深度学习

单车预测神经网络

张江

北京师范大学系统科学学院

集智学园、集智俱乐部

今日内容

- 神经网络预测器1
  - 神经元细胞
  - 神经网络
  - 神经网络的工作原理
- 神经网络预测器2
  - 数据预处理
  - 利用pytorch构建神经网络
  - 预测结果及其分析
- 对神经网络的解剖

共享单车热遍全国

摩拜的苦恼

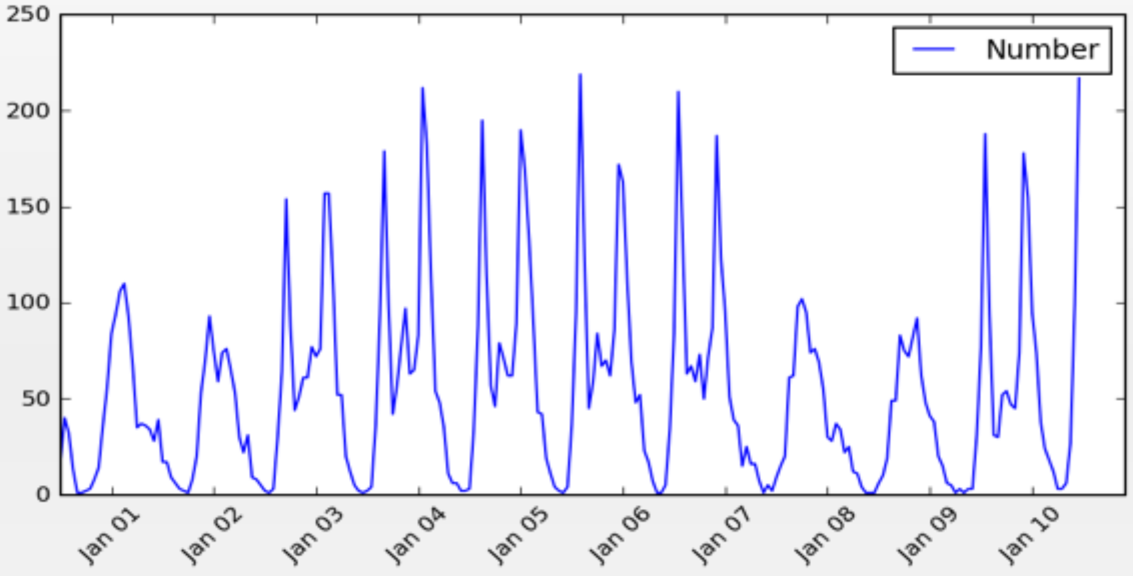
- 问题引出:
  - 究竟什么时间派送工人去搬运
  - 应该从哪里搬运到哪里?
  - 应该搬运多少辆单车?

