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:

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

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

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

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:

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

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

```python
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

```python
# Fit degree 1 polynomial
g1_coefficients = np.polyfit(x_values, y_values, 1)
g1 = np.poly1d(g1_coefficients)

# Fit degree 3 polynomial
```

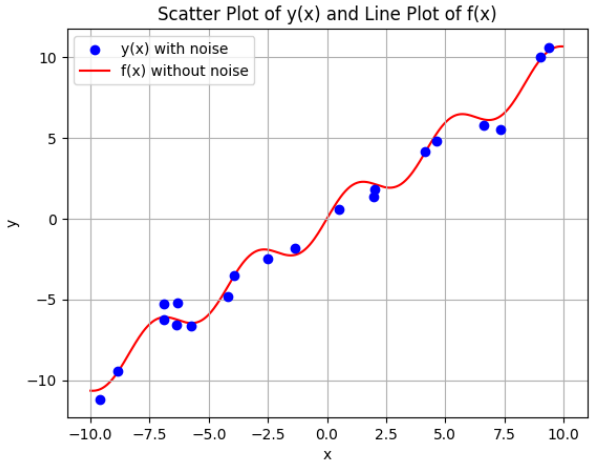


Figure 1: Scatter plot of noisy data and the true function

```
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()
```

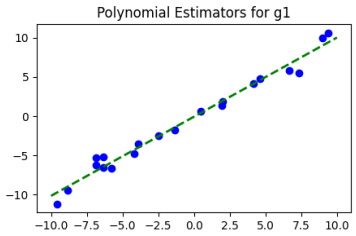


Figure 2: g1 (Degree 1)

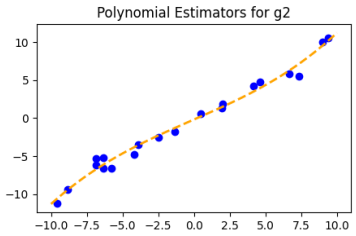


Figure 3: g3 (Degree 3)

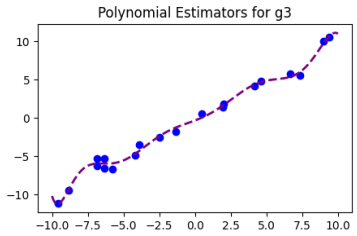


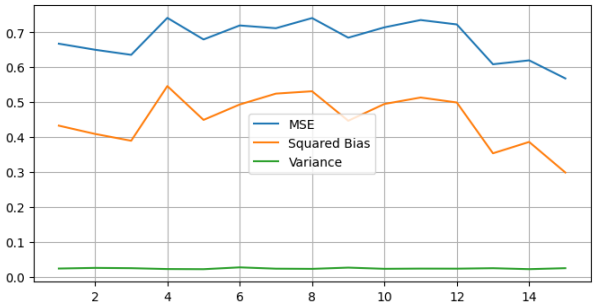
Figure 4: g10 (Degree 10)

Answer: $g1$ seems to be underfitting and $g10$ 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_reshaped = 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_reshaped = x.reshape(-1,1)

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

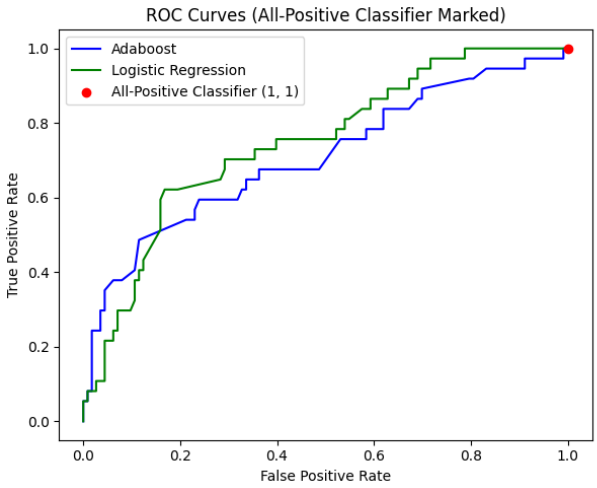
2 Question 2

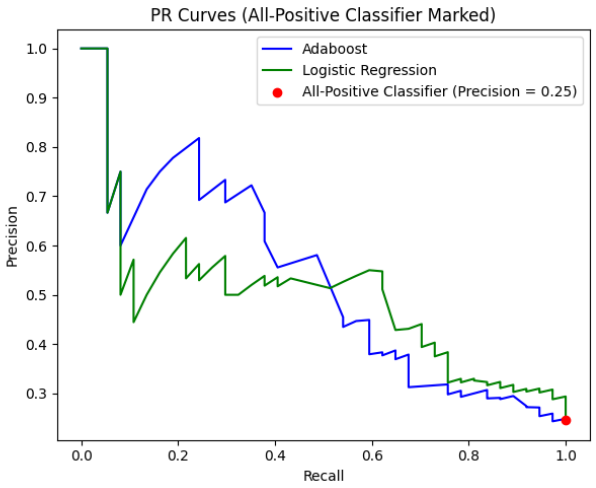
2.1

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

Unable to find out the graph as the model is not training on Google Colab, it keeps crashing. I will update the Github Repo if I am able to answer this question later.

Attaching the code for now.

