Homework Source Code: https://github.com/tiffanysentosa/Practical-Deep-Learning-Systems/tree/main

Tiffany Sentosa

Problem 1 - Bias, Variance Tradeoff, Regularization

1.1 Bias-Variance decomposition for a regression problem

Begin with the expected mean squared error (MSE):

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

Using the $\textbf{linearity of expectation}$, we can bring the expectation inside the summation:

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

Next, we $\textbf{expand the square}(f(x_i)+\epsilon-g(x_i))^2$ using the identity $((a+b+c)^2 = a^2+2ab+b^2)$:

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

Now we break down the expectation into three separate terms:

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

Since the noise $(epsilon)$ is independent of $(f(x_i)-g(x_i)$ and $(E[\epsilon]=0)$ (assuming unbiased noise), the second term simplifies to zero:\

$$E\left[(f(x_i)-g(x_i))\epsilon\right]=0$$

Also, $(E[\epsilon^2])$ is simply the $\textbf{variance of the noise}$, which we denote as (σ^2):

$$E[\epsilon^2]=\sigma^2$$

Substituting these into the expression, we get:

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

The first term represents the $\textbf{bias squared}$, while the second term is the $\textbf{noise variance}$:

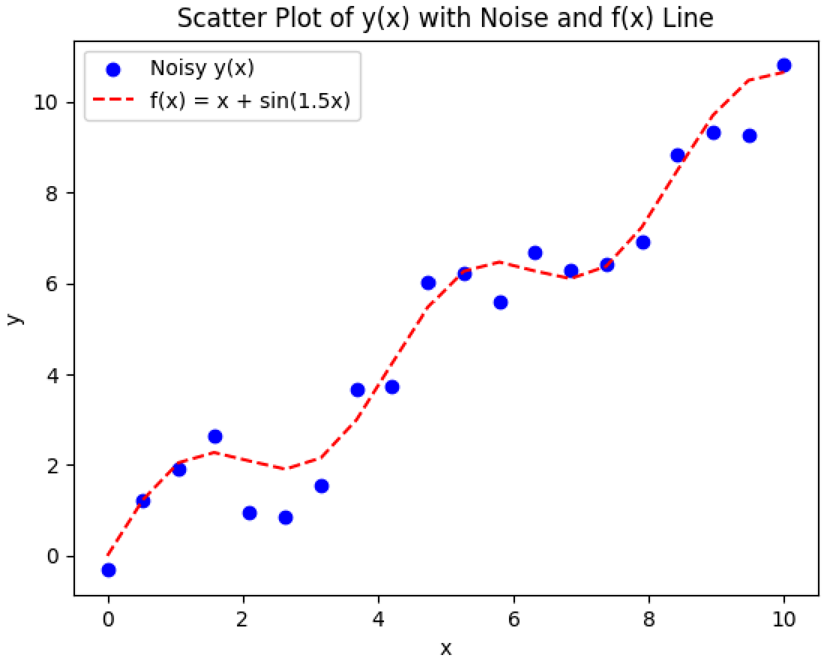
$$E[\text{MSE}] = \text{Bias}^2+\text{Variance}+\text{Noise}$$

1.2 Generate and Plot with Random Data Points

1. **Define the Function (f(x)) and Noise**:
   The function $f(x)$ is defined and given to us as: $f(x)=x+\sin(1.5x)$ The noise $epsilon$ follows a normal distribution $\mathcal{N}(0,0.3)$, meaning it has a mean of 0 and a variance of 0.3. This adds unpredictable variation to the values of $f(x)$.
2. **Generate Data Points**:
   To create the dataset, we generate 20 equally spaced points for x in the range from 0 to 10. For each point x_i, we compute the corresponding $y(x_i)$ as:

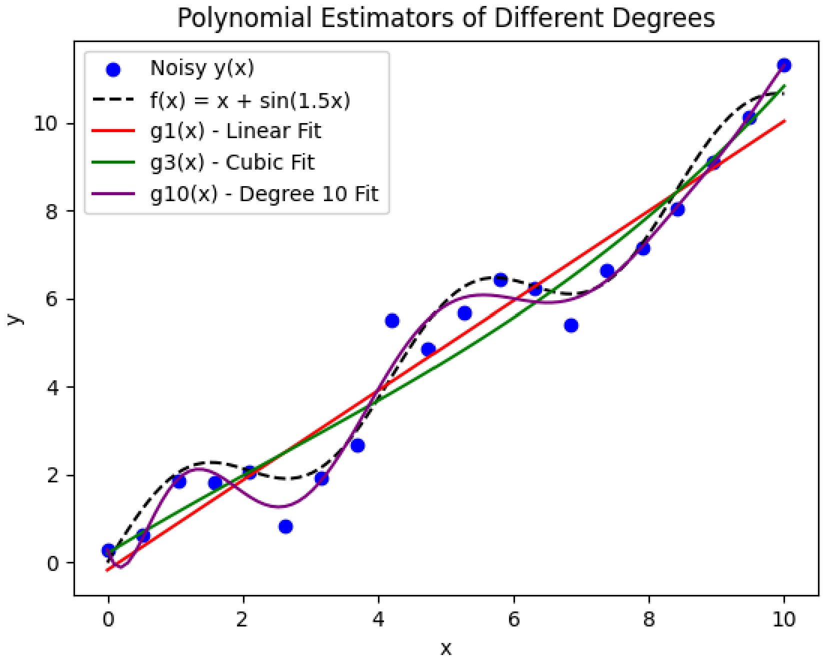
   $$y(x_i)=f(x_i)+\epsilon_i$$

   where $epsilon_i$ is a random sample from $\mathcal{N}(0,0.3)$. This means that each point $y(x_i)$ represents the true function $f(x_i)$ plus some random variation.



