Question 1

(a)

The **dual norm** for ℓ_2, is defined as:

$$\|\mathbf{a}\|_2^\dagger = \max_{\mathbf{b}\in\mathbb{R}^n, \|\mathbf{b}\|_2=1} \mathbf{a}^T\mathbf{b}$$

The dot product between vectors $\mathbf{a}$ and $\mathbf{b}$ can be written as:

$$\mathbf{a}^T\mathbf{b} = \|\mathbf{a}\|_2\|\mathbf{b}\|_2 \cos(\theta)$$

Where:

- $\|\mathbf{a}\|_2$ is the ℓ_2-norm of vector $\mathbf{a}$,
- $\|\mathbf{b}\|_2 = 1$ is the constraint on $\mathbf{b}$,
- θ is the angle between vectors $\mathbf{a}$ and $\mathbf{b}$.

The maximum value of the dot product occurs when $\theta = 0$, meaning $\mathbf{a}$ and $\mathbf{b}$ are aligned in the same direction. In this case, $\cos(\theta) = 1$, and the maximum value of the dot product is:

$$\mathbf{a}^T\mathbf{b} = \|\mathbf{a}\|_2 \cdot 1 \cdot 1 = \|\mathbf{a}\|_2$$

Thus, the dot product is maximized when $\mathbf{b}$ is aligned with $\mathbf{a}$, i.e.,

$$\mathbf{b} = \frac{\mathbf{a}}{\|\mathbf{a}\|_2}$$

Therefore, the dual norm $\|\mathbf{a}\|_2^\dagger$ is simply:

$$\|\mathbf{a}\|_2^\dagger = \|\mathbf{a}\|_2$$

(b)

Given

$$\|\mathbf{b}\|_\infty = \max_i |b_i|$$

The dual norm is defined as:

$$\|\mathbf{a}\|_\infty^\dagger = \max_{\mathbf{b}:\|\mathbf{b}\|_\infty=1} \mathbf{a}^T\mathbf{b}$$

We want to maximize the dot product $\mathbf{a}^T\mathbf{b}$, subject to the constraint that $\|\mathbf{b}\|_\infty = 1$. Written component-wise, this becomes:

$$\max_{\mathbf{b}} (a_1b_1 + a_2b_2 + \cdots + a_nb_n)$$

subject to:

$$\max_i |b_i| = 1 \quad \text{or} \quad -1 \leq b_i \leq 1, \forall i$$

We can break down the problem into maximizing each component a_ib_i separately, under the constraint $|b_i| \leq 1$. This results in the following independent optimization problems:

$$\max_{b_i} a_ib_i \quad \text{subject to} \quad -1 \leq b_i \leq 1$$

The optimal solution occurs when $b_i = \text{sign}(a_i)$. Therefore, the solution to each individual problem is:

$$b_i^* = \text{sign}(a_i)$$

Given that each $b_i^* = \text{sign}(a_i)$, the total maximized value of the dot product is:

$$\mathbf{a}^T\mathbf{b} = \sum_{i=1}^n a_i \cdot \text{sign}(a_i) = \sum_{i=1}^n |a_i|$$

Thus, the dual norm $\|\mathbf{a}\|_\infty^\dagger$ is the ℓ_1-norm of $\mathbf{a}$:

$$\|\mathbf{a}\|_\infty^\dagger = \|\mathbf{a}\|_1$$

Question 2

We are given the optimization problem:

$$\arg\min_{\Delta w \in \mathbb{R}^n} \left[\mathbf{g}^T \Delta w + \frac{\lambda}{2} \|\Delta w\|^2 \right]$$

Let:

$$\Delta w = \alpha \mathbf{t}, \quad \text{where } \|\mathbf{t}\| = 1$$

Substitute into the objective function:

$$\mathbf{g}^T(\alpha \mathbf{t}) + \frac{\lambda}{2} \|\alpha \mathbf{t}\|^2$$

Since $\|\mathbf{t}\| = 1$, we simplify the expression to:

$$\alpha(\mathbf{g}^T \mathbf{t}) + \frac{\lambda}{2} \alpha^2$$

Now, minimize with respect to α:

The function to minimize is:

$$f(\alpha) = \alpha(\mathbf{g}^T \mathbf{t}) + \frac{\lambda}{2} \alpha^2$$

Take the derivative and set it equal to zero:

$$\frac{d}{d\alpha} \left[\alpha(\mathbf{g}^T \mathbf{t}) + \frac{\lambda}{2} \alpha^2 \right] = \mathbf{g}^T \mathbf{t} + \lambda \alpha = 0$$

Solving for α, we get:

$$\alpha = -\frac{\mathbf{g}^T \mathbf{t}}{\lambda}$$

Substitute the optimal α back into the objective function:

$$f(\alpha) = -\frac{(\mathbf{g}^T \mathbf{t})^2}{2\lambda}$$

Now, we maximize $\mathbf{g}^T \mathbf{t}$ subject to $\|\mathbf{t}\| = 1$. The maximizing $\mathbf{t}$ is in the direction of $\mathbf{g}$, and the maximum value of $\mathbf{g}^T \mathbf{t}$ is the **dual norm** $\|\mathbf{g}\|^\dagger$:

$$\mathbf{t}^* = \arg\max_{\|\mathbf{t}\|=1} \mathbf{g}^T \mathbf{t}$$

The dual norm is:

$$\|\mathbf{g}\|^\dagger = \max_{\|\mathbf{t}\|=1} \mathbf{g}^T \mathbf{t}$$

Thus, the dual formulation is:

$$\arg\min_{\Delta w} \left[\mathbf{g}^T \Delta w + \frac{\lambda}{2} \|\Delta w\|^2 \right] = -\frac{\|\mathbf{g}\|^\dagger}{\lambda} \cdot \arg\max_{\|\mathbf{t}\|=1} \mathbf{g}^T \mathbf{t}$$

