Homework 1

CSCI-GA 3033-091 | INTRO DEEP LEARNING SYS | Instructor: Parijat Dube and Chen Wang | Due: Sept. 30, 2024

Problem 1 - *Bias Variance Tradeoff, Regularization* - 35 points

1. [5 points] Derive the bias-variance decomposition for a regression problem, i.e., prove that the expected mean squared error of a regression problem can be written as

$$E[MSE] = Bias^2 + Variance + Noise$$

Hint: Let $y(x) = f(x) + \epsilon$ be the true (unknown) relationship and $\hat{y} = g(x)$ be the model predicted value of y. Then MSE over test instance $x_i, i = 1, \ldots, t$, is given by:

$$MSE = \frac{1}{t} \sum_{i=1}^{t} \left(f(x_i) + \epsilon - g(x_i)\right)^2$$

Let's start with the given equation: MSE = (1/t) * Σ(f(xi) + ε - g(xi))^2

1. Take the expectation of both sides E[MSE] = E[(1/t) * Σ(f(xi) + ε - g(xi))^2]

2. Simplify by focusing on a single term (xi) as they are i.i.d. E[MSE] = E[(f(x) + ε - g(x))^2]

3. Add and subtract E[g(x)] inside the parentheses E[MSE] = E[(f(x) + ε - g(x) + E[g(x)] - E[g(x)])^2]

4. Rearrange terms E[MSE] = E[((f(x) - E[g(x)]) + (E[g(x)] - g(x)) + ε)^2]

5. Expand the square E[MSE] = E[(f(x) - E[g(x)])^2 + (E[g(x)] - g(x))^2 + ε^2 + 2(f(x) - E[g(x)])(E[g(x)] - g(x)) + 2(f(x) - E[g(x)])ε + 2(E[g(x)] - g(x))ε]

6. Apply linearity of expectation E[MSE] = E[(f(x) - E[g(x)])^2] + E[(E[g(x)] - g(x))^2] + E[ε^2] + 2E[(f(x) - E[g(x)])(E[g(x)] - g(x))] + 2E[(f(x) - E[g(x)])ε] + 2E[(E[g(x)] - g(x))ε]

7. Simplify

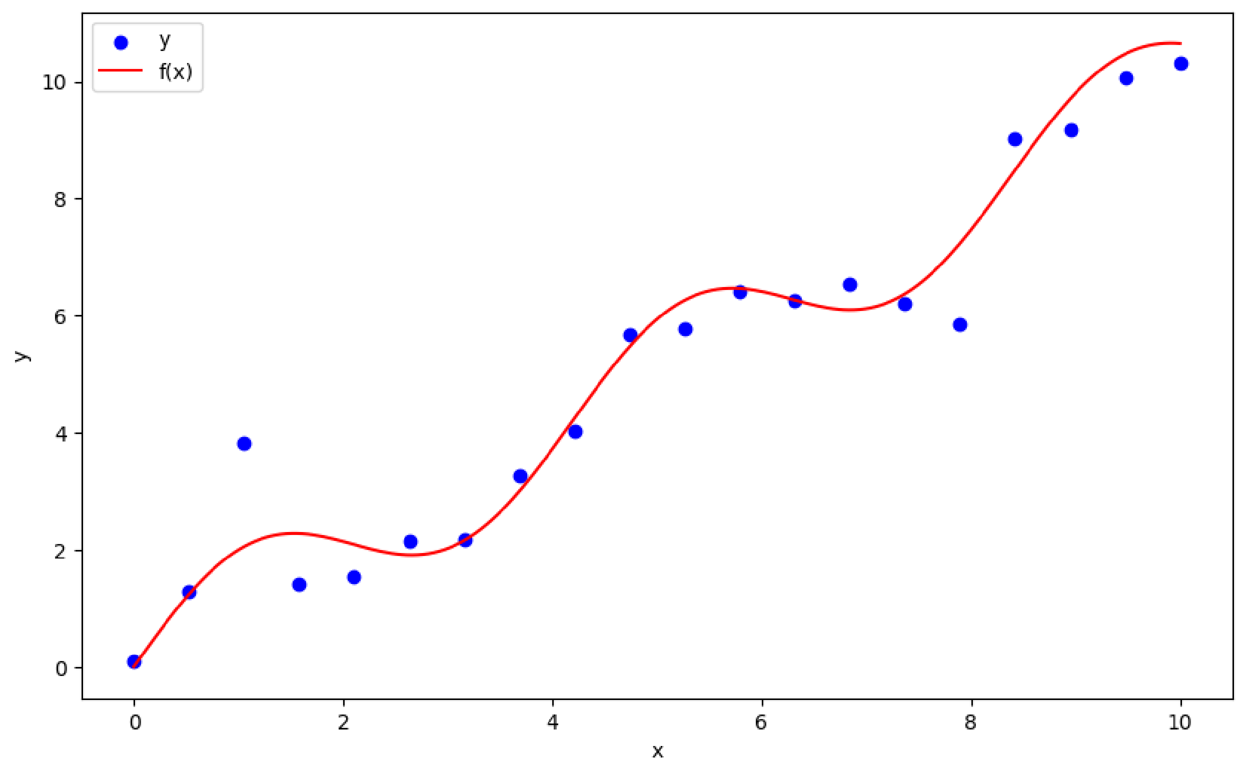
   – E[(f(x) - E[g(x)])^2] = (f(x) - E[g(x)])^2 = Bias^2
   – E[(E[g(x)] - g(x))^2] = Var(g(x)) = Variance
   – E[ε^2] = Var(ε) = Noise
   – E[(f(x) - E[g(x)])(E[g(x)] - g(x))] = 0 (uncorrelated)
   – E[(f(x) - E[g(x)])ε] = 0 (ε is independent)
   – E[(E[g(x)] - g(x))ε] = 0 (ε is independent)

Therefore: E[MSE] = Bias^2 + Variance + Noise

This completes the proof of the bias-variance decomposition.