Please find the code in Homework 1 > 1.2_plot.py

1.3 Polynomial Regression: Fitting Different Degrees to Estimate a Function



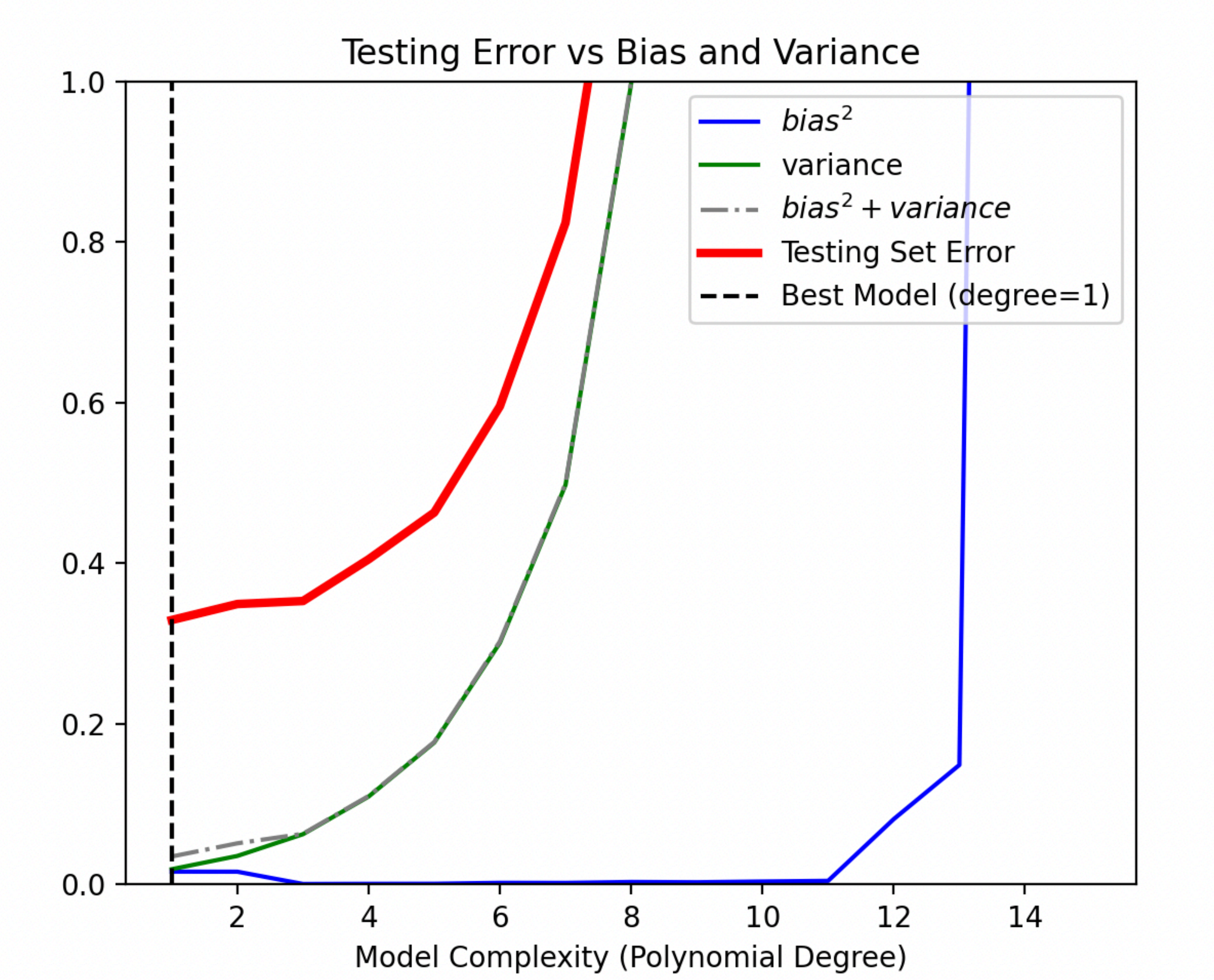
- **Blue scatter points**: The noisy data $y(x)$.
- **Black dashed line**: The true function $f(x) = x + sin(1.5x)$.
- **Red line**: The linear fit $g_1(x)$ (degree 1).
- **Green line**: The cubic fit $g_3(x)$ (degree 3).
- **Purple line**: The 10th-degree polynomial fit $g_{10}(x)$.

1. **Linear Fit $g_1(x)$ (Red Line)**:

- **Underfitting**: The linear fit is just a straight line, so it doesn't follow the wave-like pattern in the data. It's oversimplifying things and not capturing the ups and downs of the true function. The linear fit **underfits** the data because it's too basic to handle the curve in the data.

2. **Cubic Fit $g_3(x)$ (Green Line)**:
   - **Best Fit**: The cubic fit does a good job of capturing the overall trend in the data. It bends with the waves and fits the true function pretty closely without getting too caught up in the random noise. The cubic fit seems like the **best choice** here. It's flexible enough to capture the pattern but not so complicated that it starts modeling random noise.
3. **10th-Degree Polynomial Fit $g_{10}(x)$ (Purple Line)**:
   - **Overfitting**: The 10th-degree polynomial tries to follow every little bump in the data, including the random noise. This makes it way too wiggly and complicated. While it may look like it's fitting the data closely, it's actually picking up on noise that shouldn't be part of the model. The 10th-degree fit is **overfitting** because it's trying to model the random variations (noise) in the data, which means it won't do well with new data.

1.4 Bias-Variance Tradeoff with Multiple Datasets and Polynomial Models



The **polynomial of degree 1** minimizes the testing error while avoiding overfitting, making it the best model.

1.5 Bias and MSE with Regularization

Standard Polynomial Model (Degree 10):
Mean MSE for g_{10} model: 0.4122
Mean $Variance$ for g_{10} model: 0.1080
Mean $Bias^2$ for g_{10} model: 0.0030

Regularized Polynomial Model (Degree 10, L_2 Regularization):
Mean MSE for g_{10} model: 0.3508
Mean $Variance$ for g_{10} model: 0.0248
Mean $Bias^2$ for g_{10} model: 0.0228x

The g_{10} model exhibits overfitting, which leads to high variance and low bias. Regularization increases bias slightly but significantly reduces variance, resulting in an overall decrease in MSE.

