simplenn-cifar10

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]:
```
# define the SimpleNN mode;
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, 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)

    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
```

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(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    ])

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=False,     # 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

# 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

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

# Add optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=INITIAL_LR,␣
 ↪momentum=MOMENTUM, weight_decay=REG)
#############################################
```

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 = 60

DECAY_EPOCHS = 2
DECAY = 0.9

# 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)
        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
```

```python
    # 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.pth'))

    print('')

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

```
==> Training starts!
==================================================
Epoch 0:
Training loss: 1.9625, Training accuracy: 0.2727
Validation loss: 1.6071, Validation accuracy: 0.4076

Epoch 1:
Training loss: 1.4716, Training accuracy: 0.4700
Validation loss: 1.3839, Validation accuracy: 0.5200

Current learning rate has decayed to 0.009000
Epoch 2:
Training loss: 1.2947, Training accuracy: 0.5405
Validation loss: 1.2453, Validation accuracy: 0.5636

Epoch 3:
Training loss: 1.1725, Training accuracy: 0.5868
Validation loss: 1.1866, Validation accuracy: 0.5772

Current learning rate has decayed to 0.008100
Epoch 4:
Training loss: 1.0768, Training accuracy: 0.6200
Validation loss: 1.1469, Validation accuracy: 0.5984

Epoch 5:
Training loss: 1.0068, Training accuracy: 0.6466
Validation loss: 1.0857, Validation accuracy: 0.6162
```

```
Current learning rate has decayed to 0.007290
Epoch 6:
Training loss: 0.9371, Training accuracy: 0.6709
Validation loss: 1.0392, Validation accuracy: 0.6356

Epoch 7:
Training loss: 0.8823, Training accuracy: 0.6894
Validation loss: 1.0673, Validation accuracy: 0.6312

Current learning rate has decayed to 0.006561
Epoch 8:
Training loss: 0.8204, Training accuracy: 0.7125
Validation loss: 1.0355, Validation accuracy: 0.6460

Epoch 9:
Training loss: 0.7790, Training accuracy: 0.7285
Validation loss: 1.0057, Validation accuracy: 0.6542

Current learning rate has decayed to 0.005905
Epoch 10:
Training loss: 0.7194, Training accuracy: 0.7486
Validation loss: 1.0120, Validation accuracy: 0.6566

Epoch 11:
Training loss: 0.6883, Training accuracy: 0.7591
Validation loss: 1.0438, Validation accuracy: 0.6514

Current learning rate has decayed to 0.005314
Epoch 12:
Training loss: 0.6451, Training accuracy: 0.7768
Validation loss: 1.1184, Validation accuracy: 0.6428

Epoch 13:
Training loss: 0.6130, Training accuracy: 0.7856
Validation loss: 1.0902, Validation accuracy: 0.6482

Current learning rate has decayed to 0.004783
Epoch 14:
Training loss: 0.5675, Training accuracy: 0.8002
Validation loss: 1.0817, Validation accuracy: 0.6616

Epoch 15:
Training loss: 0.5302, Training accuracy: 0.8143
Validation loss: 1.1328, Validation accuracy: 0.6496

Current learning rate has decayed to 0.004305
Epoch 16:
```

```
Training loss: 0.4984, Training accuracy: 0.8237
Validation loss: 1.1454, Validation accuracy: 0.6544

Epoch 17:
Training loss: 0.4636, Training accuracy: 0.8387
Validation loss: 1.1782, Validation accuracy: 0.6596

Current learning rate has decayed to 0.003874
Epoch 18:
Training loss: 0.4231, Training accuracy: 0.8528
Validation loss: 1.2403, Validation accuracy: 0.6540

Epoch 19:
Training loss: 0.4029, Training accuracy: 0.8584
Validation loss: 1.2890, Validation accuracy: 0.6476

Current learning rate has decayed to 0.003487
Epoch 20:
Training loss: 0.3637, Training accuracy: 0.8747
Validation loss: 1.3256, Validation accuracy: 0.6540

Epoch 21:
Training loss: 0.3403, Training accuracy: 0.8810
Validation loss: 1.3676, Validation accuracy: 0.6522

Current learning rate has decayed to 0.003138
Epoch 22:
Training loss: 0.3051, Training accuracy: 0.8954
Validation loss: 1.4623, Validation accuracy: 0.6486

Epoch 23:
Training loss: 0.2903, Training accuracy: 0.8997
Validation loss: 1.5443, Validation accuracy: 0.6494

Current learning rate has decayed to 0.002824
Epoch 24:
Training loss: 0.2557, Training accuracy: 0.9140
Validation loss: 1.5894, Validation accuracy: 0.6510

Epoch 25:
Training loss: 0.2384, Training accuracy: 0.9192
Validation loss: 1.6426, Validation accuracy: 0.6482

