Lecture 06

Automatic Differentiation with PyTorch
Today

Computing partial derivatives more easily (and automatically) with PyTorch
Lecture Overview

1. PyTorch Resources
2. Computation Graphs
3. Automatic Differentiation in PyTorch
4. Training ADALINE Manually Vs Automatically in PyTorch
5. A Closer Look at the PyTorch API
Learning More About PyTorch

1. PyTorch Resources
2. Computation Graphs
3. Automatic Differentiation in PyTorch
4. Training ADALINE Manually Vs Automatically in PyTorch
5. A Closer Look at the PyTorch API
At a Glance:

- Based on Torch 7, which was based on Lua and inspired by Lush
- PyTorch started in 2016
- Focuses on flexibility and minimizing cognitive overhead
- Dynamic nature of autograd API inspired by Chainer
- Core features
  - Automatic differentiation
  - Dynamic computation graphs
  - NumPy integration
- written in C++ and CUDA (CUDA is like C++ for the GPU)
- Python is the usability glue
At a Glance:

- Based on Torch 7, which was based on Lua and inspired by Lush
- PyTorch started in 2016
- Focuses on flexibility and minimizing cognitive overhead
- Dynamic nature of autograd API inspired by Chainer
- Core features
  - Automatic differentiation
  - Dynamic computation graphs
  - NumPy integration
- written in C++ and CUDA (CUDA is like C++ for Python)
- Python is the usability glue

PyTorch vs NumPy

- Support GPU
- distribute ops across multiple devices
- keep track of computation graph and ops that created them
At a Glance:

- Based on Torch 7, which was based on Lua and inspired by Lush
- PyTorch started in 2016
- Focuses on flexibility and minimizing cognitive overhead
- Dynamic nature of autograd API inspired by Chainer
- Core features
  - Automatic differentiation

"the speedup gained by taking Python out of the computation is 10% or less"
-- Stevens et al.: Deep Learning with PyTorch

- written in C++ and CUDA (CUDA is like C++ for the GPU)
- Python is the usability glue
Installation

Recommendation for Laptop (e.g., MacBook)

Recommendation for Desktop (Linux) with GPU

https://pytorch.org/

As mention in the installation tips on Canvas

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/tutorials/
Many Useful Tutorials (recommend that you read some of them)

https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
Very Active & Friendly Community and Help/Discussion Forum

**PyTorch**

Do you want live notifications when people reply to your posts? Enable Notifications

<table>
<thead>
<tr>
<th>Topic</th>
<th>Replies</th>
<th>Views</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using MSELoss instead of CrossEntropy for Ordinal Regression/Classification</td>
<td>2</td>
<td>83</td>
<td>1h</td>
</tr>
<tr>
<td>Optimizer.load_state_dict() weird behaviour with Adam optimizer</td>
<td>7</td>
<td>2.0k</td>
<td>1h</td>
</tr>
<tr>
<td>Is there a way to train 3 dataloaders using multiprocessing?</td>
<td>0</td>
<td>11</td>
<td>2h</td>
</tr>
<tr>
<td>Getting different feature vectors from frozen layers after training</td>
<td>5</td>
<td>86</td>
<td>2h</td>
</tr>
<tr>
<td>Libtorch_CUDA.so is too large (&gt;2GB)</td>
<td>22</td>
<td>346</td>
<td>2h</td>
</tr>
<tr>
<td>Undo pruning - How to ‘unmask’ pruned weights</td>
<td>0</td>
<td>8</td>
<td>2h</td>
</tr>
<tr>
<td>If input.dim() == 2 and bias is not None: AttributeError: ‘tuple’ object has no attribute ‘dim’</td>
<td>3</td>
<td>41</td>
<td>2h</td>
</tr>
<tr>
<td>Export unsupported/compound ops to ONNX</td>
<td>0</td>
<td>9</td>
<td>2h</td>
</tr>
</tbody>
</table>

[https://discuss.pytorch.org](https://discuss.pytorch.org)
Understanding Automatic Differentiation via Computation Graphs

1. PyTorch Resources

2. Computation Graphs

3. Automatic Differentiation in PyTorch

4. Training ADALINE Manually Vs Automatically in PyTorch

5. A Closer Look at the PyTorch API
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:

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

**ReLU = Rectified Linear Unit**

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

You may note that

\[ \sigma'(z) = \begin{cases} 
  0 & \text{if } z < 0 \\
  1 & \text{if } z > 0 \\
  \text{DNE} & \text{if } z = 0 
\end{cases} \]

