



UNIVERSIDADE
CATÓLICA
PORTUGUESA

BRAGA

Deep Learning

Session 4

Introduction to PyTorch

Applied Data Science

2024/2025

What is PyTorch?

- **Open-source** deep learning framework developed by Facebook's AI Research lab.
- Allows for flexible **model building and debugging**.
- Pythonic nature with **intuitive APIs**.
- **Extensive resources**, tutorials, and an active user community.
- **Optimized for both CPU and GPU** computation.

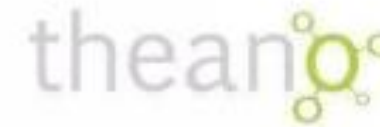
Other Libraries



Caffe



PYTORCH



$\partial y / \text{net}$



Getting Started with PyTorch

- Installation:
 - <https://pytorch.org/get-started/locally/>
- PyTorch installation depends on:
 - OS: Linux, Mac, Windows
 - Package manager: Conda, Pip, LibTorch, from Source
 - Language: Python, C++/Java
 - Compute Platform: CPU, CUDA

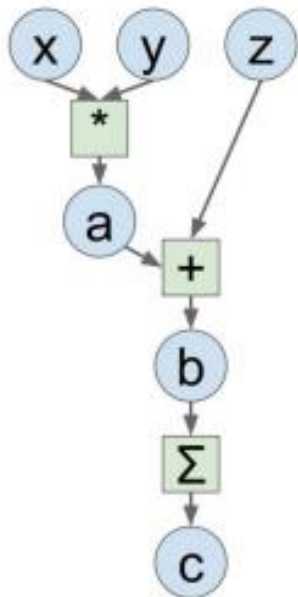
Pytorch Preview



UNIVERSIDADE
CATOLICA
PORTUGUESA

BRAGA

Computation Graph



Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

Tensorflow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                    feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch

N, D = 3, 4

x = torch.rand((N, D), requires_grad=True)
y = torch.rand((N, D), requires_grad=True)
z = torch.rand((N, D), requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
```

Tensors

- A tensor is a multi-dimensional array that generalizes scalars, vectors, and matrices (**similar to numpy arrays**).
- Easy integration with **GPU for accelerated computation**, unlike standard NumPy arrays.
- **Tensor operations are also similar to numpy** (rand, ones, zeros, indexing, slicing, reshape, transpose, cross product, matrix product, element wise multiplication, etc).

Tensors



- Attributes of a tensor:
 - `t = torch.randn(1)`
- *requires_grad* – making a trainable parameter
 - By default False
 - To turn on:
 - `t.requires_grad_()` or
 - `t = torch.randn(1, requires_grad=True)`
 - Accessing tensor values:
 - `t.data`
 - Accessing tensor gradient:
 - `t.grad`
- *grad_fn* – history of operations for autograd
 - `t.grad_fn`

```
1 import torch
2
3 ROWS, COLS = 3, 4
4
5 x = torch.rand(ROWS, COLS, requires_grad=True)
6 y = torch.rand(ROWS, COLS, requires_grad=True)
7 z = torch.rand(ROWS, COLS, requires_grad=True)
8
9 a = x + y
10 b = a * z
11 c = torch.sum(b)
12
13 c.backward()
14
15 print(c.grad_fn)
16 print(x.data)
17 print(x.grad)
```

```
<SumBackward0 object at 0x7f2c1e2fbfd0>
tensor([[0.5857, 0.0051, 0.9537, 0.9890],
        [0.8084, 0.9338, 0.6297, 0.6445],
        [0.9427, 0.1605, 0.8390, 0.8590]])
tensor([[0.3204, 0.1332, 0.5344, 0.2068],
        [0.5323, 0.0115, 0.6808, 0.1606],
        [0.7863, 0.6672, 0.3762, 0.0300]])
```

Loading Data, Devices and CUDA

- Numpy arrays to PyTorch tensors:
 - `torch.from_numpy(x)` - returns a cpu tensor!

- PyTorch tensor to numpy:
 - `t.numpy()`

- Using GPU acceleration:
 - `t.to()`
 - Sends to the chosen device (cuda or cpu)

- Fallback to cpu if gpu is unavailable:
 - `torch.cuda.is_available()`

