# PyTorch Installation

https://pytorch.org/

<table>
<thead>
<tr>
<th>PyTorch Build</th>
<th>Stable (1.7.0)</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</td>
<td>C++ / Java</td>
</tr>
<tr>
<td>CUDA</td>
<td>9.2</td>
<td>10.1</td>
</tr>
<tr>
<td>Run this Command:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conda install pytorch torchvision torchaudio cudatoolkit=11.0 -c pytorch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Previous versions of PyTorch ➔
Check if CUDA is available

```python
import torch
torch.cuda.is_available()
Out[55]: True
torch.cuda.current_device()
Out[56]: 0
torch.cuda.device(0)
Out[57]: <torch.cuda.device at 0x7f2b51842310>
torch.cuda.device_count()
Out[58]: 1
torch.cuda.get_device_name(0)
Out[59]: 'GeForce RTX 2080 with Max-Q Design'
```
Using GPU with pytorch

```python
a = torch.rand(4,3)
an
tensor([[0.0762, 0.0727, 0.4076],
        [0.1441, 0.2818, 0.7420],
        [0.7289, 0.9615, 0.6206],
        [0.7240, 0.0518, 0.3923]])

a.device
Out[101]: device(type='cpu')

device = torch.device('cuda')

a.to(device)
Out[103]:
tensor([[0.0762, 0.0727, 0.4076],
        [0.1441, 0.2818, 0.7420],
        [0.7289, 0.9615, 0.6206],
        [0.7240, 0.0518, 0.3923]],
        device='cuda:0')

cf = myNetwork()
cf.to(torch.device("cuda:0"))
```
DataLoading

DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
    batch_sampler=None, num_workers=0, collate_fn=None,
    pin_memory=False, drop_last=False, timeout=0,
    worker_init_fn=None, *, prefetch_factor=2,
    persistent_workers=False)

>>> class MyIterableDataset(torch.utils.data.IterableDataset):
...     def __init__(self, start, end):
...         super(MyIterableDataset).__init__()
...         assert end > start
...         self.start = start
...         self.end = end
...     def __iter__(self):
...         return iter(range(self.start, self.end))
PyTorch Models (torch.nn.Module)

class Mnist_CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1)

    def forward(self, xb):
        xb = xb.view(-1, 1, 28, 28)
        xb = F.relu(self.conv1(xb))
        xb = F.relu(self.conv2(xb))
        xb = F.relu(self.conv3(xb))
        xb = F.avg_pool2d(xb, 4)
        return xb.view(-1, xb.size(1))

No activation by default!

Pretty good documentation: https://pytorch.org/docs/stable/nn.html
Sequential models

```python
model = nn.Sequential(
    nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.AvgPool2d(4),
    Lambda(lambda x: x.view(x.size(0), -1)),
)
```

Defines a single model by applying layers in a sequence with pre-defined methods (i.e. `forward`).
Optimizers

```python
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

The optimizer is pre-defined with the model parameters!

Can provide parameter-specific options!
Losses

Just another nn layer

```python
>>> loss = nn.MSELoss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5)
>>> output = loss(input, target)
>>> output.backward()
```

https://pytorch.org/docs/stable/nn.html#loss-functions
Optimization loop

```python
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

- `optimizer.zero_grad()` zeroes out previously computed gradients.
- `loss.backward()` computes all model grads - maybe less efficient than TF!
- `optimizer.step()` applies new gradient only to parameters used to initialize it.
Computing gradients (e.g. for MAML)

```python
mymodel = Mnist_CNN()
data = torch.rand(16, 1, 28, 28)
loss = torch.mean(torch.max(mymodel(data), axis = -1)[0])
grad = torch.autograd.grad(loss, mymodel.parameters())
```

Currently in beta:

```python
torch.autograd.functional.jacobian(func, inputs, create_graph=False, strict=False)
torch.autograd.functional.hessian(func, inputs, create_graph=False, strict=False)
```
model = MyModel()
opt = torch.optim.Adam(model.parameters())

with higher.innerloop_ctx(model, opt) as (fmodel, diffopt):
    for xs, ys in data:
        logits = fmodel(xs)  # modified `params` can also be passed as a kwarg
        loss = loss_function(logits, ys)  # no need to call loss.backwards()
        diffopt.step(loss)  # note that `step` must take `loss` as an argument!
        # The line above gets P[t+1] from P[t] and loss[t]. `step` also returns
        # these new parameters, as an alternative to getting them from
        # `fmodel.fast_params` or `fmodel.parameters()` after calling
        # `diffopt.step`.

        # At this point, or at any point in the iteration, you can take the
        # gradient of `fmodel.parameters()` (or equivalently
        # `fmodel.fast_params`) w.r.t. `fmodel.parameters(time=0)` (equivalently
        # `fmodel.init_fast_params`). i.e. `fast_params` will always have
        # `grad_fn` as an attribute, and be part of the gradient tape.

You can even nest two higher loops within each other (Check MACAW)!
Backpack package (for higher-order gradients)

https://docs.backpack.pt/en/master/main-api.html#
Recurrent Layers

LSTM

Applying a multi-layer long short-term memory (LSTM) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

\[
\begin{align*}
h_t &= o(W_h x_t + b_h + W_i h_{t-1} + b_i) \\
    f_t &= o(W_f x_t + b_f + W_f h_{t-1} + b_f) \\
    g_t &= \tanh(W_c x_t + b_c + W_c h_{t-1} + b_c) \\
    c_t &= f_t \odot c_{t-1} + g_t \odot \sigma_t \\
    h_t &= o \odot \tanh(c_t)
\end{align*}
\]

where \( h_t \) is the hidden state at time \( t \), \( c_t \) is the cell state at time \( t \), \( h_{t-1} \) is the hidden state of the layer at time \( t-1 \) or the initial hidden state at time \( 0 \), and \( i_t, f_t, g_t, \sigma_t \) are the input, forget, cell, and output gates, respectively. \( \sigma \) is the sigmoid function, and \( \odot \) is the Hadamard product.

