

Ứng dụng Trí tuệ nhân tạo trong Nuôi trồng thủy sản

NGUYỄN HẢI TRIỀU¹

¹ Bộ môn Kỹ thuật phần mềm,
Khoa Công nghệ thông tin, Trường ĐH Nha Trang

NhaTrang, September 2024

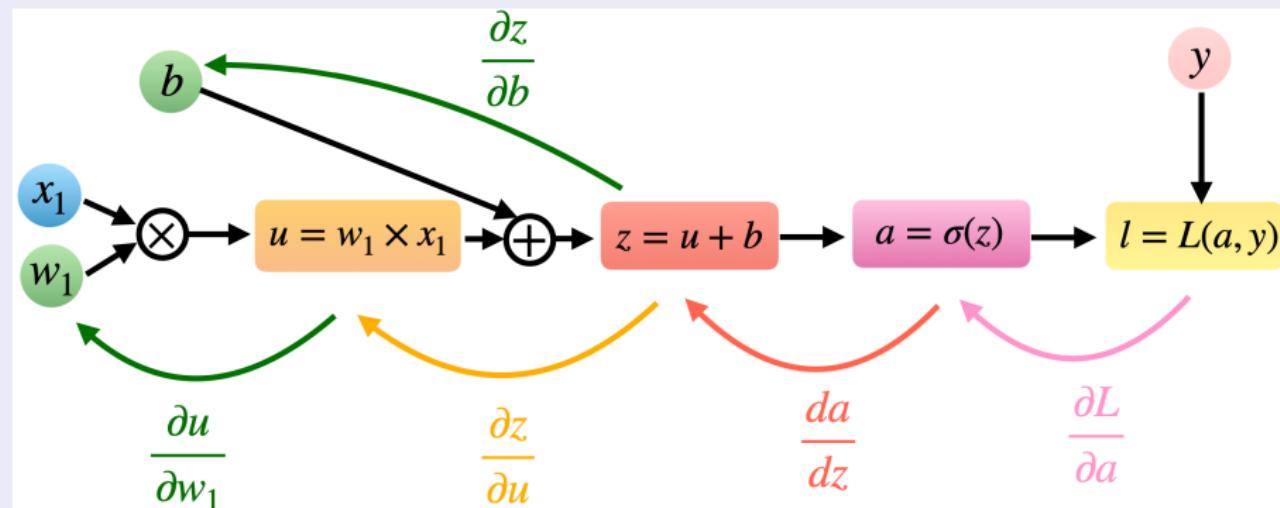
1 Using Logistic Regression for Classification

- Single Layer Neural Networks
- Activation Function
- Logistic regression loss function
- Model Training with Stochastic Gradient Descent
- Automatic Differentiation in PyTorch
- The PyTorch API

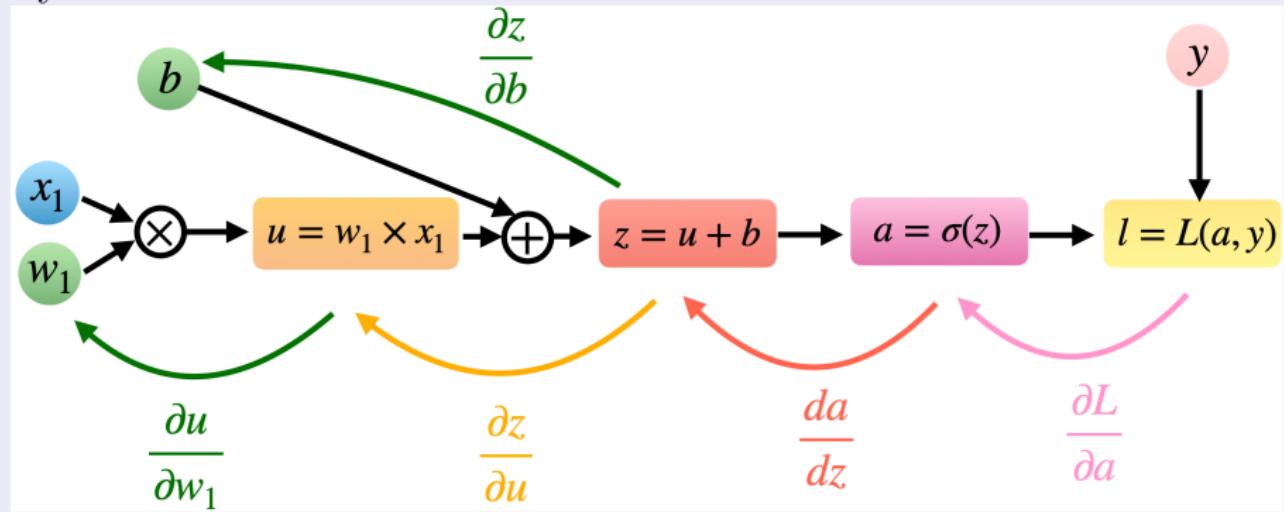
Automatic Differentiation in PyTorch

Trong phần này chúng ta sẽ tìm hiểu cách tính đạo hàm riêng tự động trong PyTorch cho mô hình Logistic regression.

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial b} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial w_1} \times \frac{\partial z}{\partial u} \times \frac{\partial a}{\partial z} \times \frac{\partial L}{\partial a} \\ \frac{\partial z}{\partial b} \times \frac{\partial a}{\partial z} \times \frac{\partial L}{\partial a} \end{bmatrix} \quad (1)$$



How we can compute these gradients automatically by using PyTorch?