Question 3

(a)

As we proved in question 2

$$\arg\min_{\Delta w\in\mathbb{R}^n}\left[\mathbf{g}^T\Delta w+\frac{\lambda}{2}\|\Delta w\|_2^2\right]=-\frac{\|\mathbf{g}\|_2^\dagger}{\lambda}\cdot\arg\max_{\|\mathbf{t}\|_2=1}\mathbf{g}^T\mathbf{t}$$

Use the results we derive from question 1(a):

$$\arg\max_{\|\mathbf{t}\|_2=1}\mathbf{g}^T\mathbf{t}=\frac{\mathbf{g}}{\|\mathbf{g}\|_2}$$

$$\|\mathbf{g}\|_2^\dagger=\|\mathbf{g}\|_2$$

$$\arg\min_{\Delta w\in\mathbb{R}^n}\left[\mathbf{g}^T\Delta w+\frac{\lambda}{2}\|\Delta w\|_2^2\right]=-\frac{\|\mathbf{g}\|_2}{\lambda}\cdot\frac{\mathbf{g}}{\|\mathbf{g}\|_2}=-\frac{\mathbf{g}}{\lambda}$$

(b)

As we proved in question 2:

$$\arg\min_{\Delta w\in\mathbb{R}^n}\left[\mathbf{g}^T\Delta w+\frac{\lambda}{2}\|\Delta w\|_\infty^2\right]=-\frac{\|\mathbf{g}\|_\infty^\dagger}{\lambda}\cdot\arg\max_{\|\mathbf{t}\|_\infty=1}\mathbf{g}^T\mathbf{t}$$

From question 1(b), we know that the optimal solution for $\mathbf{t}$ under the ℓ_∞-norm is:

$$\arg\max_{\|\mathbf{t}\|_\infty=1}\mathbf{g}^T\mathbf{t}=\operatorname{sign}(\mathbf{g})$$

$$\|\mathbf{g}\|_\infty^\dagger=\|\mathbf{g}\|_1$$

Thus, the solution is:

$$\arg\min_{\Delta w\in\mathbb{R}^n}\left[\mathbf{g}^T\Delta w+\frac{\lambda}{2}\|\Delta w\|_\infty^2\right]=-\frac{\|\mathbf{g}\|_1}{\lambda}\cdot\operatorname{sign}(\mathbf{g})=-\frac{\mathbf{g}}{\lambda}$$

Question4

(b)

We are tasked with solving the following optimization problem:

$$\underset{\Delta\mathbf{W}\in\mathbb{R}^{m\times n}}{\arg\min}\left[\mathrm{trace}(\mathbf{G}^T\Delta\mathbf{W})+\frac{\lambda}{2}\|\Delta\mathbf{W}\|_*^2\right]$$

Given the SVD of $\mathbf{G}$, we can express $\mathbf{G}$ as:

$$\mathbf{G}=\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

where $\mathbf{U}\in\mathbb{R}^{m\times m}$ and $\mathbf{V}\in\mathbb{R}^{n\times n}$ are orthogonal matrices, and $\mathbf{\Sigma}$ is a diagonal matrix containing the singular values of $\mathbf{G}$.

The trace term can be rewritten as:

$$\mathrm{trace}(\mathbf{G}^T\Delta\mathbf{W})=\mathrm{trace}(\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T\Delta\mathbf{W})$$

From the given relation:

$$\max_{\mathbf{T}\in\mathbb{R}^{m\times n},\|\mathbf{T}\|_*=1}\mathrm{trace}(\mathbf{G}^T\mathbf{T})\leq\mathrm{trace}(\mathbf{\Sigma})$$

The trace is maximized when $\Delta\mathbf{W}$ aligns with the singular vectors of $\mathbf{G}$, and its magnitude is proportional to the singular values.

Using the dual formulation of steepest descent (Equation 5), we know that:

$$\Delta\mathbf{W}=-\frac{\|\mathbf{\Sigma}\|_*}{\lambda}\cdot\underset{\mathbf{T}\in\mathbb{R}^{m\times n},\|\mathbf{T}\|_*=1}{\arg\max}\mathrm{trace}(\mathbf{G}^T\mathbf{T})$$

Here, we need to maximize $\mathrm{trace}(\mathbf{G}^T\mathbf{T})$, which is the Frobenius inner product of the matrices $\mathbf{G}$ and $\mathbf{T}$. The Frobenius inner product of two matrices is maximized when the matrices are aligned, which means that $\mathbf{T}$ must align with the left and right singular vectors of $\mathbf{G}$ from its SVD.

The optimal update $\Delta\mathbf{W}$ is given by:

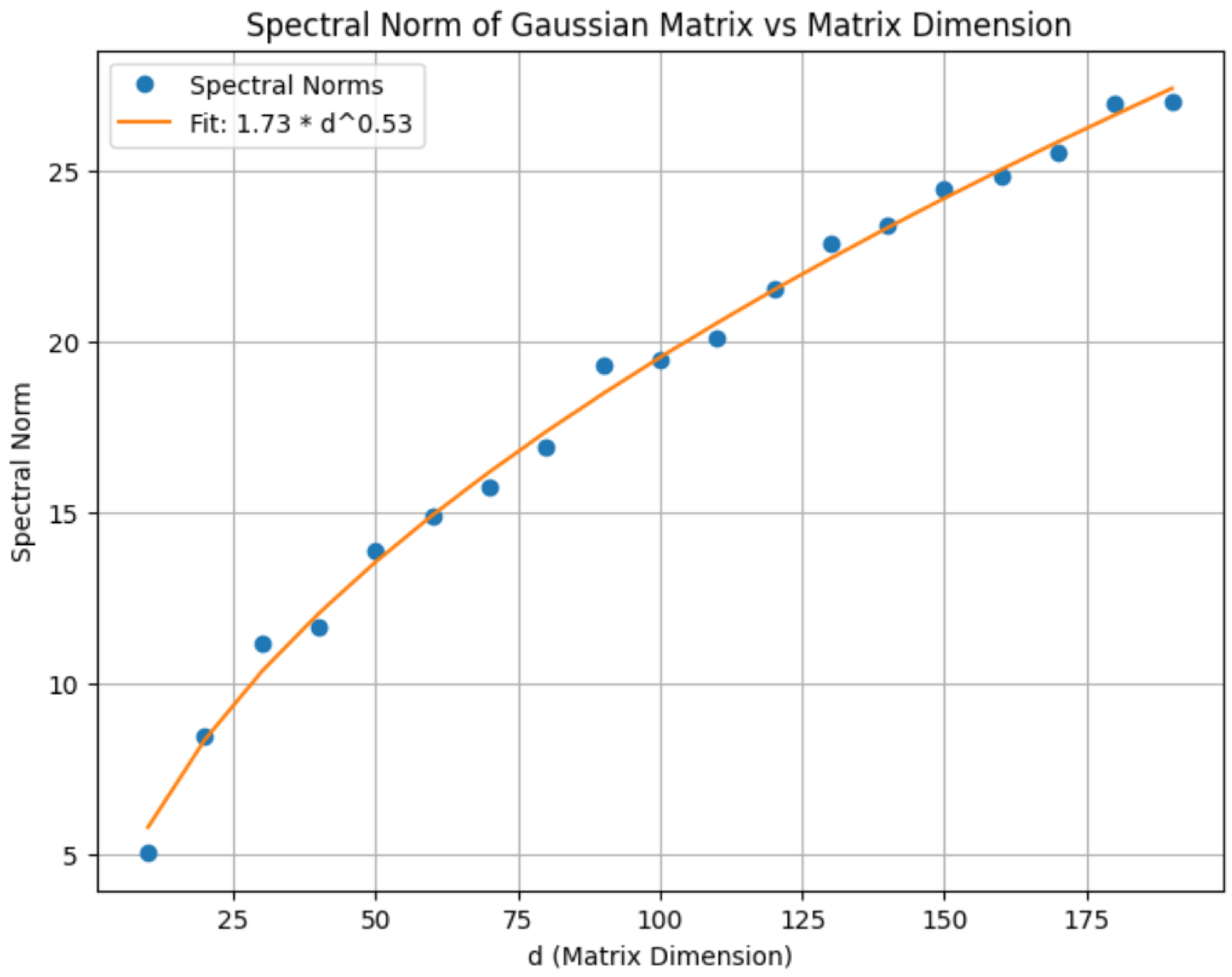
$$\Delta\mathbf{W}=-\frac{1}{\lambda}\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

where $\mathbf{U}$ and $\mathbf{V}$ are the orthogonal matrices from the SVD of $\mathbf{G}$, and $\mathbf{\Sigma}$ contains the singular values of $\mathbf{G}$. This is the steepest descent direction for the spectral norm.

Question 6

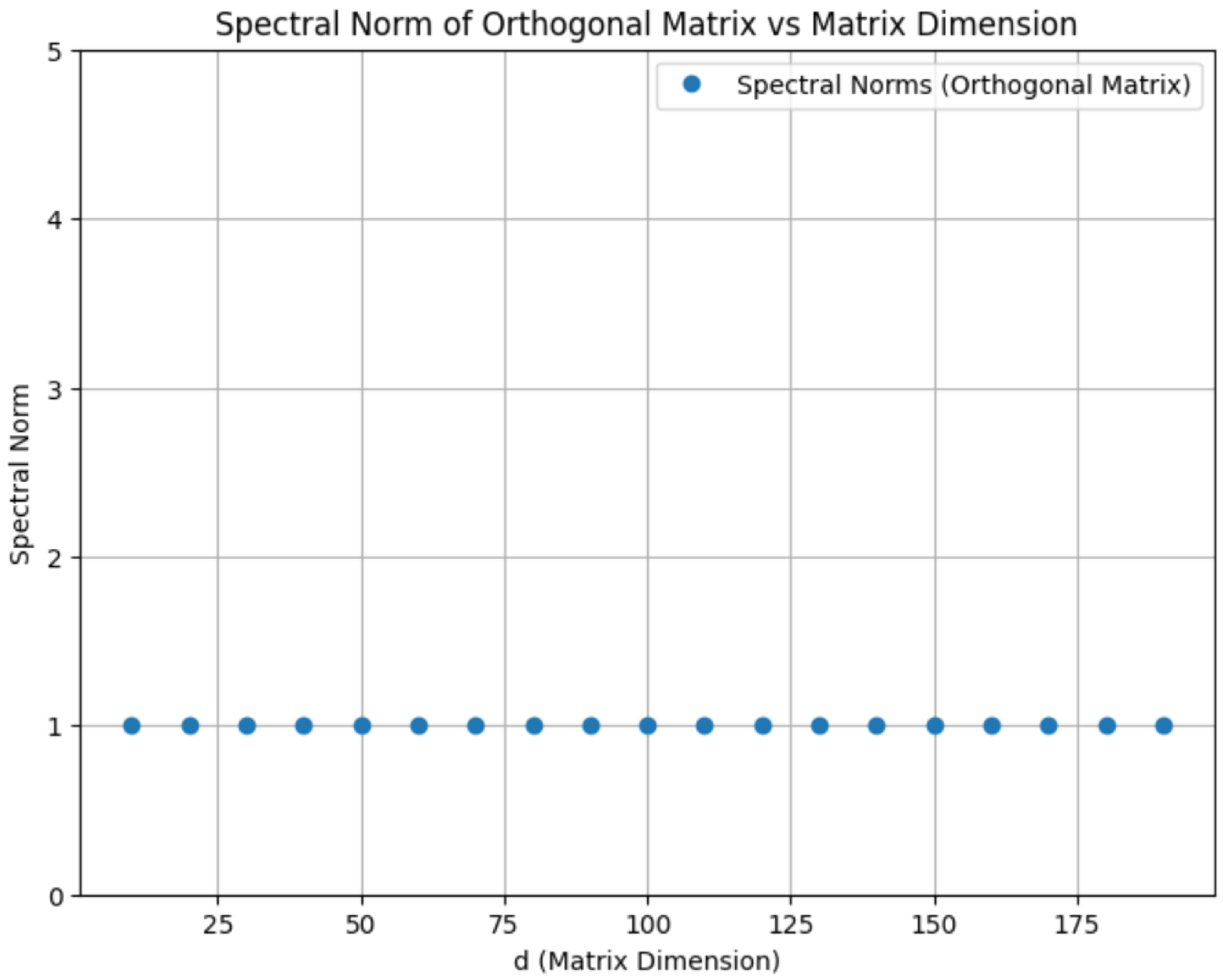
Below is the plot showing the fitted values for α and β and the spectral norms against the matrix dimension d, with the fitted curve overlaid.