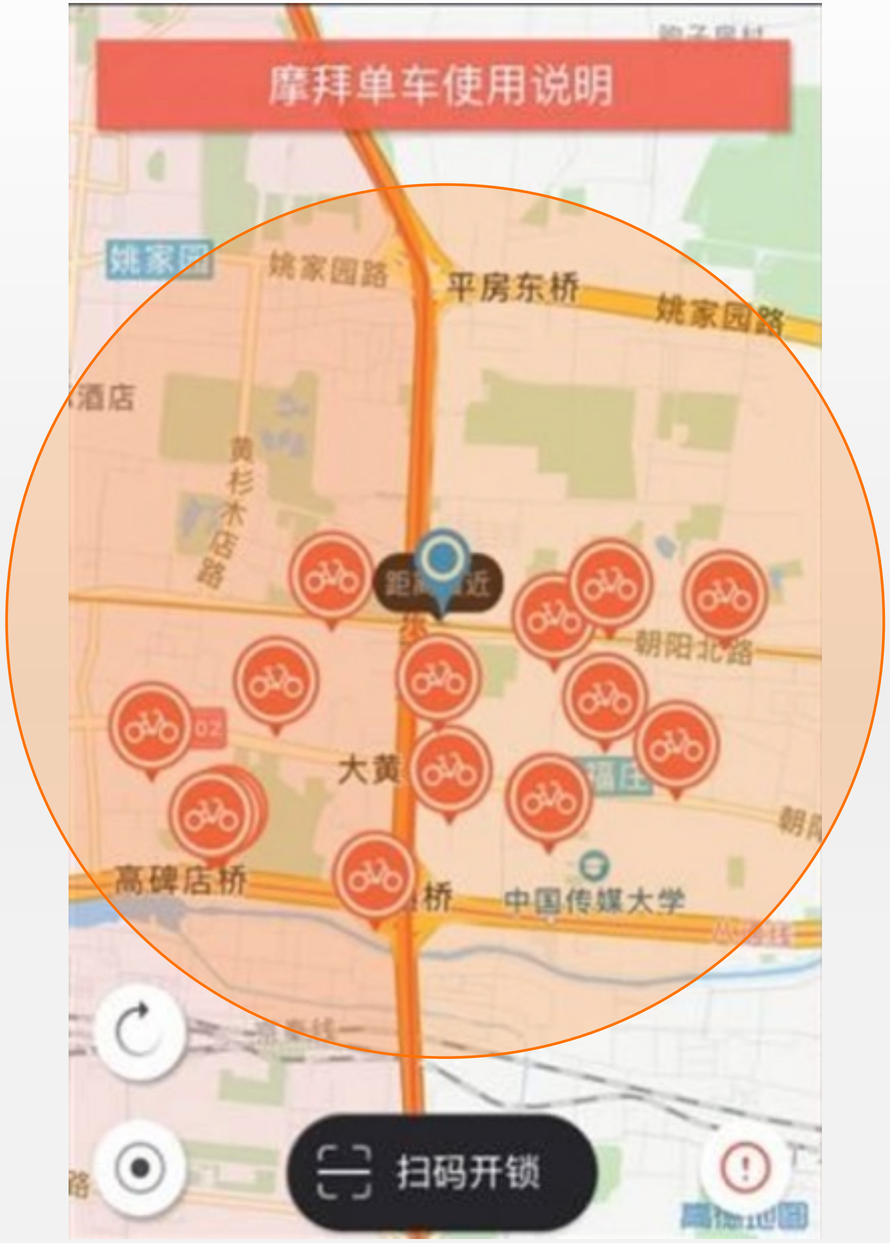
摩拜的苦恼

单车预测器

- 问题引出：
  - 究竟什么时间派送工人去搬运
  - 应该从哪里搬运到哪里？
  - 应该搬运多少辆单车？

单车预测问题



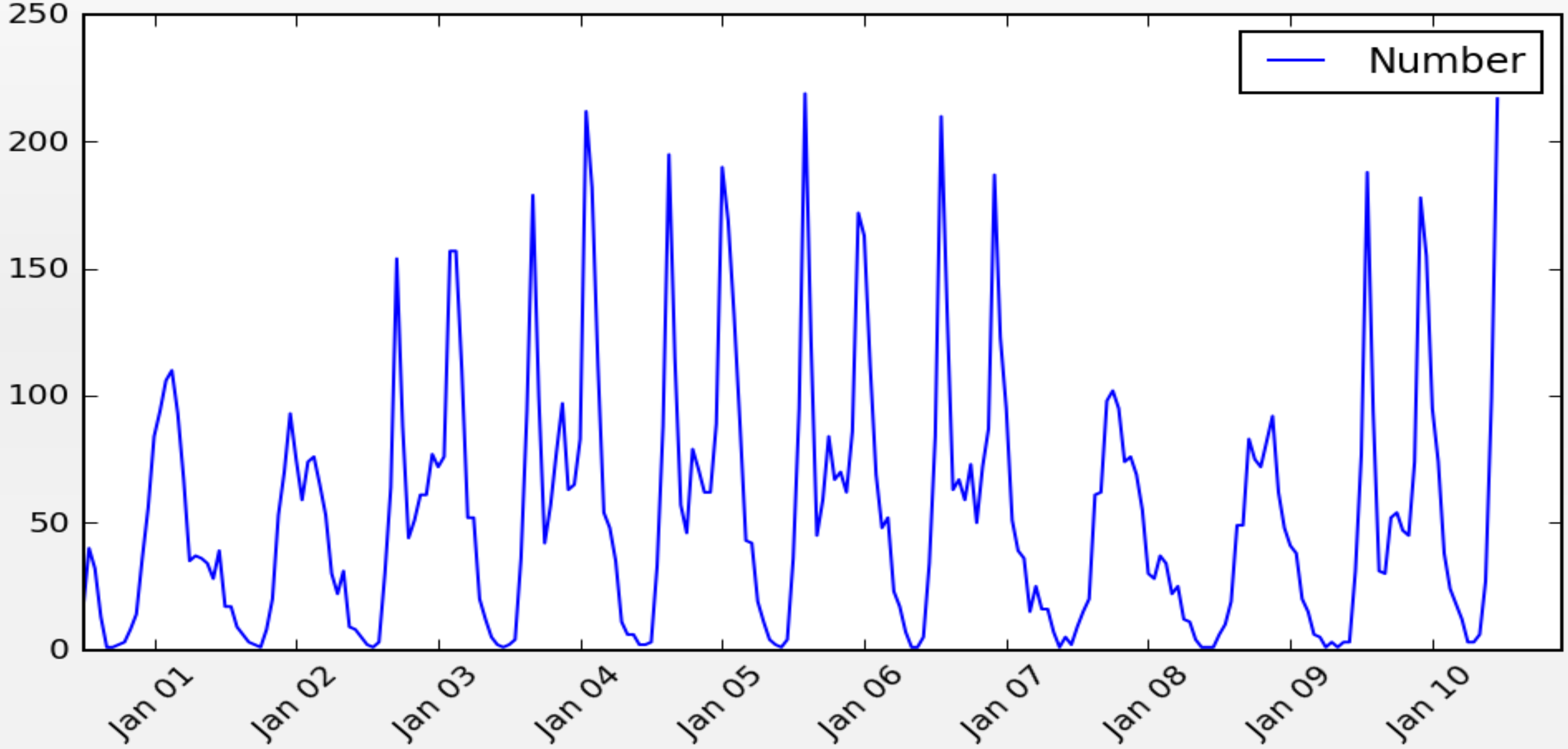


- 预测未来的曲线?

