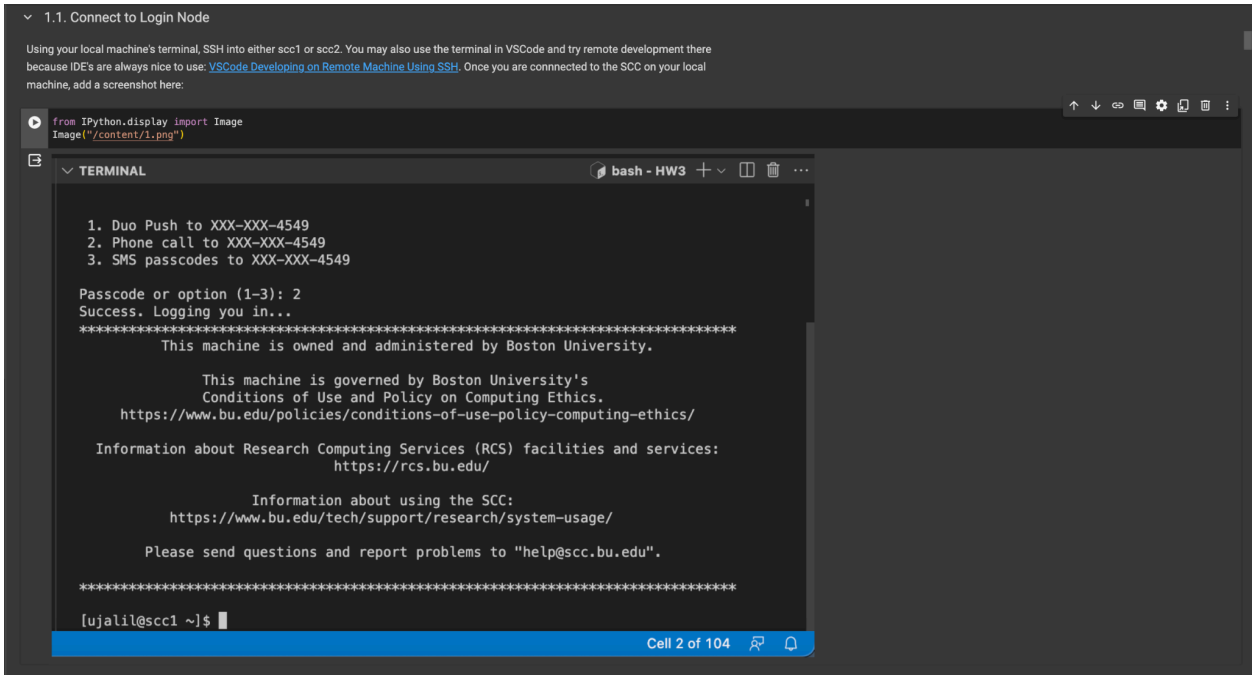
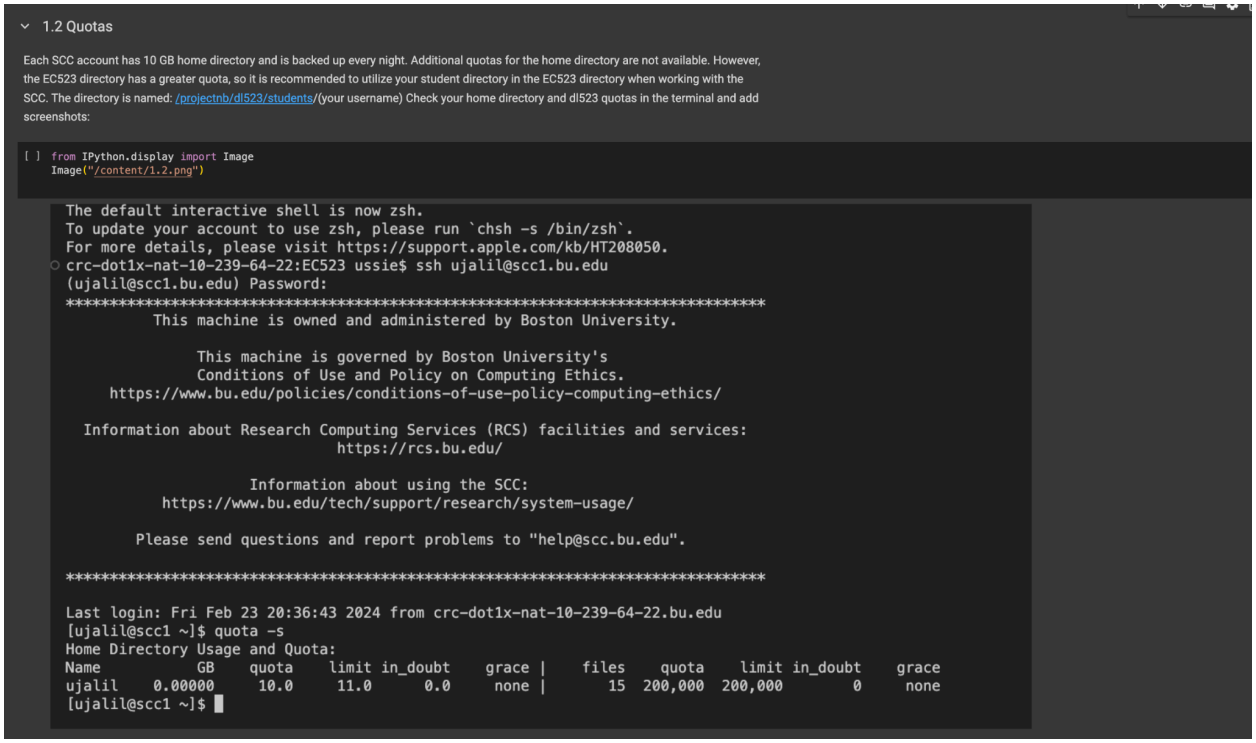
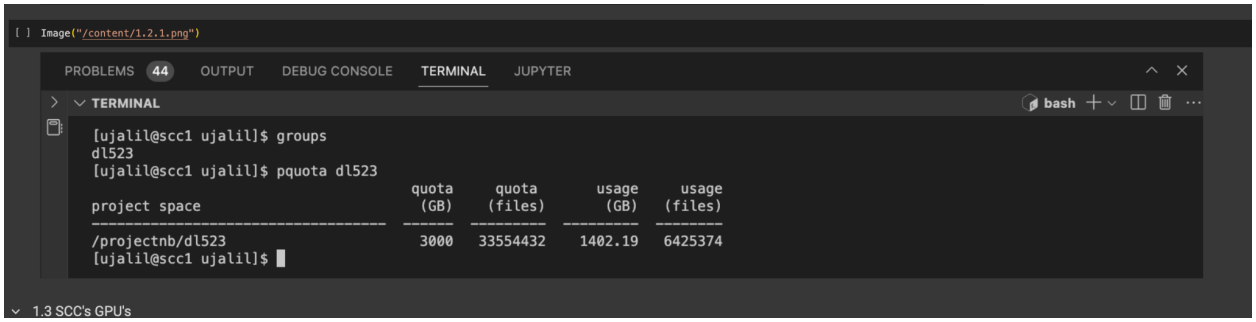
Question 1