α = 1.7260283992297267, β = 0.5269442163363144



Question 7

the spectral norm of an orthogonal matrix, it is always 1, regardless of the dimension d, as show in the below plot. This is because an orthogonal matrix preserves lengths, and its largest singular value (which corresponds to the spectral norm) is exactly 1.



Question 8

Below is the computed results

- Power Iteration:
  Iteration Step = 10: Spectral norm = 85.78, time = 0.033 seconds
  Iteration Step = 20: Spectral norm = 87.48, time = 0.004 seconds
  Iteration Step = 30: Spectral norm = 88.91, time = 0.01 seconds
- PyTorch Built-in:
  Spectral norm = ~89.19 (consistently across all runs), time = ~0.6 seconds\

Observations

In the results, the power iteration consistently under-estimates the spectral norm compared to PyTorch's built-in method. The difference is more pronounced with fewer iterations (e.g., at step = 10, it's much lower). As the number of iterations increases, the approximation improves and becomes closer to the actual value (e.g., step = 30 almost matches the built-in result).

With 30 iterations, the result is 88.91, which is very close to the built-in result of 89.51.

As for speed, power iteration is much faster, especially with fewer iterations, making it suitable for cases where a quick approximation is needed. However, more iterations can bring it close to the actual value without significantly increasing runtime.

Question 9

Code in the Train and Test Loop

```
def loop(net, train, eta):
    dataloader = train_loader if train else test_loader
    description = "Training... " if train else "Testing... "

    acc_list = []

    for data, target in tqdm(dataloader, desc=description):
        data, target = data.cuda(), target.cuda()
        output = net(data)

        loss = (
            output.logsumexp(dim=1).mean()
            - output[range(target.shape[0]), target].mean()
        )  # cross-entropy loss
        acc = (output.max(dim=1)[1] == target).sum() / target.shape[0]  # accuracy
        acc_list.append(acc.item())

        if train:
            loss.backward()

            for p in net.parameters():
                fan_out, fan_in = p.shape
                update = torch.sign(p.grad)
                ## START BLOCK that you should modify
                scale_factor = torch.sqrt(torch.tensor(fan_out / fan_in, dtype=torch.float32,
device=p.device))
                norm_factor = spectral_norm(p.grad, 30).item() # use power iteration for approximation
                update *= (scale_factor / norm_factor)
                ## END BLOCK that you should modify
                p.data -= eta * update
            net.zero_grad()

    return np.mean(acc_list)
```

Code in the Initialization Funcion

```
def initialize_matrix(p):
    fan_out, fan_in = p.shape
    ## START BLOCK that you should modify ##
    torch.nn.init.orthogonal_(p)
    scale_factor = math.sqrt(fan_out / fan_in)
    p.data *= scale_factor
    ## END BLOCK that you should modify
```

Observations

- **Before Learning Rate Transfer Mechanism:**
    - Small Width Network (width=32):
      Best learning rate: 0.001
      Training accuracy: 35.4%, Test accuracy: 37.0%
    - Large Width Network (width=4096):
        - Learning rate 0.001 performed poorly:
          Training accuracy: 23.2%, Test accuracy: 26.4%
        - Learning rate 0.0001 performed better: Training accuracy: 38.0%, Test accuracy: 40.0%
- **After Implementing Learning Rate Transfer Mechanism:**
    - Small Width Network (width=32):
      Best learning rate: 0.001
      Training accuracy: 33.5%, Test accuracy: 37.7%
    - Large Width Network (width=4096):
      With learning rate 0.001:
      Training accuracy: 24.2%, Test accuracy: 36.1%
- **Spectral Norm Calculation:** PyTorch's built-in spectral norm took over 12 hours on large-width networks, while power iteration reduced the time to 20 seconds with comparable results.
- **Learning Rate Transfer**: Initially, the learning rate did not transfer well between architectures. After implementing the transfer mechanism, learning rate 0.001 transferred more effectively between the small and large networks, though further tuning may be needed.