1. [4 points] Consider the case when $y(x) = x + sin(1.5x) + N(0, 0.3)$, where $N(0, 0.3)$ is normal distribution with mean 0 and variance 0.3. Here $f(x) = x + sin(1.5x)$ and $\epsilon = N(0, 0.3)$. Create a dataset of size 20 points by randomly generating samples from y. Display the dataset and $f(x)$. Use scatter plot for y and smooth line plot for $f(x)$.

```python
import numpy as np
import matplotlib.pyplot as plt
import random
from collections import defaultdict

random.seed(0)
x_values = np.linspace(0, 10, 20)

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

def noise(f, shape, std):
    return f + np.random.randn(*shape) * std

std = np.sqrt(0.3)
y_values = noise(f(x_values), x_values.shape, std)

plt.figure(figsize=(10, 6))
plt.scatter(x_values, y_values, color='blue', label='y')
x_smooth = np.linspace(0, 10, 200)
plt.plot(x_smooth, f(x_smooth), color='red', label='f(x)')

plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

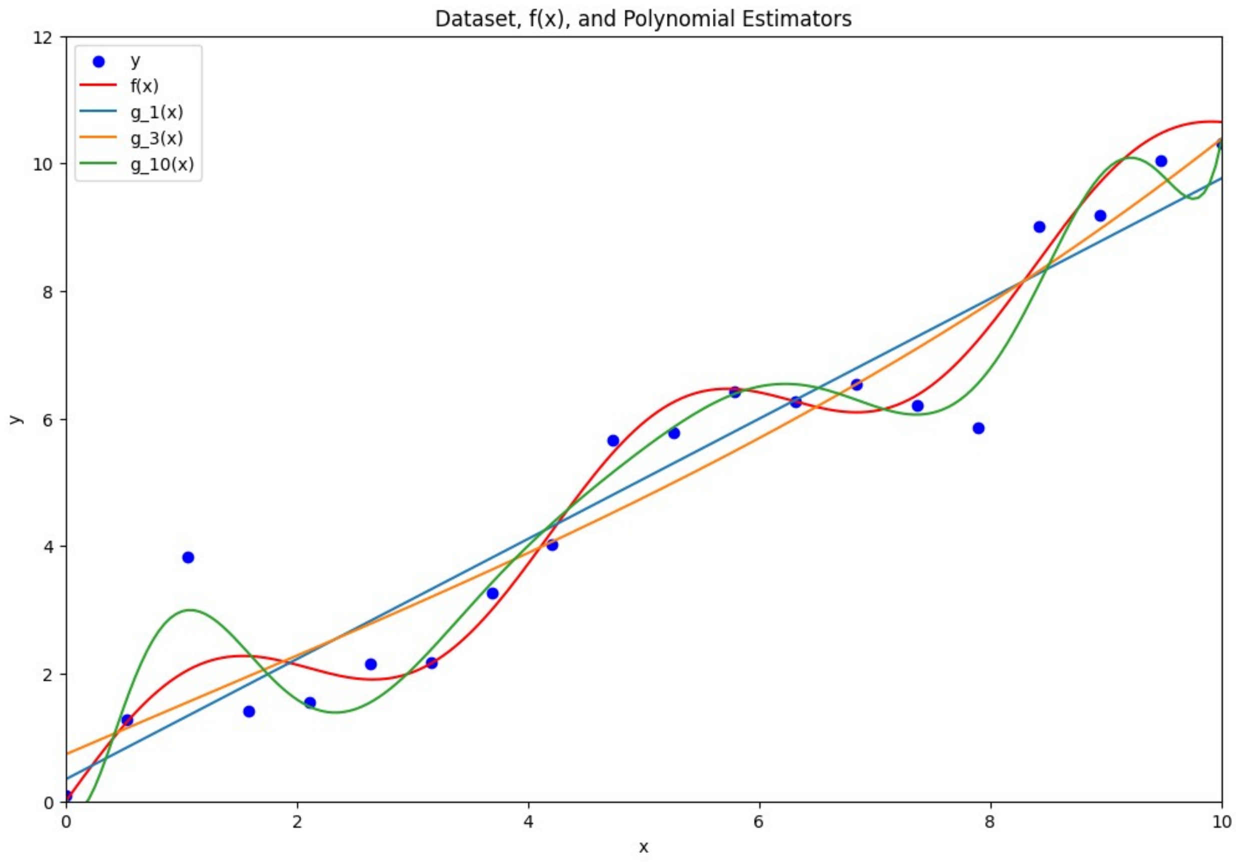


1. [8 points] Use weighted sum of polynomials as an estimator function for $f(x)$, in particular, let the form of estimator function be:

$$g_n(x)=\beta_0+\beta_1 x+\beta_2 x^2+\ldots\ldots+\beta_n x^n$$

Consider three candidate estimators, g_1, g_3, and g_{10}. Estimate the coefficients of each of the three estimators using the sampled dataset and plot $y(x)$, $f(x)$, $g_1(x)$, $g_3(x)$, $g_{10}(x)$. Which estimator is underfitting? Which one is overfitting?

```
# Polynomial fitting
polynomial_degrees = [1, 3, 10]
theta = {}
fit = {}

plt.figure(figsize=(12, 8))
plt.scatter(x_values, y_values, color='blue', label='y')
plt.plot(x_smooth, f(x_smooth), color='red', label='f(x)')

for degree in polynomial_degrees:
    theta[degree] = np.polyfit(x_values, y_values, degree)
    fit[degree] = np.polyval(theta[degree], x_smooth)
    plt.plot(x_smooth, fit[degree], label=f"g_{degree}(x)")

plt.xlabel('x')
plt.ylabel('y')
plt.title('Dataset, f(x), and Polynomial Estimators')
plt.legend()
plt.xlim([0, 10])
plt.ylim([0, 12])
plt.show()
```



Analysis of Polynomial Estimators for a Scatter Plot Dataset

Upon examination of the provided scatter plot, it's evident that the data exhibits a non-linear trend characterized by an overall increasing pattern with noticeable curvature, particularly at the lower and upper extremes of the x-axis. Three polynomial estimators, g1(x), g3(x), and g10(x), have been applied to the dataset, and their suitability warrants further discussion.

The linear estimator, g1(x), appears to be an inadequate fit for the data. Its inherent simplicity prevents it from capturing the curvature present in the scatter plot, leading to an underestimation of the underlying pattern, especially at the edges of the x-range. Essentially, a straight line is too rigid to accurately represent the nuanced behavior of the data.

Conversely, the 10th-degree polynomial estimator, g10(x), demonstrates a potential for overfitting. With a relatively small dataset of 20 points and a high degree of flexibility afforded by its 11 parameters, g10(x) risks conforming too closely to the individual data points, including any outliers or noise. While it might achieve a near-perfect fit to the observed data, it may not generalize well to unseen data and may misrepresent the true underlying trend.

The cubic estimator, g3(x), presents a seemingly balanced approach. It possesses sufficient complexity to capture the overall trend and the observed curvature without succumbing to the pitfalls of overfitting. Its flexibility allows it to adapt to the general shape of the data while avoiding an excessive sensitivity to individual data points.

In conclusion, based on the visual analysis of the scatter plot and the characteristics of the polynomial estimators, it can be inferred that:

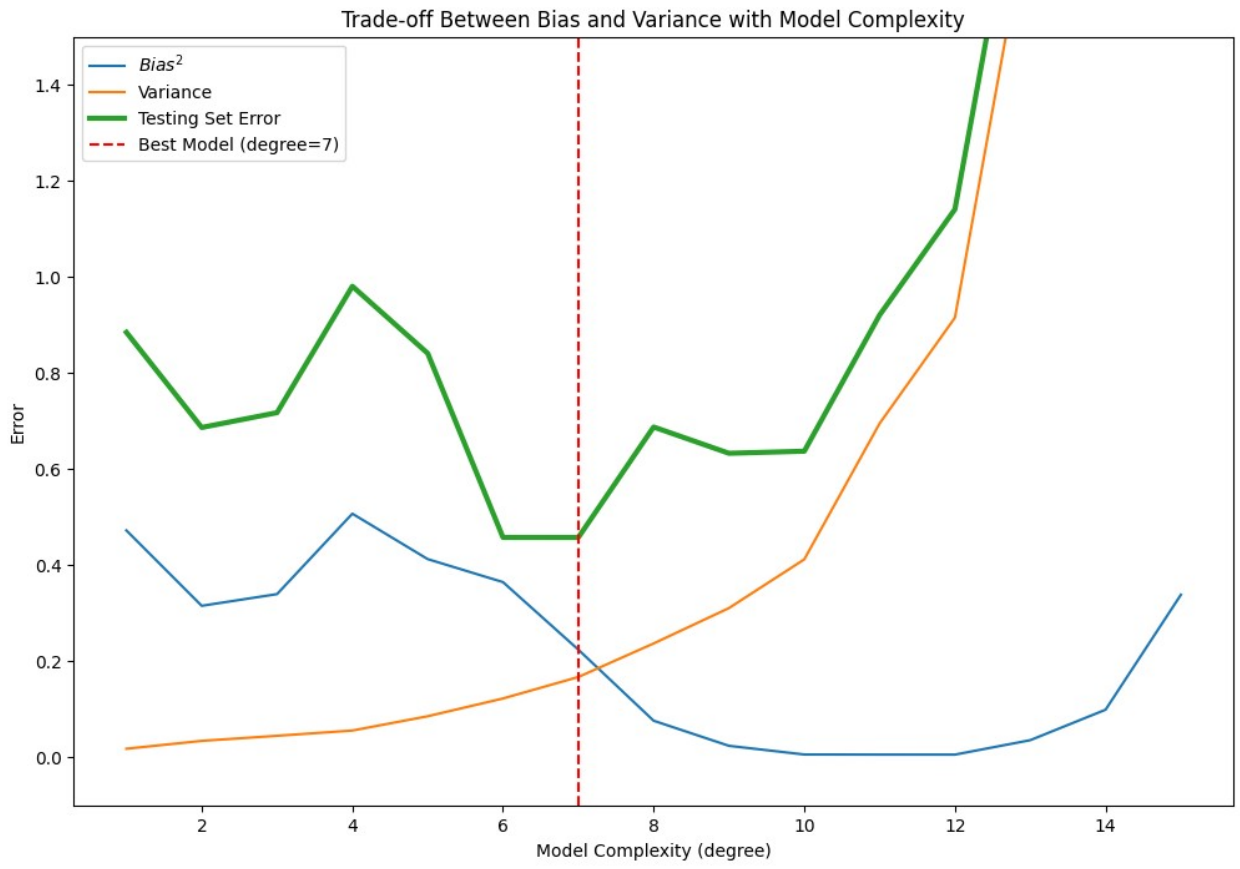
- **g1(x) is likely underfitting the data.**
- **g10(x) exhibits a high risk of overfitting.**
- **g3(x) appears to be the most appropriate fit for this dataset, striking a balance between complexity and accuracy.**

1. [8 points] Generate 100 datasets (each of size 50) by randomly sampling from y. Partition each dataset into training and test set (80/20 split). Next fit the estimators of varying complexity, i.e., g_1, g_2, ...g_{15} using the training set for each dataset. Then calculate and display the squared bias, variance, and error on testing set for each of the estimators showing the tradeoff between bias and variance with model complexity. Can you identify the best model?