1.1. Connect to Login Node
Using your local machine's terminal, SSH into either scc1 or scc2. You may also use the terminal in VSCode and try remote development there because IDE's are always nice to use: VSCode Developing on Remote Machine Using SSH. Once you are connnected to the SCC on your local machine, add a screenshot here:
from IPython.display import Image
Image("/content/1.png")
TERMINAL
bash - HW3
1. Duo Push to XXX-XXX-4549
2. Phone call to XXX-XXX-4549
3. SMS passcodes to XXX-XXX-4549
Passcode or option (1-3): 2
Success. Logging you in...
This machine is owned and administered by Boston University.
This machine is governed by Boston University's
Conditions of Use and Policy on Computing Ethics.
https://www.bu.edu/policies/conditions-of-use-policy-computing-ethics/
Information about Research Computing Services (RCS) facilities and services:
https://rcs.bu.edu/
Information about using the SCC:
https://www.bu.edu/tech/support/research/system-usage/
Please send questions and report problems to "help@scc.bu.edu".
[ujalil@scc1 ~]$
Cell 2 of 104


1.2 Quotas
Each SCC account has 10 GB home directory and is backed up every night. Additional quotas for the home directory are not available. However, the EC523 directory has a greater quota, so it is recommended to utilize your student directory in the EC523 directory when working with the SCC. The directory is named: /projectnb/dl523/students/(your username) Check your home directory and dl523 quotas in the terminal and add screenshots:
from IPython.display import Image
Image("/content/1.2.png")
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
crc-dot1x-nat-10-239-64-22:EC523 ussie$ ssh ujalil@scc1.bu.edu
(ujalil@scc1.bu.edu) Password:
This machine is owned and administered by Boston University.
This machine is governed by Boston University's
Conditions of Use and Policy on Computing Ethics.
https://www.bu.edu/policies/conditions-of-use-policy-computing-ethics/
Information about Research Computing Services (RCS) facilities and services:
https://rcs.bu.edu/
Information about using the SCC:
https://www.bu.edu/tech/support/research/system-usage/
Please send questions and report problems to "help@scc.bu.edu".
Last login: Fri Feb 23 20:36:43 2024 from crc-dot1x-nat-10-239-64-22.bu.edu
[ujalil@scc1 ~]$ quota -s
Home Directory Usage and Quota:
Name GB quota limit in_doubt grace | files quota limit in_doubt grace
ujalil 0.00000 10.0 11.0 0.0 none | 15 200,000 200,000 0 none
[ujalil@scc1 ~]$



1.3 SCC's GPU's

In the node, print out the available GPUs available to you on the SCC and add a screenshot. Choose and rank 3 of the GPUs and explain your ranking.

```
[ ] Image("/content/1.3.png")
```

```
[ujalil@scc1 ujalil]$ qgpus
gpu_type  total  in_use  available
--------  -----  ------  ---------
A100          5       1          4
A100-80G     24      19          5
A40          68      57         11
A6000        30      30          0
K40m         14       5          9
L40           6       2          4
P100         28      20          8
P100-16G     23       9         14
RTX6000       5       5          0
RTX8000       9       9          0
TitanV        8       0          8
TitanXp      10       0         10
V100         65      51         11
V100-32G      4       3          1
[ujalil@scc1 ujalil]$ qgpus -s
gpu_type  total  in_use  available
--------  -----  ------  ---------
A100-80G      4       4          0
A40           4       4          0
K40m          2       2          0
P100          8       8          0
V100          4       4          0
[ujalil@scc1 ujalil]$
```

I decided to rank the GPUs based on data bandwith since it indicates the GPU's ability to handle large datasets efficiently. I feel as though this is very relevant for deep learning, where fast data access is important for performance optimization.

