Lecture 06

Automatic Differentiation with PyTorch

STAT 479: Deep Learning, Spring 2019
Sebastian Raschka
http://stat.wisc.edu/~sraschka/teaching/stat479-ss2019/
PyTorch

FROM
RESEARCH TO
PRODUCTION

An open source deep learning platform that provides a seamless path from research prototyping to production deployment.

https://pytorch.org/
Installation

Recommendation for Laptop (e.g., MacBook)

<table>
<thead>
<tr>
<th>PyTorch Build</th>
<th>Stable (1.0)</th>
<th>Preview (Nightly)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your OS</td>
<td>Linux</td>
<td>Mac</td>
</tr>
<tr>
<td>Package</td>
<td>Conda</td>
<td>Pip</td>
</tr>
<tr>
<td>Language</td>
<td>Python 2.7</td>
<td>Python 3.5</td>
</tr>
<tr>
<td>CUDA</td>
<td>8.0</td>
<td>9.0</td>
</tr>
<tr>
<td>Run this Command</td>
<td>conda install pytorch torchvision &lt; c pytorch</td>
<td></td>
</tr>
</tbody>
</table>

Recommendation for Desktop (Linux) with GPU

<table>
<thead>
<tr>
<th>PyTorch Build</th>
<th>Stable (1.0)</th>
<th>Preview (Nightly)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your OS</td>
<td>Linux</td>
<td>Mac</td>
</tr>
<tr>
<td>Package</td>
<td>Conda</td>
<td>Pip</td>
</tr>
<tr>
<td>Language</td>
<td>Python 2.7</td>
<td>Python 3.5</td>
</tr>
<tr>
<td>CUDA</td>
<td>8.0</td>
<td>9.0</td>
</tr>
<tr>
<td>Run this Command</td>
<td>conda install pytorch torchvision cudatoolkit=10.0 &lt; c pytorch</td>
<td></td>
</tr>
</tbody>
</table>

https://pytorch.org/

Installation Tips:

https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/other/pytorch-installation-tips.md

And don't forget that you import PyTorch as "import torch," not "import pytorch" :)
Many Useful Tutorials (recommend that you read some of them)

https://pytorch.org/resources
Very Active & Friendly Community and Help/Discussion Forum

https://pytorch.org/resources
DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ

Author: Soumith Chintala

Goal of this tutorial:
- Understand PyTorch's Tensor library and neural networks at a high level.
- Train a small neural network to classify images

This tutorial assumes that you have a basic familiarity of numpy

**NOTE**

Make sure you have the torch and torchvision packages installed.

What is PyTorch?  |  Autograd: Automatic Differentiation  |  Neural Networks

Generally speaking, `torch.autograd` is an engine for computing vector-Jacobian product. That is, given any vector \( \mathbf{v} = \begin{pmatrix} v_1 & v_2 & \cdots & v_m \end{pmatrix}^T \), compute the product \( \mathbf{v}^T \cdot J \). If \( \mathbf{v} \) happens to be the gradient of a scalar function \( l = g(\mathbf{y}) \), that is, \( \mathbf{v} = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}^T \), then by the chain rule, the vector-Jacobian product would be the gradient of \( l \) with respect to \( \mathbf{x} \):

\[
J^T \cdot \mathbf{v} = \begin{pmatrix} \\
\frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\
\vdots & \ddots & \vdots \\
\frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n}
\end{pmatrix} \begin{pmatrix} \\
\frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m}
\end{pmatrix} = \begin{pmatrix} \\
\frac{\partial l}{\partial x_1} \\
\vdots \\
\frac{\partial l}{\partial x_n}
\end{pmatrix}
\]

Text source: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py
In the context of deep learning (and PyTorch) it is helpful to think about neural networks as computation graphs.
Computation Graphs

Suppose we have the following activation function:

\[ a(x, w, b) = relu(w \cdot x + b) \]

ReLU = Rectified Linear Unit
(prob. the most commonly used activation function in DL)
Side-note about ReLU Function

You may note that

\[ f'(x) = \begin{cases} 
0 & \text{if } x < 0 \\
x & \text{if } x > 0 \\
\text{DNE} & \text{if } x = 0 
\end{cases} \]

But in the computer science context, for convenience, we can just say

\[ f'(x) = \begin{cases} 
0 & \text{if } x \leq 0 \\
x & \text{if } x > 0 
\end{cases} \]

\[ f'(x) = \lim_{x \to 0} \frac{\max(0, x + \Delta x) - \max(0, x)}{\Delta x} \]

\[ f'(x) = \lim_{x \to 0} \frac{\max(0, x + \Delta x) - \max(0, x)}{\Delta x} \]

\[ f'(0) = \lim_{x \to 0^+} \frac{0 + \Delta x - 0}{\Delta x} = 1 \]

\[ f'(0) = \lim_{x \to 0^-} \frac{0 - 0}{\Delta x} = 0 \]
Computation Graphs

Suppose we have the following activation function:

$$a(x, w, b) = \text{relu}(w \cdot x + b)$$

multivariable function

activation function

weight parameter

bias

feature

(suppose only 1 training example)

