

Problem Set 3

Designed by Xide Xia, with help from Ashok Cutkosky and Brian Kulis.

This assignment will introduce you to:

1. Train a CNN model.
2. Save/restore and fine-tune with model weights.
3. Tensorboard toolkit (optional).
4. Neural style transfer learning.

This code has been tested on Colab.

Preamble

To run and solve this assignment, you need an interface to edit and run ipython notebooks (.ipynb files). The easiest way to complete this assignment is to use Google Colab. You can just copy the assignment notebook to your google drive and open it, edit it and run it on Google Colab. All libraries you need are pre-installed on Colab.

Local installation

The alternative is to have a local installation, although we do not recommend it. If you are working on Google Colab, feel free to skip to the next section "More instructions". We recommend using virtual environments for all your installations. Following is one way to set up a working environment on your local machine for this assignment, using [Anaconda](#):

- Download and install Anaconda following the instructions [here](#)
- Create a conda environment using `conda create --name dl_env python=3` (You can change the name of the environment instead of calling it `dl_env`)
- Now activate the environment using: `conda activate dl_env`
- Install jupyter lab, which is the [jupyter project's](#) latest notebook interface: `pip install jupyterlab`. You can also use the classic jupyter notebooks and there isn't any difference except the interface.
- Install other necessary libraries. For this assignment you need `numpy`, `scipy`, [pytorch](#) and `matplotlib`, all of which can be installed using: `pip install <lib_name>`. Doing this in the environment, would install these libraries for `dl_env`. You can also use `conda install`.
- Now download the assignment notebook in a local directory and launching `jupyter lab` in the same directory should open a jupyter lab session in your default browser, where you can

open and edit the ipython notebook.

- For deactivating the environment when you are done with it, use : `conda deactivate`.

For users running a Jupyter server on a remote machine :

- Launch Jupyter lab on the remote server (in the directory with the homework ipynb file) using :
`jupyter lab --no-browser --ip=0.0.0.0`
- To access the jupyter lab interface on your local browser, you need to set up ssh port forwarding. This can be done by running : `ssh -N -f -L localhost:8888:localhost:8888 <remoteuser>@<remotehost>`. You can now open `localhost:8888` on your local browser to access jupyter lab. This assumes you are running jupyter lab on its default port 8888 on the server.
- Check "Making life easy" section at the end of [this post](#) to find how to add functions to your bash run config to do this more easily each time. The post mentions functions for jupyter notebook, but just replace those with jupyter lab if you are using that interface.

The above instructions specify one way of working on the assignment. You can use other virtual environments/ipython notebook interfaces etc. (**not recommended**).

More instructions

If you are new to Python or its scientific library, Numpy, there are some nice tutorials [here](#) and [here](#).

In an ipython notebook, to run code in a cell or to render [Markdown](#)+[LaTeX](#) press `Ctrl+Enter` or `[>|]` (like "play") button above. To edit any code or text cell (double) click on its content. To change cell type, choose "Markdown" or "Code" in the drop-down menu above.

To enter your solutions for the written questions, put down your derivations into the corresponding cells below using LaTeX. Show all steps when proving statements. If you are not familiar with LaTeX, you should look at some tutorials and at the examples listed below between $..$. We will not accept handwritten solutions.

Put your solutions into boxes marked with **[double click here to add a solution]** and press `Ctrl+Enter` to render text. (Double) click on a cell to edit or to see its source code. You can add cells via `+` sign at the top left corner.

Submission instructions: please upload your completed solution file (having run all code cells and rendered all markdown/Latex) to the Google Form posted on Piazza by the due date (see Schedule for due dates and late policy).

Note: `vector` stands for column vector below.

▼ Problem 1: Convolutional Networks (40 points)

In this part, we will experiment with CNNs in PyTorch. You will need to read the documentation of the functions provided below to understand how they work.

GPU Training. Smaller networks will train fine on a CPU, but you may want to use GPU training for this part of the homework. You can run your experiments on Colab's GPUs or on BU's [Shared Computing Cluster \(SCC\)](#). You may find this SCC tutorial helpful: [SCC tutorial](#). To get access to a GPU on Colab, go to `Edit->Notebook Settings` in the notebook and set the hardware accelerator to "GPU".

▼ 1.1 Training a CNN on SVHN

In last homework, we implemented a 2-layer MLP network on the [SVHN Dataset](#).

The SVHN dataset consists of photos of house numbers, collected automatically using Google's Street View. Recognizing multi-digit numbers in photographs captured at street level is an important component of modern-day map making. Google's Street View imagery contains hundreds of millions of geo-located 360 degree panoramic images. The ability to automatically transcribe an address number from a geo-located patch of pixels and associate the transcribed number with a known street address helps pinpoint, with a high degree of accuracy, the location of the building it represents. Below are example images from the dataset. Note that for this dataset, each image (32x32 pixels) has been cropped around a single number in its center, which is the number we want to classify.

