COMPSCI 389
Introduction to Machine Learning

Days: Tu/Th. **Time:** 2:30 – 3:45 **Building:** Morrill 2 **Room:** 222

Topic 10.4: Introduction to PyTorch

Prof. Philip S. Thomas (pthomas@cs.umass.edu)

Note: This presentation covers (and provides additional context/information regarding) `10.5 Introduction to PyTorch.ipynb`

Autograd

- Can be slow because it executes Python code.
- Is designed for differentiating arbitrary code
  - It does not have extra functionality for machine learning

Deep Learning Libraries

- There are many deep learning libraries that extend autograd to:
  - Leverage low-level compiled code for faster runtimes.
  - Enable forward and backwards passes on the GPU rather than CPU (more on this later).
  - Have built-in implementations of
    - Common loss functions
    - Common activation functions
    - Common network layers
      - Fully connected feed-forward
      - Convolutional layers
      - Pooling layers
      - Etc.

Deep Learning Libraries

- PyTorch
  - The most commonly used today.
  - What we will use in class.
- Tensorflow
  - Produced and maintained by Google
  - Integrates nicely with Google’s cloud computing platforms
  - Steeper learning curve and more verbose syntax
- Keras, Caffe, MXNet, etc.
  - Many less popular alternatives

PyTorch

You can install PyTorch with:

```
pip install torch torchvision
```

We will use the following imports:

```python
# New to this topic:
import torch
import torch.nn as nn                  # For defining our neural network model
import torch.optim as optim            # For training the model using data
from torch.utils.data import TensorDataset, DataLoader  # For making mini-batches
```

Defining a Neural Network Architecture
Defining a Parametric Model

- Extend the `nn.Module` base class
  - The base class provides functionality for tracking trainable parameters (and their gradients), moving parameters to the GPU, saving and loading models, etc.
- Implement two functions:
  - `__init__(self)`: Define the different layers (number of units, number of inputs) and different activation functions that will be used.
  - `forward(self, x)`: Perform a forward pass on input x.
- You do *not* need to implement any gradients or the backwards pass!
  - PyTorch uses reverse mode automatic differentiation to automatically compute gradients.

Note: This model is bigger than needed for the GPA prediction problem. This allows us to more easily compare runtimes later, and to show a phenomenon called "overfitting".

```
class FullyConnectedNetwork(nn.Module):
    def __init__(self):
        # First call the nn.Module constructor to initialize other parts of the model. Always do this first.
        super(FullyConnectedNetwork, self).__init__()

        # Define layers. The lines below create the layers (memory is allocated for the weights here).
        self.fc1 = nn.Linear(9, 1024)  # First hidden layer with 1024 neurons and 9 inputs.
        self.fc2 = nn.Linear(1024, 512) # Second hidden layer with 512 neurons and 1024 inputs.
        self.fc3 = nn.Linear(512, 128)  # Third hidden layer with 128 neurons and 512 inputs.
        self.fc4 = nn.Linear(128, 1)   # Output layer with 1 neuron and 128 inputs.

        # Define activation function.
        self.relu = nn.ReLU()

    def forward(self, x):
        # Pass data through the network
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)                # No activation after the output layer
        return x
```

`nn.Linear` represents a linear parametric model with no basis. That is, a perceptron without an activation function.

We can now create an instance of this model:

```
# Create an instance of the network
net = FullyConnectedNetwork()

# The network structure is printed as a sanity check
print(net)
```

```
FullyConnectedNetwork(
  (fc1): Linear(in_features=9, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=1, bias=True)
  (relu): ReLU()
)
```

`bias=True` indicates that each perceptron includes an extra feature that is always equal to 1 (and hence one extra weight beyond the number of outputs from the previous layer). This is what we discussed previously when we talked about appending a 1 to the columns of a data set to implement the “y-intercept” in linear regression. For perceptrons and neural networks, this extra weight is called the **bias**.

Next, let's load the GPA data, split it into training and testing, and standardize it.

+ Code + Markdown

```
df = pd.read_csv("https://people.cs.umass.edu/~pthomas/courses/COMPSCI_389_Spring2024/GPA.csv", delimiter=',')
#df = pd.read_csv("data/GPA.csv", delimiter=',')

# We already loaded X and y, but do it again as a reminder
X = df.iloc[:, :-1]
y = df.iloc[:, -1]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=True)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train) # This sets the min/max values from the training data (without looking
X_test = scaler.transform(X_test)       # This uses the min/max scaling values chosen during training! (transfo
```

Python

PyTorch has its own objects for storing data, called PyTorch tensors. These are simply multidimensional arrays. Let's convert our data to these tensor objects. Note that the `tensor` constructor is not compatible with `pandas.Series` objects, so we call `y_train.values` and `y_test.values` to convert these to `numpy.ndarray` objects.

```python
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.float32).view(-1,1)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.float32).view(-1,1)
```

Python

Loss Function

- PyTorch has many built-in loss functions, including MSE:

```
loss_function = nn.MSELoss()
```

Optimizer

- PyTorch has many built-in loss optimizers, including gradient descent (SGD), and Adam (SGD with a specific adaptive step size method).
  - Several optimizers are discussed in the Jupyter notebook.
  - Adam is the most common, and what we will use.

```
optimizer = optim.Adam(net.parameters())
```

`net.parameters()` contains the weights, and after backwards passes will also contain the gradient information. The optimizer uses this gradient information to update the weights.