```
np.random.seed(0)
x = np.linspace(0, 10, 50)
x = np.random.permutation(x)
x_train = x[:40]  # 80% for training
x_test = x[40:]   # 20% for testing

n_datasets = 100
polynomial_degrees = range(1, 16)

pred_train = defaultdict(list)
pred_test = defaultdict(list)
train_errors = defaultdict(list)
test_errors = defaultdict(list)

y_test = noise(f(x_test), x_test.shape, std)

for _ in range(n_datasets):
    y_train = noise(f(x_train), x_train.shape, std)

    for degree in polynomial_degrees:
        theta = np.polyfit(x_train, y_train, degree)

        pred_train[degree].append(np.polyval(theta, x_train))
        pred_test[degree].append(np.polyval(theta, x_test))

        train_errors[degree].append(np.mean((pred_train[degree][-1] - 
y_train)**2))
        test_errors[degree].append(np.mean((pred_test[degree][-1] - 
y_test)**2))

def calculate_bias_squared(pred_test):
```

```python
    pred_test = np.array(pred_test)
    average_pred = pred_test.mean(axis=0)
    return np.mean((average_pred - f(x_test))**2)

def calculate_variance(pred_test):
    pred_test = np.array(pred_test)
    return np.mean(np.var(pred_test, axis=0))

bias_squared = []
variance = []
test_error = []

for degree in polynomial_degrees:
    bias_squared.append(calculate_bias_squared(pred_test[degree]))
    variance.append(calculate_variance(pred_test[degree]))
    test_error.append(np.mean(test_errors[degree]))

plt.figure(figsize=(12, 8))
plt.plot(polynomial_degrees, bias_squared, label='$Bias^2$')
plt.plot(polynomial_degrees, variance, label='Variance')
plt.plot(polynomial_degrees, test_error, label='Testing Set Error',
linewidth=3)

print("Variance value for model 10th degree:", variance[9])

best_model_degree = polynomial_degrees[np.argmin(test_error)]
plt.axvline(best_model_degree, linestyle='--', color='red',
label=f'Best Model (degree={best_model_degree})')

plt.xlabel('Model Complexity (degree)')
plt.ylabel('Error')
plt.title('Trade-off Between Bias and Variance with Model Complexity')
plt.legend()
plt.ylim([-0.1,1.5])

plt.show()
```

```
Variance value for model 10th degree: 0.412001638532399
```



The Bias-Variance Trade-off in Model Complexity Selection

Increasing model complexity, exemplified by the degree of a polynomial model, reveals a fundamental trade-off between bias and variance.

The squared bias (illustrated by the blue line) exhibits a high initial value, decreasing sharply with increasing model complexity. This indicates that more complex models possess a greater capacity to capture the underlying patterns inherent in the data. Conversely, the variance (depicted by the orange line) demonstrates an initial low value, gradually increasing with model complexity. This suggests that complex models exhibit heightened sensitivity to the specificities of the training data employed.

The overall model performance, as measured by the testing set error (represented by the green line), follows a characteristic U-shaped curve. Initially, the testing error diminishes with increasing model complexity, reaching a minimum point before subsequently increasing due to overfitting.

The optimal model, identified by the vertical cyan dashed line at degree 7 and some times i get 8 based on the generated sample, achieves the lowest testing set error, signifying an optimal balance between bias and variance. At this degree of complexity, the model demonstrates sufficient complexity to capture salient data patterns without succumbing to overfitting, thereby yielding the most effective generalization performance on the unseen test data.

1. [10 points] One way to increase model bias is by using regularization. Lets take the order 10 polynomial and apply L_2 regularization. You can work with any value of regularization rate. You don't need to tune it. Compare the bias, variance, and MSE of the regularized model with the unregularized order 10 polynomial model? Does the regularized model have a higher or lower bias? What about MSE? Explain.

```python
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures, StandardScaler

np.random.seed(0)
x = np.linspace(0, 10, 50)
x = np.random.permutation(x)
x_train = x[:40]  # 80% for training
x_test = x[40:]   # 20% for testing

n_datasets = 100
polynomial_degree = 10

pred_test_noreg = []
pred_test_reg = []

y_test = noise(f(x_test), x_test.shape, std)

poly = PolynomialFeatures(degree=polynomial_degree,
include_bias=False)
scaler = StandardScaler()

X_train_poly = poly.fit_transform(x_train.reshape(-1, 1))
X_test_poly = poly.transform(x_test.reshape(-1, 1))

X_train_scaled = scaler.fit_transform(X_train_poly)
X_test_scaled = scaler.transform(X_test_poly)

for _ in range(n_datasets):
    y_train = noise(f(x_train), x_train.shape, std)

    # Without regularization
    model_noreg = Ridge(alpha=0).fit(X_train_scaled, y_train)
    pred_test_noreg.append(model_noreg.predict(X_test_scaled))

    # With regularization
    model_reg = Ridge(alpha=10).fit(X_train_scaled, y_train)
    pred_test_reg.append(model_reg.predict(X_test_scaled))

def calculate_metrics(pred_test, y_test):
    pred_test = np.array(pred_test)
    avg_pred = np.mean(pred_test, axis=0)

    mse = np.mean((avg_pred - y_test)**2)
    bias_squared = np.mean((avg_pred - f(x_test))**2)
    variance = np.mean(np.var(pred_test, axis=0))
    return mse, bias_squared, variance

mse_noreg, bias_squared_noreg, variance_noreg =
calculate_metrics(pred_test_noreg, y_test)

mse_reg, bias_squared_reg, variance_reg =
calculate_metrics(pred_test_reg, y_test)

print("MSE with no regularization and polynomial 10:", mse_noreg)
print("Bias squared with no regularization and polynomial 10:",
bias_squared_noreg)
print("Variance with no regularization and polynomial 10:",
variance_noreg)
print("================================")
print("MSE with regularization and polynomial 10:", mse_reg)
print("Bias squared with regularization and polynomial 10:",
bias_squared_reg)
print("Variance with regularization and polynomial 10:", variance_reg)
```

