CSCI-UA 0480-042 Computer Vision

Homework 2

Enter your name and NetID below.

Name: Yi Yang

NetID: yy2324

The main goals of this assignment include:

1. Giving an introduction to PyTorch
2. Loading images using PyTorch DataLoader
3. Performing Inference on a Pretrained Resnet-18 model using a small dataset
4. Doing some analysis on the classification results

Also accompanying parts 3-4, there are a few questions (**8 questions in total**). Please give your answers in the space provided. In most of the questions, you will be asked to complete a code snippet. You can quickly navigate to those questions by searching (Ctrl/Cmd-F) for `TODO:` .

In this homework, we'll be covering few basic PyTorch concepts. More will be covered in the next homework.

Part 1: Introduction to PyTorch

PyTorch (https://pytorch.org/) and TensorFlow (https://www.tensorflow.org/) are open source machine learning frameworks that are used to develop and train neural network models. PyTorch is primarily developed by Facebook AI Research lab (FAIR), whereas TensorFlow is developed by the Google Brain team. You can read more about the differences between PyTorch and Tensorflow from these 2 articles:

1. Real Python blog post by Ray Johns, "PyTorch vs TensorFlow for Your Python Deep Learning Project" (https://realpython.com/pytorch-vs-tensorflow/)
2. Towards Data Science article by Kirill Dubovikov, "PyTorch vs TensorFlow — spotting the difference" (https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b)

In this course, we will use PyTorch.

A neural network consists of several layers. Each layer takes an input array, computes a linear product, applies some non linear activation function to give an output. These inputs and outputs are just n-dimensional arrays, and are called as `Tensor` s in PyTorch. These are similar to what we have in `NumPy` except that `Tensor` s can run on GPUs. PyTorch provides several functions for operating on these `Tensor` s. In the background, during forward and backward propagation, `Tensor` s can keep track of the computational graph and its weights' gradients.

We'll now show how one can build a neural network in PyTorch using a simple example. [Credits: Deep Learning with PyTorch: A 60 Minute Blitz by Soumith Chintala (https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#neural-networks)]

Consider this network for classifying digit images:

Neural Network Architecture for classifying digit images

This network can be built in PyTorch using the `torch.nn` package. `torch.nn` implements various neural network layers (full-connected, convolution, pooling, etc.), and activation functions (softmax, sigmoid, etc.).

In [68]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

# nn.Module is a container that's used as the base class for
# building any network. It has built-in methods for loading parameters
# from a pretrained network, exporting weights from a network, moving
# the model from CPU to GPU, etc.

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # Docs:
        # https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torc
h.nn.Conv2d
        # https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torc
h.nn.Linear

        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)

        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Docs:
        # https://pytorch.org/docs/stable/nn.functional.html#max-pool2d
        # https://pytorch.org/docs/stable/tensor_view.html#tensor-view-doc
        # https://pytorch.org/docs/stable/nn.functional.html#relu

        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

`nn.Module` subclass requires us to fill in 3 methods:

1. `__init__` : Calls the base constructor, and instantiates all the network layers (ones that have trainable parameters).
2. `forward` : Defines the computation flow for forward propagation/inference by specifying a chain of operations. Given an input Tensor `x` , it computes output from one layer, feeds it to the next, and so on, till the final output [i.e. the digit class (0-9) the image belongs to] is computed.
3. `num_flat_features` : Just a helper method used by `forward()` . This method does some shape calculation on the output from the 2nd `max_pool2d` layer, which is then used to generate a view from the `max_pool2d` layer's output. Views are used to reshape an existing Tensor without creating a new data copy, thus allowing us to do operations on Tensors faster.

You can read more from this PyTorch tutorial on Neural Networks (https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html). First 2 sections (*Neural Networks* and *Define the network*) are enough. We will not cover backprop/autograd in this homework -- that'll be discussed in the next homework.

With the network class defined, let's instantiate the network and print the layers inside it.

