Agreement: This assignment represents my own work. I did not work on this assignment with others. All coding was done by myself.

Q1

(a)

Soft decision tree routes a sample through both children, each of whom has a specific probability weight, by calculating the probability of picking each branch at a decision node. In contrast, there is no probability involved in the routing process in a **hard decision tree** because each decision node generates a strict binary split based on a predetermined feature threshold.

Advantages (Soft): Firstly, the soft decision tree is differentiable and back-propagation compatible. Secondly, the probabilistic nature allows for smoother decision boundaries.

Advantages (Hard): Hard decision trees are easier to interpret since only the nodes along a path account for the ultimate prediction.

(b)

Roll out the recursive definition:

$$traverse(\boldsymbol{P_1}, \boldsymbol{z_i}) = sim(\boldsymbol{P_1}, \boldsymbol{z_i})\boldsymbol{c_3} + \big(1 - sim(\boldsymbol{P_1}, \boldsymbol{z_i})\big)traverse(\boldsymbol{P_2}, \boldsymbol{z_i})$$

$$traverse(\boldsymbol{P_2}, \boldsymbol{z_i}) = sim(\boldsymbol{P_2}, \boldsymbol{z_i})\boldsymbol{c_2} + \big(1 - sim(\boldsymbol{P_2}, \boldsymbol{z_i})\big)\boldsymbol{c_1}$$

Therefore, the vector of class logits is computed as:

$$f(\boldsymbol{z_i}) = sim(\boldsymbol{P_1}, \boldsymbol{z_i})\boldsymbol{c_3} + \big(1 - sim(\boldsymbol{P_1}, \boldsymbol{z_i})\big)\big(sim(\boldsymbol{P_2}, \boldsymbol{z_i})\boldsymbol{c_2} + \big(1 - sim(\boldsymbol{P_2}, \boldsymbol{z_i})\big)\boldsymbol{c_1}\big)$$

$$f(\boldsymbol{z_i}) = e^{-\| z_i - \boldsymbol{P}_1 \|}\boldsymbol{c_3} + \left(1 - e^{-\| z_i - \boldsymbol{P}_1 \|}\right)\left(e^{-\| z_i - \boldsymbol{P}_2 \|}\boldsymbol{c_2} + \left(1 - e^{-\| z_i - \boldsymbol{P}_2 \|}\right)\boldsymbol{c_1}\right)$$

Where $\boldsymbol{z_i}$ is the minimizer over all prototypes, $\boldsymbol{P}_{j\in 1,2}$ is the prototype of each non-leaf node, and $\boldsymbol{c}_{k\in 1,2,3}$ is the classification vector for each leaf.

(c)

According to the paper, the ground-truth label y_i is one-hot encoded, therefore, in this case, the loss can be written as:

$$\ell(\boldsymbol{y_i}, \boldsymbol{\hat{y}_i}) = -y_{i,a}\log(\hat{y}_{i,a})$$

Therefore,

$$\frac{\partial \ell}{\partial \hat{y}_{i,a}} = -y_{i,a}\frac{1}{\hat{y}_{i,a}} = -\frac{y_{i,a}}{\hat{y}_{i,a}} \qquad (1)$$

From **(b)**, we know that:

$$\hat{y}_i = e^{-\| z_i - \boldsymbol{P}_1 \|}\boldsymbol{c_3} + \left(1 - e^{-\| z_i - \boldsymbol{P}_1 \|}\right)\left(e^{-\| z_i - \boldsymbol{P}_2 \|}\boldsymbol{c_2} + \left(1 - e^{-\| z_i - \boldsymbol{P}_2 \|}\right)\boldsymbol{c_1}\right)$$

Therefore,

$$\hat{y}_{i,a} = e^{-\| z_i - \boldsymbol{P}_1 \|}c_{3,a} + \left(1 - e^{-\| z_i - \boldsymbol{P}_1 \|}\right)\left(e^{-\| z_i - \boldsymbol{P}_2 \|}c_{2,a} + \left(1 - e^{-\| z_i - \boldsymbol{P}_2 \|}\right)c_{1,a}\right)$$

Where $c_{1,a}, c_{2,a}, c_{3,a}$ represent the term related to class a in vector $\boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{c_3}$ respectively.

Let $h = \| \boldsymbol{z}_\mathrm{i} - \boldsymbol{P_1} \|$, therefore,

$$\hat{y}_{i,a} = e^{-h}c_{3,a} + (1 - e^{-h})\left(e^{-\| z_\mathrm{i} - \boldsymbol{P}_2 \|}c_{2,a} + \left(1 - e^{-\| z_\mathrm{i} - \boldsymbol{P}_2 \|}\right)c_{1,a}\right)$$

Then we can derive:

$$\frac{\partial \hat{y}_{i,a}}{\partial h} = -e^{-h}c_{3,a} + e^{-h}\left(e^{-\| z_\mathrm{i} - \boldsymbol{P}_2 \|}c_{2,a} + \left(1 - e^{-\| z_\mathrm{i} - \boldsymbol{P}_2 \|}\right)c_{1,a}\right) \qquad (2)$$

Now look at h:

$$h = \| \boldsymbol{z}_{\mathbf{i}} - \boldsymbol{P}_{\mathbf{1}} \| = \sqrt{\sum_{k} (z_i^{(k)} - p_1^{(k)})^2}$$

Therefore,

$$\frac{\partial h}{\partial p_1^{(k)}} = \frac{1}{2\sqrt{\sum_k \left(z_i^{(k)} - p_1^{(k)}\right)^2}} 2\left(z_i^{(k)} - p_1^{(k)}\right)(-1) = \frac{p_1^{(k)} - z_i^{(k)}}{\sqrt{\sum_k \left(z_i^{(k)} - p_1^{(k)}\right)^2}} \quad (3)$$

By multiplying (1) (2) (3) together, the partial derivative $\frac{\partial \ell}{\partial p_1^{(k)}}$ can be derived as:

$$\frac{\partial \ell}{\partial p_1^{(k)}} = \frac{\partial \ell}{\partial \hat{y}_{i,a}} \frac{\partial \hat{y}_{i,a}}{\partial h} \frac{\partial h}{\partial p_1^{(k)}}$$

$$\frac{\partial \ell}{\partial p_1^{(k)}} = -\frac{y_{i,a}}{\hat{y}_{i,a}} \left(-e^{-h} c_{3,a} + e^{-h}\left(e^{-\| \boldsymbol{z}_{\mathbf{i}} - \boldsymbol{P}_{2} \|} c_{2,a} + \left(1 - e^{-\| \boldsymbol{z}_{\mathbf{i}} - \boldsymbol{P}_{2} \|}\right) c_{1,a}\right)\right) \frac{p_1^{(k)} - z_i^{(k)}}{\sqrt{\sum_k \left(z_i^{(k)} - p_1^{(k)}\right)^2}}$$

Because $h = \| \boldsymbol{z}_{\mathbf{i}} - \boldsymbol{P}_{\mathbf{1}} \| = \sqrt{\sum_k (z_{\mathrm{i}}^{(k)} - p_1^{(k)})^2}$,

$$\frac{\partial \ell}{\partial p_1^{(k)}} = \frac{-y_{i,a}}{\hat{y}_{i,a}} \left(-e^{-\| \boldsymbol{z}_{\mathbf{i}} - \boldsymbol{P}_{\mathbf{1}} \|} c_{3,a} + e^{-\| \boldsymbol{z}_{\mathbf{i}} - \boldsymbol{P}_{\mathbf{1}} \|}\left(e^{-\| \boldsymbol{z}_{\mathbf{i}} - \boldsymbol{P}_{2} \|} c_{2,a} + \left(1 - e^{-\| \boldsymbol{z}_{\mathbf{i}} - \boldsymbol{P}_{2} \|}\right) c_{1,a}\right)\right) \frac{p_1^{(k)} - z_i^{(k)}}{\| \boldsymbol{z}_{\mathbf{i}} - \boldsymbol{P}_{\mathbf{1}} \|}$$

(d)

Each prototype P_{n} is replaced with its nearest latent patch present in the training data $\tilde{z}_n^*$ by using the equation below:

$$P_{\mathrm{n}} \leftarrow \tilde{z}_n^*, \ \ \tilde{z}_n^* = \mathrm{argmin} \| \tilde{z}^* - P_{\mathrm{n}} \|$$

Then, the prototype P_{n} can be visualized as a patch of training image corresponding to $\tilde{z}_n^*$.

