# simplenn-cifar10-dev

October 3, 2023

## 1 Training SimpleNN on CIFAR-10

In this project, you will use the SimpleNN model to perform image classification on CIFAR-10. CIFAR-10 orginally contains 60K images from 10 categories. We split it into 45K/5K/10K images to serve as train/valiation/test set. We only release the ground-truth labels of training/validation dataset to you.

### 1.1 Step 0: Set up the SimpleNN model

As you have practiced to implement simple neural networks in Homework 1, we just prepare the implementation for you.

[2]:
```
# import necessary dependencies
import argparse
import os, sys
import time
import datetime
from tqdm import tqdm_notebook as tqdm

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

[3]:
```
def my_swish(x):
    return torch.sigmoid(x) * x
## implementing swish

# define the SimpleNN mode;
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.conv1_bn = nn.BatchNorm2d(num_features=8, eps=1e-05, momentum=0.1)
        ## adds first batch normalisation layer
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.conv2_bn = nn.BatchNorm2d(num_features=16, eps=1e-05, momentum=0.1)
        ## adds second batch normalisation layer
        self.fc1   = nn.Linear(16*6*6, 120)
```

```
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        # out = F.relu(self.conv1_bn(self.conv1(x)))  # relu(BN(conv))
        # out = F.relu(self.conv1(x))  # relu(conv)

        ## applies batch normalisation layer and swish
        out = my_swish(self.conv1_bn(self.conv1(x)))
        out = F.max_pool2d(out, 2)

        # out = F.relu(self.conv2_bn(self.conv2(out)))  # relu(BN(conv))
        # out = F.relu(self.conv2(out))  # relu(conv)

        ## applies batch normalisation layer and swish
        out = my_swish(self.conv2_bn(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
```

#### 1.1.1 Question (a)

Here is a sanity check to verify the implementation of SimpleNN. You need to: 1. Write down your code. 2. **In the PDF report**, give a brief description on how the code helps you know that SimpleNN is implemented correctly.

[4]:
```
##############################################
# your code here
# sanity check for the correctness of SimpleNN
test_in = torch.rand((1, 3, 32, 32))
test_model = SimpleNN()
test_out = test_model.forward(test_in)
print(test_out.shape)
##############################################
```

```
torch.Size([1, 10])
```

### 1.2 Step 1: Set up preprocessing functions

Preprocessing is very important as discussed in the lecture. You will need to write preprocessing functions with the help of *torchvision.transforms* in this step. You can find helpful tutorial/API at here.

### 1.2.1 Question (b)

For the question, you need to: 1. Complete the preprocessing code below. 2. **In the PDF report**, briefly describe what preprocessing operations you used and what are the purposes of them.

Hint: 1. Only two operations are necessary to complete the basic preprocessing here. 2. The raw input read from the dataset will be PIL images. 3. Data augmentation operations are not mendatory, but feel free to incorporate them if you want. 4. Reference value for mean/std of CIFAR-10 images (assuming the pixel values are within [0,1]): mean (RGB-format): (0.4914, 0.4822, 0.4465), std (RGB-format): (0.2023, 0.1994, 0.2010)

[5]:
```
# useful libraries
import torchvision
import torchvision.transforms as transforms

################################################
# your code here
# specify preprocessing function ToTensor (PIL to tensor) and Normalise (mean␣
 ↪becomes 0, stdev becomes 1)
# please see pdf for more details

transform_train =  transforms.Compose([
    transforms.ToTensor(),  ## data augmentation below for training only
    transforms.RandomCrop(size=(32, 32), padding=4, pad_if_needed=False,␣
 ↪padding_mode='edge'),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    ])

# validation set: no data augmentation
transform_val = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    ])
################################################
```

## 1.3 Step 2: Set up dataset and dataloader

### 1.3.1 Question (c)

Set up the train/val datasets and dataloders that are to be used during the training. Check out the official API for more information about **torch.utils.data.DataLoader**.

Here, you need to: 1. Complete the code below.

[6]:
```
# do NOT change these
from tools.dataset import CIFAR10
from torch.utils.data import DataLoader
```

