This can be run run on Google Colab using this link

Coding Assignment 2.1: Training a Deep Network

```
%shell wget
'https://raw.githubusercontent.com/NEU-Silicon-Valley/CS7150-CA2/refs/
heads/main/hw2utils.py'

--2026-02-12 19:15:08--  https://raw.githubusercontent.com/NEU-
Silicon-Valley/CS7150-CA2/refs/heads/main/hw2utils.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.111.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|
185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3643 (3.6K) [text/plain]
Saving to: ‘hw2utils.py.4’

 hw2utils.py.4         0%[                    ]       0  --.-KB/s
hw2utils.py.4       100%[===================>]   3.56K  --.-KB/s    in
0s

2026-02-12 19:15:08 (46.8 MB/s) - ‘hw2utils.py.4’ saved [3643/3643]



%shell wget
'https://raw.githubusercontent.com/NEU-Silicon-Valley/CS7150-CA2/refs/
heads/main/tiny-classification.npz'

--2026-02-12 19:15:08--  https://raw.githubusercontent.com/NEU-
Silicon-Valley/CS7150-CA2/refs/heads/main/tiny-classification.npz
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|
185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 60685 (59K) [application/octet-stream]
Saving to: ‘tiny-classification.npz.1’

           tiny-clas   0%[                    ]       0  --.-KB/s
tiny-classification 100%[===================>]  59.26K  --.-KB/s    in
0.04s

2026-02-12 19:15:08 (1.52 MB/s) - ‘tiny-classification.npz.1’ saved
[60685/60685]
```

```
%shell wget
'https://raw.githubusercontent.com/NEU-Silicon-Valley/CS7150-CA2/refs/
heads/main/hard-classification.npz'

--2026-02-12 19:15:08--  https://raw.githubusercontent.com/NEU-
Silicon-Valley/CS7150-CA2/refs/heads/main/hard-classification.npz
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.111.133, 185.199.110.133, 185.199.109.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|
185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 60668 (59K) [application/octet-stream]
Saving to: ‘hard-classification.npz.1’

           hard-clas   0%[                    ]       0  --.-KB/s
hard-classification 100%[===================>]  59.25K  --.-KB/s    in
0.04s

2026-02-12 19:15:09 (1.50 MB/s) - ‘hard-classification.npz.1’ saved
[60668/60668]
```

Overview

We will start by exploring the optimization aspects of deep network training. Throughout this journey, you will gain insights into:

Part 1:

- The fundamentals of simple gradient descent.
- The concept of weight decay.
- A deep understanding of PyTorch autograd and PyTorch optimizers.

Part 2:

- Analyzing raw gradients, means, and RMS (Root Mean Square).
- Delving into exponential moving averages.
- Exploring the workings of the ADAM optimization algorithm.

Part 3:

- Strategies for optimizing neural network parameters.
- Selecting appropriate nonlinearities, architectures, and layers to tackle the vanishing gradient problem.
- Leveraging techniques like regularization, parameterization, and specific layer choices to enhance generalization.

- Unpacking the roles and impacts of ADAM, ReLU activation, weight decay, network depth, network width, residual architectures, and batch normalization in deep learning.

Note

- **You do not need to tune hyper parameters in the regular tasks.**
- **You do not install any additional packages inside the Colab environment.**
- **If you collaborate or get assistance from classmates, online resources, AI, or other sources, then you must then you must explicitly write down the sources that you used to credit them.**
- **Attend office hours and post on the Teams group chat if you have any questions.**
- **You have sufficient time to work on this assignment. Please refrain from asking for extensions.**

Setup

```
# Import Libraries

import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import matplotlib
from torch.nn import Sequential, Module
from matplotlib import pyplot as plt
from scipy.stats import norm
from hw2utils import LossFunctionWithPlot, ConstantVectorNetwork
```

Part 1: Simple Gradient Descent

Gradient descent is an iterative optimization algorithm for finding the minimum of a function. It works by starting at a point and then moving in the direction of the steepest descent until it reaches a minimum. The steepest descent is the direction in which the function is decreasing most rapidly.

To train a model using gradient descent, we start with a set of initial parameters. These parameters are the values of the variables in the model. We then repeatedly apply gradient descent to update the parameters. Each update moves the parameters in the direction of the steepest descent. This continues until the parameters converge to a minimum of the function.

The choice of the learning rate is important for gradient descent. The learning rate is the size of the steps that are taken in the direction of the steepest descent. If the learning rate is too small,

the algorithm will converge slowly. If the learning rate is too large, the algorithm may diverge and never converge.

Let's say we have a function $L(x)$ that we want to minimize. The gradient of $L(x)$ is a vector that points in the direction of the steepest descent of $f(x)$. The gradient can be calculated using the following equation:

\begin{align} \nabla \mathcal{L}(x) &= \begin{bmatrix} \frac{\partial \mathcal{L}(x)}{\partial x_1} \\ \frac{\partial \mathcal{L}(x)}{\partial x_2} \\ \vdots \\ \frac{\partial \mathcal{L}(x)}{\partial x_n} \end{bmatrix} \end{align}

The gradient descent algorithm can be used to minimize f(x) by repeatedly taking steps in the direction of the gradient. The update rule for gradient descent is given by the following equation:

\begin{equation} x_\text{new} = x_\text{old} - \alpha \cdot \nabla \mathcal{L}(x_\text{old}) \\ \end{equation}

where x_{old} is the current value of x, x_{new} is the new value of x, α is the learning rate, and $\nabla L(x_{\text{old}})$ is the gradient of $L(x)$ evaluated at x_{old}.

Simple Implementation of Gradient Descent on a quadratic loss

Below we demonstrate gradient descent optimization in action.

We iteratively update the `x` to try to minimize a quadratic function `L` defined by the `LossFunctionWithPlot` class. The trajectory of updates and corresponding losses are stored and plotted to visualize the optimization progress.

Provide an implementation of simple gradient descent below. In 21 steps you can make the loss go down to about 3 or better, and drive `x` somewhat towards the center of the minimum of the loss function.

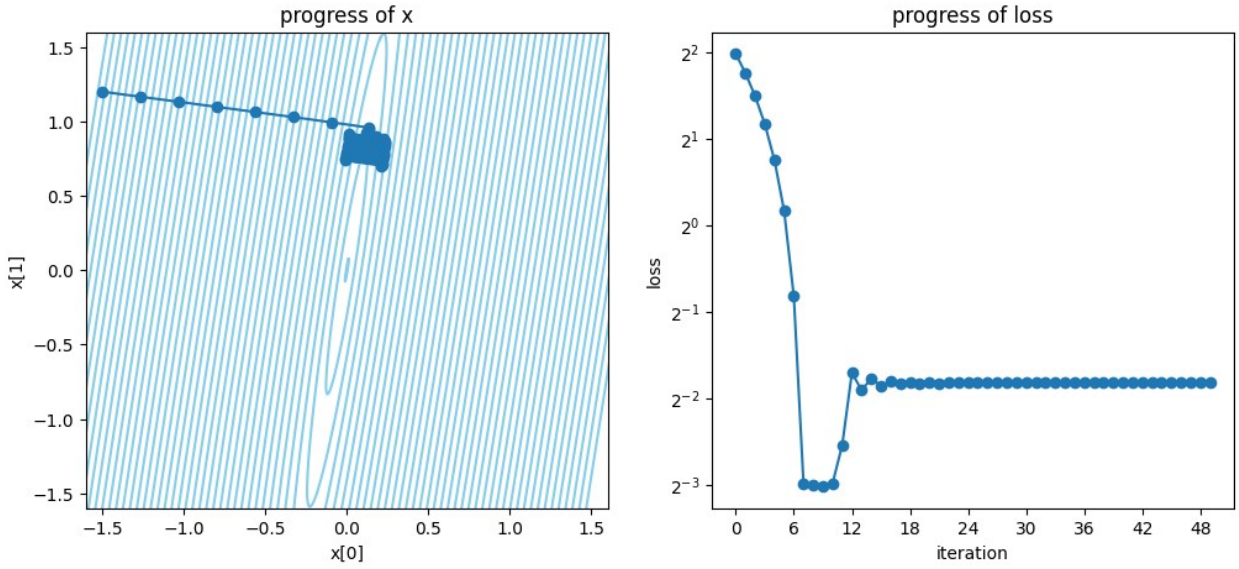
- Use `loss.backward()` (read https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html)
- Use `x.grad` to get the gradient.
- Update `x` in-place using `x -=`*something*, and know why `torch.no_grad()` is needed.
- Understand why gradients need to be zeroed: https://stackoverflow.com/questions/48001598/why-do-we-need-to-call-zero-grad-in-pytorch

Task 1.1 - Implement simple gradient descent (1 point)

```
# Do not change the starting x.
x = torch.tensor([-1.5, 1.2])
x.requires_grad = True
L = LossFunctionWithPlot()

# Start by using a learning rate of 0.1.
learning_rate = 0.1

for iter in range(50):
```

```
    loss = L(x)
    if iter % 7 == 0: print(f'Loss at step {iter} is
{loss.item():.3f}')
    loss.backward()
    with torch.no_grad():

##########################################################################
######
    # TODO: Implement Simple Gradient Descent and update variable 'x'
    # Read Documentation to compute a gradient of a parameter -
    # https://pytorch.org/docs/stable/autograd.html

##########################################################################
######
        x -= learning_rate * x.grad

##########################################################################
######
    #                                END OF YOUR CODE
#

##########################################################################
######
    x.grad = None
L.plot_history()
```

```
Loss at step 0 is 3.937
Loss at step 7 is 0.126
Loss at step 14 is 0.293
Loss at step 21 is 0.282
Loss at step 28 is 0.283
Loss at step 35 is 0.283
Loss at step 42 is 0.283
Loss at step 49 is 0.283
```



One reason it is hard for gradient descent to move towards the bottom of the bowl is that there is no single ideal learning rate for all dimensions. In the problem in 1.1, notice that:

- the loss changes direction very quickly (the curvature is high) in the horiztonal direction of `x[0]`
- the loss is very slow-changing (the curvature is lower) in the vertical direction of `x[1]`.

The "sawtooth" loss curves and the "zig-zag" paths are symptoms of an optimizer that is taking steps that are too large: the path could be repeatedly jumping over a valley in the loss surface and ending up at another point of high, or even higher loss. A lower learning rate can help, but it can lead to another problem. (What problem? Try it.)

Another fancy idea is to mix learning rates, with different learning rates for each parameter. Although you can still make a learning rate too high or too low.

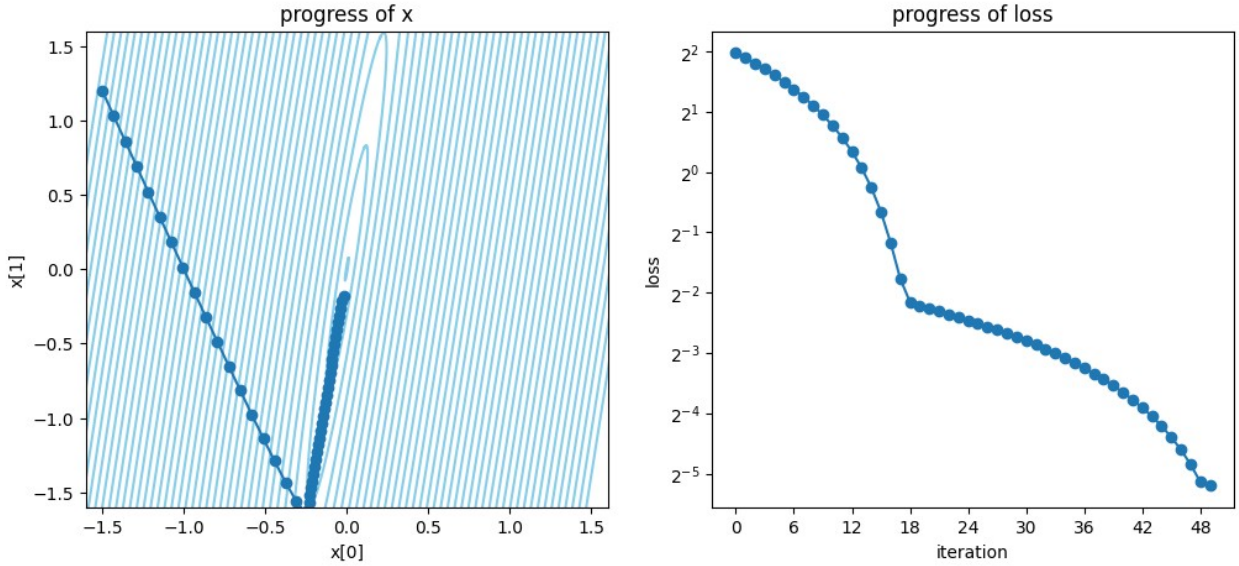
Task 1.2 - Explore Simple Gradient Descent using various learning rates (_ points)

Now copy your code from 1.1 below here, but **experiment with learning rates**, including **unequal learning rates** for `x[0]` and `x[1]` by setting to a tensor with two values. Try to find a pair of learning rates that move x to the bottom of the bowl and stays there with near-zero loss.

```
##########################################################################
##########
# TODO: Copy your solution from Task 1.1 here and attempt to discover
a pair of
# learning rates that guide the optimization process to place 'x' at
the lowest
#point of the bowl and maintain it there with a nearly zero loss.
##########################################################################
##########
x = torch.tensor([-1.5, 1.2])
x.requires_grad = True
L = LossFunctionWithPlot()
```

```
learning_rate = torch.tensor([0.03, 0.5])

for iter in range(50):
    loss = L(x)
    if iter % 7 == 0: print(f'Loss at step {iter} is
{loss.item():.3f}')
    loss.backward()
    with torch.no_grad():
        x -= learning_rate * x.grad
    x.grad = None
L.plot_history()

##########################################################################
#####
#                                END OF YOUR CODE
#
##########################################################################
#####
```

```
Loss at step 0 is 3.937
Loss at step 7 is 2.372
Loss at step 14 is 0.838
Loss at step 21 is 0.201
Loss at step 28 is 0.157
Loss at step 35 is 0.112
Loss at step 42 is 0.067
Loss at step 49 is 0.027
```



Inline Question 1.A (1 point): What bad things tend to happen when the learning rate is too high?

Answer:

When the learning rate is too high, the parameter updates overshoot the minimum. Instead of converging, the optimization oscillates wildly around or diverges away from the minimum entirely, causing the loss to increase or fluctuate without settling. The values could even blow up to infinity.

Inline Question 1.B (1 point): What bad things tend to happen when the learning rate is too low?

Answer:

When the learning rate is too low, convergence becomes much slower as the optimizer takes steps that are too small, requiring too many iterations to reach the minimum. It also becomes more susceptible to getting stuck in relatively shallow local minima, since the steps are too small to escape them.

Inline Question 1.C (1 point): Should a higher or lower learning rate be used on a dimension with higher curvature (with sharper changes)?

Answer:

A lower learning rate should be used on dimensions with higher curvature. Higher curvature means that the gradient changes rapidly in that direction, because the loss surface is sharper. A large step along a high-curvature dimension would probably overshoot the minimum along that dimension and cause too much oscillation. A smaller learning rate keeps the updates stable in steep directions.

II) Weight Decay, aka L2 regularization

Weight decay, aslo known as L2 regularization, adds the following term to the total loss:

\begin{equation} \mathcal{L}(\theta) = \mathcal{L}_\text{objective}(\theta) + \frac{\lambda}{2} ||\theta||^2 \end{equation}

where:

- $L(\theta)$ - is the regularized objective function.
- $L_{\text{objective}}(\theta)$ - is the original objective function (without regularization).
- θ is the vector of model parameters.
- λ is the regularization parameter.

1.D) Inline Question (1 point): What is the gradient $\nabla L(\theta)$ in terms of $\nabla L_{\text{objective}}(\theta)$ and θ?

Answer:

Take the gradient with respect to θ:

$$\nabla L(\theta) = \nabla L_{\text{objective}}(\theta) + \frac{\lambda}{2} \cdot 2\theta$$

Simplifying the 2s:

$$\nabla L(\theta) = \nabla L_{\text{objective}}(\theta) + \lambda\theta$$

1.E) Inline Question (1 point): Suppose 'v' is the learning rate. Then what should be the update rule for Θ?

Answer:

Applying the standard gradient descent update rule and substituting the gradient from part D:

$$\theta_{\text{new}} = \theta_{\text{old}} - v\left(\nabla L_{\text{objective}}\left(\theta_{\text{old}}\right) + \lambda \theta_{\text{old}}\right)$$

Opening the bracket:

$$\theta_{\text{new}} = \theta_{\text{old}} - v \nabla L_{\text{objective}}\left(\theta_{\text{old}}\right) - v \lambda \theta_{\text{old}}$$

Take out θ_{old}:

$$\theta_{\text{new}} = \left(1 - v \lambda\right)\theta_{\text{old}} - v \nabla L_{\text{objective}}\left(\theta_{\text{old}}\right)$$

Task 1.3 - Implement a `SimpleGradientDescent` optimizer class in pytorch (2 point)

Pytorch encapsulates optimization algorithms into optimizer classes that hold on to a list of parameters being optimized, and that update the parameters in-place based on gradients using a `step()` method.

Here we see how that pattern works.

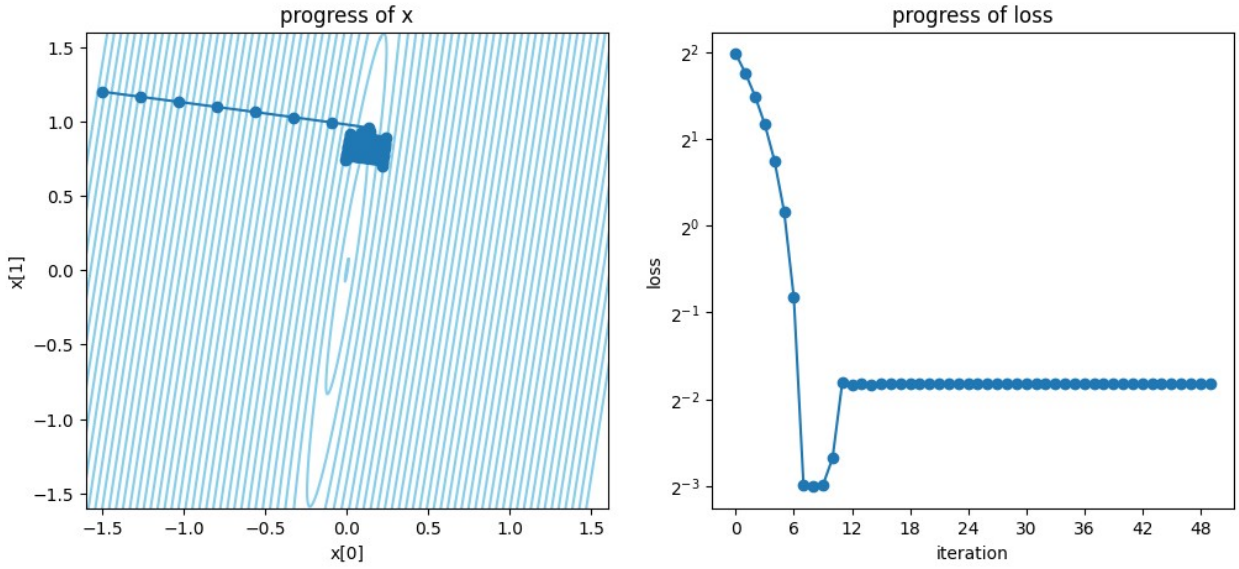
Implement the `SimpleGradientDescent` class as a pytorch-style optimizer by completing the code below.

Incorporate weight decay by adding the term you computed above, where λ is the `weight_decay` hyperparameter.

```
class SimpleGradientDescent():
    def __init__(self, parameters, lr=0.1, weight_decay=0.0):
        self.lr = lr
        self.weight_decay = weight_decay
        self.parameters = []
        for x in parameters:
            self.parameters.append(x)
    def step(self):
      with torch.no_grad():
            for x in self.parameters:

########################################################################
##
        # TODO:  Implement Simple Gradient Descent with the inclusion
of
        # weight_decay, and then update the results in variable x.

########################################################################
##
                x -= self.lr * (x.grad + self.weight_decay * x)
```

```
##########################################################################
##
        #                                END OF YOUR CODE
#

##########################################################################
##
    def zero_grad(self):
        for x in self.parameters:
            x.grad = None

x = torch.tensor([-1.5, 1.2])
x.requires_grad = True # This tells PyTorch that the x variable will
be used to calculate gradients.

L = LossFunctionWithPlot()

learning_rate = 0.1
optimizer = SimpleGradientDescent([x], lr=learning_rate,
weight_decay=1e-3)

for iter in range(50):
    loss = L(x)
    if iter % 7 == 0: print(f'Loss at step {iter} is
{loss.item():.3f}')
    loss.backward()
    with torch.no_grad():
        optimizer.step()
    optimizer.zero_grad()

L.plot_history()
```

```
Loss at step 0 is 3.937
Loss at step 7 is 0.126
Loss at step 14 is 0.282
Loss at step 21 is 0.283
Loss at step 28 is 0.283
Loss at step 35 is 0.283
Loss at step 42 is 0.283
Loss at step 49 is 0.283
```



Part 2: The ADAM optimizer

I) Analyzing Raw Gradients, Means, and RMS (Root Mean Square).

Measure mean of the gradient on each dimension

First, let us plot and examine the mean of the gradient.

Using the code below as a starting point, calculate and fill in the following mean gradient over the 50 iterations in the specific optimization below:

$$mean_g rads[i] = \frac{1}{N}\sum_{t=1}^{N}\frac{\partial L}{\partial x_i}\left(x^{(t)}\right)$$

2.A) Inline Questions (1 point):

$$\text{1. Mean gradient component with respect to x[0]} = \fbox{ -0.2982 }$$

$$\text{2. Mean gradient component with respect to x[1]} = \fbox{ 0.0940 }$$

Measure root-mean-square of the gradient on each dimension

Second, let's plot the root mean square (RMS) of the gradient.

Using the code below as a starting point, calculate and fill in the following root-mean-square gradient over the 50 iterations in the specific optimization below:

$$rms_g rads[i] = \sqrt{\frac{1}{N}\sum_{t=1}^{N}\left(\frac{\partial L}{\partial x_i}\left(x^{(t)}\right)\right)^2}$$

2.B) Inline Questions (1 point):

$$\text{ 1. RMS gradient component with respect to x[0]} = \fbox{ 2.1092 }$$

$$\text{ 2. RMS gradient component with respect to x[1]} = \fbox{ 0.3078 }$$

Understanding the challanges faced by simple gradient descent.

The optimization problem in Part I causes gradient descent to run into a few different problems:

- after initially descending the loss jumps back up.
- after making initial quick progress, x gets stuck instead of heading towards the middle.

