a3_cifar10

August 5, 2019

- assignment problem. The red color indicates the task that should be done
- debugging. The green tells you what is expected outcome. Its primarily goal is to help you get the correct answer
- comments, hints.

1 Assignment 3 (CNN)

Useful References: * official pytorch cifar10 tutorial
https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html * tensorboard in colab
https://www.tensorflow.org/tensorboard/r2/tensorboard_in_notebooks * tensorboard & colab
https://colab.research.google.com/drive/1NbEqqB42VSzYt-mmb4ESc8yxL05U2TIV
https://medium.com/looka-engineering/how-to-use-tensorboard-with-pytorch-in-google-colab-1f76a938bc34

1.1 Preliminaries

Check the environment

[1]:
```
!pwd
```

```
/home/dima/UCU/ComputerVision/Repos/ComputerVision/homework5/assignments
```

[2]:
```
!nvidia-smi
```

```
Mon Aug  5 02:12:51 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.116                Driver Version: 390.116                   |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 105...  Off  | 00000000:01:00.0 Off |                  N/A |
| N/A   46C    P3    N/A /  N/A |    572MiB /  4042MiB |      5%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
```

```
| Processes:                                                                 GPU Memory |
|  GPU       PID   Type   Process name                                       Usage      |
|=============================================================================|
|    0      2360      G   /usr/lib/xorg/Xorg                                257MiB |
|    0      3255      G   /usr/bin/gnome-shell                              126MiB |
|    0      3995      G   ...quest-channel-token=5650203069834999801         74MiB |
|    0      4247      G   ...quest-channel-token=7231672911215649333         79MiB |
|    0     11465      G   ...quest-channel-token=6908275127681227519         33MiB |
+-----------------------------------------------------------------------------+
```

Install missing packages

[3]:
```
# [colab version] latest version of tensorflow in order to use tensorboard␣
 ↪inside the notebook
# !pip install -q tf-nightly-2.0-preview
```

Load libs, set settings

[4]:
```
import numpy as np
import matplotlib.pyplot as plt

import datetime

import torch
import torchvision
import torchvision.transforms as transforms
import time

# for tensorboard
from torch.utils.tensorboard import SummaryWriter
```

[5]:
```
# random seed settings
torch.manual_seed(42)
np.random.seed(42)
```

[6]:
```
# Load the TensorBoard notebook extension
%load_ext tensorboard

# for auto-reloading external modules (files, etc.)
%load_ext autoreload
%autoreload 2

# to be able to make plots inline the notebook
# (actually no need for the colab version)
%matplotlib inline

# make plots a bit nicer
plt.matplotlib.rcParams.update({'font.size': 18, 'font.family': 'serif'})
```

1.2 Define pathes

[7]:
```
# path for dataset (will be not there after end of session)
path_data = "data/cifar10"
```

1.3 Data

Using `torchvision` load CIFAR10.

The output of torchvision datasets are PILImage images of range [0, 1]. We transform them to Tensors of normalized range [-1, 1].

[8]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root=path_data, train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root=path_data, train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[9]:
```
ls data/cifar10
```

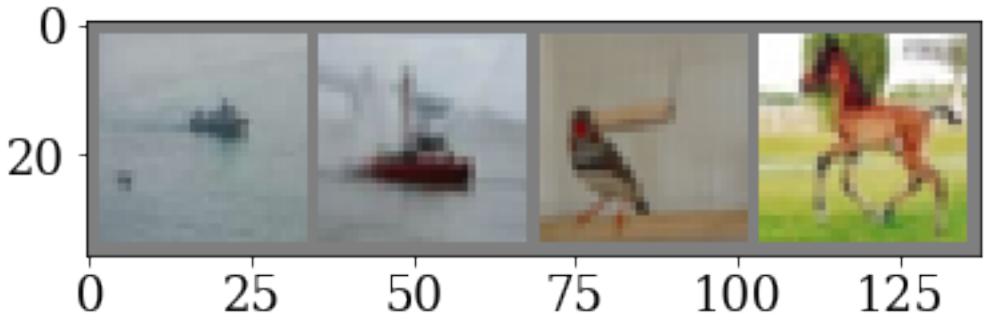
```
cifar-10-batches-py/  cifar-10-python.tar.gz
```

Let us show some of the training images, for fun.

