Artificial Intelligence

Building NNs using PyTorch

Woohwan Jung

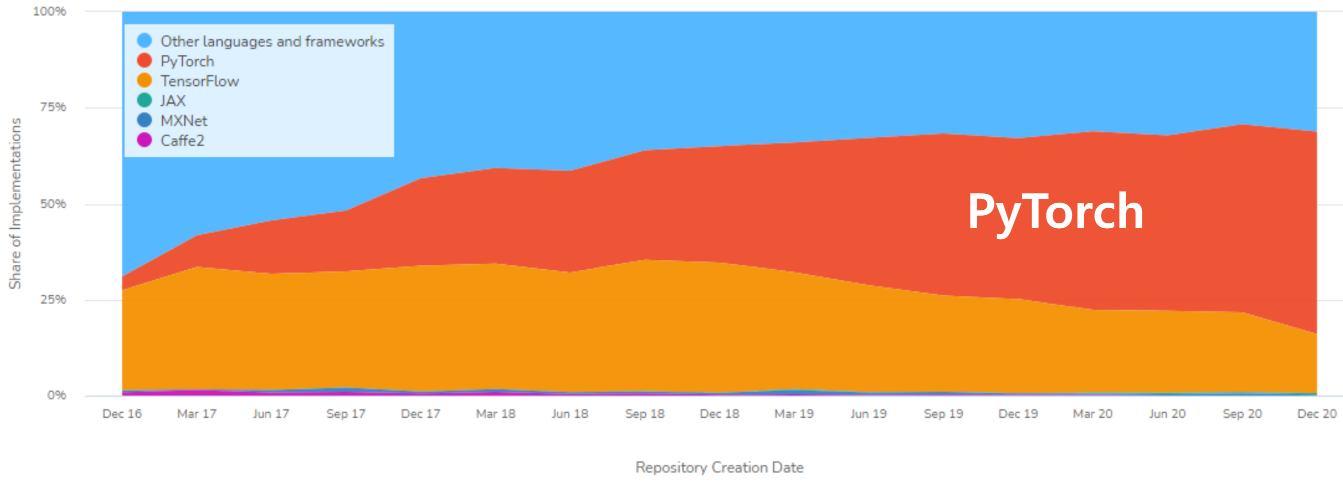
한양대학교 ERICA
소프트웨어융합대학
COLLEGE OF COMPUTING

인공지능학과
Department of
Artificial Intelligence

Shallow Neural Networks

- Logistic Regression + a Hidden Layer
- Hidden layer
  - Linear function with model parameters
    - i.e.) $\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}$
  - Nonlinear activation function
    - Ex) ReLU, tanh, LeakyReLU

PyTorch



- Open source DL framework
- Provides two high-level features
  - Tensor computing (like NumPy) with strong acceleration via GPU
  - Automatic differentiation system
- Easy to learn
- Growing ecosystem

Let's `import torch`

With and without PyTorch: Model

```python
class shallow_neural_network():
    def __init__(self, num_input_features, num_hiddens):
        self.num_input_features = num_input_features
        self.num_hiddens = num_hiddens

        self.W1 = np.random.normal(size = (num_hiddens, num_input_features))
        self.b1 = np.random.normal(size = num_hiddens)
        self.W2 = np.random.normal(size = num_hiddens)
        self.b2 = np.random.normal(size = 1)


    def sigmoid(self,z):
        return 1/(1 + np.exp(-z))

    def predict(self,x):
        z1 = np.matmul(self.W1,x) + self.b1
        a1 = np.tanh(z1)
        z2 = np.matmul(self.W2,a1)+ self.b2
        a2 = self.sigmoid(z2)
        return a2, (z1,a1,z2, a2)
```

Forward pass

```python
class shallow_neural_network(nn.Module):
    def __init__(self, num_input_features, num_hiddens):
        super().__init__()
        self.num_input_features = num_input_features
        self.num_hiddens = num_hiddens

        self.linear1 = nn.Linear(num_input_features, num_hiddens)
        self.linear2 = nn.Linear(num_hiddens, 1)

        self.tanh = torch.nn.Tanh()
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self,x):
        z1 = self.linear1(x)
        a1 = self.tanh(z1)
        z2 = self.linear2(a1)
        a2 = self.sigmoid(z2)
        return a2
```