SVHN images

In this homework, we will create and train a convolutional network (CNN) on the [SVHN Dataset](#).

```
import torch
import torchvision
import torchvision.transforms as transforms
```

▼ 1.1.0 Data Download

First, download the SVHN dataset using `torchvision` and display the RGB images in the first batch. Take a look at the [Training a Classifier](#) tutorial for an example. Follow the settings used there, such as the normalization, batch size of 4 for the `torch.utils.data.DataLoader`, etc.

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

trainset = torchvision.datasets.SVHN(root='./data', split='train',
```

```

transform=transform, download=True)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                           shuffle=True, num_workers=2)
testset = torchvision.datasets.SVHN(root='./data', split='test',
                                     transform=transform, download=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                          shuffle=False, num_workers=2)

classes = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')

    Using downloaded and verified file: ./data/train_32x32.mat
    Using downloaded and verified file: ./data/test_32x32.mat

```

```

import matplotlib.pyplot as plt
import numpy as np

# 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
dataiter = iter(trainloader)
images, labels = dataiter.next()

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

```

▼ 1.1.1 CNN Model

Next, we will train a CNN on the data. We have defined a simple CNN for you with two convolutional layers and two fully-connected layers below.

```

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, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)  # flatten features

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()

```

Instantiate the cross-entropy loss criterion, and an SGD optimizer from the `torch.optim` package with learning rate `.001` and momentum `.9`. You may also want to enable GPU training using `torch.device()`.

```

# solution here

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

net.to(device)

cuda:0
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=32, bias=True)
  (fc3): Linear(in_features=32, out_features=10, bias=True)
)

```

▼ 1.1.2 Training

Write the training loop that makes two full passes through the dataset (two epochs) using SGD. Your batch size should be 4.

Go slack off for a while...



```
for epoch in range(2): # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
```

```

        (epoch + 1, i + 1, running_loss / 2000))
    running_loss = 0.0

print('Finished Training')

[1, 2000] loss: 0.326
[1, 4000] loss: 0.343
[1, 6000] loss: 0.346
[1, 8000] loss: 0.364
[1, 10000] loss: 0.370
[1, 12000] loss: 0.362
[1, 14000] loss: 0.361
[1, 16000] loss: 0.340
[1, 18000] loss: 0.343
[2, 2000] loss: 0.321
[2, 4000] loss: 0.319
[2, 6000] loss: 0.341
[2, 8000] loss: 0.338
[2, 10000] loss: 0.330
[2, 12000] loss: 0.327
[2, 14000] loss: 0.347
[2, 16000] loss: 0.320
[2, 18000] loss: 0.327
Finished Training

```

▼ 1.1.3 Test Accuracy

Load the test data (don't forget to move it to GPU if using). Make predictions on it using the trained network and compute the accuracy. You should see an accuracy of above 80%.

```

def test_on_SVHN(net, testloader):
    dataiter = iter(testloader)
    images, labels = dataiter.next()[0].to(device), dataiter.next()[1].to(device)
    #print images
    imshow(torchvision.utils.make_grid(images.cpu()))
    print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(4)))
    outputs = net(images)
    _, predicted = torch.max(outputs, 1)
    print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}' for j in range(4)))
    correct = 0
    total = 0
    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            # calculate outputs by running images through the network
            outputs = net(images)
            # the class with the highest energy is what we choose as prediction
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

```

```

print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}
# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1
# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
return accuracy

