In [1]:

```
# %load_ext tensorboard
%reload_ext tensorboard
```

Part 1

Importing Libraries

In [2]:

```
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets, decomposition, metrics, model_selection, preprocessing
```

Initializing Functions

In [3]:

```
def generate_data():
    '''
    generate data
    :return: X: input data, y: given labels
    '''
    np.random.seed(0)
    X, y = datasets.make_moons(200, noise=0.20)
    # y = y.reshape(-1, 1)

    return X, y
```

In [4]:

```
def plot_decision_boundary_2D(pred_func, X, y, binary_class=True):
    '''
    plot the decision boundary
    :param pred_func: function used to predict the label
    :param X: input data
    :param y: given labels
    :return:
    '''
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

    h = 0.01

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])

    if binary_class:
      Z = Z.reshape(xx.shape)
      plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
      plt.scatter(X[:, 0], X[:, 1], c=y.reshape(-1), cmap=plt.cm.Spectral)

    else:
      Z = np.argmax(Z, axis=1).reshape(xx.shape)
      plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
      plt.scatter(X[:, 0], X[:, 1], c=np.argmax(y, axis=1), cmap=plt.cm.Spectral)

    plt.title("2D Decision Boundary")
    plt.show()
```

In [5]:

```
def plot_decision_boundary_3D(pred_func, X, y):
    '''
    plot the decision boundary
    :param pred_func: function used to predict the label
    :param X: input data
    :param y: given labels
    :return:
    '''
    fig = plt.figure()

    ax = fig.add_subplot(111, projection='3d')

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    z_min, z_max = X[:, 2].min() - .5, X[:, 2].max() + .5

    h = 0.01

    xx, yy, zz = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h),
                             np.arange(z_min, z_max, h))

    Z = pred_func(np.c_[xx.ravel(), yy.ravel(), zz.ravel()])

    Z = np.argmax(Z, axis=1).reshape(xx.shape)

    for i in range(Z.shape[2]):
        z = zz[:, :, i]
        z_value = z[0, 0]
        ax.contourf(xx[:, :, i], yy[:, :, i], z, Z[:, :, i], levels=[-0.5, -0.25, 0.0, 0
.25, 0.5], alpha=0.7)

    for i, color in zip(range(3), ['r', 'g', 'b']):
        idx = np.where(np.argmax(y, axis=1) == i)
        ax.scatter(X[idx, 0], X[idx, 1], X[idx, 2], c=color, label=f'Class {i}', alpha=0
.3, edgecolors='k')

    ax.set_xlabel('PCA1')
    ax.set_ylabel('PCA2')
    ax.set_zlabel('PCA3')
    plt.title("3D Decision Boundary")
    plt.show()
```

In [6]:

```
def plot_decision_boundary_3D_no_hyperp(pred_func, X, y):
    '''
    plot the decision boundary
    :param pred_func: function used to predict the label
    :param X: input data
    :param y: given labels
    :return:
    '''
    fig = plt.figure()

    ax = fig.add_subplot(111, projection='3d')

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    z_min, z_max = X[:, 2].min() - .5, X[:, 2].max() + .5

    h = 0.01

    xx, yy, zz = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h),
                             np.arange(z_min, z_max, h))

    Z = pred_func(np.c_[xx.ravel(), yy.ravel(), zz.ravel()])
```

```
    Z = np.argmax(Z, axis=1).reshape(xx.shape)

    for i, color in zip(range(3), ['r', 'g', 'b']):
        idx = np.where(np.argmax(y, axis=1) == i)
        ax.scatter(X[idx, 0], X[idx, 1], X[idx, 2], c=color, label=f'Class {i}', alpha=0
.6, edgecolors='k')

    ax.set_xlabel('PCA1')
    ax.set_ylabel('PCA2')
    ax.set_zlabel('PCA3')
    plt.title("3D Decision Boundary")
    plt.show()
```

Initializing NeuralNetwork object

In [7]:

```
class NeuralNetwork(object):
    """
    This class builds and trains a neural network
    """
    def __init__(self, nn_input_dim, nn_hidden_dim , nn_output_dim, actFun_type='tanh',
reg_lambda=0.01, seed=0):
        '''
        :param nn_input_dim: input dimension
        :param nn_hidden_dim: the number of hidden units
        :param nn_output_dim: output dimension
        :param actFun_type: type of activation function. 3 options: 'tanh', 'sigmoid', 'r
elu'
        :param reg_lambda: regularization coefficient
        :param seed: random seed
        '''
        self.nn_input_dim = nn_input_dim
        self.nn_hidden_dim = nn_hidden_dim
        self.nn_output_dim = nn_output_dim
        self.actFun_type = actFun_type
        self.reg_lambda = reg_lambda

        # initialize the weights and biases in the network
        np.random.seed(seed)
        self.W1 = np.random.randn(self.nn_input_dim, self.nn_hidden_dim) / np.sqrt(self.
nn_input_dim)
        self.b1 = np.zeros((1, self.nn_hidden_dim))
        self.W2 = np.random.randn(self.nn_hidden_dim, self.nn_output_dim) / np.sqrt(self
.nn_hidden_dim)
        self.b2 = np.zeros((1, self.nn_output_dim))

    def actFun(self, z, af_type):
        '''
        actFun computes the activation functions
        :param z: net input
        :param af_type: Tanh, Sigmoid, or ReLU
        :return: activations
        '''
        if af_type == 'Tanh':
            return np.tanh(z)
        elif af_type == 'ReLU':
            return np.maximum(0, z)
        elif af_type == 'Sigmoid':
            return 1 / (1 + np.exp(-z))
        else:
            raise ValueError("Invalid activation function type")


    def diff_actFun(self, z, af_type):
        '''
        diff_actFun computes the derivatives of the activation functions wrt the net inpu
t
        :param z: net input
        :param af_type: Tanh, Sigmoid, or ReLU
```

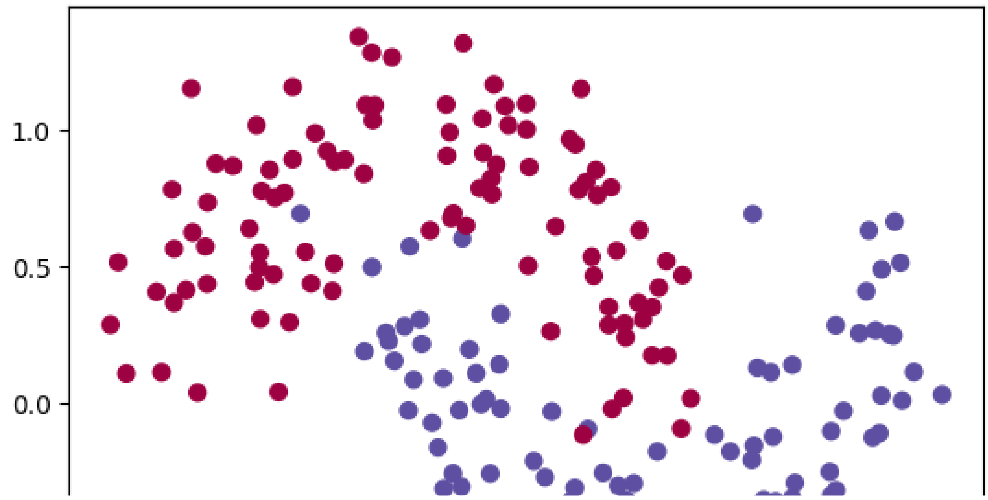
```
        :return: the derivatives of the activation functions wrt the net input
        '''
        if af_type == 'Sigmoid':
            s = 1 / (1 + np.exp(-z))
            return s * (1 - s)
        elif af_type == 'Tanh':
            return 1 - np.tanh(z)**2
        elif af_type == 'ReLU':
            return (z > 0).astype(float)
        else:
            raise ValueError("Invalid activation function type")


    def feedforward(self, X, actFun):
        '''
        feedforward builds a 3-layer neural network and computes the two probabilities,
        one for class 0 and one for class 1
        :param X: input data
        :param actFun: activation function
        :return:
        '''
        self.z1 = X.dot(self.W1) + self.b1
        self.a1 = actFun(z=self.z1, af_type=self.actFun_type)
        self.z2 = self.a1.dot(self.W2) + self.b2
        exp_scores = np.exp(self.z2)
        self.probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)


    def calculate_loss(self, X, y):
        '''
        calculate_loss computes the loss for prediction
        :param X: input data
        :param y: given labels
        :return: the loss for prediction
        '''
        self.feedforward(X, lambda z, af_type: self.actFun(z=z, af_type=af_type))

        # Calculating the loss
        corect_logprobs = -np.log(self.probs[range(len(X)), y])
        data_loss = np.sum(corect_logprobs)

        # Add regulatization term to loss (optional)
        data_loss += self.reg_lambda / 2 * (np.sum(np.square(self.W1)) + np.sum(np.squar
e(self.W2)))
        return (1. / len(X)) * data_loss

    def predict(self, X):
        '''
        predict infers the label of a given data point X
        :param X: input data
        :return: label inferred
        '''
        self.feedforward(X, lambda z, af_type: self.actFun(z=z, af_type=af_type))
        return np.argmax(self.probs, axis=1)

    def backprop(self, X, y):
        '''
        backprop implements backpropagation to compute the gradients used to update the p
arameters in the backward step
        :param X: input data
        :param y: given labels
        :return: dL/dW1, dL/b1, dL/dW2, dL/db2
        '''
        delta3 = self.probs
        delta3[range(len(X)), y] -= 1
        dW2 = (self.a1.T).dot(delta3)
        db2 = np.sum(delta3, axis=0, keepdims=True)
        delta2 = delta3.dot(self.W2.T) * self.diff_actFun(z=self.z1, af_type=self.actFun
_type)
        dW1 = np.dot(X.T, delta2)
        db1 = np.sum(delta2, axis=0)
```

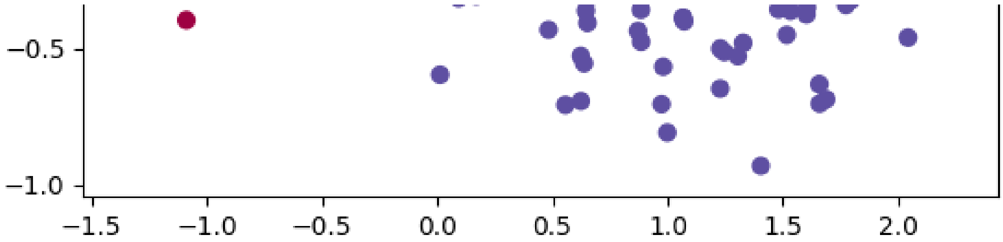
```
        return dW1, dW2, db1, db2

    def fit_model(self, X, y, epsilon=0.01, num_passes=20000, print_loss=True):
        '''
        fit_model uses backpropagation to train the network
        :param X: input data
        :param y: given labels
        :param num_passes: the number of times that the algorithm runs through the whole
dataset
        :param print_loss: print the loss or not
        :return:
        '''
        # Gradient descent.
        for i in range(0, num_passes):
            # Forward propagation
            self.feedforward(X, lambda z, af_type: self.actFun(z=z, af_type=af_type))
            # Backpropagation
            dW1, dW2, db1, db2 = self.backprop(X, y)

            # Add regularization terms (b1 and b2 don't have regularization terms)
            dW2 += self.reg_lambda * self.W2
            dW1 += self.reg_lambda * self.W1

            # Gradient descent parameter update
            self.W1 += -epsilon * dW1
            self.b1 += -epsilon * db1
            self.W2 += -epsilon * dW2
            self.b2 += -epsilon * db2

            # Optionally print the loss.
            # This is expensive because it uses the whole dataset, so we don't want to do
it too often.
            if print_loss and i % 1000 == 0:
                print("Loss after iteration %i: %f" % (i, self.calculate_loss(X, y)))

    def visualize_decision_boundary(self, X, y):
        '''
        visualize_decision_boundary plots the decision boundary created by the trained ne
twork
        :param X: input data
        :param y: given labels
        :return:
        '''
        plot_decision_boundary_2D(lambda x: self.predict(x), X, y)
```

Plot original "Make Moons" dataset

In [8]:

```
X, y = generate_data()

plt.scatter(X[:, 0], X[:, 1], s=40, c=y, cmap=plt.cm.Spectral)
plt.show()
```



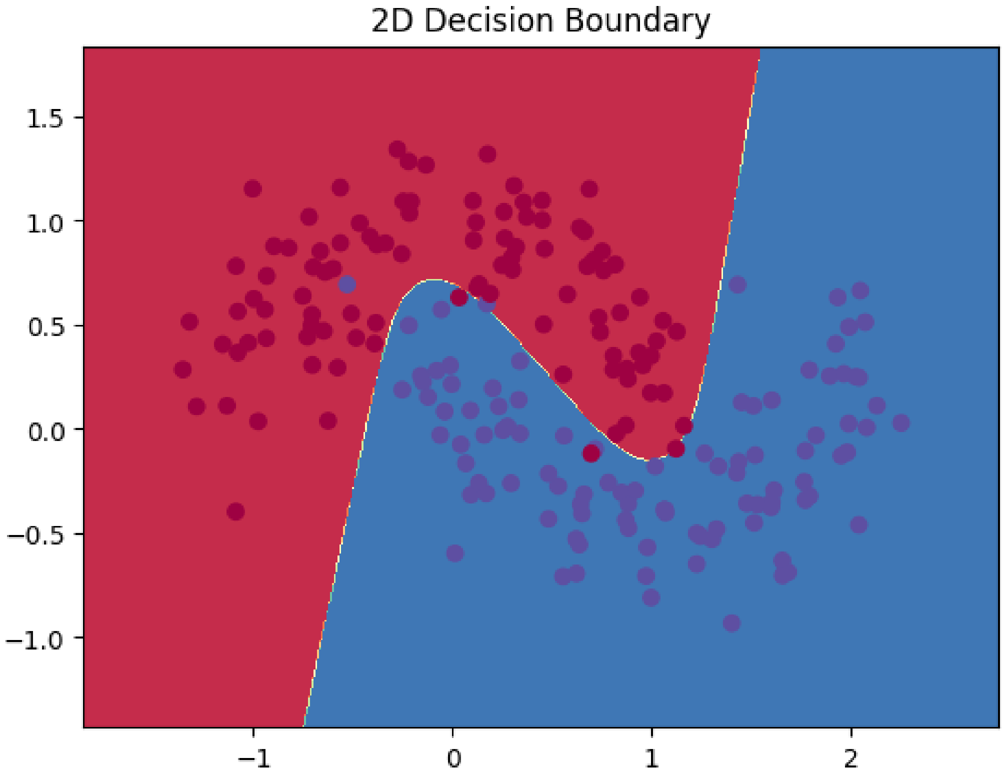


Fit 3-hidden-layer neural network with "TanH" as its activation function

In [9]:

```
model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 , nn_output_dim=2, actFun_type='Ta
nh')
model.fit_model(X, y)
model.visualize_decision_boundary(X, y)
```

```
Loss after iteration 0: 0.432387
Loss after iteration 1000: 0.068947
Loss after iteration 2000: 0.068950
Loss after iteration 3000: 0.071218
Loss after iteration 4000: 0.071253
Loss after iteration 5000: 0.071278
Loss after iteration 6000: 0.071293
Loss after iteration 7000: 0.071303
Loss after iteration 8000: 0.071308
Loss after iteration 9000: 0.071312
Loss after iteration 10000: 0.071314
Loss after iteration 11000: 0.071315
Loss after iteration 12000: 0.071315
Loss after iteration 13000: 0.071316
Loss after iteration 14000: 0.071316
Loss after iteration 15000: 0.071316
Loss after iteration 16000: 0.071316
Loss after iteration 17000: 0.071316
Loss after iteration 18000: 0.071316
Loss after iteration 19000: 0.071316
```



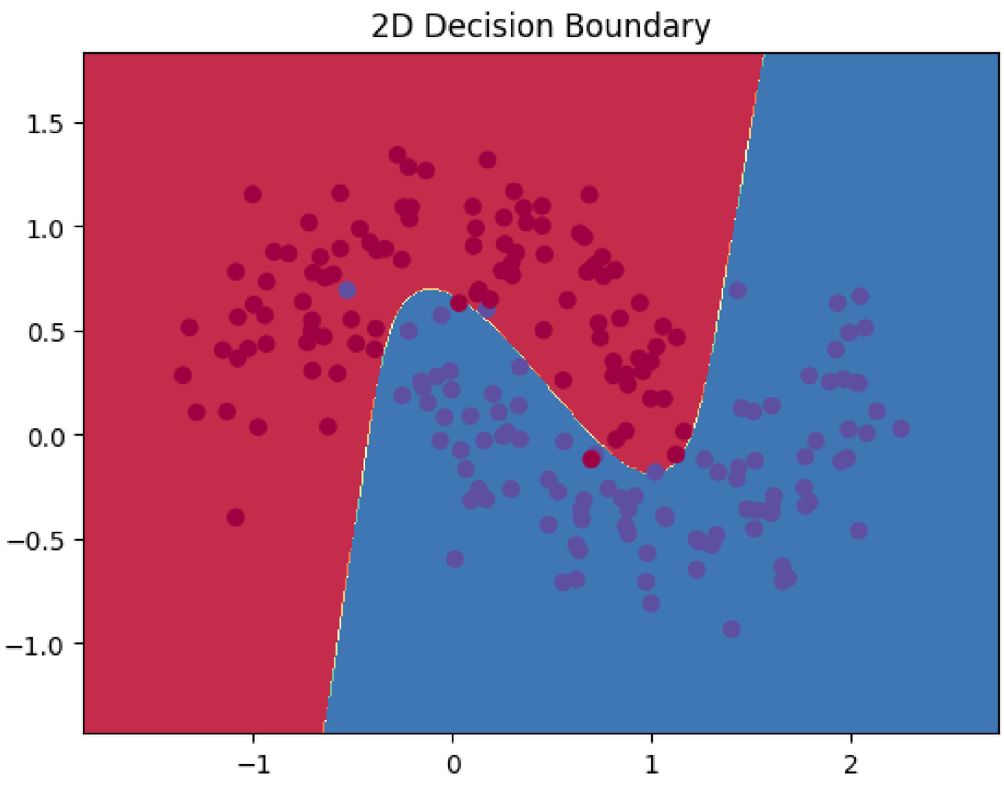
Fit 3-hidden-layer neural network with "Sigmoid" as its activation function

function

In [10]:

```
model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 , nn_output_dim=2, actFun_type='Si
gmoid')
model.fit_model(X, y)
model.visualize_decision_boundary(X, y)
```

```
Loss after iteration 0: 0.628571
Loss after iteration 1000: 0.088431
Loss after iteration 2000: 0.079598
Loss after iteration 3000: 0.078604
Loss after iteration 4000: 0.078330
Loss after iteration 5000: 0.078233
Loss after iteration 6000: 0.078192
Loss after iteration 7000: 0.078174
Loss after iteration 8000: 0.078166
Loss after iteration 9000: 0.078161
Loss after iteration 10000: 0.078159
Loss after iteration 11000: 0.078158
Loss after iteration 12000: 0.078157
Loss after iteration 13000: 0.078156
Loss after iteration 14000: 0.078156
Loss after iteration 15000: 0.078156
Loss after iteration 16000: 0.078156
Loss after iteration 17000: 0.078156
Loss after iteration 18000: 0.078156
Loss after iteration 19000: 0.078155
```



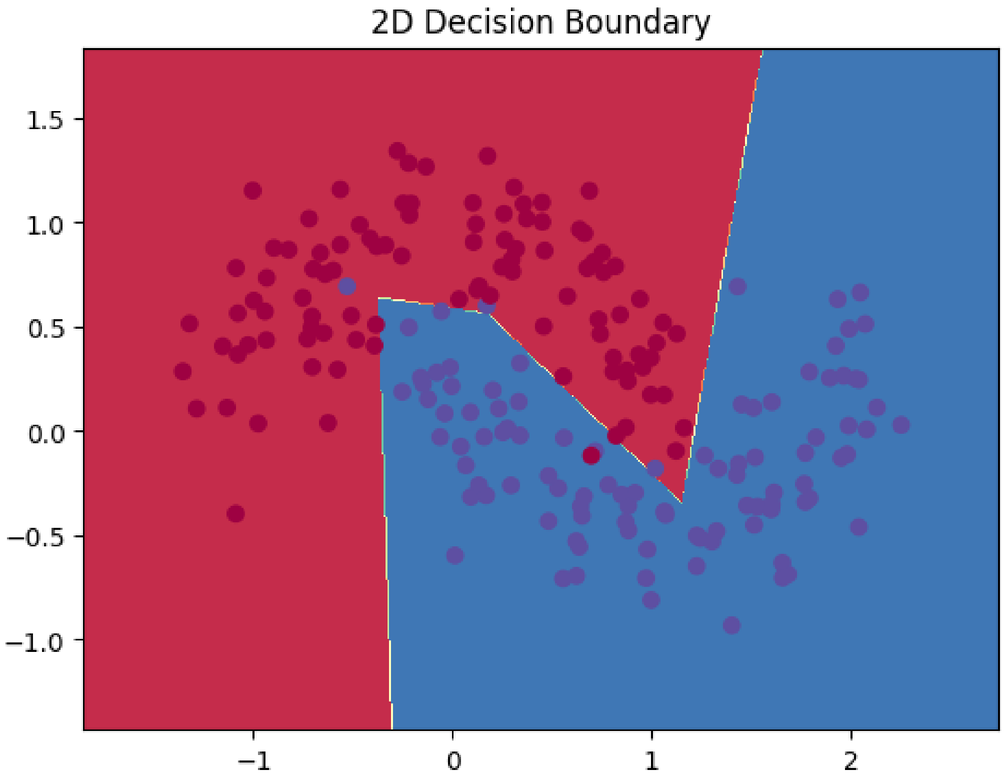
Fit 3-hidden-layer neural network with "ReLU" as its activation function

In [11]:

```
model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 , nn_output_dim=2, actFun_type='Re
LU')
model.fit_model(X, y)
model.visualize_decision_boundary(X, y)
```

```
Loss after iteration 0: 0.560274
Loss after iteration 1000: 0.072179
Loss after iteration 2000: 0.071301
Loss after iteration 3000: 0.071159
```

```
Loss after iteration 3000: 0.071159
Loss after iteration 4000: 0.071190
Loss after iteration 5000: 0.071136
Loss after iteration 6000: 0.071276
Loss after iteration 7000: 0.071090
Loss after iteration 8000: 0.071265
Loss after iteration 9000: 0.071084
Loss after iteration 10000: 0.071090
Loss after iteration 11000: 0.071087
Loss after iteration 12000: 0.071086
Loss after iteration 13000: 0.071069
Loss after iteration 14000: 0.071114
Loss after iteration 15000: 0.071074
Loss after iteration 16000: 0.071113
Loss after iteration 17000: 0.071071
Loss after iteration 18000: 0.071090
Loss after iteration 19000: 0.071219
```

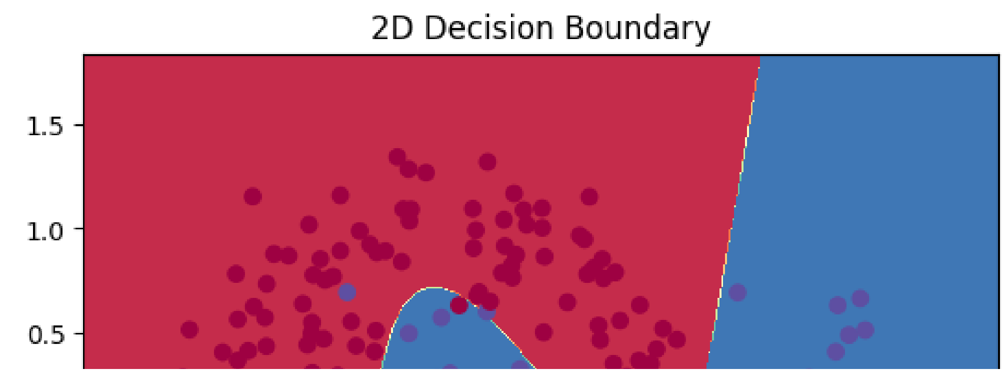


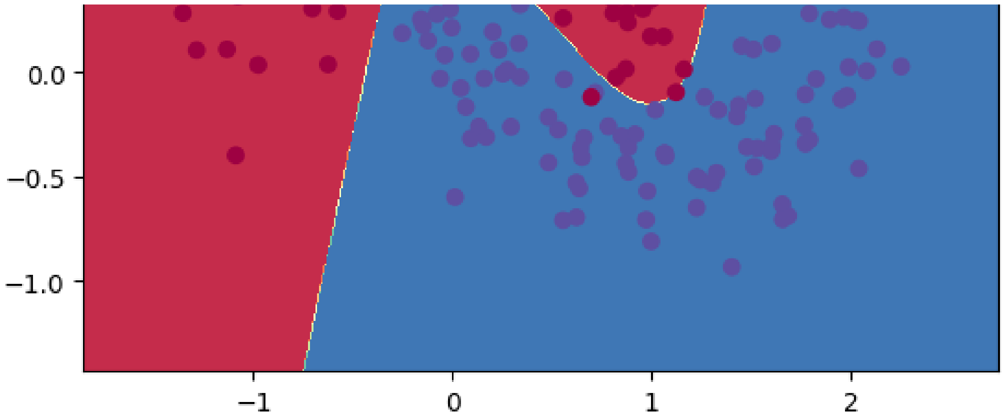
Testing several n-hidden-layer neural networks with "TanH" as its activation function

In [12]:

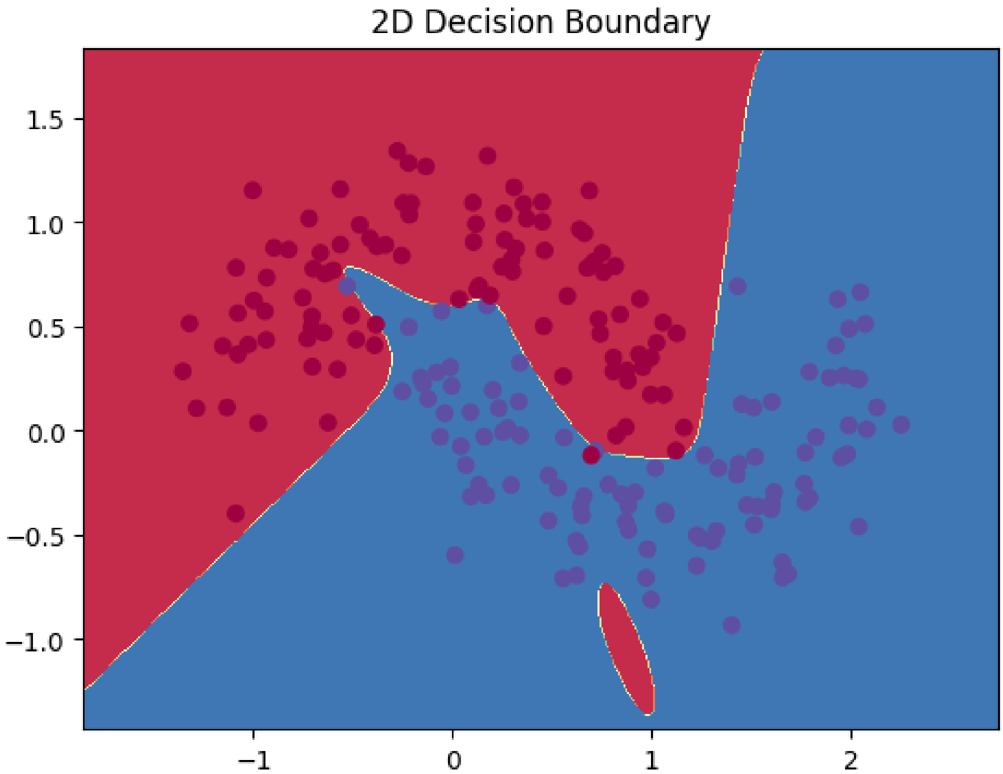
```
for hidden_dim in [3, 5, 10, 20, 30]:
    model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=hidden_dim, nn_output_dim=2, act
Fun_type='Tanh')
    print(f"\n{hidden_dim} hidden dims:")
    model.fit_model(X, y, print_loss=False)
    plot_decision_boundary_2D(lambda x: model.predict(x), X, y)
```

```
3 hidden dims:
```

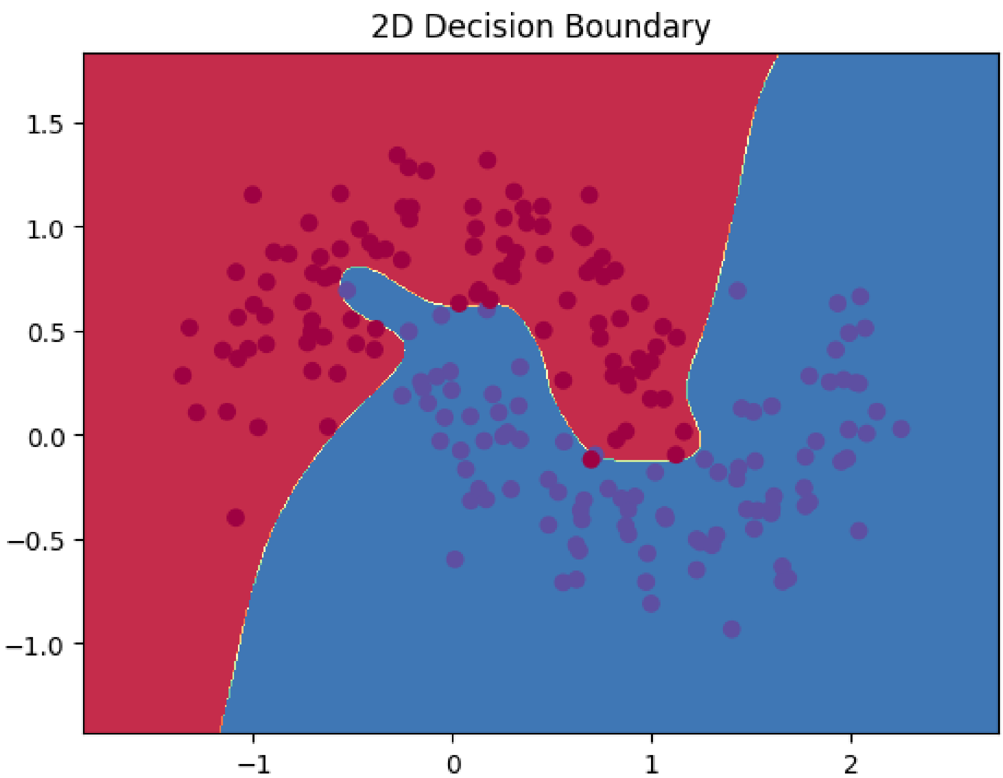




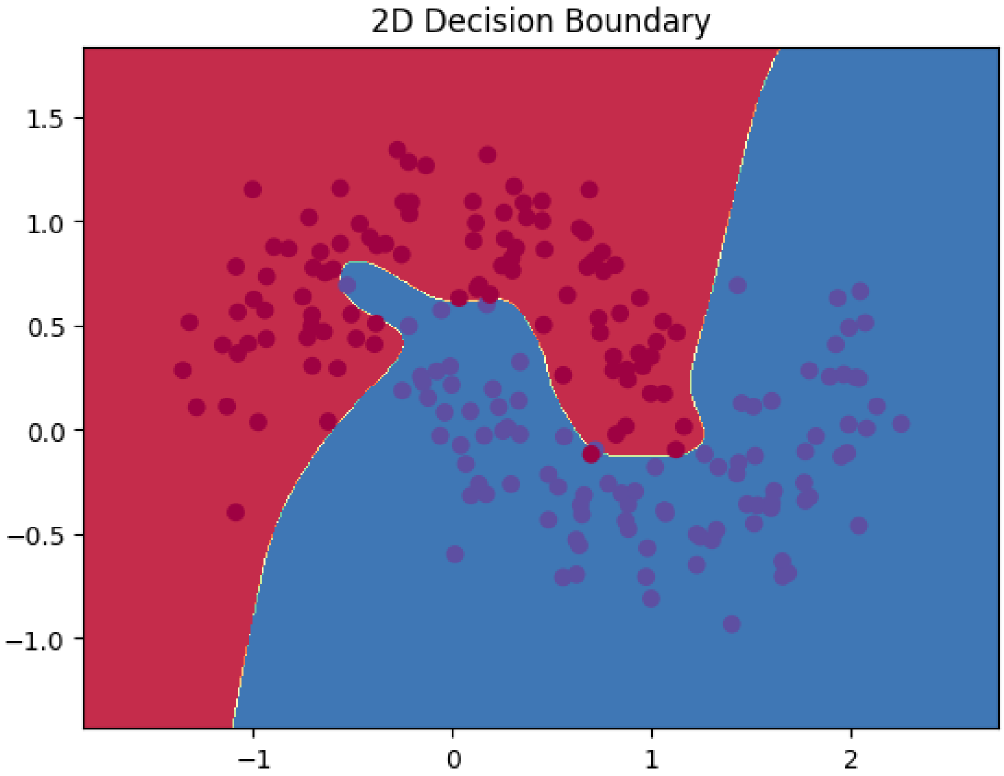
5 hidden dims:



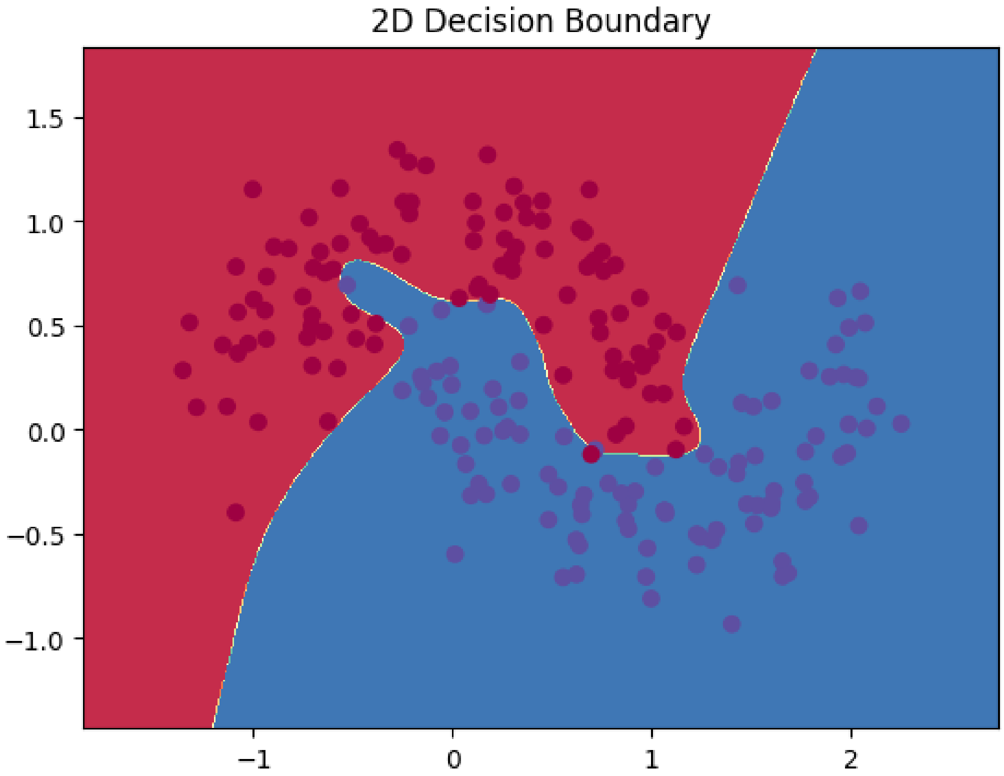
10 hidden dims:



20 hidden dims:



30 hidden dims:



Configurable Neural Network experiment (n number of layers and m_{n_i} layer sizes)

Initializing Generic Layer Object

In [13]:

```
class Layer:
    def __init__(self, input_size, output_size, activation='relu'):
        self.input_size = input_size
```

```
        self.output_size = output_size
        self.activation = activation
        # Initialize weights with small random values
        self.W = np.random.randn(input_size, output_size) * 0.01
        self.b = np.zeros((1, output_size))

    def activate(self, x):
        if self.activation == 'relu':
            return np.maximum(0, x)
        elif self.activation == 'sigmoid':
            return 1 / (1 + np.exp(-x))
        elif self.activation == 'softmax':
            exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
            return exp_x / np.sum(exp_x, axis=1, keepdims=True)
        else:
            return x  # linear activation

    def feedforward(self, a_prev):
        self.a_prev = a_prev
        self.z = np.dot(a_prev, self.W) + self.b
        self.a = self.activate(self.z)
        return self.a

    def backprop(self, delta_next, W_next, learning_rate):
        if self.activation == 'relu':
            delta = np.dot(delta_next, W_next.T) * (self.z > 0)
        elif self.activation == 'sigmoid':
            delta = np.dot(delta_next, W_next.T) * (self.a * (1 - self.a))
        else:  # linear or softmax, delta will be directly passed
            delta = delta_next

        dW = np.dot(self.a_prev.T, delta)
        db = np.sum(delta, axis=0, keepdims=True)
        self.W -= learning_rate * dW
        self.b -= learning_rate * db
        return delta
```

Initializing the modular Deep Neural Network

In [14]:

```
class DeepNeuralNetwork:
    def __init__(self, layer_sizes, activation_functions):
        self.layers = []
        for i in range(len(layer_sizes) - 1):
            layer = Layer(layer_sizes[i], layer_sizes[i + 1], activation=activation_func
tions[i])
            self.layers.append(layer)

    def feedforward(self, X):
        a = X
        for layer in self.layers:
            a = layer.feedforward(a)
        return a

    def backprop(self, y, output, learning_rate):
        m = y.shape[0]
        delta_output = output - y
        for i in reversed(range(len(self.layers))):
            if i == len(self.layers) - 1:
                delta = delta_output
            else:
                delta = self.layers[i].backprop(delta, self.layers[i + 1].W, learning_ra
te)

    def calculate_loss(self, X, y, reg_lambda):
        output = self.feedforward(X)
        loss = -np.sum(y * np.log(output))
        for layer in self.layers:
            loss += reg_lambda / 2 * np.sum(layer.W ** 2)
```

```
        return loss / X.shape[0]

    def fit_model(self, X, y, epochs=100000, learning_rate=0.01, reg_lambda=0.01, print_
loss=True):
        for epoch in range(epochs):
            output = self.feedforward(X)
            self.backprop(y, output, learning_rate)
            if print_loss and epoch % 1000 == 0:
                loss = self.calculate_loss(X, y, reg_lambda)
                print(f"Loss after iteration {epoch}: {loss}")

    def predict(self, X):
        output = self.feedforward(X)
        return np.round(output)
```

Regenerate "Make Moons" dataset for new DNN compatibility with labels

In [15]:

```
X, y = generate_data()

y = y.reshape(-1, 1)
```

Initialize the deep neural network with 3 layers: [2, 4, 1]

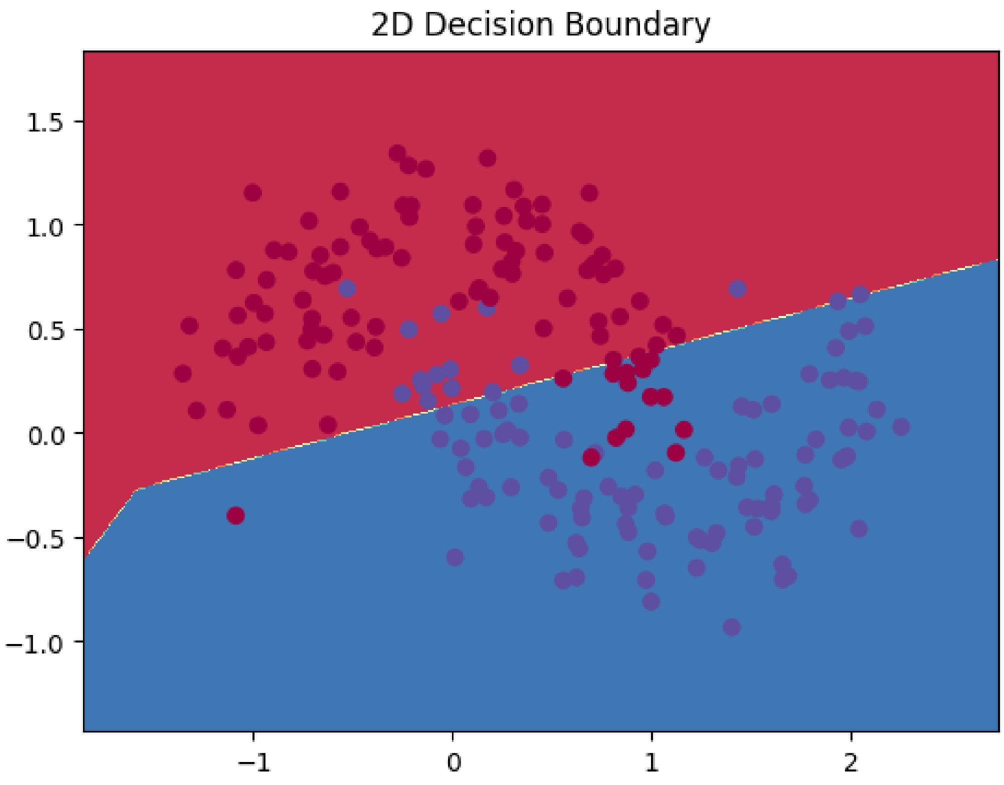
In [16]:

```
layer_sizes = [2, 4, 1]

activation_functions = ['relu', 'sigmoid']

dnn = DeepNeuralNetwork(layer_sizes, activation_functions)
```

Train the model and plot decision boundary

In [17]:

```
dnn.fit_model(X, y, epochs=100000, print_loss=False)

plot_decision_boundary_2D(lambda x: dnn.predict(x), X, y)
```



Evaluate the model

In [18]:

```
y_pred = dnn.predict(X)

print(f"Accuracy: {metrics.accuracy_score(y, y_pred) * 100:.2f}%")
```

```
Accuracy: 84.50%
```

Experiments with different configurations for binary classification

Function to run experiments on Make_Moons dataset

In [19]:

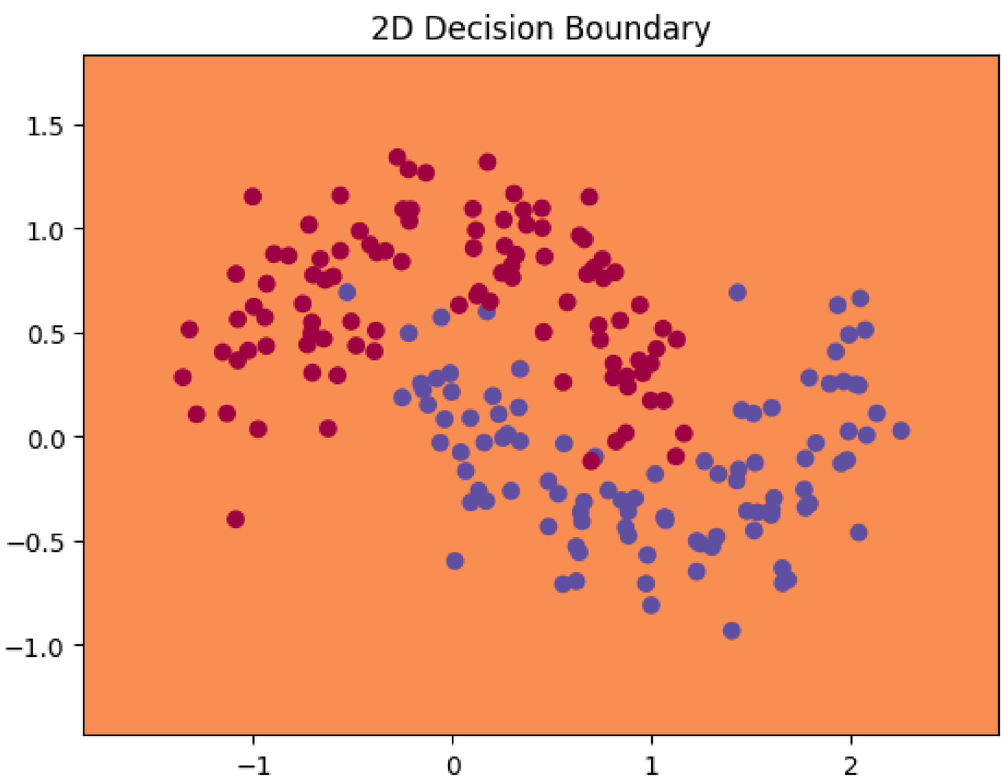
```
def run_experiment(layer_sizes, activation_functions, epochs=100000, learning_rate=0.01,
reg_lambda=0.01, plot_title=""):
    print(f"\nRunning experiment with layer_sizes = {layer_sizes}, activation_functions
= {activation_functions}")
    dnn_exp = DeepNeuralNetwork(layer_sizes, activation_functions)
    dnn_exp.fit_model(X, y, epochs=epochs, learning_rate=learning_rate, reg_lambda=reg_l
ambda, print_loss=False)
    print(f"Accuracy: {metrics.accuracy_score(y, dnn_exp.predict(X)) * 100:.2f}%")
    plot_decision_boundary_2D(lambda x: dnn_exp.predict(x), X, y)
```

Shallow vs Deep Networks

In [20]:

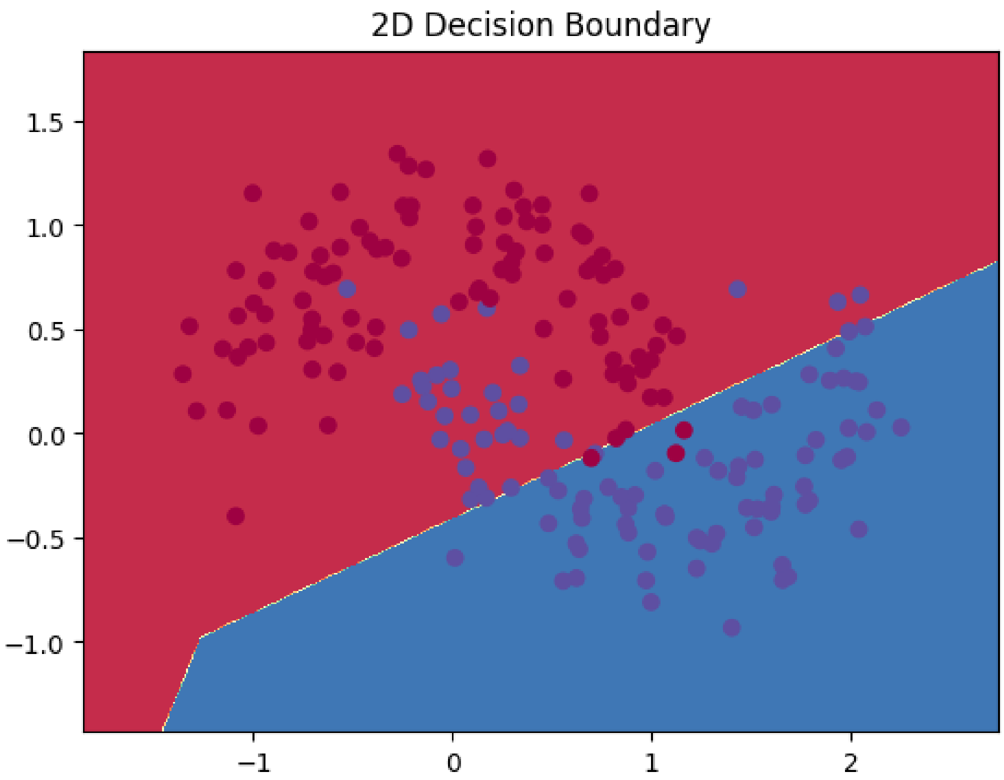
```
run_experiment([2, 4, 4, 4, 1], ['relu', 'relu', 'relu', 'sigmoid'], plot_title="Deep Ne
twork: [2, 4, 4, 4, 1]")
run_experiment([2, 4, 1], ['relu', 'sigmoid'], plot_title="Shallow Network: [2, 4, 1]")
```

```
Running experiment with layer_sizes = [2, 4, 4, 4, 1], activation_functions = ['relu', 'r
elu', 'relu', 'sigmoid']
Accuracy: 50.00%
```




```
Running experiment with layer_sizes = [2, 4, 1], activation_functions = ['relu', 'sigmoid
']
```

']
Accuracy: 82.00%

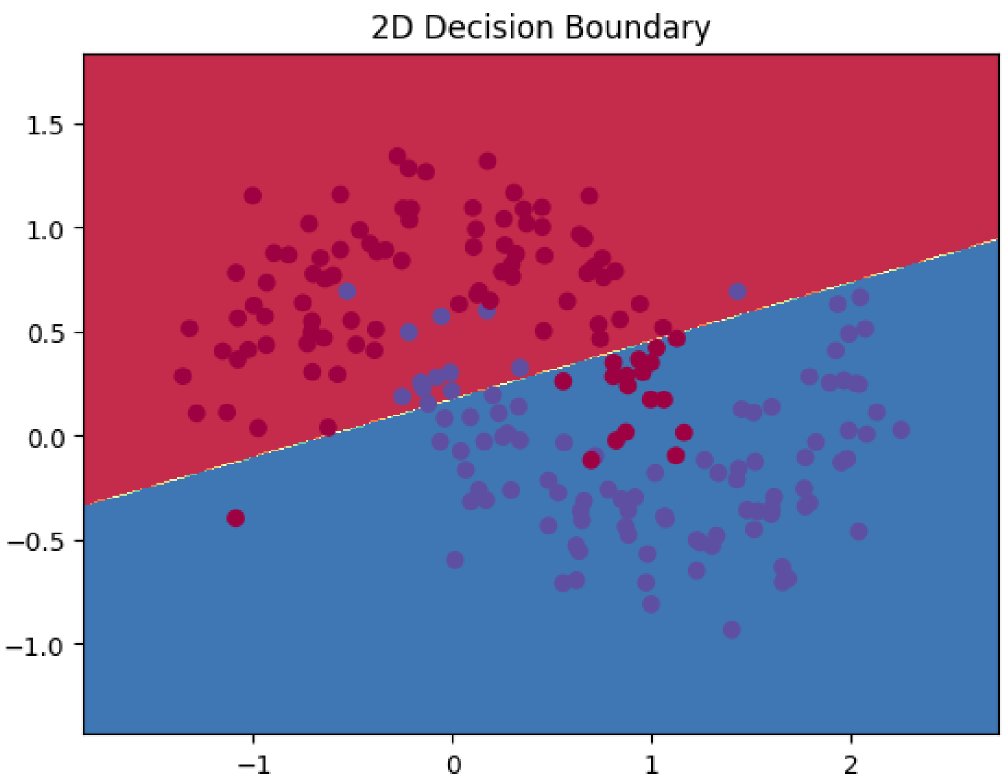


Different Activation Functions in hidden layers (other than relu)

In [21]:

```
run_experiment([2, 4, 1], ['sigmoid', 'sigmoid'], plot_title="Shallow Network with Sigmoid: [2, 4, 1]")
```