In [69]:
```
net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=576, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Part 2: Loading images using Dataloaders

As you've read in the above PyTorch tutorial, we can compute the output of a network on an image by calling `net()` on its tensor. This will return an output `Tensor` , with one value for each of the 10 classes. The network's predicted label for this image would be the label corresponding to the index with the highest value.

Consider this example below. An image of size [1, 32, 32] is randomly generated and is fed into the network to get its label. The output is a tensor containing 10 `float` values.

In [70]:
```
# Use a random 32x32 input
# 1st dimension = 1 refers to batch size = 1 (only one example/image in this b
atch of examples)
# 2nd dimension = 1 refers to the number of input channels = 1 (this is a BW i
mage)
input_tensor = torch.randn(1, 1, 32, 32)

# Do inference on this example
output_tensor = net(input_tensor)
print(output_tensor)
```

```
tensor([[ 0.0761,  0.0119, -0.0127,  0.0243,  0.0402,  0.1831,  0.0302, -0.09
30,
          0.0311, -0.0924]], grad_fn=<AddmmBackward>)
```

In [32]:
```
print("The predicted class index of the random image is", output_tensor.argmax
().numpy())

# argmax -> returns the index of the maximum value in the output_tensor
# numpy -> converts a PyTorch Tensor into a Numpy array
```

```
The predicted class index of the random image is 9
```

However, this approach (of calling the network on the input tensor to get its output) doesn't scale well. Typically, during the training phase, one would shuffle all the images in a dataset, load a batch of images from the shuffle, preprocess all the images in that batch (by applying some transformations such as random cropping, rescaling the image, etc.), and then provide the resulting image tensor to the model. If we simply do all these operations by iterating over the dataset using a `for` loop, these operations can turn out to be inefficient, especially when working on large datasets. For every batch, we have to generate the input tensor, and only after this step is complete, the network will be able to compute the output on this tensor.

To avoid this problem, we use a PyTorch utility called `DataLoader` , which is capable of performing all the above operations using `multiprocessing` workers (i.e. multiple threads). By delegating these tasks to separate workers, these workers can fetch the next batch while training on the previous batch is still in progress, thereby reducing the batch generation bottleneck. In other words, the training process doesn't need to wait for the worker threads to fetch the batch for training. Furthermore, this is necessary because loading all the images in a dataset into memory and then processing them may not always be feasible because of memory constraints.

`DataLoader` takes as input a `Dataset` object, which is a PyTorch abstract class for holding and operating on a dataset. This also provides a homogenous way of working with different datasets.

`torchvision.datasets` (https://pytorch.org/vision/0.8/datasets.html#) package provides some commonly used CV-related datasets such as `MNIST` (http://yann.lecun.com/exdb/mnist/), `CIFAR10` (https://www.cs.toronto.edu/~kriz/cifar.html), `Places365` (http://places2.csail.mit.edu/index.html), etc.

Using `Dataset` and `DataLoader` , here's how the dataloading phase in a PyTorch program works:

In [52]:
```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset

# Transforms refer to a set of operations one would apply to each
# image before sending it to the network. For example, all images
# in the dataset may not be of the same size, in which case, we could
# use a transform to scale the image to a given size.
# PyTorch provides a variety of transforms:
# https://pytorch.org/docs/stable/torchvision/transforms.html
transform = transforms.Compose([
    transforms.ToTensor(), # first, convert image to PyTorch tensor
    transforms.Normalize((0.1307,), (0.3081,)) # normalize inputs
])
# We're telling MNIST dataset is in the folder "./mnist"
# If it doesn't exist in the folder, we're allowing the package
# to download it (by setting download=True)
mnist = datasets.MNIST('../shared/hw2/mnist', download=False, transform=transf
orm)

# We don't want to use all the 60000 training images in MNIST.
# Instead we'll just use a subset of the training set (64 images)
# to make the computation faster
mnist = Subset(mnist, range(64))

# DataLoader requires a Dataset object.
# Each call to dataloader will return a batch of 16 images
# Batches are generated after shuffling the dataset
dataloader = DataLoader(mnist, batch_size=16, shuffle=True)