acc = test_on_SVHN(net, testloader)
print('Accuracy of the network on the 10000 test images: %d %%' % (acc))

```

▼ 1.2. Understanding the CNN Architecture

Explain the definition of the following terms. What are the trade-offs of various choices in each setting?

- Stride

- Padding
- Non-linearity
- Pooling
- Loss function
- Optimizer
- Learning rate
- Momentum

Your answer:

Stride: The distance between the receptive fields of adjacent neurons in a convolutional layer. A smaller stride results in a larger, more detailed feature map, while a larger stride results in a smaller, less detailed feature map.

Padding: The number of pixels added to the sides of an image to make it fit the size of the convolutional filter. Padding can improve the performance of a convolutional neural network by making the feature map size more consistent and avoiding boundary effects.

Non-linearity: A function that allows a neural network to learn non-linear relationships between input and output. Common non-linearities include RELU and sigmoid functions.

Pooling: A method of reducing the dimensionality of a convolutional feature map by combining the values of adjacent neurons. Pooling can improve the performance of a convolutional neural network by reducing the amount of data that needs to be processed and avoiding overfitting.

Loss function: A function that quantifies the error of a neural network. The loss function is used to train the neural network by adjusting the weights of the neurons.

Optimizer: An algorithm that adjusts the weights of a neural network to minimize the loss function. Common optimizers include stochastic gradient descent (SGD) and Adam.

Learning rate: The step size of the optimizer. A smaller learning rate results in a slower, but more precise training process. A larger learning rate results in a faster, but less precise training process.

Momentum: A parameter that determines how much the optimizer relies on the previous training iteration. A larger momentum results in a smoother training process, while a smaller momentum results in a more erratic training process.

▼ 1.3. Improving SVHN accuracy

We will now vary the architecture and training hyper-parameters of the network to try and achieve a higher accuracy on the SVHN dataset. Note that human performance on this dataset achieves around a 2% error, as reported in

- *Convolutional Neural Networks Applied to House Numbers Digit Classification*, Pierre Sermanet, Soumith Chintala, Yann LeCun [[pdf](#)]

You may want to read the above paper, as it provides the first exploration of deep learning for this problem. It is possible to achieve an error of less than 2% with modern deep learning methods, see for example this paper:

- *Batch-normalized Maxout Network in Network*, by Chang and Chen [[pdf](#)]

We do not expect you to achieve such results, but you should be able to improve on the initial accuracy we got and increase it to above 90%. It is more important that you try varying different architecture and training settings to understand how they affect (or not) the results.

1.3.1 Architecture

First, try to vary the structure of the network. For example, you can still use two convolutional layers, but vary their parameters. You can also try adding more convolutional, pooling and/or fully-connected (FC) layers.

Keep careful track of performance as a function of architecture changes using a table or a plot. For example, you can report the final test accuracy on 3 different numbers of channels (filters), 3 different sizes of kernels, 3 different strides, and 3 different dimensions of the final fully connected layer, using a table like the one below. Each time when you vary one parameter, you should **keep the others fixed at the original value**.

# of Filter	Accuracy
/	/
/	/
/	/
Kernel size	Accuracy
/	/
/	/
/	/
Stride	Accuracy
/	/
/	/
/	/
FC size	Accuracy
/	/
/	/
/	/

Explain your results. Note, you're welcome to decide how many training epochs to use, but do report the number you used and keep it the same for all architecture changes (as well as other training

hyper-parameters). Be careful not to change more than one thing between training/test runs, otherwise you will not know which of the multiple changes caused the results to change.

Please implement your experiments in a separate cell, DO NOT change your codes in Q1.1 for this question. During submission, you are Not required to submit any code for this question.

```
class Modified_Net(nn.Module):
    def __init__(self, fc_size):
        super(Modified_Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 10)
        self.fc4 = nn.Linear(10, fc_size)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)  # flatten features

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = self.fc4(x)
        return x

new_net1 = Modified_Net(5)
new_net2 = Modified_Net(2)
new_net3 = Modified_Net(1)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

new_net1.to(device)
new_net2.to(device)
new_net3.to(device)

def train(epochs):
    for epoch in range(epochs):  # loop over the dataset multiple times

        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data[0].to(device), data[1].to(device)

            # zero the parameter gradients
            optimizer.zero_grad()
```

```

# forward + backward + optimize
outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()

# print statistics
running_loss += loss.item()
if i % 2000 == 1999:    # print every 2000 mini-batches
    print('[%d, %5d] loss: %.3f' %
          (epoch + 1, i + 1, running_loss / 2000))
    running_loss = 0.0

print('Finished Training')

print("Unnfinished")

Unnfinished

```

solution here

▼ 1.3.2 Training Hyper-Parameters

Repeat the process for training hyper-parameters, exploring at least three of the following:

- training iterations, optionally with early stopping
- learning rate
- momentum
- optimizer
- initialization
- dropout
- batch normalization
- dataset augmentation

Document your results with tables or figures, and explain what happened. You may want to use Tensorboard (see Problem 2 below) but this is optional.

What is the best accuracy you were able to achieve on the test set, and which factors contributed the most to the improvement? (A good result should be above 85%.)

Please implement your experiments in a separate cell, DO NOT change your codes in Q1.1 for this question. During submission, you are Not required to submit any code for this question.

```
# solution here
```

solution here

▼ 1.3.3 The power of CNN

Compare to the results you got from HW2 Q4.6.3, which model learns better on image dataset - MLP or CNN? Explain it.

Hint: You can discuss on accuracy, runtime, number of parameters, etc.

```
# solution here
```

▼ Problem 2: Tensorboard (Optional)

Explore your network using Tensorboard. Tensorboard is a nice tool for visualizing how your network's training is progressing. The following tutorial provides an introduction to Tensorboard

- [Visualizing models, data and training with Tensorboard](#)

For using tensorboard in colab, run the following cell and it should open a tensorboard interface in the output of the cell.

Double-click (or enter) to edit

```
%load_ext tensorboard
%tensorboard --logdir logs
```

▼ Problem 3: Save and restore model weights (30 points)

In this section you will learn to save the weights of a trained model, and to load the weights of a saved model. This is really useful when we would like to load an already trained model in order to continue training or to fine-tune it. Often times we save “snapshots” of the trained model as training progresses in case the training is interrupted, or in case we would like to fall back to an earlier model, this is called snapshot saving.

▼ 3.1 Saving and Loading Weights

In this section you will learn how to [save and load pytorch models for inference](#).

3.1.1 State_dict

In PyTorch, the learnable parameters (i.e. weights and biases) of an `torch.nn.Module` model are contained in the model's parameters (accessed with `model.parameters()`). A `state_dict` is simply a Python dictionary object that maps each layer to its parameter tensor. Because `state_dict` objects

are Python dictionaries, they can be easily saved, updated, altered, and restored, adding a great deal of modularity to PyTorch models and optimizers.

Print out the keys of `state_dict` of the model you trained in Q1.1. (Note `state_dict` is a property of the module object.)