Ranking based on data bandwidth:

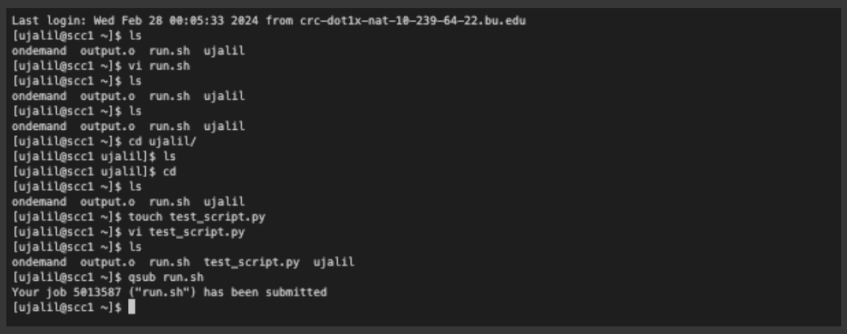
1. A100-80GB GPU (2 TB/s)
2. V100 GPU (900 GB/s)
3. P100 GPU (720 GB/s)

1.4 Running a Batch Job

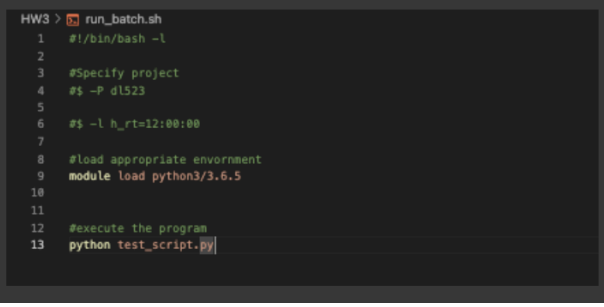
Run a batch job to execute the following Python script and store it in a text file: test_script.py

Add screenshots of your submitted bash file and the output of the code from the SCC:

```
[ ]  Image("/content/1.4.png")
```




```
Image("/content/1.41.png")
```



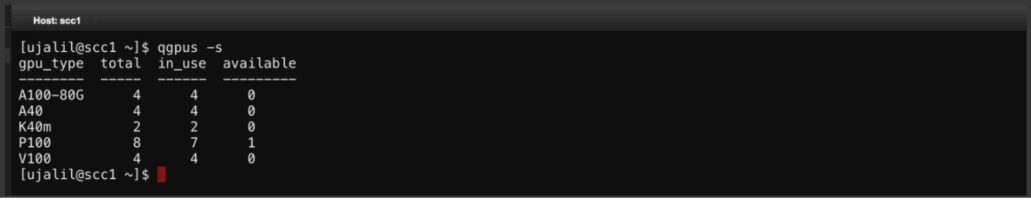
1.5 Interactive Apps

Launch a Desktop session from the SCC web browser with these specifications:

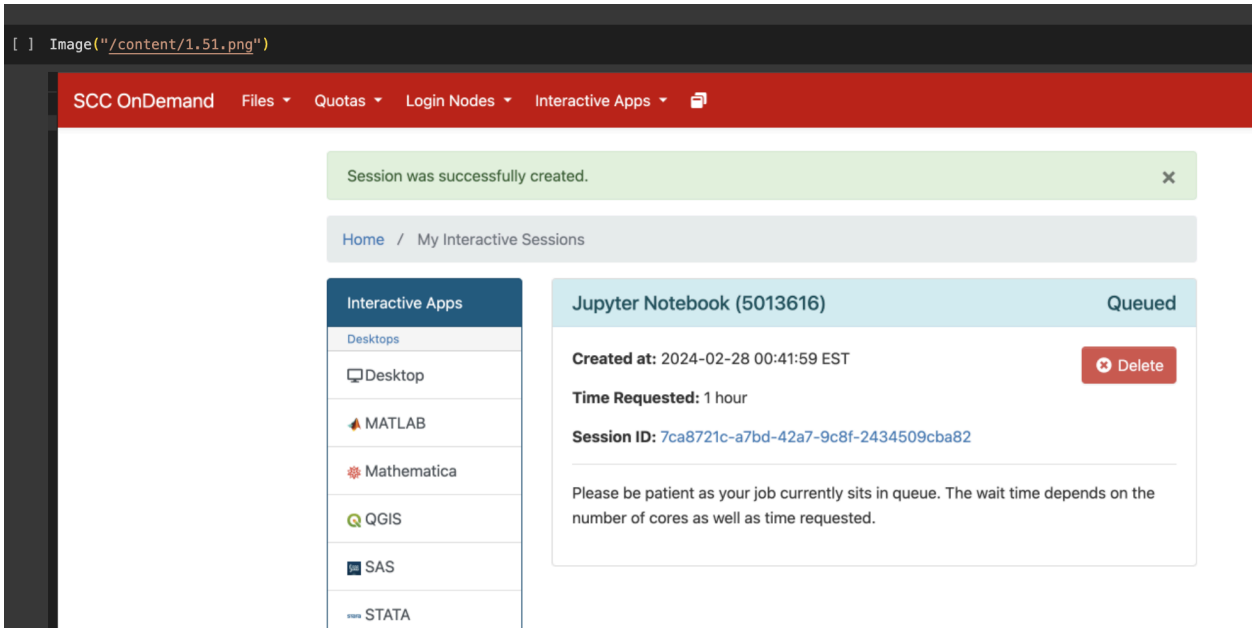
- Number of hours: 1
- Number of cores: 2
- Number of GPUs: 1
- GPU compute compatibility: 6.0 (P100 or V100)
- Project: dl523

Once your desktop session has launched, open up the terminal in the Desktop session and print out the GPU the session is using and add a screenshot:

```
[ ]  Image("/content/1.5.png")
```



```
[ ] Image("/content/1.51.png")
```



Question 2

Problem 2: Convolutional Networks (50 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.