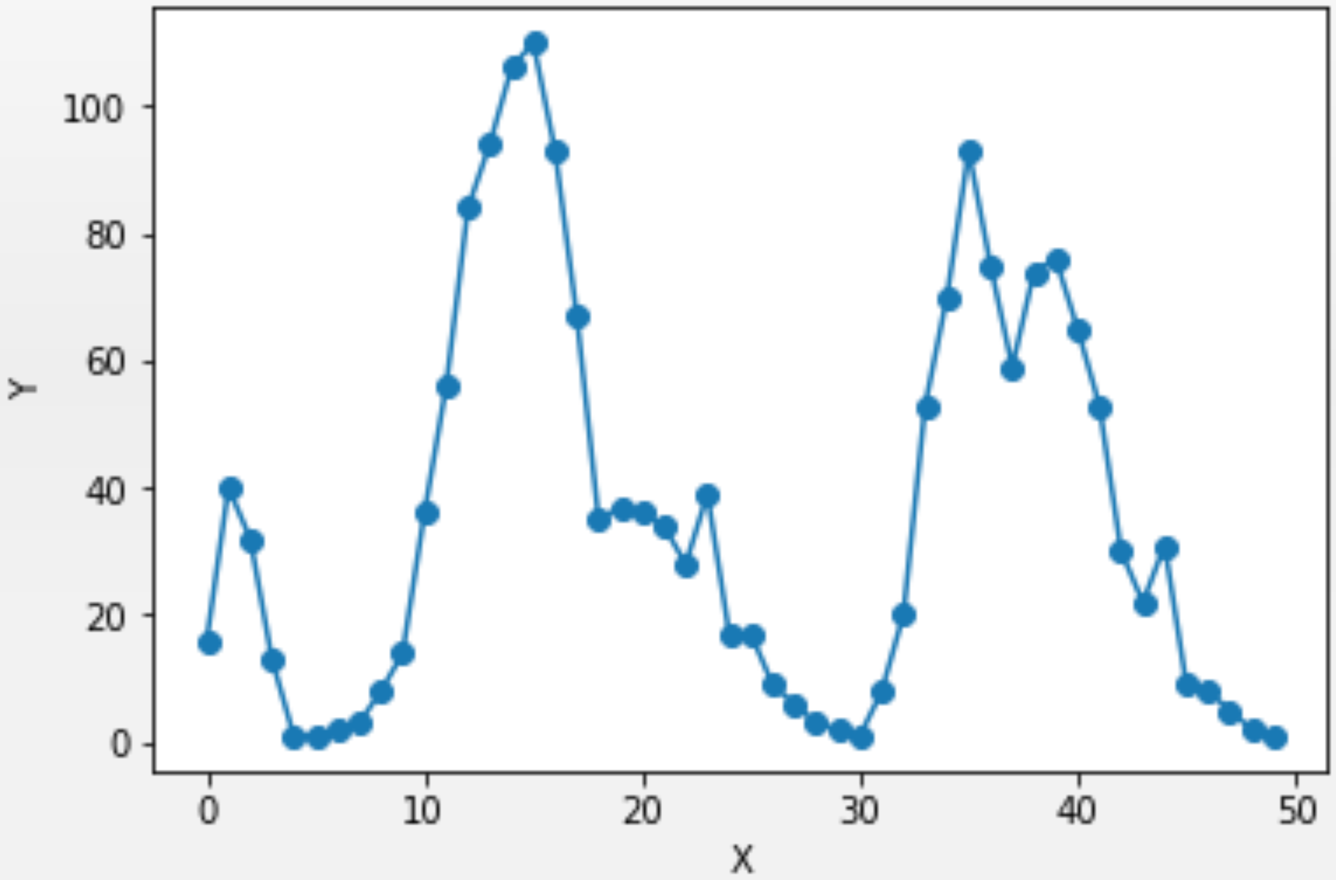
数据来源

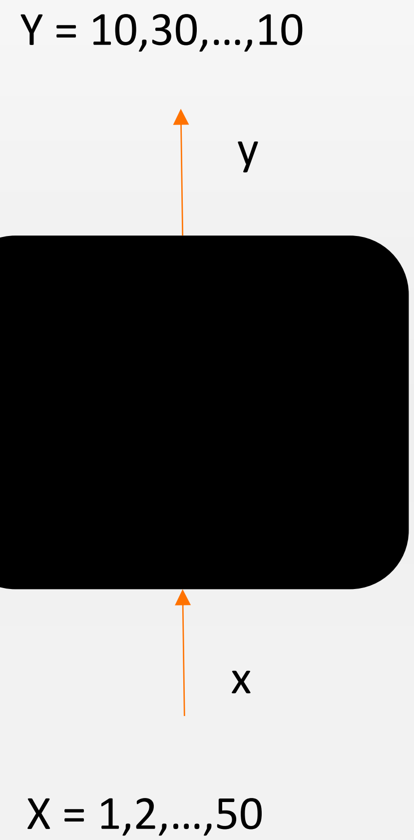
http://capitalbikeshare.com/system-data

http://www.freemeteo.com

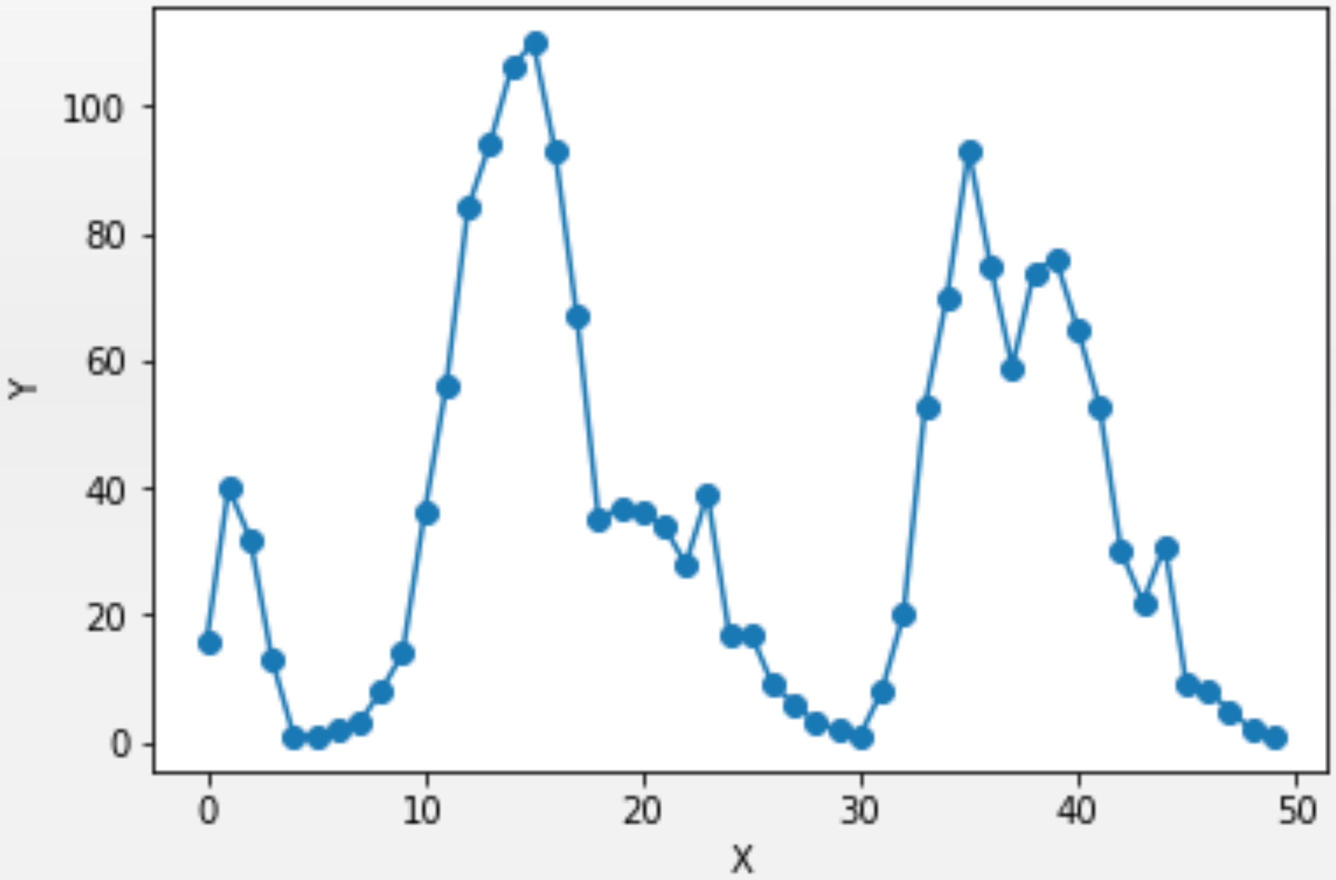


