Intro to Pytorch

Data Science Club

Agenda

- Introduction to PyTorch
- Introduction to Tensors / PyTorch basics
- How PyTorch supports a ML Workflow
- Extensions of NN’s
- Extra Tools in PyTorch

Introduction to PyTorch

- PyTorch is an Open-source deep learning framework
- Developed by Facebook’s AI Research lab
- Strengths of PyTorch
  - Pythonic syntax
  - Ability to build both simple and complex models
  - Large and active community
  - Supports Automatic Differentiation for Neural networks

PyTorch v.s other Frameworks

- Comparison with TensorFlow:
  - TensorFlow uses static computation graphs (older versions)
  - PyTorch uses dynamic computation graphs (easier for debugging)
  - Both support large-scale machine learning deployments
- Why PyTorch?
  - More flexibility (especially for R&D and experimentation)
  - Natural integration with Python libraries (e.g., NumPy)
  - Simple, readable syntax

PyTorch Fundamentals: Tensors

What are Tensors?

- Multi-dimensional arrays (similar to NumPy arrays)
- Generalization of vectors and matrices
- Used to represent input data and parameters in deep learning

```
import torch
x = torch.tensor([1, 2, 3])
print(x)
```

```
tensor([1, 2, 3])
```

PyTorch Tensor vs NumPy Array:

- PyTorch tensors support GPU acceleration
- PyTorch tensors come with autograd support
- Can easily convert between NumPy arrays and PyTorch tensors

Tensor Operations

- Element-wise Operations:
  - Operations like addition, subtraction, multiplication, and division are performed element-wise between tensors of the same shape.
- Matrix Multiplication:
  - PyTorch supports matrix multiplication using either the @ operator or torch.matmul() function.
- Reshaping Tensors:
  - You can change the shape of tensors using .view() or .reshape(), which is especially useful when preparing data for neural networks.
- Slicing and Indexing:
  - Similar to NumPy, tensors in PyTorch can be sliced and indexed.

Tensor Operations: Sample code

```
import torch

x = torch.tensor([1.0, 2.0, 3.0])
y = torch.rand(1, 3)  # Random tensor with values sampled from [0, 1)

print("X: ", x)
print("Y: ", y)

z = torch.zeros(3, 3)  # 3x3 matrix of zeros
print("Z: ", z)

result = torch.add(x, y)  # x + y
print("Result of add: ", result)
result = torch.mul(x, y)  # x * y
print("Result of mul: ", result)

x = torch.randn(2, 3)
print("ran X: ", x)
x_reshaped = x.view(3, 2)  # Reshaping the tensor
print("reshaped tensor", x_reshaped)

x[:, 1]  # All rows, second column
print("sliced tensor",x[:, 1] )
```

```
X:  tensor([1., 2., 3.])
Y:  tensor([[0.6740, 0.8895, 0.6841]])
Z:  tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
Result of add:  tensor([[1.6740, 2.8895, 3.6841]])
Result of mul:  tensor([[0.6740, 1.7790, 2.0523]])
ran X:  tensor([[-2.1624, -0.5272,  0.9708],
        [-0.3355, -1.9378, -0.9678]])
reshaped tensor tensor([[-2.1624, -0.5272],
        [ 0.9708, -0.3355],
        [-1.9378, -0.9678]])
sliced tensor tensor([-0.5272, -1.9378])
```

Data Loading with PyTorch

Efficient Data Loading:

- Neural networks require training on large datasets. PyTorch provides the `torch.utils.data.DataLoader` class to simplify data loading and manage mini-batches during training.

`Dataset` and `DataLoader`:

- **`torch.utils.data.Dataset`**: Represents your data and how to access it. You can create a custom dataset by subclassing this class and overriding `__len__` and `__getitem__` methods.
- **`torch.utils.data.DataLoader`**: Handles batching, shuffling, and loading data in parallel using workers (multiprocessing).

