In [ ]:
```
# Yihao Zhong Larry, yz7654, date: 02/23/2024 - onwards
#
```

Problem 1 - Softmax Activation Function

1.1. logarithmic derivative

Given that for each real-valued outputs v_1, v_2, ..., v_k, $o_i = \frac{exp(v_i)}{\sum_{j=1}^k exp(v_j)}$

For the case where $i = j$, we consider the derivative of o_i with respect to v_i.

Take logarithm:

$$ln(o_i) = ln(\frac{exp(v_i)}{\sum_{j=1}^k exp(v_j)}) = v_i - ln(\sum_{j=1}^k exp(v_j))$$

Taking the derivative of both sides with respect to v_i:

$$\frac{d}{dv_i} ln(o_i) = 1 - \frac{1}{\sum_{j=1}^k exp(v_j)} e^{v_i}$$

Since $o_i = \frac{exp(v_i)}{\sum_{j=1}^k exp(v_j)}$, we have

$$\frac{d}{dv_i} ln(o_i) = 1 - o_i$$

To find $\frac{\partial o_i}{\partial v_j}$, we know that $\frac{\partial}{\partial x_j} ln(f(x)) = \frac{1}{f(x)} \frac{\partial f(x)}{\partial x_j}$ by logarithmic derivative.

Therefore,

$$\frac{\partial o_i}{\partial v_j} = o_i(1 - o_i)$$

For the case $i \neq j$, we have start similarly by applying the logarithm to i, but take the derivative with respect to v_j.

Take logarithm:

$$ln(o_i) = ln(\frac{exp(v_i)}{\sum_{j=1}^k exp(v_j)})$$

Taking the derivative with respect to v_j:

$$\frac{d}{dv_j} ln(o_i) = -\frac{e^{v_j}}{\sum_{j=1}^k exp(v_j)}$$

Simply this by definition of v_j we have:

$$\frac{d}{dv_j} ln(o_i) = -o_j$$

To find $\frac{\partial o_i}{\partial v_j}$, we know that $\frac{\partial}{\partial x_j} ln(f(x)) = \frac{1}{f(x)} \frac{\partial f(x)}{\partial x_j}$ by logarithmic derivative.

Therefore,

$$\frac{\partial o_i}{\partial v_j} = -o_i o_j$$

1.2.

We know that $\frac{\partial L}{\partial v_i} = \frac{\partial L}{\partial o_i} \frac{\partial o_i}{\partial v_i} = -\sum_{i=1}^k \frac{y_i}{o_i} \frac{\partial o_i}{\partial v_i}$

We simply it using kronecker delta: $\frac{\partial L}{\partial v_i} = -\sum_{j=1}^k \frac{y_j}{o_j} \left(\delta_{ij} o_i (1 - o_i) - (1 - \delta_{ij}) o_i o_j\right)$

That is: δ_{ij} is the kronecker delta, this is 1 if $i = j$ and 0 otherwise

Therefore:

$$\frac{\partial L}{\partial v_i} = \sum_{j=1}^k \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial v_i}$$

$$\frac{\partial L}{\partial v_i} = -\sum_{j=1}^{k} \frac{y_j}{o_j}\left(\delta_{ij} o_i (1 - o_i) - (1 - \delta_{ij}) o_i o_j\right)$$

$$\frac{\partial L}{\partial v_i} = \frac{y_1}{o_1} o_1 o_i + \frac{y_2}{o_2} o_2 o_i + \ldots - \frac{y_i}{o_i} o_i (1 - o_i) + \ldots + \frac{y_k}{o_k} o_k o_i$$

$$\frac{\partial L}{\partial v_i} = y_1 o_i + y_2 o_i + \ldots - y_i (1 - o_i) + \ldots + y_k o_i$$

$$\frac{\partial L}{\partial v_i} = o_i - y_i$$

Hence we have prove: $\frac{\partial L}{\partial v_i} = o_i - y_i$

Problem 2 - Neural Network Training and Backpropagation

In [ ]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat
%matplotlib inline

data = loadmat('ex3data1.mat')
X = data['X']
y = data['y']

print(X.shape, y.shape)

from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse=False)
y_onehot = encoder.fit_transform(y)
print(y_onehot.shape)
```

```
(5000, 400) (5000, 1)
(5000, 10)
/Users/zhongyihao/anaconda3/lib/python3.10/site-packages/sklearn/preprocessing/_encoders.py:828: FutureWarning: `spar
se` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `sparse_output` is ignored unless you l
eave `sparse` to its default value.
  warnings.warn(
```

2.1.

In [ ]:
```python
def scaled_sigmoid(z):
    return 1 / (1 + np.exp(-2 * z))
```

2.2. forward_propagate()

In [ ]:
```python
def forward_propagate(X, theta1, theta2, theta3):
    m = X.shape[0]

    # Input layer activations (adding bias unit)
    a1 = np.insert(X, 0, values=np.ones(m), axis=1)

    z2 = a1 * (theta1.T)
    a2 = np.insert(scaled_sigmoid(z2), 0, values=np.ones(m), axis=1)

    z3 = a2 * (theta2.T)
    a3 = np.insert(scaled_sigmoid(z3), 0, values=np.ones(m), axis=1)

    z4 = a3 * (theta3.T)
    h = scaled_sigmoid(z4)

    return a1, z2, a2, z3, a3, z4, h
```

2.3. cost()

In [ ]:
```python
def cost(params, input_size, hidden_size1, hidden_size2, num_labels, X, y, learning_rate):
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)

    end_theta1 = hidden_size1 * (input_size + 1)
    end_theta2 = end_theta1 + hidden_size2 * (hidden_size1 + 1)
    theta1 = np.reshape(params[:end_theta1], (hidden_size1, input_size + 1))
    theta2 = np.reshape(params[end_theta1:end_theta2], (hidden_size2, hidden_size1 + 1))
    theta3 = np.reshape(params[end_theta2:], (num_labels, hidden_size2 + 1))

    # just need the output of the last layer
    _, _, _, _, _, _, h = forward_propagate(X, theta1, theta2, theta3)

    # Compute the cost
    J = 0
    for i in range(m):
        first_term = np.multiply(-y[i, :], np.log(h[i, :]))
        second_term = np.multiply((1 - y[i, :]), np.log(1 - h[i, :]))
        J += np.sum(first_term - second_term)
    J = J / m

    return J

def cost_with_regulazation(params, input_size, hidden_size1, hidden_size2, num_labels, X, y, learning_rate):
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)

    end_theta1 = hidden_size1 * (input_size + 1)
    end_theta2 = end_theta1 + hidden_size2 * (hidden_size1 + 1)
    theta1 = np.reshape(params[:end_theta1], (hidden_size1, input_size + 1))
    theta2 = np.reshape(params[end_theta1:end_theta2], (hidden_size2, hidden_size1 + 1))
    theta3 = np.reshape(params[end_theta2:], (num_labels, hidden_size2 + 1))

    # just need the output of the last layer
    _, _, _, _, _, _, h = forward_propagate(X, theta1, theta2, theta3)

    # Compute the cost
    J = 0
    for i in range(m):
        first_term = np.multiply(-y[i, :], np.log(h[i, :]))
        second_term = np.multiply((1 - y[i, :]), np.log(1 - h[i, :]))
        J += np.sum(first_term - second_term)
    J = J / m

    J += (float(learning_rate) / (2 * m)) \
           * (np.sum(np.square(theta1[:, 1:])) \
              + np.sum(np.square(theta2[:, 1:])) + np.sum(np.square(theta3[:, 1:])))
    return J
```

In [ ]:

```
# Initial setup
input_size = 400
hidden_size1 = 20
hidden_size2 = 20
num_labels = 10
learning_rate = 1

m = X.shape[0]
X = np.matrix(X)
y = np.matrix(y)

params_size = (input_size + 1) * hidden_size1 + (hidden_size1 + 1) * hidden_size2 + (hidden_size2 + 1) * num_labels
params = (np.random.random(size=params_size) - 0.5) * 0.25

end_theta1 = hidden_size1 * (input_size + 1)
end_theta2 = end_theta1 + hidden_size2 * (hidden_size1 + 1)

theta1 = np.matrix(np.reshape(params[:end_theta1], (hidden_size1, input_size + 1)))
theta2 = np.matrix(np.reshape(params[end_theta1:end_theta2], (hidden_size2, hidden_size1 + 1)))
theta3 = np.matrix(np.reshape(params[end_theta2:], (num_labels, hidden_size2 + 1)))

print(f"Theta1 shape: {theta1.shape}")
print(f"Theta2 shape: {theta2.shape}")
print(f"Theta3 shape: {theta3.shape}")
```

```
Theta1 shape: (20, 401)
Theta2 shape: (20, 21)
Theta3 shape: (10, 21)
```

In [ ]:

```
print("cost without regularization: ",
      cost(params, input_size, hidden_size1, hidden_size2, num_labels, X, y_onehot, learning_rate))
print("cost with regularization: ",
      cost_with_regulazation(params, input_size, hidden_size1, hidden_size2, num_labels, X, y_onehot, learning_rate))
a1, z2, a2, z3, a3, z4, h = forward_propagate(X, theta1, theta2, theta3)
print(f"a1: {a1.shape}, z2: {z2.shape}, a2: {a2.shape}, z3: {z3.shape}, a3: {a3.shape}, z4: {z4.shape}, h: {h.shape}"
```

```
cost without regularization:  7.072360984488249
cost with regularization:  7.076853605717871
a1: (5000, 401), z2: (5000, 20), a2: (5000, 21), z3: (5000, 20), a3: (5000, 21), z4: (5000, 10), h: (5000, 10)
```

2.4. sigmoid gradient

In [ ]:

```
def scaled_sigmoid_gradient(z):
    return np.multiply(scaled_sigmoid(z), (1 - scaled_sigmoid(z)))
```

2.5. backprop()

In [ ]:

```
def get_theta_shapes(input_size, hidden_size1, hidden_size2, num_labels):
    """
    Returns the shapes of theta matrices for each layer in the network.
    """
    theta1_shape = (hidden_size1, input_size + 1)
    theta2_shape = (hidden_size2, hidden_size1 + 1)
    theta3_shape = (num_labels, hidden_size2 + 1)
    return theta1_shape, theta2_shape, theta3_shape
```

```python
def reshape_params(params, shapes):
    """
    Reshapes the flattened parameter array into individual theta matrices.
    """
    theta1_shape, theta2_shape, theta3_shape = shapes
    size1 = theta1_shape[0] * theta1_shape[1]
    size2 = size1 + theta2_shape[0] * theta2_shape[1]

    theta1 = params[:size1].reshape(theta1_shape)
    theta2 = params[size1:size2].reshape(theta2_shape)
    theta3 = params[size2:].reshape(theta3_shape)

    return theta1, theta2, theta3
```

In [ ]:
```python
def backprop(params, input_size, hidden_size1, hidden_size2, num_labels, X, y, learning_rate):
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)

    theta_shapes = get_theta_shapes(input_size, hidden_size1, hidden_size2, num_labels)
    theta1, theta2, theta3 = reshape_params(params, theta_shapes)

    a1, z2, a2, z3, a3, z4, h = forward_propagate(X, theta1, theta2, theta3)

    # Gradients initialization
    delta1 = np.zeros(theta1.shape)
    delta2 = np.zeros(theta2.shape)
    delta3 = np.zeros(theta3.shape)

    # Compute cost
    J = cost_with_regulazation(params, input_size, hidden_size1, hidden_size2, num_labels, X, y, learning_rate)

    # Compute gradients (Backpropagation)
    for t in range(m):
        a1t = a1[t,:]
        z2t = z2[t,:]
        a2t = a2[t,:]
        ht = h[t,:]
        yt = y[t,:]

        d4 = ht - yt
        d3 = np.multiply((theta3.T * d4.T).T, scaled_sigmoid_gradient(np.insert(z3[t], 0, values=1)))
        d2 = np.multiply((theta2.T * d3[:,1:].T).T, scaled_sigmoid_gradient(np.insert(z2[t], 0, values=1)))

        delta3 += d4.T * a3[t]
        delta2 += d3[:,1:].T * a2t
        delta1 += d2[:,1:].T * a1t

    delta1 = delta1 / m
    delta2 = delta2 / m
    delta3 = delta3 / m

    # Unroll gradients
    grad = np.concatenate((np.ravel(delta1), np.ravel(delta2), np.ravel(delta3)))


    return J, grad

def backprop_with_reg(params, input_size, hidden_size1, hidden_size2, num_labels, X, y, learning_rate):
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)

    theta_shapes = get_theta_shapes(input_size, hidden_size1, hidden_size2, num_labels)
    theta1, theta2, theta3 = reshape_params(params, theta_shapes)

    a1, z2, a2, z3, a3, z4, h = forward_propagate(X, theta1, theta2, theta3)

    # Gradients initialization
    delta1 = np.zeros(theta1.shape)
    delta2 = np.zeros(theta2.shape)
    delta3 = np.zeros(theta3.shape)

    # Compute cost
    J = cost_with_regulazation(params, input_size, hidden_size1, hidden_size2, num_labels, X, y, learning_rate)

    # Compute gradients (Backpropagation)
    for t in range(m):
        a1t = a1[t,:]
        z2t = z2[t,:]
        a2t = a2[t,:]
        ht = h[t,:]
        yt = y[t,:]

        d4 = ht - yt
        d3 = np.multiply((theta3.T * d4.T).T, scaled_sigmoid_gradient(np.insert(z3[t], 0, values=1)))
        d2 = np.multiply((theta2.T * d3[:,1:].T).T, scaled_sigmoid_gradient(np.insert(z2[t], 0, values=1)))

        delta3 += d4.T * a3[t]
        delta2 += d3[:,1:].T * a2t
        delta1 += d2[:,1:].T * a1t
    delta1 = delta1 / m
    delta2 = delta2 / m
    delta3 = delta3 / m

    delta1[:, 1:] += (theta1[:, 1:] * learning_rate) / m
    delta2[:, 1:] += (theta2[:, 1:] * learning_rate) / m
    delta3[:, 1:] += (theta3[:, 1:] * learning_rate) / m

    # Unroll gradients
    grad = np.concatenate((np.ravel(delta1), np.ravel(delta2), np.ravel(delta3)))

    return J, grad
```

In [ ]:
```
J, grad = backprop(params, input_size, hidden_size1, hidden_size2, num_labels, X, y_onehot, learning_rate)
print(f"Backpropagation cost without regularization: {J}, grad shape: {grad.shape}")

J_reg, grad_reg = backprop_with_reg(params, input_size, hidden_size1, hidden_size2, num_labels, X, y_onehot, learning_
print(f"Backpropagation cost with regularization: {J_reg}, grad shape: {grad_reg.shape}")
```

```
Backpropagation cost without regularization: 7.076853605717871, grad shape: (8650,)
Backpropagation cost with regularization: 7.076853605717871, grad shape: (8650,)
```

2.6. & 2.7. Train

In [ ]:
```
from scipy.optimize import minimize

options = {'maxiter': 250}


result = minimize(fun=backprop_with_reg, x0=params,
                  args=(input_size, hidden_size1, hidden_size2, num_labels, X, y_onehot, learning_rate),
                  method='TNC', jac=True, options=options)


params_optimized = result.x
cost_optimized = result.fun

print(result)
print(f"Optimized cost: {cost_optimized}")
```

```
/var/folders/v1/6k_h9wg90q56lxft6dxycw200000gn/T/ipykernel_75688/1817137912.py:6: DeprecationWarning: 'maxiter' has b
een deprecated in favor of 'maxfun' and will be removed in SciPy 1.11.0.
  result = minimize(fun=backprop_with_reg, x0=params,
 message: Max. number of function evaluations reached
 success: False
  status: 3
     fun: 0.17435324897494292
       x: [-3.269e-01 -9.918e-03 ... -7.996e-01  9.404e-01]
     nit: 31
     jac: [-6.816e-04 -1.984e-06 ... -3.383e-05 -1.583e-04]
    nfev: 250
Optimized cost: 0.17435324897494292
```

In [ ]:
```
end_theta1 = hidden_size1 * (input_size + 1)
end_theta2 = end_theta1 + hidden_size2 * (hidden_size1 + 1)


theta1 = np.matrix(np.reshape(result.x[:end_theta1], (hidden_size1, input_size + 1)))
theta2 = np.matrix(np.reshape(result.x[end_theta1:end_theta2], (hidden_size2, hidden_size1 + 1)))
theta3 = np.matrix(np.reshape(result.x[end_theta2:], (num_labels, hidden_size2 + 1)))


X = np.matrix(X)
# X = np.insert(X, 0, values=np.ones(X.shape[0]), axis=1)

a1, z2, a2, z3, a3, z4, h = forward_propagate(X, theta1, theta2, theta3)


y_pred = np.array(np.argmax(h, axis=1) + 1)

# y = np.squeeze(np.array(y))

correct = [1 if a == b else 0 for (a, b) in zip(y_pred, y)]
accuracy = (sum(correct) / float(len(correct)))
print('Accuracy = {0}%'.format(accuracy * 100))
```

```
Accuracy = 99.52%
```

2.8.

We achieve an accuracy of 99.52% with a 3-layer neural network compared to 99.2% with a 2-layer network from the notebooks. Basically, we add an additional hidden layer increases the complexity of the model. This means the network has more parameters (weights and biases) to adjust during training, allowing it to capture more intricate patterns in the data. A deeper network can learn more complex function. By adding more layers, the network can create more complex representations of the input data.

Problem 3 - Weight Initialization, Dead Neurons, Leaky ReLU

3.1.

In [ ]:

```python
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Dropout, Flatten
from keras import backend as K
import keras
from matplotlib import pyplot as plt
from matplotlib import rcParamsDefault
import tensorflow as tf
## use of code from github.com/intoli/intoli-article-materials/blob/master/articles/neural-network-initialization/uti

def grid_axes_it(n_plots, n_cols=3, enumerate=False, fig=None):
    """
    Iterate through Axes objects on a grid with n_cols columns and as many
    rows as needed to accommodate n_plots many plots.