```
1 import torch
2
3 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
4
5 x = torch.tensor(data=[1, 2, 3, 4, 5], device=device)
6
7 y = torch.tensor([1, 2, 3, 4, 5])
8 print(y.type())
9 y = y.to(device)
10 print(y.type())
```

```
torch.LongTensor
torch.cuda.LongTensor
```



64-bit integer (signed)

torch.LongTensor

torch.cuda.LongTensor

Autograd

- **Automatic Differentiation Package**
- We do not need to worry about **partial differentiation, chain rule**, etc.
- *backward()* does the trick!
 - *loss.backward()*
- Gradients are accumulated for each step by default:
 - Need to zero out gradients after each update:
 - *t.zero_grad()*

```
1 import torch
2
3 # Create tensors
4 x = torch.tensor(data: 1., requires_grad=True)
5 w = torch.tensor(data: 2., requires_grad=True)
6 b = torch.tensor(data: 3., requires_grad=True)
7
8 # Build a computational graph
9 y = w * x + b
10
11 # Compute gradients
12 y.backward()
13
14 # Print out the gradients
15 print(x.grad) # x.grad = 2
16 print(w.grad) # w.grad = 1
17 print(b.grad) # b.grad = 1
```

```
tensor(2.)
tensor(1.)
tensor(1.)
```

- Manual Weight Update Example

```
1 import torch
2
3 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
4
5 x = torch.tensor(data: [[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32, device=device)
6 y = torch.tensor(data: [[0.2], [0.7], [0.9], [0.1]], dtype=torch.float32, device=device)
7
8 w = torch.randn(2, 1, requires_grad=True, device=device)
9 b = torch.randn(1, requires_grad=True, device=device)
10
11 lr = 0.01
12
13 for epoch in range(10):
14     y_pred = x @ w + b
15     error = y - y_pred
16     loss = (error ** 2).mean()
17
18     loss.backward()
19
20     with torch.no_grad():
21         w -= lr * w.grad
22         b -= lr * b.grad
23
24         w.grad.zero_()
25         b.grad.zero_()
26
27     print(f'Epoch {epoch + 1}, Loss: {loss.item()}')
28
29 print(f'Weights: {w}')
30 print(f'Bias: {b}')
```

```
Epoch 1, Loss: 2.6950721740722656
Epoch 2, Loss: 2.549285411834717
Epoch 3, Loss: 2.412501811981201
Epoch 4, Loss: 2.2841575145721436
Epoch 5, Loss: 2.163724899291992
Epoch 6, Loss: 2.0507094860076904
Epoch 7, Loss: 1.9446465969085693
Epoch 8, Loss: 1.8451015949249268
Epoch 9, Loss: 1.7516674995422363
Epoch 10, Loss: 1.6639615297317505
Weights: tensor([[0.3468],
                 [1.2845]], device='cuda:0', requires_grad=True)
Bias: tensor([0.6287], device='cuda:0', requires_grad=True)
```

Optimizers



- Optimizers (optim subpackage):
 - Adam, Adagrad, Adadelata, SGD, etc
 - Manually updating is ok if small number of weights
 - Imagine updating 100k parameters!
 - An optimizer takes the **parameters** we want to update, the **learning rate** we want to use (and possibly many other hyper-parameters as well!) and performs the **updates**.

```
1 import torch
2
3 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
4
5 w = torch.tensor( data: 1, requires_grad=True, dtype=torch.float32, device=device)
6 b = torch.tensor( data: 2, requires_grad=True, dtype=torch.float32, device=device)
7
8 x_train = torch.tensor( data: [1, 2, 3, 4], dtype=torch.float32, device=device)
9 y_train = torch.tensor( data: [2, 4, 6, 8], dtype=torch.float32, device=device)
10
11 lr = 0.01
12 epochs = 10
13
14 # define a SGD optimizer to update the parameters
15 optimizer = torch.optim.SGD( params=[w, b], lr=lr)
16
17 for epoch in range(epochs):
18     y_pred = w * x_train + b
19     loss = torch.mean((y_pred - y_train) ** 2)
20
21     loss.backward()
22
23     optimizer.step()
24
25     optimizer.zero_grad()
26
27     print(f'Epoch {epoch + 1}/{epochs}, loss: {loss.item()}')
28
29 print(f'w: {w.item()}, b: {b.item()}')
```