Forward pass

NumPy implementation

PyTorch implementation

With and without PyTorch: Training

```python
def train(X, Y, model, lr = 0.1):
    dW1 = np.zeros_like(model.W1)
    db1 = np.zeros_like(model.b1)
    dW2 = np.zeros_like(model.W2)
    db2 = np.zeros_like(model.b2)
    m = len(X)
    cost = 0.0
    for x,y in zip(X,Y):
        a2, (z1,a1,z2, _) = model.predict(x)
        if y == 1:
            cost -= np.log(a2)
        else:
            cost -= np.log(1-a2)

        diff = a2-y
        # layer 2
        db2 += diff
        dW2 += diff * a1

        #layer 1
        db1_tmp = diff * model.W2 * (1-a1**2)
        db1 += db1_tmp
        dW1 += np.outer(db1_tmp, x)

    cost /= m
    model.W1 -= lr * dW1/m
    model.b1 -= lr * db1/m
    model.W2 -= lr * dW2/m
    model.b2 -= lr * db2/m

    return cost
```

```python
model = shallow_neural_network(2,3)
```

```python
for epoch in range(100):
    cost = train(X,Y, model, 1.0)
    if epoch %10 == 0:
        print(epoch, cost)
```

Backpropagation

NumPy implementation

```python
num_epochs = 100
lr = 1.0
num_hiddens = 3

model = shallow_neural_network(2,num_hiddens)
optimizer = optim.SGD(model.parameters(), lr = lr)
loss = nn.BCELoss()
```

```python
for epoch in range(num_epochs):
    optimizer.zero_grad()

    cost  = 0.0
    for x,y in zip(X,Y):
        x_torch = torch.from_numpy(x)
        y_torch = torch.FloatTensor([y])

        y_hat = model(x_torch)

        loss_val = loss(y_hat, y_torch)
        cost += loss_val

    cost = cost / len(X)
    cost.backward()
    optimizer.step()

    if epoch %10==0:
        print(epoch, cost)
```

Backpropagation

PyTorch implementation

Overview

- Tensor
- Tensor operations
- Autograd
- Module

torch.Tensor

torch.Tensor

- Like tensors in linear algebra, PyTorch tensors are arrays which can be multi-dimensional
- PyTorch tensors are similar to NumPy ndarrays except for GPU acceleration
  - PyTorch provides most of tensor operations in NumPy

Initialize with Python lists

```
import torch
import numpy as np
```

```
arr = [[1,2],[2,3]]
```

In [4]:

```
arr_n = np.array(arr)
print(type(arr_n))
print(arr_n)
```

```
<class 'numpy.ndarray'>
[[1 2]
 [2 3]]
```

NumPy

In [5]:

```
arr_t = torch.Tensor(arr)
print(type(arr_t))
print(arr_t)
```

```
<class 'torch.Tensor'>
tensor([[1., 2.],
        [2., 3.]])
```

PyTorch

Initialization: ones & zeros

- PyTorch tensors are similar to NumPy ndarrays

In [35]:

```
print(np.ones((2,3)))
print(np.zeros((2,3)))
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
[[0. 0. 0.]
 [0. 0. 0.]]
```

NumPy

In [34]:

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

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

PyTorch

Initialization: ones_like & zeros_like

In [8]:
```
print(np.ones_like(arr_n))
print(np.zeros_like(arr_n))
```

```
[[1 1]
 [1 1]]
[[0 0]
 [0 0]]
```

In [7]:
```
print(torch.ones_like(arr_t))
print(torch.zeros_like(arr_t))
```

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

NumPy

PyTorch

Two ways of specifying data type

- 1. Use keyword argument `dtype`

```
In [4]: print(torch.ones((2,3), dtype = torch.int))
        print(torch.ones((2,3), dtype = torch.float))

tensor([[1, 1, 1],
        [1, 1, 1]], dtype=torch.int32)
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

- 2. Use typed tensors

```
ft = torch.FloatTensor([1,2])
print(ft)
print(ft.dtype)

tensor([1., 2.])
torch.float32
```

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point 1	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTensor`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`	`torch.cuda.LongTensor`
Boolean	`torch.bool`	`torch.BoolTensor`	`torch.cuda.BoolTensor`

Default type

For more types, please refer to
https://pytorch.org/docs/stable/tensors.html

Tensor operations

Accessing elements

- Access

```
In [42]: arr_t = torch.Tensor([[1,2],[2,3]])

In [43]: arr_t[0,1]
Out[43]: tensor(2.)
```

Similar to NumPy
But, it always returns Tensor

- Get a Python number
  - torch.Tensor.item()
    - Get a Python number from a tensor containing a single value

```
In [44]: arr_t[0,1].item()
Out[44]: 2.0
```

- Update
  - Same with NumPy

```
In [45]: arr_t[0,1] = 0

In [46]: arr_t
Out[46]: tensor([[1., 0.],
                 [2., 3.]])
```

Slicing

```
In [74]: t = torch.Tensor([[1,2,3,4],[2,3,4,5],[5,6,7,8]])
         print(t)

tensor([[1., 2., 3., 4.],
        [2., 3., 4., 5.],
        [5., 6., 7., 8.]])
```

- Taking elements from one given index to another index
  - Pass slice instead of index: `[start: end]`
    - Start (inclusive), end (exclusive)
    - Default values (`start`: 0, end: length)

