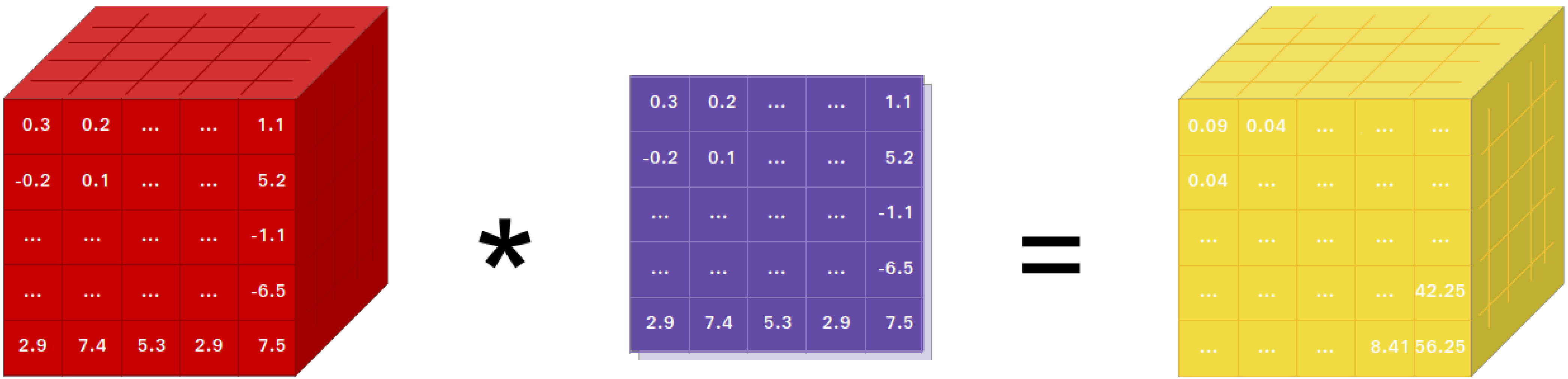
torch

Lecture 17

Dr. Colin Rundel

PyTorch

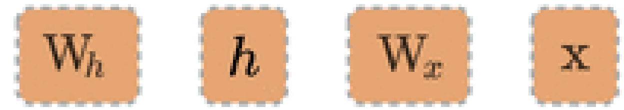
> PyTorch is a Python package that provides two high-level features:
>
> - Tensor computation (like NumPy) with strong GPU acceleration
> - Deep neural networks built on a tape-based autograd system




```python
import torch
torch.__version__
```

```
'2.6.0'
```

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)
```

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`, `jax`, etc.)

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

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

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

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

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

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

        [[0., 0.],
         [0., 0.]]])
```

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

```
<torch._C.Generator object at
0x30217f490>
```

```python
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]]]])
```

Constants

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

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

```
tensor(1)
```

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

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

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

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

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

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

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

```
ValueError: expected sequence of length 3 at dim 1 (got 2)
```

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

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

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

```
tensor([[True]])
```

Tensor Types

Data type	`dtype`	`type()`	Comment
32-bit float	`float32` or `float`	`FloatTensor`	Default float
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`	
64-bit integer (signed)	`int64` or `long`	`LongTensor`	Default integer
Boolean	`bool`	`BoolTensor`	

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.float1
```

```
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.float6
```

```
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([128], dtype=torch.int8)
```

```
RuntimeError: value cannot be converted to type
int8 without overflow
```

```
torch.tensor([128]).to(torch.int8)
```

```
tensor([-128], dtype=torch.int8)
```

```
torch.tensor([255]).to(torch.uint8)
```

```
tensor([255], dtype=torch.uint8)
```

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

```
tensor([44], dtype=torch.uint8)
```

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

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

```
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)
```

```
torch.tensor(1/3, dtype=torch.bfloat16)
```

```
tensor(0.33398438, dtype=torch.bfloat16)
```

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()
```

```
array([[1, 1],
       [1, 1]])
```

Math & Logic

Just like NumPy torch `tensor` objects support basic mathematical and logical operations with scalars and other tensors - torch provides implementations of most commonly needed mathematical functions.