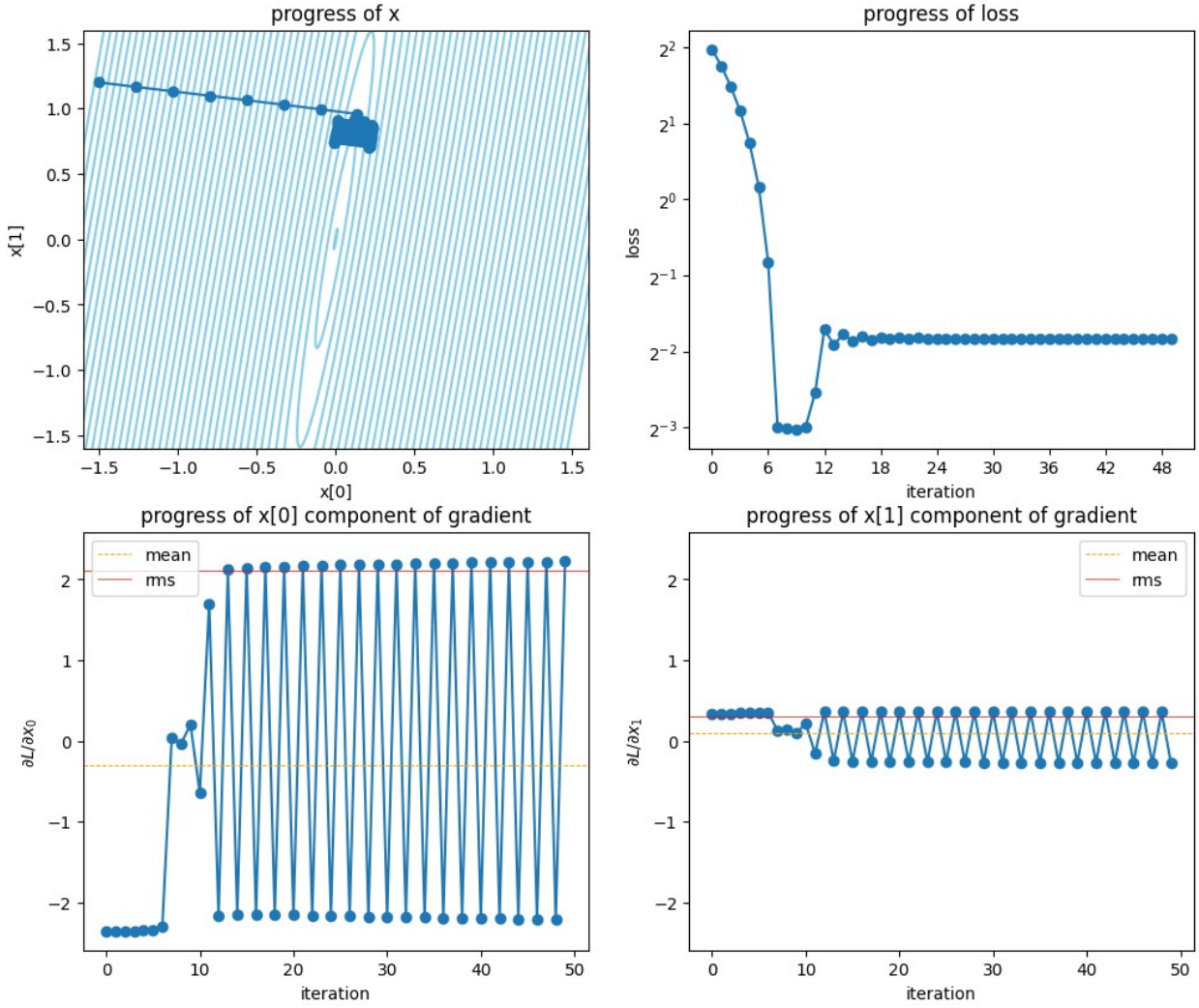
Task 2.1 Insights from Raw Gradient Plots and Statistical Metrics (2 points)

To understand the problems, run the code below to see plots of the raw gradient, and then compute the Mean and root-mean-square (RMS) of each component of the gradient over all the iterations.

```
x = torch.tensor([-1.5, 1.2])
x.requires_grad = True # This tells PyTorch that the x variable will
be used to calculate gradients.

L = LossFunctionWithPlot()

learning_rate = 0.1
optimizer = torch.optim.SGD([x], lr=learning_rate)
grads = []

for iter in range(50):
    loss = L(x)
    loss.backward()
    with torch.no_grad():
        optimizer.step()
    grads.append(x.grad.clone())
    optimizer.zero_grad()

grads = torch.stack(grads)

########################################################################
#############
# TODO: compute mean derivatives and root mean square derivatives for
x[0] and x[1]
########################################################################
#############

mean_grads = grads.mean(dim=0) # should be computed as a pair of
means, one for each dimesnion
rms_grads = (grads ** 2).mean(dim=0).sqrt() # should be computed as a
pair of root-mean-squares, one for each dimension

########################################################################
##########
#                               END OF YOUR CODE
#
```

```
##########################################################################
##########

print('Mean grads for two dimensions separately:', mean_grads)
print('Root mean square grads for two dimensions separately:',
rms_grads)

L.plot_history(grads=grads, mean_grads=mean_grads,
rms_grads=rms_grads)
```

```
Mean grads for two dimensions separately: tensor([-0.2982,  0.0940])
Root mean square grads for two dimensions separately: tensor([2.1092,
0.3078])
```



Facts about mean and RMS

To understand the role of the mean and RMS, answer the following questions.

2.C) Inline Question (2 points):

$$\text{1. When they differ, which is guaranteed to be smaller: mean or RMS?} = \fbox{ Mean }$$

$$\text{2. Which is a better representation of the "average size", mean or RMS?} = \fbox{ RMS }$$

$$\text{3. When the derviative is consistently positive, will the mean tend to be larger or smaller?} = \fbox{ Larger }$$

$$\text{4. When the derviative sign changes frequently, will the mean tend to be larger or smaller?} = \fbox{ Smaller }$$

Anticipating problems in optimization using Mean and RMS

Based on the problems we are seeing, we want the optimizer to slow down when the mean is much smaller than the RMS, and speed up when the mean and RMS are about the same size. That will:

1. Make sure the updates to be **small enough** when the gradient starts becoming bumpy, instead of oscillating.
2. Make sure the updates to be **large enough** when gradient is smooth but happens to be small.

The mean and RMS of the gradient can be used to deal with both these problems. In the next section, we will see how they are directly applied in the ADAM optimizer.

The idea of the ADAM optimizer

The ADAM optimizer automatically chooses a different learning rate for each parameter by using a heuristic that shrinks the update size in regions where the gradient is changing more quickly, while normalizing the update size so that it is a consistent size even in regions where the gradient is very small. Update magnitudes are calculated per-parameter, so ADAM can help deal with parameters that behave very differently from each other.

The idea behind ADAM is to choose an update that is proportional to a fraction between a weighted mean of the gradient and a weighted RMS of the gradient:

$$\Delta x = -\alpha \frac{\text{mean gradient}}{\text{rms gradient}} = -\alpha \frac{\sum_i w_i g_i}{\sqrt{\sum_i u_i g_i^2}}$$

In the definition above, the g_i are samples of the gradient from previous steps, and w_i and u_i are the weights to use for averaging.

Why one might divide the mean gradient by the RMS of the gradient?

To understand, answer the following question.

2.D) Inline Question (2 points):

Suppose there is a new problem which is scaled by some constant K so that all the new gradients are uniformly scaled larger. We want to understand whether ADAM speeds up or slows down when gradients are scaled up. Precisely: If gradients $\hat{g}_i = K\, g_i$ are scaled up with

$K>1$, how will the ADAM update $\Delta\hat{x}$ relate to the original problem's ADAM update Δx i? (e.g., which is larger?)

Answer.

$\Delta\hat{x}=\Delta x.$

They are equal as the ADAM update is invariant to uniform gradient scaling.

Proof:

Let all gradients be uniformly scaled by $K>1$.

$\therefore \hat{g}_i=K g_i.$

The ADAM update for the scaled problem is:

$$\Delta\hat{x}=-\alpha\frac{\sum_i w_i\hat{g}_i}{\sqrt{\sum_i u_i\hat{g}_i^2}}$$

We can substitute $\hat{g}_i=K g_i$:

$$\Delta\hat{x}=-\alpha\frac{\sum_i w_i\left(K g_i\right)}{\sqrt{\sum_i u_i\left(K g_i\right)^2}}$$

We can take out K from the numerator and K^2 from the square root in the denominator:

$$\Delta\hat{x}=-\alpha\frac{K\sum_i w_i g_i}{\sqrt{K^2\sum_i u_i g_i^2}}$$

$$\Delta\hat{x}=-\alpha\frac{K\sum_i w_i g_i}{K\sqrt{\sum_i u_i g_i^2}}$$

The two Ks cancel each other out:

$$\Delta\hat{x}=-\alpha\frac{\sum_i w_i g_i}{\sqrt{\sum_i u_i g_i^2}}=\Delta x$$

Thus, ADAM's updates are always **scale-invariant** because it uniformly scales all gradients by any constant $K>1$ which does not change the magnitude or direction. Neither update is larger as they are identical. It automatically normalizes the gradient.

This property means that ADAM will not go too fast nor too slow just because the average size of the gradient is too large. The only thing that will cause ADAM to slow down is when the mean is

much smaller than the RMS, which happens when the gradient frequently changes sign, for example, when the optimizer is oscillating around a minima.

Because the mean and RMS will change during the optimization, in practice the ADAM algorithm is based on using **exponential moving averages** which will adapt as optimization proceeds.

II) Exponential Moving Average (EMA) on Time Series Data

The exponential Moving Average (EMA) represents a moving average variant that assigns greater importance to the most recent data points within a time series. It achieves this by progressively diminishing the influence of older data points. Unlike simple moving averages, where all data points hold the same weight, EMA's differential weighting scheme enhances its sensitivity to recent data alterations, rendering it highly attuned to the latest fluctuations within the data.

EMA is a weighted average where, the weight of a sample of age $t - i$ is decayed exponentially by $\beta^{|t-i|}$, where β<1 is the smoothing parameter. That is, using geometric series identities,

$$\mathrm{EMA}_t = \frac{\beta^{t-1} x_1 + \beta^{t-2} x_1 + \ldots + \beta x_{t-1} + x_t}{\beta^{t-1} + \beta^{t-2} + \ldots + \beta + 1} = \frac{(1-\beta) \cdot \sum_i \beta^{t-i} x_i}{1 - \beta^t}$$

As t gets large, the denominator becomes indistinguishable from one, and EMA can be estimated by computing just the numerator $\mathrm{EMA}_t^{\iota} = (1-\beta) \cdot \sum_i \beta^{t-i} x_i$. The numerator has the advantage that maintaining a running average only requires a single number be remembered: the most recent numerator $\mathrm{EMA}_{t-1}^{\iota}$. The formula for calculating the numerator EMA_t^{ι} is as follows:

\begin{align*} \text{EMA}_0^* &= 0 \ \text{EMA}*t^* &= *beta *cdot *text{EMA}*{t-1}^* + (1 - \beta) \cdot x_t \end{align*}

When t is small, the numerator can be much smaller than one, so it must be included, so the full formula for the EMA is:

\begin{align*} \text{EMA}_t = \frac{\text{EMA}_t^*}{1 - *beta^{t}} *end{align*}

Where,

- EMA_t - is the Exponential Moving Average at time t.
- EMA_t^{ι} - is the Exponential Moving Average Numerator, which $\approx \mathrm{EMA}_t$ when t is large.
- x_t - is the data point at time t that you want to include in the EMA calculation.
- $\mathrm{EMA}_{t-1}^{\iota}$ - is the EMA numerator calculated at the previous time step t−1.
- β is the smoothing factor

Task 2.2 Implement EMA (2 points)

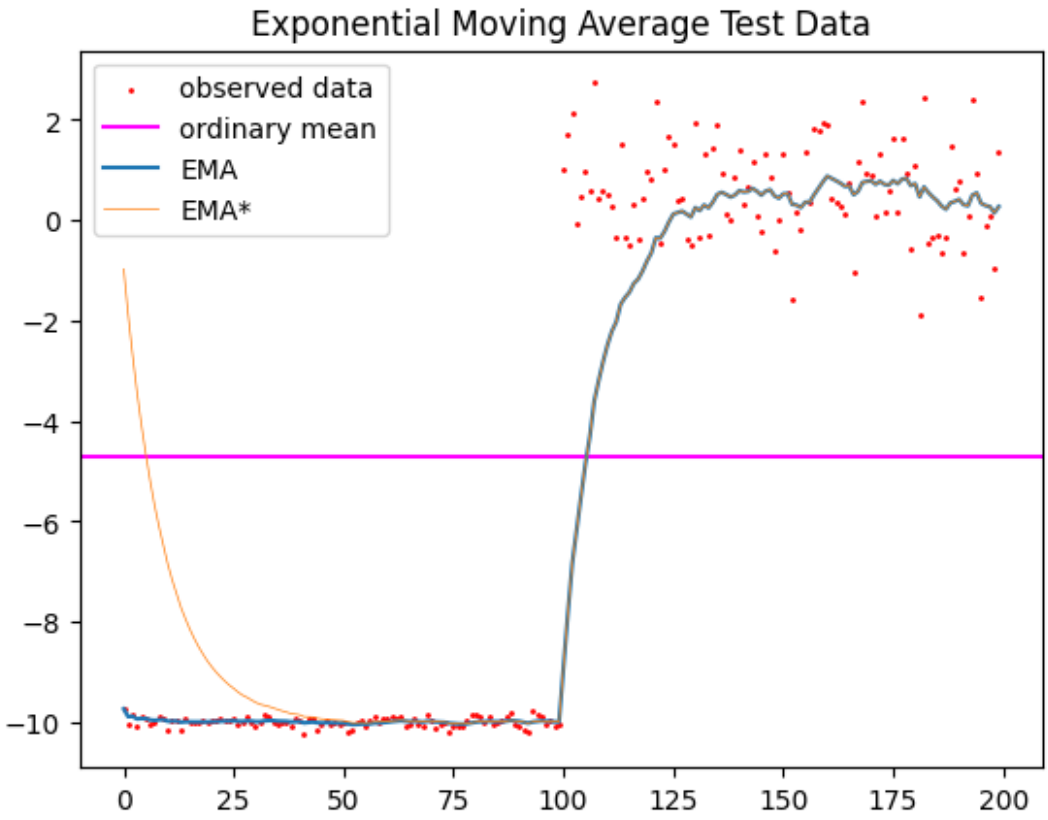
Based on the definitions above, implement ema_update below, and produce the plot of the EMA of the synthetic time series data.

As you can see, unlike the orindary mean, EMA adapts to changes in the data over time.

Also notice the difference between EMA* and EMA at the beginning of the series. Notice that EMA* is not unbiased: instead it has a clear bias towards zero.

```
def ema_update(x_t, beta, t, ema_star_old):
    ema_star_t = 0.0
    ema = 0.0

################################################################################
######
    # TODO: 1) compute ema_star_t from ema_star_old, beta, and x_t.
    #       2) compute ema from ema_star_t, beta, and t

################################################################################
######

    ema_star_t = beta * ema_star_old + (1 - beta) * x_t
    ema = ema_star_t / (1 - beta ** t)


################################################################################
######
    #                                END OF YOUR CODE
#

################################################################################
######

    return ema, ema_star_t

timestamps = 100
time_series = torch.cat([
    torch.randn(timestamps) * 0.1 - 10.0,
    torch.randn(timestamps) + 0.5,
])
mean = time_series.mean()


beta = 0.9
ema_star = 0.0
history, history_star = [], []

for t, d in enumerate(time_series):

    # This is the ema update
    ema, ema_star = ema_update(d, beta, t+1, ema_star)
    history.append(ema)
    history_star.append(ema_star)

plt.title('Exponential Moving Average Test Data')
plt.scatter(range(len(time_series)), time_series, s=1, color='red',
```

```
label='observed data')
plt.axhline(mean, label='ordinary mean', color='magenta')
plt.plot(history, label='EMA')
plt.plot(history_star, label='EMA*', linewidth=0.5)
plt.legend()
plt.show()
```



III) ADAM Optimizer

ADAM (Adaptive Moment Estimation) is a popular optimization algorithm used for training machine learning and deep learning models. ADAM is known for its efficiency, robustness, and ability to handle a wide range of optimization problems.

Here are the key components and features of the ADAM optimizer:

- Adaptive Learning Rates: ADAM adapts the learning rates for each parameter during training. It maintains a separate learning rate for each parameter based on the historical gradients of that parameter. This adaptability helps the algorithm converge faster and handle sparse gradients effectively.
- EMA Momentum: ADAM incorporates the concept of momentum, similar to the SGD with momentum optimizer. It uses exponential moving averages of past gradients to help the optimization process. This momentum term smooths the optimization trajectory and accelerates convergence.

The update rule for ADAM is as follows:

\begin{align*} m_t^ & = \beta_1 \cdot m_{t-1}^* + (1 - \beta_1) \cdot g_t & & \text{(First Moment)} \ m_t & = \frac{m_t^ }{1 - |beta_1^t} & & |text{(Bias correction)} | v_t^ & = \beta_2 \cdot v_{t-1}^* + (1 - \beta_2) \cdot g_t^2 & & \text{(Second Moment)} \ v_t & = \frac{v_t^*}{1 - \beta_2^t} & & \text{(Bias correction)} \ x_{t+1} & = x_t - \alpha \cdot \frac{{m}_t}{\sqrt{{v}_t} + \epsilon} & &\text{(Parameter Update)} \end{align*}

Where:

- m_t and v_t - are the EMA estimates of the first and second moments of the gradients at time step t, respectively.
- β_1 and β_2 - are exponential decay rates for the first and second moments, typically close to 1
- g_t - is the gradient of the parameter θ at time step t, which is used to calculate these moving averages.
- α is the learning rate.
- ϵ is a small constant, just to avoid division-by-zero.

This can also be written as \begin{align} m_t, m_t^ & = \mathtt{ema_update}(g_t, \beta_1, t, m_{t-1}^) & &|text{(EMA Update)}| v_t, v_t^ & = \mathtt{ema_update}(g_t^2, \beta_2, t, v_{t-1}^*) & &\text{(EMA Update)} \ x_{t+1} & = x_t - \alpha \cdot \frac{{m}_t}{\sqrt{{v}_t} + \epsilon} & &\text{(Parameter Update)} \end{align*}

As a final detail, we can incorporate weight decay by defining g_t to include the weight decay term as worked out in a previous exercise.

Task 2.3 - Implement the ADAM optimizer (_ points)

Now implement your own ADAMOptimizer, using the following code as a starting point.

- Within the loop, you can use `ema_update` to make the code simpler.
- Remember that there are two smoothing factors, `beta_1` and `beta_2`.
- Remember to incorporate hyperparameters `learning rate`, `epsilon`, and `weight_decay`

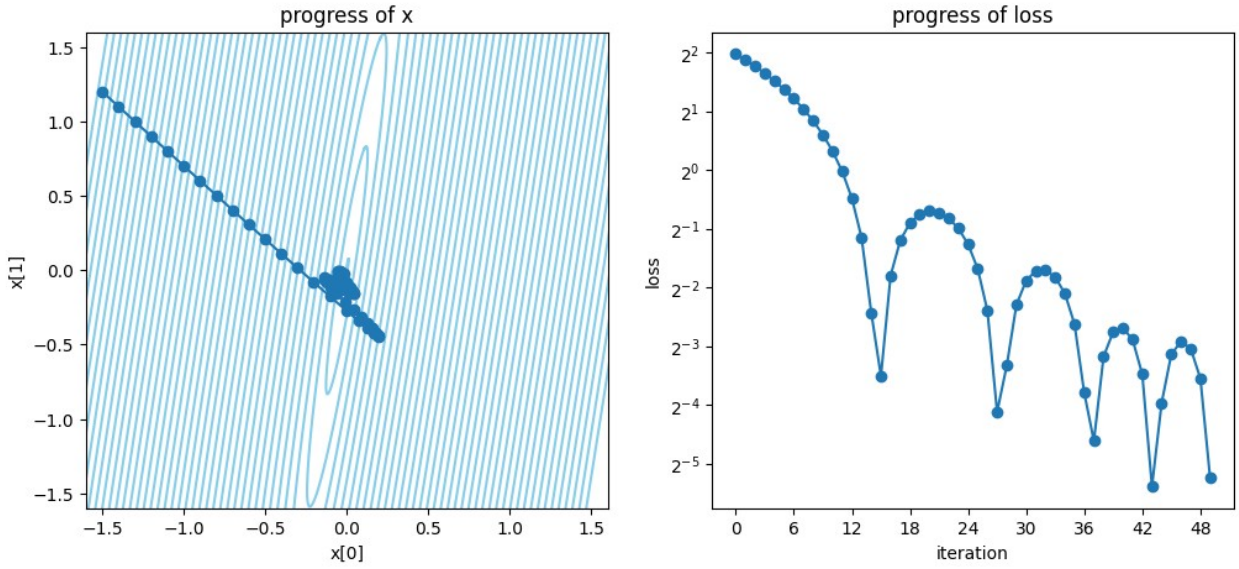
You can test your code by swapping between `torch.optim.Adam` and your own `ADAMOptimizer`. On this test, they should behave the same.

```
class ADAMOptimizer():
    def __init__(self, parameters, lr=0.1, betas=[0.9, 0.999], eps=1e-
8, weight_decay=0.0):
        self.lr = lr
        self.beta_1, self.beta_2 = betas
        self.eps = eps
        self.weight_decay = weight_decay

        # t is a running timestamp.
        self.t = 0
        self.parameters = []
```

```
        self.m_star = []
        self.v_star = []
        for x in parameters:
            self.parameters.append(x)
            self.m_star.append(torch.zeros_like(x))
            self.v_star.append(torch.zeros_like(x))

    def step(self):
        self.t += 1
        with torch.no_grad():
            for x, m_star, v_star in zip(self.parameters, self.m_star,
self.v_star):
                g = x.grad + self.weight_decay * x # Gradient with L2
regularization
                m, m_star[...] = ema_update(g, self.beta_1, self.t,
m_star) # mean of gradients
                v, v_star[...] = ema_update(g ** 2, self.beta_2,
self.t, v_star) # mean of squared gradients

################################################################
                # TODO: Implement ADAM optimization
                #
                # There are three steps:
                # (1) m and m_star are updated to incorporate g with
smoothing
                # beta_1 using ema_update(...).
                # (2) v and v_star are updated to incorporate g^2 with
smoothing
                # beta_2 using ema_update(...).
                #(3) x is updated according to ratio between the mean
and RMS
                # gradient estimates, using lr and epsilon.
                #
                # Remember to advance the timestep t before each step.
                # Remember to update m_star and v_star in-place.
                # There is a difference between saying m_star =
something and
                # m_star[...] = something.

################################################################

                x[...] = x - self.lr * m / (torch.sqrt(v) + self.eps)
# bias corrected moment estimates


################################################################
                #                                 END OF YOUR CODE
#

################################################################
```

```
    def zero_grad(self):
        for x in self.parameters:
            x.grad = None


# TEST CODE BELOW
# 'x' variable = is the current estimate which will be updated
iteratively during the optimization.
# 'L' function = the quadratic loss function we will use.  This one
can also plot its inputs and outputs.
# 'learning_rate' = is the step size that is used to update the
solution

x = torch.tensor([-1.5, 1.2])
x.requires_grad = True # This tells PyTorch that the x variable will
be used to calculate gradients.

L = LossFunctionWithPlot()

learning_rate = 0.1
weight_decay = 0.1

# If you switch this for torch.optim.Adam, it should behave exactly
the same.
optimizer = ADAMOptimizer([x], lr=learning_rate,
weight_decay=weight_decay)

for iter in range(50):
    loss = L(x)
    if iter % 7 == 0: print(f'Loss at step {iter} is
{loss.item():.3f}')
    loss.backward()
    with torch.no_grad():
        optimizer.step()
    optimizer.zero_grad()

L.plot_history()
```

```
Loss at step 0 is 3.937
Loss at step 7 is 2.054
Loss at step 14 is 0.186
Loss at step 21 is 0.605
Loss at step 28 is 0.100
Loss at step 35 is 0.162
Loss at step 42 is 0.090
Loss at step 49 is 0.027
```



Part 3: Training a Neural Network

```
# ================================================================
#                  TENSORBOARD INTEGRATION: SETUP
# ================================================================
# This cell should be placed at the beginning of Part 3.

# 1. Import the SummaryWriter class
from torch.utils.tensorboard import SummaryWriter

# 2. Load the TensorBoard notebook extension
%load_ext tensorboard
# ================================================================
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

Tiny Classification Problem

The following code loads raw data for a tiny classification problem.

The training data has 10000 samples, each a vector of 36 numbers along with a corresponding set of 10000 labels, assigning 0 or 1 to each sample. The test data has 2000 samples and labels that are disjoint from the training data.

```
train_data, train_labels, test_data, test_labels = [
    torch.tensor(m[k]).float()
    for m in [numpy.load('tiny-classification.npz')]
    for k in 'train_data train_labels val_data val_labels'.split()]

print(f'The training data has {train_data.size(0)} samples, each a
vector of {train_data.size(1)} numbers along with')
print(f'a corresponding set of {train_labels.size(0)} labels,
```

```
assigning {train_labels.min()} or {train_labels.max()} to each
sample.')

print(f'The test data has {test_data.size(0)} samples and labels that
are disjoint from the training data.')
```

```
The training data has 8000 samples, each a vector of 36 numbers along
with
a corresponding set of 8000 labels, assigning 0.0 or 1.0 to each
sample.
The test data has 1000 samples and labels that are disjoint from the
training data.
```

The following code cell serves the purpose of training a neural network, tracking its performance throughout the training process, and generating visual representations of the training progress. It is crucial for students to grasp the functionality of the `run_test` function and `Supervise` class as it will be frequently utilized in the subsequent tasks.

Comments have been thoughtfully included to enhance code comprehension.