简单神经网络预测器

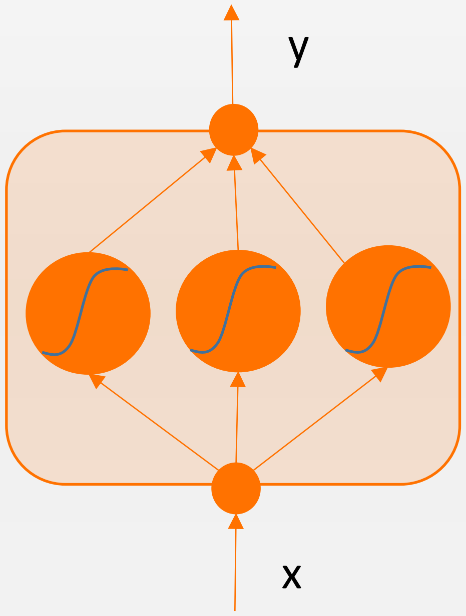




简单神经网络预测器

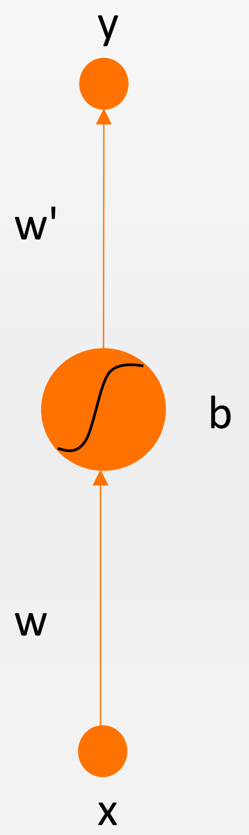


Y = 10,30,...,10



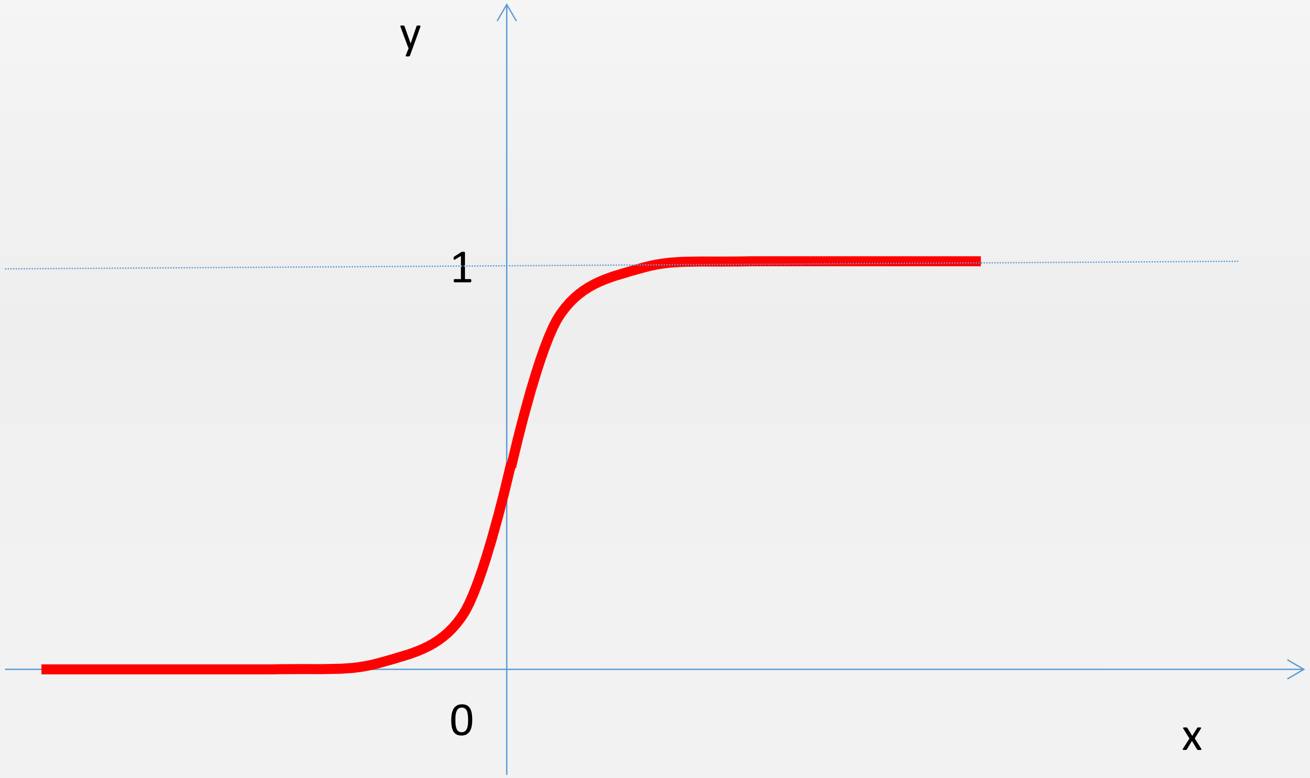