```
Epoch 1/10, loss: 1.5
Epoch 2/10, loss: 1.2613500356674194
Epoch 3/10, loss: 1.094437599182129
Epoch 4/10, loss: 0.9773091673851013
Epoch 5/10, loss: 0.894733726978302
Epoch 6/10, loss: 0.8361415863037109
Epoch 7/10, loss: 0.7941985130310059
Epoch 8/10, loss: 0.7638153433799744
Epoch 9/10, loss: 0.74146127708080566
Epoch 10/10, loss: 0.724685788154602
w: 1.2615748643875122, b: 2.0198426246643066
```

Loss



- Various predefined loss functions:
 - L1 (MAE), MSE, CrossEntropy, BCE, etc

```
1 import torch
2
3 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
4
5 w = torch.tensor( data: 1, requires_grad=True, dtype=torch.float32, device=device)
6 b = torch.tensor( data: 2, requires_grad=True, dtype=torch.float32, device=device)
7
8 x_train = torch.tensor( data: [1, 2, 3, 4], dtype=torch.float32, device=device)
9 y_train = torch.tensor( data: [2, 4, 6, 8], dtype=torch.float32, device=device)
10
11 lr = 0.01
12 epochs = 10
13
14 # define a loss function
15 loss_fn = torch.nn.MSELoss()
16
17 # define a SGD optimizer to update the parameters
18 optimizer = torch.optim.SGD( params: [w, b], lr=lr)
19
20 for epoch in range(epochs):
21     y_pred = w * x_train + b
22
23     loss = loss_fn(y_pred, y_train)
24
25     loss.backward()
26
27     optimizer.step()
28
29     optimizer.zero_grad()
30
31     print(f'Epoch {epoch + 1}/{epochs}, loss: {loss.item()}')
32
33 print(f'w: {w.item()}, b: {b.item()}')
```

```
Epoch 1/10, loss: 1.5
Epoch 2/10, loss: 1.2613500356674194
Epoch 3/10, loss: 1.094437599182129
Epoch 4/10, loss: 0.9773091673851013
Epoch 5/10, loss: 0.894733726978302
Epoch 6/10, loss: 0.8361415863037109
Epoch 7/10, loss: 0.7941985130310059
Epoch 8/10, loss: 0.7638153433799744
Epoch 9/10, loss: 0.7414612770080566
Epoch 10/10, loss: 0.724685788154602
w: 1.2615748643875122, b: 2.0198426246643066
```

- In PyTorch, a model is represented by a regular Python class that inherits from the Module class.
 - Two components:
 - `__init__(self)`: it defines the parts that make up the model - in our case, two parameters, `w` and `b` (but can be anything!)
 - `forward(self, x)`: it performs the actual computation, that is, it outputs a prediction, given the input `x`

Model

- Example:

```
1 import torch
2 from torch import nn
3
4
5 class ManualLinearRegression(nn.Module):
6     def __init__(self):
7         super().__init__()
8         self.w = nn.Parameter(torch.randn(()))
9         self.b = nn.Parameter(torch.randn(()))
10
11     def forward(self, x):
12         return self.w * x + self.b
```

- Properties:

- `model = ManualLinearRegression()`

- `model.state_dict()` - returns a dictionary of trainable parameters with their current values
 - `model.parameters()` - returns a list of all trainable parameters in the model
 - `model.train()` or `model.eval()`

Putting Things Together