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

\[ \sigma'(z) = \begin{cases} 
  0 & \text{if } z \leq 0 \\
  1 & \text{if } z > 0 
\end{cases} \]

Why not differentiable?
Derivative does not exist (DNE) at 0, because the derivative is different if we approach the limit from the left or right:

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

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

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

Suppose we have the following activation function:

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

activation function

multivariable function

weight parameter

(suppose only 1 training example)

bias

net input

feature

(suppose only 1 input feature)

ReLU activation function:

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

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

\[ u = wx \]

\[ v = u + b \]

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

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

\[ v = u + b \]

\[ u = wx \]

\[ b = 1 \]

\[ x \]

\[ w = 2 \]
Computation Graphs

\[ u = wx \]

\[ b = 1 \]

\[ v = u + b \]

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

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

\frac{\partial v}{\partial b}

\frac{da}{dv}
Computation Graphs

\[
\begin{align*}
\frac{\partial a}{\partial b} &= 0 \quad ? \\
\frac{\partial v}{\partial b} &= 1 \\
\frac{da}{dv} &= 1 \\
\frac{da}{dv} &= 1 \\
\frac{\partial v}{\partial u} &= 1 \\
\frac{\partial a}{\partial v} &= 1 \\
\frac{\partial a}{\partial u} &= 1 \\
\frac{\partial u}{\partial w} &= 1 \\
\frac{\partial u}{\partial w} &= 1
\end{align*}
\]
\[
\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v}
\]

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

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

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

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

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

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

\[
\begin{align*}
\frac{\partial a}{\partial b} &= \frac{\partial v}{\partial b} \frac{\partial a}{\partial v} \\
\frac{\partial v}{\partial b} &= \\
\frac{\partial a}{\partial w} &= \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \\
&= \frac{\partial u}{\partial w} \frac{\partial v}{\partial u} \frac{\partial a}{\partial v} \\
&= \frac{\partial u}{\partial w} \frac{\partial v}{\partial u} \frac{\partial a}{\partial v}
\end{align*}
\]
Computation Graphs

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

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

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

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

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

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

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

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

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

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

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

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

<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

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

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

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

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

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

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

\[
\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \quad \frac{\partial u}{\partial w} = 3 \quad \frac{\partial u}{\partial v} = 1
\]

\[
\frac{\partial u}{\partial w} = \frac{\partial u}{\partial v} \frac{\partial v}{\partial a} = 3 \cdot 1 \cdot 1 = 3
\]
Some More Computation Graphs
Graph with Single Path

\[ \mathcal{L}(y, \sigma_1(w_1 \cdot x_1)) \]

\[ \frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} \] (univariate chain rule)
Graph with Weight Sharing

\[ \mathcal{L}(y, \sigma_3[\sigma_1(w_1 \cdot x_1), \sigma_2(w_1 \cdot x_1)]) \]

\[
\begin{align*}
\sigma_1(z_1) &= a_1 \\
\frac{\partial a_1}{\partial w_1} &\quad \frac{\partial o}{\partial a_1} \\
\sigma_2(z_1) &= a_2 \\
\frac{\partial a_2}{\partial w_1}
\end{align*}
\]

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)

\[
\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(1)}} \cdot \frac{\partial a_2^{(1)}}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}
\]

\( L(y, o) = l \)
Automatic Differentiation with PyTorch
-- An Autograd Example

1. PyTorch Resources
2. Computation Graphs
3. **Automatic Differentiation in PyTorch**
4. Training ADALINE Manually Vs Automatically in PyTorch
5. A Closer Look at the PyTorch API
PyTorch Autograd Example

https://github.com/rasbt/stat453-deep-learning-ss21/tree/master/L06/code/pytorch-autograd.ipynb
Training an Adaptive Linear Neuron in PyTorch

1. PyTorch Resources
2. Computation Graphs
3. Automatic Differentiation in PyTorch
4. Training ADALINE Manually Vs Automatically in PyTorch
5. A Closer Look at the PyTorch API
PyTorch ADALINE (neuron model) Example

Using PyTorch: A Closer Look at the Object-Oriented and Functional APIs

1. PyTorch Resources
2. Computation Graphs
3. Automatic Differentiation in PyTorch
4. Training ADALINE Manually Vs Automatically in PyTorch
5. A Closer Look at the PyTorch API
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

Backward will be inferred automatically if we use the nn.Module class!