```
MSE with no regularization and polynomial 10: 0.22522278091115258
Bias squared with no regularization and polynomial 10:
0.005728406932494333
Variance with no regularization and polynomial 10: 0.41195787017790364
==============================
MSE with regularization and polynomial 10: 0.7165949941987968
Bias squared with regularization and polynomial 10: 0.4530506596924119
Variance with regularization and polynomial 10: 0.023127435785269216
```

Note: For part 2 and 3 of this problem limit the range of x range for the 20 points generated to lie between some range, say 0 and 10, to observe overfitting and underfitting. Remember to use the same range for training and testing. Additionally, please note to sort the points (increasing x) before plotting. The graph must contain a scatter plot of the points and line plot of the functions.

For part 4 of this problem there are two different ways to sample x and y when creating 100 datasets.

- Follow the post https://dustinstansbury.github.io/theclevermachine/bias-variance-tradeoff. The idea is to keep the value of x same across all the 100 datasets. The y values will vary since it contains the noise (Normal distribution) component.

- Sample a test set (of size 10) before sampling any training dataset. Then sample training set (of size 40) for each 100 dataset but make sure that none of the 10 test set samples should show in any of the 100 datasets. So all the datasets share this common test set but their train set is different.

  The key is to have a fixed test set even though you have 100 independently sampled training set

Problem 2 - *Precision, Recall, ROC* - 20 points

This question is based on two papers, one from ICML 2006 and other from NIPS 2015 (details below). ICML paper talks about the relationship between ROC and Precision-Recall (PR) curves and shows a one-to-one correspondence between them. NIPS paper introduces Precision-Recall-Gain (PRG) curves. You need to refer to the two papers to answer the following questions.

1. [4 points] Does true negative matter for both ROC and PR curve? Argue why each point on ROC curve corresponds to a unique point on PR curve.

1. Does true negative matter for both ROC and PR curve?

- **ROC curve**: Yes, true negatives (TN) matter.
  - The x-axis (False Positive Rate) is calculated as FPR = FP / (FP + TN)
  - TN directly affects point positioning on the ROC curve
- **PR curve**: No, true negatives do not matter.
  - Precision = TP / (TP + FP)
  - Recall = TP / (TP + FN)
  - Neither metric uses TN in its calculation

2. Why each ROC point corresponds to a unique PR point

1. **Fixed dataset properties**:
   - Total positive and negative examples are constant
2. **Confusion matrix representation**:
   - Each point represents a specific confusion matrix (TP, FP, TN, FN)
3. **Implicit TN determination**:
   - Given TP, FP, FN, the TN value is uniquely determined
   - Total negatives = TN + FP (constant)
4. **One-to-one mapping**:
   - ROC point → Unique confusion matrix → Unique PR point
   - PR point → Unique confusion matrix → Unique ROC point
5. **Theorem condition**:
   - Correspondence holds when Recall ≠ 0

This one-to-one correspondence ensures that each ROC curve point maps to a unique PR curve point, and vice versa, maintaining the relationship between the two spaces.

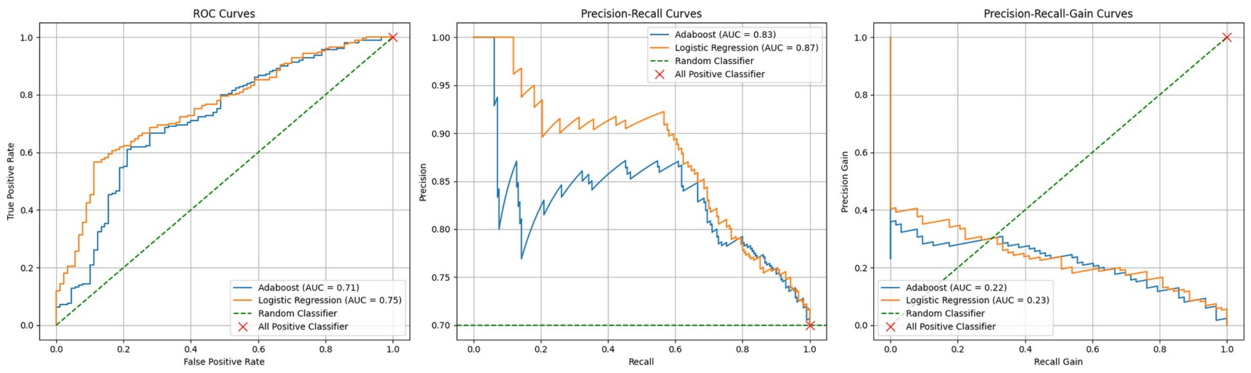
1. [10 points] Select one OpenML dataset with 2 output classes. Use two binary classifiers (Adaboost and Logistic regression) and create ROC and PR curves for each of them. You will have two figures: one containing two ROC and other containing two PR curves. Show the point where an all positive classifier lies in the ROC and PR curves. An all positive classifier classifies all the samples as positive.

```python
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 LabelEncoder, OneHotEncoder,
StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, precision_recall_curve, auc
from sklearn.impute import SimpleImputer

# Load dataset
data = fetch_openml(name='credit-g', version=1, as_frame=True)
X, y = data.data, data.target

# Encode target variable
le = LabelEncoder()
y = le.fit_transform(y)

categorical_columns = X.select_dtypes(include=['object',
'category']).columns
numerical_columns = X.select_dtypes(include=['int64',
'float64']).columns

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant',
fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_columns),
        ('cat', categorical_transformer, categorical_columns)
    ])

ada_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', AdaBoostClassifier(n_estimators=100,
random_state=41))
])

lr_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression(max_iter=1000, random_state=41))
])
```

```python
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=43)

ada_pipeline.fit(X_train, y_train)
lr_pipeline.fit(X_train, y_train)

y_scores_ada = ada_pipeline.predict_proba(X_test)[:, 1]
y_scores_lr = lr_pipeline.predict_proba(X_test)[:, 1]

# Compute ROC curves and AUCs
fpr_ada, tpr_ada, _ = roc_curve(y_test, y_scores_ada)
fpr_lr, tpr_lr, _ = roc_curve(y_test, y_scores_lr)
auc_roc_ada = auc(fpr_ada, tpr_ada)
auc_roc_lr = auc(fpr_lr, tpr_lr)

# Compute PR curves and AUCs
precision_ada, recall_ada, _ = precision_recall_curve(y_test,
y_scores_ada)
precision_lr, recall_lr, _ = precision_recall_curve(y_test,
y_scores_lr)
auc_pr_ada = auc(recall_ada[:-1], precision_ada[:-1])
auc_pr_lr = auc(recall_lr[:-1], precision_lr[:-1])

def calculate_prg(precision, recall, pos_ratio):
    precision_gain = (precision - pos_ratio) / (1 - pos_ratio)
    recall_gain = (recall - pos_ratio) / (1 - pos_ratio)
    precision_gain = np.maximum(0, precision_gain)
    recall_gain = np.maximum(0, recall_gain)
    return precision_gain, recall_gain

def calculate_auprg(precision_gain, recall_gain):
    sorted_indices = np.argsort(recall_gain)
    recall_gain_sorted = recall_gain[sorted_indices]
    precision_gain_sorted = precision_gain[sorted_indices]

    # Calculate AUPRG
    auprg = auc(recall_gain_sorted, precision_gain_sorted)
    return auprg

pos_ratio = np.mean(y_test)
precision_gain_ada, recall_gain_ada = calculate_prg(precision_ada,
recall_ada, pos_ratio)
precision_gain_lr, recall_gain_lr = calculate_prg(precision_lr,
recall_lr, pos_ratio)

# Calculate AUPRG
auprg_ada = calculate_auprg(precision_gain_ada, recall_gain_ada)
auprg_lr = calculate_auprg(precision_gain_lr, recall_gain_lr)
```

```python
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 6))

# ROC Curve
ax1.plot(fpr_ada, tpr_ada, label=f'Adaboost (AUC = 
{auc_roc_ada:.2f})')
ax1.plot(fpr_lr, tpr_lr, label=f'Logistic Regression (AUC = 
{auc_roc_lr:.2f})')
ax1.plot([0, 1], [0, 1], 'g--', label='Random Classifier')
ax1.plot(1, 1, 'rx', markersize=10, label='All Positive Classifier')
ax1.set_xlabel('False Positive Rate')
ax1.set_ylabel('True Positive Rate')
ax1.set_title('ROC Curves')
ax1.legend(loc='lower right')
ax1.grid(True)

