BME695DL/ECE695: Homework 4

Spring 2021
Due Date: Monday, March 15,2021 (11:59pm)

Turn in your solutions via BrightSpace.

1 Introduction

This homework consists of the following three goals:

1. To get you to start using convolutional layers in a network meant for classifying images. [80% weight]

   For this homework, you will use the COCO dataset that has been specially curated for this homework so that you can start working on the classification network right away.

2. To have you write a image downloader script for the COCO dataset. [10% weight]

3. To have you write a dataloader function for the COCO images you will be downloading yourself with your own image downloader script. [10% weight]

2 About the COCO Dataset

HW2 introduced you to the ImageNet dataset.

The goal of the present homework is to provide you with a first introduction to the Microsoft COCO dataset that is just as famous as the ImageNet dataset, but a bit more complex to use because of its rich annotations.

The acronym COCO stands for "Common Objects in COntext". Go ahead and download the paper that introduced this dataset by clicking on:

`https://arxiv.org/pdf/1405.0312.pdf`

You should at least read the Introduction section of the paper for why this dataset was created.

We wanted you to become familiar with this dataset but, at the same time, we did not want you to get bogged down with all the issues related to the downloading of the COCO images, their annotations, and with the creation of an appropriate dataloader for the training loop.

Therefore, for the main part of this homework, we have already downloaded a set of COCO images, downsampled them to 64×64 size and made them available for this homework through BrightSpace. With the repackaging of these images, you should be able to use the same dataloader in your image classification network that you used for HW2 (with minor modifications).

However, in order to provide you with incentive to dig deeper into the COCO dataset, this homework comes with two smaller parts: for creating an image downloader and for writing the code for the dataloader for your classification network.

3 About the Image Classification Network You Will Write Code For

First of all, remember that this part carries 80% weight for this homework. So this part deserves your immediate attention.

A good starting point for you would be to review the networks you see in the inner class "ExperimentsWithCIFAR" of the DLStudio module. The network you will be creating is likely to be very similar to the two examples — `Net` and `Net2` — shown in that section of DLStudio. After installing DLStudio, play with the two networks by changing the parameters of the convolutional and the fully connected layers and see what that does to the classification accuracy. For that, you will need to execute the following script in the Examples directory of the DLStudio module[1]

```
playing_with_cifar10.py
```

[1]You will need to install the CIFAR-10 dataset in your computer to run this script. The CIFAR image dataset, made available by the University of Toronto, is considered to be the fruit-fly of DL. The dataset consists of 32×32 images, 50,000 for training and 10,000 for testing that can easily be processed in your laptop. Just Google "download CIFAR-10 dataset" for the website from where you can download the dataset.

As with the DLStudio example code mentioned above, the classification network you will create will use a certain number of convolutional layers and, at its top, will contain one or more fully connected (FC) layers (also known as Linear layers). The number of output nodes at the final layer will be 10, which is equal to the number of image classes you will be working with.

In your experiments with the classification network, pay attention to the changing resolution in the image array as it is pushed up the resolution hierarchy of a CNN. This is particularly important when you are trying to estimate the number of nodes you need in the first fully connected layer at the top of the network. Depending on the sizes of the convolutional kernels you will use, you may also need to pay attention to the role played by padding in the convolutional layers.

4 About Creating Your Own COCO Downloader

Remember, this carries only 10% of the overall weight for this homework. So embark on this only after you have finished creating and testing your image classification network

The COCO dataset comes in 2014 and 2017 versions. For the programming task you will create your own training subset by combining images from 2014 and 2017 datasets for the multi-class classification task using CNNs. As to how one can access the images and labels, the creators of the dataset have provided COCO API and the annotation JSON files.

To do this part of the homework, you need to install the `cocoapi/PythonAPI` which is the Python version of the cocoapi. The main reason for installing this API is to make it easier for a user to access the different types of annotations that the COCO dataset makes available for each image. Here are the steps for what it takes to install this API:

1. You need to install the pycocotools package:

   ```
   sudo pip install pycocotools
   ```

2. You need to click on the following link

   ```
   https://github.com/cocodataset/cocoapi
   ```

and download from the website the `cocoapi/PythonAPI` package. Make a directory and download this ZIP archive into that directory. It will have a name like

```
cocoapi-8c9bcc3cf640524c4c20a9c40e89cb6a2f2fa0e9.zip
```

3. Unzipping the above will create a directory with the name like

   ```
   cocoapi-8c9bcc3cf640524c4c20a9c40e89cb6a2f2fa0e9
   ```

   When you cd into this directory, you will see a subdirectory named `PythonAPI`.