Running experiment with layer_sizes = [2, 4, 1], activation_functions = ['sigmoid', 'sigmoid']
Accuracy: 84.50%

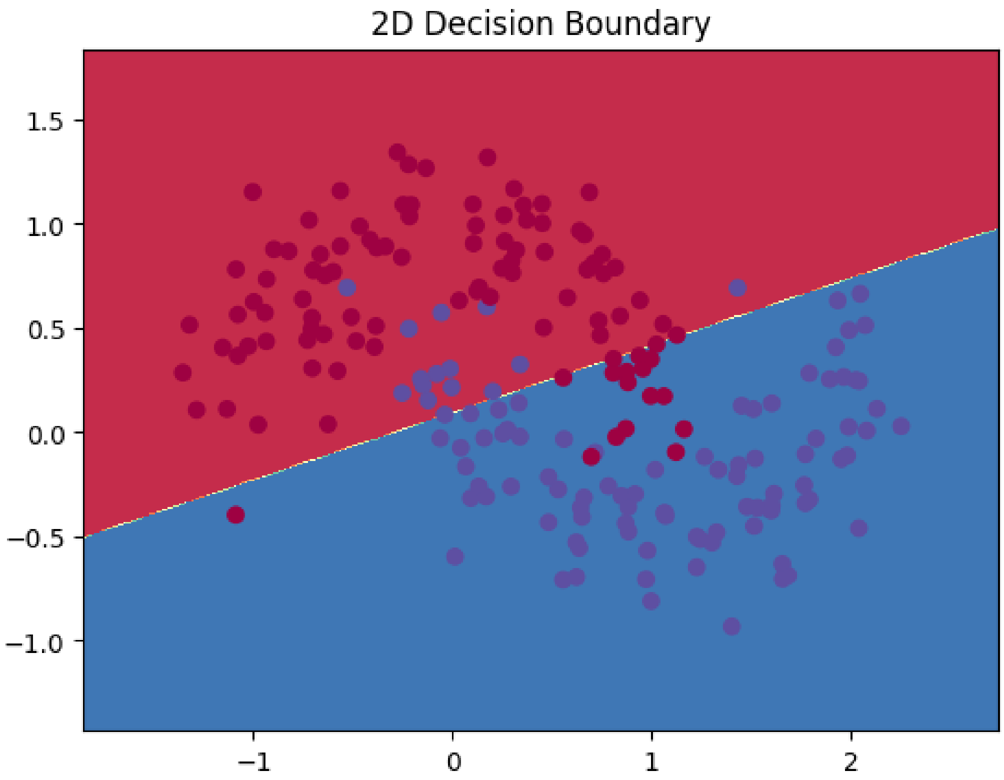


Different Layer Sizes in hidden layers

In [22]:

```
run_experiment([2, 8, 1], ['relu', 'sigmoid'], plot_title="Shallow Network with Larger H
idden Layer: [2, 8, 1]")
```

```
Running experiment with layer_sizes = [2, 8, 1], activation_functions = ['relu', 'sigmoid
']
Accuracy: 85.00%
```



Experiments on different dataset for multiclassification ("load_iris" w/ three distinct class labels)

Load Iris dataset

In [23]:

```
X_iris = datasets.load_iris().data
y_iris = datasets.load_iris().target.reshape(-1, 1)
```

One-hot encode the labels

In [24]:

```
# encoder = preprocessing.OneHotEncoder(sparse=False)
y_iris_onehot = preprocessing.OneHotEncoder(sparse=False).fit_transform(y_iris)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWar
ning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4.
`sparse_output` is ignored unless you leave `sparse` to its default value.
  warnings.warn(
```

Split the "load_iris" data into training and test sets

In [25]:

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X_iris, y_iris_oneho
t, test_size=0.2, random_state=42)
```

Initialize and train the deep neural network with 3 layers: [4, 5, 3]

In [26]:

```
layer_sizes_iris = [4, 5, 3]

activation_functions_iris = ['relu', 'softmax']

dnn_iris = DeepNeuralNetwork(layer_sizes_iris, activation_functions_iris)

dnn_iris.fit_model(X_train, y_train, epochs=100000, learning_rate=0.005, reg_lambda=0.01
, print_loss=False)
```

Evaluate the model on test data

In [27]:

```
# y_test_pred = dnn_iris.predict(X_test)
# y_test_class = np.argmax(y_test, axis=1)
# y_test_pred_class = np.argmax(y_test_pred, axis=1)
print(f"Accuracy on Iris dataset: {metrics.accuracy_score(np.argmax(y_test, axis=1), np.a
rgmax(dnn_iris.predict(X_test), axis=1)) * 100:.2f}%")
```

```
Accuracy on Iris dataset: 63.33%
```

Apply PCA to reduce dimensions to 2 and 3 features for plotting

Two-dimensional PCA

Initialize 2D PCA

In [28]:

```
pca_2d = decomposition.PCA(n_components=2)

X_train_2d = pca_2d.fit_transform(X_train)

X_test_2d = pca_2d.transform(X_test)
```

Initialize and train models on the reduced feature sets

In [29]:

```
dnn_2d = DeepNeuralNetwork([2, 5, 3], ['relu', 'softmax'])

dnn_2d.fit_model(X_train_2d, y_train, epochs=100000, learning_rate=0.01, reg_lambda=0.01
)
```

```
Loss after iteration 0: 1.098385487369922
Loss after iteration 1000: 0.9082320421692393
Loss after iteration 2000: 0.8231367063929823
Loss after iteration 3000: 0.7795857381058384
Loss after iteration 4000: 0.7551306229952008
Loss after iteration 5000: 0.7408932257080784
Loss after iteration 6000: 0.7327473547236737
Loss after iteration 7000: 0.7285624492795755
Loss after iteration 8000: 0.7272470774868071
Loss after iteration 9000: 0.7280486205948803
Loss after iteration 10000: 0.7304193782206853
Loss after iteration 11000: 0.734009740187467
Loss after iteration 12000: 0.7385320796999898
Loss after iteration 13000: 0.7438091251492599
Loss after iteration 14000: 0.7498052404825255
Loss after iteration 15000: 0.7563492959637887
Loss after iteration 16000: 0.7633475390868166
```

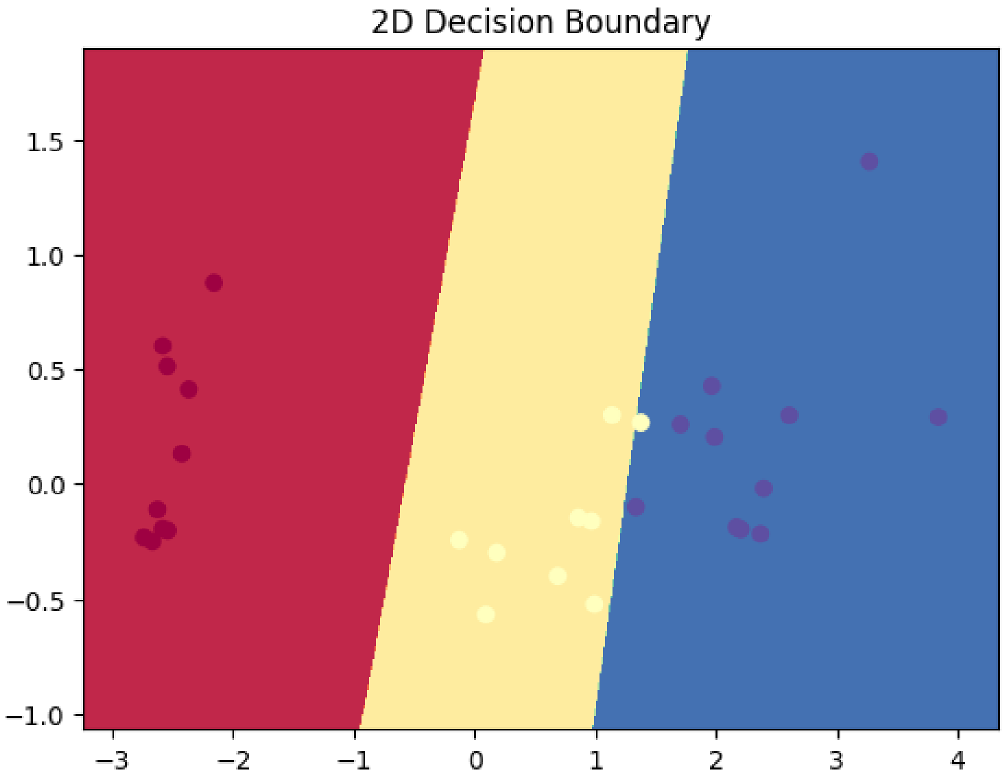
```
Loss after iteration 17000: 0.7707086289712995
Loss after iteration 18000: 0.7783732157500224
Loss after iteration 19000: 0.7862022814997052
Loss after iteration 20000: 0.7942069448539932
Loss after iteration 21000: 0.8024224219136324
Loss after iteration 22000: 0.8108296682118941
Loss after iteration 23000: 0.8194091578102504
Loss after iteration 24000: 0.8281473269705322
Loss after iteration 25000: 0.8371378211306559
Loss after iteration 26000: 0.8462949807550545
Loss after iteration 27000: 0.8555776149744854
Loss after iteration 28000: 0.8649757527552101
Loss after iteration 29000: 0.8743739134281417
Loss after iteration 30000: 0.8837462909782624
Loss after iteration 31000: 0.8931936480858284
Loss after iteration 32000: 0.902713672623411
Loss after iteration 33000: 0.9123041164101013
Loss after iteration 34000: 0.9219627422755351
Loss after iteration 35000: 0.9316873006393902
Loss after iteration 36000: 0.9414755235366878
Loss after iteration 37000: 0.9513251284259302
Loss after iteration 38000: 0.9612338269819372
Loss after iteration 39000: 0.9712114267424873
Loss after iteration 40000: 0.9812755575943851
Loss after iteration 41000: 0.9913860676337667
Loss after iteration 42000: 1.0015445330375379
Loss after iteration 43000: 1.011747981121853
Loss after iteration 44000: 1.0219943682788926
Loss after iteration 45000: 1.0322814730488965
Loss after iteration 46000: 1.0426074340958558
Loss after iteration 47000: 1.0529697430576828
Loss after iteration 48000: 1.0633625609454895
Loss after iteration 49000: 1.0737816276754828
Loss after iteration 50000: 1.0842272834749431
Loss after iteration 51000: 1.0946991157060395
Loss after iteration 52000: 1.105195052117422
Loss after iteration 53000: 1.1157147574512762
Loss after iteration 54000: 1.1262562177467148
Loss after iteration 55000: 1.1368191571720243
Loss after iteration 56000: 1.147402190775243
Loss after iteration 57000: 1.1580045258869436
Loss after iteration 58000: 1.168624822157291
Loss after iteration 59000: 1.1792626247034819
Loss after iteration 60000: 1.189916632603836
Loss after iteration 61000: 1.200586436769633
Loss after iteration 62000: 1.2112604684876347
Loss after iteration 63000: 1.2219457673338032
Loss after iteration 64000: 1.2326451985667781
Loss after iteration 65000: 1.2433575103839907
Loss after iteration 66000: 1.2540820484010744
Loss after iteration 67000: 1.2648157796949053
Loss after iteration 68000: 1.2755597466176616
Loss after iteration 69000: 1.2863141537097782
Loss after iteration 70000: 1.2970775447821257
Loss after iteration 71000: 1.3078499400179806
Loss after iteration 72000: 1.3186307889438866
Loss after iteration 73000: 1.329418707503522
Loss after iteration 74000: 1.3402048251438166
Loss after iteration 75000: 1.3509914548169235
Loss after iteration 76000: 1.3617784482255681
Loss after iteration 77000: 1.3725656504678758
Loss after iteration 78000: 1.3833526043739621
Loss after iteration 79000: 1.3941397327469969
Loss after iteration 80000: 1.404926274493352
Loss after iteration 81000: 1.4157123473737139
Loss after iteration 82000: 1.4264977714837024
Loss after iteration 83000: 1.4372823636388141
Loss after iteration 84000: 1.4480662301341467
Loss after iteration 85000: 1.4588488902432128
Loss after iteration 86000: 1.469630152496996
Loss after iteration 87000: 1.4804098249147783
Loss after iteration 88000: 1.4911880097173684
```

```
Loss after iteration 89000: 1.5019645112125806
Loss after iteration 90000: 1.512739136248707
Loss after iteration 91000: 1.5235113968246918
Loss after iteration 92000: 1.5342816841325775
Loss after iteration 93000: 1.5450492184356055
Loss after iteration 94000: 1.5558143914707325
Loss after iteration 95000: 1.5666178299954119
Loss after iteration 96000: 1.5774560514434388
Loss after iteration 97000: 1.5882877231311738
Loss after iteration 98000: 1.5991122954043147
Loss after iteration 99000: 1.6099281054696855
```

Plot the 2D decision boundary

In [30]:

```
plot_decision_boundary_2D(lambda x: dnn_2d.feedforward(x), X_test_2d, y_test, binary_cla
ss=False)
```



Three-dimensional PCA

Initialize 3D PCA

In [31]:

```
pca_3d = decomposition.PCA(n_components=3)

X_train_3d = pca_3d.fit_transform(X_train)

X_test_3d = pca_3d.transform(X_test)
```

Initialize and train models on the reduced feature sets

In [32]:

```
dnn_3d = DeepNeuralNetwork([3, 5, 3], ['relu', 'softmax'])

dnn_3d.fit_model(X_train_3d, y_train, epochs=100000, learning_rate=0.01, reg_lambda=0.01
)
```

```
Loss after iteration 0: 1.098308475136801
```

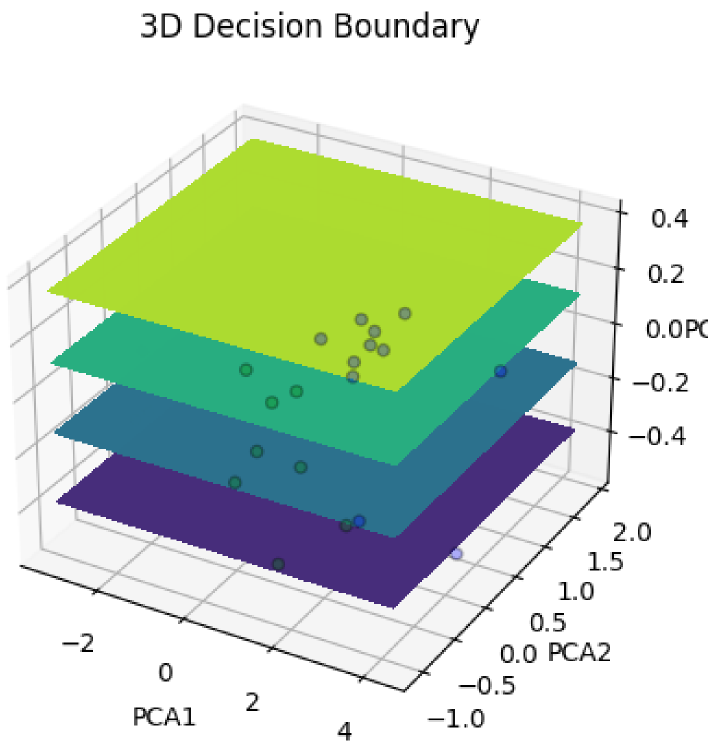
```
Loss after iteration 0: [illegible]
Loss after iteration 1000: 0.9906444252937125
Loss after iteration 2000: 0.954871982066615
Loss after iteration 3000: 0.9382897174596468
Loss after iteration 4000: 0.9295555776334246
Loss after iteration 5000: 0.9250086217352269
Loss after iteration 6000: 0.9233099561248992
Loss after iteration 7000: 0.9234668772213295
Loss after iteration 8000: 0.9251553347527113
Loss after iteration 9000: 0.9280403926440154
Loss after iteration 10000: 0.9318139862591991
Loss after iteration 11000: 0.9363266168132901
Loss after iteration 12000: 0.9414682388323565
Loss after iteration 13000: 0.9471550104335591
Loss after iteration 14000: 0.9533243677289496
Loss after iteration 15000: 0.9599471913864742
Loss after iteration 16000: 0.9669945494845744
Loss after iteration 17000: 0.9743826510281886
Loss after iteration 18000: 0.9820774328769377
Loss after iteration 19000: 0.9901496105127336
Loss after iteration 20000: 0.9985429398760287
Loss after iteration 21000: 1.0071724657006675
Loss after iteration 22000: 1.0160006959847072
Loss after iteration 23000: 1.0250090065042863
Loss after iteration 24000: 1.03430877600418
Loss after iteration 25000: 1.0437623000757241
Loss after iteration 26000: 1.0533333949259818
Loss after iteration 27000: 1.0630261032926922
Loss after iteration 28000: 1.0728027420728516
Loss after iteration 29000: 1.0826527745261605
Loss after iteration 30000: 1.0925669030544223
Loss after iteration 31000: 1.1025369108916316
Loss after iteration 32000: 1.1125905742150373
Loss after iteration 33000: 1.122762865367005
Loss after iteration 34000: 1.1329702951488871
Loss after iteration 35000: 1.1432073523671527
Loss after iteration 36000: 1.1534690106380154
Loss after iteration 37000: 1.1637686124335214
Loss after iteration 38000: 1.174126948430416
Loss after iteration 39000: 1.1844904585002147
Loss after iteration 40000: 1.1948554315060038
Loss after iteration 41000: 1.2052185317514286
Loss after iteration 42000: 1.2155767896712042
Loss after iteration 43000: 1.2259276008913116
Loss after iteration 44000: 1.2362685346625855
Loss after iteration 45000: 1.246597695095871
Loss after iteration 46000: 1.2569132129563003
Loss after iteration 47000: 1.267213504153687
Loss after iteration 48000: 1.2774972476357624
Loss after iteration 49000: 1.2877633152472205
Loss after iteration 50000: 1.2980105423525528
Loss after iteration 51000: 1.3082382507694574
Loss after iteration 52000: 1.3184456315207036
Loss after iteration 53000: 1.3286321309299056
Loss after iteration 54000: 1.3387967895728243
Loss after iteration 55000: 1.3489387204826273
Loss after iteration 56000: 1.359057676906956
Loss after iteration 57000: 1.3691534231565676
Loss after iteration 58000: 1.3792258729345597
Loss after iteration 59000: 1.3892749348458036
Loss after iteration 60000: 1.3993005607258537
Loss after iteration 61000: 1.4093030240918918
Loss after iteration 62000: 1.4192822256451687
Loss after iteration 63000: 1.429238361852851
Loss after iteration 64000: 1.439362168792677
Loss after iteration 65000: 1.4495791734953252
Loss after iteration 66000: 1.4598210353875902
Loss after iteration 67000: 1.4700308705602292
Loss after iteration 68000: 1.4802090834277675
Loss after iteration 69000: 1.49035611753951
Loss after iteration 70000: 1.5004724508551075
Loss after iteration 71000: 1.5105585914460478
Loss after iteration 72000: 1.520615073584561
```

```
Loss after iteration 73000: 1.530642454185246
Loss after iteration 74000: 1.540641309567938
Loss after iteration 75000: 1.5506122325133525
Loss after iteration 76000: 1.560600876359902
Loss after iteration 77000: 1.5705746312902156
Loss after iteration 78000: 1.5805149165130148
Loss after iteration 79000: 1.5904224606047281
Loss after iteration 80000: 1.6002979983379118
Loss after iteration 81000: 1.6101422688339226
Loss after iteration 82000: 1.6199560138908622
Loss after iteration 83000: 1.629739976471422
Loss after iteration 84000: 1.6394948993367406
Loss after iteration 85000: 1.6492215238135575
Loss after iteration 86000: 1.6589205886831728
Loss after iteration 87000: 1.6687889935044495
Loss after iteration 88000: 1.6787470244069633
Loss after iteration 89000: 1.6886714132252485
Loss after iteration 90000: 1.6985629286215354
Loss after iteration 91000: 1.708422339118272
Loss after iteration 92000: 1.7182504118529893
Loss after iteration 93000: 1.728047911454258
Loss after iteration 94000: 1.737815599028462
Loss after iteration 95000: 1.7475542312479908
Loss after iteration 96000: 1.757314467304572
Loss after iteration 97000: 1.7670632704236298
Loss after iteration 98000: 1.776773978431374
Loss after iteration 99000: 1.7864474506764032
```

Plot the 3D decision boundaries

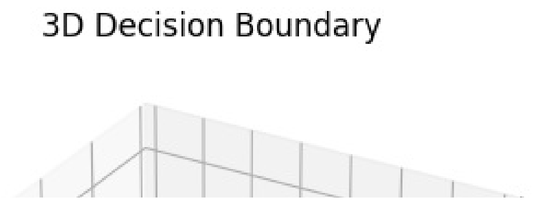
In [33]:

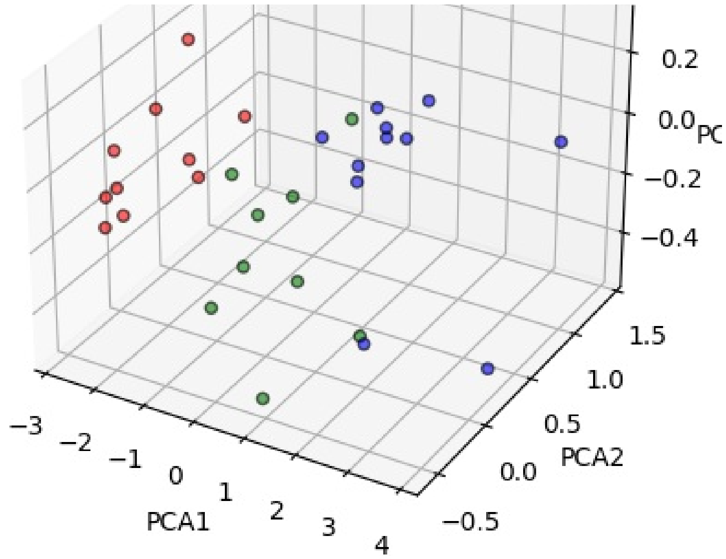
```
plot_decision_boundary_3D(lambda x: dnn_3d.feedforward(x), X_test_3d, y_test)
```



In [34]:

```
plot_decision_boundary_3D_no_hyperp(lambda x: dnn_3d.feedforward(x), X_test_3d, y_test)
```





Part 2

PyTorch Tutorial

In [35]:

```
import torch
import numpy as np
```

In [36]:

```
ndarray = np.array([0, 1, 2])
t = torch.from_numpy(ndarray)
print(t)

print(t.shape)
print(t.dtype)
print(t.device)
```

```
tensor([0, 1, 2])
torch.Size([3])
torch.int64
cpu
```

In [37]:

```
t = torch.tensor([0, 1, 2])
print(t)
```

```
tensor([0, 1, 2])
```

In [38]:

```
ndarray = np.array([[0, 1, 2], [3, 4, 5]])
t = torch.from_numpy(ndarray)
print(t)
```

```
tensor([[0, 1, 2],
        [3, 4, 5]])
```

In [39]:

```
new_t = torch.rand_like(t, dtype=torch.float)
print(new_t)
```

```
tensor([[0.9808, 0.2595, 0.0774],
        [0.0887, 0.7406, 0.3836]])
```

In [40]:

```
my_shape = (3, 3)
rand_t = torch.rand(my_shape)
print(rand_t)
```

```
tensor([[0.3803, 0.7897, 0.2752],
        [0.6571, 0.2989, 0.4621],
        [0.1073, 0.4017, 0.4276]])
```

In [41]:

```
zeros_tensor = torch.zeros((2, 3))
print(zeros_tensor)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

In [42]:

```
print(zeros_tensor[1])
print(zeros_tensor[:, 0])
```

```
tensor([0., 0., 0.])
tensor([0., 0.])
```

In [43]:

```
transposed = zeros_tensor.T
print(transposed)
```

```
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

In [44]:

```
ones_tensor = torch.ones(3, 3)
product = torch.matmul(zeros_tensor, ones_tensor)
print(product)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

In [45]:

```
import matplotlib.pyplot as plt
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.MNIST(root=".", train=True, download=True, transform=ToTensor())

test_data = datasets.MNIST(root=".", train=False, download=True, transform=ToTensor())
```

In [46]:

```
training_data[0]
```

Out[46]:

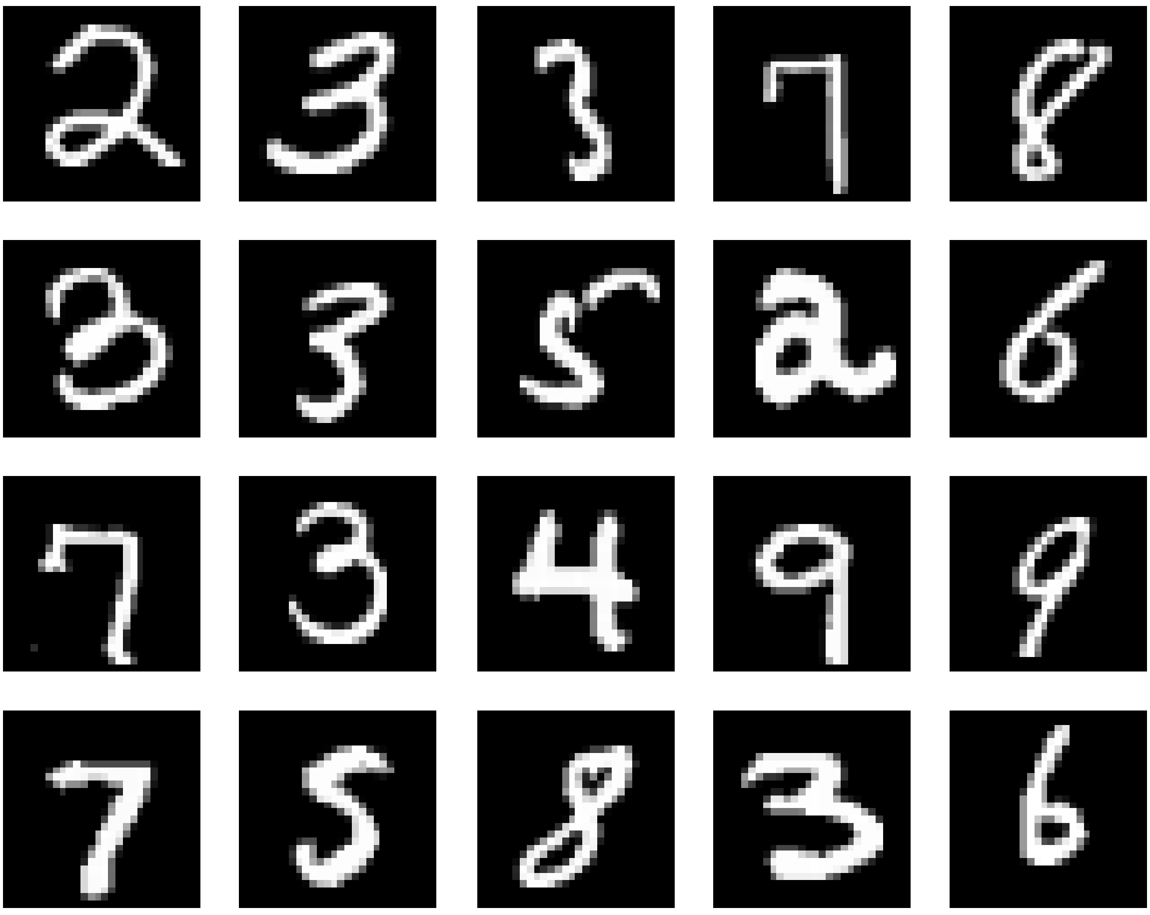
```
(tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
```

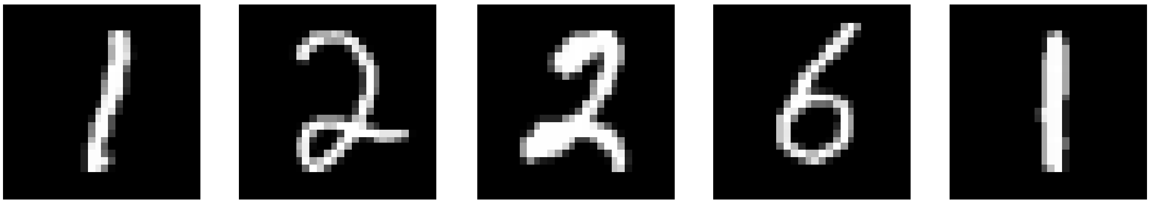
```
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0118, 0.0706, 0.0706, 0.0706,
 0.4941, 0.5333, 0.6863, 0.1020, 0.6510, 1.0000, 0.9686, 0.4980,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.1176, 0.1412, 0.3686, 0.6039, 0.6667, 0.9922, 0.9922, 0.9922,
 0.9922, 0.9922, 0.8824, 0.6745, 0.9922, 0.9490, 0.7647, 0.2510,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1922,
 0.9333, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922,
 0.9922, 0.9843, 0.3647, 0.3216, 0.3216, 0.2196, 0.1529, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0706,
 0.8588, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922, 0.7765, 0.7137,
 0.9686, 0.9451, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.3137, 0.6118, 0.4196, 0.9922, 0.9922, 0.8039, 0.0431, 0.0000,
 0.1686, 0.6039, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0549, 0.0039, 0.6039, 0.9922, 0.3529, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.5451, 0.9922, 0.7451, 0.0078, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0431, 0.7451, 0.9922, 0.2745, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.1373, 0.9451, 0.8824, 0.6275,
 0.4235, 0.0039, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.3176, 0.9412, 0.9922,
 0.9922, 0.4667, 0.0980, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1765, 0.7294,
 0.9922, 0.9922, 0.5882, 0.1059, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0627,
 0.3647, 0.9882, 0.9922, 0.7333, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.9765, 0.9922, 0.9765, 0.2510, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1804, 0.5098,
 0.7176, 0.9922, 0.9922, 0.8118, 0.0078, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.1529, 0.5804, 0.8980, 0.9922,
 0.9922, 0.9922, 0.9804, 0.7137, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0941, 0.4471, 0.8667, 0.9922, 0.9922, 0.9922,
 0.9922, 0.7882, 0.3059, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
```

```
         0.0902, 0.2588, 0.8353, 0.9922, 0.9922, 0.9922, 0.9922, 0.7765,
         0.3176, 0.0078, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0706, 0.6706,
         0.8588, 0.9922, 0.9922, 0.9922, 0.9922, 0.7647, 0.3137, 0.0353,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.2157, 0.6745, 0.8863, 0.9922,
         0.9922, 0.9922, 0.9922, 0.9569, 0.5216, 0.0431, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.5333, 0.9922, 0.9922, 0.9922,
         0.8314, 0.5294, 0.5176, 0.0627, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000]]]),
 5)
```

In [47]:

```
figure = plt.figure(figsize=(8, 8))
cols, rows = 5, 5

for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```





In [48]:

```
training_data.classes
```

Out[48]:

```
['0 - zero',
 '1 - one',
 '2 - two',
 '3 - three',
 '4 - four',
 '5 - five',
 '6 - six',
 '7 - seven',
 '8 - eight',
 '9 - nine']
```

In [49]:

```
from torch.utils.data import DataLoader

loaded_train = DataLoader(training_data, batch_size=64, shuffle=True)
loaded_test = DataLoader(test_data, batch_size=64, shuffle=True)
```

In [50]:

```
from torch import nn

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

In [51]:

```
model = NeuralNetwork()
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

In [52]:

```
loss_function = nn.CrossEntropyLoss()
```

In [53]:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

In [54]:

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        pred = model(X)
        loss = loss_fn(pred, y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 1000 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

In [55]:

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \
n")
```

In [56]:

```
torch.save(model, "model.pth")
model = torch.load("model.pth")
```

In [57]:

```
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(loaded_train, model, loss_function, optimizer)
    test(loaded_test, model, loss_function)
print("Done!")
```

```
Epoch 1
-------------------------------
loss: 2.313453  [    0/60000]
Test Error: 
 Accuracy: 12.5%, Avg loss: 2.304398

Epoch 2
-------------------------------
loss: 2.294496  [    0/60000]
Test Error: 
 Accuracy: 12.5%, Avg loss: 2.304418

Epoch 3
-------------------------------
loss: 2.309762  [    0/60000]
Test Error: 
 Accuracy: 12.5%, Avg loss: 2.304309
```

```
Epoch 4
-------------------------------
loss: 2.309878  [    0/60000]
Test Error:
 Accuracy: 12.5%, Avg loss: 2.304389

Epoch 5
-------------------------------
loss: 2.303256  [    0/60000]
Test Error:
 Accuracy: 12.5%, Avg loss: 2.304383

Done!
```

Writing Network Code

MNIST Data Wrangling

Importing libraries

In [58]:

```
import os
from datetime import datetime
from pathlib import Path

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from torch.utils.data import random_split
from torch.utils.tensorboard import SummaryWriter
from torchvision import datasets, transforms
```

Initializing hyperparameters/variables

In [59]:

```
batch_size = 64
test_batch_size = 1000
epochs = 10
lr = 0.01
try_cuda = True
seed = 1000
logging_interval = 10
logging_dir = None
```

Setting up the logging

In [60]:

```
datetime_str = datetime.now().strftime('%b%d_%H-%M-%S')

if logging_dir is None:
    base_folder = Path("./runs/")
    base_folder.mkdir(parents=True, exist_ok=True)
    logging_dir = base_folder / Path(datetime_str)
    logging_dir.mkdir(exist_ok=True)
    logging_dir = str(logging_dir.absolute())

writer = SummaryWriter(log_dir=logging_dir)
```

Deciding whether to send to the cpu or not if available

In [61]:

```
if torch.cuda.is_available() and try_cuda:
    cuda = True
    torch.cuda.mnaual_seed(seed)

else:
    cuda = False
    torch.manual_seed(seed)

print(cuda)
```

```
False
```

Setting up the data loaders

In [62]:

```
transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.01307,), (0.3081,))
])

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        'data',
        train=True,
        download=True,
        transform=transform,
        ),
    batch_size=batch_size,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        'data',
        train=False,
        download=True,
        transform=transform,
    ),
    batch_size=batch_size,
    shuffle=True,
)
```

In [63]:

```
len(train_loader), len(test_loader)
```

Out[63]:

```
(938, 157)
```

MNIST DCN

In [64]:

```
# Defining Architecture, loss, and optimizer
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        print(self.conv1, "\n", self.conv2)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)
```

```
        # super(Net, self).__init__()
        # self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        # self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        # self.fc1 = nn.Linear(1024, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)  # (batch_size, units)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        x = F.softmax(x, dim=1)
        # x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        # x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # x = x.view(-1, 1024) # (batch_size, units)
        # x = F.relu(self.fc1(x))
        # x = F.dropout(x, training=self.training)
        # x = F.softmax(x,dim=1)

        return x

model = Net()
if cuda:
    model.cuda()

optimizer = optim.Adam(model.parameters(), lr=lr)


# Visualize network as a graph on TensorBoard
input_tensor = torch.Tensor(1,1,28,28)
if cuda:
    input_tensor = input_tensor.cuda()
writer.add_graph(model, input_to_model=input_tensor)
```

```
Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
 Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
```

Training and Testing Functions

In [65]:

```
eps = 1e-13
```

defining the trainig loop

In [66]:

```
def train(epoch):
    model.train()
    criterion = nn.NLLLoss()
    #criterion = nn.CrossEntropyLoss()

    for batch_idx, (data, target) in enumerate(train_loader):
        if cuda:
            data, target = data.cuda(), target.cuda()

        optimizer.zero_grad()
        output = model(data) # forward
        loss = criterion(torch.log(output+eps), target) # = sum_k(-t_k * log(y_k))
        loss.backward()
        optimizer.step()

        if batch_idx % logging_interval == 0:
            print(f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.data
set)} \
                  ({100. * batch_idx / len(train_loader):.0f}%)]\tLoss: {loss.item():.
6f}')
```

```
            # Log train/loss to TensorBoard at every iteration
            n_iter = (epoch - 1) * len(train_loader) + batch_idx + 1
            writer.add_scalar('train/loss', loss.item(), n_iter)

    # Log model parameters to TensorBoard at every epoch
    for name, param in model.named_parameters():
        layer, attr = os.path.splitext(name)
        attr = attr[1:]
        writer.add_histogram(
            f'{layer}/{attr}',
            param.clone().cpu().data.numpy(),
            n_iter)
```

defining the testing loop

In [67]:

```
def test(epoch):
    model.eval()
    correct = 0
    test_loss = 0
    criterion = nn.NLLLoss(size_average = False)
    #criterion = nn.CrossEntropyLoss(size_average = False)

    for data, target in test_loader:
        if cuda:
            data, target = data.cuda(), target.cuda()

        output = model(data)

        # sum up batch loss (later, averaged over all test samples)
        test_loss += criterion(torch.log(output+eps), target,).item()

        # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]

        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    test_loss /= len(test_loader.dataset)
    test_accuracy = 100. * correct / len(test_loader.dataset)

    print(f'\nTest set: Average loss: {test_loss:.4f}, \
          Accuracy: {correct}/{len(test_loader.dataset)} \
          ({test_accuracy:.2f}%)\n')

    # Log test/loss and test/accuracy to TensorBoard at every epoch
    n_iter = epoch * len(train_loader)
    writer.add_scalar('test/loss', test_loss, n_iter)
    writer.add_scalar('test/accuracy', test_accuracy, n_iter)
```

