PyTorch Tutorial

Setup using Virtual Environments

Virtual environments are a great way to package your dependencies for different projects. You will be using them a lot in the future so we highly reccommend setting up a virtual environment for this homework. If you are unfamiliar with using virtual environments, follow the following instructions for setting up a virtual environment using **Anaconda**.

This is optional but highly reccommended and it will make development in the future much easier. If you get stuck on this part, just move on.

0. If you don't already have Anaconda, install it here: https://www.anaconda.com/distribution/#macos (https://www.anaconda.com/distribution/#macos)
1. Create the environment by running: `conda create -n nmep python=3.6`
2. Enter the virtual environment: `conda activate nmep`

   NOTE: If you get a `CommandNotFoundError` , you can run `conda init bash` and then open a new window.

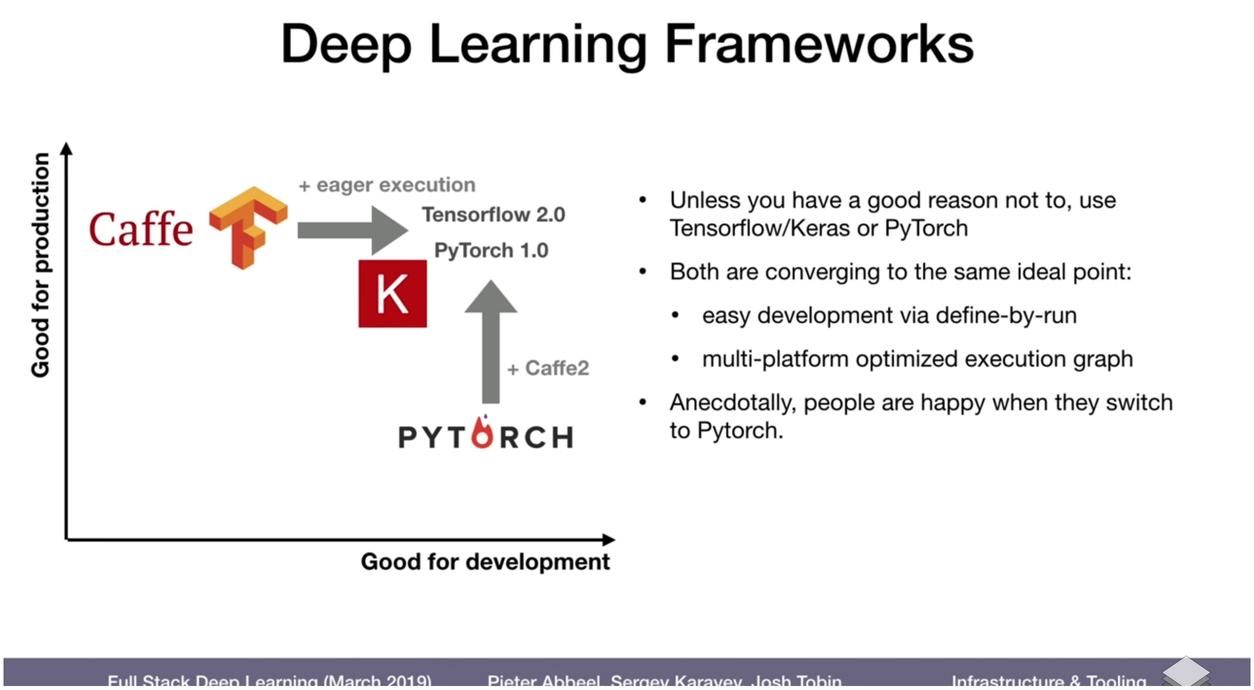
3. Install PyTorch and torchvision in your environment: `conda install pytorch torchvision -c pytorch` (https://pytorch.org/ (https://pytorch.org/) for more distributions, if you want to use CUDA for example)
4. Install Keras: `conda install -c conda-forge keras`
5. Installing your Virtual Env as a jupyter kernel:
   A. `pip install --user ipykernel`
   B. `python -m ipykernel install --user --name=nmep`
   C. In the menu bar of the jupyter notebook go to `Kernel > Change Kernel` and select `nmep`
6. To deactivate your environment: `conda deactivate`

Let's Get Started!

This homework is going to require a lot of reading to truly understand what pytorch is and how it works. Please put in the time right now to fully understand what is going on because it will only help you in the future! It is also super important that you learn how to use pytorch for next weeks homework (which will be very difficult).

So.... What is PyTorch?