Code PyTorch minh họa tính đạo hàm riêng của hàm Loss theo sơ đồ tính toán của mô hình Logistic regression

```

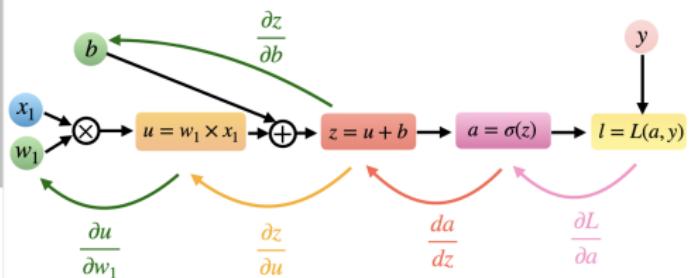
1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ", z)
13 # compute sigmoid activation function
14 a = torch.sigmoid(z)
15 print("a = ", a)
16 # compute the loss of logistic regression model
17 import torch.nn.functional as F
18 l = F.binary_cross_entropy(a,y)
19 print('l = ', l)
20 # compute gradients
21 from torch.autograd import grad
22 grad_L_w_1 = grad(l,w_1,retain_graph=True)
23 print("dL/dw_1 = ", grad_L_w_1)
24 grad_L_b = grad(l,b,retain_graph=True)
25 print("dL/db = ", grad_L_b)

```

```

... z = tensor([1.9000], grad_fn=<AddBackward0>)
a = tensor([0.8699], grad_fn=<SigmoidBackward0>)
l = tensor(0.1394, grad_fn=<BinaryCrossEntropyBackward0>)
dL/dw_1 = (tensor([-0.4814]),)
dL/db = (tensor([-0.1301]),)

```



Chúng ta sử dụng `torch.tensor()` để khởi tạo giá trị cho các tham số mô hình. Lưu ý: ta sử dụng `requires_grad=True` với mục đích **cần tính gradient** của hàm Loss theo các tham số này.

```

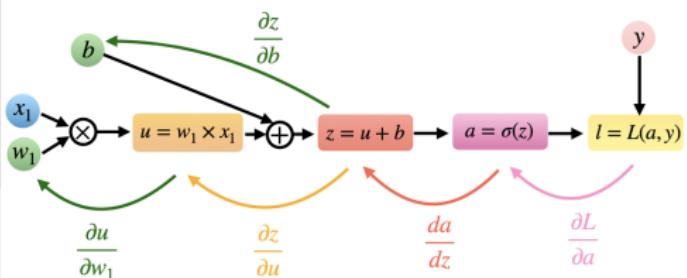
1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ", z)
13 # compute sigmoid activation function
14 a = torch.sigmoid(z)
15 print("a = ", a)
16 # compute the loss of logistic regression model
17 import torch.nn.functional as F
18 l = F.binary_cross_entropy(a,y)
19 print('l = ', l)
20 # compute gradients
21 from torch.autograd import grad
22 grad_L_w_1 = grad(l,w_1,retain_graph=True)
23 print("dL/dw_1 = ", grad_L_w_1)
24 grad_L_b = grad(l,b,retain_graph=True)
25 print("dL/db = ", grad_L_b)

```

```

... z = tensor([1.9000], grad_fn=<AddBackward0>)
a = tensor([0.8699], grad_fn=<SigmoidBackward0>)
l = tensor(0.1394, grad_fn=<BinaryCrossEntropyBackward0>)
dL/dw_1 = (tensor([-0.4814]),)
dL/db = (tensor([-0.1301]),)

```



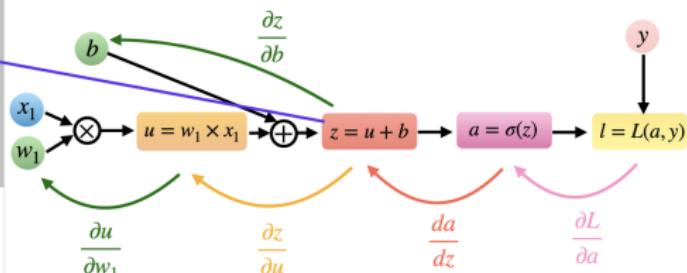
Tiếp theo chúng ta tính *weighted sum u* và *net input z* như sơ đồ tính toán.

```

1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ", z)
13 # compute sigmoid activation function
14 a = torch.sigmoid(z)
15 print("a = ", a)
16 # compute the loss of logistic regression model
17 import torch.nn.functional as F
18 l = F.binary_cross_entropy(a,y)
19 print('l = ',l)
20 # compute gradients
21 from torch.autograd import grad
22 grad_L_w_1 = grad(l,w_1,retain_graph=True)
23 print("dL/dw_1 = ", grad_L_w_1)
24 grad_L_b = grad(l,b,retain_graph=True)
25 print("dL/db = ", grad_L_b)

```

...
 $z = \text{tensor}(1.9000, \text{grad_fn}=\text{<AddBackward0>})$
 $a = \text{tensor}(0.8699, \text{grad_fn}=\text{<SigmoidBackward0>})$
 $l = \text{tensor}(0.1394, \text{grad_fn}=\text{<BinaryCrossEntropyBackward0>})$
 $dL/dw_1 = (\text{tensor}([-0.4814]),)$
 $dL/db = (\text{tensor}([-0.1301]),)$



Áp dụng sigmoid activation ta thu được kết quả $a = 0.8699$.

```

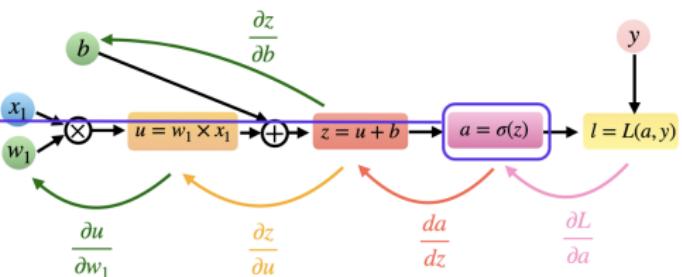
1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ", z)
13 # compute sigmoid activation function
14 a = torch.sigmoid(z)
15 print("a = ", a)
16 # compute the loss of logistic regression model
17 import torch.nn.functional as F
18 l = F.binary_cross_entropy(a,y)
19 print('l = ',l)
20 # compute gradients
21 from torch.autograd import grad
22 grad_L_w_1 = grad(l,w_1,retain_graph=True)
23 print("dL/dw_1 = ", grad_L_w_1)
24 grad_L_b = grad(l,b,retain_graph=True)
25 print("dL/db = ", grad_L_b)