4. cd into the `PythonAPI` directory and execute

   ```
   sudo python3 setup.py build_ext install
   ```

5. Finally, download the annotation files by clicking on the following links:

   ```
   http://images.cocodataset.org/annotations/annotations_
                        trainval2014.zip
   http://images.cocodataset.org/annotations/annotations_
                        trainval2017.zip
   ```

   Unzipping them will deposit both sets of annotations in a directory called `annotations`. You will see around a dozen JSON files in this directory. Our interest is going to be primarily in the following files:

   ```
   instances_train2014.json

   instances_val2014.json

   instances_train2017.json
   ```

The `cocoapi/PythonAPI` provides the necessary functionalities for loading these JSON files and accessing images using class names. The `pycocoDemo.ipynb` demo available on the cocoapi GitHub repository is a useful resource to familiarize yourself with the cocoapi.

5 About Creating a Dataloader for the Training Loop

Note again that this carries only 10% of the overall weight for this homework. So embark on this only after you have finished creating and testing your image classification network

For this part, recall your custom dataset class from HW02 for the binary classification problem. For this homework, you will extend that class to load multi-class data. The dataset made available to you through BrightSpace is specially curated by us to make it easier for you to do the main part of this homework.

The following is a brief summary for custom dataset loading using PyTorch. In general, you start with defining a custom dataset class that is derived from the `torch.utils.data.Dataset` class. At the least, this custom dataset class must provide an implementation for the `__getitem__()` function that tells the system how to fetch one sample from the dataset. And also, depending on how the images are organized in the subdirectories, the implementation of the dataset may also need to provide for indexing the samples.

6 Programming Tasks

6.1 Task1: Image Classification using CNNs – Training and Validation

Download and extract the provided `hw04_data.zip` from BrightSpace. You will notice a folder hierarchy that is similar to what you saw for HW02, except now you have 10 classes. Your first step would be to make minor modifications to your HW02 dataset class for loading this multi-class data.

Then you will implement and test the following three CNN tasks.

CNN Task 1: In the following network, you will notice that we are constructing an instance of `torch.nn.Conv2d` in the mode in which it only uses the valid pixels for the convolutions. But, as you now know based on the Week 6 lecture (and slides), this is going to cause the image to shrink as it goes up the convolutional stack. In what has been shown below, this reduction in the image size has already been accounted for when we go from the convolutional layer to the fully-connected layer.

Your first task is to run the network as shown. You're expected to see the loss going down every epoch with a batch size of 10, learning rate of 10^{-3}, and momentum of 0.9. If you print the average loss value every 500th iteration you will notice the drop in the loss value from roughly 2.0 to 1.5 after the first epoch. The precise value of your loss at the end of the first epoch would obviously also depend on how the network was initialized by the random number generator. Let's call this single layer CNN as *Net1*.

```
class TemplateNet(nn.Module):
    def __init__(self):
        super(TemplateNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 128, 3)                              ## (A)
        self.conv2 = nn.Conv2d(128, 128, 3)                            ## (B)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(XXXX, 1000)                             ## (C)
        self.fc2 = nn.Linear(1000, XX)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        ##  Uncomment the next statement and see what happens to the
        ##  performance of your classifier with and without padding.
        ##  Note that you will have to change the first arg in the
        ##  call to Linear in line (C) above and in the line (E)
        ##  shown below.  After you have done this experiment, see
        ##  if the statement shown below can be invoked twice with
        ##  and without padding.  How about three times?
#       x = self.pool(F.relu(self.conv2(x)))                           ## (D)
        x = x.view(-1, XXXX)                                          ## (E)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Note that the value for 'XXXX' will vary for each CNN architecture and finding this parameter for each CNN is your homework task. 'XX' denotes the number of classes. In order to experiment with a network like the one shown above, your training function can be as simple as:

```
def run_code_for_training(net):
    net = net.to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(net.parameters(), lr=1e-3, momentum=0.9)
    for epoch in range(epochs):
```