```
def run_test(net, optmaker, experiment_name=None, hparam_dict=None,
test_every=10):
    # ================= TENSORBOARD INTEGRATION =================
    # Create a SummaryWriter to log data for this specific run
    if experiment_name:
        writer = SummaryWriter(f'runs/{experiment_name}')
    # ===========================================================

    # Set up the Loss Function and Optimizer
    optimizer = optmaker(net.parameters()) # Initialize the optimizer
with model parameters
    print(f'{sum([p.numel() for p in net.parameters()])} parameters')
    train_losses, train_accs, test_accs = [], [], []

    # ================= TENSORBOARD INTEGRATION =================
    # Add the model graph to TensorBoard for visualization
    # We use a sample of the training data as input.
    if experiment_name:
        writer.add_graph(net.net, train_data.float())
    # ===========================================================

    for epoch in range(2000):
        loss = net(train_data.float(), train_labels.float())
        loss.backward()

        # ================= TENSORBOARD INTEGRATION =================
        # Log the training loss at every epoch
        if experiment_name:
            writer.add_scalar('Loss/train', loss.item(), epoch)
        # ===========================================================
```

```
        train_losses.append([epoch, loss.item()])
        optimizer.step() # Update model parameters using the
optimizer's update rule
        if epoch % test_every == test_every - 1:
            grads = torch.stack([p.grad.abs().max() for p in
net.parameters()])
            maxg, ming = grads.abs().max(), grads.abs().min()
            train_outputs = net.net(train_data.float())
            train_preds = (train_outputs.squeeze() > 0.5).float()
            train_accuracy = (train_preds ==
train_labels).float().mean()
            train_accs.append([epoch + 1, train_accuracy])
            net.eval()
            test_outputs = net.net(test_data.float())
            net.train()
            test_preds = (test_outputs.squeeze() > 0.5).float()
            test_accuracy = (test_preds == test_labels).float().mean()
            test_accs.append([epoch + 1, test_accuracy])

            # ================= TENSORBOARD INTEGRATION
=================
            # Log training and test accuracy at each evaluation step
            if experiment_name:
                writer.add_scalar('Accuracy/train',
train_accuracy.item(), epoch)
                writer.add_scalar('Accuracy/test',
test_accuracy.item(), epoch)
            #
============================================================

            print(f'Epoch {epoch+1}, Loss: {loss.item():.5f}, Grad
range {maxg:.1e} to {ming:.1e}, '
                  f'Train Accuracy: {train_accuracy.item()}, Test
Accuracy: {test_accuracy.item()}', end='   \r')
            if test_accuracy.item() > 0.999:
                break
        optimizer.zero_grad()

    # Test the Model
    with torch.no_grad():
        train_outputs = net.net(train_data.float())
        train_preds = (train_outputs.squeeze() > 0.5).float()
        train_accuracy = (train_preds == train_labels).float().mean()
        net.eval()
        test_outputs = net.net(test_data.float())
        net.train()
        test_preds = (test_outputs.squeeze() > 0.5).float()
        test_accuracy = (test_preds == test_labels).float().mean()
        print(f'\nTrain Accuracy: {train_accuracy.item():.5f}, Test
```

```
Accuracy: {test_accuracy.item():.5f}')

    # ================= TENSORBOARD INTEGRATION =================
    # Log the hyperparameters and final metrics for this run
    if experiment_name:
        final_metrics = {
            'accuracy/final_train': train_accuracy.item(),
            'accuracy/final_test': test_accuracy.item()
        }
        writer.add_hparams(hparam_dict, final_metrics)
        writer.close() # Make sure to close the writer
    # ===========================================================


    # Visulaiztion
    fig, ax = plt.subplots()
    ax2 = ax.twinx()
    ax.plot(*zip(*train_losses), label="Training loss")
    ax.set_yscale('log')
    ax2.plot(*zip(*train_accs), color="orange", label="Training
accuracy")
    ax2.plot(*zip(*test_accs), color="red", label="Test accuracy")
    ax2.set_ylim(0.0, 1.0)
    for a in [ax, ax2]:
        for pos in 'top right bottom left'.split():
            a.spines[pos].set_visible(False)
    ax.set_xlabel('Epochs')
    ax.set_ylabel('Loss')
    ax2.set_ylabel('Accuracy')
    fig.legend(loc="lower left", bbox_to_anchor=(0, 0),
bbox_transform=ax.transAxes)
    fig.show()

print(f'Data width {train_data.size(1)}; Constant baseline accuracy
{max(test_labels.sum(), len(test_labels) - test_labels.sum()) /
len(test_labels):.3f}')
```

```
Data width 36; Constant baseline accuracy 0.500
```

The `Supervise` class is a wrapper that combines a neural network model and a loss function to facilitate supervised learning tasks. It computes the loss by performing a forward pass through the neural network and comparing the predicted values to the true labels.

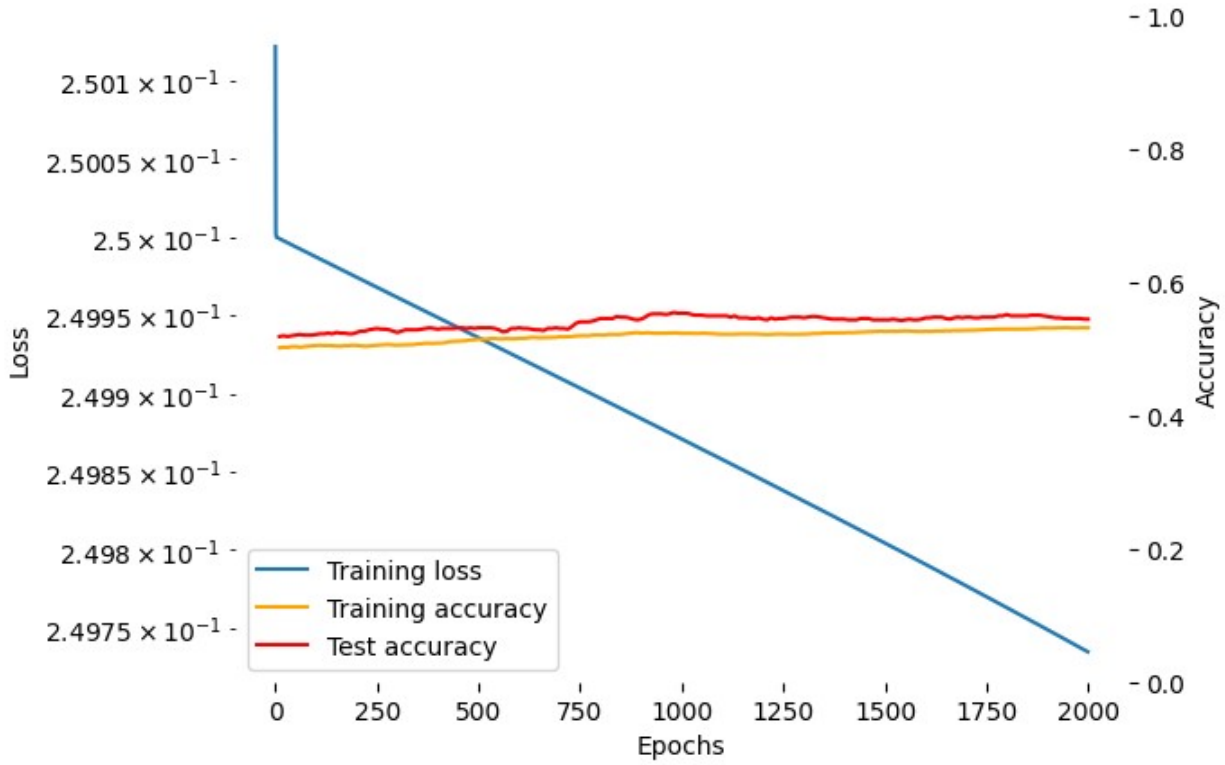
```
class Supervise(Module):
    def __init__(self, criterion, net):
        super().__init__()
        self.net = net
        self.criterion = criterion
    def forward(self, x, y):
```

```
        out = self.net(x).squeeze()
        return self.criterion(out, y)
```

Here is an example on how to train a model using `Supervise` and `run_test` function

```
##############################################################################
##########
# TODO: Build a Neural Network which has the architecture as follows:-
# Hidden Dimension - 256
# Loss function - Mean Squared Error (MSE)
# Optimizer - Simple GradientDescent (lr=0.1) [Using the optimizer
built in Task 1.3]
# Network Architecture - (Linear + Sigmoid) -> (Linear + Sigmoid) ->
(Linear + Sigmoid)
##############################################################################
##########

input_size=train_data.size(1)
hidden_dims= 256
output_dims=1

run_test(
    Supervise(
    nn.MSELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.Sigmoid(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.Sigmoid(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: SimpleGradientDescent(p,lr=0.1),
    experiment_name="Example-Sigmoid_SimpleGradientDescent-0.1", #
This name will appear in TensorBoard, feel free use any convention you
like
    hparam_dict = {
        'activation': 'Sigmoid',
        'optimizer': 'SimpleGradientDescent',
        'loss': 'MSE',
        'lr': 0.1,
        'hidden_dim': hidden_dims,
        'depth': 3
    }
)
##############################################################################
##########
#                               END OF YOUR CODE
```

```
#
##########################################################################
##########

75521 parameters
Epoch 2000, Loss: 0.24973, Grad range 1.8e-04 to 2.2e-06, Train
Accuracy: 0.5318750143051147, Test Accuracy: 0.5450000166893005
Train Accuracy: 0.53188, Test Accuracy: 0.54500
```



Configuring and training various Neural Network Architectures

Let's construct a range of neural network architectures with varying configurations to explore their impact on training performance when considering factors such as:

- Activation functions
- Choice of optimizers
- Regulariztion
- Hidden layer dimensions
- Network depth
- Batch normalization
- Residual networks.

This experimentation will help us gain insights into how these factors influence the training process on our dataset.

Note - Utilize the previously defined `SimpleGradientDescent` and `ADAMOptimizer` optimizers exclusively, unless otherwise specified in the task to employ PyTorch's built-in optimizers.

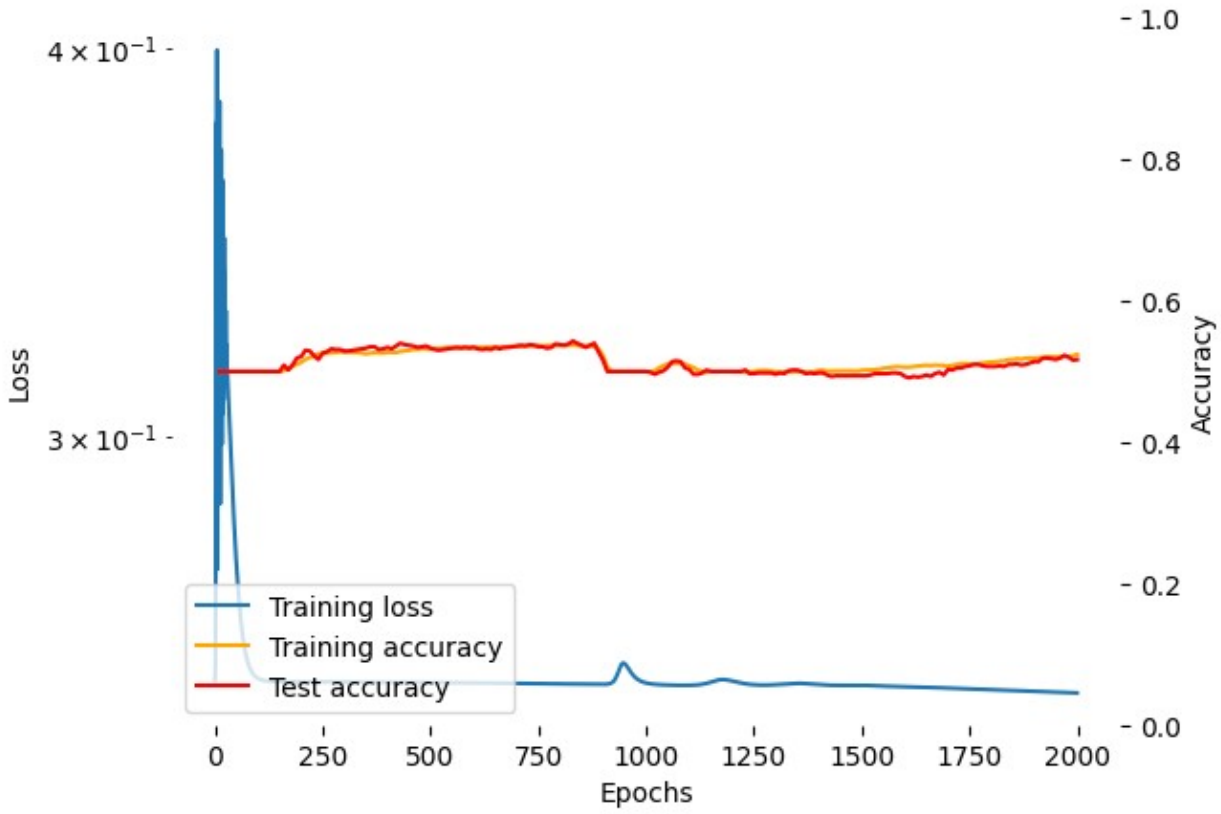
1) Activation functions

Task 3.1 - Test sigmoid activations in a three-layer network with `run_Test` (1 point)

Let's apply our new `run_test` function together with your `SimpleGradientDescent` optimizer class to test (assuming that is the class name you gave it). You will need to put in your three-layer sigmoid network definition to make the example work.

```
torch.manual_seed(7150) # (Leave it here for deterministic behavior
and easier grading)
##########################################################################
##########
# TODO: Create an architecture similar to the example but with reduced
hidden
# dimension
# Hidden Dimension - 128
# Loss - MSELoss()
# Optimizer - Simple GradientDescent - (lr = 1.0) [Use the Optimizer
built above]
# Network Architecture - (Linear + Sigmoid) -> (Linear + Sigmoid) ->
(Linear + Sigmoid)
##########################################################################
##########

input_size = train_data.size(1)
hidden_dims = 128
output_dims = 1

run_test(
    Supervise(
        nn.MSELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.Sigmoid(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.Sigmoid(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: SimpleGradientDescent(p, lr=1.0),
    experiment_name="Task3.1-Sigmoid_SimpleGD-1.0",
    hparam_dict={
        'activation': 'Sigmoid',
        'optimizer': 'SimpleGradientDescent',
        'loss': 'MSE',
```

```
        'lr': 1.0,
        'hidden_dim': hidden_dims,
        'depth': 3
    }
)

##############################################################################
##########
#                                END OF YOUR CODE
#
##############################################################################
##########

21377 parameters
Epoch 2000, Loss: 0.24789, Grad range 1.5e-02 to 3.6e-04, Train
Accuracy: 0.5241249799728394, Test Accuracy: 0.5170000195503235
Train Accuracy: 0.52412, Test Accuracy: 0.51700
```



Task 3.2 - Test Tanh activations in a three-layer network (1 point)

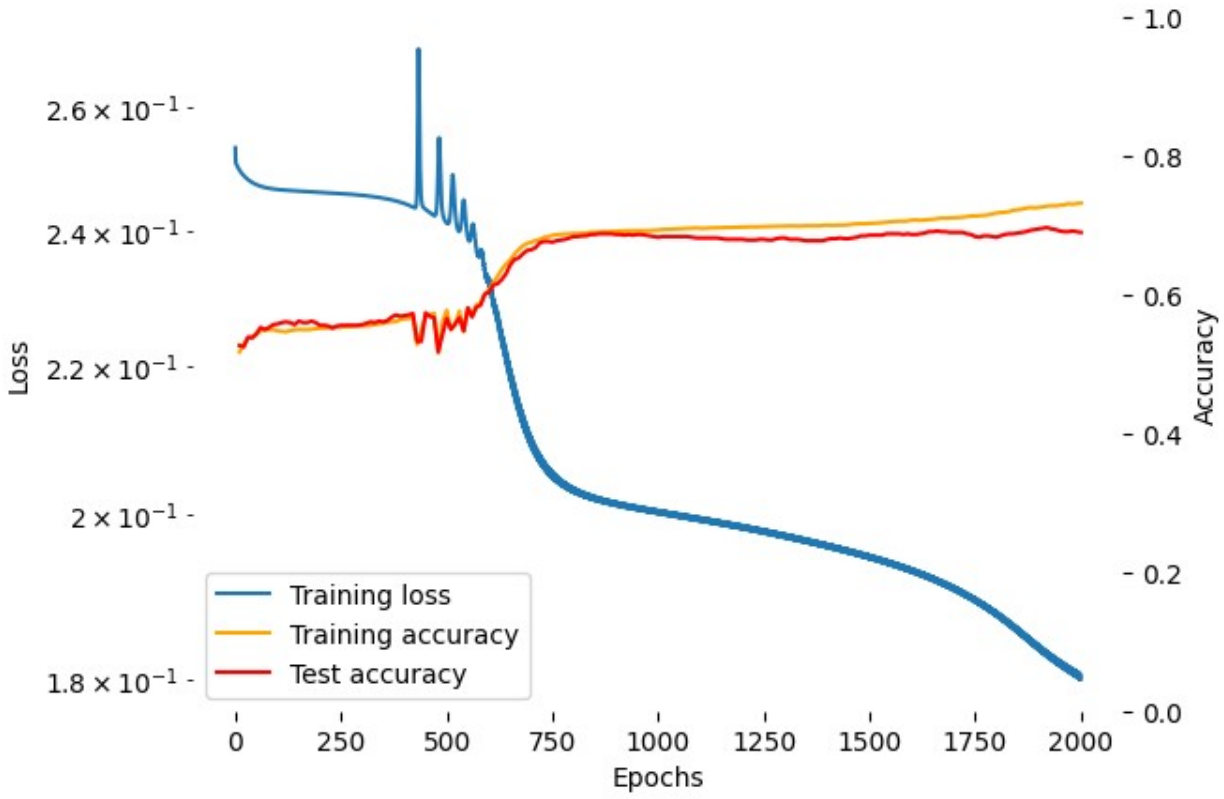
Implement the same neural network architecture

Read documentation - https://pytorch.org/docs/stable/generated/torch.nn.Tanh.AC

Now let us compare Sigmoid to other activations. Switch every `nn.Sigmoid()` to `nn.Tanh()`. Tanh balances positive and negative outputs and sometimes work better. Implement and decide whether it is helping in this case.

```
torch.manual_seed(7150) # (Leave it here for deterministic behavior
and easier grading)
##########################################################################
##########
# TODO: Create an architecture similar to the previous one, but
replace every
# nn.Sigmoid() activation function with nn.Tanh() (except for the last
one).
# Hidden Dimension - 128
# Loss - MSELoss()
# Optimizer - Simple GradientDescent - (lr = 1.0) [Use the Optimizer
built above]
# Network Architecture - (Linear + Tanh) -> (Linear + Tanh) -> (Linear
+ Sigmoid)
##########################################################################
##########

input_size = train_data.size(1)
hidden_dims = 128
output_dims = 1

run_test(
    Supervise(
        nn.MSELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.Tanh(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.Tanh(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: SimpleGradientDescent(p, lr=1.0),
    experiment_name="Task3.2-Tanh_SimpleGD-1.0",
    hparam_dict={
        'activation': 'Tanh',
        'optimizer': 'SimpleGradientDescent',
        'loss': 'MSE',
        'lr': 1.0,
        'hidden_dim': hidden_dims,
        'depth': 3
    }
)

##########################################################################
```

```
#####
#                               END OF YOUR CODE
#
##############################################################################
#####

21377 parameters
Epoch 2000, Loss: 0.17992, Grad range 2.6e-02 to 7.6e-03, Train
Accuracy: 0.7322499752044678, Test Accuracy: 0.6899999976158142
Train Accuracy: 0.73225, Test Accuracy: 0.69000
```

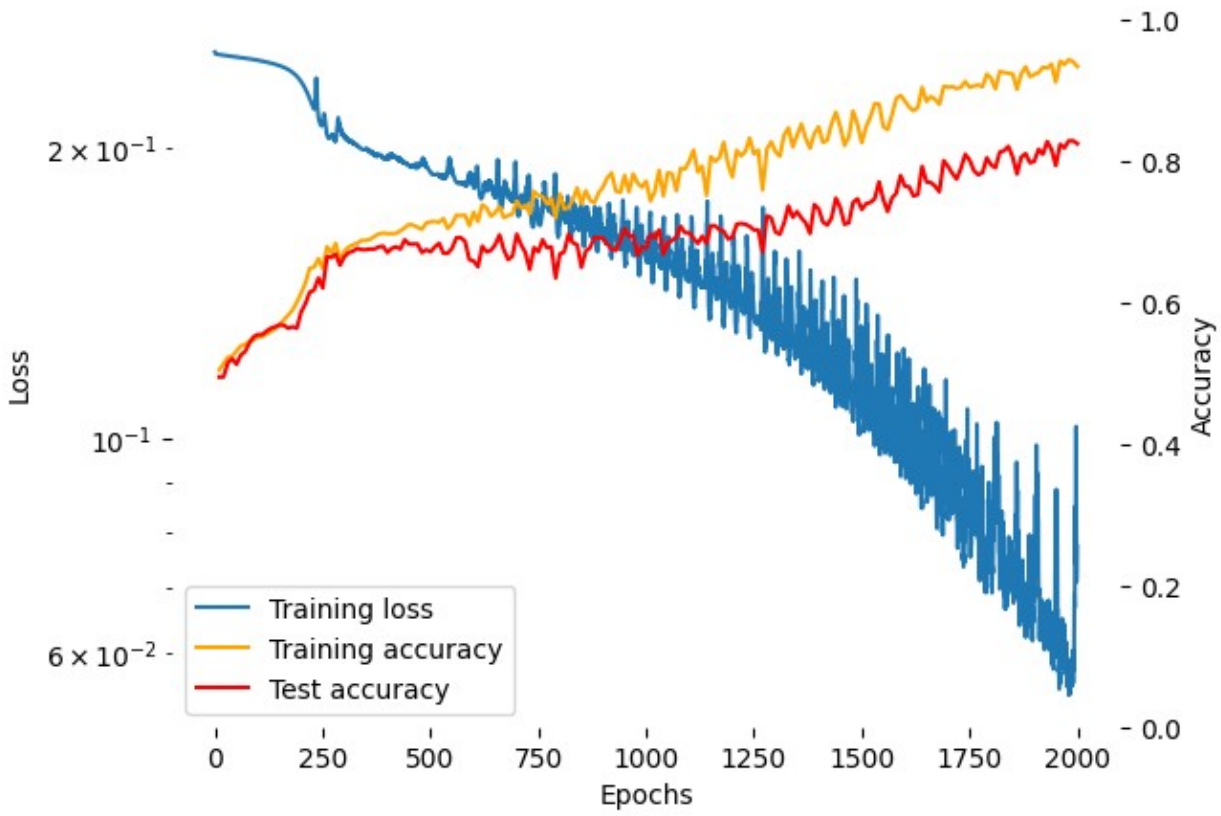


Task 3.3 - Test ReLU activations in a three-layer network (1 point)

The ReLU activation was studied closely by Glorot, which we have discussed in class. It tends to be very effective at avoiding vanishing gradients, because on the positive side it never saturates. Replace all your nonlinearties with ReLU while keeping the architecture otherwise the same. Does ReLU help in this case?

```
torch.manual_seed(7150) # (Leave it here for deterministic behavior
and easier grading)
##############################################################################
##########
# TODO: Create an architecture similar to the previous one, but
replace every
# nn.Tanh() activation function with nn.ReLU() (except for the last
```

```
one).
# Hidden Dimension - 128
# Loss - MSELoss()
# Optimizer - SimpleGradientDescent - (lr = 1.0)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear
+ Sigmoid)
##############################################################################
###########

input_size = train_data.size(1)
hidden_dims = 128
output_dims = 1

run_test(
    Supervise(
        nn.MSELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: SimpleGradientDescent(p, lr=1.0),
    experiment_name="Task3.3-ReLU_SimpleGD-1.0",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'SimpleGradientDescent',
        'loss': 'MSE',
        'lr': 1.0,
        'hidden_dim': hidden_dims,
        'depth': 3
    }
)

##############################################################################
#####
#                             END OF YOUR CODE
#
##############################################################################
#####

21377 parameters
Epoch 2000, Loss: 0.07740, Grad range 3.3e-02 to 1.3e-02, Train
Accuracy: 0.9341250061988831, Test Accuracy: 0.824999988079071
Train Accuracy: 0.93413, Test Accuracy: 0.82500
```



3.A) Inline Question (1 point):
In the above Tasks, we employed three different activation functions—sigmoid, tanh, and relu—in our neural network architecture. Describe the difference in the behavior of the optimization process that you observed between Sigmoid, TanH, and ReLU. Do your results confirm or contradict the results that Glorot reported in his 2010 study?

The results in our notebook **confirm** Glorot's findings. They showed that:

- Sigmoid activations cause gradients to vanish in deeper layers, making training increasingly difficult as depth grows.
- Tanh activations improve over Sigmoid due to zero-centering and a steeper gradient, yet still suffer from saturation.
- ReLU-type activations address the vanishing gradient problem because they maintain a constant gradient of 1 for positive inputs.

The experiments in our notebook reproduce this exact hierarchy: **Sigmoid ≪ Tanh < ReLU**, both in terms of final accuracy and in terms of the gradient magnitudes observed during training.

Task 3.4 - Implement Binary Cross Entropy Loss (1 point)

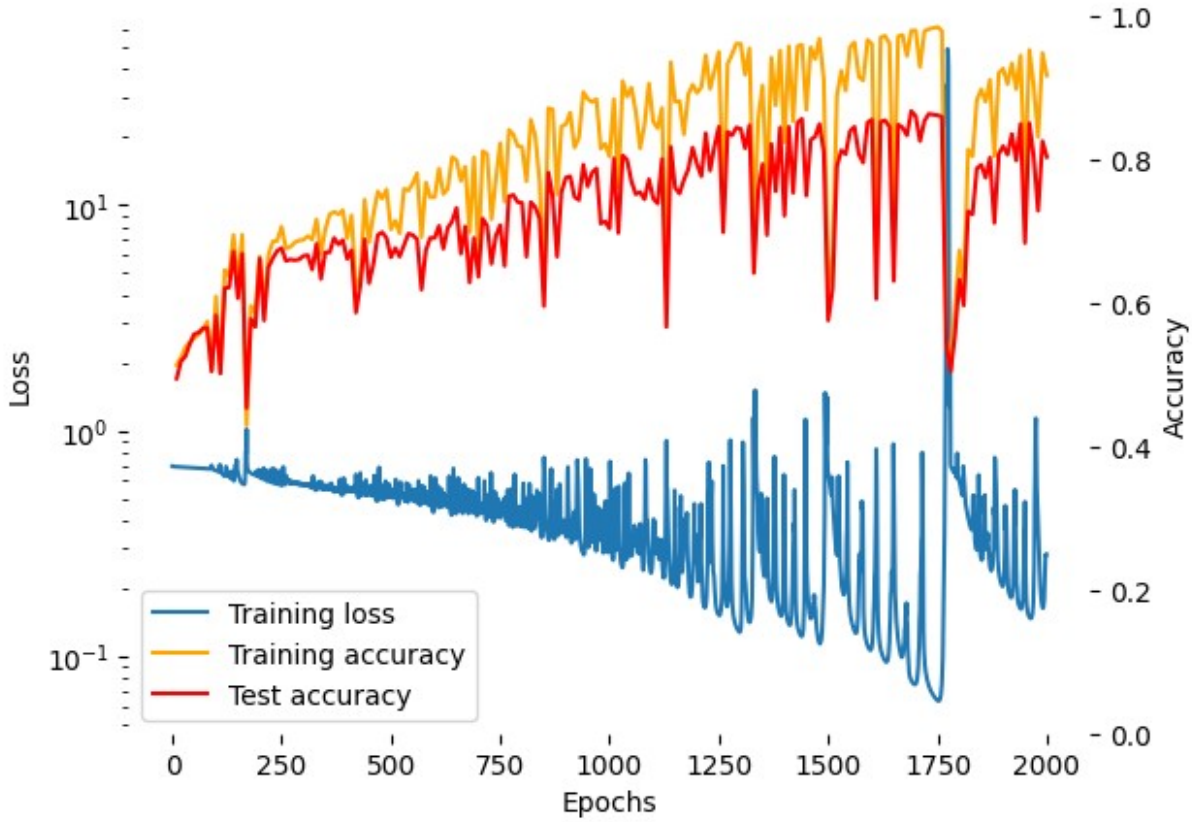
In a classification setting, we often prefer to interpret the outputs as probabilities and drive the probability distribution towards the true distribution. The standard way to achieve that is to use the cross-entropy loss. Cross-entropy (as seen in HW1) also test to avoid saturation when compared to MSE, when used in combination with softmax.

Replace the supervision with `BCELoss` rather than mean square error, and observe any differences.

Read documentation for BCE Loss - https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html

