Lec 22 - pytorch

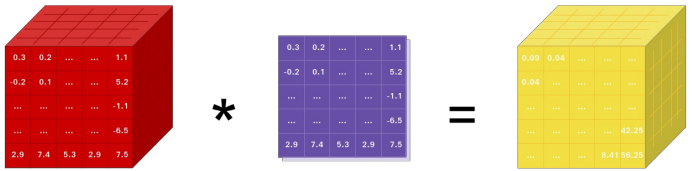
Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

PyTorch

> PyTorch is a Python package that provides two high-level features:
>
> - Tensor computation (like NumPy) with strong GPU acceleration
> - Deep neural networks built on a tape-based autograd system



A graph is created on the fly


```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

```
import torch
torch.__version__
```

Tensors

are the basic data abstraction in PyTorch and are implemented by the `torch.Tensor` class. The behave in much the same was as the other array libraries we've seen so far (`numpy`, `theano`, etc.)

```
torch.zeros(3)
```

```
## tensor([0., 0., 0.])
```

```
torch.ones(3,2)
```

```
## tensor([[1., 1.],
##         [1., 1.],
##         [1., 1.]])
```

```
torch.empty(2,2,2)
```

```
## tensor([[[ 0.0000e+00, -3.6893e+19],
##          [ 4.2163e-29, -3.6893e+19]],
##
##         [[ 1.0000e+00,  1.0000e+00],
##          [ 4.6725e+20,  1.4013e-45]]])
```

```
torch.manual_seed(1234)
```

```
## <torch._C.Generator object at 0x2aef26c50>
```

```
torch.rand(2,2,2,2)
```

```
## tensor([[[[0.0290, 0.4019],
##           [0.2598, 0.3666]],
##
##          [[0.0583, 0.7006],
##           [0.0518, 0.4681]]],
##
##
##         [[[0.6738, 0.3315],
##           [0.7837, 0.5631]],
##
##          [[0.7749, 0.8208],
##           [0.2793, 0.6817]]]])
```

Constant values

As expected, tensors can be constructed from constant numeric values in lists or tuples.

```
torch.tensor(1)
```

```
## tensor(1)
```

```
torch.tensor((1,2))
```

```
## tensor([1, 2])
```

```
torch.tensor([[1,2,3], [4,5,6]])
```

```
## tensor([[1, 2, 3],
##         [4, 5, 6]])
```

```
torch.tensor([(1,2,3), [4,5,6]])
```

```
## tensor([[1, 2, 3],
##         [4, 5, 6]])
```

```
torch.tensor([(1,1,1), [4,5]])
```

```
## ValueError: expected sequence of length 3 at dim 1 (
```

```
torch.tensor([["A"]])
```

```
## ValueError: too many dimensions 'str'
```

```
torch.tensor([[True]]).dtype
```

```
## torch.bool
```

Note using `tensor()` in this way results in a full copy of the data.

Tensor Types

Data type	dtype	type()	Comment
32-bit float	`float32` or `float`	`FloatTensor`	Default float type
64-bit float	`float64` or `double`	`DoubleTensor`	
16-bit float	`float16` or `half`	`HalfTensor`	
16-bit brain float	`bfloat16`	`BFloat16Tensor`	
64-bit complex float	`complex64`		
128-bit complex float	`complex128` or `cdouble`		
8-bit integer (unsigned)	`uint8`	`ByteTensor`	
8-bit integer (signed)	`int8`	`CharTensor`	
16-bit integer (signed)	`int16` or `short`	`ShortTensor`	
32-bit integer (signed)	`int32` or `int`	`IntTensor`	

We've left off quantized integer types here

Specifying types

Just like NumPy and Pandas, types are specified via the `dtype` argument and can be inspected via the dtype attribute.

```
a = torch.tensor([1,2,3])
a
```

```
## tensor([1, 2, 3])
```

```
a.dtype
```

```
## torch.int64
```

```
b = torch.tensor([1,2,3], dtype=torch.float16)
b
```

```
## tensor([1., 2., 3.], dtype=torch.float16)
```

```
b.dtype
```

```
## torch.float16
```

```
c = torch.tensor([1.,2.,3.])
c
```

```
## tensor([1., 2., 3.])
```

```
c.dtype
```

```
## torch.float32
```

```
d = torch.tensor([1,2,3], dtype=torch.float64)
d
```

```
## tensor([1., 2., 3.], dtype=torch.float64)
```

```
d.dtype
```

```
## torch.float64
```

Type precision

When using types with less precision it is important to be careful about underflow and overflow (ints) and rounding errors (floats).

