Hw 5 Jerry Chen 6648517090

$$a = h(s) = \frac{1}{\sum_{m=1}^{M} e^{s_m}} \begin{bmatrix} e^{s_1} \\ e^{s_2} \\ \vdots \\ e^{s_m} \end{bmatrix} \qquad C = -\sum_{i=1}^{n} y_i \ln a_i$$

$$\frac{\partial C}{\partial a_i} = \frac{\partial(-y_i \ln(a_i))}{\partial a_i}$$

$$= -y_i \cdot \frac{1}{a_i}$$

$$= -\frac{y_i}{a_i}$$

1. case 1 $i = j$

$$a_i = \frac{e^{s_i}}{\sum_{m=1}^{M} e^{s_m}}$$

$$\frac{\partial a_i}{\partial s_i} = \frac{\partial \frac{e^{s_i}}{\sum_{m=1}^{M} e^{s_m}}}{\partial s_i}$$

$$= \frac{\frac{\partial e^{s_i}}{\partial s_i} \cdot \sum_{m=1}^{M} e^{s_m} - e^{s_i} \cdot \frac{\partial \sum_{m=1}^{M} e^{s_m}}{\partial s_i}}{\left(\sum_{m=1}^{M} e^{s_m}\right)^2}$$

$$= \frac{e^{s_i} \cdot \sum_{m=1}^{M} e^{s_m} - e^{s_i} \cdot e^{s_i}}{\left(\sum_{m=1}^{M} e^{s_m}\right)^2}$$

$$= \frac{e^{s_i}\left(\sum_{m=1}^{M} e^{s_m} - e^{s_i}\right)}{\left(\sum_{m=1}^{M} e^{s_m}\right)^2} \qquad a_i = \frac{e^{s_i}}{\sum_{m=1}^{M} e^{s_m}}$$

$$= a_i(1-a_i)$$

Case 2: $i \neq j$

$$a_i = \frac{e^{s_i}}{\sum_{m=1}^{M} e^{s_m}}$$

$$\frac{\partial a_i}{\partial s_j} = \frac{\frac{\partial e^{s_i}}{\partial s_j} \cdot \sum_{m=1}^{M} e^{s_m} - e^{s_i} \cdot \frac{\partial \sum_{m=1}^{M} e^{s_m}}{\partial s_j}}{\left(\sum_{m=1}^{M} e^{s_m}\right)^2}$$

$$= \frac{0 - e^{s_i} \cdot e^{s_j}}{\left(\sum_{m=1}^{M} e^{s_m}\right)^2}$$

$$= -a_i \cdot a_j$$

$$\frac{\partial a_i}{\partial s_j} = a_i(\delta_{ij} - a_j) \qquad \delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}$$

$$\delta_j = \frac{\partial C}{\partial s_j} = \sum_{i=1}^{M} \frac{\partial C}{\partial a_i} \cdot \frac{\partial a_i}{\partial s_j}$$

$$= \sum_{i=1}^{M} \left(-\frac{y_i}{a_i}\right) \cdot a_i(\delta_{ij} - a_j)$$

$$= -\sum_{i=1}^{M} y_i(\delta_{ij} - a_j)$$

$$= -y_j(1-a_j) + \sum_{i \neq j} (-y_i)(-a_j)$$

$$= -y_j(1-a_j) + 0$$

$$= a_j - y_j$$

$$\boxed{\therefore \delta = a - y}$$

2, (b)

(i) $$p(y=k \mid x_n) = \frac{\exp(w_k^T x_n)}{\sum_{j=1}^{K} \exp(w_j^T x_n)}$$

$$L(w) = \prod_{n=1}^{N} p(y=k_n \mid x_n)$$

$$= \sum_{n=1}^{N} \log p(y=k_n \mid x_n)$$

$$= \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \log p(y=k \mid x_n)$$

$$\frac{\partial L(w)}{\partial w_k} = \sum_{n=1}^{N} \frac{\partial}{\partial w_k} \log p(y=k \mid x_n)$$

$$= \frac{1}{p(y=k \mid x_n)} \cdot \frac{\partial p(y=k \mid x_n)}{\partial w_k}$$

case 1: $i = k$

$$\frac{\partial p(y=k \mid x_n)}{\partial w_k} = p(y=k \mid x_n)(1 - p(y=k \mid x_n))x_n$$