PyTorch is an open-source ML framework created by Facebook that makes it super easy to build and train models in python. It also provides functionality for distributed training and data loading (among other things) to make training your models as efficient as possible. There are also other libraries including TensorFlow and Keras that you will likely encounter as you read and write ML code.



Intro to PyTorch

BEFORE YOU GET STARTED please read the following excellent articles on how PyTorch works. This is one of the best articles on PyTorch I've read so go through it THOROUGHLY:

1. https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html (https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)

Lets build our first Neural Network

In [1]:
```
#Import the necessary libraries
import os
#Disable the warnings for now cuz they are annoying
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import torch
import torch.nn as nn
```

Build our PyTorch system

Essentially, PyTorch training involves the following parts:

1. **Module**: main model which will be trained (e.g. class neural_network(nn.Module):)
   1a. __ **init** __: define all layers as attributes of the Module so parameters can be saved
   1b. **forward**: a mathematical function that uses layers define in __init__ to calculate output
2. **Loss Measure**: guide for optimization of model parameters
3. **Optimization Method**: update method for tuning model parameters

These are all described in more detail in the above article. If you are confused about any of these components, I would highly recommend taking a look at it.

Source: https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py (https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py)

Model:

1. The first layer is given to you. It creates a hidden layer layer of size [input, 256] (256 neurons in the first hidden layer) and a bias variable for each neuron. The weights will be randomly intialized automatically.
2. TODO: Add a ReLU activation after the first layer.
3. TODO: Repeat a similar process for another layer with 256 neurons
4. TODO: Repeat a similar process for the output layer. How many neurons should this layer have? (hint: don't use ReLU activation after the last layer)
5. TODO: Implement the forward method, in which you call all the layers defined in __init__().
6. Initialize model

In [7]:
```
#MODEL --> This will be a shallow network because we don't want our laptops

class Model(nn.Module):

    def __init__(self):
        super(Model, self).__init__()
        self.linear1 = nn.Linear(784, 256)
        self.relu1 = nn.ReLU()
        self.linear2 = nn.Linear(256, 256)
        self.relu2 = nn.ReLU()
        self.linear3 = nn.Linear(256, 10)

    def forward(self, input):
        x = self.linear1(input)
        x = self.relu1(x)
        x = self.linear2(x)
        x = self.relu2(x)
        x = self.linear3(x)
        return x

model = Model()
```

Question: Why do we need to define the learnable layers as attributes of the Module, and not just define and call them in forward? *hint: something something parameters*
Answer: We need the layers to retain their state after calling forward multiple times. Thus, they need to be parameters of the neural network as a whole, and cannot be defined and called in the same function.

Loss Measure:

1. TODO: Define the loss function to be minimized during the training loop

In [8]:
```
#LOSS MEASURE
criterion = nn.CrossEntropyLoss()
```

Optimization Method:

This part defines an optimizer and learning rate and creates an operation in the graph to minimize the loss that we defined earlier. This is where the learning (gradient computation and weight updates) happen. We used the Adam optimizer, but feel free to experiment.

In [9]:
```
#OPTIMIZATION METHOD
optimizer = torch.optim.Adam(model.parameters())
```

Loading the Data

Since our dataset is small enough we can load all the data into memory; however, in reality if you are working with a large dataset/large images, you will have to batch your data and read each batch from disk during every training loop...more on this later.

PyTorch provides a convenient interface for many datasets through torchvision (CV tools), including MNIST. This makes it really easy to test your code on a dataset that is commonly used. The code below shows you how to create a Dataset and Dataloader with MNIST data.

Datasets control the data, and custom Datasets are very easy to create: you just override the __len__() and __getitem__() methods (not necessary here, just good to know). Many data sets are too large to fit into your computer's memory, so you can use a Dataset to only load more data into memory when the DataLoader calls for it. DataLoaders take in a dataset as an argument and allow you to iterate through batches of the Dataset, only retrieving data when necessary.