[10]:
```
# functions to show an image
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))


# get some random training images
images, labels = next(iter(trainloader))

# show images
```

```
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

```
 ship  ship  bird horse
```



1.4 Define a Convolution Neural Network

Copy the neural network from the Neural Networks section before and modify it to take 3-channel images (instead of 1-channel images as it was defined).

[11]:
```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)
        self.dropout1 = nn.Dropout(0.2)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv4 = nn.Conv2d(64, 64, 3, padding=1)
        self.dropout2 = nn.Dropout(0.2)
        self.conv5 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv6 = nn.Conv2d(128, 128, 3, padding=1)
        self.dropout3 = nn.Dropout(0.2)
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.dropout4 = nn.Dropout(0.2)
        self.fc2 = nn.Linear(512, 128)
        self.dropout5 = nn.Dropout(0.2)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
```

```
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = self.dropout1(x)

        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = self.dropout2(x)

        x = F.relu(self.conv5(x))
        x = F.relu(self.conv6(x))
        x = self.pool(x)
        x = self.dropout3(x)

        x = x.view(-1, 128 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = self.dropout4(x)
        x = F.relu(self.fc2(x))
        x = self.dropout5(x)
        x = self.fc3(x)
        return x

net_name = "vgg_3block_v2_dropout"
net = Net()
```

1.5 Define a Loss function

Let's use a Classification Cross-Entropy loss

[12]:
```
criterion = nn.CrossEntropyLoss()
```

1.6 Set-up training

[13]:
```
def calc_accuracy(dataloader, model, device):
    correct = 0
    total = 0
    with torch.no_grad():
        for input_data, target in dataloader:
            target = target.to(device=device)
            input_data = input_data.to(device=device)
            outputs = model(input_data)
            _, predicted = torch.max(outputs.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()
    return 100 * correct / total
```

```
def calc_loss(dataloader, model, device):
    correct = 0
    total = 0
    losses = []
    with torch.no_grad():
        for input_data, target in dataloader:
            target = target.to(device=device)
            input_data = input_data.to(device=device)
            outputs = model(input_data)
            loss = criterion(outputs, target)
            losses.append(loss.item())
    return np.mean(losses)
```

[14]:
```
def train_epoch(train_loader, model, criterion, optimizer, scheduler,
                epoch, device, log_interval, globaliter):

    # switch to train mode
    model.train()

    # adjust_learning_rate
    if scheduler is not None:
        scheduler.step()

    for batch_idx, (input_data, target) in enumerate(train_loader):

        # TODO: do in other way (this is global batch index, for logging)
        globaliter += 1

        # extract batch data
        target = target.to(device=device)
        input_data = input_data.to(device=device)

        # compute output
        output = model(input_data)
        loss = criterion(output, target)

        # compute gradient and do optimizer step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # logging
        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                      epoch, batch_idx * len(input_data), len(train_loader.
 ↪dataset),
                      100. * batch_idx / len(train_loader), loss.item()))
```

```
            # log loss
            writer.add_scalar('Train/IterationsVsLoss', loss.item(), globaliter)
            # log LR
            lr = scheduler.get_lr()[0]
            writer.add_scalar('Train/LearningRate', lr, globaliter)

    return globaliter
```

1.7 Logging

[15]:
```
# path for dataset (will be not there after end of session)
current_time = str(datetime.datetime.now().timestamp())
path_log = 'logs/tensorboard/' + net_name + "_" + current_time

# set-up writer
writer = SummaryWriter(path_log)
```

[16]:
```
# Model (and Images) to TensorBoard (TB)

images, labels = next(iter(trainloader))
grid = torchvision.utils.make_grid(images)
writer.add_image('images', grid, 0)

# net to TB
device = torch.device('cpu')
net = net.to(device=device)
writer.add_graph(net, images)
```

[17]:
```
%tensorboard --logdir logs/tensorboard
```

```
Reusing TensorBoard on port 6006 (pid 9744), started 3:56:38 ago. (Use '!kill 9744' to kill it
```

```
<IPython.lib.display.IFrame at 0x7efb9cdac9e8>
```

1.8 Training

[18]:
```
# set number of epoch
n_epochs = 5

# set-up optimizer and scheduler
optimizer = torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                            step_size=5,
                                            gamma=0.1)
# device = 'cpu'
```

```
device = 'cuda'
globaliter = 0
device = torch.device(device)
net = net.to(device=device)

start = time.time()
for epoch in range(1, n_epochs + 1):
    globaliter = train_epoch(trainloader, net, criterion, optimizer, scheduler,␣
 ↪epoch, device, 500, globaliter)
    writer.add_scalar('Train/EpochsVsLoss', calc_loss(trainloader, net,␣
 ↪device), epoch)
    writer.add_scalar('Test/EpochsVsLoss', calc_loss(testloader, net, device),␣
 ↪epoch)
    writer.add_scalar('Train/EpochsVsAccuracy', calc_accuracy(trainloader, net,␣
 ↪device), epoch)
    writer.add_scalar('Test/EpochsVsAccuracy', calc_accuracy(testloader, net,␣
 ↪device), epoch)

end = time.time()
writer.add_scalar('Train/ExecutionTime_{}'.format(device), end - start)
```