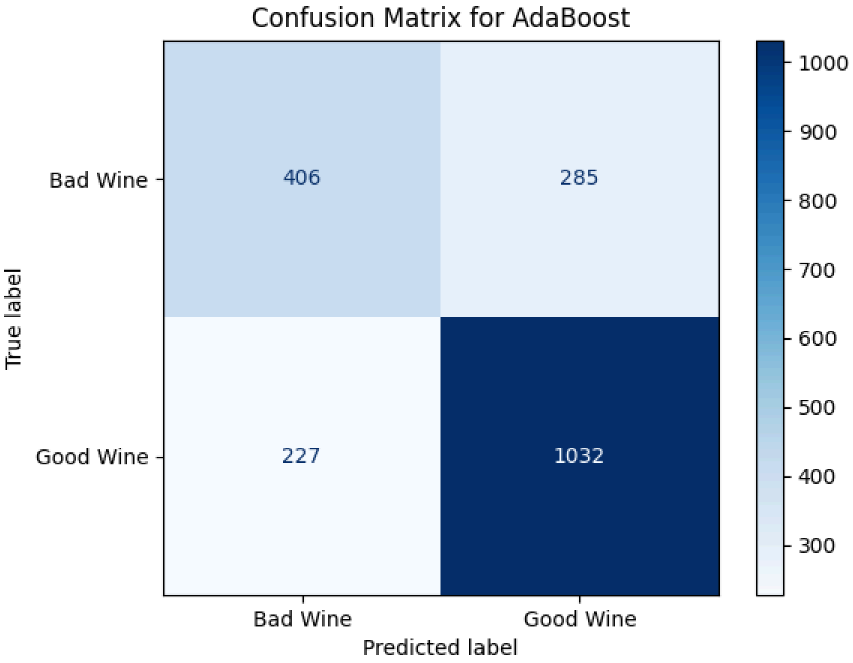
Problem 2 - Precision, Recall, ROC

2.1 True Negatives in ROC and PR curves

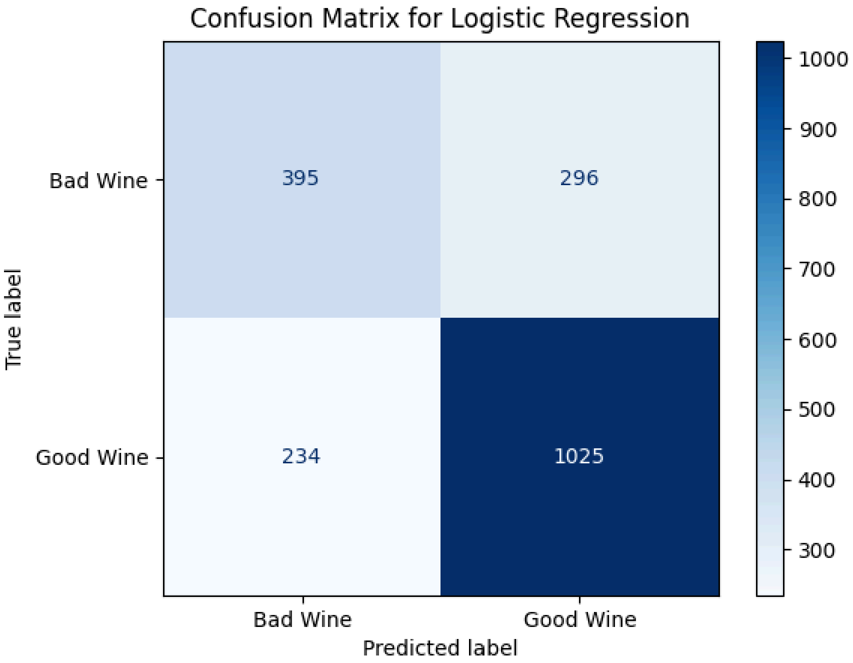
1. Does true negative matter for both ROC and PR curve? Argue why each point on ROC curve corresponds to a unique point on PR curve?
    1. **Does true negative matter for both ROC and PR curve?**
        1. **True negatives matter in ROC curves** because ROC curves is the curve plotting the true positive rate (TPR) against the false positive rate (FPR). True negatives directly influence the false positive rate (FPR) as the FPR is defined as $FPR = \frac{FP}{FP+TN}$.
        2. True negatives do not matter in PR curves. This is because PR curves plot precision against recall. Precision is defined as $Precision = \frac{TP}{TP+FP}$ and recall as $Recall = \frac{TP}{TP+FN}$. Neither of the two depend on true negatives (TN), therefore TNs do not matter for the plotting of PR curves.
    2. **Argue why each point on ROC curve corresponds to a unique point on PR curve?**
        1. The reason why each point on the ROC curve corresponds to a unique point on the PR curve is because both curves are derived from the same set of the confusion matrices. This is explained by Theorem 3.1 Davis & Goadrich (2006) pg. 234 (given below).
        2. "Theorem 3.1. For a given dataset of positive and negative examples, there exists a one-to-one correspondence between a curve in ROC space and a curve in PR space, such that the curves contain exactly the same confusion matrices, if $Recall \neq 0$."

2.2 Binary Datasets and ROC and PR Curves

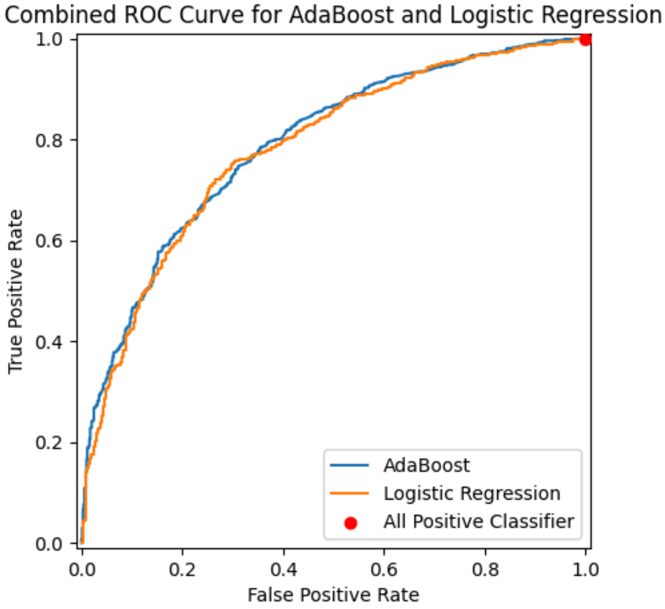
1. Select one OpenML dataset with 2 output classes. Use two binary classifiers (Adaboost and Logistic regression) and create ROC and PR curves for each of them. You will have two figures: one containing two ROC and other containing two PR curves. Show the point where an all positive classifier lies in the ROC and PR curves. An all positive classifier classifies all the samples as positive.
   The dataset I chose contains red and white variants of the Portuguese "Vinho Verde" wine, with quality ratings ranging from 0 (bad) to 10 (good). For this classification task, we’ve categorized wines with ratings of 5 or below as "bad" and those above 5 as "good" to simplify the quality assessment.