```
torch.tensor([300], dtype=torch.int8)
```

```
## tensor([44], dtype=torch.int8)
```

```
torch.tensor([-300]).to(torch.int8)
```

```
## tensor([-44], dtype=torch.int8)
```

```
torch.tensor([-300]).to(torch.uint8)
```

```
## tensor([212], dtype=torch.uint8)
```

```
torch.tensor([300]).to(torch.int16)
```

```
## tensor([300], dtype=torch.int16)
```

```
torch.set_printoptions(precision=8)

torch.tensor(1/3, dtype=torch.float16)
```

```
## tensor(0.33325195, dtype=torch.float16)
```

```
torch.tensor(1/3, dtype=torch.float32)
```

```
## tensor(0.33333334)
```

```
torch.tensor(1/3, dtype=torch.float64)
```

```
## tensor(0.33333333, dtype=torch.float64)
```

NumPy conversion

It is possible to easily move between NumPy arrays and Tensors via the `from_numpy()` function and `numpy()` method.

```
a = np.eye(3,3)
torch.from_numpy(a)
```

```
## tensor([[1., 0., 0.],
##         [0., 1., 0.],
##         [0., 0., 1.]], dtype=torch.float64)
```

```
b = np.array([1,2,3])
torch.from_numpy(b)
```

```
## tensor([1, 2, 3])
```

```
c = torch.rand(2,3)
c.numpy()
```

```
## array([[0.28367, 0.65673, 0.23876],
##        [0.73128, 0.60122, 0.30433]], dtype=float32)
```

```
d = torch.ones(2,2, dtype=torch.int64)
d.numpy()
```

Math & Logic

Just like NumPy tensors support basic mathematical and logical operations with scalars and other tensors - the PyTorch library provides implementations of most commonly needed mathematical and related functions.

```
torch.ones(2,2) * 7 -1
```

```
## tensor([[6., 6.],
##         [6., 6.]])
```

```
torch.ones(2,2) + torch.tensor([[1,2], [3,4]])
```

```
## tensor([[2., 3.],
##         [4., 5.]])
```

```
2 ** torch.tensor([[1,2], [3,4]])
```

```
## tensor([[ 2,  4],
##         [ 8, 16]])
```

```
2 ** torch.tensor([[1,2], [3,4]]) > 5
```

```
## tensor([[False, False],
```

```
x = torch.rand(2,2)

torch.ones(2,2) @ x
```

```
## tensor([[1.22126317, 1.36931109],
##         [1.22126317, 1.36931109]])
```

```
torch.clamp(x*2-1, -0.5, 0.5)
```

```
## tensor([[-0.49049568,  0.25872374],
##         [ 0.50000000,  0.47989845]])
```

```
torch.mean(x)
```

```
## tensor(0.64764357)
```

```
torch.sum(x)
```

Broadcasting

Like NumPy in cases where tensor dimensions do not match, the broadcasting algorithm is used. The rules for broadcasting are:

> - Each tensor must have at least one dimension - no empty tensors.
> - Comparing the dimension sizes of the two tensors, going from last to first:
>   - Each dimension must be equal, or
>   - One of the dimensions must be of size 1, or
>   - The dimension does not exist in one of the tensors

Exercise 1

Consider the following 6 tensors:

```
a = torch.rand(4, 3, 2)
b = torch.rand(3, 2)
c = torch.rand(2, 3)
d = torch.rand(0)
e = torch.rand(3, 1)
f = torch.rand(1, 2)
```

which of the above could be multiplied together and produce a valid result via broadcasting (e.g. `a*b`, `a*c`, `a*d`, etc.).

Explain why or why not broadcasting was able to be applied in each case.

Inplace modification

In instances where we need to conserve memory it is possible to apply many functions in a way where a new tensor is not created but the original values are replaced. These functions share the same name with the original functions but have a _ suffix.