```
# a few arguments, do NOT change these
DATA_ROOT = "./data"
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100

################################################
# your code here
# construct dataset
train_set = CIFAR10(
    root=DATA_ROOT,
    mode='train',
    download=True,
    transform=transform_train    # your code
)
val_set = CIFAR10(
    root=DATA_ROOT,
    mode='val',
    download=True,
    transform=transform_val    # your code
)

# construct dataloader
train_loader = DataLoader(
    train_set,
    batch_size=TRAIN_BATCH_SIZE,  # your code
    shuffle=True,     # your code
    num_workers=4
)
val_loader = DataLoader(
    val_set,
    batch_size=VAL_BATCH_SIZE,  # your code
    shuffle=True,     # your code
    num_workers=4
)
################################################
```

```
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
```

## 1.4 Step 3: Instantiate your SimpleNN model and deploy it to GPU devices.

### 1.4.1 Question (d)

You may want to deploy your model to GPU device for efficient training. Please assign your model to GPU if possible. If you are training on a machine without GPUs, please deploy your model to CPUs.

Here, you need to: 1. Complete the code below. 2. **In the PDF report**, briefly describe how you verify that your model is indeed deployed on GPU. (Hint: check `nvidia-smi`.)

[7]:
```
# specify the device for computation
#############################################
# your code here

device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
# Construct our model by instantiating the class defined above
model = SimpleNN()
# Copy to CUDA device. This is very important.
model = model.to(device)
#############################################
```

```
cuda
```

## 1.5 Step 4: Set up the loss function and optimizer

Loss function/objective function is used to provide "feedback" for the neural networks. Typically, we use multi-class cross-entropy as the loss function for classification models. As for the optimizer, we will use SGD with momentum.

### 1.5.1 Question (e)

Here, you need to: 1. Set up the cross-entropy loss as the criterion. (Hint: there are implemented functions in **torch.nn**) 2. Specify a SGD optimizer with momentum. (Hint: there are implemented functions in **torch.optim**)

[8]:
```
import torch.nn as nn
import torch.optim as optim



##### my addition
L1_or_L2 = 'L1'
##### end of my addition



# hyperparameters, do NOT change right now
# initial learning rate
```

```
INITIAL_LR = 0.01

# momentum for optimizer
MOMENTUM = 0.9

# L2 regularization strength
REG = 1e-4  # 1e-4 is orig reg strength (L2 and L1)

################################################
# your code here
# create loss function
criterion = nn.CrossEntropyLoss()

# Add optimizer
assert MOMENTUM == 0.9, "momentum isn't 0.9"

# using L2 optimisation
if L1_or_L2 == 'L2':
    optimizer = torch.optim.SGD(model.parameters(), lr=INITIAL_LR,␣
 ↪momentum=MOMENTUM, weight_decay=REG)

# using L1 optimisation, weight_decay=0 (no L2)
if L1_or_L2 == 'L1':
    optimizer = torch.optim.SGD(model.parameters(), lr=INITIAL_LR,␣
 ↪momentum=MOMENTUM, weight_decay=0)

assert L1_or_L2 == 'L1' or L1_or_L2 == 'L2', 'L1_or_L2 should be either str L1␣
 ↪or L2'
################################################
```

## 1.6 Step 5: Start the training process.

### 1.6.1 Question (f)/(g)

Congratulations! You have completed all of the previous steps and it is time to train our neural network.

Here you need to: 1. Complete the training codes. 2. Actually perform the training.

Hint: Training a neural network usually repeats the following 4 steps:

**i) Get a batch of data from the dataloader and copy it to your device (GPU).**

**ii) Do a forward pass to get the outputs from the neural network and compute the loss. Be careful about your inputs to the loss function. Are the inputs required to be the logits or softmax probabilities?)**

**iii) Do a backward pass (back-propagation) to compute gradients of all weights with respect to the loss.**

**iiii) Update the model weights with the optimizer.**

You will also need to compute the accuracy of training/validation samples to track your model's performance over each epoch (the accuracy should be increasing as you train for more and more epochs).

[9]:
```
# some hyperparameters
# total number of training epochs
EPOCHS = 80