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

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

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

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

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

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

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

```
tensor([[False, False],
        [ True,  True]])
```

```python
x = torch.rand(2,2)
torch.ones(2,2) @ x
```

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

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

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

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

```
tensor(0.64764357)
```

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

```
tensor(2.59057426)
```

```python
torch.min(x)
```

```
tensor(0.25475216)
```

Broadcasting

Like NumPy, cases where tensor dimensions do not match use the broadcasting heuristic.

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 such that a new tensor is not created but the original value(s) 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, 1.45181489]])
```

```
print(a)
```

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

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

```
tensor([[1.37521553, 1.33750737],
        [1.52136779, 1.45181489]])
```

```
print(a)
```

```
tensor([[1.37521553, 1.33750737],
        [1.52136779, 1.45181489]])
```

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.

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

```python
y
```

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

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

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

```python
y
```

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

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

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

```python
y.view(6)
```

```
RuntimeError: view size is not
compatible with input tensor's
size and stride (at least one
dimension spans across two
   contiguous subspaces). Use
.reshape(...) instead.
```

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

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

```python
y
```

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

```python
z
```

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

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])
```

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

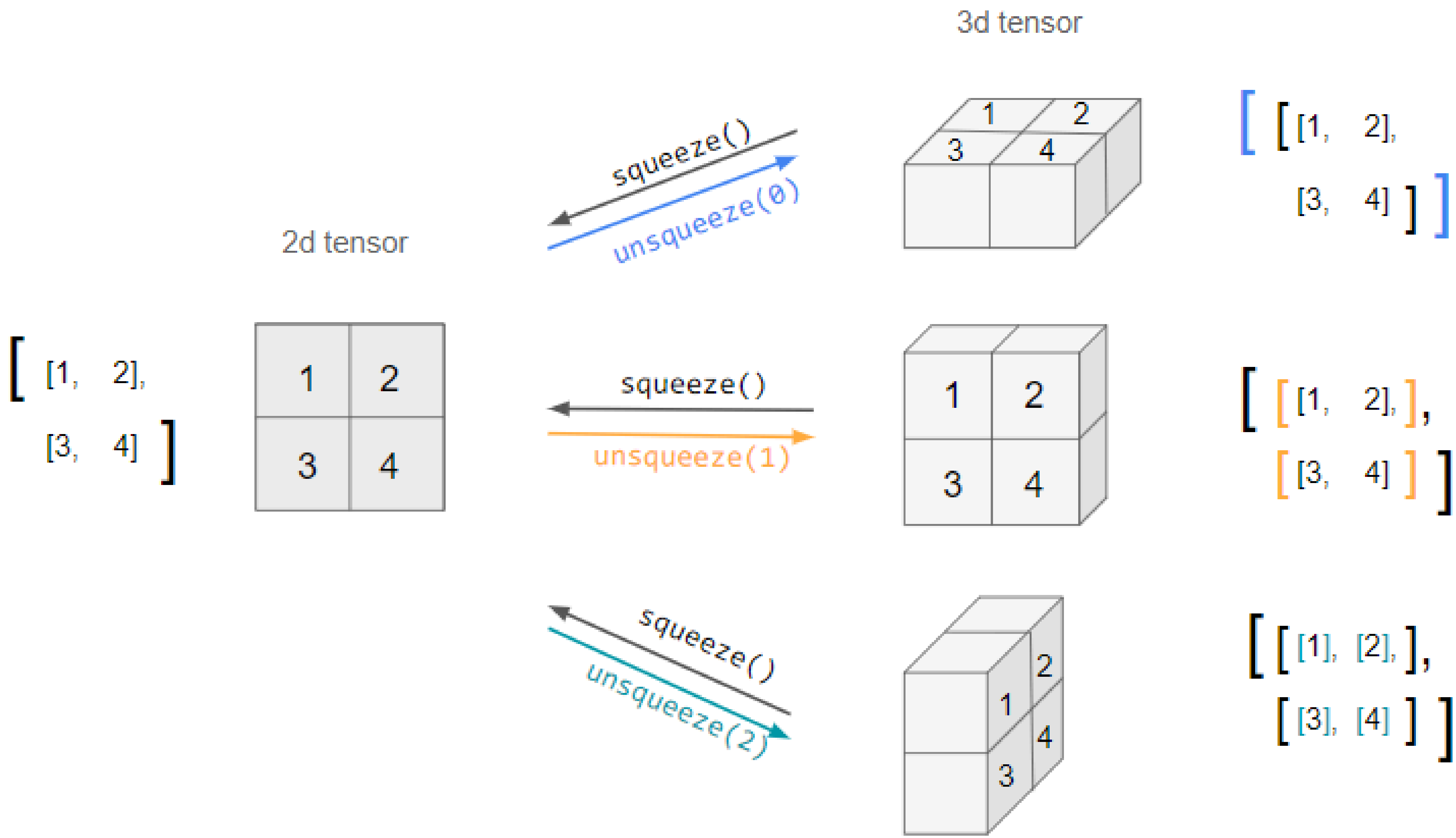
```
torch.Size([1, 3, 2])
```

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

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

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

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


3d tensor
2d tensor
squeeze()
unsqueeze(0)
squeeze()
unsqueeze(1)
squeeze()
unsqueeze(2)
[[1, 2],
[3, 4]]
[[[1, 2],
[3, 4]]]
[[[1, 2]],
[[3, 4]]]
[[[1], [2]],
[[3], [4]]]
1 2 3 4

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 the tensor or the `enable_grad()` context manager used (via `with`).

```
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, 1.29999995, 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, 5.89999962, 6.19999981, 6.50000000, 6.80000019, 7.10000038,
        7.39999962, 7.69999981, 8.00000000], grad_fn=<AddBackward0>)
```

Computational graph

Basics of the computation graph can be explored via the `next_functions` attribute

```
y.grad_fn
```

```
<AddBackward0 object at 0x30981f730>
```

```
y.grad_fn.next_functions
```

```
((<MulBackward0 object at 0x11ff295a0>, 0), (None, 0))
```

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

```
((<AccumulateGrad object at 0x30981fa90>, 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 *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
```

```
x.grad
```

```
tensor([3., 3., 3., 3., 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, 1.29999995, 1.39999998, 1.50000000, 1.60000002, 1.70000005,
        1.79999995, 1.89999998, 2.00000000])
```

In context you can interpret `x.grad` and `m.grad` as the gradient of `y` with respect to `x` or `m` respectively.

High-level autograd API

This allows for the automatic calculation and evaluation of the jacobian and hessian for a function defined using 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, inputs),"\n")
```

```
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.]]))

x = 1.0 y =  1.0
(tensor([[4.]]), tensor([[5.]]))
```

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

```
((tensor([[0.]]), tensor([[1.]])), (tensor([[1.]]), tensor([[4.]])))
```

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