    Args:
        n_plots: Number of plots to plot onto figure.
        n_cols: Number of columns to divide the figure into.
        fig: Optional figure reference.

    Yields:
        n_plots many Axes objects on a grid.
    """
    n_rows = int(n_plots / n_cols + int(n_plots % n_cols > 0))

    if not fig:
        default_figsize = rcParamsDefault['figure.figsize']
        fig = plt.figure(figsize=(
            default_figsize[0] * n_cols,
            default_figsize[1] * n_rows
        ))

    for i in range(1, n_plots + 1):
        ax = plt.subplot(n_rows, n_cols, i)
        yield ax


def create_mlp_model(
    n_hidden_layers,
    dim_layer,
    input_shape,
    n_classes,
    kernel_initializer,
    bias_initializer,
    activation,
):
    """Create Multi-Layer Perceptron with given parameters."""
    model = Sequential()
    model.add(Dense(dim_layer, input_shape=input_shape, kernel_initializer=kernel_initializer,
                    bias_initializer=bias_initializer))
    for i in range(n_hidden_layers):
        model.add(Dense(dim_layer, activation=activation, kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer))
    model.add(Dense(n_classes, activation='softmax', kernel_initializer=kernel_initializer,
                    bias_initializer=bias_initializer))
    return model


def create_cnn_model(input_shape, num_classes, kernel_initializer='glorot_uniform',
                     bias_initializer='zeros'):
    """Create CNN model similar to
       https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py."""
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=input_shape,
                     kernel_initializer=kernel_initializer,
                     bias_initializer=bias_initializer))
    model.add(Conv2D(64, (3, 3), activation='relu',
                     kernel_initializer=kernel_initializer,
                     bias_initializer=bias_initializer))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(128, activation='relu',
                    kernel_initializer=kernel_initializer,
                    bias_initializer=bias_initializer))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax',
                    kernel_initializer=kernel_initializer,
                    bias_initializer=bias_initializer))
    return model


def compile_model(model):
    model.compile(loss = keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.legacy.RMSprop(),
                  metrics=['accuracy'])
    return model
```

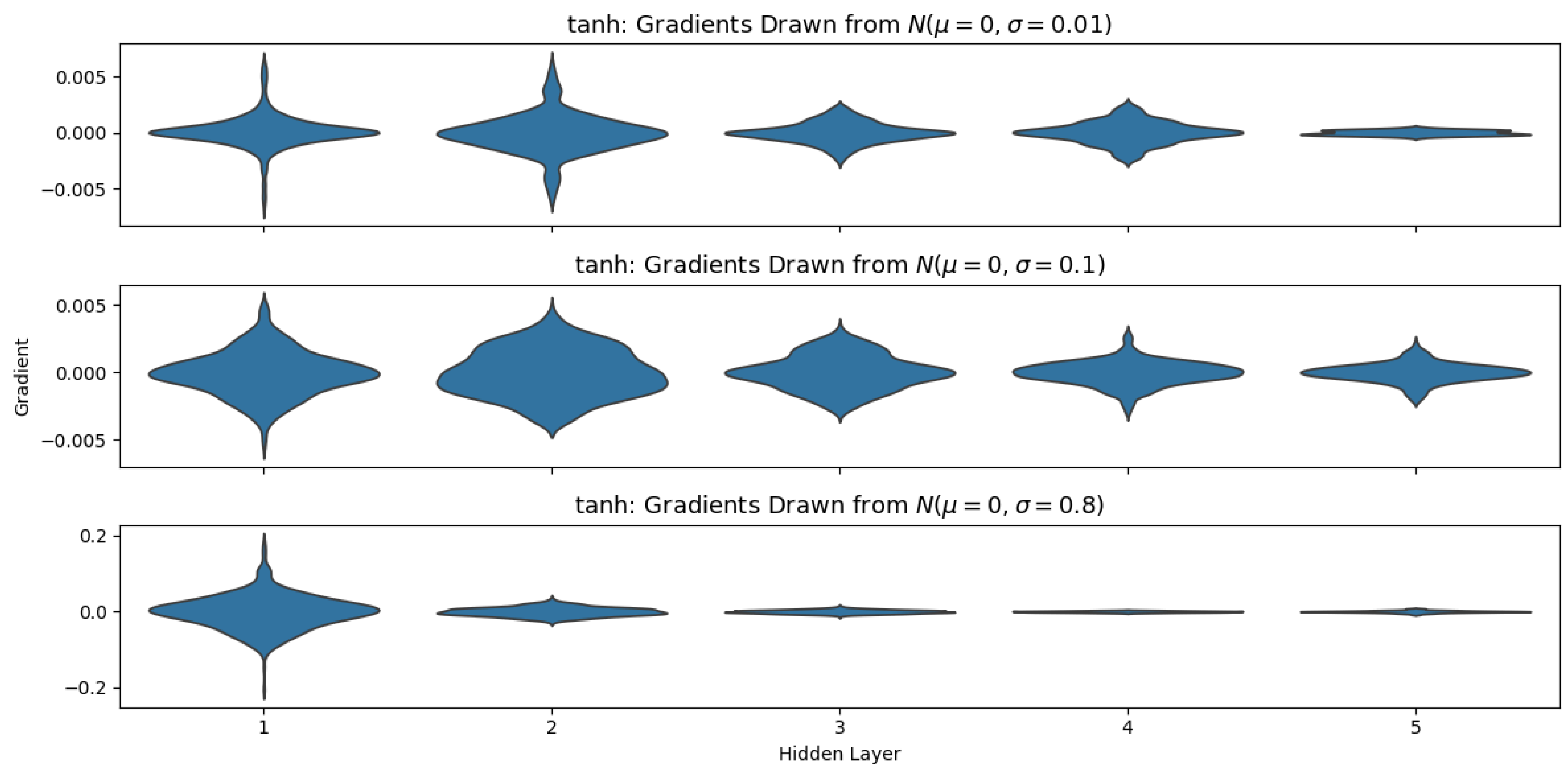
```
def get_init_id(init):
    """
    Returns string ID summarizing initialization scheme and its parameters.

    Args:
        init: Instance of some initializer from keras.initializers.
    """
    try:
        init_name = str(init).split('.')[2].split(' ')[0]
    except:
        init_name = str(init).split(' ')[0].replace('.', '_')

    param_list = []
    config = init.get_config()
    for k, v in config.items():
        if k == 'seed':
            continue
        param_list.append('{k}-{v}'.format(k=k, v=v))
    init_params = '__'.join(param_list)

    return '|'.join([init_name, init_params])


def get_activations(model, x, mode=0.0):
    """Extract activations with given model and input vector x."""
    outputs = [layer.output for layer in model.layers]
    activations = K.function([model.input], outputs)
    output_elts = activations([x, mode])
    return output_elts


class LossHistory(keras.callbacks.Callback):
    """A custom keras callback for recording losses during network training."""

    def on_train_begin(self, logs={}):
        self.losses = []
        self.epoch_losses = []
        self.epoch_val_losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))

    def on_epoch_end(self, epoch, logs={}):
        self.epoch_losses.append(logs.get('loss'))
        self.epoch_val_losses.append(logs.get('val_loss'))
```

In [ ]:
```
def calculate_gradients(network, inputs, targets):

    loss_function = tf.keras.losses.CategoricalCrossentropy()


    with tf.GradientTape(persistent= True) as gradient_tape:
        predictions = network(inputs)
        loss = loss_function(targets, predictions)


    gradients_list = []

    for layer in network.layers:
        if layer.trainable:
            for weight in layer.trainable_weights:
                if 'bias' not in weight.name:
                    weight_gradient = gradient_tape.gradient(loss, weight)
                    gradients_list.append(weight_gradient)

    return gradients_list
```

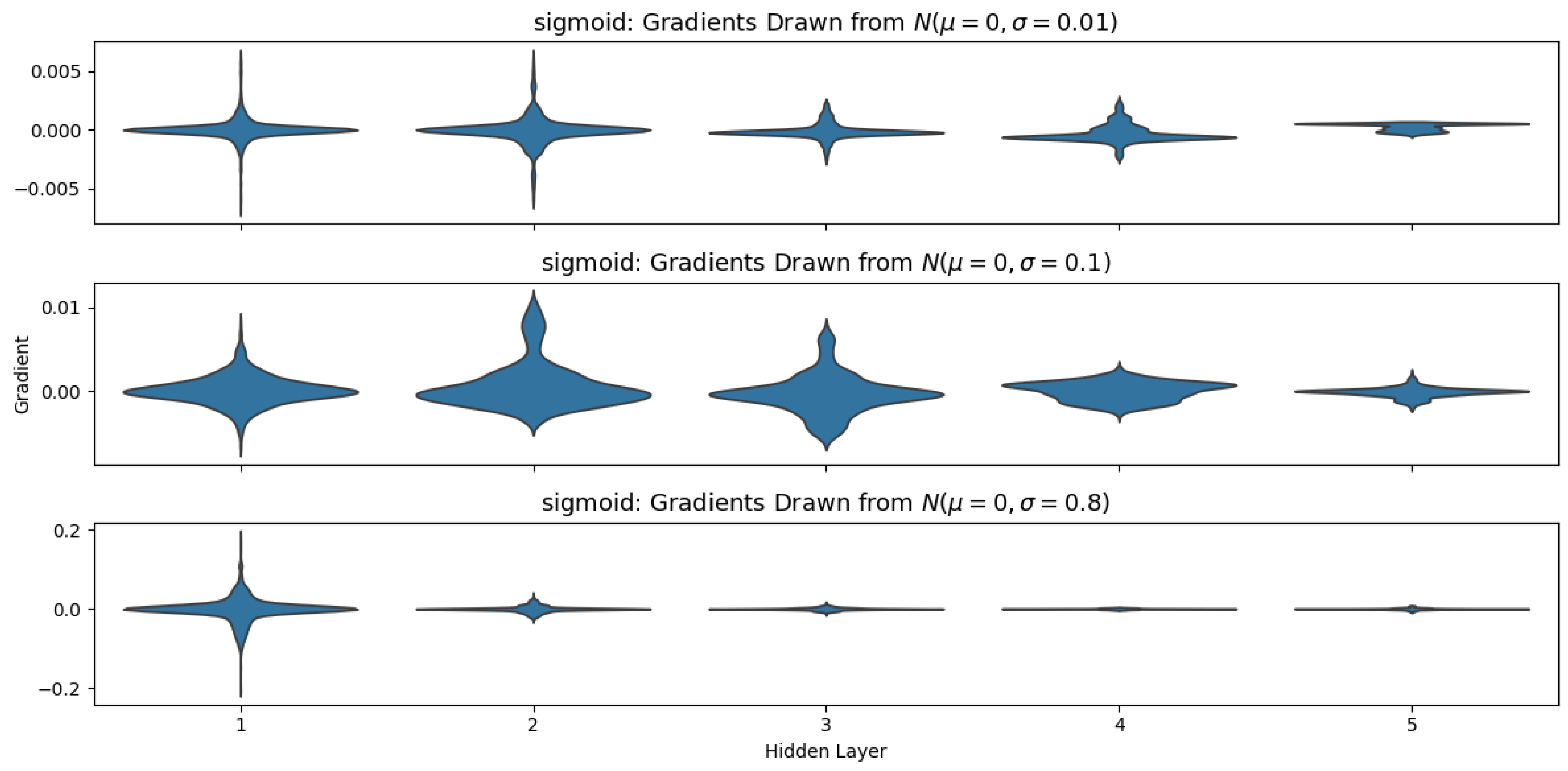
In [ ]:
```
import keras
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from keras import initializers
from keras.datasets import mnist
import warnings
warnings.filterwarnings('ignore')


seed = 10

# Number of points to plot
n_train = 1000
n_test = 100
n_classes = 10

# Network params
n_hidden_layers = 5
dim_layer = 100
batch_size = n_train
epochs = 1
```

```python
# Load and prepare MNIST dataset.
n_train = 60000
n_test = 10000

(x_train, y_train), (x_test, y_test) = mnist.load_data()
num_classes = len(np.unique(y_test))
data_dim = 28 * 28

x_train = x_train.reshape(60000, 784).astype('float32')[:n_train]
x_test = x_test.reshape(10000, 784).astype('float32')[:n_train]
x_train /= 255
x_test /= 255

y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)


rows = []
sigmas = [0.01, 0.1, 0.8]
activations = ['tanh', 'sigmoid']
for activation in activations:
    for stddev in sigmas:
        init = initializers.RandomNormal(mean=0.0, stddev=stddev, seed=seed)
        activation = activation

        model = create_mlp_model(
            n_hidden_layers,
            dim_layer,
            (data_dim,),
            n_classes,
            init,
            'zeros',
            activation
        )
        compile_model(model)
        model.fit(x_train, y_train)
        output_elts = calculate_gradients(model, x_test, y_test)
        n_layers = len(model.layers)
        i_output_layer = n_layers - 1

        for i, out in enumerate(output_elts[:-1]):
            if i > 0 and i != i_output_layer:
                for out_i in out.numpy().ravel()[::20]:
                    rows.append([i, stddev, out_i])

    df = pd.DataFrame(rows, columns=['Hidden Layer', 'Standard Deviation', 'Output'])

    fig = plt.figure(figsize=(12, 6))
    axes = grid_axes_it(len(sigmas), 1, fig=fig)

    for sig in sigmas:
        ax = next(axes)
        ddf = df[df['Standard Deviation'] == sig]
        sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count', inner=None)

        ax.set_xlabel('')
        ax.set_ylabel('')

        ax.set_title(f'{activation}: Gradients Drawn from $N(\mu = 0, \sigma = {sig})$', fontsize=13)

        if sig == sigmas[1]:
            ax.set_ylabel("Gradient")
        if sig != sigmas[-1]:
            ax.set_xticklabels(())
        else:
            ax.set_xlabel("Hidden Layer")
# optimizer `tf.keras.optimizers.RMSprop` runs slowly on M1/M2 Macs, please use the legacy Keras optimizer instead
    plt.tight_layout()
    plt.show()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
1875/1875 [==============================] - 7s 3ms/step - loss: 1.4614 - accuracy: 0.4328
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2643 - accuracy: 0.9190
1875/1875 [==============================] - 6s 3ms/step - loss: 5.2943 - accuracy: 0.2697
```