Development Loop

In [68]:

```
for epoch in range(1, epochs + 1):
    train(epoch)
    test(epoch)

writer.close()
```

```
Train Epoch: 1 [0/60000                    (0%)] Loss: 2.337935
Train Epoch: 1 [640/60000                    (1%)] Loss: 1.974368
Train Epoch: 1 [1280/60000                    (2%)] Loss: 1.351631
Train Epoch: 1 [1920/60000                    (3%)] Loss: 1.224358
Train Epoch: 1 [2560/60000                    (4%)] Loss: 1.083730
Train Epoch: 1 [3200/60000                    (5%)] Loss: 1.212512
Train Epoch: 1 [3840/60000                    (6%)] Loss: 1.353354
Train Epoch: 1 [4480/60000                    (7%)] Loss: 0.698632
Train Epoch: 1 [5120/60000                    (9%)] Loss: 0.606955
Train Epoch: 1 [5760/60000                    (10%)] Loss: 0.806120
```

```
Train Epoch: 1 [5760/60000 (10%)] Loss: 0.896120
Train Epoch: 1 [6400/60000 (11%)] Loss: 0.943553
Train Epoch: 1 [7040/60000 (12%)] Loss: 0.737437
Train Epoch: 1 [7680/60000 (13%)] Loss: 0.858857
Train Epoch: 1 [8320/60000 (14%)] Loss: 0.597314
Train Epoch: 1 [8960/60000 (15%)] Loss: 0.677044
Train Epoch: 1 [9600/60000 (16%)] Loss: 0.364073
Train Epoch: 1 [10240/60000 (17%)] Loss: 0.530638
Train Epoch: 1 [10880/60000 (18%)] Loss: 0.737668
Train Epoch: 1 [11520/60000 (19%)] Loss: 0.739057
Train Epoch: 1 [12160/60000 (20%)] Loss: 0.425409
Train Epoch: 1 [12800/60000 (21%)] Loss: 0.664458
Train Epoch: 1 [13440/60000 (22%)] Loss: 0.554559
Train Epoch: 1 [14080/60000 (23%)] Loss: 0.659615
Train Epoch: 1 [14720/60000 (25%)] Loss: 0.221411
Train Epoch: 1 [15360/60000 (26%)] Loss: 0.446667
Train Epoch: 1 [16000/60000 (27%)] Loss: 0.482461
Train Epoch: 1 [16640/60000 (28%)] Loss: 0.459395
Train Epoch: 1 [17280/60000 (29%)] Loss: 0.538316
Train Epoch: 1 [17920/60000 (30%)] Loss: 0.284874
Train Epoch: 1 [18560/60000 (31%)] Loss: 0.571532
Train Epoch: 1 [19200/60000 (32%)] Loss: 0.772050
Train Epoch: 1 [19840/60000 (33%)] Loss: 0.339456
Train Epoch: 1 [20480/60000 (34%)] Loss: 0.922521
Train Epoch: 1 [21120/60000 (35%)] Loss: 0.556076
Train Epoch: 1 [21760/60000 (36%)] Loss: 0.703250
Train Epoch: 1 [22400/60000 (37%)] Loss: 0.364796
Train Epoch: 1 [23040/60000 (38%)] Loss: 0.365412
Train Epoch: 1 [23680/60000 (39%)] Loss: 0.187608
Train Epoch: 1 [24320/60000 (41%)] Loss: 0.362121
Train Epoch: 1 [24960/60000 (42%)] Loss: 0.468077
Train Epoch: 1 [25600/60000 (43%)] Loss: 0.445361
Train Epoch: 1 [26240/60000 (44%)] Loss: 0.443709
Train Epoch: 1 [26880/60000 (45%)] Loss: 0.370622
Train Epoch: 1 [27520/60000 (46%)] Loss: 0.482532
Train Epoch: 1 [28160/60000 (47%)] Loss: 0.403658
Train Epoch: 1 [28800/60000 (48%)] Loss: 0.366789
Train Epoch: 1 [29440/60000 (49%)] Loss: 0.378956
Train Epoch: 1 [30080/60000 (50%)] Loss: 0.337710
Train Epoch: 1 [30720/60000 (51%)] Loss: 0.482696
Train Epoch: 1 [31360/60000 (52%)] Loss: 0.370162
Train Epoch: 1 [32000/60000 (53%)] Loss: 0.412182
Train Epoch: 1 [32640/60000 (54%)] Loss: 0.732846
Train Epoch: 1 [33280/60000 (55%)] Loss: 0.436693
Train Epoch: 1 [33920/60000 (57%)] Loss: 0.342616
Train Epoch: 1 [34560/60000 (58%)] Loss: 0.423645
Train Epoch: 1 [35200/60000 (59%)] Loss: 0.282153
Train Epoch: 1 [35840/60000 (60%)] Loss: 0.381530
Train Epoch: 1 [36480/60000 (61%)] Loss: 0.756993
Train Epoch: 1 [37120/60000 (62%)] Loss: 0.464018
Train Epoch: 1 [37760/60000 (63%)] Loss: 0.662769
Train Epoch: 1 [38400/60000 (64%)] Loss: 0.494743
Train Epoch: 1 [39040/60000 (65%)] Loss: 0.329158
Train Epoch: 1 [39680/60000 (66%)] Loss: 0.421385
Train Epoch: 1 [40320/60000 (67%)] Loss: 0.390769
Train Epoch: 1 [40960/60000 (68%)] Loss: 0.397396
Train Epoch: 1 [41600/60000 (69%)] Loss: 0.476441
Train Epoch: 1 [42240/60000 (70%)] Loss: 0.671805
Train Epoch: 1 [42880/60000 (71%)] Loss: 0.505239
Train Epoch: 1 [43520/60000 (72%)] Loss: 0.508506
Train Epoch: 1 [44160/60000 (74%)] Loss: 0.414785
Train Epoch: 1 [44800/60000 (75%)] Loss: 0.349092
Train Epoch: 1 [45440/60000 (76%)] Loss: 0.461332
Train Epoch: 1 [46080/60000 (77%)] Loss: 0.302404
Train Epoch: 1 [46720/60000 (78%)] Loss: 0.282701
Train Epoch: 1 [47360/60000 (79%)] Loss: 0.679439
Train Epoch: 1 [48000/60000 (80%)] Loss: 0.597437
Train Epoch: 1 [48640/60000 (81%)] Loss: 0.339818
Train Epoch: 1 [49280/60000 (82%)] Loss: 0.481274
Train Epoch: 1 [49920/60000 (83%)] Loss: 0.559902
Train Epoch: 1 [50560/60000 (84%)] Loss: 0.571816
Train Epoch: 1 [51200/60000 (85%)] Loss: 0.733927
```

```
Train Epoch: 1 [51840/60000                    (86%)] Loss: 0.308710
Train Epoch: 1 [52480/60000                    (87%)] Loss: 0.300418
Train Epoch: 1 [53120/60000                    (88%)] Loss: 0.493699
Train Epoch: 1 [53760/60000                    (90%)] Loss: 0.383605
Train Epoch: 1 [54400/60000                    (91%)] Loss: 0.382422
Train Epoch: 1 [55040/60000                    (92%)] Loss: 0.336135
Train Epoch: 1 [55680/60000                    (93%)] Loss: 0.567889
Train Epoch: 1 [56320/60000                    (94%)] Loss: 0.301691
Train Epoch: 1 [56960/60000                    (95%)] Loss: 0.399286
Train Epoch: 1 [57600/60000                    (96%)] Loss: 0.303430
Train Epoch: 1 [58240/60000                    (97%)] Loss: 0.323305
Train Epoch: 1 [58880/60000                    (98%)] Loss: 0.322109
Train Epoch: 1 [59520/60000                    (99%)] Loss: 0.371338
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/_reduction.py:42: UserWarning: size_aver
age and reduce args will be deprecated, please use reduction='sum' instead.
  warnings.warn(warning.format(ret))
```

```
Test set: Average loss: 0.1208,             Accuracy: 9639/10000             (96.39%)

Train Epoch: 2 [0/60000                    (0%)] Loss: 0.337136
Train Epoch: 2 [640/60000                    (1%)] Loss: 0.479274
Train Epoch: 2 [1280/60000                    (2%)] Loss: 0.225007
Train Epoch: 2 [1920/60000                    (3%)] Loss: 0.384488
Train Epoch: 2 [2560/60000                    (4%)] Loss: 0.165874
Train Epoch: 2 [3200/60000                    (5%)] Loss: 0.128303
Train Epoch: 2 [3840/60000                    (6%)] Loss: 0.707741
Train Epoch: 2 [4480/60000                    (7%)] Loss: 0.321983
Train Epoch: 2 [5120/60000                    (9%)] Loss: 0.952283
Train Epoch: 2 [5760/60000                    (10%)] Loss: 0.260391
Train Epoch: 2 [6400/60000                    (11%)] Loss: 0.222487
Train Epoch: 2 [7040/60000                    (12%)] Loss: 0.469909
Train Epoch: 2 [7680/60000                    (13%)] Loss: 0.701855
Train Epoch: 2 [8320/60000                    (14%)] Loss: 0.179554
Train Epoch: 2 [8960/60000                    (15%)] Loss: 0.458847
Train Epoch: 2 [9600/60000                    (16%)] Loss: 0.513000
Train Epoch: 2 [10240/60000                    (17%)] Loss: 0.651630
Train Epoch: 2 [10880/60000                    (18%)] Loss: 0.669549
Train Epoch: 2 [11520/60000                    (19%)] Loss: 0.348878
Train Epoch: 2 [12160/60000                    (20%)] Loss: 0.384195
Train Epoch: 2 [12800/60000                    (21%)] Loss: 0.246505
Train Epoch: 2 [13440/60000                    (22%)] Loss: 0.257513
Train Epoch: 2 [14080/60000                    (23%)] Loss: 0.257987
Train Epoch: 2 [14720/60000                    (25%)] Loss: 0.363980
Train Epoch: 2 [15360/60000                    (26%)] Loss: 0.196778
Train Epoch: 2 [16000/60000                    (27%)] Loss: 0.248525
Train Epoch: 2 [16640/60000                    (28%)] Loss: 0.423651
Train Epoch: 2 [17280/60000                    (29%)] Loss: 0.338681
Train Epoch: 2 [17920/60000                    (30%)] Loss: 0.619952
Train Epoch: 2 [18560/60000                    (31%)] Loss: 0.292972
Train Epoch: 2 [19200/60000                    (32%)] Loss: 0.336553
Train Epoch: 2 [19840/60000                    (33%)] Loss: 0.751823
Train Epoch: 2 [20480/60000                    (34%)] Loss: 0.198866
Train Epoch: 2 [21120/60000                    (35%)] Loss: 0.582023
Train Epoch: 2 [21760/60000                    (36%)] Loss: 0.402101
Train Epoch: 2 [22400/60000                    (37%)] Loss: 0.284591
Train Epoch: 2 [23040/60000                    (38%)] Loss: 0.233107
Train Epoch: 2 [23680/60000                    (39%)] Loss: 0.322134
Train Epoch: 2 [24320/60000                    (41%)] Loss: 0.417522
Train Epoch: 2 [24960/60000                    (42%)] Loss: 0.460273
Train Epoch: 2 [25600/60000                    (43%)] Loss: 0.306307
Train Epoch: 2 [26240/60000                    (44%)] Loss: 0.762786
Train Epoch: 2 [26880/60000                    (45%)] Loss: 0.384040
Train Epoch: 2 [27520/60000                    (46%)] Loss: 0.343098
Train Epoch: 2 [28160/60000                    (47%)] Loss: 0.390398
Train Epoch: 2 [28800/60000                    (48%)] Loss: 0.239739
Train Epoch: 2 [29440/60000                    (49%)] Loss: 1.071127
Train Epoch: 2 [30080/60000                    (50%)] Loss: 0.501941
Train Epoch: 2 [30720/60000                    (51%)] Loss: 0.511122
Train Epoch: 2 [31360/60000                    (52%)] Loss: 0.303646
Train Epoch: 2 [32000/60000                    (53%)] Loss: 0.385205
Train Epoch: 2 [32640/60000                    (54%)] Loss: 0.554963
Train Epoch: 2 [33280/60000                    (55%)] Loss: 0.302876
```

```
Train Epoch: 2 [33920/60000 (57%)] Loss: 0.196659
Train Epoch: 2 [34560/60000 (58%)] Loss: 0.329665
Train Epoch: 2 [35200/60000 (59%)] Loss: 0.352431
Train Epoch: 2 [35840/60000 (60%)] Loss: 0.617813
Train Epoch: 2 [36480/60000 (61%)] Loss: 0.228329
Train Epoch: 2 [37120/60000 (62%)] Loss: 0.422546
Train Epoch: 2 [37760/60000 (63%)] Loss: 0.596287
Train Epoch: 2 [38400/60000 (64%)] Loss: 0.616060
Train Epoch: 2 [39040/60000 (65%)] Loss: 0.257114
Train Epoch: 2 [39680/60000 (66%)] Loss: 0.326699
Train Epoch: 2 [40320/60000 (67%)] Loss: 0.466682
Train Epoch: 2 [40960/60000 (68%)] Loss: 0.439112
Train Epoch: 2 [41600/60000 (69%)] Loss: 0.240317
Train Epoch: 2 [42240/60000 (70%)] Loss: 0.140975
Train Epoch: 2 [42880/60000 (71%)] Loss: 0.256579
Train Epoch: 2 [43520/60000 (72%)] Loss: 0.506550
Train Epoch: 2 [44160/60000 (74%)] Loss: 0.488903
Train Epoch: 2 [44800/60000 (75%)] Loss: 0.220152
Train Epoch: 2 [45440/60000 (76%)] Loss: 0.292367
Train Epoch: 2 [46080/60000 (77%)] Loss: 0.481088
Train Epoch: 2 [46720/60000 (78%)] Loss: 0.433307
Train Epoch: 2 [47360/60000 (79%)] Loss: 0.402098
Train Epoch: 2 [48000/60000 (80%)] Loss: 0.668445
Train Epoch: 2 [48640/60000 (81%)] Loss: 0.679566
Train Epoch: 2 [49280/60000 (82%)] Loss: 0.311963
Train Epoch: 2 [49920/60000 (83%)] Loss: 0.259018
Train Epoch: 2 [50560/60000 (84%)] Loss: 0.291222
Train Epoch: 2 [51200/60000 (85%)] Loss: 0.308323
Train Epoch: 2 [51840/60000 (86%)] Loss: 0.265978
Train Epoch: 2 [52480/60000 (87%)] Loss: 0.363620
Train Epoch: 2 [53120/60000 (88%)] Loss: 0.346468
Train Epoch: 2 [53760/60000 (90%)] Loss: 0.625935
Train Epoch: 2 [54400/60000 (91%)] Loss: 0.235442
Train Epoch: 2 [55040/60000 (92%)] Loss: 0.286900
Train Epoch: 2 [55680/60000 (93%)] Loss: 0.358709
Train Epoch: 2 [56320/60000 (94%)] Loss: 0.276832
Train Epoch: 2 [56960/60000 (95%)] Loss: 0.531221
Train Epoch: 2 [57600/60000 (96%)] Loss: 0.510090
Train Epoch: 2 [58240/60000 (97%)] Loss: 0.500545
Train Epoch: 2 [58880/60000 (98%)] Loss: 0.525299
Train Epoch: 2 [59520/60000 (99%)] Loss: 0.421123

Test set: Average loss: 0.1304, Accuracy: 9624/10000 (96.24%)

Train Epoch: 3 [0/60000 (0%)] Loss: 0.143986
Train Epoch: 3 [640/60000 (1%)] Loss: 0.695911
Train Epoch: 3 [1280/60000 (2%)] Loss: 0.530430
Train Epoch: 3 [1920/60000 (3%)] Loss: 0.371876
Train Epoch: 3 [2560/60000 (4%)] Loss: 0.351873
Train Epoch: 3 [3200/60000 (5%)] Loss: 0.097900
Train Epoch: 3 [3840/60000 (6%)] Loss: 0.373057
Train Epoch: 3 [4480/60000 (7%)] Loss: 0.328354
Train Epoch: 3 [5120/60000 (9%)] Loss: 0.349329
Train Epoch: 3 [5760/60000 (10%)] Loss: 0.465759
Train Epoch: 3 [6400/60000 (11%)] Loss: 0.409909
Train Epoch: 3 [7040/60000 (12%)] Loss: 0.345538
Train Epoch: 3 [7680/60000 (13%)] Loss: 0.283638
Train Epoch: 3 [8320/60000 (14%)] Loss: 0.635366
Train Epoch: 3 [8960/60000 (15%)] Loss: 0.467434
Train Epoch: 3 [9600/60000 (16%)] Loss: 0.415051
Train Epoch: 3 [10240/60000 (17%)] Loss: 0.300160
Train Epoch: 3 [10880/60000 (18%)] Loss: 0.399905
Train Epoch: 3 [11520/60000 (19%)] Loss: 0.157745
Train Epoch: 3 [12160/60000 (20%)] Loss: 0.197733
Train Epoch: 3 [12800/60000 (21%)] Loss: 0.226561
Train Epoch: 3 [13440/60000 (22%)] Loss: 0.518948
Train Epoch: 3 [14080/60000 (23%)] Loss: 0.666111
Train Epoch: 3 [14720/60000 (25%)] Loss: 0.366006
Train Epoch: 3 [15360/60000 (26%)] Loss: 0.197712
Train Epoch: 3 [16000/60000 (27%)] Loss: 0.382115
Train Epoch: 3 [16640/60000 (28%)] Loss: 0.351189
Train Epoch: 3 [17280/60000 (29%)] Loss: 0.263923
```

```
Train Epoch: 3 [17920/60000         (30%)] Loss: 0.495450
Train Epoch: 3 [18560/60000         (31%)] Loss: 0.289360
Train Epoch: 3 [19200/60000         (32%)] Loss: 0.170334
Train Epoch: 3 [19840/60000         (33%)] Loss: 0.391904
Train Epoch: 3 [20480/60000         (34%)] Loss: 0.346157
Train Epoch: 3 [21120/60000         (35%)] Loss: 0.383141
Train Epoch: 3 [21760/60000         (36%)] Loss: 0.318312
Train Epoch: 3 [22400/60000         (37%)] Loss: 0.685658
Train Epoch: 3 [23040/60000         (38%)] Loss: 0.377422
Train Epoch: 3 [23680/60000         (39%)] Loss: 0.305419
Train Epoch: 3 [24320/60000         (41%)] Loss: 0.240928
Train Epoch: 3 [24960/60000         (42%)] Loss: 0.405327
Train Epoch: 3 [25600/60000         (43%)] Loss: 0.526315
Train Epoch: 3 [26240/60000         (44%)] Loss: 0.395687
Train Epoch: 3 [26880/60000         (45%)] Loss: 0.338942
Train Epoch: 3 [27520/60000         (46%)] Loss: 0.505238
Train Epoch: 3 [28160/60000         (47%)] Loss: 0.210290
Train Epoch: 3 [28800/60000         (48%)] Loss: 0.333995
Train Epoch: 3 [29440/60000         (49%)] Loss: 0.370425
Train Epoch: 3 [30080/60000         (50%)] Loss: 0.231636
Train Epoch: 3 [30720/60000         (51%)] Loss: 0.621049
Train Epoch: 3 [31360/60000         (52%)] Loss: 0.530123
Train Epoch: 3 [32000/60000         (53%)] Loss: 0.413550
Train Epoch: 3 [32640/60000         (54%)] Loss: 0.135045
Train Epoch: 3 [33280/60000         (55%)] Loss: 0.371043
Train Epoch: 3 [33920/60000         (57%)] Loss: 0.378734
Train Epoch: 3 [34560/60000         (58%)] Loss: 0.401920
Train Epoch: 3 [35200/60000         (59%)] Loss: 0.367261
Train Epoch: 3 [35840/60000         (60%)] Loss: 0.283215
Train Epoch: 3 [36480/60000         (61%)] Loss: 0.571325
Train Epoch: 3 [37120/60000         (62%)] Loss: 0.466318
Train Epoch: 3 [37760/60000         (63%)] Loss: 0.189224
Train Epoch: 3 [38400/60000         (64%)] Loss: 0.234057
Train Epoch: 3 [39040/60000         (65%)] Loss: 0.417677
Train Epoch: 3 [39680/60000         (66%)] Loss: 0.305710
Train Epoch: 3 [40320/60000         (67%)] Loss: 0.340737
Train Epoch: 3 [40960/60000         (68%)] Loss: 0.489766
Train Epoch: 3 [41600/60000         (69%)] Loss: 0.279524
Train Epoch: 3 [42240/60000         (70%)] Loss: 0.418107
Train Epoch: 3 [42880/60000         (71%)] Loss: 0.512998
Train Epoch: 3 [43520/60000         (72%)] Loss: 0.474564
Train Epoch: 3 [44160/60000         (74%)] Loss: 0.255928
Train Epoch: 3 [44800/60000         (75%)] Loss: 0.322918
Train Epoch: 3 [45440/60000         (76%)] Loss: 0.177344
Train Epoch: 3 [46080/60000         (77%)] Loss: 0.549125
Train Epoch: 3 [46720/60000         (78%)] Loss: 0.399815
Train Epoch: 3 [47360/60000         (79%)] Loss: 0.271971
Train Epoch: 3 [48000/60000         (80%)] Loss: 0.456737
Train Epoch: 3 [48640/60000         (81%)] Loss: 0.383172
Train Epoch: 3 [49280/60000         (82%)] Loss: 0.172426
Train Epoch: 3 [49920/60000         (83%)] Loss: 0.482662
Train Epoch: 3 [50560/60000         (84%)] Loss: 0.321574
Train Epoch: 3 [51200/60000         (85%)] Loss: 0.519809
Train Epoch: 3 [51840/60000         (86%)] Loss: 0.324275
Train Epoch: 3 [52480/60000         (87%)] Loss: 0.235553
Train Epoch: 3 [53120/60000         (88%)] Loss: 0.563037
Train Epoch: 3 [53760/60000         (90%)] Loss: 0.550453
Train Epoch: 3 [54400/60000         (91%)] Loss: 0.623710
Train Epoch: 3 [55040/60000         (92%)] Loss: 0.599679
Train Epoch: 3 [55680/60000         (93%)] Loss: 0.301909
Train Epoch: 3 [56320/60000         (94%)] Loss: 0.355210
Train Epoch: 3 [56960/60000         (95%)] Loss: 0.267676
Train Epoch: 3 [57600/60000         (96%)] Loss: 0.635964
Train Epoch: 3 [58240/60000         (97%)] Loss: 0.465695
Train Epoch: 3 [58880/60000         (98%)] Loss: 0.350577
Train Epoch: 3 [59520/60000         (99%)] Loss: 0.709676

Test set: Average loss: 0.1059,         Accuracy: 9702/10000          (97.02%)

Train Epoch: 4 [0/60000         (0%)] Loss: 0.108460
Train Epoch: 4 [640/60000         (1%)] Loss: 0.425560
Train Epoch: 4 [1280/60000         (2%)] Loss: 0.254301
```

```
Train Epoch: 4 [1920/60000 (3%)] Loss: 0.428108
Train Epoch: 4 [2560/60000 (4%)] Loss: 0.343296
Train Epoch: 4 [3200/60000 (5%)] Loss: 0.369928
Train Epoch: 4 [3840/60000 (6%)] Loss: 0.300075
Train Epoch: 4 [4480/60000 (7%)] Loss: 0.872673
Train Epoch: 4 [5120/60000 (9%)] Loss: 0.695217
Train Epoch: 4 [5760/60000 (10%)] Loss: 0.225474
Train Epoch: 4 [6400/60000 (11%)] Loss: 0.523622
Train Epoch: 4 [7040/60000 (12%)] Loss: 0.495387
Train Epoch: 4 [7680/60000 (13%)] Loss: 0.440046
Train Epoch: 4 [8320/60000 (14%)] Loss: 0.232337
Train Epoch: 4 [8960/60000 (15%)] Loss: 0.160292
Train Epoch: 4 [9600/60000 (16%)] Loss: 0.446541
Train Epoch: 4 [10240/60000 (17%)] Loss: 0.382764
Train Epoch: 4 [10880/60000 (18%)] Loss: 0.400303
Train Epoch: 4 [11520/60000 (19%)] Loss: 0.372205
Train Epoch: 4 [12160/60000 (20%)] Loss: 0.523375
Train Epoch: 4 [12800/60000 (21%)] Loss: 0.293484
Train Epoch: 4 [13440/60000 (22%)] Loss: 0.205108
Train Epoch: 4 [14080/60000 (23%)] Loss: 0.522204
Train Epoch: 4 [14720/60000 (25%)] Loss: 0.269795
Train Epoch: 4 [15360/60000 (26%)] Loss: 0.544264
Train Epoch: 4 [16000/60000 (27%)] Loss: 0.407836
Train Epoch: 4 [16640/60000 (28%)] Loss: 0.303444
Train Epoch: 4 [17280/60000 (29%)] Loss: 0.433247
Train Epoch: 4 [17920/60000 (30%)] Loss: 0.123767
Train Epoch: 4 [18560/60000 (31%)] Loss: 0.503479
Train Epoch: 4 [19200/60000 (32%)] Loss: 0.199913
Train Epoch: 4 [19840/60000 (33%)] Loss: 0.248016
Train Epoch: 4 [20480/60000 (34%)] Loss: 0.321681
Train Epoch: 4 [21120/60000 (35%)] Loss: 0.379008
Train Epoch: 4 [21760/60000 (36%)] Loss: 0.464656
Train Epoch: 4 [22400/60000 (37%)] Loss: 0.300058
Train Epoch: 4 [23040/60000 (38%)] Loss: 0.800661
Train Epoch: 4 [23680/60000 (39%)] Loss: 0.210334
Train Epoch: 4 [24320/60000 (41%)] Loss: 0.269489
Train Epoch: 4 [24960/60000 (42%)] Loss: 0.247709
Train Epoch: 4 [25600/60000 (43%)] Loss: 0.372970
Train Epoch: 4 [26240/60000 (44%)] Loss: 0.535515
Train Epoch: 4 [26880/60000 (45%)] Loss: 0.174940
Train Epoch: 4 [27520/60000 (46%)] Loss: 0.352128
Train Epoch: 4 [28160/60000 (47%)] Loss: 0.293153
Train Epoch: 4 [28800/60000 (48%)] Loss: 0.175991
Train Epoch: 4 [29440/60000 (49%)] Loss: 0.236847
Train Epoch: 4 [30080/60000 (50%)] Loss: 0.241978
Train Epoch: 4 [30720/60000 (51%)] Loss: 0.516029
Train Epoch: 4 [31360/60000 (52%)] Loss: 0.362029
Train Epoch: 4 [32000/60000 (53%)] Loss: 0.425949
Train Epoch: 4 [32640/60000 (54%)] Loss: 0.669070
Train Epoch: 4 [33280/60000 (55%)] Loss: 0.340179
Train Epoch: 4 [33920/60000 (57%)] Loss: 0.419187
Train Epoch: 4 [34560/60000 (58%)] Loss: 0.487478
Train Epoch: 4 [35200/60000 (59%)] Loss: 0.384564
Train Epoch: 4 [35840/60000 (60%)] Loss: 0.377897
Train Epoch: 4 [36480/60000 (61%)] Loss: 0.207883
Train Epoch: 4 [37120/60000 (62%)] Loss: 0.308626
Train Epoch: 4 [37760/60000 (63%)] Loss: 0.200205
Train Epoch: 4 [38400/60000 (64%)] Loss: 0.181627
Train Epoch: 4 [39040/60000 (65%)] Loss: 0.257840
Train Epoch: 4 [39680/60000 (66%)] Loss: 0.307480
Train Epoch: 4 [40320/60000 (67%)] Loss: 0.447075
Train Epoch: 4 [40960/60000 (68%)] Loss: 0.398623
Train Epoch: 4 [41600/60000 (69%)] Loss: 0.275165
Train Epoch: 4 [42240/60000 (70%)] Loss: 0.248029
Train Epoch: 4 [42880/60000 (71%)] Loss: 0.421756
Train Epoch: 4 [43520/60000 (72%)] Loss: 0.270988
Train Epoch: 4 [44160/60000 (74%)] Loss: 0.254186
Train Epoch: 4 [44800/60000 (75%)] Loss: 0.390284
Train Epoch: 4 [45440/60000 (76%)] Loss: 0.169280
Train Epoch: 4 [46080/60000 (77%)] Loss: 0.644407
Train Epoch: 4 [46720/60000 (78%)] Loss: 0.405275
Train Epoch: 4 [47360/60000 (79%)] Loss: 0.357460
```

```
Train Epoch: 4 [48000/60000 (80%)] Loss: 0.198657
Train Epoch: 4 [48640/60000 (81%)] Loss: 0.227988
Train Epoch: 4 [49280/60000 (82%)] Loss: 0.383270
Train Epoch: 4 [49920/60000 (83%)] Loss: 0.220222
Train Epoch: 4 [50560/60000 (84%)] Loss: 0.256078
Train Epoch: 4 [51200/60000 (85%)] Loss: 0.445944
Train Epoch: 4 [51840/60000 (86%)] Loss: 0.456606
Train Epoch: 4 [52480/60000 (87%)] Loss: 0.145958
Train Epoch: 4 [53120/60000 (88%)] Loss: 0.171615
Train Epoch: 4 [53760/60000 (90%)] Loss: 0.475618
Train Epoch: 4 [54400/60000 (91%)] Loss: 0.323357
Train Epoch: 4 [55040/60000 (92%)] Loss: 0.441469
Train Epoch: 4 [55680/60000 (93%)] Loss: 0.225519
Train Epoch: 4 [56320/60000 (94%)] Loss: 0.559752
Train Epoch: 4 [56960/60000 (95%)] Loss: 0.432842
Train Epoch: 4 [57600/60000 (96%)] Loss: 0.419061
Train Epoch: 4 [58240/60000 (97%)] Loss: 0.505451
Train Epoch: 4 [58880/60000 (98%)] Loss: 0.318146
Train Epoch: 4 [59520/60000 (99%)] Loss: 0.496282

Test set: Average loss: 0.1069, Accuracy: 9671/10000 (96.71%)

Train Epoch: 5 [0/60000 (0%)] Loss: 0.279480
Train Epoch: 5 [640/60000 (1%)] Loss: 0.583582
Train Epoch: 5 [1280/60000 (2%)] Loss: 0.114471
Train Epoch: 5 [1920/60000 (3%)] Loss: 0.228138
Train Epoch: 5 [2560/60000 (4%)] Loss: 0.360272
Train Epoch: 5 [3200/60000 (5%)] Loss: 0.232679
Train Epoch: 5 [3840/60000 (6%)] Loss: 0.380263
Train Epoch: 5 [4480/60000 (7%)] Loss: 0.464436
Train Epoch: 5 [5120/60000 (9%)] Loss: 0.382125
Train Epoch: 5 [5760/60000 (10%)] Loss: 0.271823
Train Epoch: 5 [6400/60000 (11%)] Loss: 0.368081
Train Epoch: 5 [7040/60000 (12%)] Loss: 0.531454
Train Epoch: 5 [7680/60000 (13%)] Loss: 0.312927
Train Epoch: 5 [8320/60000 (14%)] Loss: 0.164462
Train Epoch: 5 [8960/60000 (15%)] Loss: 0.349191
Train Epoch: 5 [9600/60000 (16%)] Loss: 0.238632
Train Epoch: 5 [10240/60000 (17%)] Loss: 0.426839
Train Epoch: 5 [10880/60000 (18%)] Loss: 0.415183
Train Epoch: 5 [11520/60000 (19%)] Loss: 0.266665
Train Epoch: 5 [12160/60000 (20%)] Loss: 0.225294
Train Epoch: 5 [12800/60000 (21%)] Loss: 0.339763
Train Epoch: 5 [13440/60000 (22%)] Loss: 0.529783
Train Epoch: 5 [14080/60000 (23%)] Loss: 0.500715
Train Epoch: 5 [14720/60000 (25%)] Loss: 0.245331
Train Epoch: 5 [15360/60000 (26%)] Loss: 0.436559
Train Epoch: 5 [16000/60000 (27%)] Loss: 0.481812
Train Epoch: 5 [16640/60000 (28%)] Loss: 0.211769
Train Epoch: 5 [17280/60000 (29%)] Loss: 0.550403
Train Epoch: 5 [17920/60000 (30%)] Loss: 0.350280
Train Epoch: 5 [18560/60000 (31%)] Loss: 0.155272
Train Epoch: 5 [19200/60000 (32%)] Loss: 0.522046
Train Epoch: 5 [19840/60000 (33%)] Loss: 0.385714
Train Epoch: 5 [20480/60000 (34%)] Loss: 0.371923
Train Epoch: 5 [21120/60000 (35%)] Loss: 0.233176
Train Epoch: 5 [21760/60000 (36%)] Loss: 0.488653
Train Epoch: 5 [22400/60000 (37%)] Loss: 0.258055
Train Epoch: 5 [23040/60000 (38%)] Loss: 0.412481
Train Epoch: 5 [23680/60000 (39%)] Loss: 0.142940
Train Epoch: 5 [24320/60000 (41%)] Loss: 0.409342
Train Epoch: 5 [24960/60000 (42%)] Loss: 0.098702
Train Epoch: 5 [25600/60000 (43%)] Loss: 0.525860
Train Epoch: 5 [26240/60000 (44%)] Loss: 0.253256
Train Epoch: 5 [26880/60000 (45%)] Loss: 0.363933
Train Epoch: 5 [27520/60000 (46%)] Loss: 0.194624
Train Epoch: 5 [28160/60000 (47%)] Loss: 0.347276
Train Epoch: 5 [28800/60000 (48%)] Loss: 0.336600
Train Epoch: 5 [29440/60000 (49%)] Loss: 0.562904
Train Epoch: 5 [30080/60000 (50%)] Loss: 0.330007
Train Epoch: 5 [30720/60000 (51%)] Loss: 0.622185
Train Epoch: 5 [31360/60000 (52%)] Loss: 0.307415
```

```
Train Epoch: 5 [32000/60000 (53%)] Loss: 0.447527
Train Epoch: 5 [32640/60000 (54%)] Loss: 0.341777
Train Epoch: 5 [33280/60000 (55%)] Loss: 0.324765
Train Epoch: 5 [33920/60000 (57%)] Loss: 0.290968
Train Epoch: 5 [34560/60000 (58%)] Loss: 0.237752
Train Epoch: 5 [35200/60000 (59%)] Loss: 0.512075
Train Epoch: 5 [35840/60000 (60%)] Loss: 0.248750
Train Epoch: 5 [36480/60000 (61%)] Loss: 0.354318
Train Epoch: 5 [37120/60000 (62%)] Loss: 0.256978
Train Epoch: 5 [37760/60000 (63%)] Loss: 0.343810
Train Epoch: 5 [38400/60000 (64%)] Loss: 0.502602
Train Epoch: 5 [39040/60000 (65%)] Loss: 0.257192
Train Epoch: 5 [39680/60000 (66%)] Loss: 0.522747
Train Epoch: 5 [40320/60000 (67%)] Loss: 0.621672
Train Epoch: 5 [40960/60000 (68%)] Loss: 0.319940
Train Epoch: 5 [41600/60000 (69%)] Loss: 0.288031
Train Epoch: 5 [42240/60000 (70%)] Loss: 0.702151
Train Epoch: 5 [42880/60000 (71%)] Loss: 0.452771
Train Epoch: 5 [43520/60000 (72%)] Loss: 0.431340
Train Epoch: 5 [44160/60000 (74%)] Loss: 0.407420
Train Epoch: 5 [44800/60000 (75%)] Loss: 0.443665
Train Epoch: 5 [45440/60000 (76%)] Loss: 0.375876
Train Epoch: 5 [46080/60000 (77%)] Loss: 0.382248
Train Epoch: 5 [46720/60000 (78%)] Loss: 0.479976
Train Epoch: 5 [47360/60000 (79%)] Loss: 0.266127
Train Epoch: 5 [48000/60000 (80%)] Loss: 0.140895
Train Epoch: 5 [48640/60000 (81%)] Loss: 0.198195
Train Epoch: 5 [49280/60000 (82%)] Loss: 0.394249
Train Epoch: 5 [49920/60000 (83%)] Loss: 0.207947
Train Epoch: 5 [50560/60000 (84%)] Loss: 0.310662
Train Epoch: 5 [51200/60000 (85%)] Loss: 0.398560
Train Epoch: 5 [51840/60000 (86%)] Loss: 0.389851
Train Epoch: 5 [52480/60000 (87%)] Loss: 0.092229
Train Epoch: 5 [53120/60000 (88%)] Loss: 0.525212
Train Epoch: 5 [53760/60000 (90%)] Loss: 0.402774
Train Epoch: 5 [54400/60000 (91%)] Loss: 0.538168
Train Epoch: 5 [55040/60000 (92%)] Loss: 0.816550
Train Epoch: 5 [55680/60000 (93%)] Loss: 0.287489
Train Epoch: 5 [56320/60000 (94%)] Loss: 0.393158
Train Epoch: 5 [56960/60000 (95%)] Loss: 0.186453
Train Epoch: 5 [57600/60000 (96%)] Loss: 0.339153
Train Epoch: 5 [58240/60000 (97%)] Loss: 0.248818
Train Epoch: 5 [58880/60000 (98%)] Loss: 0.302968
Train Epoch: 5 [59520/60000 (99%)] Loss: 0.388227

Test set: Average loss: 0.1036, Accuracy: 9696/10000 (96.96%)

Train Epoch: 6 [0/60000 (0%)] Loss: 0.229695
Train Epoch: 6 [640/60000 (1%)] Loss: 0.387925
