

# COMS6998\_015 Prac Deep Learning System HW1

Name: Allen Chen

UNI #: atc2160

Notes:

- See the end of this notebook for Problem 3 code.
- Everything used to produce this notebook:  
<https://github.com/AllenChenCU/coms6998-dl-sys/tree/main/hw1>

## Problem 1 - Bias Variance Tradeoff, Regularization

### 1.1 Derive the bias-variance decomposition for a regression problem

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

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

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

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

$$\mathbb{E}[\text{MSE}] = \frac{1}{t} \mathbb{E}\left[\sum_{i=1}^t (f(x_i) - g(x_i))^2\right] + \sigma^2$$

$$\mathbb{E}[\text{MSE}] = \frac{1}{t} \mathbb{E}\left[\sum_{i=1}^t (f(x_i) - \mathbb{E}[g(x_i)] + \mathbb{E}[g(x_i)] - g(x_i))^2\right] + \sigma^2$$

$$\mathbb{E}[\text{MSE}] = \frac{1}{t} \mathbb{E}\left[\sum_{i=1}^t (f(x_i) - \mathbb{E}[g(x_i)])^2 + 2(f(x_i) - \mathbb{E}[g(x_i)])(\mathbb{E}[g(x_i)] - g(x_i)) + (\mathbb{E}[g(x_i)] - g(x_i))^2\right] + \sigma^2$$

$$\mathbb{E}[\text{MSE}] = \frac{1}{t} \mathbb{E}\left[\sum_{i=1}^t (f(x_i) - \mathbb{E}[g(x_i)])^2 + (\mathbb{E}[g(x_i)] - g(x_i))^2\right] + \sigma^2$$

$$\mathbb{E}[\text{MSE}] = \frac{1}{t} \sum_{i=1}^t (f(x_i) - \mathbb{E}[g(x_i)])^2 + \frac{1}{t} \mathbb{E}\left[\sum_{i=1}^t (\mathbb{E}[g(x_i)] - g(x_i))^2\right] + \sigma^2$$

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

### 1.2 Scatter plot for y and f(x)

$$y(x) = x + \sin(1.5x) + N(0, 0.3)$$

$$f(x) = x + \sin(1.5x)$$

```
In [1]: %load_ext autoreload
%autoreload 2

import numpy as np
import pandas as pd
import random
import math
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

random.seed(42)

def y_func(x, mu, sigma):
    noise = random.gauss(mu=mu, sigma=sigma)
    return x + math.sin(1.5*x) + noise

def f_func(x):
    return x + math.sin(1.5*x)

MU = 0
SIGMA = math.sqrt(0.3)
N = 20
XMAX = 10

X = [random.randint(0, XMAX) for _ in range(N)]
Y = [y_func(x, MU, SIGMA) for x in X]
F = [f_func(x) for x in X]

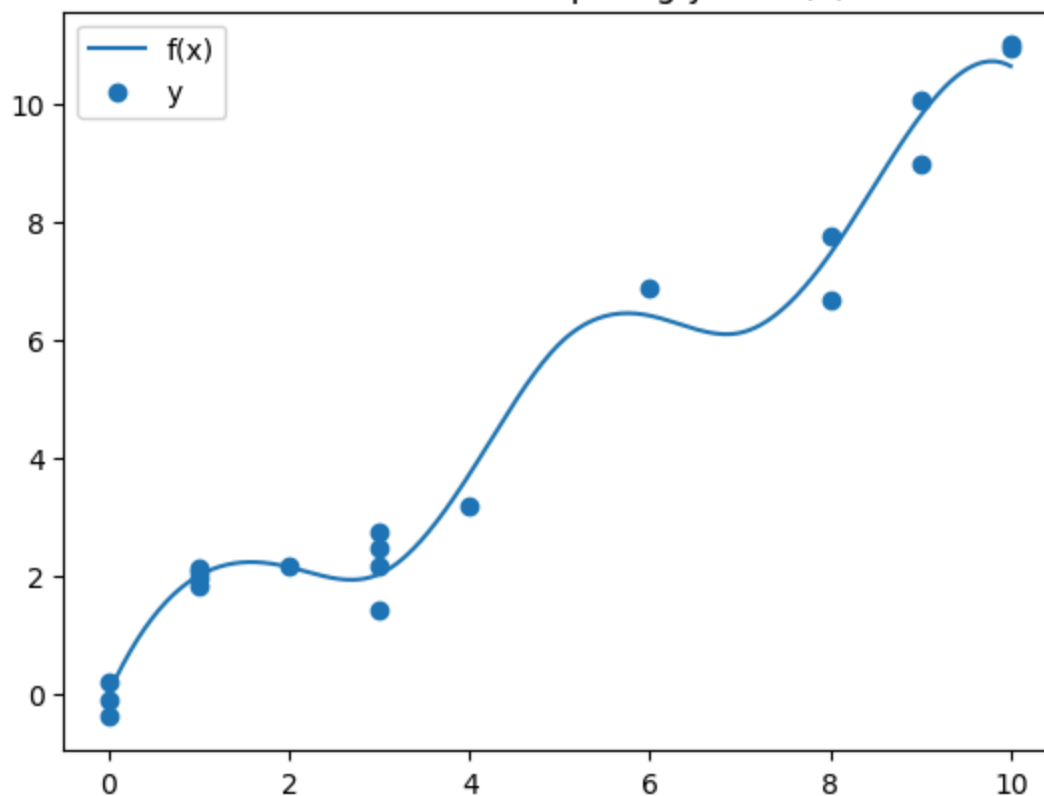
# display dataset
dataset = pd.DataFrame({"x": X, "y": Y, "f(x)": F})
print(dataset)

# Plot
X_1 = np.arange(XMAX + 1)
F_1 = [f_func(x) for x in X_1]
x_smooth = np.linspace(0, XMAX, 500)
smooth_func = interp1d(X_1, F_1, kind='cubic')

plt.title("Problem 1.2: Comparing y and f(x)")
plt.plot(x_smooth, smooth_func(x_smooth), label="f(x)")
plt.scatter(X, Y, label="y")
plt.legend()
plt.show()
```

	x	y	f(x)
0	10	11.009942	10.650288
1	1	2.058022	1.997495
2	0	-0.404395	0.000000
3	4	3.164831	3.720585
4	3	2.157397	2.022470
5	3	2.740578	2.022470
6	2	2.163936	2.141120
7	1	1.939259	1.997495
8	10	10.941554	10.650288
9	8	6.667288	7.463427
10	1	1.826454	1.997495
11	9	10.072367	9.803784
12	6	6.890502	6.412118
13	0	-0.131798	0.000000
14	0	0.206272	0.000000
15	1	2.133447	1.997495
16	3	2.450968	2.022470
17	3	1.412733	2.022470
18	8	7.774671	7.463427
19	9	8.974247	9.803784

Problem 1.2: Comparing y and f(x)



### 1.3

$$g_n(x) = B_0 + B_1x + B_2x^2 + \dots + B_nx^n$$

```
In [2]: from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
```

```

x_train = dataset[["x"]]
y_train = dataset["y"].values
f_train = dataset["f(x)"].values

# fit estimators
model_poly_1 = LinearRegression().fit(x_train, y_train)
dataset["g_1"] = model_poly_1.predict(x_train)
print(f"Model g_1 intercept: {model_poly_1.intercept_}")
print(f"Model g_1 coefficients: {model_poly_1.coef_}")

x_train_poly_3 = PolynomialFeatures(degree=3).fit_transform(x_train)
model_poly_3 = LinearRegression().fit(x_train_poly_3, y_train)
dataset["g_3"] = model_poly_3.predict(x_train_poly_3)
print(f"Model g_3 intercept: {model_poly_3.intercept_}")
print(f"Model g_3 coefficients: {model_poly_3.coef_}")

x_train_poly_10 = PolynomialFeatures(degree=10).fit_transform(x_train)
model_poly_10 = LinearRegression().fit(x_train_poly_10, y_train)
dataset["g_10"] = model_poly_10.predict(x_train_poly_10)
print(f"Model g_10 intercept: {model_poly_10.intercept_}")
print(f"Model g_10 coefficients: {model_poly_10.coef_}")

# plot
X_1 = np.arange(XMAX + 1)
F_1 = [f_func(x) for x in X_1]
x_smooth = np.linspace(0, XMAX, 500)
smooth_func = interp1d(X_1, F_1, kind='cubic')
dataset.sort_values(["x"], inplace=True)

plt.title("Problem 1.3")
plt.plot(x_smooth, smooth_func(x_smooth), label="f(x)")
plt.scatter(dataset['x'], dataset['y'], label="y(x)")
plt.scatter(dataset['x'], dataset['g_1'], label="g_1(x)")
plt.scatter(dataset['x'], dataset['g_3'], label="g_3(x)")
plt.scatter(dataset['x'], dataset['g_10'], label="g_10(x)")
plt.legend()
plt.show()

```

Model g\_1 intercept: 0.026180552877201002

Model g\_1 coefficients: [1.01859347]