```
1875/1875 [==============================] - 6s 3ms/step - loss: 2.1959 - accuracy: 0.1323
1875/1875 [==============================] - 6s 3ms/step - loss: 0.8682 - accuracy: 0.7118
1875/1875 [==============================] - 6s 3ms/step - loss: 0.8815 - accuracy: 0.7191
```



For the tanh activation function:

With σ = 0.01, the gradients are relatively concentrated around zero but with a noticeable spread, suggesting moderate variance in the gradient values. With σ = 0.1, the gradients have a more spread. This suggests that while the gradients are small, they are still significantly different from zero, which is good for learning. With σ = 0.8, the gradients are very small and tightly concentrated around zero, especially in deeper layers. This is vanishing gradients problem, where gradients become smaller as we move through the layers during backpropagation, leading to very little or no learning in the initial layers of the network.

For the sigmoid activation function:

With σ = 0.01, the gradients are small and concentrated, even smaller than with the tanh activation, which is expected because the sigmoid function saturates more easily than tanh.

With σ = 0.1, there is a slight increase in the spread, but they are still quite small, so that learning is slow.

With σ = 0.8, the gradients are almost non-existent in the deeper layers, which is the vanishing gradients problem. This is that large initial weights can be detrimental to learning, especially with sigmoid activations.

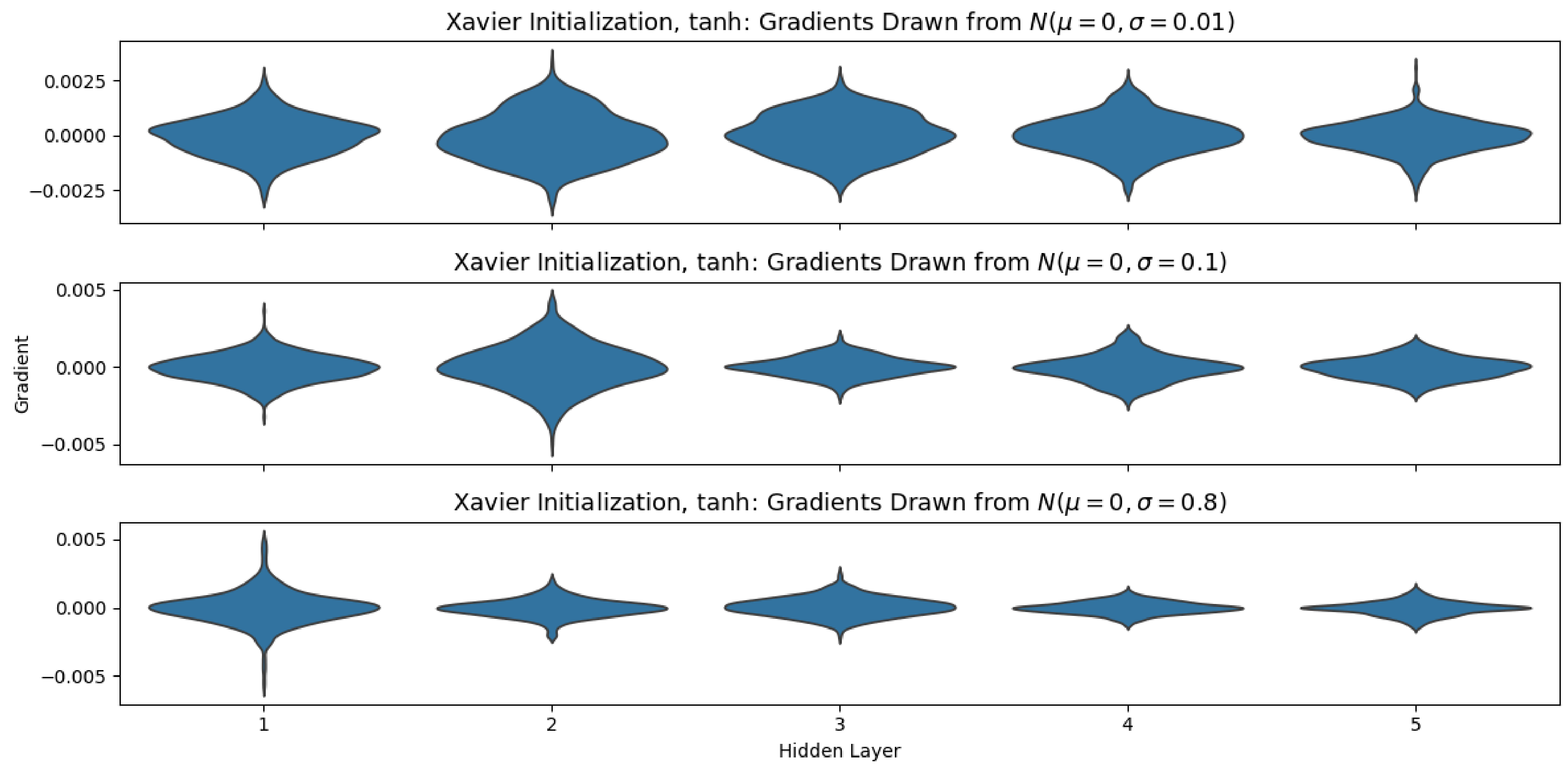
Vanishing Gradients Explanation:

The vanishing gradients phenomenon is observed when the gradients of the network's loss with respect to the weights become increasingly smaller as we move backward from the output layer to the input layer. This is problematic in deep networks with many layers and when doing backprogation. If the gradients are small, multiplying many of these small numbers together will make the gradient even smaller as it propagates backward through the network.

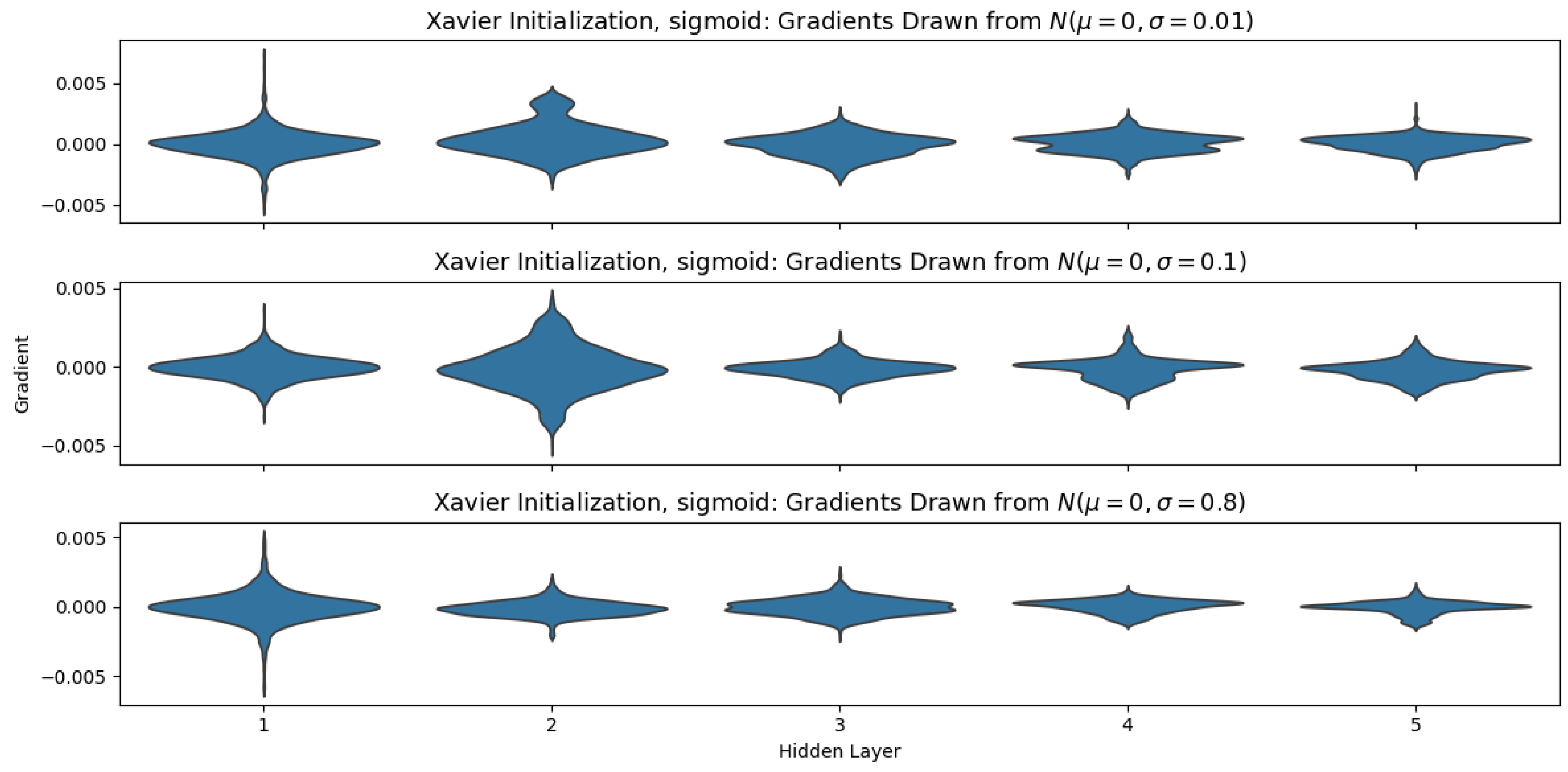
3.2

In [ ]:
```
seed = 10

# Number of points to plot
n_train = 1000
n_test = 100
n_classes = 10

# Network params
n_hidden_layers = 5
dim_layer = 100
batch_size = n_train
epochs = 1

# Load and prepare MNIST dataset.
n_train = 60000
n_test = 10000

(x_train, y_train), (x_test, y_test) = mnist.load_data()
num_classes = len(np.unique(y_test))
data_dim = 28 * 28

x_train = x_train.reshape(60000, 784).astype('float32')[:n_train]
x_test = x_test.reshape(10000, 784).astype('float32')[:n_train]
x_train /= 255
x_test /= 255

y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Run the data through a few MLP models and save the activations from
# each layer into a Pandas DataFrame.
rows = []
sigmas = [0.01, 0.1, 0.8]
activations = ['tanh', 'sigmoid']
for activation in activations:
    for stddev in sigmas:
        init = initializers.GlorotNormal(seed=seed)
        activation = activation

        model = create_mlp_model(
            n_hidden_layers,
            dim_layer,
            (data_dim,),
            n_classes,
            init,
            'zeros',
            activation
        )
        compile_model(model)
        model.fit(x_train, y_train)
        output_elts = calculate_gradients(model, x_test, y_test)
        n_layers = len(model.layers)
        i_output_layer = n_layers - 1

        for i, out in enumerate(output_elts[:-1]):
            if i > 0 and i != i_output_layer:
                for out_i in out.numpy().ravel()[::20]:
                    rows.append([i, stddev, out_i])

    df = pd.DataFrame(rows, columns=['Hidden Layer', 'Standard Deviation', 'Output'])

    fig = plt.figure(figsize=(12, 6))
    axes = grid_axes_it(len(sigmas), 1, fig=fig)

    for sig in sigmas:
        ax = next(axes)
        ddf = df[df['Standard Deviation'] == sig]
        sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count', inner=None)

        ax.set_xlabel('')
        ax.set_ylabel('')

        ax.set_title(f'Xavier Initialization, {activation}: Gradients Drawn from $N(\mu = 0, \sigma = {sig})$', fonts

        if sig == sigmas[1]:
            ax.set_ylabel("Gradient")
        if sig != sigmas[-1]:
            ax.set_xticklabels(())
        else:
            ax.set_xlabel("Hidden Layer")

    plt.tight_layout()
    plt.show()
```

```
1875/1875 [==============================] - 7s 3ms/step - loss: 0.2825 - accuracy: 0.9147
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2814 - accuracy: 0.9145
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2829 - accuracy: 0.9142
```



```
1875/1875 [==============================] - 6s 3ms/step - loss: 0.7765 - accuracy: 0.7441
1875/1875 [==============================] - 6s 3ms/step - loss: 0.7525 - accuracy: 0.7571
1875/1875 [==============================] - 6s 3ms/step - loss: 0.8179 - accuracy: 0.7303
```



with tanh Activation Function:

For all three standard deviation values, the gradients are more evenly distributed across the layers compared to using RandomNormal. The spread of the gradient values is consistent across the layers, which indicates that the Xavier initialization is maintaining a healthy flow of gradients. And there is no sign of vanishing gradients, even in deeper layer with different random draw from initialization.

with sigmoid Activation Function:

Similar to the tanh activation function, the gradients are more consistent across layers. I think Xavier initialization is preventing the gradients from vanishing too much in deeper layers, which is a common problem with sigmoid functions when not using a proper initialization technique. The gradients are maintained across different standard deviations.

The gradients in the Xavier initialization are less likely to explode or vanish, offering a improvement over RandomNormal for both activation functions.

3.3.

In [ ]:

```python
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import RMSprop


num_simulations = 1000
num_points = 3000
num_hidden_layers = 10
units_per_layer = 2
```

```
batch_size = 64


relu_collapses = 0

relu = 'relu'


def create_network(activation):
    model = Sequential()
    model.add(Dense(units_per_layer, input_shape=(1,), activation=activation))
    for _ in range(num_hidden_layers - 1):
        model.add(Dense(units_per_layer, activation=activation))
    model.add(Dense(1, activation='relu'))  # Output layer
    return model

def perform_experiment(activation_function):
    collapses = 0
    for i in range(num_simulations):
        if i % 100 == 0:
          print(f"{i}th sim")
        # Generate uniform data from the interval [-7, 7]
        x_train, x_test = np.random.uniform(-7, 7, num_points), np.random.uniform(-7, 7, int(num_points*0.25))
        y_train, y_test = np.abs(x_train), np.abs(x_test)

        model = create_network(activation_function)
        model.compile(optimizer=RMSprop(), loss='mse')


        model.fit(x_train, y_train, batch_size=batch_size, epochs=1, verbose=0)

        if len(np.unique(model.predict(x_test, verbose = 0).flatten())) == 1:
            collapses += 1

    return collapses


relu_collapses = perform_experiment(relu)

print(f"Fraction of collapses with ReLU: {relu_collapses / num_simulations}")
```

```
0th sim
100th sim
200th sim
300th sim
400th sim
500th sim
600th sim
700th sim
800th sim
900th sim
Fraction of collapses with ReLU: 0.954
```

I chose f(x) = |x|. The collapse fraction out of 1000 sim is 95.4%, which is close to Lu et al, showing a similar problem of dying ReLU.

3.4. Discussion

In [ ]:
```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import RMSprop

num_simulations = 1000
num_points = 3000
num_hidden_layers = 10
units_per_layer = 2
batch_size = 64

leaky_relu = 'LeakyReLU'
leaky_relu_collapses = 0
from keras.layers import LeakyReLU, Input
def create_leaky_network():
    model = Sequential()
    model.add(Input(shape = (1,)))
    for _ in range(num_hidden_layers - 1):
        model.add(Dense(2, activation=LeakyReLU(alpha=0.01)))
    model.add(Dense(1))  # Output layer
    return model

def perform_leaky_experiment():
    collapses = 0
    for i in range(num_simulations):
        if i % 100 == 0:
          print(f"{i}th sim")
        x_train, x_test = np.random.uniform(-7, 7, num_points), np.random.uniform(-7, 7, int(num_points*0.25))
        y_train, y_test = np.abs(x_train), np.abs(x_test)

        model = create_leaky_network()
        model.compile(optimizer=RMSprop(), loss='mse')
        model.fit(x_train, y_train, batch_size=batch_size, epochs=1, verbose=0)

        if len(np.unique(model.predict(x_test, verbose = 0).flatten())) == 1:
            collapses += 1

    return collapses

leaky_relu_collapses = perform_leaky_experiment()
print(f"Fraction of collapses with Leaky ReLU: {leaky_relu_collapses / num_simulations}")
print("number of collapse with leaky ReLU", leaky_relu_collapses)
```

```
0th sim
100th sim
200th sim
300th sim
400th sim
500th sim
600th sim
700th sim
800th sim
900th sim
Fraction of collapses with Leaky ReLU: 0.09
number of collapse with leaky ReLU 90
```

The fraction of collapse with Leaky ReLU drop to 9%, so it help in preventing dying neurons. This is because leaky ReLU allow a small, positive gradient for negative inputs. Unlike the ReLU activation function, which outputs zero for any negative input and therefore can cause neurons to become dead, Leaky ReLU allows for a small, non-zero gradient when the input is negative (with slow slope 0.01 when $z < 0$), thus reducing the risk of gradient vanishing for certain neurons and improving the network's overall ability to learn from the data.

Problem 4 - Batch Normalization, Dropout, MNIST

4.1.

Co-adaptation in neural networks refers to where neurons in a layer rely too much on the behavior of other neurons during training. neurons are supposed to learn features that are useful independently, but co-adaptation can lead to overfitting where neurons only work well in the presence of the activations of other neurons they have co-adapted with. Dropout is a technique to reduce co-adaptation by randomly "dropping out" some neuron outputs during training, forcing the network to learn more robust features.

Internal covariate shift refers to the change in the distribution of network activations due to the updating of weights during training. When the input distribution to a network layer changes, it can make the learning process less efficient because the layer has to relearn to adapt to the new distribution. Batch normalization addresses this problem by normalizing the inputs to each layer so that they have a mean of zero and a standard deviation of one.

4.2.

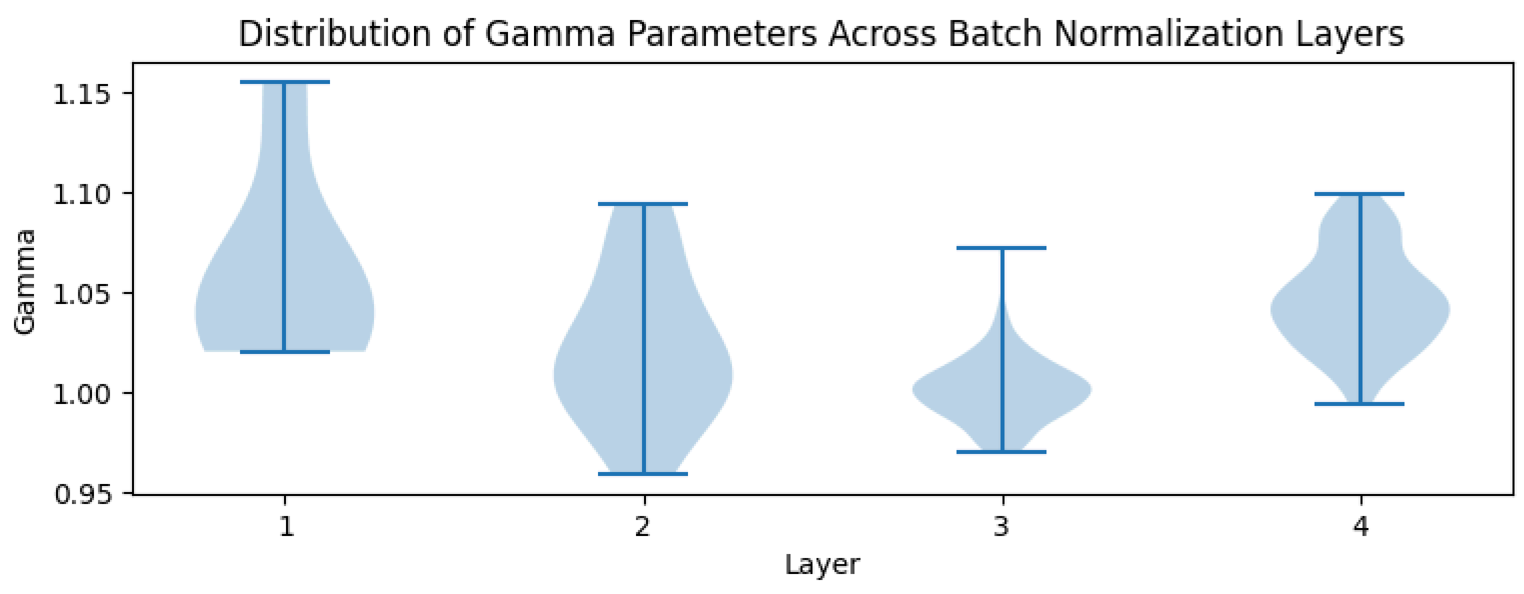
In [ ]:
```
from keras.datasets import mnist
from keras.layers import Dense, Conv2D, AveragePooling2D, Flatten, BatchNormalization, Normalization
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

X_train = np.expand_dims(X_train, axis=-1)
X_test = np.expand_dims(X_test, axis=-1)

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Define the model
st_model = Sequential([
    Normalization(),

    Conv2D(6, kernel_size=5, activation='tanh', input_shape=(28, 28, 1)),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),
    BatchNormalization(),

    Conv2D(16, kernel_size=5, activation='tanh'),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),
    BatchNormalization(),

    Flatten(),
    Dense(120, activation='tanh'),
    BatchNormalization(),
    Dense(84, activation='tanh'),
    BatchNormalization(),
    Dense(10, activation='softmax')
])