Train Epoch: 6 [1280/60000 (2%)] Loss: 0.312874
Train Epoch: 6 [1920/60000 (3%)] Loss: 0.200146
Train Epoch: 6 [2560/60000 (4%)] Loss: 0.297047
Train Epoch: 6 [3200/60000 (5%)] Loss: 0.221484
Train Epoch: 6 [3840/60000 (6%)] Loss: 0.309829
Train Epoch: 6 [4480/60000 (7%)] Loss: 0.144698
Train Epoch: 6 [5120/60000 (9%)] Loss: 0.417942
Train Epoch: 6 [5760/60000 (10%)] Loss: 0.235661
Train Epoch: 6 [6400/60000 (11%)] Loss: 0.290786
Train Epoch: 6 [7040/60000 (12%)] Loss: 0.237960
Train Epoch: 6 [7680/60000 (13%)] Loss: 0.575022
Train Epoch: 6 [8320/60000 (14%)] Loss: 0.300228
Train Epoch: 6 [8960/60000 (15%)] Loss: 0.473434
Train Epoch: 6 [9600/60000 (16%)] Loss: 0.335970
Train Epoch: 6 [10240/60000 (17%)] Loss: 0.276764
Train Epoch: 6 [10880/60000 (18%)] Loss: 0.099117
Train Epoch: 6 [11520/60000 (19%)] Loss: 0.360143
Train Epoch: 6 [12160/60000 (20%)] Loss: 0.526211
Train Epoch: 6 [12800/60000 (21%)] Loss: 0.303238
Train Epoch: 6 [13440/60000 (22%)] Loss: 0.338592
Train Epoch: 6 [14080/60000 (23%)] Loss: 0.215090
Train Epoch: 6 [14720/60000 (25%)] Loss: 0.322847
Train Epoch: 6 [15360/60000 (26%)] Loss: 0.153402
```

```
Train Epoch: 6 [16000/60000          (27%)] Loss: 0.533954
Train Epoch: 6 [16640/60000          (28%)] Loss: 0.487382
Train Epoch: 6 [17280/60000          (29%)] Loss: 0.208998
Train Epoch: 6 [17920/60000          (30%)] Loss: 0.764552
Train Epoch: 6 [18560/60000          (31%)] Loss: 0.512401
Train Epoch: 6 [19200/60000          (32%)] Loss: 0.375841
Train Epoch: 6 [19840/60000          (33%)] Loss: 0.616413
Train Epoch: 6 [20480/60000          (34%)] Loss: 0.341068
Train Epoch: 6 [21120/60000          (35%)] Loss: 0.385144
Train Epoch: 6 [21760/60000          (36%)] Loss: 0.393741
Train Epoch: 6 [22400/60000          (37%)] Loss: 0.252085
Train Epoch: 6 [23040/60000          (38%)] Loss: 0.495450
Train Epoch: 6 [23680/60000          (39%)] Loss: 0.440192
Train Epoch: 6 [24320/60000          (41%)] Loss: 0.251723
Train Epoch: 6 [24960/60000          (42%)] Loss: 0.291375
Train Epoch: 6 [25600/60000          (43%)] Loss: 0.420710
Train Epoch: 6 [26240/60000          (44%)] Loss: 0.472655
Train Epoch: 6 [26880/60000          (45%)] Loss: 0.427120
Train Epoch: 6 [27520/60000          (46%)] Loss: 0.308083
Train Epoch: 6 [28160/60000          (47%)] Loss: 0.216019
Train Epoch: 6 [28800/60000          (48%)] Loss: 0.323822
Train Epoch: 6 [29440/60000          (49%)] Loss: 0.671619
Train Epoch: 6 [30080/60000          (50%)] Loss: 0.357273
Train Epoch: 6 [30720/60000          (51%)] Loss: 0.260498
Train Epoch: 6 [31360/60000          (52%)] Loss: 0.404606
Train Epoch: 6 [32000/60000          (53%)] Loss: 0.419295
Train Epoch: 6 [32640/60000          (54%)] Loss: 0.350933
Train Epoch: 6 [33280/60000          (55%)] Loss: 0.532115
Train Epoch: 6 [33920/60000          (57%)] Loss: 0.350708
Train Epoch: 6 [34560/60000          (58%)] Loss: 0.283400
Train Epoch: 6 [35200/60000          (59%)] Loss: 0.629881
Train Epoch: 6 [35840/60000          (60%)] Loss: 0.216300
Train Epoch: 6 [36480/60000          (61%)] Loss: 0.136666
Train Epoch: 6 [37120/60000          (62%)] Loss: 0.283540
Train Epoch: 6 [37760/60000          (63%)] Loss: 0.514523
Train Epoch: 6 [38400/60000          (64%)] Loss: 0.287523
Train Epoch: 6 [39040/60000          (65%)] Loss: 0.421855
Train Epoch: 6 [39680/60000          (66%)] Loss: 0.610001
Train Epoch: 6 [40320/60000          (67%)] Loss: 0.255489
Train Epoch: 6 [40960/60000          (68%)] Loss: 0.312748
Train Epoch: 6 [41600/60000          (69%)] Loss: 0.666175
Train Epoch: 6 [42240/60000          (70%)] Loss: 0.220162
Train Epoch: 6 [42880/60000          (71%)] Loss: 0.450981
Train Epoch: 6 [43520/60000          (72%)] Loss: 0.731593
Train Epoch: 6 [44160/60000          (74%)] Loss: 0.464104
Train Epoch: 6 [44800/60000          (75%)] Loss: 0.486290
Train Epoch: 6 [45440/60000          (76%)] Loss: 0.571150
Train Epoch: 6 [46080/60000          (77%)] Loss: 0.344816
Train Epoch: 6 [46720/60000          (78%)] Loss: 0.173954
Train Epoch: 6 [47360/60000          (79%)] Loss: 0.434259
Train Epoch: 6 [48000/60000          (80%)] Loss: 0.379379
Train Epoch: 6 [48640/60000          (81%)] Loss: 0.200702
Train Epoch: 6 [49280/60000          (82%)] Loss: 0.242128
Train Epoch: 6 [49920/60000          (83%)] Loss: 0.264166
Train Epoch: 6 [50560/60000          (84%)] Loss: 0.370711
Train Epoch: 6 [51200/60000          (85%)] Loss: 0.694037
Train Epoch: 6 [51840/60000          (86%)] Loss: 0.308189
Train Epoch: 6 [52480/60000          (87%)] Loss: 0.292568
Train Epoch: 6 [53120/60000          (88%)] Loss: 0.224421
Train Epoch: 6 [53760/60000          (90%)] Loss: 0.394205
Train Epoch: 6 [54400/60000          (91%)] Loss: 0.591992
Train Epoch: 6 [55040/60000          (92%)] Loss: 0.283203
Train Epoch: 6 [55680/60000          (93%)] Loss: 0.455354
Train Epoch: 6 [56320/60000          (94%)] Loss: 0.490419
Train Epoch: 6 [56960/60000          (95%)] Loss: 0.367567
Train Epoch: 6 [57600/60000          (96%)] Loss: 0.299195
Train Epoch: 6 [58240/60000          (97%)] Loss: 0.267722
Train Epoch: 6 [58880/60000          (98%)] Loss: 0.263468
Train Epoch: 6 [59520/60000          (99%)] Loss: 0.267713

Test set: Average loss: 0.1056,      Accuracy: 9677/10000       (96.77%)
```

```
Train Epoch: 7 [0/60000 (0%)] Loss: 0.837890
Train Epoch: 7 [640/60000 (1%)] Loss: 0.189089
Train Epoch: 7 [1280/60000 (2%)] Loss: 0.180046
Train Epoch: 7 [1920/60000 (3%)] Loss: 0.704030
Train Epoch: 7 [2560/60000 (4%)] Loss: 0.116552
Train Epoch: 7 [3200/60000 (5%)] Loss: 0.297312
Train Epoch: 7 [3840/60000 (6%)] Loss: 0.531063
Train Epoch: 7 [4480/60000 (7%)] Loss: 0.350385
Train Epoch: 7 [5120/60000 (9%)] Loss: 0.338195
Train Epoch: 7 [5760/60000 (10%)] Loss: 0.212885
Train Epoch: 7 [6400/60000 (11%)] Loss: 0.278701
Train Epoch: 7 [7040/60000 (12%)] Loss: 0.450169
Train Epoch: 7 [7680/60000 (13%)] Loss: 0.220673
Train Epoch: 7 [8320/60000 (14%)] Loss: 0.250956
Train Epoch: 7 [8960/60000 (15%)] Loss: 0.579347
Train Epoch: 7 [9600/60000 (16%)] Loss: 0.199489
Train Epoch: 7 [10240/60000 (17%)] Loss: 0.358259
Train Epoch: 7 [10880/60000 (18%)] Loss: 0.214763
Train Epoch: 7 [11520/60000 (19%)] Loss: 0.278322
Train Epoch: 7 [12160/60000 (20%)] Loss: 0.385798
Train Epoch: 7 [12800/60000 (21%)] Loss: 0.388552
Train Epoch: 7 [13440/60000 (22%)] Loss: 0.262660
Train Epoch: 7 [14080/60000 (23%)] Loss: 0.351227
Train Epoch: 7 [14720/60000 (25%)] Loss: 0.128479
Train Epoch: 7 [15360/60000 (26%)] Loss: 0.463506
Train Epoch: 7 [16000/60000 (27%)] Loss: 0.288715
Train Epoch: 7 [16640/60000 (28%)] Loss: 0.523283
Train Epoch: 7 [17280/60000 (29%)] Loss: 0.375977
Train Epoch: 7 [17920/60000 (30%)] Loss: 0.390240
Train Epoch: 7 [18560/60000 (31%)] Loss: 0.428068
Train Epoch: 7 [19200/60000 (32%)] Loss: 0.518812
Train Epoch: 7 [19840/60000 (33%)] Loss: 0.691415
Train Epoch: 7 [20480/60000 (34%)] Loss: 0.273742
Train Epoch: 7 [21120/60000 (35%)] Loss: 0.400496
Train Epoch: 7 [21760/60000 (36%)] Loss: 0.469732
Train Epoch: 7 [22400/60000 (37%)] Loss: 0.199493
Train Epoch: 7 [23040/60000 (38%)] Loss: 0.418602
Train Epoch: 7 [23680/60000 (39%)] Loss: 0.532779
Train Epoch: 7 [24320/60000 (41%)] Loss: 0.226882
Train Epoch: 7 [24960/60000 (42%)] Loss: 0.240844
Train Epoch: 7 [25600/60000 (43%)] Loss: 0.750183
Train Epoch: 7 [26240/60000 (44%)] Loss: 0.281831
Train Epoch: 7 [26880/60000 (45%)] Loss: 0.570632
Train Epoch: 7 [27520/60000 (46%)] Loss: 0.396159
Train Epoch: 7 [28160/60000 (47%)] Loss: 0.346033
Train Epoch: 7 [28800/60000 (48%)] Loss: 0.698006
Train Epoch: 7 [29440/60000 (49%)] Loss: 0.692041
Train Epoch: 7 [30080/60000 (50%)] Loss: 0.234712
Train Epoch: 7 [30720/60000 (51%)] Loss: 0.248681
Train Epoch: 7 [31360/60000 (52%)] Loss: 0.568996
Train Epoch: 7 [32000/60000 (53%)] Loss: 0.258500
Train Epoch: 7 [32640/60000 (54%)] Loss: 0.488713
Train Epoch: 7 [33280/60000 (55%)] Loss: 0.337966
Train Epoch: 7 [33920/60000 (57%)] Loss: 0.187882
Train Epoch: 7 [34560/60000 (58%)] Loss: 0.417114
Train Epoch: 7 [35200/60000 (59%)] Loss: 0.374555
Train Epoch: 7 [35840/60000 (60%)] Loss: 0.138717
Train Epoch: 7 [36480/60000 (61%)] Loss: 0.280312
Train Epoch: 7 [37120/60000 (62%)] Loss: 0.452184
Train Epoch: 7 [37760/60000 (63%)] Loss: 0.286474
Train Epoch: 7 [38400/60000 (64%)] Loss: 0.170632
Train Epoch: 7 [39040/60000 (65%)] Loss: 0.591377
Train Epoch: 7 [39680/60000 (66%)] Loss: 0.131033
Train Epoch: 7 [40320/60000 (67%)] Loss: 0.223469
Train Epoch: 7 [40960/60000 (68%)] Loss: 0.284062
Train Epoch: 7 [41600/60000 (69%)] Loss: 0.288565
Train Epoch: 7 [42240/60000 (70%)] Loss: 0.183048
Train Epoch: 7 [42880/60000 (71%)] Loss: 0.297716
Train Epoch: 7 [43520/60000 (72%)] Loss: 0.236378
Train Epoch: 7 [44160/60000 (74%)] Loss: 0.481150
Train Epoch: 7 [44800/60000 (75%)] Loss: 0.369506
Train Epoch: 7 [45440/60000 (76%)] Loss: 0.356308
```

```
Train Epoch: 7 [46080/60000 (77%)] Loss: 0.457204
Train Epoch: 7 [46720/60000 (78%)] Loss: 0.371238
Train Epoch: 7 [47360/60000 (79%)] Loss: 0.453482
Train Epoch: 7 [48000/60000 (80%)] Loss: 0.240517
Train Epoch: 7 [48640/60000 (81%)] Loss: 0.303087
Train Epoch: 7 [49280/60000 (82%)] Loss: 0.224447
Train Epoch: 7 [49920/60000 (83%)] Loss: 0.388276
Train Epoch: 7 [50560/60000 (84%)] Loss: 0.322746
Train Epoch: 7 [51200/60000 (85%)] Loss: 0.307661
Train Epoch: 7 [51840/60000 (86%)] Loss: 0.116977
Train Epoch: 7 [52480/60000 (87%)] Loss: 0.381625
Train Epoch: 7 [53120/60000 (88%)] Loss: 0.392064
Train Epoch: 7 [53760/60000 (90%)] Loss: 0.524096
Train Epoch: 7 [54400/60000 (91%)] Loss: 0.539240
Train Epoch: 7 [55040/60000 (92%)] Loss: 0.548295
Train Epoch: 7 [55680/60000 (93%)] Loss: 0.339566
Train Epoch: 7 [56320/60000 (94%)] Loss: 0.303376
Train Epoch: 7 [56960/60000 (95%)] Loss: 0.334295
Train Epoch: 7 [57600/60000 (96%)] Loss: 0.341817
Train Epoch: 7 [58240/60000 (97%)] Loss: 0.182137
Train Epoch: 7 [58880/60000 (98%)] Loss: 0.351808
Train Epoch: 7 [59520/60000 (99%)] Loss: 0.517167

Test set: Average loss: 0.1013, Accuracy: 9686/10000 (96.86%)

Train Epoch: 8 [0/60000 (0%)] Loss: 0.167613
Train Epoch: 8 [640/60000 (1%)] Loss: 0.208143
Train Epoch: 8 [1280/60000 (2%)] Loss: 0.154171
Train Epoch: 8 [1920/60000 (3%)] Loss: 0.258984
Train Epoch: 8 [2560/60000 (4%)] Loss: 0.409571
Train Epoch: 8 [3200/60000 (5%)] Loss: 0.322532
Train Epoch: 8 [3840/60000 (6%)] Loss: 0.476947
Train Epoch: 8 [4480/60000 (7%)] Loss: 0.350938
Train Epoch: 8 [5120/60000 (9%)] Loss: 0.186628
Train Epoch: 8 [5760/60000 (10%)] Loss: 0.333380
Train Epoch: 8 [6400/60000 (11%)] Loss: 0.114562
Train Epoch: 8 [7040/60000 (12%)] Loss: 0.702907
Train Epoch: 8 [7680/60000 (13%)] Loss: 0.319610
Train Epoch: 8 [8320/60000 (14%)] Loss: 0.275609
Train Epoch: 8 [8960/60000 (15%)] Loss: 0.257658
Train Epoch: 8 [9600/60000 (16%)] Loss: 0.494542
Train Epoch: 8 [10240/60000 (17%)] Loss: 0.348974
Train Epoch: 8 [10880/60000 (18%)] Loss: 0.292364
Train Epoch: 8 [11520/60000 (19%)] Loss: 0.277454
Train Epoch: 8 [12160/60000 (20%)] Loss: 0.361938
Train Epoch: 8 [12800/60000 (21%)] Loss: 0.509336
Train Epoch: 8 [13440/60000 (22%)] Loss: 0.456967
Train Epoch: 8 [14080/60000 (23%)] Loss: 0.236169
Train Epoch: 8 [14720/60000 (25%)] Loss: 0.228939
Train Epoch: 8 [15360/60000 (26%)] Loss: 0.553882
Train Epoch: 8 [16000/60000 (27%)] Loss: 0.298616
Train Epoch: 8 [16640/60000 (28%)] Loss: 0.299919
Train Epoch: 8 [17280/60000 (29%)] Loss: 0.288489
Train Epoch: 8 [17920/60000 (30%)] Loss: 0.403352
Train Epoch: 8 [18560/60000 (31%)] Loss: 0.122656
Train Epoch: 8 [19200/60000 (32%)] Loss: 0.118218
Train Epoch: 8 [19840/60000 (33%)] Loss: 0.285583
Train Epoch: 8 [20480/60000 (34%)] Loss: 0.422011
Train Epoch: 8 [21120/60000 (35%)] Loss: 0.688237
Train Epoch: 8 [21760/60000 (36%)] Loss: 0.337284
Train Epoch: 8 [22400/60000 (37%)] Loss: 0.515490
Train Epoch: 8 [23040/60000 (38%)] Loss: 0.214353
Train Epoch: 8 [23680/60000 (39%)] Loss: 0.325923
Train Epoch: 8 [24320/60000 (41%)] Loss: 0.267237
Train Epoch: 8 [24960/60000 (42%)] Loss: 0.432661
Train Epoch: 8 [25600/60000 (43%)] Loss: 0.444008
Train Epoch: 8 [26240/60000 (44%)] Loss: 0.669615
Train Epoch: 8 [26880/60000 (45%)] Loss: 0.346699
Train Epoch: 8 [27520/60000 (46%)] Loss: 0.514951
Train Epoch: 8 [28160/60000 (47%)] Loss: 0.233266
Train Epoch: 8 [28800/60000 (48%)] Loss: 0.199712
Train Epoch: 8 [29440/60000 (49%)] Loss: 0.470421
```

```
Train Epoch: 8 [30080/60000                    (50%)] Loss: 0.256436
Train Epoch: 8 [30720/60000                    (51%)] Loss: 0.279074
Train Epoch: 8 [31360/60000                    (52%)] Loss: 0.321409
Train Epoch: 8 [32000/60000                    (53%)] Loss: 0.350736
Train Epoch: 8 [32640/60000                    (54%)] Loss: 0.107329
Train Epoch: 8 [33280/60000                    (55%)] Loss: 0.462671
Train Epoch: 8 [33920/60000                    (57%)] Loss: 0.310852
Train Epoch: 8 [34560/60000                    (58%)] Loss: 0.288262
Train Epoch: 8 [35200/60000                    (59%)] Loss: 0.153833
Train Epoch: 8 [35840/60000                    (60%)] Loss: 0.611765
Train Epoch: 8 [36480/60000                    (61%)] Loss: 0.560434
Train Epoch: 8 [37120/60000                    (62%)] Loss: 0.448362
Train Epoch: 8 [37760/60000                    (63%)] Loss: 0.348619
Train Epoch: 8 [38400/60000                    (64%)] Loss: 0.549396
Train Epoch: 8 [39040/60000                    (65%)] Loss: 0.409769
Train Epoch: 8 [39680/60000                    (66%)] Loss: 0.173933
Train Epoch: 8 [40320/60000                    (67%)] Loss: 0.644762
Train Epoch: 8 [40960/60000                    (68%)] Loss: 0.394223
Train Epoch: 8 [41600/60000                    (69%)] Loss: 0.231997
Train Epoch: 8 [42240/60000                    (70%)] Loss: 0.166776
Train Epoch: 8 [42880/60000                    (71%)] Loss: 0.321544
Train Epoch: 8 [43520/60000                    (72%)] Loss: 0.256221
Train Epoch: 8 [44160/60000                    (74%)] Loss: 0.358050
Train Epoch: 8 [44800/60000                    (75%)] Loss: 0.374187
Train Epoch: 8 [45440/60000                    (76%)] Loss: 0.524191
Train Epoch: 8 [46080/60000                    (77%)] Loss: 0.245598
Train Epoch: 8 [46720/60000                    (78%)] Loss: 0.639706
Train Epoch: 8 [47360/60000                    (79%)] Loss: 0.345146
Train Epoch: 8 [48000/60000                    (80%)] Loss: 0.557186
Train Epoch: 8 [48640/60000                    (81%)] Loss: 0.304694
Train Epoch: 8 [49280/60000                    (82%)] Loss: 0.403146
Train Epoch: 8 [49920/60000                    (83%)] Loss: 0.258142
Train Epoch: 8 [50560/60000                    (84%)] Loss: 0.589908
Train Epoch: 8 [51200/60000                    (85%)] Loss: 0.570742
Train Epoch: 8 [51840/60000                    (86%)] Loss: 0.321190
Train Epoch: 8 [52480/60000                    (87%)] Loss: 0.308157
Train Epoch: 8 [53120/60000                    (88%)] Loss: 0.346942
Train Epoch: 8 [53760/60000                    (90%)] Loss: 0.519119
Train Epoch: 8 [54400/60000                    (91%)] Loss: 0.183560
Train Epoch: 8 [55040/60000                    (92%)] Loss: 0.200081
Train Epoch: 8 [55680/60000                    (93%)] Loss: 0.279761
Train Epoch: 8 [56320/60000                    (94%)] Loss: 0.340631
Train Epoch: 8 [56960/60000                    (95%)] Loss: 0.160399
Train Epoch: 8 [57600/60000                    (96%)] Loss: 0.316809
Train Epoch: 8 [58240/60000                    (97%)] Loss: 0.597524
Train Epoch: 8 [58880/60000                    (98%)] Loss: 0.340963
Train Epoch: 8 [59520/60000                    (99%)] Loss: 0.288645

Test set: Average loss: 0.1165,              Accuracy: 9652/10000              (96.52%)

Train Epoch: 9 [0/60000                    (0%)] Loss: 0.169700
Train Epoch: 9 [640/60000                    (1%)] Loss: 0.243249
Train Epoch: 9 [1280/60000                    (2%)] Loss: 0.532874
Train Epoch: 9 [1920/60000                    (3%)] Loss: 0.280194
Train Epoch: 9 [2560/60000                    (4%)] Loss: 0.350955
Train Epoch: 9 [3200/60000                    (5%)] Loss: 0.346968
Train Epoch: 9 [3840/60000                    (6%)] Loss: 0.296314
Train Epoch: 9 [4480/60000                    (7%)] Loss: 0.176825
Train Epoch: 9 [5120/60000                    (9%)] Loss: 0.586302
Train Epoch: 9 [5760/60000                    (10%)] Loss: 0.274271
Train Epoch: 9 [6400/60000                    (11%)] Loss: 0.190513
Train Epoch: 9 [7040/60000                    (12%)] Loss: 0.457308
Train Epoch: 9 [7680/60000                    (13%)] Loss: 0.208888
Train Epoch: 9 [8320/60000                    (14%)] Loss: 0.526673
Train Epoch: 9 [8960/60000                    (15%)] Loss: 0.606389
Train Epoch: 9 [9600/60000                    (16%)] Loss: 0.286861
Train Epoch: 9 [10240/60000                    (17%)] Loss: 0.454417
Train Epoch: 9 [10880/60000                    (18%)] Loss: 0.171716
Train Epoch: 9 [11520/60000                    (19%)] Loss: 0.378621
Train Epoch: 9 [12160/60000                    (20%)] Loss: 0.476416
Train Epoch: 9 [12800/60000                    (21%)] Loss: 0.307236
Train Epoch: 9 [13440/60000                    (22%)] Loss: 0.499397
```

```
Train Epoch: 9 [14080/60000 (23%)] Loss: 0.336605
Train Epoch: 9 [14720/60000 (25%)] Loss: 0.626056
Train Epoch: 9 [15360/60000 (26%)] Loss: 0.238846
Train Epoch: 9 [16000/60000 (27%)] Loss: 0.195958
Train Epoch: 9 [16640/60000 (28%)] Loss: 0.175957
Train Epoch: 9 [17280/60000 (29%)] Loss: 0.156235
Train Epoch: 9 [17920/60000 (30%)] Loss: 0.459341
Train Epoch: 9 [18560/60000 (31%)] Loss: 0.354459
Train Epoch: 9 [19200/60000 (32%)] Loss: 0.681841
Train Epoch: 9 [19840/60000 (33%)] Loss: 0.605146
Train Epoch: 9 [20480/60000 (34%)] Loss: 0.133440
Train Epoch: 9 [21120/60000 (35%)] Loss: 0.405487
Train Epoch: 9 [21760/60000 (36%)] Loss: 0.393784
Train Epoch: 9 [22400/60000 (37%)] Loss: 0.384011
Train Epoch: 9 [23040/60000 (38%)] Loss: 0.231696
Train Epoch: 9 [23680/60000 (39%)] Loss: 0.238498
Train Epoch: 9 [24320/60000 (41%)] Loss: 0.135217
Train Epoch: 9 [24960/60000 (42%)] Loss: 0.197532
Train Epoch: 9 [25600/60000 (43%)] Loss: 0.285111
Train Epoch: 9 [26240/60000 (44%)] Loss: 0.499086
Train Epoch: 9 [26880/60000 (45%)] Loss: 0.305744
Train Epoch: 9 [27520/60000 (46%)] Loss: 0.482932
Train Epoch: 9 [28160/60000 (47%)] Loss: 0.184272
Train Epoch: 9 [28800/60000 (48%)] Loss: 0.557988
Train Epoch: 9 [29440/60000 (49%)] Loss: 0.411541
Train Epoch: 9 [30080/60000 (50%)] Loss: 0.202577
Train Epoch: 9 [30720/60000 (51%)] Loss: 0.378262
Train Epoch: 9 [31360/60000 (52%)] Loss: 0.615596
Train Epoch: 9 [32000/60000 (53%)] Loss: 0.459101
Train Epoch: 9 [32640/60000 (54%)] Loss: 0.382366
Train Epoch: 9 [33280/60000 (55%)] Loss: 0.333661
Train Epoch: 9 [33920/60000 (57%)] Loss: 0.389638
Train Epoch: 9 [34560/60000 (58%)] Loss: 0.465811
Train Epoch: 9 [35200/60000 (59%)] Loss: 0.371905
Train Epoch: 9 [35840/60000 (60%)] Loss: 0.180698
Train Epoch: 9 [36480/60000 (61%)] Loss: 0.611916
Train Epoch: 9 [37120/60000 (62%)] Loss: 0.567188
Train Epoch: 9 [37760/60000 (63%)] Loss: 0.498344
Train Epoch: 9 [38400/60000 (64%)] Loss: 0.160315
Train Epoch: 9 [39040/60000 (65%)] Loss: 0.320574
Train Epoch: 9 [39680/60000 (66%)] Loss: 0.533084
Train Epoch: 9 [40320/60000 (67%)] Loss: 0.329050
Train Epoch: 9 [40960/60000 (68%)] Loss: 0.420700
Train Epoch: 9 [41600/60000 (69%)] Loss: 0.202165
Train Epoch: 9 [42240/60000 (70%)] Loss: 0.347387
Train Epoch: 9 [42880/60000 (71%)] Loss: 0.162860
Train Epoch: 9 [43520/60000 (72%)] Loss: 0.271953
Train Epoch: 9 [44160/60000 (74%)] Loss: 0.356759
Train Epoch: 9 [44800/60000 (75%)] Loss: 0.560521
Train Epoch: 9 [45440/60000 (76%)] Loss: 0.330899
Train Epoch: 9 [46080/60000 (77%)] Loss: 0.648139
Train Epoch: 9 [46720/60000 (78%)] Loss: 0.417379
Train Epoch: 9 [47360/60000 (79%)] Loss: 0.135176
Train Epoch: 9 [48000/60000 (80%)] Loss: 0.190184
Train Epoch: 9 [48640/60000 (81%)] Loss: 0.365274
Train Epoch: 9 [49280/60000 (82%)] Loss: 0.487520
Train Epoch: 9 [49920/60000 (83%)] Loss: 0.279231
Train Epoch: 9 [50560/60000 (84%)] Loss: 0.200188
Train Epoch: 9 [51200/60000 (85%)] Loss: 0.457125
Train Epoch: 9 [51840/60000 (86%)] Loss: 0.348882
Train Epoch: 9 [52480/60000 (87%)] Loss: 0.178141
Train Epoch: 9 [53120/60000 (88%)] Loss: 0.398479
Train Epoch: 9 [53760/60000 (90%)] Loss: 0.443322
Train Epoch: 9 [54400/60000 (91%)] Loss: 0.336012
Train Epoch: 9 [55040/60000 (92%)] Loss: 0.378820
Train Epoch: 9 [55680/60000 (93%)] Loss: 0.471646
Train Epoch: 9 [56320/60000 (94%)] Loss: 0.433204
Train Epoch: 9 [56960/60000 (95%)] Loss: 0.231810
Train Epoch: 9 [57600/60000 (96%)] Loss: 0.267413
Train Epoch: 9 [58240/60000 (97%)] Loss: 0.212249
Train Epoch: 9 [58880/60000 (98%)] Loss: 0.286779
Train Epoch: 9 [59520/60000 (99%)] Loss: 0.198763
```

```
Test set: Average loss: 0.1123, Accuracy: 9681/10000 (96.81%)

Train Epoch: 10 [0/60000 (0%)] Loss: 0.383010
Train Epoch: 10 [640/60000 (1%)] Loss: 0.277066
Train Epoch: 10 [1280/60000 (2%)] Loss: 0.218355
Train Epoch: 10 [1920/60000 (3%)] Loss: 0.380143
Train Epoch: 10 [2560/60000 (4%)] Loss: 0.205005
Train Epoch: 10 [3200/60000 (5%)] Loss: 0.425667
Train Epoch: 10 [3840/60000 (6%)] Loss: 0.425182
Train Epoch: 10 [4480/60000 (7%)] Loss: 0.420155
Train Epoch: 10 [5120/60000 (9%)] Loss: 0.131133
Train Epoch: 10 [5760/60000 (10%)] Loss: 0.332233
Train Epoch: 10 [6400/60000 (11%)] Loss: 0.370097
Train Epoch: 10 [7040/60000 (12%)] Loss: 0.490312
Train Epoch: 10 [7680/60000 (13%)] Loss: 0.352214
Train Epoch: 10 [8320/60000 (14%)] Loss: 0.288417
Train Epoch: 10 [8960/60000 (15%)] Loss: 0.376205
Train Epoch: 10 [9600/60000 (16%)] Loss: 0.472742
Train Epoch: 10 [10240/60000 (17%)] Loss: 0.208089
Train Epoch: 10 [10880/60000 (18%)] Loss: 0.185488
Train Epoch: 10 [11520/60000 (19%)] Loss: 0.437391
Train Epoch: 10 [12160/60000 (20%)] Loss: 0.353009
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.278874
Train Epoch: 10 [13440/60000 (22%)] Loss: 0.112826
Train Epoch: 10 [14080/60000 (23%)] Loss: 0.166523
Train Epoch: 10 [14720/60000 (25%)] Loss: 0.347804
Train Epoch: 10 [15360/60000 (26%)] Loss: 0.144403
Train Epoch: 10 [16000/60000 (27%)] Loss: 0.337187
Train Epoch: 10 [16640/60000 (28%)] Loss: 0.281081
Train Epoch: 10 [17280/60000 (29%)] Loss: 0.431080
Train Epoch: 10 [17920/60000 (30%)] Loss: 0.248372
Train Epoch: 10 [18560/60000 (31%)] Loss: 0.558700
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.397511
Train Epoch: 10 [19840/60000 (33%)] Loss: 0.230046
Train Epoch: 10 [20480/60000 (34%)] Loss: 0.310099
Train Epoch: 10 [21120/60000 (35%)] Loss: 0.373435
Train Epoch: 10 [21760/60000 (36%)] Loss: 0.331420
Train Epoch: 10 [22400/60000 (37%)] Loss: 0.246949
Train Epoch: 10 [23040/60000 (38%)] Loss: 0.492376
Train Epoch: 10 [23680/60000 (39%)] Loss: 0.240741
Train Epoch: 10 [24320/60000 (41%)] Loss: 0.255109
Train Epoch: 10 [24960/60000 (42%)] Loss: 0.275360
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.301749
Train Epoch: 10 [26240/60000 (44%)] Loss: 0.573375
Train Epoch: 10 [26880/60000 (45%)] Loss: 0.195788
Train Epoch: 10 [27520/60000 (46%)] Loss: 0.690665
Train Epoch: 10 [28160/60000 (47%)] Loss: 0.254778
Train Epoch: 10 [28800/60000 (48%)] Loss: 0.514807
Train Epoch: 10 [29440/60000 (49%)] Loss: 0.198609
Train Epoch: 10 [30080/60000 (50%)] Loss: 0.505396
Train Epoch: 10 [30720/60000 (51%)] Loss: 0.196715
Train Epoch: 10 [31360/60000 (52%)] Loss: 0.328238
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.239519
Train Epoch: 10 [32640/60000 (54%)] Loss: 0.280905
Train Epoch: 10 [33280/60000 (55%)] Loss: 0.253024
Train Epoch: 10 [33920/60000 (57%)] Loss: 0.315063
Train Epoch: 10 [34560/60000 (58%)] Loss: 0.368183
Train Epoch: 10 [35200/60000 (59%)] Loss: 0.527836
Train Epoch: 10 [35840/60000 (60%)] Loss: 0.355607
Train Epoch: 10 [36480/60000 (61%)] Loss: 0.475712
Train Epoch: 10 [37120/60000 (62%)] Loss: 0.656337
Train Epoch: 10 [37760/60000 (63%)] Loss: 0.273347
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.727771
Train Epoch: 10 [39040/60000 (65%)] Loss: 0.471544
Train Epoch: 10 [39680/60000 (66%)] Loss: 0.209420
Train Epoch: 10 [40320/60000 (67%)] Loss: 0.350493
Train Epoch: 10 [40960/60000 (68%)] Loss: 0.409450
Train Epoch: 10 [41600/60000 (69%)] Loss: 0.244843
Train Epoch: 10 [42240/60000 (70%)] Loss: 0.362997
Train Epoch: 10 [42880/60000 (71%)] Loss: 0.280947
Train Epoch: 10 [43520/60000 (72%)] Loss: 0.378039
```

```
Train Epoch: 10 [44160/60000                    (74%)] Loss: 0.390458
Train Epoch: 10 [44800/60000                    (75%)] Loss: 0.205114
Train Epoch: 10 [45440/60000                    (76%)] Loss: 0.370930
Train Epoch: 10 [46080/60000                    (77%)] Loss: 0.239301
Train Epoch: 10 [46720/60000                    (78%)] Loss: 0.236183
Train Epoch: 10 [47360/60000                    (79%)] Loss: 0.473879
Train Epoch: 10 [48000/60000                    (80%)] Loss: 0.450681
Train Epoch: 10 [48640/60000                    (81%)] Loss: 0.484007
Train Epoch: 10 [49280/60000                    (82%)] Loss: 0.527784
Train Epoch: 10 [49920/60000                    (83%)] Loss: 0.417889
Train Epoch: 10 [50560/60000                    (84%)] Loss: 0.389166
Train Epoch: 10 [51200/60000                    (85%)] Loss: 0.563477
Train Epoch: 10 [51840/60000                    (86%)] Loss: 0.662653
Train Epoch: 10 [52480/60000                    (87%)] Loss: 0.284285
Train Epoch: 10 [53120/60000                    (88%)] Loss: 0.620631
Train Epoch: 10 [53760/60000                    (90%)] Loss: 0.525693
Train Epoch: 10 [54400/60000                    (91%)] Loss: 0.366103
Train Epoch: 10 [55040/60000                    (92%)] Loss: 0.377109
Train Epoch: 10 [55680/60000                    (93%)] Loss: 0.421128
Train Epoch: 10 [56320/60000                    (94%)] Loss: 0.587034
Train Epoch: 10 [56960/60000                    (95%)] Loss: 0.387451
Train Epoch: 10 [57600/60000                    (96%)] Loss: 0.314108
Train Epoch: 10 [58240/60000                    (97%)] Loss: 0.208940
Train Epoch: 10 [58880/60000                    (98%)] Loss: 0.139073
Train Epoch: 10 [59520/60000                    (99%)] Loss: 0.586233

Test set: Average loss: 0.1026,              Accuracy: 9683/10000              (96.83%)
```