Current learning rate has decayed to 0.002542
Epoch 26:
Training loss: 0.2102, Training accuracy: 0.9316
Validation loss: 1.7288, Validation accuracy: 0.6422
```

```
Epoch 27:
Training loss: 0.1948, Training accuracy: 0.9361
Validation loss: 1.7892, Validation accuracy: 0.6470

Current learning rate has decayed to 0.002288
Epoch 28:
Training loss: 0.1703, Training accuracy: 0.9460
Validation loss: 1.8600, Validation accuracy: 0.6464

Epoch 29:
Training loss: 0.1528, Training accuracy: 0.9532
Validation loss: 1.9480, Validation accuracy: 0.6468

Current learning rate has decayed to 0.002059
Epoch 30:
Training loss: 0.1333, Training accuracy: 0.9608
Validation loss: 1.9924, Validation accuracy: 0.6426

Epoch 31:
Training loss: 0.1150, Training accuracy: 0.9686
Validation loss: 2.0966, Validation accuracy: 0.6384

Current learning rate has decayed to 0.001853
Epoch 32:
Training loss: 0.0998, Training accuracy: 0.9743
Validation loss: 2.2083, Validation accuracy: 0.6372

Epoch 33:
Training loss: 0.0892, Training accuracy: 0.9781
Validation loss: 2.2950, Validation accuracy: 0.6384

Current learning rate has decayed to 0.001668
Epoch 34:
Training loss: 0.0755, Training accuracy: 0.9835
Validation loss: 2.3423, Validation accuracy: 0.6446

Epoch 35:
Training loss: 0.0656, Training accuracy: 0.9870
Validation loss: 2.3982, Validation accuracy: 0.6518

Current learning rate has decayed to 0.001501
Epoch 36:
Training loss: 0.0557, Training accuracy: 0.9905
Validation loss: 2.5021, Validation accuracy: 0.6432

Epoch 37:
Training loss: 0.0485, Training accuracy: 0.9925
Validation loss: 2.5632, Validation accuracy: 0.6418
```

```
Current learning rate has decayed to 0.001351
Epoch 38:
Training loss: 0.0418, Training accuracy: 0.9942
Validation loss: 2.6383, Validation accuracy: 0.6406

Epoch 39:
Training loss: 0.0374, Training accuracy: 0.9954
Validation loss: 2.6903, Validation accuracy: 0.6422

Current learning rate has decayed to 0.001216
Epoch 40:
Training loss: 0.0329, Training accuracy: 0.9964
Validation loss: 2.7536, Validation accuracy: 0.6440

Epoch 41:
Training loss: 0.0297, Training accuracy: 0.9971
Validation loss: 2.8155, Validation accuracy: 0.6398

Current learning rate has decayed to 0.001094
Epoch 42:
Training loss: 0.0266, Training accuracy: 0.9979
Validation loss: 2.8463, Validation accuracy: 0.6412

Epoch 43:
Training loss: 0.0245, Training accuracy: 0.9980
Validation loss: 2.9160, Validation accuracy: 0.6382

Current learning rate has decayed to 0.000985
Epoch 44:
Training loss: 0.0224, Training accuracy: 0.9985
Validation loss: 2.9329, Validation accuracy: 0.6412

Epoch 45:
Training loss: 0.0209, Training accuracy: 0.9988
Validation loss: 2.9714, Validation accuracy: 0.6406

Current learning rate has decayed to 0.000886
Epoch 46:
Training loss: 0.0193, Training accuracy: 0.9988
Validation loss: 3.0046, Validation accuracy: 0.6396

Epoch 47:
Training loss: 0.0183, Training accuracy: 0.9990
Validation loss: 3.0422, Validation accuracy: 0.6392

Current learning rate has decayed to 0.000798
Epoch 48:
```

```
Training loss: 0.0172, Training accuracy: 0.9991
Validation loss: 3.0792, Validation accuracy: 0.6378

Epoch 49:
Training loss: 0.0165, Training accuracy: 0.9991
Validation loss: 3.1001, Validation accuracy: 0.6414

Current learning rate has decayed to 0.000718
Epoch 50:
Training loss: 0.0156, Training accuracy: 0.9992
Validation loss: 3.1270, Validation accuracy: 0.6384

Epoch 51:
Training loss: 0.0150, Training accuracy: 0.9993
Validation loss: 3.1526, Validation accuracy: 0.6402

Current learning rate has decayed to 0.000646
Epoch 52:
Training loss: 0.0143, Training accuracy: 0.9994
Validation loss: 3.1702, Validation accuracy: 0.6382

Epoch 53:
Training loss: 0.0139, Training accuracy: 0.9994
Validation loss: 3.1846, Validation accuracy: 0.6422

Current learning rate has decayed to 0.000581
Epoch 54:
Training loss: 0.0133, Training accuracy: 0.9995
Validation loss: 3.2113, Validation accuracy: 0.6400

