Problem 1

This paper explores zeroth-order (ZO) optimization techniques, which eliminate the need for back-propagation by approximating gradients through function value differences rather than explicit gradient calculations. As LLMs continue to grow in size, the memory overhead generated by back-propagation during gradient computation presents a significant obstacle for applications like on-device training, where memory constraints are critical. And this technique addresses the challenge of memory inefficiency in fine-tuning large language models (LLMs) due to the extensive memory requirements of first-order (FO) optimization methods, such as stochastic gradient descent (SGD) and Adam. Though scaling up ZO optimization for deep model training is exceedingly challenging due to its high variance. Nevertheless, LLM pre-training offers a unique advantage by enabling the fine-tuner to start from a well-optimized pre-trained model state. This graceful model initialization makes ZO optimization potentially scalable to LLM finetuning tasks.

A key contribution of the paper is the benchmarking of six ZO optimization methods, such as ZO-SGD and ZO-Adam, and comparing them with FO methods. The benchmark reveals the trade-offs between memory efficiency and fine-tuning accuracy, showing that methods like ZO-SGD can significantly reduce memory usage while maintaining competitive accuracy, making them well-suited for memory-constrained settings. Furthermore, the authors introduce novel improvements to ZO optimization, including block-wise descent, which reduces variance in gradient estimation, and hybrid training, combining ZO and FO methods to strike a balance between memory efficiency and performance. These innovations demonstrate that ZO optimization offers a promising solution for scaling LLM fine-tuning to resource-limited environments.

Strength

- **Comprehensive Benchmarking**: This paper provides a detailed benchmark of different ZO optimization methods across multiple LLM architectures, tasks, and fine-tuning strategies. This helps researchers understand the trade-offs between different methods and gives a clearer picture of how ZO optimization compares to traditional FO approaches, particularly in terms of memory usage and query efficiency.
- **Innovation in Extended Study**: The introduction of block-wise descent and hybrid ZO-FO fine-tuning represents an important innovation. The proposed enhancements can further improve the finetuning accuracy while maintaining the memory efficiency.

Weakness

- **High Variance in Performance**: One of the weaknesses of ZO optimization identified in the paper is the high variance in performance across different ZO methods. For example, ZO-SGD and ZO-Adam exhibit fluctuating accuracy depending on the model and task, which can make it difficult to predict performance consistently across different LLMs. This indicates that the stability of ZO methods still needs improvement.
- **Limited Scalability for Larger Models and Complex Tasks**: While ZO optimization methods offer clear advantages in terms of memory efficiency, the paper highlights that as model size increases and task complexity grows, FO methods still outperform ZO methods in terms of accuracy. This suggests that ZO optimization might struggle with scaling up to the largest LLMs and more complex tasks.

Future Directions

- **Reducing Variance in ZO Optimization**: Further exploration of variance reduction techniques, such as importance sampling introduced in the previous paper, could help stabilize gradient estimates in ZO optimization. These techniques can reduce the noise introduced by ZO's gradient approximation and make the method more robust across different tasks and LLM architectures.
- **Privacy-Preserving Optimization**: An interesting possible direction for future research is applying ZO optimization in privacy-preserving machine learning. Since ZO methods do not require access to gradients, they can be more privacy-preserving compared to FO methods.

Problem 2

This paper introduces an input-aware dynamic backdoor attack, where the trigger is dynamically generated based on the input image which aims to improve the stealthiness and effectiveness of backdoor attacks on deep neural networks. Traditional backdoor attacks rely on static, fixed triggers that remain the same for all input images, making them detectable and vulnerable to defense methods. While this method ensures that each input has a unique trigger, significantly enhancing the stealthiness of the attack and bypassing existing defense mechanisms that rely on detecting fixed triggers.

The trigger generator used in this method is trained using a diversity loss to ensure that different input images produce distinct triggers, and a cross-trigger test ensures that triggers cannot be reused across images. By doing so, the attack can effectively poison models while maintaining high accuracy on clean data. The attack was tested on standard datasets such as MNIST, CIFAR-10, and GTSRB, achieving near-perfect attack success rates while bypassing state-of-the-art defense mechanisms like Neural Cleanse, STRIP, and Fine-Pruning. Additionally, the method was shown to resist common image regularization defenses such as spatial smoothing, further demonstrating its robustness.

Strength

- **Stealthiness and Resistance to Defenses**: By generating unique triggers for each input, the attack introduced in the paper becomes much harder to detect and defend against, as it breaks the assumption that backdoor triggers are static. This makes existing defense methods that rely on detecting fixed triggers ineffective.
- **Generalization Across Datasets**: The method is shown to be effective across different datasets and model architectures, demonstrating that it can generalize well to a variety of tasks. The high attack success rate coupled with the ability to maintain performance on clean data indicates that the attack is both powerful and adaptable.