```
In [75]: t[:2]

Out[75]: tensor([[1., 2., 3., 4.],
                 [2., 3., 4., 5.]])
```

```
In [76]: t[:,1:]

Out[76]: tensor([[2., 3., 4.],
                 [3., 4., 5.],
                 [6., 7., 8.]])
```

```
In [81]: t[1:,1:3]

Out[81]: tensor([[3., 4.],
                 [0., 0.]])
```

```
In [82]: t[1:,1:3] = 0
         print(t)

tensor([[1., 2., 3., 4.],
        [2., 0., 0., 5.],
        [5., 0., 0., 8.]])
```

Negative slicing

```
In [74]: t = torch.Tensor([[1,2,3,4],[2,3,4,5],[5,6,7,8]])
         print(t)

tensor([[1., 2., 3., 4.],
        [2., 3., 4., 5.],
        [5., 6., 7., 8.]])
```

- Use minus operator to refer to an index from the end

Index	0	1	2	…	N-2	N-1
Negative index	N-1	N-2	N-3	…	-2	-1

```
In [77]: t[:,:-1]

Out[77]: tensor([[1., 2., 3.],
                 [2., 3., 4.],
                 [5., 6., 7.]])
```

```
In [13]: t[:,-3:-1]

Out[13]: tensor([[2., 3.],
                 [3., 4.],
                 [6., 7.]])
```

Shape & Transpose (matrix)

- Transpose of 2-D tensor (matrix)
  - Tensor.T

```
X
```

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

```
X.T
```

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

- Shape
  - A tuple of tensor dimensions

```
print(X.shape)
print(X.T.shape)
```

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

Sum

- `torch.sum`(*input, *, dtype=None*) → Tensor
  - Parameters
    - **input** (Tensor): the input tensor
  - Keyword arguments
    - **dtype** (torch.dtype, *optional*): desired datatype (default: None)
      - If specified, the input tensor is casted to dtype before the operation is performed
- Equivalent representations: $X.sum()$ and $torch.sum(X)$
- Example)

```
X
```

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

```
print(X.sum())
```

```
tensor(21.)
```

Sum

- torch.sum(*input*, *dim*, *keepdim=False*, ***, *dtype=None*) → Tensor
  - Parameters
    - **input** (Tensor): the input tensor
    - **dim** (int or tupke, *optional*): the dimensions to reduce
  - Keyword arguments
    - **dtype** (torch.dtype, *optional*): desired datatype (default: None)
    - **keepdim** (bool): whether the output tensor has *dim* retained or not
- Example)

```
X
```

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

```
print(X.sum(0))
print(X.sum(1))
```

```
tensor([5., 7., 9.])
tensor([ 6., 15.])
```

```
print(X.sum(0, keepdim = True))
print(X.sum(1, keepdim = True))
```

```
tensor([[5., 7., 9.]])
tensor([[ 6.],
        [15.]])
```

Mean

- `torch.mean`(*input*, *dim*, *keepdim=False*, ***) → Tensor
  - Parameters
    - **input** (Tensor): the input tensor
    - **dim** (int or tuple, *optional*): the dimensions to reduce
- Example)

```
X
```

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

```
print(X.mean())
print(X.mean(0))
print(X.mean(1))
```

```
tensor(3.5000)
tensor([2.5000, 3.5000, 4.5000])
tensor([2., 5.])
```

Max

- `torch.max`(*input, dim, keepdim=False, **) → Tensor
  - Parameters
    - **input** (Tensor): the input tensor
    - **dim** (int or tuple, *optional*): the dimensions to reduce
  - Output
    - **out** (Tensor, if dim is specified): the input tensor (max, max_indices)
- You can use torch.argmax to get only max_indices

```
X
```

```
tensor([[1., 7., 3.],
        [4., 5., 6.]])
```

```
print(X.max())
print(X.max(0))
print(X.max(1))
```

```
tensor(7.)
torch.return_types.max(
values=tensor([4., 7., 6.]),
indices=tensor([1, 0, 1]))
torch.return_types.max(
values=tensor([7., 6.]),
indices=tensor([1, 2]))
```

Binary Operators

```
X
```

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

```
Y
```

```
tensor([[1., 0., 2.],
        [1., 0., 1.]])
```

- Addition $Z = X + Y$
  - $z_{ij} = x_{ij} + y_{ij}$

```
X+Y
```

```
tensor([[2, 3, 4],
        [5, 6, 7]])
```

- Element-wise multiplication
  - $z_{ij} = x_{ij} * y_{ij}$

```
X*Y
```

```
tensor([[1., 0., 6.],
        [4., 0., 6.]])
```

- Matrix multiplication

```
torch.matmul(X.T,Y)
```

```
tensor([[ 5.,  0.,  6.],
        [ 7.,  0.,  9.],
        [ 9.,  0., 12.]])
```