X = 1,2,...,50

单个神经元

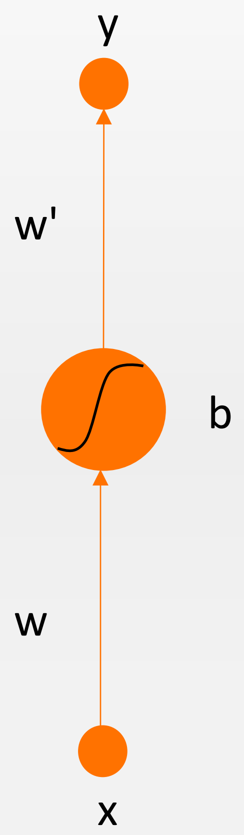


$$y = w'\sigma(wx + b)$$

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

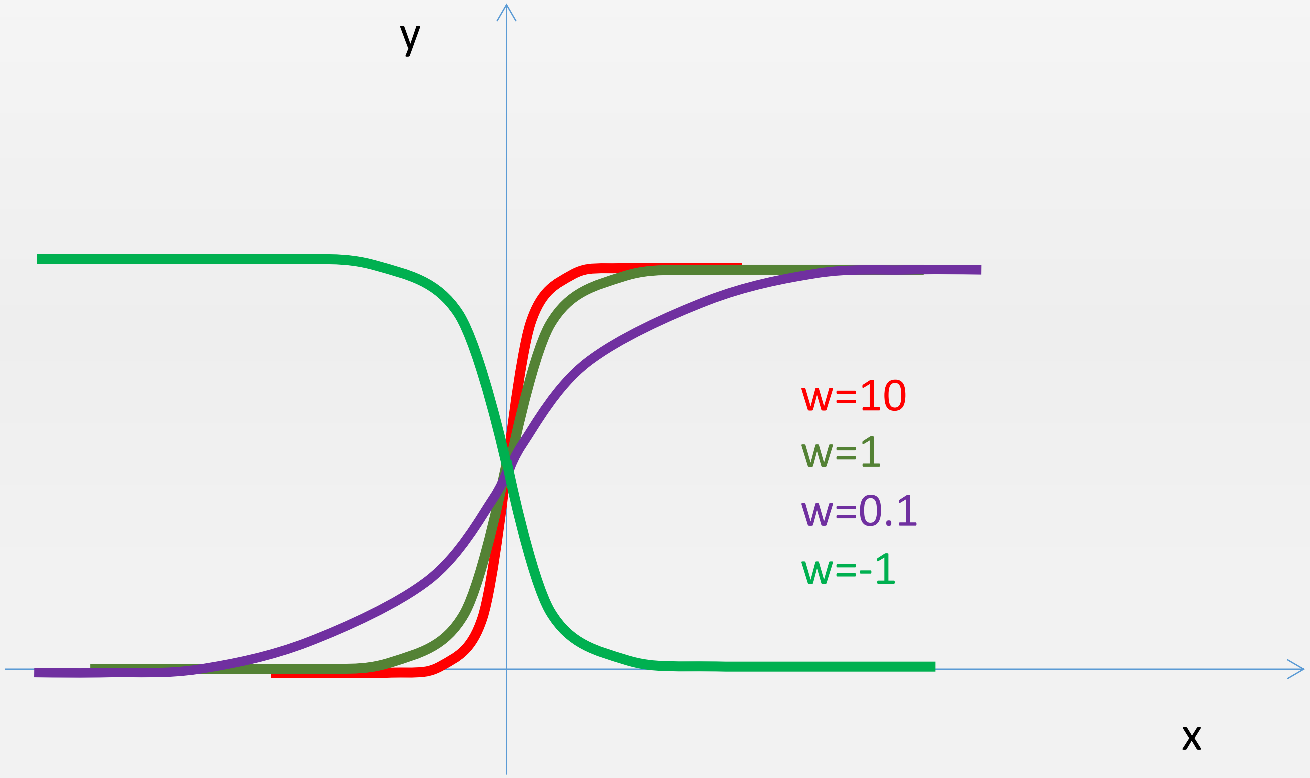