```
Train Epoch: 1 [0/50000 (0%)]   Loss: 2.302413
Train Epoch: 1 [2000/50000 (4%)]        Loss: 2.306021
Train Epoch: 1 [4000/50000 (8%)]        Loss: 2.279253
Train Epoch: 1 [6000/50000 (12%)]       Loss: 2.313576
Train Epoch: 1 [8000/50000 (16%)]       Loss: 2.283106
Train Epoch: 1 [10000/50000 (20%)]      Loss: 2.325875
Train Epoch: 1 [12000/50000 (24%)]      Loss: 2.297396
Train Epoch: 1 [14000/50000 (28%)]      Loss: 2.281273
Train Epoch: 1 [16000/50000 (32%)]      Loss: 2.322986
Train Epoch: 1 [18000/50000 (36%)]      Loss: 2.314445
Train Epoch: 1 [20000/50000 (40%)]      Loss: 2.296417
Train Epoch: 1 [22000/50000 (44%)]      Loss: 2.287430
Train Epoch: 1 [24000/50000 (48%)]      Loss: 2.322195
Train Epoch: 1 [26000/50000 (52%)]      Loss: 2.283132
Train Epoch: 1 [28000/50000 (56%)]      Loss: 2.302885
Train Epoch: 1 [30000/50000 (60%)]      Loss: 2.305017
Train Epoch: 1 [32000/50000 (64%)]      Loss: 2.298969
Train Epoch: 1 [34000/50000 (68%)]      Loss: 2.259380
Train Epoch: 1 [36000/50000 (72%)]      Loss: 2.298090
Train Epoch: 1 [38000/50000 (76%)]      Loss: 2.298517
Train Epoch: 1 [40000/50000 (80%)]      Loss: 2.266521
Train Epoch: 1 [42000/50000 (84%)]      Loss: 2.269654
Train Epoch: 1 [44000/50000 (88%)]      Loss: 2.276579
Train Epoch: 1 [46000/50000 (92%)]      Loss: 1.807100
Train Epoch: 1 [48000/50000 (96%)]      Loss: 2.206672
Train Epoch: 2 [0/50000 (0%)]   Loss: 2.037696
```

```
Train Epoch: 2 [2000/50000 (4%)]        Loss: 2.807856
Train Epoch: 2 [4000/50000 (8%)]        Loss: 2.522194
Train Epoch: 2 [6000/50000 (12%)]       Loss: 1.994397
Train Epoch: 2 [8000/50000 (16%)]       Loss: 1.336990
Train Epoch: 2 [10000/50000 (20%)]      Loss: 2.429588
Train Epoch: 2 [12000/50000 (24%)]      Loss: 2.823310
Train Epoch: 2 [14000/50000 (28%)]      Loss: 1.412503
Train Epoch: 2 [16000/50000 (32%)]      Loss: 1.725552
Train Epoch: 2 [18000/50000 (36%)]      Loss: 1.541113
Train Epoch: 2 [20000/50000 (40%)]      Loss: 1.420707
Train Epoch: 2 [22000/50000 (44%)]      Loss: 1.163422
Train Epoch: 2 [24000/50000 (48%)]      Loss: 1.535677
Train Epoch: 2 [26000/50000 (52%)]      Loss: 1.567748
Train Epoch: 2 [28000/50000 (56%)]      Loss: 1.596470
Train Epoch: 2 [30000/50000 (60%)]      Loss: 1.345752
Train Epoch: 2 [32000/50000 (64%)]      Loss: 1.506351
Train Epoch: 2 [34000/50000 (68%)]      Loss: 1.405900
Train Epoch: 2 [36000/50000 (72%)]      Loss: 1.676985
Train Epoch: 2 [38000/50000 (76%)]      Loss: 2.233533
Train Epoch: 2 [40000/50000 (80%)]      Loss: 1.469631
Train Epoch: 2 [42000/50000 (84%)]      Loss: 1.790094
Train Epoch: 2 [44000/50000 (88%)]      Loss: 1.672816
Train Epoch: 2 [46000/50000 (92%)]      Loss: 1.320435
Train Epoch: 2 [48000/50000 (96%)]      Loss: 0.924370
Train Epoch: 3 [0/50000 (0%)]   Loss: 1.914663
Train Epoch: 3 [2000/50000 (4%)]        Loss: 1.676053
Train Epoch: 3 [4000/50000 (8%)]        Loss: 1.889041
Train Epoch: 3 [6000/50000 (12%)]       Loss: 1.087935
Train Epoch: 3 [8000/50000 (16%)]       Loss: 1.668571
Train Epoch: 3 [10000/50000 (20%)]      Loss: 1.254669
Train Epoch: 3 [12000/50000 (24%)]      Loss: 1.006042
Train Epoch: 3 [14000/50000 (28%)]      Loss: 1.053289
Train Epoch: 3 [16000/50000 (32%)]      Loss: 1.795834
Train Epoch: 3 [18000/50000 (36%)]      Loss: 0.605243
Train Epoch: 3 [20000/50000 (40%)]      Loss: 1.402145
Train Epoch: 3 [22000/50000 (44%)]      Loss: 1.598189
Train Epoch: 3 [24000/50000 (48%)]      Loss: 0.825855
Train Epoch: 3 [26000/50000 (52%)]      Loss: 1.731425
Train Epoch: 3 [28000/50000 (56%)]      Loss: 0.708877
Train Epoch: 3 [30000/50000 (60%)]      Loss: 0.693768
Train Epoch: 3 [32000/50000 (64%)]      Loss: 1.573269
Train Epoch: 3 [34000/50000 (68%)]      Loss: 0.501931
Train Epoch: 3 [36000/50000 (72%)]      Loss: 1.014184
Train Epoch: 3 [38000/50000 (76%)]      Loss: 1.292787
Train Epoch: 3 [40000/50000 (80%)]      Loss: 1.714407
Train Epoch: 3 [42000/50000 (84%)]      Loss: 1.773646
Train Epoch: 3 [44000/50000 (88%)]      Loss: 1.042924
Train Epoch: 3 [46000/50000 (92%)]      Loss: 0.879903
```

