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

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

Getting Started with PyTorch

- Installation:

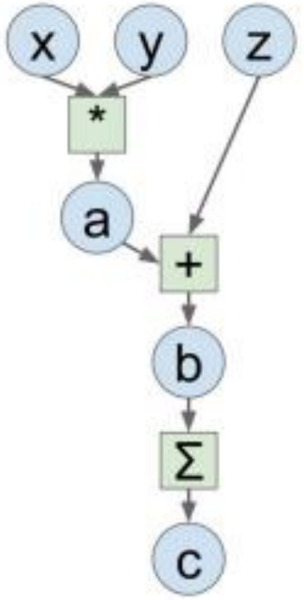
  - https://pytorch.org/get-started/locally/

- PyTorch installation depends on:
  - OS: Linux, Mac, Windows
  - Package manager: Conda, Pip, LibTorch, from Source
  - Language: Python, C++/Java
  - Cimpute Platform: CPU, CUDA

Pytorch Preview

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

Computation Graph



Numpy

```python
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

```python
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

```python
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

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

- 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* – hystory of operations for autograd
  - *t.grad_fn*

```python
import torch

ROWS, COLS = 3, 4

x = torch.rand(ROWS, COLS, requires_grad=True)
y = torch.rand(ROWS, COLS, requires_grad=True)
z = torch.rand(ROWS, COLS, requires_grad=True)

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

c.backward()

print(c.grad_fn)
print(x.data)
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

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

- 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()*

```
import torch

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

x = torch.tensor( data: [1, 2, 3, 4, 5], device=device)

y = torch.tensor([1, 2, 3, 4, 5])
print(y.type())
y = y.to(device)
print(y.type())
```

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

64-bit integer (signed)	torch.LongTensor	torch.cuda.LongTensor

Autograd

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

- **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()*

```python
import torch

# Create tensors
x = torch.tensor(data: 1., requires_grad=True)
w = torch.tensor(data: 2., requires_grad=True)
b = torch.tensor(data: 3., requires_grad=True)

# Build a computational graph
y = w * x + b

# Compute gradients
y.backward()

# Print out the gradients
print(x.grad)   # x.grad = 2
print(w.grad)   # w.grad = 1
print(b.grad)   # b.grad = 1
```

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

Autograd

- Manual Weight Update Example

```python
import torch

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

x = torch.tensor( data: [[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32, device=device)
y = torch.tensor( data: [[0.2], [0.7], [0.9], [0.1]], dtype=torch.float32, device=device)

w = torch.randn(2, 1, requires_grad=True, device=device)
b = torch.randn(1, requires_grad=True, device=device)

lr = 0.01

for epoch in range(10):
    y_pred = x @ w + b
    error = y - y_pred
    loss = (error ** 2).mean()

    loss.backward()

    with torch.no_grad():
        w -= lr * w.grad
        b -= lr * b.grad

        w.grad.zero_()
        b.grad.zero_()

    print(f'Epoch {epoch + 1}, Loss: {loss.item()}')

print(f'Weights: {w}')
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, Adadelta, 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.**

```python
import torch

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

w = torch.tensor( data: 1, requires_grad=True, dtype=torch.float32, device=device)
b = torch.tensor( data: 2, requires_grad=True, dtype=torch.float32, device=device)

x_train = torch.tensor( data: [1, 2, 3, 4], dtype=torch.float32, device=device)
y_train = torch.tensor( data: [2, 4, 6, 8], dtype=torch.float32, device=device)

lr = 0.01
epochs = 10

# define a SDG optimizer to update the parameters
optimizer = torch.optim.SGD( params: [w, b], lr=lr)

for epoch in range(epochs):
    y_pred = w * x_train + b
    loss = torch.mean((y_pred - y_train) ** 2)

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()

    print(f'Epoch {epoch + 1}/{epochs}, loss: {loss.item()}')

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
```

Loss

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

```python
import torch

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

w = torch.tensor( data: 1, requires_grad=True, dtype=torch.float32, device=device)
b = torch.tensor( data: 2, requires_grad=True, dtype=torch.float32, device=device)

x_train = torch.tensor( data: [1, 2, 3, 4], dtype=torch.float32, device=device)
y_train = torch.tensor( data: [2, 4, 6, 8], dtype=torch.float32, device=device)

lr = 0.01
epochs = 10

# define a loss function
loss_fn = torch.nn.MSELoss()

# define a SDG optimizer to update the parameters
optimizer = torch.optim.SGD( params: [w, b], lr=lr)

for epoch in range(epochs):
    y_pred = w * x_train + b

    loss = loss_fn(y_pred, y_train)

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()

    print(f'Epoch {epoch + 1}/{epochs}, loss: {loss.item()}')

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
```

Model

- In PyTorch, a model is represented by a regular Python class that inherits ftom 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:

```python
import torch
from torch import nn


class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        self.w = nn.Parameter(torch.randn(()))
        self.b = nn.Parameter(torch.randn(()))

    def forward(self, x):
        return self.w * x + self.b
```

- Properties:
    - model = ManualLinearRegression()
        - model.state_dic() - 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 Thinghs Together