HW5_Q2

October 30, 2023

[1]:
```
# import necessary dependencies
import argparse
import os, sys
import time
import datetimek
from tqdm import tqdm_notebook as tqdm
import numpy as np

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

import torch.nn as nn
import torch.optim as optim

import matplotlib.pyplot as plt

import random
```

[2]:
```
def set_all_seeds(RANDOM_SEED):
    random.seed(RANDOM_SEED)     # python random generator
    np.random.seed(RANDOM_SEED)  # numpy random generator

    torch.manual_seed(RANDOM_SEED)
    torch.cuda.manual_seed_all(RANDOM_SEED)

    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
set_all_seeds(42)
```

[3]:
```
class SimpleCIFAR10Classifier(nn.Module):
    def __init__(self):
        super(SimpleCIFAR10Classifier, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.fc1   = nn.Linear(16*6*6, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)
```

```python
    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.conv2(out))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out
```

[4]:

```python
# useful libraries
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader


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

##################
# YOUR CODE HERE #
##################

# adjust batch size to your need
batch_size = 64

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
val_size = int(0.5 * len(testset))
test_size = len(testset) - val_size
valset, testset = torch.utils.data.random_split(testset, [val_size, test_size])


val_loader = torch.utils.data.DataLoader(valset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

test_loader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

```
print(len(trainset), len(valset), len(testset))
```

```
Files already downloaded and verified
Files already downloaded and verified
50000 5000 5000
```

1 (a)

[5]:
```
net = SimpleCIFAR10Classifier().cuda()
INITIAL_LR = 0.01
MOMENTUM = 0.9
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=INITIAL_LR, momentum=MOMENTUM)

EPOCHS = 30
CHECKPOINT_FOLDER = "./saved_model"
best_val_acc = 0

# Training Loop
train_losses = []
val_losses = []
for i in range(0, EPOCHS):

    net.train()

    print("Epoch %d:" %i)

    total_examples = 0
    correct_examples = 0

    # Record training loss
    train_loss = 0

    # Looping through training loader
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        ##################
        # YOUR CODE HERE #
        ##################

        # Send input and target to device
        inputs, targets = inputs.cuda(), targets.cuda()
        # compute the model output logits and training loss
        outputs = net(inputs)
        loss = criterion(outputs, targets)
        # back propogation & optimizer update parametes
        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()
        # calculate predictions
        _, predictions = torch.max(outputs, 1)
        correct_examples += (predictions == targets).sum().item()
        total_examples += inputs.shape[0]
        train_loss += loss.cpu().detach().numpy()

    # calculate average training loss and accuracy
    avg_loss = train_loss / len(train_loader)
    avg_acc = correct_examples / total_examples
    print("Training loss: %.4f, Training accuracy: %.4f" %(avg_loss, avg_acc))
    train_losses.append(avg_loss)


    # Evaluate the validation set performance
    net.eval()

    total_examples = 0
    correct_examples = 0

    # Record validation loss
    val_loss = 0

    # disable gradient during validation, which can save GPU memory
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(val_loader):
            ##################
            # YOUR CODE HERE #
            ##################

            # Send input and target to device
            inputs, targets = inputs.cuda(), targets.cuda()
            # compute the model output logits and training loss
            outputs = net(inputs)
            loss = criterion(outputs, targets)
            # count the number of correctly predicted samples in the current␣
↪batch
            _, predictions = torch.max(outputs, 1)
            correct_examples += (predictions == targets).sum().item()
            total_examples += inputs.shape[0]
            val_loss += loss.cpu().detach().numpy()

    # calculate average validation loss and accuracy
    avg_loss = val_loss / len(val_loader)
    avg_acc = correct_examples / total_examples
    print("Validation loss: %.4f, Validation accuracy: %.4f" % (avg_loss,␣
↪avg_acc))
```

```python
    val_losses.append(avg_loss)

    # save the model checkpoint
    current_learning_rate = optimizer.state_dict()['param_groups'][0]['lr']
    if avg_acc > best_val_acc:
        best_val_acc = avg_acc

        if not os.path.exists(CHECKPOINT_FOLDER):
           os.makedirs(CHECKPOINT_FOLDER)
        print("Saving ...")
        state = {'state_dict': net.state_dict(),
                'epoch': i,
                'lr': current_learning_rate}
        torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'best_model.bin'))

    print('')

print(f"Best validation accuracy: {best_val_acc:.4f}")
```

```
Epoch 0:
Training loss: 1.8587, Training accuracy: 0.3090
Validation loss: 1.5058, Validation accuracy: 0.4594
Saving …

Epoch 1:
Training loss: 1.3745, Training accuracy: 0.5023
Validation loss: 1.3045, Validation accuracy: 0.5366
Saving …

Epoch 2:
Training loss: 1.1939, Training accuracy: 0.5760
Validation loss: 1.1641, Validation accuracy: 0.5888
Saving …

Epoch 3:
Training loss: 1.0741, Training accuracy: 0.6174
Validation loss: 1.1005, Validation accuracy: 0.6140
Saving …

Epoch 4:
Training loss: 0.9947, Training accuracy: 0.6489
Validation loss: 1.0457, Validation accuracy: 0.6350
Saving …

Epoch 5:
Training loss: 0.9217, Training accuracy: 0.6745
Validation loss: 1.0606, Validation accuracy: 0.6342
```

```
Epoch 6:
Training loss: 0.8584, Training accuracy: 0.6966
Validation loss: 1.0272, Validation accuracy: 0.6518
Saving …

Epoch 7:
Training loss: 0.8094, Training accuracy: 0.7156
Validation loss: 1.0099, Validation accuracy: 0.6476

Epoch 8:
Training loss: 0.7581, Training accuracy: 0.7310
Validation loss: 1.0078, Validation accuracy: 0.6548
Saving …

Epoch 9:
Training loss: 0.7111, Training accuracy: 0.7477
Validation loss: 1.0560, Validation accuracy: 0.6482

Epoch 10:
Training loss: 0.6745, Training accuracy: 0.7601
Validation loss: 1.0871, Validation accuracy: 0.6482

Epoch 11:
Training loss: 0.6360, Training accuracy: 0.7736
Validation loss: 1.0997, Validation accuracy: 0.6426

Epoch 12:
Training loss: 0.6000, Training accuracy: 0.7859
Validation loss: 1.1334, Validation accuracy: 0.6558
Saving …

Epoch 13:
Training loss: 0.5781, Training accuracy: 0.7921
Validation loss: 1.1609, Validation accuracy: 0.6418

Epoch 14:
Training loss: 0.5427, Training accuracy: 0.8049
Validation loss: 1.2202, Validation accuracy: 0.6306

Epoch 15:
Training loss: 0.5203, Training accuracy: 0.8129
Validation loss: 1.2312, Validation accuracy: 0.6490

Epoch 16:
Training loss: 0.5012, Training accuracy: 0.8201
Validation loss: 1.3026, Validation accuracy: 0.6452

Epoch 17:
```

```
Training loss: 0.4744, Training accuracy: 0.8292
Validation loss: 1.2977, Validation accuracy: 0.6468

Epoch 18:
Training loss: 0.4565, Training accuracy: 0.8365
Validation loss: 1.3221, Validation accuracy: 0.6458

Epoch 19:
Training loss: 0.4412, Training accuracy: 0.8414
Validation loss: 1.4556, Validation accuracy: 0.6382

Epoch 20:
Training loss: 0.4235, Training accuracy: 0.8481
Validation loss: 1.5195, Validation accuracy: 0.6318

Epoch 21:
Training loss: 0.4129, Training accuracy: 0.8513
Validation loss: 1.5120, Validation accuracy: 0.6432

Epoch 22:
Training loss: 0.3956, Training accuracy: 0.8586
Validation loss: 1.5268, Validation accuracy: 0.6270

Epoch 23:
Training loss: 0.3856, Training accuracy: 0.8616
Validation loss: 1.6055, Validation accuracy: 0.6316

Epoch 24:
Training loss: 0.3657, Training accuracy: 0.8709
Validation loss: 1.7347, Validation accuracy: 0.6336

Epoch 25:
Training loss: 0.3621, Training accuracy: 0.8700
Validation loss: 1.6591, Validation accuracy: 0.6300

Epoch 26:
Training loss: 0.3598, Training accuracy: 0.8712
Validation loss: 1.7372, Validation accuracy: 0.6390

Epoch 27:
Training loss: 0.3449, Training accuracy: 0.8761
Validation loss: 1.7351, Validation accuracy: 0.6188

Epoch 28:
Training loss: 0.3367, Training accuracy: 0.8798
Validation loss: 1.9336, Validation accuracy: 0.6136

Epoch 29:
```

```
Training loss: 0.3397, Training accuracy: 0.8795
Validation loss: 1.8522, Validation accuracy: 0.6248