```
Train Epoch: 3 [48000/50000 (96%)]      Loss: 1.580692
Train Epoch: 4 [0/50000 (0%)]   Loss: 1.063759
Train Epoch: 4 [2000/50000 (4%)]        Loss: 0.874704
Train Epoch: 4 [4000/50000 (8%)]        Loss: 1.496957
Train Epoch: 4 [6000/50000 (12%)]       Loss: 1.856838
Train Epoch: 4 [8000/50000 (16%)]       Loss: 0.779423
Train Epoch: 4 [10000/50000 (20%)]      Loss: 1.515180
Train Epoch: 4 [12000/50000 (24%)]      Loss: 1.272280
Train Epoch: 4 [14000/50000 (28%)]      Loss: 1.163113
Train Epoch: 4 [16000/50000 (32%)]      Loss: 1.429261
Train Epoch: 4 [18000/50000 (36%)]      Loss: 0.928337
Train Epoch: 4 [20000/50000 (40%)]      Loss: 0.575294
Train Epoch: 4 [22000/50000 (44%)]      Loss: 1.103197
Train Epoch: 4 [24000/50000 (48%)]      Loss: 1.484552
Train Epoch: 4 [26000/50000 (52%)]      Loss: 0.683538
Train Epoch: 4 [28000/50000 (56%)]      Loss: 0.918258
Train Epoch: 4 [30000/50000 (60%)]      Loss: 1.458449
Train Epoch: 4 [32000/50000 (64%)]      Loss: 0.996354
Train Epoch: 4 [34000/50000 (68%)]      Loss: 1.728029
Train Epoch: 4 [36000/50000 (72%)]      Loss: 1.410805
Train Epoch: 4 [38000/50000 (76%)]      Loss: 0.754200
Train Epoch: 4 [40000/50000 (80%)]      Loss: 1.306446
Train Epoch: 4 [42000/50000 (84%)]      Loss: 0.971885
Train Epoch: 4 [44000/50000 (88%)]      Loss: 2.018604
Train Epoch: 4 [46000/50000 (92%)]      Loss: 1.191027
Train Epoch: 4 [48000/50000 (96%)]      Loss: 0.970706
Train Epoch: 5 [0/50000 (0%)]   Loss: 1.144580
Train Epoch: 5 [2000/50000 (4%)]        Loss: 0.533292
Train Epoch: 5 [4000/50000 (8%)]        Loss: 0.777728
Train Epoch: 5 [6000/50000 (12%)]       Loss: 1.482499
Train Epoch: 5 [8000/50000 (16%)]       Loss: 0.803737
Train Epoch: 5 [10000/50000 (20%)]      Loss: 0.318600
Train Epoch: 5 [12000/50000 (24%)]      Loss: 0.777497
Train Epoch: 5 [14000/50000 (28%)]      Loss: 0.128679
Train Epoch: 5 [16000/50000 (32%)]      Loss: 0.601152
Train Epoch: 5 [18000/50000 (36%)]      Loss: 0.820511
Train Epoch: 5 [20000/50000 (40%)]      Loss: 0.102109
Train Epoch: 5 [22000/50000 (44%)]      Loss: 1.546815
Train Epoch: 5 [24000/50000 (48%)]      Loss: 1.148021
Train Epoch: 5 [26000/50000 (52%)]      Loss: 0.517927
Train Epoch: 5 [28000/50000 (56%)]      Loss: 0.666994
Train Epoch: 5 [30000/50000 (60%)]      Loss: 0.460318
Train Epoch: 5 [32000/50000 (64%)]      Loss: 1.258019
Train Epoch: 5 [34000/50000 (68%)]      Loss: 0.364448
Train Epoch: 5 [36000/50000 (72%)]      Loss: 0.949031
Train Epoch: 5 [38000/50000 (76%)]      Loss: 0.382683
Train Epoch: 5 [40000/50000 (80%)]      Loss: 1.604599
Train Epoch: 5 [42000/50000 (84%)]      Loss: 0.233570
```

```
Train Epoch: 5 [44000/50000 (88%)]      Loss: 0.336459
Train Epoch: 5 [46000/50000 (92%)]      Loss: 0.553643
Train Epoch: 5 [48000/50000 (96%)]      Loss: 0.137549
```

[19]:
```
ls logs/tensorboard/
```