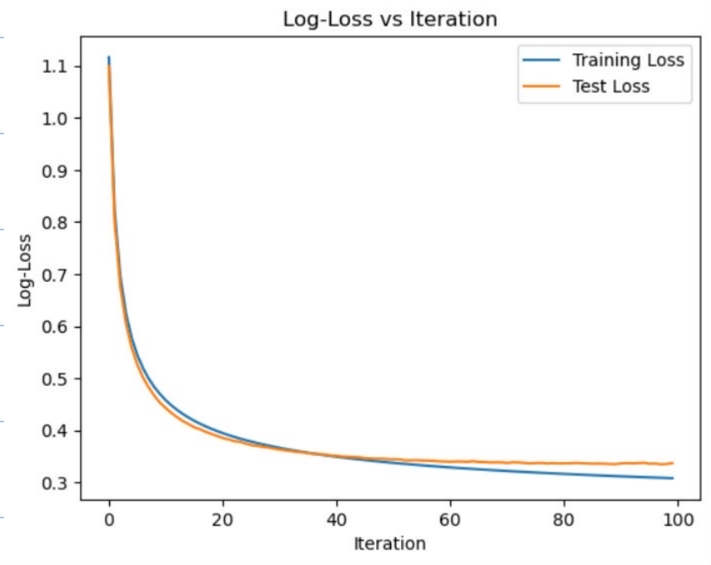
case 2: $i \neq k$

$$\frac{\partial p(y=k \mid x_n)}{\partial w_i} = -p(y=k \mid x_n)\, p(y=i \mid x_n)\, x_n$$

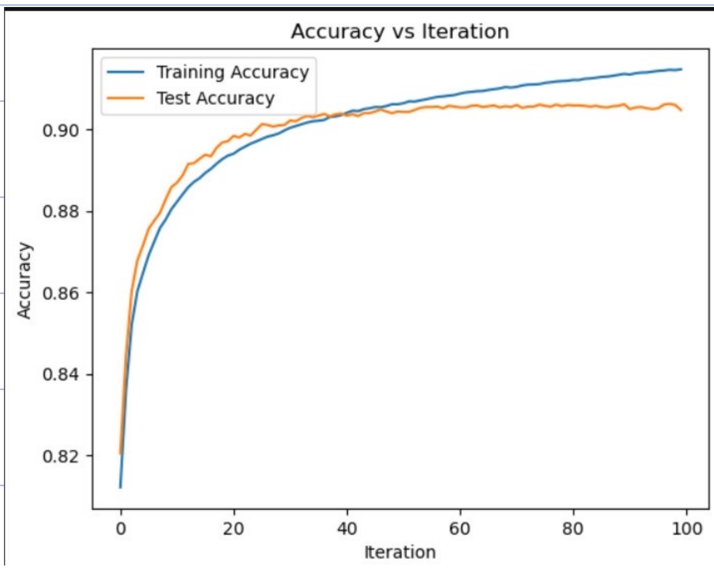
$$\boxed{\frac{\partial L(w)}{\partial w_k} = \sum_{n=1}^{N} x_n (p(y=k \mid x_n - y_{nk})}$$

(ii) I use 0.005 as my learning rate

(iii)




log-loss of the training set and test set.



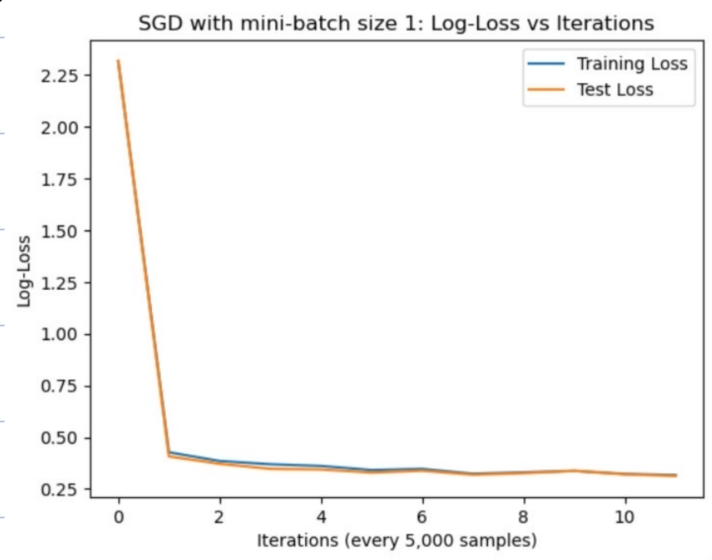

Accuracy of the training set and test set.