Best validation accuracy: 0.6558
```

2 (b)

[6]:
```
##################
# YOUR CODE HERE #
##################

# load trained model weight
net = SimpleCIFAR10Classifier().cuda()
net.load_state_dict(torch.load(os.path.join(CHECKPOINT_FOLDER, 'best_model.
 ↪bin'))['state_dict'])
```

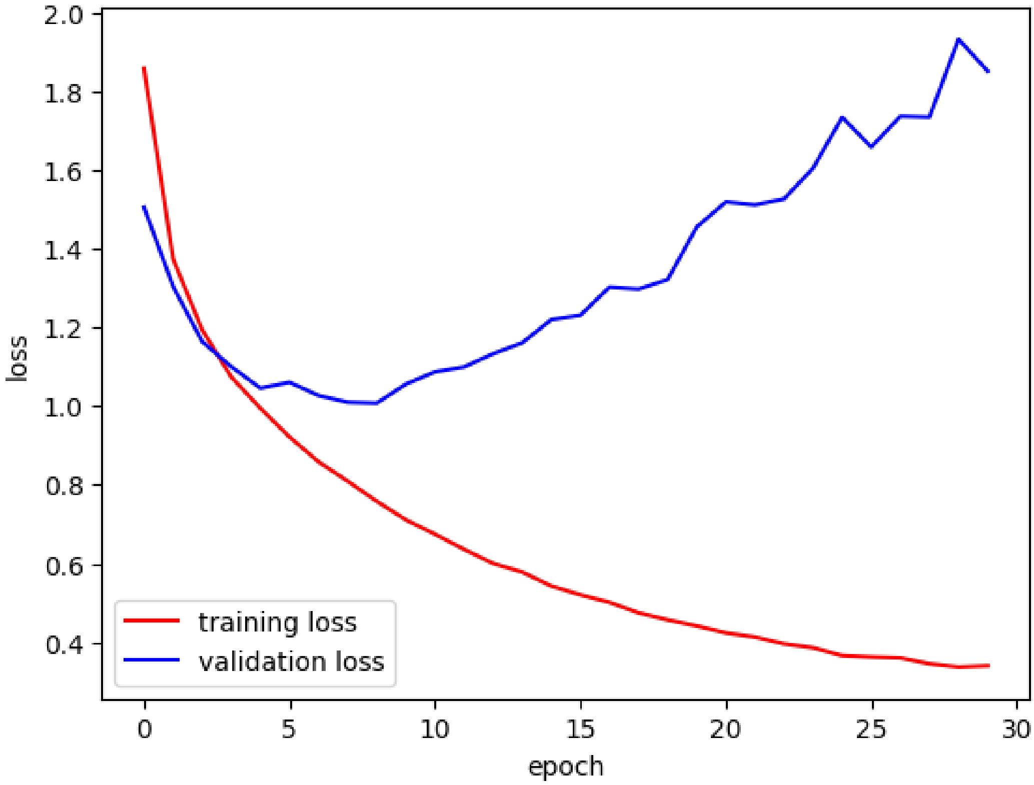
[6]:
```
<All keys matched successfully>
```

[7]:
```
##################
# YOUR CODE HERE #
##################

# write another loop to evaluate trained model performance on the test split
net.eval()
total_examples = 0
correct_examples = 0
with torch.no_grad():
    for batch_idx, (inputs, targets) in enumerate(test_loader):
        inputs, targets = inputs.cuda(), targets.cuda()
        outputs = net(inputs)
        _, predictions = torch.max(outputs, 1)
        correct_examples += (predictions == targets).sum().item()
        total_examples += inputs.shape[0]

avg_acc = correct_examples / total_examples
print("Testing accuracy: %.4f" % (avg_acc))
```

```
Testing accuracy: 0.6512
```

[8]:
```
##################
# YOUR CODE HERE #
##################

# visualize model loss curves (both train and loss)
plt.plot(range(0, EPOCHS), train_losses, color='r')
plt.plot(range(0, EPOCHS), val_losses, color='b')
plt.xlabel('epoch')
```

```
plt.ylabel('loss')
plt.legend(['training loss', 'validation loss'])
plt.show()
```



As can be seen from the above figure, when the epoch increases, the training loss decreases, while the validation loss decreases and then increases, which means that this model is overfitting. By comparing training accuracy and testing accuracy, we can also draw the conclusion of overfitting.

3 (C)

3.0.1 L1 Regularization

[9]:
```
INITIAL_LR = 0.01
net = SimpleCIFAR10Classifier().cuda()
MOMENTUM = 0.9
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=INITIAL_LR, momentum=MOMENTUM)

EPOCHS = 30
CHECKPOINT_FOLDER = "./saved_model"
```

```
#################
# YOUR CODE HERE #
#################

# set your own L1 regularization weight
REG = 1e-4

# write training loops with L1 regularization and validation loops (Hint:␣
↪similar to (a))
best_val_acc = 0

# Training Loop
train_losses = []
val_losses = []
for i in range(0, EPOCHS):

    net.train()

    print("Epoch %d:" %i)

    total_examples = 0
    correct_examples = 0

    # Record training loss
    train_loss = 0

    # Looping through training loader
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        # Send input and target to device
        inputs, targets = inputs.cuda(), targets.cuda()
        # compute the model output logits and training loss
        outputs = net(inputs)

        # calculate L1 term
        L1_term = torch.tensor(0.).cuda()
        for name, weights in net.named_parameters():
            if 'bias' not in name:
                L1_term += torch.norm(weights, 1)

        CE_term = criterion(outputs, targets)
        loss = CE_term + REG * L1_term

        # back propogation & optimizer update parametes
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # calculate predictions
```

```
        _, predictions = torch.max(outputs, 1)
        correct_examples += (predictions == targets).sum().item()
        total_examples += inputs.shape[0]
        train_loss += loss.cpu().detach().numpy()

    # calculate average training loss and accuracy
    avg_loss = train_loss / len(train_loader)
    avg_acc = correct_examples / total_examples
    print("Training loss: %.4f, Training accuracy: %.4f" %(avg_loss, avg_acc))
    train_losses.append(avg_loss)


    # Evaluate the validation set performance
    net.eval()

    total_examples = 0
    correct_examples = 0

    # Record validation loss
    val_loss = 0

    # disable gradient during validation, which can save GPU memory
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(val_loader):
            # Send input and target to device
            inputs, targets = inputs.cuda(), targets.cuda()
            # compute the model output logits and training loss
            outputs = net(inputs)
            loss = criterion(outputs, targets)
            # count the number of correctly predicted samples in the current␣
↪batch
            _, predictions = torch.max(outputs, 1)
            correct_examples += (predictions == targets).sum().item()
            total_examples += inputs.shape[0]
            val_loss += loss.cpu().detach().numpy()

    # calculate average validation loss and accuracy
    avg_loss = val_loss / len(val_loader)
    avg_acc = correct_examples / total_examples
    print("Validation loss: %.4f, Validation accuracy: %.4f" % (avg_loss,␣
↪avg_acc))
    val_losses.append(avg_loss)

    # save the model checkpoint
    current_learning_rate = optimizer.state_dict()['param_groups'][0]['lr']
    if avg_acc > best_val_acc:
        best_val_acc = avg_acc
```

```python
        if not os.path.exists(CHECKPOINT_FOLDER):
            os.makedirs(CHECKPOINT_FOLDER)
        print("Saving ...")
        state = {'state_dict': net.state_dict(),
                 'epoch': i,
                 'lr': current_learning_rate}
        torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'best_model_L1.bin'))

    print('')

print(f"Best validation accuracy: {best_val_acc:.4f}")
```

```
Epoch 0:
Training loss: 2.0072, Training accuracy: 0.3284
Validation loss: 1.5097, Validation accuracy: 0.4496
Saving …

Epoch 1:
Training loss: 1.5350, Training accuracy: 0.5071
Validation loss: 1.2679, Validation accuracy: 0.5592
Saving …

Epoch 2:
Training loss: 1.3480, Training accuracy: 0.5849
Validation loss: 1.1599, Validation accuracy: 0.5884
Saving …

Epoch 3:
Training loss: 1.2504, Training accuracy: 0.6189
Validation loss: 1.1450, Validation accuracy: 0.5996
Saving …

Epoch 4:
Training loss: 1.1912, Training accuracy: 0.6475
Validation loss: 1.0420, Validation accuracy: 0.6408
Saving …

Epoch 5:
Training loss: 1.1346, Training accuracy: 0.6691
Validation loss: 1.0064, Validation accuracy: 0.6468
Saving …

Epoch 6:
Training loss: 1.1016, Training accuracy: 0.6867
Validation loss: 0.9673, Validation accuracy: 0.6588
Saving …
```