More Visuals

New MNIST Data Wrangling

Importing libraries again

In [69]:

```python
from datetime import datetime
from pathlib import Path

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from torch.utils.data import random_split
from torch.utils.tensorboard import SummaryWriter
from torchvision import datasets, transforms
```

Initializing hyperparameters again (new "logging_interval" value)

In [70]:

```python
batch_size = 64
test_batch_size = 1000
epochs = 10
lr = 0.01
try_cuda = True
seed = 1000
logging_interval = 100
logging_dir = None
```

Setting up the logging w/ new dir

In [71]:

```
datetime_str = datetime.now().strftime('%b%d_%H-%M-%S')

if logging_dir is None:
    base_folder = Path("./logged_runs/")
    base_folder.mkdir(parents=True, exist_ok=True)
    logging_dir = base_folder / Path(datetime_str)
    logging_dir.mkdir(exist_ok=True)
    logging_dir = str(logging_dir.absolute())

writer = SummaryWriter(log_dir=logging_dir)
```

Deciding whether to send to the cpu or not if available

In [72]:

```
if torch.cuda.is_available() and try_cuda:
    cuda = True
    torch.cuda.mnaual_seed(seed)

else:
    cuda = False
    torch.manual_seed(seed)

print(cuda)
```

```
False
```

Setting up the data loaders (new "valid_loader" validation dataset).

In [73]:

```
transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.01307,), (0.3081,))
])

train_data = datasets.MNIST(
        'data',
        train=True,
        download=True,
        transform=transform,
        )

train_size = int(0.9 * len(train_data))
valid_size = len(train_data) - train_size

actual_train_dataset, validation_dataset =\
    random_split(train_data, [train_size, valid_size])

train_loader = torch.utils.data.DataLoader(
    actual_train_dataset,
    batch_size=batch_size,
    shuffle=True)

valid_loader = torch.utils.data.DataLoader(
    validation_dataset,
    batch_size=batch_size,
    shuffle=True)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        'data',
        train=False,
        download=True,
        transform=transform,
    ),
    batch_size=batch_size,
    shuffle=True,
```

```
)
```

MNIST DCN Logger

In [74]:

```
# Defining new Architecture with logging stats
class LoggerNet(nn.Module):
    def __init__(self):
        super(LoggerNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        self.stats = {}
        x = self.conv1(x)
        self.stats['conv1_weights'] = self.conv1.weight.data.cpu().numpy()
        self.stats['conv1_biases'] = self.conv1.bias.data.cpu().numpy()
        self.stats['conv1_net_input'] = x.data.cpu().numpy()
        x = F.relu(x)
        self.stats['conv1_activations_after_relu'] = x.data.cpu().numpy()
        x = F.max_pool2d(x, 2)
        self.stats['conv1_activations_after_maxpool'] = x.data.cpu().numpy()

        x = self.conv2(x)
        self.stats['conv2_weights'] = self.conv2.weight.data.cpu().numpy()
        self.stats['conv2_biases'] = self.conv2.bias.data.cpu().numpy()
        self.stats['conv2_net_input'] = x.data.cpu().numpy()
        x = F.relu(x)
        self.stats['conv2_activations_after_relu'] = x.data.cpu().numpy()
        x = F.max_pool2d(x, 2)
        self.stats['conv2_activations_after_maxpool'] = x.data.cpu().numpy()

        x = x.view(-1, 320)   # (batch_size, units)

        x = self.fc1(x)
        self.stats['fc1_weights'] = self.fc1.weight.data.cpu().numpy()
        self.stats['fc1_biases'] = self.fc1.bias.data.cpu().numpy()
        self.stats['fc1_net_input'] = x.data.cpu().numpy()
        x = F.relu(x)
        self.stats['fc1_activations_after_relu'] = x.data.cpu().numpy()

        x = F.dropout(x, training=self.training)

        x = self.fc2(x)
        self.stats['fc2_weights'] = self.fc2.weight.data.cpu().numpy()
        self.stats['fc2_biases'] = self.fc2.bias.data.cpu().numpy()
        self.stats['fc2_net_input'] = x.data.cpu().numpy()

        x = F.softmax(x, dim=1)

        return x


model = LoggerNet()
if cuda:
    model.cuda()

optimizer = optim.Adam(model.parameters(), lr=lr)


# Visualize network as a graph on TensorBoard
input_tensor = torch.Tensor(1, 1, 28, 28)
if cuda:
    input_tensor = input_tensor.cuda()
writer.add_graph(model, input_to_model=input_tensor)
```

```
<ipython-input-74-395ccfbab5d3>:14: TracerWarning: Converting a tensor to a NumPy array m
```

```
<ipython-input-74-395ccfbab5d3>:14: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_weights'] = self.conv1.weight.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:15: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_biases'] = self.conv1.bias.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:16: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_net_input'] = x.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:18: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_activations_after_relu'] = x.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:20: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_activations_after_maxpool'] = x.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:23: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_weights'] = self.conv2.weight.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:24: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_biases'] = self.conv2.bias.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:25: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_net_input'] = x.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:27: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_activations_after_relu'] = x.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:29: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_activations_after_maxpool'] = x.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:34: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc1_weights'] = self.fc1.weight.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:35: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc1_biases'] = self.fc1.bias.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:36: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc1_net_input'] = x.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:38: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc1_activations_after_relu'] = x.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:43: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
```

```
ot generalize to other inputs!
  self.stats['fc2_weights'] = self.fc2.weight.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:44: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc2_biases'] = self.fc2.bias.data.cpu().numpy()
<ipython-input-74-395ccfbab5d3>:45: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc2_net_input'] = x.data.cpu().numpy()
```

Logging DCN's Training, Validation, and Testing functions

In [75]:

```
eps = 1e-13
```

Training Function for Logging DCN

In [76]:

```
def log_train(epoch):
    model.train()
    criterion = nn.NLLLoss()
    # criterion = nn.CrossEntropyLoss()

    for batch_idx, (data, target) in enumerate(train_loader):
        if cuda:
            data, target = data.cuda(), target.cuda()

        optimizer.zero_grad()
        output = model(data)   # forward
        # loss = sum_k(-t_k * log(y_k))
        loss = criterion(torch.log(output+eps), target)
        loss.backward()
        optimizer.step()

        if batch_idx % logging_interval == 0:
            print(f'Train Epoch: {epoch} \
                  [{batch_idx * len(data)}/{len(train_loader.dataset)} \
                    ({100. * batch_idx / len(train_loader):.0f}%)]\t\
                        Loss: {loss.item():.6f}')

            # Log train/loss to TensorBoard at every iteration
            n_iter = (epoch - 1) * len(train_loader) + batch_idx + 1
            writer.add_scalar('train/loss', loss.item(), n_iter)
            for key, value in model.stats.items():
                value_flat = value.flatten()
                writer.add_scalar(
                    f'statistics/{key}_mean',
                    np.mean(value_flat),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{key}_std',
                    np.std(value_flat),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{key}_min',
                    np.min(value_flat),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{key}_max',
                    np.max(value_flat),
```

```
                n_iter
            )
            writer.add_histogram(
                f'histogram/{key}',
                value_flat,
                n_iter
            )
        for name, param in model.named_parameters():
            if 'weight' in name:
                writer.add_scalar(
                    f'statistics/{name}_min',
                    param.min().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_max',
                    param.max().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_mean',
                    param.mean().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_std',
                    param.std().item(),
                    n_iter
                )
                writer.add_histogram(
                    f'histogram/{name}',
                    param,
                    n_iter
                )
            elif 'bias' in name:
                writer.add_scalar(
                    f'statistics/{name}_min',
                    param.min().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_max',
                    param.max().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_mean',
                    param.mean().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_std',
                    param.std().item(),
                    n_iter
                )
                writer.add_histogram(
                    f'histogram/{name}',
                    param,
                    n_iter
                )
```

Validation Function for Logging DCN

In [77]:

```
def validation(epoch):
    model.eval()
    correct = 0
    valid_loss = 0
    criterion = nn.NLLLoss(size_average=False)
```

```
    for data, target in valid_loader:
        if cuda:
            data, target = data.cuda(), target.cuda()
        output = model(data)
        valid_loss += criterion(torch.log(output+eps), target).item()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

    valid_loss /= len(valid_loader.dataset)
    valid_accuracy = 100. * correct / len(valid_loader.dataset)
    print(f'Validation set: Average loss: {valid_loss:.4f}, \
         Accuracy: {correct}/{len(valid_loader.dataset)} \
         ({valid_accuracy:.2f}%)\n')

    # Log test/loss and test/accuracy to TensorBoard at every epoch
    n_iter = epoch * len(valid_loader)
    writer.add_scalar('valid/loss', valid_loss, n_iter)
    writer.add_scalar('valid/accuracy', valid_accuracy, n_iter)
    writer.add_scalar('valid/error', 100. - valid_accuracy, n_iter)
```

Testing Function for Logging DCN

In [78]:

```
def log_testing(epoch):
    model.eval()
    correct = 0
    test_loss = 0
    criterion = nn.NLLLoss(size_average=False)
    # criterion = nn.CrossEntropyLoss(size_average=False)

    for data, target in test_loader:
        if cuda:
            data, target = data.cuda(), target.cuda()

        output = model(data)

        # sum up batch loss (later, averaged over all test samples)
        test_loss += criterion(torch.log(output+eps), target,).item()
        # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    test_loss /= len(test_loader.dataset)
    test_accuracy = 100. * correct / len(test_loader.dataset)
    print(f'\nTest set: Average loss: {test_loss:.4f}, \
         Accuracy: {correct}/{len(test_loader.dataset)} \
         ({test_accuracy:.2f}%)\n')

    # Log test/loss and test/accuracy to TensorBoard at every epoch
    n_iter = epoch * len(test_loader)
    writer.add_scalar('test/loss', test_loss, n_iter)
    writer.add_scalar('test/accuracy', test_accuracy, n_iter)
    writer.add_scalar('test/error', 100. - test_accuracy, n_iter)
```

Logged Development Loop

In [79]:

```
for epoch in range(1, epochs + 1):
    log_train(epoch)
    validation(epoch)
    log_testing(epoch)
writer.close()
```

```
Train Epoch: 1                [0/54000                    (0%)]
Loss: 2.348830
Train Epoch: 1                [6400/54000                    (12%)]
```

```
Loss: 0.644235
Train Epoch: 1                  [12800/54000                    (24%)]
Loss: 0.400018
Train Epoch: 1                  [19200/54000                    (36%)]
Loss: 0.375166
Train Epoch: 1                  [25600/54000                    (47%)]
Loss: 0.595027
Train Epoch: 1                  [32000/54000                    (59%)]
Loss: 0.328699
Train Epoch: 1                  [38400/54000                    (71%)]
Loss: 0.585336
Train Epoch: 1                  [44800/54000                    (83%)]
Loss: 0.260204
Train Epoch: 1                  [51200/54000                    (95%)]
Loss: 0.109617
Validation set: Average loss: 0.1149,         Accuracy: 5783/6000             (96.
38%)


Test set: Average loss: 0.1013,             Accuracy: 9673/10000             (96.73%)

Train Epoch: 2                  [0/54000                    (0%)]
Loss: 0.172939
Train Epoch: 2                  [6400/54000                    (12%)]
Loss: 0.300560
Train Epoch: 2                  [12800/54000                    (24%)]
Loss: 0.205040
Train Epoch: 2                  [19200/54000                    (36%)]
Loss: 0.384614
Train Epoch: 2                  [25600/54000                    (47%)]
Loss: 0.272137
Train Epoch: 2                  [32000/54000                    (59%)]
Loss: 0.240116
Train Epoch: 2                  [38400/54000                    (71%)]
Loss: 0.323737
Train Epoch: 2                  [44800/54000                    (83%)]
Loss: 0.262507
Train Epoch: 2                  [51200/54000                    (95%)]
Loss: 0.122860
Validation set: Average loss: 0.0960,         Accuracy: 5861/6000             (97.
68%)


Test set: Average loss: 0.0860,             Accuracy: 9798/10000             (97.98%)

Train Epoch: 3                  [0/54000                    (0%)]
Loss: 0.260876
Train Epoch: 3                  [6400/54000                    (12%)]
Loss: 0.452025
Train Epoch: 3                  [12800/54000                    (24%)]
Loss: 0.160304
Train Epoch: 3                  [19200/54000                    (36%)]
Loss: 0.118868
Train Epoch: 3                  [25600/54000                    (47%)]
Loss: 0.169762
Train Epoch: 3                  [32000/54000                    (59%)]
Loss: 0.665644
Train Epoch: 3                  [38400/54000                    (71%)]
Loss: 0.080182
Train Epoch: 3                  [44800/54000                    (83%)]
Loss: 0.194716
Train Epoch: 3                  [51200/54000                    (95%)]
Loss: 0.298285
Validation set: Average loss: 0.1047,         Accuracy: 5865/6000             (97.
75%)


Test set: Average loss: 0.0851,             Accuracy: 9789/10000             (97.89%)

Train Epoch: 4                  [0/54000                    (0%)]
Loss: 0.075072
Train Epoch: 4                  [6400/54000                    (12%)]
```

```
Loss: 0.047196
Train Epoch: 4                [12800/54000                 (24%)]
Loss: 0.091742
Train Epoch: 4                [19200/54000                 (36%)]
Loss: 0.238872
Train Epoch: 4                [25600/54000                 (47%)]
Loss: 0.116513
Train Epoch: 4                [32000/54000                 (59%)]
Loss: 0.188632
Train Epoch: 4                [38400/54000                 (71%)]
Loss: 0.161212
Train Epoch: 4                [44800/54000                 (83%)]
Loss: 0.106673
Train Epoch: 4                [51200/54000                 (95%)]
Loss: 0.103287
Validation set: Average loss: 0.0960,           Accuracy: 5848/6000           (97.
47%)


Test set: Average loss: 0.0887,           Accuracy: 9747/10000           (97.47%)

Train Epoch: 5                [0/54000                 (0%)]
Loss: 0.104163
Train Epoch: 5                [6400/54000                 (12%)]
Loss: 0.148269
Train Epoch: 5                [12800/54000                 (24%)]
Loss: 0.175344
Train Epoch: 5                [19200/54000                 (36%)]
Loss: 0.269765
Train Epoch: 5                [25600/54000                 (47%)]
Loss: 0.171820
Train Epoch: 5                [32000/54000                 (59%)]
Loss: 0.105960
Train Epoch: 5                [38400/54000                 (71%)]
Loss: 0.487016
Train Epoch: 5                [44800/54000                 (83%)]
Loss: 0.421033
Train Epoch: 5                [51200/54000                 (95%)]
Loss: 0.080383
Validation set: Average loss: 0.0960,           Accuracy: 5850/6000           (97.
50%)


Test set: Average loss: 0.0835,           Accuracy: 9786/10000           (97.86%)

Train Epoch: 6                [0/54000                 (0%)]
Loss: 0.075624
Train Epoch: 6                [6400/54000                 (12%)]
Loss: 0.414105
Train Epoch: 6                [12800/54000                 (24%)]
Loss: 0.148393
Train Epoch: 6                [19200/54000                 (36%)]
Loss: 0.430653
Train Epoch: 6                [25600/54000                 (47%)]
Loss: 0.138468
Train Epoch: 6                [32000/54000                 (59%)]
Loss: 0.100153
Train Epoch: 6                [38400/54000                 (71%)]
Loss: 0.106494
Train Epoch: 6                [44800/54000                 (83%)]
Loss: 0.237655
Train Epoch: 6                [51200/54000                 (95%)]
Loss: 0.216577
Validation set: Average loss: 0.1328,           Accuracy: 5803/6000           (96.
72%)


Test set: Average loss: 0.1342,           Accuracy: 9658/10000           (96.58%)

Train Epoch: 7                [0/54000                 (0%)]
Loss: 0.482705
Train Epoch: 7                [6400/54000                 (12%)]
```

```
Loss: 0.079652
Train Epoch: 7                [12800/54000                 (24%)]
Loss: 0.198897
Train Epoch: 7                [19200/54000                 (36%)]
Loss: 0.083859
Train Epoch: 7                [25600/54000                 (47%)]
Loss: 0.172761
Train Epoch: 7                [32000/54000                 (59%)]
Loss: 0.056938
Train Epoch: 7                [38400/54000                 (71%)]
Loss: 0.146611
Train Epoch: 7                [44800/54000                 (83%)]
Loss: 0.143602
Train Epoch: 7                [51200/54000                 (95%)]
Loss: 0.228041
Validation set: Average loss: 0.1036,          Accuracy: 5861/6000            (97.
68%)


Test set: Average loss: 0.0871,            Accuracy: 9772/10000           (97.72%)

Train Epoch: 8                [0/54000                  (0%)]
Loss: 0.063642
Train Epoch: 8                [6400/54000                 (12%)]
Loss: 0.412957
Train Epoch: 8                [12800/54000                 (24%)]
Loss: 0.238586
Train Epoch: 8                [19200/54000                 (36%)]
Loss: 0.054792
Train Epoch: 8                [25600/54000                 (47%)]
Loss: 0.165892
Train Epoch: 8                [32000/54000                 (59%)]
Loss: 0.211161
Train Epoch: 8                [38400/54000                 (71%)]
Loss: 0.171858
Train Epoch: 8                [44800/54000                 (83%)]
Loss: 0.104013
Train Epoch: 8                [51200/54000                 (95%)]
Loss: 0.385878
Validation set: Average loss: 0.1009,          Accuracy: 5849/6000            (97.
48%)


Test set: Average loss: 0.0816,            Accuracy: 9774/10000           (97.74%)

Train Epoch: 9                [0/54000                  (0%)]
Loss: 0.206377
Train Epoch: 9                [6400/54000                 (12%)]
Loss: 0.468197
Train Epoch: 9                [12800/54000                 (24%)]
Loss: 0.091172
Train Epoch: 9                [19200/54000                 (36%)]
Loss: 0.207672
Train Epoch: 9                [25600/54000                 (47%)]
Loss: 0.147956
Train Epoch: 9                [32000/54000                 (59%)]
Loss: 0.107348
Train Epoch: 9                [38400/54000                 (71%)]
Loss: 0.217135
Train Epoch: 9                [44800/54000                 (83%)]
Loss: 0.134008
Train Epoch: 9                [51200/54000                 (95%)]
Loss: 0.368295
Validation set: Average loss: 0.1208,          Accuracy: 5848/6000            (97.
47%)


Test set: Average loss: 0.1058,            Accuracy: 9753/10000           (97.53%)

Train Epoch: 10                [0/54000                  (0%)]
Loss: 0.217871
Train Epoch: 10                [6400/54000                 (12%)]
```

```
Loss: 0.132886
Train Epoch: 10                    [12800/54000                    (24%)]
Loss: 0.129986
Train Epoch: 10                    [19200/54000                    (36%)]
Loss: 0.149658
Train Epoch: 10                    [25600/54000                    (47%)]
Loss: 0.311660
Train Epoch: 10                    [32000/54000                    (59%)]
Loss: 0.253167
Train Epoch: 10                    [38400/54000                    (71%)]
Loss: 0.365745
Train Epoch: 10                    [44800/54000                    (83%)]
Loss: 0.207317
Train Epoch: 10                    [51200/54000                    (95%)]
Loss: 0.172122
Validation set: Average loss: 0.1236,          Accuracy: 5839/6000             (97.
32%)


Test set: Average loss: 0.0913,              Accuracy: 9778/10000            (97.78%)
```