单个神经元



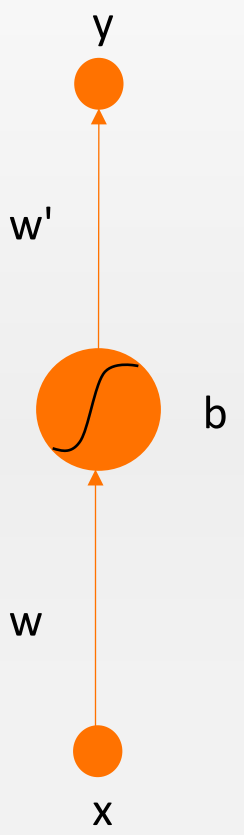
$$y = w'\sigma(wx + b)$$

$$\sigma(wx) = \frac{1}{1+\exp(-wx)}$$



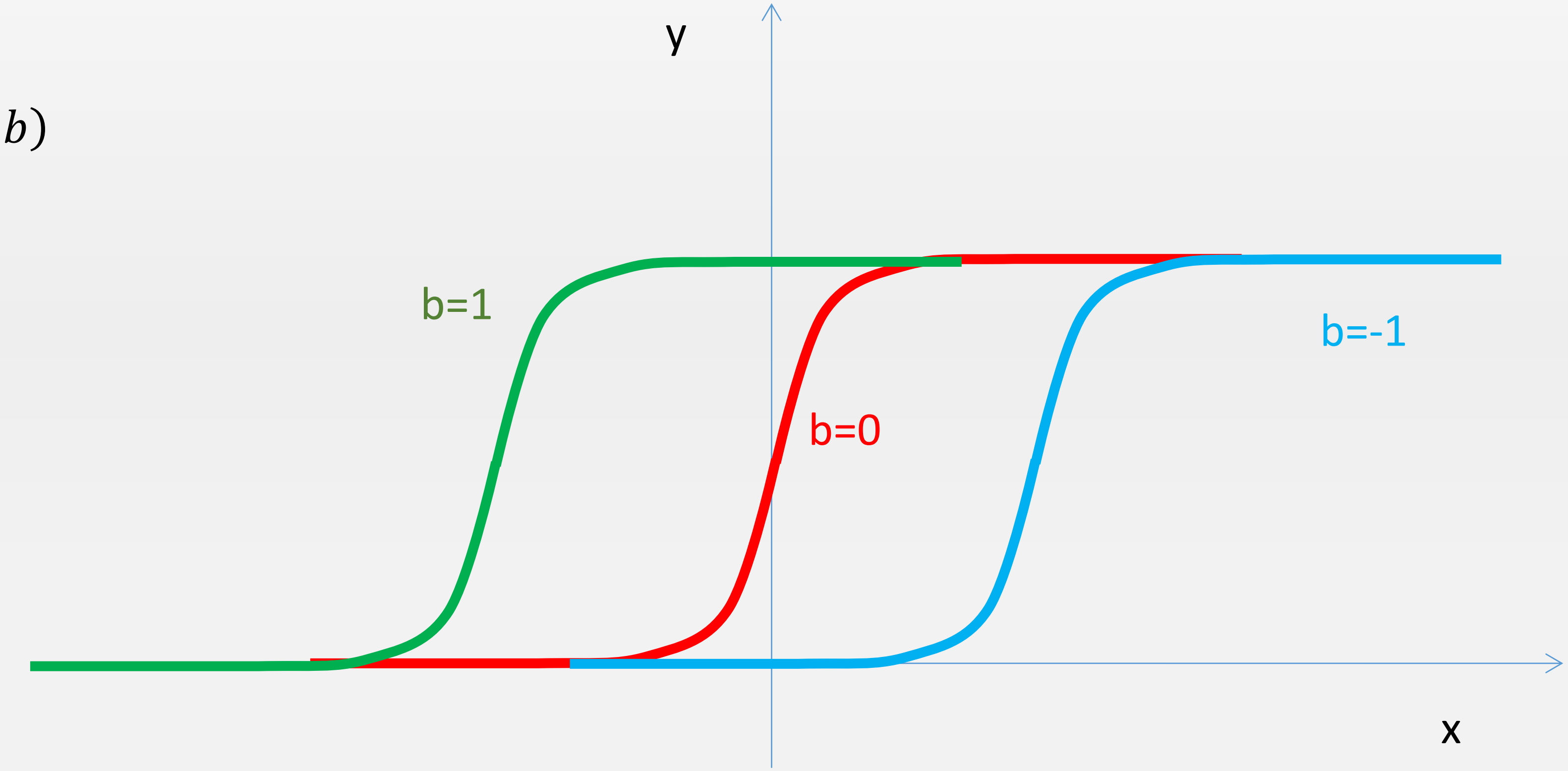
w控制着曲线的弯曲程度

单个神经元