```
torch.matmul(X,Y.T)
```

```
tensor([[ 7.,  4.],
        [16., 10.]])
```

Inner product

```
x = torch.FloatTensor([1,2])
y = torch.FloatTensor([1,1])
```

```
print(torch.inner(x,y))
```

```
tensor(3.)
```

Tensor manipulation

- You will struggle with dimensions and shapes

```
torch.matmul(X,Y)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-54-22cf2a5648e8> in <module>
----> 1 torch.matmul(X,Y)

RuntimeError: mat1 and mat2 shapes cannot be multiplied (2x3 and 2x3)
```

- View, Squeeze, Unsqueeze, …

View

- Returns a tensor with the same data and number of elements as self but with the specified shape
- X.view(*shape)
  - **shape** (int or tuple): the desired shape
- Example

```
print(X.shape)
print(X)
```

```
torch.Size([2, 3, 3])
tensor([[[1., 3., 1.],
         [0., 2., 1.],
         [1., 2., 5.]],

        [[0., 4., 2.],
         [1., 1., 2.],
         [3., 2., 1.]]])
```

```
Y = X.view(3,2,3)
print(Y.shape)
print(Y)
```

```
torch.Size([3, 2, 3])
tensor([[[1., 3., 1.],
         [0., 2., 1.]],

        [[1., 2., 5.],
         [0., 4., 2.]],

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

```
Y = X.view(6,3)
print(Y.shape)
print(Y)
```

```
torch.Size([6, 3])
tensor([[1., 3., 1.],
        [0., 2., 1.],
        [1., 2., 5.],
        [0., 4., 2.],
        [1., 1., 2.],
        [3., 2., 1.]])
```

Squeeze/Unsqueeze

X.squeeze(dim = 1)

X

Shape: (4,1,3)

Y

Shape: (4,3)

Y.unsqueeze(dim = 1)

Broadcasting

- If a PyTorch operation supports broadcast, its Tensor arguments can be automatically expanded to be of equal sizes

```
X = torch.FloatTensor([[1,2,3],[4,5,6]])
print(X)

y = torch.FloatTensor([1,2])
print(y)
```

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

```
print(X+1)
```

```
tensor([[2., 3., 4.],
        [5., 6., 7.]])
```

```
print(X+y)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-16-e28833481e72> in <module>
----> 1 print(X+y)

RuntimeError: The size of tensor a (3) must match the size of tensor b (2) a
```

```
print(X.T+y)
```

```
tensor([[2., 6.],
        [3., 7.],
        [4., 8.]])
```

Broadcasting

- If a PyTorch operation supports broadcast, its Tensor arguments can be automatically expanded to be of equal sizes
- Two non-empty tensors are "broadcastable" if
  - When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be
    1. one of them is 1
    2. one of them does not exist

```
print(X.shape)
print(X)
```

```
torch.Size([2, 3, 3])
tensor([[[1., 3., 1.],
         [0., 2., 1.],
         [1., 2., 5.]],

        [[0., 4., 2.],
         [1., 1., 2.],
         [3., 2., 1.]]])
```

```
Y = torch.ones((1,1,3))
```

```
Y = torch.ones((1,3))
```

```
Y = torch.ones(3)
```

```
Y+X
```