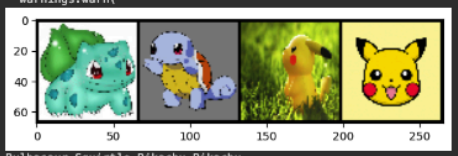
GPU Training. Smaller networks will train fine on a CPU, but you may want to use GPU training for this part of the homework. You can run your experiments on Colab's GPUs or on BU's Shared Computing Cluster (SCC). You may find this SCC tutorial helpful: SCC tutorial. To get access to a GPU on Colab, go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

```
[ ] # check GPU status
    !nvidia-smi
```

```
Sun Mar  3 20:24:53 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   68C    P8              12W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

```
# show some examples
import matplotlib.pyplot as plt
import numpy as np
import os

dir_path = '/content/drive/MyDrive/Pokemon/'
classes = os.listdir(dir_path)
print(classes)

# Get a list of all items (files and directories) in the directory
items = os.listdir(PATH_OF_DATA + "Pokemon_train")
print(PATH_OF_DATA + "Pokemon_train")

classes_labels = ['Bulbasaur', 'Charmander', 'Mewtwo', 'Pikachu', 'Psyduck', 'Squirtle']

# Convert the list of labels to a tuple
labels_tuple = tuple(classes_labels)

print('The labels are: ', labels_tuple)

# functions to show an image
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


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

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes_labels[labels[j]]:5s}' for j in range(4)))
```

```
['Pokemon_test', 'Pokemon_train']
/content/drive/MyDrive/Pokemon/Pokemon_train
The labels are:  ('Bulbasaur', 'Charmander', 'Mewtwo', 'Pikachu', 'Psyduck', 'Squirtle')
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias parameter of all the resizing transforms
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias parameter of all the resizing transforms
  warnings.warn(
```




```
Bulbasaur Squirtle Pikachu Pikachu
```

2.1.1 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(in_channels = 3, out_channels = 16, kernel_size = 3, padding = 1)
        self.conv2 = nn.Conv2d(in_channels = 16, out_channels = 32, kernel_size = 3, padding = 1)
        self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(32 * 16 * 16, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 6)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)
        x = x.view(-1, 32 * 16 * 16)
        x = self.fc1(x)
        x = 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 .005 and momentum .9. You may also want to enable GPU training using `torch.device()`.

```
[ ]  ## -- ! code required
     import torch
     import torch.nn as nn
     import torch.optim as optim

     device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
     net = Net().to(device)
     criterion = nn.CrossEntropyLoss()
     optimizer = optim.SGD(net.parameters(), lr=0.005, momentum=0.9)
```

```python
def train_on_Pokemon(net, optimizer, device, trainloader):
  if torch.cuda.is_available():
      net.cuda()
  net.train()

  for epoch in range(10):
      running_loss = 0.0
      for i, data in enumerate(trainloader):
          ## -- ! code required
          inputs, labels = data
          inputs, labels = inputs.to(device), labels.to(device)
          optimizer.zero_grad()
          outputs = net(inputs)
          loss = criterion(outputs, labels)
          loss.backward()
          optimizer.step()

  print('Finished Training')
  return net
```

```python
# kick off the training
net = train_on_Pokemon(net, optimizer, device, trainloader_no_norm)
```

```
Finished Training
```

2.1.3 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 60%.

```python
def test_on_Pokemon(net, testloader):

    ## -- ! code required
    correct = 0
    total = 0
    net.eval()
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    return accuracy



acc = test_on_Pokemon(net, testloader_no_norm)

print(f'Accuracy of the network on the test images: {acc} %')
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias parameter of all the resizing tr
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias parameter of all the resizing tr
  warnings.warn(
Accuracy of the network on the test images: 80.0 %
```

2.2 Understanding the CNN Architecture

Explain the definition of the following terms. What is the corresponding setting in our net? Are there any other choices?

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

Solution:

Stride

- Refers to the number of pixels the filter moves across the input image.
- In the CNN, the stride is set to 2.
- I could set the stride parameter in the convolutional layers to a value other than 1 if you want the filter to move across the input with a different step size.

Padding

- A technique of adding extra pixels around the border of the input image before applying the filter.
- In this net, I've set the padding parameter to 1 in both nn.Conv2d layers to ensure that the spatial dimensions of the feature maps remain the same.
- I could choose different padding values based on the desired output size and the filter size.

Non-linearity

- Refers to the activation function applied after the convolutional layers to allow the model to adapt with a variety of data and to differentiate between different outcomes.
- I used the ReLU activation function (nn.ReLU()) after each convolutional layer.
- There are other activation functions like Sigmoid, Tanh, Leaky ReLU.

Pooling

- A downsampling operation that reduces the spatial dimensions of the feature maps by taking the maximum or average value in a local region.
- I used max pooling (nn.MaxPool2d) with a kernel size of 2x2 and a stride of 2 after each convolutional layer.
- I could have used the average pooling instead of max pooling and experiment with different kernel sizes and strides.

Loss function

- Measures the difference between the predicted outputs of the network and the actual labels. We want to minimize the value of the loss function during the backpropagation step in order to make the neural network better.
- I've used the cross-entropy loss (nn.CrossEntropyLoss()) as the loss function.
- There are different loss functions such as Mean Squared Error (MSE) and Binary Cross-Entropy.

Optimizer

- Updates the parameters of the network during training to minimize the loss function.
- I've used the SGD optimizer (optim.SGD) with a learning rate of 0.005 and momentum of 0.9.
- There are various optimizers with different update rules.

Learning rate

- Determines the step size at which the optimizer updates the parameters during training.
- I've set the learning rate to 0.005.
- I could do some tuning to see different learning rates to find the optimal value for faster convergence and better performance.

Momentum

- Helps accelerate the optimizer in the relevant direction
- I've set the momentum to 0.9 in the SGD optimizer.
- I could adjust the momentum parameter to control the effect of previous gradients on the update step.

2.3 Understanding the effect of normalization

In this section, we explore the effect of data normalization on model training. Specificly, we add the normalization in data transform and re-train the model. Run the following cells. Then write analysis that compare the results with and without data augmentation.

2.3.1 Re-train the model with data augmentation