```
Epoch 7:
Training loss: 1.0705, Training accuracy: 0.7016
Validation loss: 0.9538, Validation accuracy: 0.6688
Saving …

Epoch 8:
Training loss: 1.0444, Training accuracy: 0.7139
Validation loss: 0.9571, Validation accuracy: 0.6774
Saving …

Epoch 9:
Training loss: 1.0288, Training accuracy: 0.7232
Validation loss: 0.9681, Validation accuracy: 0.6638

Epoch 10:
Training loss: 1.0009, Training accuracy: 0.7353
Validation loss: 0.9956, Validation accuracy: 0.6660

Epoch 11:
Training loss: 0.9881, Training accuracy: 0.7441
Validation loss: 0.9653, Validation accuracy: 0.6730

Epoch 12:
Training loss: 0.9785, Training accuracy: 0.7505
Validation loss: 0.9690, Validation accuracy: 0.6764

Epoch 13:
Training loss: 0.9592, Training accuracy: 0.7598
Validation loss: 0.9990, Validation accuracy: 0.6778
Saving …

Epoch 14:
Training loss: 0.9555, Training accuracy: 0.7657
Validation loss: 1.0464, Validation accuracy: 0.6626

Epoch 15:
Training loss: 0.9368, Training accuracy: 0.7732
Validation loss: 0.9986, Validation accuracy: 0.6748

Epoch 16:
Training loss: 0.9342, Training accuracy: 0.7759
Validation loss: 1.0015, Validation accuracy: 0.6704

Epoch 17:
Training loss: 0.9355, Training accuracy: 0.7780
Validation loss: 1.0244, Validation accuracy: 0.6646

Epoch 18:
```

```
Training loss: 0.9214, Training accuracy: 0.7855
Validation loss: 0.9922, Validation accuracy: 0.6860
Saving …

Epoch 19:
Training loss: 0.9120, Training accuracy: 0.7910
Validation loss: 1.0662, Validation accuracy: 0.6568

Epoch 20:
Training loss: 0.9062, Training accuracy: 0.7940
Validation loss: 1.0720, Validation accuracy: 0.6708

Epoch 21:
Training loss: 0.9025, Training accuracy: 0.7981
Validation loss: 1.0943, Validation accuracy: 0.6542

Epoch 22:
Training loss: 0.8958, Training accuracy: 0.8027
Validation loss: 1.0644, Validation accuracy: 0.6666

Epoch 23:
Training loss: 0.8971, Training accuracy: 0.8031
Validation loss: 1.0926, Validation accuracy: 0.6724

Epoch 24:
Training loss: 0.8991, Training accuracy: 0.8035
Validation loss: 1.1183, Validation accuracy: 0.6638

Epoch 25:
Training loss: 0.8935, Training accuracy: 0.8084
Validation loss: 1.1439, Validation accuracy: 0.6608

Epoch 26:
Training loss: 0.8869, Training accuracy: 0.8146
Validation loss: 1.1514, Validation accuracy: 0.6614

Epoch 27:
Training loss: 0.8864, Training accuracy: 0.8167
Validation loss: 1.1763, Validation accuracy: 0.6630

Epoch 28:
Training loss: 0.8788, Training accuracy: 0.8192
Validation loss: 1.1614, Validation accuracy: 0.6708

Epoch 29:
Training loss: 0.8819, Training accuracy: 0.8204
Validation loss: 1.1843, Validation accuracy: 0.6654
```

```
Best validation accuracy: 0.6860
```

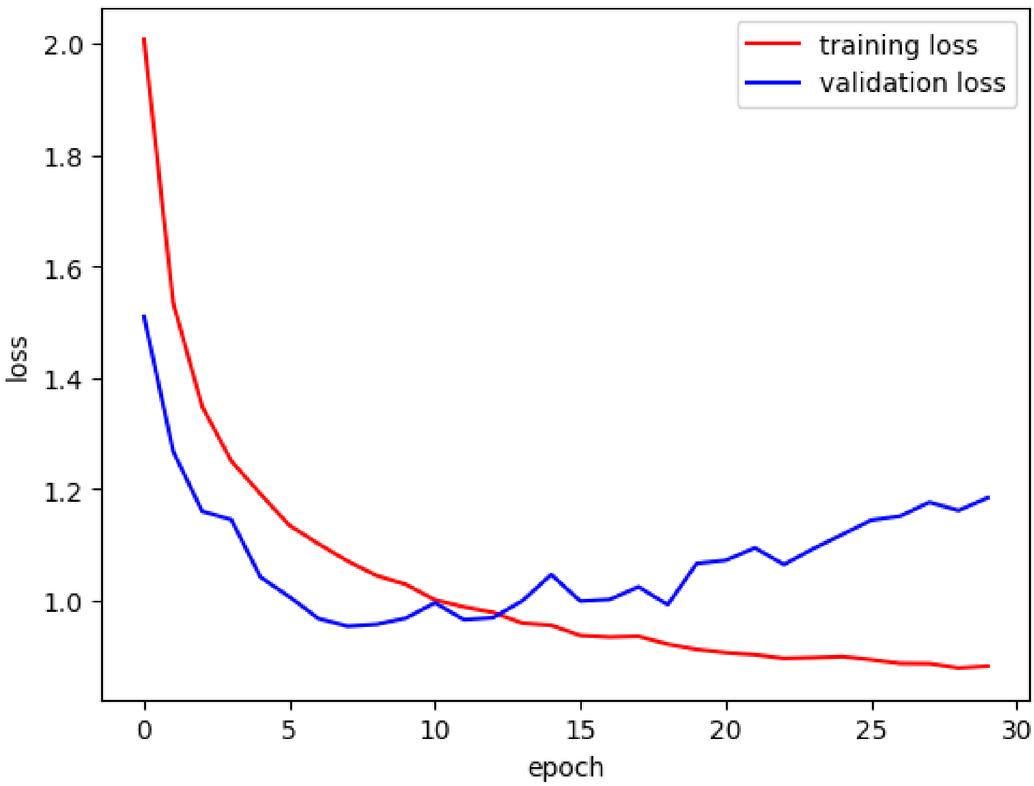
[10]:
```python
##################
# YOUR CODE HERE #
##################

# load trained model weight
net = SimpleCIFAR10Classifier().cuda()
net.load_state_dict(torch.load(os.path.join(CHECKPOINT_FOLDER, 'best_model_L1.
↪bin'))['state_dict'])
```

[10]:
```
<All keys matched successfully>
```

[11]:
```python
##################
# YOUR CODE HERE #
##################

# write another loop to evaluate trained model performance on the test split
net.eval()
total_examples = 0
correct_examples = 0
with torch.no_grad():
    for batch_idx, (inputs, targets) in enumerate(test_loader):
        inputs, targets = inputs.cuda(), targets.cuda()
        outputs = net(inputs)
        _, predictions = torch.max(outputs, 1)
        correct_examples += (predictions == targets).sum().item()
        total_examples += inputs.shape[0]

avg_acc = correct_examples / total_examples
print("Testing accuracy: %.4f" % (avg_acc))
```

```
Testing accuracy: 0.6844
```

[12]:
```python
##################
# YOUR CODE HERE #
##################