# Typically one would want to train using a GPU.
# To be able to do that, torch needs to locate/access the GPU.
# cuda = torch.device("cuda") # use "cuda:0", "cuda:1" instead to use the firs
t and second GPU respectively

for batch_idx, (inputs, targets) in enumerate(dataloader):
    # To move the tensors to the GPU device, one would do
    # inputs = inputs.to(cuda)
    # targets = targets.to(cuda)
    # However, we currently don't have access to a
    # GPU, so leave the above 2 lines commented.
    print("Batch Idx", batch_idx, "; Input size", inputs.shape, "; Target siz
e", targets.shape)

    # Forward propagation logic sits here
```

```
Batch Idx 0 ; Input size torch.Size([16, 1, 28, 28]) ; Target size torch.Size
([16])
Batch Idx 1 ; Input size torch.Size([16, 1, 28, 28]) ; Target size torch.Size
([16])
Batch Idx 2 ; Input size torch.Size([16, 1, 28, 28]) ; Target size torch.Size
([16])
Batch Idx 3 ; Input size torch.Size([16, 1, 28, 28]) ; Target size torch.Size
([16])
```

Each batch has 16 images as we've specified in the `DataLoader`. Each target has 16 values, where each value is the index of the true label the image belongs to. The 2nd dimension in the `inputs` tensor refers to the number of input channels, which is 1 in this case, because MNIST contains BW images. 3rd and 4th dimension refer to the image's height and width respectively.

Also check out this thread (https://twitter.com/_ScottCondron/status/1363494433715552259) [yes, it's a tweet :)] and this article (https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel).

Part 3: Inference on the sample Imagenet dataset

To demonstrate inference on a pretrained network, we've created a dataset, consisting of 100 images by sampling 5 images from 20 classes in the Imagenet dataset. Imagenet dataset comprises approximately 1 million images and 1000 classes, and in the interest of keeping things simple, we've provided a smaller subset.

Question 1

To work with this dataset, you'll need to create a `Dataset` object, since Imagenet dataset cannot be downloaded directly via the `torchvision.datasets` package. In order to do that, you need to define a class subclassing the abstract `Dataset` class. Any custom `Dataset` class should contain the following:

1. `__init__()` method: Constructor method. We've provided the list of attributes the class would need.
2. `__len__()` method: Returns the number of samples in the dataset.
3. `__getitem__()` method: Given an index, it returns the (input, target) pair at that index.

In [65]:
```
from torch.utils.data import Dataset
import glob
import numpy as np
from PIL import Image

class SampleImagenetDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        '''
        Args:
        root_dir: path where all the 100 jpg images are located
        transform: set of transforms to apply on a dataset sample
        '''
        # TODO: Generate a list containing
        # paths to all the 100 jpg images. Use the glob library
        # to do this.
        self.image_list = glob.glob('../shared/imagenet_samples/**.jpg')

        # TODO: There's a file called 'labels_to_ids.txt' inside `../shared/im
agenet_samples`.
        # Read the file contents to generate a dictionary containing
        # label to idx mapping. The resulting dictionary should have a `str`
        # type key mapped to a `int` type value representing the index
        dic = {}
        with open("../shared/imagenet_samples/labels_to_ids.txt") as f:
            for line in f:
                (key, val) = line.split()
                dic[key] = val

        self.labels_to_ids = dic

        self.transform = transform

    def __len__(self):
        return len(self.image_list)

    def __getitem__(self, idx):
        img_path = self.image_list[idx]

        classname = img_path.split("_", 3)[-1].split(".")[0]
        label = self.labels_to_ids[classname]

        # TODO: Open the image using PIL.Image package
        image = Image.open(img_path)

        if self.transform:
            # TODO: Apply the transform to the `image`
            # and overwrite the output back to the `image` variable
            image = self.transform(image)

        return image, label
```

Question 2

Having defined our custom dataset class, we'll instantiate a `SampleImagenetDataset` class object. And then we'll set up a `DataLoader` object.

In [66]:
```
from torchvision import transforms

data_transform = transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

sample_dataset = SampleImagenetDataset('../shared/imagenet_samples', data_tran
sform)