Question 11

Given that weight decay updates a weight matrix $\mathbf{W}$ as follows:

$$\mathbf{W} \to \mathbf{W} \times 0.999$$

The singular value decomposition (SVD) of $\mathbf{W}$ is:

$$\mathbf{W} = \sum_{i=1}^{\text{rank}(\mathbf{W})} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

Where σ_i are the singular values, and $\mathbf{u}_i$ and $\mathbf{v}_i$ are the singular vectors.

After applying weight decay, each entry of $\mathbf{W}$ is scaled by a factor of 0.999. Therefore, the singular values σ_i are also scaled by 0.999:

$$\sigma_i \to 0.999 \times \sigma_i$$

The singular vectors $\mathbf{u}_i$ and $\mathbf{v}_i$ are **unchanged** by the weight decay. This is because weight decay scales the entire matrix $\mathbf{W}$, and scaling $\mathbf{W}$ does not alter the directions of its singular vectors—only the magnitudes of the singular values are affected.

Thus, we have:

$$\mathbf{u}_i \to \mathbf{u}_i$$

$$\mathbf{v}_i \to \mathbf{v}_i$$

Question 12

CNN Arch

Code

```python
import torch.nn as nn


def make_cnn(
    num_outputs: int, activation: str, num_conv_layers: int = 4, num_fc_layers: int = 2
) -> nn.Module:
    if activation == "relu":
        act_cls = nn.ReLU
    elif activation == "tanh":
        act_cls = nn.Tanh
    elif activation == "none":
        act_cls = nn.Identity
    else:
        raise ValueError(f"Unexpected activation={repr(activation)}")
    net = [
        nn.Conv2d(
            in_channels=3,
            out_channels=12,
            kernel_size=5,
            padding_mode="reflect",
            padding=2,
            stride=1,
        ),
        act_cls(),
    ]
    num_out = 12
    for _ in range(num_conv_layers - 1):
        net.extend(
            [
                nn.Conv2d(
                    in_channels=num_out,
                    out_channels=64,
                    kernel_size=3,
                    padding=1,
                    stride=2,
                ),
                act_cls(),
            ]
        )
        num_out = 64
    net.extend([nn.Flatten()])
    for _ in range(num_fc_layers - 1):
        net.extend([nn.LazyLinear(out_features=128), act_cls()])
    net.extend([nn.LazyLinear(out_features=num_outputs)])
    return nn.Sequential(*net)
```

[ ]

Python

Print Results

```
num_outputs=100, tanh, num_conv_layers=5, num_fc_layers=4:
Sequential(
(0): Conv2d(3, 12, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), padding_mode=reflect)
(1): Tanh()
(2): Conv2d(12, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
(3): Tanh()
(4): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
(5): Tanh()
(6): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
(7): Tanh()
(8): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
(9): Tanh()
(10): Flatten(start_dim=1, end_dim=-1)
(11): LazyLinear(in_features=0, out_features=128, bias=True)
(12): Tanh()
(13): LazyLinear(in_features=0, out_features=128, bias=True)
(14): Tanh()
(15): LazyLinear(in_features=0, out_features=128, bias=True)
(16): Tanh()
(17): LazyLinear(in_features=0, out_features=100, bias=True)
)

num_outputs=3, no activation, num_conv_layers=3, num_fc_layers=2:
Sequential(
(0): Conv2d(3, 12, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), padding_mode=reflect)
(1): Identity()
(2): Conv2d(12, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
(3): Identity()
(4): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
(5): Identity()
(6): Flatten(start_dim=1, end_dim=-1)
(7): LazyLinear(in_features=0, out_features=128, bias=True)
(8): Identity()
(9): LazyLinear(in_features=0, out_features=3, bias=True)
)
```

Question 13

train_epoch

```python
device = None
import torch
import tqdm


def train_epoch(
    epoch: int,
    model: nn.Module,
    train_loader: torch.utils.data.DataLoader,
    optim: torch.optim.Optimizer,
):
    loss_values: List[float] = []

    loss_fn = nn.CrossEntropyLoss()

    for data, target in tqdm(train_loader, desc=f"Training @ epoch {epoch}"):
        optim.zero_grad()
        data = data.to(device)
        target = target.to(device)
        # make predictions
        prediction = model(data)

        # compute loss
        loss = loss_fn(input=prediction, target=target)
        loss_values.append(loss.item())
        loss.backward()
        optim.step()

    return loss_values
```

[ ] Python

evaluate

```python
class EvaluateResult:
    r"""
    A collection containing everything we need to know about the evaluate results.
    See `evaluate` docstring for meanings of the members of this class
    """

    acc: float  # overall accuracy

    correct_predictions: torch.Tensor  # size |dataset|

    confidence: torch.Tensor  # size |dataset|


@torch.no_grad()
def evaluate(model: nn.Module, loader: torch.utils.data.DataLoader) -> EvaluateResult:
    correct_predictions = []
    confidence = []
    model.eval()
    for data, target in loader:
        data = data.to(device)
        target = target.to(device)
        output = model(data)
        prob = torch.softmax(output, dim=1)
        pred = torch.argmax(prob, dim=1)
        conf = torch.max(prob, dim=1).values
        correct_predictions.append((pred == target).detach().cpu())
        confidence.append(conf.detach().cpu())

    correct_predictions = torch.cat(correct_predictions).cpu()
    confidence = torch.cat(confidence).cpu()
    acc = correct_predictions.float().mean().item()
    eval_result = EvaluateResult(acc, correct_predictions, confidence)
    return eval_result
```