# Compile the model
```

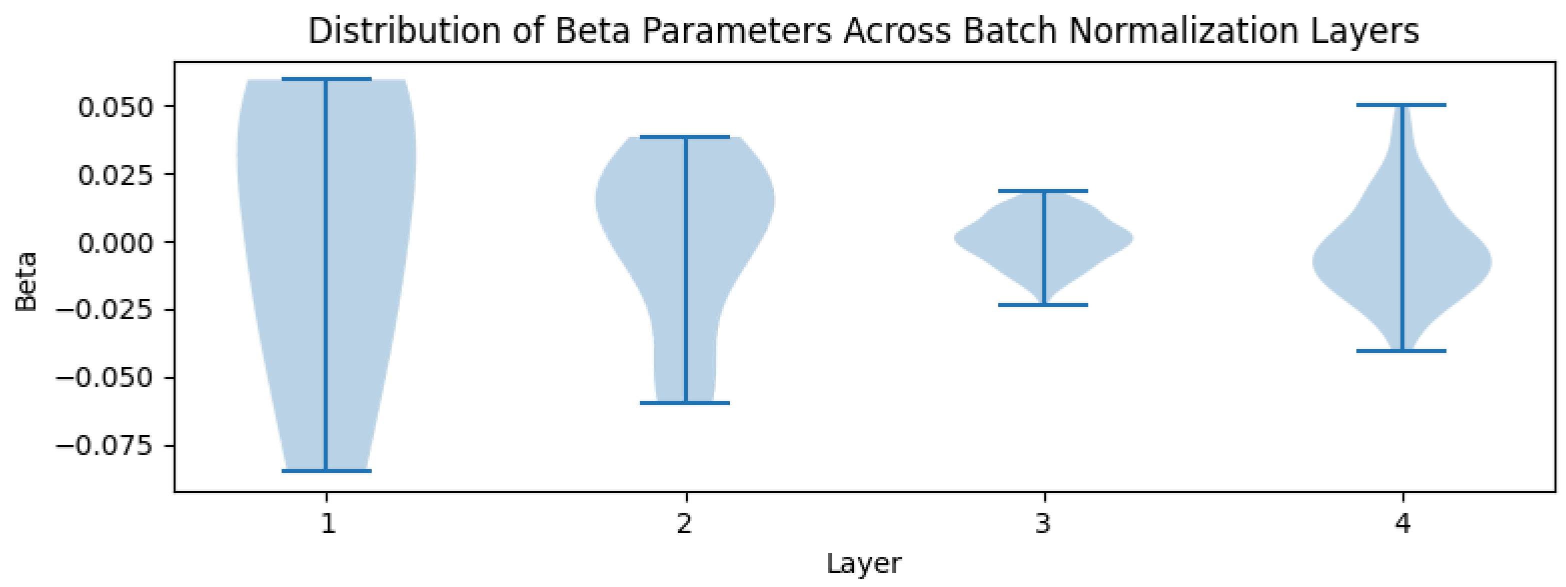
```
st_model.compile(loss='categorical_crossentropy', optimizer=SGD(), metrics=['accuracy'])

# Train
history = st_model.fit(X_train, y_train, \
                       validation_data=(X_test, y_test), epochs=10, batch_size=128)


batch_norm_parameters = []
for layer in st_model.layers:
    if "batch_normalization" in layer.name:
        gamma, beta, mean, variance = layer.get_weights()
        batch_norm_parameters.append((gamma, beta, mean, variance))
```

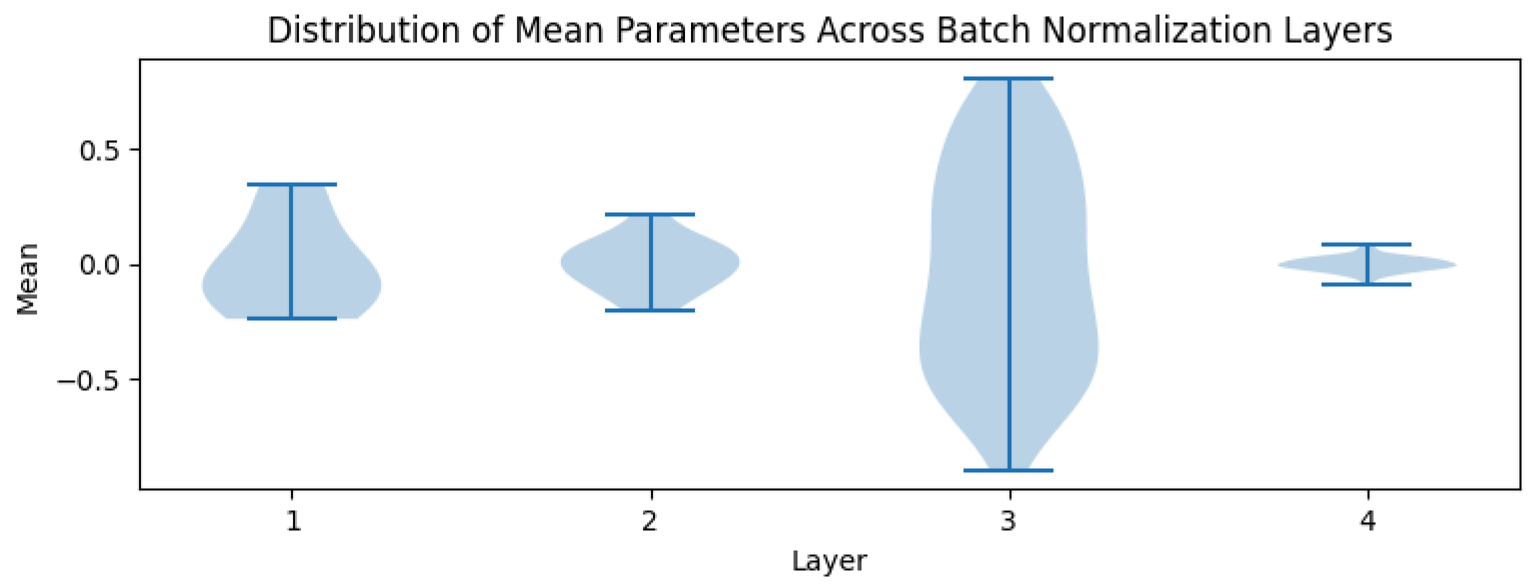
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
Epoch 1/10
469/469 [==============================] - 9s 7ms/step - loss: 0.4342 - accuracy: 0.8768 - val_loss: 0.2495 - val_acc
uracy: 0.9311
Epoch 2/10
469/469 [==============================] - 3s 6ms/step - loss: 0.2236 - accuracy: 0.9378 - val_loss: 0.1757 - val_acc
uracy: 0.9491
Epoch 3/10
469/469 [==============================] - 3s 6ms/step - loss: 0.1704 - accuracy: 0.9524 - val_loss: 0.1378 - val_acc
uracy: 0.9606
Epoch 4/10
469/469 [==============================] - 3s 6ms/step - loss: 0.1401 - accuracy: 0.9601 - val_loss: 0.1172 - val_acc
uracy: 0.9650
Epoch 5/10
469/469 [==============================] - 3s 6ms/step - loss: 0.1205 - accuracy: 0.9656 - val_loss: 0.1038 - val_acc
uracy: 0.9682
Epoch 6/10
469/469 [==============================] - 3s 6ms/step - loss: 0.1069 - accuracy: 0.9691 - val_loss: 0.0935 - val_acc
uracy: 0.9730
Epoch 7/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0951 - accuracy: 0.9724 - val_loss: 0.0878 - val_acc
uracy: 0.9732
Epoch 8/10
469/469 [==============================] - 3s 5ms/step - loss: 0.0871 - accuracy: 0.9752 - val_loss: 0.0794 - val_acc
uracy: 0.9763
Epoch 9/10
469/469 [==============================] - 3s 5ms/step - loss: 0.0802 - accuracy: 0.9765 - val_loss: 0.0752 - val_acc
uracy: 0.9770
Epoch 10/10
469/469 [==============================] - 3s 5ms/step - loss: 0.0748 - accuracy: 0.9783 - val_loss: 0.0718 - val_acc
uracy: 0.9782
```

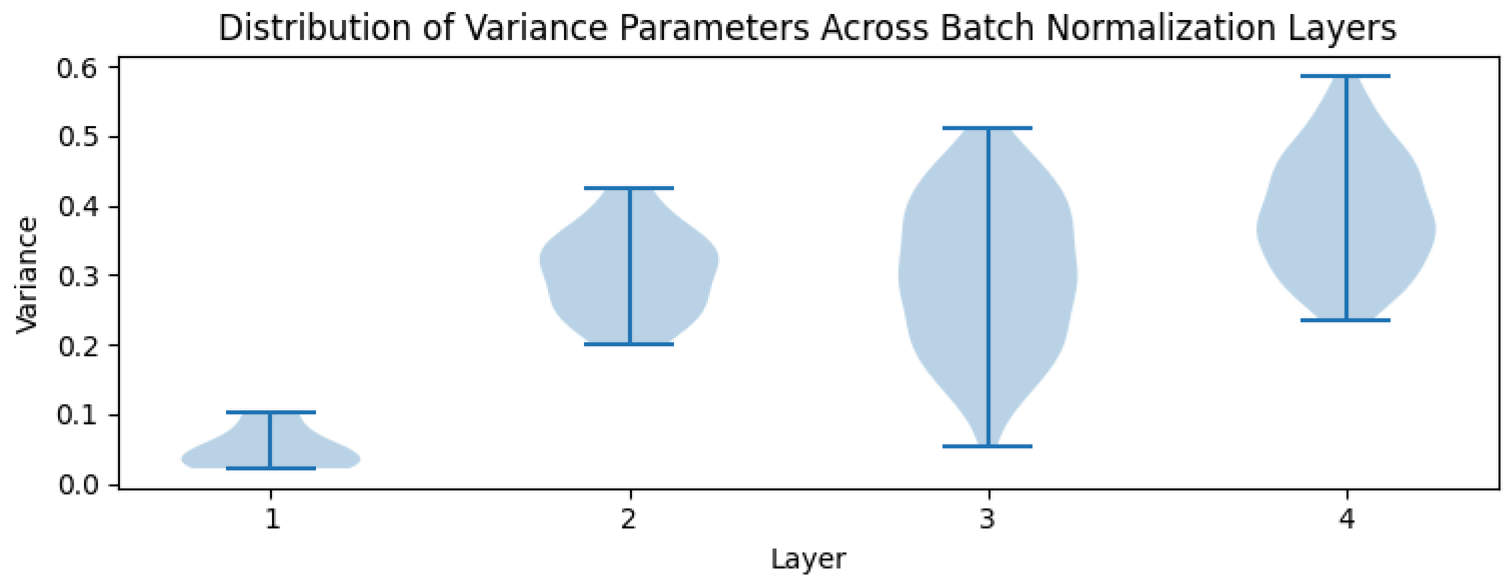
In [ ]:

```
fig, axes = plt.subplots(nrows=4, ncols=1, figsize=(8, 12))


gamma_values = [params[0] for params in batch_norm_parameters]
axes[0].violinplot(gamma_values)
axes[0].set_title('Distribution of Gamma Parameters Across Batch Normalization Layers')
axes[0].set_ylabel('Gamma')
axes[0].set_xlabel('Layer')
# show layer 1 to 4
axes[0].set_xticks(range(1, 5))

beta_values = [params[1] for params in batch_norm_parameters]
axes[1].violinplot(beta_values)
axes[1].set_title('Distribution of Beta Parameters Across Batch Normalization Layers')
axes[1].set_ylabel('Beta')
axes[1].set_xlabel('Layer')
axes[1].set_xticks(range(1, 5))

mean_values = [params[2] for params in batch_norm_parameters]
axes[2].violinplot(mean_values)
axes[2].set_title('Distribution of Mean Parameters Across Batch Normalization Layers')
axes[2].set_ylabel('Mean')
axes[2].set_xlabel('Layer')
axes[2].set_xticks(range(1, 5))

variance_values = [params[3] for params in batch_norm_parameters]
axes[3].violinplot(variance_values)
axes[3].set_title('Distribution of Variance Parameters Across Batch Normalization Layers')
axes[3].set_ylabel('Variance')
axes[3].set_xlabel('Layer')
axes[3].set_xticks(range(1, 5))

fig.tight_layout()
plt.show()
```









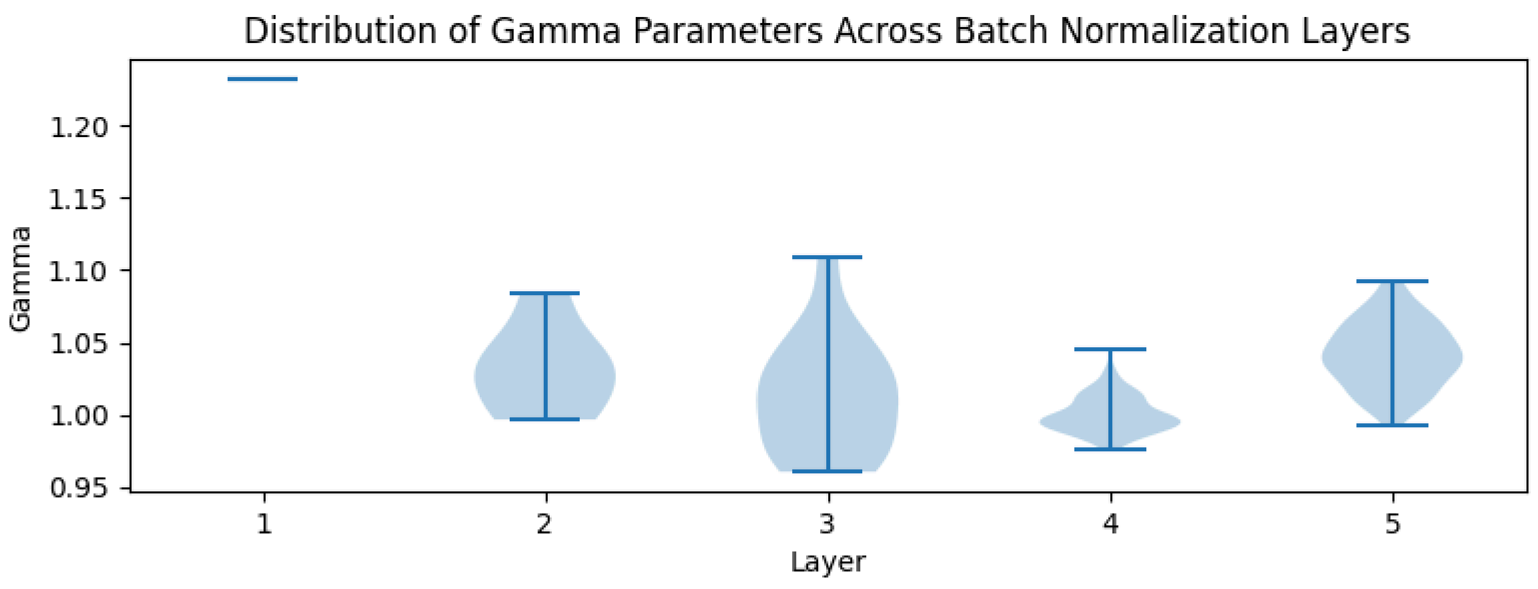
4.3.

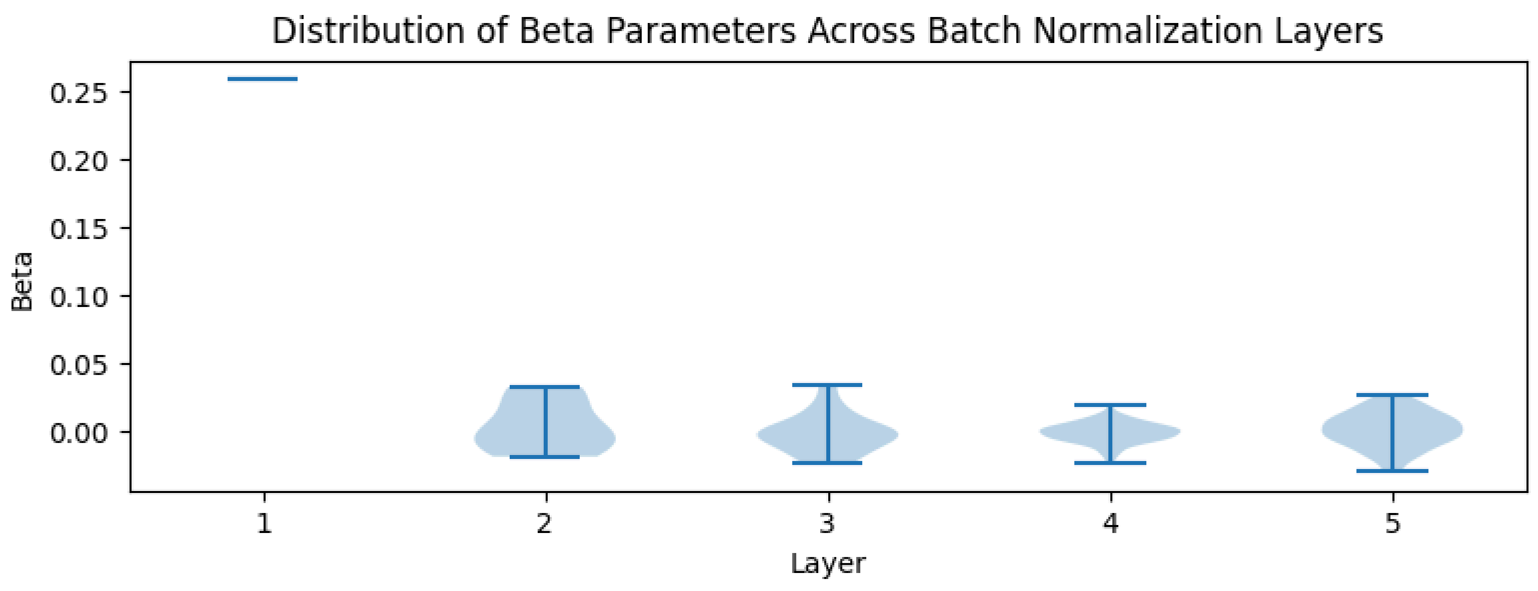
In [ ]:
```
batch_model = Sequential([
    BatchNormalization(),

    Conv2D(6, kernel_size=5, activation='tanh', input_shape=(28, 28, 1)),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),
    BatchNormalization(),

    Conv2D(16, kernel_size=5, activation='tanh'),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),
    BatchNormalization(),

    Flatten(),
    Dense(120, activation='tanh'),
    BatchNormalization(),
    Dense(84, activation='tanh'),
    BatchNormalization(),
    Dense(10, activation='softmax')
])

# Compile the model
batch_model.compile(loss='categorical_crossentropy', optimizer=SGD(), metrics=['accuracy'])

# Train
history_batch = batch_model.fit(X_train, y_train, \
                                validation_data=(X_test, y_test), epochs=10, batch_size=128)

# Batch Normalization layer's gamma and beta
batch_norm_parameters_b = []
for layer in batch_model.layers:
    if "batch_normalization" in layer.name:
        gamma, beta, mean, variance = layer.get_weights()
        batch_norm_parameters_b.append((gamma, beta, mean, variance))
```

