ECE 285 – MLIP – Assignment 3
Transfer Learning

Written by Anurag Paul and Charles Deledalle. Last Updated on April 30, 2019.

In Assignments 1 and 2, we were focusing on classification on the MNIST Dataset. In this assignment, we will focus on the best practices for managing a deep learning project and will use transfer learning for solving a classification problem. You will learn to use the PyTorch's DataLoader, and create checkpoints to stop and restart model training.

We want to learn how to predict the species of a bird given its picture. We will be using Caltech-UCSD Birds-200-2011 (CUB-200-2011) dataset. The dataset is located on DSMLP here `/datasets/ee285s-public/caltech_ucsd_birds/` and is also downloadable from http://www.vision.caltech.edu/visipedia-data/CUB-200-2011/CUB_200_2011.tgz. It has 12,000 images of 200 bird species. We will be working on a small subset of this dataset with 20 bird species having 743 training images and 372 images for validation. This directory contains a folder `CUB_200_2011` with all the images and two files: `train.csv` and `val.csv`. Each line of these files correponds to a sample described by the file path of the image, the bounding box values surrounding the bird, and the respective class label for each species from 0 to 19 (separated by commas). Given the very small size of this subset, we will rely on transfer learning (otherwise we will be facing the curse of dimensionality).

It is possible that DSMLP will be too busy to let you run everything from this assignment. We know what the results should look like, so we will only grade the code.

1 Getting started

First of all, connect to DSMLP (`ieng6` server) and start a *pod* with enabled GPU/CUDA capabilities

```
$ launch-py3torch-gpu.sh
```

Next connect to your Jupyter Notebook from your web browser (refer to Assignment 0 for more details). Create a new notebook `assignment3.ipynb` and import

```
%matplotlib notebook

import os
import numpy as np
import torch
from torch import nn
from torch.nn import functional as F
import torch.utils.data as td
import torchvision as tv
import pandas as pd
from PIL import Image
from matplotlib import pyplot as plt
```

If any of the above libraries is not available, install it using

```
$ pip install --user <library_name>
```

Select the relevant device

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

For the following questions, please write your code and answers directly in your notebook. Organize your notebook with headings, markdown and code cells (following the numbering of the questions).

2 Data Loader

In order to start training a classifier, we first need to build routines for loading the data. We will achieve this using the data management tools provided in the package `torch.utils.data`.

1. Create a variable `dataset_root_dir` and make it point to the Bird dataset directory.

   Advice: a good habit is to set the value of such a variable according to ***`socket.gethostname()`*** *and* ***`getpass.getuser()`****. This allows you and your collaborators to use the same piece of code on different hosts or clusters in which data may be stored at different locations.*

2. Central to `torch.utils.data`, is the abstract class `Dataset` that will be useful for managing our training and testing data. Please refer to the documentation here https://pytorch.org/docs/stable/data.html. Interpret and complete the following piece of code:

```
class BirdsDataset(td.Dataset):

    def __init__(self, root_dir, mode="train", image_size=(224, 224)):
        super(BirdsDataset, self).__init__()
        self.image_size = image_size
        self.mode = mode
        self.data = pd.read_csv(os.path.join(root_dir, "%s.csv" % mode))
        self.images_dir = os.path.join(root_dir, "CUB_200_2011/images")

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

    def __repr__(self):
        return "BirdsDataset(mode={}, image_size={})". \
               format(self.mode, self.image_size)

    def __getitem__(self, idx):
        img_path = os.path.join(self.images_dir, \
                                self.data.iloc[idx]['file_path'])
        bbox = self.data.iloc[idx][['x1', 'y1', 'x2', 'y2']]
        img = Image.open(img_path).convert('RGB')
        img = img.crop([bbox[0], bbox[1], bbox[2], bbox[3]])
        transform = tv.transforms.Compose([
            # COMPLETE
            ])
```

```
        x = transform(img)
        d = self.data.iloc[idx]['class']
        return x, d

    def number_of_classes(self):
        return self.data['class'].max() + 1
```

The method `__getitem__` returns the image `x` of index `idx` together with its class label `d`. It crops the image according to the bounding box indicated in the `csv` file. Refer to https://pytorch.org/docs/stable/torchvision/transforms.html and complete the relevant section to use `torchvision` to resize it to the size indicated by `image_size`, convert it to a tensor and then normalize it to the range $[-1, 1]$.