Weakness

- **Lack of Real-World Testing**: While the method performs well in controlled experimental settings and it is claimed to be possibly useful in more scenarios, it has not yet been thoroughly evaluated in more complex, real-world applications, such as speech or face recognition tasks. Further validation is necessary to confirm the generalization of the attack to these domains.
- **Complexity in Implementation**: The input-aware trigger generation method requires careful design and training, including the use of a trigger generator and enforcing a diversity loss to prevent triggers from collapsing into fixed patterns. This added complexity may limit the practical deployment of the attack in many real-world scenarios .

Future Directions

- **Adversarial Noise as Triggers**: To address the problem mentioned in the paper that the generated triggers can appear unnatural, one possible approach would be to blend adversarial noise techniques with input-aware triggers, making the triggers indistinguishable from typical adversarial perturbations. This could further blur the line between backdoor attacks and common adversarial examples, making detection even more difficult.

- **Improving the Efficiency and Scalability of the Attack**: The current trigger generator architecture may introduce computational overhead, especially when applied to large-scale datasets or real-time applications. A future research direction could focus on improving the efficiency and scalability of the attack.
- **Adapting to Fine-Tuning Scenarios**: In modern AI, model fine-tuning on pre-trained models is a common practice. Then a key question is whether the dynamically generated triggers remain effective after the model undergoes fine-tuning on new tasks or datasets. Research could explore ways to ensure that the triggers are resilient to changes in the model weights, preserving their effectiveness even after fine-tuning.

Problem 3

(a) $L(w) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(1+\exp(-\langle x_i, w\rangle)) + (1-y_i)\langle x_i, w\rangle\right)$

$$\Rightarrow \nabla_w L(w) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i \cdot \frac{1}{1+\exp(-x_i^T w)} \cdot \exp(-x_i^T w)\cdot(-x_i) + (1-y_i)x_i\right)$$

$$\Rightarrow \nabla_w^2 L(w) = \frac{1}{N}\sum_{i=1}^{N}\left(-y_i x_i \cdot \frac{\exp(-x_i^T w)\cdot(-x_i)(1+\exp(-x_i^T w)) - \exp(-x_i^T w)\cdot\exp(-x_i^T w)\cdot(-x_i)}{(1+\exp(-x_i^T w))^2}\right)$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(y_i x_i x_i^T \frac{\exp(-x_i^T w)}{(1+\exp(-x_i^T w))^2}\right)$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(y_i x_i x_i^T \cdot \frac{1}{1+\exp(-x_i^T w)} \cdot \frac{\exp(-x_i^T w)}{1+\exp(-x_i^T w)}\right)$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(y_i \cdot \sigma(x_i^T w)\cdot(1-\sigma(x_i^T w))\, x_i x_i^T\right)$$

sigmoid function $\sigma(x_i^T w) = \frac{1}{1+\exp(-x_i^T w)}$

① In the expression of Hessian,

$x_i x_i^T$ is postive semi-definite

y_i is a scalar, $y_i \in \{0, +1\}$

$\sigma(x_i^T w)\cdot(1-\sigma(x_i^T w)) > 0$, because any sigmoid function $\in (0,1)$

so, the Hessian is also positive semi-definite

Then from this we know that $L(w)$ is convex

② Then, take $x_i^T w$ as the variable

$\sigma(x_i^T w)\cdot(1-\sigma(x_i^T w))$ will reach maximum when $\sigma(x_i^T w) = \frac{1}{2} \Leftrightarrow x_i^T w = 0$

and the maximum is $\frac{1}{4}$

So, the Hessian is upper bounded

$$\nabla_w L(w) \le \frac{1}{4N}\sum_{i=1}^{N} y_i \cdot x_i x_i^T$$

And also, $\frac{1}{4N}\sum_{i=1}^{N} y_i x_i x_i^T \le \frac{1}{4N}\sum_{i=1}^{N}(y_i\|x_i\|^2) Id$

↑ largest eigenvalue of $x_i x_i^T$ is $\|x_i\|^2$

so, we get $\nabla_w^2 L(w) \le \frac{1}{4N}\sum_{i=1}^{N}(y_i\|x_i\|^2)\cdot Id$

take $L = \frac{1}{4N}\sum_{i=1}^{N} y_i\|x_i\|^2$

then $\nabla_w^2 L(w) \le LId$, so $L(w)$ is L-smooth

③ Lastly, when $x_i^T w \to -\infty$ or $x_i^T w \to +\infty$, $\sigma(x_i^T w)(1-\sigma(x_i^T w)) \to 0$

which means that Hessian can approach 0 and it's impossible to lower bound the Hessian

so we can't find such a $\mu > 0$ to satisfy $\mu Id \le \nabla_w^2 L(w)$ for all w