```
[30] # Comment 0: define transformation that you wish to apply on image
     data_transforms = transforms.Compose([transforms.ToTensor(),
          transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
          transforms.Resize((64, 64))])
     # Comment 1 : Load the datasets with ImageFolder
     trainset = datasets.ImageFolder(root = PATH_OF_DATA + "/Pokemon_train",
                                     transform = data_transforms)
     # Comment 2: Using the image datasets and the transforms, define the dataloaders
     train_sampler = torch.utils.data.RandomSampler(trainset)
     trainloader = torch.utils.data.DataLoader(trainset, batch_size = 4, sampler = train_sampler, shuffle = False, num_workers = 2)

     testset = datasets.ImageFolder(root = PATH_OF_DATA + "/Pokemon_test",
                                    transform = data_transforms)
     testloader = torch.utils.data.DataLoader(testset, batch_size = 4, shuffle = False, num_workers = 2)
```

```
[32] ## -- ! code required
       # train the model with data normalization
     net_norm = train_on_Pokemon(net, optimizer, device, trainloader)
```

```
acc = test_on_Pokemon(net_norm, testloader)

print(f'Accuracy of the network on the the test images: {acc} %')
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the anti
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the anti
  warnings.warn(
Accuracy of the network on the the test images: 83.33333333333333 %
```

+ Code + Text

Comparison:

The accuracy of 83.30% achieved with data augmentation represents an improvement over the 80% accuracy achieved without data augmentation. This indicates that data augmentation enhances the model's ability to generalize better leading to a more robust accruacy.

Accuracy of 83.33*%

2.3.2 Explore the effect of normalization layer in model

In this section, we explore the effect of adding normalization layer into the model. In the code block below, insert a batch normalization layer after each convolutional layer.

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

     class Net_with_BatchNorm(nn.Module):
         def __init__(self):
             super(Net_with_BatchNorm, self).__init__()
             self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
             self.bn1 = nn.BatchNorm2d(16)
             self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
             self.bn2 = nn.BatchNorm2d(32)
             self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
             self.relu = nn.ReLU()
             self.fc1 = nn.Linear(32 * 16 * 16, 128)
             self.fc2 = nn.Linear(128, 64)
             self.fc3 = nn.Linear(64, 6)

         def forward(self, x):
             x = self.conv1(x)
             x = self.bn1(x)
             x = self.relu(x)
             x = self.pool(x)
             x = self.conv2(x)
             x = self.bn2(x)
             x = self.relu(x)
             x = self.pool(x)
             x = x.view(-1, 32 * 16 * 16)
             x = self.fc1(x)
             x = self.fc2(x)
             x = self.fc3(x)

             return x

     net_bn = Net_with_BatchNorm()
     optimizer = torch.optim.SGD(net_bn.parameters(), lr=0.005, momentum=0.9)
```

Next, we re-train the model with batch normalization layer on the dataset without data augmentation used. Write about your findings.

```
net_bn = train_on_Pokemon(net_bn, optimizer, device, trainloader_no_norm)

acc = test_on_Pokemon(net_bn, testloader_no_norm)

print(f'Accuracy of the network on the test images: {acc} %')
```

```
Accuracy of the network on the test images: 82.22222222222223 %
```

Findings: Adding batch normalization layers slightly decreased the accuracy to 82.22%, suggesting that in this case, it might not have provided significant performance improvement compared to the model without batch normalization.

2.3.3 Explore different model normalization methods

In this section, we experiment with different model normalization layers. In the code block below, insert a layer normalization layer after each convolutional layer.

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

class Net_with_LayerNorm(nn.Module):
    def __init__(self):
        super(Net_with_LayerNorm, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.ln1 = nn.LayerNorm((16, 64, 64))
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.ln2 = nn.LayerNorm((32, 32, 32))
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(32 * 16 * 16, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 6)

    def forward(self, x):
        x = self.conv1(x)
        x = self.ln1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.ln2(x)
        x = self.relu(x)
        x = self.pool(x)
        x = x.view(-1, 32 * 16 * 16)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x

net_ln = Net_with_LayerNorm()

optimizer = torch.optim.SGD(net_ln.parameters(), lr=0.005, momentum=0.9)
```

Next, we re-train the model with layer normalization layer on the dataset without data augmentation used. Write about your findings.

```
net_ln = train_on_Pokemon(net_ln, optimizer, device, trainloader_no_norm)

acc = test_on_Pokemon(net_ln, testloader_no_norm)

print(f'Accuracy of the network on the test images: {acc} %')
```

```
Accuracy of the network on the test images: 85.55555555555556 %
```

Findings: After inserting layer normalization layers after each convolutional layer, the accuracy improved to 85.56% on the test images, suggesting that layer normalization might be more effective than batch normalization for this particular model and dataset configuration.

Experimental Results:

```python
class ModifiedNet(nn.Module):
    def __init__(self):
        super(ModifiedNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.bn2 = nn.BatchNorm2d(64)

        self.pool = nn.AvgPool2d(kernel_size=2, stride=2)
        self.relu = nn.LeakyReLU()
        self.dropout = nn.Dropout(p=0.5)
        self.fc1 = nn.Linear(64 * 16 * 16, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 6)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.pool(x)
        x = x.view(-1, 64 * 16 * 16)
        x = self.dropout(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x

net_modified = ModifiedNet()
optimizer = optim.SGD(net_modified.parameters(), lr=0.001, momentum=0.9)
```