3. Copy paste the following function

```
def myimshow(image, ax=plt):
    image = image.to('cpu').numpy()
    image = np.moveaxis(image, [0, 1, 2], [2, 0, 1])
    image = (image + 1) / 2
    image[image < 0] = 0
    image[image > 1] = 1
    h = ax.imshow(image)
    ax.axis('off')
    return h
```

Create the object `train_set` as an instance of `BirdsDataset`. Sample the element with index 10. Store it in a variable `x`. Use `myimshow` function to display the image `x`.

4. The main advantage of using PyTorch's `Dataset` is to use its data loader mechanism with `DataLoader`. Read the documentation and create `train_loader`: the object that loads the training set and split it into shuffled mini-batches of size `B=16`. Use `pin_memory=True`. What is the advantage of using `pin_memory`? How many mini-batches are there?

5. In a different cell, display the first image and label pair of the first four mini-batches. Re-evaluate your cell, what do you observe?

6. Also create `val_set` as an instance of `BirdsDataset` using `mode='val'`. Then, create `val_loader` similar to `train_loader` but without shuffle. Why do you think we need to shuffle the dataset for training but not for validation?

3 Abstract Neural Network Model

In this section, we will use the mechanism of abstract classes and inheritance to define all the common methods which will be shared between different deep learning models that we will build in the future works in this course and beyond. These functionalities are provided to you in a homemade package `/datasets/ee285s-public/nntools.py`. From your terminal, create a symbolic link on this package into your working/current directory

```
$ ln -s /datasets/ee285s-public/nntools.py .
```

Back to your Jupyter Notebook, you can now import its functions as:

```
import nntools as nt
```

A main concept introduced in `nntools` is the abstract class `NeuralNetwork`. This class describes the general architecture and functionalities that a neural network object is expected to have. In particular it has two methods `forward` and `criterion` used respectively to make a forward pass in the network and compute the loss. Read its documentation by typing `help(nt.NeuralNetwork)` and next open `nntools.py` to inspect its code. As you can observe these methods are tagged as *abstract* and as a result the class is said to be abstract. Note you already used abstract classes: `nn.Module` and `nn.Dataset`.

7. Try to instantiate a neural network as `net = nt.NeuralNetwork()`. What do you observe?

An abstract class does not implement all of its methods and cannot be instantiated. This is because the implementation of `forward` and `criterion` will depend on the specific type and architecture of neural networks we will be considering. The implementation of these two methods will be done in sub-classes following the principle of inheritance.

For instance, we can define the subclass `NNClassifier` that inherits from `NeuralNetwork` and implements the method `criterion` as being the cross entropy loss

```
class NNClassifier(nt.NeuralNetwork):

    def __init__(self):
        super(NNClassifier, self).__init__()
        self.cross_entropy = nn.CrossEntropyLoss()

    def criterion(self, y, d):
        return self.cross_entropy(y, d)
```

Compare to `NeuralNetwork`, this class is more specific as it considers only neural networks that will produce one-hot codes and that are then classifiers. Nevertheless, this class is still abstract as it does not implement the method `forward`. Indeed, the method `forward` still depend on the specific architecture of the classification network we will be considering.

4 VGG-16 Transfer Learning

We are going to consider a new classifier built on the principle of transfer learning from a pretrained network. VGG (by Simonyan and Zisserman, 2014) was runner-up of the ILSVRC-2014 challenge. It is a very popular network for transfer learning and is widely used as a feature extractor for multiple computer vision tasks. Here, we will use the 16-layer version of the VGG with batch norm from `torchvision` package (`vgg16_bn`). We will replace the final fully connected (FC) layer with a one specific to our task. We will then train only this task-specific last FC layer and will keep all other layers as frozen (*i.e.*, they will not be trained). The main advantage of transfer learning is that we are enabling transfer of knowledge gained by model on one task to be adapted to learn another task. It also reduces drastically the number of parameters to learn, hence, avoiding overfitting and making the training to converge in just a few epochs.

8. In your notebook, evaluate the following

```
vgg = tv.models.vgg16_bn(pretrained=True)
```

Print the network and inspect its learnable parameters (as done in Assignment 2).

9. Copy/paste the code of `NNClassifier` in your notebook, and create a new subclass `VGG16Transfer` that inherits from `NNClassifier` by completing the following:

```
class VGG16Transfer(NNClassifier):

    def __init__(self, num_classes, fine_tuning=False):
        super(VGG16Transfer, self).__init__()
        vgg = tv.models.vgg16_bn(pretrained=True)
        for param in vgg.parameters():
            param.requires_grad = fine_tuning
        self.features = vgg.features
        # COMPLETE
        num_ftrs = vgg.classifier[6].in_features
        self.classifier[6] = nn.Linear(num_ftrs, num_classes)

    def forward(self, x):
        # COMPLETE
        y = self.classifier(f)
        return y
```

Note that `fine_tuning=True` can be used if we were willing to retrain (fine tune) all other layers, but this will take much more time and memory. This would also require more data (or using a very small learning rate and carefully monitoring the loss) as the number of parameters would be much higher, and the procedure subject to overfitting.