```
((tensor([[0.]]), tensor([[1.]])), (tensor([[1.]]), tensor([[4.]])))
```

Demo 1 - Linear Regression w/ PyTorch

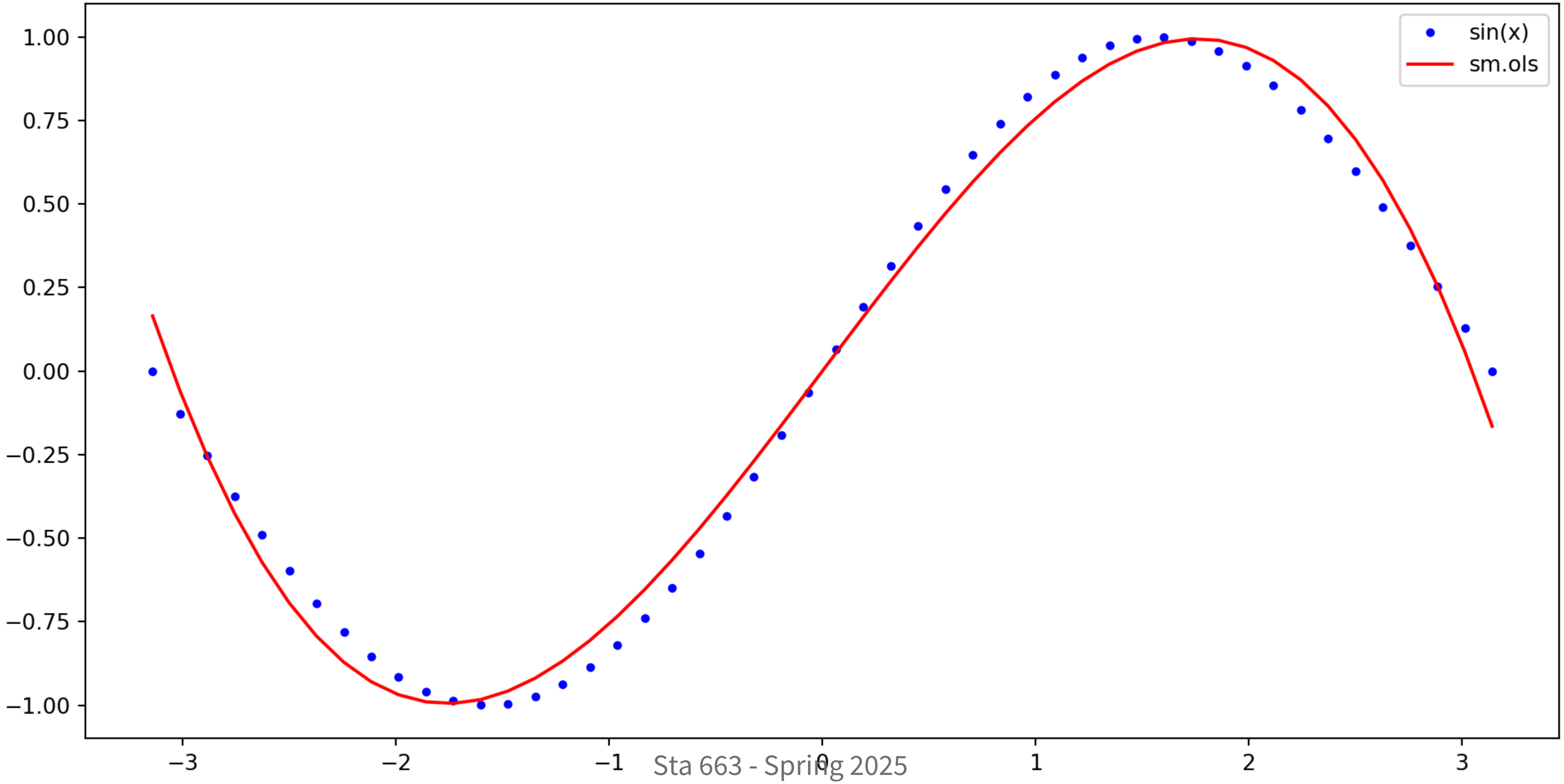
A basic model

```python
x = np.linspace(-math.pi, math.pi, 50)
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:                        0.990
Model:                            OLS   Adj. R-
squared:                   0.989
Method:                 Least Squares   F-
statistic:                      1455.
Date:                Tue, 18 Mar 2025   Prob
(F-statistic):           1.44e-45
Time:                        17:20:44   Log-
Likelihood:                60.967
No. Observations:                  50   AIC:
-113.9
Df Residuals:                      46   BIC:
-106.3
Df Model:                           3
```

Predictions

```
plt.figure(figsize=(10,5), layout="constrained")
plt.plot(x, y, ".b", label="sin(x)")
plt.plot(x, lm.predict(), "-r", label="sm.ols")
plt.legend()
plt.show()
```



Making tensors

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

```python
yt
```

```
tensor([-1.22464680e-16, -1.27877162e-01,
-2.53654584e-01, -3.75267005e-01,
        -4.90717552e-01, -5.98110530e-01,
-6.95682551e-01, -7.81831482e-01,
        -8.55142763e-01, -9.14412623e-01,
-9.58667853e-01, -9.87181783e-01,
        -9.99486216e-01, -9.95379113e-01,
-9.74927912e-01, -9.38468422e-01,
        -8.86599306e-01, -8.20172255e-01,
-7.40277997e-01, -6.48228395e-01,
        -5.45534901e-01, -4.33883739e-01,
-3.15108218e-01, -1.91158629e-01,
        -6.40702200e-02,  6.40702200e-02,
1.91158629e-01,  3.15108218e-01,
         4.33883739e-01,  5.45534901e-01,
```