Model g\_3 intercept: 0.3929045262658426

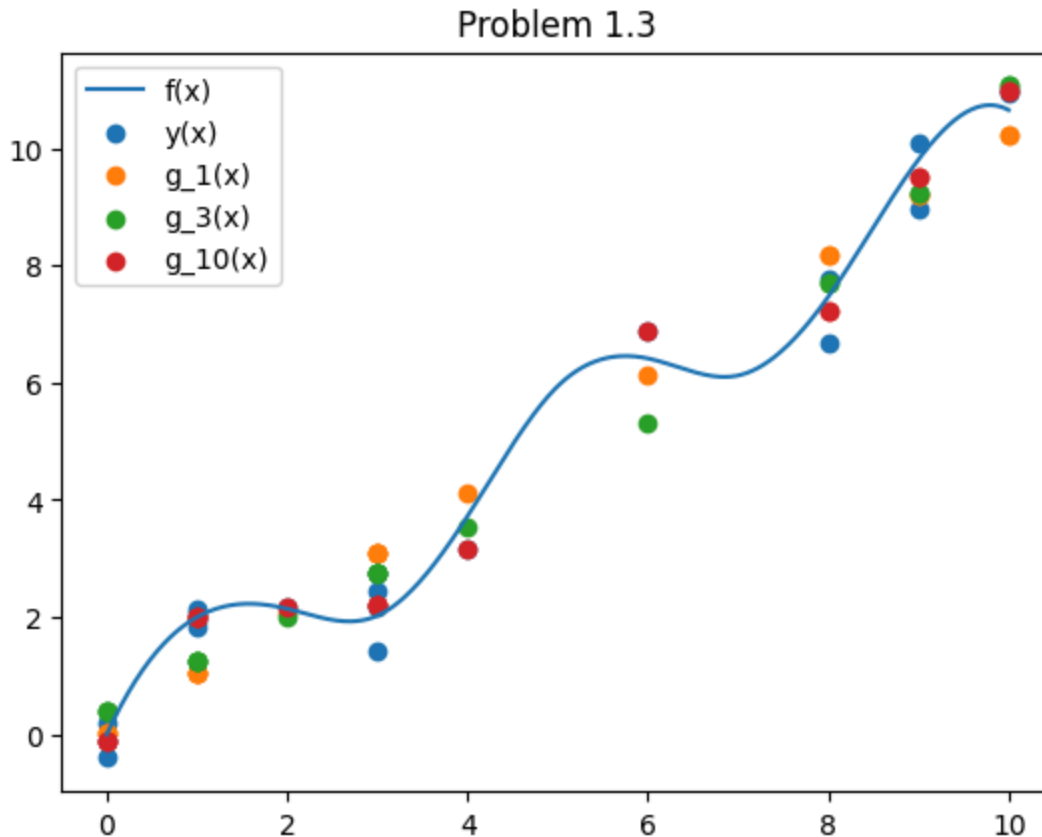
Model g\_3 coefficients: [ 0. 0.89013749 -0.055449 0.00731094]

Model g\_10 intercept: -0.10997356210055464

Model g\_10 coefficients: [ 0.00000000e+00 2.39270701e+00 5.71449973e-01 -  
7.67018200e-01

-5.73316016e-01 7.01805044e-01 -2.76610179e-01 5.62215600e-02

-6.33669879e-03 3.75754822e-04 -9.14406761e-06]



Estimator  $g_1$  is underfitting, and estimator  $g_{10}$  is overfitting.

## 1.4

$$Bias^2 = \frac{1}{t} \sum_{i=1}^t (f(x_i) - \mathbb{E}[g(x_i)])^2$$

The term  $f(x_i)$  is the truth value at  $x_i$

The Term expected value of the prediction  $\mathbb{E}[g(x_i)]$  is the average of the prediction for  $x_i$  across all possible datasets

$$Variance = \frac{1}{t} \mathbb{E}[\sum_{i=1}^t (\mathbb{E}[g(x_i)] - g(x_i))^2]$$

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

```
In [3]: from collections import defaultdict
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import Ridge

# config
MU = 0
SIGMA = math.sqrt(0.3)
N = 50 # dataset size
N_test = 10
N_train = 40
```

```

XMAX = 10
M = 100 # number of datasets
DEGREE = 15 # try polynomial regression up to degree DEGREE

random.seed(21)
np.random.seed(21)

x = np.linspace(0, XMAX, N)
x = np.random.permutation(x)
X_train = np.array(x[:N_train])
X_test = np.array(x[N_train:])

def simulate(degree, X_train, X_test, regularize=False):

    y_preds = []
    y_tests = []

    for _ in range(M):
        # Generate dataset
        y_train = np.array([y_func(x, MU, SIGMA) for x in X_train])
        y_test = np.array([y_func(x, MU, SIGMA) for x in X_test])

        # Transform dataset
        poly_transformer = PolynomialFeatures(degree=degree)
        X_train_poly = poly_transformer.fit_transform(np.expand_dims(X_train, axis=0))
        X_test_poly = poly_transformer.transform(np.expand_dims(X_test, axis=0))

        # Train
        if regularize:
            model = Ridge(alpha=0.5).fit(X_train_poly, y_train)
        else:
            model = LinearRegression().fit(X_train_poly, y_train)

        # Predict
        y_pred = model.predict(X_test_poly)
        y_preds.append(y_pred)
        y_tests.append(y_test)

    y_preds = np.stack(y_preds, axis=0)
    y_tests = np.stack(y_tests, axis=0)
    f = np.array([f_func(x) for x in X_test])

    # compute MSE
    mse = mean_squared_error(y_tests, y_preds)

    # compute bias-squared
    exp_preds = np.mean(y_preds, axis=0)

    bias_squared = np.mean(np.square(f - exp_preds))
    variance = np.mean(np.mean(np.square(exp_preds - y_preds), axis=0))
    computed_noise = mse - (bias_squared + variance)

    return {
        "mse": float(mse),
        "bias_squared": float(bias_squared),
        "variance": float(variance),
    }

```

```

        "computed_noise": float(computed_noise),
    }

results_map = {}

for d in range(1, DEGREE + 1):
    results_map[f"g_{d}"] = simulate(d, X_train, X_test)

results_df = pd.DataFrame(results_map)
display(results_df)
complexity = np.arange(1, 16)

# plot
plt.title("Model complexity vs Error")
plt.plot(complexity, results_df.T["mse"].values, label="mse")
plt.plot(complexity, results_df.T["bias_squared"].values, label="bias_squared")
plt.plot(complexity, results_df.T["variance"].values, label="variance")
plt.legend()
plt.show()

```

	g_1	g_2	g_3	g_4	g_5	g_6	g_7
mse	1.190326	1.000286	1.032534	0.917501	0.990884	0.526866	0.46559
bias_squared	0.833505	0.654311	0.685812	0.619258	0.654237	0.166760	0.10789
variance	0.013945	0.015273	0.028701	0.038775	0.035863	0.054237	0.06135
computed_noise	0.342875	0.330703	0.318021	0.259469	0.300784	0.305870	0.29635



The Best model has a polynomial degree of 8, where it has the lowest MSE and the optimal balance between bias\_squared and variance.

## 1.5 Regularization

```
In [5]: import warnings
warnings.filterwarnings("ignore")

results_map = {}

results_map[f"g_10"] = simulate(10, X_train, X_test, False)
results_map[f"g_10_L2"] = simulate(10, X_train, X_test, True)

results_df = pd.DataFrame(results_map)
results_df
```

```
Out[5]:
```

	g_10	g_10_L2
mse	0.408168	0.376128
bias_squared	0.001975	0.015525
variance	0.090800	0.077387
computed_noise	0.315392	0.283216

Going from the unregularized order 10 polynomial model to regularized model,

1. MSE decreased slightly
2.  $Bias^2$  increased
3. Variance decreased

Therefore, the regularized model has a higher bias, lower variance, and lower MSE.

This is because the regularized model penalized complex model with larger coefficients, so that the model doesn't fit too closely to the data. This leads to better generalization to unseen data. Thus, it results in lower MSE on testing dataset. Bias increased because regularization simplified the model, and variance decreased because it limited the model's complexity.

## Problem 2 - Precision, Recall, ROC

### 2.1

Q: 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.

A: True negative is important in the calculation of false positive rate (FPR), which is the x-axis of the ROC curve. However, true negative does not matter for PR curve since



precision and recall metrics don't use true negative in their calculation. Each point on ROC curve corresponds to an unique point on PR curve because both corresponding points depend on the same set of predictions at a certain threshold. These predictions at a certain threshold form a specific confusion matrix, which consists of TP, TN, FP, FN that are directly used to calculate precision, recall, false positive rate, and true positive rate. Precision and recalls are used to plot PR curve, and FPR and TPR are used to plot ROC curve.