```
torch.manual_seed(7150) # (Leave it here for deterministic behavior and easier grading)
################################################################################
# TODO: Build a neural network with following architecture:
# Hidden Dimension - 128
# Loss - Binary Cross Entropy
# Optimizer - SimpleGradientDescent - (lr = 1.0)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear + Sigmoid)
################################################################################

input_size = train_data.size(1)
hidden_dims = 128
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: SimpleGradientDescent(p, lr=1.0),
    experiment_name="Task3.4-ReLU_BCE_SimpleGD-1.0",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'SimpleGradientDescent',
        'loss': 'BCE',
        'lr': 1.0,
        'hidden_dim': hidden_dims,
        'depth': 3
    }
)

################################################################################
#                               END OF YOUR CODE
```

```
#
##########################################################################
#####

21377 parameters
Epoch 2000, Loss: 0.28317, Grad range 1.2e-01 to 2.9e-02, Train
Accuracy: 0.9177500009536743, Test Accuracy: 0.8040000200271606
Train Accuracy: 0.91775, Test Accuracy: 0.80400
```



2) Optimizers

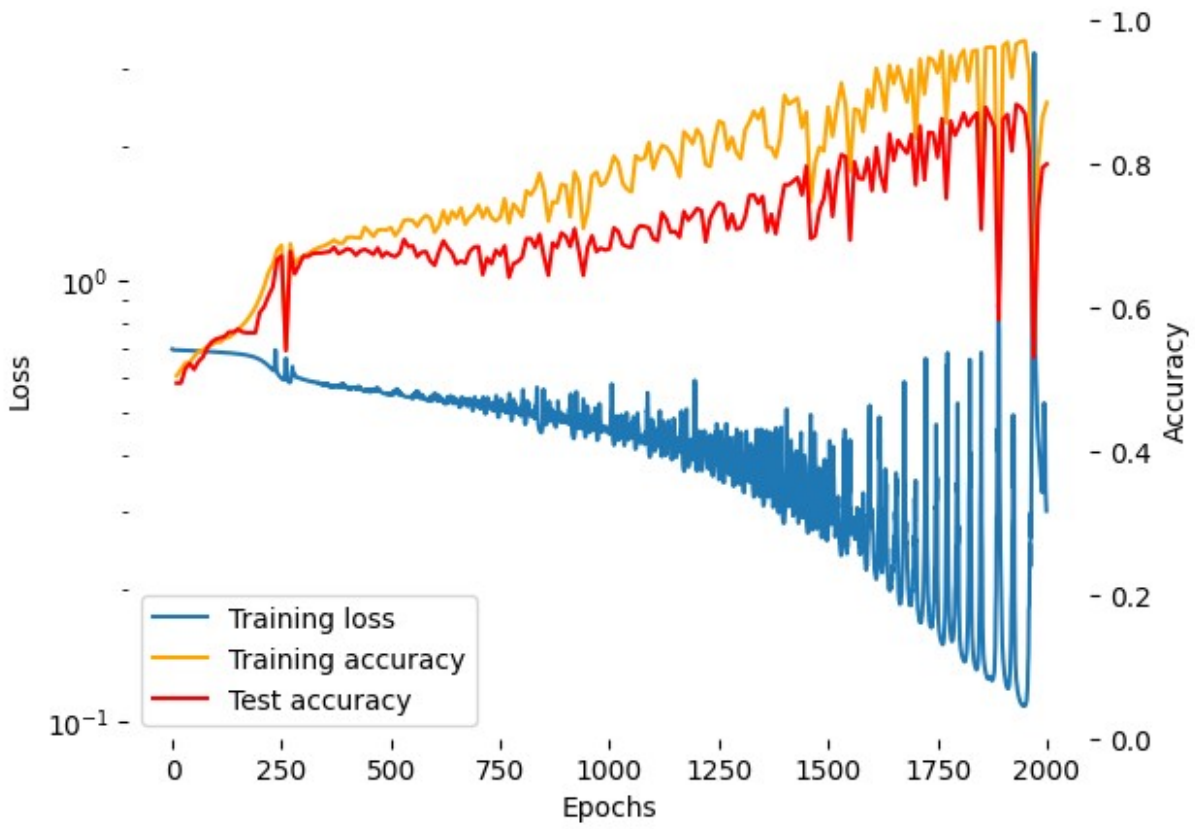
Task 3.5 Implement using SGD Optimizer (1 point)

So far we have been using our own `SimpleGradientDescent`. Now try comparing results with pytorch's built-in `torch.optim.SGD` class. How does your implementation compare? Is it the same?

Read Documentation for SGD Optimizer using Pytorch - https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

```
torch.manual_seed(7150) # (Leave it here for deterministic behavior
and easier grading)
##########################################################################
##########
```

```
# TODO: Build a neural network with following architecture, Careful,
Hidden
# Dimension has changed.
# Hidden Dimension - 128
# Loss - Binary Cross Entropy
# Optimizer - SGD - (lr = 0.5)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear
+ Sigmoid)
##########################################################################
##########

input_size = train_data.size(1)
hidden_dims = 128
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.SGD(p, lr=0.5),
    experiment_name="Task3.5-ReLU_BCE_SGD-0.5",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'SGD',
        'loss': 'BCE',
        'lr': 0.5,
        'hidden_dim': hidden_dims,
        'depth': 3
    }
)

##########################################################################
#####
#                               END OF YOUR CODE
#
##########################################################################
#####

21377 parameters
Epoch 2000, Loss: 0.30000, Grad range 3.2e-02 to 1.6e-02, Train
Accuracy: 0.8855000138282776, Test Accuracy: 0.800000011920929
Train Accuracy: 0.88550, Test Accuracy: 0.80000
```



Task 3.6 - ADAM Optimizer (1 point)

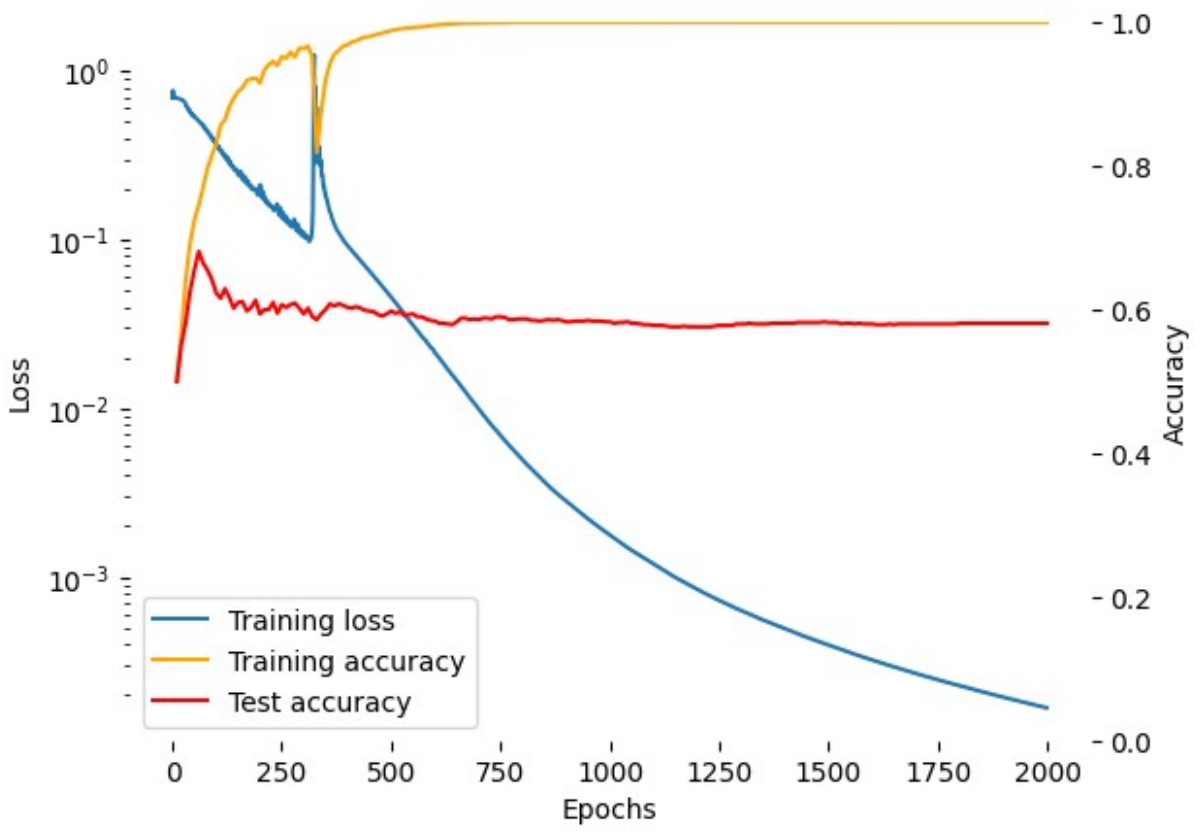
ADAM is a very powerful optimizer and should improve results.

Use your own `ADAMOptimizer` class here to see how it behaves. If you wish to debug against the standard ADAM optimizer, then you can try it out as well, but when you hand in your results, show what your `ADAMOptimzer` does. Ideally, they should behave the same.

Read Documentation for ADAM Optimizer using Pytorch - https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
########################################################################
##########
# TODO: Build a neural network with following architecture:
# Hidden Dimension - 128
# Loss - Binary Cross Entropy
# Optimizer - ADAM (weight decay = 0 ,lr = 0.01)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear
+ Sigmoid)
########################################################################
##########

input_size = train_data.size(1)
```

```
hidden_dims = 128
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: ADAMOptimizer(p, lr=0.01, weight_decay=0.0),
    experiment_name="Task3.6-ReLU_BCE_ADAM-0.01",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'ADAMOptimizer',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 3
    }
)

#########################################################################
#####
#                               END OF YOUR CODE
#
#########################################################################
#####

21377 parameters
Epoch 2000, Loss: 0.00017, Grad range 1.5e-04 to 1.9e-06, Train
Accuracy: 1.0, Test Accuracy: 0.5809999704360962
Train Accuracy: 1.00000, Test Accuracy: 0.58100
```



3.B) Inline Question (1 point): Does ADAMOptimzer do better? Explain how the optimization behaves differently than you observed with Simple Gradient Descent.

ADAM **converges faster** but **generalizes worse**.

Convergence Speed: ADAM reaches 100% training accuracy by ~epoch 500, with the loss dropping smoothly and monotonically from ~10^0 down to ~10^{-3}. In contrast, both SimpleGradientDescent and SGD struggle through 2000 epochs with spikey and noisy loss curves that never fully converge, ending at losses of around 0.283 and 0.300.

Stability: ADAM's adaptive per-parameter learning rates produce a smooth optimization trajectory. SGD and SimpleGD exhibit excessive spikes or "oscillations" in both loss and accuracy throughout training, indicating that a single global learning rate is not well-suited for this loss landscape.

Generalization: Despite optimization, ADAM **severely overfits**:

- ADAM: Train Accuracy = 100%, Test Accuracy = 58.1% (gap of ~42%)
- SimpleGD: Train Accuracy = 91.8%, Test Accuracy = 80.4% (gap of ~11%)
- SGD: Train Accuracy = 88.6%, Test Accuracy = 80.0% (gap of ~9%)

In this experiment, ADAM's powerful optimization appears to effectively memorize the training data (loss = 0.00017), driving the model into a sharp minimum that does not generalize well. The SGD variants, while noisier and slower, act as implicit regularizers: their inability to perfectly fit

the training data somehow yields better test performance. This demonstrates that a better optimizer doesn't automatically mean a better model.

The Battle Against Overfitting

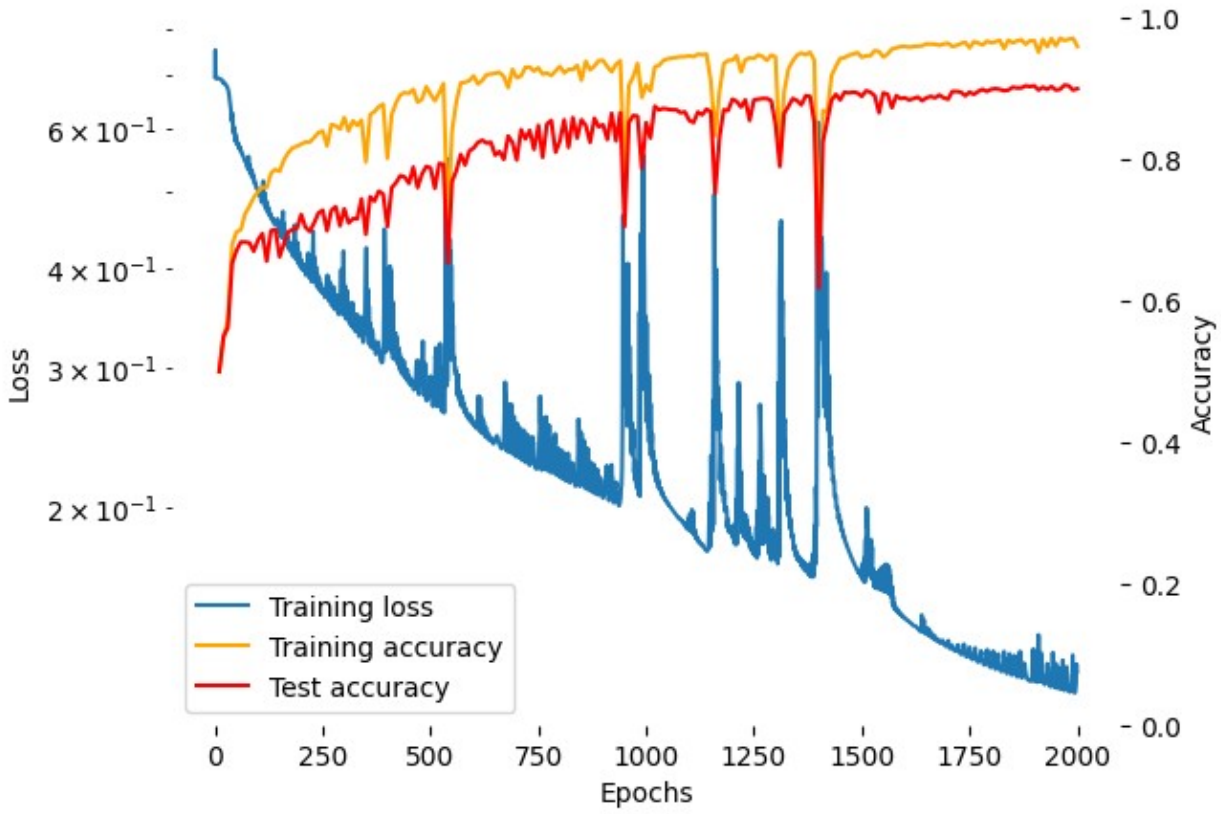
Let's apply some techniques to enhance and refine our model.

A) The Art of Regularization

Task 3.7 - Add a penalty term to the loss function that discourages large weights. This helps prevent the model from fitting noise in the data. (1 point)

```
torch.manual_seed(7150) # (Leave it here for deterministic behavior
and easier grading)
##########################################################################
##########
# TODO: Design a neural network with the specified architecture and
introduce a
# regularization term in the loss function to discourage the growth of
large
# weights.
# Hidden Dimension - 128
# Loss - Binary Cross Entropy
# Optimizer - Adam with weight decay - (lr = 0.01, weight_decay=1e-3)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear
+ Sigmoid)
##########################################################################
##########

input_size = train_data.size(1)
hidden_dims = 128
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.7-ReLU_BCE_ADAM-WeightDecay",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
```

```
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 3,
        'weight_decay': 1e-3
    }
)

##############################################################################
#####
#                               END OF YOUR CODE
#
##############################################################################
#####

21377 parameters
Epoch 2000, Loss: 0.12376, Grad range 2.0e-01 to 1.2e-02, Train
Accuracy: 0.9597499966621399, Test Accuracy: 0.8999999761581421
Train Accuracy: 0.95975, Test Accuracy: 0.90000
```



B) Slimming Down for Success

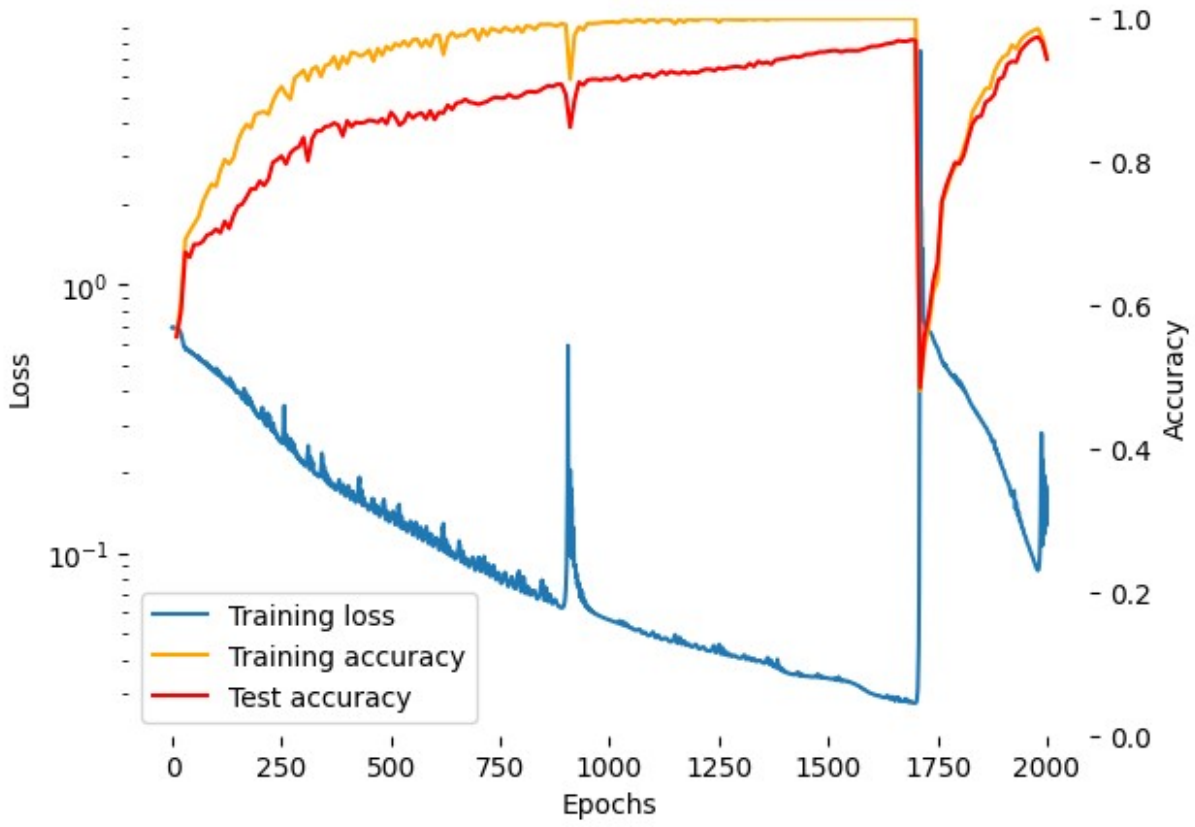
Reducing Hidden Dimension - Fewer parameters means that the model has less flexibility to fit the training data and it is forced to learn simpler features that are more likely to generalize to new data.

Task 3.8 - Going Deeper with Shrinking Hidden Dimensions (1 point)

Let's explore the effectiveness of dimension reduction as a technique by reducing the number of hidden dimensions from 128 to 64.

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
################################################################################
# TODO: Let's create a neural network with following architecture:
# Hidden Dimension - 64
# Loss - Binary Cross Entropy
# Optimizer - Adam with weight decay - (lr = 0.01, weight_decay=1e-3)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear
+ Sigmoid)
################################################################################

input_size = train_data.size(1)
hidden_dims = 64
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.8-ReLU_BCE_ADAM-HidDim64",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 3,
        'weight_decay': 1e-3
    }
)

################################################################################
#####
#                               END OF YOUR CODE
```

```
#
##########################################################################
#####

6593 parameters
Epoch 2000, Loss: 0.17321, Grad range 5.4e-01 to 4.1e-02, Train
Accuracy: 0.9481250047683716, Test Accuracy: 0.9430000185966492
Train Accuracy: 0.94813, Test Accuracy: 0.94300
```

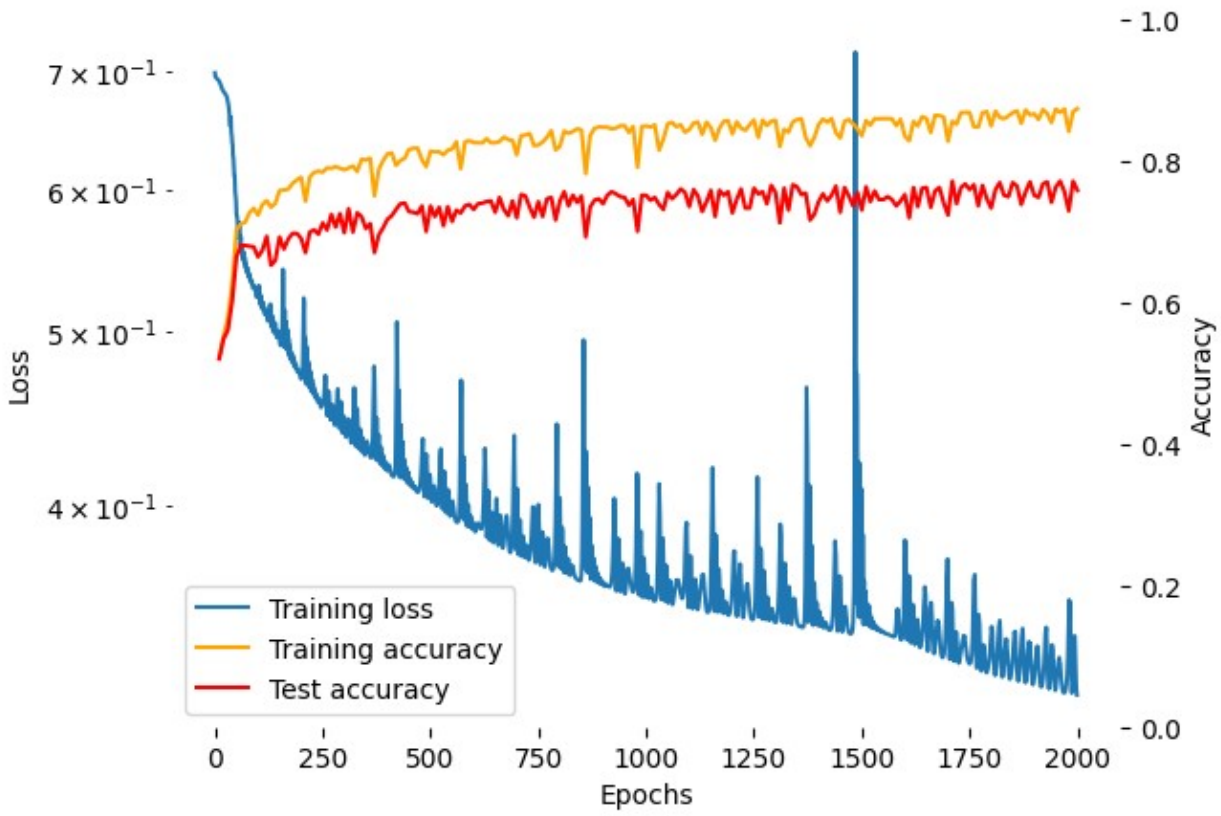


Task 3.10 - Going more Deeper with Shrinking Hidden Dimensions (1 point)

Dimension reduction improved our outcome. Let's continue by further reducing the hidden dimension from 64 to 32.

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
##########################################################################
##########
# TODO: Let's create a neural network with following architecture:
# Hidden Dimension - 32
# Loss - Binary Cross Entropy
# Optimizer - Adam with weight decay - (lr = 0.01, weight_decay=1e-3)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear
+ Sigmoid)
```

```
##########################################################################
##########

input_size = train_data.size(1)
hidden_dims = 32
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.10-ReLU_BCE_ADAM-HidDim32",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 3,
        'weight_decay': 1e-3
    }
)

##########################################################################
#####
#                             END OF YOUR CODE
#
##########################################################################
#####

2273 parameters
Epoch 2000, Loss: 0.31305, Grad range 1.2e-02 to 8.6e-05, Train
Accuracy: 0.8743749856948853, Test Accuracy: 0.7590000033378601
Train Accuracy: 0.87437, Test Accuracy: 0.75900
```