(iv)

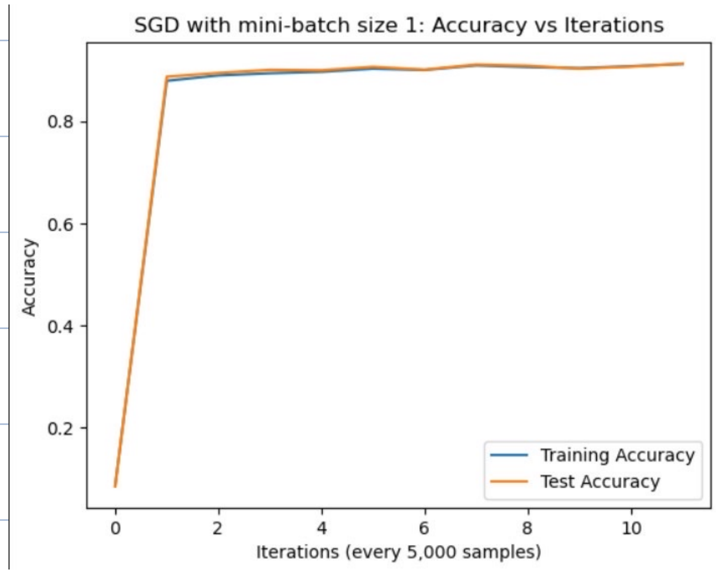
Final Train Loss: 0.3081253444765434
Final Test Loss: 0.3355847460977488‌5
Final Train Accuracy: 0.91475
Final Test Accuracy: 0.9059

(c)

(i)




log-loss of SGD with mini-batch size of 1.




Accuracy of SGD with mini-batch size of 1.

After about 10 iteration. SGD reached a performance Comparable to BGD in terms of both log-loss and accuracy.

The learning rate plays a critical role in SGD's performance. A higher learning rate allowing the model to converge faster. but if it is too high, it might cause instability. I try with 0.001, 0.01. 0.05. and 0.01 worked well. with a higher learning rate. (0.01) the model converge quickly in about 10 iteration,

With a lower learning rate (0.001), SGD requires more iteration to achieve same level of performance.

(ii)