## 2.2

```
In [6]: from scipy.io import arff
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, precision_recall_curve
from sklearn.metrics import PrecisionRecallDisplay
from sklearn.metrics import auc
from sklearn.metrics._ranking import _binary_clf_curve

# load data
arff_file = arff.loadarff('dataset_37_diabetes.arff')
prob2_2_data = pd.DataFrame(arff_file[0])
prob2_2_data['class'] = prob2_2_data['class'].map({b"tested_positive": 1, b"tested_negative": 0})
X = prob2_2_data[[col for col in prob2_2_data.columns if col != 'class']]
y = prob2_2_data['class'].values

# split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

def make_pipe(model_name, model):
    return Pipeline([("scaler", StandardScaler()), (model_name, model)])

results_map = {}
models = [
    ("adaboost", AdaBoostClassifier(n_estimators=100, algorithm="SAMME", random_state=42)),
    ("log_reg", LogisticRegression(random_state=42)),
]

for model_name, model in models:
    pipe = make_pipe(model_name, model)
    pipe.fit(X_train, y_train)
    y_pred = pipe.predict_proba(X_test)[:, 1]
    fpr, tpr, thresholds_roc = roc_curve(y_test, y_pred)
    precision, recall, thresholds_pr = precision_recall_curve(y_test, y_pred)
    auROC = auc(fpr, tpr)
    aupR = auc(recall, precision)
    pi = 268/768 # proportion of positives in dataset
    recallG = [float((r - pi) / ((1-pi)*r + 0.0001)) for r in recall]
    precisionG = [float((p - pi) / ((1-pi)*p + 0.0001)) for p in precision]
    aupRG = auc(recallG, precisionG)
    results_map[model_name] = {
        'auROC': auROC,
        'aupR': aupR,
        'aupRG': aupRG
    }
```

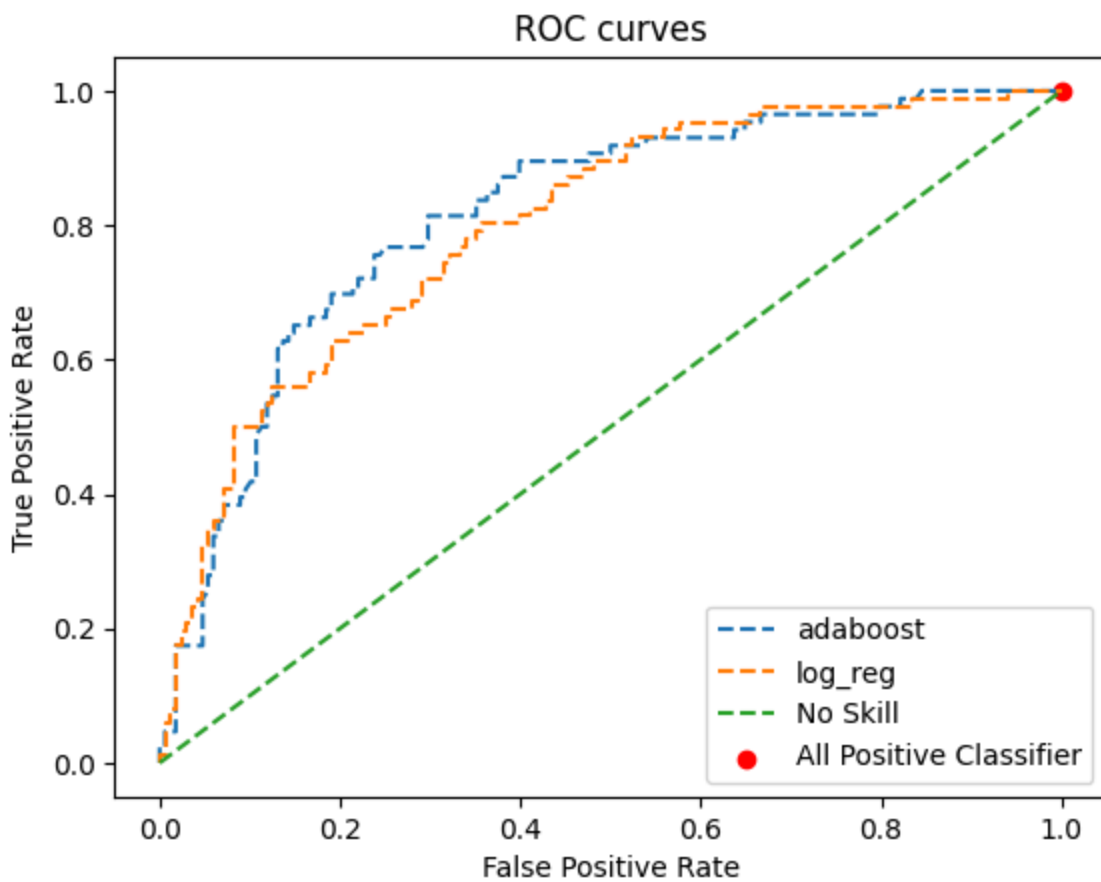
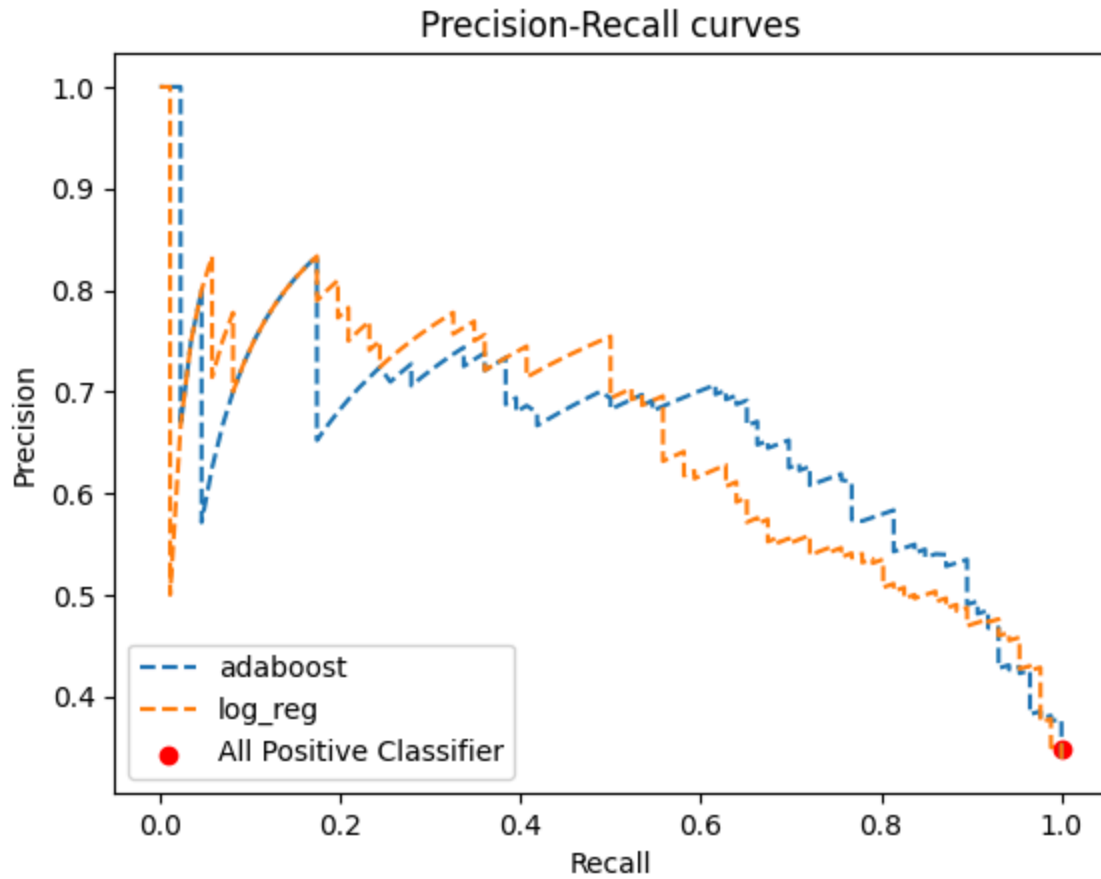
```

        "fpr": fpr,
        "tpr": tpr,
        "precision": precision,
        "recall": recall,
        "thresholds_pr": thresholds_pr,
        "thresholds_roc": thresholds_roc,
        "auroc": auroc,
        "aupr": aupr,
        "auprg": auprg,
    }

plt.title("Precision-Recall curves")
plt.plot(results_map["adaboost"]["recall"], results_map["adaboost"]["precision"])
plt.plot(results_map["log_reg"]["recall"], results_map["log_reg"]["precision"])
plt.scatter([1], [1], label="All Positive Classifier", color="red")
plt.ylabel("Precision")
plt.xlabel("Recall")
plt.legend(loc="lower left")
plt.show()

plt.title("ROC curves")
plt.plot(results_map["adaboost"]["fpr"], results_map["adaboost"]["tpr"], '--')
plt.plot(results_map["log_reg"]["fpr"], results_map["log_reg"]["tpr"], '--')
plt.plot(np.linspace(0, 1, 100), np.linspace(0, 1, 100), '--', label="No Skill")
plt.scatter([1], [1], label="All Positive Classifier", color="red")
plt.ylabel("True Positive Rate")
plt.xlabel("False Positive Rate")
plt.legend()
plt.show()

```



Note: For all positive classifier, we have

$$\text{TPR} = 1; \text{FPR} = 1; \text{Precision} = \frac{P}{(P+N)} = \frac{268}{768}; \text{Recall} = 1$$

## 2.3 PR gain curve

```
In [7]: print(f"AUROC for adaboost {results_map["adaboost"]["auroc"]}")
print(f"AUPR for adaboost {results_map["adaboost"]["aupr"]}")
print(f"AUPRG for adaboost {results_map["adaboost"]["auprg"]}")

print(f"AUROC for logistic regression {results_map["log_reg"]["auroc"]}")
print(f"AUPR for logistic regression {results_map["log_reg"]["aupr"]}")
print(f"AUPRG for logistic regression {results_map["log_reg"]["auprg"]}")
```

```
AUROC for adaboost 0.8160991140642303
AUPR for adaboost 0.6583825843463507
AUPRG for adaboost 3484.9656016837307
AUROC for logistic regression 0.7994878183831672
AUPR for logistic regression 0.6522517882377267
AUPRG for logistic regression 3476.754018649303
```

I disagree that PR gain curves should be used over PR curves. There could be unstable values that fall into the negative range when calculating the precision gain and recall gain values. In that case, my AUPRG value ends up becoming larger than 1.

