HW6: Neural Networks

Due Mar 21, 2021 by 2:30pm **Points** 100 **Submitting** a file upload
Available until Mar 21, 2021 at 2:30pm

This assignment was locked Mar 21, 2021 at 2:30pm.

Assignment Goals

- Get Pytorch set up for your environment
- Familiarize yourself with the tools
- Perform some basic neural network tasks using Pytorch's utilities
- Happy deep learning! :)

Summary

Home-brewing every machine learning solution is not only time-consuming but potentially error-prone (as you may have discovered). 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, as their utilities have been developed by a team of professionals and undergo rigorous testing and verification.

In this homework, we'll be exploring the **Pytorch** **(https://pytorch.org/)** framework. Please complete the functions in this template: **intro_pytorch.py** **(https://canvas.wisc.edu/courses/230450/files/18735425/download?download_frd=1)** .

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.

We suggest that you use the CS lab computers for this homework. You can also work on your personal system for the initial development, but finally, you will have to test your model on the CSL lab computers. Find more instructions: **How to access CSL Machines Remotely** **(https://csl.cs.wisc.edu/)**

The following are the installation steps for Linux (CSL machines are recommended).

You will be working on Python 3 (instead of Python 2 which is no longer supported). Read more about Pytorch and Python version **here (https://pytorch.org/get-started/locally/)**. 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)

Step1: For simplicity, we use the **venv (https://docs.python.org/3/library/venv.html)** module (feel free to use other virtual envs such as **Conda (https://www.anaconda.com/)**). To set up a Python Virtual Environment named Pytorch:

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

For example, if you want to put the virtual environment in your home directory:

```
python3 -m venv --system-site-packages ~/Pytorch
```

If you want to learn more about Python virtual environments, a very good tutorial can be found **here (https://realpython.com/python-virtual-environments-a-primer/)**.

Here the name of our virtual environment is Pytorch (you can use any other name if you want).

Step2: Activate the environment:

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

Step3: From your virtual environment shell, run the following commands to upgrade pip and install the CPU version of Pytorch 1.8 (this homework only uses the basic functionality of Pytorch, so feel free to use older versions such as 1.6):

```
pip install --upgrade pip
```

```
pip install torch==1.8.0+cpu torchvision==0.9.0+cpu torchaudio==0.8.0 -f https://download.pytorc
h.org/whl/torch_stable.html
```

You can check the version 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 models, and make predictions on test data using this model. We expect you to 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, and the 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 and test images,
  - 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 MNIST dataset for a better understanding of the dataset and the labels. But it is totally optional and does not carry any points.

Import necessary packages

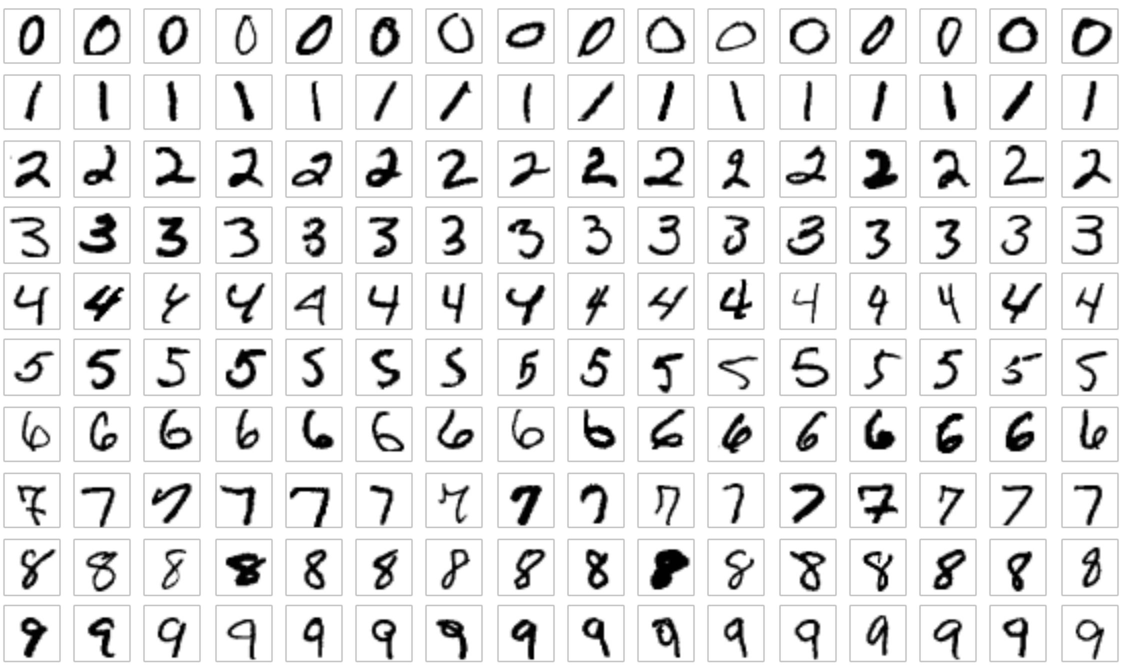
Here are some of the useful modules that may help us save a ton of efforts in the project:

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

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

Get the DataLoader

We will use the "hello world" dataset **MNIST** (http://yann.lecun.com/exdb/mnist/) which consists of handwritten digits from 0 to 9.