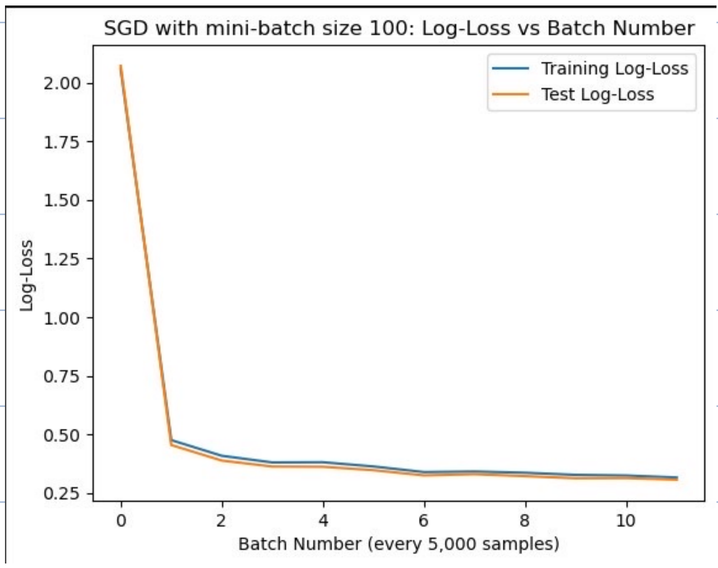
For BGD, $N = 60000$ traing samples

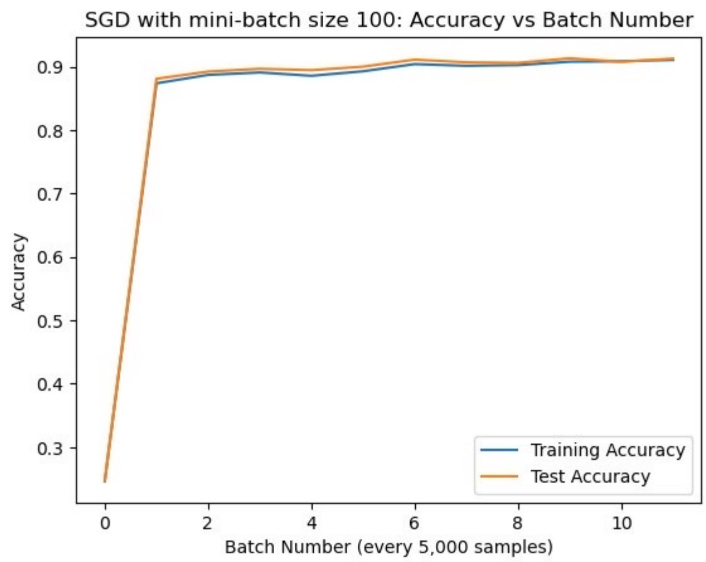
Total complexity (BGD) $= O(100 \times 60000) = O(6{,}000{,}000)$

For SGD, Total complexity (SGD) $= O(1 \times 50000) = O(50000)$

SGD (with mini-batch size $=1$) is computationally less expensive than BGD for this problem.

(iii)




log-loss of SGD with mini-batch size of 100.

Accuracy of SGD with mini-batch size of 100.

After about 10 iteration. SGD reached a performance comparable to BGD in terms of both log-loss and accuracy.

With a mini-batch size $= 100$, we need to increase learning rate (0.5) to allowed it to converge faster, reaching comparable performance to BGD with 10 iterations.

(iv)

Total Complexity of SGD (mini-batch size = 100) = O(100 × 10)

= O(1000)

Comparison:

| | Total Complexity |
|---|---|
| SGD (Mini-batch size = 100) | O(1000) |
| SGD (Mini-batch size = 1) | O(50000) |
| BGD | O(6,000,000) |

SGD with mini-batch size = 100 is much more efficient than both SGD with mini-batch size = 1 and BGD in terms of total computational complexity.