## Problem 3 - Learning Rate, Batch Size, FashionMNIST

Note: Code at the end of this notebook.

### 3.1

```
In [8]: import json

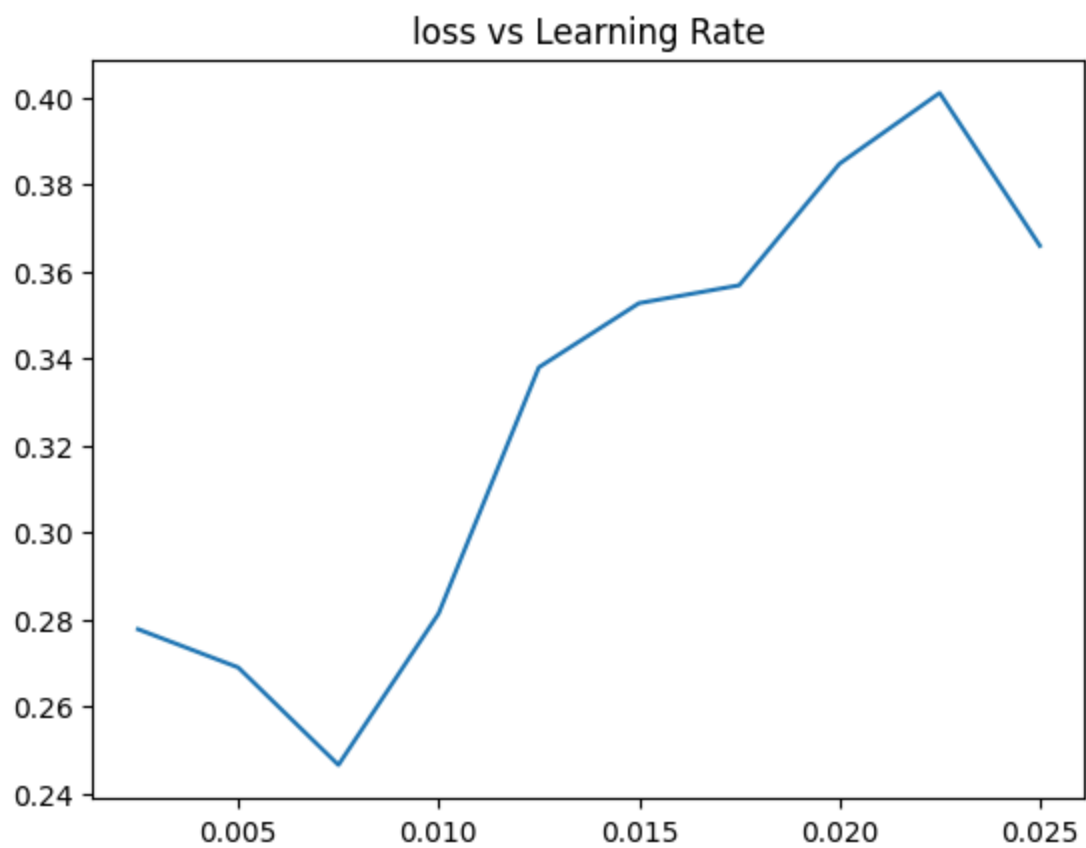
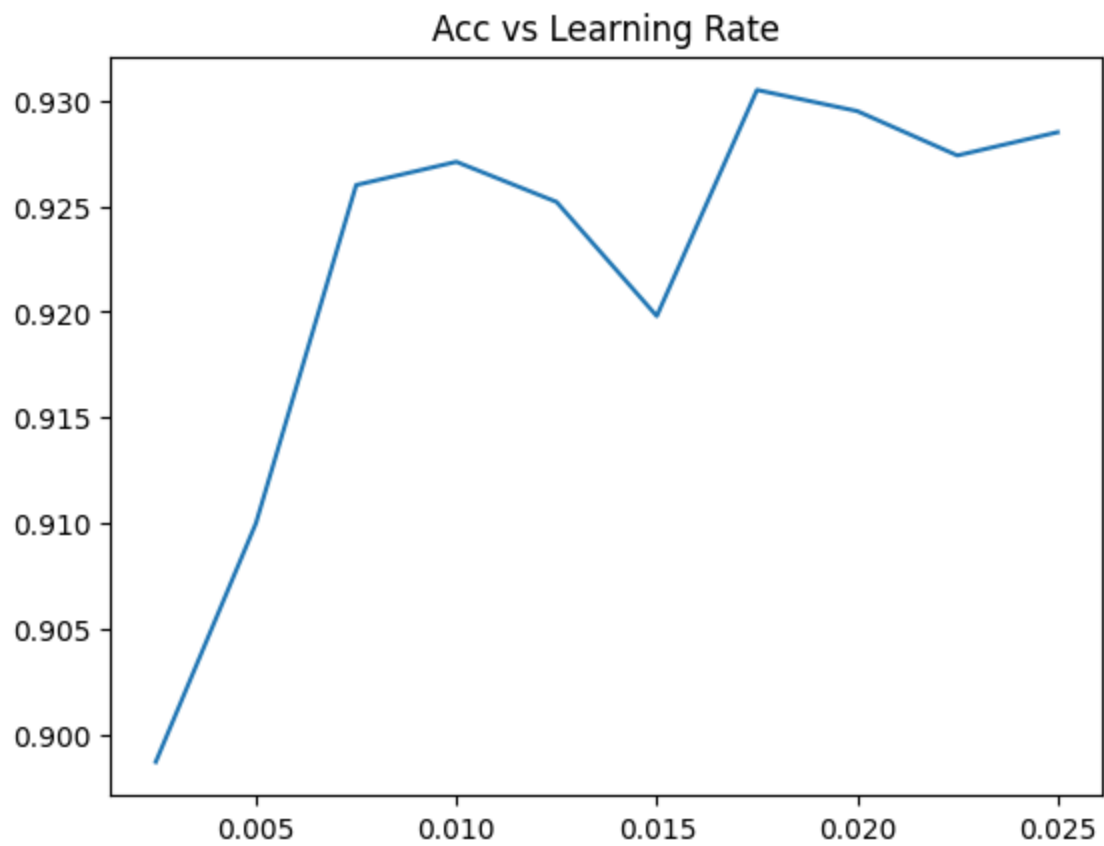
with open("prob3_1_results_1.json", "r") as f:
    prob3_1_results = json.load(f)

lrs = []
accs = []
losses = []
for i, results in prob3_1_results.items():
    lrs.append(results["lr"])
    accs.append(results["val_acc"][-1])
    losses.append(results["val_loss"][-1])

prob3_1_results_df = pd.DataFrame({"lr": lrs, "acc": accs, "loss": losses})
prob3_1_results_df.sort_values(["lr"], inplace=True)

plt.title("Acc vs Learning Rate")
plt.plot(prob3_1_results_df["lr"], prob3_1_results_df["acc"])
plt.show()
```

```
plt.title("loss vs Learning Rate")  
plt.plot(prob3_1_results_df["lr"], prob3_1_results_df["loss"])  
plt.show()
```