```

```

... z = tensor([1.9000], grad_fn=<AddBackward0>)
a = tensor([0.8699], grad_fn=<SigmoidBackward0>)
l = tensor(0.1394, grad_fn=<BinaryCrossEntropyBackward0>)
dL/dw_1 = (tensor([-0.4814]),)
dL/db = (tensor([-0.1301]),)

```



Loss của Logistic regression được tính bằng thư viện `import torch.nn.functional as F`. Trong PyTorch, hàm Loss của Logistic regression có tên gọi là `.binary_cross_entropy()` -> $l = 0.1394$

```

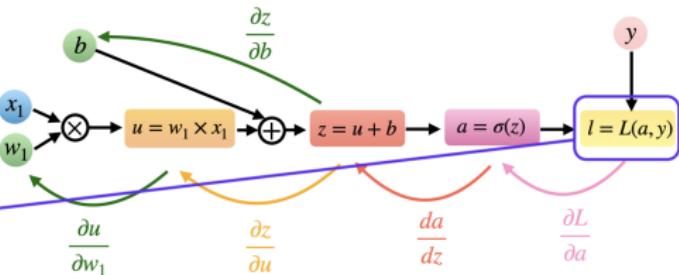
1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ",z)
13 # compute sigmoid activation function
14 a = torch.sigmoid(z)
15 print("a = ", a)
16 # compute the loss of logistic regression model
17 import torch.nn.functional as F
18 l = F.binary_cross_entropy(a,y)
19 print('l = ',l)
20 # compute gradients
21 from torch.autograd import grad
22 grad_L_w_1 = grad(l,w_1,retain_graph=True)
23 print("dL/dw_1 = ", grad_L_w_1)
24 grad_L_b = grad(l,b,retain_graph=True)
25 print("dL/db = ", grad_L_b)

```

```

... z = tensor([1.9000], grad_fn=<AddBackward0>)
a = tensor([0.8699], grad_fn=<SigmoidBackward0>)
l = tensor(0.1394, grad_fn=<BinaryCrossEntropyBackward0>)
dL/dw_1 = (tensor([-0.4814]),)
dL/db = (tensor([-0.1301]),)

```



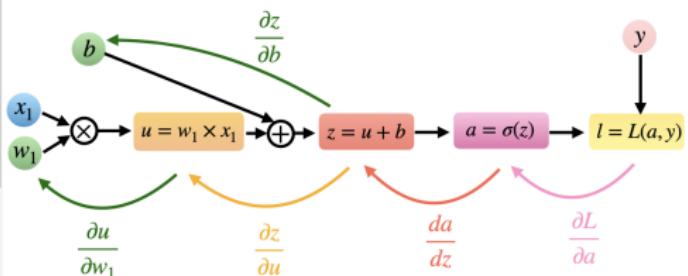
Quá trình tính toán tới giá trị hàm Loss thông thường được gọi là *Forward propagation*

```

1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ", z)
13 # compute sigmoid activation function
14 a = torch.sigmoid(z)
15 print("a = ", a)
16 # compute the loss of logistic regression model
17 import torch.nn.functional as F
18 l = F.binary_cross_entropy(a,y)
19 print('l = ',l)
20 # compute gradients
21 from torch.autograd import grad
22 grad_L_w_1 = grad(l,w_1,retain_graph=True)
23 print("dL/dw_1 = ", grad_L_w_1)
24 grad_L_b = grad(l,b,retain_graph=True)
25 print("dL/db = ", grad_L_b)

```

...
 $z = \text{tensor}(1.9000, \text{grad_fn}=<\text{AddBackward0}>)$
 $a = \text{tensor}(0.8699, \text{grad_fn}=<\text{SigmoidBackward0}>)$
 $l = \text{tensor}(0.1394, \text{grad_fn}=<\text{BinaryCrossEntropyBackward0}>)$
 $dL/dw_1 = (\text{tensor}([-0.4814]),)$
 $dL/db = (\text{tensor}([-0.1301]),)$



Bây giờ chúng ta sẽ tính đạo hàm riêng của loss L theo các tham số mô hình, quá trình này được gọi là *Backward propagation*.

```

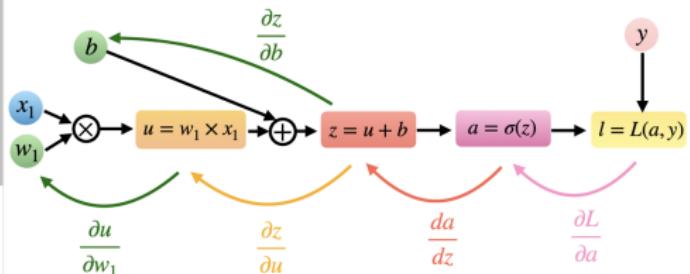
1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ", z)
13 # compute sigmoid activation function
14 a = torch.sigmoid(z)
15 print("a = ", a)
16 # compute the loss of logistic regression model
17 import torch.nn.functional as F
18 l = F.binary_cross_entropy(a,y)
19 print('l = ', l)
20 # compute gradients
21 from torch.autograd import grad
22 grad_L_w_1 = grad(l,w_1,retain_graph=True)
23 print("dL/dw_1 = ", grad_L_w_1)
24 grad_L_b = grad(l,b,retain_graph=True)
25 print("dL/db = ", grad_L_b)