```
default_1564920516.411322/
default_1564925160.082206/
default_3layer_kernel3_1564922865.229081/
default_3layer_kernel3_2_1564939973.285632/
default_double_kernel3_1564921040.465951/
default_double_kernel3_v2_1564923879.888557/
vgg_1block_v2_1564941454.841095/
vgg_3block_v2_100epochs_1564944242.370245/
vgg_3block_v2_1564942707.596526/
vgg_3block_v2_dropout_100epochs_1564956667.466318/
vgg_3block_v2_dropout_1564960374.486479/
vgg_3block_v2_dropout_comparsion_1564952494.902232/
vgg_3block_v2_dropout_comparsion_1564954757.227014/
```

[20]:
```
rm -r logs/tensorboard/1564914403.503957/
```

```
rm: cannot remove 'logs/tensorboard/1564914403.503957/': No such file or
directory
```

1.9 Test the network on the test data

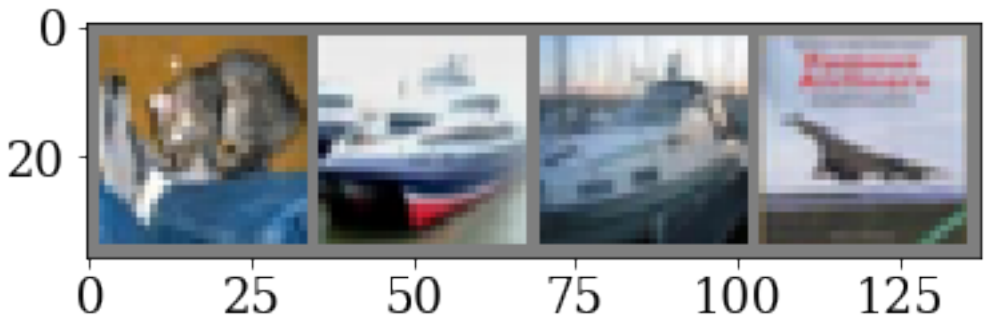
[21]:
```
# switch to test mode
net.eval()
```

[21]:
```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (dropout1): Dropout(p=0.2)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (dropout2): Dropout(p=0.2)
  (conv5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv6): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (dropout3): Dropout(p=0.2)
  (fc1): Linear(in_features=2048, out_features=512, bias=True)
  (dropout4): Dropout(p=0.2)
  (fc2): Linear(in_features=512, out_features=128, bias=True)
  (dropout5): Dropout(p=0.2)
  (fc3): Linear(in_features=128, out_features=10, bias=True)
```

```
)
```

[22]:
```
dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

```
GroundTruth:    cat  ship  ship plane
```



Okay, now let us see what the neural network thinks these examples above are:

[23]:
```
images = images.to(device=device)
labels = labels.to(device=device)
outputs = net(images)
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

```
Predicted:    cat  ship  ship plane
```

Let us look at how the network performs on the whole dataset.

[24]:
```
print('Accuracy of the network on the 10000 test images: %d %%' %␣
 ↪calc_accuracy(testloader, net, device))
```

```
Accuracy of the network on the 10000 test images: 72 %
```

the classes that performed well, and the classes that did not perform well:

[25]:
```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
```