$lr_{min} = 0.0025$  $lr_{max} = 0.01$ 

## 3.2

```
In [9]: with open("prob3_2_results_1.json", "r") as f:
        prob3_2_results = json.load(f)

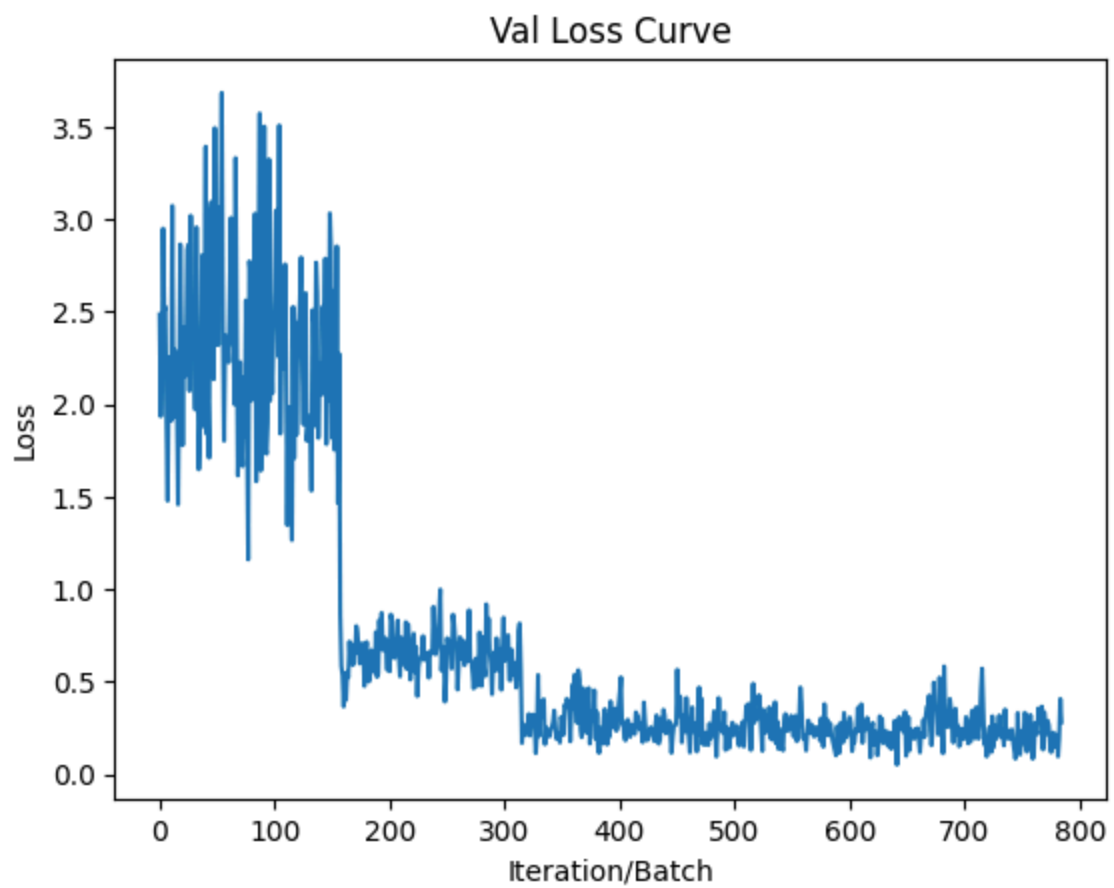
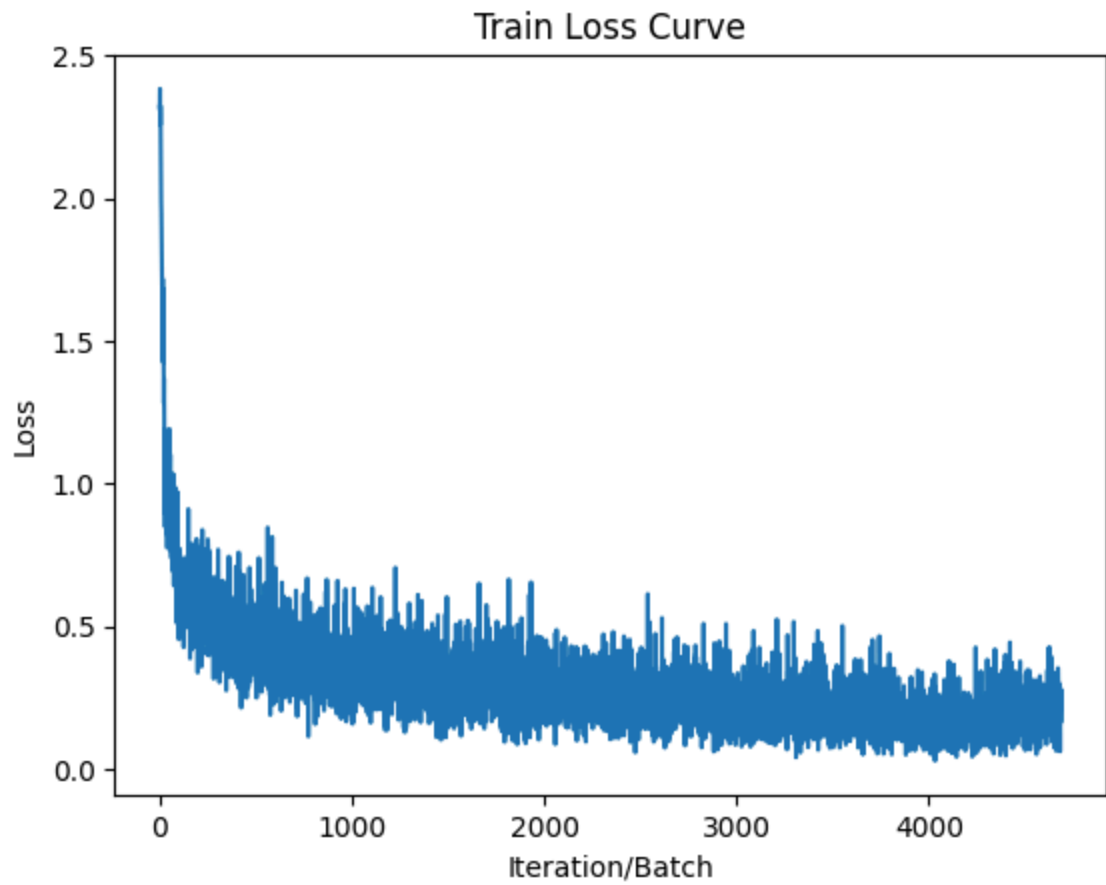
        num_iteration_train = np.arange(len(prob3_2_results["train_batch_losses"]))
        num_iteration_val = np.arange(len(prob3_2_results["val_batch_losses"]))

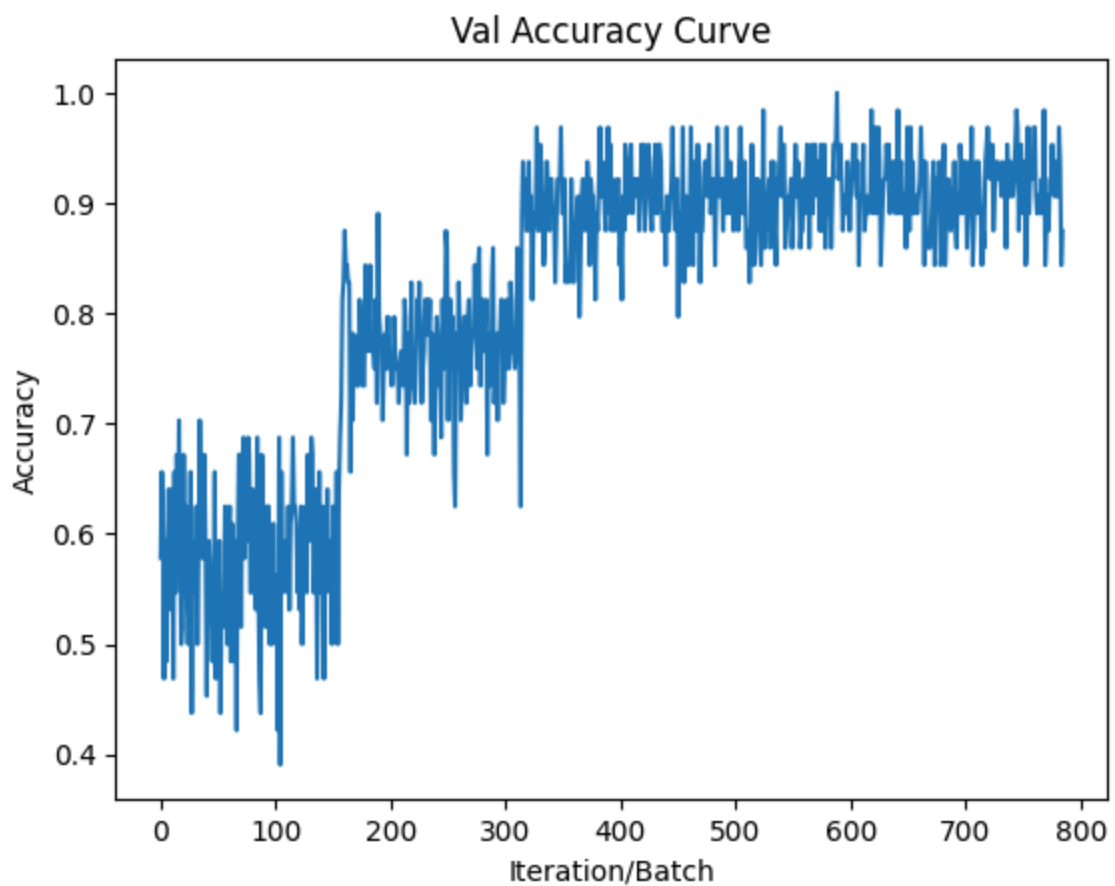
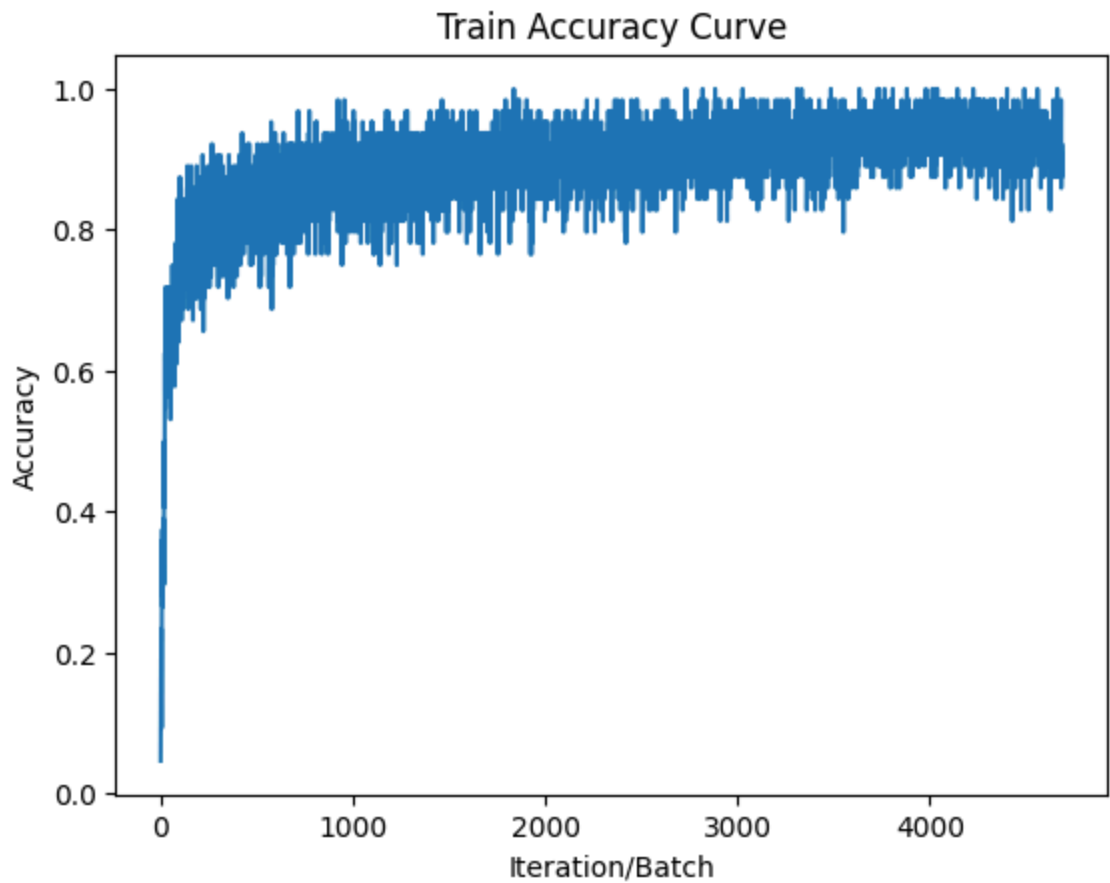
        plt.title("Train Loss Curve")
        plt.plot(num_iteration_train, prob3_2_results["train_batch_losses"])
        plt.xlabel("Iteration/Batch")
        plt.ylabel("Loss")
        plt.show()

        plt.title("Val Loss Curve")
        plt.plot(num_iteration_val, prob3_2_results["val_batch_losses"])
        plt.xlabel("Iteration/Batch")
        plt.ylabel("Loss")
        plt.show()

        plt.title("Train Accuracy Curve")
        plt.plot(num_iteration_train, prob3_2_results["train_batch_accs"])
        plt.xlabel("Iteration/Batch")
        plt.ylabel("Accuracy")
        plt.show()

        plt.title("Val Accuracy Curve")
        plt.plot(num_iteration_val, prob3_2_results["val_batch_accs"])
        plt.xlabel("Iteration/Batch")
        plt.ylabel("Accuracy")
        plt.show()
```