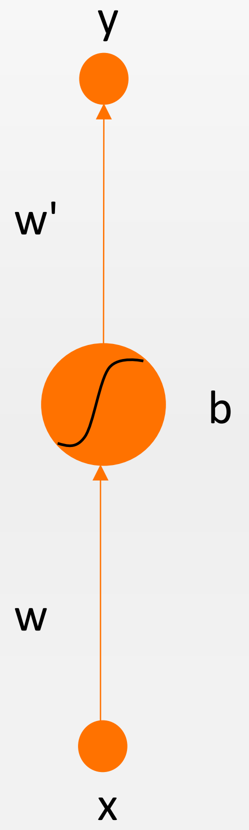
$$y = w'\sigma(wx + b)$$

$$\sigma(wx + b) = \frac{1}{1 + \exp(-(wx + b))}$$



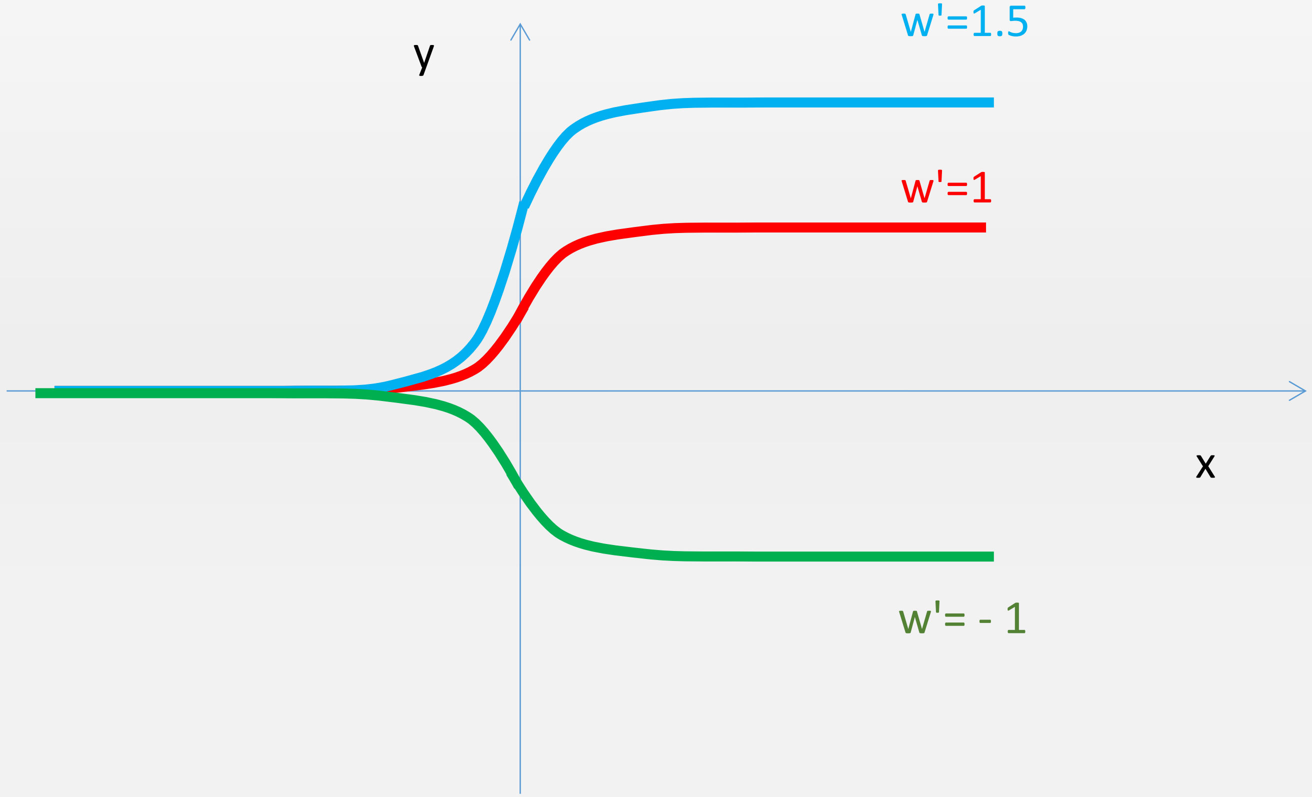
b控制着曲线的竖直位置

单个神经元