```

```

... z = tensor([1.9000], grad_fn=<AddBackward0>)
a = tensor([0.8699], grad_fn=<SigmoidBackward0>)
l = tensor(0.1394, grad_fn=<BinaryCrossEntropyBackward0>)
dL/dw_1 = (tensor([-0.4814]),)
dL/db = (tensor([-0.1301]),)

```



Backward propagation

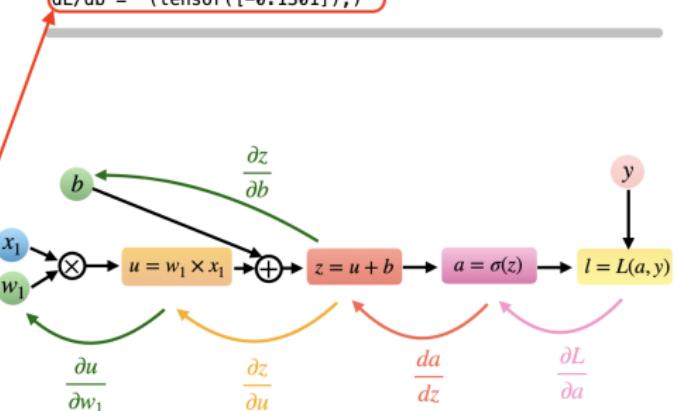
Chúng ta có thể tính gradient theo các tham số w_1 , b một cách tự động bằng cách sử dụng hàm `grad()` từ thư viện `from torch.autograd import grad`

```

1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ", z)
13 # compute sigmoid activation function
14 a = torch.sigmoid(z)
15 print("a = ", a)
16 # compute the loss of logistic regression model
17 import torch.nn.functional as F
18 l = F.binary_cross_entropy(a,y)
19 print('l = ',l)
20 # compute gradients
21 from torch.autograd import grad
22 grad_l_w_1 = grad(l,w_1,retain_graph=True)
23 print("dL/dw_1 = ", grad_l_w_1)
24 grad_l_b = grad(l,b,retain_graph=True)
25 print("dL/db = ", grad_l_b)

```

...
 $z = \text{tensor}(1.9000)$, grad_fn=<AddBackward0>
 $a = \text{tensor}(0.8699)$, grad_fn=<SigmoidBackward0>
 $l = \text{tensor}(0.1394)$, grad_fn=<BinaryCrossEntropyBackward0>
 $dL/dw_1 = (\text{tensor}([-0.4814]),)$
 $dL/db = (\text{tensor}([-0.1301]),)$



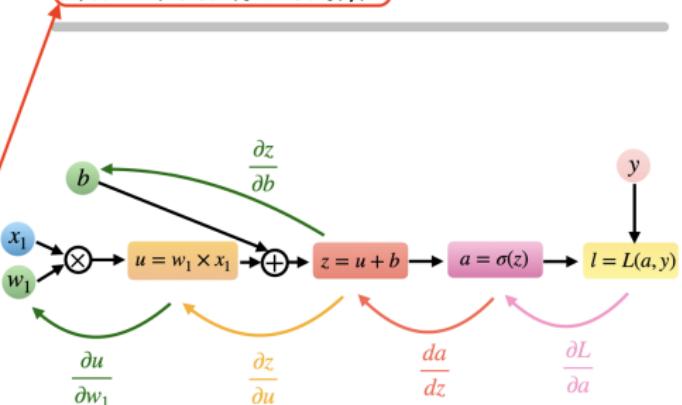
Khi sử dụng hàm `grad()`, ta có truyền vào đối số `retain_graph=True` với mục đích muốn lưu trữ quá trình tính toán trong bộ nhớ. Nếu không sử dụng, PyTorch sẽ tự động huỷ.

```

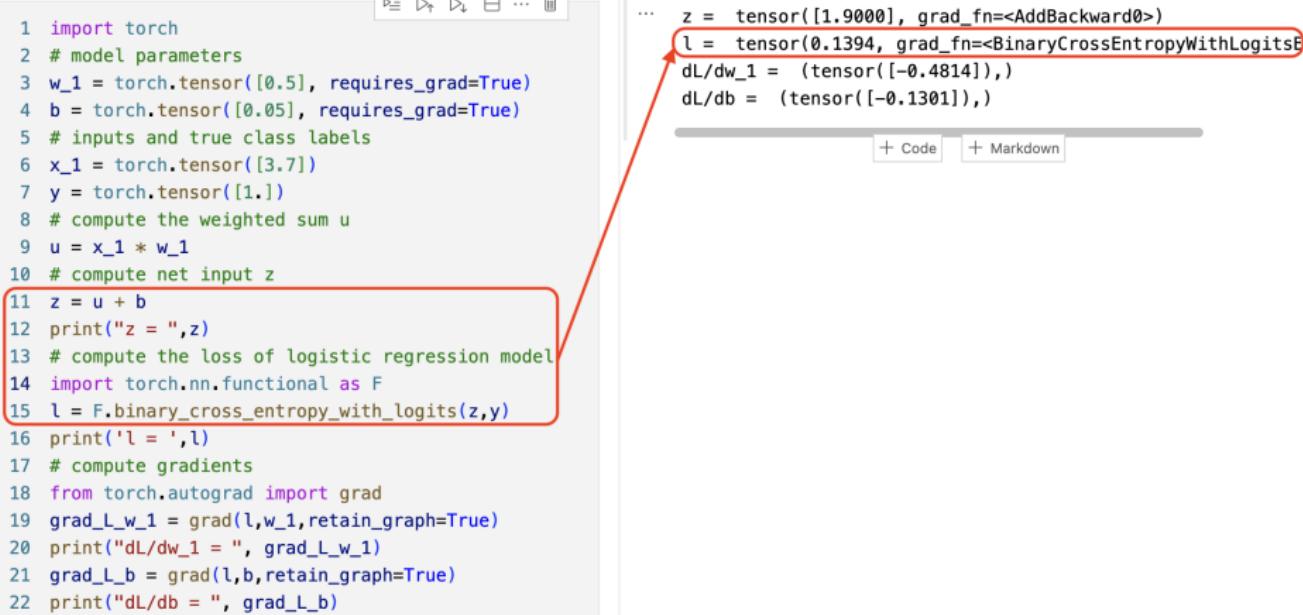
1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ", z)
13 # compute sigmoid activation function
14 a = torch.sigmoid(z)
15 print("a = ", a)
16 # compute the loss of logistic regression model
17 import torch.nn.functional as F
18 l = F.binary_cross_entropy(a,y)
19 print('l = ',l)
20 # compute gradients
21 from torch.autograd import grad
22 grad_L_w_1 = grad(l,w_1,retain_graph=True)
23 print("dL/dw_1 = ", grad_L_w_1)
24 grad_L_b = grad(l,b,retain_graph=True)
25 print("dL/db = ", grad_L_b)