### 3.3



```
In [10]: with open("prob3_3_results.json", "r") as f:
    prob3_3_results = json.load(f)

    train_losses = prob3_3_results["train_loss"]
    batch_size = 32
    batch_sizes = []
    while batch_size <= 8192:
        batch_sizes.append(batch_size)
        batch_size *= 2

    # each epoch uses one batch_size
    # epoch0: 32
    # epoch1: 64
    # ...
    plt.title("Training Loss vs Batch size")
    plt.plot(batch_sizes, train_losses)
    plt.xlabel("Batch Size")
    plt.ylabel("Loss")
    plt.show()
```



The final model from problem 3.3 as shown in the plot above is similar to the Training Loss curve shown in the plot in Problem 3.2 with cyclical learning rate policy. They both show similar steep downward trend.

# Problem 4 - Convolutional Neural Networks Architectures

## 4.1

Table 1. VGG19 Activation units and paramteters calculation

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
Conv3-128	$112 * 112 * 128 = 1.6M$	$(3 * 3 * 64) * 128 = 73,728$
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	16.5M	143,652,544

## 4.2

(a) According to the Googlenet paper, the general idea behind designing an inception module in a CNN is based on finding out how an optimal local sparse structure in a convolutional vision network can be approximated and covered by readily available dense components.

(b) Calculate output sizes

Naive version:

Concatenate  $(32 \times 32 \times 128) + (32 \times 32 \times 192) + (32 \times 32 \times 96) + (32 \times 32 \times 256)$  at the channel/filter dimension  $\Rightarrow (32 \times 32 \times 672)$

Dimension reductions version:

Concatenate  $128 + 192 + 96 + 64 \Rightarrow (32 \times 32 \times 480)$

(c) Calculate number of convolutional operations

Naive version:

$$(1 * 1 * 256 * 128) + (3 * 3 * 256 * 192) + (5 * 5 * 256 * 96) + 0$$

$$= 32768 + 442368 + 614400$$

$$= 1,089,536$$

Dimension reductions version:

$$(1 * 1 * 256 * 128) + (1 * 1 * 256 * 128) + (3 * 3 * 128 * 192) + (1 * 1 * 256 * 32) + (5 * 5 * 32 * 96) + 0 + (1 * 1 * 256 * 64)$$

$$= 388,096$$

(d) How dimension reductions technique helps?

The inception module with dimension reductions technique is significantly faster than the naive architecture with much lower computational steps. The number of convolutional computation decreased from 1,089,536 to 388,096, which is a 701,440 operations reduction or 64.4% decrease.

## Problem 5

Staleness at the start of the compute of the following gradients:

$$g[L_1, 1] = 0$$

$$g[L_1, 2] = 0$$

$$g[L_1, 3] = 1$$

$$g[L_1, 4] = 0$$

$$g[L_2, 1] = 2$$

$g[L_2, 2] = 0$  (Since PS receives gradients and updates weights from  $g[L_1, 4]$  and  $g[L_2, 1]$  at the same time at second 5. The gradients read at the beginning of the compute of  $g[L_2, 2]$  is not stale)