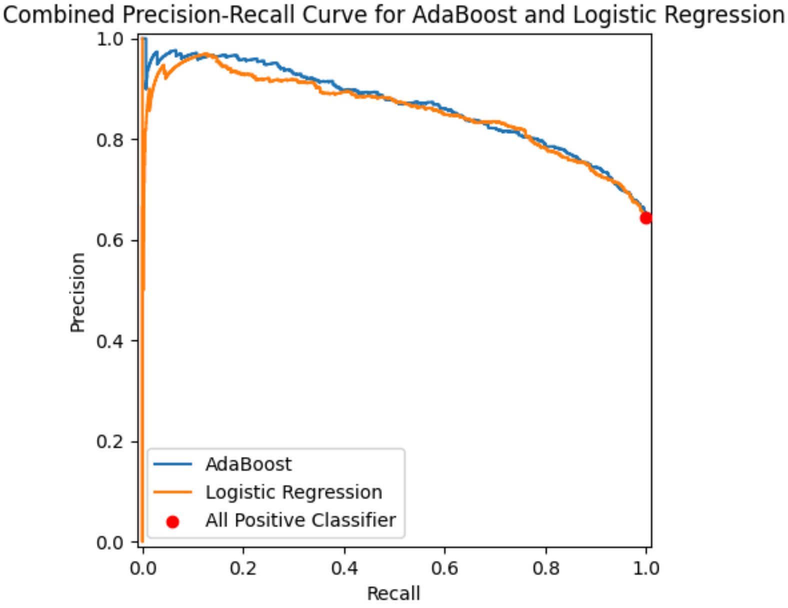


Confusion Matrix for AdaBoost



Confusion Matrix for Logistic Regression





Validation check for PR Curve:
Total Positives / total test set = 1259/1950=0.646

2.3 PR Gain curve, AUROC, AUPR, and AUPRG.

1. 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.
    1. Area under the ROC curve (AUROC)
        1. AdaBoost AUROC: 0.7901
        2. Logistic Regression AUROC: 0.7826
        3. Reason why this matters: The AUROC measures the ability of the classifier to distinguish between the positive and negative classes across different thresholds.
        4. Interpretation: Both classifiers perform similarly, with AdaBoost slightly better. A value closer to 1 indicates better performance.

2. AUPR
    1. AdaBoost AUPR: 0.8679
    2. Logistic Regression AUPR: 0.8546
    3. Again, both classifiers are close in performance, with AdaBoost slightly better. The AUPR is particularly useful when dealing with imbalanced classes, as it focuses more on the performance of the positive class.
3. AUPRG

    1. AdaBoost AUPRG: 0.6271
    2. Logistic Regression AUPRG: 0.5898
    3. - The PR Gain curve provides an alternative to the PR curve that accounts for the baseline performance (proportion of positive samples). It tends to show a lower value because it corrects for the "optimism" that can occur in traditional PR curves.
    4. Here too, AdaBoost performs better, but with a noticeable difference between AUPR and AUPRG values, especially for Logistic Regression.

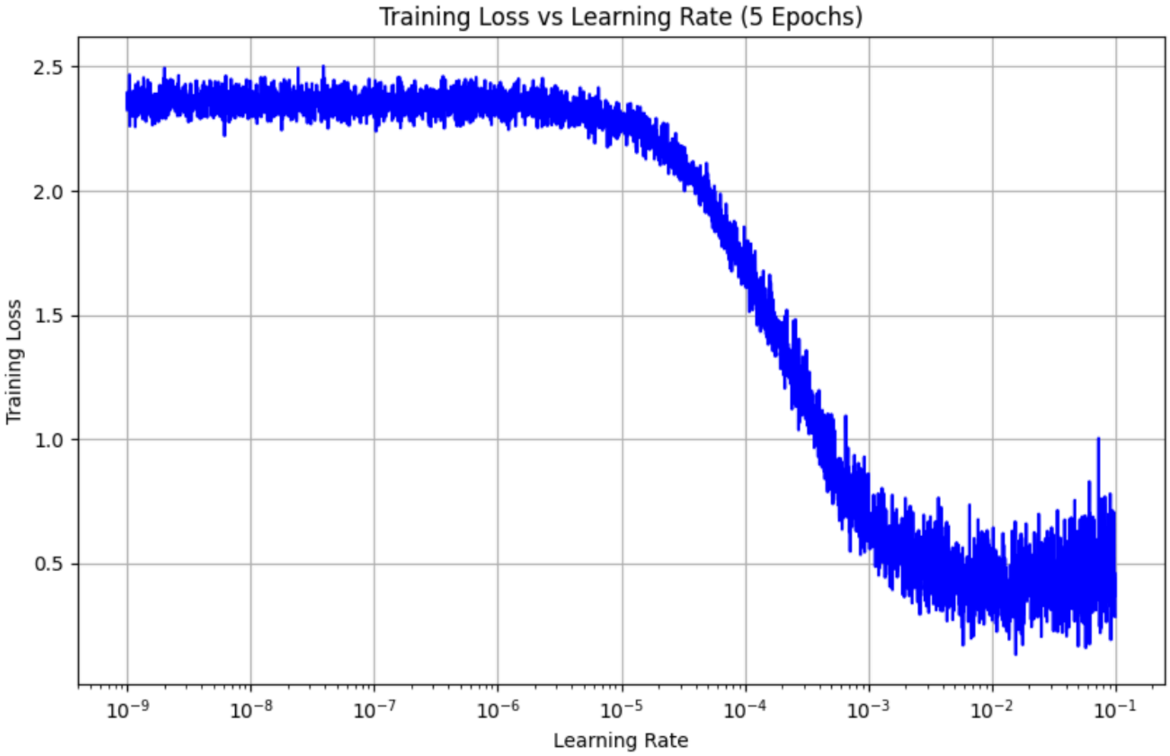
    The NIPS paper suggests that using PR Gain curves is a better way to evaluate classifiers, especially when dealing with imbalanced datasets. They argue that traditional PR curves can be misleading because they don't factor in the baseline of how many positives are actually in the dataset, which can make a model look better than it really is. In my experiment, I saw this in action: both AdaBoost and Logistic Regression had pretty high AUPR scores, making them seem like they were doing great. But when I looked at the AUPRG scores, they were much lower, showing that the real performance wasn't as impressive once I accounted for the baseline. This aligns with what the paper says, and I agree that PR Gain curves give a more honest view of how well a model is doing, especially when positives are scarce. It's a good reality check against the false sense of success that traditional PR curves might give.

References:

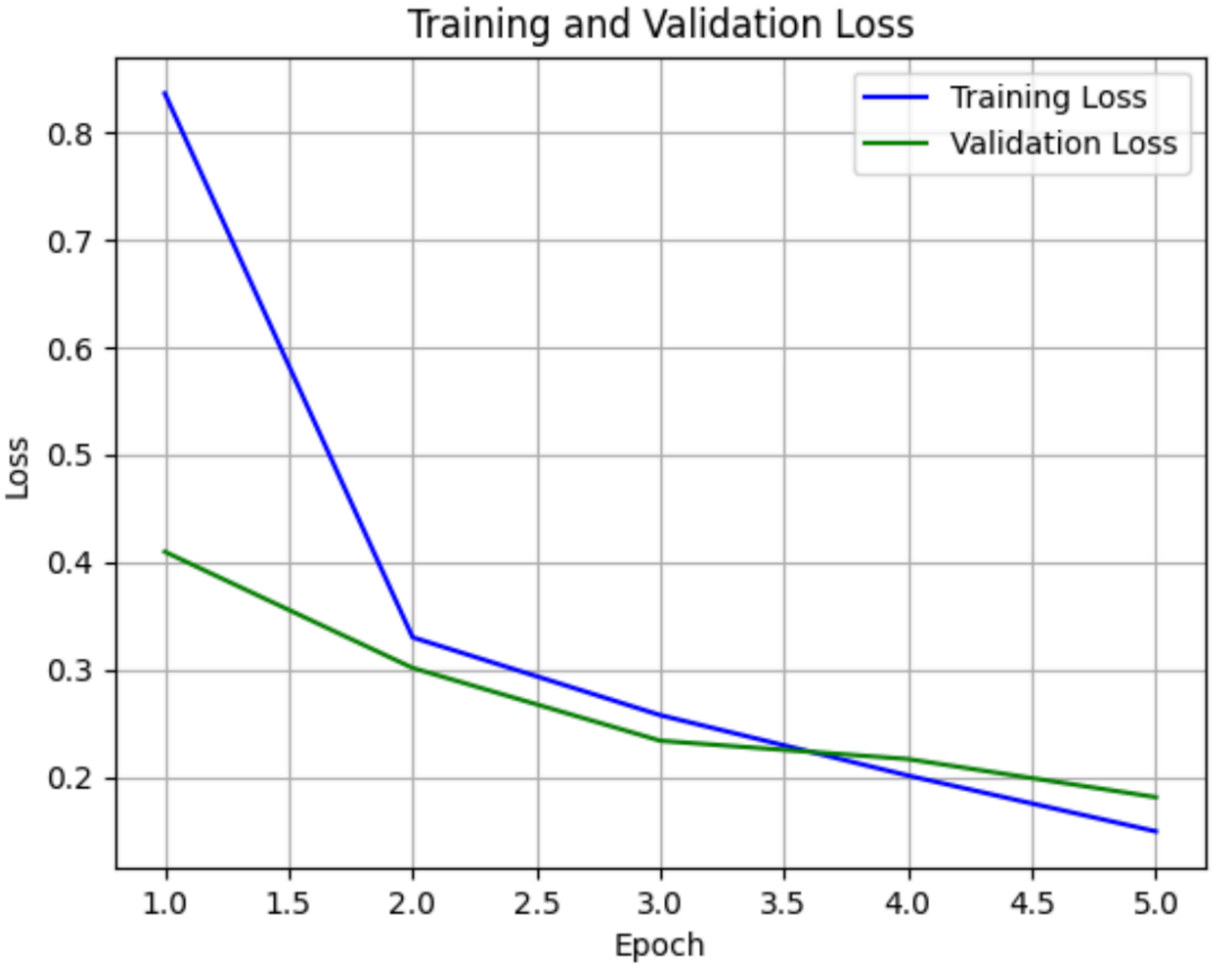
% P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis.
% Modeling wine preferences by data mining from physicochemical properties.
% In Decision Support Systems, Elsevier, 47(4):547-553. ISSN: 0167-9236.
%
% Available at: [@Elsevier] http://dx.doi.org/10.1016/j.dss.2009.05.016
% [Pre-press (pdf)] http://www3.dsi.uminho.pt/pcortez/winequality09.pdf
% [bib] http://www3.dsi.uminho.pt/pcortez/dss09.bib

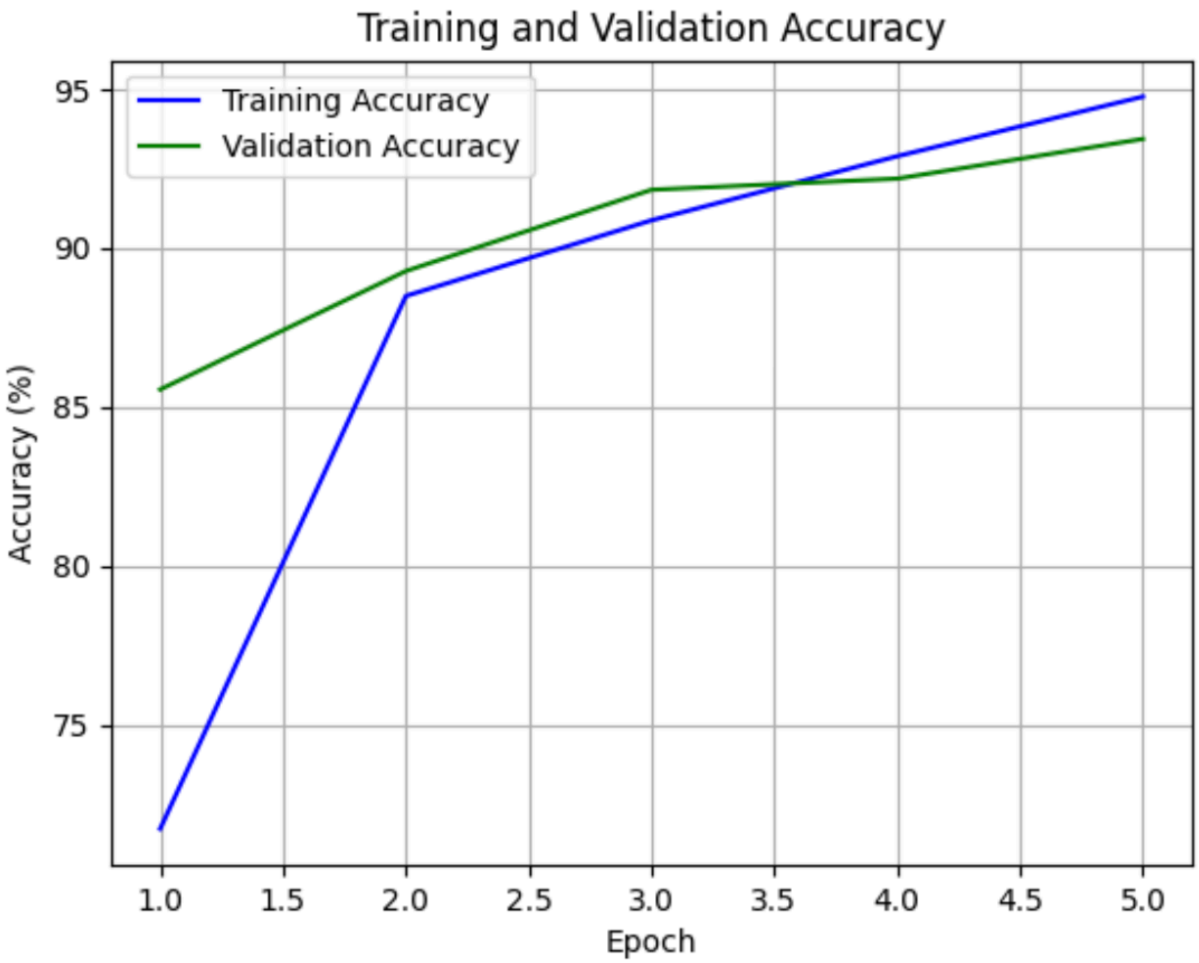
Problem 3: Learning Rate, Batch Size, FashionMNIST

3.1

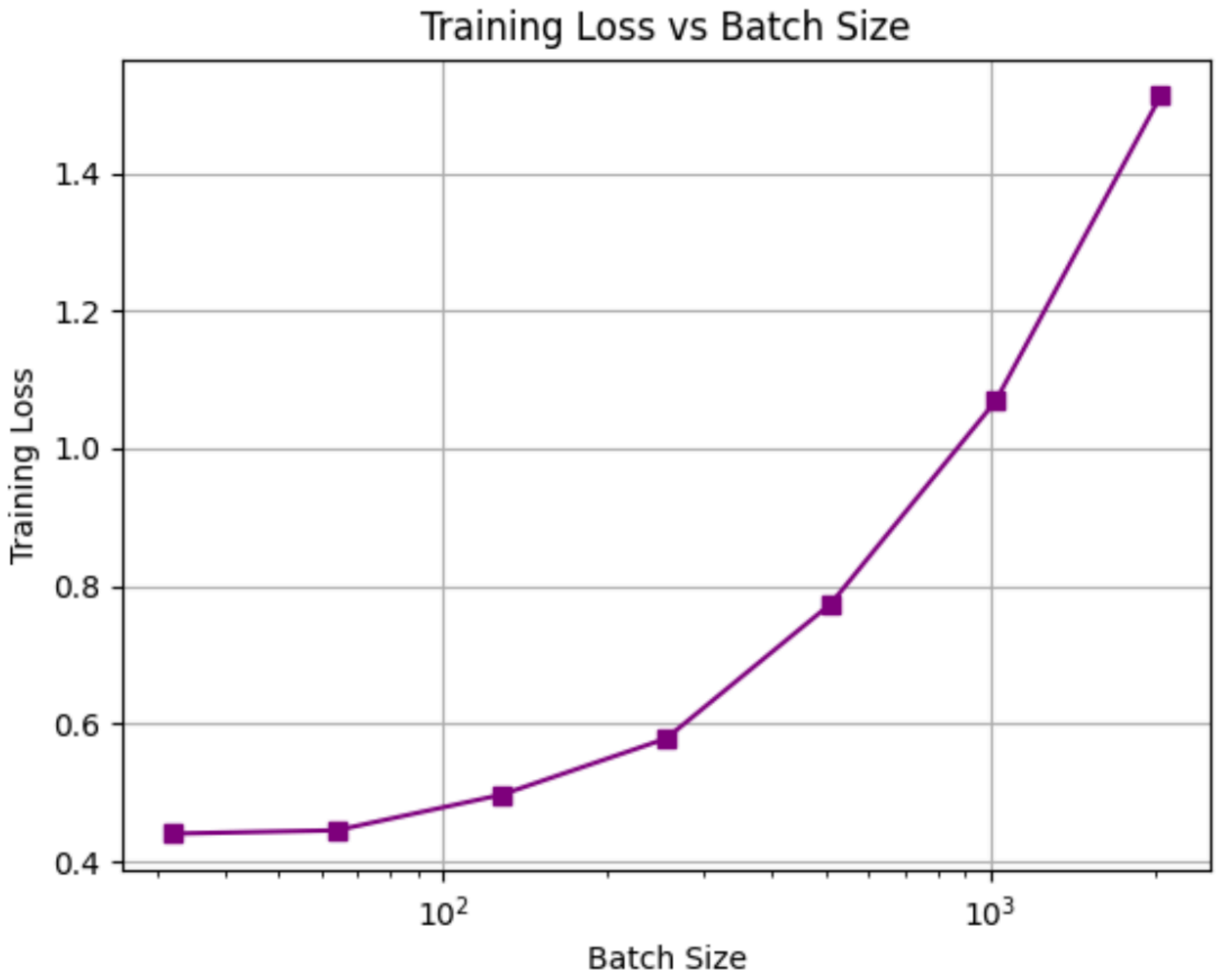

Training Loss vs Learning Rate (5 Epochs)
Training Loss
2.5
2.0
1.5
1.0
0.5
Learning Rate
10^{-9}
10^{-8}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}

3.2


Training and Validation Loss
Training Loss
Validation Loss
Loss
0.8
0.7
0.6
0.5
0.4
0.3
0.2
Epoch
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy (%)
95
90
85
80
75
Epoch
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

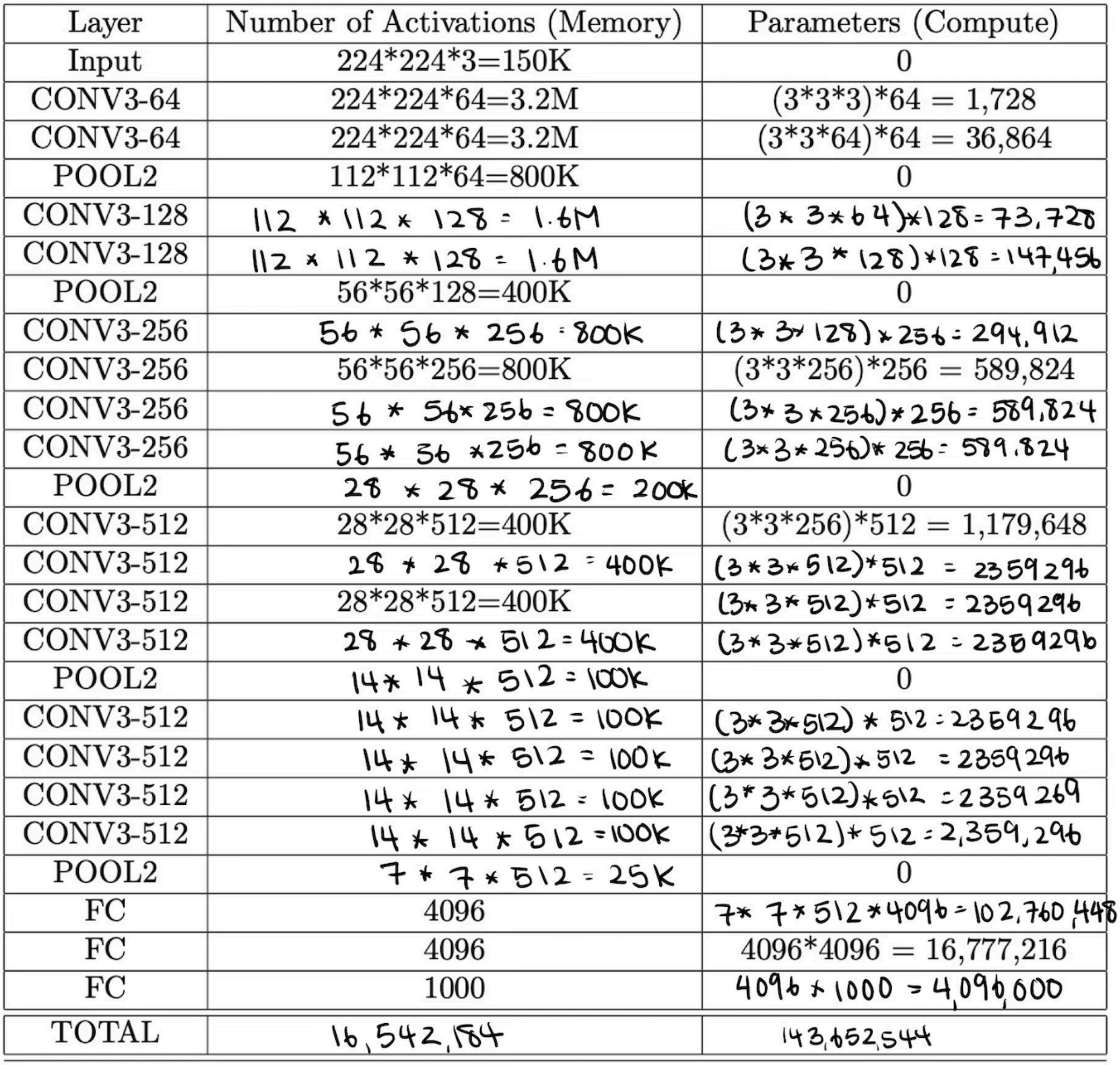
3.3



The graph shows how changing the batch size affects how well a machine learning model learns. When we use very small batches, the model learns quickly but can make mistakes. As we increase the batch size, the model's performance improves, reaching its best point at a medium batch size. However, if we keep making the batches bigger, the model starts to learn more slowly and doesn't do as well. This is similar to how changing the learning rate affects the model, but it's easier to fine-tune the learning rate than the batch size. The graph helps us understand that finding the right batch size is important for getting the best results from our model. It's about balancing how fast the model learns with how stable and accurate its learning is. This information is useful for people training machine learning models to get better results.

Problem 4: Convolutional Neural Networks Architectures

4.1



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 = 2359296
CONV3-512	28*28*512=400K	(3 * 3 * 512)*512 = 2359296
CONV3-512	28 * 28 * 512 = 400K	(3 * 3 * 512)*512 = 2359296
POOL2	14 * 14 * 512 = 100K	0
CONV3-512	14 * 14 * 512 = 100K	(3 * 3 * 512) * 512 = 2359296
CONV3-512	14 * 14 * 512 = 100K	(3 * 3 * 512) * 512 = 2359296
CONV3-512	14 * 14 * 512 = 100K	(3 * 3 * 512) * 512 = 2359269
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	16,542,184	143,652,544

Table 1: VGG19 memory and weights

4.2 Inception Module Analysis from GoogLeNet

4.2.1. Inception Model

The idea behind designing an Inception module, as proposed in the GoogLeNet paper by Szegedy et al., is to efficiently capture multi-scale features within a convolutional neural network (CNN) through parallel convolutional filters of different sizes (1x1, 3x3, 5x5) and pooling operations. By applying these operations in parallel and concatenating their outputs, the network can simultaneously capture fine-grained details and more abstract features, enabling a more comprehensive feature representation. The 1x1 convolutions play a critical role in reducing the dimensionality of the input before the more computationally expensive 3x3 and 5x5 convolutions, which helps keep the computational cost manageable without sacrificing performance. Pooling layers are also used to capture the most dominant features in a region, offering translational invariance. This multi-scale approach allows the network to process information at various levels of abstraction while maintaining efficiency by optimizing the use of computational resources (Szegedy et al., 2014, Section 4). The resulting architecture enhances the network's ability to handle both fine and coarse visual details, making it well-suited for complex tasks like image classification and object detection (Szegedy et al., 2014, Section 3).

4.2.2 Output Size Calculations

Naive Architecture:

The output channels from each operation in the naive inception module are:

- **1x1 Convolution:** 128 filters
- **3x3 Convolution:** 192 filters
- **5x5 Convolution:** 96 filters
- **3x3 Max Pooling:** Followed by 256 channels
  Thus, the total number of output channels is:

$$128 + 192 + 96 + 256 = 672$$

So, the output size after concatenation is:

$$32 \times 32 \times 672$$

Dimension Reduction Architecture:

In the dimension reduction version, 1x1 convolutions are applied before 3x3 and 5x5 convolutions to reduce the number of input channels. The output channels are:

- **1x1 Convolution (before 3x3 and 5x5):** Reduced the input depth
- **Output from the inception module:** 128, 192, 96, and 64 channels after dimension reduction.