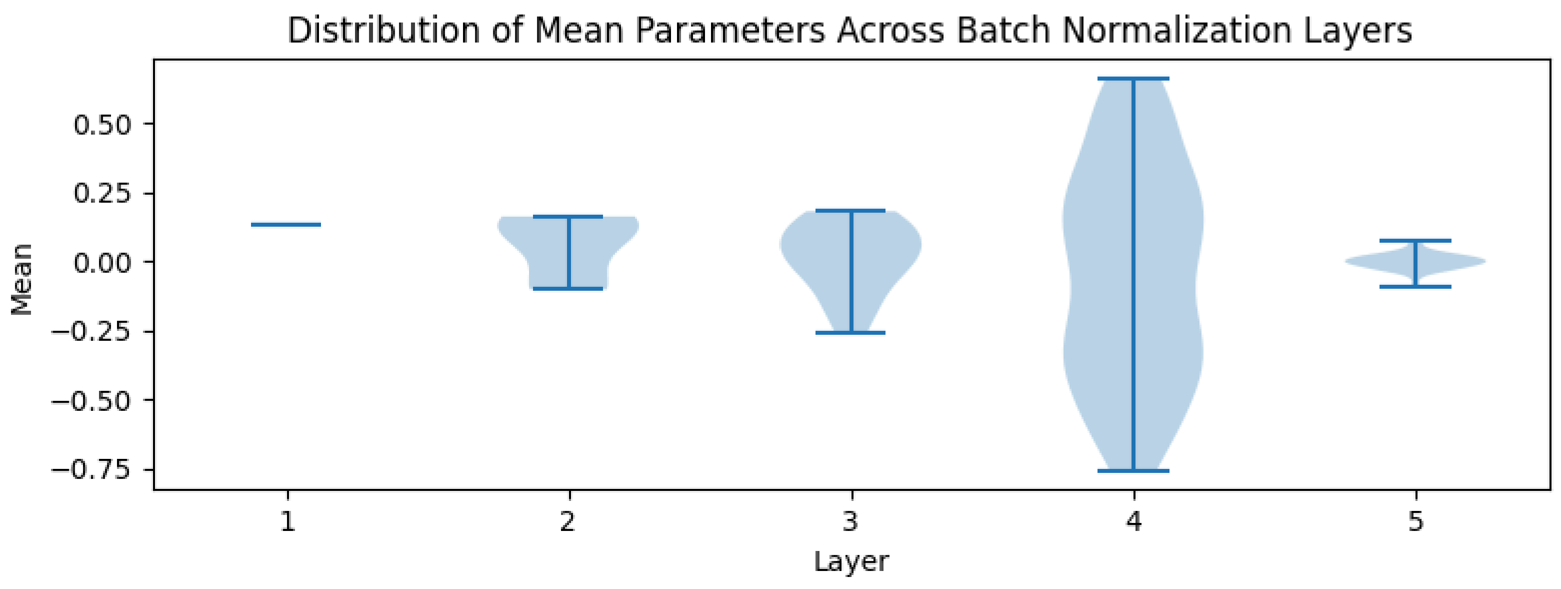
```
Epoch 1/10
469/469 [==============================] - 6s 9ms/step - loss: 0.3978 - accuracy: 0.8875 - val_loss: 0.2234 - val_acc
uracy: 0.9388
Epoch 2/10
469/469 [==============================] - 3s 7ms/step - loss: 0.1852 - accuracy: 0.9489 - val_loss: 0.1354 - val_acc
uracy: 0.9617
Epoch 3/10
469/469 [==============================] - 3s 7ms/step - loss: 0.1289 - accuracy: 0.9641 - val_loss: 0.1030 - val_acc
uracy: 0.9695
Epoch 4/10
469/469 [==============================] - 3s 6ms/step - loss: 0.1012 - accuracy: 0.9715 - val_loss: 0.0833 - val_acc
uracy: 0.9768
Epoch 5/10
469/469 [==============================] - 3s 7ms/step - loss: 0.0868 - accuracy: 0.9754 - val_loss: 0.0742 - val_acc
uracy: 0.9786
Epoch 6/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0757 - accuracy: 0.9788 - val_loss: 0.0663 - val_acc
uracy: 0.9815
Epoch 7/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0684 - accuracy: 0.9806 - val_loss: 0.0633 - val_acc
uracy: 0.9823
Epoch 8/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0621 - accuracy: 0.9826 - val_loss: 0.0559 - val_acc
uracy: 0.9839
Epoch 9/10
469/469 [==============================] - 3s 7ms/step - loss: 0.0571 - accuracy: 0.9839 - val_loss: 0.0541 - val_acc
uracy: 0.9834
Epoch 10/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0531 - accuracy: 0.9848 - val_loss: 0.0506 - val_acc
uracy: 0.9858
```

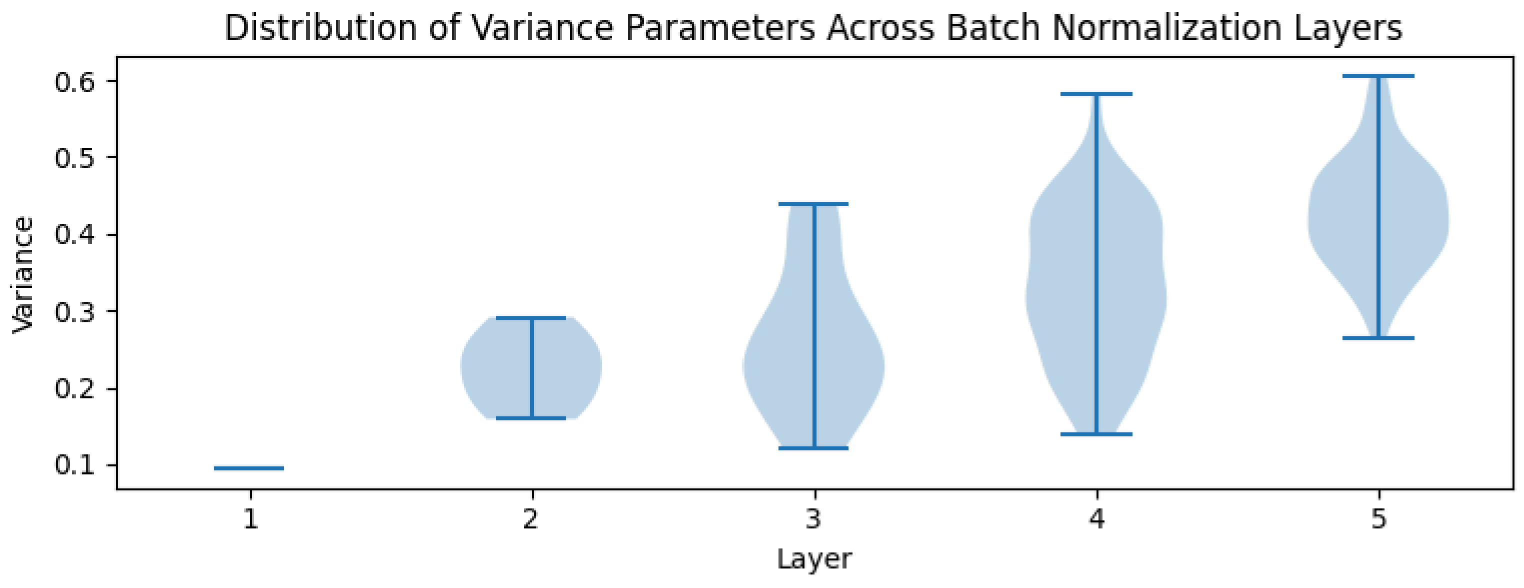
In [ ]:

```
fig, axes = plt.subplots(nrows=4, ncols=1, figsize=(8, 12))


gamma_values = [params[0] for params in batch_norm_parameters_b]
axes[0].violinplot(gamma_values)
axes[0].set_title('Distribution of Gamma Parameters Across Batch Normalization Layers')
axes[0].set_ylabel('Gamma')
axes[0].set_xlabel('Layer')

beta_values = [params[1] for params in batch_norm_parameters_b]
axes[1].violinplot(beta_values)
axes[1].set_title('Distribution of Beta Parameters Across Batch Normalization Layers')
axes[1].set_ylabel('Beta')
axes[1].set_xlabel('Layer')

mean_values = [params[2] for params in batch_norm_parameters_b]
axes[2].violinplot(mean_values)
axes[2].set_title('Distribution of Mean Parameters Across Batch Normalization Layers')
axes[2].set_ylabel('Mean')
axes[2].set_xlabel('Layer')

variance_values = [params[3] for params in batch_norm_parameters_b]
axes[3].violinplot(variance_values)
axes[3].set_title('Distribution of Variance Parameters Across Batch Normalization Layers')
axes[3].set_ylabel('Variance')
axes[3].set_xlabel('Layer')

fig.tight_layout()
plt.show()
```

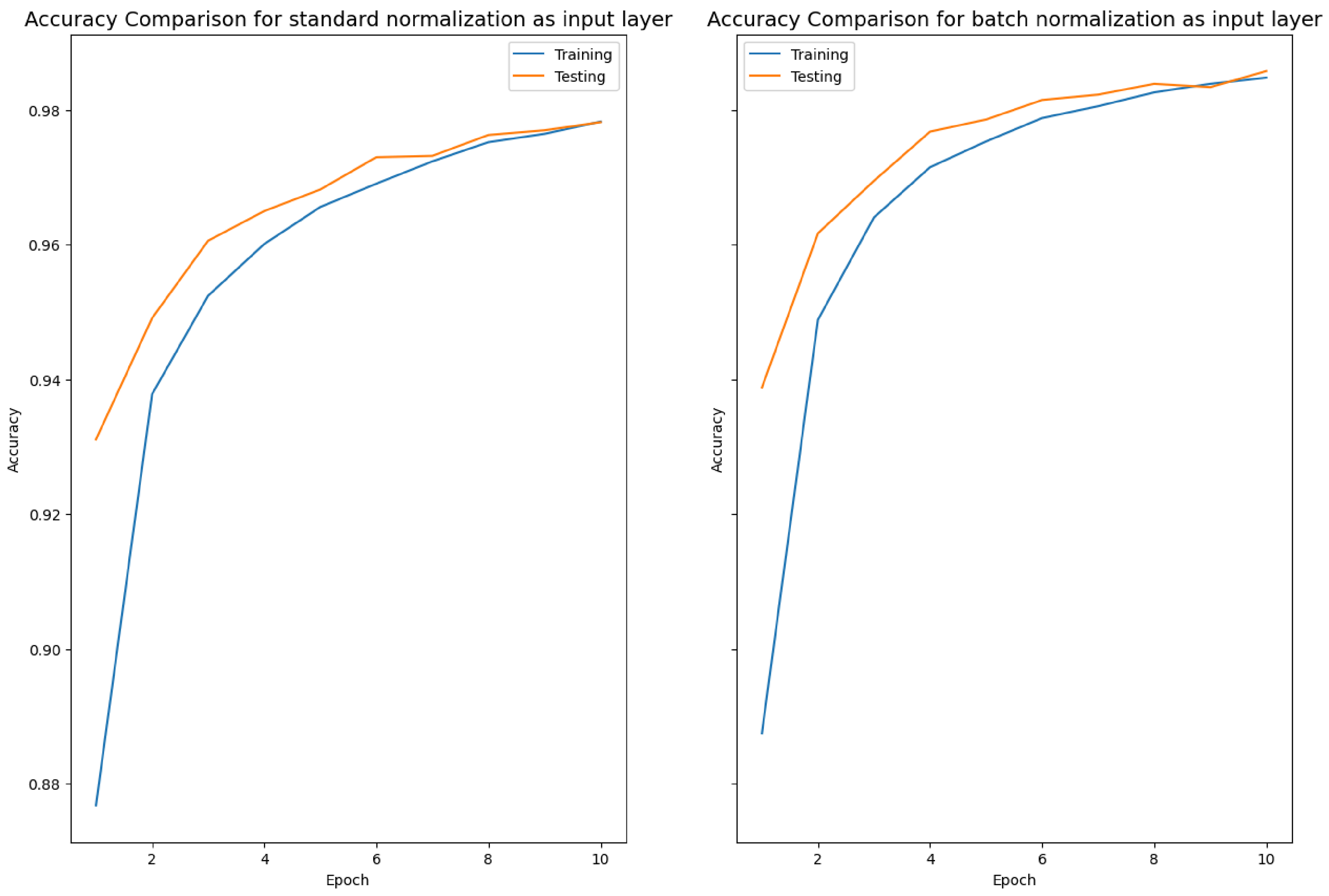




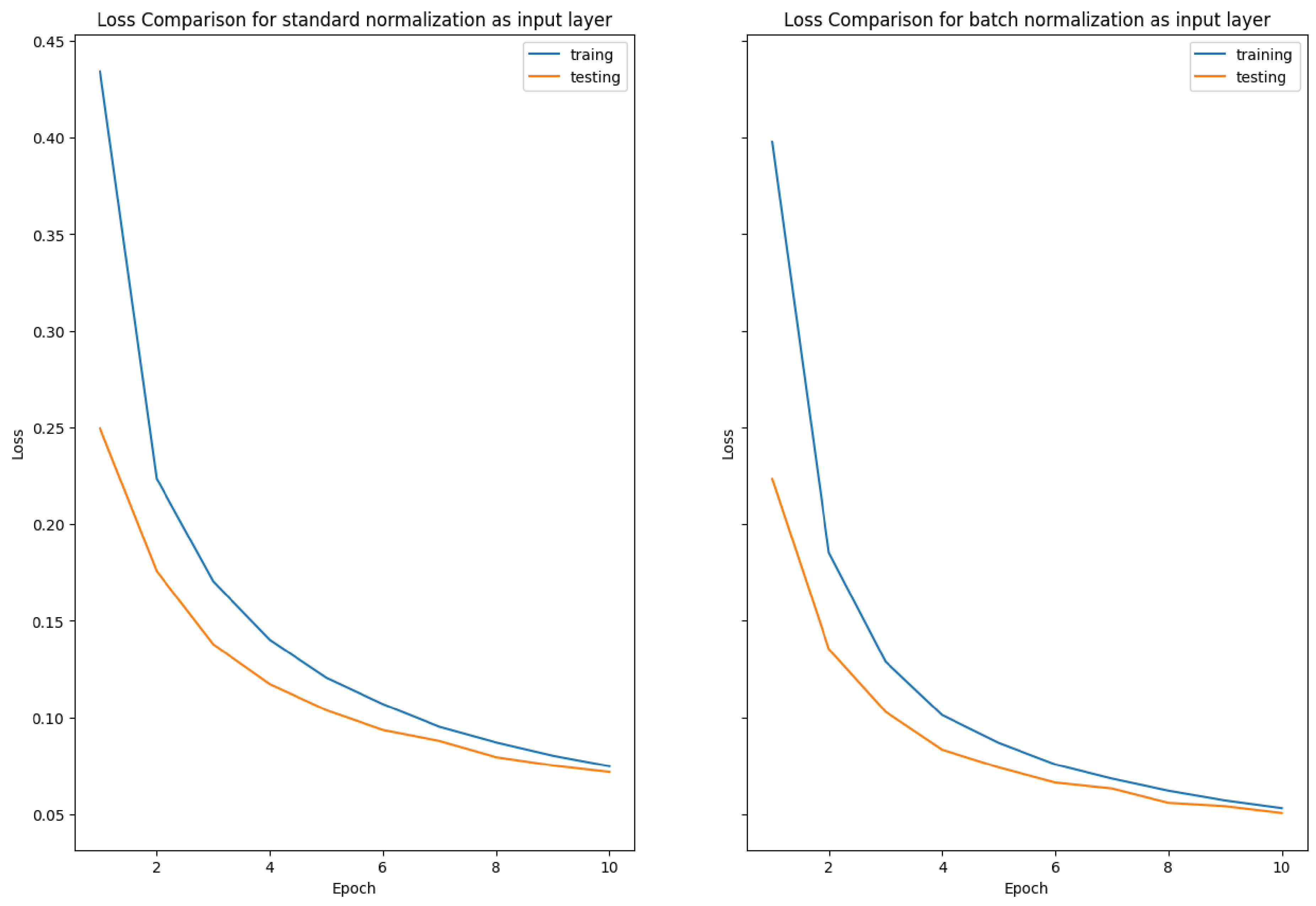




In [ ]:
```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(15, 10), sharey=True, sharex=True)

# Train accuracy for both models
plt.sca(axs[0])
plt.plot(np.arange(1,11), history.history['accuracy'], label='Training')
plt.plot(np.arange(1,11), history.history['val_accuracy'], label='Testing')
plt.title('Accuracy Comparison for standard normalization as input layer', size = 14)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.sca(axs[1])
plt.plot(np.arange(1,11), history_batch.history['accuracy'], label='Training')
plt.plot(np.arange(1,11), history_batch.history['val_accuracy'], label='Testing')
plt.title('Accuracy Comparison for batch normalization as input layer', size = 14)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



```
In [ ]: fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(15, 10), sharey=True, sharex=True)