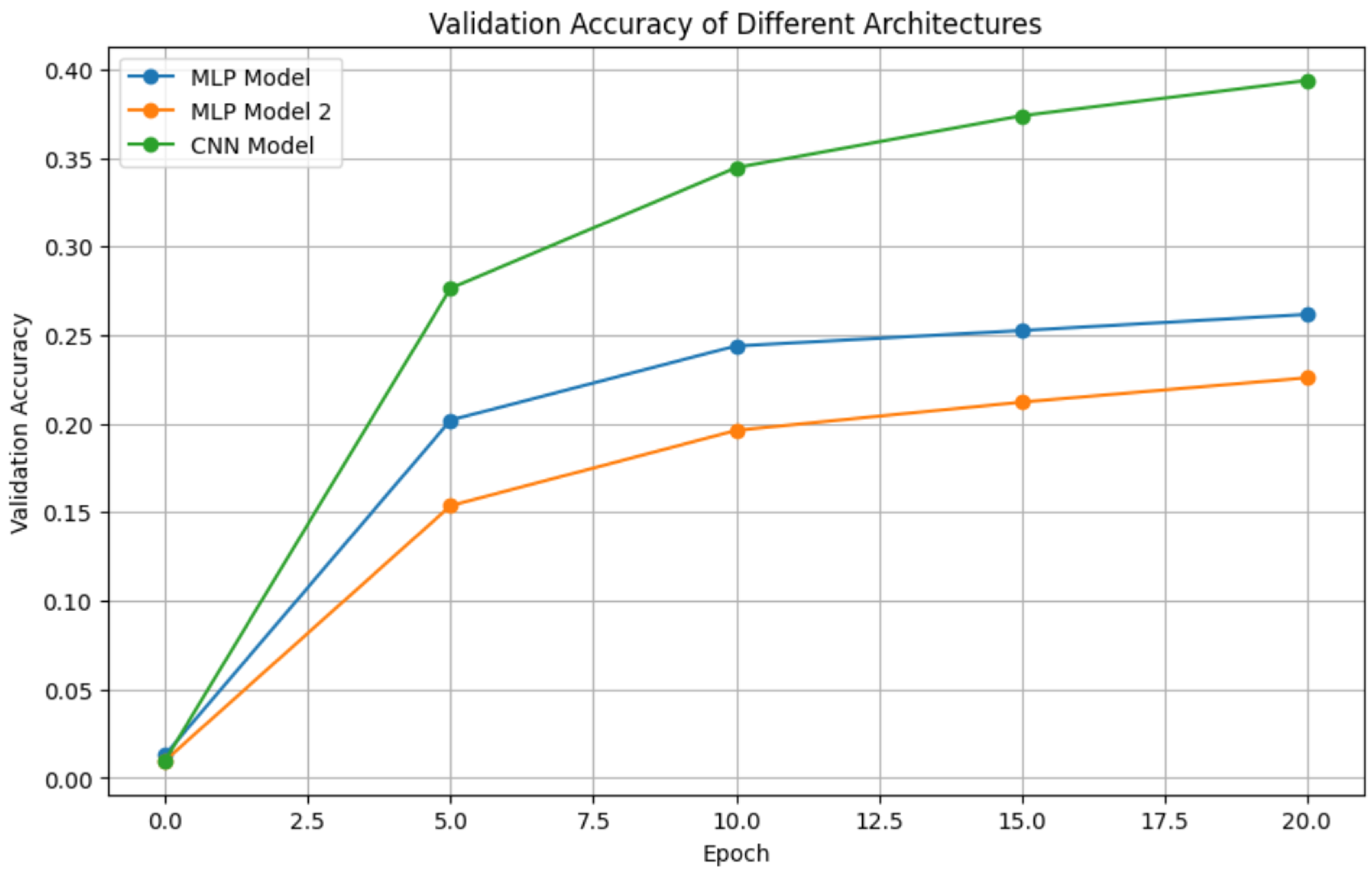
[ ] Python

train

```python
optim = torch.optim.Adam(model.parameters(), lr=lr)
scheduler = torch.optim.lr_scheduler.MultiStepLR(
    optim, milestones=[5, 50], gamma=0.3
)
```

Question 14

(a)



(b)

From the plot above, CNN model outperforms both MLP models, reaching a validation accuracy of nearly 40% by epoch 20.

Why did CNN perform better?

CNNs perform best because they have an inherent inductive bias tailored for image data, specifically shift invariance and local connectivity. CNNs use convolutional filters to capture spatial relationships, allowing them to detect patterns (e.g., edges, textures) anywhere in the image. This weight sharing across the image leads to fewer parameters, better generalization, and the ability to recognize objects regardless of their position (shift invariance). In contrast, MLPs treat each pixel independently, lacking this spatial awareness, resulting in poorer performance for image-related tasks.

Would arbitrarily deeper MLPs significantly improve accuracy?

No, arbitrarily deeper MLPs (like the MLP Model 2 in the plot) would not significantly improve validation or test accuracy for the following reasons:

- MLPs cannot capture spatial hierarchies or local patterns in the data, which are essential for images. Deeper MLPs simply add more layers but do not introduce the convolutional layers necessary to handle images effectively.
- Adding more layers increases the number of parameters, leading to a higher likelihood of overfitting and making the model harder to train without much gain in performance.

Question 15

(a)

$$g(h_u) = h_u$$

$$f\left(\sum_{u \in \mathcal{N}(v)} g(h_u)\right) = \frac{1}{|\mathcal{N}(v)|} \sum_{u \in \mathcal{N}(v)} g(h_u)$$

PyTorch Code Implementation:

```
import torch
import torch.nn.functional as F

def f(x, num_neighbors):
    # average aggregation
    return x / num_neighbors

def g(x):
    return x
```

(b)

$$g(h_u) = \exp(h_u)$$

$$f\left(\sum_{u \in \mathcal{N}(v)} g(h_u)\right) = \log\left(\sum_{u \in \mathcal{N}(v)} g(h_u)\right)$$

PyTorch Code Implementation:

```
import torch

def f(x):
    return torch.exp(h_u)

def g(h_u):
    return torch.log(x)
```

Question 16

(a)

Yes,see below discussion of both $generic_f, g$ and $\max$ AGGREGATE function:

1. Generic Aggregation Case:

- **Aggregation**: Define the aggregation function as averaging neighbor features, therefore node features will remain the same during iterations since the initial features are the same
- **Update Function**: Can be flexible, such as identity function
- **READOUT Function**: The final **READOUT** can sum the features of all nodes:

$$\text{READOUT} = \frac{\sum_{v \in G} h_v^{(k)}}{\text{mean}(h_v^{(k)})}$$

Since $\text{mean}(h_v^{(k)}) = h_v$ (as all features are equal), this is equivalent to summing the node features, which gives the total number of nodes.