DECAY_EPOCHS = 4
DECAY = 0.95

# the folder where the trained model is saved
CHECKPOINT_FOLDER = "./saved_model"

# start the training/validation process
# the process should take about 5 minutes on a GTX 1070-Ti
# if the code is written efficiently.
best_val_acc = 0
current_learning_rate = INITIAL_LR




##### my addition
accuracy_arr_train = []
accuracy_arr_val = []
##### end of my addition




print("==> Training starts!")
print("="*50)
for i in range(0, EPOCHS):
    # handle the learning rate scheduler.
    if i % DECAY_EPOCHS == 0 and i != 0:
        current_learning_rate = current_learning_rate * DECAY
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_learning_rate
        print("Current learning rate has decayed to %f" %current_learning_rate)

    ########################
    # your code here
    # switch to train mode
    model.train()
```

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

    print("Epoch %d:" %i)
    # this help you compute the training accuracy
    total_examples = 0
    correct_examples = 0

    train_loss = 0 # track training loss if you want

    # Train the model for 1 epoch.
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        # if batch_idx == 0 and i == 0:
        #     print(torch.mean(inputs.flatten()))
        ###################################
        # your code here
        # copy inputs to device
        inputs, targets = inputs.to(device), targets.to(device)

        # compute the output and loss
        y_pred = model(inputs)
        loss = criterion(y_pred, targets)



        ## L1 normalisation
##### my addition
        if L1_or_L2 == 'L1':
            for name, param in model.named_parameters():
                if 'bias' not in name:
                    loss += REG * torch.sum(torch.abs(param))
##### end of my addition




        train_loss += loss

        # zero the gradient
        optimizer.zero_grad()

        # backpropagation
        loss.backward()

        # apply gradient and update the weights
        optimizer.step()

        # count the number of correctly predicted samples in the current batch
```

```
        total_examples += len(targets.view(targets.size(0), -1))
        correct_examples += sum(torch.argmax(y_pred, dim=1, keepdim=False) ==␣
↪targets)

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

    avg_loss = train_loss / len(train_loader)
    avg_acc = correct_examples / total_examples
    print("Training loss: %.4f, Training accuracy: %.4f" %(avg_loss, avg_acc))



##### my addition
    accuracy_arr_train.append(avg_acc)
##### end of my addition



    # Validate on the validation dataset
    ########################
    # your code here
    # switch to eval mode
    model.eval()

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

    # this helps you compute the validation accuracy
    total_examples = 0
    correct_examples = 0

    val_loss = 0 # again, track the validation loss if you want

    # 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
            # copy inputs to device
            inputs, targets = inputs.to(device), targets.to(device)

            # compute the output and loss
            y_pred = model(inputs)
            loss = criterion(y_pred, targets)
            val_loss += loss

            # count the number of correctly predicted samples in the current␣
↪batch
```

```
            total_examples += len(targets.view(targets.size(0), -1))
            correct_examples += sum(torch.argmax(y_pred, dim=1, keepdim=False)␣
 ↪== targets)
            ####################################

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



##### my addition
    accuracy_arr_val.append(avg_acc)
##### end of my addition



    # save the model checkpoint
    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': model.state_dict(),
        #          'epoch': i,
        #          'lr': current_learning_rate}
        # torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'simplenn-dev.pth'))

    print('')

print("="*50)
print(f"==> Optimization finished! Best validation accuracy: {best_val_acc:.
 ↪4f}")
```

```
==> Training starts!
==================================================
Epoch 0:
Training loss: 1.9695, Training accuracy: 0.3468
Validation loss: 1.4438, Validation accuracy: 0.4830

Epoch 1:
Training loss: 1.6324, Training accuracy: 0.4778
Validation loss: 1.2835, Validation accuracy: 0.5426

Epoch 2:
Training loss: 1.4969, Training accuracy: 0.5282
```

```
Validation loss: 1.1686, Validation accuracy: 0.5896

Epoch 3:
Training loss: 1.4209, Training accuracy: 0.5537
Validation loss: 1.1410, Validation accuracy: 0.5832

Current learning rate has decayed to 0.009500
Epoch 4:
Training loss: 1.3520, Training accuracy: 0.5788
Validation loss: 1.0623, Validation accuracy: 0.6254

Epoch 5:
Training loss: 1.3077, Training accuracy: 0.5946
Validation loss: 1.0455, Validation accuracy: 0.6296

Epoch 6:
Training loss: 1.2726, Training accuracy: 0.6114
Validation loss: 0.9927, Validation accuracy: 0.6500

Epoch 7:
Training loss: 1.2434, Training accuracy: 0.6199
Validation loss: 0.9730, Validation accuracy: 0.6582

Current learning rate has decayed to 0.009025
Epoch 8:
Training loss: 1.2144, Training accuracy: 0.6328
Validation loss: 1.0359, Validation accuracy: 0.6318

Epoch 9:
Training loss: 1.2039, Training accuracy: 0.6383
Validation loss: 0.9408, Validation accuracy: 0.6682

Epoch 10:
Training loss: 1.1899, Training accuracy: 0.6428
Validation loss: 0.9123, Validation accuracy: 0.6810

Epoch 11:
Training loss: 1.1634, Training accuracy: 0.6538
Validation loss: 0.8942, Validation accuracy: 0.6824

Current learning rate has decayed to 0.008574
Epoch 12:
Training loss: 1.1556, Training accuracy: 0.6531
Validation loss: 0.9181, Validation accuracy: 0.6774

Epoch 13:
Training loss: 1.1460, Training accuracy: 0.6610
Validation loss: 0.8995, Validation accuracy: 0.6780
```

```
Epoch 14:
Training loss: 1.1447, Training accuracy: 0.6618
Validation loss: 0.9080, Validation accuracy: 0.6766

Epoch 15:
Training loss: 1.1201, Training accuracy: 0.6734
Validation loss: 0.9119, Validation accuracy: 0.6732

Current learning rate has decayed to 0.008145
Epoch 16:
Training loss: 1.1150, Training accuracy: 0.6731
Validation loss: 0.8545, Validation accuracy: 0.6982

Epoch 17:
Training loss: 1.1091, Training accuracy: 0.6742
Validation loss: 0.8717, Validation accuracy: 0.6902

Epoch 18:
Training loss: 1.1033, Training accuracy: 0.6792
Validation loss: 0.8639, Validation accuracy: 0.6904

Epoch 19:
Training loss: 1.0967, Training accuracy: 0.6817
Validation loss: 0.8311, Validation accuracy: 0.7138

Current learning rate has decayed to 0.007738
Epoch 20:
Training loss: 1.0868, Training accuracy: 0.6882
Validation loss: 0.8621, Validation accuracy: 0.6982

Epoch 21:
Training loss: 1.0844, Training accuracy: 0.6867
Validation loss: 0.8236, Validation accuracy: 0.7078

Epoch 22:
Training loss: 1.0795, Training accuracy: 0.6882
Validation loss: 0.8711, Validation accuracy: 0.6966

Epoch 23:
Training loss: 1.0714, Training accuracy: 0.6894
Validation loss: 0.8002, Validation accuracy: 0.7158