```
1 import torch
2 from torch import nn
3
4 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
5 x_train = torch.tensor(data: [[1., 2.], [2., 3.], [3., 4.], [4., 5.], [5., 6.]], device=device)
6 y_train = torch.tensor(data: [[2.], [3.], [4.], [5.], [6.]], device=device)
7
8 class ManuallinearRegression(nn.Module): 1 usage
9     def __init__(self):
10         super().__init__()
11         self.w = nn.Parameter(torch.randn(1, 2, requires_grad=True, dtype=torch.float32))
12         self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float32))
13
14     def forward(self, x):
15         return torch.mm(x, self.w.t()) + self.b
16
17 # Instantiate the model
18 model = ManuallinearRegression().to(device)
19 # Inspect the parameters
20 print(model.state_dict())
21 # Instantiate the loss
22 criterion = nn.MSELoss()
23 # Instantiate the optimizer
24 optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
25
26 # Training loop
27 n_epochs = 10
28 for epoch in range(n_epochs):
29     model.train()
30     optimizer.zero_grad()
31     y_pred = model(x_train)
32     loss = criterion(y_pred, y_train)
33     loss.backward()
34     optimizer.step()
35     print(f'Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}')
36
37 # Inspect the parameters
38 print(model.state_dict())
```

```
OrderedDict([('w', tensor([[ -1.1242, -0.1247]], device='cuda:0')), ('b', tensor([ -1.0909], device='cuda:0'))])
Epoch 1/10, Loss: 90.43248748779297
Epoch 2/10, Loss: 14.810317039489746
Epoch 3/10, Loss: 2.4655263423919678
Epoch 4/10, Loss: 0.45000359416007996
Epoch 5/10, Loss: 0.12061251699924469
Epoch 6/10, Loss: 0.06646423786878586
Epoch 7/10, Loss: 0.057249199599027634
Epoch 8/10, Loss: 0.055371999740600586
Epoch 9/10, Loss: 0.054695576429367065
Epoch 10/10, Loss: 0.05421821400523186
OrderedDict([('w', tensor([[ -0.0807,  1.2335]], device='cuda:0')), ('b', tensor([ -0.7761], device='cuda:0'))])
```

Complex Models

- Predefined "layer" modules

```
1  from torch import nn
2
3
4  class LayerLinearRegression(nn.Module): 1 usage
5      def __init__(self):
6          super(LayerLinearRegression, self).__init__()
7          self.linear = nn.Linear(in_features=1, out_features=1)
8
9      def forward(self, x):
10         return self.linear(x)
```

- "Sequential" layer modules

```
1  from torch import nn
2
3
4  class LayerLinearRegression(nn.Module): 1 usage
5      def __init__(self):
6          super(LayerLinearRegression, self).__init__()
7          self.seq_linear = nn.Sequential(
8              nn.Linear(in_features=1, out_features=2),
9              nn.ReLU(),
10             nn.Linear(in_features=2, out_features=1)
11         )
12
13     def forward(self, x):
14         return self.seq_linear(x)
```


Dataset



- In PyTorch, a dataset is represented by a regular Python class that inherits from the ***Dataset*** class.
- 3 components:
 - `__init__(self)`
 - `__getitem__(self, index)`
 - `__len__(self)`
- Unless the dataset is huge and cannot fit in memory, you don't explicitly need to define this class. Use ***TensorDataset***!

```
1 import numpy as np
2 import torch
3 from torch.utils.data import Dataset, TensorDataset
4
5 class MyDataset(Dataset):
6     def __init__(self, x_tensor, y_tensor):
7         self.x = x_tensor
8         self.y = y_tensor
9
10    def __len__(self):
11        return len(self.x)
12
13    def __getitem__(self, idx):
14        return self.x[idx], self.y[idx]
15
16    x_numpy = np.random.rand(100, 10)
17    y_numpy = np.random.rand(100, 1)
18
19    x_train = torch.from_numpy(x_numpy).float()
20    y_train = torch.from_numpy(y_numpy).float()
21
22    train_dataset = MyDataset(x_train, y_train)
23    print(train_dataset[0])
24
25    train_dataset = TensorDataset(*train_dataset.tensors)
26    print(train_dataset[0])
```