```
# solution here
```

```
print(list(net.state_dict().keys()))
```

```
['conv1.weight', 'conv1.bias', 'conv2.weight', 'conv2.bias', 'fc1.weight', 'fc1.b
```

▼ 3.1.2 Save state_dict

Save the `state_dict` of the model in Q1.1.4 with the `torch.save()` function to a local path.

```
# solution here
```

```
PATH = './SVHN_net.pth'
torch.save(net.state_dict(), PATH)
```

▼ 3.1.3 Load state_dict

Now let's initiate `net2` which has the same structure, and load the weights you saved to `net2` by using `load_state_dict()`.

```
net2 = Net().to(device)
```

```
## -- ! code required
net2.load_state_dict(torch.load(PATH))
```

```
net2.eval()
```

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=32, bias=True)
  (fc3): Linear(in_features=32, out_features=10, bias=True)
)
```

Test `net2`'s performance on SVHN.

```
# solution here
```

```
acc2 = test_on_SVHN(net2, testloader)
print('Accuracy of the network on the 10000 test images: %d %%' % (acc2))
```

3.2 Fine-tune a pre-trained model on CIFAR-10

[CIFAR-10](#) is another popular benchmark for image classification.



▼ 3.2.0 Data Download

Similar to Q1.1.2, download the CIFAR-10 dataset using `torchvision` and display the RGB images in the first batch.

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

cifar10_trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                                transform=transform, download=True)
cifar10_trainloader = torch.utils.data.DataLoader(cifar10_trainset, batch_size=4,
                                                  shuffle=True, num_workers=2)
cifar10_testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                                transform=transform, download=True)
cifar10_testloader = torch.utils.data.DataLoader(cifar10_testset, batch_size=4,
```



```
shuffle=False, num_workers=2)
```

```
# get some random training images
cifar10_dataiter = iter(cifar10_trainloader)
images, labels = cifar10_dataiter.next()

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

▼ 3.2.1 Load state_dict partially

Let's define `net_cifar = Net()`, and only load selected weights in `selected_layers`.

```
net_cifar = Net().to(device)
selected_layers = ['conv1.weight', 'conv1.bias', 'conv2.weight', 'conv2.bias', 'fc1.w

## -- ! code required
model_dict = torch.load(PATH)
updated_dict = {k: v for k,v in model_dict.items() if k in selected_layers}
model_dict.update(updated_dict)
net_cifar.load_state_dict(model_dict)
print(list(net_cifar.state_dict().keys()))

['conv1.weight', 'conv1.bias', 'conv2.weight', 'conv2.bias', 'fc1.weight', 'fc1.l

print(np.array_equal(net_cifar.state_dict()['conv1.weight'].cpu().numpy(), net.state_c
print(np.array_equal(net_cifar.state_dict()['fc3.weight'].cpu().numpy(), net.state_dic

True
True
```

▼ 3.2.2 Fine-tune net_cifar on CIFAR-10

Fine-tune the `net_cifar` on CIFAR-10 (you need to train and test the accuracy of `net_cifar` on CIFAR10 test set instead of on SVHN test set), and show the plot of training loss.

```

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net_cifar.parameters(), lr=0.001, momentum=0.9)

losses = []

for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs, labels = inputs.cuda(), labels.cuda()

        # zero parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            losses.append(running_loss/2000)
            running_loss = 0.0

print('Finished Training')

PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.cuda(), labels.cuda()
        outputs = net_cifar(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')

[1, 2000] loss: 0.314
[1, 4000] loss: 0.318
[1, 6000] loss: 0.319
[1, 8000] loss: 0.321

