

torch

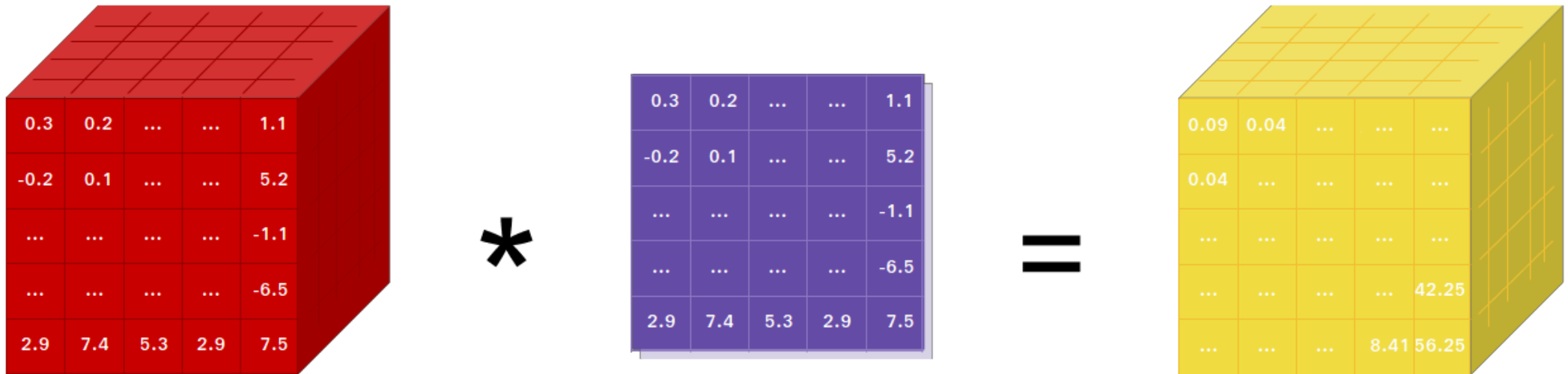
Lecture 17

Dr. Colin Rundel

PyTorch

PyTorch is a Python package that provides two high-level features:

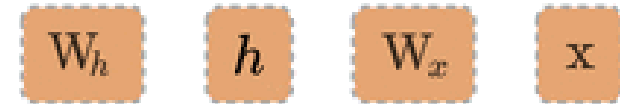
- Tensor computation (like NumPy) with strong GPU acceleration
- Deep neural networks built on a tape-based autograd system



```
1 import torch
2 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. They behave in much the same way as the other array libraries we've seen so far (`numpy`, `jax`, etc.)

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

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

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

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

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

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

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

```
<torch._C.Generator object at  
0x15d5cb410>
```

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

```
tensor([[[[0.02898, 0.40190],  
          [0.25984, 0.36664]],  
        [[0.05830, 0.70064],  
          [0.05180, 0.46814]]],  
        [[0.67381, 0.33146],  
          [0.78371, 0.56306]],  
        [[0.77485, 0.82080],  
          [0.27928, 0.68171]]]])
```

Constants

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

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

```
tensor(1)
```

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

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

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

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

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

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

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

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

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

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

```
1 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	Default integer
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	
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.

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

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

```
1 a.dtype
```

```
torch.int64
```

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

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

```
1 b.dtype
```

```
torch.float16
```

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

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

```
1 c.dtype
```

```
torch.float32
```

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

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

```
1 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).

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

RuntimeError: value cannot be converted to type int8 without overflow

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

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

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

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

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

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

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

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

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

tensor(0.33325195, dtype=torch.float16)

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

tensor(0.33333334)

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

tensor(0.33333333, dtype=torch.float64)

```
1 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
tensor(0.64764357)
```

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

```
tensor(2.59057426)
```

```
1 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:

```
1 a = torch.rand(4, 3, 2)
2 b = torch.rand(3, 2)
3 c = torch.rand(2, 3)
4 d = torch.rand(0)
5 e = torch.rand(3, 1)
6 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.

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

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

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

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

```
1 print(a)
```

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

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

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

```
1 print(a)
```

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

Inplace arithmetic

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

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

```
1 a+b
```

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

```
1 print(a)
```

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

```
1 print(b)
```

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

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

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

```
1 print(a)
```

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

```
1 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.

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

```
1 y
```

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

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

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

```
1 y
```

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

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

```
1 x.view(6)
```

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

```
1 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.

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

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

```
1 y
```

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

```
1 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.

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

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

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

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

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

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

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

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

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

```
1 x = torch.zeros(3,2)
```

```
1 x.unsqueeze(0).shape
```

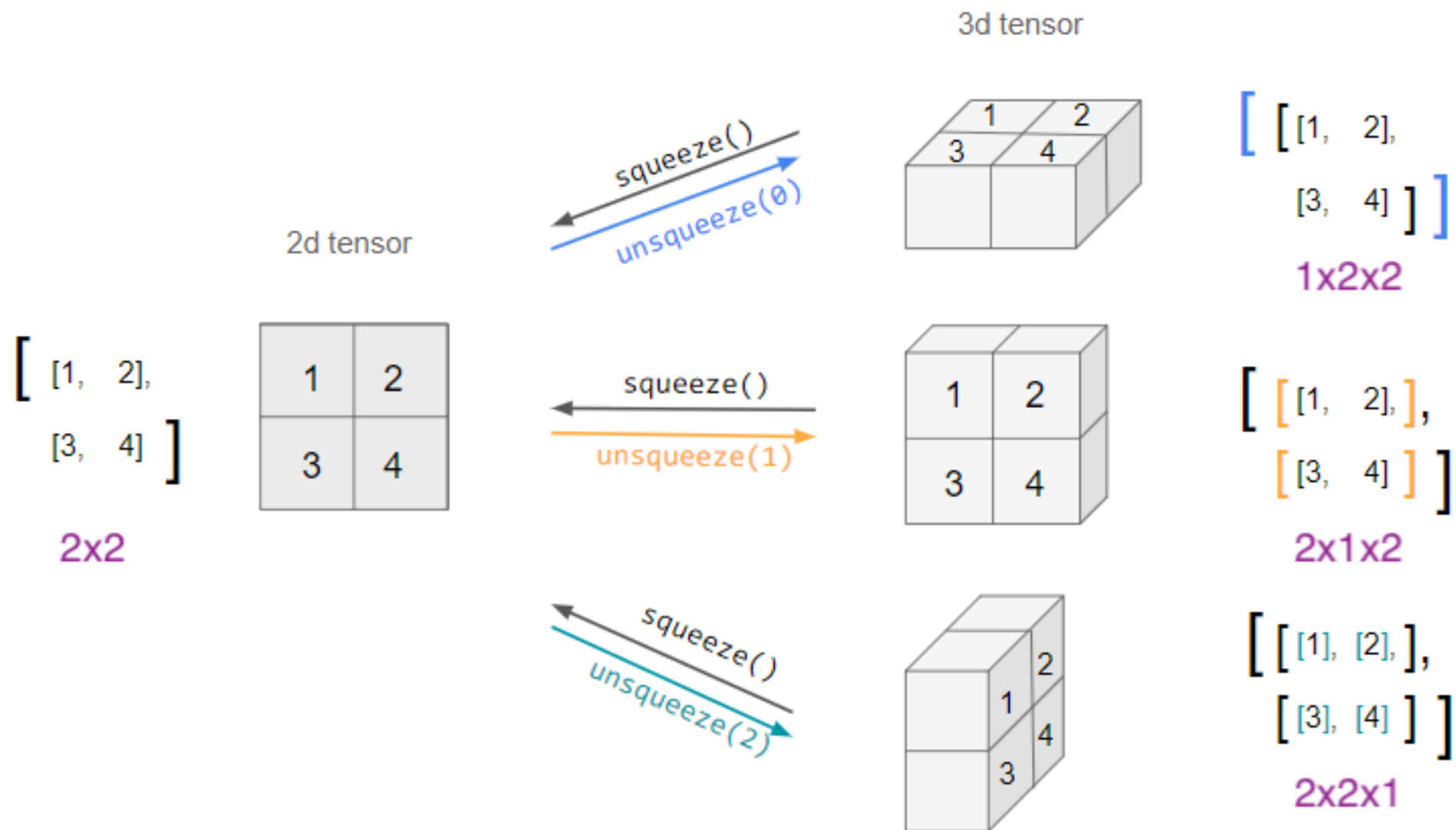
```
torch.Size([1, 3, 2])
```

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

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

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

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

Exercise 2

Given the following tensors,

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

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

```
1 y = 3*x + 2
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

```
1 y.grad_fn
```

```
<AddBackward0 object at 0x1652f4250>
```

```
1 y.grad_fn.next_functions
```

```
((<MulBackward0 object at 0x165b93850>, 0), (None, 0))
```

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

```
((<AccumulateGrad object at 0x165d2c610>, 0), (None, 0))
```

```
1 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.

```
1 out = y.sum()  
2 out.backward()  
3 out
```

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

```
1 y.grad
```

```
1 x.grad
```

```
tensor([3., 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

```
1 n = 21
2 x = torch.linspace(0, 2, steps=n, requires_grad=True)
3 m = torch.rand(n, requires_grad=True)
4
5 y = m*x + 2
6
7 y.backward(torch.ones(n))
```

```
1 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])
```

```
1 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])
```

High-level autograd API

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

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

```
1 for x in [0.,1.]:  
2     for y in [0.,1.]:  
3         print("x =",x, "y = ",y)  
4         inputs = (torch.tensor([x]), torch.tensor([y]))  
5         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.])))
```



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

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

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

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

Demo 1 - Linear Regression w/ PyTorch

A basic model

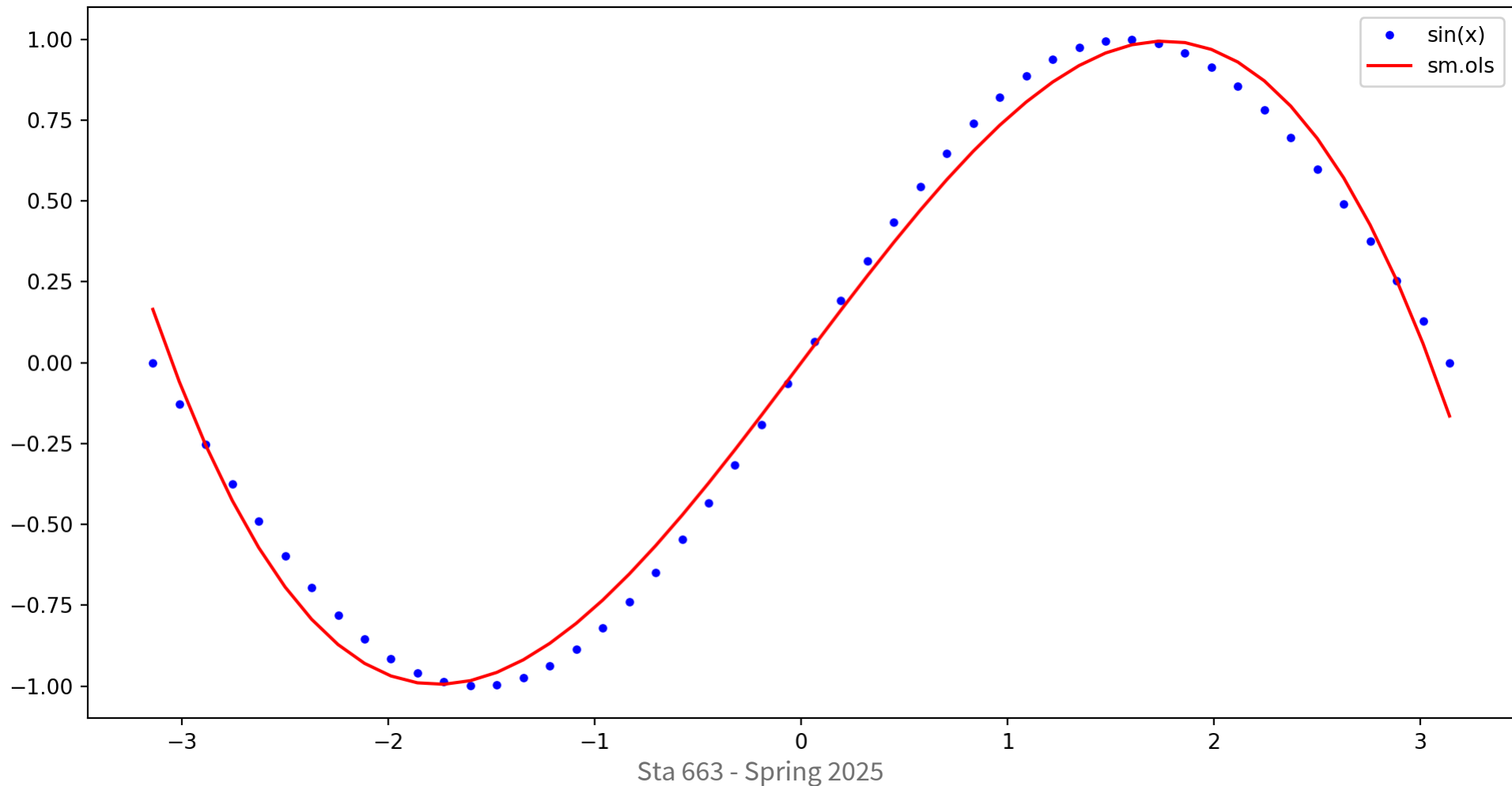
```
1 x = np.linspace(-math.pi, math.pi, 50)
2 y = np.sin(x)
3
4 lm = smf.ols(
5     "y~x+I(x**2)+I(x**3)",
6     data=pd.DataFrame({"x": x, "y": y})
7 ).fit()
8
9 print(lm.summary())
```

OLS Regression Results						
=====						
Dep. Variable:	y	R-squared:	0.990			
Model:	OLS	Adj. R-squared:	0.989			
Method:	Least Squares	F-statistic:	1455.			
Date:	Wed, 19 Mar 2025	Prob (F-statistic):	1.44e-45			
Time:	09:16:02	Log-Likelihood:	60.967			
No. Observations:	50	AIC:	-113.9			
Df Residuals:	46	BIC:	-106.3			
Df Model:	3					
Covariance Type:	nonrobust					
=====						
	coef	std err	t	P> t	[0.025	0.975]

Intercept	-7.958e-17	0.016	-5.03e-15	1.000	-0.032	0.032
x	0.8476	0.014	59.444	0.000	0.819	0.876
I(x ** 2)	3.692e-17	0.003	1.07e-14	1.000	-0.007	0.007
I(x ** 3)	-0.0912	0.002	-42.977	0.000	-0.095	-0.087
=====						

Predictions

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



Making tensors

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

```
1 yt.shape
```

```
torch.Size([50])
```

```
1 Xt.shape
```

```
torch.Size([50, 4])
```

```
1 bt.shape
```

```
torch.Size([4, 1])
```

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

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

```
2119.277704016523
```

Gradient descent

```
1 learning_rate = 1e-6
2
3 loss.backward() # Compute the backward pass
4
5 with torch.no_grad():
6     bt -= learning_rate * bt.grad # Make the step
7
8     bt.grad = None # Reset the gradients
```

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

2069.4881821807053

Putting it together

```
1 yt = torch.tensor(y).unsqueeze(1)
2 Xt = torch.tensor(lm.model.exog)
3 bt = torch.randn((Xt.shape[1], 1), dtype=torch.float64, requires_grad=True)
4
5 learning_rate = 1e-5
6 for i in range(5001):
7
8     yt_pred = Xt @ bt
9
10    loss = (yt_pred - yt).pow(2).sum()
11    if i % 500 == 0:
12        print(f"Step: {i}, \tloss: {loss.item()}")
13
14    loss.backward()
15
16    with torch.no_grad():
17        bt -= learning_rate * bt.grad
18        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
Step: 5000,   loss: 0.38513809895450724
```

```
1 print(bt)
```

```
tensor([[ 0.03141952],
        [ 0.78487683],
        [-0.00520719],
        [-0.08261045]], dtype=torch.float64,
requires_grad=True)
```

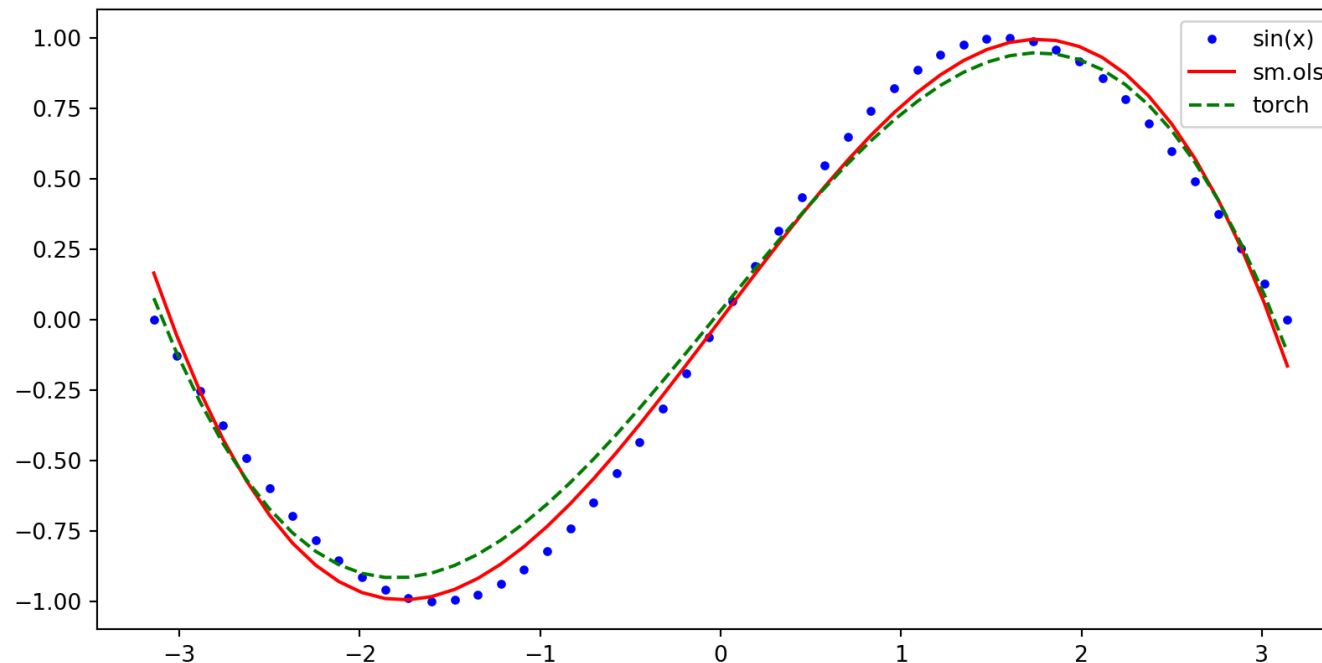

Comparing results

1 lm.params

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

1 bt

```
tensor([[ 0.03141952],  
        [ 0.78487683],  
        [-0.00520719],  
        [-0.08261045]],  
        dtype=torch.float64, requires_grad=True)
```



Demo 2 - Using a torch model

A simple model

```
1 class Model(torch.nn.Module):
2     def __init__(self, beta):
3         super().__init__()
4         beta.requires_grad = True
5         self.beta = torch.nn.Parameter(beta)
6
7     def forward(self, X):
8         return X @ self.beta
9
10 def training_loop(model, X, y, optimizer, n=1000):
11     losses = []
12     for i in range(n):
13         y_pred = model(X)
14
15         loss = (y_pred.squeeze() - y.squeeze()).pow(2).sum()
16         loss.backward()
17
18         optimizer.step()
19         optimizer.zero_grad()
20
21         losses.append(loss.item())
22
23     return losses
```

Fitting

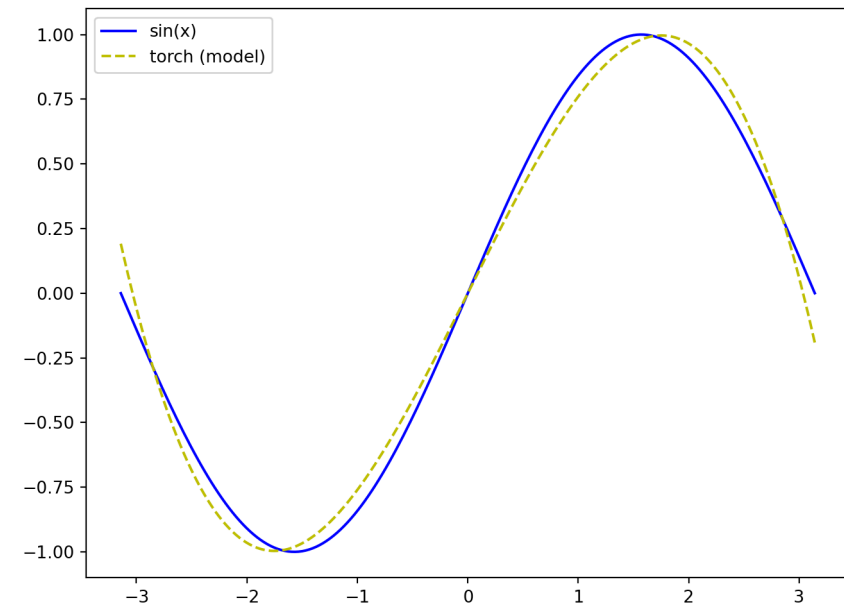
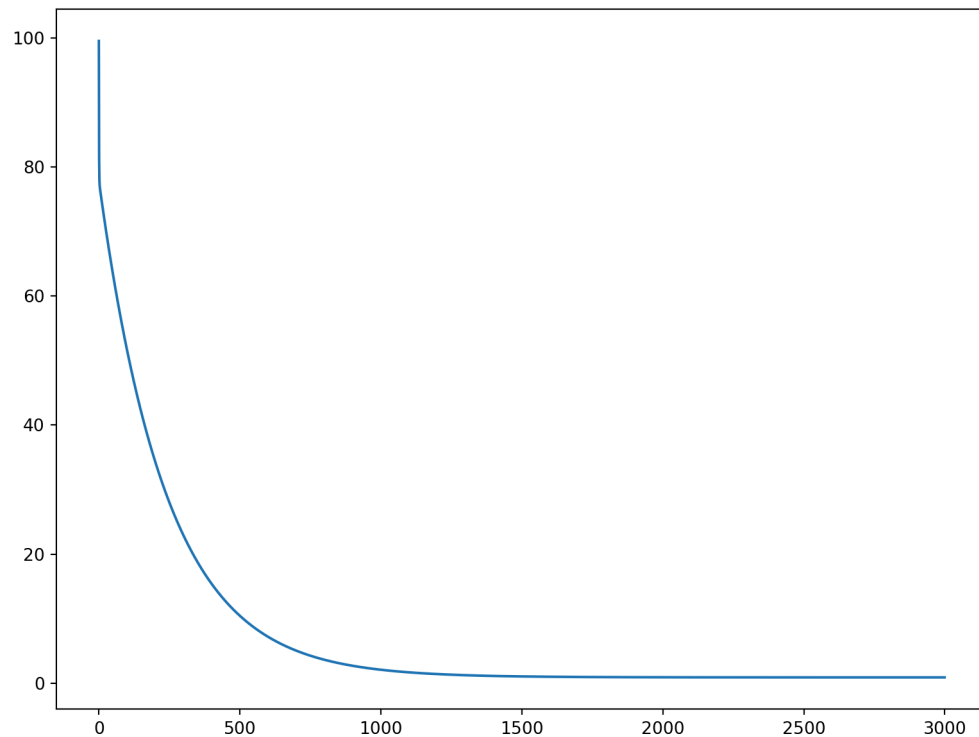
```
1 x = torch.linspace(-math.pi, math.pi, 200)
2 y = torch.sin(x)
3
4 X = torch.vstack((
5     torch.ones_like(x),
6     x,
7     x**2,
8     x**3
9 )) .T
10
11 m = Model(beta = torch.zeros(4))
12 opt = torch.optim.SGD(m.parameters(), lr=1e-5)
13
14 losses = training_loop(m, X, y, opt, n=3000)
```

Results

1 m.beta

Parameter containing:

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

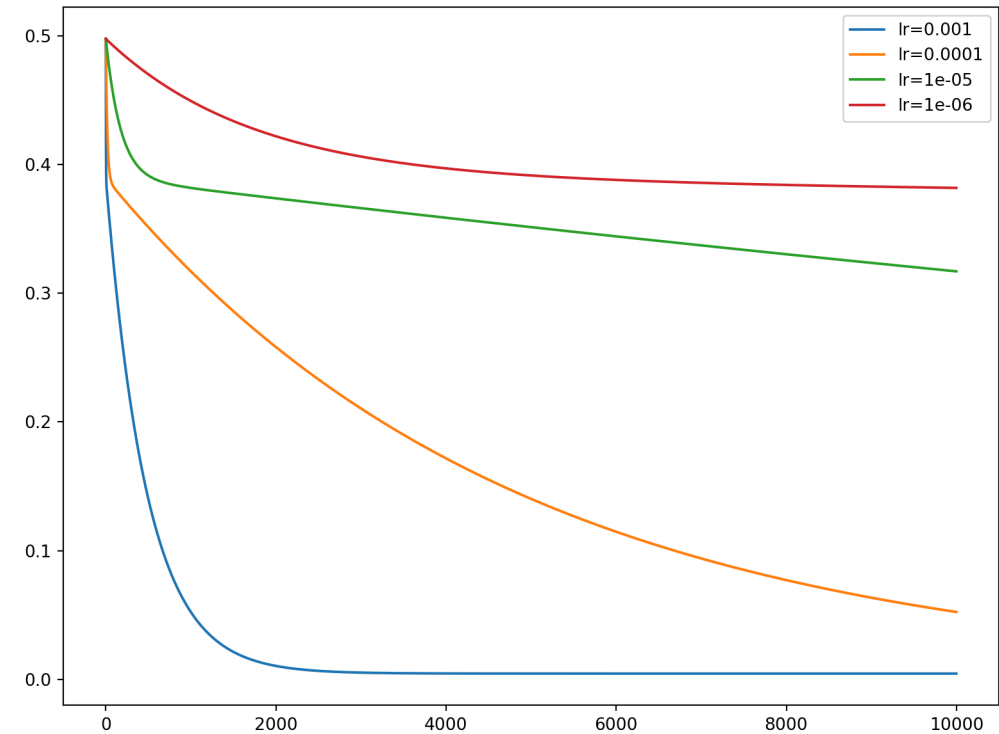


An all-in-one model

```
1 class Model(torch.nn.Module):
2     def __init__(self, X, y, beta=None):
3         super().__init__()
4         self.X = X
5         self.y = y
6         if beta is None:
7             beta = torch.zeros(X.shape[1])
8             beta.requires_grad = True
9             self.beta = torch.nn.Parameter(beta)
10
11     def forward(self, X):
12         return X @ self.beta
13
14     def fit(self, opt, n=1000, loss_fn = torch.nn.MSELoss()):
15         losses = []
16         for i in range(n):
17             loss = loss_fn(self.forward(self.X).squeeze(), self.y.squeeze())
18             loss.backward()
19             opt.step()
20             opt.zero_grad()
21             losses.append(loss.item())
```

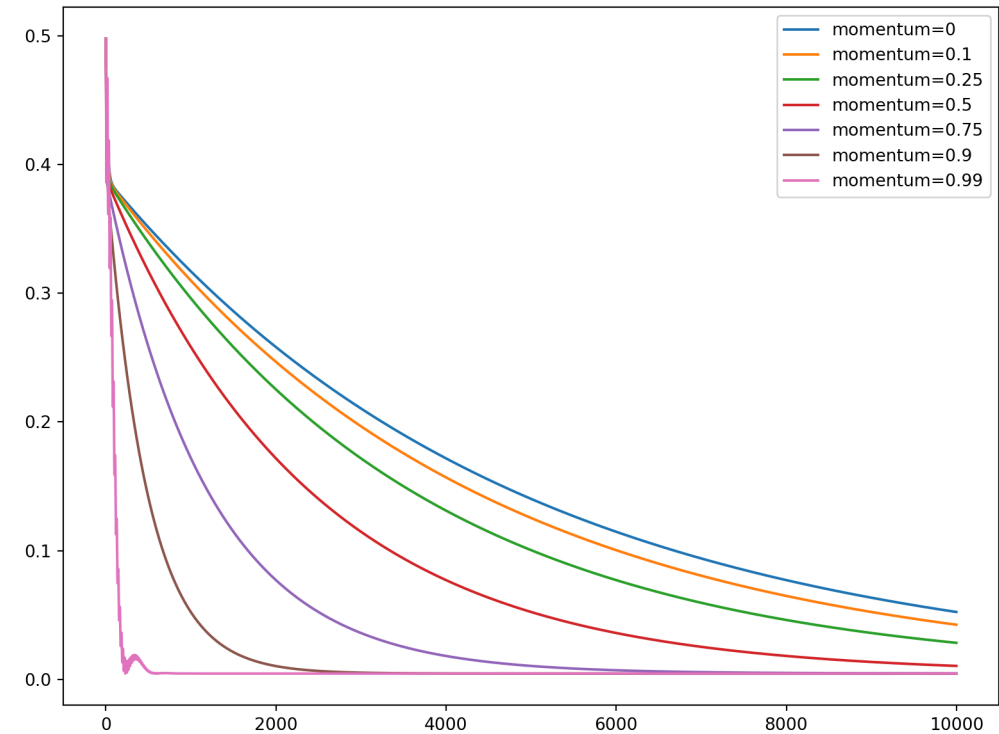
Learning rate and convergence

```
1 plt.figure(figsize=(8,6), layout="constrained")
2
3 for lr in [1e-3, 1e-4, 1e-5, 1e-6]:
4     m = Model(X, y)
5     opt = torch.optim.SGD(m.parameters())
6     losses = m.fit(opt, n=10000)
7
8     plt.plot(losses, label=f"{lr}")
9
10 plt.legend()
11 plt.show()
```



Momentum and convergence

```
1 plt.figure(figsize=(8,6), layout="constrained")
2
3 for momentum in [0, 0.1, 0.25, 0.5, 0.75, 0.9, 0.99]:
4     m = Model(X, y)
5     opt = torch.optim.SGD(
6         m.parameters(),
7         lr = 1e-4,
8         momentum = momentum
9     )
10    losses = m.fit(opt, n=10000)
11
12    plt.plot(losses, label=f"{momentum=}")
13
14 plt.legend()
15 plt.show()
```



Optimizers and convergence

```
1 plt.figure(figsize=(8,6), layout="constrained")
2
3 opts = (torch.optim.SGD,
4         torch.optim.Adam,
5         torch.optim.Adagrad)
6
7 for opt_fn in opts:
8     m = Model(X, y)
9     opt = opt_fn(m.parameters(), lr=1e-4)
10    losses = m.fit(opt, n=10000)
11
12    plt.plot(losses, label=f"{opt_fn}")
13
14 plt.legend()
15 plt.show()
```