```
(tensor([0.2782, 0.2468, 0.5286, 0.6951, 0.2164, 0.3121, 0.2749, 0.5096, 0.4903,
         0.1141]), tensor([0.2687]))
(tensor([0.2782, 0.2468, 0.5286, 0.6951, 0.2164, 0.3121, 0.2749, 0.5096, 0.4903,
         0.1141]), tensor([0.2687]))
```

- But if we have a huge dataset? We need to train in '**batches**'!
- Use PyTorch's ***Dataloader*** class!
 - We tell it which **dataset** to use, the desired **mini-batch size** and if we'd like to **shuffle** it or not. That's it!
 - Our loader will behave like an iterator, so we can loop over it and fetch a different mini-batch every time.

```
29 from torch.utils.data import DataLoader
30
31 train_loader = DataLoader(dataset=train_dataset, batch_size=16, shuffle=True)
32 print(next(iter(train_loader)))
```

```
(tensor([0.1614, 0.0563, 0.1415, 0.1973, 0.7612, 0.6448, 0.4764, 0.3098, 0.3750,
0.3178]), tensor([0.9990]))
(tensor([0.1614, 0.0563, 0.1415, 0.1973, 0.7612, 0.6448, 0.4764, 0.3098, 0.3750,
0.3178]), tensor([0.9990]))
[tensor([[0.9054, 0.8853, 0.9626, 0.8340, 0.2862, 0.8917, 0.1765, 0.7521, 0.7377,
0.3932],
[0.3919, 0.8569, 0.2240, 0.5022, 0.6886, 0.4805, 0.7269, 0.0316, 0.6883,
0.2496],
```

DataLoader in Practice



```
28 from torch.utils.data import DataLoader
29
30 train_loader = DataLoader(dataset=train_dataset, batch_size=16, shuffle=True)
31
32 model = ManualLinearRegression().to(device)
33
34 loss_fn = nn.MSELoss()
35 optimizer = torch.optim.SGD(model.parameters(), lr=lr)
36
37 for epoch in range(n_epochs):
38     for x_batch, y_batch in train_loader:
39         x_batch = x_batch.to(device)
40         y_batch = y_batch.to(device)
41
42         # set model to train mode
43         model.train()
44
45         # forward pass
46         y_pred = model(x_batch)
47
48         # compute loss
49         loss = loss_fn(y_pred, y_batch)
50
51         # zero gradients
52         optimizer.zero_grad()
53
54         # backward pass
55         loss.backward()
56
57         # update weights
58         optimizer.step()
59
60     # print loss
61     print(f'Epoch {epoch+1}, Loss: {loss.item()}')
62
63 print(model.state_dict())
```

```
Epoch 1, Loss: 0.5525757074356079
Epoch 2, Loss: 0.1374179720878601
Epoch 3, Loss: 0.06684266030788422
Epoch 4, Loss: 0.03609754517674446
Epoch 5, Loss: 0.029070958495140076
Epoch 6, Loss: 0.026194017380475998
Epoch 7, Loss: 0.022452618926763535
Epoch 8, Loss: 0.1915893256664276
Epoch 9, Loss: 0.05336730182170868
Epoch 10, Loss: 0.1300673484802246
OrderedDict([('linear.weight', tensor([[ 0.3641,  0.3745, -0.1619,  0.2516, -0.1009,  0.0481,  0.3232, -0.0076,
      0.0424,  0.0021]], device='cuda:0')), ('linear.bias', tensor([-0.0796], device='cuda:0'))])
```

Split Data

- Random train, validationa and test split
 - *random_split()*

```
1 import numpy as np
2 import torch
3 from torch.utils.data import TensorDataset
4
5 x_numpy = np.random.rand(100, 10)
6 y_numpy = np.random.rand(100, 1)
7
8 x_train = torch.from_numpy(x_numpy).float()
9 y_train = torch.from_numpy(y_numpy).float()
10
11 dataset = TensorDataset(*tensors: x_train, y_train)
12
13 train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, lengths: [60, 20, 20])
14
15 train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=10, shuffle=True)
16 val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=10, shuffle=False)
17 test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=10, shuffle=False)
```

Saving/Loading Weights

- Method 1:
 - Only inference/evaluation - we only need *state_dict*
 - *Save*:

```
torch.save(model.state_dict(), PATH)
```
 - *Load*:

```
model = TheModelClass(*args, **kwargs)  
model.load_state_dict(torch.load(PATH, weights_only=True))  
model.eval()
```
- A common PyTorch convention is to save models using either a *.pt* or *.pth* file extension.

Saving/Loading Weights

- Method 2:
 - Checkpoints – to resume training / inference

- Save:

```
torch.save({  
    'epoch': epoch,  
    'model_state_dict': model.state_dict(),  
    'optimizer_state_dict': optimizer.state_dict(),  
    'loss': loss,  
    ...  
}, PATH)
```

- Load:

```
model = TheModelClass(*args, **kwargs)  
optimizer = TheOptimizerClass(*args, **kwargs)  
  
checkpoint = torch.load(PATH, weights_only=True)  
model.load_state_dict(checkpoint['model_state_dict'])  
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])  
epoch = checkpoint['epoch']  
loss = checkpoint['loss']  
  
model.eval()  
# - OR -  
model.train()
```

Evaluation

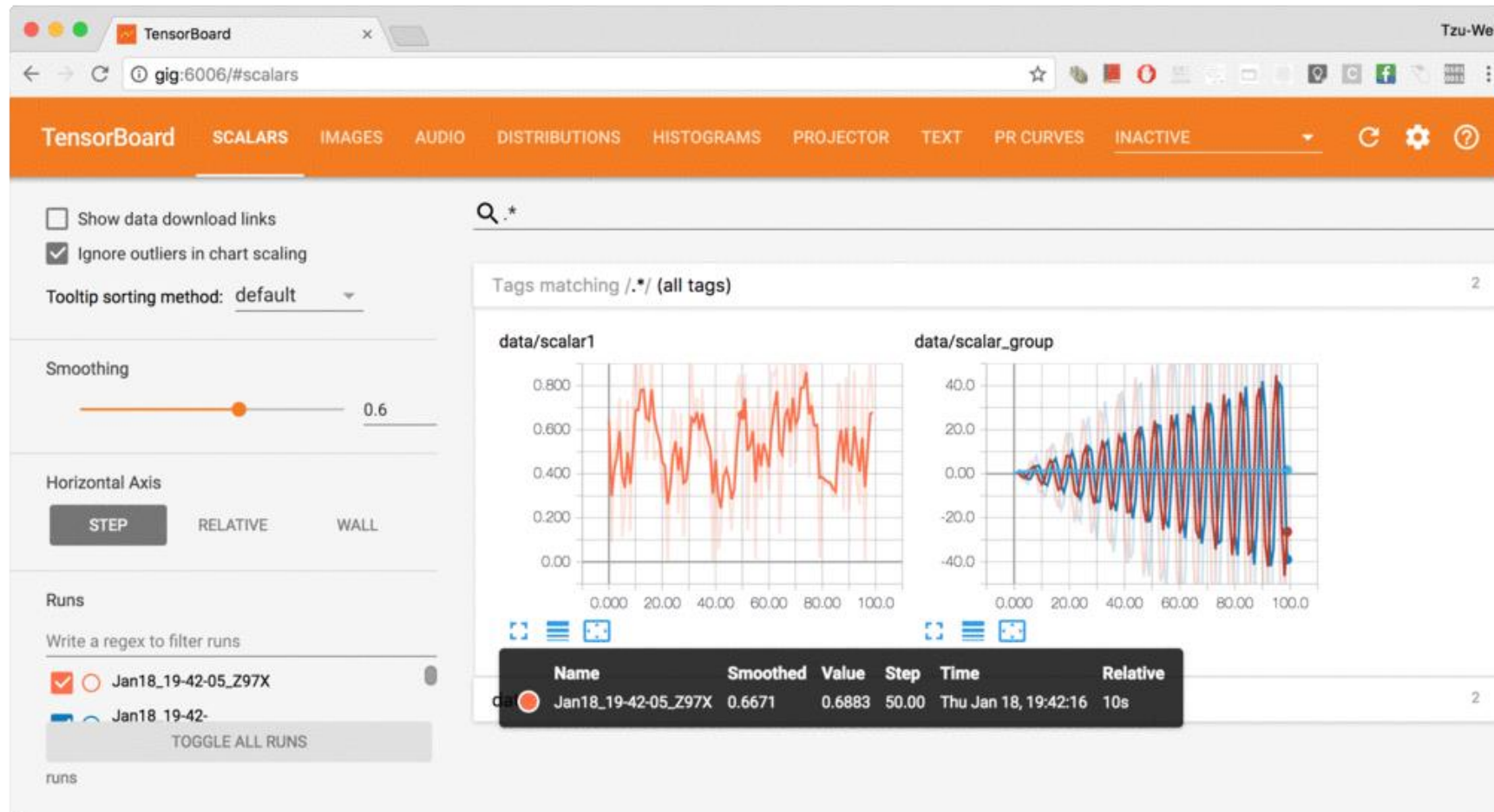


- Two important things:
 - *torch.no_grad()*
 - Don't store the history of all computations.
 - *model.eval()*
 - Tell the compiler which mode to run on.

