CS440/ECE448 Lecture 10: Pytorch

Mark Hasegawa-Johnson

2/2023

Outline

- Review: linear regression
- Training a linear regression model using numpy
- The same example with pytorch
- torch.nn module: using predefined neural net components

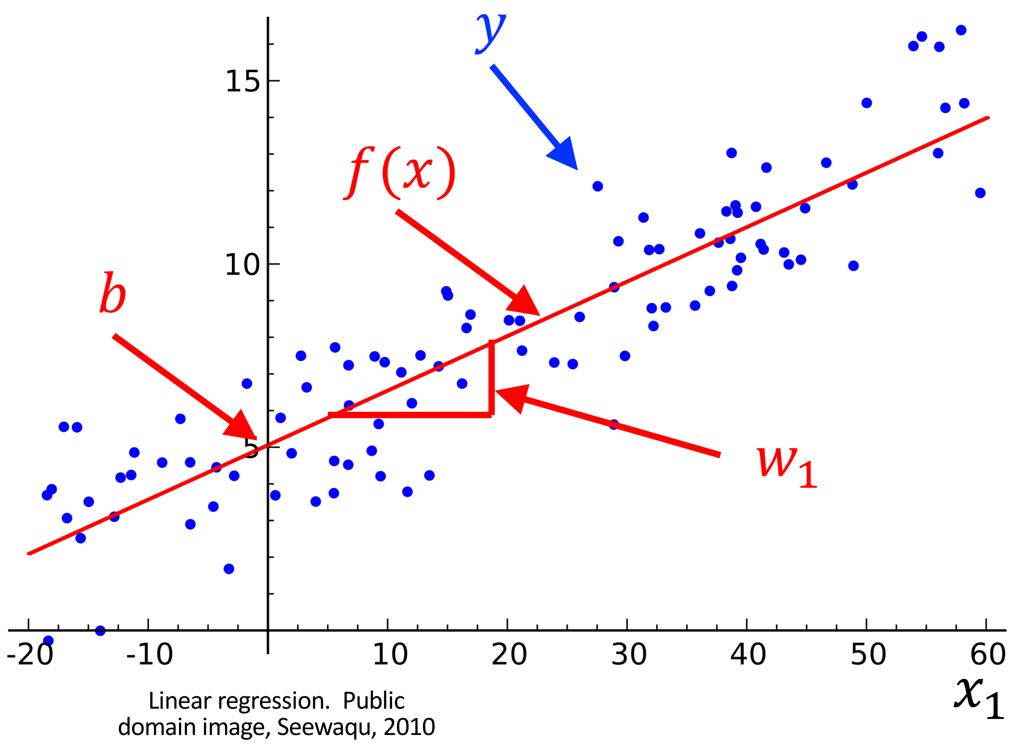
Linear regression

Linear regression is used to estimate a real-valued target variable, y, using a linear combination of real-valued input variables:

$$f(x) = w@x = b + \sum_{j=1}^{D} w_j x_j$$

… so that …

$$f(x) \approx y$$



Linear regression. Public domain image, Seewaqu, 2010

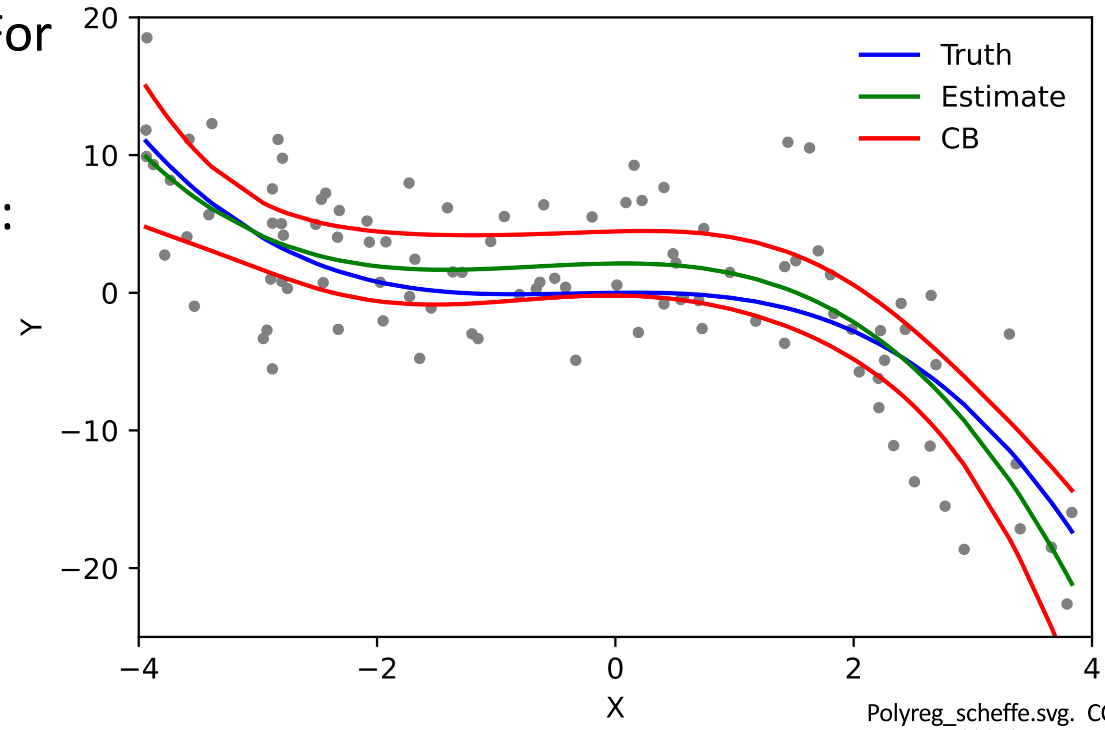
Polynomial regression = multivariate linear regression

We can use linear regression to solve nonlinear regression problems by simply augmenting the features. For example, suppose we start with a scalar variable, x, but suppose we expand it to four variables like this:

$$z = \begin{bmatrix} x \\ x^2 \\ x^3 \\ 1 \end{bmatrix}$$

Then

$$f(x) = w@z$$
$$= b + w_1 x + w_2 x^2 + w_3 x^3$$



Polyreg_scheffe.svg. CC-BY 3.0, Skbkekas, 2009

Minimizing the MSE

Our goal is to find the coefficients $w = [w_0, \dots, w_{D-1}, b]^T$ that minimize the MSE loss function:

$$\mathcal{L} = \frac{1}{n}\sum_{i=0}^{n-1}(x_i@w - y_i)^2$$

The gradient descent algorithm

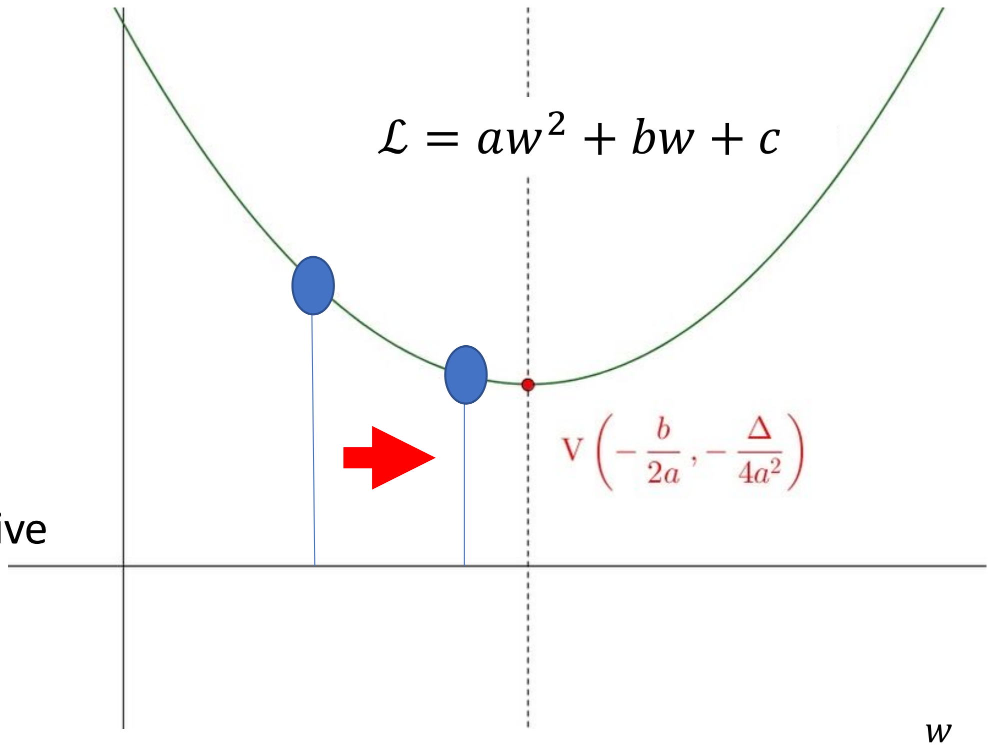
- Start from a random initial value of $\vec{w}$.
- Calculate the derivative of MSE with respect to $\vec{w}$:

$$\nabla_w \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_0} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{D-1}} \\ \frac{\partial \mathcal{L}}{\partial b} \end{bmatrix}$$

- Take a step "downhill" (in the direction of the negative gradient

$$w \leftarrow w - \frac{\eta}{2} \nabla_w \mathcal{L}$$

...where η is a constant called the "learning rate," that determines how big of a step you take. Usually, you need to adjust η in order to get optimum performance on a dev set, but often $\eta \approx 0.001$.


$\mathcal{L} = aw^2 + bw + c$
$V\left(-\frac{b}{2a}, -\frac{\Delta}{4a^2}\right)$
w

Gradient descent

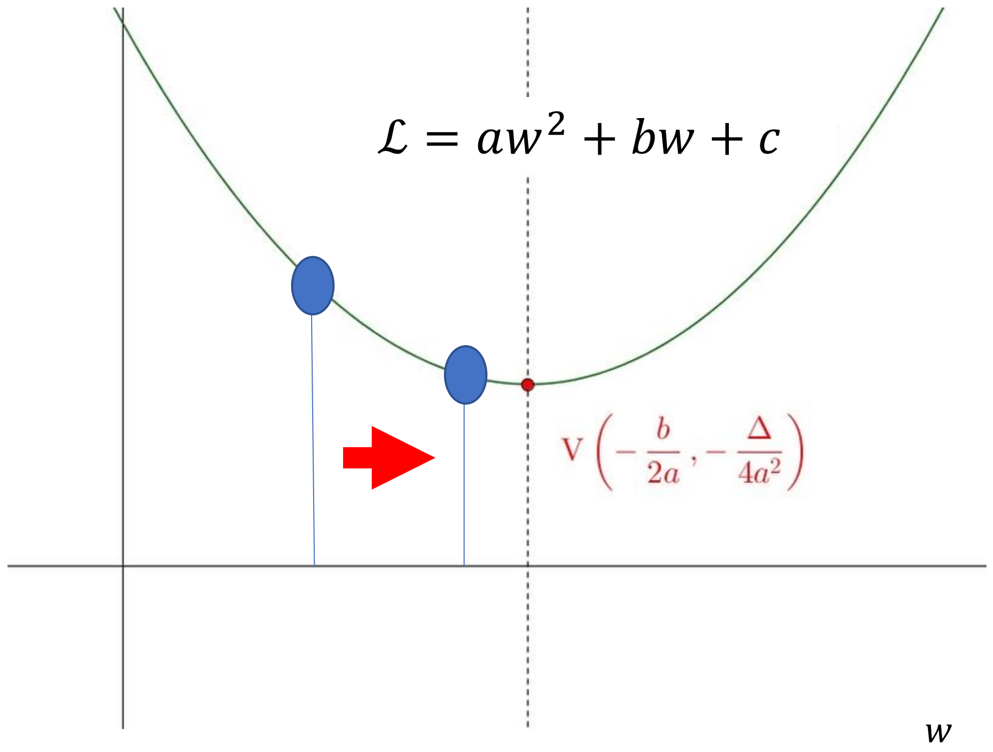
$$\mathcal{L} = \frac{1}{n}\sum_{i=0}^{n-1} {\epsilon_i}^2 = \frac{1}{n}\sum_{i=0}^{n-1} (x_i@w - y_i)^2$$

If we differentiate that, we discover that:

$$\nabla_w \mathcal{L} = \frac{2}{n}\sum_{i=1}^{n} \epsilon_i x_i$$

…so the gradient descent algorithm is:

$$w \leftarrow w - \frac{\eta}{n}\sum_{i=0}^{n-1} \epsilon_i x_i$$


$\mathcal{L} = aw^2 + bw + c$
$V\left(-\frac{b}{2a}, -\frac{\Delta}{4a^2}\right)$
w

Outline

- Review: linear regression
- Training a linear regression model using numpy
- The same example with pytorch
- torch.nn module: using predefined neural net components

Running example: neural net regression

- For example, suppose $y = \sin(x)$
- Suppose that the network can only model functions of the form

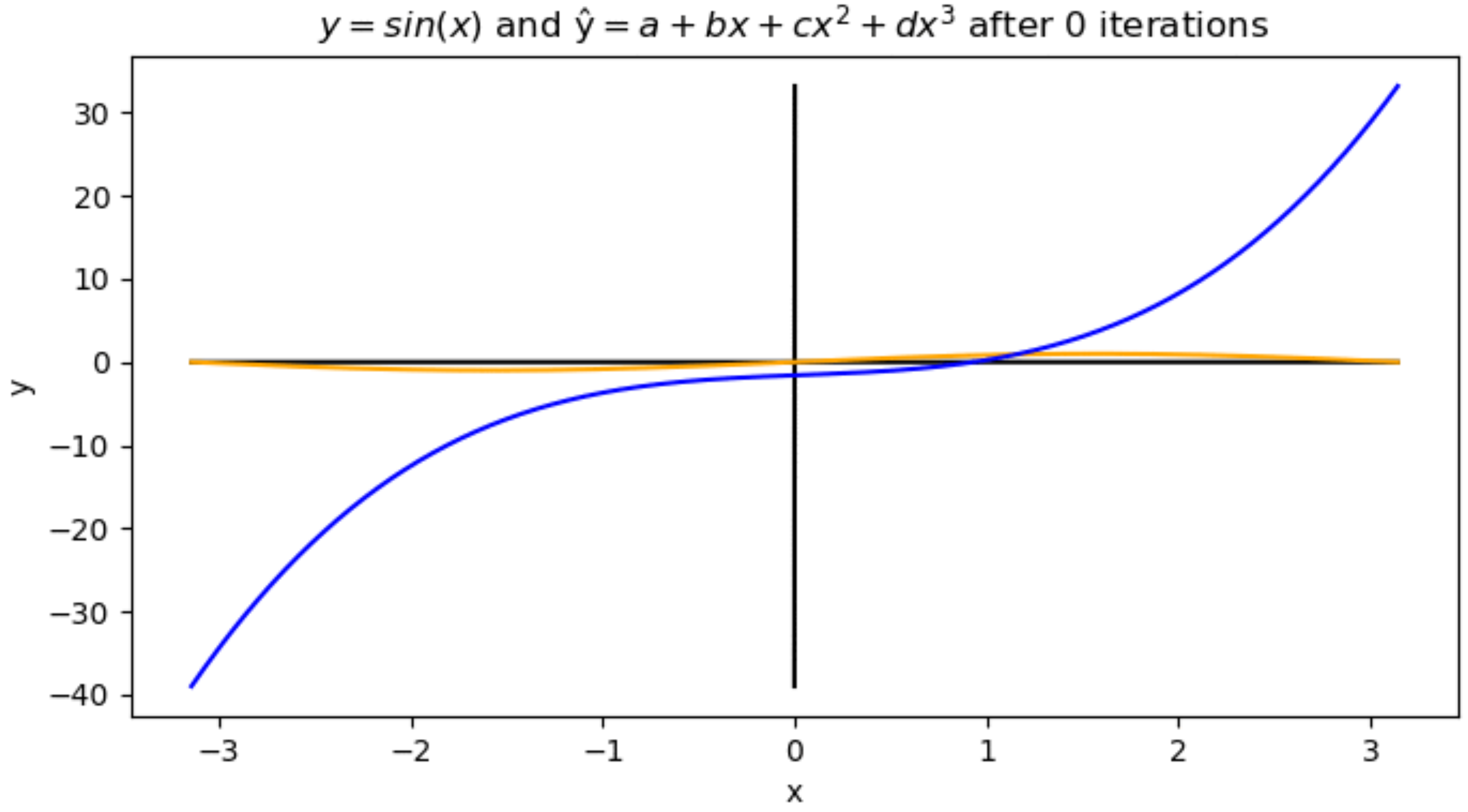
$$f(x) = a + bx + cx^2 + dx^3 = z@w$$

…where we're defining…

$$w = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}, x = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix}$$

- We want to learn a, b, c, d so that $f(x) \approx y$

Running example: neural net regression



Mean-squared error

First, let’s define the loss function.

$$f(x_i) = a + bx_i + c{x_i}^2 + d{x_i}^3,$$

$$\epsilon_i = f(x_i) - y_i,$$

$$\mathcal{L} = \frac{1}{n}\sum_{i=0}^{n-1} {\epsilon_i}^2$$

Loss gradient

Next, find the derivative of the loss.

$$f(x_i) = a + bx_i + cx_i^2 + dx_i^3, \qquad \epsilon_i = f(x_i) - y_i, \qquad \mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} \epsilon_i^2$$

$$\nabla_w \mathcal{L} = \begin{bmatrix} \frac{d\mathcal{L}}{da} \\ \frac{d\mathcal{L}}{db} \\ \frac{d\mathcal{L}}{dc} \\ \frac{d\mathcal{L}}{dd} \end{bmatrix} = \frac{2}{n}\sum_{i=1}^{n} \epsilon_i \nabla_w \epsilon_i = \frac{2}{n}\sum_{i=1}^{n} \epsilon_i \begin{bmatrix} \frac{df(x_i)}{da} \\ \frac{df(x_i)}{db} \\ \frac{df(x_i)}{dc} \\ \frac{df(x_i)}{dd} \end{bmatrix} = \frac{2}{n}\sum_{i=1}^{n} \epsilon_i \begin{bmatrix} 1 \\ x_i \\ x_i^2 \\ x_i^3 \end{bmatrix}$$

Gradient update

Now, update the weights by subtracting the gradient.

$$a = a - \frac{\eta}{2}\frac{d\mathcal{L}}{da} = a - \frac{\eta}{n}\sum_{i=1}^{n}(f(x_i) - y_i)\,,$$

$$b = b - \eta\frac{d\mathcal{L}}{db} = b - \frac{\eta}{n}\sum_{i=1}^{n}(f(x_i) - y_i)x_i\,,$$

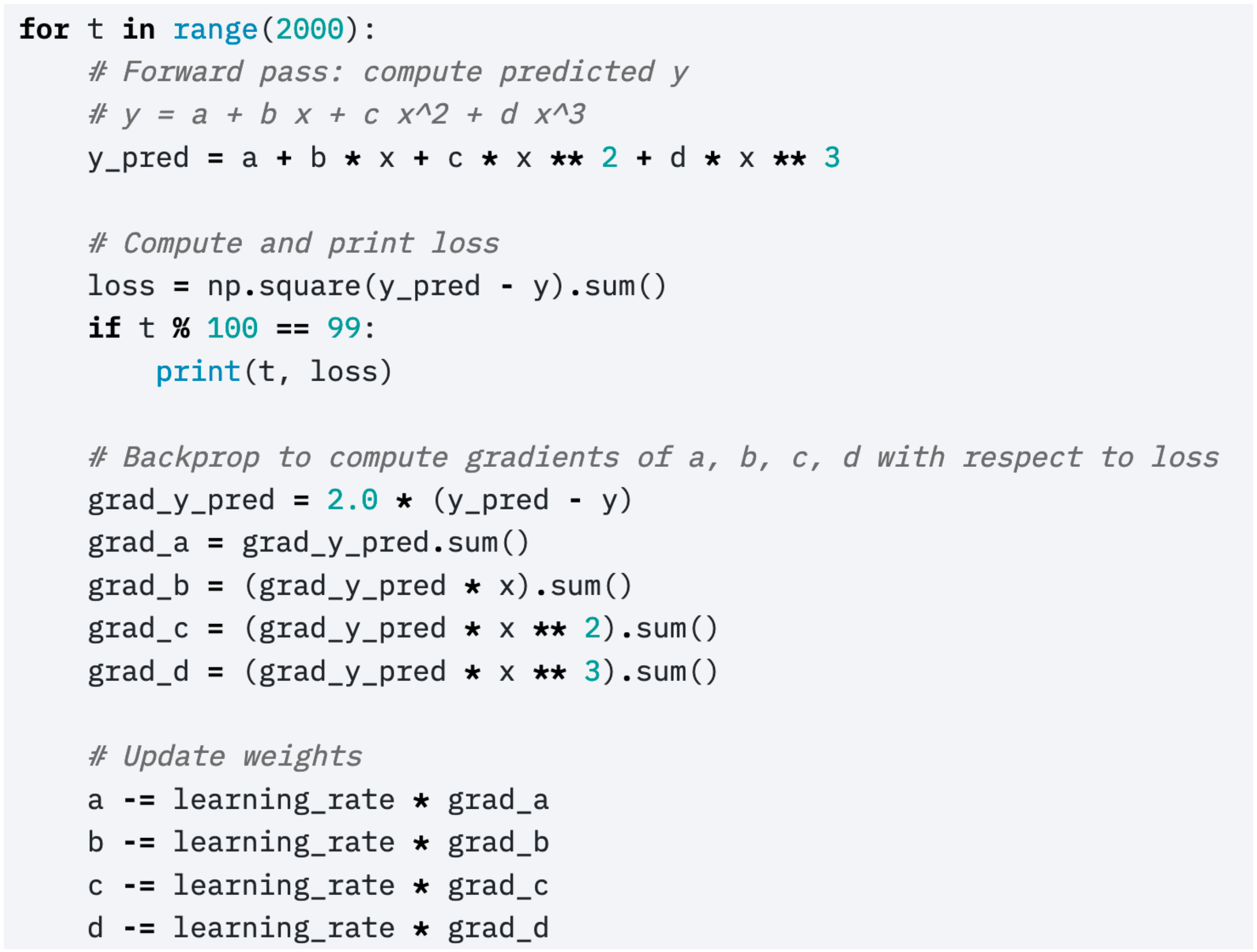
$$c = c - \eta\frac{d\mathcal{L}}{dc} = c - \frac{\eta}{n}\sum_{i=1}^{n}(f(x_i) - y_i){x_i}^2\,,$$

$$d = d - \eta\frac{d\mathcal{L}}{dd} == d - \frac{\eta}{n}\sum_{i=1}^{n}(f(x_i) - y_i){x_i}^3$$

How a neural network is trained

Here's Justin Johnson's code for doing those things:

(https://pytorch.org/tutorials/beginner/pytorch_with_examples.html)



```python
for t in range(2000):
    # Forward pass: compute predicted y
    # y = a + b x + c x^2 + d x^3
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d
```

© 2021 Pytorch,
https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

Outline

- Review: linear regression
- Training a linear regression model using numpy
- The same example with pytorch
- torch.nn module: using predefined neural net components

Autograd: Main idea

- A neural network is a complicated function $f(x)$, made up of many simple components
- If we try to take all the derivatives, $d\mathcal{L}/dw_{l,j,k}$, all at once, in a big mass of spaghetti code, then the code will be really ugly.
- HOWEVER: Each of the components is simple to compute. Furthermore, the derivative of its output w.r.t. its input is simple.

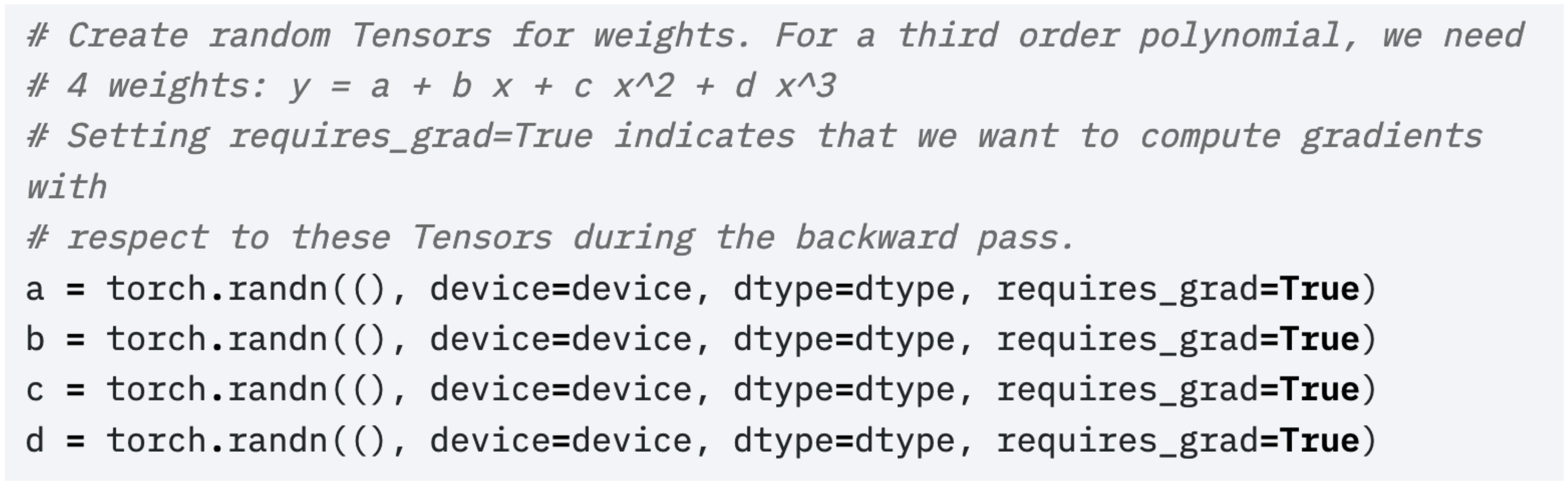
Autograd: Tensor objects

The basic idea of autograd is to create a new kind of object that takes responsibility for its own gradient.

- For example, the object might be a network weight, $w_{l,j,k}$

Autograd: Tensor objects

- In pytorch, variables that take responsibility for their own gradients are called “tensors” (https://pytorch.org/docs/stable/tensors.html)
- Here’s how Justin Johnson defines tensors for the polynomial regression problem:



```
# Create random Tensors for weights. For a third order polynomial, we need
# 4 weights: y = a + b x + c x^2 + d x^3
# Setting requires_grad=True indicates that we want to compute gradients
with
# respect to these Tensors during the backward pass.
a = torch.randn((), device=device, dtype=dtype, requires_grad=True)
b = torch.randn((), device=device, dtype=dtype, requires_grad=True)
c = torch.randn((), device=device, dtype=dtype, requires_grad=True)
d = torch.randn((), device=device, dtype=dtype, requires_grad=True)
```

Autograd: Overloaded operators

The basic idea of autograd is to create a new kind of object that takes responsibility for its own gradient.

- For example, the object might be a network weight, $w_{l,j,k}$
- These new objects have overloaded operators, so that any time we use them to compute some output, the input is cached. For example, it might be used to compute

$$f_j = b_{1,j} + \sum_k w_{1,j,k} h_k$$

```
f[j] = b[1,j]+np.sum(w[1,j,:]*h)
```

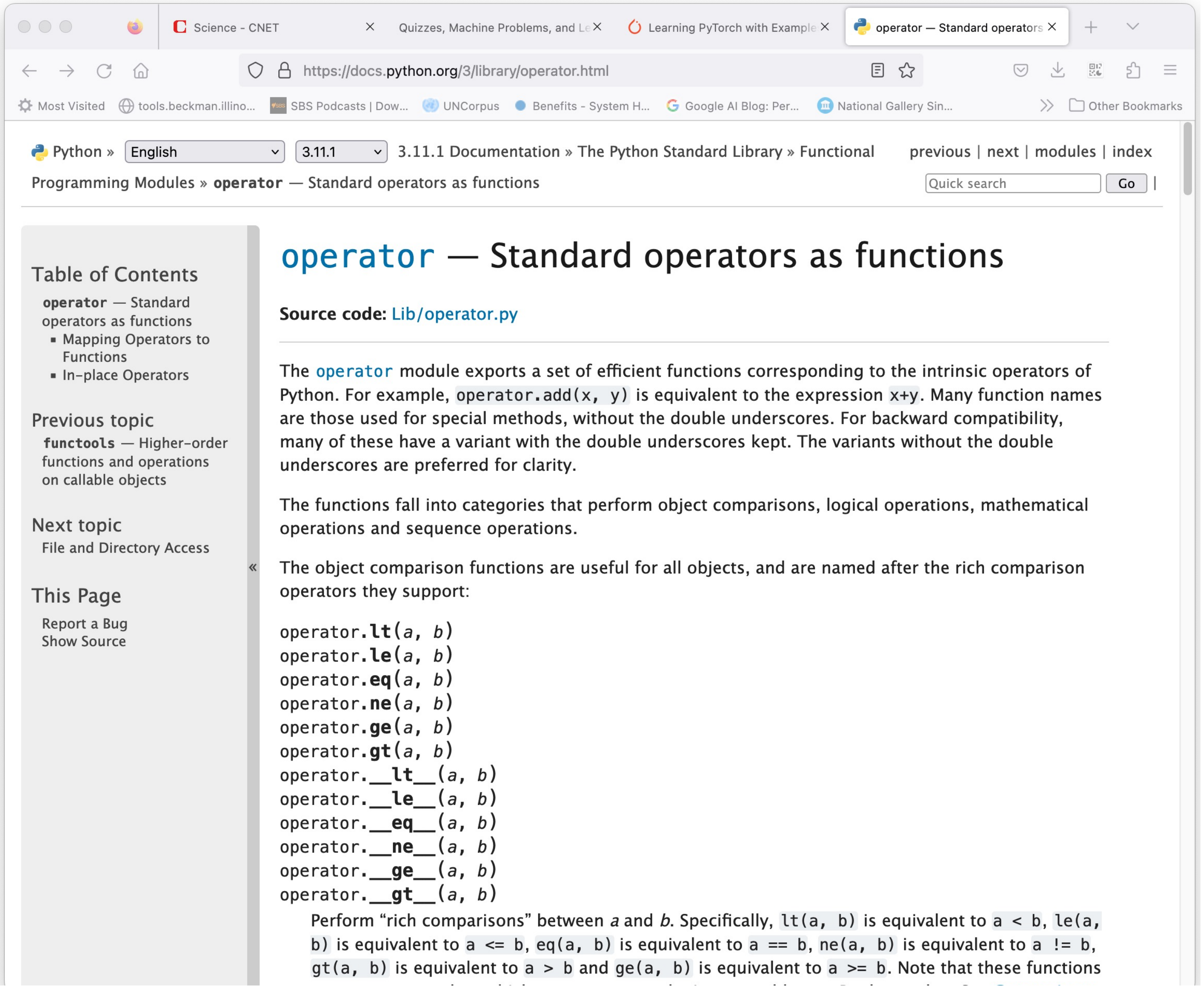
Autograd: Overloaded operators

Here's why it works: if "b" is an object that has a method named __mul__, then the python expression

f=b*x

…actually calls:

f=b.__mul__(x)



Science - CNET
Quizzes, Machine Problems, and Le
Learning PyTorch with Example
operator — Standard operators
https://docs.python.org/3/library/operator.html
Most Visited
tools.beckman.illino...
SBS Podcasts | Dow...
UNCorpus
Benefits - System H...
Google AI Blog: Per...
National Gallery Sin...
Other Bookmarks
Python » English 3.11.1 3.11.1 Documentation » The Python Standard Library » Functional Programming Modules » operator — Standard operators as functions
previous | next | modules | index
Quick search
Go

Table of Contents
- operator — Standard operators as functions
  - Mapping Operators to Functions
  - In-place Operators

Previous topic
functools — Higher-order functions and operations on callable objects

Next topic
File and Directory Access

This Page
Report a Bug
Show Source

operator — Standard operators as functions

Source code: Lib/operator.py

The operator module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, `operator.add(x, y)` is equivalent to the expression `x+y`. Many function names are those used for special methods, without the double underscores. For backward compatibility, many of these have a variant with the double underscores kept. The variants without the double underscores are preferred for clarity.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations and sequence operations.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

```
operator.lt(a, b)
operator.le(a, b)
operator.eq(a, b)
operator.ne(a, b)
operator.ge(a, b)
operator.gt(a, b)
operator.__lt__(a, b)
operator.__le__(a, b)
operator.__eq__(a, b)
operator.__ne__(a, b)
operator.__ge__(a, b)
operator.__gt__(a, b)
```

Perform "rich comparisons" between *a* and *b*. Specifically, `lt(a, b)` is equivalent to `a < b`, `le(a, b)` is equivalent to `a <= b`, `eq(a, b)` is equivalent to `a == b`, `ne(a, b)` is equivalent to `a != b`, `gt(a, b)` is equivalent to `a > b` and `ge(a, b)` is equivalent to `a >= b`. Note that these functions

Autograd: Overloaded operators

The operator overload code looks something like this:

```
class Tensor(torch.autograd.Function):
    def __init__(self, weight):
        self.weight = weight
        self.saved_tensors = ()
    def __mul__(self, other):
        self.saved_tensors = (self.saved_tensors[:], other)
        returnvalue = self.weight * other
        return Tensor(returnvalue)
```

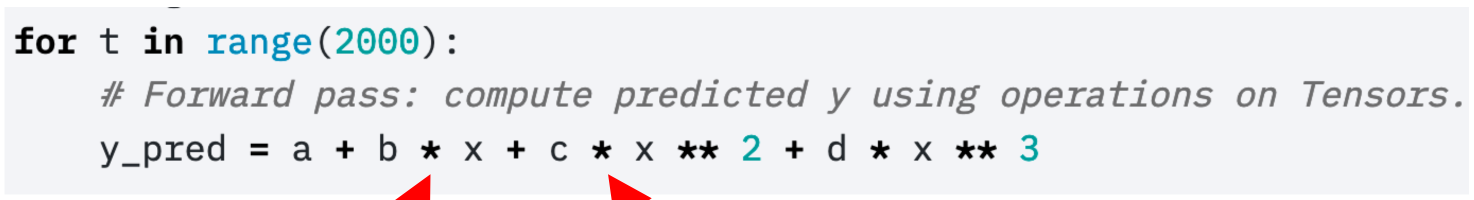
Cache x in self.saved_tensors, so we can use it later…

Then calculate the output of the multiply operation,

… and cast the return value as a Tensor.

Autograd: Overloaded operators

Here’s how it gets used:



```
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
```

Stores x in b.saved_tensors

Stores x**2 in c.saved_tensors

Python overloaded operators: the expression “b*x” actually calls b.__mul__(x).

Autograd: the Loss tensor

The basic idea of autograd is to create a new kind of object, a tensor, that takes responsibility for its own gradient. Any time we use tensors to compute some output, the input is cached. For example, these operations:

$$f(x_i) = a + bx_i + cx_i^2 + dx_i^3$$

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2$$

```
f = a + b*x + c*x**2 + d*x**3
loss = (f-y).pow(2)
```

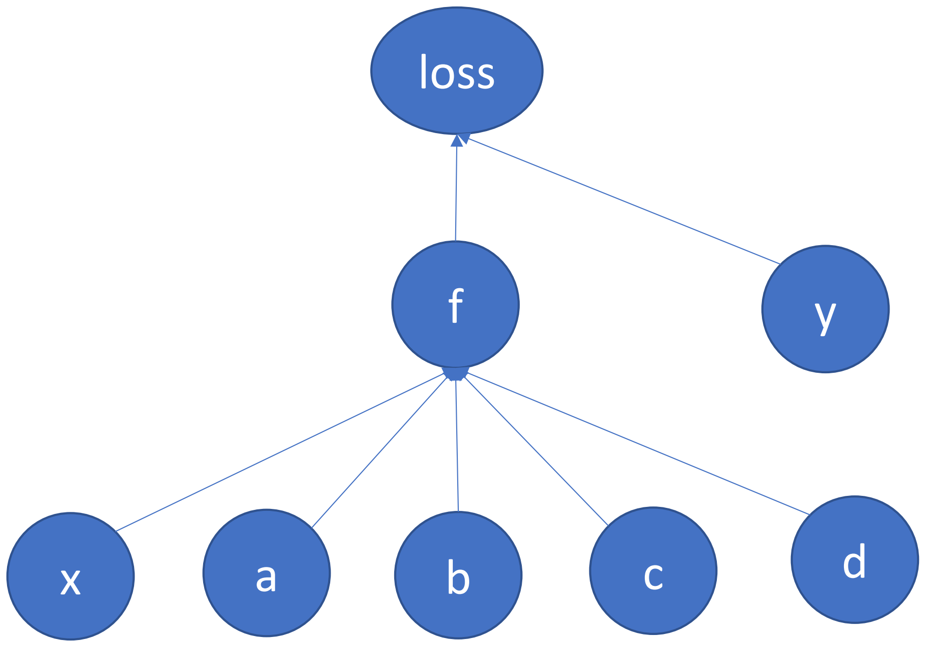
...will calculate the loss, but will also store some extra information in loss.saved_tensors, f.saved_tensors, a.saved_tensors, b.saved_tensors, c.saved_tensors, d.saved_tensors, and x.saved_tensors.

Autograd: the Loss tensor

Notice the flow diagram that was implied by those lines of code.

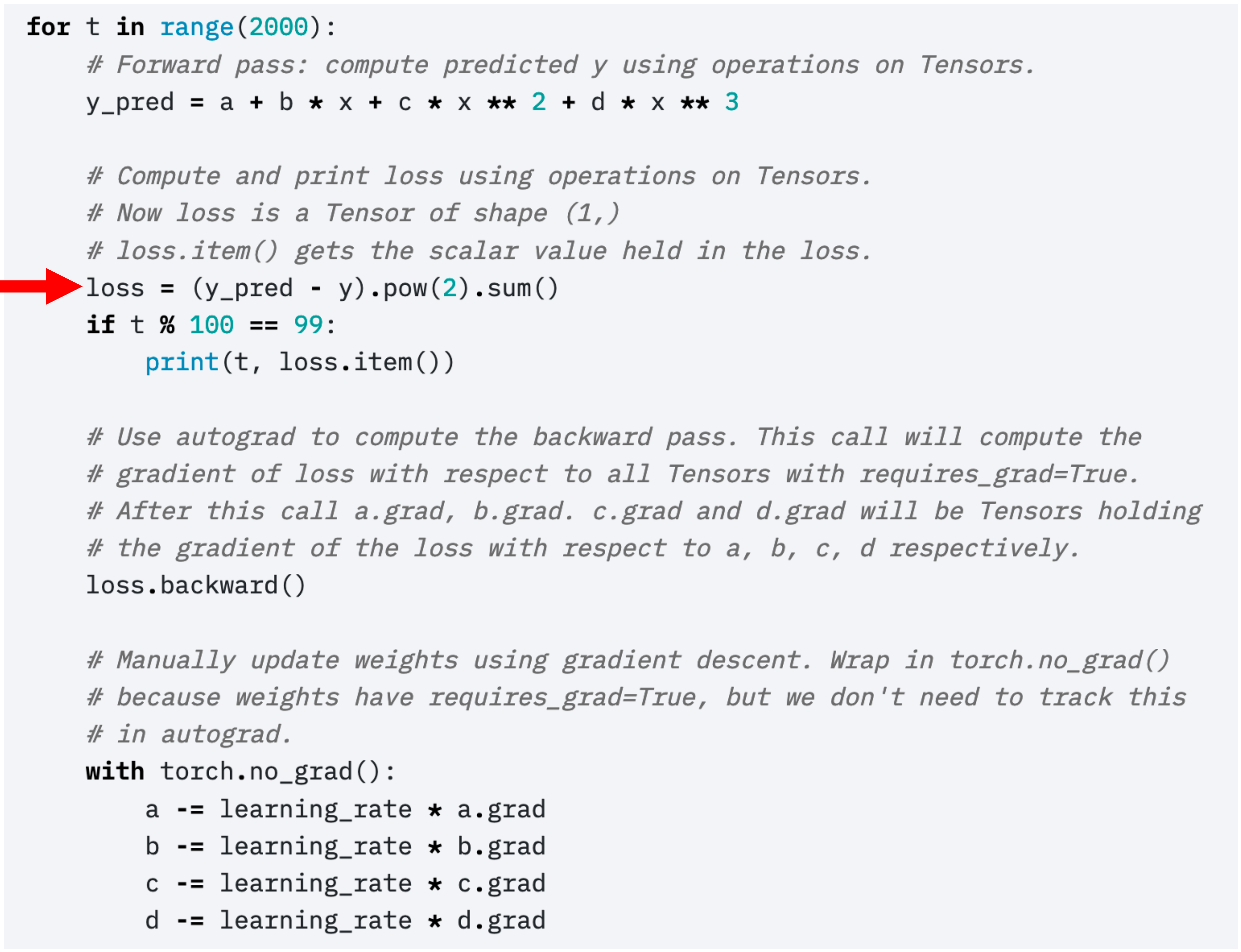
Each tensor's overloaded __mul__ operator keeps track of the variables used to compute it:

- loss.saved_tensors has pointers to f and y
- f.saved_tensors has pointers to x, a, b, c, and d



Autograd

loss depends on y_pred, which depends on a, b, c, d.



```python
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
```

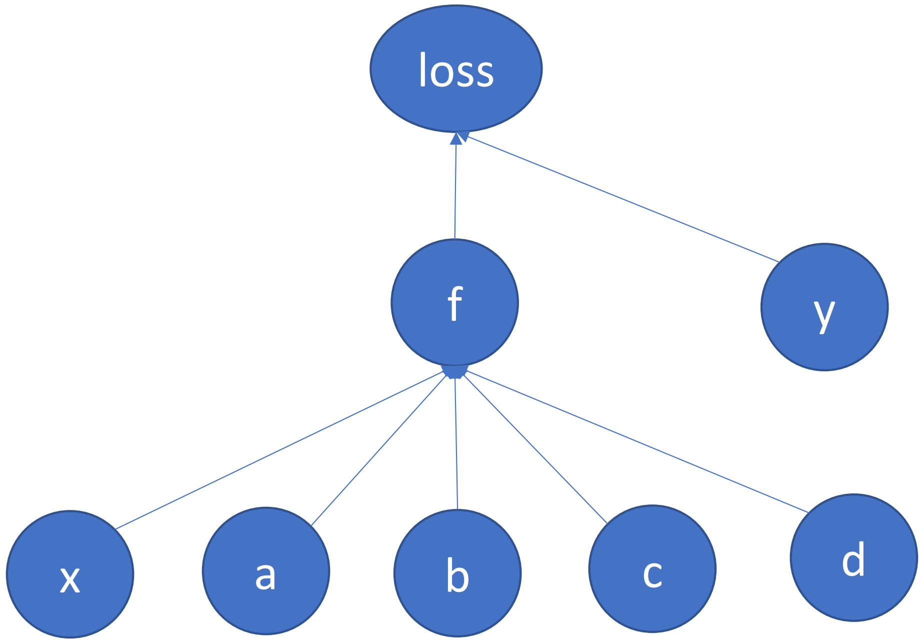
Autograd: the backward function

Every tensor object has a method called backward().

If backward() is called with no arguments, it calculates the derivative with respect to the inputs:

- loss.backward() calculates

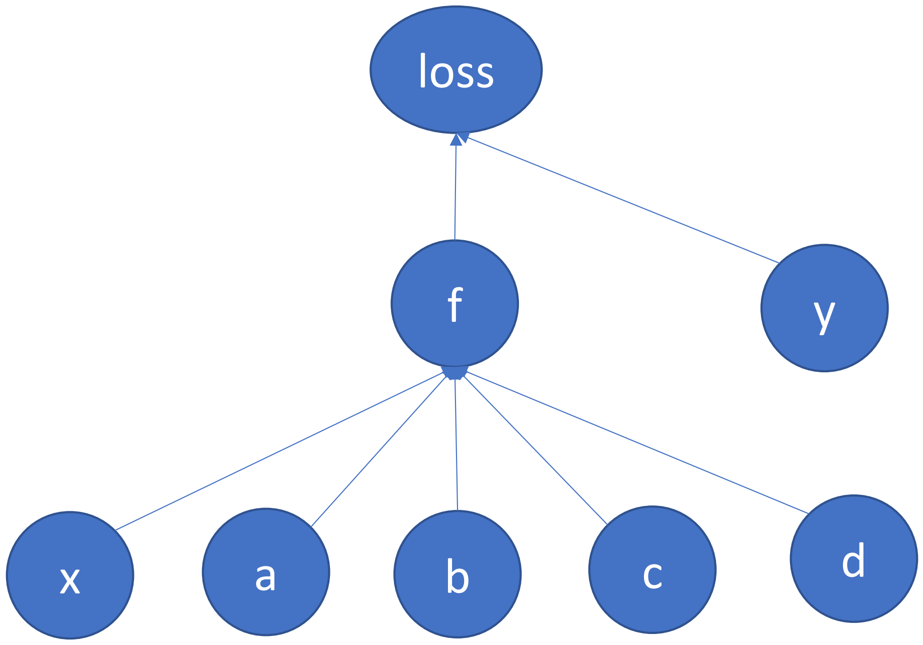
tmp= $\frac{d\mathcal{L}}{df}$, then calls the method f.backward(tmp).



Autograd: the backward function

If f.backward(tmp) is called with the argument tmp= $\frac{d\mathcal{L}}{df}$, it does three things:

- Store f.grad= $\frac{d\mathcal{L}}{df}$
- Calculate derivative w.r.t. each input, for example, tmpc=
$$\frac{d\mathcal{L}}{dc} = \frac{d\mathcal{L}}{df} \times \frac{df}{dc}$$
- Pass the input derivatives back to the inputs, e.g., call c.backward(tmpc)

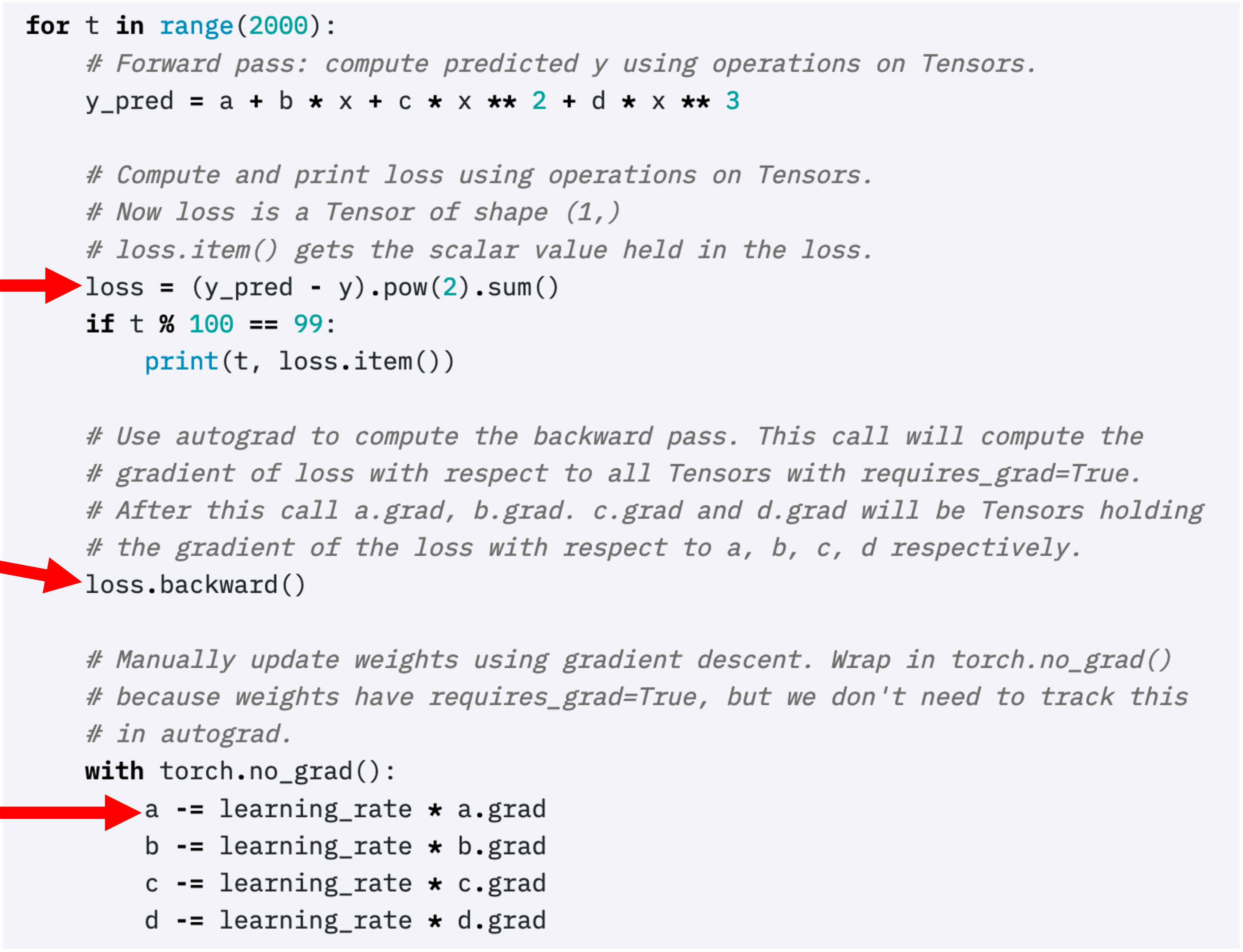


Autograd

loss depends on y_pred, which depends on a, b, c, d.

Calculates the derivative of the loss w.r.t. each of its input tensors.

Uses the resulting derivatives to update the weights.



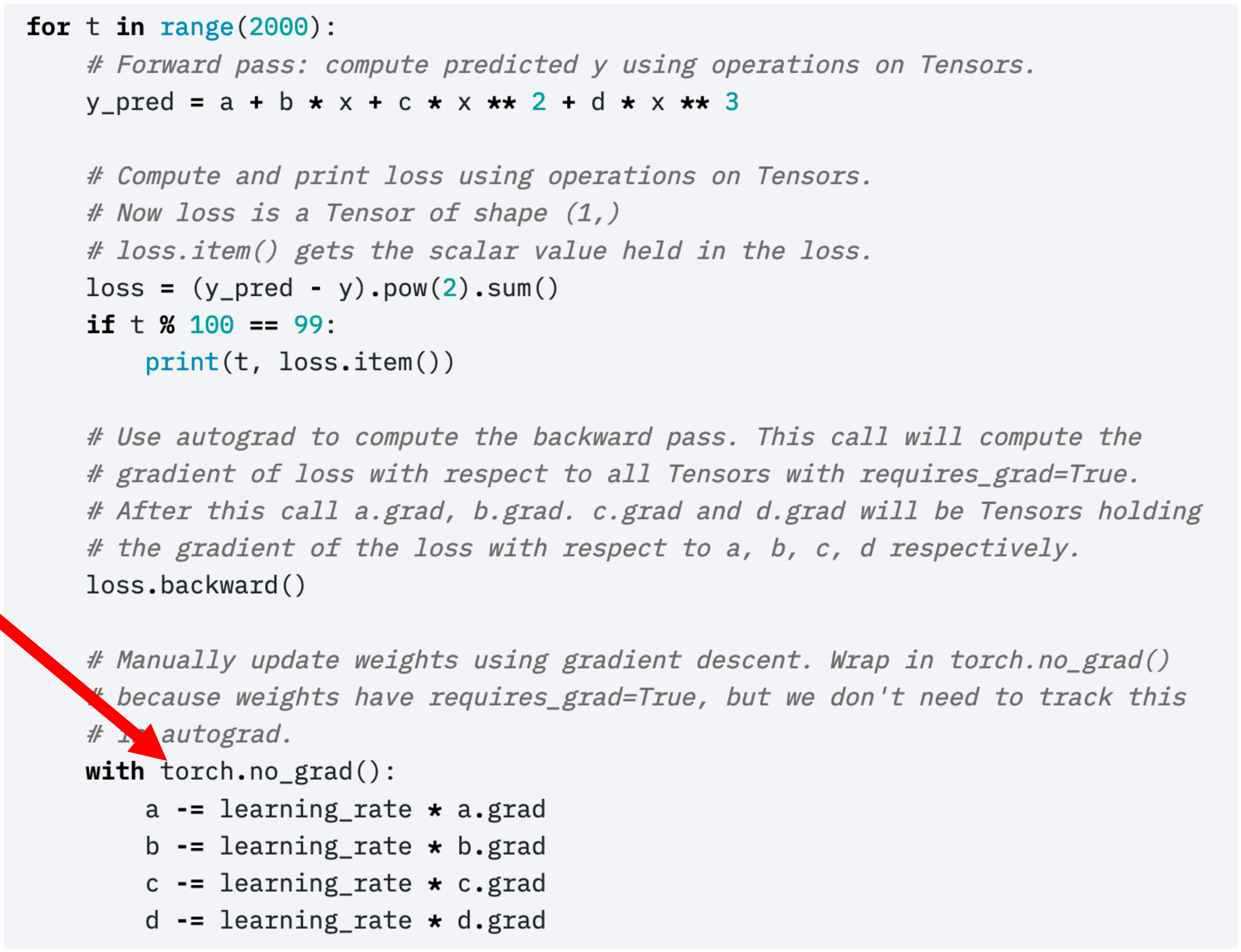
```python
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
```

Details: How to turn off autograd

- As you know, every time you add, subtract, multiply or divide a tensor by anything, the tensor stores data in self.saved_tensors, so it can use that information later to compute the gradient
- How do you turn this behavior off?

Dynamically turning off Autograd

These weight updates are not part of the neural network forward pass.



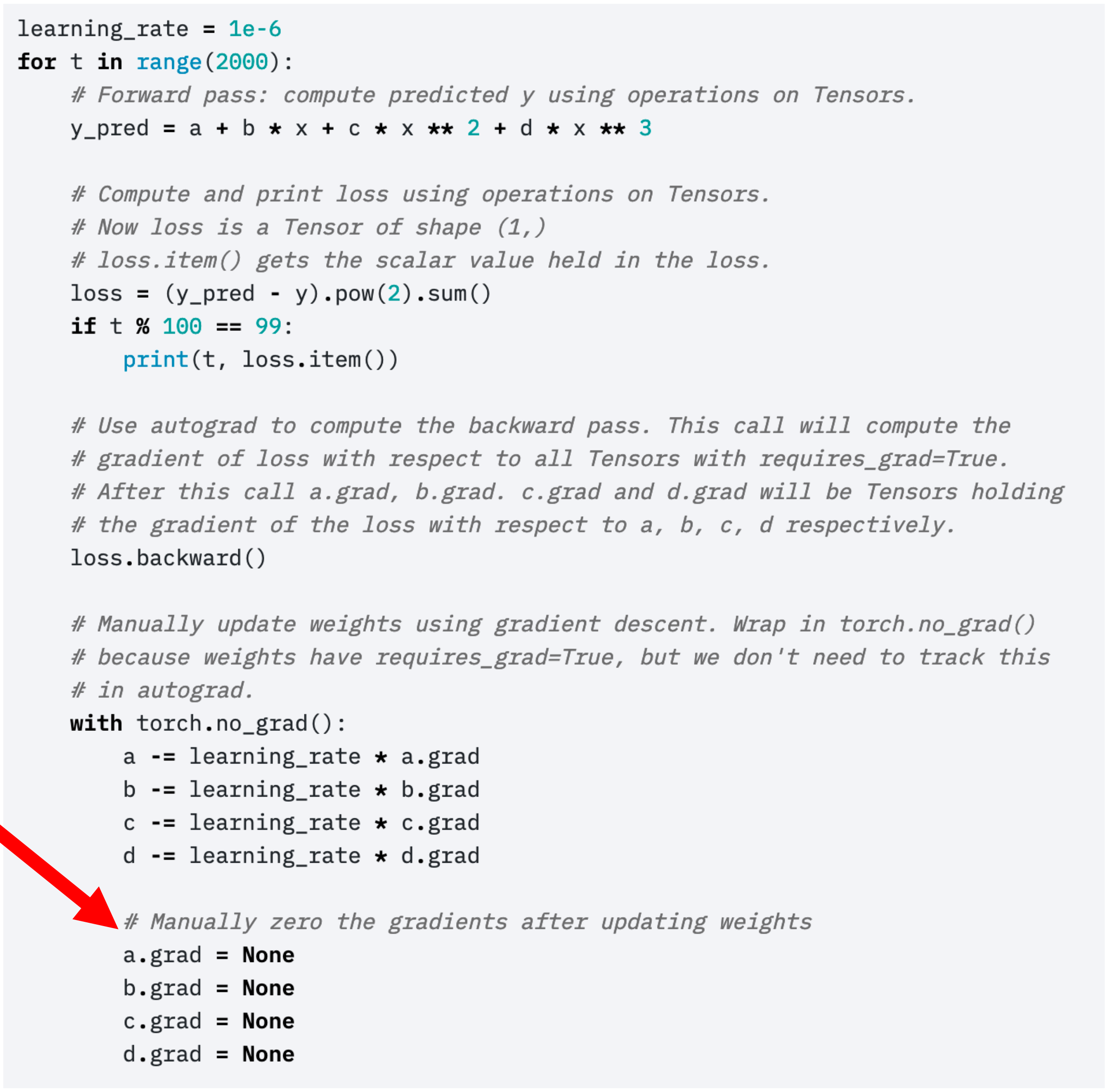
```python
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
```

How to zero out the gradients

- When you call backward() over a tensor, it doesn’t zero out any previous gradients
- Instead, it adds the current gradient to the previous gradients
- A very very very common mistake: running 2000 iterations, with the gradient accumulating from each iteration to the next, instead of zeroing it out in between iterations

Manually zeroing out the gradients

Here's the part I didn't show you before.



```python
learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        # Manually zero the gradients after updating weights
        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None
```

Outline

- Review: linear regression
- Training a linear regression model using numpy
- The same example with pytorch
- torch.nn module: using predefined neural net components

Pytorch nn module

- The autograd feature of pytorch allows you to define only the forward propagation of your neural net. As long as all of the component operations are in pytorch’s library, the back-propagation will be computed for you.
- Tensors just do multiplication and addition. What about other types of operations?
- General operations are contained in the nn module, using the formalism of a “layer.”

Some types of layers

- torch.nn.Linear: a layer that computes $o = i@w^T + b$
- torch.nn.Softmax: a layer that computes $o_j = \frac{\exp(i_j)}{\sum_k \exp(i_k)}$
- torch.nn.Sigmoid: a layer that computes $o_j = \frac{1}{1-\exp(-i_j)}$
- torch.nn.ReLU: a layer that computes $o_j = \max(0, i_j)$
- torch.nn.Sequential: a model that takes a sequence of layers as its arguments, and applies them, one after the other, in order

m=torch.nn.Linear(n_1,n_2)

- This creates a callable object, m, such that o=m(i) treats each row of i as a vector, and generates a corresponding row of o using the operation:

$$o = i@w^T + b$$

i can be a tensor of any size, as long as its last dimension (the dimension of each row) is n_1

- o is then a tensor of the same shape as i, except that its last dimension (the row length) is now n_2
- m.weight (w) is a matrix of size (n_2,n_1)
- m.bias (b) is a row vector of length n_2

Example: Linear, Sigmoid, Softmax

- Here's an example flowgraph. We could create the layers as:

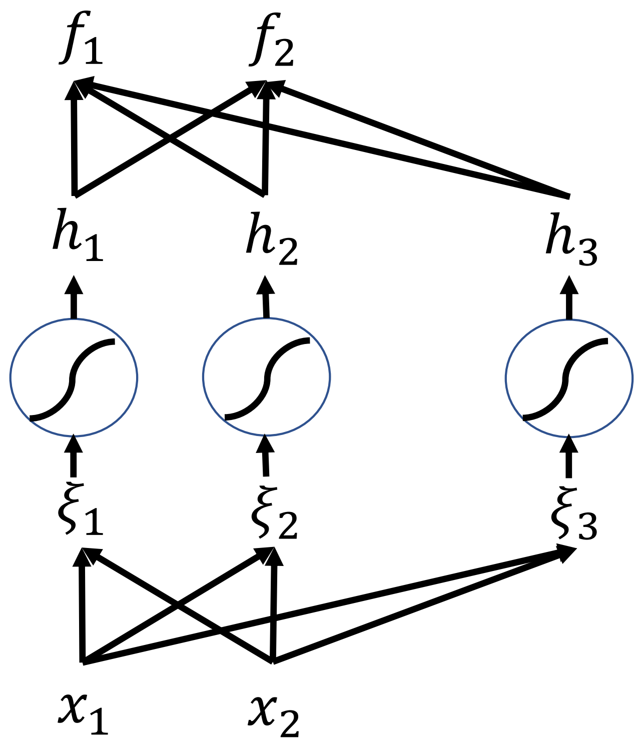
```
linearlayer1 = torch.nn.Linear(2,3)
sigmoidlayer1 = torch.nn.Sigmoid()
linearlayer2 = torch.nn.Linear(3,2)
loss_function = torch.nn.MSEloss()
```

- Having created them, we could then run forward pass as:

```
xi = linearlayer1(x)
h = sigmoidlayer1(xi)
f = linearlayer2(h)
loss = loss_function(f,y)
```

- Then we could calculate all of the gradients by running

```
loss.backward()
```



torch.nn.Sequential

- torch.nn.Sequential is a special module that creates a sequence of layers, where each layer's output is the next layer's input. For example:

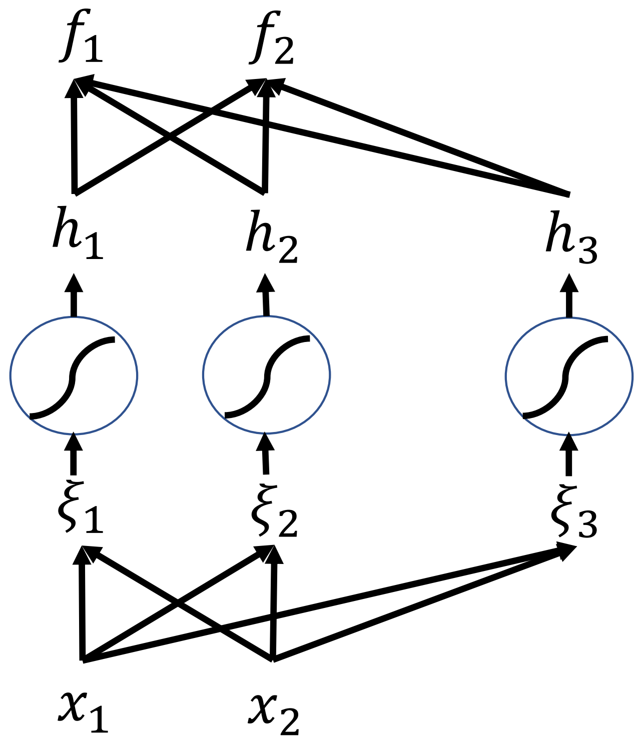
```
model = torch.nn.Sequential(
        torch.nn.Linear(2,3),
        torch.nn.Sigmoid(),
        torch.nn.Linear(3,2))
loss_function = torch.nn.MSEloss()
```

- Then you can run forward pass by just typing:

```
f = model(x)
loss = loss_function(f,y)
```

- You can still calculate all of the gradients by running

```
loss.backward()
```



torch.nn.Sequential: where are the parameters?

- The layers each have their own parameters, for example, a model created using the commands on the previous slide would have

```
model[0].weight
model[0].bias
model[2].weight
model[2].bias
```

- Accessing them that way requires you to know which layers have weights and biases, and which don't. An easier way is to use the function model.parameters(), which iterates through all trainable parameters, regardless of where they are actually stored:

```
for param in model.parameters():
    param -= learning_rate * param.grad
```

Outline

- Review: linear regression
- Training a linear regression model using numpy
  - https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#warm-up-numpy
- The same example with pytorch
  - https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-tensors
- torch.nn module: using predefined neural net components
  - https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#nn-module