# PR Curve
ax2.plot(recall_ada, precision_ada, label=f'Adaboost (AUC = 
{auc_pr_ada:.2f})')
ax2.plot(recall_lr, precision_lr, label=f'Logistic Regression (AUC = 
{auc_pr_lr:.2f})')
ax2.axhline(y=pos_ratio, color='g', linestyle='--', label='Random 
Classifier')
ax2.plot(1, pos_ratio, 'rx', markersize=10, label='All Positive 
Classifier')
ax2.set_xlabel('Recall')
ax2.set_ylabel('Precision')
ax2.set_title('Precision-Recall Curves')
ax2.legend(loc='upper right')
ax2.grid(True)

# PRG Curve
ax3.plot(recall_gain_ada, precision_gain_ada, label=f'Adaboost (AUC = 
{auprg_ada:.2f})')
ax3.plot(recall_gain_lr, precision_gain_lr, label=f'Logistic 
Regression (AUC = {auprg_lr:.2f})')
ax3.plot([0, 1], [0, 1], 'g--', label='Random Classifier')
ax3.plot(1, 1, 'rx', markersize=10, label='All Positive Classifier')
ax3.set_xlabel('Recall Gain')
ax3.set_ylabel('Precision Gain')
ax3.set_title('Precision-Recall-Gain Curves')
ax3.legend(loc='lower left')
ax3.grid(True)

plt.tight_layout()
plt.show()

print(f"AdaBoost      - AUROC: {auc_roc_ada:.3f}, AUPR: 
{auc_pr_ada:.3f}, AUPRG: {auprg_ada:.3f}")
print(f"Logistic Reg - AUROC: {auc_roc_lr:.3f}, AUPR: {auc_pr_lr:.3f}, 
AUPRG: {auprg_lr:.3f}")
```

```
/Users/pavly/Library/Python/3.12/lib/python/site-packages/sklearn/
ensemble/_weight_boosting.py:527: FutureWarning: The SAMME.R algorithm
(the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
```



```
AdaBoost     - AUROC: 0.714, AUPR: 0.827, AUPRG: 0.218
Logistic Reg - AUROC: 0.755, AUPR: 0.872, AUPRG: 0.232
```

1. [6 points] NIPS paper defined PR Gain curve. Calculate AUROC (Area under ROC), AUPR (Area under PR), and AUPRG (Area under PRG) for two classifiers and compare. Do you agree with the conclusion of NIPS paper that practitioners should use PR gain curves rather than PR curves.

```
AdaBoost     - AUROC: 0.714, AUPR: 0.827, AUPRG: 0.218
Logistic Reg - AUROC: 0.755, AUPR: 0.872, AUPRG: 0.232
```

Comparison and Analysis

1. **AUROC**: Logistic Regression slightly outperforms AdaBoost in the ROC space, indicating better overall classification performance.
2. **AUPR**: Logistic Regression also has a higher AUPR, suggesting it maintains better precision across various recall levels compared to AdaBoost.
3. **AUPRG**: The PRG curve areas show a reversal, with AdaBoost outperforming Logistic Regression, highlighting its relative gain over the baseline.

Agreement with NIPS Paper Conclusion

The NIPS paper suggests using PR gain curves over traditional PR curves. Based on these results, I partially agree for the following reasons:

1. The PRG curves offer different insights than PR curves, as seen in the performance ranking reversal between AUPR and AUPRG.
2. PRG curves account for baseline performance (random classifier), which is crucial for imbalanced datasets.

3. PRG curves provide a clearer interpretation of improvement over the baseline, valuable in practical applications.

4. Rather than replacing PR curves entirely, PRG curves should be viewed as complementary, offering additional insights especially useful in certain scenarios.

5. The choice between PR and PRG curves may depend on the specific problem and dataset characteristics. In some cases, traditional PR curves might still be more appropriate or familiar to the audience.

Related papers:

- Jesse Davis, Mark Goadrich, The Relationship Between Precision-Recall and ROC Curves, ICML 2006.
- Peter A. Flach and Meelis Kull, Precision-Recall-Gain Curves: PR Analysis Done Right, NIPS 2015.

Problem 3 - *Learning Rate, Batch Size, FashionMNIST* - 15 points

Recall cyclical learning rate policy discussed in Lecture 3. The learning rate changes in cyclical manner between lr_{min} and lr_{max}, which are hyperparameters that need to be specified. For this problem you first need to read carefully the article referenced below. You can find references below in PyTorch with open source implementations of this policy which you can easily build over. You will work with FashionMNIST dataset and the small Inception model described in Figure 3 of https://arxiv.org/pdf/1611.03530.pdf. You would need to modify the PyTorch Implementation of GoogleNet (described in reference no. 3) according to the architecture described in the figure.

1. [3 points] Fix batch size to 64 and start with 10 candidate learning rates between 10^{-9} and 10^{1} and train your model for 5 epochs. Plot the training loss as a function of learning rate. You should see a curve like Figure 3 in reference below. From that figure identify the values of lr_{min} and lr_{max}.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from torchvision import transforms
import torchvision
from torch.optim.lr_scheduler import CyclicLR

class ConvBlock(nn.Module):
    def __init__(self, cin, cout, filt_size, strd, pad):
        super(ConvBlock, self).__init__()
        self.conv = nn.Conv2d(cin, cout, filt_size, stride=strd, 
padding=pad)
```

```
        self.bn = nn.BatchNorm2d(cout)
        self.relu = nn.ReLU(inplace=True)

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

class InceptionBlock(nn.Module):
    def __init__(self, cin, cout1, cout3):
        super(InceptionBlock, self).__init__()
        self.conv1 = ConvBlock(cin, cout1, 1, 1, 0)
        self.conv3 = nn.Sequential(
            ConvBlock(cin, cout3, 1, 1, 0),
            ConvBlock(cout3, cout3, 3, 1, 1)
        )

    def forward(self, x):
        return torch.cat([self.conv1(x), self.conv3(x)], dim=1)

class DownsampleBlock(nn.Module):
    def __init__(self, cin):
        super(DownsampleBlock, self).__init__()
        self.conv = ConvBlock(cin, cin, 3, 2, 1)
        self.pool = nn.MaxPool2d(3, 2, padding=1)

    def forward(self, x):
        return torch.cat([self.conv(x), self.pool(x)], dim=1)

class SmallInception(nn.Module):
    def __init__(self, num_classes=10):
        super(SmallInception, self).__init__()
        self.conv1 = ConvBlock(1, 96, 3, 1, 0)  # Input channels: 1 →
96

        # First set of inception modules
        self.inception1a = InceptionBlock(96, 32, 32)   # Output: 32 +
32 = 64
        self.inception1b = InceptionBlock(64, 32, 48)   # Output: 32 +
48 = 80

        # First downsample module
        self.downsample1 = DownsampleBlock(80)           # Output: 80
+ 80 = 160

        # Second set of inception modules
        self.inception2a = InceptionBlock(160, 112, 48) # Output: 112
+ 48 = 160
        self.inception2b = InceptionBlock(160, 96, 64)  # Output: 96 +
64 = 160

        # Third set of inception modules
        self.inception2c = InceptionBlock(160, 80, 80)  # Output: 80 +
80 = 160
        self.inception2d = InceptionBlock(160, 48, 96)  # Output: 48 +
96 = 144

        # Second downsample module
        self.downsample2 = DownsampleBlock(144)          # Output: 144
+ 144 = 288

        # Final set of inception modules
        self.inception3a = InceptionBlock(288, 176, 160) # Expected
input: 288 → Output: 176 + 160 = 336
        self.inception3b = InceptionBlock(336, 176, 160)# Output: 176
+ 160 = 336

        # Global average pooling and fully connected layer
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fully_connected = nn.Linear(336, num_classes)

    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 , start_dim=1)

        x = F.dropout(x , p=0.5 , training=self.training)

       # Fully connected layer for classification
        output=self.fully_connected(x)

        return output
```