so, $L(w)$ is not strongly convex

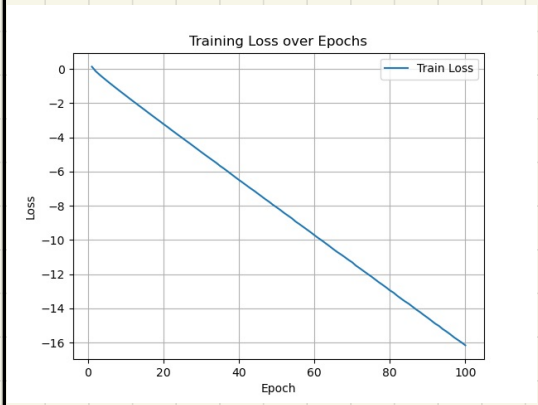
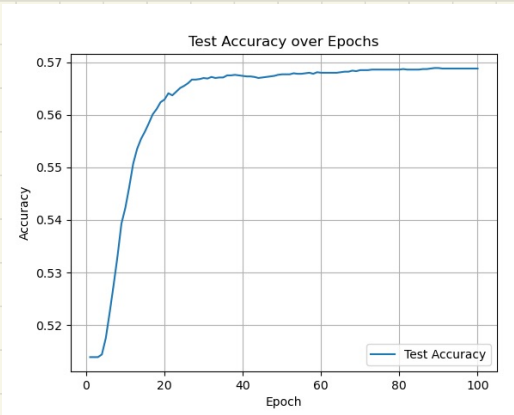
(b)

For this problem, there is an inevitable issue with loss function, that when $y_i = 0$, $L_i(w) = \langle x_i, w \rangle$ which can be negative and tends to be negative after training.

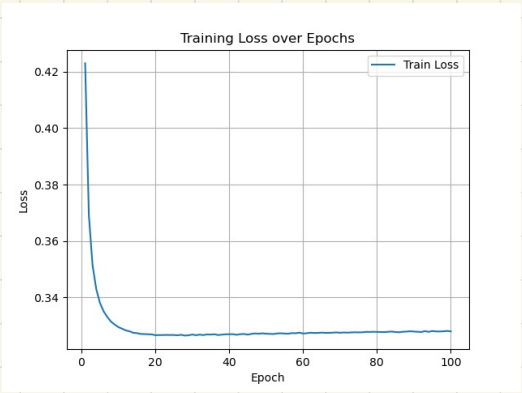
Apparently that is not the normal cases that we want to see, and also, this will lead to a BIG PROBLEM that when loss function is very likely to be Nan with slightly larger learning rate.

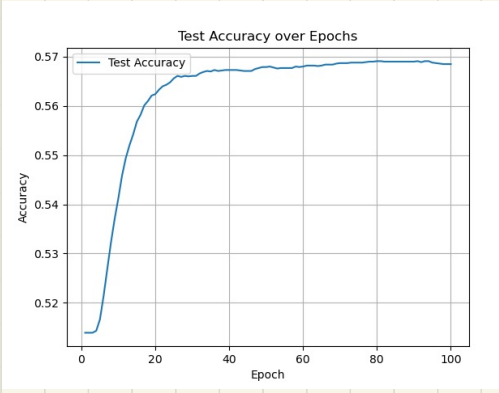
So, to address this problem, I manually change the $\langle x_i, w \rangle$ to y_pred, which is $\sigma(\langle x_i, w \rangle)$. But this will lead to another problem that our calculated gradient of loss function now doesn't match the loss function after manual limitation.

So here I plotted the original loss function and also loss function after limitation. And separately testing their accuracy on test_dataset to see whether our "manual limit" makes sense.

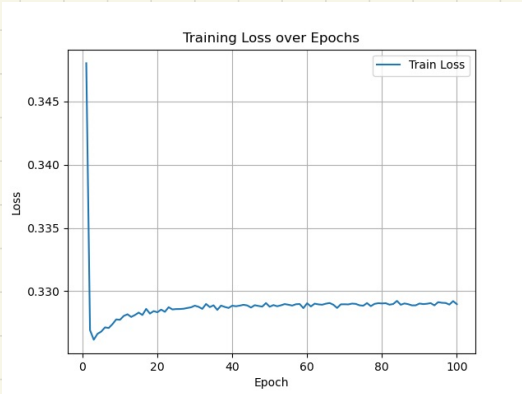
	loss	accuracy
original ($\eta = 0.001$)		

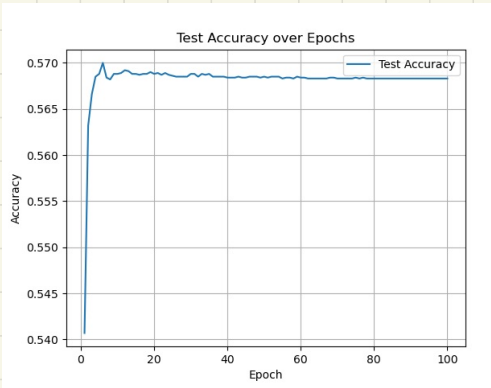
modified
($\langle x_i, w \rangle \rightarrow \sigma(\langle x_i, w \rangle)$)
($\eta = 0.001$)