```

...
 $z = \text{tensor}(1.9000)$, grad_fn=<AddBackward0>
 $a = \text{tensor}(0.8699)$, grad_fn=<SigmoidBackward0>
 $l = \text{tensor}(0.1394)$, grad_fn=<BinaryCrossEntropyBackward0>
 $dL/dw_1 = (\text{tensor}([-0.4814]),)$
 $dL/db = (\text{tensor}([-0.1301]),)$



Ở bước tính toán giá trị của L , chúng ta có thể bỏ qua bước tính sigmoid activation và sử dụng hàm kết hợp $binary_cross_entropy_with_logits(z,y)$, kết quả hàm L không đổi.



```

1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ",z)
13 # compute the loss of logistic regression model
14 import torch.nn.functional as F
15 l = F.binary_cross_entropy_with_logits(z,y)
16 print('l = ',l)
17 # compute gradients
18 from torch.autograd import grad
19 grad_L_w_1 = grad(l,w_1,retain_graph=True)
20 print("dL/dw_1 = ", grad_L_w_1)
21 grad_L_b = grad(l,b,retain_graph=True)
22 print("dL/db = ", grad_L_b)

```

...
 $z = \text{tensor}(1.9000, \text{grad_fn}=<\text{AddBackward0}>)$
 $l = \text{tensor}(0.1394, \text{grad_fn}=<\text{BinaryCrossEntropyWithLogitsE}$
 $dL/dw_1 = (\text{tensor}([-0.4814]),)$
 $dL/db = (\text{tensor}([-0.1301]),)$

Ở bước tính toán đạo hàm riêng, chúng ta có thể gọi nhanh phương thức `.backward()`, thay vì đi tính đạo hàm từng thành phần theo hàm $grad(l, w_1)$, $grad(l, b)$.

```

1 import torch
2 # model parameters
3 w_1 = torch.tensor([0.5], requires_grad=True)
4 b = torch.tensor([0.05], requires_grad=True)
5 # inputs and true class labels
6 x_1 = torch.tensor([3.7])
7 y = torch.tensor([1.])
8 # compute the weighted sum u
9 u = x_1 * w_1
10 # compute net input z
11 z = u + b
12 print("z = ",z)
13 # compute the loss of logistic regression model
14 import torch.nn.functional as F
15 l = F.binary_cross_entropy_with_logits(z,y)
16 print('l = ',l)
17 # compute gradients
18 l.backward()
19 print("backward: dL/dw_1 = ", w_1.grad)
20 print("backward: dL/db = ", b.grad)

```

```

... z = tensor([1.9000], grad_fn=<AddBackward0>)
      l = tensor(0.1394, grad_fn=<BinaryCrossEntropyWithLogitsE
backward: dL/dw_1 = tensor([-0.4814])
backward: dL/db = tensor([-0.1301])

```

Kết quả thu được là như nhau so với gọi hàm `grad()`.

Sử dụng PyTorch để huấn luyện mô hình

Cấu trúc tổng quát của các mô hình Deep Neural Networks được xây dựng trong PyTorch

```
1 import torch
2
3 class MyClassifier(torch.nn.Module):
4     def __init__(self):
5         # define model parameters
6
7     def forward(self, x):
8         # define how the model
9         # produces outputs
10
11         return outputs
```

Sử dụng PyTorch để huấn luyện mô hình

In PyTorch, we define **models as classes**. Here our *myClassifier* class inherits from *torch.nn.module*, which will give it a certain set of properties, which will be helpful later for the model training.

```
1 import torch
2
3 class MyClassifier(torch.nn.Module):
4     def __init__(self):
5         # define model parameters
6
7     def forward(self, x):
8         # define how the model
9         # produces outputs
10
11    return outputs
```

Sử dụng PyTorch để huấn luyện mô hình

We have this *init constructor*, where we would define the model parameters.

```
1 import torch
2
3 class MyClassifier(torch.nn.Module):
4     def __init__(self):
5         # define model parameters
6
7     def forward(self, x):
8         # define how the model
9         # produces outputs
10
11     return outputs
```

Sử dụng PyTorch để huấn luyện mô hình

Then we have this *forward method*, where we pass in a data point x , and here we would define how the model does the predictions.

```
1 import torch
2
3 class MyClassifier(torch.nn.Module):
4     def __init__(self):
5         # define model parameters
6
7     def forward(self, x):
8         # define how the model
9         # produces outputs
10
11     return outputs
```

Dựa vào cấu trúc tổng quát, chúng ta huấn luyện cho bài toán cụ thể Logistic regression. Trong trường hợp này, *myClassifier* class là Logistic regression.

```
1 model = MyClassifier()
2 optimizer = torch.optim.SGD(...)
3
4 for epoch in range(num_epochs):
5     for x, y in train_dataloader:
6
7         # forward pass
8         outputs = model(x)
9         loss = loss_fn(outputs, y)
```