Appendix:

```python
import h5py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder

# Load MNIST data
with h5py.File('mnist_traindata.hdf5', 'r') as f:
    X_train = np.array(f['xdata'])
    y_train = np.array(f['ydata'])

with h5py.File('mnist_testdata.hdf5', 'r') as f:
    X_test = np.array(f['xdata'])
    y_test = np.array(f['ydata'])
# Already one-hot encoded in the dataset
y_train_one_hot = y_train
y_test_one_hot = y_test

# Print to check shape
print("X_train shape:", X_train.shape)
print("y_train_one_hot shape:", y_train_one_hot.shape)
print("X_test shape:", X_test.shape)
print("y_test_one_hot shape:", y_test_one_hot.shape)
```

```python
# Initialize weights and bias
num_classes = y_train_one_hot.shape[1]  # 10 for MNIST
num_features = X_train.shape[1]  # 784 (28x28 pixels)
weights = np.random.randn(num_classes, num_features) * 0.01  # (10, 784)
bias = np.zeros((num_classes, 1))  # Shape: (10, 1)
learning_rate = 0.005
epochs = 100
batch_size = 128

# Lists to store loss and accuracy
train_loss = []
test_loss = []
train_accuracy = []
test_accuracy = []

# Softmax function
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # For numerical stability
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

# Categorical cross-entropy loss
def categorical_cross_entropy_loss(y_true, y_pred):
    epsilon = 1e-7  # Small value to prevent log(0)
    return -np.sum(y_true * np.log(y_pred + epsilon)) / y_true.shape[0]

# Compute gradient
def compute_gradient(X, y_true, y_pred):
    N = X.shape[0]  # Number of samples in batch
    dw = np.dot((y_pred - y_true).T, X) / N  # Weight gradient
    db = np.sum(y_pred - y_true, axis=0, keepdims=True).T / N  # Bias gradient
    return dw, db

# Training loop
for epoch in range(epochs):
    # Shuffle data
    indices = np.random.permutation(X_train.shape[0])
    X_train_shuffled = X_train[indices]
    y_train_one_hot_shuffled = y_train_one_hot[indices]

    # Process in mini-batches
    for i in range(0, X_train.shape[0], batch_size):
        X_batch = X_train_shuffled[i:i + batch_size]
        y_batch = y_train_one_hot_shuffled[i:i + batch_size]

        # Forward pass
        z_batch = np.dot(X_batch, weights.T) + bias.T  # Include bias in the logits
        y_pred_batch = softmax(z_batch)  # Compute softmax probabilities

        # Compute gradient and update weights and bias
        dw, db = compute_gradient(X_batch, y_batch, y_pred_batch)
        weights -= learning_rate * dw  # Update weights
        bias -= learning_rate * db  # Update bias
```

```python
# After each epoch, evaluate the full training set
z_train = np.dot(X_train, weights.T) + bias.T  # Forward pass with bias
y_pred_train = softmax(z_train)
loss_train = categorical_cross_entropy_loss(y_train_one_hot, y_pred_train)
train_loss.append(loss_train)

predictions_train = np.argmax(y_pred_train, axis=1)
true_labels_train = np.argmax(y_train_one_hot, axis=1)
accuracy_train = np.mean(predictions_train == true_labels_train)
train_accuracy.append(accuracy_train)

#Evaluate on test set
z_test = np.dot(X_test, weights.T)
y_pred_test = softmax(z_test)
loss_test = categorical_cross_entropy_loss(y_test_one_hot, y_pred_test)
test_loss.append(loss_test)

predictions_test = np.argmax(y_pred_test, axis=1)
true_labels_test = np.argmax(y_test_one_hot, axis=1)  # Convert one-hot back to labels
accuracy_test = np.mean(predictions_test == true_labels_test)
test_accuracy.append(accuracy_test)

# Print progress every 10 epochs
if epoch % 10 == 0:
    print(f"Epoch {epoch}: Train Loss = {loss_train}, Test Loss = {loss_test}")
# Compute final accuracy and loss
print(f"Final Train Loss: {train_loss[-1]}, Final Test Loss: {test_loss[-1]}")
print(f"Final Train Accuracy: {train_accuracy[-1]}, Final Test Accuracy: {test_accuracy[-1]}")

# Plot log-loss of the training set and test set on the same figure.
plt.plot(train_loss, label="Training Loss")
plt.plot(test_loss, label="Test Loss")
plt.xlabel("Iteration")
plt.ylabel("Log-Loss")
plt.title("Log-Loss vs Iteration")
plt.legend()
plt.show()

plt.plot(train_accuracy, label="Training Accuracy")
plt.plot(test_accuracy, label="Test Accuracy")
plt.xlabel("Iteration")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Iteration")
plt.legend()
plt.show()
```

```python
import h5py