Batch Size and Shuffling:

- **Batch Size:** Controls the number of samples per batch passed to the model.
- **Shuffling:** Randomizes the order of data samples, which helps to prevent the model from learning data order instead of patterns.

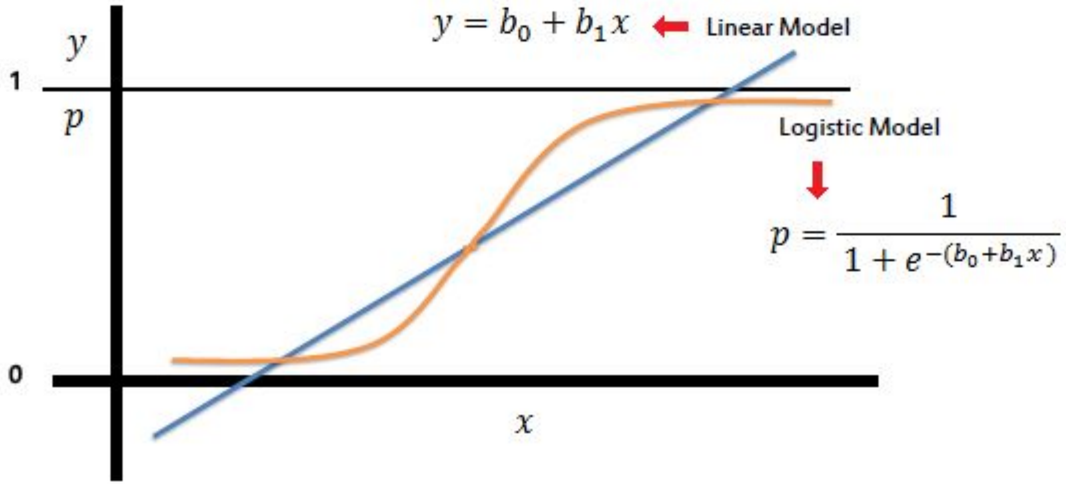
```python
from torch.utils.data import DataLoader

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
for batch in train_loader:
    inputs, labels = batch
```

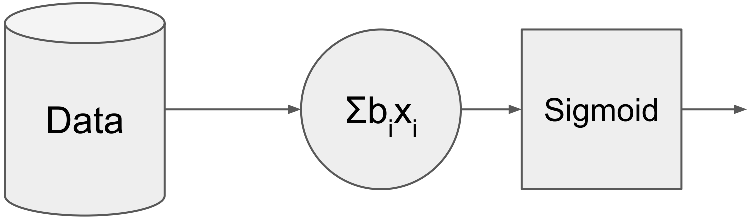
PyTorch and Neural Networks

First steps:
Logistic Regression

Remember how logistic models work: They have a set of parameters (b0, b1) and output a probability corresponding to a certain data input (x).



Neural networks take this idea and extend it.