Hình 1: The training API

Đầu tiên, ta khởi tạo *myClassifier* như là model cần huấn luyện.

```
1 model = MyClassifier()  
2 optimizer = torch.optim.SGD(...)  
3  
4 for epoch in range(num_epochs):  
5     for x, y in train_dataloader:  
6  
7         # forward pass  
8         outputs = model(x)  
9         loss = loss_fn(outputs, y)
```

Tiếp theo, ta khởi tạo *optimizer* là các thuật toán tối ưu huấn luyện mô hình, thường chọn *SGD*–Stochastic Gradient Descent với các *minibatches* trong *train_dataloader*.

```
1 model = MyClassifier()
2 optimizer = torch.optim.SGD(...)
3
4 for epoch in range(num_epochs):
5     for x, y in train_dataloader:
6
7         # forward pass
8         outputs = model(x)
9         loss = loss_fn(outputs, y)
```

Thực hiện theo thuật toán Minibatch Gradient Descent, chúng ta lặp qua các training epochs, tại mỗi epoch, chúng ta lặp qua các minibatches của train_dataloader, ở đó chúng ta sẽ tính output của model và loss.

```
1  model = MyClassifier()  
2  optimizer = torch.optim.SGD(...)  
3  
4  for epoch in range(num_epochs):  
5      for x, y in train_dataloader:  
6  
7          # forward pass  
8          outputs = model(x)  
9          loss = loss_fn(outputs, y)
```

Ta phải gọi `zero_grad()` để thiết lập gradient bằng 0 trước khi bắt đầu tính gradient cho mỗi minibatch.

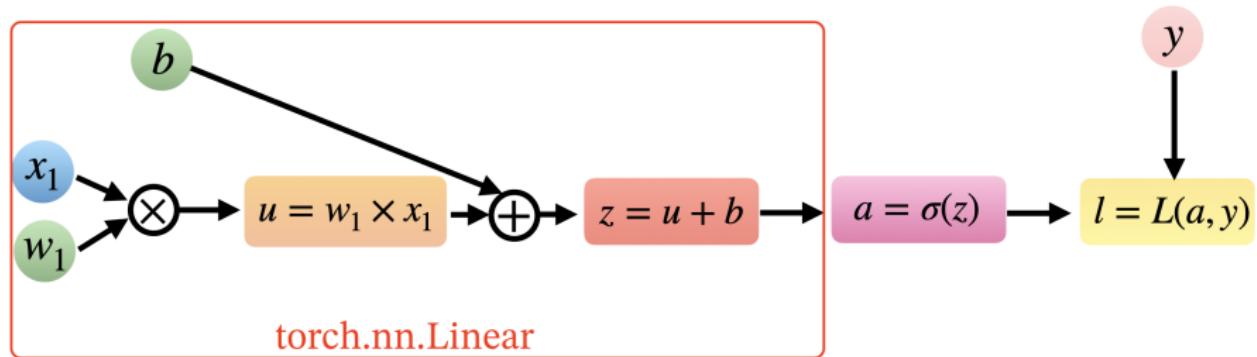
```
1 model = MyClassifier()
2 optimizer = torch.optim.SGD(...)
3
4 for epoch in range(num_epochs):
5     for x, y in train_dataloader:
6
7         # forward pass
8         outputs = model(x)
9         loss = loss_fn(outputs, y)
10
11        # backward pass (backpropagation)
12        optimizer.zero_grad()
13        loss.backward()
14
15        # update model parameters
16        optimizer.step()
```

Bước cuối cùng, chúng ta cập nhật lại tham số của mô hình sau mỗi minibatch bằng cách gọi *optimizer.step()*

```
1 model = MyClassifier()
2 optimizer = torch.optim.SGD(...)
3
4 for epoch in range(num_epochs):
5     for x, y in train_dataloader:
6
7         # forward pass
8         outputs = model(x)
9         loss = loss_fn(outputs, y)
10
11        # backward pass (backpropagation)
12        optimizer.zero_grad()
13        loss.backward()
14
15        # update model parameters
16        optimizer.step()
```

Neural network layers in PyTorch

Chúng ta có thể đơn giản việc xây dựng mô hình ML/DL bằng cách sử dụng **định nghĩa Neural Network Layers** có sẵn trong PyTorch.

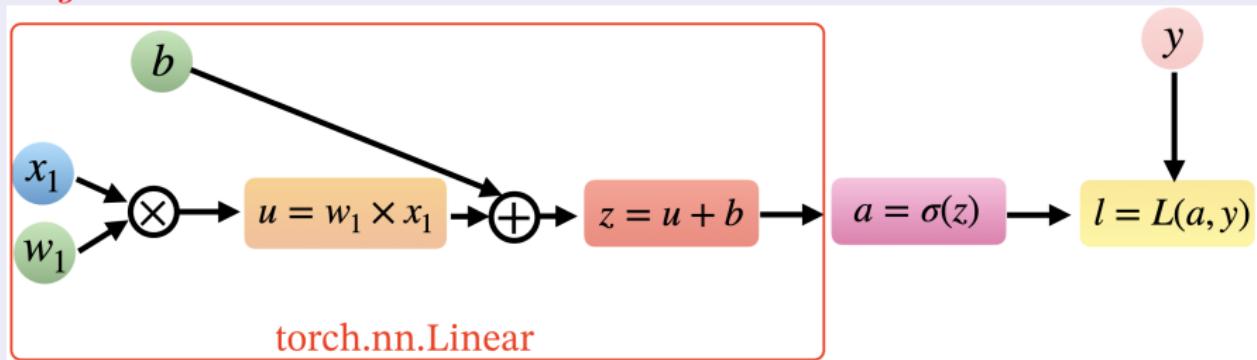


Hình 2: Ví dụ **định nghĩa** lớp **Linear** trong mô hình Logistic regression hoặc trong Deep neural networks nói chung.

Neural network layers in PyTorch

`torch.nn.Linear` là gì?