# TODO: Create a `DataLoader` object
# to load images from `sample_dataset` object created above.
# Shuffle the dataset and use a batch size of 10.
# Docs: https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader
dataset_loader = DataLoader(sample_dataset, batch_size=10, shuffle=True)
```

Question 3

Using the `DataLoader` object, we will iterate over the dataset, dividing it into batches and working on each batch per iteration. For each batch, the network takes the input tensor and outputs the prediction. All predictions and targets (true labels) will be combined and stored in 2 lists -- `outputs` and `targets` respectively.

In [71]:
```
from torchvision import models

# TODO:
# Construct a **pretrained** ResNet-18 model. Use the
# `torchvision.models` package.
resnet18 = models.resnet18(pretrained=True)

outputs = []
targets = []

for idx, (input_tensor, target_tensor) in enumerate(dataset_loader):
    # TODO: Compute the pretrained model's output on input_tensor
    output_tensor = net(input_tensor)

    # Notes:
    # 1. Using argmax() gives the index of the maximum value in the output_ten
sor
    #    This is the network's predicted label. output_tensor is a 2d tensor o
f
    #    dimensions [batch_size, num_classes]. argmax(axis=1) returns the indi
ces
    #    of the maximum values across dim=1 (across the row). Resulting tensor
    #    would be a 1D tensor of size [batch_size].
    #    See https://pytorch.org/docs/stable/generated/torch.argmax.html#torch
-argmax
    # 2. Calling numpy() converts a PyTorch tensor to a numpy array
    # 3. list(<arr>) converts a NumPy array to a list
    # 4. l1 += l2 appends elements of list l2 to list l1.
    outputs += list(output_tensor.argmax(1).numpy())
    targets += list(target_tensor.numpy())
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-71-a1392182c673> in <module>
     11 for idx, (input_tensor, target_tensor) in enumerate(dataset_loader):
     12     # TODO: Compute the pretrained model's output on input_tensor
---> 13     output_tensor = net(input_tensor)
     14 
     15     # Notes:

/opt/conda/envs/cv_sp21/lib/python3.8/site-packages/torch/nn/modules/module.p
y in _call_impl(self, *input, **kwargs)
    725             result = self._slow_forward(*input, **kwargs)
    726         else:
--> 727             result = self.forward(*input, **kwargs)
    728         for hook in itertools.chain(
    729                 _global_forward_hooks.values(),

<ipython-input-68-20a8375c18e5> in forward(self, x)
     33 
     34         # Max pooling over a (2, 2) window
---> 35         x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
     36         # If the size is a square you can only specify a single numbe
r
     37         x = F.max_pool2d(F.relu(self.conv2(x)), 2)

/opt/conda/envs/cv_sp21/lib/python3.8/site-packages/torch/nn/modules/module.p
y in _call_impl(self, *input, **kwargs)
    725             result = self._slow_forward(*input, **kwargs)
    726         else:
--> 727             result = self.forward(*input, **kwargs)
    728         for hook in itertools.chain(
    729                 _global_forward_hooks.values(),

/opt/conda/envs/cv_sp21/lib/python3.8/site-packages/torch/nn/modules/conv.py
 in forward(self, input)
    421 
    422     def forward(self, input: Tensor) -> Tensor:
--> 423         return self._conv_forward(input, self.weight)
    424 
    425 class Conv3d(_ConvNd):

/opt/conda/envs/cv_sp21/lib/python3.8/site-packages/torch/nn/modules/conv.py
 in _conv_forward(self, input, weight)
    417                             weight, self.bias, self.stride,
    418                             _pair(0), self.dilation, self.groups)
--> 419         return F.conv2d(input, weight, self.bias, self.stride,
    420                         self.padding, self.dilation, self.groups)
    421 

RuntimeError: Given groups=1, weight of size [6, 1, 3, 3], expected input[10,
3, 224, 224] to have 1 channels, but got 3 channels instead
```

Question 4

How many batches are generated in the `for` loop? What're the sizes of the `input_tensor`, `target_tensor` and `output_tensor` in any given iteration?

Answer 10 batches are generated, input size should be [10, 3, 224, 224] with target tensor being 10, and output tensor being [6, 1, 3, 3]

Part 4: Classification Analysis

From part 3, we have the network's predicted label indices and the target label (or the true label) indices. Using `scikit-learn` (https://machinelearningmastery.com/a-gentle-introduction-to-scikit-learn-a-python-machine-learning-library/), we'll perform some analysis on those results.

But before we do that, we'll convert those `int`-type indices to `str`-type labels using a idx-to-label mapping. This mapping has been provided in the `shared/imagenet_samples` folder in a file called `ids_to_labels.txt`.

Question 5

Read the contents of the file and generate a python dictionary that holds the idx-to-label mapping. In the map, the key is an `int`-type index and the value is a `str`-type label.

In [73]:
```
# TODO:
d = {}
with open("../shared/imagenet_samples/ids_to_labels.txt") as g:
    for line in g:
        (key, val) = line.split()
        d[int(key)] = val
ids_to_labels = d

# We'll convert the list of keys in the mapping to a Python `set`
ids = set(ids_to_labels.keys())
```

Now let's convert these `int`-values in the `outputs` list to a list of `str`-type labels. However, some network predictions may not be in the list of classes we've picked (remember: Imagenet has 1000 classes and we just picked 20 classes). Such indices will be mapped to label "other" using the method below:

In [74]:
```
def transform_outputs(outputs):
    return [ids_to_labels[output] if output in ids else "other" for output in 
outputs]

outputs = transform_outputs(outputs)
targets = transform_outputs(targets)
```

Question 6

Using `sklearn.metrics` (https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics) generate a classification report showing the per-class precision, recall and f1-score. Also print the accuracy of the predictions, macro and micro averages of precision, recall and f1-score across all classes.

If the method you're using does not print the micro-averaged metrics, please briefly explain why in 3-4 lines. In that case, also report the values of these 3 metrics.

In [ ]:
```
from sklearn import metrics

precision = precision_score(outputs, targets)
recall = recall_score(outputs, targets)
f1_score = f1_score(outputs, targets)
accuracy = accuracy_score(outputs, targets)
micro_precision = precision_score(outputs, targets, average='micro')
macro_precision = precision_score(outputs, targets, average='macro')
micro_recall = recall_score(outputs, targets, average='micro')
macro_recall = recall_score(outputs, targets, average='macro')
micro_f1 = f1_score(outputs, targets, average='micro')
macro_f1 = f1_score(outputs, targets, average='macro')
print("precision is: ", precision, "recall is ", recall, "f1 is ", f1_score, 
"accuracy is ", accuracy, "micro and macro precision, recall and f1 are ",
     micro_precision, macro_precision, micro_recall, macro_recall, micro_f1, m
acro_f1)
```

Question 7

In 3-4 sentences, explain what micro- and macro- averages are and how they're different from each other. A micro average is when sum up the individual true positives, false positives, and false negatives of the system for different sets and the apply them to get the statistics. alternatively, macro average is the average of the precision and recall of the system on different sets. They are different because micro measure different sizes of data, while macro tests the performance across the data sets

Answer

Question 8

A confusion matrix (or an error matrix) allows us to visualize model's performance on a per-class basis. From Wikipedia (https://en.wikipedia.org/wiki/Confusion_matrix), "each row of the matrix represents the instances in a predicted class, while each column represents the instances in an actual class (or vice versa)".

Using `sklearn.metrics` , compute the confusion matrix.

In [ ]:
```
ids_to_labels[0] = "other"
labels_list = list(ids_to_labels.values())
# TODO:
conf_matrix = ...
```

Now let's plot the confusion matrix:

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline

fig, ax = plt.subplots(figsize=(7, 7))
plot = ax.imshow(conf_matrix, interpolation="nearest")
ax.set_xticks(np.arange(0, 21))
ax.set_xticklabels(labels_list)
plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor"
)
ax.set_yticks(np.arange(0, 21))
ax.set_yticklabels(labels_list);
ax.set_xlabel("Predicted class")
ax.set_ylabel("Target class")
plt.colorbar(plot, ax=ax);
```