Current learning rate has decayed to 0.007351
Epoch 24:
Training loss: 1.0681, Training accuracy: 0.6917
Validation loss: 0.8195, Validation accuracy: 0.7122
```

```
Epoch 25:
Training loss: 1.0682, Training accuracy: 0.6947
Validation loss: 0.8217, Validation accuracy: 0.7118

Epoch 26:
Training loss: 1.0576, Training accuracy: 0.6974
Validation loss: 0.8289, Validation accuracy: 0.7114

Epoch 27:
Training loss: 1.0566, Training accuracy: 0.6961
Validation loss: 0.8132, Validation accuracy: 0.7136

Current learning rate has decayed to 0.006983
Epoch 28:
Training loss: 1.0500, Training accuracy: 0.6989
Validation loss: 0.7995, Validation accuracy: 0.7194

Epoch 29:
Training loss: 1.0451, Training accuracy: 0.7024
Validation loss: 0.8104, Validation accuracy: 0.7174

Epoch 30:
Training loss: 1.0433, Training accuracy: 0.7028
Validation loss: 0.8458, Validation accuracy: 0.7094

Epoch 31:
Training loss: 1.0417, Training accuracy: 0.7052
Validation loss: 0.8000, Validation accuracy: 0.7252

Current learning rate has decayed to 0.006634
Epoch 32:
Training loss: 1.0344, Training accuracy: 0.7084
Validation loss: 0.8187, Validation accuracy: 0.7112

Epoch 33:
Training loss: 1.0290, Training accuracy: 0.7066
Validation loss: 0.7925, Validation accuracy: 0.7268

Epoch 34:
Training loss: 1.0238, Training accuracy: 0.7110
Validation loss: 0.8062, Validation accuracy: 0.7192

Epoch 35:
Training loss: 1.0285, Training accuracy: 0.7102
Validation loss: 0.8106, Validation accuracy: 0.7250

Current learning rate has decayed to 0.006302
Epoch 36:
```

```
Training loss: 1.0231, Training accuracy: 0.7111
Validation loss: 0.7762, Validation accuracy: 0.7266

Epoch 37:
Training loss: 1.0206, Training accuracy: 0.7112
Validation loss: 0.7960, Validation accuracy: 0.7240

Epoch 38:
Training loss: 1.0224, Training accuracy: 0.7123
Validation loss: 0.7829, Validation accuracy: 0.7278

Epoch 39:
Training loss: 1.0238, Training accuracy: 0.7089
Validation loss: 0.7657, Validation accuracy: 0.7306

Current learning rate has decayed to 0.005987
Epoch 40:
Training loss: 1.0100, Training accuracy: 0.7164
Validation loss: 0.7733, Validation accuracy: 0.7266

Epoch 41:
Training loss: 1.0072, Training accuracy: 0.7168
Validation loss: 0.7629, Validation accuracy: 0.7352

Epoch 42:
Training loss: 1.0055, Training accuracy: 0.7194
Validation loss: 0.7642, Validation accuracy: 0.7328

Epoch 43:
Training loss: 1.0020, Training accuracy: 0.7182
Validation loss: 0.7700, Validation accuracy: 0.7284

Current learning rate has decayed to 0.005688
Epoch 44:
Training loss: 1.0022, Training accuracy: 0.7193
Validation loss: 0.7838, Validation accuracy: 0.7278

Epoch 45:
Training loss: 1.0006, Training accuracy: 0.7205
Validation loss: 0.7628, Validation accuracy: 0.7348

Epoch 46:
Training loss: 0.9984, Training accuracy: 0.7216
Validation loss: 0.7515, Validation accuracy: 0.7330

Epoch 47:
Training loss: 0.9948, Training accuracy: 0.7221
Validation loss: 0.7685, Validation accuracy: 0.7332
```

```
Current learning rate has decayed to 0.005404
Epoch 48:
Training loss: 0.9892, Training accuracy: 0.7252
Validation loss: 0.7630, Validation accuracy: 0.7338

Epoch 49:
Training loss: 0.9822, Training accuracy: 0.7244
Validation loss: 0.7835, Validation accuracy: 0.7330

Epoch 50:
Training loss: 0.9911, Training accuracy: 0.7224
Validation loss: 0.7681, Validation accuracy: 0.7308

Epoch 51:
Training loss: 0.9908, Training accuracy: 0.7228
Validation loss: 0.7556, Validation accuracy: 0.7384

Current learning rate has decayed to 0.005133
Epoch 52:
Training loss: 0.9860, Training accuracy: 0.7232
Validation loss: 0.7491, Validation accuracy: 0.7406

Epoch 53:
Training loss: 0.9856, Training accuracy: 0.7264
Validation loss: 0.7824, Validation accuracy: 0.7292

Epoch 54:
Training loss: 0.9798, Training accuracy: 0.7252
Validation loss: 0.7358, Validation accuracy: 0.7430

Epoch 55:
Training loss: 0.9792, Training accuracy: 0.7276
Validation loss: 0.7205, Validation accuracy: 0.7514

Current learning rate has decayed to 0.004877
Epoch 56:
Training loss: 0.9701, Training accuracy: 0.7299
Validation loss: 0.7431, Validation accuracy: 0.7396

Epoch 57:
Training loss: 0.9727, Training accuracy: 0.7292
Validation loss: 0.7261, Validation accuracy: 0.7470

Epoch 58:
Training loss: 0.9739, Training accuracy: 0.7277
Validation loss: 0.7697, Validation accuracy: 0.7300
```

```
Epoch 59:
Training loss: 0.9648, Training accuracy: 0.7328
Validation loss: 0.7259, Validation accuracy: 0.7484

Current learning rate has decayed to 0.004633
Epoch 60:
Training loss: 0.9671, Training accuracy: 0.7308
Validation loss: 0.7408, Validation accuracy: 0.7424

Epoch 61:
Training loss: 0.9649, Training accuracy: 0.7318
Validation loss: 0.7314, Validation accuracy: 0.7460

Epoch 62:
Training loss: 0.9590, Training accuracy: 0.7319
Validation loss: 0.7318, Validation accuracy: 0.7452

Epoch 63:
Training loss: 0.9603, Training accuracy: 0.7374
Validation loss: 0.7345, Validation accuracy: 0.7440

Current learning rate has decayed to 0.004401
Epoch 64:
Training loss: 0.9530, Training accuracy: 0.7343
Validation loss: 0.7230, Validation accuracy: 0.7474

Epoch 65:
Training loss: 0.9533, Training accuracy: 0.7348
Validation loss: 0.7517, Validation accuracy: 0.7432

Epoch 66:
Training loss: 0.9600, Training accuracy: 0.7340
Validation loss: 0.7132, Validation accuracy: 0.7538

Epoch 67:
Training loss: 0.9560, Training accuracy: 0.7363
Validation loss: 0.7211, Validation accuracy: 0.7506

Current learning rate has decayed to 0.004181
Epoch 68:
Training loss: 0.9528, Training accuracy: 0.7342
Validation loss: 0.7118, Validation accuracy: 0.7554

Epoch 69:
Training loss: 0.9525, Training accuracy: 0.7361
Validation loss: 0.7406, Validation accuracy: 0.7460

Epoch 70:
```

```
Training loss: 0.9493, Training accuracy: 0.7349
Validation loss: 0.7211, Validation accuracy: 0.7474

Epoch 71:
Training loss: 0.9498, Training accuracy: 0.7357
Validation loss: 0.7193, Validation accuracy: 0.7442

Current learning rate has decayed to 0.003972
Epoch 72:
Training loss: 0.9447, Training accuracy: 0.7386
Validation loss: 0.7206, Validation accuracy: 0.7408

Epoch 73:
Training loss: 0.9417, Training accuracy: 0.7379
Validation loss: 0.7402, Validation accuracy: 0.7436

Epoch 74:
Training loss: 0.9449, Training accuracy: 0.7374
Validation loss: 0.7181, Validation accuracy: 0.7484

Epoch 75:
Training loss: 0.9415, Training accuracy: 0.7397
Validation loss: 0.7116, Validation accuracy: 0.7556

Current learning rate has decayed to 0.003774
Epoch 76:
Training loss: 0.9375, Training accuracy: 0.7397
Validation loss: 0.7272, Validation accuracy: 0.7476

Epoch 77:
Training loss: 0.9413, Training accuracy: 0.7411
Validation loss: 0.7454, Validation accuracy: 0.7368

Epoch 78:
Training loss: 0.9380, Training accuracy: 0.7414
Validation loss: 0.7188, Validation accuracy: 0.7522

Epoch 79:
Training loss: 0.9357, Training accuracy: 0.7432
Validation loss: 0.7252, Validation accuracy: 0.7474

================================================
==> Optimization finished! Best validation accuracy: 0.7556
```