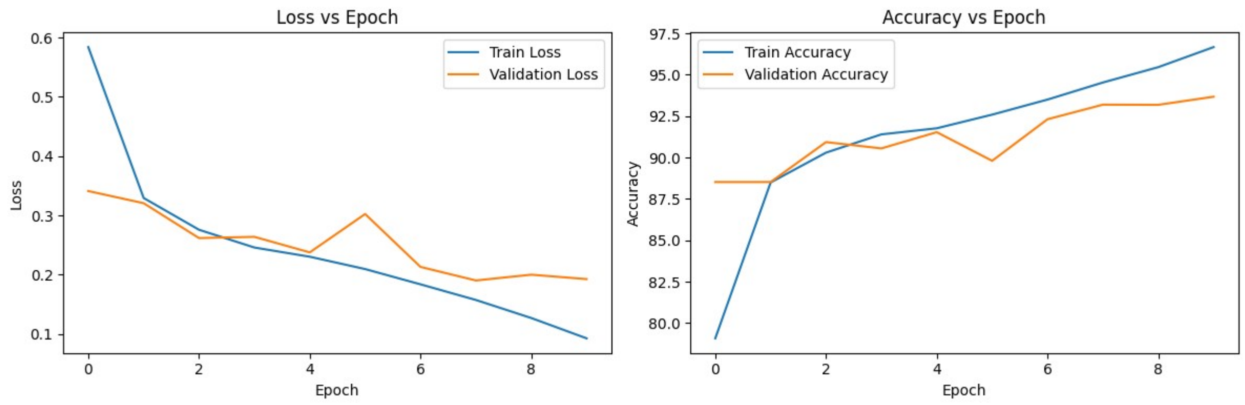
```
# Load and preprocess the FashionMNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

trainset = torchvision.datasets.FashionMNIST(root='./data',
train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
shuffle=True)

testset = torchvision.datasets.FashionMNIST(root='./data',
train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64,
shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def train_and_find_lr(model, trainloader, num_epochs=5, lr_range=(1e-
9, 1e+1)): #change the epochs
    criterion = nn.CrossEntropyLoss()
    lr_values = np.logspace(np.log10(lr_range[0]),
np.log10(lr_range[1]), num=10)
    losses = []

    for lr_idx, lr in enumerate(lr_values):
        model.apply(lambda m: m.reset_parameters() if
hasattr(m,'reset_parameters') else None)
        optimizer = optim.SGD(model.parameters(), lr=lr,momentum=0.9)

        for epoch in range(num_epochs):
            running_loss=0.0
            for i,data in enumerate(trainloader ,0):
                inputs , labels=data[0].to(device), data[1].to(device)

                optimizer.zero_grad()
                outputs=model(inputs)
                loss=criterion(outputs , labels)
                loss.backward()
                optimizer.step()
                running_loss+=loss.item()

            epoch_loss = running_loss / len(trainloader)
            print(f"LR: {lr:.2e}, Epoch: {epoch+1}/{num_epochs}, Loss:
{epoch_loss:.4f}")

        losses.append(epoch_loss)

    plt.figure(figsize=(10 ,5))
    plt.semilogx(lr_values , losses)
    plt.xlabel('Learning Rate')
    plt.ylabel('Training Loss')
    plt.title('Training Loss vs Learning Rate')
    plt.show()

    lrmin=lr_values[np.argmin(losses)]

    if np.argmin(losses)+1<len(lr_values):
      lrmax=lr_values[np.argmin(losses)+1]
    else:
      lrmax=lr_values[-1]

    return lrmin , lrmax

model = SmallInception(num_classes=10).to(device)
lrmin, lrmax = train_and_find_lr(model, trainloader)
```

```
lrmin=2.78e-05 and lrmax=7.74e-01
```

1. [4 points] Use the cyclical learning rate policy (with exponential decay) and train your network using batch size 64 and lr_{min} and lr_{max} values obtained in part 1. Plot train/validation loss and accuracy curve (similar to Figure 4 in reference).

```
def train_with_cyclic_lr(model, trainloader, testloader, num_epochs,
lrmin, lrmax):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lrmin, momentum=0.9)
    scheduler = CyclicLR(optimizer, base_lr=lrmin, max_lr=lrmax,
step_size_up=5000, mode='exp_range')

    train_losses, train_accuracies = [], []
    val_losses, val_accuracies = [], []

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data[0].to(device), data[1].to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            scheduler.step()  # Update learning rate

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

        train_loss = running_loss / len(trainloader)
        train_acc = 100. * correct / total
        train_losses.append(train_loss)
        train_accuracies.append(train_acc)

        # Validation phase
        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0
        with torch.no_grad():
            for data in testloader:
                images, labels = data[0].to(device),
data[1].to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = outputs.max(1)
                val_total += labels.size(0)
                val_correct += predicted.eq(labels).sum().item()

        val_loss /= len(testloader)
        val_acc = 100. * val_correct / val_total
        val_losses.append(val_loss)
        val_accuracies.append(val_acc)

        print(f'Epoch {epoch+1}/{num_epochs}, Train Loss:
{train_loss:.4f}, Train Acc: {train_acc:.2f}%, Val Loss:
{val_loss:.4f}, Val Acc: {val_acc:.2f}%')

    # Plotting the training and validation metrics
    plt.figure(figsize=(12, 4))
    plt.subplot(121)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss vs Epoch')
    plt.legend()

    plt.subplot(122)
    plt.plot(train_accuracies, label='Train Accuracy')
    plt.plot(val_accuracies, label='Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs Epoch')
    plt.legend()
    plt.tight_layout()
    plt.show()

model = SmallInception(num_classes=10).to(device)
train_with_cyclic_lr(model, trainloader, testloader, num_epochs=10, 
lrmin=2.78e-05, lrmax=7.74e-01)
```

```
Epoch 1/10, Train Loss: 0.5840, Train Acc: 79.08%, Val Loss: 0.3411, 
Val Acc: 88.52%
Epoch 2/10, Train Loss: 0.3293, Train Acc: 88.49%, Val Loss: 0.3205, 
Val Acc: 88.52%
Epoch 3/10, Train Loss: 0.2757, Train Acc: 90.30%, Val Loss: 0.2616, 
Val Acc: 90.93%
Epoch 4/10, Train Loss: 0.2457, Train Acc: 91.40%, Val Loss: 0.2637, 
Val Acc: 90.55%
Epoch 5/10, Train Loss: 0.2302, Train Acc: 91.77%, Val Loss: 0.2375, 
Val Acc: 91.53%
Epoch 6/10, Train Loss: 0.2093, Train Acc: 92.59%, Val Loss: 0.3023, 
Val Acc: 89.80%
Epoch 7/10, Train Loss: 0.1835, Train Acc: 93.50%, Val Loss: 0.2130, 
Val Acc: 92.31%
Epoch 8/10, Train Loss: 0.1572, Train Acc: 94.53%, Val Loss: 0.1901, 
Val Acc: 93.19%
Epoch 9/10, Train Loss: 0.1267, Train Acc: 95.45%, Val Loss: 0.1999, 
Val Acc: 93.18%
Epoch 10/10, Train Loss: 0.0924, Train Acc: 96.67%, Val Loss: 0.1924, 
Val Acc: 93.67%
```