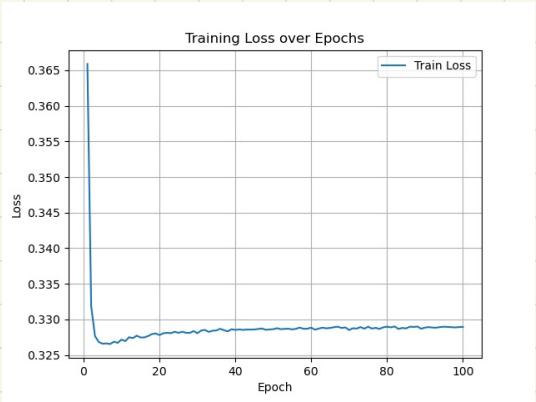


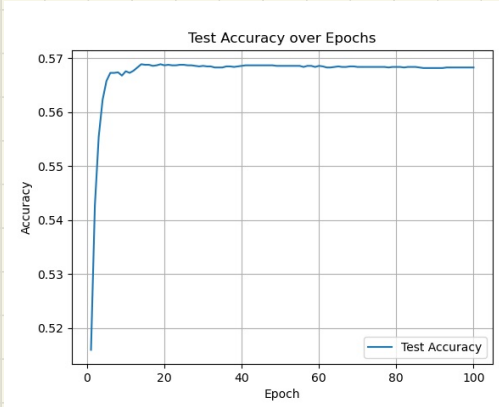
modified
($\eta = 0.005$)
$\eta = 0.005$, original one will crash





modified
($\eta = 0.01$)





Code for P3_b

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist

# Load and preprocess the MNIST dataset
def load_preprocess_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    # Flatten the 28x28 images into 784-dimensional vectors
    x_train = x_train.reshape(x_train.shape[0], 28*28).astype('float32') / 255.0
    x_test = x_test.reshape(x_test.shape[0], 28*28).astype('float32') / 255.0

    # Binarize the labels: 0-4 are class 0, 5-9 are class 1
    y_train_binary = np.where(y_train < 5, 0, 1)
    y_test_binary = np.where(y_test < 5, 0, 1)

    return x_train, y_train_binary, x_test, y_test_binary

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# 3. Logistic loss function
def logistic_loss(y_true, z, y_pred):
    # return np.mean(y_true * np.log(1 + np.exp(-z)) + (1 - y_true) * z)
    return np.mean(y_true * np.log(1 + np.exp(-y_pred)) + (1 - y_true) * y_pred)

# 4. Logistic regression model class
class LogisticRegression:
    def __init__(self, input_size):
        # Initialize weights
        self.w = np.random.randn(input_size) * 0.01

    def predict(self, X):
        # Linear combination of inputs and weights
        z = np.dot(X, self.w)
        # Apply sigmoid function to get probabilities
        y_pred = sigmoid(z)
        y_pred = np.clip(y_pred, 1e-9, 1 - 1e-9) # Avoiding infinite or zero log result
        return z, y_pred

    def compute_gradient(self, X, y_true, y_pred):
        # Compute the gradient of the loss function with respect to weights
        return np.dot(X.T, (1 - 2 * y_true + y_true * y_pred)) / X.shape[0]

    def update_weights(self, gradient, learning_rate):
        # Update weights using the gradient and learning rate
        self.w -= learning_rate * gradient
```

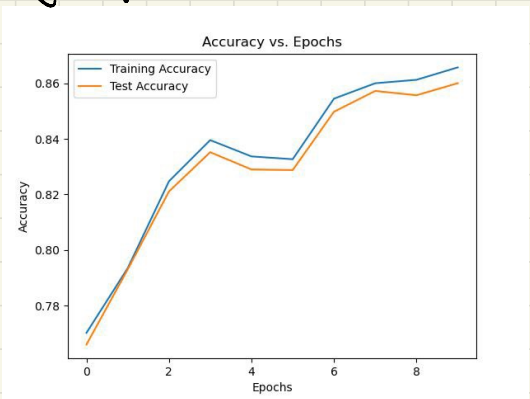
```
# # 5. Training function using SGD
# def train_model(model, X_train, y_train, epochs, batch_size, learning_rate):
#     losses = []

#     for epoch in range(epochs):
#         # Shuffle the training data
#         perm = np.random.permutation(len(X_train))
#         X_train_shuffled = X_train[perm]
#         y_train_shuffled = y_train[perm]

#         batch_losses = []

#         # Mini-batch gradient descent
#         for i in range(0, len(X_train), batch_size):
#             X_batch = X_train_shuffled[i:i + batch_size]
#             y_batch = y_train_shuffled[i:i + batch_size]

#             # Forward pass: make predictions
#             z, y_pred = model.predict(X_batch)

#             # Compute the loss
#             loss = logistic_loss(y_batch, z, y_pred)
#             batch_losses.append(loss)

#             # Backward pass: compute gradient
#             gradient = model.compute_gradient(X_batch, y_batch, y_pred)

#             # Update the model weights
#             model.update_weights(gradient, learning_rate)

#         # Average loss for the epoch
#         epoch_loss = np.mean(batch_losses)
#         losses.append(epoch_loss)
#         if (epoch+1) % 10 == 0:
#             print(f'Epoch {epoch+1}/{epochs}, Loss: {epoch_loss}')