```
        labels = labels.to(device=device)
        images = images.to(device=device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of plane : 72 %
Accuracy of   car : 88 %
Accuracy of  bird : 52 %
Accuracy of   cat : 52 %
Accuracy of  deer : 65 %
Accuracy of   dog : 67 %
Accuracy of  frog : 81 %
Accuracy of horse : 79 %
Accuracy of  ship : 81 %
Accuracy of truck : 84 %
```

[ ]:

1.10 Tasks

[ ]:

1.10.1 MORE Tensorboard:

- create separate plots with 'epoch vs. loss' (in addition to the current 'interation vs. loss')
- create separate plots with 'epoch vs. accuracy(train)'
- create separate plots with 'epoch vs. accuracy(test)' (for this create separate tag 'Test')

1.10.2 IMPROVE THE MODEL:

Experiment and try to get the best performance that you can on CIFAR-10 using a ConvNet. Here are some ideas to get you started: (tips based on 'cs231n course')

Things to try:

- Filter size
- Number of filters

- Batch normalization
- Network architecture. Some good architectures to try include:
  - [conv-relu-pool]xN - conv - relu - [FC]xM
  - [conv-relu-pool]XN - [FC]XM
  - [conv-relu-conv-relu-pool]xN - [FC]xM

Tips for training For each network architecture that you try, you should tune the learning rate and regularization strength. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the course-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.

Going above and beyond If you are feeling adventurous there are many other features you can implement to try and improve your performance.

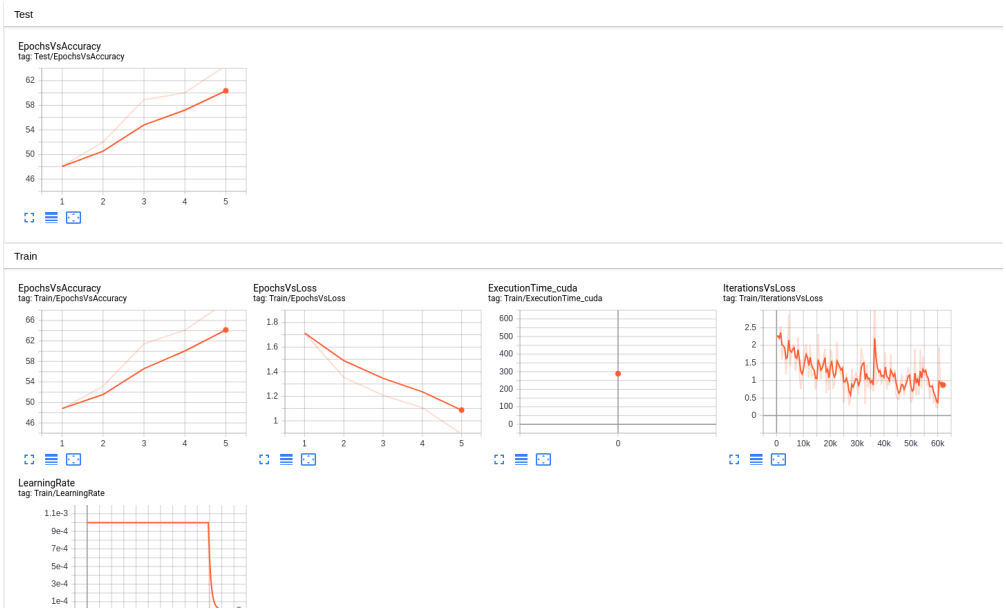
- Alternative update steps: SGD+momentum, RMSprop, and Adam; you could try alternatives like AdaGrad or AdaDelta.
- Alternative activation functions such as leaky ReLU, parametric ReLU, or MaxOut.
- Model ensembles
- Data augmentation

What to expect At the very least, you should be able to train a ConvNet that gets at least 65% accuracy on the validation set. This is just a lower bound - if you are careful it should be possible to get accuracies much higher than that!

1.10.3 CPU vs GPU:

compare time of training for CPU, GPU for different models

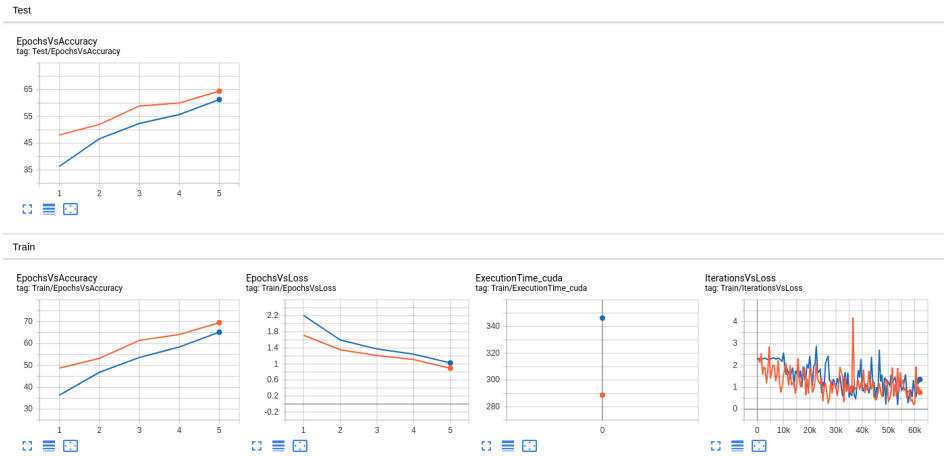
1.10.4 MORE Tensorboard:



title

1.10.5 IMPROVE THE MODEL:

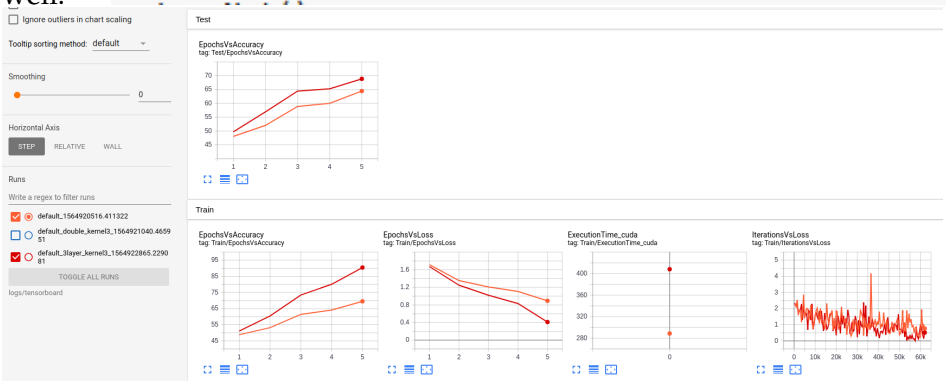
1) I tried to change the kernel size from 5 to 3 and to save the dimensions I used 4 layers instead of 2. Results became worse:



2) I tried to change the kernel size from 5 to 3 and changed dimensions as

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 3)
        self.pool1 = nn.MaxPool2d(2, 1)
        self.conv2 = nn.Conv2d(6, 16, 3)
        self.conv3 = nn.Conv2d(16, 25, 3)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(25 * 12 * 12, 1000)
        self.fc2 = nn.Linear(1000, 100)
        self.fc3 = nn.Linear(100, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool1(F.relu(self.conv2(x)))
        x = self.pool2(F.relu(self.conv3(x)))
        x = x.view(-1, 25 * 12 * 12)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net_name = "default_3layer_kernel3"
```

well: Results:



As seen from results, training accuracy is around 90 and test 68. It performed better than default net (64 vs 68 accuracy), but it overfitted, as training accuracy is around 90

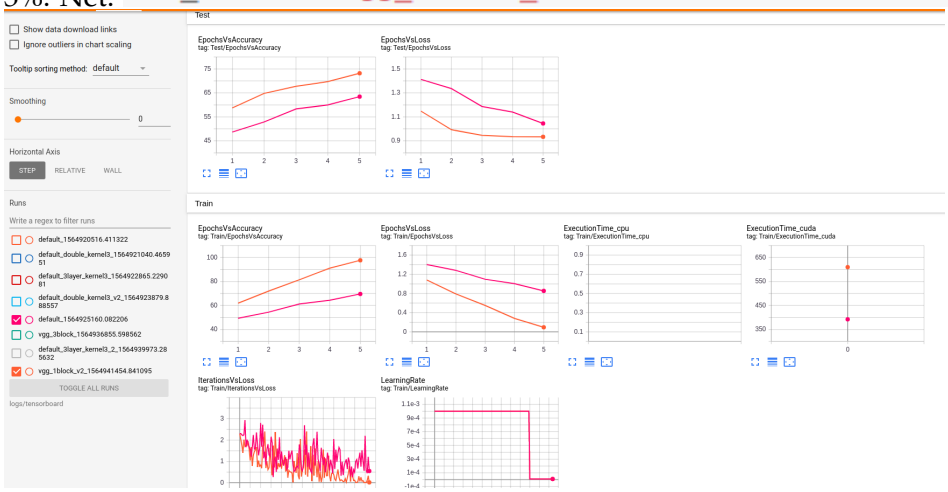
To track the overfitting I added EpochsVsLoss for Test set as well. To improve the model I used the guide for tuning the CNN: https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/

3) VGG with only one block has much better results compared to the default: accuracy

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 16 * 16, 1000)
        self.fc2 = nn.Linear(1000, 100)
        self.fc3 = nn.Linear(100, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 32 * 16 * 16)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net_name = "vgg_1block_v2"
```

around 73%. Net:

Results:



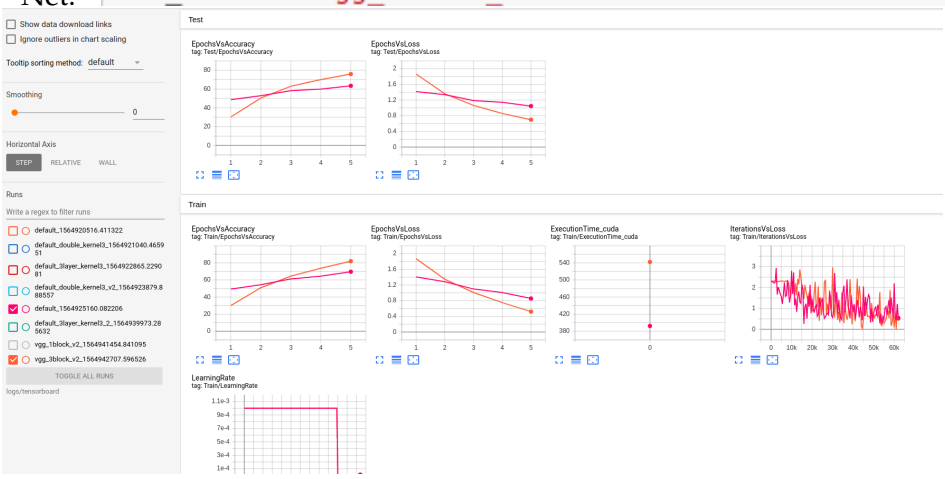
Let's try add more blocks

3) VGG with three blocks has even better results than with 1 block: accuracy ~

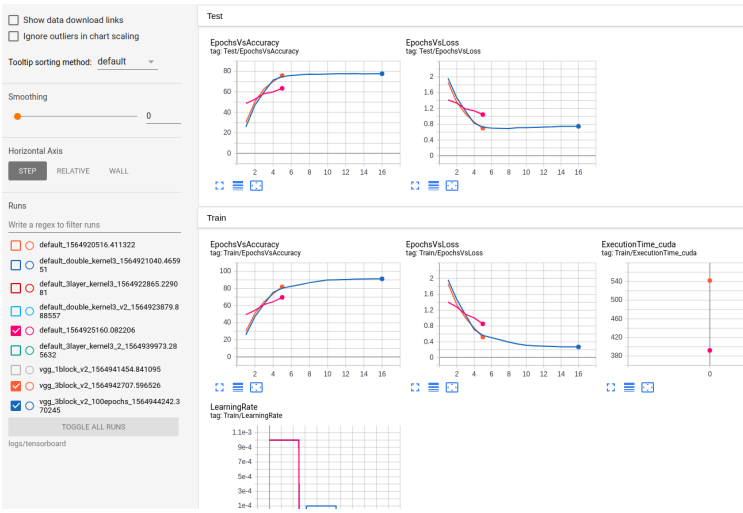
75%. Net:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv4 = nn.Conv2d(64, 64, 3, padding=1)
        self.conv5 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv6 = nn.Conv2d(128, 128, 3, padding=1)
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = self.pool(F.relu(self.conv4(x)))
        x = F.relu(self.conv5(x))
        x = self.pool(F.relu(self.conv6(x)))
        x = x.view(-1, 128 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net_name = "vgg_3block_v2"
```