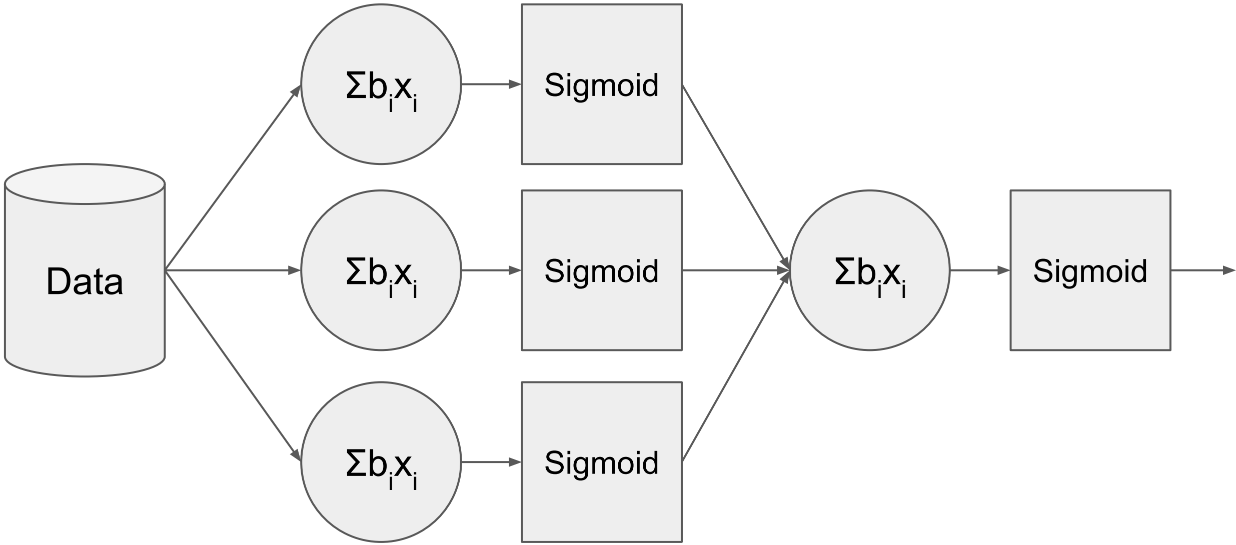


Next step:
A Neural Network

We can instead have multiple logistic regressions in parallel that each have different values for b.

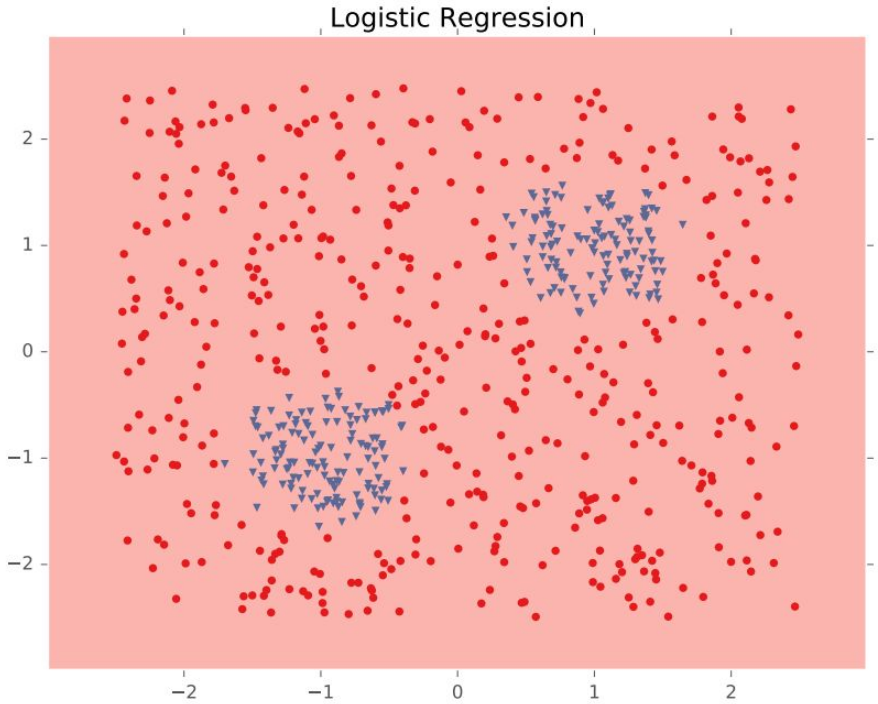
Now each one of these outputs can be used as the data input for a next logistic regression.