```python
Xt
```

```
tensor([[ 1.00000000e+00,
-3.14159265e+00,  9.86960440e+00,
-3.10062767e+01],
        [ 1.00000000e+00,
-3.01336438e+00,  9.08036490e+00,
-2.73624482e+01],
        [ 1.00000000e+00,
-2.88513611e+00,  8.32401038e+00,
-2.40159029e+01],
        [ 1.00000000e+00,
-2.75690784e+00,  7.60054083e+00,
-2.09539906e+01],
        [ 1.00000000e+00,
-2.62867957e+00,  6.90995627e+00,
-1.81640609e+01],
```

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

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

```
2119.277704016523
```

Gradient descent

```
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()
```

```
2069.4881821807053
```

Putting it together

```python
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(f"Step: {i},\tloss: {loss.item()}")

  loss.backward()

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

Putting it together

```
Step: 0,    loss: 70161.1580804254
Step: 500,  loss: 14.791178300540242
Step: 1000, loss: 8.825181658035252
Step: 1500, loss: 5.311942717260374
Step: 2000, loss: 3.2416251317783518
Step: 2500, loss: 2.020671792951764
Step: 3000, loss: 1.300022038356929
Step: 3500, loss: 0.8742816442183533
Step: 4000, loss: 0.6225166364100523
Step: 4500, loss: 0.473473387453477
```

```
print(bt)
```

```
tensor([[ 0.03143311],
        [ 0.78484316],
        [-0.00520945],
        [-0.08260584]], dtype=torch.float64,
requires_grad=True)
```

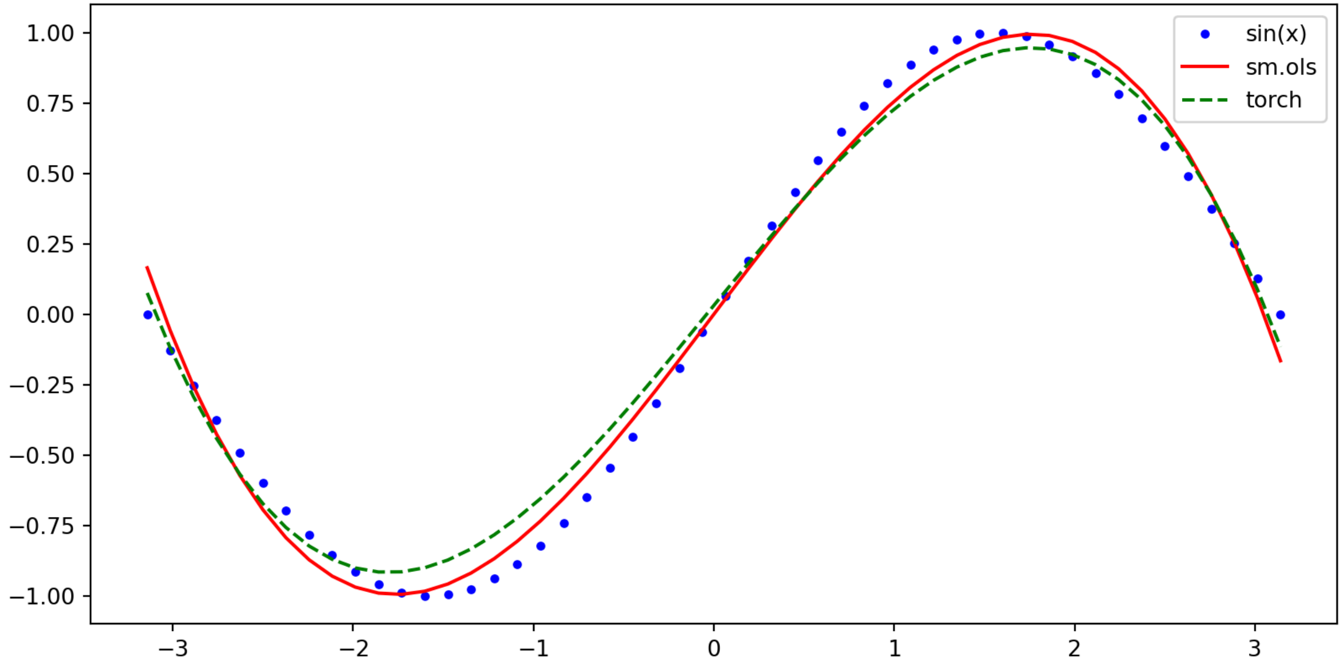
Comparing results

```
lm.params
```

```
Intercept    -7.958044e-17
x             8.476289e-01
I(x ** 2)     3.691708e-17
I(x ** 3)    -9.120167e-02
dtype: float64
```

```
bt
```

```
tensor([[ 0.03143311],
        [ 0.78484316],
        [-0.00520945],
        [-0.08260584]],
dtype=torch.float64, requires_grad=True)
```