Combinations of training algorithms & non-linearities w/ Xavier initialization technique

Import Relevant Libraries

In [80]:

```
import torch
import torch.nn as nn
import torch.optim as optim

from torch.utils.data import random_split
from torch.utils.tensorboard import SummaryWriter
from torchvision import datasets, transforms
```

Initializing hyperparameters for redundancy

in case we need to run only the combinations of optimizing algorithms & non-linearity activation functions with xavier initialization

In [81]:

```
batch_size = 64
test_batch_size = 1000
epochs = 10
lr = 0.01
try_cuda = True
seed = 1000
logging_interval = 100
logging_dir = None
```

Setting up the logging w/ new "configurable" dir

In [82]:

```
datetime_str = datetime.now().strftime('%b%d_%H-%M-%S')

if logging_dir is None:
    base_folder = Path("./configurable_runs/")
    base_folder.mkdir(parents=True, exist_ok=True)
    logging_dir = base_folder / Path(datetime_str)
    logging_dir.mkdir(exist_ok=True)
    logging_dir = str(logging_dir.absolute())
```

```
writer = SummaryWriter(log_dir=logging_dir)
```

Deciding whether to send to the cpu or not if available

In [83]:

```
if torch.cuda.is_available() and try_cuda:
    cuda = True
    torch.cuda.mnaual_seed(seed)

else:
    cuda = False
    torch.manual_seed(seed)

print(cuda)
```

```
False
```

MNIST Data Wrangling for Configurable DCN

Setting up the data loaders (new "valid_loader" validation dataset)

In [84]:

```
transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.01307,), (0.3081,))
])

train_data = datasets.MNIST(
        'data',
        train=True,
        download=True,
        transform=transform,
        )

train_size = int(0.9 * len(train_data))
valid_size = len(train_data) - train_size

actual_train_dataset, validation_dataset =\
    random_split(train_data, [train_size, valid_size])

train_loader = torch.utils.data.DataLoader(
    actual_train_dataset,
    batch_size=batch_size,
    shuffle=True)

valid_loader = torch.utils.data.DataLoader(
    validation_dataset,
    batch_size=batch_size,
    shuffle=True)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        'data',
        train=False,
        download=True,
        transform=transform,
    ),
    batch_size=batch_size,
    shuffle=True,
)
```

Define Configurations of Activation Functions & Optimizers

In [85]:

```
ACTIVATION_FUNCTION = 'LeakyReLU'  # Options: 'ReLU', 'Tanh', 'Sigmoid', 'LeakyReLU'

OPTIMIZER = 'Adagrad'  # Options: 'Adam', 'SGD', 'Momentum', 'Adagrad'
```

Function for Optimizer Selection

(SGD, Momentum-based Methods, or Adagrad..)

In [86]:

```
def get_optimizer(name, parameters, lr=0.001):
    """
    Returns the optimizer corresponding to the given name.

    Args:
    name (str): The name of the optimizer ('Adam', 'SGD', 'Momentum', 'Adagrad').
    parameters (iterable): The parameters to optimize.
    lr (float, optional): Learning rate for the optimizer. Defaults to 0.001.

    Returns:
    torch.optim.Optimizer: The optimizer class from PyTorch.
    """
    if name == 'Adam':
        return optim.Adam(parameters, lr=lr)
    elif name == 'SGD':
        return optim.SGD(parameters, lr=lr)
    elif name == 'Momentum':
        return optim.SGD(parameters, lr=lr, momentum=0.9)
    elif name == 'Adagrad':
        return optim.Adagrad(parameters, lr=lr)
    else:
        raise ValueError(f"Optimizer {name} not recognized")
```

Function for Activation Function Selection

(tanh, sigmoid, or leaky-ReLU)

In [87]:

```
def get_activation_function(name):
    """
    Returns the activation function corresponding to the given name.

    Args:
        name (str): The name of the activation function. Options include 'ReLU', 'Tanh',
'Sigmoid', and 'LeakyReLU'.

    Returns:
        torch.nn.Module: The activation function as a PyTorch module.

    Raises:
        ValueError: If the provided activation function name is not recognized.
    """
    if name == 'ReLU':
        return nn.ReLU()
    elif name == 'Tanh':
        return nn.Tanh()
    elif name == 'Sigmoid':
        return nn.Sigmoid()
    elif name == 'LeakyReLU':
        return nn.LeakyReLU()
    else:
        raise ValueError(f"Activation function {name} not recognized")
```

Configurable MNIST DCN

In [88]:

```python
# Defining Configurable Architecture w/ Logging
class ConfNet(nn.Module):
    def __init__(self):
        super(ConfNet, self).__init__()
        self.activation = get_activation_function(ACTIVATION_FUNCTION)

        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        nn.init.xavier_uniform_(self.conv1.weight)

        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        nn.init.xavier_uniform_(self.conv2.weight)

        self.conv2_drop = nn.Dropout2d()

        self.fc1 = nn.Linear(320, 50)
        nn.init.xavier_uniform_(self.fc1.weight)

        self.fc2 = nn.Linear(50, 10)
        nn.init.xavier_uniform_(self.fc2.weight)

    def forward(self, x):
        self.stats = {}

        x = self.conv1(x)
        self.stats['conv1_weights'] = self.conv1.weight.data.cpu().numpy()
        self.stats['conv1_biases'] = self.conv1.bias.data.cpu().numpy()
        self.stats['conv1_net_input'] = x.data.cpu().numpy()

        x = self.activation(x)
        self.stats['conv1_activations_after_activation'] = x.data.cpu().numpy()

        x = F.max_pool2d(x, 2)
        self.stats['conv1_activations_after_maxpool'] = x.data.cpu().numpy()

        x = self.conv2(x)
        self.stats['conv2_weights'] = self.conv2.weight.data.cpu().numpy()
        self.stats['conv2_biases'] = self.conv2.bias.data.cpu().numpy()
        self.stats['conv2_net_input'] = x.data.cpu().numpy()

        x = self.activation(x)
        self.stats['conv2_activations_after_activation'] = x.data.cpu().numpy()

        x = F.max_pool2d(x, 2)
        self.stats['conv2_activations_after_maxpool'] = x.data.cpu().numpy()

        # (batch_size, units)
        x = x.view(-1, 320)

        x = self.fc1(x)
        self.stats['fc1_weights'] = self.fc1.weight.data.cpu().numpy()
        self.stats['fc1_biases'] = self.fc1.bias.data.cpu().numpy()
        self.stats['fc1_net_input'] = x.data.cpu().numpy()

        x = self.activation(x)
        self.stats['fc1_activations_after_activation'] = x.data.cpu().numpy()

        x = F.dropout(x, training=self.training)

        x = self.fc2(x)
        self.stats['fc2_weights'] = self.fc2.weight.data.cpu().numpy()
        self.stats['fc2_biases'] = self.fc2.bias.data.cpu().numpy()
        self.stats['fc2_net_input'] = x.data.cpu().numpy()

        x = F.softmax(x, dim=1)

        return x


model = ConfNet()
if cuda:
    model.cuda()
```

```python
optimizer = get_optimizer(OPTIMIZER, model.parameters(), lr=lr)


# Visualize network as a graph on TensorBoard
input_tensor = torch.Tensor(1, 1, 28, 28)
if cuda:
    input_tensor = input_tensor.cuda()
writer.add_graph(model, input_to_model=input_tensor)
```

```
<ipython-input-88-adf8a18bdc1e>:25: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_weights'] = self.conv1.weight.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:26: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_biases'] = self.conv1.bias.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:27: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_net_input'] = x.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:30: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_activations_after_activation'] = x.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:33: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv1_activations_after_maxpool'] = x.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:36: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_weights'] = self.conv2.weight.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:37: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_biases'] = self.conv2.bias.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:38: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_net_input'] = x.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:41: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_activations_after_activation'] = x.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:44: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['conv2_activations_after_maxpool'] = x.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:50: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc1_weights'] = self.fc1.weight.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:51: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc1_biases'] = self.fc1.bias.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:52: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
```

```
ot generalize to other inputs!
  self.stats['fc1_net_input'] = x.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:55: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc1_activations_after_activation'] = x.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:60: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc2_weights'] = self.fc2.weight.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:61: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc2_biases'] = self.fc2.bias.data.cpu().numpy()
<ipython-input-88-adf8a18bdc1e>:62: TracerWarning: Converting a tensor to a NumPy array m
ight cause the trace to be incorrect. We can't record the data flow of Python values, so
this value will be treated as a constant in the future. This means that the trace might n
ot generalize to other inputs!
  self.stats['fc2_net_input'] = x.data.cpu().numpy()
```

Configurable DCN's Training, Validation, and Testing functions

In [89]:

```
eps=1e-13
```

Training Function for Configurable DCN

In [90]:

```
def config_train(epoch):
    model.train()
    criterion = nn.NLLLoss()
    # criterion = nn.CrossEntropyLoss()

    for batch_idx, (data, target) in enumerate(train_loader):
        if cuda:
            data, target = data.cuda(), target.cuda()

        optimizer.zero_grad()
        output = model(data)   # forward
        # loss = sum_k(-t_k * log(y_k))
        loss = criterion(torch.log(output+eps), target)
        loss.backward()
        optimizer.step()

        if batch_idx % logging_interval == 0:
            print(f'Train Epoch: {epoch} \
                  [{batch_idx * len(data)}/{len(train_loader.dataset)} \
                    ({100. * batch_idx / len(train_loader):.0f}%)]\t\
                       Loss: {loss.item():.6f}')

            # Log train/loss to TensorBoard at every iteration
            n_iter = (epoch - 1) * len(train_loader) + batch_idx + 1
            writer.add_scalar('train/loss', loss.item(), n_iter)
            for key, value in model.stats.items():
                value_flat = value.flatten()
                writer.add_scalar(
                    f'statistics/{key}_mean',
                    np.mean(value_flat),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{key}_std',
                    np.std(value_flat),
                    n_iter
```

```
            )
            writer.add_scalar(
                f'statistics/{key}_min',
                np.min(value_flat),
                n_iter
            )
            writer.add_scalar(
                f'statistics/{key}_max',
                np.max(value_flat),
                n_iter
            )
            writer.add_histogram(
                f'histogram/{key}',
                value_flat,
                n_iter
            )
        for name, param in model.named_parameters():
            if 'weight' in name:
                writer.add_scalar(
                    f'statistics/{name}_min',
                    param.min().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_max',
                    param.max().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_mean',
                    param.mean().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_std',
                    param.std().item(),
                    n_iter
                )
                writer.add_histogram(
                    f'histogram/{name}',
                    param,
                    n_iter
                )
            elif 'bias' in name:
                writer.add_scalar(
                    f'statistics/{name}_min',
                    param.min().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_max',
                    param.max().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_mean',
                    param.mean().item(),
                    n_iter
                )
                writer.add_scalar(
                    f'statistics/{name}_std',
                    param.std().item(),
                    n_iter
                )
                writer.add_histogram(
                    f'histogram/{name}',
                    param,
                    n_iter
                )
```

Validation Function for Configurable DCN

Validation Function for Configurable DCN

In [91]:

```
def new_validation(epoch):
    model.eval()
    correct = 0
    valid_loss = 0
    criterion = nn.NLLLoss(size_average=False)

    for data, target in valid_loader:
        if cuda:
            data, target = data.cuda(), target.cuda()
        output = model(data)
        valid_loss += criterion(torch.log(output+eps), target).item()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

    valid_loss /= len(valid_loader.dataset)
    valid_accuracy = 100. * correct / len(valid_loader.dataset)
    print(f'Validation set: Average loss: {valid_loss:.4f}, \
          Accuracy: {correct}/{len(valid_loader.dataset)} \
          ({valid_accuracy:.2f}%)\n')

    # Log test/loss and test/accuracy to TensorBoard at every epoch
    n_iter = epoch * len(valid_loader)
    writer.add_scalar('valid/loss', valid_loss, n_iter)
    writer.add_scalar('valid/accuracy', valid_accuracy, n_iter)
    writer.add_scalar('valid/error', 100. - valid_accuracy, n_iter)
```

Testing Function for Configurable DCN

In [92]:

```
def config_testing(epoch):
    model.eval()
    correct = 0
    test_loss = 0
    criterion = nn.NLLLoss(size_average=False)
    # criterion = nn.CrossEntropyLoss(size_average=False)

    for data, target in test_loader:
        if cuda:
            data, target = data.cuda(), target.cuda()

        output = model(data)

        # sum up batch loss (later, averaged over all test samples)
        test_loss += criterion(torch.log(output+eps), target,).item()
        # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    test_loss /= len(test_loader.dataset)
    test_accuracy = 100. * correct / len(test_loader.dataset)
    print(f'\nTest set: Average loss: {test_loss:.4f}, \
          Accuracy: {correct}/{len(test_loader.dataset)} \
          ({test_accuracy:.2f}%)\n')

    # Log test/loss and test/accuracy to TensorBoard at every epoch
    n_iter = epoch * len(test_loader)
    writer.add_scalar('test/loss', test_loss, n_iter)
    writer.add_scalar('test/accuracy', test_accuracy, n_iter)
    writer.add_scalar('test/error', 100. - test_accuracy, n_iter)
```

Development Loop for Configurable DCN

In [93]:

```
for epoch in range(1, epochs + 1):
```

```
    config_train(epoch)
    new_validation(epoch)
    config_testing(epoch)
writer.close()
```

```
Train Epoch: 1                [0/54000                        (0%)]
Loss: 2.438369
Train Epoch: 1                [6400/54000                        (12%)]
Loss: 0.626828
Train Epoch: 1                [12800/54000                        (24%)]
Loss: 0.309523
Train Epoch: 1                [19200/54000                        (36%)]
Loss: 0.436894
Train Epoch: 1                [25600/54000                        (47%)]
Loss: 0.745237
Train Epoch: 1                [32000/54000                        (59%)]
Loss: 0.291944
Train Epoch: 1                [38400/54000                        (71%)]
Loss: 0.486573
Train Epoch: 1                [44800/54000                        (83%)]
Loss: 0.186915
Train Epoch: 1                [51200/54000                        (95%)]
Loss: 0.381107
Validation set: Average loss: 0.1224,          Accuracy: 5803/6000          (96.
72%)


Test set: Average loss: 0.1047,          Accuracy: 9717/10000          (97.17%)

Train Epoch: 2                [0/54000                        (0%)]
Loss: 0.208937
Train Epoch: 2                [6400/54000                        (12%)]
Loss: 0.352698
Train Epoch: 2                [12800/54000                        (24%)]
Loss: 0.164752
Train Epoch: 2                [19200/54000                        (36%)]
Loss: 0.188190
Train Epoch: 2                [25600/54000                        (47%)]
Loss: 0.398461
Train Epoch: 2                [32000/54000                        (59%)]
Loss: 0.227092
Train Epoch: 2                [38400/54000                        (71%)]
Loss: 0.080970
Train Epoch: 2                [44800/54000                        (83%)]
Loss: 0.163251
Train Epoch: 2                [51200/54000                        (95%)]
Loss: 0.311918
Validation set: Average loss: 0.1005,          Accuracy: 5848/6000          (97.
47%)


Test set: Average loss: 0.0777,          Accuracy: 9793/10000          (97.93%)

Train Epoch: 3                [0/54000                        (0%)]
Loss: 0.056179
Train Epoch: 3                [6400/54000                        (12%)]
Loss: 0.169053
Train Epoch: 3                [12800/54000                        (24%)]
Loss: 0.193481
Train Epoch: 3                [19200/54000                        (36%)]
Loss: 0.182267
Train Epoch: 3                [25600/54000                        (47%)]
Loss: 0.267626
Train Epoch: 3                [32000/54000                        (59%)]
Loss: 0.168700
Train Epoch: 3                [38400/54000                        (71%)]
Loss: 0.255610
Train Epoch: 3                [44800/54000                        (83%)]
Loss: 0.249681
Train Epoch: 3                [51200/54000                        (95%)]
Loss: 0.082164
Validation set: Average loss: 0.1381,          Accuracy: 5818/6000          (96.
```

```
97%)


Test set: Average loss: 0.1090,            Accuracy: 9741/10000            (97.41%)

Train Epoch: 4                 [0/54000                     (0%)]
Loss: 0.113561
Train Epoch: 4                 [6400/54000                     (12%)]
Loss: 0.079668
Train Epoch: 4                 [12800/54000                     (24%)]
Loss: 0.202648
Train Epoch: 4                 [19200/54000                     (36%)]
Loss: 0.102469
Train Epoch: 4                 [25600/54000                     (47%)]
Loss: 0.169789
Train Epoch: 4                 [32000/54000                     (59%)]
Loss: 0.185241
Train Epoch: 4                 [38400/54000                     (71%)]
Loss: 0.171008
Train Epoch: 4                 [44800/54000                     (83%)]
Loss: 0.260988
Train Epoch: 4                 [51200/54000                     (95%)]
Loss: 0.107392
Validation set: Average loss: 0.0988,            Accuracy: 5837/6000            (97.
28%)


Test set: Average loss: 0.0898,            Accuracy: 9739/10000            (97.39%)

Train Epoch: 5                 [0/54000                     (0%)]
Loss: 0.046984
Train Epoch: 5                 [6400/54000                     (12%)]
Loss: 0.107251
Train Epoch: 5                 [12800/54000                     (24%)]
Loss: 0.058935
Train Epoch: 5                 [19200/54000                     (36%)]
Loss: 0.071399
Train Epoch: 5                 [25600/54000                     (47%)]
Loss: 0.129493
Train Epoch: 5                 [32000/54000                     (59%)]
Loss: 0.083428
Train Epoch: 5                 [38400/54000                     (71%)]
Loss: 0.222058
Train Epoch: 5                 [44800/54000                     (83%)]
Loss: 0.192097
Train Epoch: 5                 [51200/54000                     (95%)]
Loss: 0.126780
Validation set: Average loss: 0.1011,            Accuracy: 5865/6000            (97.
75%)


Test set: Average loss: 0.0752,            Accuracy: 9815/10000            (98.15%)

Train Epoch: 6                 [0/54000                     (0%)]
Loss: 0.132886
Train Epoch: 6                 [6400/54000                     (12%)]
Loss: 0.126748
Train Epoch: 6                 [12800/54000                     (24%)]
Loss: 0.065807
Train Epoch: 6                 [19200/54000                     (36%)]
Loss: 0.119576
Train Epoch: 6                 [25600/54000                     (47%)]
Loss: 0.228738
Train Epoch: 6                 [32000/54000                     (59%)]
Loss: 0.226352
Train Epoch: 6                 [38400/54000                     (71%)]
Loss: 0.267017
Train Epoch: 6                 [44800/54000                     (83%)]
Loss: 0.222689
Train Epoch: 6                 [51200/54000                     (95%)]
Loss: 0.076570
Validation set: Average loss: 0.1059,            Accuracy: 5855/6000            (97.
```

```
58%)

Test set: Average loss: 0.0771,           Accuracy: 9817/10000           (98.17%)

Train Epoch: 7                [0/54000                     (0%)]
Loss: 0.116782
Train Epoch: 7                [6400/54000                    (12%)]
Loss: 0.061585
Train Epoch: 7                [12800/54000                    (24%)]
Loss: 0.268620
Train Epoch: 7                [19200/54000                    (36%)]
Loss: 0.129445
Train Epoch: 7                [25600/54000                    (47%)]
Loss: 0.092006
Train Epoch: 7                [32000/54000                    (59%)]
Loss: 0.217859
Train Epoch: 7                [38400/54000                    (71%)]
Loss: 0.201172
Train Epoch: 7                [44800/54000                    (83%)]
Loss: 0.120687
Train Epoch: 7                [51200/54000                    (95%)]
Loss: 0.158155
Validation set: Average loss: 0.1173,          Accuracy: 5858/6000          (97.
63%)

Test set: Average loss: 0.0781,           Accuracy: 9816/10000           (98.16%)

Train Epoch: 8                [0/54000                     (0%)]
Loss: 0.101097
Train Epoch: 8                [6400/54000                    (12%)]
Loss: 0.209340
Train Epoch: 8                [12800/54000                    (24%)]
Loss: 0.328837
Train Epoch: 8                [19200/54000                    (36%)]
Loss: 0.041354
Train Epoch: 8                [25600/54000                    (47%)]
Loss: 0.086324
Train Epoch: 8                [32000/54000                    (59%)]
Loss: 0.534101
Train Epoch: 8                [38400/54000                    (71%)]
Loss: 0.151895
Train Epoch: 8                [44800/54000                    (83%)]
Loss: 0.287575
Train Epoch: 8                [51200/54000                    (95%)]
Loss: 0.135314
Validation set: Average loss: 0.1139,          Accuracy: 5851/6000          (97.
52%)

Test set: Average loss: 0.0844,           Accuracy: 9816/10000           (98.16%)

Train Epoch: 9                [0/54000                     (0%)]
Loss: 0.217314
Train Epoch: 9                [6400/54000                    (12%)]
Loss: 0.152466
Train Epoch: 9                [12800/54000                    (24%)]
Loss: 0.553281
Train Epoch: 9                [19200/54000                    (36%)]
Loss: 0.141260
Train Epoch: 9                [25600/54000                    (47%)]
Loss: 0.192347
Train Epoch: 9                [32000/54000                    (59%)]
Loss: 0.137014
Train Epoch: 9                [38400/54000                    (71%)]
Loss: 0.159838
Train Epoch: 9                [44800/54000                    (83%)]
Loss: 0.248008
Train Epoch: 9                [51200/54000                    (95%)]
Loss: 0.398625
Validation set: Average loss: 0.1705,          Accuracy: 5837/6000          (97.
```

```
28%)


Test set: Average loss: 0.1149,            Accuracy: 9771/10000            (97.71%)

Train Epoch: 10                  [0/54000                     (0%)]
Loss: 0.237380
Train Epoch: 10                  [6400/54000                     (12%)]
Loss: 0.119530
Train Epoch: 10                  [12800/54000                     (24%)]
Loss: 0.410528
Train Epoch: 10                  [19200/54000                     (36%)]
Loss: 0.025118
Train Epoch: 10                  [25600/54000                     (47%)]
Loss: 0.183509
Train Epoch: 10                  [32000/54000                     (59%)]
Loss: 0.073769
Train Epoch: 10                  [38400/54000                     (71%)]
Loss: 0.096612
Train Epoch: 10                  [44800/54000                     (83%)]
Loss: 0.105562
Train Epoch: 10                  [51200/54000                     (95%)]
Loss: 0.043238
Validation set: Average loss: 0.1181,            Accuracy: 5857/6000            (97.
62%)


Test set: Average loss: 0.0936,            Accuracy: 9802/10000            (98.02%)
```