# visualize model loss curves (both train and loss)
plt.plot(range(0, EPOCHS), train_losses, color='r')
plt.plot(range(0, EPOCHS), val_losses, color='b')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['training loss', 'validation loss'])
plt.show()
```

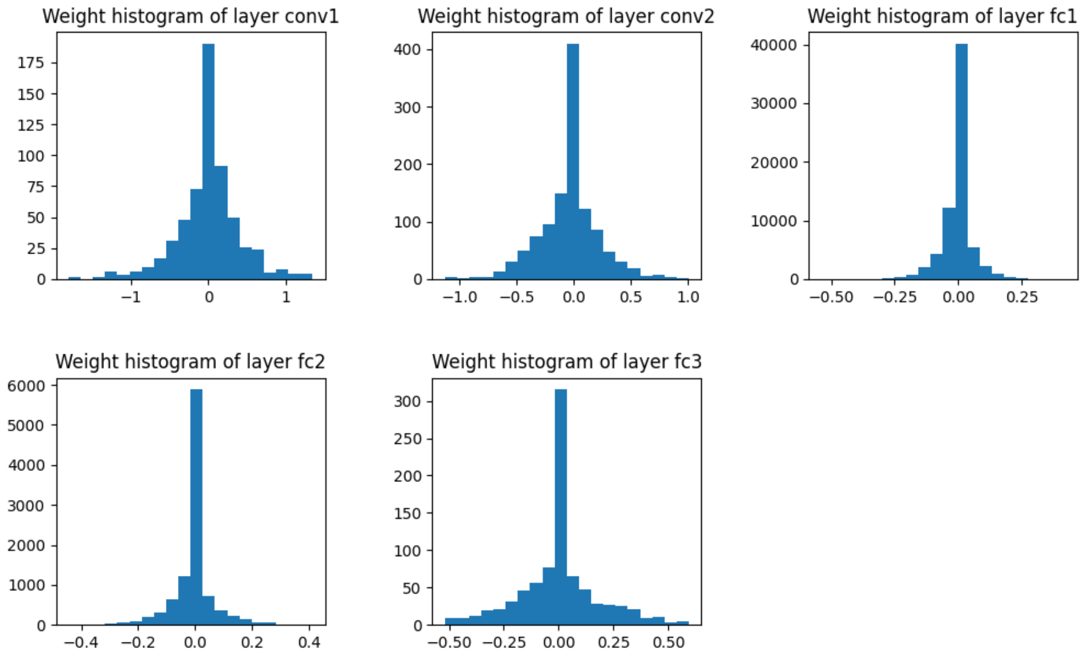


The figure above shows that this model is less overfitting than the model without L1 regularization. And the testing accuracy of this model (0.6844) is larger than the model without L1 regularization (0.6512). Therefore, this model is better.

3.0.2 Visualize the model weights

[13]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(12,7))
plt.subplots_adjust(wspace = 0.4, hspace = 0.4)
plt_i = 0
for name, module in net.named_modules():
    if 'conv' in name or 'fc' in name:
        ##################
        # YOUR CODE HERE #
        ##################

        # extract weight from layers
        weight = module.weight.cpu().detach().numpy().flatten()

        # Visualize the weights
        plt_i += 1
```

```
        plt.subplot(230 + plt_i)
        _ = plt.hist(weight, bins=20)
        plt.title("Weight histogram of layer " + name)
plt.show()
```



4 (d)

4.0.1 L2 Regularization

[14]:

```
INITIAL_LR = 0.01
net = SimpleCIFAR10Classifier().cuda()
MOMENTUM = 0.9
REG = 1e-3
criterion = nn.CrossEntropyLoss()
EPOCHS = 30
CHECKPOINT_FOLDER = "./saved_model"

##################
# YOUR CODE HERE #
##################

# set your own L2 regularization weight
REG = 1e-3
```

```python
optimizer = optim.SGD(net.parameters(), lr=INITIAL_LR, momentum=MOMENTUM,␣
 ↪weight_decay=REG)

# write training loops with L2 regularization and validation loops (Hint:␣
 ↪similar to (a))
best_val_acc = 0

# Training Loop
train_losses = []
val_losses = []
for i in range(0, EPOCHS):

    net.train()

    print("Epoch %d:" %i)

    total_examples = 0
    correct_examples = 0

    # Record training loss
    train_loss = 0

    # Looping through training loader
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        # Send input and target to device
        inputs, targets = inputs.cuda(), targets.cuda()
        # compute the model output logits and training loss
        outputs = net(inputs)
        loss = criterion(outputs, targets)
        # back propogation & optimizer update parametes
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # calculate predictions
        _, predictions = torch.max(outputs, 1)
        correct_examples += (predictions == targets).sum().item()
        total_examples += inputs.shape[0]
        train_loss += loss.cpu().detach().numpy()

    # calculate average training loss and accuracy
    avg_loss = train_loss / len(train_loader)
    avg_acc = correct_examples / total_examples
    print("Training loss: %.4f, Training accuracy: %.4f" %(avg_loss, avg_acc))
    train_losses.append(avg_loss)


    # Evaluate the validation set performance
```

```
    net.eval()

    total_examples = 0
    correct_examples = 0

    # Record validation loss
    val_loss = 0

    # disable gradient during validation, which can save GPU memory
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(val_loader):
            # Send input and target to device
            inputs, targets = inputs.cuda(), targets.cuda()
            # compute the model output logits and training loss
            outputs = net(inputs)
            loss = criterion(outputs, targets)
            # count the number of correctly predicted samples in the current␣
↪batch
            _, predictions = torch.max(outputs, 1)
            correct_examples += (predictions == targets).sum().item()
            total_examples += inputs.shape[0]
            val_loss += loss.cpu().detach().numpy()

    # calculate average validation loss and accuracy
    avg_loss = val_loss / len(val_loader)
    avg_acc = correct_examples / total_examples
    print("Validation loss: %.4f, Validation accuracy: %.4f" % (avg_loss,␣
↪avg_acc))
    val_losses.append(avg_loss)

    # save the model checkpoint
    current_learning_rate = optimizer.state_dict()['param_groups'][0]['lr']
    if avg_acc > best_val_acc:
        best_val_acc = avg_acc

        if not os.path.exists(CHECKPOINT_FOLDER):
           os.makedirs(CHECKPOINT_FOLDER)
        print("Saving ...")
        state = {'state_dict': net.state_dict(),
                 'epoch': i,
                 'lr': current_learning_rate}
        torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'best_model_L2.bin'))

    print('')

print(f"Best validation accuracy: {best_val_acc:.4f}")
```

```
Epoch 0:
Training loss: 1.8003, Training accuracy: 0.3377
Validation loss: 1.4931, Validation accuracy: 0.4670
Saving …

Epoch 1:
Training loss: 1.3692, Training accuracy: 0.5064
Validation loss: 1.3354, Validation accuracy: 0.5248
Saving …

Epoch 2:
Training loss: 1.1990, Training accuracy: 0.5730
Validation loss: 1.1813, Validation accuracy: 0.5764
Saving …

Epoch 3:
Training loss: 1.0901, Training accuracy: 0.6149
Validation loss: 1.0992, Validation accuracy: 0.6090
Saving …

Epoch 4:
Training loss: 1.0045, Training accuracy: 0.6429
Validation loss: 1.0363, Validation accuracy: 0.6386
Saving …

Epoch 5:
Training loss: 0.9417, Training accuracy: 0.6660
Validation loss: 1.0312, Validation accuracy: 0.6464
Saving …

Epoch 6:
Training loss: 0.8864, Training accuracy: 0.6877
Validation loss: 1.0074, Validation accuracy: 0.6474
Saving …

Epoch 7:
Training loss: 0.8403, Training accuracy: 0.7028
Validation loss: 1.0510, Validation accuracy: 0.6424

Epoch 8:
Training loss: 0.7966, Training accuracy: 0.7190
Validation loss: 1.0413, Validation accuracy: 0.6488
Saving …

Epoch 9:
Training loss: 0.7618, Training accuracy: 0.7339
Validation loss: 1.0218, Validation accuracy: 0.6522
Saving …
```

```
Epoch 10:
Training loss: 0.7266, Training accuracy: 0.7451
Validation loss: 1.0391, Validation accuracy: 0.6550
Saving …

Epoch 11:
Training loss: 0.7070, Training accuracy: 0.7504
Validation loss: 1.0023, Validation accuracy: 0.6642
Saving …

Epoch 12:
Training loss: 0.6698, Training accuracy: 0.7624
Validation loss: 1.0192, Validation accuracy: 0.6600

Epoch 13:
Training loss: 0.6457, Training accuracy: 0.7699
Validation loss: 1.0219, Validation accuracy: 0.6608

Epoch 14:
Training loss: 0.6316, Training accuracy: 0.7766
Validation loss: 1.0383, Validation accuracy: 0.6626

Epoch 15:
Training loss: 0.6154, Training accuracy: 0.7820
Validation loss: 1.0313, Validation accuracy: 0.6638

Epoch 16:
Training loss: 0.5845, Training accuracy: 0.7917
Validation loss: 1.0596, Validation accuracy: 0.6676
Saving …

Epoch 17:
Training loss: 0.5693, Training accuracy: 0.7968
Validation loss: 1.1095, Validation accuracy: 0.6494

Epoch 18:
Training loss: 0.5504, Training accuracy: 0.8042
Validation loss: 1.0569, Validation accuracy: 0.6740
Saving …

Epoch 19:
Training loss: 0.5445, Training accuracy: 0.8053
Validation loss: 1.0697, Validation accuracy: 0.6554

Epoch 20:
Training loss: 0.5247, Training accuracy: 0.8130
Validation loss: 1.0893, Validation accuracy: 0.6612
```

```
Epoch 21:
Training loss: 0.5156, Training accuracy: 0.8154
Validation loss: 1.0968, Validation accuracy: 0.6624

Epoch 22:
Training loss: 0.4984, Training accuracy: 0.8213
Validation loss: 1.1513, Validation accuracy: 0.6476

Epoch 23:
Training loss: 0.4988, Training accuracy: 0.8212
Validation loss: 1.1073, Validation accuracy: 0.6726

Epoch 24:
Training loss: 0.4839, Training accuracy: 0.8281
Validation loss: 1.1281, Validation accuracy: 0.6596

Epoch 25:
Training loss: 0.4690, Training accuracy: 0.8325
Validation loss: 1.1648, Validation accuracy: 0.6530

Epoch 26:
Training loss: 0.4645, Training accuracy: 0.8341
Validation loss: 1.2120, Validation accuracy: 0.6484

Epoch 27:
Training loss: 0.4596, Training accuracy: 0.8361
Validation loss: 1.1712, Validation accuracy: 0.6542

Epoch 28:
Training loss: 0.4459, Training accuracy: 0.8419
Validation loss: 1.2016, Validation accuracy: 0.6516

Epoch 29:
Training loss: 0.4470, Training accuracy: 0.8407
Validation loss: 1.1980, Validation accuracy: 0.6500

Best validation accuracy: 0.6740
```