Epoch 55:
Training loss: 0.0129, Training accuracy: 0.9996
Validation loss: 3.2298, Validation accuracy: 0.6404

Current learning rate has decayed to 0.000523
Epoch 56:
Training loss: 0.0125, Training accuracy: 0.9996
Validation loss: 3.2405, Validation accuracy: 0.6386

Epoch 57:
Training loss: 0.0122, Training accuracy: 0.9996
Validation loss: 3.2621, Validation accuracy: 0.6366

Current learning rate has decayed to 0.000471
Epoch 58:
Training loss: 0.0118, Training accuracy: 0.9996
Validation loss: 3.2762, Validation accuracy: 0.6382
```

```
Epoch 59:
Training loss: 0.0115, Training accuracy: 0.9997
Validation loss: 3.2823, Validation accuracy: 0.6394

==================================================
==> Optimization finished! Best validation accuracy: 0.6616
```

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.

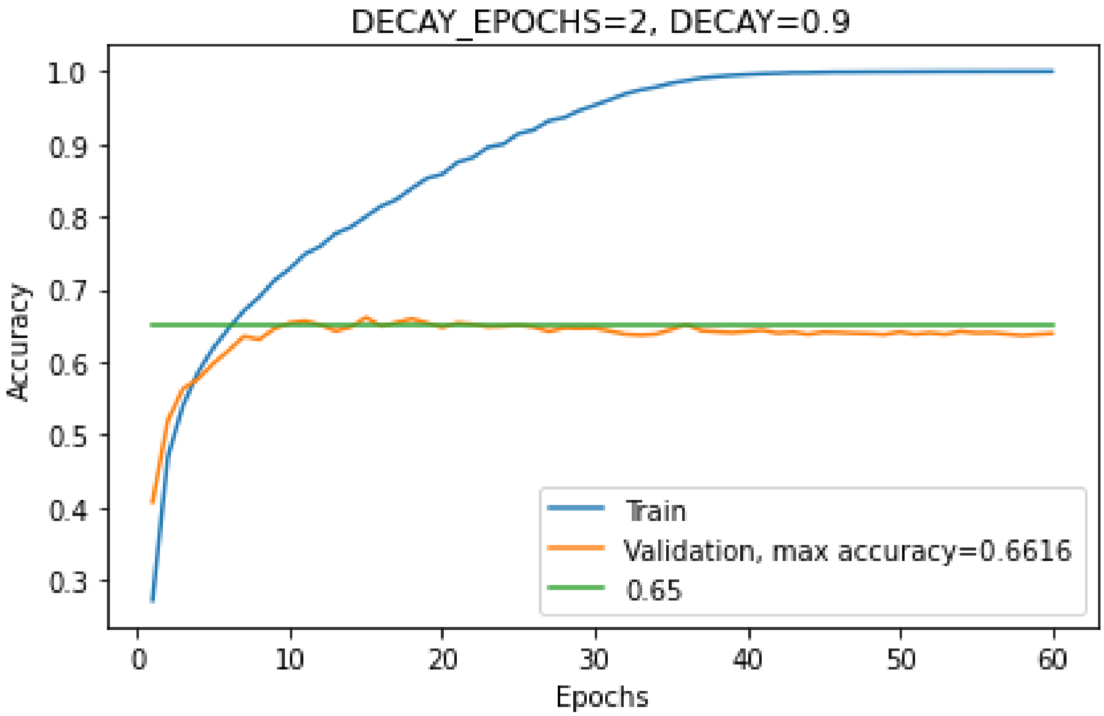
```
if i % DECAY_EPOCHS == 0 and i != 0:
    current_learning_rate = current_learning_rate * DECAY
    for param_group in optimizer.param_groups:
        param_group['lr'] = current_learning_rate
    print("Current learning rate has decayed to %f" %current_learning_rate)
```

[11]:
```
##### my addition
import matplotlib.pyplot as plt
import numpy as np

save = 'y'

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

ax.plot(xx, accuracy_arr_train, label='Train')
ax.plot(xx, 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}'.format(DECAY_EPOCHS, DECAY)
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('q1h_DECAY_EPOCHS_{:d}_DECAY_{:g}.pdf'.format(DECAY_EPOCHS,␣
 ↪DECAY), dpi=500, bbox_inches='tight')
    plt.savefig('q2a_DECAY_EPOCHS_{:d}_DECAY_{:g}_wo_aug.pdf'.
                format(DECAY_EPOCHS, DECAY), dpi=500, bbox_inches='tight')
##### end of my addition
```


DECAY_EPOCHS=2, DECAY=0.9
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
Accuracy
Train
Validation, max accuracy=0.6616
0.65
0
10
20
30
40
50
60
Epochs

```
[ ]:
```