## Reference

```
In [ ]: import warnings
from collections import namedtuple
from functools import partial
from typing import Any, Callable, List, Optional, Tuple
import json

import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
from torch import Tensor
from tqdm import tqdm
import matplotlib.pyplot as plt
import numpy as np
import torchvision
import torchvision.transforms as transforms
from torchvision.transforms._presets import ImageClassification
from torchvision.utils import _log_api_usage_once
from torchvision.models._api import register_model, Weights, WeightsEnum
from torchvision.models._meta import _IMAGENET_CATEGORIES
from torchvision.models._utils import _overwrite_named_param, handle_legacy_i

__all__ = ["GoogLeNet", "GoogLeNetOutputs", "_GoogLeNetOutputs", "GoogLeNet_

GoogLeNetOutputs = namedtuple("GoogLeNetOutputs", ["logits", "aux_logits2",
GoogLeNetOutputs.__annotations__ = {"logits": Tensor, "aux_logits2": Optiona

# Script annotations failed with _GoogLeNetOutputs = namedtuple ...
# _GoogLeNetOutputs set here for backwards compat
_GoogLeNetOutputs = GoogLeNetOutputs

class GoogLeNet(nn.Module):
    __constants__ = ["aux_logits", "transform_input"]

    def __init__(
        self,
        num_classes: int = 1000,
        aux_logits: bool = True,
```

```

transform_input: bool = False,
init_weights: Optional[bool] = None,
blocks: Optional[List[Callable[..., nn.Module]]] = None,
dropout: float = 0.2,
dropout_aux: float = 0.7,
) -> None:
    super().__init__()
    _log_api_usage_once(self)
    if blocks is None:
        blocks = [BasicConv2d, Inception, InceptionAux]
    if init_weights is None:
        warnings.warn(
            "The default weight initialization of GoogleNet will be changed to 'kaiming_uniform' in "
            "torchvision. If you wish to keep the old behavior (which leads to some "
            "due to scipy/scipy#11299), please set init_weights=True.",
            FutureWarning,
        )
        init_weights = True
    if len(blocks) != 3:
        raise ValueError(f"blocks length should be 3 instead of {len(blocks)}")
    conv_block = blocks[0]
    inception_block = blocks[1]
    inception_aux_block = blocks[2]

    self.aux_logits = aux_logits
    self.transform_input = transform_input

    self.conv0 = nn.Conv2d(in_channels=1, out_channels=3, kernel_size=1)
    self.conv1_module = conv_block(3, 96, kernel_size=3, stride=1)
    self.inception2_1_module = InceptionModule(96, 32, 32)
    self.inception2_2_module = InceptionModule(64, 32, 48)
    self.downsample2_module = DownsampleModule(80, 80, 1)
    self.inception3_1_module = InceptionModule(160, 112, 48)
    self.inception3_2_module = InceptionModule(160, 96, 64)
    self.inception3_3_module = InceptionModule(160, 80, 80)
    self.inception3_4_module = InceptionModule(160, 48, 96)
    self.downsample3_module = DownsampleModule(144, 96, 0)
    self.inception4_1 = InceptionModule(240, 176, 160)
    self.inception4_2 = InceptionModule(336, 176, 160)
    #self.mean_pooling4 = nn.AvgPool2d(7)
    self.fc4 = nn.Linear(336, num_classes)

    self.conv1 = conv_block(3, 64, kernel_size=7, stride=2, padding=3)
    self.maxpool1 = nn.MaxPool2d(3, stride=2, ceil_mode=True)
    self.conv2 = conv_block(64, 64, kernel_size=1)
    self.conv3 = conv_block(64, 192, kernel_size=3, padding=1)
    self.maxpool2 = nn.MaxPool2d(3, stride=2, ceil_mode=True)

    self.inception3a = inception_block(192, 64, 96, 128, 16, 32, 32)
    self.inception3b = inception_block(256, 128, 128, 192, 32, 96, 64)
    self.maxpool3 = nn.MaxPool2d(3, stride=2, ceil_mode=True)

    self.inception4a = inception_block(480, 192, 96, 208, 16, 48, 64)
    self.inception4b = inception_block(512, 160, 112, 224, 24, 64, 64)
    self.inception4c = inception_block(512, 128, 128, 256, 24, 64, 64)
    self.inception4d = inception_block(512, 112, 144, 288, 32, 64, 64)

```

```

self.inception4e = inception_block(528, 256, 160, 320, 32, 128, 128)
self.maxpool4 = nn.MaxPool2d(2, stride=2, ceil_mode=True)

self.inception5a = inception_block(832, 256, 160, 320, 32, 128, 128)
self.inception5b = inception_block(832, 384, 192, 384, 48, 128, 128)

if aux_logits:
    self.aux1 = inception_aux_block(512, num_classes, dropout=dropout)
    self.aux2 = inception_aux_block(528, num_classes, dropout=dropout)
else:
    self.aux1 = None # type: ignore[assignment]
    self.aux2 = None # type: ignore[assignment]

self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.dropout = nn.Dropout(p=dropout)
self.fc = nn.Linear(1024, num_classes)

if init_weights:
    for m in self.modules():
        if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
            torch.nn.init.trunc_normal_(m.weight, mean=0.0, std=0.01)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)

def _transform_input(self, x: Tensor) -> Tensor:
    if self.transform_input:
        x_ch0 = torch.unsqueeze(x[:, 0], 1) * (0.229 / 0.5) + (0.485 - 0.5)
        x_ch1 = torch.unsqueeze(x[:, 1], 1) * (0.224 / 0.5) + (0.456 - 0.5)
        x_ch2 = torch.unsqueeze(x[:, 2], 1) * (0.225 / 0.5) + (0.406 - 0.5)
        x = torch.cat((x_ch0, x_ch1, x_ch2), 1)
    return x

def forward(self, x: Tensor) -> Tensor: #Tuple[Tensor, Optional[Tensor]],

    # N x 1 x 28 x 28
    x = self.conv0(x)
    # N x 3 x 28 x 28
    x = self.conv1_module(x)
    x = self.inception2_1_module(x)
    x = self.inception2_2_module(x)
    x = self.downsample2_module(x)
    x = self.inception3_1_module(x)
    x = self.inception3_2_module(x)
    x = self.inception3_3_module(x)
    x = self.inception3_4_module(x)
    x = self.downsample3_module(x)
    x = self.inception4_1(x)
    x = self.inception4_2(x)
    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.dropout(x)
    x = self.fc4(x)
    return x

```

```

class InceptionModule(nn.Module):
    def __init__(
        self,
        in_channel,
        ch1,
        ch3
    ):
        super().__init__()
        self.conv_module1 = BasicConv2d(in_channel, ch1, kernel_size=1, stride=1)
        self.conv_module2 = BasicConv2d(in_channel, ch3, kernel_size=3, stride=1)

    def forward(self, x: Tensor) -> Tensor:
        x1 = self.conv_module1(x)
        x2 = self.conv_module2(x)
        return torch.cat([x1, x2], 1)

class DownsampleModule(nn.Module):
    def __init__(
        self,
        in_channel,
        ch3,
        padding
    ):
        super().__init__()
        self.conv_module = BasicConv2d(in_channel, ch3, kernel_size=3, stride=1)
        self.max_pool = nn.MaxPool2d(3, stride=2, ceil_mode=True, padding=0)

    def forward(self, x: Tensor) -> Tensor:
        x1 = self.conv_module(x)
        x2 = self.max_pool(x)
        return torch.cat([x1, x2], 1)

class Inception(nn.Module):
    def __init__(
        self,
        in_channels: int,
        ch1x1: int,
        ch3x3red: int,
        ch3x3: int,
        ch5x5red: int,
        ch5x5: int,
        pool_proj: int,
        conv_block: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__init__()
        if conv_block is None:
            conv_block = BasicConv2d
        self.branch1 = conv_block(in_channels, ch1x1, kernel_size=1)

        self.branch2 = nn.Sequential(
            conv_block(in_channels, ch3x3red, kernel_size=1), conv_block(ch3x3red, ch3x3, kernel_size=3)
        )

        self.branch3 = nn.Sequential(

```

```

        conv_block(in_channels, ch5x5red, kernel_size=1),
        # Here, kernel_size=3 instead of kernel_size=5 is a known bug.
        # Please see https://github.com/pytorch/vision/issues/906 for details
        conv_block(ch5x5red, ch5x5, kernel_size=3, padding=1),
    )

    self.branch4 = nn.Sequential(
        nn.MaxPool2d(kernel_size=3, stride=1, padding=1, ceil_mode=True),
        conv_block(in_channels, pool_proj, kernel_size=1),
    )

    def _forward(self, x: Tensor) -> List[Tensor]:
        branch1 = self.branch1(x)
        branch2 = self.branch2(x)
        branch3 = self.branch3(x)
        branch4 = self.branch4(x)

        outputs = [branch1, branch2, branch3, branch4]
        return outputs

    def forward(self, x: Tensor) -> Tensor:
        outputs = self._forward(x)
        return torch.cat(outputs, 1)

class InceptionAux(nn.Module):
    def __init__(
        self,
        in_channels: int,
        num_classes: int,
        conv_block: Optional[Callable[..., nn.Module]] = None,
        dropout: float = 0.7,
    ) -> None:
        super().__init__()
        if conv_block is None:
            conv_block = BasicConv2d
        self.conv = conv_block(in_channels, 128, kernel_size=1)

        self.fc1 = nn.Linear(2048, 1024)
        self.fc2 = nn.Linear(1024, num_classes)
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, x: Tensor) -> Tensor:
        # aux1: N x 512 x 14 x 14, aux2: N x 528 x 14 x 14
        x = F.adaptive_avg_pool2d(x, (4, 4))
        # aux1: N x 512 x 4 x 4, aux2: N x 528 x 4 x 4
        x = self.conv(x)
        # N x 128 x 4 x 4
        x = torch.flatten(x, 1)
        # N x 2048
        x = F.relu(self.fc1(x), inplace=True)
        # N x 1024
        x = self.dropout(x)
        # N x 1024
        x = self.fc2(x)
        # N x 1000 (num_classes)