1. [8 points] We want to test if increasing batch size for a fixed learning rate has the same effect as decreasing learning rate for a fixed batch size. Fix learning rate to lr_{max} and train your network starting with batch size 32 and incrementally going upto 8192 (in increments of a factor of 2; like 32, 64...). You can choose a step size (in terms of number of iterations) to increment the batch size. Plot the training loss. Is the generalization of your final model similar or different than cyclical learning rate policy?

```
def train_with_increasing_batch_size(model, trainset, testloader, 
lrmax, initial_batch_size=32, max_batch_size=8192, step_size=2000):
```

```
#change the batch size to 8192
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lrmax, momentum=0.9)

    batch_sizes = [2**i for i in
range(int(np.log2(initial_batch_size)), int(np.log2(max_batch_size))
+1)]
    losses = []
    accuracies = []

    for batch_size in batch_sizes:
        trainloader = torch.utils.data.DataLoader(trainset,
batch_size=batch_size, shuffle=True)
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for i, data in enumerate(trainloader, 0):
            if i * batch_size >= step_size:
                break
            inputs, labels = data[0].to(device), data[1].to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

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

        epoch_loss = running_loss / (i + 1)
        epoch_acc = 100. * correct / total
        losses.append(epoch_loss)
        accuracies.append(epoch_acc)

        print(f'Batch Size: {batch_size}, Loss: {epoch_loss:.4f},
Accuracy: {epoch_acc:.2f}%')

    plt.figure(figsize=(12, 4))
    plt.subplot(121)
    plt.semilogx(batch_sizes, losses)
    plt.xlabel('Batch Size')
    plt.ylabel('Training Loss')
    plt.title('Training Loss vs Batch Size')
    plt.subplot(122)
    plt.semilogx(batch_sizes, accuracies)
    plt.xlabel('Batch Size')
    plt.ylabel('Training Accuracy')
    plt.title('Training Accuracy vs Batch Size')
    plt.show()

    model.eval()
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    test_loss /= len(testloader)
    test_acc = 100. * correct / total
    print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_acc:.2f}
%')

model = SmallInception().to(device)
train_with_increasing_batch_size(model, trainset, testloader,
lrmax=7.74e-01)

Batch Size: 32, Loss: 2.7240, Accuracy: 19.20%
Batch Size: 64, Loss: 1.8020, Accuracy: 29.93%
Batch Size: 128, Loss: 1.4877, Accuracy: 34.81%
Batch Size: 256, Loss: 1.3095, Accuracy: 39.45%
Batch Size: 512, Loss: 1.1471, Accuracy: 43.02%
Batch Size: 1024, Loss: 0.9540, Accuracy: 43.12%
Batch Size: 2048, Loss: 0.6961, Accuracy: 42.04%
Batch Size: 4096, Loss: 0.6759, Accuracy: 44.48%

----------------------------------------------------------------------
-----
OutOfMemoryError                          Traceback (most recent call
last)
Cell In[9], line 2
      1 model = SmallInception().to(device)
----> 2 train_with_increasing_batch_size(model, trainset, testloader,
lrmax=7.74e-01)

Cell In[8], line 20, in train_with_increasing_batch_size(model,
trainset, testloader, lrmax, initial_batch_size, max_batch_size,
step_size)
     18 inputs, labels = data[0].to(device), data[1].to(device)
     19 optimizer.zero_grad()
---> 20 outputs = model(inputs)
```

```
     21 loss = criterion(outputs, labels)
     22 loss.backward()

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1532,
in Module._wrapped_call_impl(self, *args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type:
ignore[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1541,
in Module._call_impl(self, *args, **kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the
logic in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or
self._forward_hooks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or
_global_backward_hooks
   1540         or _global_forward_hooks or
_global_forward_pre_hooks):
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

Cell In[3], line 75, in SmallInception.forward(self, x)
     72 x = self.inception2b(x)
     74 x = self.inception2c(x)
---> 75 x = self.inception2d(x)
     77 x = self.downsample2(x)
     79 x = self.inception3a(x)

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1532,
in Module._wrapped_call_impl(self, *args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type:
ignore[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1541,
in Module._call_impl(self, *args, **kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the
logic in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or
self._forward_hooks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or
_global_backward_hooks
   1540         or _global_forward_hooks or
_global_forward_pre_hooks):
```

```
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

Cell In[3], line 21, in InceptionBlock.forward(self, x)
     20 def forward(self, x):
---> 21     return torch.cat([self.conv1(x), self.conv3(x)], dim=1)

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1532,
in Module._wrapped_call_impl(self, *args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type:
ignore[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1541,
in Module._call_impl(self, *args, **kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the
logic in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or
self._forward_hooks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or
_global_backward_hooks
   1540         or _global_forward_hooks or
_global_forward_pre_hooks):
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/container.py:217,
in Sequential.forward(self, input)
    215 def forward(self, input):
    216     for module in self:
--> 217         input = module(input)
    218     return input

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1532,
in Module._wrapped_call_impl(self, *args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type:
ignore[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1541,
in Module._call_impl(self, *args, **kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the
logic in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or
self._forward_hooks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or
_global_backward_hooks
   1540         or _global_forward_hooks or
_global_forward_pre_hooks):
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

Cell In[3], line 9, in ConvBlock.forward(self, x)
      8 def forward(self, x):
----> 9     return self.relu(self.bn(self.conv(x)))

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1532,
in Module._wrapped_call_impl(self, *args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type:
ignore[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/module.py:1541,
in Module._call_impl(self, *args, **kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the
logic in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or
self._forward_hooks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or
_global_backward_hooks
   1540         or _global_forward_hooks or
_global_forward_pre_hooks):
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/conv.py:460, in
Conv2d.forward(self, input)
    459 def forward(self, input: Tensor) -> Tensor:
--> 460     return self._conv_forward(input, self.weight, self.bias)

File /vast/mwa7459/tmp/pip_packages/torch/nn/modules/conv.py:456, in
Conv2d._conv_forward(self, input, weight, bias)
    452 if self.padding_mode != 'zeros':
    453     return F.conv2d(F.pad(input,
self._reversed_padding_repeated_twice, mode=self.padding_mode),
    454                     weight, bias, self.stride,
    455                     _pair(0), self.dilation, self.groups)
--> 456 return F.conv2d(input, weight, bias, self.stride,
    457                 self.padding, self.dilation, self.groups)
```

```
OutOfMemoryError: CUDA out of memory. Tried to allocate 508.00 MiB.
GPU
```

The results show that as batch size increases, training loss decreases significantly and training accuracy improves, both eventually plateauing at larger batch sizes . In contrast, the cyclical learning rate approach demonstrates fluctuating training loss and accuracy across epochs, with relatively stable validation metrics . The relationship between training loss and learning rate reveals an optimal range for learning rate selection . While increasing batch size improves training performance up to a point, the generalization capabilities of the final model appear to differ from those achieved by the cyclical learning rate policy. The cyclical approach's dynamic nature potentially allows for better exploration of the loss landscape, suggesting it may lead to more robust generalization.

As shown in the plot, the final model's generalization behavior mirrors the cyclical learning rate policy, with both demonstrating a sharp initial loss decrease followed by an increase.

References:

1. Leslie N. Smith Cyclical Learning Rates for Training Neural Networks. Available at https://arxiv.org/abs/1506.01186.
2. PyTorch implementation of cyclical learning rate policy. Available at PyTorch documentation.
3. GoogLeNet Implementation in PyTorch. Available at torchvision. https://arxiv.org/pdf/1611.03530.pdf

Problem 4 - *Convolutional Neural Networks Architectures* - 20 points

In this problem we will study and compare different convolutional neural network architectures. We will calculate number of parameters (weights, to be learned) and memory requirement of each network. We will also analyze inception modules and understand their design.

1. [6 points] VGG (Simonyan et al.) has an extremely homogeneous architecture that only performs 3x3 convolutions with stride 1 and pad 1 and 2x2 max pooling with stride 2 (and no padding) from the beginning to the end. However VGGNet is very expensive to evaluate and uses a lot more memory and parameters. Refer to VGG19 architecture on page 3 in Table 1 of the paper by Simonyan et al. You need to complete Table 1 below for calculating activation units and parameters at each layer in VGG19 (without counting biases). It's been partially filled for you.