(weight parameter)

(assume only 1 input feature)

Suppose we have the following activation function:

$$\text{relu}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Graph:

relu($x$) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}
Computation Graphs

\[ a(x, w, b) = \text{relu}(w \cdot x + b) \]

\[ u = wx \]

\[ v = u + b \]

\[ a = \text{relu}(v) \]
Computation Graphs

\[ a(x, w, b) = \text{relu}(w \cdot x + b) \]

\[ u = wx \]

\[ v = u + b \]

\[ a = \text{relu}(v) \]
Computation Graphs

\[ u = wx \]

\[ b = 1 \]

\[ v = u + b \]

\[ a = \text{relu}(v) \]

\[ \frac{da}{dv} \]
Computation Graphs

\[ u = wx \]
\[ b = 1 \]
\[ v = u + b \]
\[ a = \text{relu}(v) \]

\[ \frac{da}{dv} \]
Computation Graphs

\[ u = wx \]

\[ b = 1 \]

\[ v = u + b \]

\[ a = \text{relu}(v) \]

\[ \frac{\partial v}{\partial b} \]

\[ \frac{da}{dv} \]
\[ u = wx \]
\[ b = 1 \]
\[ v = u + b \]
\[ a = \text{relu}(v) \]

**Computation Graphs**

\[ \frac{\partial a}{\partial b} = ? \]

\[ \frac{\partial v}{\partial b} \]

\[ \frac{\partial v}{\partial u} \]

\[ \frac{\partial u}{\partial w} \]
\[
\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v}
\]

Computation Graphs
Computation Graphs

\[ \frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v} \]

\[ \frac{\partial v}{\partial b} \]

\[ \frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \]

\[ = \frac{\partial u}{\partial w} \frac{\partial v}{\partial u} \frac{\partial a}{\partial v} \]

b = 1

x = 3

w = 2

u = wx

v = u + b

a = relu(v)
Computation Graphs

\[
\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v}
\]

\[
\frac{\partial v}{\partial b}
\]

\[
\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u}
\]

\[
= \frac{\partial u}{\partial w} \frac{\partial v}{\partial u} \frac{\partial a}{\partial v}
\]

\[
\frac{\partial v}{\partial u}
\]

\[
relu(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  0 & \text{otherwise}
\end{cases}
\]
Computation Graphs

\[ \frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v} \]

\[ \frac{\partial v}{\partial b} \]

\[ \frac{da}{dv} = 1 \]

\[ a = \text{relu}(v) \]

\[ \text{relu}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \]

\[ \frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \]

\[ = \frac{\partial u}{\partial w} \frac{\partial v}{\partial u} \frac{\partial a}{\partial v} \]

\[ \frac{\partial v}{\partial u} \]

\[ + \]

\[ v = u + b \]

\[ u = wx \]

\[ x = 3 \]

\[ w = 2 \]

\[ b = 1 \]
Computation Graphs

\[ \frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v} \]

\[ \frac{\partial v}{\partial b} = ? \]

\[ \frac{da}{dv} = 1 \]

\[ \frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \]

\[ \frac{\partial u}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \]

\[ \frac{\partial v}{\partial u} = ? \]

<table>
<thead>
<tr>
<th>Function</th>
<th>Derivative</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) + g(x) )</td>
<td>( f'(x) + g'(x) )</td>
</tr>
</tbody>
</table>
Computation Graphs

\[ \frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v} \]

\[ \frac{\partial v}{\partial b} = 1 \]

\[ \frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \]

\[ = \frac{\partial u}{\partial w} \frac{\partial v}{\partial u} \frac{\partial a}{\partial v} \]
Computation Graphs

\[ \frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial b} = 1 \]

\[ \frac{\partial a}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial v} = 1 \]

\[ \frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} = 3 \]

\[ \frac{\partial u}{\partial w} = 3 \]

\[ \frac{\partial v}{\partial u} = 1 \]

\[ \frac{\partial u}{\partial v} = 3 \]

\[ \frac{\partial a}{\partial v} = 3 \]

\[ \frac{\partial v}{\partial w} = 1 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]

\[ \frac{\partial v}{\partial u} = 3 \]

\[ \frac{\partial a}{\partial w} = 3 \]

\[ \frac{\partial u}{\partial v} = 1 \]
PyTorch Autograd Example

https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L06_pytorch/code/pytorch-autograd.ipynb
Gradients of intermediate variables
(usually not required in practice outside research)

https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L06_pytorch/code/grad-intermediate-var.ipynb
Some More Computation Graphs
Graph with Single Path

\[ w_1 \cdot x_1 = z_1 \]
\[ \sigma_1(z_1) = a_1 \]
\[ \frac{\partial l}{\partial o} \]
\[ \mathcal{L}(y, o) = l \]

\[ \frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} \]