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

        [[1., 5., 3.],
         [2., 2., 3.],
         [4., 3., 2.]]])
```

ndarray <->tensor

torch.from_numpy(a)

a

NumPy ndarray

t

PyTorch tensor

t.numpy()

Autograd

Easy?

$$\frac{\partial L(a^{[2]},y)}{\partial b^{[2]}} = \frac{\partial L(a^{[2]},y)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial b^{[2]}}$$

$$= \left(\frac{-y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right)\sigma\left(z^{[2]}\right)\left(1-\sigma\left(z^{[2]}\right)\right)$$

$$= \left(\frac{-y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right)a^{[2]}\left(1-a^{[2]}\right)$$

$$= -y\left(1-a^{[2]}\right) + a^{[2]}(1-y) = a^{[2]} - \mathrm{y}$$

$$\frac{\partial L(a^{[2]},y)}{\partial w_i^{[2]}} = \frac{\partial L(a^{[2]},y)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial w_i^{[2]}}$$

$$= \left(a^{[2]} - y\right)\frac{\partial z^{[2]}}{\partial w_i^{[2]}}$$

$$= \left(a^{[2]} - y\right)\ a_i^{[1]}$$

$$\frac{\partial L(a^{[2]},y)}{\partial b_i^{[1]}} = \frac{\partial L(a^{[2]},y)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a_i^{[1]}} \frac{\partial a_i^{[1]}}{\partial z_i^{[1]}} \frac{\partial z_i^{[1]}}{\partial b_i^{[1]}}$$

$$= \left(a^{[2]} - y\right)\frac{\partial z^{[2]}}{\partial a_i^{[1]}} \frac{\partial a_i^{[1]}}{\partial z_i^{[1]}} \frac{\partial z_i^{[1]}}{\partial b_i^{[1]}}$$

$$= \left(a^{[2]} - y\right)w_i^{[2]}\left(1 - \tanh^2 z_i^{[1]}\right)\cdot 1$$

$$= \left(a^{[2]} - y\right)w_i^{[2]}\left(1 - {a_i^{[1]}}^2\right)$$

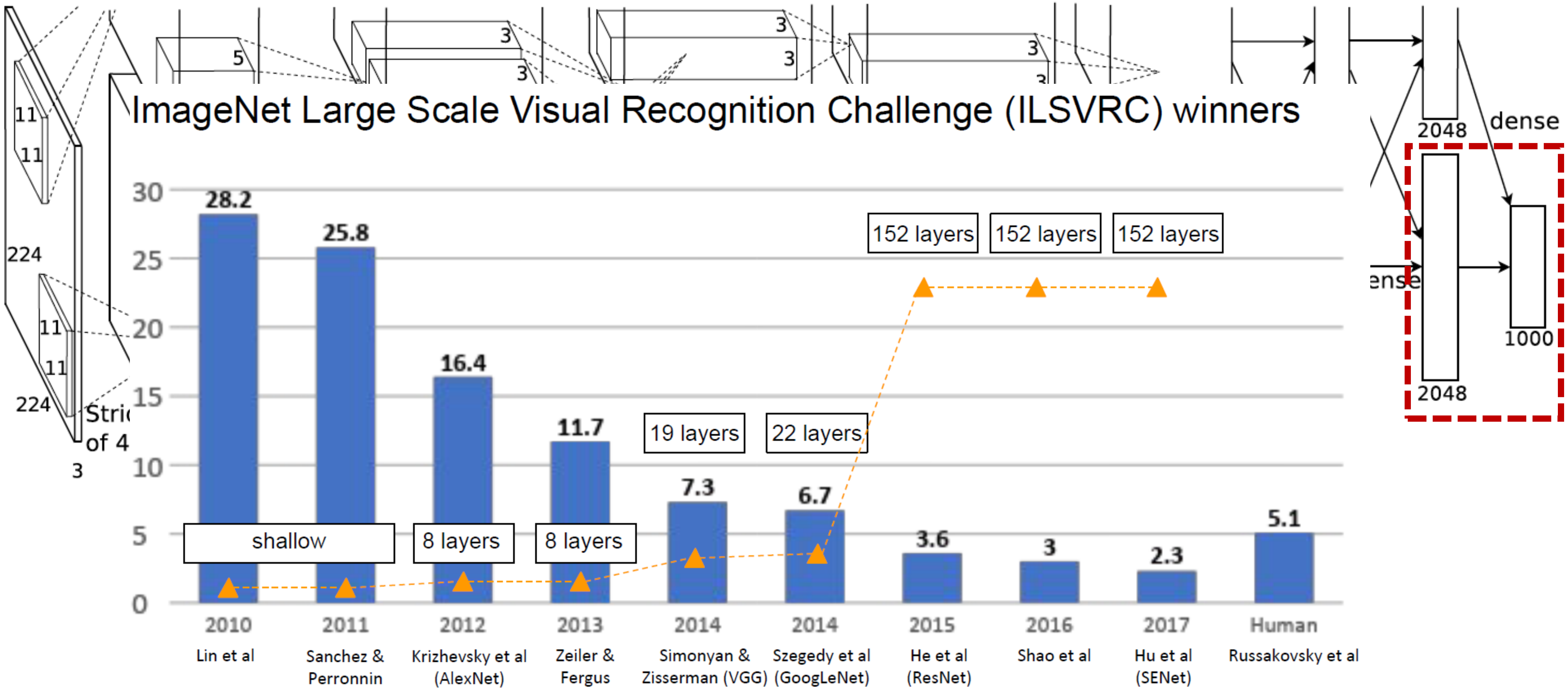
$$\frac{\partial L(a^{[2]},y)}{\partial W_{ij}^{[1]}} = \frac{\partial L(a^{[2]},y)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a_i^{[1]}} \frac{\partial a_i^{[1]}}{\partial z_i^{[1]}} \frac{\partial z_i^{[1]}}{\partial W_{ij}^{[1]}}$$

$$= \left(a^{[2]} - y\right)w_i^{[2]}\left(1 - {a_i^{[1]}}^2\right)\frac{\partial z_i^{[1]}}{\partial W_{ij}^{[1]}}$$

$$= \left(a^{[2]} - y\right)w_i^{[2]}\left(1 - {a_i^{[1]}}^2\right)x_j$$

How about this?

AlexNet, by Alex Krizhevsky et al. (2012)



Autograd

- torch.autograd is PyTorch's automatic differentiation engine that powers neural network training.
- Initialization
  - Set requires_grad to True if you want to track the gradient

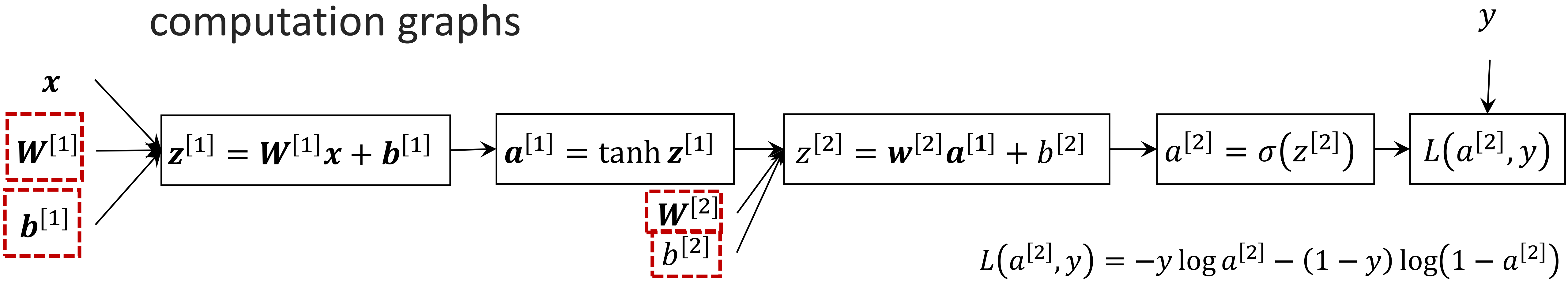
```
w = torch.randn(2, requires_grad = True)
x = torch.ones(2)
```