        # Train accuracy for both models
        plt.sca(axs[0])
        plt.plot(np.arange(1,11), history.history['loss'], label='traing')
        plt.plot(np.arange(1,11), history.history['val_loss'], label='testing')
        plt.title('Loss Comparison for standard normalization as input layer')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()


        plt.sca(axs[1])
        plt.plot(np.arange(1,11), history_batch.history['loss'], label='training')
        plt.plot(np.arange(1,11), history_batch.history['val_loss'], label='testing')
        plt.title('Loss Comparison for batch normalization as input layer')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()
```



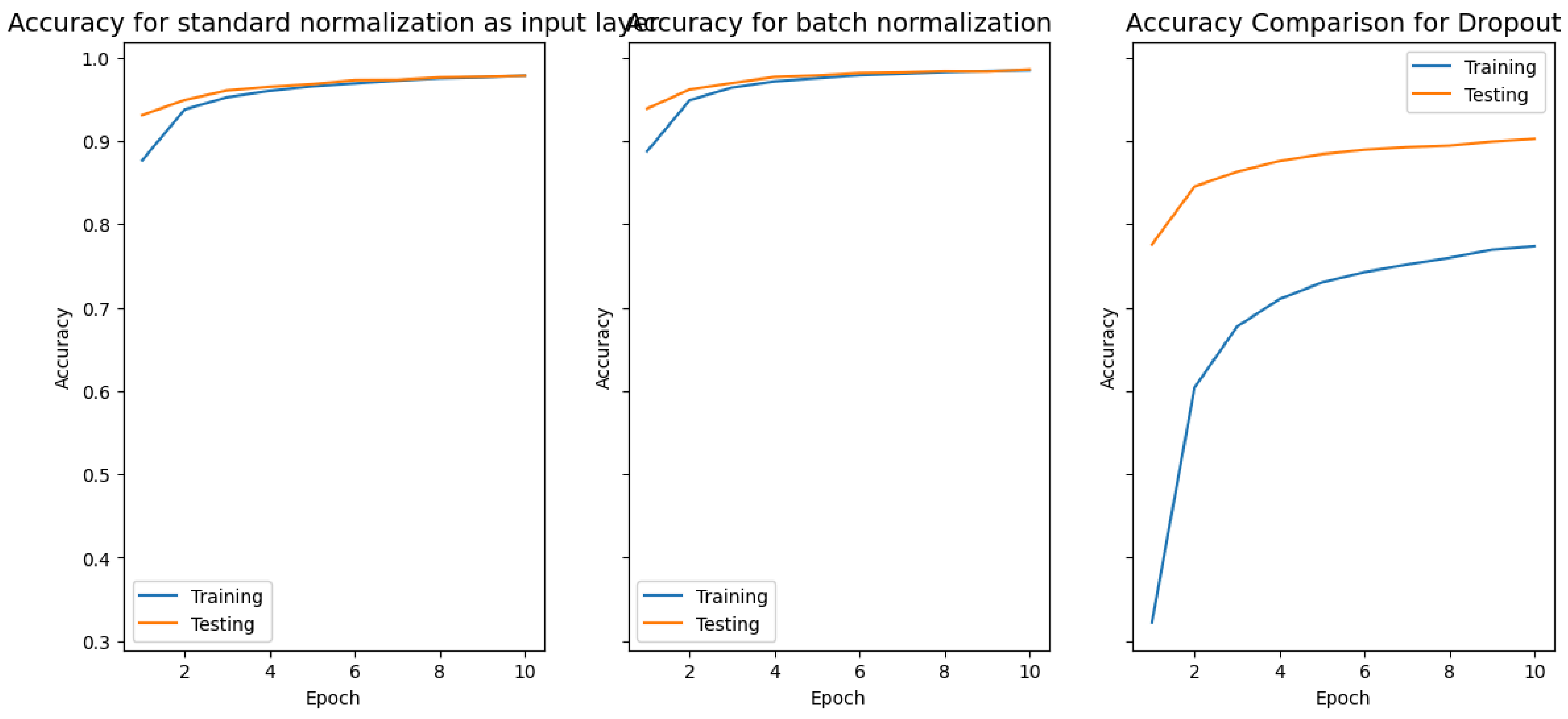
Batch normalization for the input layer improve performance. For the accuracy when using batch normalization as input layer, we see an increase in both training and testing accuracy compared to using standard normalization. And we see a decrease in loss for training and testing for batch normalization compared to using standard normalization. In general, at the input layer, batch normalization does better job in helping in stabilizing the learning process.

4.4.

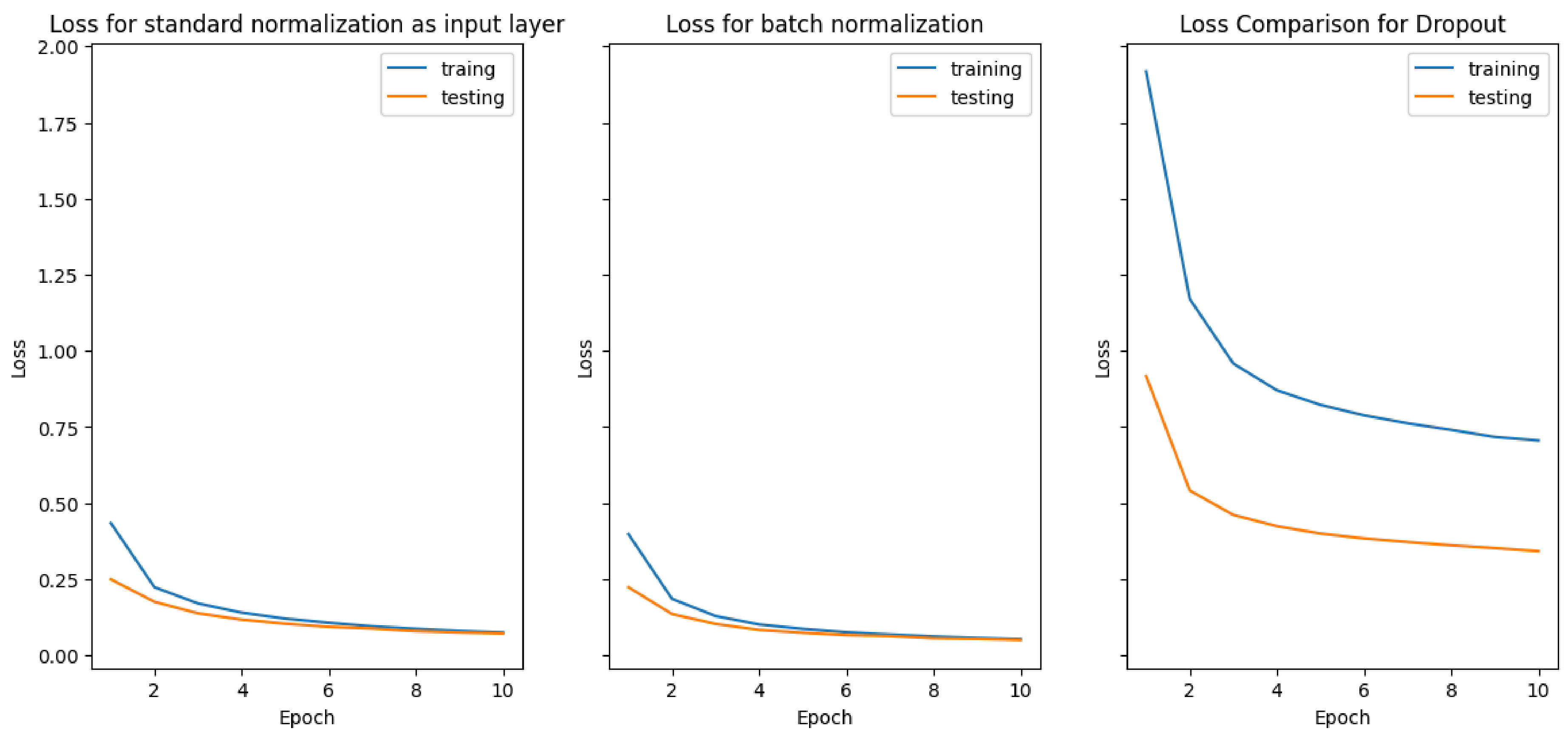
In [ ]:
```
from keras.layers import Dense, Conv2D, AveragePooling2D, Flatten, Dropout
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt

dropout_model = Sequential([
    Dropout(0.2),

    Conv2D(6, kernel_size=5, activation='tanh', input_shape=(28, 28, 1)),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),
    Dropout(0.5),

    Conv2D(16, kernel_size=5, activation='tanh'),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),
    Dropout(0.5),

    Flatten(),
    Dense(120, activation='tanh'),
    Dropout(0.5),
    Dense(84, activation='tanh'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])


# Compile the model
dropout_model.compile(loss='categorical_crossentropy', optimizer=SGD(), metrics=['accuracy'])

# Train
history_dropout = dropout_model.fit(X_train, y_train, \
                                    validation_data=(X_test, y_test), epochs=10, batch_size=128)
```

```
Epoch 1/10
469/469 [==============================] - 4s 5ms/step - loss: 1.9182 - accuracy: 0.3221 - val_loss: 0.9171 - val_acc
uracy: 0.7757
Epoch 2/10
469/469 [==============================] - 2s 5ms/step - loss: 1.1709 - accuracy: 0.6037 - val_loss: 0.5414 - val_acc
uracy: 0.8453
Epoch 3/10
469/469 [==============================] - 2s 5ms/step - loss: 0.9585 - accuracy: 0.6771 - val_loss: 0.4611 - val_acc
uracy: 0.8630
Epoch 4/10
469/469 [==============================] - 2s 5ms/step - loss: 0.8705 - accuracy: 0.7101 - val_loss: 0.4242 - val_acc
uracy: 0.8759
Epoch 5/10
469/469 [==============================] - 2s 5ms/step - loss: 0.8226 - accuracy: 0.7301 - val_loss: 0.3999 - val_acc
uracy: 0.8841
Epoch 6/10
469/469 [==============================] - 2s 5ms/step - loss: 0.7883 - accuracy: 0.7426 - val_loss: 0.3840 - val_acc
uracy: 0.8894
Epoch 7/10
469/469 [==============================] - 2s 5ms/step - loss: 0.7622 - accuracy: 0.7516 - val_loss: 0.3720 - val_acc
uracy: 0.8925
Epoch 8/10
469/469 [==============================] - 2s 4ms/step - loss: 0.7405 - accuracy: 0.7597 - val_loss: 0.3614 - val_acc
uracy: 0.8944
Epoch 9/10
469/469 [==============================] - 2s 5ms/step - loss: 0.7174 - accuracy: 0.7695 - val_loss: 0.3522 - val_acc
uracy: 0.8992
Epoch 10/10
469/469 [==============================] - 2s 5ms/step - loss: 0.7057 - accuracy: 0.7737 - val_loss: 0.3423 - val_acc
uracy: 0.9026
```

In [ ]:
```python
fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(15, 6), sharey=True, sharex=True)

# Train accuracy for both models
plt.sca(axs[0])
plt.plot(np.arange(1,11), history.history['accuracy'], label='Training')
plt.plot(np.arange(1,11), history.history['val_accuracy'], label='Testing')
plt.title('Accuracy for standard normalization', size = 14)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.sca(axs[1])
plt.plot(np.arange(1,11), history_batch.history['accuracy'], label='Training')
plt.plot(np.arange(1,11), history_batch.history['val_accuracy'], label='Testing')
plt.title('Accuracy for batch normalization', size = 14)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.sca(axs[2])
plt.plot(np.arange(1,11), history_dropout.history['accuracy'], label='Training')
plt.plot(np.arange(1,11), history_dropout.history['val_accuracy'], label='Testing')
plt.title('Accuracy Comparison for Dropout', size = 14)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



In [ ]:
```python
fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(14, 6), sharey=True, sharex=True)

# loss for both models
plt.sca(axs[0])
plt.plot(np.arange(1,11), history.history['loss'], label='traing')
plt.plot(np.arange(1,11), history.history['val_loss'], label='testing')
plt.title('Loss for standard normalization as input layer')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()


plt.sca(axs[1])
plt.plot(np.arange(1,11), history_batch.history['loss'], label='training')
plt.plot(np.arange(1,11), history_batch.history['val_loss'], label='testing')
plt.title('Loss for batch normalization')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.sca(axs[2])
plt.plot(np.arange(1,11), history_dropout.history['loss'], label='training')
plt.plot(np.arange(1,11), history_dropout.history['val_loss'], label='testing')
plt.title('Loss Comparison for Dropout')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



The accuracy and loss graphs for dropout show a larger gap between the training and testing accuracy compared to the other two cases, also the absolute effect for accuracy and loss are worse for dropout compared to the other two normalization. It could be dropout is preventing overfitting to some extent as the model is not as accurate on the training data as it could be without dropout, which is good for generalization. But in general, dropout, by its own, in this case, did not improve performance in terms of testing accuracy or loss compared to batch normalization.

4.5.

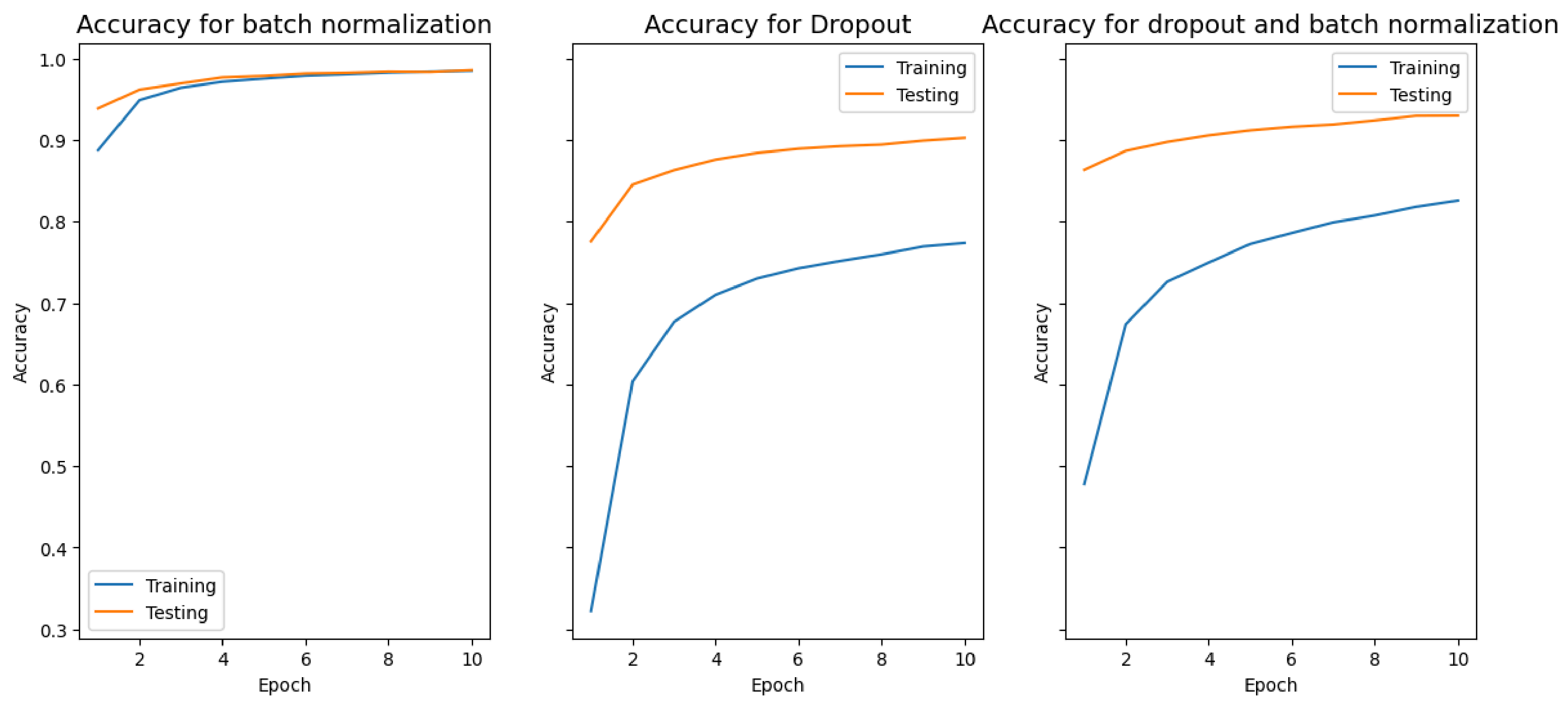
```
In [ ]: from keras.layers import Dense, Conv2D, AveragePooling2D, Flatten, Dropout
        from keras.models import Sequential
        from keras.optimizers import SGD
        from keras.utils import to_categorical
        import numpy as np
        import matplotlib.pyplot as plt

        combine_model = Sequential([
            BatchNormalization(),
            Dropout(0.2),

            Conv2D(6, kernel_size=5, activation='tanh', input_shape=(28, 28, 1)),
            AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),
            BatchNormalization(),
            Dropout(0.5),

            Conv2D(16, kernel_size=5, activation='tanh'),
            AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),
            BatchNormalization(),
            Dropout(0.5),

            Flatten(),
            Dense(120, activation='tanh'),
            BatchNormalization(),
            Dropout(0.5),
            Dense(84, activation='tanh'),
            BatchNormalization(),
            Dropout(0.5),
            Dense(10, activation='softmax')
        ])
```

```
# Compile the model
combine_model.compile(loss='categorical_crossentropy', optimizer=SGD(), metrics=['accuracy'])

# Train
history_combine = combine_model.fit(X_train, y_train, \
                                    validation_data=(X_test, y_test), \
                                    epochs=10, batch_size=128)
```

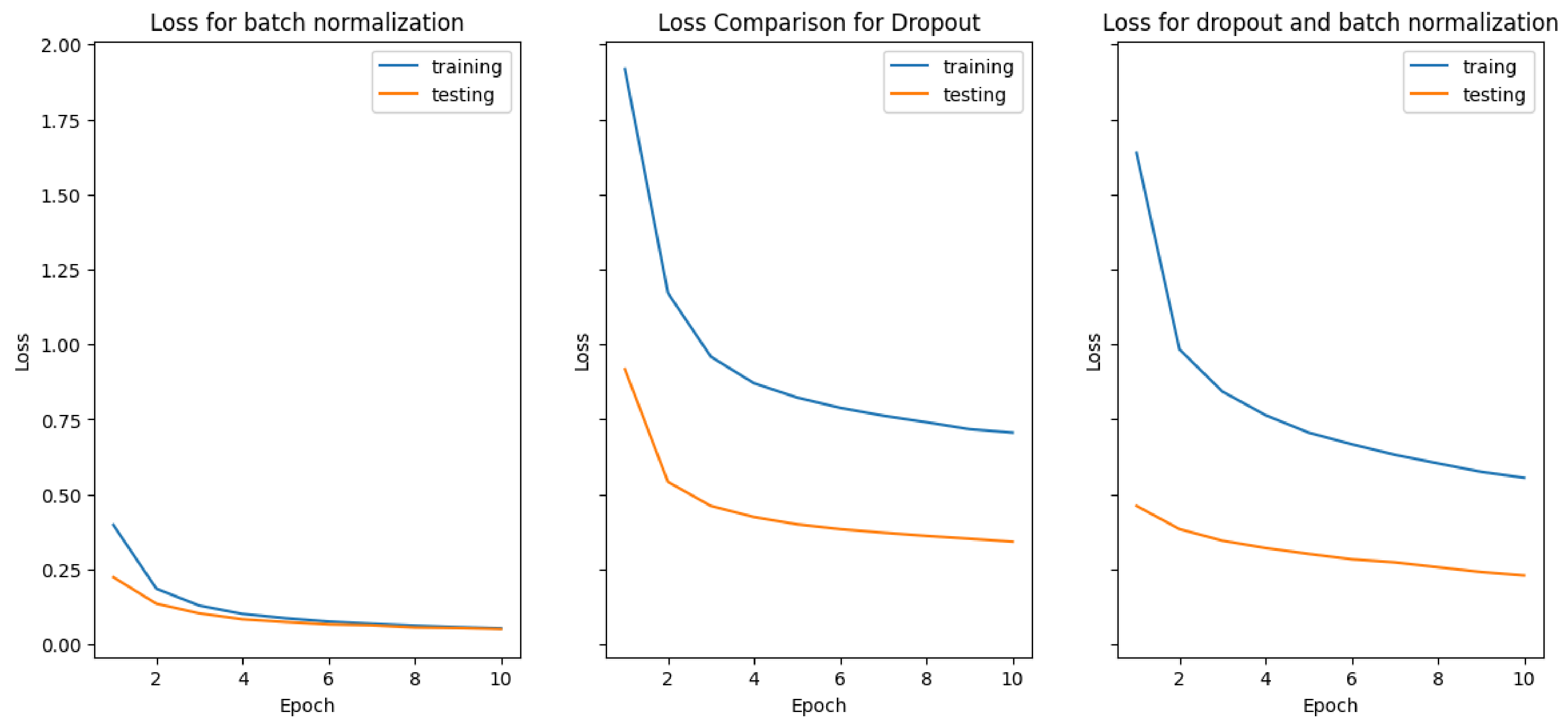
```
Epoch 1/10
469/469 [==============================] - 6s 8ms/step - loss: 1.6390 - accuracy: 0.4779 - val_loss: 0.4617 - val_acc
uracy: 0.8634
Epoch 2/10
469/469 [==============================] - 4s 9ms/step - loss: 0.9831 - accuracy: 0.6740 - val_loss: 0.3840 - val_acc
uracy: 0.8867
Epoch 3/10
469/469 [==============================] - 3s 7ms/step - loss: 0.8424 - accuracy: 0.7261 - val_loss: 0.3453 - val_acc
uracy: 0.8976
Epoch 4/10
469/469 [==============================] - 3s 7ms/step - loss: 0.7638 - accuracy: 0.7497 - val_loss: 0.3210 - val_acc
uracy: 0.9057
Epoch 5/10
469/469 [==============================] - 3s 7ms/step - loss: 0.7057 - accuracy: 0.7724 - val_loss: 0.3014 - val_acc
uracy: 0.9119
Epoch 6/10
469/469 [==============================] - 4s 8ms/step - loss: 0.6669 - accuracy: 0.7859 - val_loss: 0.2834 - val_acc
uracy: 0.9162
Epoch 7/10
469/469 [==============================] - 3s 7ms/step - loss: 0.6319 - accuracy: 0.7990 - val_loss: 0.2731 - val_acc
uracy: 0.9190
Epoch 8/10
469/469 [==============================] - 3s 7ms/step - loss: 0.6032 - accuracy: 0.8075 - val_loss: 0.2570 - val_acc
uracy: 0.9240
Epoch 9/10
469/469 [==============================] - 3s 7ms/step - loss: 0.5752 - accuracy: 0.8181 - val_loss: 0.2409 - val_acc
uracy: 0.9297
Epoch 10/10
469/469 [==============================] - 4s 8ms/step - loss: 0.5554 - accuracy: 0.8256 - val_loss: 0.2295 - val_acc
uracy: 0.9299
```

In [ ]:

```
fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(14, 6), sharey=True, sharex=True)

# Train accuracy for both models

plt.sca(axs[0])
plt.plot(np.arange(1,11), history_batch.history['accuracy'], label='Training')
plt.plot(np.arange(1,11), history_batch.history['val_accuracy'], label='Testing')
plt.title('Accuracy for batch normalization', size = 14)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.sca(axs[1])
plt.plot(np.arange(1,11), history_dropout.history['accuracy'], label='Training')
plt.plot(np.arange(1,11), history_dropout.history['val_accuracy'], label='Testing')
plt.title('Accuracy for Dropout', size = 14)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.sca(axs[2])
plt.plot(np.arange(1,11), history_combine.history['accuracy'], label='Training')
plt.plot(np.arange(1,11), history_combine.history['val_accuracy'], label='Testing')
plt.title('Accuracy for dropout and batch normalization', size = 14)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()


plt.show()
```



In [ ]:
```
fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(14, 6), sharey=True, sharex=True)

# loss for both models

plt.sca(axs[0])
plt.plot(np.arange(1,11), history_batch.history['loss'], label='training')
plt.plot(np.arange(1,11), history_batch.history['val_loss'], label='testing')
plt.title('Loss for batch normalization')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.sca(axs[1])
plt.plot(np.arange(1,11), history_dropout.history['loss'], label='training')
plt.plot(np.arange(1,11), history_dropout.history['val_loss'], label='testing')
plt.title('Loss Comparison for Dropout')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.sca(axs[2])
plt.plot(np.arange(1,11), history_combine.history['loss'], label='traing')
plt.plot(np.arange(1,11), history_combine.history['val_loss'], label='testing')
plt.title('Loss for dropout and batch normalization')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



Combining both techniques, the graph indicates a slight discrepancy between training and testing accuracy, with training accuracy being higher. However, the testing accuracy is not as high as with batch normalization alone and the testing loss is higher compared to batch normalization alone. Batch normalization seems to have the best performance in terms of both accuracy and loss. Dropout improve generalization to some extent but does not achieve the same level of performance. When combined, batch normalization and dropout do show an additive benefit than just dropout but, in fact, might be less effective than batch normalization on its own.

Problem 5: Learning Rate, Batch Size, FashionMNIST

5.1

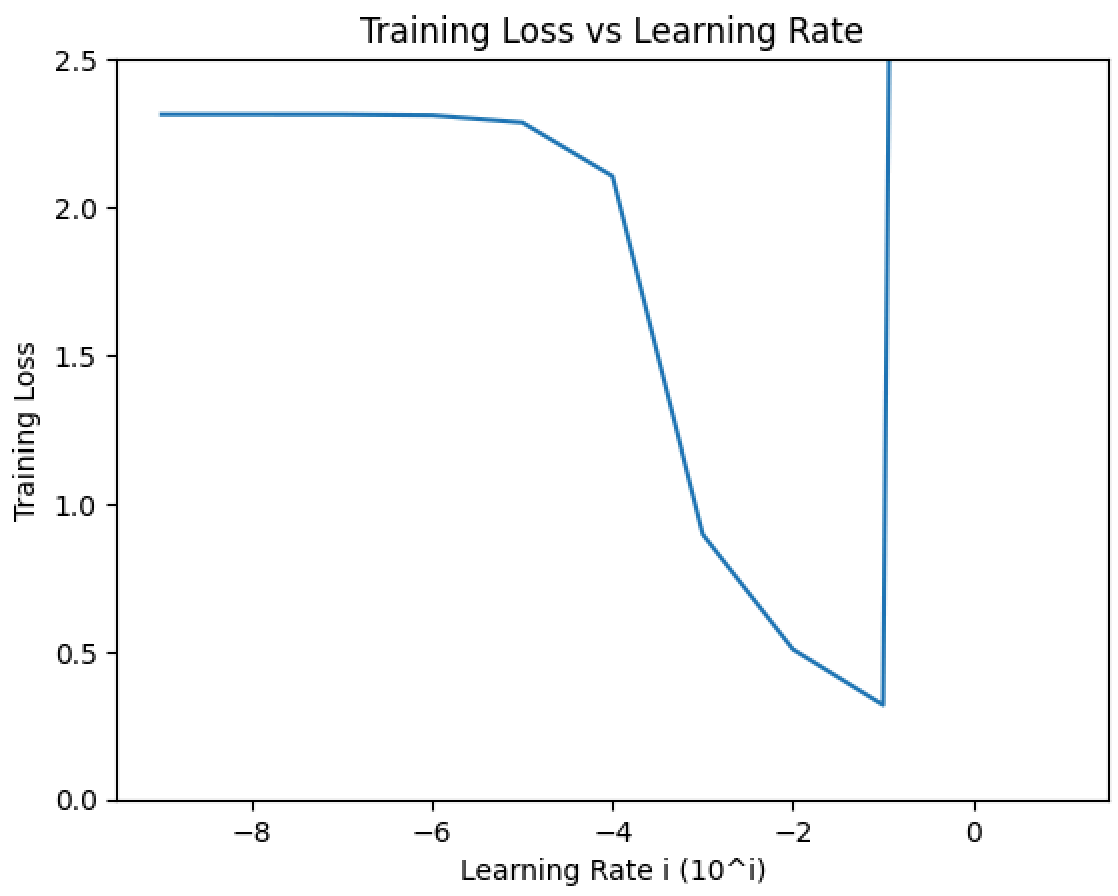
In [ ]:
```python
from keras.layers import Dense, Conv2D, AveragePooling2D, Flatten, Dropout
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

(X_train_fashion, y_train_fashion), (X_test_fashion, y_test_fashion) = tf.keras.datasets.fashion_mnist.load_data()
X_train_fashion = X_train_fashion.astype('float32') / 255.0
X_test_fashion = X_test_fashion.astype('float32') / 255.0

X_train_fashion = np.expand_dims(X_train_fashion, axis=-1)
X_test_fashion = np.expand_dims(X_test_fashion, axis=-1)

y_train_fashion = to_categorical(y_train_fashion)
y_test_fashion = to_categorical(y_test_fashion)

lr_model = Sequential([
    Conv2D(6, kernel_size=5, strides=(1, 1), activation='tanh', input_shape=(28, 28, 1)),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),

    Conv2D(16, kernel_size=5, strides=(1, 1), activation='tanh'),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),

    Flatten(),
    Dense(120, activation='tanh'),

    Dense(84, activation='tanh'),

    Dense(10, activation='softmax')
])


# Compile the model
lr_model.compile(loss='categorical_crossentropy', optimizer=SGD(), metrics=['accuracy'])

# Train
lr_list = [10**i for i in range(-9, 2)]
lr_loss = []
for i in lr_list:
    lr_model.optimizer.lr = i
    history_lr = lr_model.fit(X_train_fashion, y_train_fashion, \
                             validation_data=(X_test_fashion, y_test_fashion), \
                             epochs=5, batch_size=64, verbose = 0)
    lr_loss.append(history_lr.history['loss'][-1])
    print(f"{i} finished")
```

```
1e-09 finished
1e-08 finished
1e-07 finished
1e-06 finished
1e-05 finished
0.0001 finished
0.001 finished
0.01 finished
0.1 finished
1 finished
10 finished
```

In [ ]:
```python
print(lr_loss)
```

```
[2.3135311603546143, 2.3135263919830322, 2.3133230209350586, 2.310471296310425, 2.286162853240967, 2.104422569274902
3, 0.8966323733329773, 0.5077552795410156, 0.3196070194244385, 33.91877365112305, 384.50726318359375]
```

In [ ]:
```python
# plot the training loss against learning rate
plt.plot(range(-9,2), lr_loss)
plt.xlabel('Learning Rate i (10^i)')
plt.ylabel('Training Loss')
plt.ylim(0,2.5)
plt.title('Training Loss vs Learning Rate')
plt.show()
```



By examing the graph, we find where the learning rate is too small for the network to actually learn anything and where the learning rate is far too high for our model to learn. Therefore, we have:

$$lr_{min} = 10^{-1}, lr_{max} = 10^{-5}$$

5.2

In [ ]:
~~~
# Reference: https://github.com/bckenstler/CLR

from tensorflow.keras.callbacks import *
from tensorflow.keras import backend as K
import numpy as np

class CyclicLR(Callback):
    """This callback implements a cyclical learning rate policy (CLR).
    The method cycles the learning rate between two boundaries with
    some constant frequency, as detailed in this paper (https://arxiv.org/abs/1506.01186).
    The amplitude of the cycle can be scaled on a per-iteration or
    per-cycle basis.
    This class has three built-in policies, as put forth in the paper.
    "triangular":
        A basic triangular cycle w/ no amplitude scaling.
    "triangular2":
        A basic triangular cycle that scales initial amplitude by half each cycle.
    "exp_range":
        A cycle that scales initial amplitude by gamma**(cycle iterations) at each
        cycle iteration.
    For more detail, please see paper.

    # Example
        ```python
            clr = CyclicLR(base_lr=0.001, max_lr=0.006,
                                step_size=2000., mode='triangular')
            model.fit(X_train, Y_train, callbacks=[clr])
        ~~~

    Class also supports custom scaling functions:
        ```python
            clr_fn = lambda x: 0.5*(1+np.sin(x*np.pi/2.))
            clr = CyclicLR(base_lr=0.001, max_lr=0.006,
                                step_size=2000., scale_fn=clr_fn,
                                scale_mode='cycle')
            model.fit(X_train, Y_train, callbacks=[clr])
        ```
    # Arguments
        base_lr: initial learning rate which is the
            lower boundary in the cycle.
        max_lr: upper boundary in the cycle. Functionally,
            it defines the cycle amplitude (max_lr - base_lr).
            The lr at any cycle is the sum of base_lr
            and some scaling of the amplitude; therefore
            max_lr may not actually be reached depending on
            scaling function.
        step_size: number of training iterations per
            half cycle. Authors suggest setting step_size
            2-8 x training iterations in epoch.
        mode: one of {triangular, triangular2, exp_range}.
            Default 'triangular'.
            Values correspond to policies detailed above.
            If scale_fn is not None, this argument is ignored.
        gamma: constant in 'exp_range' scaling function:
```
```

```
            gamma**(cycle iterations)
        scale_fn: Custom scaling policy defined by a single
            argument lambda function, where
            0 <= scale_fn(x) <= 1 for all x >= 0.
            mode paramater is ignored
        scale_mode: {'cycle', 'iterations'}.
            Defines whether scale_fn is evaluated on
            cycle number or cycle iterations (training
            iterations since start of cycle). Default is 'cycle'.
    """

    def __init__(self, base_lr=0.001, max_lr=0.006, step_size=2000., mode='triangular',
                 gamma=1., scale_fn=None, scale_mode='cycle'):
        super(CyclicLR, self).__init__()

        self.base_lr = base_lr
        self.max_lr = max_lr
        self.step_size = step_size
        self.mode = mode
        self.gamma = gamma
        if scale_fn == None:
            if self.mode == 'triangular':
                self.scale_fn = lambda x: 1.
                self.scale_mode = 'cycle'
            elif self.mode == 'triangular2':
                self.scale_fn = lambda x: 1/(2.**(x-1))
                self.scale_mode = 'cycle'
            elif self.mode == 'exp_range':
                self.scale_fn = lambda x: gamma**(x)
                self.scale_mode = 'iterations'
        else:
            self.scale_fn = scale_fn
            self.scale_mode = scale_mode
        self.clr_iterations = 0.
        self.trn_iterations = 0.
        self.history = {}

        self._reset()

    def _reset(self, new_base_lr=None, new_max_lr=None,
               new_step_size=None):
        """Resets cycle iterations.
        Optional boundary/step size adjustment.
        """
        if new_base_lr != None:
            self.base_lr = new_base_lr
        if new_max_lr != None:
            self.max_lr = new_max_lr
        if new_step_size != None:
            self.step_size = new_step_size
        self.clr_iterations = 0.

    def clr(self):
        cycle = np.floor(1+self.clr_iterations/(2*self.step_size))
        x = np.abs(self.clr_iterations/self.step_size - 2*cycle + 1)
        if self.scale_mode == 'cycle':
            return self.base_lr + (self.max_lr-self.base_lr)*np.maximum(0, (1-x))*self.scale_fn(cycle)
        else:
            return self.base_lr + (self.max_lr-self.base_lr)*np.maximum(0, (1-x))*self.scale_fn(self.clr_iterations)

    def on_train_begin(self, logs={}):
        logs = logs or {}

        if self.clr_iterations == 0:
            K.set_value(self.model.optimizer.lr, self.base_lr)
        else:
            K.set_value(self.model.optimizer.lr, self.clr())

    def on_batch_end(self, epoch, logs=None):

        logs = logs or {}
        self.trn_iterations += 1
        self.clr_iterations += 1

        self.history.setdefault('lr', []).append(K.get_value(self.model.optimizer.lr))
        self.history.setdefault('iterations', []).append(self.trn_iterations)

        for k, v in logs.items():
            self.history.setdefault(k, []).append(v)

        K.set_value(self.model.optimizer.lr, self.clr())
```

In [ ]:
```
from keras.layers import Dense, Conv2D, AveragePooling2D, Flatten, Dropout
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

clr_model = Sequential([
    Conv2D(6, kernel_size=5, strides=(1, 1), activation='tanh', input_shape=(28, 28, 1)),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),

    Conv2D(16, kernel_size=5, strides=(1, 1), activation='tanh'),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),