```

```
[1, 10000] loss: 0.327
[1, 12000] loss: 0.315
[1, 14000] loss: 0.314
[1, 16000] loss: 0.322
[1, 18000] loss: 0.309
[2, 2000] loss: 0.312
[2, 4000] loss: 0.317
[2, 6000] loss: 0.302
[2, 8000] loss: 0.309
[2, 10000] loss: 0.320
[2, 12000] loss: 0.319
[2, 14000] loss: 0.330
[2, 16000] loss: 0.322
[2, 18000] loss: 0.331
Finished Training
Accuracy of the network on the 10000 test images: 86 %
```

```
plt.figure()
plt.plot(losses, 'g', label='training losses')

plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.legend()
plt.grid()

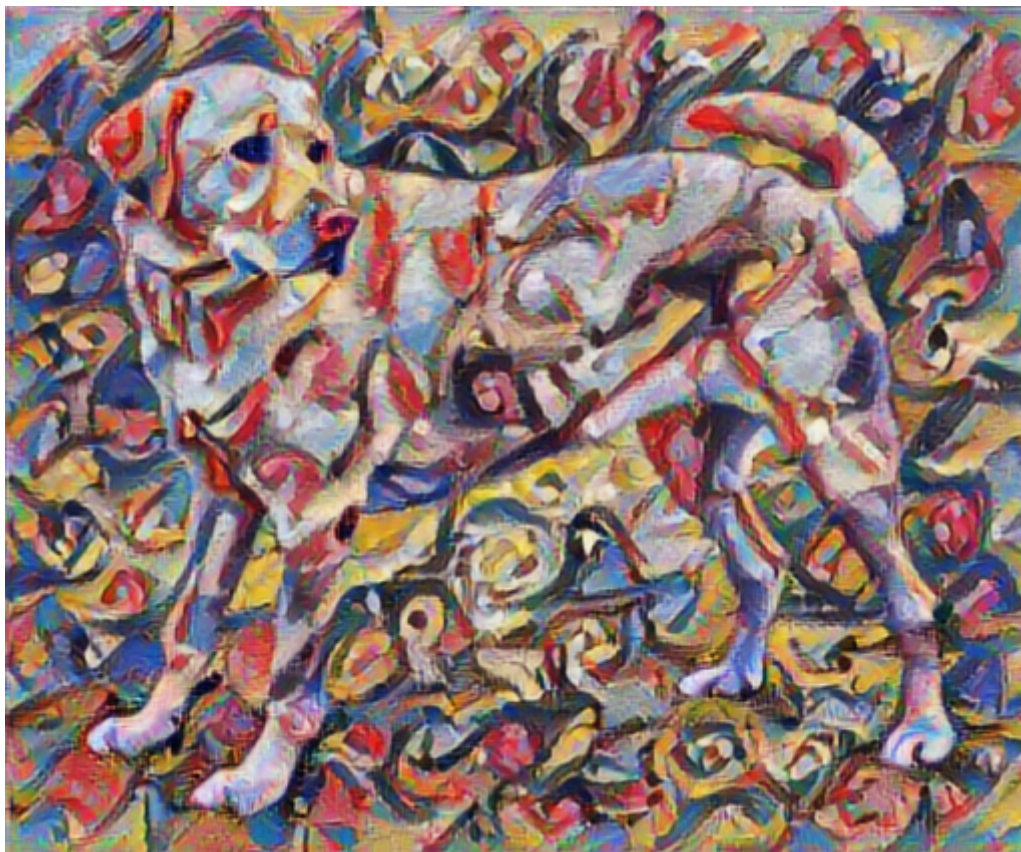
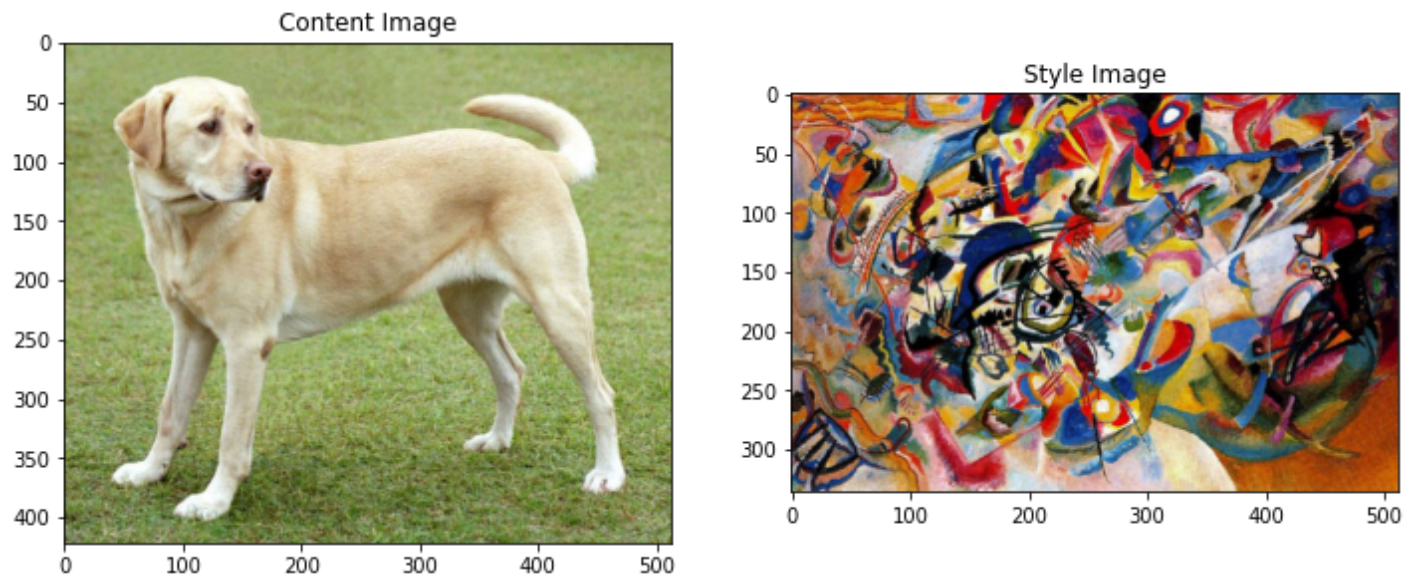
plt.show()
```