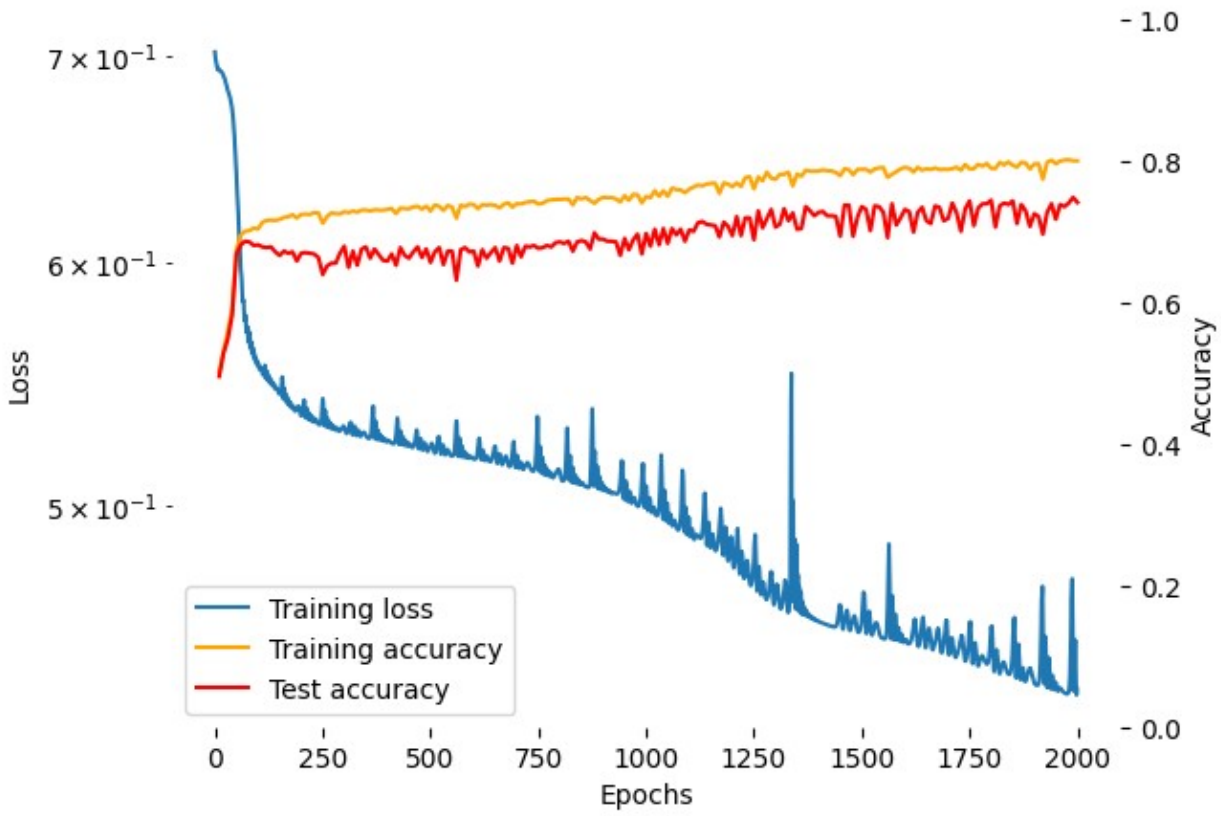
Task 3.11 - The Last Shrink (1 point)

Did it help to reduce to 32 dimensions?

Let's try one more time, maybe it will work? (Below, please try reducing the hidden dimension from 32 to 16.)

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
#########################################################################
#####################################
# TODO: Let's create a neural network with following architecture:
# Hidden Dimension - 16
# Loss - Binary Cross Entropy
# Optimizer - Adam with weight decay - (lr = 0.01, weight_decay=1e-3)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear
+ Sigmoid)
#########################################################################
#####################################

input_size = train_data.size(1)
hidden_dims = 16
output_dims = 1

run_test(
```

```
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.11-ReLU_BCE_ADAM-HidDim16",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 3,
        'weight_decay': 1e-3
    }
)

##############################################################################
#####
#                               END OF YOUR CODE
#
##############################################################################
#####

881 parameters
Epoch 2000, Loss: 0.43562, Grad range 1.1e-01 to 2.5e-02, Train
Accuracy: 0.8007500171661377, Test Accuracy: 0.7419999837875366
Train Accuracy: 0.80075, Test Accuracy: 0.74200
```



3.C) Inline Question - In the above Tasks [3.8 - 3.11] (Slimming Down for success) what were your key observations? (1 point)

Reducing hidden dimensions acts as a form of implicit regularization by constraining the model's capacity, but there is a clear sweet spot beyond which further reduction becomes counterproductive.

Task	Hidden Dim	Parameters	Train Acc	Test Acc	Gap
3.7	128	21,377	95.97%	90.00%	~6.0%
3.8	64	6,593	94.81%	94.30%	~0.5%
3.10	32	2,273	87.44%	75.90%	~11.5%
3.11	16	881	80.08%	74.20%	~5.9%

Note: Claude Opus 4.6 Thinking helped format this table.

1. **From 128 → 64 (Task 3.7 → 3.8):** Reducing hidden dimensions improved generalization. Test accuracy jumped from 90.0% to 94.3%, and the train–test gap collapsed from ~6% to ~0.5%. The model with 128 hidden units was overparameterized and still overfitting despite weight decay. Shrinking to 64 forced the network to learn more compact and generalizable representations. The loss curve also became much smoother and more stable.

2. **From 64 → 32 (Task 3.8 → 3.10):** This reduction went too far. Both training accuracy (87.4%) and test accuracy (75.9%) dropped, and the train–test gap widened to ~11.5%. The model now lacks sufficient capacity to learn the underlying decision boundary: it is **underfitting**. The loss curve shows the model struggling to decrease loss meaningfully, with persistent spikes indicating instability.

3. **From 32 → 16 (Task 3.10 → 3.11):** Performance degraded further. Training accuracy fell to 80.1% and test accuracy to 74.2%. With only 881 parameters, the network is severely underfitting: it can't effectively represent the complexity/pattern of the classification boundary. The loss plateaus at a relatively high value (~0.44) and the accuracy curves are essentially flat after the initial learning.

Conclusion:

There is a bias–variance tradeoff at play here. 64 hidden dimensions represent the optimal balance for this dataset: large enough to capture the true decision boundary, but small enough to avoid memorizing noise. Shrinking below this point does not improve generalization, but instead introduces underfitting where the model is too simple to learn even the training data well. Thus, we learned here that regularization via dimension reduction is effective only up to the point where the model's capacity to learn patterns still exceeds the complexity of the underlying pattern.

C) Layer by Layer

Increasing the number of layers.

Geoff Hinton likes to assert that deeper layers can capture increasingly abstract and high-level features in the data. This hierarchy allows the network to focus on relevant patterns and discard noise, making it less prone to fitting random variations in the training data.

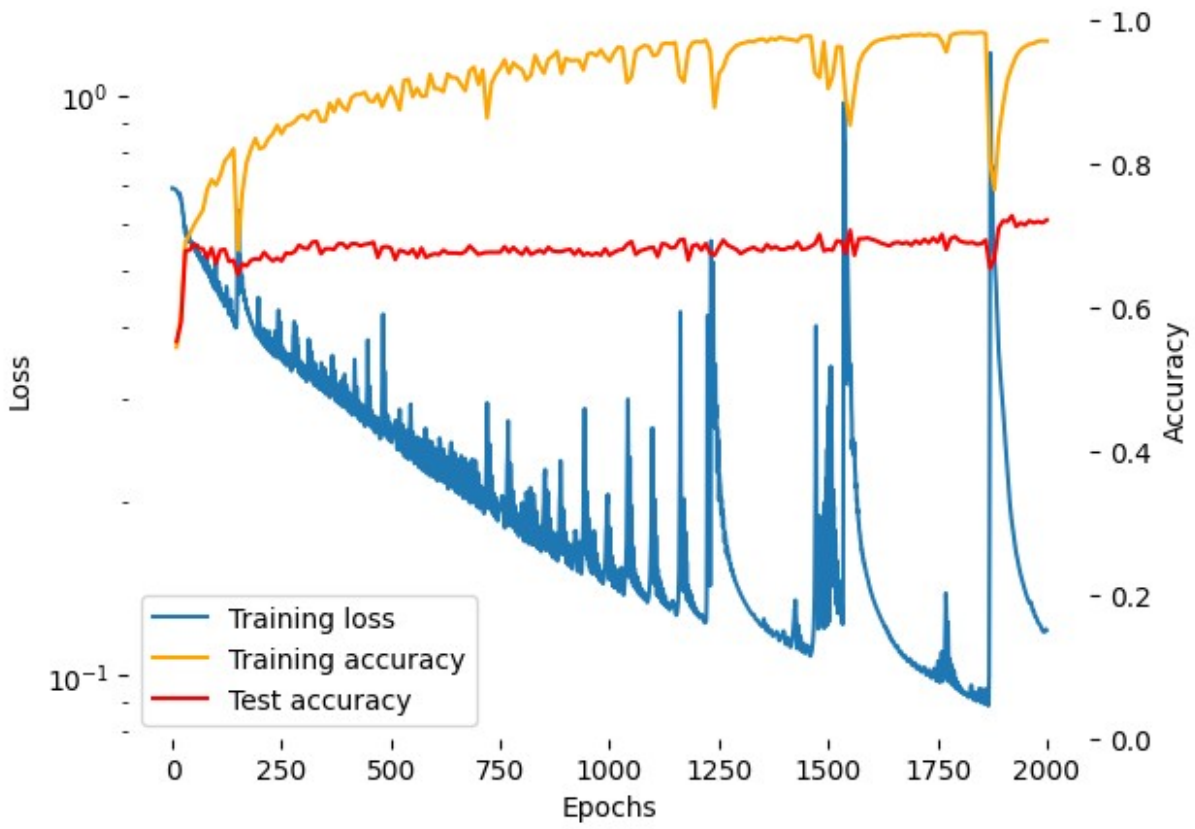
Is it true? Let's try it.

Task 3.12 - Increasing the Depth of the network (1 point)

Let's enhance our model by introducing an additional layer, creating a network with four layers.

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
#########################################################################
##########
# TODO: Let's create a neural network with following architecture:
# Hidden Dimension - 64
# Loss - Binary Cross Entropy
# Optimizer - Adam (lr = 0.01, weight_decay=1e-3)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear
+ ReLU)
# -> (Linear + Sigmoid)
#########################################################################
##########
```

```
input_size = train_data.size(1)
hidden_dims = 64
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.12-ReLU_BCE_ADAM-4Layers-HidDim64",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 4,
        'weight_decay': 1e-3
    }
)

#########################################################################
#####
#                               END OF YOUR CODE
#
#########################################################################
#####

10753 parameters
Epoch 2000, Loss: 0.11988, Grad range 1.1e-01 to 1.0e-02, Train
Accuracy: 0.9713749885559082, Test Accuracy: 0.722000002861023
Train Accuracy: 0.97137, Test Accuracy: 0.72200
```

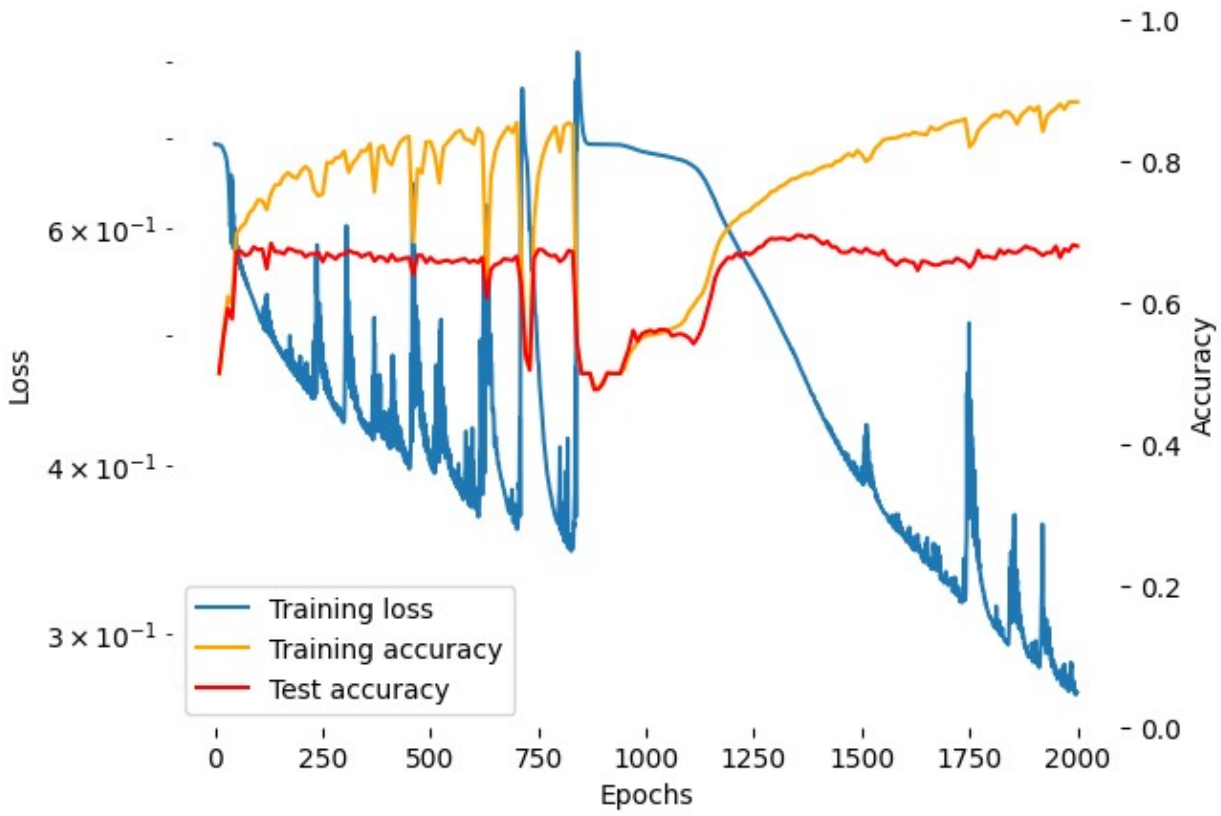


Task 3.13 - Increasing the Depth and reducing the width of the network (1 point)

The recent modification yielded remarkable results. Now, let's take it a step further by enhancing our model's architecture: we'll increase the number of layers from 4 to 6 and reduce the hidden dimension from 64 to 32.

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
##########################################################################
##########
# TODO: Let's create a neural network with following architecture:
# Hidden Dimension - 32
# Loss - Binary Cross Entropy
# Optimizer - Adam with weight decay - (lr=0.005, weight_decay=1e-4)
# Network Architecture - (Linear + ReLU) -> (Linear + ReLU) -> (Linear
+ ReLU)
#-> (Linear + ReLU) -> (Linear + ReLU) -> (Linear + Sigmoid)
##########################################################################
##########

input_size = train_data.size(1)
hidden_dims = 32
output_dims = 1
```

```
run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.005, weight_decay=1e-4),
    experiment_name="Task3.13-ReLU_BCE_ADAM-6Layers-HidDim32",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.005,
        'hidden_dim': hidden_dims,
        'depth': 6,
        'weight_decay': 1e-4
    }
)

##########################################################################
#####
#                               END OF YOUR CODE
#
##########################################################################
#####

5441 parameters
Epoch 2000, Loss: 0.27081, Grad range 3.9e-01 to 9.0e-03, Train
Accuracy: 0.8842499852180481, Test Accuracy: 0.6800000071525574
Train Accuracy: 0.88425, Test Accuracy: 0.68000
```



D) Batch Normalization

Batch Normalization is a technique used to improve the training of deep neural networks. It works by normalizing the activations of each layer, which helps to prevent the network from becoming too sensitive to the initialization of the weights and the order of the training data."

Pytorch documentation - https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html#torch.nn.BatchNorm1d

Task 3.14 Adding Batch Normalization (1 point)

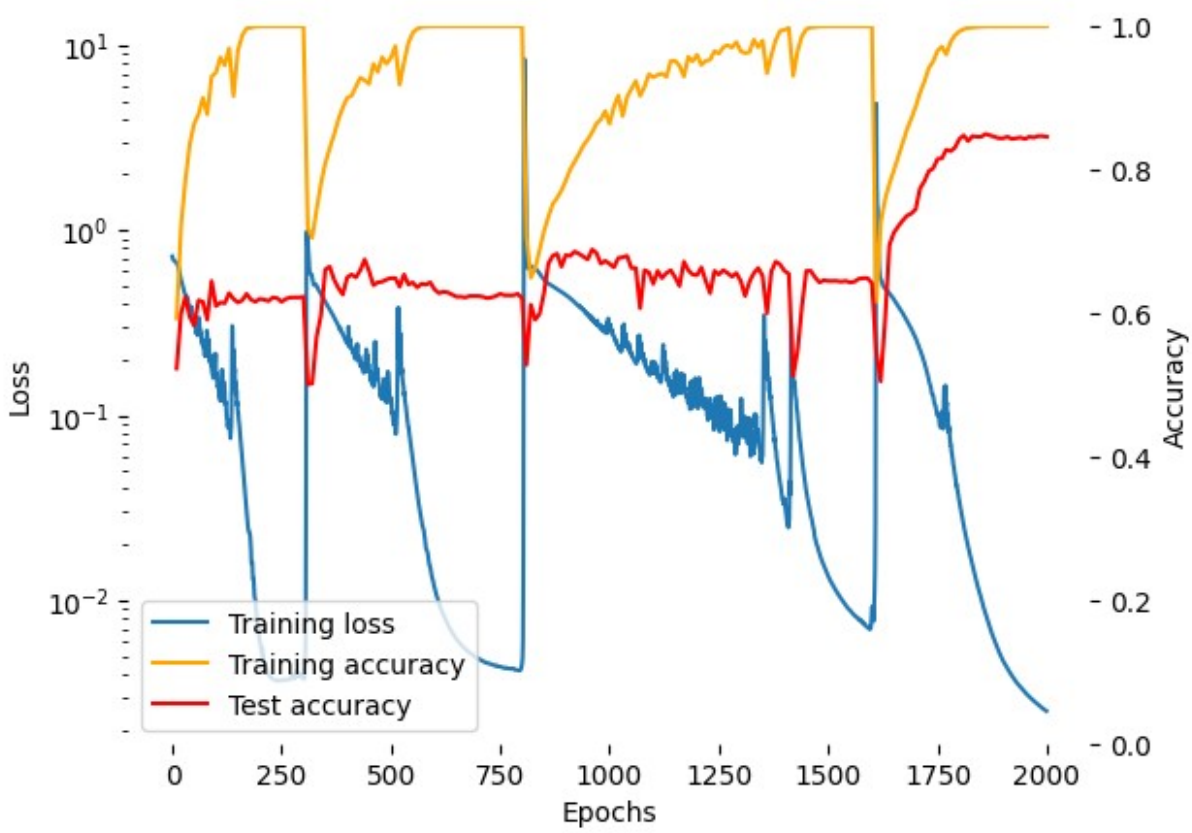
In the previous tasks, our model with a hidden dimension of 32 didn't deliver the desired performance. To address this, let's incorporate Batch Normalization into that architecture and assess whether it can enhance its performance.

Read documentation - https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
#########################################################################
##########
# TODO: Let's create a neural network with following architecture:
# Hidden Dimension - 32
# Loss - Binary Cross Entropy
# Optimizer - Adam (lr = 0.01, weight_decay=1e-3)
# Network Architecture - (Linear + ReLU + batch_normalization) ->
```

```
# (Linear + ReLU) -> (Linear + ReLU) -> (Linear + ReLU) ->
# (Linear + ReLU + batch_normalization) -> (Linear + Sigmoid)
##########################################################################
##########

input_size = train_data.size(1)
hidden_dims = 32
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.BatchNorm1d(hidden_dims),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.BatchNorm1d(hidden_dims),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.14-ReLU_BCE_ADAM-BatchNorm-HidDim32",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 6,
        'weight_decay': 1e-3
    }
)

##########################################################################
#####
#                             END OF YOUR CODE
#
##########################################################################
#####
```

```
5569 parameters
Epoch 2000, Loss: 0.00252, Grad range 2.5e-03 to 2.3e-04, Train
Accuracy: 1.0, Test Accuracy: 0.8460000157356262
Train Accuracy: 1.00000, Test Accuracy: 0.84600
```



E) Residual Networks

Residual Networks (ResNets) are a type of deep neural network that are designed to address the problem of vanishing gradients.

Task 3.15 - Modifiy the code below so that that `ResidualSequence` does not just implement `y = f(x)` but instead implements `y = f(x) + x`. The template code has a bug and only implement `y=x`. (1 point)

```
##########################################################################
##########
# TODO: Correct the below code to get f(x) + x as output
##########################################################################
##########
class ResidualSequence(Sequential):
    def forward(self, x):
        side_result = super().forward(x)
        final_result = side_result + x
        return final_result
```

```
##########################################################################
##########
#                                END OF YOUR CODE
#
##########################################################################
##########
```

Example of Residual Block Architecture

```
torch.nn.Sequential(
    nn.Linear(train_data.size(1), hidden_dims),
    ResidualSequence(
        nn.ReLU(),
        nn.Linear(hidden_dims, fan_out_dims),
        nn.ReLU(),
        nn.Linear(fan_out_dims, hidden_dims),
    ),
    nn.Linear(hidden_dims, 1),
    nn.Sigmoid()
)
```

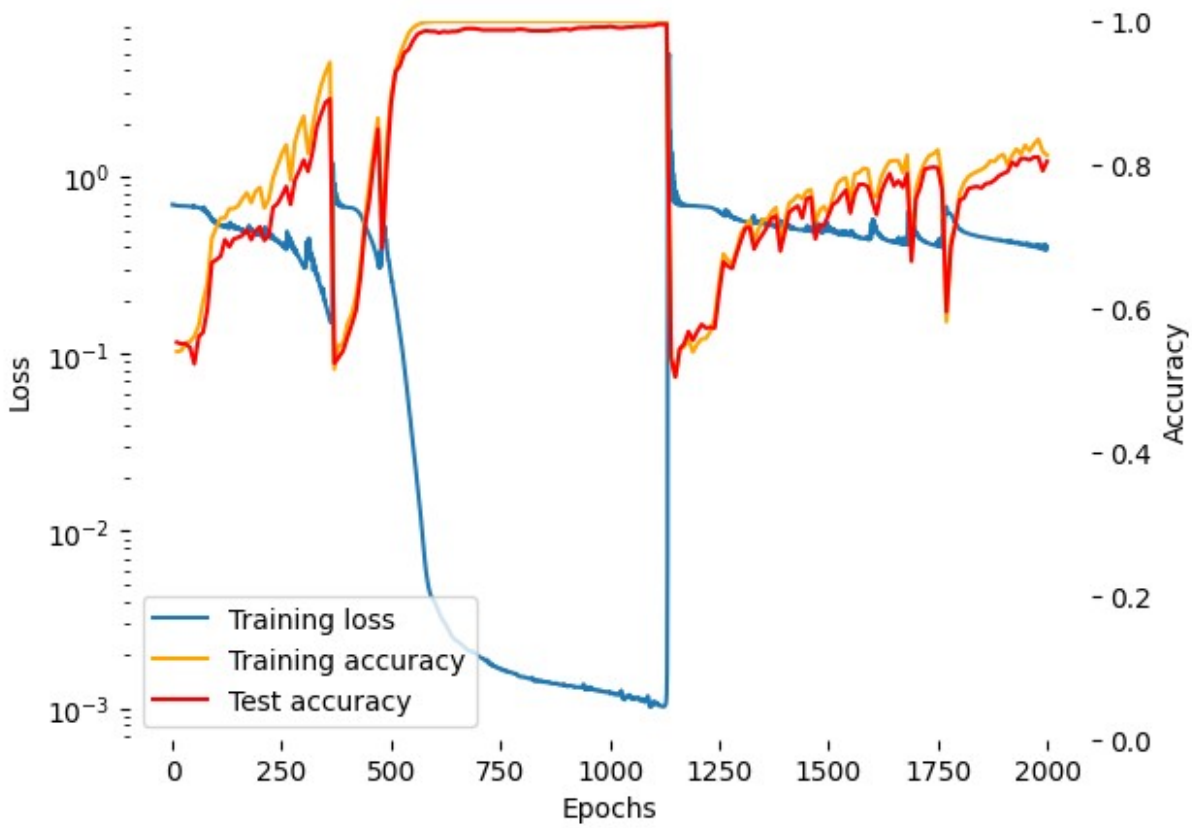
Task 3.16 - Implement Residual blocks in a Neural Network architecture (1 point)

Design a neural network with four Residual blocks, each composed according to the specifications outlined below.