```
a = torch.rand(2,2)
print(a)
```

```
## tensor([[0.31861043, 0.29080772],
##         [0.41960979, 0.37281448]])
```

```
print(torch.exp(a))
```

```
## tensor([[1.37521553, 1.33750737],
##         [1.52136779, 1.45181501]])
```

```
print(a)
```

```
## tensor([[0.31861043, 0.29080772],
##         [0.41960979, 0.37281448]])
```

```
print(torch.exp_(a))
```

```
## tensor([[1.37521553, 1.33750737],
##         [1.52136779, 1.45181501]])
```

```
print(a)
```

```
## tensor([[1.37521553, 1.33750737],
##         [1.52136779, 1.45181501]])
```

For functions without a _ variant, check if the have a `to` argument which can then be used instead - see `torch.matmul()`

Inplace arithmetic

All arithmetic functions are available as methods of the Tensor class,

```
a = torch.ones(2, 2)
b = torch.rand(2, 2)
```

```
a+b
```

```
## tensor([[1.37689185, 1.01077938],
##         [1.94549370, 1.76611161]])
```

```
print(a)
```

```
## tensor([[1., 1.],
##         [1., 1.]])
```

```
print(b)
```

```
## tensor([[0.37689191, 0.01077944],
##         [0.94549364, 0.76611167]])
```

```
a.add_(b)
```

```
## tensor([[1.37689185, 1.01077938],
##         [1.94549370, 1.76611161]])
```

```
print(a)
```

```
## tensor([[1.37689185, 1.01077938],
##         [1.94549370, 1.76611161]])
```

```
print(b)
```

```
## tensor([[0.37689191, 0.01077944],
##         [0.94549364, 0.76611167]])
```

Changing tensor shapes

The `shape` of a tensor can be changed using the `view()` or `reshape()` methods. The former guarantees that the result shares data with the original object (but requires contiguity),the latter may or may not copy the data.

```
x = torch.zeros(3, 2)
y = x.view(2, 3)
y
```

```
## tensor([[0., 0., 0.],
##         [0., 0., 0.]])
```

```
x.fill_(1)
```

```
## tensor([[1., 1.],
##         [1., 1.],
##         [1., 1.]])
```

```
y
```

```
## tensor([[1., 1., 1.],
##         [1., 1., 1.]])
```

```
x = torch.zeros(3, 2)
y = x.t()
z = y.view(6)
```

```
## RuntimeError: view size is not compatible with input
```

```
z = y.reshape(6)
x.fill_(1)
```

```
## tensor([[1., 1.],
##         [1., 1.],
##         [1., 1.]])
```

```
y
```

```
## tensor([[1., 1., 1.],
##         [1., 1., 1.]])
```

Adding or removing dimensions

The `squeeze()` and `unsqueeze()` methods can be used to remove or add length 1 dimension(s) to a tensor.