Demo 2 - Using a torch model

A sample model

```python
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())
```

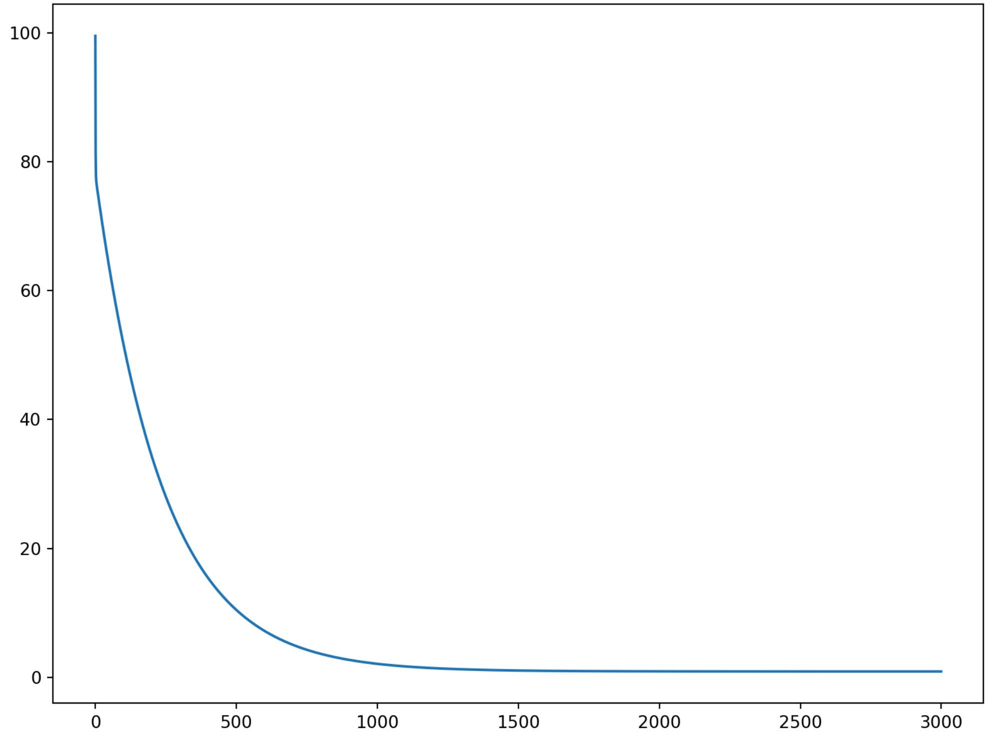
Fitting

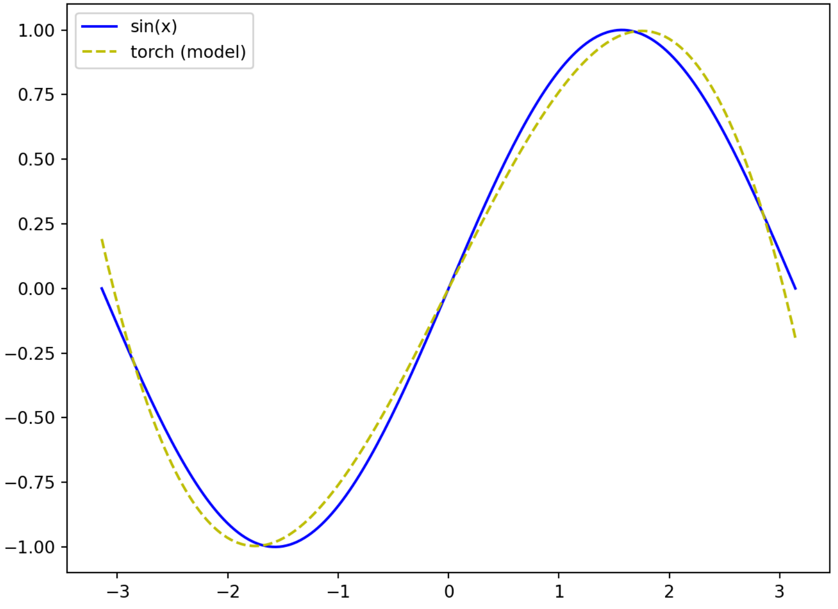
```
x = torch.linspace(-math.pi, math.pi, 200)
y = torch.sin(x)

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([-4.07514189e-10,  8.52953434e-01,  1.22972355e-10, -9.25917700e-02],
       requires_grad=True)
```



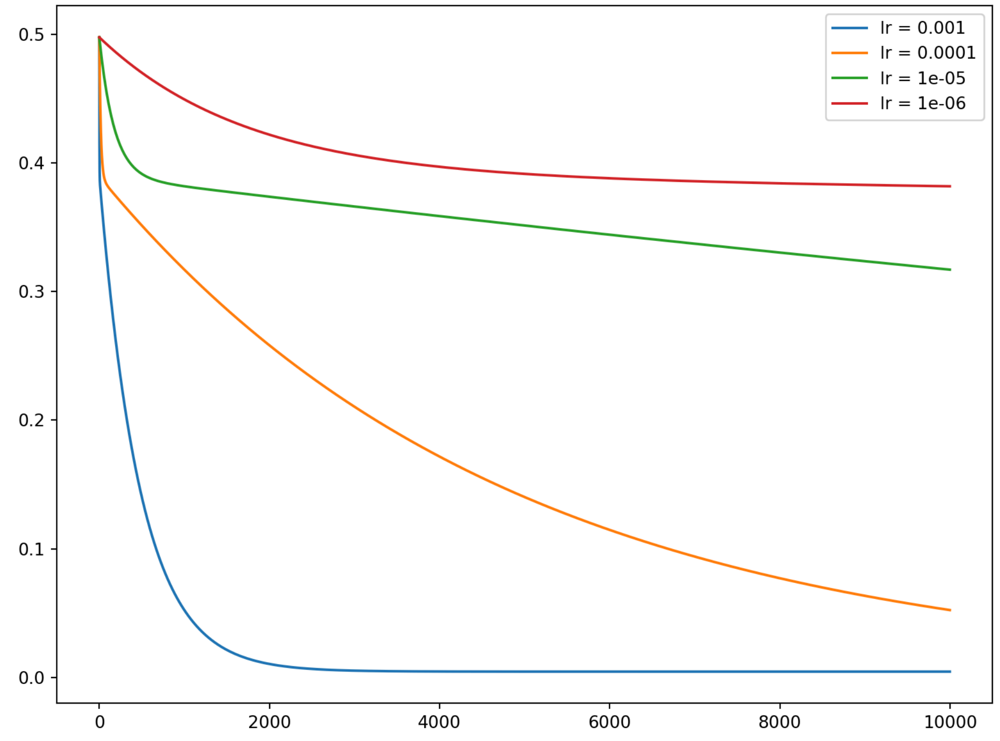


An all-in-one model

```
class Model(torch.nn.Module):
    def __init__(self, X, y, beta=None):
        super().__init__()
        self.X = X
        self.y = y
        if beta is None:
          beta = torch.zeros(X.shape[1])
        beta.requires_grad = True
        self.beta = torch.nn.Parameter(beta)

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

    def fit(self, opt, n=1000, loss_fn = torch.nn.MSELoss()):
      losses = []
      for i in range(n):
          loss = loss_fn(self(self.X).squeeze(), self.y.squeeze())
          loss.backward()
          opt.step()
          opt.zero_grad()
          losses.append(loss.item())
```