Hint1: note that Pytorch already contains various datasets for you to use, so there is no need to manually download from the Internet. Specifically, `torchvision.datasets.MNIST()` can be used to retrieve and return a Dataset object `torchvision.datasets.mnist.MNIST` which is a wrapper that contains image inputs (as 2D arrays) and labels (0 to 9) representing handwritten numbers:

```
train_set = datasets.MNIST('./data', train=True, download=True,
                       transform=custom_transform)
test_set = datasets.MNIST('./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. If you encounter HTTP Error 403, check **[here. (https://github.com/pytorch/vision/issues/1938)](https://github.com/pytorch/vision/issues/1938)**

Note that input preprocessing can be done by specifying `transform` as our `custom_transform`:

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

- `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 (https://pytorch.org/docs/stable/torchvision/transforms.html?highlight=normalize#torchvision.transforms.Normalize)](https://pytorch.org/docs/stable/torchvision/transforms.html?highlight=normalize#torchvision.transforms.Normalize)** for more details.

Hint2: After obtaining the dataset object, you may wonder how to retrieve images and labels during training and testing. Luckily, Pytorch 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 (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` = 50 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 = 50)
```

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 MNIST
    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 these in the lecture, but take a minute to look through this **simple example (https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py)** (it's nice and short) to get an idea of what the implementation logistics will look like.

We use the following layers in this order:

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. (https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity)**
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 (https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html)** container to hold these layers:

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

After building the model, the expected output should be like this:

```
>>> 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 that the Flatten layer just serves to reformat the data.

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:

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

Here we use the cross-entropy loss `nn.CrossEntropyLoss()` (this criterion 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 (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py)** for more details (please pay attention to the order of `zero_grad()`, `backward()` and `step()`).

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_model(model, train_loader, criterion, T = 5)
Train Epoch: 0   Accuracy: 47771/60000(79.62%)  Loss: 0.740
Train Epoch: 1   Accuracy: 54914/60000(91.52%)  Loss: 0.293
Train Epoch: 2   Accuracy: 55894/60000(93.16%)  Loss: 0.236
Train Epoch: 3   Accuracy: 56597/60000(94.33%)  Loss: 0.196
Train Epoch: 4   Accuracy: 57142/60000(95.24%)  Loss: 0.166
```

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 in percentage should be put inside parentheses
- Accuracy should be printed before Loss
- Loss denotes the average loss per epoch (accumulated loss in an epoch / length of the dataset)

You should be able to reach at least 90% 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: 97.92%

>>> evaluate_model(model, test_loader, criterion, show_loss = True)
Average loss: 0.0015
Accuracy: 97.92%
```

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 (https://pytorch.org/docs/stable/nn.functional.html?highlight=softmax#torch.nn.functional.softmax)** function to convert the output of your final Dense layer into probabilities (note that by default your model outputs **logits (https://developers.google.com/machine-learning/glossary#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** (assumed to be valid i.e. $0 \leq i \leq N-1$) along with their respective probabilities in percentage (again, your output will vary in its exact numbers but should follow the format below):

```
>>> predict_label(model, pred_set, 1)
two: 99.41%
three: 0.42%
one: 0.09%
```

where we assume the class names are:

```
class_names = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine']
```

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 files in a zip file named **hw6_<netid>.zip**, where you replace <netid> with your netID (your wisc.edu login). Inside your zip file, there should be one single file named:

intro_pytorch.py. Do not submit a Jupyter notebook .ipynb file.

All code should be contained in functions OR under a

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

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

Be sure to **remove all debugging output** before submission. Failure to remove debugging output will be **penalized (10pts)**.

If a regrading request isn't justifiable (the initial grade is correct and clear, subject to the instructors' judgment), the request for regrading will be **penalized (10 pts)**.

This assignment is due on March 19, 2:30 PM. We have extended the due date by 3 days due to the midterm. There will be no additional extensions beyond this. It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

Rubric

Criteria	Ratings		Pts
get_data_loader() successfully returns the desired train and test loader	**20 pts Full Marks**	**0 pts No Marks**	20 pts
build_model() returns an untrained, Sequential model with the specified components	**20 pts Full Marks**	**0 pts No Marks**	20 pts
train_model() trains the model for the specified epochs and the print format is correct	**25 pts Full Marks**	**0 pts No Marks**	25 pts
evaluate_model() displays different output based on its optional argument	**5 pts Full Marks**	**0 pts No Marks**	5 pts
predict_label() displays correctly formatted output; predict_label() displays expected top-3 categories; predict_label() displays probabilities, not logits	**30 pts Full Marks**	**0 pts No Marks**	30 pts
			Total Points: 100