#     return losses

# 5. Training function using SGD and tracking accuracy for each epoch
def train_model_with_accuracy(model, X_train, y_train, X_test, y_test, epochs, batch_size,
learning_rate):
    train_losses = []
    test_accuracies = []

    for epoch in range(epochs):
        # Shuffle the training data
        perm = np.random.permutation(len(X_train))
        X_train_shuffled = X_train[perm]
        y_train_shuffled = y_train[perm]
```

```
        batch_losses = []

        # Mini-batch gradient descent
        for i in range(0, len(X_train), batch_size):
            X_batch = X_train_shuffled[i:i + batch_size]
            y_batch = y_train_shuffled[i:i + batch_size]

            # Forward pass: make predictions
            z, y_pred = model.predict(X_batch)

            # Compute the training loss for this batch
            loss = logistic_loss(y_batch, z, y_pred)
            batch_losses.append(loss)

            # Backward pass: compute gradient
            gradient = model.compute_gradient(X_batch, y_batch, y_pred)

            # Update the model weights
            model.update_weights(gradient, learning_rate)

        # Average training loss for the epoch
        epoch_train_loss = np.mean(batch_losses)
        train_losses.append(epoch_train_loss)

        # Compute the test accuracy at the end of each epoch
        _, y_test_pred_prob = model.predict(X_test)
        y_test_pred_labels = (y_test_pred_prob >= 0.5).astype(int)
        test_accuracy = np.mean(y_test_pred_labels == y_test)  # Calculate test accuracy
        test_accuracies.append(test_accuracy)

        # Print training loss and test accuracy every 10 epochs
        if (epoch+1) % 10 == 0:
            print(f'Epoch {epoch+1}/{epochs}, Train Loss: {epoch_train_loss:.4f}, Test Accuracy:
{test_accuracy * 100:.2f}%')

    return train_losses, test_accuracies


# Main code to execute the training process
x_train, y_train, x_test, y_test = load_preprocess_data()
input_size = x_train.shape[1]  # Input size is 784 (28x28 flattened)

# Initialize the logistic regression model
logistic_model = LogisticRegression(input_size)

# Train the model using SGD
epochs = 100
batch_size = 256
```

```
learning_rate = 0.001

# Train the model and track accuracy over epochs
train_losses, test_accuracies = train_model_with_accuracy(logistic_model, x_train, y_train,
x_test, y_test, epochs, batch_size, learning_rate)

# Plot training loss over epochs
plt.plot(range(1, len(train_losses)+1), train_losses, label='Train Loss')
plt.title('Training Loss over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.savefig('p3_training_loss_lr005')
plt.clf()