Results:



Let's increase number of epochs and see how architecture with 3 blocks can perform Accuracy of the model stopped increasing after 77.67%

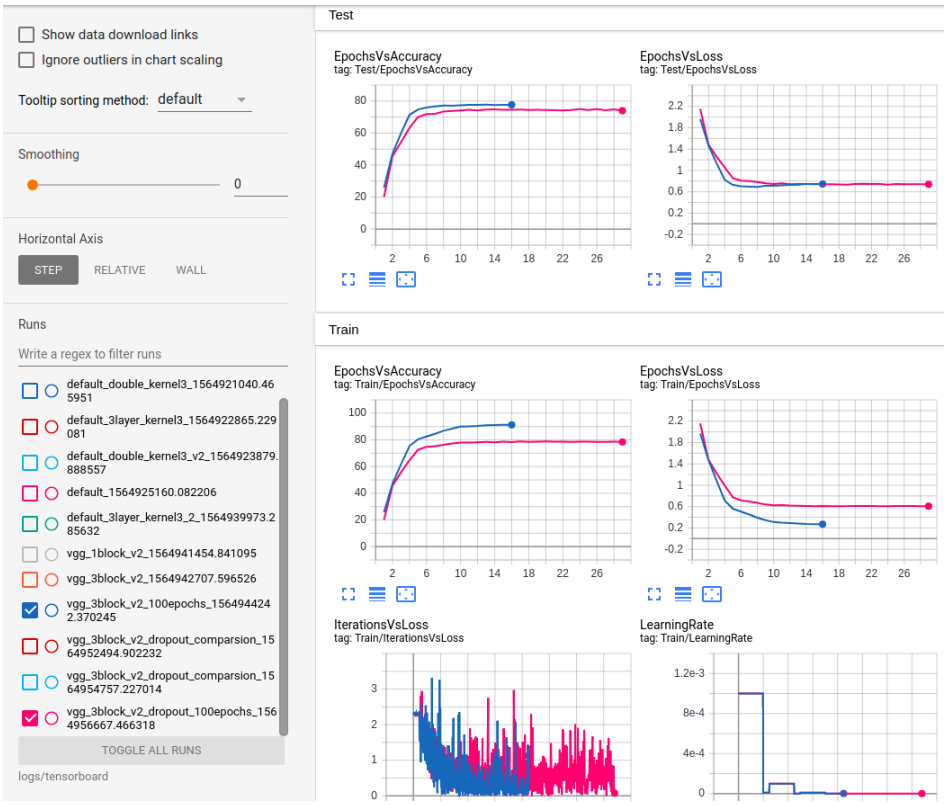


Starting from the 6th epoch model starts overfitting
Let's add Dropout to prevent overfitting

4) VGG with three blocks and dropout did not overfit as previous version, but has a little bit worse results Net:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)
        self.dropout1 = nn.Dropout(0.2)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv4 = nn.Conv2d(64, 64, 3, padding=1)
        self.dropout2 = nn.Dropout(0.2)
        self.conv5 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv6 = nn.Conv2d(128, 128, 3, padding=1)
        self.dropout3 = nn.Dropout(0.2)
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.dropout4 = nn.Dropout(0.2)
        self.fc2 = nn.Linear(512, 128)
        self.dropout5 = nn.Dropout(0.2)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = self.dropout1(x)

        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = self.dropout2(x)

        x = F.relu(self.conv5(x))
        x = F.relu(self.conv6(x))
        x = self.pool(x)
        x = self.dropout3(x)

        x = x.view(-1, 128 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = self.dropout4(x)
        x = F.relu(self.fc2(x))
        x = self.dropout5(x)
        x = self.fc3(x)
        return x

net name = "vgg 3block v2 dropout 100epochs"
```

Results:



1.10.6 CPU vs GPU:

Last model with dropout and epochs=5 has execution time 2135 seconds on CPU and 606 seconds on GPU

[ ]: