PyTorch Tutorial

07. Multiple Dimension Input

多维输入的logistic问题

Revision

x (hours)	y (points)
1	2
2	4
3	6
4	?

x (hours)	y (pass/fail)
1	0 (fail)
2	0 (fail)
3	1 (pass)
4	?

Diabetes Dataset

X1	X2	X3	X4	X5	X6	X7	X8	Y
-0.29	0.49	0.18	-0.29	0.00	0.00	-0.53	-0.03	0
-0.88	-0.15	0.08	-0.41	0.00	-0.21	-0.77	-0.67	1
-0.06	0.84	0.05	0.00	0.00	-0.31	-0.49	-0.63	0
-0.88	-0.11	0.08	-0.54	-0.78	-0.16	-0.92	0.00	1
0.00	0.38	-0.34	-0.29	-0.60	0.28	0.89	-0.60	0
-0.41	0.17	0.21	0.00	0.00	-0.24	-0.89	-0.70	1
-0.65	-0.22	-0.18	-0.35	-0.79	-0.08	-0.85	-0.83	0
0.18	0.16	0.00	0.00	0.00	0.05	-0.95	-0.73	1
-0.76	0.98	0.15	-0.09	0.28	-0.09	-0.93	0.07	0
-0.06	0.26	0.57	0.00	0.00	0.00	-0.87	0.10	0

这是一个糖尿病的数据集

样本

Sample

在数据库中

这是一个关系表

每一行叫作一个 Record（记录）

每一列叫作一个字段

ML中

行为样本，列为特征

Feature、特征

.gz 是 Unix 中的一种压缩格式

数据集准备就是把X(输入)单独放入一个矩阵

把Y(输出)单独放入一个矩阵.

Multiple Dimension Logistic Regression Model

Logistic Regression Model

$$\hat{y}^{(i)} = \sigma(x^{(i)} * \omega + b)$$

向量点乘

↓展开

Logistic Regression Model

$$\hat{y}^{(i)} = \sigma\left(\sum_{n=1}^{8} x_n^{(i)} \cdot \omega_n + b\right)$$

$x_n^{(i)}$：表示第i个特征 第n个数据

ω_n：向量中第n个量（标量）

$$X^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_8^{(i)} \end{bmatrix}$$ 向量

$$X^{(i)T} \times W = X^{(i)T} \cdot W = [x_1^{(i)}, x_2^{(i)}, \cdots, x_8^{(i)}] \begin{bmatrix} w_1^{(i)} \\ w_2 \\ \vdots \\ w_8 \end{bmatrix} = x_1 w_1 + \cdots + x_8 w_8$$ （标量）

~~8分类中，每个特征组数据（输入）先单独求线性回归（即 $\vec{x}_n^{(i)} \cdot \vec{w}_n$ ~~

~~再将其求和作为下面输入~~

~~加上偏置b~~

☆ 8组数据用的是同一个线性模型（$\vec{w}$相同）

Multiple Dimension Logistic Regression Model

Logistic Regression Model

$$\hat{y}^{(i)} = \sigma(x^{(i)} * \omega + b)$$

$$\sum_{n=1}^{8} x_n^{(i)} \cdot \omega_n = \begin{bmatrix} x_1^{(i)} & \cdots & x_N^{(i)} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_8 \end{bmatrix}$$

Logistic Regression Model

$$\hat{y}^{(i)} = \sigma\left(\sum_{n=1}^{8} x_n^{(i)} \cdot \omega_n + b\right)$$

Logistic Regression Model

$$\hat{y}^{(i)} = \sigma\left(\begin{bmatrix} x_1^{(i)} & \cdots & x_8^{(i)} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_8 \end{bmatrix} + b\right)$$

$$= \sigma(z^{(i)})$$

即第i个样本的线性计算结果

Mini-Batch (N samples)

即对每个样本都要单独计算σ（sigma）

$$\begin{bmatrix} \hat{y}^{(1)} \\ \vdots \\ \hat{y}^{(N)} \end{bmatrix} = \begin{bmatrix} \sigma(z^{(1)}) \\ \vdots \\ \sigma(z^{(N)}) \end{bmatrix} = \sigma\left(\begin{bmatrix} z^{(1)} \\ \vdots \\ z^{(N)} \end{bmatrix}\right)$$

Sigmoid function is in an element-wise fashion.

注：在 numpy，torch 中的函数都是以向量为输入的函数.

Mini-Batch (N samples)

$$\begin{bmatrix}\hat{y}^{(1)}\\ \vdots \\ \hat{y}^{(N)}\end{bmatrix} = \begin{bmatrix}\sigma(z^{(1)})\\ \vdots \\ \sigma(z^{(N)})\end{bmatrix} = \sigma\left(\begin{bmatrix}z^{(1)}\\ \vdots \\ z^{(N)}\end{bmatrix}\right)$$

Sigmoid function is in an element-wise fashion.

$$z^{(1)} = \begin{bmatrix}x_1^{(1)} & \cdots & x_8^{(1)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \vdots \\ \omega_8\end{bmatrix} + b$$

$$\vdots$$

$$z^{(N)} = \begin{bmatrix}x_1^{(N)} & \cdots & x_8^{(N)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \vdots \\ \omega_8\end{bmatrix} + b$$

$$\begin{bmatrix}z^{(1)}\\ \vdots \\ z^{(N)}\end{bmatrix} = \begin{bmatrix}x_1^{(1)} & \cdots & x_8^{(1)}\\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \cdots & x_8^{(N)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \vdots \\ \omega_8\end{bmatrix} + \begin{bmatrix}b\\ \vdots \\ b\end{bmatrix}$$

$N \times 1$ $N \times 8$ 8×1 $N \times 1$

Mini-Batch (N samples)

$$\begin{bmatrix}\hat{y}^{(1)}\\ \vdots \\ \hat{y}^{(N)}\end{bmatrix} = \begin{bmatrix}\sigma(z^{(1)})\\ \vdots \\ \sigma(z^{(N)})\end{bmatrix} = \sigma\left(\begin{bmatrix}z^{(1)}\\ \vdots \\ z^{(N)}\end{bmatrix}\right)$$

多维输出计算 模型计算的上下的唯一改变

```
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(8, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.linear(x))
        return x

model = Model()
```

$$z^{(1)} = \begin{bmatrix}x_1^{(1)} & \cdots & x_8^{(1)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \vdots \\ \omega_8\end{bmatrix} + b$$

$$\vdots$$

$$z^{(N)} = \begin{bmatrix}x_1^{(N)} & \cdots & x_8^{(N)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \vdots \\ \omega_8\end{bmatrix} + b$$

$$\begin{bmatrix}z^{(1)}\\ \vdots \\ z^{(N)}\end{bmatrix} = \begin{bmatrix}x_1^{(1)} & \cdots & x_8^{(1)}\\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \cdots & x_8^{(N)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \vdots \\ \omega_8\end{bmatrix} + \begin{bmatrix}b\\ \vdots \\ b\end{bmatrix}$$

$N \times 1$ $N \times 8$ 8×1 $N \times 1$

Linear Layer

构造多层神经网络（这两个Linear层都只有1层）

(线性)

```
self.linear = torch.nn.Linear(8, 1)
```

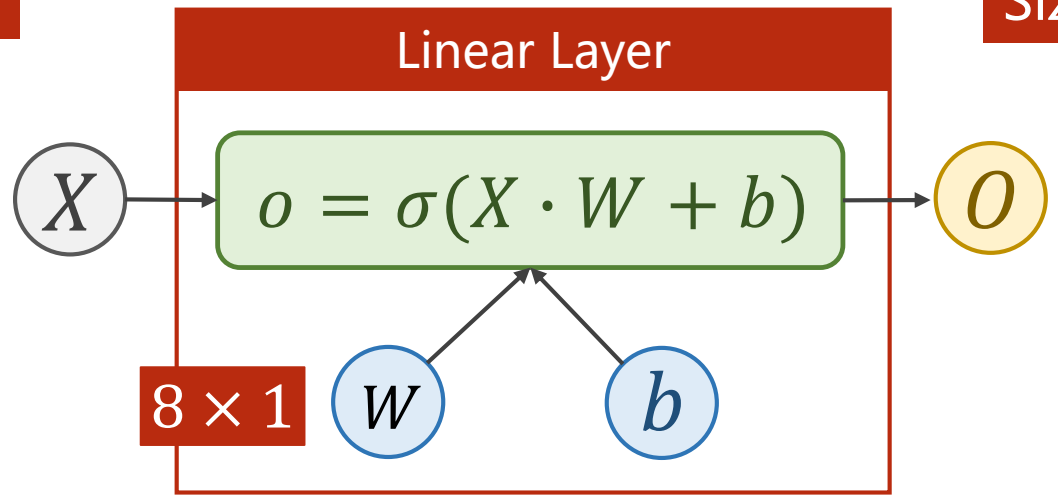
输入维度

输出维度

Size of each input sample

Size of each output sample

$$X = \begin{bmatrix} x_1^{(1)} & \cdots & x_8^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \cdots & x_8^{(N)} \end{bmatrix}$$