```
running_loss = 0.0
for i, data in enumerate(train_data_loader):
    inputs, labels = data
    inputs = inputs.to(device)
    labels = labels.to(device)
    optimizer.zero_grad()
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
    if (i+1) % 500 == 0:
        print("\n[epoch:%d, batch:%5d] loss: %.3f" %
                    (epoch + 1, i + 1, running_loss / float(500)))
        running_loss = 0.0
```

where the parameter `net` is an instance of `TemplateNet`.

CNN Task 2: Uncomment line (D). This will now give you two convolutional layers in the network. To make the network function with this change, you must also change the number of nodes in lines (C) and (E). Line (C) defines the first part of the fully-connected neural network that will sit above the convolutional layers. With the two convolutional layers and the changed fully-connected layer, train the classifier again and see what happens to the loss. Let's refer to this CNN architecture as *Net2*.

CNN Task 3: In the TemplateNet class as shown, we used the class `torch.nn.Conv2d` class without padding. In this task, construct instances of this class with padding. Specifically, add a padding of one to the first convolutional layer. Now calculate the loss again and compare with the loss for the case when no padding was used. This is the third CNN architecture, *Net3* for this homework

Write your own code for calculating the confusion matrix for the validation part of your homework. For the given dataset, your confusion matrix will be a 10×10 array of numbers, with both the rows and the columns standing for the 10 classes in the dataset. The numbers in each row should show how the test samples corresponding to that class were correctly and incorrectly classified. You might find `scikit-learn` and `seaborn` python packages useful for this task.

For the submission, bundle your implementation in the following two files

`hw04_training.py`

`hw04_validation.py`

Depending on how you decide to implement the training and validation steps, you might need some additional helper python module that will be common to both training and validation steps, such as dataloading and the CNN architecture. Feel free to create additional helper python modules, *e.g.*, `model.py` or `dataloader.py`. This is optional and the file names and code structure for these helper modules are flexible.

The required `argparse` arguments for `hw04_training.py` and `hw04_validation.py` are as follows

```
parser = argparse.ArgumentParser(description='HW04 Training/
                                 Validation')
parser.add_argument('--root_path',  required=True, type=str)
parser.add_argument('--class_list', required=True, nargs='*',
                                 type=str)
args, args_other = parser.parse_known_args()
```

The expected output files from the training and validation calls are as follows

- Training call:

  ```
  python hw04_training.py --root_path <coco_root>/Train/
   --class_list "refrigerator" "airplane" "giraffe" "cat" "elephant"
  "dog" "train" "horse" "boat" "truck"
  ```

  should produce

  `train_loss.jpg`

  `net1.pth`

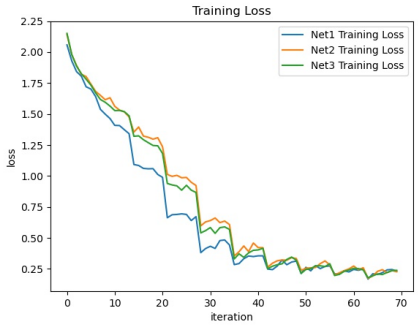
  `net2.pth`

  `net3.pth`

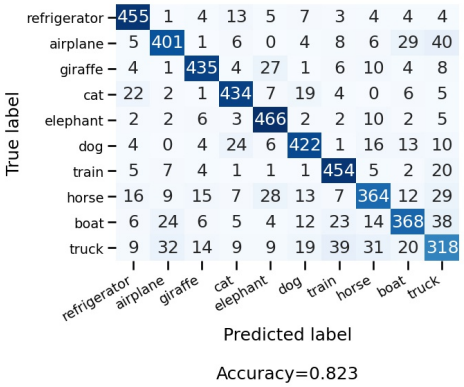
  Fig. 1a shows an example of training loss for the three CNNs.

- Validation call:

  ```
  python hw04_validation.py --root_path <coco_root>/Val/
   --class_list "refrigerator" "airplane" "giraffe" "cat" "elephant"
  "dog" "train" "horse" "boat" "truck"
  ```



(a) Training loss for the three CNNs.



(b) Sample confusion matrix.

Figure 1: Sample output, training loss and validation confusion matrix. The plotting options are flexible. Your results could vary based on your choice of hyperparamters.

should load each `net1.pth`, `net2.pth` and `net3.pth` and produce the following confusion matrices for the three CNNs using the validation set.

`net1_confusion_matrix.jpg`

`net2_confusion_matrix.jpg`

`net3_confusion_matrix.jpg`

Fig. 1b shows a sample confusion matrix on validation images.

It's important to note that your own plots and the validation accuracy could vary based on your choice of hyperparameters.

If you're wondering about the training time for the three CNNs on CPU, then just to give you a rough estimation. Depending on your hardware configuration, the per-epoch training time could vary from 10 minutes to 20 minutes with the batch size of 10. You can adjust the batch size and the number of epochs depending on your hardware configuration.

6.2 Task2: COCO Downloader

For this task, you're expected to create a script to download and downsample (64×64) a subset of COCO images using COCO API. The key learning objective here is how to query relevant image urls using cocoapi and download the images using `requests` python package similar to your HW02. Note that with COCO API this can be achieved with just 50-70 lines of python code.

Furthermore, even though the following instructions are for recreating the data provided in `hw04_data.zip`, your solution will be evaluated for generalizability. In other words, your script should be able to handle arbitrary number of classes and any number of per-class images.

To recreate the training set, you will download and downsample 2000 images per-class using the `instances_train2017.json` and 1500 images per-class using the `instances_train2014.json`. This will give you 3500 images per-class for training the CNNs. For the validation purpose, you will download and downsample 500 images per-class using the `instances_val2014.json`.

1. Create `hw04_coco_downloader.py` with the following required `argparse` arguments