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
##########################################################################
#######
# TODO: In this task, we will configure a neural network consisting of
four
# residual blocks
# Hidden Dimension = 16
# fan_out_dims = 32
# Loss - Binary Cross Entropy
# Optimizer - Adam- (lr=0.01, weight_decay=1e-3)
# Residual Block - [ReLU → Linear → ReLU → Linear]
# Network Architecture - Linear -> Residual Block -> Residual Block ->
# Residual Block -> Residual Block -> (Linear + Sigmoid)
##########################################################################
#######

input_size = train_data.size(1)
hidden_dims = 16
fan_out_dims = 32
output_dims = 1

run_test(
```

```
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            ResidualSequence(
                nn.ReLU(),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.ReLU(),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.ReLU(),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.ReLU(),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.16-ResBlocks4-HidDim16",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 4,
        'weight_decay': 1e-3
    }
)

######################################################################
#####
#                               END OF YOUR CODE
#
```

```
##########################################################################
#####

4897 parameters
Epoch 2000, Loss: 0.40298, Grad range 1.4e+00 to 4.6e-02, Train
Accuracy: 0.8136249780654907, Test Accuracy: 0.8050000071525574
Train Accuracy: 0.81362, Test Accuracy: 0.80500
```



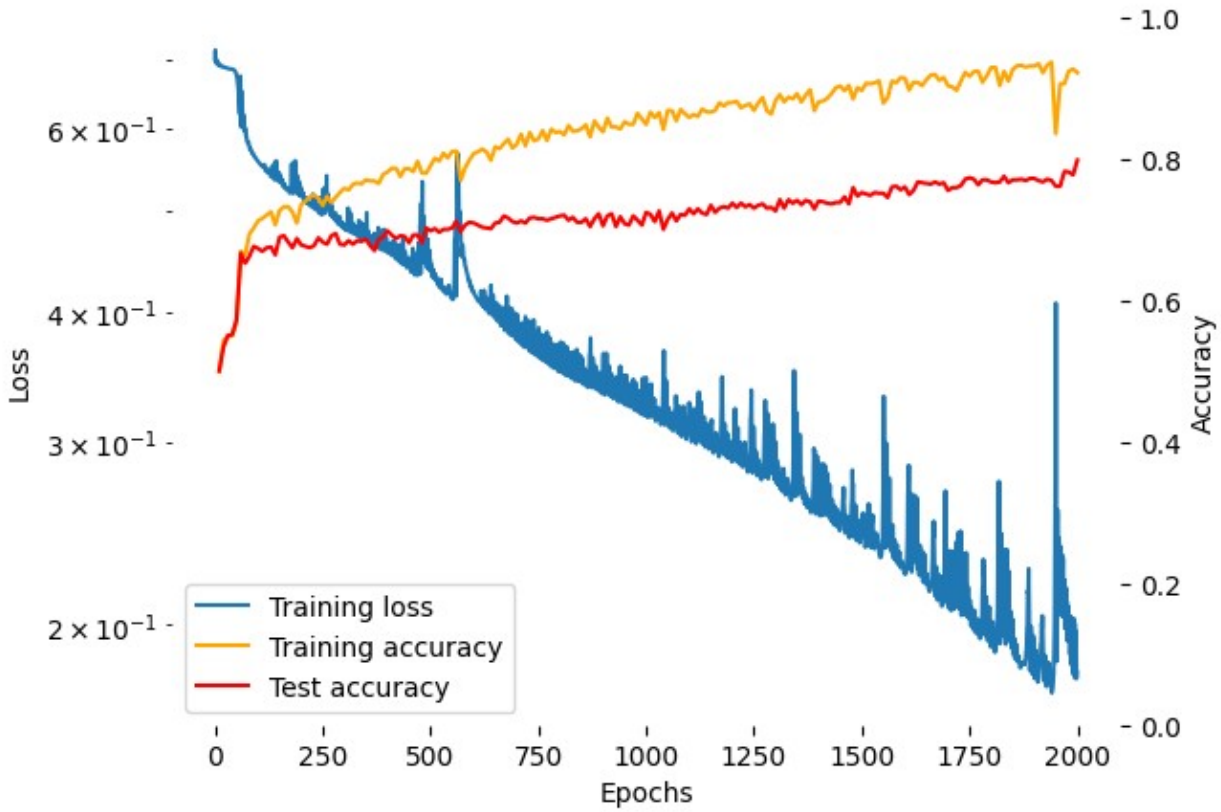
Task 3.17 - Reducing Residual Blocks (1 point)

Let's decrease the number of Residual Blocks and observe whether it has any impact on our performance.

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
hidden_dims = 16
fan_out_dims = 32
##########################################################################
##########
# TODO: In this task, we will configure a neural network consisting of
two
# residual blocks
# Hidden Dimension = 16
# fan_out_dims = 32
```

```
# Loss - Binary Cross Entropy
# Optimizer - Adam - (lr=0.01, weight_decay=1e-3)
# Residual Block - [ReLU → Linear → ReLU → Linear]
# Network Architecture - Linear -> Residual Block -> Residual Block ->
# (Linear + Sigmoid)
################################################################################
##########

input_size = train_data.size(1)
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            ResidualSequence(
                nn.ReLU(),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.ReLU(),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.17-ResBlocks2-HidDim16",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 2,
        'weight_decay': 1e-3
    }
)

################################################################################
#####
#                            END OF YOUR CODE
#
```

```
##########################################################################
#####

2753 parameters
Epoch 2000, Loss: 0.18011, Grad range 3.2e-01 to 1.9e-02, Train
Accuracy: 0.9226250052452087, Test Accuracy: 0.7990000247955322
Train Accuracy: 0.92263, Test Accuracy: 0.79900
```



Task 3.18 - Develop a Neural Network with a combination of BatchNorm and Residual Blocks (2 points)

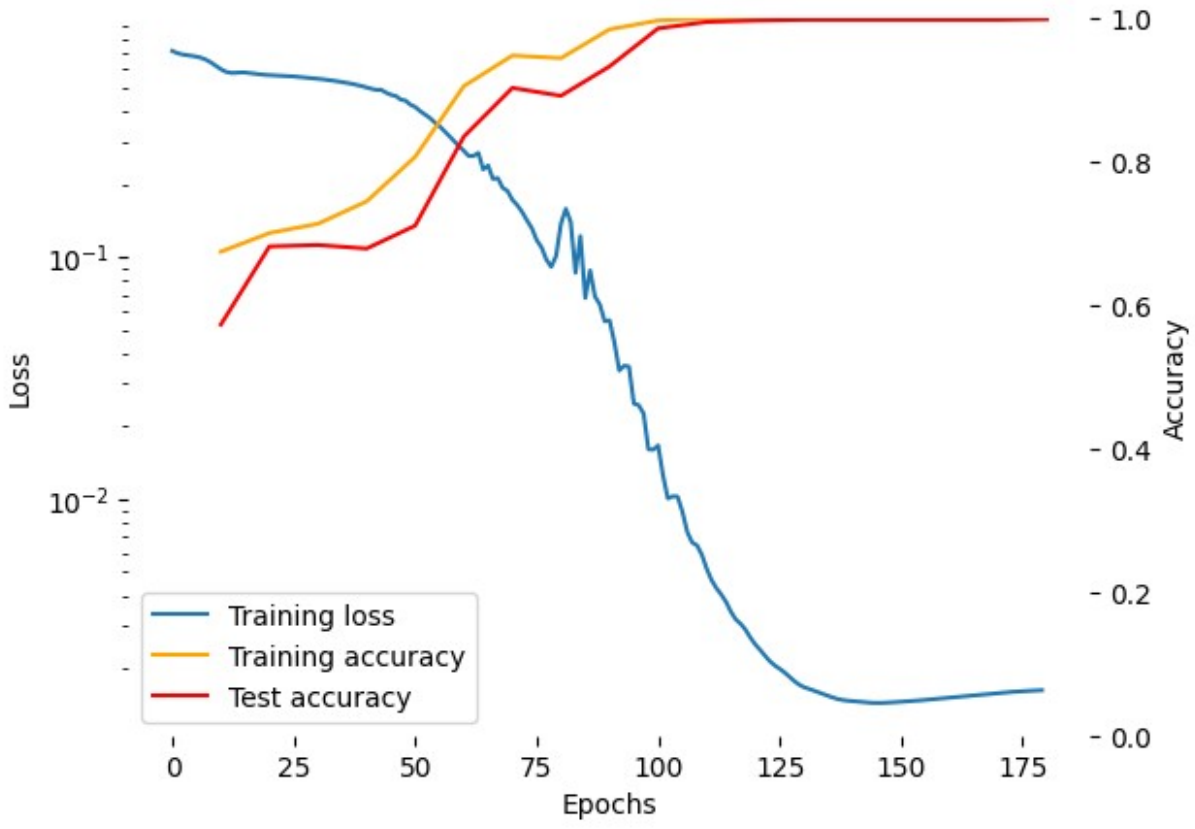
The synergy of these two components often results in improved model performance because BatchNorm stabilizes activations and enables the use of deeper networks, while Residual connections facilitate the training of deep networks and prevent degradation in performance. Together, they can enhance the model's ability to learn intricate patterns and improve its generalization to unseen data. However, it's important to strike a balance and avoid overly complex models, as they may lead to overfitting if not properly regularized.

We will create a neural network incorporating both Batch Normalization and Residual Blocks to evaluate if we can achieve favorable outcomes.

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
```

```
#########################################################################
#######
# TODO: In this task, we will configure a neural network consisting of
two
# residual blocks,
# Hidden Dimension - 16
# fan_out_dims = 32
# Loss - Binary Cross Entropy
# Optimizer - Adam - (lr = 0.01, weight_decay=1e-3)
# Residual Block - [Batch_Norm -> Linear → ReLU → Linear]
# Network Architecture - Linear -> Residual Block -> Residual Block ->
# (Batch_Norm + Linear + Sigmoid)
#########################################################################
#######

input_size = train_data.size(1)
hidden_dims = 16
fan_out_dims = 32
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            ResidualSequence(
                nn.BatchNorm1d(hidden_dims),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.BatchNorm1d(hidden_dims),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            nn.BatchNorm1d(hidden_dims),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.18-ResBlocks2-BatchNorm-HidDim16",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
```

```
        'hidden_dim': hidden_dims,
        'depth': 2,
        'weight_decay': 1e-3
    }
)

##############################################################################
#####
#                               END OF YOUR CODE
#
##############################################################################
#####

2849 parameters
Epoch 180, Loss: 0.00162, Grad range 1.2e-03 to 1.2e-08, Train
Accuracy: 1.0, Test Accuracy: 0.9990000128746033
Train Accuracy: 1.00000, Test Accuracy: 0.99900
```

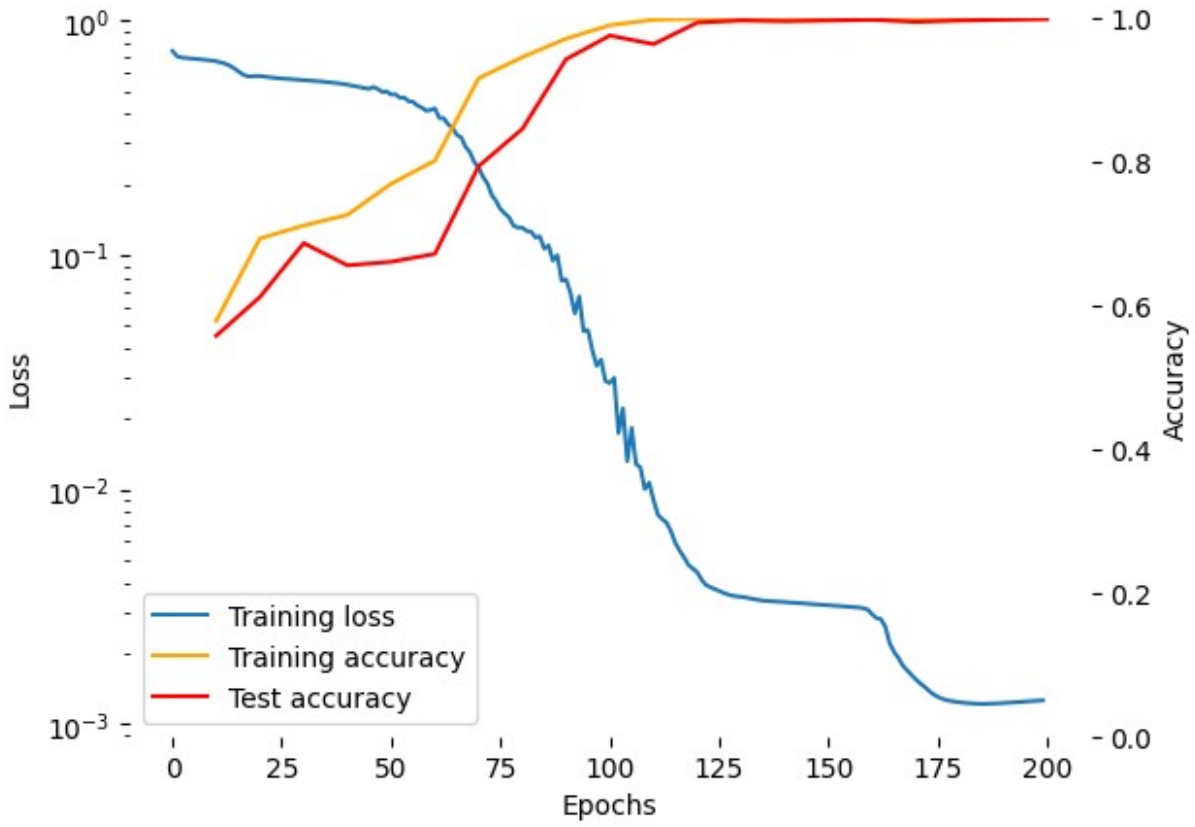


Task 3.19 - Develop a neural network that incorporates Batch Normalization, Residual connections, and an increased number of layers (2 points)

We will construct a neural network with 6 residual blocks to assess whether we can further enhance performance by increasing its depth.

```
torch.manual_seed(7150)# (Leave it here for deterministic behavior and
easier grading)
##########################################################################
##########
# TODO: In this task, we will configure a neural network consisting of
two
# residual blocks.
# Hidden Dimension - 16
# fan_out_dims = 32
# Loss - Binary Cross Entropy
# Optimizer - Adam - (lr = 0.01, weight_decay=1e-3)
# Residual Block - [Batch_Norm -> Linear → ReLU → Linear]
# Network Architecture - Linear -> Residual Block -> Residual Block ->
# Residual Block -> Residual Block -> Residual Block -> Residual Block
# -> (Batch_Norm + Linear + Sigmoid)
##########################################################################
##########

input_size = train_data.size(1)
hidden_dims = 16
fan_out_dims = 32
output_dims = 1

run_test(
    Supervise(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            ResidualSequence(
                nn.BatchNorm1d(hidden_dims),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.BatchNorm1d(hidden_dims),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.BatchNorm1d(hidden_dims),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.BatchNorm1d(hidden_dims),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
```

```
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.BatchNorm1d(hidden_dims),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            ResidualSequence(
                nn.BatchNorm1d(hidden_dims),
                nn.Linear(hidden_dims, fan_out_dims),
                nn.ReLU(),
                nn.Linear(fan_out_dims, hidden_dims),
            ),
            nn.BatchNorm1d(hidden_dims),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task3.19-ResBlocks6-BatchNorm-HidDim16",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 6,
        'weight_decay': 1e-3
    }
)

##########################################################################
##########
#                               END OF YOUR CODE
#
##########################################################################
##########

7265 parameters
Epoch 200, Loss: 0.00126, Grad range 1.4e-03 to 2.5e-09, Train
Accuracy: 1.0, Test Accuracy: 0.9990000128746033
Train Accuracy: 1.00000, Test Accuracy: 0.99900
```



Part 4: Weight Initialization

Weight initialization in deep learning refers to the procedure of assigning initial values to a neural network's weights, which are the tunable parameters learned during training. The manner in which these weights are initialized plays a critical role in shaping the training process and ultimately impacts the network's performance. Several weight initialization techniques have been developed to combat challenges such as vanishing or exploding gradients, with the goal of establishing a solid foundation for training deep neural networks. Some of the weight initialization methods are :-

1. Zero Weight Initialization
2. Random Weight Initialization
3. Xavier Initialization (Glorot Initialization)
4. He Initialization (often used in deep CNN's)

1) Zero Initialization

Zero weight initialization initializes a neural network's weights to zero, which can be effective for specific neural network types. This initialization can mitigate overfitting by making it harder for the model to fit training data perfectly.

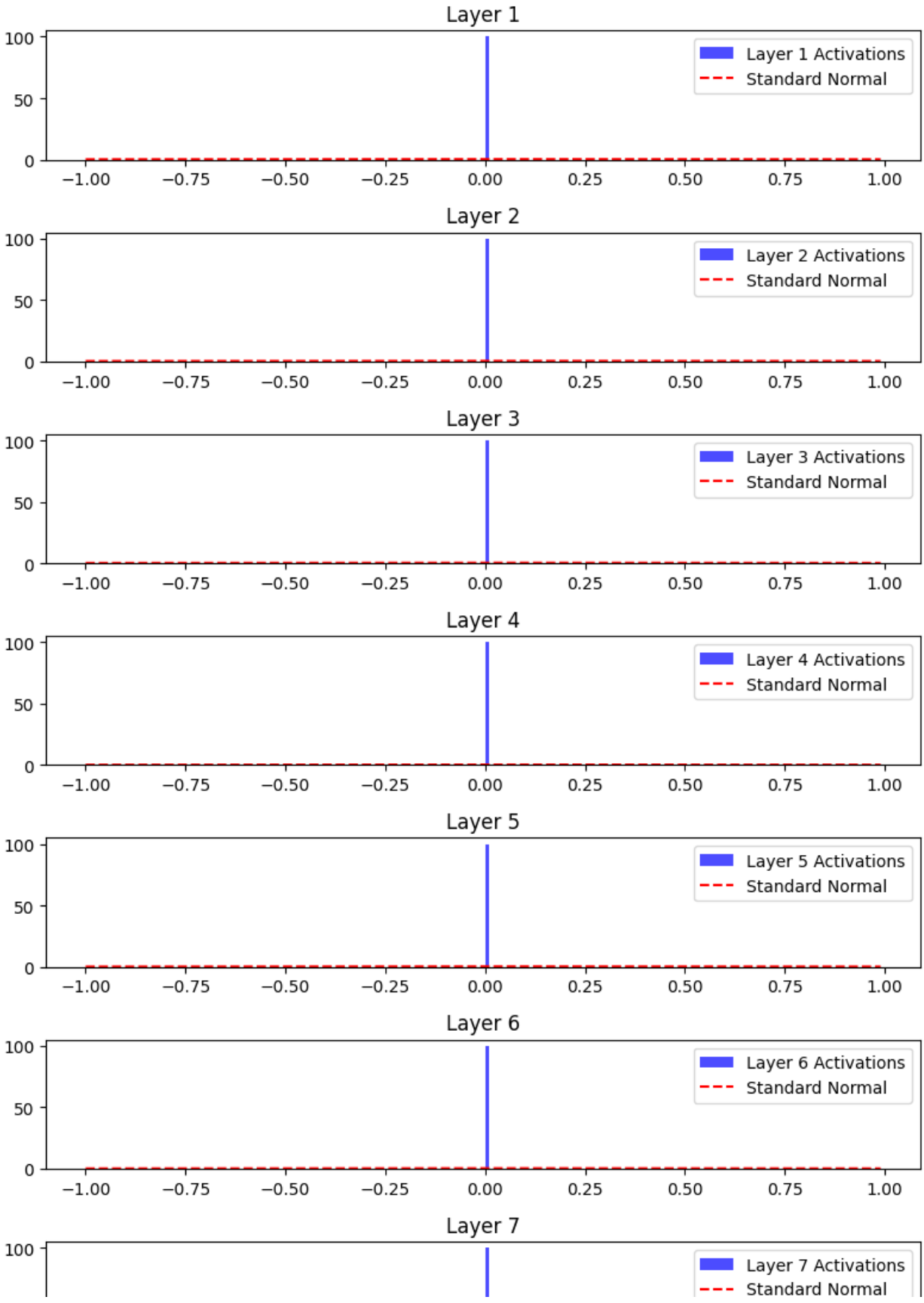
However, it has drawbacks. It hinders learning complex input-output relationships from scratch and makes the model sensitive to hyperparameters like the learning rate. This approach is generally discouraged due to the potential emergence of symmetric neurons and slow convergence as a result of weight symmetry issues.

The code visualizes layer-wise activation distributions in a neural network for a Zero Weight Initialization. It offers insights into the network's learning and can reveal potential issues.

Task 4.1 - Initialize zero weights for each layer (_ points)

```
num_layers = 7
layer_dims = [2048]*num_layers

input_data = np.random.randn(16, layer_dims[0])

activations = []

weights = []


for i in range(num_layers - 1):


##########################################################################
######
    # TODO - # Initialize zero weights for each layer (except the last
one)
    # and store it in 'W' varaiable

##########################################################################
######
    W = np.zeros((layer_dims[i], layer_dims[i + 1]))

##########################################################################
######
    #                               END OF YOUR CODE
#

##########################################################################
######
    weights.append(W)

# Forward pass through the network
for i in range(num_layers):
    W = weights[i] if i < num_layers - 1 else None
    layer_input = input_data if i == 0 else activations[i - 1]
    if W is not None:
        layer_output = np.tanh(np.dot(layer_input, W))
    else:
        layer_output = layer_input
    activations.append(layer_output)

# Create a figure with subplots for each layer's activation
distribution
fig, axes = plt.subplots(num_layers, 1, figsize=(8, 12))
```

```
x_axis = np.arange(-1, 1, 0.01)

for i in range(num_layers):
    ax = axes[i]
    ax.hist(activations[i].flatten(), bins=100, density=True,
alpha=0.7, color='blue', label=f'Layer {i+1} Activations')
    ax.plot(x_axis, norm.pdf(x_axis, 0, 1), color='red',
linestyle='--', label='Standard Normal')
    ax.set_title(f'Layer {i+1}')
    ax.legend()

plt.tight_layout()
plt.show()
```


Layer 1
Layer 1 Activations
Standard Normal
Layer 2
Layer 2 Activations
Standard Normal
Layer 3
Layer 3 Activations
Standard Normal
Layer 4
Layer 4 Activations
Standard Normal
Layer 5
Layer 5 Activations
Standard Normal
Layer 6
Layer 6 Activations
Standard Normal
Layer 7
Layer 7 Activations
Standard Normal
100
50
0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

4.A) Inline Question (2 points): Have you noticed any of the drawbacks in the results mentioned earlier? If so, please highlight your observations.

With zero weight initialization, every layer's activation distribution collapses to a single spike at exactly zero, as seen in the histograms. This demonstrates several drawbacks:

1. **Symmetry Problem**: When all weights are initialized to zero, every neuron in a given layer computes the exact same output (zero), and receives the exact same gradient during backpropagation. This means all neurons update identically and can never learn different features. The network effectively behaves as if it has only one neuron per layer, regardless of width.

2. **No Learning / Dead Network**: Since $\tanh(0) = 0$ for every neuron at every layer, the network produces a constant zero output irrespective of the input. The activations carry no information about the input data whatsoever. Comparing the blue spike at zero against the red dashed standard normal reference curve makes it obvious that zero-initialized activations bear no resemblance to a healthy activation distribution.

3. **Vanishing Gradients**: Although $\tanh'(0) = 1$ (the maximum possible gradient for tanh), the symmetry ensures that all gradient updates are identical across neurons. Combined with the zero activations flowing forward, the network has no meaningful signal to learn from, and weight updates, even if nonzero, will preserve the symmetry. This would keep the network stuck.

2. Random Weight Initialization

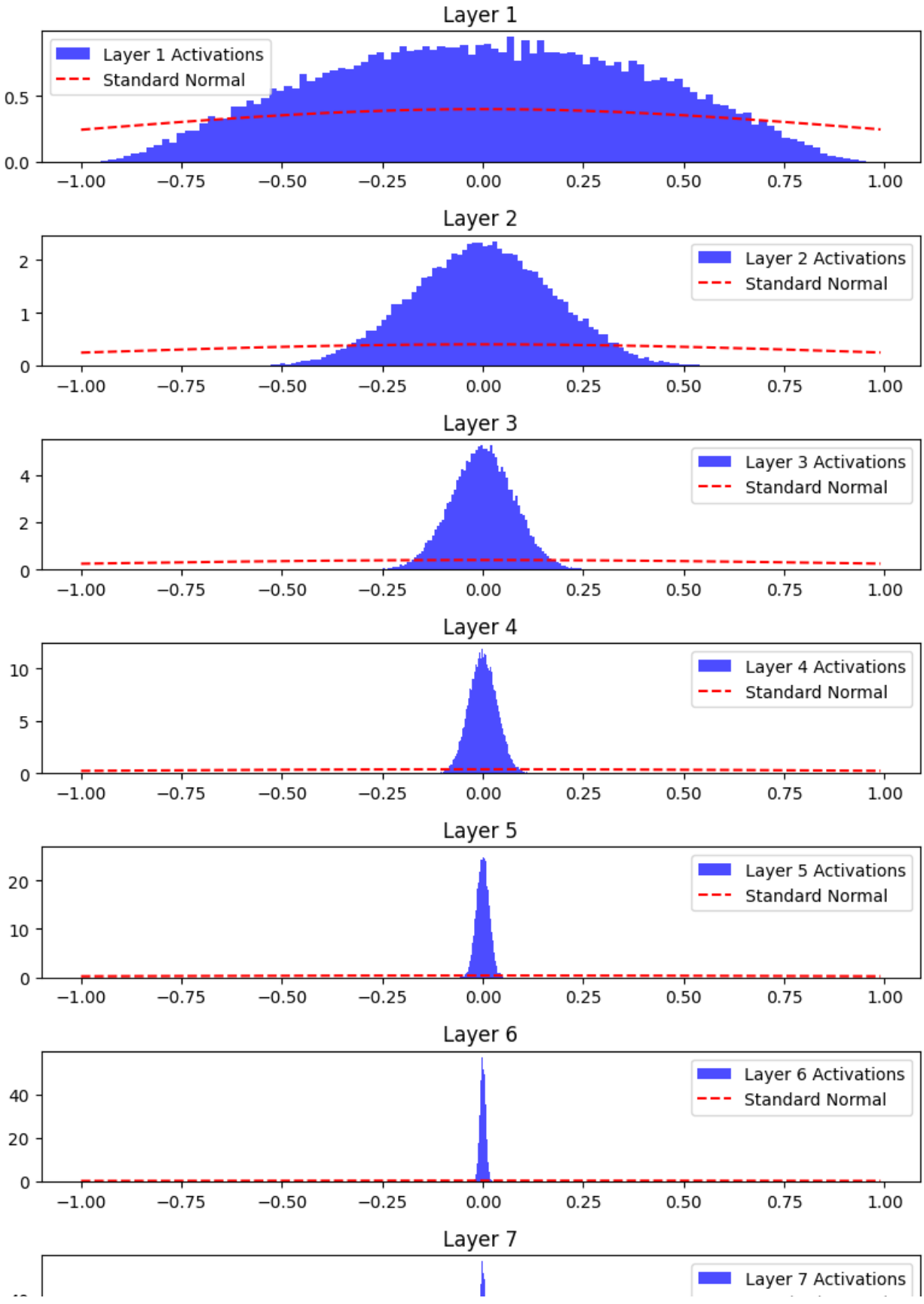
Random Weight Initialization in deep learning sets the initial weights of a neural network to random values, drawn from a distribution. It's simple and encourages diverse starting points for training, breaking symmetry among neurons. However, it can lead to vanishing/exploding gradients in deep networks, making it less effective for them. It's sensitive to initialization values and lacks control compared to specialized methods like Xavier or He initialization.

Task 4.2 Initialize Random weights for each layer (_ points)

The below code visualizes layer-wise activation distributions for a Random Weight Initialization in a neural network. It offers insights into the network's learning and can reveal potential issues.