# Plot test accuracy over epochs
plt.plot(range(1, len(test_accuracies)+1), test_accuracies, label='Test Accuracy')
plt.title('Test Accuracy over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.savefig('p3_testing_accuracy_lr005')
```

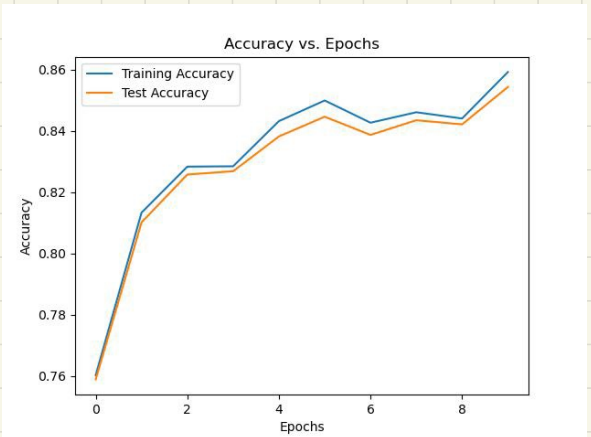
Problem 4

(a) $\eta = 0.1$

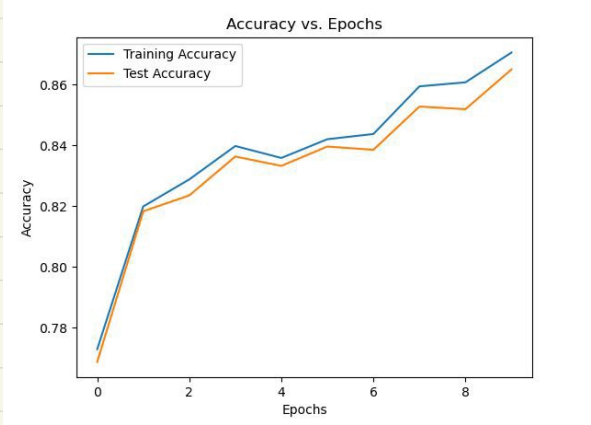


(b) $\eta = 0.1$

hidden layer : 128



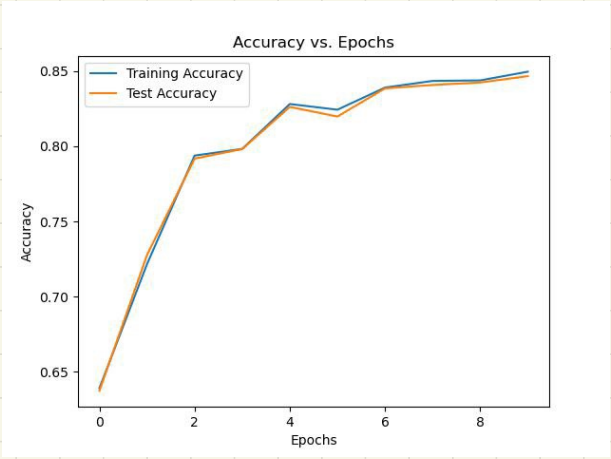
hidden layer : 512



Since the epoch is really limited, so there is no apparent conclusion for how the change of units in hidden layer can affect the performance

(C) $\gamma = 0.1$

adding another 256 hidden layer



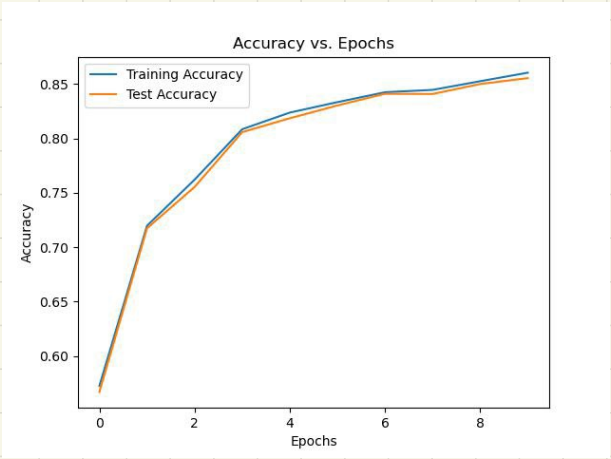
it seems like the model performs even a little bit worse than before because of adding more complexity

But still, need more epoch and tests to have a deterministic and reliable conclusion

(d)

adding dropout layer into training

dropout_rate = 0.5



I believe adding a dropout layer will make the training more robust. which can be reflected from the smoothier loss curve

But whether this will affect the final accuracy, still needing more epoches

Code for P4_abc

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Load Fashion-MNIST dataset
def load_fashion_mnist():
    fashion_mnist = fetch_openml("Fashion-MNIST")
    X = fashion_mnist.data / 255.0  # normalize pixel values
    y = fashion_mnist.target.astype(int)
    return X, y

# One-hot encoding for labels
def one_hot_encoding(y, num_classes):
    encoder = OneHotEncoder(sparse_output=False, categories='auto')
    y_one_hot = encoder.fit_transform(y.to_numpy().reshape(-1, 1))
    return y_one_hot

# ReLU activation
def relu(x):
    return np.maximum(0, x)

# Derivative of ReLU
def relu_derivative(x):
    return np.where(x > 0, 1, 0)

# Softmax function
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

# Cross-entropy loss
def cross_entropy_loss(y_true, y_pred):
    return -np.mean(np.sum(y_true * np.log(y_pred + 1e-8), axis=1))

# Compute accuracy
def accuracy(y_true, y_pred):
    return np.mean(np.argmax(y_true, axis=1) == np.argmax(y_pred, axis=1))

# # Initialize weights and biases
# def initialize_parameters(input_size, hidden_size, output_size):
#     np.random.seed(0)
#     W1 = np.random.randn(input_size, hidden_size) * 0.01
#     b1 = np.zeros((1, hidden_size))
#     W2 = np.random.randn(hidden_size, output_size) * 0.01
#     b2 = np.zeros((1, output_size))
#     return W1, b1, W2, b2
```

```
# Forward pass
def forward_pass(X, W1, b1, W2, b2, W3, b3):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = relu(Z2)
    Z3 = np.dot(A2, W3) + b3
    A3 = softmax(Z3)
    return Z1, A1, Z2, A2, Z3, A3

# Backward pass
def backward_pass(X, y_true, Z1, Z2, A1, A2, W2, A3, W3):
    m = X.shape[0]

    dZ3 = A3 - y_true
    dW3 = np.dot(A2.T, dZ3) / m
    db3 = np.sum(dZ3, axis=0, keepdims=True) / m

    dA2 = np.dot(dZ3, W3.T)
    dZ2 = dA2 * relu_derivative(Z2)
    dW2 = np.dot(A1.T, dZ2) / m
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = np.dot(X.T, dZ2) / m
    db1 = np.sum(dZ1, axis=0, keepdims=True) / m

    return dW1, db1, dW2, db2, dW3, db3

# Mini-batch SGD
def sgd_update(W1, b1, W2, b2, W3, b3, dW1, db1, dW2, db2, dW3, db3, learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    W3 -= learning_rate * dW3
    b3 -= learning_rate * db3
    return W1, b1, W2, b2, W3, b3

# Train MLP
def train_mlp(X_train, y_train, X_test, y_test, hidden_size, batch_size, epochs, learning_rate):
    input_size = X_train.shape[1]
    output_size = y_train.shape[1]

    # Initialize weights and biases
    np.random.seed(0)
    W1 = np.random.randn(input_size, hidden_size) * 0.01
```

```
    b1 = np.zeros((1, hidden_size))
    # W2 = np.random.randn(hidden_size, output_size) * 0.01
    # b2 = np.zeros((1, output_size))
    W2 = np.random.randn(hidden_size,hidden_size) * 0.01
    b2 = b1 = np.zeros((1, hidden_size))
    W3 = np.random.randn(hidden_size, output_size) * 0.01
    b3 = np.zeros((1, output_size))

    training_acc = []
    testing_acc = []

    for epoch in range(epochs):
        # Shuffle training data
        indices = np.random.permutation(X_train.shape[0])
        X_train_shuffled = X_train.iloc[indices]
        y_train_shuffled = y_train[indices]

        # Mini-batch training
        for i in range(0, X_train.shape[0], batch_size):
            X_batch = X_train_shuffled[i:i+batch_size]
            y_batch = y_train_shuffled[i:i+batch_size]

            # Forward pass
            Z1, A1, Z2, A2, Z3, A3 = forward_pass(X_batch, W1, b1, W2, b2, W3, b3)

            # Backward pass
            dW1, db1, dW2, db2, dW3, db3 = backward_pass(X_batch, y_batch, Z1, Z2, A1, A2,
W2, A3, W3)

            # Update weights
            W1, b1, W2, b2, W3, b3 = sgd_update(W1, b1, W2, b2, W3, b3, dW1, db1, dW2, db2,
dW3, db3, learning_rate)

        # Calculate accuracy for training and test sets
        _, _, _, _, _, train_pred = forward_pass(X_train, W1, b1, W2, b2, W3, b3)
        _, _, _, _, _, test_pred = forward_pass(X_test, W1, b1, W2, b2, W3, b3)

        train_acc = accuracy(y_train, train_pred)
        test_acc = accuracy(y_test, test_pred)

        training_acc.append(train_acc)
        testing_acc.append(test_acc)

        print(f'Epoch {epoch+1}/{epochs} - Training Accuracy: {train_acc:.4f}, Test Accuracy:
{test_acc:.4f}')

    return training_acc, testing_acc
```

```
if __name__ == '__main__':
    X, y = load_fashion_mnist()

    # Split into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # One-hot encode labels
    num_classes = 10
    y_train_one_hot = one_hot_encoding(y_train, num_classes)
    y_test_one_hot = one_hot_encoding(y_test, num_classes)

    # Train MLP
    hidden_size=256
    batch_size=256
    epochs=10
    learning_rate=0.1
    training_acc, testing_acc = train_mlp(X_train, y_train_one_hot, X_test, y_test_one_hot,
hidden_size, batch_size, epochs, learning_rate)

    # Plot accuracy curves
    plt.plot(training_acc, label='Training Accuracy')
    plt.plot(testing_acc, label='Test Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs. Epochs')
    plt.legend()
    plt.savefig('p4_c.jpg')
```