import numpy as np
import matplotlib.pyplot as plt

# Load MNIST data (same as before)
with h5py.File('mnist_traindata.hdf5', 'r') as f:
    X_train = np.array(f['xdata'])
    y_train = np.array(f['ydata'])

with h5py.File('mnist_testdata.hdf5', 'r') as f:
    X_test = np.array(f['xdata'])
    y_test = np.array(f['ydata'])

y_train_one_hot = y_train
y_test_one_hot = y_test

# Initialize weights and bias
num_classes = y_train_one_hot.shape[1]  # 10 for MNIST
num_features = X_train.shape[1]  # 784 (28x28 pixels)
weights = np.random.randn(num_classes, num_features) * 0.01  # (10, 784)
bias = np.zeros((num_classes, 1))  # Bias term initialized as zeros

learning_rate = 0.01
epochs = 1  # For SGD, we only pass through the data one time
```

```python
# Lists to store loss and accuracy
train_loss_sgd = []
test_loss_sgd = []
train_accuracy_sgd = []
test_accuracy_sgd = []

# Softmax function
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # For numerical stability
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

# Categorical cross-entropy loss
def categorical_cross_entropy_loss(y_true, y_pred):
    epsilon = 1e-7
    return -np.sum(y_true * np.log(y_pred + epsilon)) / y_true.shape[0]

# Compute gradient
def compute_gradient(X, y_true, y_pred):
    N = X.shape[0]  # Number of samples in batch
    dw = np.dot((y_pred - y_true).T, X)  # Gradient for weights
    db = np.sum(y_pred - y_true, axis=0, keepdims=True).T  # Gradient for bias
    return dw, db
```

```python
# SGD with mini-batch size = 1 (process one sample at a time)
for epoch in range(epochs):
    for i in range(X_train.shape[0]):
        X_sample = X_train[i:i + 1]  # One sample at a time
        y_sample = y_train_one_hot[i:i + 1]  # Corresponding label

        # Forward pass: include bias
        z_sample = np.dot(X_sample, weights.T) + bias.T  # Include bias in the logits
        y_pred_sample = softmax(z_sample)

        # Compute gradient and update weights and bias
        dw, db = compute_gradient(X_sample, y_sample, y_pred_sample)
        weights -= learning_rate * dw
        bias -= learning_rate * db

        # Record the log-loss and accuracy every 5,000 samples
        if i % 5000 == 0:
            # Training log-loss and accuracy
            z_train = np.dot(X_train, weights.T) + bias.T
            y_pred_train = softmax(z_train)
            loss_train = categorical_cross_entropy_loss(y_train_one_hot, y_pred_train)
            train_loss_sgd.append(loss_train)
            predictions_train = np.argmax(y_pred_train, axis=1)
            true_labels_train = np.argmax(y_train_one_hot, axis=1)
            accuracy_train = np.mean(predictions_train == true_labels_train)
            train_accuracy_sgd.append(accuracy_train)

            # Test log-loss and accuracy
            z_test = np.dot(X_test, weights.T) + bias.T
            y_pred_test = softmax(z_test)
            loss_test = categorical_cross_entropy_loss(y_test_one_hot, y_pred_test)
            test_loss_sgd.append(loss_test)
            predictions_test = np.argmax(y_pred_test, axis=1)
            true_labels_test = np.argmax(y_test_one_hot, axis=1)
            accuracy_test = np.mean(predictions_test == true_labels_test)
            test_accuracy_sgd.append(accuracy_test)

# Plot log-loss and accuracy for every 5,000 samples
plt.plot(train_loss_sgd, label='Training Loss')
plt.plot(test_loss_sgd, label='Test Loss')
plt.xlabel('Iterations (every 5,000 samples)')
plt.ylabel('Log-Loss')
plt.legend()
plt.title("SGD with mini-batch size 1 (with Bias): Log-Loss vs Iterations")
plt.show()

plt.plot(train_accuracy_sgd, label='Training Accuracy')
plt.plot(test_accuracy_sgd, label='Test Accuracy')
plt.xlabel('Iterations (every 5,000 samples)')
plt.ylabel('Accuracy')
plt.legend()
plt.title("SGD with mini-batch size 1 (with Bias): Accuracy vs Iterations")
plt.show()
```

```python
import h5py
import numpy as np
import matplotlib.pyplot as plt

# Load MNIST data (same as before)
with h5py.File('mnist_traindata.hdf5', 'r') as f:
    X_train = np.array(f['xdata'])
    y_train = np.array(f['ydata'])

with h5py.File('mnist_testdata.hdf5', 'r') as f:
    X_test = np.array(f['xdata'])
    y_test = np.array(f['ydata'])

y_train_one_hot = y_train
y_test_one_hot = y_test