10. The class `VGG16Transfer` is no longer abstract as it implements all of the methods of its ancestors. Create an instance of this class for a classification problem with a number of classes specified as `num_classes = train_set.number_of_classes()`. Print the network and inspect its learnable parameters.

5 Training experiment and checkpoints

The package `nntools` introduces another mechanism for running learning experiments. An important aspect when running such an experiment is to regularly create checkpoints or backups of the current model, optimization state and statistics (training loss, validation loss, accuracy, etc). In case of an unexpected error, you need to be able to restart the computation from where it stopped and you do not want to rerun everything from scratch. Typical reasons for your learning to stop are server disconnection/timeout, out of memory errors, CUDA runtime errors, quota exceeded error, etc.

The computation of statistics will be delegated to the class `StatsManager`, that provides functionalities to accumulate statistics from different mini batches and then aggregate/summarize the information at the end of each epoch. Read and interpret the code of `StatsManager`. This class is not abstract since it implements all of its methods. We could use an instance of this class to monitor the learning for our classification problem. But this class is too general and then does not compute classification accuracy. Even though the class is not abstract, we can still create a subclass by inheritance and redefine some of its methods, this is called overloading.

11. Create a new subclass `ClassificationStatsManager` that inherits from `StatsManager` and overload each method in order to track the accuracy by completing the following code

```
class ClassificationStatsManager(nt.StatsManager):
```

```
    def __init__(self):
        super(ClassificationStatsManager, self).__init__()

    def init(self):
        super(ClassificationStatsManager, self).init()
        self.running_accuracy = 0

    def accumulate(self, loss, x, y, d):
        super(ClassificationStatsManager, self).accumulate(loss, x, y, d)
        _, l = torch.max(y, 1)
        self.running_accuracy += torch.mean((l == d).float())

    def summarize(self):
        loss = super(ClassificationStatsManager, self).summarize()
        accuracy = 100 * # COMPLETE
        return {'loss': loss, 'accuracy': accuracy}
```

Experiments will be carried out by the class `Experiment` which is defined with respect to 6 inputs

- a given network,
- a given optimizer,
- a given training set,
- a given validation set,
- a given mini batch size,
- a given statistic manager.

Once instantiated, the experiment can be run for n epochs on the training set by using the method `run`. The statistics at each iteration are stored as a list in the attribute `history`.

12. An experiment can be evaluated on the validation set by the method `evaluate`. Read the code of that method and note that first `self.net` is set to `eval` mode. Read the documentation https://pytorch.org/docs/stable/nn.html#torch.nn.Module.eval and explain why we use this.

13. The `Experiment` class creates a checkpoint at each epoch and automatically restarts from the last available checkpoint. The checkpoint will be saved into (or loaded from) the directory specified by the optional argument `output_dir` of the constructor. If not specified, a new directory with an arbitrary name is created. Take time to read and interpret carefully the code of `Experiment` and run the following

```
lr = 1e-3
net = VGG16Transfer(num_classes)
net = net.to(device)
adam = torch.optim.Adam(net.parameters(), lr=lr)
stats_manager = ClassificationStatsManager()
exp1 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
                     output_dir="birdclass1", perform_validation_during_training=True)
```

    Check that a directory `birdclass1` has been created and inspect its content. Visualize the file `config.txt`. What does the file `checkpoint.pth.tar` correspond to?

14. Change the learning rate to `1e-4` and re-evaluate the cell. What do you observe? Change it back to `1e-3` and re-evaluate the cell. What do you observe? Why?