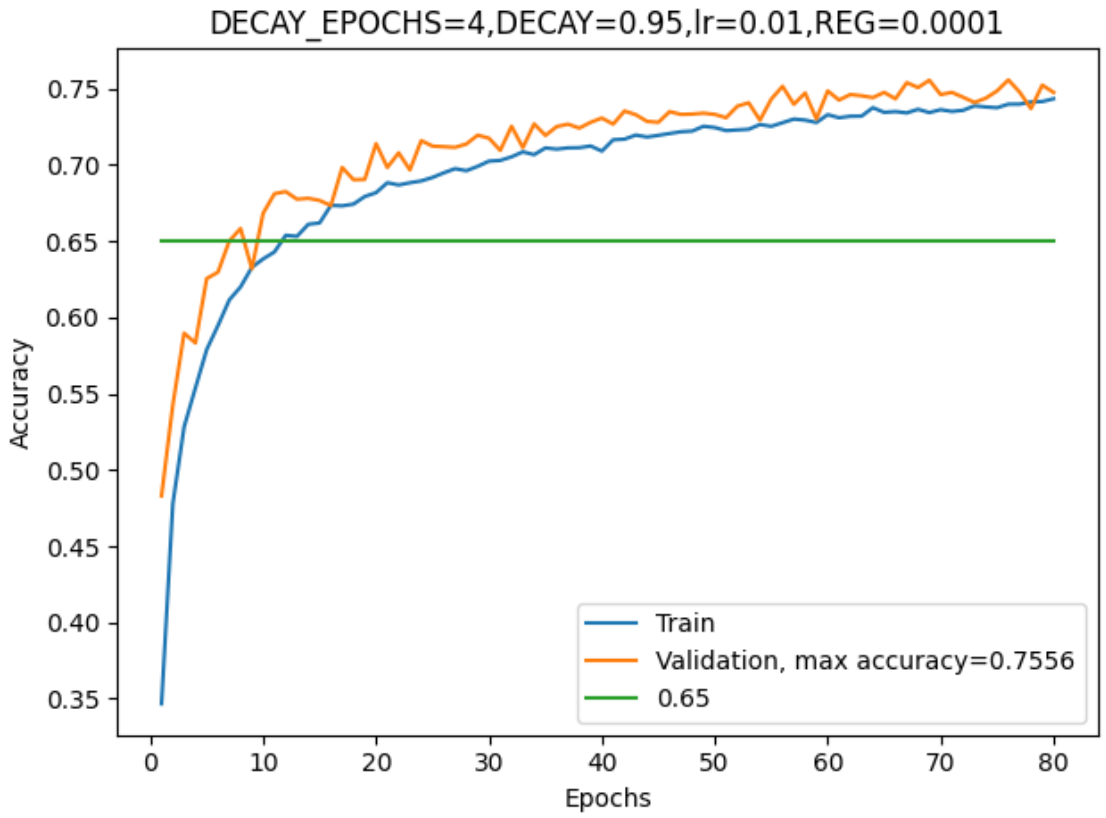
## 2 Bonus: with learning rate decay

The following code can help you adjust the learning rate during training. You need to figure out how to incorporate this code into your training loop.

```
    if i % DECAY_EPOCHS == 0 and i != 0:
        current_learning_rate = current_learning_rate * DECAY
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_learning_rate
        print("Current learning rate has decayed to %f" %current_learning_rate)
```