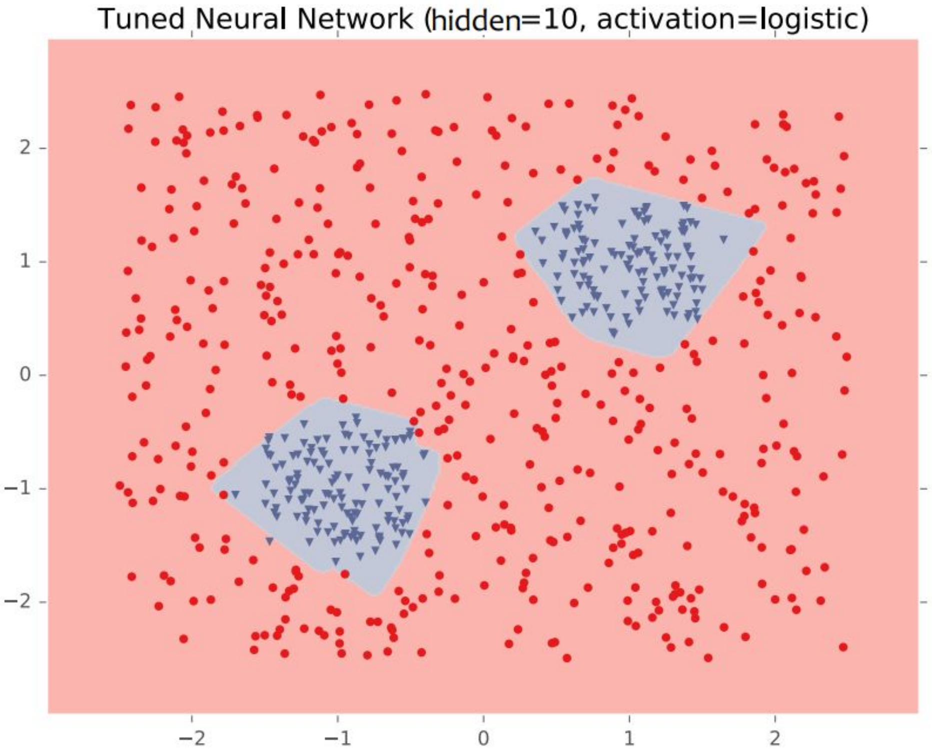
During the training process the model learns to diversify each set of b values in a way that gives more useful information.



1 layer with 3 neurons

Example of logistic regression v.s NN’s





Introduction to Neural Networks

Introduction to Neural Networks

- **What are Neural Networks?**
  - Neural networks are models inspired by the human brain that consist of interconnected layers of neurons.
  - Each neuron in a neural network performs a simple mathematical operation, and the layers are stacked to learn complex patterns.
  - In PyTorch, neural networks are typically represented as a sequence of layers (fully connected layers, convolutional layers, etc.) followed by activation functions (ReLU, sigmoid, etc.).
- **PyTorch's `torch.nn` Module:**
  - PyTorch provides the `torch.nn` module for building neural networks. This module includes layers like `Linear` for fully connected layers and functions like `ReLU` for non-linear activations.
  - Each neural network is defined as a class that subclasses `nn.Module` and implements a `forward

Building Neural Networks in PyTorch

Defining a Neural Network:

- In PyTorch, neural networks are typically defined as Python classes that inherit from `torch.nn.Module`. This class acts as a container for layers and implements the forward pass logic.
- The core structure involves two steps:
  1. **Initializing Layers:** In the `__init__()` method, define the layers (e.g., fully connected layers, convolutions) that will make up your network.
  2. **Forward Method:** In the `forward()` method, define how data moves through the layers in your model. This is where the layers you've defined are applied to the input data, layer by layer.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 128)  # Fully connected layer (input: 784, output: 128)
        self.fc2 = nn.Linear(128, 10)   # Fully connected layer (input: 128, output: 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))  # Apply ReLU activation after the first layer
        x = self.fc2(x)          # Output layer, no activation (usually combined with a loss function)
        return x
```

Training a Neural Network

Training Loop in PyTorch:

- PyTorch uses a simple, clear training loop that follows the steps:
    1. **Forward Pass:** Send the input through the model to get the predicted output.
    2. **Compute Loss:** Use a loss function to calculate the difference between the predicted output and the ground truth (target labels).
    3. **Zero Gradients:** Reset the gradients before the backward pass by calling `optimizer.zero_grad()`. Otherwise, gradients accumulate.
    4. **Backward Pass:** Call `loss.backward()` to compute the gradients of the loss with respect to each parameter.
    5. **Update Weights:** Use `optimizer.step()` to update the parameters using the computed gradients.