```python
import torch
from torch import nn

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
x_train = torch.tensor( data: [[1., 2.], [2., 3.], [3., 4.], [4., 5.], [5., 6.]], device=device)
y_train = torch.tensor( data: [[2.], [3.], [4.], [5.], [6.]], device=device)

class ManualLinearRegression(nn.Module):  1 usage
    def __init__(self):
        super().__init__()
        self.w = nn.Parameter(torch.randn(1, 2, requires_grad=True, dtype=torch.float32))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float32))

    def forward(self, x):
        return torch.mm(x, self.w.t()) + self.b

# Instantiate the model
model = ManualLinearRegression().to(device)
# Inspect the parameters
print(model.state_dict())
# Instantiate the loss
criterion = nn.MSELoss()
# Instantiate the optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
n_epochs = 10
for epoch in range(n_epochs):
    model.train()
    optimizer.zero_grad()
    y_pred = model(x_train)
    loss = criterion(y_pred, y_train)
    loss.backward()
    optimizer.step()
    print(f'Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}')

# Inspect the parameters
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

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

- Predefined "layer" modules

```python
from torch import nn


class LayerLinearRegression(nn.Module):
    def __init__(self):
        super(LayerLinearRegression, self).__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)
```

- "Sequential" layer modules

```python
from torch import nn


class LayerLinearRegression(nn.Module):
    def __init__(self):
        super(LayerLinearRegression, self).__init__()
        self.seq_linear = nn.Sequential(
            nn.Linear(1, 2),
            nn.ReLU(),
            nn.Linear(2, 1)
        )

    def forward(self, x):
        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)*
  - *__get_item__(self, index)*
  - *__len__(self)*

- Unless the dataset is huge and cannot fit in memory, you don't explictly need to define this class. Use ***TensorDataset***!

```python
import numpy as np
import torch
from torch.utils.data import Dataset, TensorDataset

class MyDataset(Dataset):  1 usage
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

x_numpy = np.random.rand(100, 10)
y_numpy = np.random.rand(100, 1)

x_train = torch.from_numpy(x_numpy).float()
y_train = torch.from_numpy(y_numpy).float()

train_dataset = MyDataset(x_train, y_train)
print(train_dataset[0])

train_dataset = TensorDataset( *tensors: x_train, y_train)
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]))
```

DataLoader

- 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.

```python
from torch.utils.data import DataLoader

train_loader = DataLoader(dataset=train_dataset, batch_size=16, shuffle=True)
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

UNIVERSIDADE CATÓLICA PORTUGUESA

BRAGA

```python
from torch.utils.data import DataLoader

train_loader = DataLoader(dataset=train_dataset, batch_size=16, shuffle=True)

model = ManualLinearRegression().to(device)

loss_fn = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr)

for epoch in range(n_epochs):
    for x_batch, y_batch in train_loader:
        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)

        # set model to train mode
        model.train()

        # forward pass
        y_pred = model(x_batch)

        # compute loss
        loss = loss_fn(y_pred, y_batch)

        # zero gradients
        optimizer.zero_grad()

        # backward pass
        loss.backward()

        # update weights
        optimizer.step()

    # print loss
    print(f'Epoch {epoch+1}, Loss: {loss.item()}')

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()*

```python
import numpy as np
import torch
from torch.utils.data import TensorDataset

x_numpy = np.random.rand(100, 10)
y_numpy = np.random.rand(100, 1)

x_train = torch.from_numpy(x_numpy).float()
y_train = torch.from_numpy(y_numpy).float()

dataset = TensorDataset( *tensors: x_train, y_train)

train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset,  lengths: [60, 20, 20])