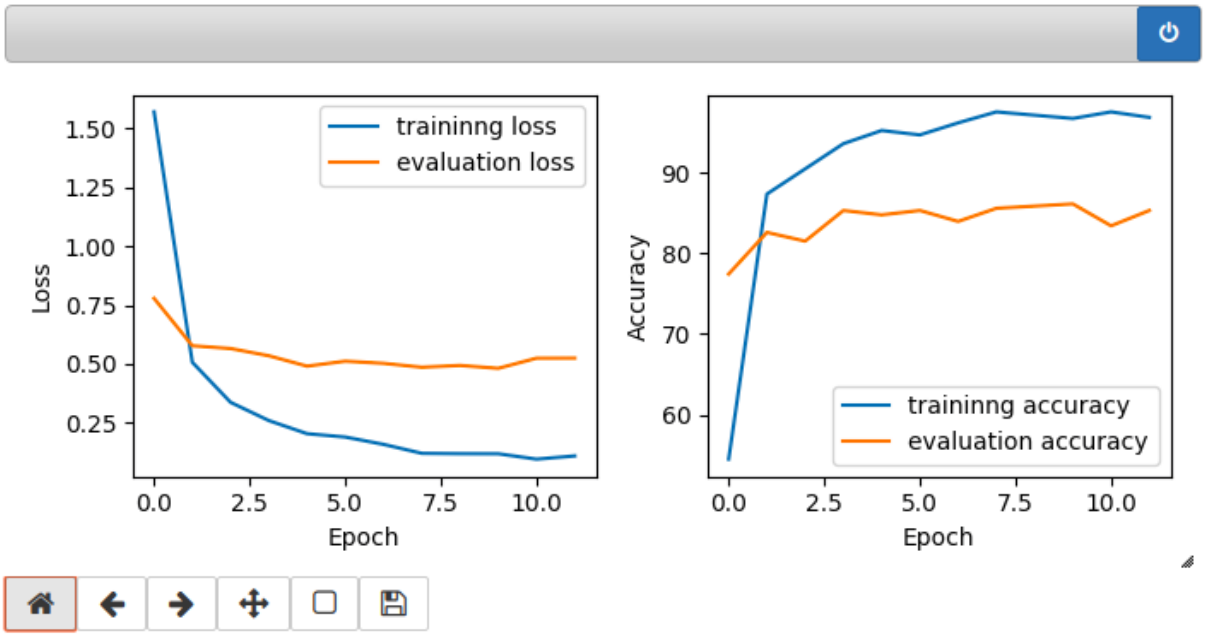
15. Copy and complete the following code running the experiment for 20 epochs

```
def plot(exp, fig, axes):
    axes[0].clear()
    axes[1].clear()
    axes[0].plot([exp.history[k][0]['loss'] for k in range(exp.epoch)],
                 label="traininng loss")
    # COMPLETE
    plt.tight_layout()
    fig.canvas.draw()
```

```
fig, axes = plt.subplots(ncols=2, figsize=(7, 3))
exp1.run(num_epochs=20, plot=lambda exp: plot(exp, fig=fig, axes=axes))
```

and displaying two plots side-by-side, one showing the losses and the other showing the evolution of accuracy over the epochs. The training should take about 8 minutes. Make sure your loss evolutions are consistent with the ones below. If they are not so, interupt it (`Esc+i+i`), check your code, delete the `output_dir`, and start again.

In [*]:
```
fig, axes = plt.subplots(ncols=2, figsize=(7, 3))
exp1.run(num_epochs=20, plot=lambda exp: plot(exp, fig=fig, axes=axes))
```




```
Start/Continue training from epoch 0
Epoch 1 (Time: 13.78s)
Epoch 2 (Time: 25.13s)
Epoch 3 (Time: 24.94s)
Epoch 4 (Time: 24.44s)
Epoch 5 (Time: 24.79s)
Epoch 6 (Time: 24.90s)
Epoch 7 (Time: 25.17s)
Epoch 8 (Time: 25.09s)
Epoch 9 (Time: 24.55s)
Epoch 10 (Time: 24.43s)
Epoch 11 (Time: 24.24s)
Epoch 12 (Time: 24.36s)
```

6 ResNet18 Transfer Learning

ResNet (by He *et al.*, 2015) is an ultra deep network with up to 152 layers which won the ILSVRC 2015 challenge. Pretrained models of ResNet are available in `torchvision` for 18, 34, 50, 101 and 152 layer versions. We will be using the 18 layer version for this assignment that can be loaded as

```
resnet = tv.models.resnet18(pretrained=True)
```

16. Similar to `VGG16Transfer`, create a subclass `Resnet18Transfer` that inherits from `NNClassifer` and that redefines the last FC layer.

17. Create an instance of `Resnet18Transfer` and create a new experiment `exp2` making backups in the directory `birdclass2`. Run the experiment for 20 epochs with Adam and learning rate `1e-3` using the same function `plot` as before.

18. Using the method `evaluate` of `Experiment`, compare the validation performance obtained by `exp1` and `exp2` using respectively `VGG16Transfer` and `Resnet18Transfer`.

7 Bonus (optional and ungraded)

19. Create a new subclass of `StatsManager` that implements top-k accuracy statistic.

20. Try different batch sizes, learning rates and try to achieve the best validation accuracy.

21. Try other pretrained models from `torchvision` and see if some other model performs better than VGG16 or Resnet18 on this task.