```python
for epoch in range(num_epochs):
    outputs = model(inputs)  # Forward pass
    loss = criterion(outputs, labels)  # Compute loss

    optimizer.zero_grad()  # Clear gradients
    loss.backward()  # Backward pass (compute gradients)
    optimizer.step()  # Update weights
```

Loss Function in PyTorch

What are Loss Functions?

- Loss functions measure how well a model's predictions match the true labels. They provide feedback to adjust model weights during training.
- PyTorch offers a variety of built-in loss functions in the `torch.nn` module.

Common Loss Functions:

- **Classification Tasks:**
  - Use `CrossEntropyLoss` for multi-class classification problems, which combines `LogSoftmax` and `NLLLoss`.
- **Regression Tasks:**
  - Use `MSELoss` (Mean Squared Error) for regression tasks where the target is a continuous variable.
- **Other Losses:**
  - **L1 Loss:** Measures the absolute differences between predicted and target values.
  - **Binary Cross-Entropy (BCE):** Commonly used for binary classification.
- **Choosing the Right Loss:**
  - Depending on your task, selecting the appropriate loss function is crucial to guide the learning process effectively.

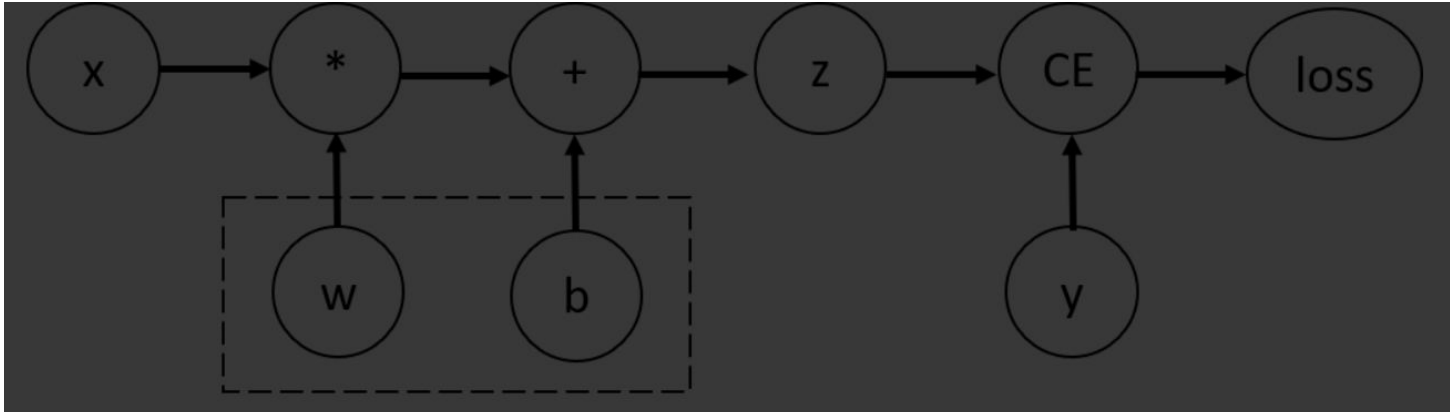
Backpropagation in PyTorch

Backpropagation Overview:

- Core to how neural networks learn
- Calculates gradients of the loss function with respect to model parameters

PyTorch's `backward()` Function:

- Computes the gradient of the loss function
- Automatically populates the `.grad` attribute for each tensor



Autograd Automatic Differentiation

What is Autograd?

- PyTorch's `autograd` module automatically computes gradients of tensor operations. These gradients are crucial for training neural networks because they indicate how to adjust model parameters to minimize the loss function.
- PyTorch tracks operations on tensors with `requires_grad=True` and constructs a computational graph. When the `.backward()` function is called, the framework traverses the graph to calculate gradients.

Why is this important?

- This automation is key to the backpropagation algorithm, which is fundamental in deep learning to adjust weights based on the loss.