```python
def train(net, optimizer, trainloader, criterion, device, epochs=10):
    net.to(device)
    for epoch in range(epochs):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data[0].to(device), data[1].to(device)
            optimizer.zero_grad()
            outputs = net(inputs)

            labels = labels.view(-1)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if i % 2000 == 1999:
                print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0
    print('Finished Training')



criterion = nn.CrossEntropyLoss()
train(net_modified, optimizer, trainloader, criterion, device)
```

```
Finished Training
```

```python
def test(net, testloader, criterion, device):
    net.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print('Accuracy of the network on the test images: %.2f %%' % accuracy)

test(net_modified, testloader, criterion, device)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias para
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias para
  warnings.warn(
Accuracy of the network on the test images: 91.11 %
```

Results:

I increased the output channels to 32 and 64, used average pooling with the same parameters, and the LeakyReLU in this modified nerual net to increase the accruacy to 91.1%

Epochs	Learning Rate	Momentum	Accuracy
10	0.001	0.95	90.00
10	0.001	0.99	63.33
10	0.01	0.95	53.33
10	0.01	0.99	16.67
15	0.001	0.95	90.00
15	0.001	0.99	48.89
15	0.01	0.95	44.44
15	0.01	0.99	16.67

The results indicate that a learning rate of 0.001 consistently yielded higher accuracies than 0.01, suggesting the importance of a smaller learning rate for effective optimization. Additionally, a momentum of 0.95 consistently outperformed 0.99, indicating the benefits of moderate momentum in preventing local minima. Surprisingly, accuracies were generally higher with 10 epochs compared to 15 epochs, suggesting that longer training may not always lead to improved performance and could potentially result in overfitting. Overall I achievd an accruacy of 90% with an epoch of 10/15, LR of .001 and momentum .95.

Question 4

4.1.1 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 Q2.3.1.

4.1.2 Save state_dict

Save the state_dict of the model in Q2.3.1 with the torch.save() function to a local path.

```python
## -- ! code required
import torch

print("Keys of state_dict:")
for key in net.state_dict():
    print(key)

PATH = "model.pth"
torch.save(net.state_dict(), PATH)
```

```
Keys of state_dict:
conv1.weight
conv1.bias
conv2.weight
conv2.bias
fc1.weight
fc1.bias
fc2.weight
fc2.bias
fc3.weight
fc3.bias
```

4.1.3 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().

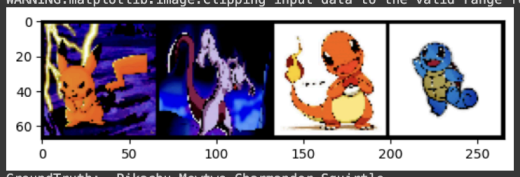
```
[48] net2 = Net()
     PATH = "model.pth"
     net2.load_state_dict(torch.load(PATH))
```

```
<All keys matched successfully>
```

Let's test net2's performance on Pokemon Dataset

```
net2.eval()
classes_labels = ['Bulbasaur', 'Charmander', 'Mewtwo', 'Pikachu', 'Psyduck', 'Squirtle']
dataiter = iter(trainloader)
images, labels = next(dataiter)


def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

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

# Make predictions
outputs = net2(images)
_, predicted = torch.max(outputs, 1)

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

```
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias parameter of all the
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias parameter of all the
  warnings.warn(
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```




```
GroundTruth:  Pikachu Mewtwo Charmander Squirtle
Predicted:  Squirtle Squirtle Mewtwo Mewtwo
```

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        #images, labels = images.to(device), labels.to(device)
        outputs = net2(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the net2 on the test images: %d %%' % (100 * correct / total))
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: Th
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: Th
  warnings.warn(
Accuracy of the net2 on the test images: 17 %
```

When I ran this I had errors, but commenting out the images/labels.to(device) worked. Not entirely sure why.

4.2.2 Load state_dict partially

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

```
net_cifar = Net()
selected_layers = ['conv1.weight', 'conv1.bias', 'conv2.weight', 'conv2.bias', 'fc1.weight', 'fc1.bias', 'fc2.weight', 'fc2.bias']

## -- ! code required
state_dict = torch.load(PATH)
filtered_state_dict = {key: value for key, value in state_dict.items() if key in selected_layers}
net_cifar.load_state_dict(filtered_state_dict, strict=False)
```

4.2.3 Fine-tune net_cifar on CIFAR-10

Fine-tune the net_cifar on CIFAR-10 and show the results.

```
## -- ! code required -- define criterion, optimizer, and device
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net_cifar.parameters(), lr=0.001, momentum=0.9)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net_cifar.to(device)
```