Note: transforms.ToTensor() is necessary since by default the MNIST data is stored as PIL images

In [5]:
```
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

MNIST_train = MNIST("../data/mnist", train=True, download=True, transform =
MNIST_test = MNIST("../data/mnist", train=False, download=True, transform =
train_dataloader = DataLoader(MNIST_train, batch_size=256, shuffle=True)
test_dataloader = DataLoader(MNIST_test, batch_size=len(MNIST_test), shuffl
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
 (http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz) to ../dat
a/mnist/MNIST/raw/train-images-idx3-ubyte.gz
```

9920512/? [05:10<00:00, 33703.84it/s]

```
Extracting ../data/mnist/MNIST/raw/train-images-idx3-ubyte.gz to ../data/
mnist/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
 (http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz) to ../dat
a/mnist/MNIST/raw/train-labels-idx1-ubyte.gz
```

0% 0/28881 [00:00<?, ?it/s]

```
Extracting ../data/mnist/MNIST/raw/train-labels-idx1-ubyte.gz to ../data/
mnist/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz (h
ttp://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz) to ../data/mni
st/MNIST/raw/t10k-images-idx3-ubyte.gz
```

1654784/? [01:05<00:00, 27595.96it/s]

```
Extracting ../data/mnist/MNIST/raw/t10k-images-idx3-ubyte.gz to ../data/m
nist/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz (h
ttp://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz) to ../data/mni
st/MNIST/raw/t10k-labels-idx1-ubyte.gz
```

0% 0/4542 [00:00<?, ?it/s]

```
Extracting ../data/mnist/MNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/m
nist/MNIST/raw
Processing...
Done!
```

Implementing the Training Loop

In [10]:
```
epochs = 10

for epoch in range(epochs):
    total_loss = 0

    for batch in train_dataloader:
        X_batch, y_batch = batch[0].view(-1, 784), batch[1]

        # THESE 5 LINES ARE IMPORTANT, UNDERSTAND WHAT IS HAPPENING HERE
        optimizer.zero_grad()
        predicted_batch = model(X_batch)
        loss = criterion(predicted_batch, y_batch)
        loss.backward()
        optimizer.step()

        total_loss += loss

    print("Epoch {0}: {1}".format(epoch, total_loss))
    if epoch%5 == 0 and epoch!= 0:
        test_batch = next(iter(test_dataloader))
        X_test, y_test = test_batch[0].view(-1, 784), test_batch[1]
        predicted = model(X_test)
        test_acc = torch.sum(y_test == torch.argmax(predicted, dim=1), dtyp
        print("\tTest Accuracy {0}".format(test_acc))
```

```
Epoch 0: 97.85601806640625
Epoch 1: 38.419776916503906
Epoch 2: 25.197256088256836
Epoch 3: 18.89733123779297
Epoch 4: 14.321187973022461
Epoch 5: 11.475135803222656
        Test Accuracy 0.9765
Epoch 6: 8.777388572692871
Epoch 7: 6.809278964996338
Epoch 8: 5.358859062194824
Epoch 9: 4.310754299163818
```

Evaluating the Model

In [12]:
```
#CALCULATE TEST ACCURACY

test_batch = next(iter(test_dataloader))
X_test, y_test = test_batch[0].view(-1, 784), test_batch[1]
predicted = model(X_test)
test_accuracy = torch.sum(y_test == torch.argmax(predicted, dim=1), dtype=t
print ("Test Accuracy {0}".format(test_accuracy))
```

```
Test Accuracy 0.9782
```

Save and restore models

When we are training, it is very important that we periodically checkpoint our model (save the weights to disk). In this case we will only be saving the weights after we finish training; however, in practice you should be doing it after each training epoch (depending on how long training takes).

Reading: [https://pytorch.org/tutorials/beginner/saving_loading_models.html (https://pytorch.org/tutorials/beginner/saving_loading_models.html)](https://pytorch.org/tutorials/beginner/saving_loading_models.html)

In [14]:
```
#SAVE YOUR MODEL STATE_DICT (weights)
torch.save(model.state_dict(), "./weights")
```

In [15]:
```
#INITIALIZE MODEL ARCHITECTURE
loaded_model = Model()

#RESTORE THE WEIGHTS FROM THE LATEST CHECKPOINT (use model_name.load_state_
loaded_model.load_state_dict(torch.load("./weights"))

#YOU SHOULD SEE A LIST OF YOUR ENTIRE MODEL PRINTED HERE
print(loaded_model)
```

```
Model(
  (linear1): Linear(in_features=784, out_features=256, bias=True)
  (relu1): ReLU()
  (linear2): Linear(in_features=256, out_features=256, bias=True)
  (relu2): ReLU()
  (linear3): Linear(in_features=256, out_features=10, bias=True)
)
```

Congrats!! You (hopefully) now know the basics of how to use PyTorch. Of course, the best way to learn is to practice! In the next homework you will be implementing an autoencoder using PyTorch! Wooooo!