```
print("w", w.requires_grad)
print("x", x.requires_grad)
```

```
w True
x False
```

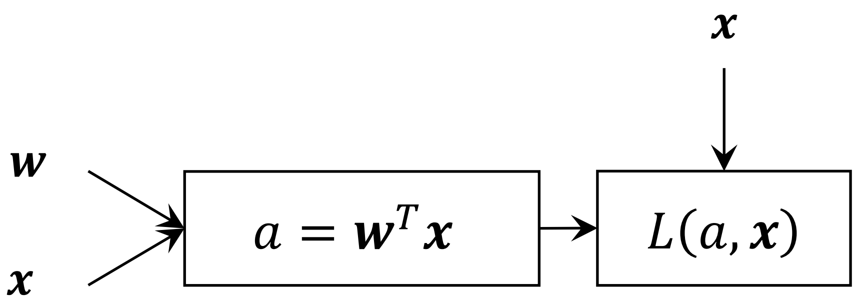
backward():

- Computes the sum of gradients of given tensors w.r.t. the leaves of computation graphs



- Accessing the gradient
  - w.grad
    - w is a tensor whose requires_grad is True

Example



$$L(a, x) = \left(a - \frac{x_1 + x_2}{2}\right)^2$$

1. Initialize

```
w = torch.randn(2, requires_grad = True)
x = torch.Tensor([1,2])
```

2. Predict output

```
y_hat = torch.inner(w,x)
```

2-2. Intermediate results

```
print(x)
print(w)
print(y_hat)
```

```
tensor([1., 2.])
tensor([-0.0444,  0.3759], requires_grad=True)
tensor(0.7075, grad_fn=<ViewBackward>)
```

3. Compute loss

```
loss = (x.mean()-y_hat)**2
print(loss)
```

```
tensor(0.6281, grad_fn=<PowBackward0>)
```

4. Backpropagation

```
loss.backward()
```

4-2. Accessing the gradient

```
w.grad
```

```
tensor([-1.5851, -3.1701])
```

Update parameters

```
In [102]: lr = 0.1
```

```
In [108]: with torch.no_grad():
              w = w - lr * w.grad
          print(w.requires_grad)
          w.requires_grad = True
```