(univariate chain rule)
Graph with Weight Sharing

\[
\sigma_1(z_1) = a_1
\]

\[
\frac{\partial a_1}{\partial w_1} \quad \frac{\partial o}{\partial a_1} \quad \frac{\partial l}{\partial o} \quad L(y, o) = l
\]

\[
\sigma_2(z_1) = a_2
\]

\[
\frac{\partial a_2}{\partial w_1} \quad \frac{\partial l}{\partial w_1} \quad \frac{\partial l}{\partial o} \quad \frac{\partial l}{\partial o} \quad \frac{\partial a_2}{\partial w_1}
\]

Upper path

\[
\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1}
\]

(multivariable chain rule)

Lower path
Graph with Fully-Connected Layers (later in this course)

\[ \frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} \]
class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_h1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_h1, num_h2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_h2, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
PyTorch Usage: Step 2 (Creation)

```python
torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features,
                            num_classes=num_classes)
model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
```

- Instantiate model (creates the model parameters)
- Define an optimization method
torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features,
                           num_classes=num_classes)
model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)

Optionally move model to GPU, where device e.g. torch.device('cuda:0')
PyTorch Usage: Step 3 (Training)

for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):
        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = F.cross_entropy(probas, targets)
        optimizer.zero_grad()

        cost.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()
        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy

        This will run the forward() method
        Define a loss function to optimize
        Set the gradient to zero (could be non-zero from a previous forward pass)
        Compute the gradients, the backward is automatically constructed by "autograd" based on the forward() method and the loss function
        Use the gradients to update the weights according to the optimization method (defined on the previous slide)
        E.g., for SGD, $w := w + \text{learning_rate} \times \text{gradient}$
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):
        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()
        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()
    model.eval()

    with torch.no_grad():
        # compute accuracy

For evaluation, set the model to eval mode (will be relevant later when we use Dropout or BatchNorm)

This prevents the computation graph for backpropagation from automatically being built in the background to save memory
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):
        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()
        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy

logits because of computational efficiency. Basically, it internally uses a log_softmax(logits) function that is more stable than log(softmax(logits)). More on logits ("net inputs" of the last layer) in the next lecture. Please also see

https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/other/pytorch-lossfunc-cheatsheet.md
PyTorch ADALINE (neuron model) Example

Objected-Oriented vs Functional* API

*Note that with "functional" I mean "functional programming" (one paradigm in CS)

```python
import torch.nn.functional as F

class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden_1)
        self.relu1 = torch.nn.ReLU()

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1, num_hidden_2)
        self.relu2 = torch.nn.ReLU()

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2, num_classes)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        out = self.linear_1(x)
        out = self.relu1(out)
        out = self.linear_2(out)
        out = self.relu2(out)
        logits = self.linear_out(out)
        probas = self.softmax(logits)
        return logits, probas
```

Unnecessary because these functions don't need to store a state but maybe helpful for keeping track of order of ops (when implementing "forward")
Objected-Oriented vs Functional API

Using "Sequential"

```
import torch.nn.functional as F

class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()
        self.my_network = torch.nn.Sequential(
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_2, num_classes)
        )
    def forward(self, x):
        logits = self.my_network(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas

class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()
        self.my_network = torch.nn.Sequential(
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_2, num_classes)
        )
    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

Much more compact and clear, but "forward" may be harder to debug if there are errors (we cannot simply add breakpoints or insert "print" statements
Objected-Oriented vs Functional API

Using "Sequential"

1) 
```python
class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()
        self.my_network = torch.nn.Sequential(
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

Much more compact and clear, but "forward" may be harder to debug if there are errors (we cannot simply add breakpoints or insert "print" statements

2) However, if you use Sequential, you can define "hooks" to get intermediate outputs. For example:
```python
model.net
```
```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU(inplace)
  (2): Linear(in_features=128, out_features=256, bias=True)
  (3): ReLU(inplace)
  (4): Linear(in_features=256, out_features=10, bias=True)
)
```
If we want to get the output from the 2nd layer during the forward pass, we can register a hook as follows:
```python
outputs = []
def hook(module, input, output):
    outputs.append(output)
model.net[2].register_forward_hook(hook)
```
```
<torch.utils.hooks.RemovableHandle at 0x7f659c6685c0>
```
Now, if we call the model on some inputs, it will save the intermediate results in the "outputs" list:
```python
outputs = [model(features)
    print(outputs)
```
```
[tensor([[0.5341, 1.0513, 2.3542, ..., 0.0000, 0.0000, 0.0000],
    [0.0000, 0.6676, 0.6620, ..., 0.0000, 0.0000, 2.4056],
    [1.1320, 0.0000, 0.0000, ..., 2.5860, 0.8992, 0.9642],
    ..., 
    [0.0000, 0.1876, 0.0000, ..., 1.3237, 0.0000, 2.5283],
    [0.5415, 0.0000, 0.0000, ..., 7.9038, 0.8244, 1.6335],
    [1.0710, 0.9085, 3.0103, ..., 0.0000, 0.0000, 0.0000]],
    device='cuda:3', grad_fn=<ThresholdBackward1>)]
```
More PyTorch features will be introduced step-by-step later in this course when we start working with more complex networks, including

- Running code on the GPU
- Using efficient data loaders
- Splitting networks across different GPUs
Reading Assignments

- **What is PyTorch**
  [https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py](https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py)

- **Autograd: Automatic Differentiation**