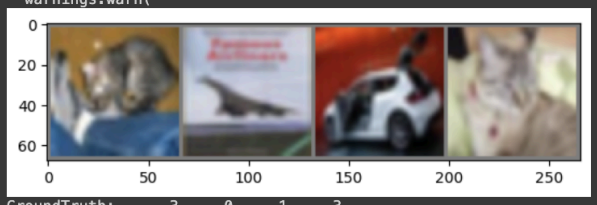
```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu): ReLU()
  (fc1): Linear(in_features=8192, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=6, bias=True)
)
```

```
## -- ! code required -- training loop
epochs = 5
for epoch in range(epochs):
    running_loss = 0.0
    for i, data in enumerate(cifar10_trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = net_cifar(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0
print('Finished Training')
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
[1,  2000] loss: 1.327
[1,  4000] loss: 1.137
[1,  6000] loss: 1.074
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
[2,  2000] loss: 0.989
[2,  4000] loss: 0.957
[2,  6000] loss: 0.955
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
[3,  2000] loss: 0.845
[3,  4000] loss: 0.898
[3,  6000] loss: 0.874
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
[4,  2000] loss: 0.789
[4,  4000] loss: 0.810
[4,  6000] loss: 0.814
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional
  warnings.warn(
[5,  2000] loss: 0.725
[5,  4000] loss: 0.759
[5,  6000] loss: 0.761
Finished Training
```

```python
# get some random training images
cifar10_dataiter = iter(cifar10_testloader)

## -- ! code required
images, labels = next(iter(cifar10_testloader))
imshow(torchvision.utils.make_grid(images.cpu()))
print('GroundTruth: ', ' '.join('%5s' % labels[j].item() for j in range(4)))
outputs = net_cifar(images.to(device))
_, predicted = torch.max(outputs, 1)
print('Predicted: ', ' '.join('%5s' % predicted[j].item()
                              for j in range(4)))
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias
  warnings.warn(
```




```
GroundTruth:      3     0     1     3
Predicted:        3     0     1     3
```

```python
[ ] correct = 0
    total = 0
    with torch.no_grad():
        for data in cifar10_testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = net_cifar(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print('Accuracy of the net_cifar on the test images: %d %%' % (100 * correct / total))
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/functional.py:1603: UserWarning: The default value of the antialias
  warnings.warn(
Accuracy of the net_cifar on the test images: 67 %
```

Question 5

5.1 Loss functions

5.1.1 Content Loss

The content of an image is represented by the values of the intermediate feature maps. Finish the ContentLoss() to match the corresponding content target representations.

```
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
```

5.1.2 Style Loss

The style of an image can be described by the means and correlations across the different feature maps. Calculate a Gram matrix that includes this information and finish the StyleLoss().

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

def gram_matrix(input):
    a, b, c, d = input.size()
    features = input.view(a * b, c * d)
    G = torch.mm(features, features.t())
    return G.div(a * b * c * d)

class StyleLoss(nn.Module):
    def __init__(self, target_feature):
        super(StyleLoss, self).__init__()
        self.target = gram_matrix(target_feature).detach()

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

5.1.3 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 th

Import a pretrained VGG-19 from torchvision.models. Make sure to set the network to evaluation mode using .e

```
## -- ! code required

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

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: Th
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Ar
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /root/.cache/tor
100%|██████████| 548M/548M [00:03<00:00, 152MB/s]
```

5.1.4 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 = torch.tensor(mean).view(-1, 1, 1)
        self.std = torch.tensor(std).view(-1, 1, 1)

    def forward(self, img):
        return (img - self.mean) / self.std
```

```
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)
            # The in-place version doesn't play very nicely with the ContentLoss and StyleLoss
            # so we replace with out-of-place ones here.
            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:
            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:
            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
```

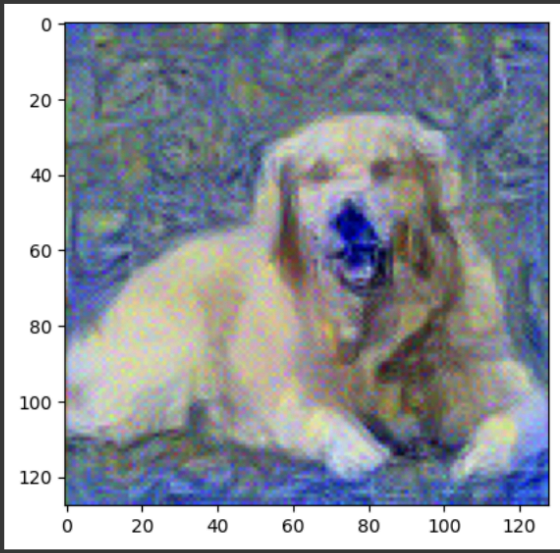
```
content_layers_selected = ['conv_4']
style_layers_selected = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']

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..')
    step_i = 0
    while step_i <= num_steps:
        input_img.data.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()

        optimizer.step()

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

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

    return input_img
```

5.2.2 Test your model

Now you have completed your codes, let's test them!

```
content_layers_selected = ['conv_4']
style_layers_selected = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']
style_weight=100000
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)

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

```
Building the style transfer model..
Optimizing..
<ipython-input-57-1417dfb7f431>:9: UserWarning: To copy construct from a tensor, it is recommended to use
  self.mean = torch.tensor(mean).view(-1, 1, 1)
<ipython-input-57-1417dfb7f431>:10: UserWarning: To copy construct from a tensor, it is recommended to us
  self.std = torch.tensor(std).view(-1, 1, 1)
run 50:
Style Loss : 70.684471 Content Loss: 28.664692

run 100:
Style Loss : 11.624935 Content Loss: 22.914890
```