Thus, the total number of output channels is:

$$128 + 192 + 96 + 64 = 480$$

So, the output size is reduced to:

$$32 \times 32 \times 480$$

4.2.3 Convolutional Operations

Naive Architecture:

For the naive architecture, the total number of operations can be calculated as follows:

- **1x1 Convolution:**

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

- **3x3 Convolution:**

$$192 \times 32 \times 32 \times 3 \times 3 \times 256 = 169,869,312 \text{ (operations)}$$

- **5x5 Convolution:**

$$96 \times 32 \times 32 \times 5 \times 5 \times 256 = 98,304,000 \text{ (operations)}$$

Thus, the total number of operations is approximately **1.12 billion operations**:

$$33,554,432 + 169,869,312 + 98,304,000 = 1.12 \text{ billion operations}$$

Dimension Reduction Architecture:

You're absolutely correct! There are two **1x1 convolution layers** with **128 filters**, which means we need to include both of them in the total number of operations.

Let’s break it down again with this additional consideration:

Corrected Dimension Reduction Architecture - Breakdown:

1. **1x1 Convolution (before 3x3)**:

   $128 \times 32 \times 32 \times 1 \times 1 \times 256 = 33,554,432 \text{ operations}$
2. **1x1 Convolution (before 5x5)**:

   $32 \times 32 \times 32 \times 1 \times 1 \times 256 = 8,388,608 \text{ operations}$
3. **1x1 Convolution (after pooling)**:

   $64 \times 32 \times 32 \times 1 \times 1 \times 256 = 16,777,216 \text{ operations}$
4. **3x3 Convolution** (after preceding 1x1 convolution, depth = 128):

$192 \times 32 \times 32 \times 3 \times 3 \times 128 = 226,492,416 \text{ operations}$

5. **5x5 Convolution** (after preceding 1x1 convolution, depth = 32):
   $96 \times 32 \times 32 \times 5 \times 5 \times 32 = 78,643,200 \text{ operations}$
6. **Second 1x1 Convolution (parallel to first)**:
   Since there is a second **1x1 convolution** layer with **128 filters**, the operation count should be the same as the first one:
   $128 \times 32 \times 32 \times 1 \times 1 \times 256 = 33,554,432 \text{ operations}$

Total Operations:
$33,554,432 + 33,554,432 + 8,388,608 + 16,777,216 + 226,492,416 + 78,643,200 = 397,410,304 \text{ operations}$

Summary of Computational Savings:

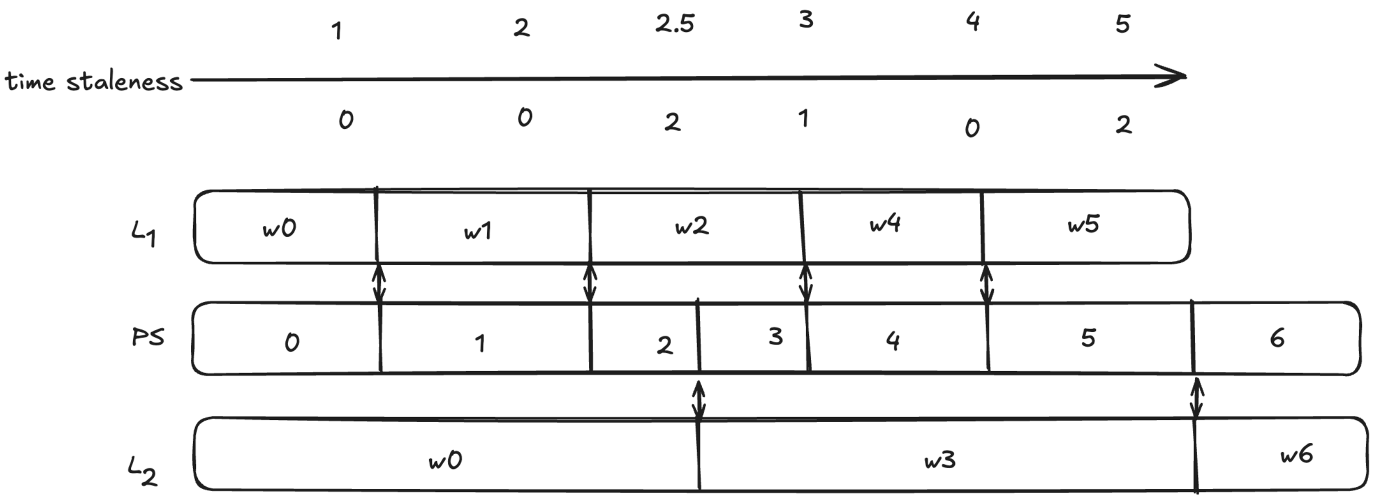
- **Naive architecture:** 1.12 billion operations
- **Dimension reduction architecture:** 397.4 million operations

The **dimensionality reduction architecture** results in a significant reduction in the number of operations compared to the naive version, saving around **68%** of computational cost.

4.2.4 Naive Architecture vs. Dimensionality Reduction

The problem with the naive architecture is that it performs very expensive convolution operations directly on the full 256-channel input. Convolutions with large filters, like 3x3 and 5x5, applied to a high number of channels result in an exponential increase in computational complexity. For example, a 3x3 convolution with 192 filters applied to 256 input channels requires 169.8 million operations, and a 5x5 convolution with 96 filters requires 98.3 million operations, making the naive version computationally expensive. The dimensionality reduction architecture solves this problem by applying 1x1 convolutions as bottleneck layers, which reduce the depth of the input before applying the more computationally expensive 3x3 and 5x5 convolutions. For instance, by reducing the depth from 256 to 128 channels for the 3x3 convolution, the number of operations is halved to 84.9 million, and for the 5x5 convolution, the depth is reduced to 32 channels, requiring only 12.3 million operations. This approach significantly reduces computational complexity while maintaining the ability to capture multi-scale features, making the dimensionality reduction technique much more efficient than the naive approach.

Problem 5: Async SGD Training System



- $g[L1, 1]$: The staleness is **0** because this is the first update, and there are no prior updates for this layer.
- $g[L1, 2]$: The staleness is **0** since the previous update also occurred at the $L1$ level, meaning no other layer has been updated in between.
- $g[L1, 3]$: The staleness is **1** because the last update was $g[L1, 2]$, meaning one step has passed since the last update for this parameter.
- $g[L1, 4]$: The staleness is **0** again because the previous update was also for the $L1$ layer, indicating that this parameter has been updated recently without any interruptions.
- $g[L2, 1]$: The staleness is **2** because there were two updates, $g[L1, 1]$ and $g[L1, 2]$, that happened before this update to the $L2$ layer, creating a delay or "staleness" of 2.
- $g[L2, 2]$: The staleness is **2** due to the updates at $g[L1, 3]$ and $g[L1, 4]$, meaning two updates occurred at the $L1$ layer before this update to the $L2$ layer.