```
False
```

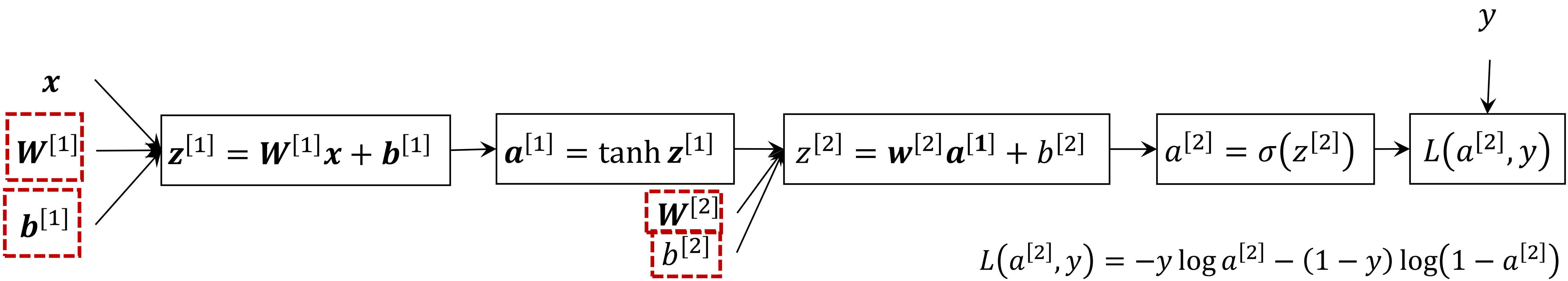
Torch.no_grad disables gradient calculation

In-place operations and Autograd

- An in-place operation is an operation that changes directly the content of a given Tensor without making a copy
  - EX) Y[1,2] = 3, A +=Y

Supporting in-place operations in autograd is a hard matter,
and we discourage their use in most cases. – PyTorch –

In-place operations and Autograd



Anyway, never use in-place operations to the tensors on the path from the parameters to the loss

Avoiding in-place operations

In-place operations

1. A += X

1. a[i:] = 0

Without In-place operations

1. A = A + X

1. mask = torch.ones_like(a)
2. mask[i:] = 0
3. a = a*mask

Implement a Shallow NN with PyTorch autograd

Data preparation & Import

```
In [1]: import numpy as np
        import torch
```

XOR data (numpy)

```
In [2]: x_seeds = np.array([(0,0),(1,0),(0,1),(1,1)],dtype=np.float32)
        y_seeds = np.array([0,1,1,0])
```

```
In [3]: N = 1000
        idxs = np.random.randint(0,4,N)
```

```
In [4]: X = x_seeds[idxs]
        Y = y_seeds[idxs]
```

```
In [5]: X += np.random.normal(scale = 0.25, size = X.shape)
```

Model

Model (torch)

```
class shallow_neural_network():
    def __init__(self, num_input_features, num_hiddens):
        self.num_input_features = num_input_features
        self.num_hiddens = num_hiddens

        self.W1 = torch.randn((num_hiddens, num_input_features), requires_grad = True)
        self.b1 = torch.randn(num_hiddens, requires_grad = True)
        self.W2 = torch.randn(num_hiddens, requires_grad = True)
        self.b2 = torch.randn(1, requires_grad = True)

        self.tanh = torch.nn.Tanh()
        self.sigmoid = torch.nn.Sigmoid()


    def predict(self,x):
        z1 = torch.matmul(self.W1,x) + self.b1
        a1 = self.tanh(z1)
        z2 = torch.matmul(self.W2,a1)+ self.b2
        a2 = self.sigmoid(z2)
        return a2
```

Training

```python
def train(X, Y, model, lr = 0.1):
    m = len(X)

    cost = 0.0
    for x,y in zip(X,Y):
        x_torch = torch.from_numpy(x)

        a2 = model.predict(x_torch)
        if y == 1:
            loss = -torch.log(a2+0.0001)
        else:
            loss = -torch.log(1.0001-a2)

        loss.backward()
        cost += loss.item()

    with torch.no_grad():
        model.W1 -= lr * model.W1.grad/m
        model.b1 -= lr * model.b1.grad/m
        model.W2 -= lr * model.W2.grad/m
        model.b2 -= lr * model.b2.grad/m

    model.W1.requires_grad = True
    model.b1.requires_grad = True
    model.W2.requires_grad = True
    model.b2.requires_grad = True

    return cost/m
```

```python
for epoch in range(100):
    cost = train(X,Y, model, 1.0)
    if epoch % 10 == 0:
        print(epoch, cost)
```

```
0 0.6836582199335098
10 0.3127332235444337
20 0.21746384638389588
30 0.23482400209485924
40 0.27920054577931797
50 0.29317299860637286
60 0.3011668978813414
70 0.28102042431628615
80 0.29269537733129686
90 0.2781521064764529
```

Testing

```
print(model.predict(torch.Tensor((0,0))))
print(model.predict(torch.Tensor((0,1))))
print(model.predict(torch.Tensor((1,0))))
print(model.predict(torch.Tensor((1,1))))
```

```
tensor([1.8461e-35], grad_fn=<SigmoidBackward>)
tensor([1.], grad_fn=<SigmoidBackward>)
tensor([1.], grad_fn=<SigmoidBackward>)
tensor([0.1467], grad_fn=<SigmoidBackward>)
```

nn.Module

Modules

- PyTorch uses modules to represent neural networks
- Modules are:
  - Building blocks of computations
    - A module represents a node or a subgraph in a computation graph
  - Tightly integrated with `autograd`
    - Make it simple to specify learnable parameters
  - Easy to work with
    - Save, restore, transfer between CPU/GPU …

A simple custom module