Code for P4-d

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Load Fashion-MNIST dataset
def load_fashion_mnist():
    fashion_mnist = fetch_openml("Fashion-MNIST")
    X = fashion_mnist.data / 255.0  # normalize pixel values
    y = fashion_mnist.target.astype(int)
    return X, y

# One-hot encoding for labels
def one_hot_encoding(y, num_classes):
    encoder = OneHotEncoder(sparse_output=False, categories='auto')
    y_one_hot = encoder.fit_transform(y.to_numpy().reshape(-1, 1))
    return y_one_hot

# ReLU activation
def relu(x):
    return np.maximum(0, x)

# Derivative of ReLU
def relu_derivative(x):
    return np.where(x > 0, 1, 0)

# Softmax function
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

# Cross-entropy loss
def cross_entropy_loss(y_true, y_pred):
    return -np.mean(np.sum(y_true * np.log(y_pred + 1e-8), axis=1))

# Compute accuracy
def accuracy(y_true, y_pred):
    return np.mean(np.argmax(y_true, axis=1) == np.argmax(y_pred, axis=1))

# # Initialize weights and biases
# def initialize_parameters(input_size, hidden_size, output_size):
#     np.random.seed(0)
#     W1 = np.random.randn(input_size, hidden_size) * 0.01
#     b1 = np.zeros((1, hidden_size))
#     W2 = np.random.randn(hidden_size, output_size) * 0.01
#     b2 = np.zeros((1, output_size))
#     return W1, b1, W2, b2
```

```
# Dropout layer function
def apply_dropout(A, dropout_rate):
    mask = np.random.rand(*A.shape) > dropout_rate  # Create mask for dropout
    A_dropout = mask * A / (1 - dropout_rate)  # Scale activations and apply mask
    return A_dropout, mask

# Forward pass
def forward_pass(X, W1, b1, W2, b2, W3, b3, dropout_rate, training):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)

    if training:  # Apply dropout only during training
        A1, mask1 = apply_dropout(A1, dropout_rate)
    else:
        mask1 = None

    Z2 = np.dot(A1, W2) + b2
    A2 = relu(Z2)

    if training:  # Apply dropout only during training
        A2, mask2 = apply_dropout(A2, dropout_rate)
    else:
        mask2 = None