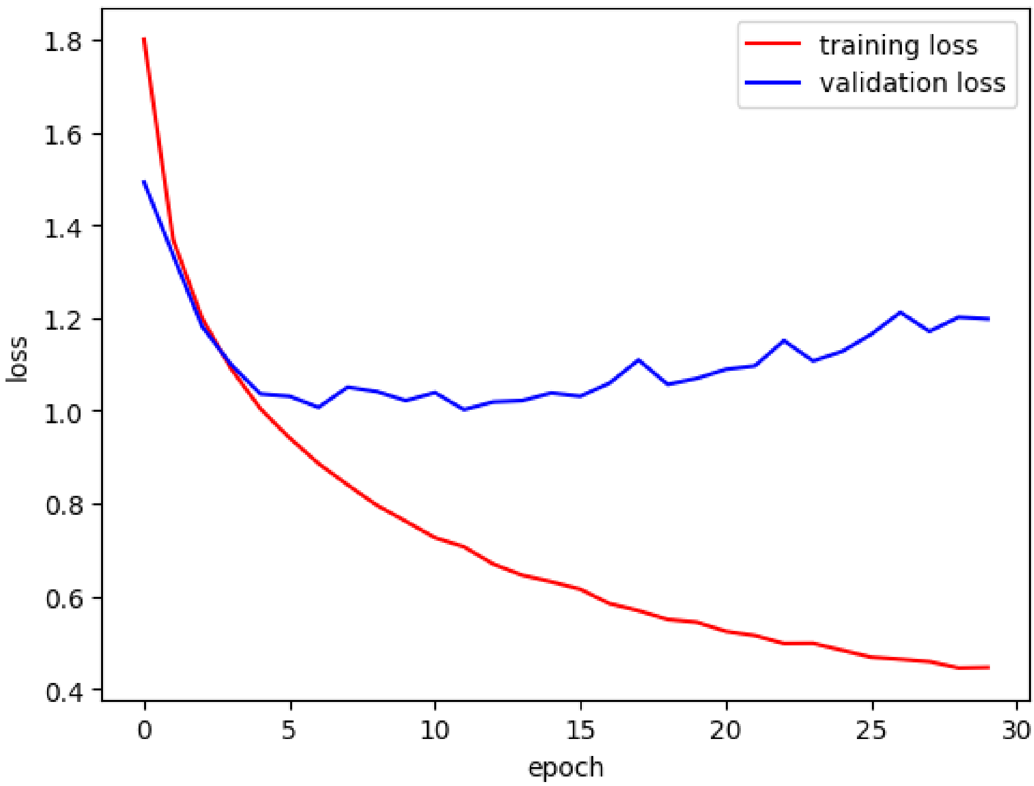
[15]:
```
##################
# YOUR CODE HERE #
##################

# load trained model weight
net = SimpleCIFAR10Classifier().cuda()
net.load_state_dict(torch.load(os.path.join(CHECKPOINT_FOLDER, 'best_model_L2.
 ↪bin'))['state_dict'])
```

[15]: <All keys matched successfully>

[16]:
```
##################
# YOUR CODE HERE #
##################

# write another loop to evaluate trained model performance on the test split
net.eval()
total_examples = 0
correct_examples = 0
with torch.no_grad():
    for batch_idx, (inputs, targets) in enumerate(test_loader):
        inputs, targets = inputs.cuda(), targets.cuda()
        outputs = net(inputs)
        _, predictions = torch.max(outputs, 1)
        correct_examples += (predictions == targets).sum().item()
        total_examples += inputs.shape[0]

avg_acc = correct_examples / total_examples
print("Testing accuracy: %.4f" % (avg_acc))
```

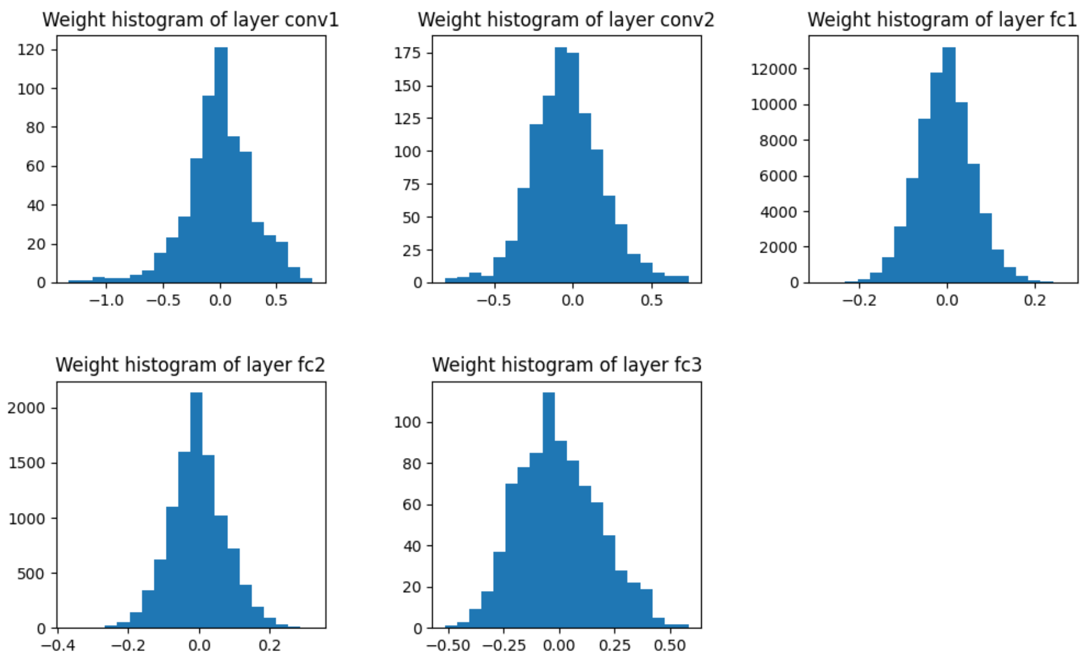
Testing accuracy: 0.6702

[17]:
```
##################
# YOUR CODE HERE #
##################

# visualize model loss curves (both train and loss)
plt.plot(range(0, EPOCHS), train_losses, color='r')
plt.plot(range(0, EPOCHS), val_losses, color='b')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['training loss', 'validation loss'])
plt.show()
```



4.0.2 Visualize the model weights

[18]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(12,7))
plt.subplots_adjust(wspace = 0.4, hspace = 0.4)
plt_i = 0
for name, module in net.named_modules():
    if 'conv' in name or 'fc' in name:
        ##################
        # YOUR CODE HERE #
        ##################