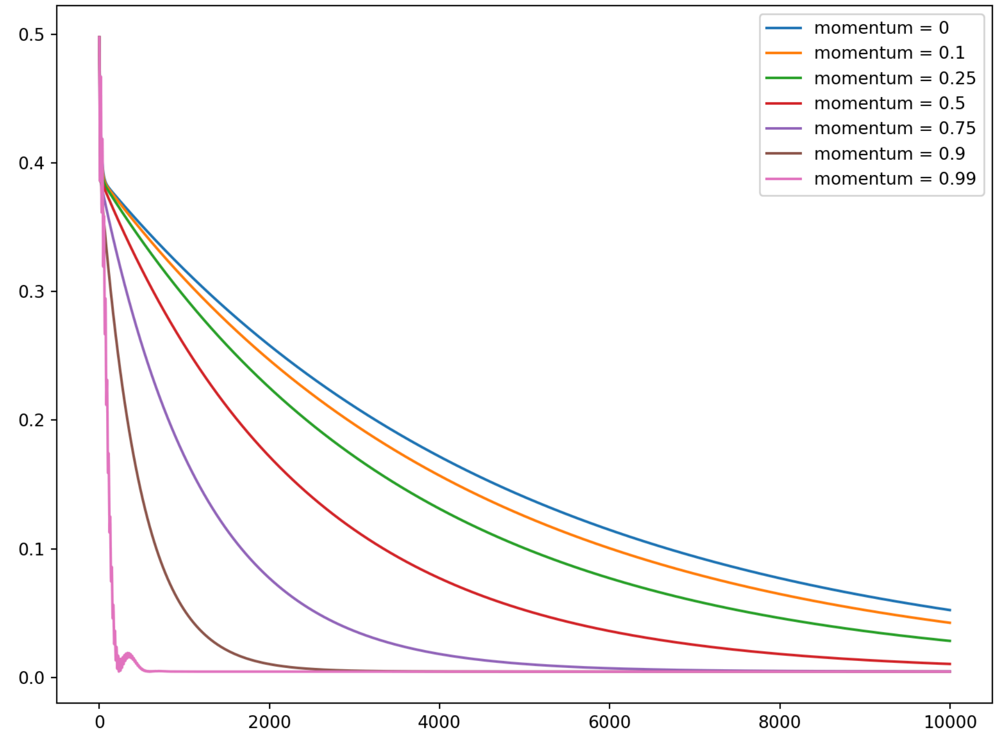
Learning rate and convergence

```
plt.figure(figsize=(8,6), layout="cons

for lr in [1e-3, 1e-4, 1e-5, 1e-6]:
  m = Model(X, y)
  opt = torch.optim.SGD(m.parameters()
  losses = m.fit(opt, n=10000)

  plt.plot(losses, label=f"lr = {lr}")

plt.legend()
plt.show()
```



Momentum and convergence

```
plt.figure(figsize=(8,6), layout="cons
     
for mt in [0, 0.1, 0.25, 0.5, 0.75, 0.
  m = Model(X, y)
  opt = torch.optim.SGD(
    m.parameters(), 
    lr = 1e-4, 
    momentum = mt
  )
  losses = m.fit(opt, n=10000)
  
  plt.plot(losses, label=f"momentum = 

plt.legend()
plt.show()
```



Optimizers and convergence

```
plt.figure(figsize=(8,6), layout="cons

opts = (torch.optim.SGD,
        torch.optim.Adam,
        torch.optim.Adagrad)

for opt_fn in opts:
  m = Model(X, y)
  opt = opt_fn(m.parameters(), lr=1e-4
  losses = m.fit(opt, n=10000)

  plt.plot(losses, label=f"opt = {opt_

plt.legend()
plt.show()
```