```
25 val_losses = []
26
27 for epoch in range(n_epochs):
28     for x_batch, y_batch in train_loader:
29         x_batch = x_batch.to(device)
30         y_batch = y_batch.to(device)
31
32         model.train()
33
34         y_pred = model(x_batch)
35         loss = loss_fn(y_pred, y_batch)
36
37         optimizer.zero_grad()
38         loss.backward()
39         optimizer.step()
40
41     with torch.no_grad():
42         for x_val, y_val in val_loader:
43             x_val = x_val.to(device)
44             y_val = y_val.to(device)
45
46             model.eval()
47
48             y_pred = model(x_val)
49             val_loss = loss_fn(y_pred, y_val)
50             val_losses.append(val_loss.item())
```

Visualization

- **TensorboardX** (visualize training)



Visualization

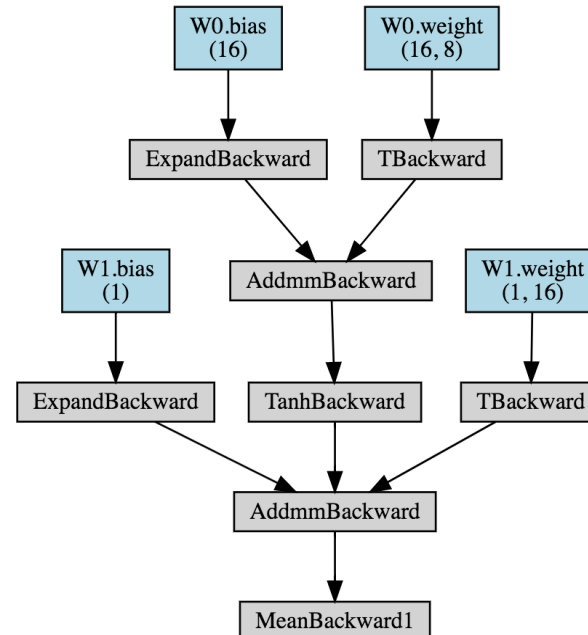
- **PyTorchViz** (visualize computation graph)

```
In [2]: model = nn.Sequential()
model.add_module('W0', nn.Linear(8, 16))
model.add_module('tanh', nn.Tanh())
model.add_module('W1', nn.Linear(16, 1))

x = Variable(torch.randn(1,8))
y = model(x)

make_dot(y.mean(), params=dict(model.named_parameters()))
```

Out[2]:



Resources

- PyTorch Documentation
 - <https://pytorch.org/>
 - <https://github.com/pytorch/pytorch>
- Tutorials:
 - <https://github.com/hunkim/PyTorchZeroToAll>
 - https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
 - <https://www.learnpytorch.io/>

Exercise

- End-to-End PyTorch exercise
 1. Load the data
 2. Define the model
 3. Define the loss function
 4. Define the optimizer
 5. Train the model
 6. Make predictions