        # extract weight from layers
        weight = module.weight.cpu().detach().numpy().flatten()

        # Visualize the weights
        plt_i += 1
        plt.subplot(230 + plt_i)
        _ = plt.hist(weight, bins=20)
        plt.title("Weight histogram of layer "+name)
plt.show()
```



5 (e)

5.0.1 comment on the differences between L1 and L2 regularization.

The weight histograms for L1 regularization exhibit sharp spikes at zero, while the weights for L2 regularization are more evenly spread around zero.

Effects on model weights: L1 regularization penalizes the absolute value of the weights, which compels the model to be sparse; while L2 penalizes the square of the weights, which encourages tiny weights but doesn't impose sparsity.

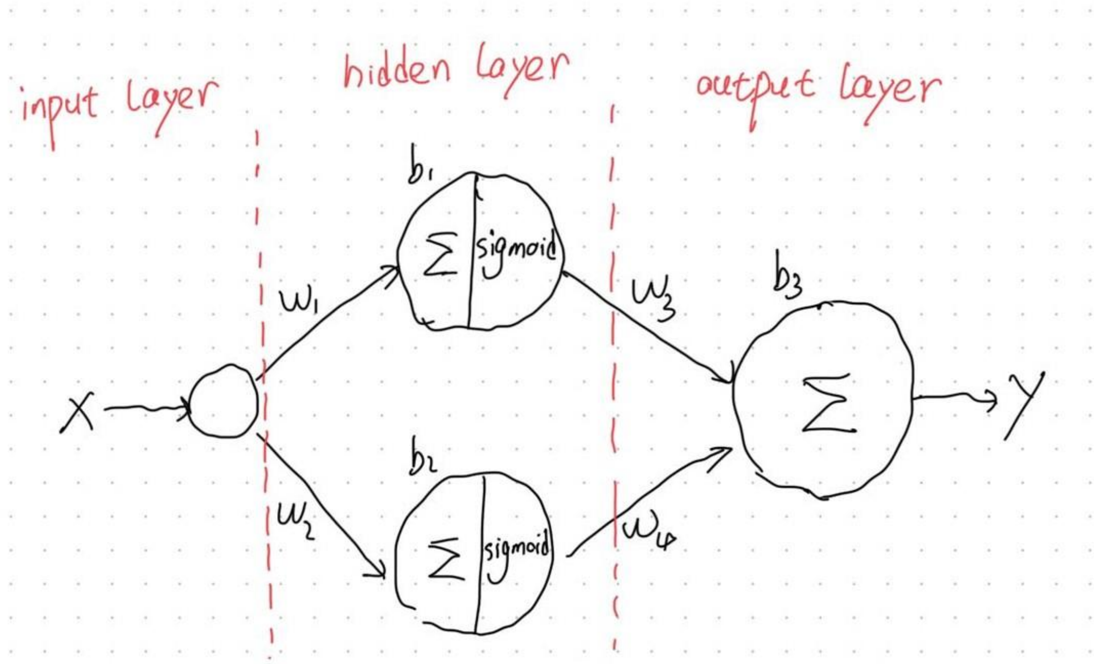
Effects on model performance: L1 Regularization sets many weights to zero, which lead to a simpler model. It works well if the dataset has irrelevant or redundant features. In comparison, L2 Regularization keeps more features, potentially improving model performance; nevertheless, failure to properly adjust the REG increases the danger of overfitting.

Q3

3.1

(a)

The NN is implemented as below:



The weight and bias values are designed as below:

$$\boldsymbol{w_1 = 500, w_2 = -500, w_3 = 1, w_4 = 1}$$

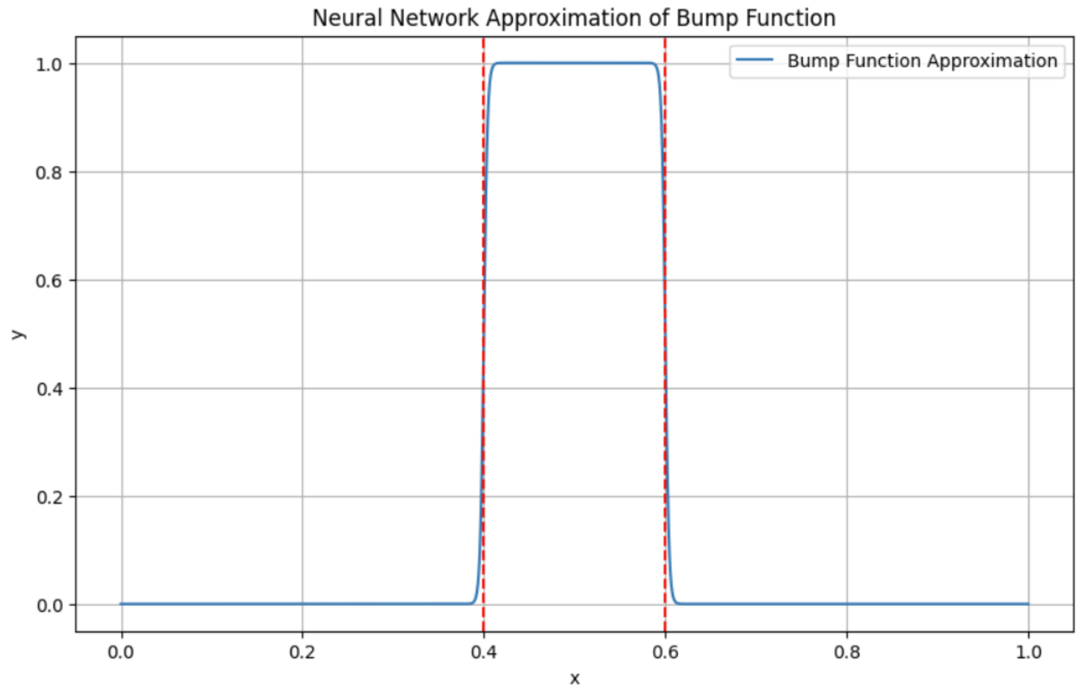
$$\boldsymbol{b_1 = -0.4w_1 = -200, b_2 = -0.6w_2 = 300, b_3 = -1}$$

Therefore, the approximated bump function is derived as:

$$y = w_3 sigmoid(w_1 x + b_1) + w_4 sigmoid(w_2 x + b_2) + b_3$$

$$\boldsymbol{y = sigmoid(500x - 200) + sigmoid(-500x + 300) - 1}$$

Then the approximated bump function is plotted as below:



The minimum number of hidden neurons for this approximation is 2, because at least two neurons are required to achieve a combination of a step-up function and a step-down function.

(b)

(1) The steepness of the step-up part is determined by w_1. The larger w_1 is, the steeper the step-up part is. The steepness of the step-down part is determined by w_2. The smaller w_2 is, the steeper the step-down part is.

(2) The step-up location is determined by b_1. The step-down location is determined by b_2. Below shows the relationship between b_1 and step-up location x_{up} and the relationship between b_2 and step-down location x_{down}.

$$b_1 = -x_{up} w_1, b_2 = -x_{down} w_2$$

(3) The height of the bump is determined by w_3 and w_4. The relationship

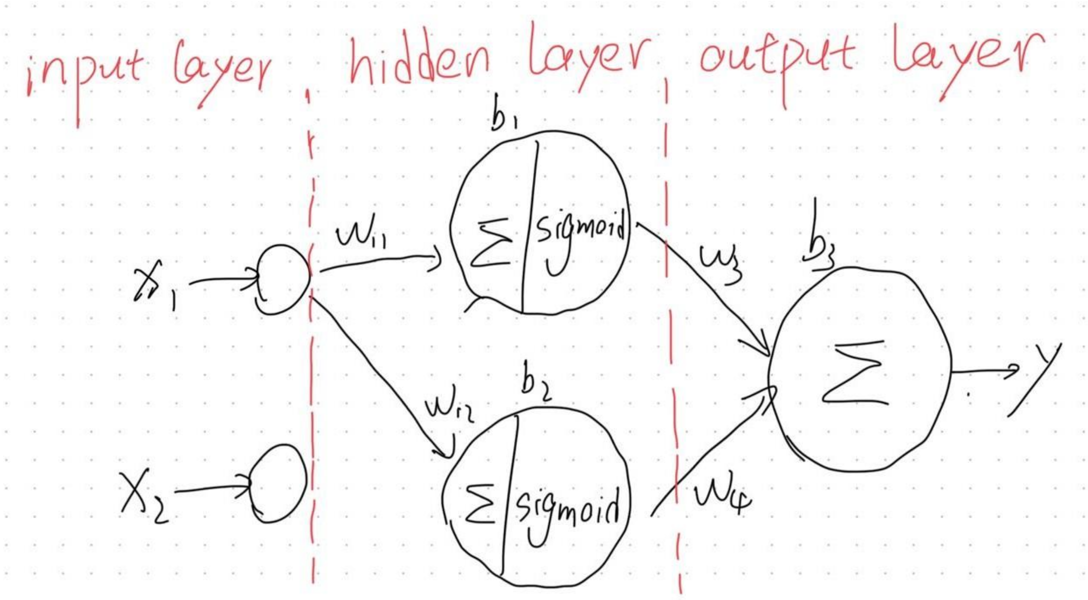
is shown below:

$$height = w_3 = w_4$$

3.2

(a)

The single layer NN with two inputs is implemented as below:



The weight and bias values are designed as below:

$$\boldsymbol{w_{11} = 500, w_{12} = -500, w_3 = 1, w_4 = 1}$$

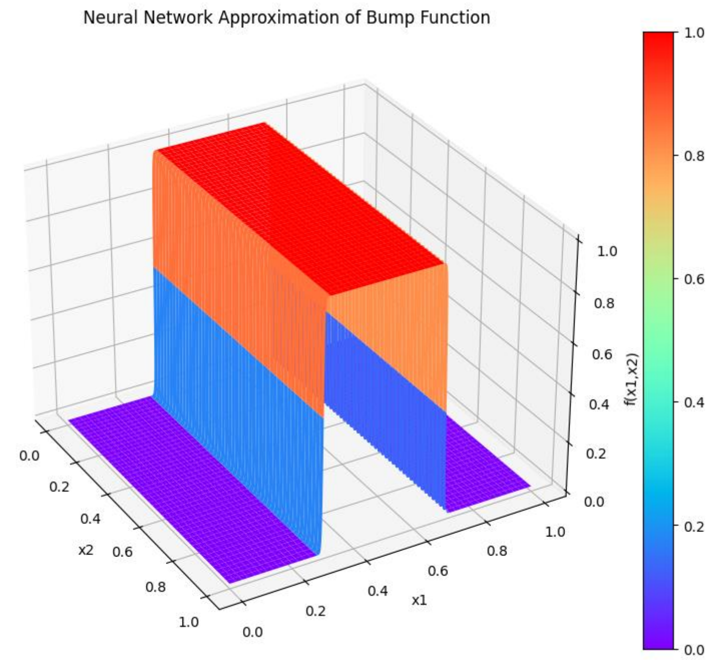
$$\boldsymbol{b_1 = -0.3w_{11} = -150, b_2 = -0.7w_{12} = 350, b_3 = -1}$$

Therefore, the approximated bump function is derived as:

$$y = w_3 sigmoid(w_{11}x_1 + b_1) + w_4 sigmoid(w_{12}x_1 + b_2) + b_3$$

$$\boldsymbol{y = sigmoid(500x_1 - 150) + sigmoid(-500x_1 + 350) - 1}$$

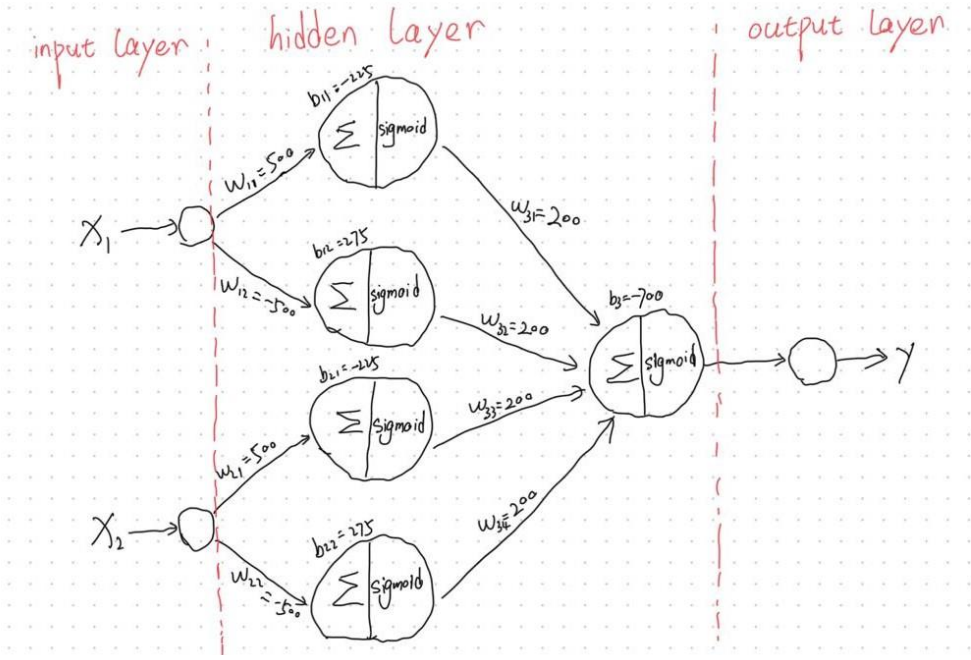
Then the approximated bump function is plotted as below:



The minimum number of hidden neurons for this approximation is 2, because at least two neurons are required to achieve a combination of a step-up function and a step-down function for x_1.

(b)

The two-layer NN with two inputs is implemented as below:

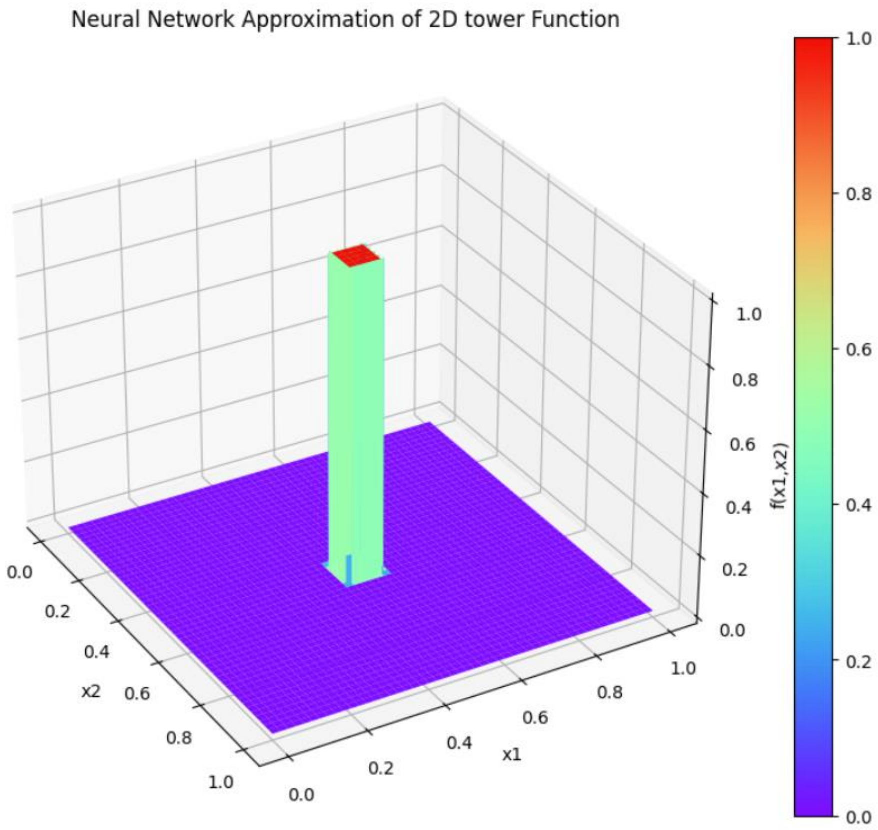


The approximated 2D tower function is derived as:

$$y = sigmoid(w_{31}sigmoid(w_{11}x_1 + b_{11}) + w_{32}sigmoid(w_{12}x_1 + b_{12}) \\ + w_{33}sigmoid(w_{21}x_2 + b_{21}) + w_{34}sigmoid(w_{22}x_2 + b_{22}) + b_3)$$

$$\boldsymbol{y = sigmoid(200sigmoid(500x_1 - 225) + 200sigmoid(-500x_1 + 275) \\ + 200sigmoid(500x_2 - 225) + 200sigmoid(-500x_2 + 275) - 700)}$$

The approximated 2D tower function is plotted as below:



The minimum number of hidden neurons for this approximation is 5, two for x_1 direction bump in the first hidden layer, two for x_2 direction bump in the first hidden layer, and one for the second hidden layer.

(c)

In the worst case, the gradient of $f(x_1, x_2)$ for both directions are always t, and the maximum error for each tower function, ϵ is the height change of $f(x_1, x_2)$ over its base square, which can be derived as below:

$$\epsilon = st$$

Where s is the base square size of each tower function.

Therefore, the area of base square, s^2 is calculated as:

$$s^2 = \left(\frac{\epsilon}{t}\right)^2$$

To fill the 2D unit square with these tower functions, the number of tower functions is determined as:

$$N = \frac{unit\ square\ area}{base\ square\ area} = \frac{1}{s^2} = \left(\frac{t}{\epsilon}\right)^2 \tag{1}$$

Therefore, the minimum number of tower functions that can guarantee to make such an approximation for all possible function $\boldsymbol{f}$ that satisfies the conditions is $\left(\frac{t}{\epsilon}\right)^2$.

In addition, according to part b and equation (1), the relationship between the gradient limit $\boldsymbol{t}$, error bound $\boldsymbol{\epsilon}$, and the total required hidden neuron number $\boldsymbol{N_{hidden}}$ is shown below:

$$\boldsymbol{N_{hidden} \propto N \rightarrow N_{hidden} \propto \left(\frac{t}{\epsilon}\right)^2}$$