Define model parameters that will be instantiated when created an object of this class

Define how and in what order the model parameters should be used in the forward pass
PyTorch Usage: Step 2 (Creation)

torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features,
    num_classes=num_classes)  # Instantiate model
    (creates the model parameters)

model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(),
    lr=learning_rate)  # Define an optimization method
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)
```

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

Run for a specified number of epochs

```python
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
```

Iterate over minibatches in epoch

If your model is on the GPU, data should also be on the GPU

Gradients at each leaf node are accumulated under the .grad attribute, not just stored. This is why we have to zero them before each backward pass.
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)  # This will run the forward() method
        loss = F.cross_entropy(logits, targets)  # Define a loss function to optimize
        optimizer.zero_grad()  # Set the gradient to zero (could be non-zero from a previous forward pass)

        loss.backward()  # Compute the gradients, the backward is automatically constructed by "autograd" based on the forward() method and the loss function

        ### UPDATE MODEL PARAMETERS
        optimizer.step()  # Use the gradients to update the weights according to the optimization method (defined on the previous slide)

        model.eval()
        with torch.no_grad():  # E.g., for SGD, \( w := w + \text{learning\_rate} \times \text{gradient} \)
            # 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

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 build 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 logsoftmax(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
Objected-Oriented vs Functional* API

*Note that with "functional" I mean "functional programming" (one paradigm in CS) torch.nn.functional = api without internal state

```
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 = F.log_softmax(logits, dim=1)
        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"

```python
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.log_softmax(logits, dim=1)
        return logits, probas

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

        self.linear_1 = torch.nn.Linear(num_features, num_hidden_1)
        self.linear_2 = torch.nn.Linear(num_hidden_1, num_hidden_2)
        self.linear_out = 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"

Using "Sequential"

```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

However, if you use Sequential, you can define "hooks" to get intermediate outputs.
For example:

```python
model.net
```

```python
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)
```

```python
<torch.nn.modules.module.Module at 0x7f659c6685c0>
```

Now, if we call the model on some inputs, it will save the intermediate results in the "outputs" list:

```python
__ = model(features)
print(outputs)
```

```python
[[0.5341, 1.0513, 2.3542, ..., 0.0000, 0.0000, 0.0000],
 [0.0000, 0.6676, 0.6620, ..., 0.0000, 0.0000, 2.4056],
 [1.1520, 0.0000, 0.0000, ..., 2.5860, 0.8992, 0.9642],
 ..., [0.0000, 0.1076, 0.0000, ..., 1.2367, 0.0000, 2.5283],
 [0.5415, 0.0000, 0.0000, ..., 2.7968, 0.8244, 1.6335],
 [1.0710, 0.9805, 3.0103, ..., 0.0000, 0.0000, 0.0000]],
device='cuda:3', grad_fn=<ThresholdBackward1>)]
```
https://github.com/IgorSusmelj/pytorch-styleguide

Jupyter Notebook vs Python Scripts

In general, we recommend to use jupyter notebooks for initial exploration/playing around with new models and code. Python scripts should be used as soon as you want to train the model on a bigger dataset where also reproducibility is more important.

Our recommended workflow:

1. Start with a jupyter notebook
2. Explore the data and models
3. Build your classes/methods inside cells of the notebook
4. Move your code to python scripts
5. Train/deploy on server

<table>
<thead>
<tr>
<th>Jupyter Notebook</th>
<th>Python Scripts</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Exploration</td>
<td>+ Running longer jobs without interruption</td>
</tr>
<tr>
<td>+ Debugging</td>
<td>+ Easy to track changes with git</td>
</tr>
<tr>
<td>- Can become a huge file</td>
<td>- Debugging mostly means rerunning the whole script</td>
</tr>
<tr>
<td>- Can be interrupted (don't use for long training)</td>
<td></td>
</tr>
<tr>
<td>- Prone to errors and become a mess</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Convention</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Packages &amp; Modules</td>
<td>lower_with_under</td>
</tr>
<tr>
<td>Classes</td>
<td>CapWords</td>
</tr>
<tr>
<td>Constants</td>
<td>CAPS_WITH_UNDER</td>
</tr>
<tr>
<td>Instances</td>
<td>lower_with_under</td>
</tr>
<tr>
<td>Methods &amp; Functions</td>
<td>lower_with_under()</td>
</tr>
<tr>
<td>Variables</td>
<td>lower_with_under</td>
</tr>
</tbody>
</table>
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