[11]:
```
##### plotting accuracy vs epoch


import matplotlib.pyplot as plt
import numpy as np

def my_to_np(lst):
    return np.array([lst[i].detach().cpu().numpy().flatten() for i in␣
 ↪range(len(lst))])

save = 'n'

fig, ax = plt.subplots(1, 1)
xx = np.linspace(1, EPOCHS, EPOCHS)

ax.plot(xx, my_to_np(accuracy_arr_train), label='Train')
ax.plot(xx, my_to_np(accuracy_arr_val), label='Validation, max accuracy={:.4f}'.
 ↪format(best_val_acc))
ax.plot(xx, np.linspace(0.65, 0.65, EPOCHS), label='0.65')
# title_ = 'DECAY_EPOCHS={:d}, DECAY={:g}, initial lr={:g}'.
 ↪format(DECAY_EPOCHS, DECAY, INITIAL_LR)
title_ = 'DECAY_EPOCHS={:d},DECAY={:g},lr={:g},REG={:g}'.format(DECAY_EPOCHS,␣
 ↪DECAY, INITIAL_LR, REG)
if DECAY == 1:
    title_ = 'No Learning Rate Decay'.format(DECAY_EPOCHS)
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy')
ax.legend()
ax.set_title(title_)
fig.tight_layout()
if save == 'y':
    # plt.savefig('q2a_DECAY_EPOCHS_{:d}_DECAY_{:g}_aug.pdf'.
 ↪format(DECAY_EPOCHS, DECAY),
    #             dpi=500, bbox_inches='tight')
    # plt.savefig('q2b1_DECAY_EPOCHS_{:d}_DECAY_{:g}_aug_BN.pdf'.
 ↪format(DECAY_EPOCHS, DECAY),
    #             dpi=500, bbox_inches='tight')
```

```
  # plt.savefig('q2b2_aug_BN_lr{:g}.pdf'.format(INITIAL_LR),
  #             dpi=500, bbox_inches='tight')
  # plt.savefig('q2b2_aug_wo_BN_lr{:g}.pdf'.format(INITIAL_LR),
  #             dpi=500, bbox_inches='tight')
  # plt.savefig('q2b3_aug_BN_swish_lr{:g}_{:d}epochs.pdf'.format(INITIAL_LR,␣
↪EPOCHS),
  #             dpi=500, bbox_inches='tight')
  # plt.savefig('q2b3_aug_BN_relu_lr{:g}_{:d}epochs.pdf'.format(INITIAL_LR,␣
↪EPOCHS),
  #              dpi=500, bbox_inches='tight')
  # plt.savefig('q2c_aug_BN_swish_lr{:g}_{:d}epochs.pdf'.format(INITIAL_LR,␣
↪EPOCHS),
  #              dpi=500, bbox_inches='tight')
  # plt.savefig('q2c2_aug_BN_swish_lr{:g}_L2{:g}.pdf'.format(INITIAL_LR,␣
↪REG),
  #              dpi=500, bbox_inches='tight')
  plt.savefig('q2c3_aug_BN_swish_lr{:g}_{:s}_{:g}.pdf'.format(INITIAL_LR,␣
↪L1_or_L2, REG),
              dpi=500, bbox_inches='tight')
```



[12]:
```
##### plotting weights distribution

save = 'n'
fig, ax = plt.subplots(1, 1)

weights_list = []

for name, module in model.named_modules():
    if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
        weights_list.extend(list((module.weight.cpu().detach().numpy()).
 ↪flatten()))

ax.hist(weights_list, bins=100)
ax.set_title(str("Weight histogram, " + L1_or_L2 + ",lr={:g},REG={:g}".
 ↪format(INITIAL_LR, REG)))

ax.set_xlabel('Epochs')
ax.set_ylabel('Frequency')
fig.tight_layout()
if save == 'y':
    plt.savefig('q2c3weights_aug_BN_swish_lr{:g}_{:s}_{:g}.pdf'.
 ↪format(INITIAL_LR, L1_or_L2, REG),
                dpi=500, bbox_inches='tight')
```