

1 Question 1

1.1

To derive the bias-variance decomposition for a regression problem, we start by defining the expected mean squared error (MSE) of a model. The MSE can be expressed as follows:

$$\text{MSE}(\hat{f}(x)) = \mathbb{E} \left[(y - \hat{f}(x))^2 \right]$$

Where: - y is the true output. - $\hat{f}(x)$ is the predicted output of our model for input x .

Now, we can express the true output y in terms of the underlying function $f(x)$ and some noise ϵ :

$$y = f(x) + \epsilon$$

Where ϵ is the noise, which has an expected value of 0, i.e., $\mathbb{E}[\epsilon] = 0$.

Substituting this into our MSE expression gives us:

$$\begin{aligned} \text{MSE}(\hat{f}(x)) &= \mathbb{E} \left[(f(x) + \epsilon - \hat{f}(x))^2 \right] \\ &= \mathbb{E} \left[(f(x) - \hat{f}(x) + \epsilon)^2 \right] \\ &= \mathbb{E} \left[(f(x) - \hat{f}(x))^2 \right] + 2\mathbb{E} \left[(f(x) - \hat{f}(x))\epsilon \right] + \mathbb{E} [\epsilon^2] \end{aligned}$$

Since the noise ϵ is independent of $f(x)$ and has an expected value of zero, the middle term vanishes:

$$\mathbb{E} \left[(f(x) - \hat{f}(x))\epsilon \right] = 0$$

Now we can write the MSE as:

$$\text{MSE}(\hat{f}(x)) = \mathbb{E} \left[(f(x) - \hat{f}(x))^2 \right] + \mathbb{E} [\epsilon^2]$$

We denote:

$$\text{Bias}(\hat{f}(x)) = \mathbb{E}[\hat{f}(x)] - f(x)$$

$$\text{Var}(\hat{f}(x)) = \mathbb{E} \left[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2 \right]$$

Using these definitions, we can express the expected squared error in terms of bias and variance:

$$\mathbb{E}[(f(x) - \hat{f}(x))^2] = \text{Bias}^2(\hat{f}(x)) + \text{Var}(\hat{f}(x))$$

Thus, we have:

$$\text{MSE}(\hat{f}(x)) = \text{Bias}^2(\hat{f}(x)) + \text{Var}(\hat{f}(x)) + \mathbb{E}[\epsilon^2]$$

If we denote $\mathbb{E}[\epsilon^2]$ as the noise term σ^2 , we obtain the final form of the bias-variance decomposition:

$$\mathbb{E}[\text{MSE}] = \text{Bias}^2 + \text{Variance} + \sigma^2$$

This completes the derivation of the bias-variance decomposition for a regression problem.

1.2

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

np.random.seed(42) # For reproducibility
x_values = np.random.uniform(-10, 10, 20)

# Compute f(x) = x + sin(1.5x)
def f(x):
    return x + np.sin(1.5 * x)

# Generate noise from a normal distribution N(0, 0.3)
noise = np.random.normal(0, np.sqrt(0.3), 20)

y_values = f(x_values) + noise

plt.scatter(x_values, y_values, color='blue', label='y(x) with noise', zorder=2)

x_smooth = np.linspace(-10, 10, 400)
f_smooth = f(x_smooth)
plt.plot(x_smooth, f_smooth, color='red', label='f(x) without noise', zorder=1)

plt.title('Scatter Plot of y(x) and Line Plot of f(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

1.3

```
# Fit degree 1 polynomial
g1_coefficients = np.polyfit(x_values, y_values, 1)
```