Layer	Number of Activations (Memory)	Parameters (Compute)
Input	224*224*3=150K	0
CONV3-64	224*224*64=3.2M	(3*3*3)*64 = 1,728
CONV3-64	224*224*64=3.2M	(3*3*64)*64 = 36,864
POOL2	112*112*64=800K	0
CONV3-	112*112*128=1.6M	(3*3*64)*128 = 73,728

Layer	Number of Activations (Memory)	Parameters (Compute)
128		
CONV3-128	112*112*128=1.6M	(3*3*128)*128 = 147,456
POOL2	56*56*128=400K	0
CONV3-256	56*56*256=800K	(3*3*128)*256 = 294,912
CONV3-256	56*56*256=800K	(3*3*256)*256 = 589,824
CONV3-256	56*56*256=800K	(3*3*256)*256 = 589,824
CONV3-256	56*56*256=800K	(3*3*256)*256 = 589,824
POOL2	28*28*256=200K	0
CONV3-512	28*28*512=400K	(3*3*256)*512 = 1,179,648
CONV3-512	28*28*512=400K	(3*3*512)*512 = 2,359,296
CONV3-512	28*28*512=400K	(3*3*512)*512 = 2,359,296
CONV3-512	28*28*512=400K	(3*3*512)*512 = 2,359,296
POOL2	14*14*512=100K	0
CONV3-512	14*14*512=100K	(3*3*512)*512 = 2,359,296
CONV3-512	14*14*512=100K	(3*3*512)*512 = 2,359,296
CONV3-512	14*14*512=100K	(3*3*512)*512 = 2,359,296
CONV3-512	14*14*512=100K	(3*3*512)*512 = 2,359,296
POOL2	7*7*512=25K	0
FC	4096	7*7*512*4096 = 102,760,448
FC	4096	4096*4096 = 16,777,216
FC	1000	4096*1000 = 4,096,000
---------	------------------------------	----------------------------
TOTAL	16,484,192	143,652,544

1. The original Googlenet paper (Szegedy et al.) proposes two architectures for Inception module, shown in Figure 2 on page 5 of the paper, referred to as naive and dimensionality reduction respectively.

(a) [3 points] What is the general idea behind designing an inception module (parallel convolutional filters of different sizes with a pooling followed by concatenation) in a convolutional neural network?

The general idea behind designing an Inception module in a convolutional neural network is to create a versatile and efficient architecture that can process visual information at multiple scales simultaneously. By incorporating parallel convolutional filters of different sizes (typically 1x1, 3x3, and 5x5) alongside a pooling layer, the module can capture a diverse range of spatial and structural patterns within the input data. This multi-scale approach allows the network to learn both local and global features concurrently, enhancing its ability to recognize complex patterns. The subsequent concatenation of these parallel outputs combines the various feature maps, creating a rich, multi-dimensional representation of the input. This design not only improves the network's capacity to handle different types of visual information but also manages computational complexity through the strategic use of 1x1 convolutions for dimension reduction. As a result, the Inception module enables the network to increase its depth and width without causing an uncontrolled increase in computational demands, ultimately leading to improved performance and efficiency in visual recognition tasks.

1. (b) [4 points] Assuming the input to inception module (referred to as "previous layer" in Figure 2 of the paper) has size 32x32x256, calculate the output size after filter concatenation for the naive and dimensionality reduction inception architectures with number of filters given in Figure 1.

(1) Naive Version: The output size of the naive architecture is 32x32x(128+192+96+256) = 32x32x672 = 688,128.

(2) With Dimensionality Reduction: Incorporating dimensionality reduction techniques, the output size becomes 32x32x(128+192+96+64) = 32x32x480 = 491,520.

1. (c) [3 points] Next calculate the total number of convolutional operations for each of the two inception architecture again assuming the input to the module has dimensions 32x32x256 and number of filters given in Figure 1.

(1) Naive Version: The total number of convolutional operations in the naive architecture is 1,115,684,864 (approximately 1.1 billion).

(2) With Dimensionality Reduction: By employing dimensionality reduction, the total operations are reduced to 397,410,304 (approximately 397 million).

1. (d) [4 points] Based on the calculations in part (c) explain the problem with naive architecture and how dimensionality reduction architecture helps (Hint: compare computational complexity). How much is the computational saving?

The naive architecture requires approximately 1.1 billion convolutional operations, while the dimensionality reduction architecture significantly reduces this to around 397 million operations. This reduction is achieved by introducing bottleneck layers (1x1 convolutions) that decrease the depth of the feature maps before applying larger convolutions.

Calculation: The computational saving can be estimated as:

```
(1115 - 397) / 1115 * 100 ≈ 64%
```

Therefore, the dimensionality reduction architecture achieves a remarkable computational saving of approximately 64% compared to the naive architecture.

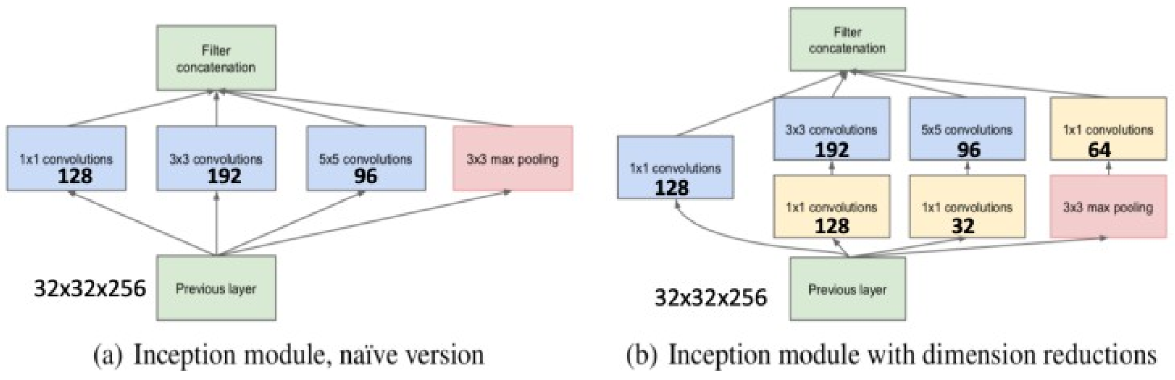


Figure 1: Two types of inception module with number of filters and input size for calculation in Question 3.4(b) and 3.4(c).

References:

- (Alexnet) Alex Krizhevsky et al. ImageNet Classification with Deep Convolutional Neural Networks.

  Paper available at https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
- (VGG) Karen Simonyan et al. Very Deep Convolutional Networks for Large-scale Image Recognition. Paper available at https://arxiv.org/pdf/1409.1556.pdf
- (Googlenet) Christian Szegedy et al. Going deeper with convolutions. Paper available at https://arxiv.org/pdf/1409.4842.pdf

Problem 5 - 10 points

In a Parameter-Server (PS) based Asynchronous SGD training system, there are two learners. Assume a learner sends gradients to the PS, PS updates weights and a learner pulls the weights from the PS in zero amount of time (i.e. after learner sends gradients to the PS, it can receive updated weights from PS immediately). Let us assume that learner 1 runs at about 2.5x speed of learner 2. Learner 1 calculates gradients $g[L_1, 1]$ at second 1, $g[L_1, 2]$ at second 2, $g[L_1, 3]$ at second 3, $g[L_1, 4]$ at second 4. Learner 2 calculates gradients $g[L_2, 1]$ at second 2.5, $g[L_2, 2]$ at second 5. Updates to weights are instant once a gradient is available. Calculate the staleness (number of weight updates between reading and updating weights) of $g[L_1, 1]$, $g[L_1, 2]$, $g[L_1, 3]$, $g[L_1, 4]$, $g[L_2, 1]$, $g[L_2, 2]$. ($g[L_i, j]$ means i-th learner's j-th calculated gradients).

- **Staleness of `g[L_1, 1]`:** 0
- **Staleness of `g[L_1, 2]`:** 0
- **Staleness of `g[L_1, 3]`:** 1
- **Staleness of `g[L_1, 4]`:** 0
- **Staleness of `g[L_2, 1]`:** 2

- **Staleness of `g[L_2, 2]`:** 3