```
import torch

x = torch.ones(2, 2, requires_grad=True)  # Track gradients
y = x + 2  # Perform operations
z = y * y * 3  # Further operations
z.backward()  # Compute gradients
print(x.grad)  # Gradients stored in x.grad
```

```
tensor([[18., 18.],
        [18., 18.]])
```

Optimizers in PyTorch

Role of Optimizers:

- After calculating gradients via backpropagation, an optimizer adjusts the model’s parameters to minimize the loss function.
- Optimizers use the computed gradients to update the model's weights. PyTorch provides several optimization algorithms in the `torch.optim` module.

Popular Optimizers:

- **SGD (Stochastic Gradient Descent):**
  - One of the simplest and most commonly used optimizers. SGD updates model parameters with learning rate and momentum:
- **Adam (Adaptive Moment Estimation)**
  - A popular choice because it adapts the learning rate for each parameter based on gradient moments. It often works well with minimal tuning.
- **RMSprop**:
  - A variant of SGD that scales the learning rate based on recent gradients, commonly used in RNNs
- **Learning Rate:**
  - The learning rate controls how big of a step the optimizer takes during each parameter update. It’s a crucial hyperparameter that needs tuning.

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Training on GPU with PyTorch

Why Use GPUs?

- GPUs are designed for fast matrix operations, making them significantly faster than CPUs for training deep neural networks. PyTorch supports GPU acceleration using CUDA.

Using CUDA in PyTorch:

- **GPU-accelerated Training:** All tensor operations performed on GPU are much faster than on CPU, especially when dealing with large datasets and models.

```
#Checking for GPU availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

#Moving model and data to GPU
model.to(device)
inputs, labels = inputs.to(device), labels.to(device)
```

Extensions of NN’s: Convolutional Neural Networks

CNNs are particularly well-suited for tasks involving:

- Image recognition
- Natural language processing
- Time series analysis

Imagine a sliding window.

- **Input:** A signal (like an image or audio).
- **Window:** A smaller, movable pattern (often called a kernel or filter).

Why is it useful?

- **Feature extraction:** CNNs use convolutions to learn and extract meaningful features from input data.
- **Pattern recognition:** By sliding the filter across the input, CNNs can detect patterns and structures.
- **Noise reduction:** Convolution can be used to smooth out noise in signals.

Transfer Learning in PyTorch

- **Why Save Models?**
  - Saving a trained model allows you to reuse it without retraining, share it with others, or deploy it in production.

- **Saving Model Weights:**
  - PyTorch provides a simple way to save model parameters using the `torch.save()` function. This saves the model's state dictionary, which contains the weights and biases.

- **Loading Model Weights:**
  - To reload the saved weights, first create an instance of the model, then load the state dictionary.

- **Saving and Loading Entire Models:**
  - It's also possible to save the entire model, including its architecture and weights. This can be useful for quick deployment.

Transfer Learning in PyTorch: sample code

```
#Saving model weights
torch.save(model.state_dict(), 'model_weights.pth')

#Loading model weights
model = Net()
model.load_state_dict(torch.load('model_weights.pth'))

#Saving/Loading entire models
torch.save(model, 'model.pth')
model = torch.load('model.pth')
```

Link to Colab exercise

- Link: https://tinyurl.com/mry2zk9e
  - Be sure to make a copy of the notebook before you start!
- Slides: https://tinyurl.com/dscptworkshopslides

Free Perplexity for 1 year!!!

(scan QR code and create account

with cmu account)