In a multilayer LSTM, the input \( x_t^{(0)} \) of the 1st layer \( (l = 0) \) is the hidden state \( h_t^{(l-1)} \) of the previous layer multiplied by dropout \( \delta_t^{(l-1)} \) where each \( \delta_t^{(l-1)} \) is a Bernoulli random variable which is 0 with probability \( \text{dropout} \).

### Parameters
- **input_size** - The number of expected features in the input \( x \)
- **hidden_size** - The number of features in the hidden state \( h \)
- **num_layers** - Number of recurrent layers. \( \text{e.g.}, \text{setting} \ num_layers=2 \ would \ mean \ stacking \ two \ LSTMs \) together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** - If \text{False}, then the layer does not use bias weights \( b_i, b_f, b_c \). Default: \text{True}
- **batch_first** - If \text{True}, then the input and output tensors are provided as \( \text{batch, seq, feature} \). Default: \text{False}
- **dropout** - If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to \( \text{dropout} \). Default: 0
- **bidirectional** - If \text{True}, becomes a bidirectional LSTM. Default: \text{False}

### Inputs: input, \((h_0, c_0)\)

- **input** of shape \( (\text{seq_len}, \text{batch}, \text{input_size}) \): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See \text{torch.nn.utils.rnn.pack_padded_sequence() or torch.nn.utils.rnn.unpack_packed_sequence()} for details.
- **h_0** of shape \( (\text{num_layers} \times \text{num_directions}, \text{batch}, \text{hidden_size}) \): tensor containing the initial hidden state for each element in the batch. If the LSTM is bidirectional, \text{num_directions} should be 2, else it should be 1.
- **c_0** of shape \( (\text{num_layers} \times \text{num_directions}, \text{batch}, \text{hidden_size}) \): tensor containing the initial cell state for each element in the batch.

If \((h_0, c_0)\) is not provided, both \( h_0 \) and \( c_0 \) default to zero.

### Outputs: output, \((h_n, c_n)\)

- **output** of shape \( (\text{seq_len}, \text{batch}, \text{num_directions} \times \text{hidden_size}) \): tensor containing the output features \( (h_t) \) from the last layer of the LSTM, for each \( t \). If a \text{torch.nn.utils.rnn.PackedSequence} has been given as the input, the output will also be a packed sequence.

For the unpacked case, the directions can be separated using \text{output.view(seq_len, batch, num_directions, hidden_size)}, with forward and backward being direction 0 and 1 respectively. Similarly, the directions can be separated in the packed case.

- **h_n** of shape \( (\text{num_layers} \times \text{num_directions}, \text{batch}, \text{hidden_size}) \), with forward and backward being direction 0 and 1 respectively. Similarly, the directions can be separated in the packed case.

- **c_n** of shape \( (\text{num_layers} \times \text{num_directions}, \text{batch}, \text{hidden_size}) \), and similarly for \( c_n \).

LSTM layer by default returns sequences (need this for HW 4).
ProTip (not that Pro): Pack padded sequence/pad packed sequence

```
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

seq = torch.tensor([[1, 2, 0], [3, 0, 0], [4, 5, 6]])

lens = [2, 1, 3]

packed = pack_padded_sequence(seq, lens, batch_first=True, enforce_sorted=False)

seq_unpacked, lens_unpacked = pad_packed_sequence(packed, batch_first=True)

>>> from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
>>> seq = torch.tensor([[1, 2, 0], [3, 0, 0], [4, 5, 6]])
>>> lens = [2, 1, 3]
>>> packed = pack_padded_sequence(seq, lens, batch_first=True, enforce_sorted=False)
>>> packed
PackedSequence(data=tensor([4, 1, 3, 5, 2, 6]), batch_sizes=tensor([3, 2, 1]), sorted_indices=tensor([2, 0, 1]), unsorted_indices=tensor([1, 2, 0]))
>>> seq_unpacked, lens_unpacked = pad_packed_sequence(packed, batch_first=True)
>>> seq_unpacked
tensor([[1, 2, 0],
        [3, 0, 0],
        [4, 5, 6]])
>>> lens_unpacked
tensor([2, 1, 3])
```

Makes RNN runs way faster than TF!
Torch Distributions

mean = torch.rand(4, 3, requires_grad = True)
Out[103]:
tensor([[0.1878, 0.6516, 0.7403],
        [0.4144, 0.9887, 0.0093],
        [0.2708, 0.2635, 0.6638],
        [0.4777, 0.6329, 0.7109]], requires_grad=True)

dist = torch.distributions.normal.Normal(loc = mean, scale = torch.exp(mean))
dist.rsample()
Out[105]:
tensor([[ 0.3194, -1.5584, -3.8187],
        [-2.6826, -0.8975,  1.1454],
        [-2.1106,  1.3008, -3.8159],
        [-0.7909,  2.2228,  2.0558]], grad_fn=<AddBackward0>)

dist.sample()
Out[106]:
tensor([[-0.8447, -1.5922, -0.2065],
        [-0.9781, -1.8587,  0.1368],
        [ 0.3973,  0.4207,  1.7271],
        [ 0.8244, -1.8930,  2.0482]])

**Parameterized** - will compute gradients through the sampling!

**Not parameterized** - will not compute gradients through the sampling!