PyTorch Basics

Ling 282/482: Deep Learning for Computational Linguistics
C.M. Downey
Fall 2025

PyTorch

- Most popular **library for Deep Learning**
- Handles **Backpropagation** for built-in and user-defined models
- Libraries like **Huggingface** are built **on top** of PyTorch
  - (Model definitions in Huggingface are written in **PyTorch syntax**)
- Handles **GPU integration**

torch.Tensor

- A tensor is an **abstraction** on vectors and matrices
  - Instead of just rows and columns, can have **many "dimensions"**
- PyTorch Tensors work **a lot like Numpy arrays**
  - Essentially **Numpy with extra stuff on top**
- Shown: defining a vector, matrix, and 3D Tensor

```
>>> import torch
>>>
>>> x = torch.Tensor([1.0, 2.0, 3.0])
>>>
>>> x
tensor([1., 2., 3.])
>>>
>>> A = torch.Tensor([[1.0, 2.0, 3.0],[4.0, 5.0, 6.0]])
>>>
>>> A
tensor([[1., 2., 3.],
        [4., 5., 6.]])
>>> A.size()
torch.Size([2, 3])
>>>
>>> B = torch.Tensor([[[1.0, 2.0],[3.0, 4.0]],[[4.0, 3.0],[2.0, 1.0]]])
>>> B
tensor([[[1., 2.],
         [3., 4.]],

        [[4., 3.],
         [2., 1.]]])
>>> B.size()
torch.Size([2, 2, 2])
>>>
```

Using a Module

- Once defined, a Module class can be **instantiated** as a Python object
- Module can be **called** on some input x with my_module(x)
  - This runs the **forward function**
- Module.parameters() returns the **trainable model parameters**
- Notice the output tensor has a **gradient function** attached
  - This is used in the **backward pass**

```
>>> my_ffnn = feedforward(4, 2)
>>>
>>> my_ffnn
feedforward(
  (linear): Linear(in_features=4, out_features=2, bias=True)
  (sigmoid): Sigmoid()
)
>>> [p for p in my_ffnn.parameters()]
[Parameter containing:
tensor([[-0.3182, -0.1286, -0.0936, -0.0154],
        [-0.4639,  0.1249,  0.2100, -0.2978]], requires_grad=True),
tensor([-0.0233, -0.3502], requires_grad=True)]
>>>
>>> x = torch.Tensor([1.0, 2.0, 3.0, 4.0])
>>>
>>> y = my_ffnn(x)
>>>
>>> y
tensor([0.2806, 0.2450], grad_fn=<SigmoidBackward0>)
>>>
```

Training Loop

- This is the **general PyTorch training loop**
- \`net\` is a Module defining the **model we want to train**
  - We **call** the model on the inputs
- The **loss function** is usually **also a Module**
  - PyTorch defines most common loss functions
- loss.backward() runs the **Backpropagation algorithm** with respect to the loss value
  - **Just calculates the gradients**, doesn't update yet
- optimizer.step() **updates the parameters**
  - optimizer defines the **optimization algorithm** (e.g. Stochastic Gradient Descent, but others too)

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Word2Vec in PyTorch

- Here's an example of **Skip-Gram with Negative Sampling**, implemented as a PyTorch Module
- You'll **learn how this works** as part of Homework 2
- What are the **learnable parameters** in this model?
- Remember: PyTorch handles the **backward pass** for any Module we define
  - As long as **each operation defines a gradient** (see torch.sum, torch.mul)

```python
class SGNS(nn.Module):
    def __init__(self, vocab_size: int, embedding_dim: int):
        super().__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.context_embeddings = nn.Embedding(vocab_size, embedding_dim)

    def forward(self, batch: dict[str, Tensor]) -> Tensor:
        """TODO: your docstring summary here

        Args:
            TODO: document the arguments

        Returns:
            TODO: document what the function returns
        """
        target_embeddings_batch = self.embeddings(batch['target_ids'])
        context_embeddings_batch = self.context_embeddings(batch['context_ids'])
        similarities = torch.sum(
            torch.mul(target_embeddings_batch, context_embeddings_batch), dim=1
        )
        return similarities
```

```
>>> from word2vec import SGNS
>>>
>>> my_sgns = SGNS(vocab_size=4, embedding_dim=2)
>>>
>>> my_sgns
SGNS(
  (embeddings): Embedding(4, 2)
  (context_embeddings): Embedding(4, 2)
)
```