CS540 Spring 2024 Homework 6

Assignment Goals

- Get Pytorch set up for your environment.
- Familiarize yourself with the tools.
- Implementing and training a basic neural network using Pytorch.
- Happy deep learning :)

Summary

Home-brewing every machine learning solution is not only time-consuming but potentially error-prone. One of the reasons we're using Python in this course is because it has some very powerful machine learning tools. Besides common scientific computing packages such as SciPy and NumPy, it's very helpful in practice to use frameworks such as Scikit-Learn, TensorFlow, PyTorch, and MXNet to support your projects. The utilities of these frameworks have been developed by a team of professionals and undergo rigorous testing and verification.

In this homework, we'll be exploring the PyTorch framework. You will complete the functions in the starter code provided, intro_pytorch.py, following the instructions below.

Part 1: Setting up the Python Virtual Environment

In this assignment, you will familiarize yourself with the Python Virtual Environment. Working in a virtual environment is an important part of working with modern ML platforms, so we want you to get a flavor of that through this assignment. Why do we prefer virtual environments? Virtual environments allow us to install packages within the virtual environment without affecting the host system setup. So you can maintain project-specific packages in respective virtual environments.

You can work on your own machine but remember to test on Gradescope. The following are the installation steps for Linux. If you don't have a Linux computer, you can use the CS lab computers for this homework. Find more instructions: How to access CSL Machines Remotely. For example, you can connect to the CSL Linux computers by using `ssh` along with your CS account username and password. In your terminal simply type:

```
ssh {csUserName}@best-linux.cs.wisc.edu
```

You can use `scp` to transfer files: `scp source destination`. For example, to upload a file to the CSL machine:

```
scp Desktop/intro_pytorch.py {csUserName}@best-linux.cs.wisc.edu:/home/{csUserName}
```

You will be working on Python 3 (instead of Python 2 which is no longer supported) with Python version >= 3.8. Read more about PyTorch and Python version here. To check your Python version use:

```
python -V or python3 -V
```

If you have an alias set for python=python3 then both should show the same version (3.x.x)

Step 1: For simplicity, we use the venv module (feel free to use other virtual envs such as Conda).

To set up a Python Virtual Environment, use the following:

```
python3 -m venv /path/to/new/virtual/environment
```

For example, if you want to set up a virtual environment named `Pytorch` in your working directory:

```
python3 -m venv Pytorch
```

(Optional: If you want to learn more about Python virtual environments, a very good tutorial can be found here.)

Step 2: Activate the virtual environment:
Suppose the name of our virtual environment is `Pytorch` (you can use any other name if you want). You can activate the environment by the following command:

```
source Pytorch/bin/activate
```

Step3: From your virtual environment shell, run the following commands to upgrade pip (the Python package installer) and install the CPU version of PyTorch. (It may take some time.)

```
pip install --upgrade pip
pip install torch==2.1.0 torchvision==0.16.0 torchaudio==2.1.0
pip install numpy==1.26.4
```

You can check the versions of the packages installed using the following command:

```
pip freeze
```

Note: to deactivate the virtual environment, just type

```
deactivate
```

Part 2: Build Your First Neural Network

In this section, we will guide you step by step to build a simple deep learning model for predicting labels of handwritten images. You will learn how to build, train, evaluate the model, and to make predictions on test data using this model.

You will implement the following functions in Python.

- **get_data_loader(training=True)**
  - Input: an optional boolean argument (default value is True for training dataset)
  - Return: Dataloader for the training set (if training = True) or the test set (if training = False)
- **build_model()**
  - Input: none
  - Return: an untrained neural network model
- **train_model(model, train_loader, criterion, T)**
  - Input: the model produced by the previous function, the train DataLoader produced by the first function, the criterion for measuring model performance, and the total number of epochs T for training
  - Return: none
- **evaluate_model(model, test_loader, criterion, show_loss=True)**
  - Input: the trained model produced by the previous function, the test DataLoader, and the criterion.
  - It prints the evaluation statistics as described below (displaying the loss metric value if and only if the optional parameter has not been set to False)
  - Return: none
- **predict_label(model, test_images, index)**
  - Input: the trained model, test images (tensor of dimension $N \times 1 \times 28 \times 28$), and an index
  - It prints the top 3 most likely labels for the image at the given index, along with their probabilities
  - Return: none

You are free to implement any other utility function. But we will only be testing the functionality using the above 5 APIs, so make sure that each of them follows the exact function signature and returns. You can also use helper methods to visualize the images from the FashionMNIST dataset for a better understanding of the dataset and the labels. But it is entirely optional and does not carry any points.