```
parser = argparse.ArgumentParser(description='HW04 COCO
                                 downloader')
parser.add_argument('--root_path',  required=True, type=
                                 str)
parser.add_argument('--coco_json_path', required=True,
                                 type=str)
parser.add_argument('--class_list', required=True, nargs=
                                 '*',type=str)
parser.add_argument('--images_per_class', required=True,
                                 type=int)
args, args_other = parser.parse_known_args()
```

2. Refer to the `pycocoDemo.ipynb` demo on the cocapi GitHub repository for how to read, *e.g.*, `instances_train2017.json`, file using COCO python module. Similar to HW02, you will have to maintain a counter variable to download the required number of per-class images. Also recall the following line from HW02 for resizing images to 64×64 using the PIL python module

   ```
   im_resized = im.resize((64, 64), Image.BOX)
   ```

3. Similar to HW02, you will create a folder hierarchy as follows

   ```
   <coco_root>/Train/refrigerator/
   <coco_root>/Train/airplane/
    .
    .
    .
   <coco_root>/Train/truck/
   <coco_root>/Val/refrigerator/
   <coco_root>/Val/airplane/
    .
   ```

```
.
.
<coco_root>/Val/truck/
```

4. After the successful completion of this task, you can reproduce the training and validation sets using the following three calls

   - For COCO2017 training images

     ```
     python hw04_coco_downloader.py --root_path <coco_root>/Train/
     --coco_json_path <path_to_instances_train2017.json>
      --class_list "refrigerator" "airplane" "giraffe" "cat" "elephant"
     "dog" "train" "horse" "boat" "truck"
      --images_per_class 2000
     ```

   - For COCO2014 training images

     ```
     python hw04_coco_downloader.py --root_path <coco_root>/Train/
     --coco_json_path <path_to_instances_train2014.json>
      --class_list "refrigerator" "airplane" "giraffe" "cat" "elephant"
     "dog" "train" "horse" "boat" "truck"
      --images_per_class 1500
     ```

   - For reproducing the validation set for the homework

     ```
     python hw04_coco_downloader.py --root_path <coco_root>/Val/
     --coco_json_path <path_to_instances_val2014.json>
      --class_list "refrigerator" "airplane" "giraffe" "cat" "elephant"
     "dog" "train" "horse" "boat" "truck"
      --images_per_class 500
     ```

   The above path variables are given for Linux OS, you might have to adjust the path variables for Windows OS. As you can notice, you are expected to combine COCO2014 and COCO2017 images as your training data.

6.3 Task3: COCO DataLoader

For this task extend your dataloader from Task1 to handle arbitrary number of classes and any number of per-class images. This can be achieved by simply avoiding using hard-coded variables in your code and instead using the arguments supplied via `argparse`.

7 Submission Instructions

Your Task1 solutions will be tested for reproducibility of your training and validation results. Your Task2 and Task3 solutions will be primarily tested for generalizability. In other words, your COCO downloader and dataloader should be able to handle arbitrary number of classes and any number of per-class images.

- Make sure to submit your code in Python 3.x and not Python 2.x.
- Create a .zip archive with the following required files:

  `hw04_coco_downloader.py`

  `hw04_training.py`

  `train_loss.jpg`

  `net1.pth`

  `net2.pth`

  `net3.pth`

  `hw04_validation.py`

  `net1_confusion_matrix.jpg`

  `net2_confusion_matrix.jpg`

  `net3_confusion_matrix.jpg`

  and optionally any additional helper python modules such as `model.py`, `dataloader.py`, etc.

  Name the .zip archive as `hw04_<Firstname><Lastname>.zip` (without any white spaces). Note that .rar file format is Windows specific so please do NOT submit your solutions in .rar format. **Your code must be your own work.**

- You can resubmit a homework assignment as many times as you want up to the deadline. Each submission will overwrite any previous submission.