```
# Define the number of layers and their dimensions
num_layers = 7
layer_dims = [2048]*num_layers

input_data = np.random.randn(16, layer_dims[0])

activations = []

weights = []


for i in range(num_layers - 1):
```

```
##########################################################################
######
    # TODO - # Initialize random weights for each layer (except the
last one)
    # and the store value in 'W' variable

##########################################################################
######
    W = np.random.randn(layer_dims[i], layer_dims[i+1])

##########################################################################
######
    #                                END OF YOUR CODE
#

##########################################################################
######
    weights.append(W*0.01)

# Forward pass through the network
for i in range(num_layers):
    W = weights[i] if i < num_layers - 1 else None
    layer_input = input_data if i == 0 else activations[i - 1]
    if W is not None:
        layer_output = np.tanh(np.dot(layer_input, W))
    else:
        layer_output = layer_input
    activations.append(layer_output)

# Create a figure with subplots for each layer's activation
distribution
fig, axes = plt.subplots(num_layers, 1, figsize=(8, 12))

x_axis = np.arange(-1, 1, 0.01)

for i in range(num_layers):
    ax = axes[i]
    ax.hist(activations[i].flatten(), bins=100, density=True,
alpha=0.7, color='blue', label=f'Layer {i+1} Activations')
    ax.plot(x_axis, norm.pdf(x_axis, 0, 1), color='red',
linestyle='--', label='Standard Normal')
    ax.set_title(f'Layer {i+1}')
    ax.legend()

plt.tight_layout()
plt.show()
```


Layer 1
Layer 1 Activations
Standard Normal
Layer 2
Layer 2 Activations
Standard Normal
Layer 3
Layer 3 Activations
Standard Normal
Layer 4
Layer 4 Activations
Standard Normal
Layer 5
Layer 5 Activations
Standard Normal
Layer 6
Layer 6 Activations
Standard Normal
Layer 7
Layer 7 Activations
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

4.B) Inline Question (2 points): We can see all the activations tend to zero for deeper network layers. what can be expected regarding the gradients dL/dW, and is there still potential for learning?

From the histograms, we can clearly observe a progressive collapse of activations toward zero as depth increases. Layer 1 has a broad, roughly uniform distribution spanning $[-1, 1)$, but by Layers 6 and 7, the distribution has degenerated into an extremely narrow spike centered at zero (the y-axis density jumps from ~0.4 in Layer 1 to ~50 in Layer 7).

What happens to the gradients $\frac{\partial L}{\partial W}$:

The gradient of the loss with respect to the weights at layer l is:

$$\frac{\partial L}{\partial W^{[l]}} = \frac{\partial L}{\partial z^{[l]}} \cdot \left(a^{[l-1]}\right)^T$$

Since the activations $a^{[l-1]} \approx 0$ in deeper layers (as shown by the histograms), $\frac{\partial L}{\partial W^{[l]}} \approx 0$ as well, regardless of the upstream gradient. Furthermore, during backpropagation the gradient signal passes through each layer's weight matrix:

$$\frac{\partial L}{\partial a^{[l-1]}} = \left(W^{[l]}\right)^T \cdot \frac{\partial L}{\partial z^{[l]}}$$

Because the weights are scaled by 0.01 (super small), the gradient is attenuated at every layer, causing the vanishing effect to compound as you go deeper into the network.

Note that $\tanh'(z) = 1 - \tanh^2(z)$, and since $\tanh(z) \approx 0$ in deeper layers, $\tanh'(z) \approx 1$, so the activation function itself is *not* the bottleneck here. The problem is entirely due to the tiny weight magnitudes shrinking both the forward signal and the backward gradient signal at every layer.

Is there still potential for learning?

Technically yes, but probably not. Unlike zero initialization, symmetry *is* broken here (each neuron has distinct random weights), so neurons can theoretically learn different features. However, the gradients are so vanishingly small in the deeper layers that weight updates will be negligible, and training would be extraordinarily slow or effectively stalled. The early layers may still receive some gradient signal, but the deeper layers will barely learn at all. So, this experiment shows us the **vanishing gradient problem** in action.

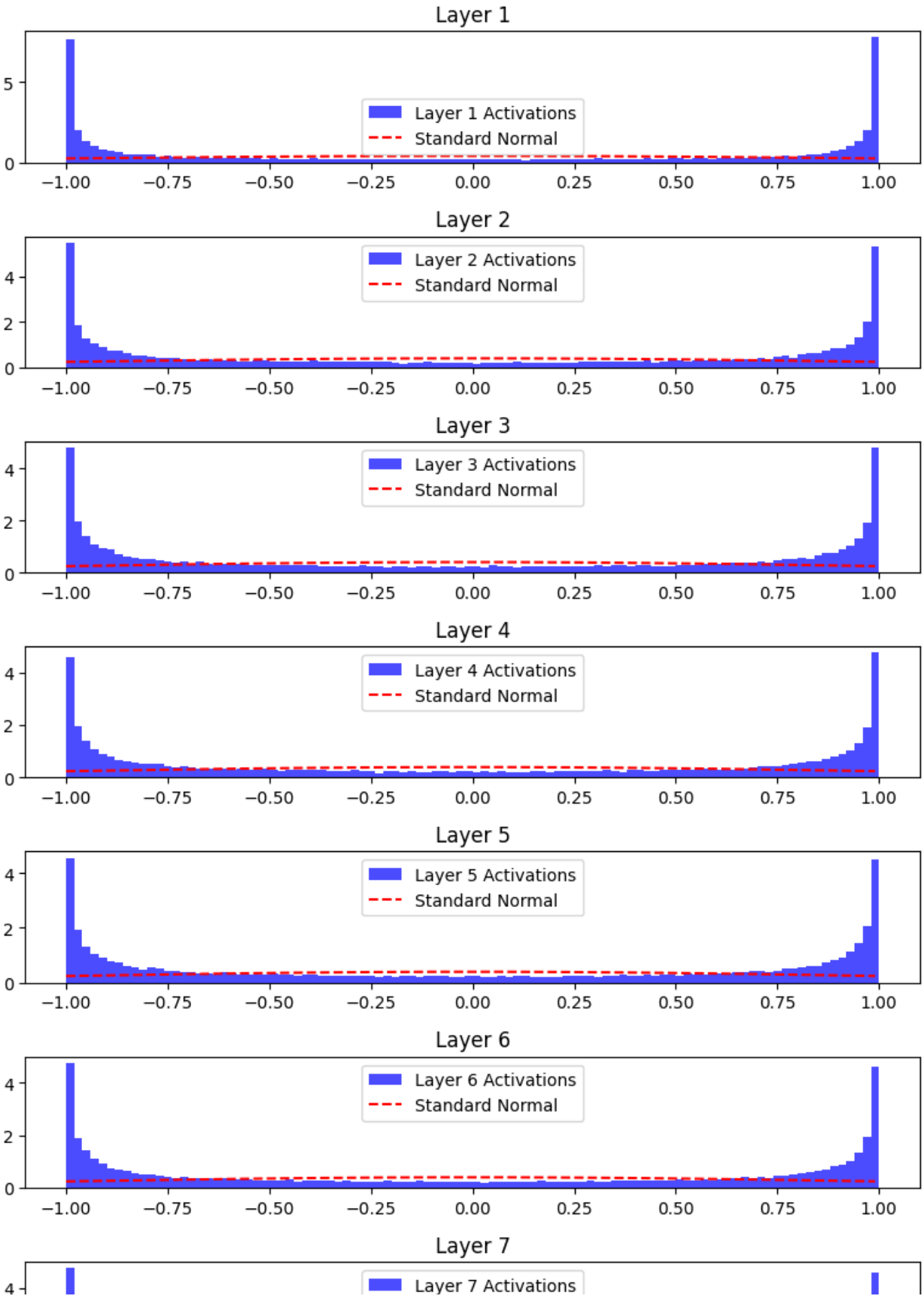
I can hear you all say try increasing weights which might resolve the issue. So, let's proceed by multiplying our weights by a factor of 5 and see if it helps.

Task 4.3 - Initialize large Random weights for each layer (_ points)

```
num_layers = 7
layer_dims = [2048]*num_layers

input_data = np.random.randn(16, layer_dims[0])
```

```
activations = []

weights = []


for i in range(num_layers - 1):


################################################################################
######
    # TODO - # Initialize random weights for each layer similar to the
last
    # task but here you scale them up by a factor of 5

################################################################################
######
    W = np.random.randn(layer_dims[i], layer_dims[i+1]) * 5

################################################################################
######
    #                                END OF YOUR CODE
#

################################################################################
######

    weights.append(W*0.01)

# Forward pass through the network
for i in range(num_layers):
    W = weights[i] if i < num_layers - 1 else None
    layer_input = input_data if i == 0 else activations[i - 1]
    if W is not None:
        layer_output = np.tanh(np.dot(layer_input, W))
    else:
        layer_output = layer_input
    activations.append(layer_output)

# Create a figure with subplots for each layer's activation
distribution
fig, axes = plt.subplots(num_layers, 1, figsize=(8, 12))

x_axis = np.arange(-1, 1, 0.01)

for i in range(num_layers):
    ax = axes[i]
    ax.hist(activations[i].flatten(), bins=100, density=True,
alpha=0.7, color='blue', label=f'Layer {i+1} Activations')
    ax.plot(x_axis, norm.pdf(x_axis, 0, 1), color='red',
```

```
linestyle='--', label='Standard Normal')
    ax.set_title(f'Layer {i+1}')
    ax.legend()

plt.tight_layout()
plt.show()
```


Layer 1
Layer 1 Activations
Standard Normal
Layer 2
Layer 2 Activations
Standard Normal
Layer 3
Layer 3 Activations
Standard Normal
Layer 4
Layer 4 Activations
Standard Normal
Layer 5
Layer 5 Activations
Standard Normal
Layer 6
Layer 6 Activations
Standard Normal
Layer 7
Layer 7 Activations
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

4.C) Inline Question (2 points): All activations saturate due to big weights. What can be expected regarding the gradients dL/dW, and is there still potential for learning?

From the histograms, we observe that activations across all layers are saturated at the extreme values of $\tanh$, -1 and +1. Even Layer 1 already shows tall spikes at ± 1 with almost no mass in the interior, and this bimodal saturation pattern persists uniformly through all 7 layers.

What happens to the gradients $\frac{\partial L}{\partial W}$:

The derivative of $\tanh$ is:

$$\tanh'(z) = 1 - \tanh^2(z)$$

When activations saturate at $\tanh(z) \approx \pm 1$, the local gradient becomes:

$$\tanh'(z) \approx 1 - (\pm 1)^2 = 1 - 1 = 0$$

During backpropagation, the gradient at each layer involves this local derivative:

$$\frac{\partial L}{\partial z^{[l]}} = \frac{\partial L}{\partial a^{[l]}} \cdot \tanh'\left(z^{[l]}\right) \approx \frac{\partial L}{\partial a^{[l]}} \cdot 0 = 0$$

Since this near-zero factor appears at every layer, the gradient signal is killed before it can propagate backward. Thus:

$$\frac{\partial L}{\partial W^{[l]}} = \frac{\partial L}{\partial z^{[l]}} \cdot \left(a^{[l-1]}\right)^T \approx 0$$

Even though $a^{[l-1]}$ is large in magnitude (saturated at ± 1), it does not help because the upstream gradient $\frac{\partial L}{\partial z^{[l]}}$ is already effectively zero due to the saturated $\tanh'$ term.

Is there still potential for learning?

Nope. This is the **vanishing gradient problem** again, but caused by saturation. Unlike Task 4.2 where the forward signal was too weak, here the forward signal is too strong: the large weights push every neuron deep into the saturated regime of $\tanh$, where the gradient is essentially flat. Weight updates will be near zero across all layers, so the network can't learn. This is the opposite extreme of Task 4.2, but leads to the same outcome: the gradients $\frac{\partial L}{\partial W}$ vanish and training stalls. This shows us that both too-small and too-large weight initialization are harmful.

3. Xavier Initialization

Xavier Initialization, also called Glorot Initialization, is a weight initialization method for deep neural networks. It sets initial weights to prevent vanishing and exploding gradients by controlling the variance of activations.This technique stabilizes training and is widely used in practice.

The weights are initialized from a Gaussian distribution with a mean of 0 and a variance of ($1/n_{in}$) :-

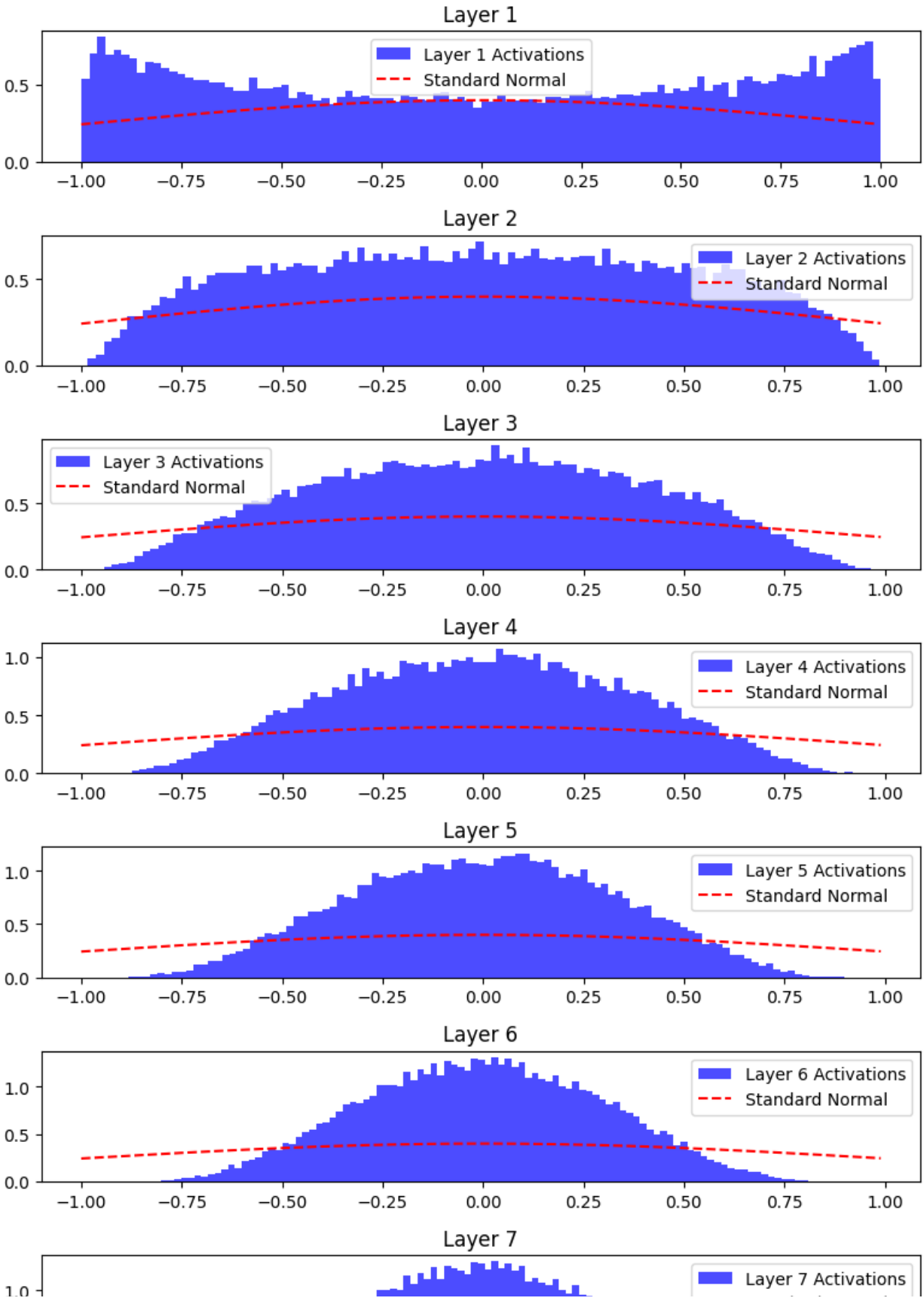
\begin{equation} \mathbf{W} = \mathcal{N}\left(0, \frac{1}{n_\text{in}}\right) \end{equation}

The below code visualizes layer-wise activation distributions for a Xavier Weight Initialization in a neural network. It offers insights into the network's learning and can reveal potential issues.

Task 4.4 - Initialize weights using Xavier method for each layer (1 point)

```
num_layers = 7
layer_dims = [2048]*num_layers

input_data = np.random.randn(16, layer_dims[0])
activations = []
weights = []


for i in range(num_layers - 1):


################################################################################
    # TODO - Initialize weights using Xavier method for each layer (except the
    # last one) and store in variable 'W'

################################################################################
    W= np.random.randn(layer_dims[i], layer_dims[i+1]) * np.sqrt(1.0 / layer_dims[i])

################################################################################
    #                              END OF YOUR CODE
#

################################################################################
    weights.append(W)

# Forward pass through the network
for i in range(num_layers):
    W = weights[i] if i < num_layers - 1 else None
    layer_input = input_data if i == 0 else activations[i - 1]
    if W is not None:
        layer_output = np.tanh(np.dot(layer_input, W))
    else:
        layer_output = layer_input
    activations.append(layer_output)

# Create a figure with subplots for each layer's activation distribution
fig, axes = plt.subplots(num_layers, 1, figsize=(8, 12))
```

```
x_axis = np.arange(-1, 1, 0.01)

for i in range(num_layers):
    ax = axes[i]
    ax.hist(activations[i].flatten(), bins=100, density=True,
alpha=0.7, color='blue', label=f'Layer {i+1} Activations')
    ax.plot(x_axis, norm.pdf(x_axis, 0, 1), color='red',
linestyle='--', label='Standard Normal')
    ax.set_title(f'Layer {i+1}')
    ax.legend()

plt.tight_layout()
plt.show()
```


Layer 1
Layer 1 Activations
Standard Normal
Layer 2
Layer 2 Activations
Standard Normal
Layer 3
Layer 3 Activations
Standard Normal
Layer 4
Layer 4 Activations
Standard Normal
Layer 5
Layer 5 Activations
Standard Normal
Layer 6
Layer 6 Activations
Standard Normal
Layer 7
Layer 7 Activations
0.0
0.5
1.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

4. He/ MSRA Initialization

It is a weight initialization technique commonly used in deep neural networks. It is designed to address the vanishing gradient problem and is particularly effective when Rectified Linear Unit (ReLU) activation functions are used.

For a layer with n_{in} input units, He Initialization initializes the weights by sampling them from a Gaussian distribution with a mean of 0 and a variance of 2 / n_{in}. The choice of variance (2) is specific to the ReLU activation function and ensures that the weights are set to values that allow ReLU units to activate in a desirable range.

The formula for He Initialization can be expressed as follows:

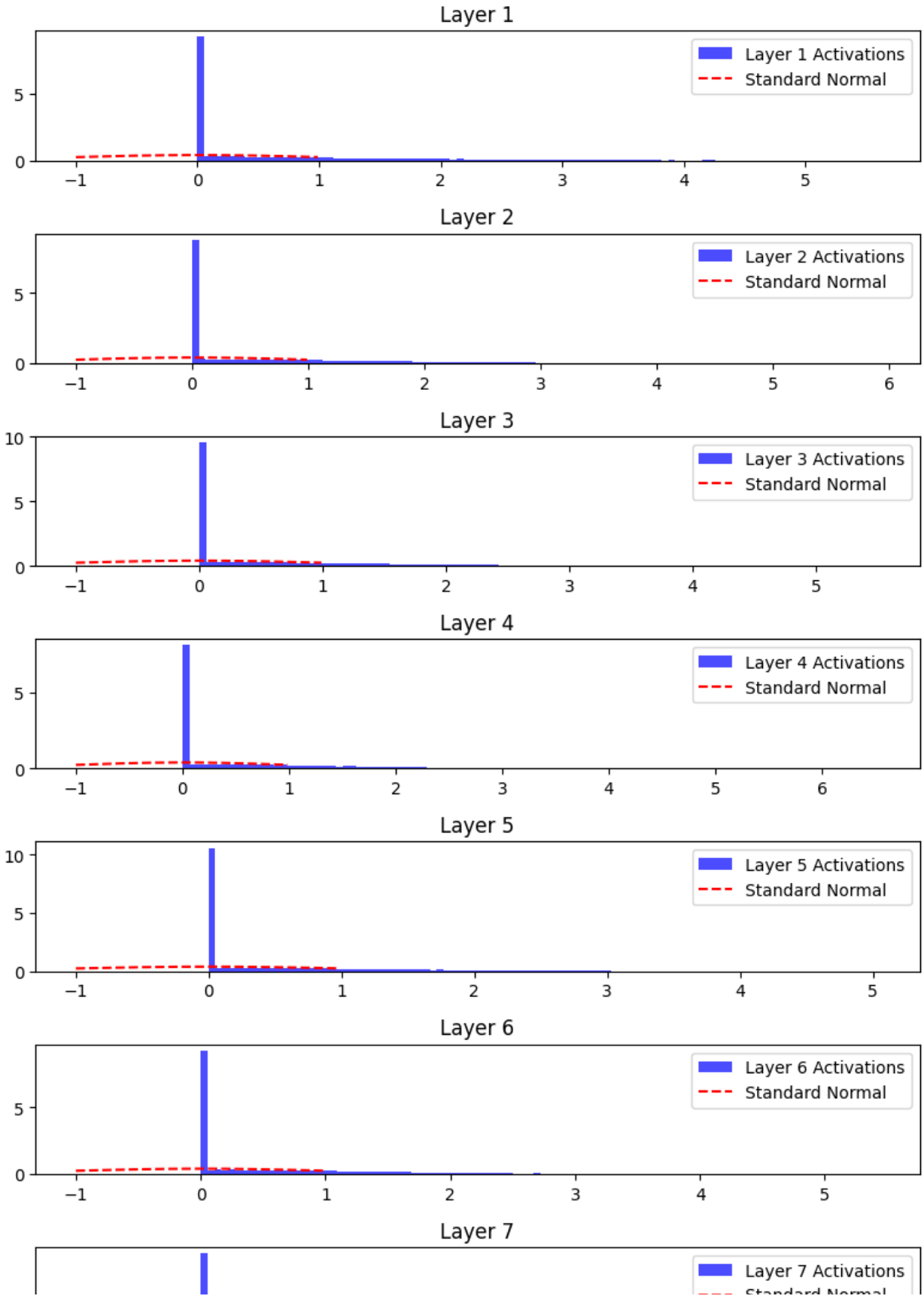
\begin{equation} \mathbf{W} = \mathcal{N}\left(0, \frac{2}{n_\text{in}}\right) \end{equation}

The below code visualizes layer-wise activation distributions for a HE/ MSRA Weight Initialization in a neural network. It offers insights into the network's learning and can reveal potential issues.

Task 4.5- Initialize weights using Kaiming He's method for each layer (1 point)

```
num_layers = 7
layer_dims = [2048]*num_layers

input_data = np.random.randn(16, layer_dims[0])

activations = []

weights = []

for i in range(num_layers - 1):


##########################################################################
######
    # TODO - Initialize weights using He method for each layer (except
the
    # last one) and store in variable 'W'

##########################################################################
######
    W = np.random.randn(layer_dims[i], layer_dims[i+1]) * np.sqrt(2.0
/ layer_dims[i])

##########################################################################
######
    #                               END OF YOUR CODE
#

##########################################################################
######
```

```
    weights.append(W)

# Forward pass through the network
for i in range(num_layers):
    W = weights[i] if i < num_layers - 1 else None
    layer_input = input_data if i == 0 else activations[i - 1]
    if W is not None:
        layer_output = np.maximum(0, layer_input.dot(W))
    else:
        layer_output = layer_input
    activations.append(layer_output)

# Create a figure with subplots for each layer's activation
distribution
fig, axes = plt.subplots(num_layers, 1, figsize=(8, 12))

x_axis = np.arange(-1, 1, 0.01)

for i in range(num_layers):
    ax = axes[i]
    ax.hist(activations[i].flatten(), bins=100, density=True,
alpha=0.7, color='blue', label=f'Layer {i+1} Activations')
    ax.plot(x_axis, norm.pdf(x_axis, 0, 1), color='red',
linestyle='--', label='Standard Normal')
    ax.set_title(f'Layer {i+1}')
    ax.legend()

plt.tight_layout()
plt.show()
```


Layer 1
Layer 1 Activations
Standard Normal
Layer 2
Layer 2 Activations
Standard Normal
Layer 3
Layer 3 Activations
Standard Normal
Layer 4
Layer 4 Activations
Standard Normal
Layer 5
Layer 5 Activations
Standard Normal
Layer 6
Layer 6 Activations
Standard Normal
Layer 7
Layer 7 Activations

5. Custom Weight Initialization in Pytorch

Custom weight initialization in PyTorch involves setting the weights of a neural network to user-defined values. This can serve different purposes, including enhancing model performance or preventing overfitting.

In PyTorch, you can achieve custom weight initialization using the init module, which offers various weight initialization techniques like normal_, uniform_, and kaiming_normal_.

Read Documentation for more information - https://pytorch.org/docs/stable/nn.init.html

An illustration of custom weight initialization for a neural network model using values sampled from a normal distribution.