Import necessary packages

Here are some of the useful modules that may help us save a ton of effort in the project:

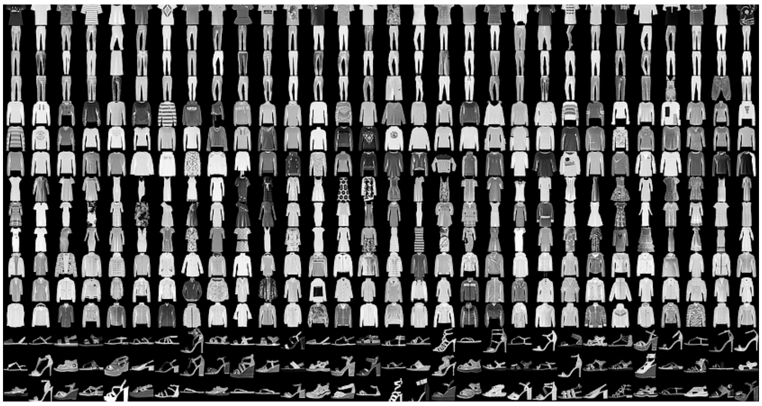
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
```

`torch`, `torchvision` and the Python standard packages are the only imports allowed on this assignment. The autograder will likely not handle any other packages.

The following 5 sections explain the details for each of the above functions you are required to implement.

Get the DataLoader

We will use the Fashion-MNIST dataset, each example is a 28×28 grayscale image, associated with a label from 10 classes.



Hint 1: Note that PyTorch already contains various datasets for you to use, so there is no need to manually download from the Internet. Specifically, the function

```
torchvision.datasets.FashionMNIST()
```

can be used to retrieve and return a Dataset object torchvision.datasets.FashionMNIST, which is a wrapper that contains image inputs (as 2D arrays) and labels ('T-shirt/top', ' Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt','Sneaker', 'Bag', 'Ankle Boot'):

```
train_set=datasets.FashionMNIST('./data',train=True,
            download=True,transform=custom_transform)
test_set=datasets.FashionMNIST('./data', train=False,
            transform=custom_transform)
```

The `train_set` contains images and labels we'll be using to train our neural network; the `test_set` contains images and labels for model evaluation. Here we set the location where the dataset is downloaded as the data folder in the current directory.
Note that input preprocessing can be done by specifying transform as our custom_transform (you don't need to change this part)

```
custom_transform= transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
    ])
```

- In the above, `transforms.To_Tensor()` converts a PIL Image or numpy.ndarray to tensor.

- `transforms.Normalize()` normalizes the tensor with a mean and standard deviation which goes as the two parameters respectively. Feel free to check the official doc for more details.

Hint 2: After obtaining the dataset object, you may wonder how to retrieve images and labels during training and testing. Luckily, PytTorch provides such a class called torch.utils.data.DataLoader that implements the iterator protocol. It also provides useful features such as:

- Batching the data
- Shuffling the data
- Load the data in parallel using multiprocessing.
- ...

Below is the full signature of the DataLoader class (for more details, check here):

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None, *, prefetch_factor=2,
           persistent_workers=False)
```

As an introductory project, we won't use complicated features. We ask you to set the batch_size = 64 for both train loader and test loader. Besides, set shuffle=False for the test loader. Given a Dataset object data_set, we can obtain its DataLoader as follows:

```
loader = torch.utils.data.DataLoader(data_set, batch_size = 64)
```

Putting it all together, you should be ready to implement the get_data_loader() function. Note that when the optional argument is unspecified, the function should return the Dataloader for the training set. If the optional argument is set to False, the Dataloader for the test set is returned. The expected output is as follows:

```
>>> train_loader = get_data_loader()
>>> print(type(train_loader))
<class 'torch.utils.data.dataloader.DataLoader'>
>>> print(train_loader.dataset)
Dataset FashionMNIST
    Number of datapoints: 60000
    Root location: ./data
    Split: Train
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.1307,), std=(0.3081,))
           )
>>> test_loader = get_data_loader(False)
```

Build Your Model

After setting up the data loaders, let's build the model we're going to use with the datasets. Neural networks in PyTorch are composed of layers. You've heard about layers in the lectures, but take a minute to look through this simple example (it's nice and short) to get an idea of what the implementation logistics will look like. We will use the following layers (in the order specified below):

1. A Flatten layer to convert the 2D pixel array to a 1D array.
2. A Dense layer with 128 nodes and a ReLU activation.
3. A Dense layer with 64 nodes and a ReLU activation.
4. A Dense layer with 10 nodes.

In this assignment, you are expected to use a Sequential container to hold these layers. As a fun practice, we ask you to fill out the positions marked with "?" with the appropriate parameters.

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(?, ?),
    nn.ReLU()
    nn.Linear(?, ?),
    ...
    )
```

After building the model, the expected output be as below. Note that the Flatten layer just serves to reformat the data.