```
x = torch.zeros(1,3,1)
x.squeeze().shape
```

```
## torch.Size([3])
```

```
x.squeeze(0).shape
```

```
## torch.Size([3, 1])
```

```
x.squeeze(1).shape
```

```
## torch.Size([1, 3, 1])
```

```
x.squeeze(2).shape
```

```
## torch.Size([1, 3])
```

```
y = x.squeeze()
x.fill_(1)
```

```
x = torch.zeros(3,2)
x.unsqueeze(0).shape
```

```
## torch.Size([1, 3, 2])
```

```
x.unsqueeze(1).shape
```

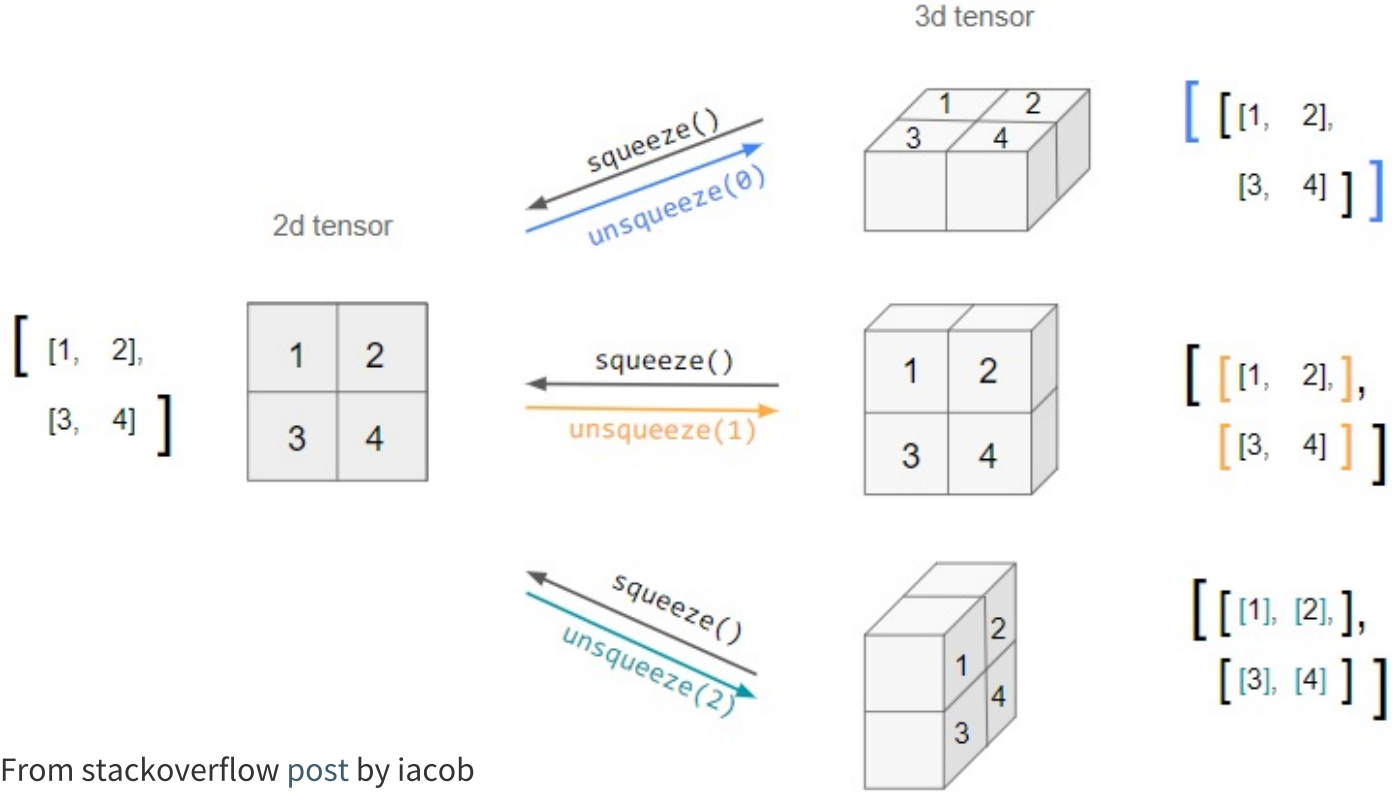
```
## torch.Size([3, 1, 2])
```

```
x.unsqueeze(2).shape
```

```
## torch.Size([3, 2, 1])
```

```
y = x.unsqueeze(1)
x.fill_(1)
```

```
## tensor([[1., 1.],
##         [1., 1.],
##         [1., 1.]])
```



From stackoverflow post by iacob

Exercise 2

Given the following tensors,

```
a = torch.ones(4,3,2)
b = torch.rand(3)
c = torch.rand(5,3)
```

what reshaping is needed to make it possible so that `a * b` and `a * c` can be calculated via broadcasting?

Autograd

Tensor expressions

Gradient tracking can be enabled using the `requires_grad` argument at initialization, alternatively the `requires_grad` flag can be set on tensor or the `enable_grad()` context manager used.

```
x = torch.linspace(0, 2, steps=21, requires_grad=True)
x
```

```
## tensor([0.00000000, 0.10000000, 0.20000000, 0.30000001, 0.40000001, 0.50000000,
##         0.60000002, 0.69999999, 0.80000001, 0.90000004, 1.00000000, 1.10000002,
##         1.20000005, 1.30000007, 1.39999998, 1.50000000, 1.60000002, 1.70000005,
##         1.79999995, 1.89999998, 2.00000000], requires_grad=True)
```