```
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

class ConvBlock(nn.Module):
    def __init__(self, cin, cout, filt_size, strd, pad):
        super(ConvBlock, self).__init__()
        self.conv2d = nn.Conv2d(in_channels=cin, out_channels=cout, kernel_size=
            filt_size, stride=strd, padding=pad)
        self.batch_norm = nn.BatchNorm2d(num_features=cout)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.relu(self.batch_norm(self.conv2d(x)))

class InceptionBlock(nn.Module):
    def __init__(self, cin, cout1, cout3):
        super(InceptionBlock, self).__init__()
        self.branch1 = ConvBlock(cin, cout1, filt_size=1, strd=1, pad=0)
        self.branch2 = ConvBlock(cin, cout3, filt_size=3, strd=1, pad=1)

    def forward(self, x):
        branches = (self.branch1, self.branch2)
        return torch.cat([branch(x) for branch in branches], 1)

class DownsampleBlock(nn.Module):
    def __init__(self, cin, cout3):
        super(DownsampleBlock, self).__init__()
        self.branch1 = ConvBlock(cin, cout3, filt_size=3, strd=2, pad=0)
        self.branch2 = nn.MaxPool2d(kernel_size=3, stride=2)

    def forward(self, x):
        branches = (self.branch1, self.branch2)
        return torch.cat([branch(x) for branch in branches], 1)

class SmallInception(nn.Module):
    def __init__(self, num_classes=10):
        super(SmallInception, self).__init__()
        self.conv1 = ConvBlock(cin=1, cout=96, filt_size=3, strd=1, pad=0)
        self.inception1a = InceptionBlock(cin=96, cout1=32, cout3=32)
        self.inception1b = InceptionBlock(cin=64, cout1=32, cout3=48)
        self.downsample1 = DownsampleBlock(cin=80, cout3=80)
        self.inception2a = InceptionBlock(cin=160, cout1=112, cout3=48)
        self.inception2b = InceptionBlock(cin=160, cout1=96, cout3=64)
        self.inception2c = InceptionBlock(cin=160, cout1=80, cout3=80)
        self.inception2d = InceptionBlock(cin=160, cout1=48, cout3=96)
        self.downsample2 = DownsampleBlock(cin=144, cout3=96)
        self.inception3a = InceptionBlock(cin=240, cout1=176, cout3=160)
        self.inception3b = InceptionBlock(cin=336, cout1=176, cout3=160)
```

```
        self.avgpool = nn.AvgPool2d(kernel_size=7, padding=1)
        self.fully_connected = nn.Linear(336, num_classes)
        self.dropout = nn.Dropout(0.9)
        self.fully_relu = nn.ReLU()

    def forward(self, x):
        x = self.conv1(x)
        x = self.inception1a(x)
        x = self.inception1b(x)
        x = self.downsample1(x)
        x = self.inception2a(x)
        x = self.inception2b(x)
        x = self.inception2c(x)
        x = self.inception2d(x)
        x = self.downsample2(x)
        x = self.inception3a(x)
        x = self.inception3b(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fully_connected(x)
        x = self.dropout(x)
        x = self.fully_relu(x)
        return x

class CyclicLR:
    def __init__(self, base_lr, max_lr, step_size):
        self.base_lr = base_lr
        self.max_lr = max_lr
        self.step_size = step_size
        self.iteration = 0

    def get_lr(self):
        cycle = self.iteration // (2 * self.step_size)
        x = abs(self.iteration / self.step_size - 2 * cycle - 1)
        lr = self.base_lr + (self.max_lr - self.base_lr) * max(0, (1 - x))
        self.iteration += 1
        return lr
model = SmallInception()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

lrs = []
losses = []

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

```
    for epoch in range(5):
        print("Running epoch ")
        print(epoch)
        for images, labels in train_loader:
            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))

# 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()
```

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 $\times K \times H \times W \times Cin \times Cout$)

Naive

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

$$\text{CONV3} = 32 \times 32 \times 9 \times 192 \times 256 = 452984832 \tag{2}$$

$$\text{CONV5} = 32 \times 32 \times 25 \times 96 \times 256 = 629145600 \tag{3}$$

$$\text{Total} = 1,115,684,864 \tag{4}$$

Dimensionality Reduction (DR)

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

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

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

$$\text{Total} = 397,410,304 \tag{8}$$

4.1.4 d)

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

5 Question5

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