▼ Problem 4: Neural style transfer (30 points)

In this problem, we will use deep learning to compose one image in the style of another image. This is known as [neural style transfer](#) and the technique is outlined in paper [A Neural Algorithm of Artistic Style \(Gatys et al.\)](#). Please read the paper before you proceed with Problem 4.

Neural style transfer is an optimization technique used to take two images—a content image and a style reference image (such as an artwork by a famous painter)—and blend them together so the output image looks like the content image, but “painted” in the style of the style reference image.

This is implemented by optimizing the output image to match the content statistics of the content image and the style statistics of the style reference image. These statistics are extracted from the images using a convolutional network.



```
from __future__ import print_function
```

```

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

from PIL import Image
import matplotlib.pyplot as plt

import torchvision.transforms as transforms
import torchvision.models as models

import copy

```

▼ 4.0 Visualize the inputs

We provide two images, `starry.jpg` and `golden.jpg`, for style and content input respectively. To save runtime, we downscale the images to (128,128). You are welcome to play with your own inputs at any resolution scale (note a larger resolution requires more runtime). To upload files to your colab notebook, you can click on `files` on the left side of your notebook then choose upload.

```

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# desired size of the output image
imgsize = (128,128)

loader = transforms.Compose([
    transforms.Resize(imgsize), # scale imported image
    transforms.ToTensor()]) # transform it into a torch tensor

def image_loader(image_name):
    image = Image.open(image_name)
    # fake batch dimension required to fit network's input dimensions
    image = loader(image).unsqueeze(0)
    return image.to(device, torch.float)

style_img = image_loader("starry.jpg")
content_img = image_loader("golden.jpg")

assert style_img.size() == content_img.size(), \
    "we need to import style and content images of the same size"

unloader = transforms.ToPILImage() # reconvert into PIL image

plt.ion()

```

```
def imshow(tensor, title=None):
    image = tensor.cpu().clone() # we clone the tensor to not do changes on it
    image = image.squeeze(0)      # remove the fake batch dimension
    image = unloader(image)
    plt.imshow(image)
    if title is not None:
        plt.title(title)
    plt.pause(0.001) # pause a bit so that plots are updated

plt.figure()
imshow(style_img, title='Style Image')

plt.figure()
imshow(content_img, title='Content Image')
```

▼ 4.1 Loss functions

The cost function has two terms - a content loss term and a style loss term, both of which are explained below.

4.1.1 Content Loss

The content of an image is represented by the values of the intermediate feature maps. This is based on the intuition that images with similar content will have similar representation in the higher layers of the network. Let p and x be the original image and the image that is generated,



where P^l is the representation of the original image and F^l is the representation of the generated image in the feature maps of layer l .

Finish the ContentLoss() to match the corresponding content target representations. (hint: You can implement the ContentLoss using pytorch builtin mse_loss())

```
class ContentLoss(nn.Module):

    def __init__(self, target,):
        super(ContentLoss, self).__init__()
        self.target = target.detach()

    def forward(self, input):
        self.loss = F.mse_loss(input, self.target)
        return input
```

▼ 4.1.2 Style Loss

The style loss is implemented similarly to the content loss. It will act as a transparent layer in a network that computes the style loss of that layer. In order to calculate the style loss, we need to compute the gram matrix. A gram matrix is the result of multiplying a given matrix by its transposed matrix.



Let a and x be the original image and the image that is generated,



Here, A^l is the representation of the original image and G^l is the representation of the generated image in layer l . N_l is the number of feature maps and M_l is the size of the flattened feature map in layer l . w_l is the weight given to the style loss of layer l .

Complete the gram_matrix() to calculate a Gram matrix that includes this information and finish the StyleLoss().

```
def gram_matrix(input):
```

```

a, b, c, d = input.size()
features = input.view(a*b, c*d)
gram = torch.mm(features, features.t())
return gram.div(a*b*c*d)

class StyleLoss(nn.Module):

    def __init__(self, target):
        super(StyleLoss, self).__init__()
        self.target = gram_matrix(target).detach()

    def forward(self, input):
        G = gram_matrix(input)
        self.loss = F.mse_loss(G, self.target)
        return input

```

▼ 4.1.3 Import a pre-trained VGG-19.

Now we need to import a pre-trained neural network. We will use a 19 layer VGG network like the one used in the paper.