```
import torch
from torch import nn

class MyLinear(nn.Module):
  def __init__(self, in_features, out_features):
    super().__init__()
    self.weight = nn.Parameter(torch.randn(in_features, out_features))
    self.bias = nn.Parameter(torch.randn(out_features))

  def forward(self, input):
    return (input @ self.weight) + self.bias
```

Call forward

```
m = MyLinear(4, 3)
sample_input = torch.randn(4)
m(sample_input)
: tensor([-0.3037, -1.0413, -4.2057], grad_fn=<AddBackward0>)
```

Modules as Building Blocks

```
import torch.nn.functional as F

class Net(nn.Module):
  def __init__(self):
    super().__init__()
    self.l0 = nn.Linear(4,3)
    self.l1 = nn.Linear(3,1)
  def forward(self, x):
    x = self.l0(x)
    x = F.relu(x)
    x = self.l1(x)
    return x
```

Training with Modules

- Initialize
  - model = YourModel()
  - optimizer = torch.optim.SGD(model.parameters(), lr = <learning_rate>
- Forward
  - y_hat = model(input)
- Backward
  - loss = compute_loss(y, y_hat)   //You can use either a built-in loss or your own loss
  - model.zero_grad()
  - loss.backward()
  - optimizer.step()

Finally..
Implementing a Shallow NN with autograd and nn.Module

Model (torch.nn.Module)

```
class shallow_neural_network(nn.Module):
    def __init__(self, num_input_features, num_hiddens):
        super().__init__()
        self.num_input_features = num_input_features
        self.num_hiddens = num_hiddens

        self.linear1 = nn.Linear(num_input_features, num_hiddens)
        self.linear2 = nn.Linear(num_hiddens, 1)

        self.tanh = torch.nn.Tanh()
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self,x):
        z1 = self.linear1(x)
        a1 = self.tanh(z1)
        z2 = self.linear2(a1)
        a2 = self.sigmoid(z2)
        return a2
```

Training

```python
num_epochs = 100
lr = 1.0
num_hiddens = 3

model = shallow_neural_network(2,num_hiddens)
optimizer = optim.SGD(model.parameters(), lr = lr)
loss = nn.BCELoss()
```

```python
for epoch in range(num_epochs):
    optimizer.zero_grad()

    cost  = 0.0
    for x,y in zip(X,Y):
        x_torch = torch.from_numpy(x)
        y_torch = torch.FloatTensor([y])

        y_hat = model(x_torch)

        loss_val = loss(y_hat, y_torch)
        cost += loss_val

    cost = cost / len(X)
    cost.backward()
    optimizer.step()

    if epoch %10==0:
        print(epoch, cost)
```

Test

```
for x,y in zip(x_seeds,y_seeds):
    print(x)
    x_torch = torch.FloatTensor(x)
    y_hat = model(x_torch)
    print(y, y_hat.item())
```

```
[0. 0.]
0 0.19255302846431732
[1. 0.]
1 0.702729344367981
[0. 1.]
1 0.7534303665161133
[1. 1.]
0 0.1685885339975357
```

Wrap up

Shallow NN: Model

```python
class shallow_neural_network(nn.Module):
    def __init__(self, num_input_features, num_hiddens):
        super().__init__()
        self.num_input_features = num_input_features
        self.num_hiddens = num_hiddens

        self.linear1 = nn.Linear(num_input_features, num_hiddens)
        self.linear2 = nn.Linear(num_hiddens, 1)

        self.tanh = torch.nn.Tanh()
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self,x):
        z1 = self.linear1(x)
        a1 = self.tanh(z1)
        z2 = self.linear2(a1)
        a2 = self.sigmoid(z2)
        return a2
```

Forward pass

Inherits nn.Module (custom module) (p.46~50)

Modules as Building Blocks (p. 49)

Modules as Building Blocks (p. 49)

Shallow NN: Training

```
num_epochs = 100
lr = 1.0
num_hiddens = 3

model  = shallow_neural_network(2,num_hiddens)
optimizer = optim.SGD(model.parameters(), lr = lr)
loss = nn.BCELoss()
```

Training with modules - initialization (p. 51, p. 54)

```
for epoch in range(num_epochs):
    optimizer.zero_grad()

    cost   = 0.0
    for x,y in zip(X,Y):
        x_torch = torch.from_numpy(x)
        y_torch = torch.FloatTensor([y])

        y_hat  = model(x_torch)

        loss_val  = loss(y_hat, y_torch)
        cost += loss_val

    cost  = cost / len(X)
    cost.backward()
    optimizer.step()

    if epoch %10==0:
        print(epoch, cost)
```

Autograd (p. 36)
Training with modules (p. 51)

PyTorch

Has much more powerful functions
to build and train deep neural networks