```
y = 3*x + 2
y
```

```
## tensor([2.00000000, 2.29999995, 2.59999990, 2.90000010, 3.20000005, 3.50000000,
##         3.80000019, 4.09999990, 4.40000010, 4.69999981, 5.00000000, 5.30000019,
##         5.60000038, 5.90000010, 6.19999981, 6.50000000, 6.80000019, 7.10000038,
##         7.39999962, 7.69999981, 8.00000000], grad_fn=<AddBackward0>)
```

Computational graph

```
y.grad_fn
```

```
## <AddBackward0 object at 0x2b1c5d790>
```

```
y.grad_fn.next_functions
```

```
## ((<MulBackward0 object at 0x2b1c5d250>, 0), (None, 0))
```

```
y.grad_fn.next_functions[0][0].next_functions
```

```
## ((<AccumulateGrad object at 0x2b1c5d4f0>, 0), (None, 0))
```

```
y.grad_fn.next_functions[0][0].next_functions[0][0].next_functions
```

```
## ()
```

Autogradient

In order to calculate the gradients we use the `backward()` method on the calculation output tensor (must be a scalar), this then makes the grad attribute available for the input (leaf) tensors.

```
out = y.sum()
out.backward()
out
```

```
## tensor(105., grad_fn=<SumBackward0>)
```

```
y.grad
```

```
## UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute
   be populated during autograd.backward(). If you indeed want the .grad field to be populated for a non-leaf Te
   use .retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access
   leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations. (Triggered internally a
   Users/distiller/project/pytorch/build/aten/src/ATen/core/TensorBody.h:475.)
##   return self._grad
```

```
x.grad
```

```
## tensor([3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
```

A bit more complex

```
n = 21
x = torch.linspace(0, 2, steps=n, requires_grad=True)
m = torch.rand(n, requires_grad=True)

y = m*x + 2

y.backward(torch.ones(n))
```

```
x.grad
```

```
## tensor([0.23227984, 0.72686875, 0.11874896, 0.39512146, 0.71987736, 0.75950843,
##         0.53108865, 0.64494550, 0.72242016, 0.44158769, 0.36338443, 0.88182861,
##         0.98741043, 0.73160070, 0.28143251, 0.06507802, 0.00649202, 0.50345892,
##         0.30815977, 0.37417805, 0.42968810])
```

```
m.grad
```

```
## tensor([0.00000000, 0.10000000, 0.20000000, 0.30000001, 0.40000001, 0.50000000,
##         0.60000002, 0.69999999, 0.80000001, 0.90000004, 1.00000000, 1.10000002,
##         1.20000005, 1.30000007, 1.39999998, 1.50000000, 1.60000002, 1.70000005,
##         1.79999995, 1.89999998, 2.00000000])
```

High-level autograd API

This allows for the automatic calculation and evaluation of the jacobian and hessian for a function defined used tensors.

```
def f(x, y):
  return 3*x + 1 + 2*y**2 + x*y
```

```
for x in [0.,1.]:
  for y in [0.,1.]:
    print("x =",x, "y = ",y)
    inputs = (torch.tensor([x]), torch.tensor([y]
    print(torch.autograd.functional.jacobian(f, i
```

```
## x = 0.0 y =  0.0
## (tensor([[3.]]), tensor([[0.]]))
##
## x = 0.0 y =  1.0
## (tensor([[4.]]), tensor([[4.]]))
##
## x = 1.0 y =  0.0
## (tensor([[3.]]), tensor([[1.]]))
##
```

```
inputs = (torch.tensor([0.]), torch.tensor([0.]))
torch.autograd.functional.hessian(f, inputs)
```

```
## ((tensor([[0.]]), tensor([[1.]])), (tensor([[1.]]),
```

```
inputs = (torch.tensor([1.]), torch.tensor([1.]))
torch.autograd.functional.hessian(f, inputs)
```

```
## ((tensor([[0.]]), tensor([[1.]])), (tensor([[1.]]),
```

Demo 1 - Linear Regression w/ PyTorch

A basic model

```
x = np.linspace(-math.pi, math.pi, 200)
y = np.sin(x)
```

```
lm = smf.ols("y~x+I(x**2)+I(x**3)", data=pd.DataFrame({"x": x, "y": y})).fit()
print(lm.summary())
```