    Z3 = np.dot(A2, W3) + b3
    A3 = softmax(Z3)

    return Z1, A1, Z2, A2, Z3, A3, mask1, mask2

# Backward pass
def backward_pass(X, y_true, Z1, Z2, A1, A2, A3, W2, W3, mask1, mask2, dropout_rate):
    m = X.shape[0]

    # Output layer gradients
    dZ3 = A3 - y_true
    dW3 = np.dot(A2.T, dZ3) / m
    db3 = np.sum(dZ3, axis=0, keepdims=True) / m

    # Second hidden layer gradients
    dA2 = np.dot(dZ3, W3.T)
    if mask2 is not None:
        dA2 = dA2 * mask2 / (1 - dropout_rate)  # Apply mask during backprop

    dZ2 = dA2 * relu_derivative(Z2)
    dW2 = np.dot(A1.T, dZ2) / m
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m

    # First hidden layer gradients
```

```
    dA1 = np.dot(dZ2, W2.T)
    if mask1 is not None:
        dA1 = dA1 * mask1 / (1 - dropout_rate)  # Apply mask during backprop

    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = np.dot(X.T, dZ1) / m
    db1 = np.sum(dZ1, axis=0, keepdims=True) / m

    return dW1, db1, dW2, db2, dW3, db3

# Mini-batch SGD
def sgd_update(W1, b1, W2, b2, W3, b3, dW1, db1, dW2, db2, dW3, db3, learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    W3 -= learning_rate * dW3
    b3 -= learning_rate * db3
    return W1, b1, W2, b2, W3, b3

# Train MLP
def train_mlp(X_train, y_train, X_test, y_test, hidden_size, batch_size, epochs, learning_rate,
dropout_rate):
    input_size = X_train.shape[1]
    output_size = y_train.shape[1]

    # Initialize weights and biases
    np.random.seed(0)
    W1 = np.random.randn(input_size, hidden_size) * 0.01
    b1 = np.zeros((1, hidden_size))
    # W2 = np.random.randn(hidden_size, output_size) * 0.01
    # b2 = np.zeros((1, output_size))
    W2 = np.random.randn(hidden_size,hidden_size) * 0.01
    b2 = b1 = np.zeros((1, hidden_size))
    W3 = np.random.randn(hidden_size, output_size) * 0.01
    b3 = np.zeros((1, output_size))

    training_acc = []
    testing_acc = []

    for epoch in range(epochs):
        # Shuffle training data
        indices = np.random.permutation(X_train.shape[0])
        X_train_shuffled = X_train.iloc[indices]
        y_train_shuffled = y_train[indices]

        # Mini-batch training
        for i in range(0, X_train.shape[0], batch_size):
            X_batch = X_train_shuffled[i:i+batch_size]
```

```
            y_batch = y_train_shuffled[i:i+batch_size]

            # Forward pass
            Z1, A1, Z2, A2, Z3, A3, mask1, mask2 = forward_pass(X_batch, W1, b1, W2, b2, W3, b3,
dropout_rate, training=True)

            # Backward pass
            dW1, db1, dW2, db2, dW3, db3 = backward_pass(X_batch, y_batch, Z1, Z2, A1, A2, A3,
W2, W3, mask1, mask2, dropout_rate)

            # Update weights
            W1, b1, W2, b2, W3, b3 = sgd_update(W1, b1, W2, b2, W3, b3, dW1, db1, dW2, db2,
dW3, db3, learning_rate)

        # Calculate accuracy for training and test sets
        _, _, _, _, _, train_pred, _, _ = forward_pass(X_train, W1, b1, W2, b2, W3, b3, dropout_rate,
training=False)
        _, _, _, _, _, test_pred, _, _ = forward_pass(X_test, W1, b1, W2, b2, W3, b3, dropout_rate,
training=False)

        train_acc = accuracy(y_train, train_pred)
        test_acc = accuracy(y_test, test_pred)

        training_acc.append(train_acc)
        testing_acc.append(test_acc)

        print(f'Epoch {epoch+1}/{epochs} - Training Accuracy: {train_acc:.4f}, Test Accuracy:
{test_acc:.4f}')

    return training_acc, testing_acc


if __name__ == '__main__':
    X, y = load_fashion_mnist()

    # Split into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # One-hot encode labels
    num_classes = 10
    y_train_one_hot = one_hot_encoding(y_train, num_classes)
    y_test_one_hot = one_hot_encoding(y_test, num_classes)

    # Train MLP
    hidden_size=256
    batch_size=256
    epochs=10
    learning_rate=0.1
    dropout_rate=0.5
```

```
    training_acc, testing_acc = train_mlp(X_train, y_train_one_hot, X_test, y_test_one_hot,
hidden_size, batch_size, epochs, learning_rate, dropout_rate)

    # Plot accuracy curves
    plt.plot(training_acc, label='Training Accuracy')
    plt.plot(testing_acc, label='Test Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs. Epochs')
    plt.legend()
    plt.savefig('p4_d.jpg')
```