Import a pretrained VGG-19 from [torchvision.models](#). Make sure to set the network to evaluation mode using `.eval()`.

```
cnn = models.vgg19(pretrained=True).features.to(device).eval()
```

▼ 4.1.4 VGG-19 pre-processing

VGG networks are trained on images with each channel normalized by `mean=[0.485, 0.456, 0.406]` and `std=[0.229, 0.224, 0.225]`.

Complete `Normalization()` to normalize the image before sending it into the network.

```

cnn_normalization_mean = torch.tensor([0.485, 0.456, 0.406]).to(device)
cnn_normalization_std = torch.tensor([0.229, 0.224, 0.225]).to(device)

# create a module to normalize input image so we can easily put it in a nn.Sequential
class Normalization(nn.Module):
    def __init__(self, mean, std):
        super(Normalization, self).__init__()
        # self.mean = mean.clone().detach()
        # self.std = std.clone().detach()
        self.mean = torch.tensor(mean).view(-1, 1, 1)
        self.std = torch.tensor(std).view(-1, 1, 1)

    def forward(self, img):
        # normalize img

```



```

# n_img = F.normalize(img)
# n_mean = F.normalize(self.mean)
# n_std = F.normalize(self.std)
# normalized_img = (n_img - n_mean) / n_std
# return normalized_img
return (img - self.mean) / self.std

```

▼ 4.1.5 Get content/style representations

Choose intermediate layers from the network to represent the style and content of the image. Use the selected intermediate layers of the model to get the content and style representations of the image. In this problem, you are using the VGG19 network architecture, a pretrained image classification network. These intermediate layers are necessary to define the representation of content and style from the images.

Complete the `get_style_model_and_losses()` so you can easily extract the intermediate layer values.

```

# { display-mode: "both" }
def get_style_model_and_losses(cnn, normalization_mean, normalization_std,
                              style_img, content_img, content_layers, style_layers):
    cnn = copy.deepcopy(cnn)

    # normalization module
    normalization = Normalization(normalization_mean, normalization_std).to(device)

    # just in order to have an iterable access to or list of content/syle losses
    content_losses = []
    style_losses = []

    # assuming that cnn is a nn.Sequential, so we make a new nn.Sequential
    # to put in modules that are supposed to be activated sequentially
    model = nn.Sequential(normalization)

    i = 0 # increment every time we see a conv
    for layer in cnn.children():
        if isinstance(layer, nn.Conv2d):
            i += 1
            name = 'conv_{}'.format(i)
        elif isinstance(layer, nn.ReLU):
            name = 'relu_{}'.format(i)
            layer = nn.ReLU(inplace=False)
        elif isinstance(layer, nn.MaxPool2d):
            name = 'pool_{}'.format(i)
        elif isinstance(layer, nn.BatchNorm2d):
            name = 'bn_{}'.format(i)
        else:
            raise RuntimeError('Unrecognized layer: {}'.format(layer.__class__.__name_

    model.add_module(name, layer)

```

```

if name in content_layers:
    # add content loss:
    target = model(content_img).detach()
    content_loss = ContentLoss(target)
    model.add_module("content_loss_{}".format(i), content_loss)
    content_losses.append(content_loss)

if name in style_layers:
    # add style loss:
    target_feature = model(style_img).detach()
    style_loss = StyleLoss(target_feature)
    model.add_module("style_loss_{}".format(i), style_loss)
    style_losses.append(style_loss)

# now we trim off the layers after the last content and style losses
for i in range(len(model) - 1, -1, -1):
    if isinstance(model[i], ContentLoss) or isinstance(model[i], StyleLoss):
        break
model = model[:i + 1]
return model, style_losses, content_losses

```

▼ 4.2 Build the model

4.2.1 Perform the neural transfer

Finally, we must define a function that performs the neural transfer. To transfer the style of an artwork a onto a photograph p we synthesise a new image that simultaneously matches the content representation of p and the style representation of a . Thus we jointly minimise the distance of the feature representations of an initial image from the content representation of the photograph in one layer and the style representation of the painting defined on a number of layers of the Convolutional Neural Network. The loss function we minimise



where α and β are the weighting factors for content and style reconstruction, respectively.

For each iteration of the networks, it is fed an updated input and computes new losses. We will run the backward methods of each loss module to dynamically compute their gradients. The paper recommends LBFGS, but Adam works okay, too.