    Flatten(),
    Dense(120, activation='tanh'),

    Dense(84, activation='tanh'),

    Dense(10, activation='softmax')
])
```

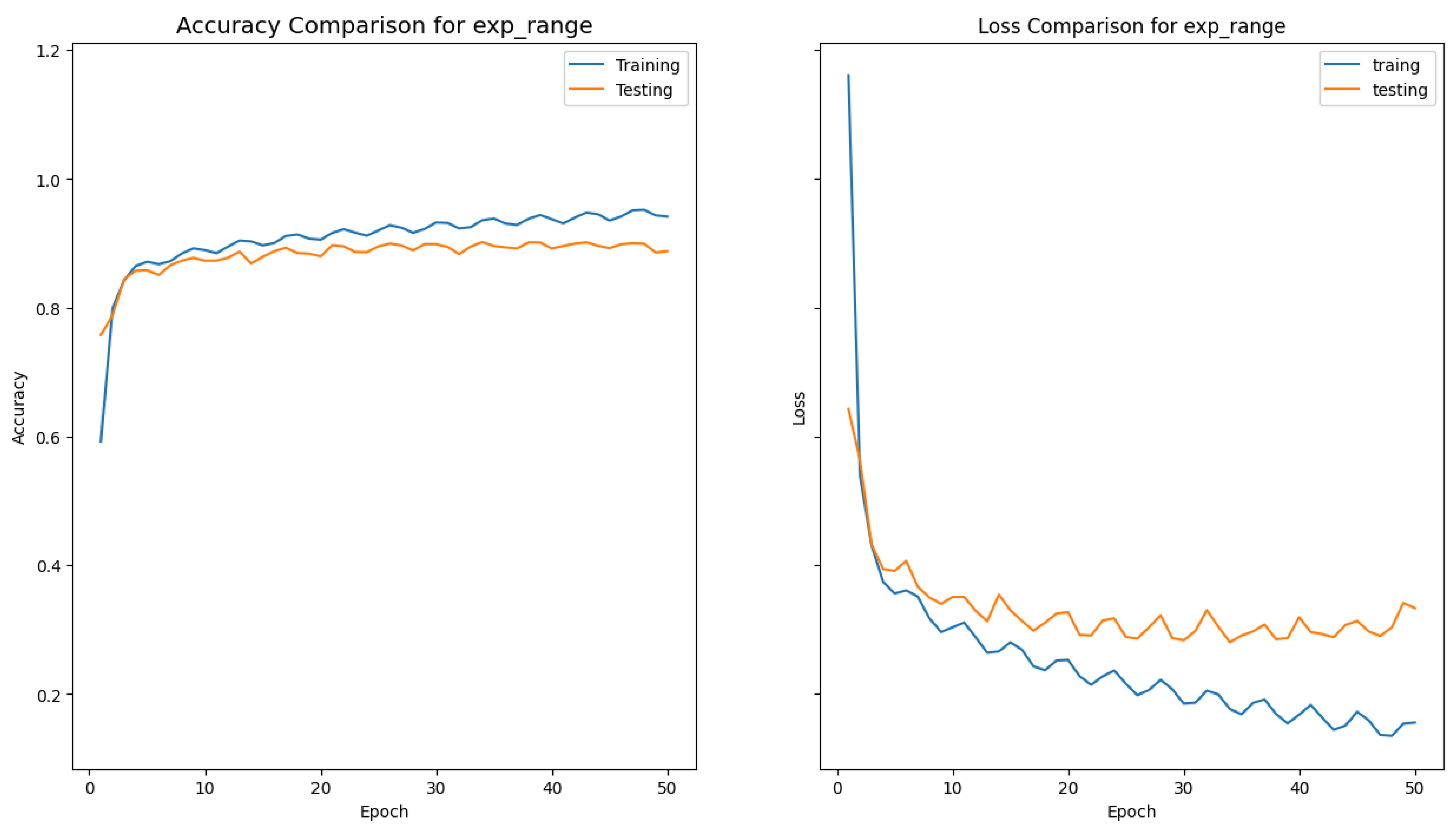
In [ ]:
```
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Input
clr_exp = CyclicLR(base_lr=10**-5, max_lr=10**-1, mode='exp_range')
clr_model.compile(loss='categorical_crossentropy', optimizer=SGD(), metrics=['accuracy'])
clr_model_history =clr_model.fit(X_train_fashion, y_train_fashion, callbacks=[clr_exp], \
              validation_data=(X_test_fashion, y_test_fashion), \
              epochs=50, batch_size=64,\
              verbose = 1)
```

```
Epoch 1/50
938/938 [==============================] - 6s 6ms/step - loss: 1.1606 - accuracy: 0.5925 - val_loss: 0.6425 - val_acc
uracy: 0.7575
Epoch 2/50
938/938 [==============================] - 5s 5ms/step - loss: 0.5391 - accuracy: 0.7996 - val_loss: 0.5637 - val_acc
uracy: 0.7885
Epoch 3/50
938/938 [==============================] - 5s 5ms/step - loss: 0.4307 - accuracy: 0.8428 - val_loss: 0.4326 - val_acc
uracy: 0.8443
Epoch 4/50
938/938 [==============================] - 5s 5ms/step - loss: 0.3744 - accuracy: 0.8649 - val_loss: 0.3942 - val_acc
uracy: 0.8575
Epoch 5/50
938/938 [==============================] - 5s 5ms/step - loss: 0.3560 - accuracy: 0.8715 - val_loss: 0.3912 - val_acc
uracy: 0.8582
Epoch 6/50
938/938 [==============================] - 5s 5ms/step - loss: 0.3610 - accuracy: 0.8676 - val_loss: 0.4068 - val_acc
uracy: 0.8509
Epoch 7/50
938/938 [==============================] - 5s 5ms/step - loss: 0.3514 - accuracy: 0.8723 - val_loss: 0.3668 - val_acc
uracy: 0.8660
Epoch 8/50
938/938 [==============================] - 5s 5ms/step - loss: 0.3177 - accuracy: 0.8844 - val_loss: 0.3501 - val_acc
uracy: 0.8732
Epoch 9/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2962 - accuracy: 0.8923 - val_loss: 0.3401 - val_acc
uracy: 0.8775
Epoch 10/50
938/938 [==============================] - 5s 6ms/step - loss: 0.3036 - accuracy: 0.8895 - val_loss: 0.3507 - val_acc
uracy: 0.8731
Epoch 11/50
938/938 [==============================] - 5s 5ms/step - loss: 0.3110 - accuracy: 0.8849 - val_loss: 0.3509 - val_acc
uracy: 0.8733
Epoch 12/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2879 - accuracy: 0.8950 - val_loss: 0.3290 - val_acc
uracy: 0.8776
Epoch 13/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2641 - accuracy: 0.9044 - val_loss: 0.3130 - val_acc
uracy: 0.8872
Epoch 14/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2661 - accuracy: 0.9031 - val_loss: 0.3544 - val_acc
uracy: 0.8687
Epoch 15/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2803 - accuracy: 0.8969 - val_loss: 0.3302 - val_acc
uracy: 0.8790
Epoch 16/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2689 - accuracy: 0.9004 - val_loss: 0.3136 - val_acc
uracy: 0.8878
Epoch 17/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2431 - accuracy: 0.9115 - val_loss: 0.2982 - val_acc
uracy: 0.8932
Epoch 18/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2369 - accuracy: 0.9139 - val_loss: 0.3107 - val_acc
uracy: 0.8851
Epoch 19/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2519 - accuracy: 0.9076 - val_loss: 0.3250 - val_acc
uracy: 0.8841
Epoch 20/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2529 - accuracy: 0.9057 - val_loss: 0.3270 - val_acc
uracy: 0.8798
Epoch 21/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2276 - accuracy: 0.9163 - val_loss: 0.2919 - val_acc
uracy: 0.8972
Epoch 22/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2143 - accuracy: 0.9221 - val_loss: 0.2908 - val_acc
uracy: 0.8954
Epoch 23/50
938/938 [==============================] - 4s 5ms/step - loss: 0.2274 - accuracy: 0.9167 - val_loss: 0.3140 - val_acc
uracy: 0.8867
Epoch 24/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2365 - accuracy: 0.9120 - val_loss: 0.3174 - val_acc
uracy: 0.8864
Epoch 25/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2161 - accuracy: 0.9202 - val_loss: 0.2887 - val_acc
uracy: 0.8954
Epoch 26/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1982 - accuracy: 0.9283 - val_loss: 0.2861 - val_acc
uracy: 0.8996
Epoch 27/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2064 - accuracy: 0.9244 - val_loss: 0.3036 - val_acc
uracy: 0.8968
Epoch 28/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2221 - accuracy: 0.9166 - val_loss: 0.3223 - val_acc
uracy: 0.8891
Epoch 29/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2074 - accuracy: 0.9225 - val_loss: 0.2868 - val_acc
uracy: 0.8988
Epoch 30/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1856 - accuracy: 0.9325 - val_loss: 0.2835 - val_acc
uracy: 0.8987
Epoch 31/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1868 - accuracy: 0.9317 - val_loss: 0.2977 - val_acc
uracy: 0.8945
```

```
Epoch 32/50
938/938 [==============================] - 5s 5ms/step - loss: 0.2053 - accuracy: 0.9232 - val_loss: 0.3302 - val_acc
uracy: 0.8834
Epoch 33/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1993 - accuracy: 0.9253 - val_loss: 0.3043 - val_acc
uracy: 0.8951
Epoch 34/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1769 - accuracy: 0.9359 - val_loss: 0.2804 - val_acc
uracy: 0.9020
Epoch 35/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1688 - accuracy: 0.9388 - val_loss: 0.2908 - val_acc
uracy: 0.8960
Epoch 36/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1864 - accuracy: 0.9309 - val_loss: 0.2971 - val_acc
uracy: 0.8939
Epoch 37/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1919 - accuracy: 0.9287 - val_loss: 0.3079 - val_acc
uracy: 0.8921
Epoch 38/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1692 - accuracy: 0.9385 - val_loss: 0.2851 - val_acc
uracy: 0.9016
Epoch 39/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1548 - accuracy: 0.9441 - val_loss: 0.2868 - val_acc
uracy: 0.9013
Epoch 40/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1682 - accuracy: 0.9377 - val_loss: 0.3192 - val_acc
uracy: 0.8920
Epoch 41/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1834 - accuracy: 0.9310 - val_loss: 0.2962 - val_acc
uracy: 0.8961
Epoch 42/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1634 - accuracy: 0.9402 - val_loss: 0.2930 - val_acc
uracy: 0.8995
Epoch 43/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1447 - accuracy: 0.9480 - val_loss: 0.2882 - val_acc
uracy: 0.9016
Epoch 44/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1512 - accuracy: 0.9454 - val_loss: 0.3070 - val_acc
uracy: 0.8963
Epoch 45/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1727 - accuracy: 0.9355 - val_loss: 0.3136 - val_acc
uracy: 0.8926
Epoch 46/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1594 - accuracy: 0.9419 - val_loss: 0.2971 - val_acc
uracy: 0.8985
Epoch 47/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1367 - accuracy: 0.9513 - val_loss: 0.2901 - val_acc
uracy: 0.9003
Epoch 48/50
938/938 [==============================] - 5s 5ms/step - loss: 0.1353 - accuracy: 0.9522 - val_loss: 0.3034 - val_acc
uracy: 0.8994
Epoch 49/50
938/938 [==============================] - 4s 5ms/step - loss: 0.1543 - accuracy: 0.9436 - val_loss: 0.3415 - val_acc
uracy: 0.8859
Epoch 50/50
938/938 [==============================] - 4s 5ms/step - loss: 0.1560 - accuracy: 0.9418 - val_loss: 0.3334 - val_acc
uracy: 0.8879
```

In [ ]:

```
# plot train/validation loss and accuracy curve over the number of epochs.

fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(15, 8), sharey=True, sharex=True)

# Train accuracy for both models
plt.sca(axs[0])
plt.plot(np.arange(1,51), clr_model_history.history['accuracy'], label='Training')
plt.plot(np.arange(1,51), clr_model_history.history['val_accuracy'], label='Testing')
plt.title('Accuracy Comparison for exp_range', size = 14)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.sca(axs[1])
plt.plot(np.arange(1,51), clr_model_history.history['loss'], label='traing')
plt.plot(np.arange(1,51), clr_model_history.history['val_loss'], label='testing')
plt.title('Loss Comparison for exp_range')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



It is correct and show similarity with the figure in the paper, where you have fluctuating curve when accuracy and loss tend to be stable because of the cyclical learning rate policy (with exponential decay).

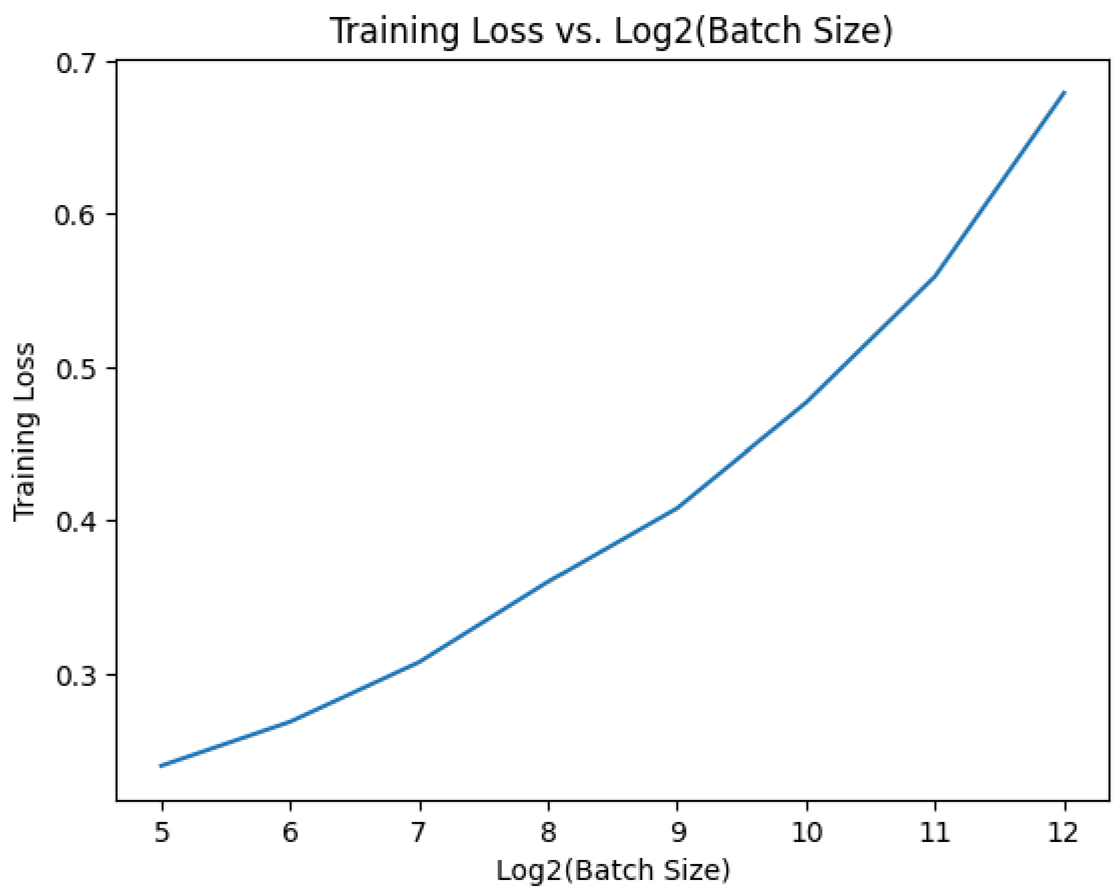
5.3

```
In [ ]: from keras.layers import Dense, Conv2D, AveragePooling2D, Flatten, Dropout
        from keras.models import Sequential
        from keras.optimizers import SGD
        from keras.utils import to_categorical
        import numpy as np
        import matplotlib.pyplot as plt
        import tensorflow as tf

        def create_lr_batch_model():
          return Sequential([
              Conv2D(6, kernel_size=5, strides=(1, 1), activation='tanh', input_shape=(28, 28, 1)),
              AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),

              Conv2D(16, kernel_size=5, strides=(1, 1), activation='tanh'),
              AveragePooling2D(pool_size=(2, 2), strides=(2, 2)),

              Flatten(),
              Dense(120, activation='tanh'),

              Dense(84, activation='tanh'),

              Dense(10, activation='softmax')
          ])
```

```
In [ ]: from keras.models import Sequential, Model
        from keras.layers import Dense, Activation, Input

        training_loss = []
        for i in range(5, 13):
          clr_batch_model = create_lr_batch_model()
          clr_batch_model.compile(loss='categorical_crossentropy', optimizer=SGD(learning_rate=10**-1),\
                                  metrics=['accuracy'])
          clr_model_history =clr_batch_model.fit(X_train_fashion, y_train_fashion, \
                          validation_data=(X_test_fashion, y_test_fashion), \
                          epochs=10, batch_size=2**i,\
                          verbose = 0)

          training_loss.append(clr_model_history.history['loss'][-1])
          print(f'Batch size: {2**i} finished')
```

```
Batch size: 32 finished
Batch size: 64 finished
Batch size: 128 finished
Batch size: 256 finished
Batch size: 512 finished
Batch size: 1024 finished
Batch size: 2048 finished
Batch size: 4096 finished
```

```
In [ ]: # Plot the training loss vs. log2(batch size).

        plt.plot(np.arange(5, 13), training_loss)
        plt.xlabel('Log2(Batch Size)')
        plt.ylabel('Training Loss')
        plt.title('Training Loss vs. Log2(Batch Size)')
        plt.show()
```



The accuracy graph for part 2 shows that the model is training consistently with the cyclical learning rate policy. And the loss shows the expected behavior of a cyclical learning rate: loss decreases as the learning rate cycles between lrmin and lrmax.

The training loss versus log2(batch size) graph has a clear trend: as the batch size increases, the training loss also increases. This could be because larger batch sizes provide a more accurate estimate of the gradient.

The cyclical learning rate policy is a balance between the loss (when the learning rate is high) and exploiting narrow areas of the loss (when learning rate is low). The cyclical nature allows the model to escape local minima and can be beneficial for generalization.

In contrast, increasing the batch size while keeping the learning rate fixed does not provide the same benefits. The lack of generalization could be due to less noise in the gradient estimates, which can lead to poorer exploration of the loss. So the generalization is different from cyclical learning rate policy.