```
class CustomModel(nn.Module):
    def __init__(self):
        super(CustomModel, self).__init__()
        self.fc1 = nn.Linear(64, 32)
        self.fc2 = nn.Linear(32, 10)

        self.initialize_weights()

    #Initialize weights with values from a normal distribution
    def initialize_weights(self):
        nn.init.normal_(self.fc1.weight, mean=0, std=0.01)
        nn.init.normal_(self.fc2.weight, mean=0, std=0.01)

    def forward(self, x):
        pass

model = CustomModel()

for name, param in model.named_parameters():
    if param.requires_grad:
        print(f'Parameter name: {name}')
        print(param.data)
```

Task 4.6 - Create a class called `Supervise_random_weights` that defines weight using Random Weight Initialization method. (1 point)

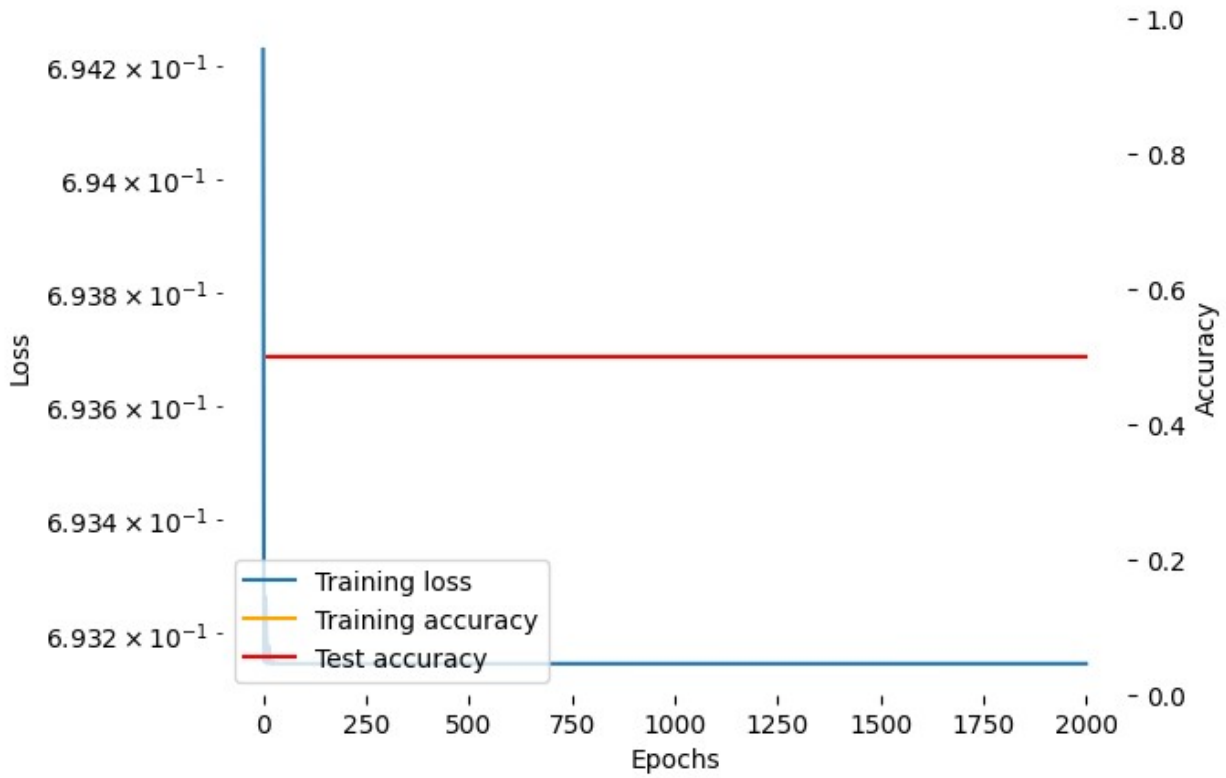
```
class Supervise_random_weights(nn.Module):
    def __init__(self, criterion, net):
        super().__init__()
        self.net = net
        self.criterion = criterion


        for module in self.net.modules():
            if isinstance(module, nn.Linear):
```

```
##############################################################
                # TODO: Set the weights to random values from a normal
                # distribution with a mean of 0 and a standard
deviation of 0.01

##############################################################
                nn.init.normal_(module.weight, mean=0.0, std=0.01)

##############################################################
                #                             END OF YOUR CODE
#

##############################################################

    def forward(self, x, y):
        out = self.net(x).squeeze()
        return self.criterion(out, y)
```

Task 4.7 - Build a Neural Network architecture using the Supervise_random_weights class (1 point)

```
torch.manual_seed(7150)
##############################################################################
##########
# TODO: In this task, we will configure a neural network with
Supervise_random_weights
# Hidden Dimension = 32
# Loss - Binary Cross Entropy
# Optimizer - Adam - (lr=0.01, weight_decay=1e-3)
# Network Architecture - (Linear + ReLU)-> (Linear + ReLU)->(Linear +
ReLU)->
# (Linear + ReLU)->(Linear + ReLU)->(Linear + Sigmoid)
##############################################################################
##########

input_size = train_data.size(1)
hidden_dims = 32
output_dims = 1

run_test(
    Supervise_random_weights(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
```

```
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task4.7-RandomWeights-ReLU_BCE_ADAM",
    hparam_dict={
        'activation': 'ReLU',
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 6,
        'weight_decay': 1e-3,
        'init': 'random_normal_0.01'
    }
)

##########################################################################
##########
#                               END OF YOUR CODE
#
##########################################################################
##########

5441 parameters
Epoch 2000, Loss: 0.69315, Grad range 4.7e-08 to 2.6e-30, Train
Accuracy: 0.5, Test Accuracy: 0.5
Train Accuracy: 0.50000, Test Accuracy: 0.50000
```



Task 4.8 - Create a class called `Supervise_Kaiming_weights` that defines weight using Kaiming He's Weight Initialization method. (1 point)

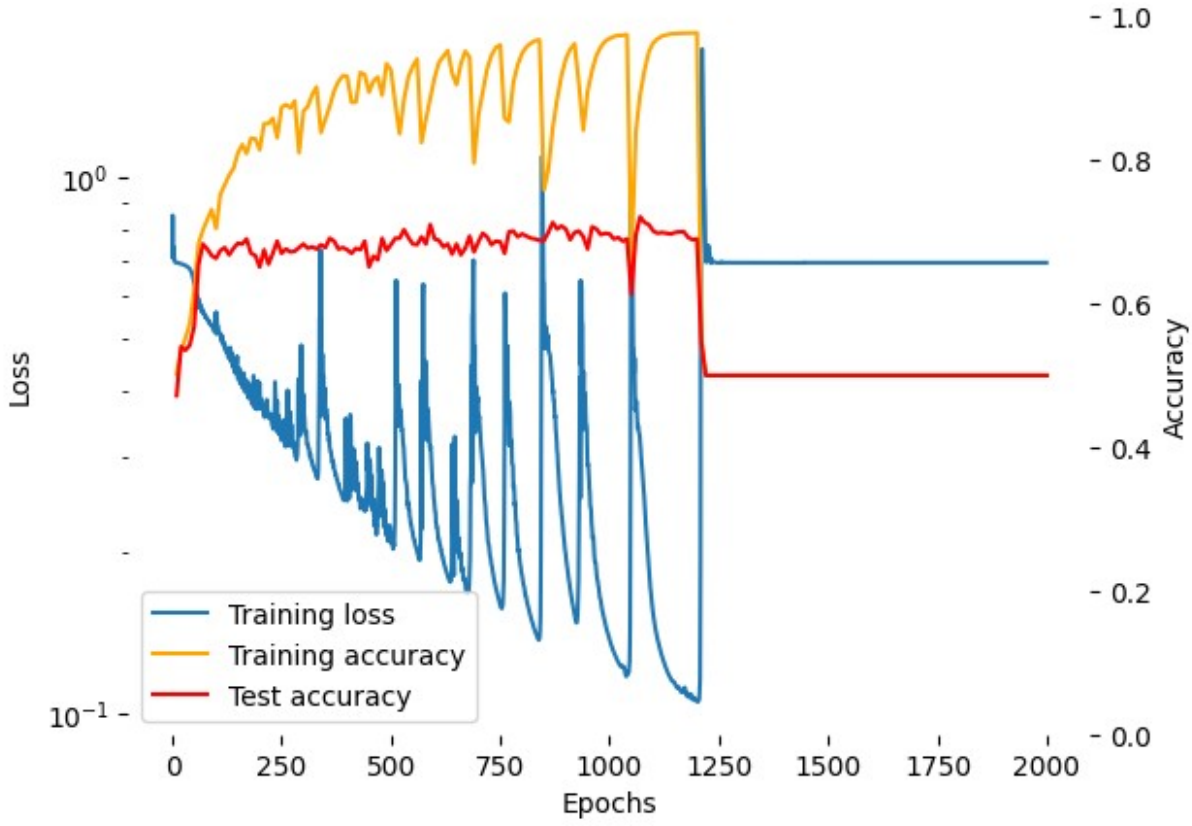
```
class Supervise_Kaiming_weights(nn.Module):
    def __init__(self, criterion, net):
        super().__init__()
        self.net = net
        self.criterion = criterion

        for module in self.net.modules():
            if isinstance(module, nn.Linear):

##################################################################
                # TODO: Set the weights using He initialization
                # [Hint: Python's torch.nn.init.kaiming would be
useful]

##################################################################
                nn.init.kaiming_normal_(module.weight,
nonlinearity='relu')

##################################################################
                #                          END OF YOUR CODE
#
```

```
##################################################################

    def forward(self, x, y):
        out = self.net(x).squeeze()
        return self.criterion(out, y)
```

Task 4.9 - Build a Neural Network architecture using the Supervise_Kaiming_weights class (1 point)

```
torch.manual_seed(7150)
##########################################################################
##########
# TODO: In this task, we will configure a neural network with
Supervise_He_weights
# Hidden Dimension = 32
# Loss - Binary Cross Entropy
# Optimizer - Adam - (lr=0.01, weight_decay=1e-3)
# Network Architecture - (Linear + ReLU)-> (Linear + ReLU)->(Linear +
ReLU)->
# (Linear + ReLU)->(Linear + ReLU)->(Linear + Sigmoid)
##########################################################################
##########

input_size = train_data.size(1)
hidden_dims = 32
output_dims = 1

run_test(
    Supervise_Kaiming_weights(
        nn.BCELoss(),
        nn.Sequential(
            nn.Linear(input_size, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, output_dims),
            nn.Sigmoid()
        )
    ),
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="Task4.9-KaimingWeights-ReLU_BCE_ADAM",
    hparam_dict={
        'activation': 'ReLU',
```

```
        'optimizer': 'Adam',
        'loss': 'BCE',
        'lr': 0.01,
        'hidden_dim': hidden_dims,
        'depth': 6,
        'weight_decay': 1e-3,
        'init': 'kaiming_he'
    }
)

################################################################################
######
#                               END OF YOUR CODE
#
################################################################################
######

5441 parameters
Epoch 2000, Loss: 0.69315, Grad range 2.5e-04 to 0.0e+00, Train
Accuracy: 0.5, Test Accuracy: 0.5
Train Accuracy: 0.50000, Test Accuracy: 0.50000
```



Note - Any idea how does pytorch initialize weights and biases for a layer by default ? Read this discussion - https://discuss.pytorch.org/t/how-are-layer-weights-and-biases-initialized-by-default/13073

```
# ######################################################################
#               TENSORBOARD INTEGRATION: LAUNCH DASHBOARD
# ######################################################################
# This command should start the TensorBoard UI right in the notebook.
# (you may need to run this cell twice)
# The --logdir points to the parent directory where we saved all our
runs.

%tensorboard --logdir runs
# ######################################################################
```

```
<IPython.core.display.Javascript object>
```

Extra Credit Questions

Extra Credit Question 1: (2 points)

Now try to train a network to classify the data in the file `hard-classification.npz`. This classification problem is very similar to the original one in `tiny-classification.npz`, with inputs that have a very similar structure. And yet the problem is harder to learn: do the same training techniques work, or is some other approach necessary? Hint: consider transfer learning or fine-tuning approaches

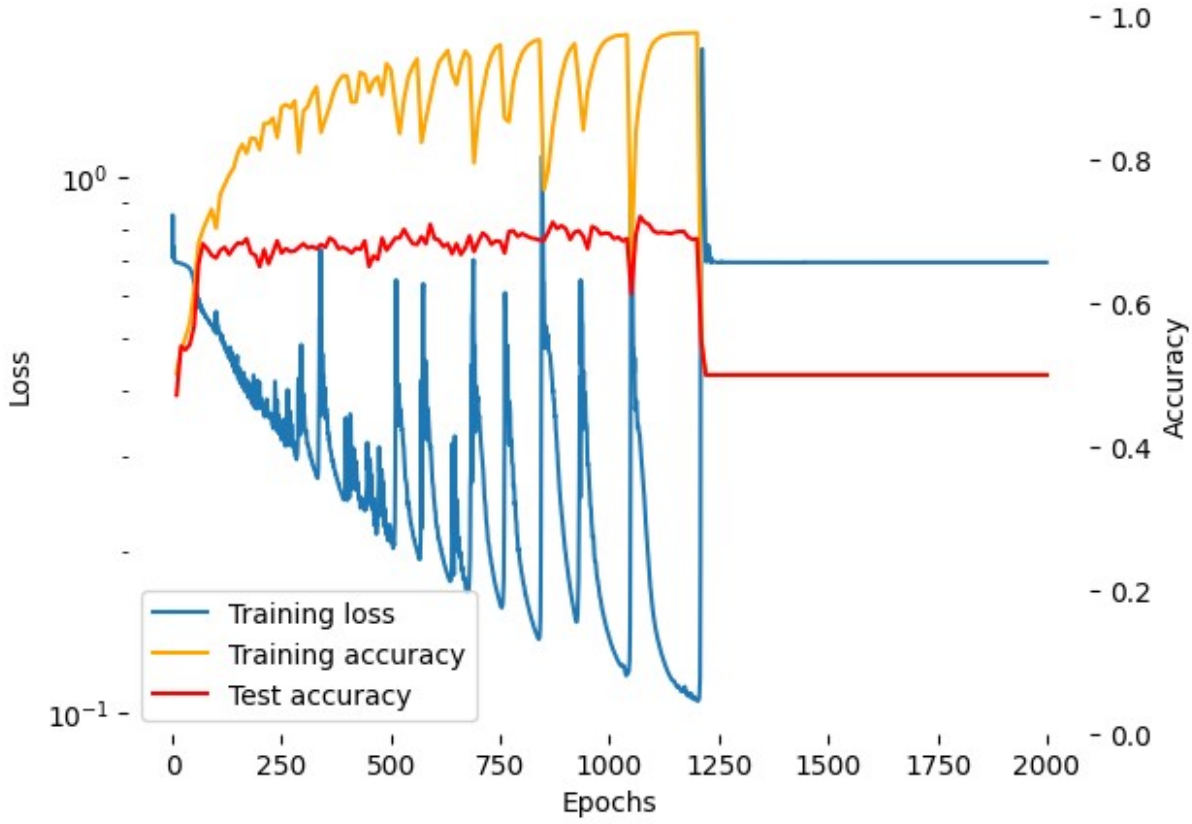
This was taking too long to run so I couldn't improve it in time. I forgot to use a GPU here to speed it up.

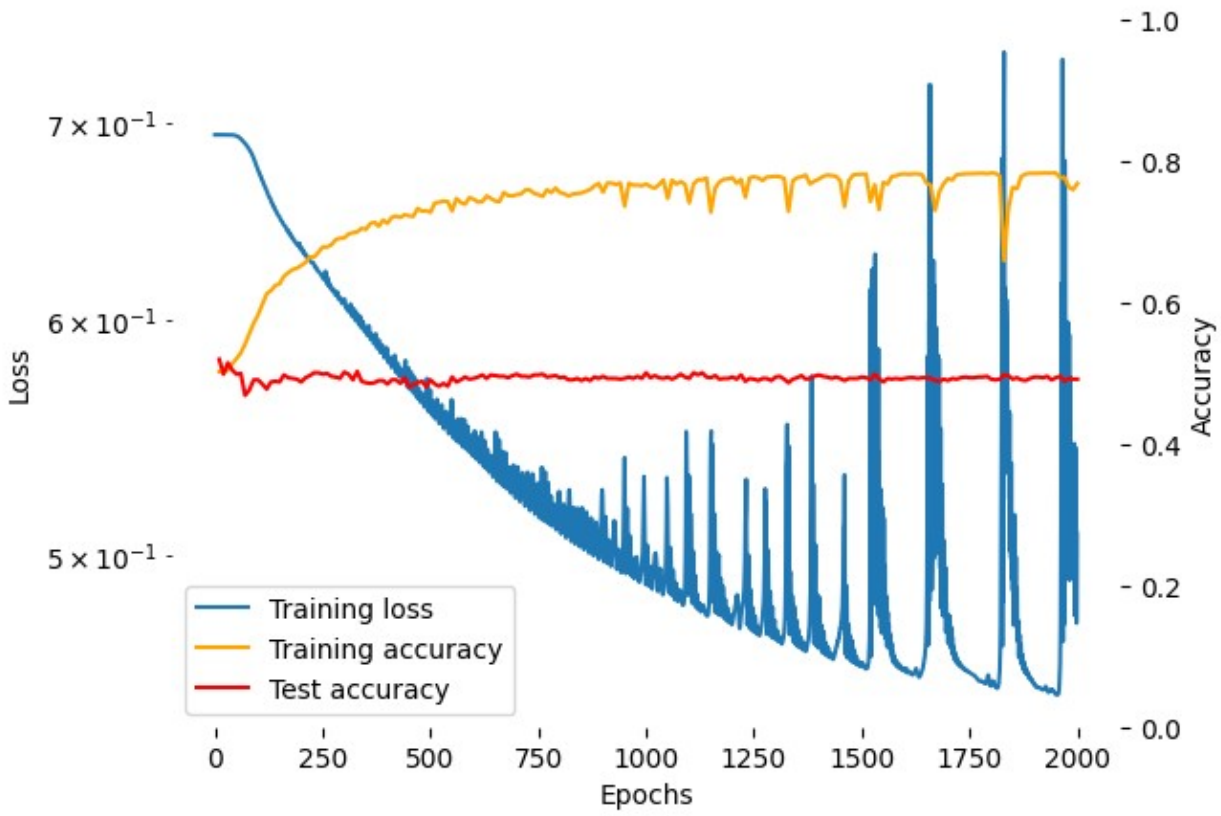
```
# Step 1: Load the hard-classification data
hard_train_data, hard_train_labels, hard_test_data, hard_test_labels =
[
    torch.tensor(m[k]).float()
    for m in [numpy.load('hard-classification.npz')]
    for k in 'train_data train_labels val_data val_labels'.split()
]

print(f'Hard training data: {hard_train_data.size(0)} samples, '
      f'each a vector of {hard_train_data.size(1)} numbers')
print(f'Hard test data: {hard_test_data.size(0)} samples')

# Step 2: Train a base model on tiny-classification using run_test
# (This is KNOWN to reach 100% from Task 4.9)
torch.manual_seed(7150)

input_size = train_data.size(1)
hidden_dims = 32
output_dims = 1
```

```
base_net = nn.Sequential(
    nn.Linear(input_size, hidden_dims),
    nn.ReLU(),
    nn.Linear(hidden_dims, hidden_dims),
    nn.ReLU(),
    nn.Linear(hidden_dims, hidden_dims),
    nn.ReLU(),
    nn.Linear(hidden_dims, hidden_dims),
    nn.ReLU(),
    nn.Linear(hidden_dims, hidden_dims),
    nn.ReLU(),
    nn.Linear(hidden_dims, output_dims),
    nn.Sigmoid()
)

base_model = Supervise_Kaiming_weights(nn.BCELoss(), base_net)

# Use run_test to train — this reaches ~100% on tiny-classification
run_test(
    base_model,
    lambda p: torch.optim.Adam(p, lr=0.01, weight_decay=1e-3),
    experiment_name="EC1-Pretrain-Tiny",
    hparam_dict={
        'activation': 'ReLU', 'optimizer': 'Adam',
        'loss': 'BCE', 'lr': 0.01,
        'hidden_dim': hidden_dims, 'depth': 6,
        'weight_decay': 1e-3, 'init': 'kaiming'
    }
)

# Step 3: Fine-tune the pretrained model on hard-classification
# Temporarily swap the global data references so run_test uses hard
data
_orig_train_data, _orig_train_labels = train_data, train_labels
_orig_test_data, _orig_test_labels = test_data, test_labels

train_data = hard_train_data
train_labels = hard_train_labels
test_data = hard_test_data
test_labels = hard_test_labels

# Fine-tune with lower lr to preserve learned features
fine_tune_model = Supervise(nn.BCELoss(), base_net)

run_test(
    fine_tune_model,
    lambda p: torch.optim.Adam(p, lr=0.001, weight_decay=1e-4),
    experiment_name="EC1-FineTune-Hard",
    hparam_dict={
        'activation': 'ReLU', 'optimizer': 'Adam',
```

```
        'loss': 'BCE', 'lr': 0.001,
        'hidden_dim': hidden_dims, 'depth': 6,
        'weight_decay': 1e-4, 'init': 'transfer_from_tiny'
    }
)
```

```
Hard training data: 8000 samples, each a vector of 36 numbers
Hard test data: 1000 samples
5441 parameters
Epoch 2000, Loss: 0.69315, Grad range 2.5e-04 to 0.0e+00, Train
Accuracy: 0.5, Test Accuracy: 0.5
Train Accuracy: 0.50000, Test Accuracy: 0.50000
5441 parameters
Epoch 2000, Loss: 0.50853, Grad range 4.9e+00 to 1.4e-02, Train
Accuracy: 0.768875002861023, Test Accuracy: 0.492000013589859
Train Accuracy: 0.76888, Test Accuracy: 0.49200
```





Extra Credit Question 2: (4 points)

One of the most serious drawbacks of deep networks is that, even if they can learn to solve a problem and recognize patterns in the data, they might not give us humans much insight about those solutions. But if we can create a network that solves a problem, it should be possible to understand that solution. As extra credit, figure out: what classification rule did the neural network learn in the above exercises when the network achieves 100% hold-out accuracy? Can you extract from the network a succinct set of rules that it implements, for example, can you decompile the network into a short python program, that can correctly assign a class to a sample?

$$ \fbox{Answer: _____}$$