$$O = \begin{bmatrix} o^{(1)} \\ \vdots \\ o^{(N)} \end{bmatrix}$$

Linear Layer

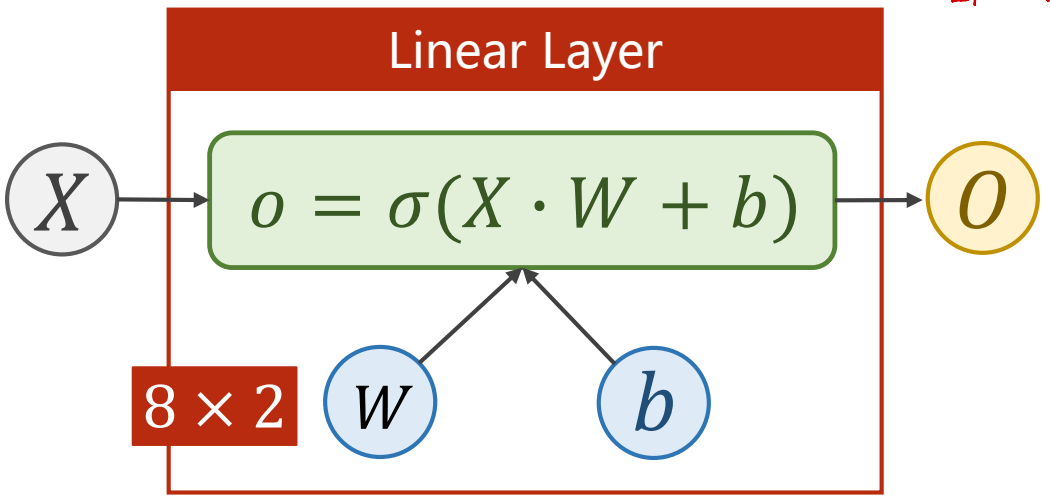
```
self.linear = torch.nn.Linear(8, 2)
```

1 → 2

又因输出只有1维，只需再在后面加入一层

Linear(2,1)

$$X = \begin{bmatrix} x_1^{(1)} & \cdots & x_8^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \cdots & x_8^{(N)} \end{bmatrix}$$

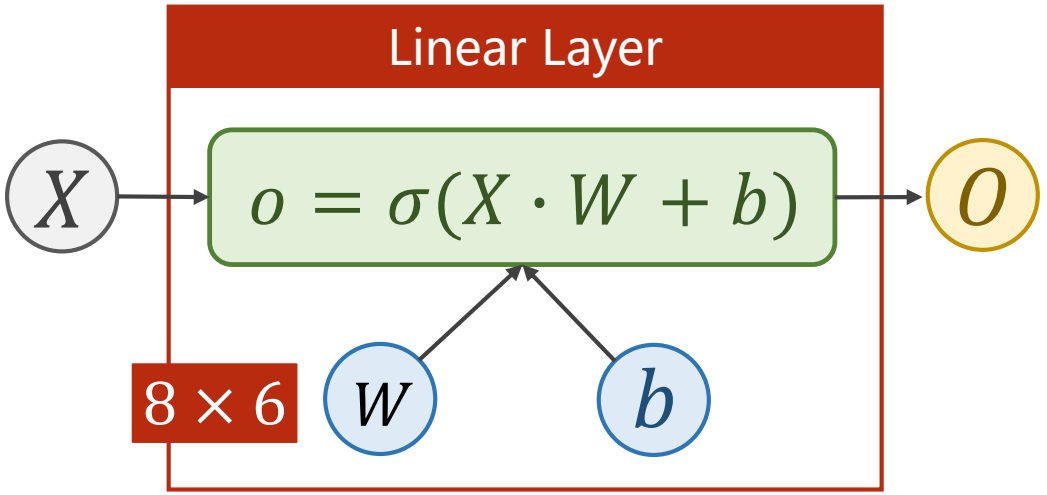


$$O = \begin{bmatrix} o_1^{(1)} & o_2^{(1)} \\ \vdots & \vdots \\ o_1^{(N)} & o_2^{(N)} \end{bmatrix}$$

Linear Layer

```
self.linear = torch.nn.Linear(8, 6)
```

$$X = \begin{bmatrix} x_1^{(1)} & \cdots & x_8^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \cdots & x_8^{(N)} \end{bmatrix}$$



$$O = \begin{bmatrix} o_1^{(1)} & \cdots & o_6^{(1)} \\ \vdots & \ddots & \vdots \\ o_1^{(N)} & \cdots & o_6^{(N)} \end{bmatrix}$$

对于矩阵运算

$$y = Ax$$
$$M\times 1 \quad M\times N \quad N\times 1$$

→ 实际是将 N 维的向量 x 通过 A 映射到 M 维向量 y

⇒ 所以，矩阵运算其实就是一种空间变换，$y=Ax$ 实际就是一个空间变换的函数。

☆ 但 $y=Ax$ 只一种线性的空间变换，但实际应用中空间变换大多是复杂的、非线性的，所以我们就需要用多个线性变换层相互组合，使其变换效果等效于我们想要的非线性变换。

通过不同的权重

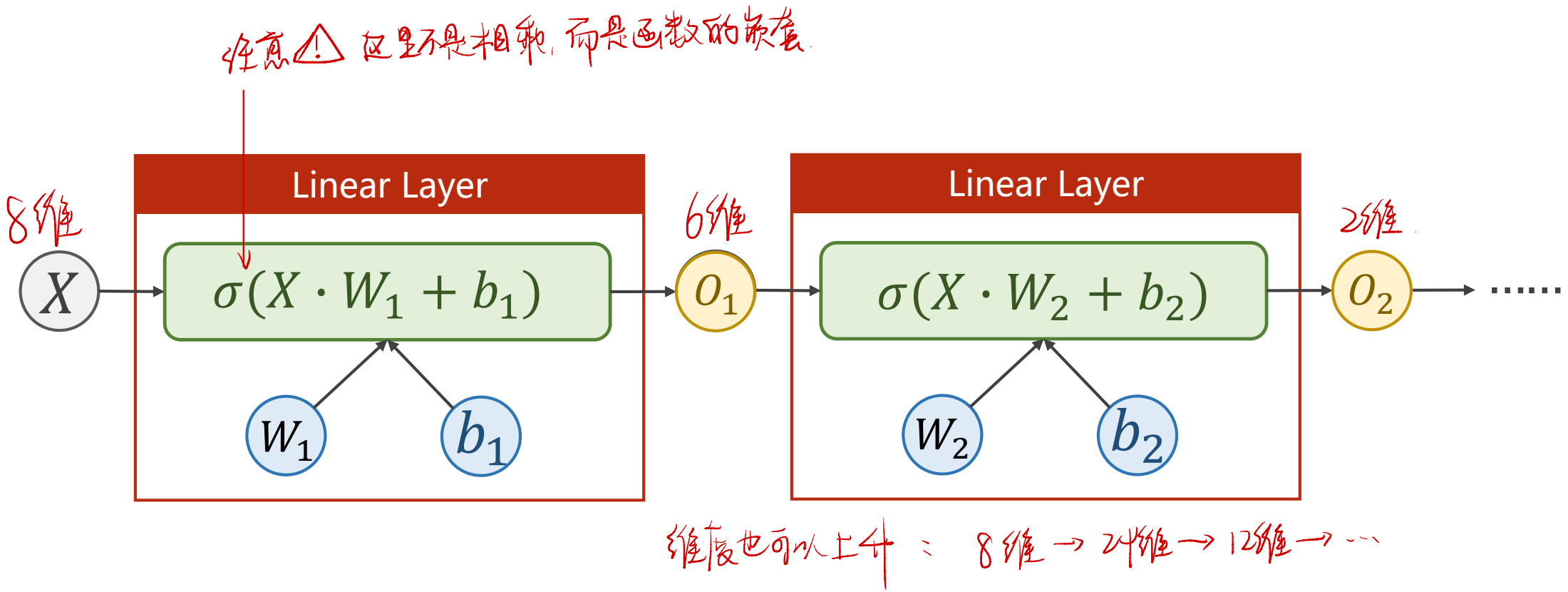
类似人类的大脑用多次求等效其它复杂函数

故神经网络本质上是寻找一种非线性的空间变换函数 → σ（sigma 函数，如 logistic 函数，非线性）就是激活函数

- 注：Linear（X·W+b）本身是不做非线性的，但 σ 是非线性，故每个 Linear 层都要引入一个 σ 函数，才可以模拟我们想要的非线性变换

（只是加 X·W+b 层是没用的，无法改变其线性性质）

Neural Network



维度也可以上升：8维 → 24维 → 128维 → ···

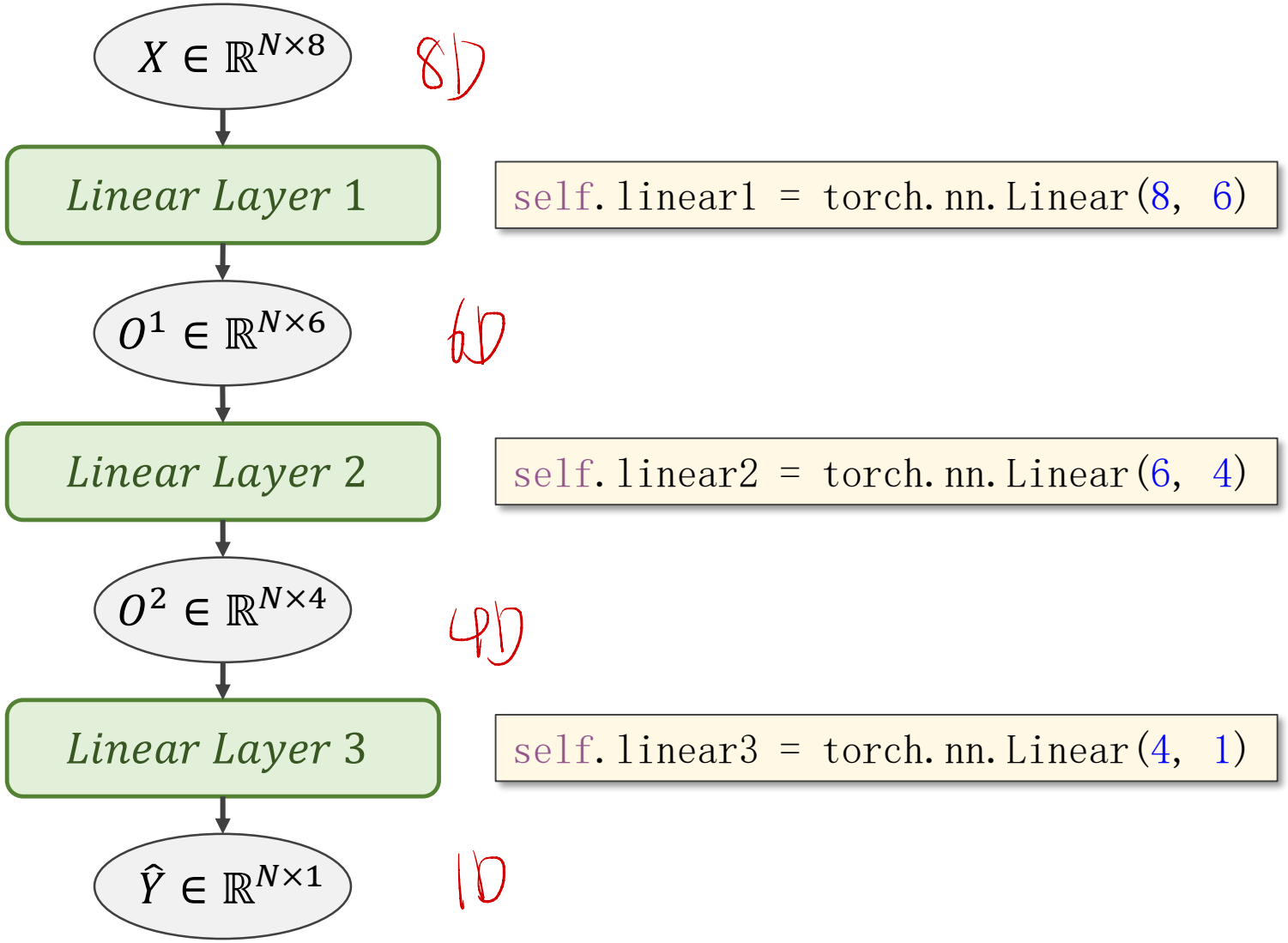
☆ 维度、层数越多，模型越精准，越复杂，学习能力越强，但同时也更容易过拟合（越敏感）

⇒ 模型必须要有泛化能力

学到数据中的噪声

类似于玄学调参，有用没有都肾过了

Example: Artificial Neural Network



Example: Diabetes Prediction

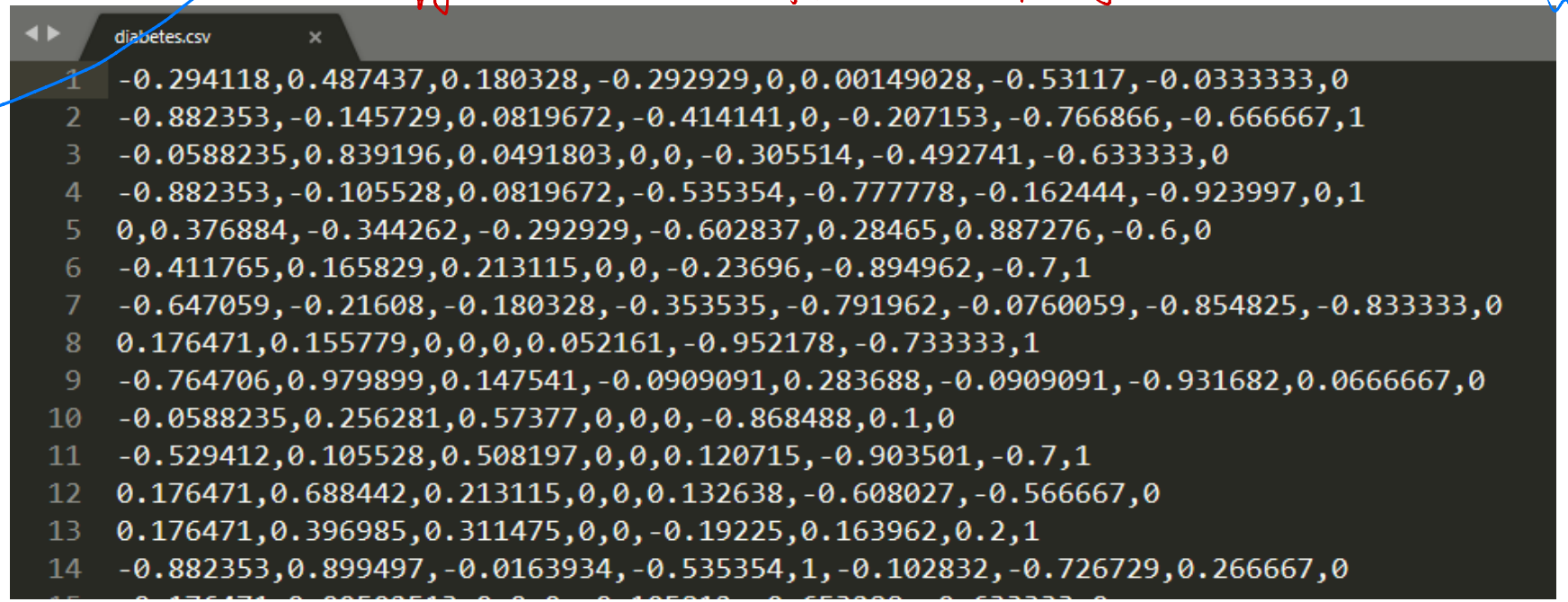
X1	X2	X3	X4	X5	X6	X7	X8	Y
-0.29	0.49	0.18	-0.29	0.00	0.00	-0.53	-0.03	0
-0.88	-0.15	0.08	-0.41	0.00	-0.21	-0.77	-0.67	1
-0.06	0.84	0.05	0.00	0.00	-0.31	-0.49	-0.63	0
-0.88	-0.11	0.08	-0.54	-0.78	-0.16	-0.92	0.00	1
0.00	0.38	-0.34	-0.29	-0.60	0.28	0.89	-0.60	0
-0.41	0.17	0.21	0.00	0.00	-0.24	-0.89	-0.70	1
-0.65	-0.22	-0.18	-0.35	-0.79	-0.08	-0.85	-0.83	0
0.18	0.16	0.00	0.00	0.00	0.05	-0.95	-0.73	1
-0.76	0.98	0.15	-0.09	0.28	-0.09	-0.93	0.07	0
-0.06	0.26	0.57	0.00	0.00	0.00	-0.87	0.10	0

Example: Diabetes Prediction

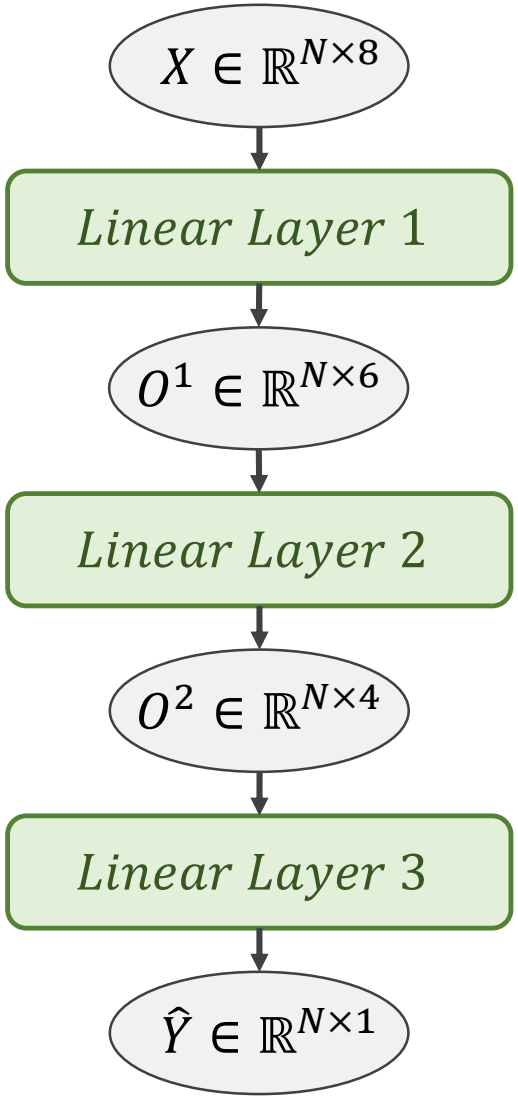
1. Prepare dataset
   we shall talk about this later
2. Design model using Class
   inherit from nn.Module
3. Construct loss and optimizer
   using PyTorch API
4. Training cycle
   forward, backward, update

Example: 1. Prepare Dataset

```
import numpy as np
xy = np.loadtxt('diabetes.csv.gz', delimiter=',', dtype=np.float32)
x_data = torch.from_numpy(xy[:,:-1])
y_data = torch.from_numpy(xy[:, [-1]])
```



Example: 2. Define Model




```python
import torch

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear1 = torch.nn.Linear(8, 6)
        self.linear2 = torch.nn.Linear(6, 4)
        self.linear3 = torch.nn.Linear(4, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.linear1(x))
        x = self.sigmoid(self.linear2(x))
        x = self.sigmoid(self.linear3(x))
        return x

model = Model()
```

Example: 3. Construct Loss and Optimizer

Mini-Batch Loss Function for Binary Classification

$$loss = -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)$$

Update

$$\omega = \omega - \alpha \frac{\partial cost}{\partial \omega}$$

cross-e-- 交叉熵)

```
criterion = torch.nn.BCELoss(size_average=True)

optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

Example: 4. Training Cycle

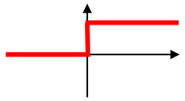
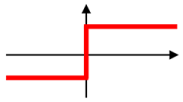
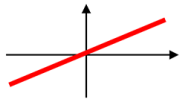
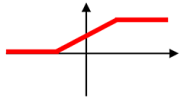
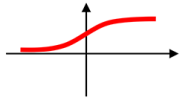
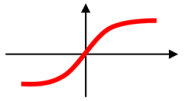
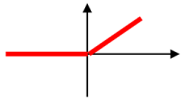
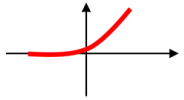
```
for epoch in range(100):
    # Forward
    y_pred = model(x_data)
    loss = criterion(y_pred, y_data)
    print(epoch, loss.item())