`torch.nn.Linear` là một neural network layer dùng để tính tổng weighted sum z .



Neural network layers in PyTorch

Tính *weighted sum z*, để đơn giản, chúng ta xét ví dụ khởi tạo một lớp Linear với *chỉ một training example có 2 đặc trưng*. Đầu ra mong muốn của lớp Linear này là một giá trị *z*.

```
1 import torch
2 torch.manual_seed(123)
3 linear = torch.nn.Linear(in_features=2, out_features=1)
4 print('weights -> ', linear.weight)
5 print('bias unit -> ', linear.bias)
```

Vì đầu vào chỉ có 2 features, nên ta truyền *in_features=2* vào *torch.nn.Linear()*, tương tự đầu ra chỉ có một *z* nên *out_features=1*. Lưu ý: *torch.manual_seed(123)* đảm bảo sinh ra ngẫu nhiên cùng một giá trị sau mỗi lần chạy code.

Sau khi khởi tạo lớp *linear*, PyTorch sẽ khởi tạo tự động trọng số mô hình *weight* và *bias unit* tương ứng với các đối số truyền vào *torch.nn.Linear* mà không cần phải khởi tạo bằng tay như phần Perceptron.

```
1 import torch
2 torch.manual_seed(123)
3 linear = torch.nn.Linear(in_features=2, out_features=1)
4 print('weights -> ', linear.weight)
5 print('bias unit -> ', linear.bias)
```

✓ 0.0s

```
weights -> Parameter containing:
tensor([[-0.2883,  0.0234]], requires_grad=True)
bias unit -> Parameter containing:
tensor([-0.3512], requires_grad=True)
```

Chúng ta có thể xem các tham số đã được khởi tạo bằng cách sử dụng *linear.weight* hoặc *.bias*

Tính weighted sum bằng *dot product* $z = \mathbf{x}\mathbf{w}^T + b$

```

1 import torch
2 torch.manual_seed(123)
3 linear = torch.nn.Linear(in_features=2, out_features=1)
4 print('weights -> ', linear.weight)
5 print('bias unit -> ', linear.bias)
6 # input single training example
7 x = torch.tensor([[1.2, 0.5]])
8 # compute weighted sum using matrix multiplication
9 z = torch.matmul(x, linear.weight.T) + linear.bias
10 print('z = ', z)
✓ 1.0s

```

```

weights -> Parameter containing:
tensor([-0.2883,  0.0234]), requires_grad=True)
bias unit -> Parameter containing:
tensor([-0.3512], requires_grad=True)
z = tensor([[-0.6855]], grad_fn=<AddBackward0>)

```

Lưu ý: nếu chỉ có 1 training example, ta biểu diễn \mathbf{x} là một tensor 2 chiều có size là (1×2) .

Chúng ta có cách tính *weighted sum* đơn giản hơn bằng cách sử dụng định nghĩa output của lớp *linear*. Kết quả thu được vẫn như nhau so với nhân ma trận ở trên.

```
1 import torch
2 torch.manual_seed(123)
3 linear = torch.nn.Linear(in_features=2, out_features=1)
4 print('weights -> ', linear.weight)
5 print('bias unit -> ', linear.bias)
6 # input single training example
7 x = torch.tensor([[1.2, 0.5]])
8 # compute weighted sum using Linear layer
9 z = linear(x)
10 print('z = ', z)
```

✓ 0.0s

```
weights -> Parameter containing:
tensor([[-0.2883,  0.0234]], requires_grad=True)
bias unit -> Parameter containing:
tensor([-0.3512], requires_grad=True)
z = tensor([[-0.6855]], grad_fn=<AddmmBackward0>)
```

Xét trường hợp Linear layer cho **một minibatch gồm nhiều training examples với 2 features**. Việc khởi tạo lớp *linear* vẫn như cũ, gồm **2 in_features**, và đầu ra vẫn là **1 out_features**.

```
1 import torch
2 torch.manual_seed(123)
3 linear = torch.nn.Linear(in_features=2 , out_features=1)
4 # input 3 training examples
5 x = torch.tensor([[1.2,0.5],[0.45,1.5],[0.3,2.]])
```

Giả sử chúng ta khởi tạo tensor x có 3 training examples.

Kết quả tính weighted sum bằng *dot product* và sử dụng *output* của lớp *linear* vẫn như nhau.

```

1 import torch
2 torch.manual_seed(123)
3 linear = torch.nn.Linear(in_features=2,
4   out_features=1)
5 print('weights -> ', linear.weight)
6 print('bias unit -> ', linear.bias)
7 # input 3 training examples
8 x = torch.tensor([[1.2,0.5],[0.45,1.5],[0.3,2.]])
9 # compute weighted sum using matrix mult.
10 z = torch.matmul(x, linear.weight.T) + linear.bias
11 print('z = ',z)
```

✓ 0.0s

Python

weights -> Parameter containing:
`tensor([[-0.2883, 0.0234]], requires_grad=True)`
 bias unit -> Parameter containing:
`tensor([-0.3512], requires_grad=True)`

`z = tensor([[-0.6855, -0.4458], [-0.3908]], grad_fn=<AddBackward0>)`

```

1 import torch
2 torch.manual_seed(123)
3 linear = torch.nn.Linear(in_features=2,
4   out_features=1)
5 print('weights -> ', linear.weight)
6 print('bias unit -> ', linear.bias)
7 # input 3 training examples
8 x = torch.tensor([[1.2,0.5],[0.45,1.5],[0.3,2.]])
9 # compute weighted sum using Linear output
10 z = linear(x)
11 print('z = ',z)
```

✓ 0.0s

Python

... weights -> Parameter containing:
`tensor([[-0.2883, 0.0234]], requires_grad=True)`
 bias unit -> Parameter containing:
`tensor([-0.3512], requires_grad=True)`