Complete the `run_style_transfer()`.

```

content_layers_selected = ['conv_4']
style_layers_selected = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']

def run_style_transfer(cnn, normalization_mean, normalization_std,
                      content_img, style_img, input_img, num_steps=300,
                      style_weight=1000000, content_weight=1,
                      content_layers=content_layers_selected,
                      style_layers=style_layers_selected):
    """Run the style transfer."""
    print('Building the style transfer model..')
    model, style_losses, content_losses = get_style_model_and_losses(cnn,
                             normalization_mean, normalization_std, style_img, content_img, content_layers,
                             style_layers)

    optimizer = optim.Adam([input_img.requires_grad_()], lr=0.1, eps=1e-1)

    print('Optimizing..')
    run = [0]
    while run[0] <= num_steps:
        def closure():
            # correct the values of updated input image
            with torch.no_grad():
                input_img.clamp_(0, 1)

            optimizer.zero_grad()
            model(input_img)
            style_score = 0
            content_score = 0

            for sl in style_losses:
                style_score += sl.loss
            for cl in content_losses:
                content_score += cl.loss

            style_score *= style_weight
            content_score *= content_weight

            loss = style_score + content_score
            loss.backward()

            run[0] += 1
            if run[0] % 50 == 0:
                print("run {}: ".format(run))
                print('Style Loss : {:.4f} Content Loss: {:.4f}'.format(
                    style_score.item(), content_score.item()))
                print()

            return style_score + content_score

        optimizer.step(closure)

```

```
# a last correction...
input_img.data.clamp_(0, 1)

return input_img
```

▼ 4.2.2 Test your model

Now you have done your coding tasks, let's test them!

```
content_layers_selected = ['conv_4']
style_layers_selected = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']
style_weight=1000000
input_img = content_img.clone().detach().requires_grad_(True)

output = run_style_transfer(cnn, cnn_normalization_mean, cnn_normalization_std,
                           content_img, style_img, input_img, num_steps=100, style_weight=style_weight,
                           content_layers=content_layers_selected,
                           style_layers=style_layers_selected)

plt.figure()
imshow(output, title='Output Image')

plt.ioff()
plt.show()
```

▼ 4.3 Content/style loss weight ratio

Try two different style loss weights: 5000 and 10. Discuss what you learn from the results.

```
content_layers_selected = ['conv_4']
style_layers_selected = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']
style_weight=5000
input_img = content_img.clone().detach().requires_grad_(True)

output = run_style_transfer(cnn, cnn_normalization_mean, cnn_normalization_std,
                           content_img, style_img, input_img, num_steps=100, style_weight=style_weight,
                           content_layers=content_layers_selected,
                           style_layers=style_layers_selected)

plt.figure()
imshow(output, title='Output Image')

plt.ioff()
plt.show()
```

```

content_layers_selected = ['conv_4']
style_layers_selected = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']
style_weight=10
input_img = content_img.clone().detach().requires_grad_(True)

output = run_style_transfer(cnn, cnn_normalization_mean, cnn_normalization_std,
                           content_img, style_img, input_img, num_steps=100, style_weight=style_weight,
                           content_layers=content_layers_selected,
                           style_layers=style_layers_selected)

plt.figure()
imshow(output, title='Output Image')

plt.ioff()
plt.show()

```

Solution:

Because the style_weight parameter affects the amount of influence that the style of the reference image has on the generated image, higher style_weight at 5000 that the generated image will be more similar to the style of the reference image and we can see more of Van Gough's influences on

this golden retriever. On the other hand a lower style_weight such as 10 means that the generated image will be more similar to the content of the content image.

▼ 4.4 Choose different intermediate layers

Try three different intermediate layers for style representations: conv_1, conv_3 and conv_5. Discuss what you learn from the results.

```
content_layers_selected = ['conv_4']
style_layers_selected = ['conv_1']
style_weight=1000000
input_img = content_img.clone().detach().requires_grad_(True)

output = run_style_transfer(cnn, cnn_normalization_mean, cnn_normalization_std,
                           content_img, style_img, input_img, num_steps=100, style_weight=style_weight,
                           content_layers=content_layers_selected,
                           style_layers=style_layers_selected)

plt.figure()
imshow(output, title='Output Image')

plt.ioff()
plt.show()
```

```
content_layers_selected = ['conv_4']
style_layers_selected = ['conv_3']
style_weight=1000000
input_img = content_img.clone().detach().requires_grad_(True)

output = run_style_transfer(cnn, cnn_normalization_mean, cnn_normalization_std,
                           content_img, style_img, input_img, num_steps=100, style_weight=style_weight,
                           content_layers=content_layers_selected,
                           style_layers=style_layers_selected)

plt.figure()
imshow(output, title='Output Image')

plt.ioff()
plt.show()
```

```
content_layers_selected = ['conv_4']
style_layers_selected = ['conv_5']
style_weight=1000000
```



```
input_img = content_img.clone().detach().requires_grad_(True)

output = run_style_transfer(cnn, cnn_normalization_mean, cnn_normalization_std,
                             content_img, style_img, input_img, num_steps=100, style_weight=1,
                             content_layers=content_layers_selected,
                             style_layers=style_layers_selected)

plt.figure()
imshow(output, title='Output Image')

plt.ioff()
plt.show()
```

your solution:

From the above three outputs, I can see that Conv1 has the least style transfer whereas Conv5 has the most intense style transfer. At the same time, the higher the focus on the style, the higher the losses are as well for content and style features. However, the trade off that I can see is that convolutional layers that are deeper in the network are more intense in the style attributes.

✓ 0s completed at 4:47 PM

● ×