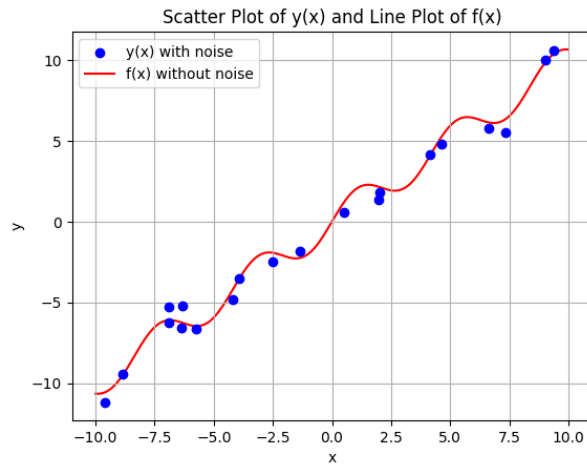


Figure 1: Scatter plot of noisy data and the true function

```
g1 = np.poly1d(g1_coefficients)

# Fit degree 3 polynomial
g3_coefficients = np.polyfit(x_values, y_values, 3)
g3 = np.poly1d(g3_coefficients)

# Fit degree 10 polynomial
g10_coefficients = np.polyfit(x_values, y_values, 10)
g10 = np.poly1d(g10_coefficients)

print("Degree 1 coefficients: ", g1_coefficients)
print("Degree 3 coefficients: ", g3_coefficients)
print("Degree 10 coefficients: ", g10_coefficients)

# Step 4: Plot the results
plt.figure(figsize=(5, 3))
plt.title('Polynomial Estimators for g1')
plt.scatter(x_values, y_values, color='blue', label='y(x) with noise', zorder=2)
plt.plot(x_smooth, g1(x_smooth), color='green', linestyle='--', label='g1(x) - Degree 1',
         , linewidth=2)
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(5, 3))
plt.title('Polynomial Estimators for g3')
plt.scatter(x_values, y_values, color='blue', label='y(x) with noise', zorder=2)
plt.plot(x_smooth, g3(x_smooth), color='orange', linestyle='--', label='g3(x) - Degree 3',
         , linewidth=2)
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(5, 3))
plt.title('Polynomial Estimators for g10')
plt.scatter(x_values, y_values, color='blue', label='y(x) with noise', zorder=2)
plt.plot(x_smooth, g10(x_smooth), color='purple', linestyle='--', label='g10(x) - Degree 10', linewidth=2)
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

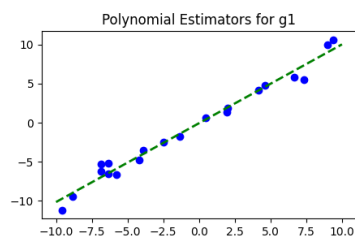


Figure 2: g_1 (Degree 1)

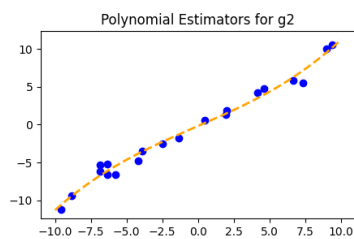


Figure 3: g_3 (Degree 3)

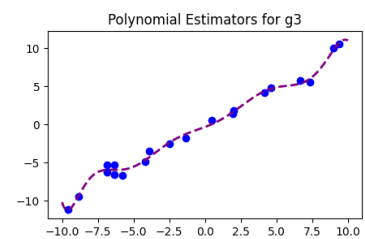


Figure 4: g_{10} (Degree 10)

Answer: g_1 seems to be underfitting and g_{10} seems to be overfitting.

1.4

```
from sklearn.metrics import mean_squared_error, r2_score
from mlxtend.evaluate import bias_variance_decomp
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Generate data
x = 20 * np.random.random(50)
x = np.sort(x)
f = lambda x: x + np.sin(1.5 * x)
datasets = []
degrees = range(1, 16)

# Create 100 datasets
for _ in range(100):
    y = f(x) + np.random.normal(0, 0.3, 50)
    datasets.append(y)

x_resaped = x.reshape(-1, 1)
datasets_array = np.array(datasets)
```

```

mse_list = []
bias_list = []
variance_list = []

# Split into train/test in 80/20 ratio
for dataset_index in range(100):
    X_train, X_test, y_train, y_test = train_test_split(x_reshaped, datasets_array[
        dataset_index], test_size=0.20, random_state=0)

    for degree in degrees:
        polynomial_features = PolynomialFeatures(degree=degree)
        x_poly = polynomial_features.fit_transform(x_reshaped)
        model = LinearRegression()
        model.fit(X_train, y_train)

        mse, bias, var = bias_variance_decomp(model, X_train, y_train.flatten(), X_test,
            y_test.flatten(), loss='mse', num_rounds=50)
        mse_list.append(mse)
        bias_list.append(bias**2)
        variance_list.append(var)

# Initialize lists to store averaged metrics
mse_averaged = []
bias_averaged = []
variance_averaged = []

# Set the range for averaging
start_index = 0
end_index = 100

# Averaging the metrics for different model complexities
while end_index <= len(mse_list):
    mse_averaged.append(np.mean(mse_list[start_index:end_index]))
    bias_averaged.append(np.mean(bias_list[start_index:end_index])) # Use
        bias_squared_list for bias
    variance_averaged.append(np.mean(variance_list[start_index:end_index]))

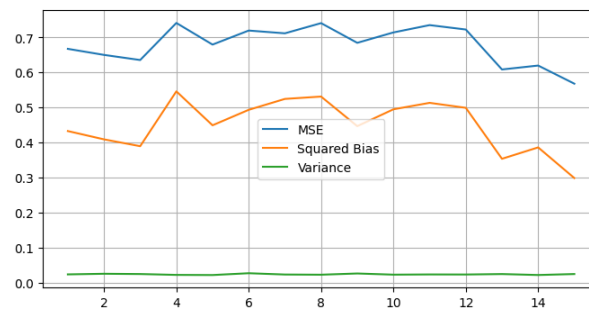
    # Move to the next segment of 100
    start_index += 100
    end_index += 100

complexity=range(1,16)

#Plotting
fig = plt.figure(figsize=(8,4))
plt.plot(complexity,mse_averaged,label='MSE')
plt.plot(complexity,bias_averaged,label='Squared Bias')
plt.plot(complexity,variance_averaged, label='Variance')

```

```
plt.grid()
plt.legend()
```



Answer: A good model typically has low bias and low variance, which results in the lowest error.

1.5

```
from sklearn.model_selection import train_test_split, cross_val_score
from statistics import mean
from sklearn.linear_model import Ridge

x = np.random.random(50)
fx = x + np.sin(1.5*x)
e = np.random.normal(0, 0.3, 50)
y = fx + e
x_resaped = x.reshape(-1,1)

X_train, X_test, y_train, y_test = train_test_split(x_resaped, y, test_size=0.20)
poly = PolynomialFeatures(degree=10)
x_poly = poly.fit_transform(x_resaped)
linear_regression = LinearRegression()
linear_regression.fit(x_resaped, y)

rp = linear_regression.predict(X_test)

mse, bias, var = bias_variance_decomp(linear_regression, X_train, y_train, X_test,
    y_test, loss = 'mse', num_rounds=200, random_seed=1)

print('MSE: %.4f' % mse)
print('Bias: %.4f' % bias)
print('Variance: %.4f' % var)
print('\n')

cross_val_scores_ridge = []
alpha = []

for i in range(1, 9):
    ridgeModel = Ridge(alpha = i * 0.25)
```

```

ridgeModel.fit(X_train, y_train)
scores = cross_val_score(ridgeModel, x_rs, y, cv = 10)
avg_cross_val_score = mean(scores)*100
cross_val_scores_ridge.append(avg_cross_val_score)
alpha.append(i * 0.25)

ridgeModelChosen = Ridge(alpha = 2)
ridgeModelChosen.fit(X_train, y_train)

rp = ridgeModelChosen.predict(X_test)
mse2, bias2, var2 = bias_variance_decomp(ridgeModel, X_train, y_train, X_test, y_test,
    loss='mse', num_rounds=200, random_seed=1)
# summarize results
print('MSE Regularized: %.3f' % mse2)
print('Bias Regularized: %.3f' % bias2)
print('Variance Regularized: %.3f' % var2)

```

Table 1: Comparison of MSE, Bias, and Variance

Metric	Unregularized	Regularized
MSE	0.0723	0.117
Bias	0.0681	0.113
Variance	0.0042	0.004

2 Question 2

2.1

True Negatives matter only for ROC curves as ROC curves use False Positive Rate to calculate the curve. The False positive rate = $FP / (FP + TN)$. PR curve on the other hand uses only Precision and Recall to calculate PR and neither of those use FP.

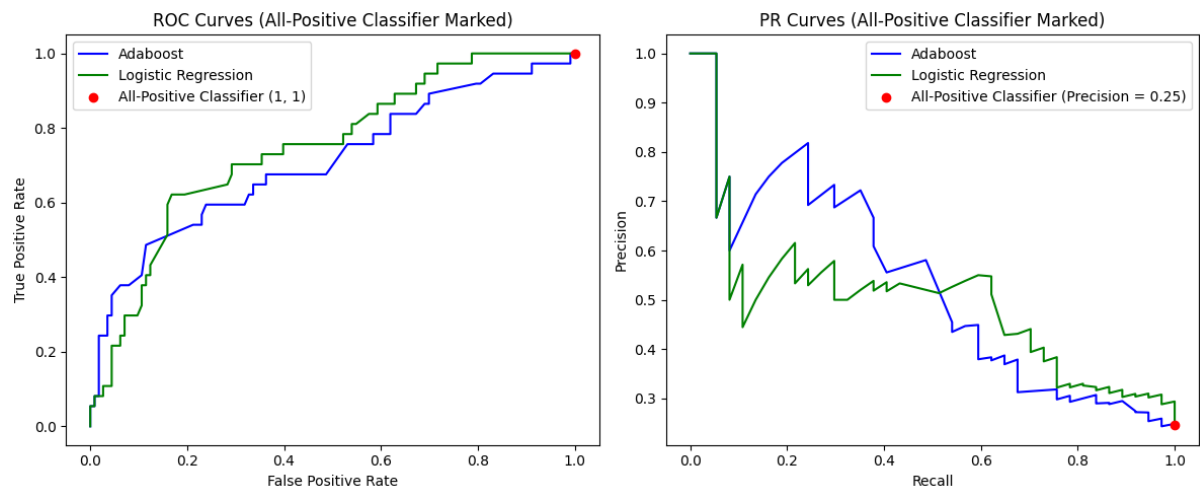
The paper argues that as the ROC space defines a matrix where the data set is fixed. And if we have a fixed number of positive and negative examples, the false negatives have also to be fixed/uniquely determined. Further if recall is 0, we are unable to recover FP and thus cannot find a unique confusion matrix. Consequently, we have a one-to-one mapping between confusion matrices and points in PR space. This implies that we also have a one-to-one mapping between points (each defined by a confusion matrix) in ROC space and PR space; hence, we can translate a curve in ROC space to PR space and vice-versa.

```

# Q2

import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, precision_recall_curve
from sklearn.metrics import roc_auc_score, auc, confusion_matrix

```



```

from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import fetch_openml

dataset = fetch_openml(data_id=1464)
X = dataset.data
y = dataset.target.astype(int) - 1 # Adjust target labels from {1, 2} to {0, 1}

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
    =42)

# Train two classifiers
adaboost = AdaBoostClassifier(n_estimators=100, random_state=42)
log_reg = LogisticRegression(max_iter=1000, random_state=42)

adaboost.fit(X_train, y_train)
log_reg.fit(X_train, y_train)

# Get prediction probabilities
y_scores_adaboost = adaboost.predict_proba(X_test)[: , 1]
y_scores_log_reg = log_reg.predict_proba(X_test)[: , 1]

# Calculate ROC and PR curves for both classifiers
fpr_adaboost, tpr_adaboost, _ = roc_curve(y_test, y_scores_adaboost)
fpr_log_reg, tpr_log_reg, _ = roc_curve(y_test, y_scores_log_reg)

precision_adaboost, recall_adaboost, _ = precision_recall_curve(y_test,
    y_scores_adaboost)
precision_log_reg, recall_log_reg, _ = precision_recall_curve(y_test, y_scores_log_reg)

# Find and mark the point for an all-positive classifier
positive_rate = np.sum(y_test) / len(y_test)
print(f"Positive Rate (All Positive Classifier Precision): {positive_rate}")

```



```

# Add all-positive classifier points to the plots
plt.figure(figsize=(12, 5))

# For an all-positive classifier:
# TPR = 1 (because all positives are correctly identified).
# FPR = 1 (because all negatives are incorrectly classified as positives).
# Hence, the all-positive classifier is represented by the point (1, 1) on the ROC curve
.

# Recall is always 1.
# Precision is equal to the positive rate of the dataset (since it predicts all samples
    as positive).
# Therefore, the point for an all-positive classifier on the PR curve is (1, positive
    rate

# ROC Curve with all-positive classifier point
plt.subplot(1, 2, 1)
plt.plot(fpr_adaboost, tpr_adaboost, label="Adaboost", color='blue')
plt.plot(fpr_log_reg, tpr_log_reg, label="Logistic Regression", color='green')
plt.scatter(1, 1, color='red', label="All-Positive Classifier (1, 1)", zorder=5)
plt.title("ROC Curves (All-Positive Classifier Marked)")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend()

# PR Curve with all-positive classifier point
plt.subplot(1, 2, 2)
plt.plot(recall_adaboost, precision_adaboost, label="Adaboost", color='blue')
plt.plot(recall_log_reg, precision_log_reg, label="Logistic Regression", color='green')
plt.scatter(1, positive_rate, color='red', label=f"All-Positive Classifier (Precision =
    {positive_rate:.2f})", zorder=5)
plt.title("PR Curves (All-Positive Classifier Marked)")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.legend()

plt.tight_layout()
plt.show()

```

2.2

Table 2: Performance Metrics for Adaboost and Logistic Regression

Metric	Adaboost	Logistic Regression
AUROC	0.7090	0.7481
AUPR	0.5265	0.4946
AUPRG	∞	∞

```

# Calculate AUROC
auroc_adaboost = roc_auc_score(y_test, y_scores_adaboost)
auroc_log_reg = roc_auc_score(y_test, y_scores_log_reg)

# Calculate AUPR
aupr_adaboost = auc(recall_adaboost, precision_adaboost)
aupr_log_reg = auc(recall_log_reg, precision_log_reg)

## AdaBoost (PRG)

precision_gain_adaboost = (precision_adaboost - np.pi)/((1-np.pi)*precision_adaboost)
recall_gain_adaboost = (recall_adaboost - np.pi)/((1-np.pi)*recall_adaboost)

#Logistic Regression (PRG)

precision_gain_log_reg = (precision_log_reg - np.pi)/((1-np.pi)*precision_log_reg)
recall_gain_log_reg = (recall_log_reg - np.pi)/((1-np.pi)*recall_log_reg)

#Plotting PRG
plt.title('PR Gain Curve')
plt.xlabel('Recall Gain')
plt.ylabel('Precision Gain')
plt.plot(recall_gain_adaboost, precision_gain_adaboost, label='AdaBoost')
plt.plot(recall_gain_log_reg, precision_gain_log_reg, label='LogisticRegression')
plt.grid()

#AUPRG
auprg_adaboost = auc(recall_gain_adaboost, precision_gain_adaboost)
auprg_log_reg = auc(recall_gain_log_reg, precision_gain_log_reg)

# Display Results
print(f"AUROC for Adaboost: {auroc_adaboost:.4f}")
print(f"AUROC for Logistic Regression: {auroc_log_reg:.4f}")
print(f"AUPR for Adaboost: {aupr_adaboost:.4f}")
print(f"AUPR for Logistic Regression: {aupr_log_reg:.4f}")
print(f"AUPRG for Adaboost: {auprg_adaboost:.4f}")
print(f"AUPRG for Logistic Regression: {auprg_log_reg:.4f}")

```

3 Question3

3.1 a)

```

import torch
import torch.nn as nn

```

```

import torch.optim as optim
import torchvision.transforms as transforms
from torchvision.datasets import FashionMNIST
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Define transformations
transform = transforms.Compose([transforms.ToTensor()])

# Load datasets
train_dataset = FashionMNIST(root='./data', train=True, download=True, transform=
    transform)
test_dataset = FashionMNIST(root='./data', train=False, download=True, transform=
    transform)

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

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

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

class ConvModule(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=0):
        super(ConvModule, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        return self.relu(x)

class InceptionModule(nn.Module):
    def __init__(self, in_channels, ch1, ch3):
        super(InceptionModule, self).__init__()
        # 1x1 branch
        self.branch1x1 = ConvModule(in_channels, ch1, kernel_size=1)

        # 3x3 branch
        self.branch3x3 = ConvModule(in_channels, ch3, kernel_size=3, padding=1)

```

```

def forward(self, x):
    out1x1 = self.branch1x1(x)
    out3x3 = self.branch3x3(x)
    # Concatenate along the channel dimension
    return torch.cat([out1x1, out3x3], dim=1)

class DownsampleModule(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownsampleModule, self).__init__()
        # 3x3 conv branch with stride 2
        self.conv = ConvModule(in_channels, out_channels, kernel_size=3, stride=2,
                                padding=1)

        # Max pooling branch
        self.pool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

    def forward(self, x):
        out_conv = self.conv(x)
        out_pool = self.pool(x)
        # Concatenate along the channel dimension
        return torch.cat([out_conv, out_pool], dim=1)

class SmallInception(nn.Module):
    def __init__(self, num_classes=10):
        super(SmallInception, self).__init__()

        # Initial conv layer: Adjusted input channels to 1 for FashionMNIST (grayscale
        # input)
        self.conv1 = ConvModule(1, 96, kernel_size=3, stride=1, padding=1)

        # First Inception Block
        self.inception1 = InceptionModule(96, ch1=32, ch3=32) # Output: 32+32=64
        # channels
        self.inception2 = InceptionModule(64, ch1=32, ch3=48) # Output: 32+48=80
        # channels

        # Downsampling
        self.downsample1 = DownsampleModule(80, 80)

        # Second Inception Block
        self.inception3 = InceptionModule(160, ch1=112, ch3=48) # Output: 112+48=160
        # channels
        self.inception4 = InceptionModule(160, ch1=96, ch3=64) # Output: 96+64=160
        # channels
        self.inception5 = InceptionModule(160, ch1=80, ch3=80) # Output: 80+80=160
        # channels
        self.inception6 = InceptionModule(160, ch1=48, ch3=96) # Output: 48+96=144
        # channels

```

```

# Downsampling
self.downsample2 = DownsampleModule(144, 48)

# Final Inception Block
self.inception7 = InceptionModule(192, ch1=176, ch3=160) # Output: 176+160=336
channels
self.inception8 = InceptionModule(336, ch1=176, ch3=160) # Output: 176+160=336
channels

# Global average pooling
self.global_pool = nn.AvgPool2d(kernel_size=7)

# Fully connected layer for classification
self.fc = nn.Linear(336, num_classes)

def forward(self, x):
    x = self.conv1(x)
    x = self.inception1(x)
    x = self.inception2(x)
    x = self.downsample1(x)
    x = self.inception3(x)
    x = self.inception4(x)
    x = self.inception5(x)
    x = self.inception6(x)
    x = self.downsample2(x)
    x = self.inception7(x)
    x = self.inception8(x)
    x = self.global_pool(x)
    x = torch.flatten(x, 1)
    return self.fc(x)

# Assuming you have the SmallInception model and train_loader defined
model = SmallInception()
criterion = nn.CrossEntropyLoss()

# Change optimizer to SGD with momentum (optional)
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9) # You can adjust the
lr and momentum

lrs = []
losses = []

# Testing different learning rates
for lr in torch.logspace(-9, 1, 10):
    optimizer.param_groups[0]['lr'] = lr
    running_loss = 0.0

```

```

print("Running for lr ", lr)
for epoch in range(5):
    print("Running epoch ", epoch)
    for i, (images, labels) in enumerate(train_loader):

        if i > 10: # Limiting to 10 batches per epoch for speedup
            break
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

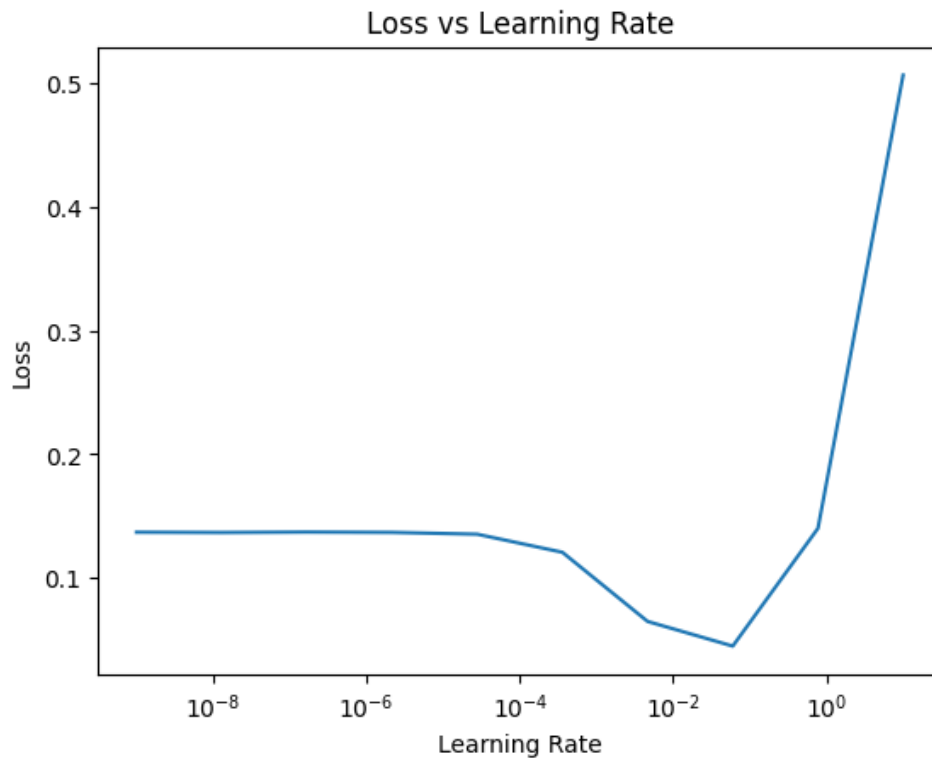
    lrs.append(lr)
    losses.append(running_loss / len(train_loader))

plt.plot(lrs, losses)
plt.xscale('log')
plt.xlabel('Learning Rate')
plt.ylabel('Loss')
plt.title('Loss vs Learning Rate')
plt.show()

# Plot the results
plt.plot(lrs, losses)
plt.xscale('log')
plt.xlabel('Learning Rate')
plt.ylabel('Loss')
plt.title('Loss vs Learning Rate')
plt.show()

```

The loss vs learning curve looks like: The lr-min and lr-max can be **[0.01, 1]**



3.2 b)

```
import torch
import torch.optim as optim
from torch.optim.lr_scheduler import CyclicalLR
import matplotlib.pyplot as plt

# Assuming model is already defined and instantiated
model = SmallInception() # Using the SmallInception model

# Criterion (Loss Function) and Optimizer
criterion = torch.nn.CrossEntropyLoss()

# Initializing the optimizer
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

# Learning rate range obtained from part 1
lrmin = 1e-2 # You should replace this with the lrmin found in part 1
lrmax = 1 # You should replace this with the lrmax found in part 1

# Cyclical Learning Rate Scheduler
scheduler = CyclicalLR(optimizer, base_lr=lrmin, max_lr=lrmax, step_size_up=2000, mode='
    exp_range', gamma=0.99)

# Containers to store metrics
train_losses, val_losses = [], []
train_accuracies, val_accuracies = [], []
```

```

# Training Loop
num_epochs = 30
for epoch in range(num_epochs):
    model.train()
    running_loss, correct = 0.0, 0
    total = 0

    for i, (images, labels) in enumerate(train_loader):
        optimizer.zero_grad()
        if i > 10: # Limiting to 10 batches per epoch for speedup
            break

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()

        # Update weights
        optimizer.step()
        scheduler.step() # Update the learning rate per batch

        # Track training loss and accuracy
        running_loss += loss.item()
        _, predicted = outputs.max(1)
        correct += predicted.eq(labels).sum().item()
        total += labels.size(0)

    train_loss = running_loss / len(train_loader)
    train_acc = correct / total
    train_losses.append(train_loss)
    train_accuracies.append(train_acc)

# Validation Phase
model.eval()
val_loss, correct = 0.0, 0
total = 0

with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)
        val_loss += loss.item()
        _, predicted = outputs.max(1)
        correct += predicted.eq(labels).sum().item()
        total += labels.size(0)

val_loss = val_loss / len(test_loader)
val_acc = correct / total

```



```

val_losses.append(val_loss)
val_accuracies.append(val_acc)

# Print metrics
print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Train Accuracy: {train_acc:.4f}, '
      f'Val Loss: {val_loss:.4f}, Val Accuracy: {val_acc:.4f}')

# Decay the lrmin and lrmax values exponentially after each epoch
lrmin *= 0.99 # Adjust decay factor as needed
lrmax *= 0.99
scheduler.base_lrs = [lrmin] # Update the scheduler's learning rate range

```



3.3 c)

```

# Define transformations
transform = transforms.Compose([transforms.ToTensor()])

# Load datasets
train_dataset = FashionMNIST(root='./data', train=True, download=True, transform=
    transform)
test_dataset = FashionMNIST(root='./data', train=False, download=True, transform=
    transform)

# Learning Rate (Set this to lrmax found in part 1)
lrmax = 1 # Replace with your lrmax value

# Batch sizes to test
batch_sizes = [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]

# Store losses for each batch size
batch_loss = {}

# Training Loop for each batch size

```

```

for batch_size in batch_sizes:
    print(f"Training with batch size: {batch_size}")

    # Create DataLoader with the current batch size
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    # Define model, criterion and optimizer
    model = SmallInception() # Using the SmallInception model
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lrmax, momentum=0.9)

    # Store training loss
    running_loss = []
    num_epochs = 5 # You can adjust this based on your requirement

    for epoch in range(num_epochs):
        model.train()
        epoch_loss = 0.0

        for i, (images, labels) in enumerate(train_loader):
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()

        # Average loss for the epoch
        running_loss.append(epoch_loss / len(train_loader))
        print(f'Epoch {epoch+1}, Loss: {running_loss[-1]:.4f}')

    # Store the final loss for the current batch size
    batch_loss[batch_size] = running_loss[-1]

# Plotting the training loss for each batch size
plt.figure(figsize=(10, 6))
plt.plot(batch_sizes, list(batch_loss.values()), marker='o')
plt.xscale('log')
plt.title('Training Loss vs Batch Size')
plt.xlabel('Batch Size (log scale)')
plt.ylabel('Training Loss')
plt.xticks(batch_sizes, batch_sizes) # Show actual batch sizes on x-axis
plt.grid()
plt.show()

```

4 Question4

4.1

- **CONV3-128:**

Memory: $112 \times 112 \times 128 = 1,605,632$

Params: $3 \times 3 \times 64 \times 128 = 73,728$

- **CONV3-128:**

Memory: $112 \times 112 \times 128 = 1,605,632$

Params: $3 \times 3 \times 128 \times 128 = 147,456$

- **CONV3-256:**

Memory: $56 \times 56 \times 256 = 802,816$

Params: $3 \times 3 \times 128 \times 256 = 294,912$

- **CONV3-256:**

Memory: $56 \times 56 \times 256 = 802,816$

Params: $3 \times 3 \times 256 \times 256 = 589,824$

- **CONV3-256:**

Memory: $56 \times 112 \times 256 = 802,816$

Params: $3 \times 3 \times 256 \times 256 = 589,824$

- **POOL2 Memory:**

Memory: $28 \times 28 \times 256 = 200,704$

- **CONV3-512:**

Memory: $28 \times 28 \times 512 = 401,408$

Params: $3 \times 3 \times 256 \times 512 = 1,179,648$

- **CONV3-512:**

Memory: $28 \times 28 \times 512 = 401,408$

Params: $3 \times 3 \times 512 \times 512 = 2,359,296$

- **CONV3-512:**

Memory: $28 \times 28 \times 512 = 401,408$

Params: $3 \times 3 \times 512 \times 512 = 2,359,296$

- **POOL2 Memory:**

Memory: $14 \times 14 \times 512 = 100,352$

- **CONV3-512:**

Memory: $14 \times 14 \times 512 = 100,352$

Params: $3 \times 3 \times 512 \times 512 = 2,359,296$

- **CONV3-512:**

Memory: $14 \times 14 \times 512 = 100,352$

Params: $3 \times 3 \times 512 \times 512 = 2,359,296$

- **CONV3-512:**

Memory: $14 \times 14 \times 512 = 100,352$

Params: $3 \times 3 \times 512 \times 512 = 2,359,296$

- **POOL2 Memory:**

Memory: $7 \times 7 \times 512 = 25,088$

- **FC:**

Memory: 4096

Params: $4096 \times 7 \times 7 \times 512 = 102,760,448$

- **FC:**

Memory: 4096

Params: $4096 \times 4096 = 4,096,000$

- **Total Memory:**

Approximately 15M

Total Parameters: Approximately 138.3M

4.1.1 a)

The main idea behind the inception module is to boost the network's capabilities without making it more computationally heavy by just increasing its size. It does this by using matrix multiplication routines, turning sparse matrices from the kernels into a denser format. This approach helps the model learn complex patterns in the data more efficiently while keeping resource use in check.

4.1.2 b)

- Naive: $32 \times 32 \times (128 + 192 + 96 + 256) = 32 \times 32 \times 672$
- Dimensionality Reduction: $32 \times 32 \times (128 + 192 + 96 + 64) = 32 \times 32 \times 480$

4.1.3 c)

The number of operations is calculated as $(K \times K \times H \times W \times C_{in} \times C_{out})$

Naive

$$\text{CONV1} = 32 \times 32 \times 1 \times 128 \times 256 = 33,554,432 \quad (1)$$

$$\text{CONV3} = 32 \times 32 \times 9 \times 192 \times 256 = 452984832 \quad (2)$$

$$\text{CONV5} = 32 \times 32 \times 25 \times 96 \times 256 = 629145600 \quad (3)$$

$$\text{Total} = 1,115,684,864 \quad (4)$$

Dimensionality Reduction (DR)

$$\text{CONV1} = (1 \times 1 \times 32 \times 32 \times 256 \times 128) + (32 \times 32 \times 256 \times 128) + (32 \times 32 \times 256 \times 64) = 92,274,688 \quad (5)$$

$$\text{CONV3} = 32 \times 32 \times 9 \times 128 \times 192 = 226,492,416 \quad (6)$$

$$\text{CONV5} = 32 \times 32 \times 25 \times 32 \times 96 = 78,643,200 \quad (7)$$

$$\text{Total} = 397,410,304 \quad (8)$$

4.1.4 d)

The naive architecture is 2.96 times more expensive than Dimensionality Reduction. By having smaller outputs for every inception module, DR is less computationally intensive.

5 Question5

$$g[L1, 1] = 0 \quad g[L1, 2] = 0 \quad g[L1, 3] = 1 \quad g[L1, 4] = 0 \quad g[L2, 1] = 2 \quad g[L2, 2] = 2$$