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=10, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=10, shuffle=False)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=10, shuffle=False)
```

Saving/Loading Weights

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

- 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

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

- 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

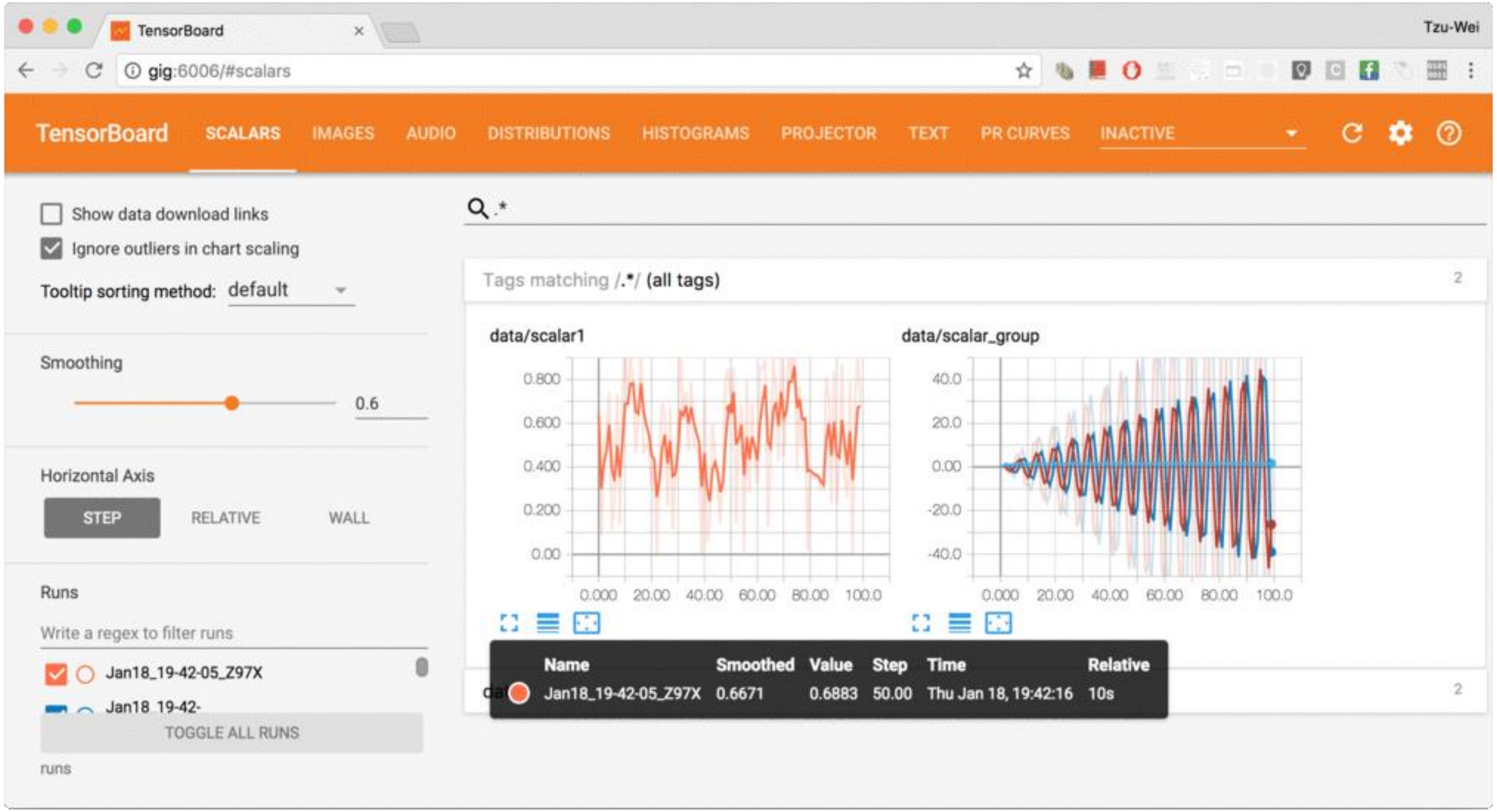
- Two important things:
  - *torch.no_grad()*
    - Don't store the history of all computations.
  - *model.eval()*
    - Tell the compliler which mode to run on.

```python
val_losses = []

for epoch in range(n_epochs):
    for x_batch, y_batch in train_loader:
        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)

        model.train()

        y_pred = model(x_batch)
        loss = loss_fn(y_pred, y_batch)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    with torch.no_grad():
        for x_val, y_val in val_loader:
            x_val = x_val.to(device)
            y_val = y_val.to(device)

            model.eval()

            y_pred = model(x_val)
            val_loss = loss_fn(y_pred, y_val)
            val_losses.append(val_loss.item())
```

Visualization

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

- **TensorboardX** (visualize training)

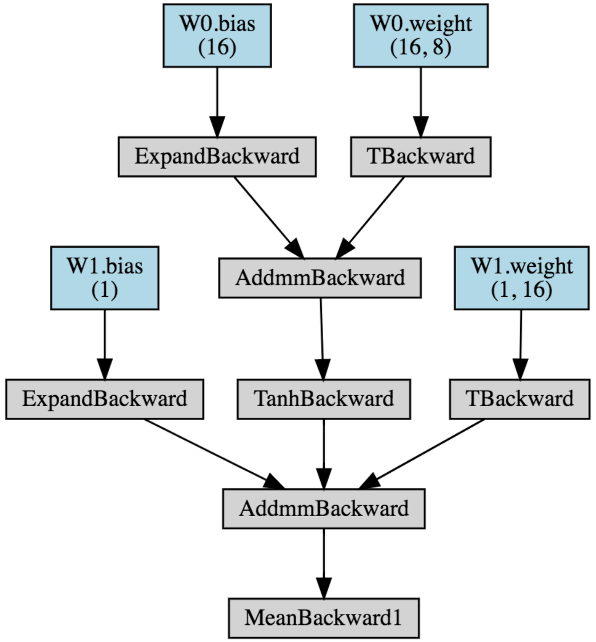


Visualization

- **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

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

- 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

UNIVERSIDADE CATÓLICA PORTUGUESA
BRAGA

- 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