```
>>> model = build_model()
>>> print(model)
Sequential(
  (0): Flatten()
  (1): Linear(in_features=?, out_features=?, bias=True)
  (2): ReLU()
  (3): Linear(in_features=?, out_features=?, bias=True)
  ...
)
```

Note: Be careful not to add large parameter sized model to Gradescope. The auto-grader will throw a timeout error on doing so.

Train Your Model

After building the model, now we are ready to implement the training procedure. One of the parameters of train_model(..., criterion, ...) is the criterion, which can be specified as (we will also use this in the autograder):

```
criterion = nn.CrossEntropyLoss()
```

Here we use the cross-entropy loss nn.CrossEntropyLoss(), which combines nn.LogSoftmax() and nn.NLLLoss(). Inside the function train_model(), you may need to pick your favorite optimization algorithm by setting up an optimizer first: here we use stochastic gradient descent (SGD) with a learning rate of 0.001 and momentum of 0.9:

```
opt = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

A note on the major difference between gradient descent (GD) and SGD: in GD, all samples in the training set are used to update parameters in a particular iteration; while in SGD, only a random subset of training samples are used to update parameters in a particular iteration. SGD often converges much faster than GD for large datasets.

The standard training procedure contains 2 for loops: the outer for loop iterates over epochs, while the inner for loop iterates over batches of (images, labels) pairs from the train DataLoader. Feel free to check the Train the network part in this official tutorial for more details. Please pay attention to the order of the three commands `zero_grad()`, `backward()` and `step()`. These commands serve distinctive functions in the backpropogation step, which result in the model weights being updated. A kind reminder: please set your model to train mode before iterating over the dataset. This can be done with the following call:

```
model.train()
```

We ask you to print the training status after every epoch of training in the following format (it should have 3 components per line):
Train Epoch: ? Accuracy: ?/?(??.??%) Loss: ?.???
Then the training process (for 5 epochs) will be similar to the following (numbers can be different):

```
Train Epoch: 0   Accuracy: 42954/60000(71.59%) Loss: 0.833
Train Epoch: 1   Accuracy: 49602/60000(82.67%) Loss: 0.489
Train Epoch: 2   Accuracy: 50730/60000(84.55%) Loss: 0.436
Train Epoch: 3   Accuracy: 51383/60000(85.64%) Loss: 0.405
Train Epoch: 4   Accuracy: 51820/60000(86.37%) Loss: 0.383
```

Here are a few specific requirements for the format:

- We count the first epoch as Epoch 0
- All the information should be summarized in one line for each epoch. (e.g. in total you should print 5 lines if you train for 5 epochs)
- Accuracy (with 2 decimal places) in percentage should be put inside parentheses
- Accuracy should be printed before Loss
- Loss (with 3 decimal places) denotes the average loss per epoch (sum of all images' loss in an epoch divided by number of images in the dataset). Note that `nn.CrossEntropyLoss()` by default makes `loss.item()` return the average loss of one batch instead of the total loss. Also, you may want to consider if all batches' sizes are the same.
- You should be able to reach at least 80% accuracy after 5 epochs of training.

Evaluate Your Model

After the model is trained, we need to evaluate how good it is on the test set. The process is very similar to that of training, except that you need to turn the model into evaluation mode:

```
model.eval()
```

Besides, there is no need to track gradients during testing, which can be disabled with the context manager:

```
with torch.no_grad():
    for data, labels in test_loader:
        ...
```

You are expected to print both the test Loss and the test Accuracy if show_loss is set to True (print Accuracy only otherwise) in the following format:

```
>>> evaluate_model(model, test_loader, criterion, show_loss = False)
Accuracy: 85.39%

>>> evaluate_model(model, test_loader, criterion, show_loss = True)
Average loss: 0.4116
Accuracy: 85.39%
```

Format the Accuracy with two decimal places and the accuracy should be shown as a percentage. Format the Loss with four decimal places. The loss should be printed in a separate line before Accuracy (as shown above).

Predict the Labels

Instead of testing on a whole dataset, sometimes it's more convenient to examine the model's output on a single image.

As it's easier for humans to read and interpret probabilities, we need to use a Softmax function to convert the output of your final Dense layer into probabilities (note that by default your model outputs logits). Generally, Softmax is often used as the activation for the last layer of a classification network because the result can be interpreted as a categorical distribution. Specifically, once we obtain the logits, we can use:

```
prob = F.softmax(logits, dim=?)
```

You can assume the input test_images in `predict_label(model, test_images, index)` is a torch tensor with the shape Nx1x28x28. Your implementation should display the top three most likely class labels (in descending order of predicted probability; three lines in total) for the image at the given index along with their respective probabilities in percentage (again, your output will vary in its exact numbers but should follow the format below):

```
>>> test_images = next(iter(test_loader))[0]
>>> predict_label(model, test_images, 1)
Pullover: 92.48%
Shirt: 5.93%
Coat: 1.48%
```

The index are assumed to be valid. We assume the class names are (note that there is no white space in any class name):

```
class_names = ['T-shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt'
                ,'Sneaker','Bag','Ankle Boot']
```

Deliverable

A single file named intro_pytorch.py containing the methods mentioned in the program specification section. Please pay close attention to the format of the print statements in your functions. Incorrect format will lead to point deduction.

Submission

Please submit your file "intro_pytorch.py" to Gradescope. Do ***not*** submit a Jupyter notebook .ipynb file. All code except imports should be contained in functions ***or*** under the following check:

```
if __name__=="__main__":
```

so that it will not run if your code is imported to another program.

This assignment's due date is on Canvas. We strongly encourage you to start working on it early.