```
##                             OLS Regression Results
## ==============================================================================
## Dep. Variable:                      y   R-squared:                       0.991
## Model:                            OLS   Adj. R-squared:                  0.991
## Method:                 Least Squares   F-statistic:                     7041.
## Date:                Tue, 29 Mar 2022   Prob (F-statistic):          2.96e-199
## Time:                        14:21:12   Log-Likelihood:                254.95
## No. Observations:                 200   AIC:                            -501.9
## Df Residuals:                     196   BIC:                            -488.7
## Df Model:                           3
## Covariance Type:            nonrobust
## ==============================================================================
##                  coef    std err          t      P>|t|      [0.025      0.975]
## ------------------------------------------------------------------------------
## Intercept   6.104e-17      0.007   8.42e-15      1.000      -0.014       0.014
## x              0.8546      0.007    128.977      0.000       0.842       0.868
## I(x ** 2)    4.93e-18      0.002   3.03e-15      1.000      -0.003       0.003
```

Making tensors

```
yt = torch.tensor(y)
Xt = torch.tensor(lm.model.exog)
bt = torch.randn((Xt.shape[1], 1), dtype=torch.float64, requires_grad=True)
```

```
yt_pred = (Xt @ bt).squeeze()
```

```
loss = (yt_pred - yt).pow(2).sum()
loss.item()
```

```
## 8060.669052389756
```

Gradient descent

Going back to our discussion of optimization and gradient descent awhile back - we can update our guess for `b` / `bt` by moving in the direction of the negative gradient. The step size is refered to as the learning rate which we will pick a relatively small value for.

```
learning_rate = 1e-6

loss.backward() # Compute the backward pass

with torch.no_grad():
  bt -= learning_rate * bt.grad # Make the step

  bt.grad = None # Reset the gradients
```

```
yt_pred = (Xt @ bt).squeeze()
loss = (yt_pred - yt).pow(2).sum()
loss.item()
```

```
## 7380.58278232608
```

Putting it together

```
yt = torch.tensor(y).unsqueeze(1)
Xt = torch.tensor(lm.model.exog)
bt = torch.randn((Xt.shape[1], 1), dtype=torch.float64, requires_grad=True)

learning_rate = 1e-5
for i in range(5000):

  yt_pred = Xt @ bt

  loss = (yt_pred - yt).pow(2).sum()
  if i % 500 == 0:
    print(i, loss.item())

  loss.backward()

  with torch.no_grad():
    bt -= learning_rate * bt.grad
    bt.grad = None
```

```
## 0 257680.8304254537
## 500 13.07780707986047
## 1000 2.4863342128971935
## 1500 1.1208859061116399
## 2000 0.9423484068960166
```

Comparing results

```
bt
```

```
## tensor([[ 4.79584652e-05],
##         [ 8.54550246e-01],
##         [-8.19655854e-06],
##         [-9.28168699e-02]], dtype=torch.float64, requires_grad=True)
```

```
lm.params
```

```
## Intercept    6.104058e-17
## x            8.545770e-01
## I(x ** 2)    4.929732e-18
## I(x ** 3)   -9.282065e-02
## dtype: float64
```

Demo 2 - Using a model

A sample model

```
class Model(torch.nn.Module):
    def __init__(self, beta):
        super().__init__()
        beta.requires_grad = True
        self.beta = torch.nn.Parameter(beta)

    def forward(self, X):
        return X @ self.beta

def training_loop(model, X, y, optimizer, n=1000):
    losses = []
    for i in range(n):
        y_pred = model(X)

        loss = (y_pred.squeeze() - y.squeeze()).pow(2).sum()
        loss.backward()

        optimizer.step()
        optimizer.zero_grad()

        losses.append(loss.item())

    return losses
```

Fitting

```
x = torch.linspace(-math.pi, math.pi, 200)
y = torch.sin(x)

X = torch.vstack((
  torch.ones_like(x),
  x,
  x**2,
  x**3
)).T

m = Model(beta = torch.zeros(4))
opt = torch.optim.SGD(m.parameters(), lr=1e-5)

losses = training_loop(m, X, y, opt, n=3000)
```

Results

```
m.beta
```

```
## Parameter containing:
## tensor([-8.40055758e-09,  8.52953434e-01,  2.83126012e-09, -9.25917700e-02],
##        requires_grad=True)
```

```
plt.figure(figsize=(8,6), layout="constrained")
plt.plot(losses)
plt.show()
```