```



```

        return x

class BasicConv2d(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, **kwargs: Any) -
        super().__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, bias=False, **kwargs)
        self.bn = nn.BatchNorm2d(out_channels, eps=0.001)

    def forward(self, x: Tensor) -> Tensor:
        x = self.conv(x)
        x = self.bn(x)
        return F.relu(x, inplace=True)

class GoogLeNet_Weights(WeightsEnum):
    IMAGENET1K_V1 = Weights(
        url="https://download.pytorch.org/models/googlenet-1378be20.pth",
        transforms=partial(ImageClassification, crop_size=224),
        meta={
            "num_params": 6624904,
            "min_size": (15, 15),
            "categories": _IMAGENET_CATEGORIES,
            "recipe": "https://github.com/pytorch/vision/tree/main/reference",
            "_metrics": {
                "ImageNet-1K": {
                    "acc@1": 69.778,
                    "acc@5": 89.530,
                }
            },
            "_ops": 1.498,
            "_file_size": 49.731,
            "_docs": """These weights are ported from the original paper."""
        },
    )
    DEFAULT = IMAGENET1K_V1

@register_model()
@handle_legacy_interface(weights=("pretrained", GoogLeNet_Weights.IMAGENET1K
def googlenet(*, weights: Optional[GoogLeNet_Weights] = None, progress: bool
    """GoogLeNet (Inception v1) model architecture from
    `Going Deeper with Convolutions <http://arxiv.org/abs/1409.4842>`_.

    Args:
        weights (:class:`~torchvision.models.GoogLeNet_Weights`, optional):
            pretrained weights for the model. See
            :class:`~torchvision.models.GoogLeNet_Weights` below for
            more details, and possible values. By default, no pre-trained
            weights are used.
        progress (bool, optional): If True, displays a progress bar of the
            download to stderr. Default is True.
        **kwargs: parameters passed to the ``torchvision.models.GoogLeNet``
            base class. Please refer to the `source code
            <https://github.com/pytorch/vision/blob/main/torchvision/models/

```

```

        for more details about this class.
    .. autoclass:: torchvision.models.GoogLeNet_Weights
       :members:
    """
    weights = GoogLeNet_Weights.verify(weights)

    original_aux_logits = kwargs.get("aux_logits", False)
    if weights is not None:
        if "transform_input" not in kwargs:
            _overwrite_named_param(kwargs, "transform_input", True)
        _overwrite_named_param(kwargs, "aux_logits", True)
        _overwrite_named_param(kwargs, "init_weights", False)
        _overwrite_named_param(kwargs, "num_classes", len(weights.meta["categories"]))

    model = GoogLeNet(**kwargs)

    if weights is not None:
        model.load_state_dict(weights.get_state_dict(progress=progress, check_hash=True))
        if not original_aux_logits:
            model.aux_logits = False
            model.aux1 = None # type: ignore[assignment]
            model.aux2 = None # type: ignore[assignment]
        else:
            warnings.warn(
                "auxiliary heads in the pretrained googlenet model are NOT pretrained"
            )

    return model

class Trainer:
    def __init__(self, model, optimizer=None, criterion=None, scheduler=None, device=None):
        self.model = model
        self.optimizer = optim.Adam(model.parameters(), lr=0.001) if optimizer is None else optimizer
        #self.criterion = nn.CrossEntropyLoss() if criterion is None else criterion
        self.criterion = nn.NLLLoss() if criterion is None else criterion
        self.scheduler = scheduler
        self.device = "cpu" if device is None else device

        self.model = self.model.to(self.device)

    def train(self, epochs, train_dataloader, val_dataloader=None):
        results = {
            "train_loss": [],
            "train_acc": [],
            "train_batch_losses": [],
            "train_batch_accs": [],
        }
        if val_dataloader:
            results["val_loss"] = []
            results["val_acc"] = []
            results["val_batch_losses"] = []
            results["val_batch_accs"] = []

        #batch_size = 32

```

```

# Iterate over epochs
for epoch in tqdm(range(epochs)):
    # Training the epoch
    #train_dataloader = torch.utils.data.DataLoader(training_set, batch_size=batch_size, shuffle=True, num_workers=0)
    train_loss, train_acc, train_batch_losses, train_batch_accs = self.train(
        dataloader=train_dataloader,
    )
    results["train_loss"].append(train_loss)
    results["train_acc"].append(train_acc)
    results["train_batch_losses"].extend(train_batch_losses)
    results["train_batch_accs"].extend(train_batch_accs)

    # eval the epoch
    if val_dataloader:
        #val_dataloader = torch.utils.data.DataLoader(validation_set, batch_size=batch_size, shuffle=False, num_workers=0)
        val_loss, val_acc, val_batch_losses, val_batch_accs = self.validate(
            dataloader=val_dataloader,
        )
        results["val_loss"].append(val_loss)
        results["val_acc"].append(val_acc)
        results["val_batch_losses"].extend(val_batch_losses)
        results["val_batch_accs"].extend(val_batch_accs)
    #batch_size *= 2
return results

def _train_epoch(self, dataloader):
    # Training mode
    self.model.train()

    # Iterate through batches
    total_samples = 0
    epoch_loss = 0.0
    epoch_acc = 0.0
    batch_losses = []
    batch_accs = []
    for x, y in tqdm(dataloader):
        x = x.to(self.device)
        y = y.to(self.device)

        # train steps
        probs = self.model(x)
        loss = self.criterion(probs, y)
        loss.backward()
        self.optimizer.step()
        self.optimizer.zero_grad()
        if self.scheduler:
            scheduler.step()

        # compute metrics
        num_in_batch = len(x)
        total_samples += num_in_batch
        epoch_loss += loss.detach().item() * num_in_batch
        batch_losses.append(loss.detach().item())
        #probs = F.softmax(logits, dim=1)
        preds = torch.max(probs, dim=1).indices
        acc = torch.sum(torch.eq(preds, y)).detach().item()

```

```

        batch_accs.append(acc/num_in_batch)
        epoch_acc += acc

    avg_epoch_loss = epoch_loss / total_samples
    avg_epoch_acc = epoch_acc / total_samples
    return avg_epoch_loss, avg_epoch_acc, batch_losses, batch_accs

def _eval_epoch(self, dataloader):

    # Eval mode
    self.model.eval()

    # Iterate through batches
    total_samples = 0
    epoch_loss = 0.0
    epoch_acc = 0.0
    batch_losses = []
    batch_accs = []
    for x, y in tqdm(dataloader):
        x = x.to(self.device)
        y = y.to(self.device)

        with torch.no_grad():
            probs = self.model(x)
            loss = self.criterion(probs, y)

            # compute metrics
            num_in_batch = len(x)
            total_samples += num_in_batch
            epoch_loss += loss.detach().item() * num_in_batch
            batch_losses.append(loss.detach().item())
            #probs = F.softmax(logits, dim=1)
            preds = torch.max(probs, dim=1).indices
            acc = torch.sum(torch.eq(preds, y)).detach().item()
            batch_accs.append(acc/num_in_batch)
            epoch_acc += acc

    avg_epoch_loss = epoch_loss / total_samples
    avg_epoch_acc = epoch_acc / total_samples
    return avg_epoch_loss, avg_epoch_acc, batch_losses, batch_accs

## Small model for testing preliminary code
class GarmentClassifier(nn.Module):
    def __init__(self):
        super(GarmentClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 4 * 4)

```

```

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

if __name__ == "__main__":

    # BATCH_SIZE = 64
    BATCH_SIZE = 32
    # EPOCHS = 5
    EPOCHS = 9

    transform = transforms.Compose(
        [
            transforms.ToTensor(),
            transforms.Normalize((0.5,), (0.5,)),
        ]
    )

    # Create datasets for training & validation, download if necessary
    training_set = torchvision.datasets.FashionMNIST('./data', train=True, transform=transform)
    validation_set = torchvision.datasets.FashionMNIST('./data', train=False, transform=transform)

    # Create data loaders for our datasets; shuffle for training, not for validation
    training_loader = torch.utils.data.DataLoader(training_set, batch_size=BATCH_SIZE, shuffle=True)
    validation_loader = torch.utils.data.DataLoader(validation_set, batch_size=BATCH_SIZE, shuffle=False)

    model = GoogLeNet(
        num_classes=10,
        aux_logits=True,
        transform_input=False,
        init_weights=None,
        blocks=None,
        dropout=0.2,
        dropout_aux=0.7,
    )
    #model = GarmentClassifier()
    criterion = torch.nn.CrossEntropyLoss()

    # # prob3.1
    # lrs = []
    # lr = 0.0025
    # for _ in range(10):
    #     lrs.append(lr)
    #     lr += 0.0025
    # prob3_1_results = {}
    # for i, lr in enumerate(tqdm(lrs)):
    #     optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    #     #optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)

    #     trainer = Trainer(model, optimizer=optimizer, criterion=criterion)

    #     results = trainer.train(EPOCHS, training_loader, validation_loader)
    #     results['lr'] = lr
    #     prob3_1_results[str(i)] = results

```

```
# with open('prob3_1_results.json', 'w') as fp:
#     json.dump(prob3_1_results, fp)

# prob3.2
# optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
# scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer, base_lr=0.002)
# trainer = Trainer(model, optimizer=optimizer, criterion=criterion, scheduler=scheduler)
# prob3_2_results = trainer.train(EPOCHS, training_loader, validation_loader)
# with open('prob3_2_results.json', 'w') as fp:
#     json.dump(prob3_2_results, fp)

# prob3.3
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
trainer = Trainer(model, optimizer=optimizer, criterion=criterion) #, device=device
prob3_3_results = trainer.train(EPOCHS, training_loader, validation_loader)

with open('prob3_3_results.json', 'w') as fp:
    json.dump(prob3_3_results, fp)
```