2. Max Aggregation Case:

- **Aggregation**: Max aggregation results in all nodes having the same feature throughout the iterations since the initial features are the same.
- **Update Function**: Can be flexible, such as identity function
- **READOUT Function**: We can compute the total number of nodes using the following readout:

$$\text{READOUT} = \frac{\sum_{v \in G} h_v^{(k)}}{\max(h_v^{(k)})}$$

Since $\max(h_v^{(k)}) = h_v$ (as all features are equal), this is equivalent to summing the node features, which gives the total number of nodes.

(b)

Yes, see discussion below

1. Generic Aggregation Case:

- **Aggregation Functions**:
  - $g(h_u) = h_u + 1$ (increment distance for neighbors)
  - $f(x) = \min(x)$ (retain the smallest distance)
- **Update Function**: Nodes propagate distance information, and each node updates its feature to the minimum distance from its neighbors.

$$h_u^{(k+1)} = \min\left(h_u^{(k)}, \min_{w \in \mathcal{N}(u)}\left(h_w^{(k)}\right)\right)$$

- **Readout Function**:

$$\text{READOUT} = \max_{u \in G} h_u^{(k)}$$

This computes the maximum shortest distance after the final iteration.

2. Max Aggregation Case:

- **Update Function**:

$$h_u^{(k+1)} = \min\left(h_u^{(k)}, \min_{w \in \mathcal{N}(u)}\left(h_w^{(k)} + 1\right)\right)$$

This ensures that nodes propagate the smallest feature (tracking the shortest distance) and increment the path length at each iteration.

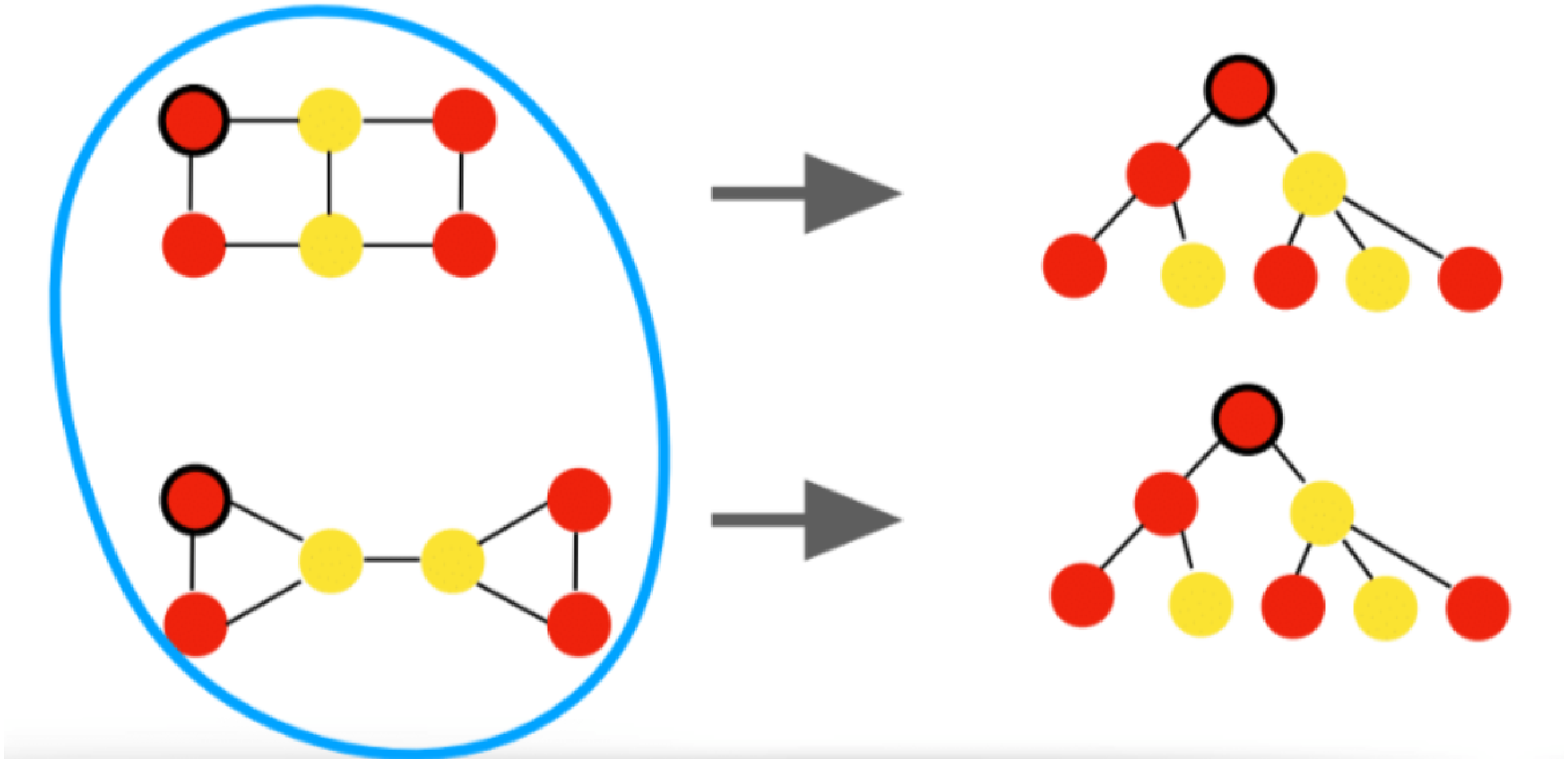
- **Readout Function**:

$$\text{READOUT} = \max_{u \in G} h_u^{(k)}$$

This gives the maximum shortest distance after the final iteration.