# Initialize weights and bias
num_classes = y_train_one_hot.shape[1]  # 10 for MNIST
num_features = X_train.shape[1]  # 784 (28x28 pixels)
weights = np.random.randn(num_classes, num_features) * 0.01  # (10, 784)
bias = np.zeros((num_classes, 1))  # Bias term initialized as zeros

learning_rate = 0.5  # Use a higher learning rate to improve convergence
epochs = 1  # Pass through the dataset once
batch_size = 100  # Mini-batch size of 100

# Lists to store loss and accuracy
train_loss_sgd = []
test_loss_sgd = []
train_accuracy_sgd = []
test_accuracy_sgd = []
batch_numbers = []  # To track the batch number for x-axis

# Softmax function
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # For numerical stability
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

# Categorical cross-entropy loss
def categorical_cross_entropy_loss(y_true, y_pred):
    epsilon = 1e-7
    return -np.sum(y_true * np.log(y_pred + epsilon)) / y_true.shape[0]

# Compute gradient for both weights and bias
def compute_gradient(X, y_true, y_pred):
    N = X.shape[0]  # Number of samples in batch
    dw = np.dot((y_pred - y_true).T, X) / N
    db = np.sum(y_pred - y_true, axis=0, keepdims=True).T / N  # Bias gradient
    return dw, db

# SGD with mini-batch size = 100
batch_counter = 0
```

```python
for epoch in range(epochs):
    # Shuffle the training data at the beginning of each epoch
    indices = np.random.permutation(X_train.shape[0])
    X_train_shuffled = X_train[indices]
    y_train_one_hot_shuffled = y_train_one_hot[indices]

    for i in range(0, X_train.shape[0], batch_size):
        X_batch = X_train_shuffled[i:i + batch_size]
        y_batch = y_train_one_hot_shuffled[i:i + batch_size]

        # Forward pass with bias
        z_batch = np.dot(X_batch, weights.T) + bias.T
        y_pred_batch = softmax(z_batch)

        # Compute gradient and update weights and bias
        dw, db = compute_gradient(X_batch, y_batch, y_pred_batch)
        weights -= learning_rate * dw
        bias -= learning_rate * db

        # Log performance every 5,000 samples
        if batch_counter % 50 == 0:  # 50 mini-batches of size 100 equals 5,000 samples
            # Training log-loss and accuracy
            z_train = np.dot(X_train, weights.T) + bias.T
            y_pred_train = softmax(z_train)
            loss_train = categorical_cross_entropy_loss(y_train_one_hot, y_pred_train)
            train_loss_sgd.append(loss_train)
            predictions_train = np.argmax(y_pred_train, axis=1)
            true_labels_train = np.argmax(y_train_one_hot, axis=1)
            accuracy_train = np.mean(predictions_train == true_labels_train)
            train_accuracy_sgd.append(accuracy_train)

            # Test log-loss and accuracy
            z_test = np.dot(X_test, weights.T) + bias.T
            y_pred_test = softmax(z_test)
            loss_test = categorical_cross_entropy_loss(y_test_one_hot, y_pred_test)
            test_loss_sgd.append(loss_test)
            predictions_test = np.argmax(y_pred_test, axis=1)
            true_labels_test = np.argmax(y_test_one_hot, axis=1)
            accuracy_test = np.mean(predictions_test == true_labels_test)
            test_accuracy_sgd.append(accuracy_test)

            # Append batch number for plotting
            batch_numbers.append(batch_counter // 50)

        batch_counter += 1

# Plot log-loss vs batch number
plt.plot(batch_numbers, train_loss_sgd, label='Training Log-Loss')
plt.plot(batch_numbers, test_loss_sgd, label='Test Log-Loss')
plt.xlabel('Batch Number (every 5,000 samples)')
plt.ylabel('Log-Loss')
plt.legend()
plt.title('SGD with mini-batch size 100: Log-Loss vs Batch Number')
plt.show()

# Plot accuracy vs batch number
plt.plot(batch_numbers, train_accuracy_sgd, label='Training Accuracy')
plt.plot(batch_numbers, test_accuracy_sgd, label='Test Accuracy')
plt.xlabel('Batch Number (every 5,000 samples)')
plt.ylabel('Accuracy')
plt.legend()
plt.title('SGD with mini-batch size 100: Accuracy vs Batch Number')
plt.show()
```

```python
# Save the model parameters
with h5py.File('mnist_network_params.hdf5', 'w') as hf:
    hf.create_dataset('w', data=np.asarray(weights))
    hf.create_dataset('b', data=np.asarray(bias))
print("Shape of weights (w):", weights.shape)
print("Shape of bias (b):", bias.shape)
```