`z = tensor([[-0.6855, -0.4458], [-0.3908]], grad_fn=<AddmmBackward0>)`

Kết quả tính weighted sum bằng *dot product* và sử dụng *output* của lớp *linear* vẫn như nhau.

```

1 import torch
2 torch.manual_seed(123)
3 linear = torch.nn.Linear(in_features=2,
4   out_features=1)
5 print('weights -> ', linear.weight)
6 print('bias unit -> ', linear.bias)
7 # input 3 training examples
8 x = torch.tensor([[1.2,0.5],[0.45,1.5],[0.3,2.]])
9 # compute weighted sum using matrix mult.
10 z = torch.matmul(x, linear.weight.T) + linear.bias
11 print('z = ',z)
```

✓ 0.0s

Python

weights -> Parameter containing:
`tensor([[-0.2883, 0.0234]], requires_grad=True)`
 bias unit -> Parameter containing:
`tensor([-0.3512], requires_grad=True)`

`z = tensor([[-0.6855, -0.4458], [-0.3908]], grad_fn=<AddBackward0>)`

```

1 import torch
2 torch.manual_seed(123)
3 linear = torch.nn.Linear(in_features=2,
4   out_features=1)
5 print('weights -> ', linear.weight)
6 print('bias unit -> ', linear.bias)
7 # input 3 training examples
8 x = torch.tensor([[1.2,0.5],[0.45,1.5],[0.3,2.]])
9 # compute weighted sum using Linear output
10 z = linear(x)
11 print('z = ',z)
```

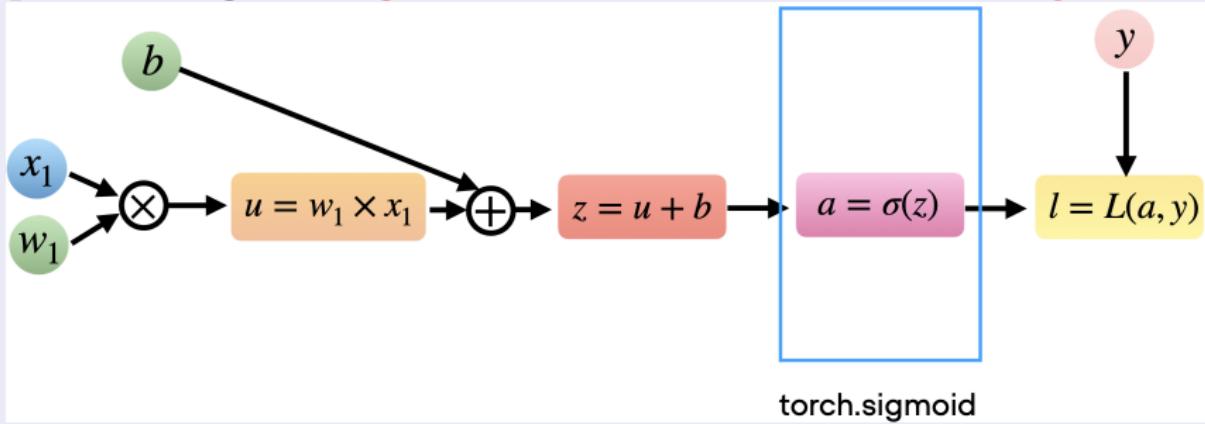
✓ 0.0s

Python

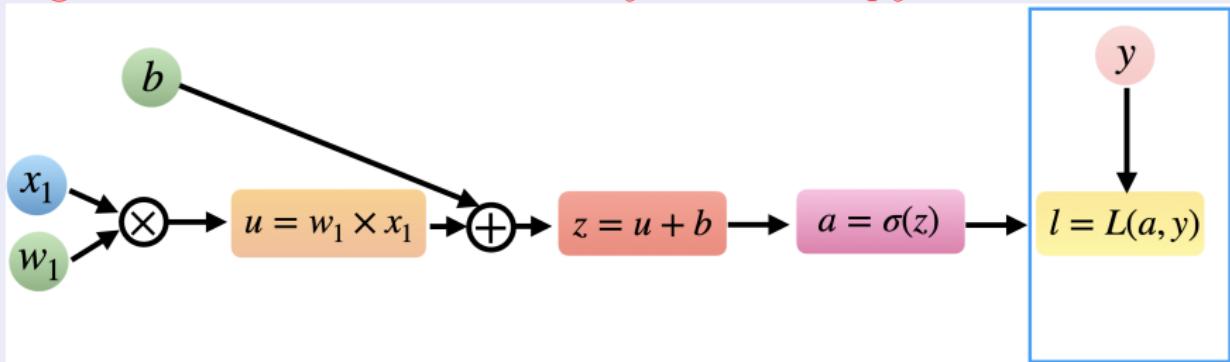
... weights -> Parameter containing:
`tensor([[-0.2883, 0.0234]], requires_grad=True)`
 bias unit -> Parameter containing:
`tensor([-0.3512], requires_grad=True)`

`z = tensor([[-0.6855, -0.4458], [-0.3908]], grad_fn=<AddmmBackward0>)`

Sau khi tính được tổng trọng số, net input z , PyTorch cũng cung cấp cho chúng ta **công cụ tính các hàm kích hoạt như sigmoid**



công cụ tính hàm Loss như *binary cross entropy*



`torch.nn.functional.binary_cross_entropy`

Tài liệu tham khảo

-  **Sebastian Raschka, Yuxi (Hayden) Liu, Vahid Mirjalili**
Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python (2022). Published by Packt Publishing Ltd, ISBN 978-1-80181-931-2.
-  **Sebastian Raschka**
MACHINE LEARNING Q AND AI: 30 Essential Questions and Answers on Machine Learning and AI (2024). ISBN-13: 978-1-7185-0377-9 (ebook).
-  **LightningAI**
LightningAI: PyTorch Lightning (2024) .