(c)

No, a GNN cannot differentiate between two graphs with the same unrolled tree structures at each node, even if they contain a different number of triangles. This is because GNNs aggregate node features based on local neighborhood information, and if two nodes share identical unrolled tree structures, the GNN will compute the same feature representation for them.See below examples from lecture slides:



- The two graphs have identical local structures (as seen by their unrolled tree structures), but the upper graph has 0 triangles, while the lower graph has 2 triangles. Because GNNs cannot differentiate between nodes with the same neighborhood structures, they will treat these two graphs as isomorphic, even though they have a different number of triangles.

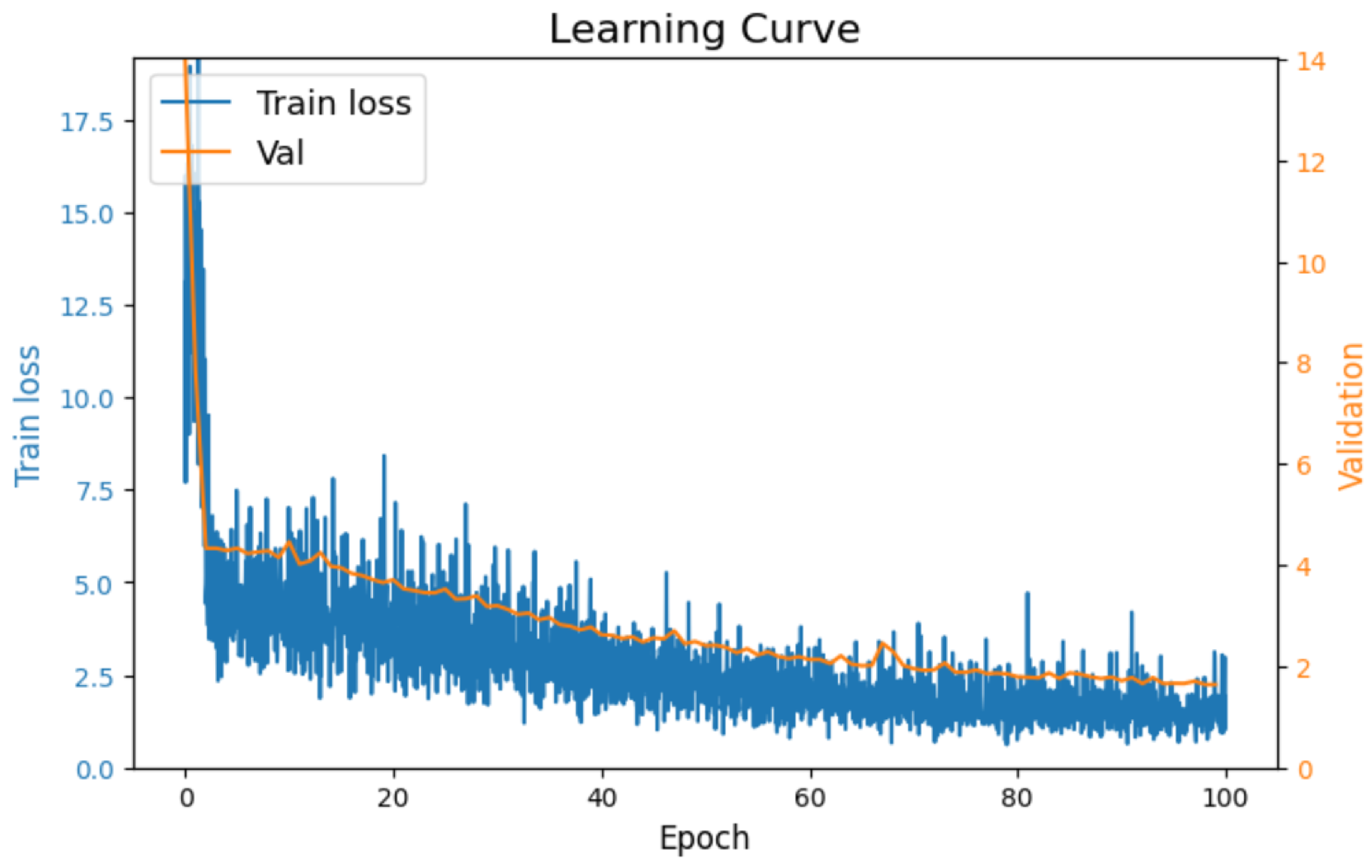
Question 17

(a)

No, MP-GNNs using the generic aggregation function cannot differentiate between mirror images of chiral molecules. This limitation arises because MP-GNNs operate similarly to the 1-dimensional Weisfeiler-Lehman (WL) test, which is used to determine graph isomorphism. Since the mirror images of chiral molecules have the same local neighborhood structure (i.e., the same unrolled tree structure), the 1-WL algorithm cannot distinguish between them. Therefore, MP-GNNs, which rely on this kind of aggregation, cannot differentiate between such mirror images.

Question 18

The validation losses are plotted below:


Learning Curve
Train loss
Val
Train loss
Validation
Epoch