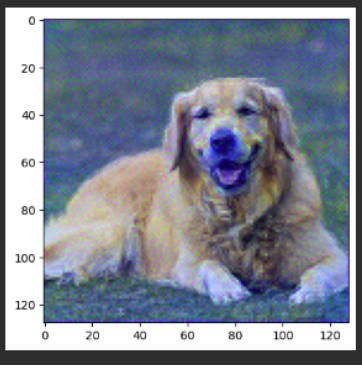


5.3 Content/style loss weight ratio

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

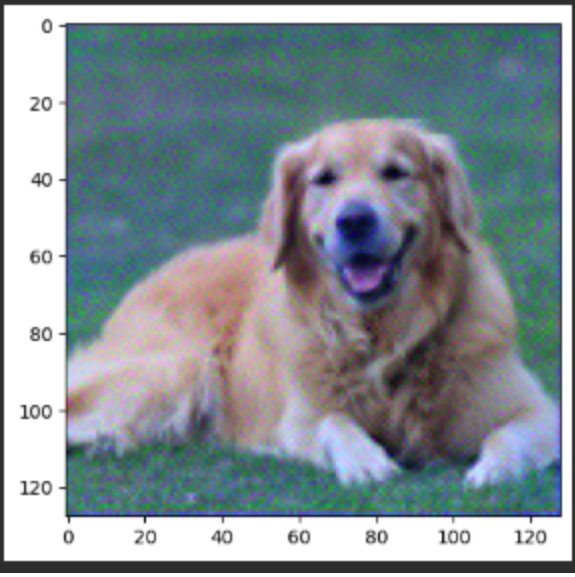
```
[15] ## -- ! code required -- style loss weight 5000
     # First run with style weight 5000
     style_weight=5000
     output = run_style_transfer(cnn, cnn_normalization_mean, cnn_normalization_std,
                                 content_img, style_img, input_img, num_steps=100,
                                 style_weight=style_weight, content_layers=content_layers_selected,
                                 style_layers=style_layers_selected)

     plt.figure()
     imshow(output)

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

```
Building the style transfer model..
Optimizing..
<ipython-input-11-a959762d6eaf>:8: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requ
  self.mean = torch.tensor(mean).view(-1, 1, 1)
<ipython-input-11-a959762d6eaf>:9: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requ
  self.std = torch.tensor(std).view(-1, 1, 1)
run 50:
Style Loss : 2.390841 Content Loss: 5.558018

run 100:
Style Loss : 2.167830 Content Loss: 4.896179
```

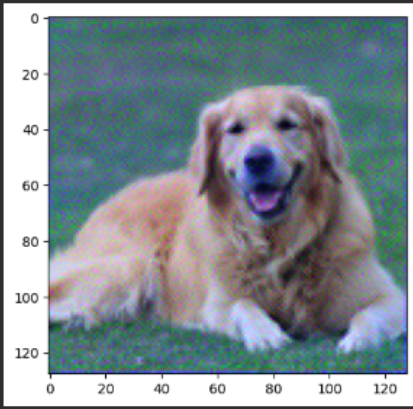


```
## -- ! code required -- style loss weight 10

# Second run with style weight 10
style_weight=10

output_10 = 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)

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

```
Building the style transfer model..
Optimizing..
<ipython-input-11-a959762d6eaf>:8: UserWarning: To copy construct from a tensor, it is recommended
  self.mean = torch.tensor(mean).view(-1, 1, 1)
<ipython-input-11-a959762d6eaf>:9: UserWarning: To copy construct from a tensor, it is recommended
  self.std = torch.tensor(std).view(-1, 1, 1)
run 50:
Style Loss : 0.060196 Content Loss: 0.839146

run 100:
Style Loss : 0.064072 Content Loss: 0.585675
```



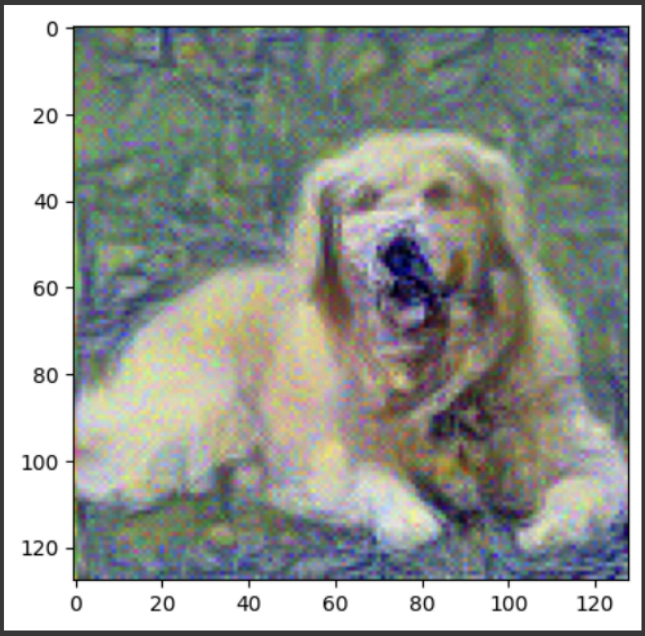
```
run 100:
Style Loss : 0.064072 Content Loss: 0.585675
```



Discussion: When using a style weight of 5000, the generated image leaned heavily towards resembling the style image, with less focus on retaining the original content. Whereas with a style weight of 10, the image preserved more of its original content, with only a litle influence of the style. Overall, different style weights affects how much the image looks like the style versus the original.

```python
## -- ! code required --your solution(s)
style_layers_selected = ['conv_1', 'conv_3', 'conv_5']
style_weight=100000
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_selecte
                            style_layers=style_layers_selected)

plt.figure()
imshow(output)

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

```
Building the style transfer model..
Optimizing..
<ipython-input-57-1417dfb7f431>:9: UserWarning: To copy construct from a tensor, it is recomm
  self.mean = torch.tensor(mean).view(-1, 1, 1)
<ipython-input-57-1417dfb7f431>:10: UserWarning: To copy construct from a tensor, it is recor
  self.std = torch.tensor(std).view(-1, 1, 1)
run 50:
Style Loss : 48.458111 Content Loss: 26.454697

run 100:
Style Loss : 12.778237 Content Loss: 20.798550
```



Discussion:

When using conv_1, conv_3, and conv_5 as intermediate layers for style representation, theres a significant reduction in the style loss compared to the previous experiments. This indicates that these layers capture lower-level features of the style image, resulting in a smoother style transfer. However, there is also a slight decrease in content loss, suggesting that the content of the output image may be slightly compromised.