    # Backward
    optimizer.zero_grad()
    loss.backward()

    # Update
    optimizer.step()
```

NOTICE:

This program has not use **Mini-Batch** for training.

We shall talk about **DataLoader** later.

Exercise: Try different activate function

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

常用

Linear不是 $\vec{x} \cdot \vec{W} + \vec{b}$，而是1次

↓

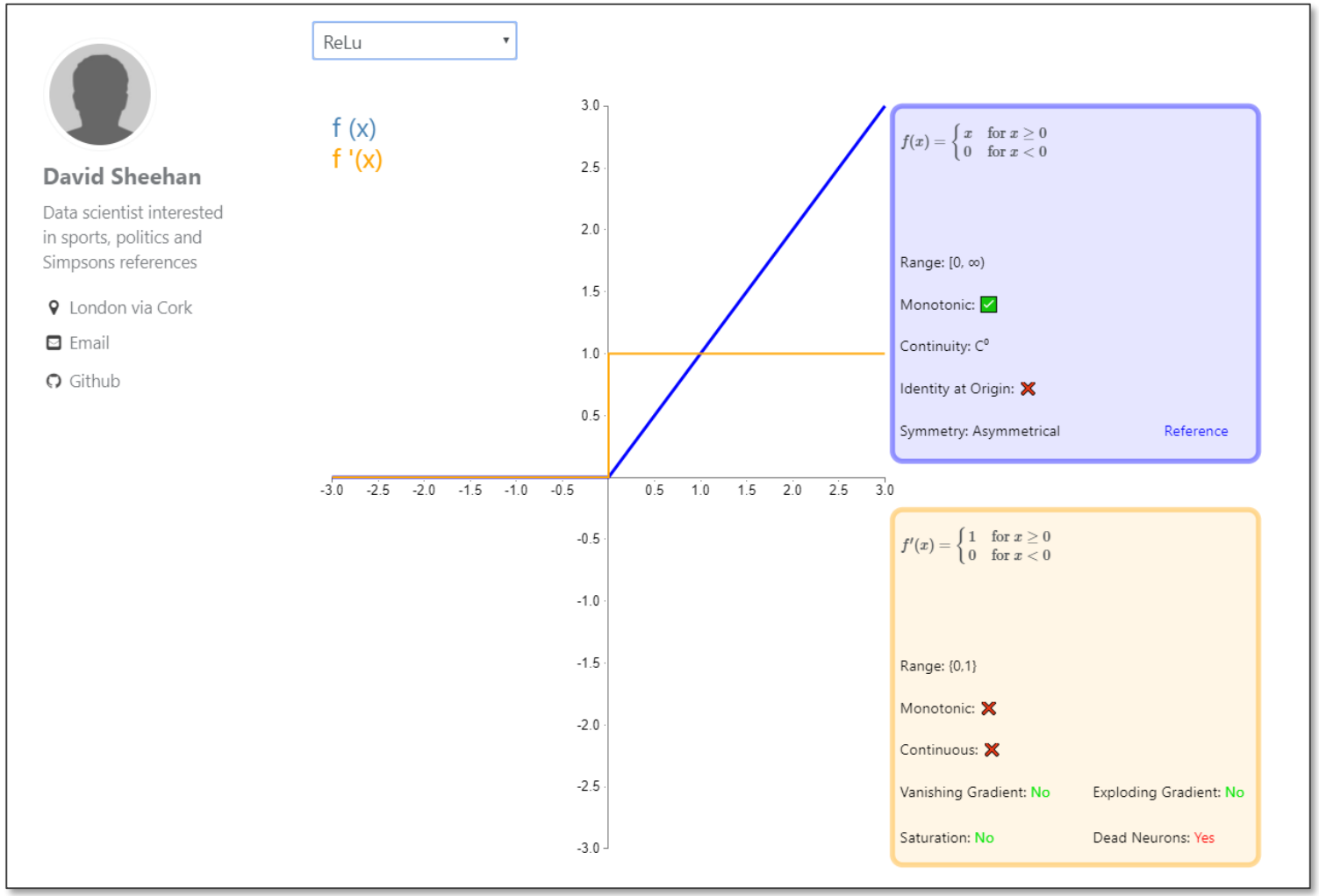
每层中都含有，无论激活函数

可以看作 W 在此层与前层所连接的弧上，

即此层的输入就直接为 $\vec{x} \cdot \vec{W} + \vec{b}$

http://rasbt.github.io/mlxtend/user_guide/general_concepts/activation-functions/#activation-functions-for-artificial-neural-networks

Exercise: Try different activate function



https://dashee87.github.io/data%20science/deep%20learning/visualising-activation-functions-in-neural-networks/

Exercise: Try different activate function

pytorch 中的激发活函数

- Non-linear activations (weighted sum, nonlinearity)
  - ELU
  - Hardshrink
  - Hardtanh
  - LeakyReLU
  - LogSigmoid
  - PReLU
  - ReLU
  - ReLU6
  - RReLU
  - SELU
  - Sigmoid
  - Softplus
  - Softshrink
  - Softsign
  - Tanh
  - Tanhshrink
  - Threshold

Non-linear activations (weighted sum, nonlinearity)

class **torch.nn.ELU**(*alpha=1.0, inplace=False*) [source]

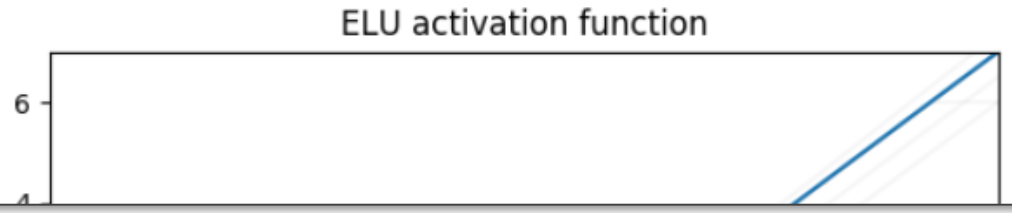
Applies element-wise, $\text{ELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x) - 1))$

Parameters:

- **alpha** – the α value for the ELU formulation. Default: 1.0
- **inplace** – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

Exercise: Try different activate function

```
import torch

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear1 = torch.nn.Linear(8, 6)
        self.linear2 = torch.nn.Linear(6, 4)
        self.linear3 = torch.nn.Linear(4, 1)
        self.activate = torch.nn.ReLU()

    def forward(self, x):
        x = self.activate(self.linear1(x))
        x = self.activate(self.linear2(x))
        x = self.activate(self.linear3(x))
        return x

model = Model()
```

PyTorch Tutorial

07. Multiple Dimension Input