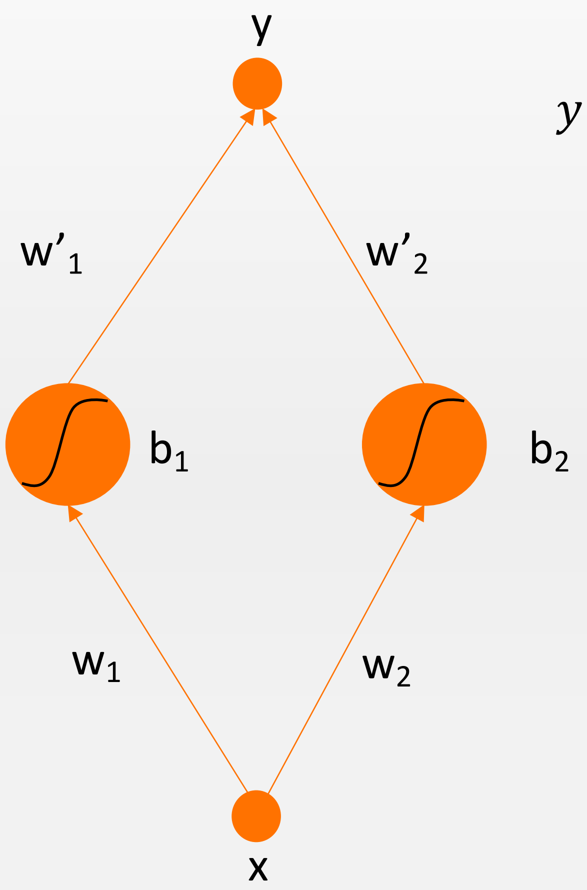
$$y = w'\sigma(wx + b)$$

$$w'\sigma(wx + b) = \frac{w'}{1 + \exp(-w(x + b))}$$



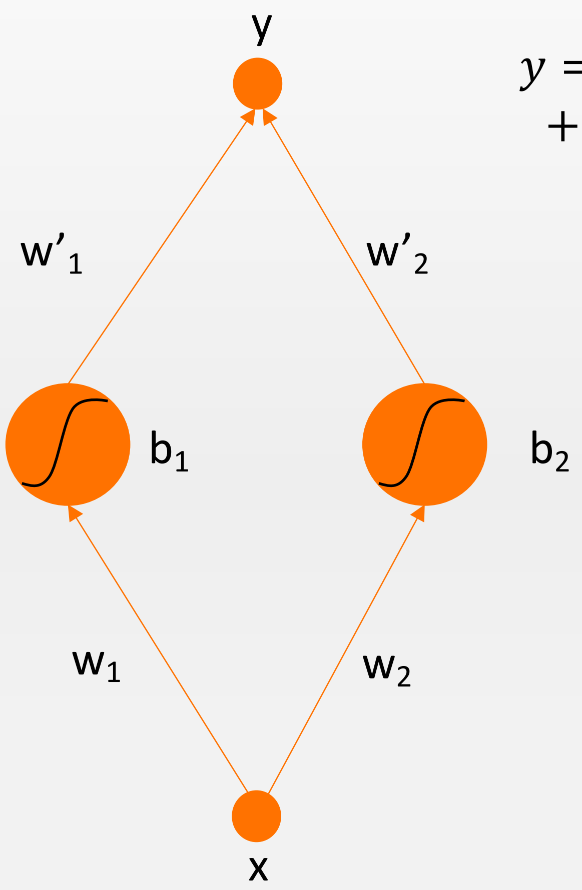
w'控制着曲线的高矮

两个神经元



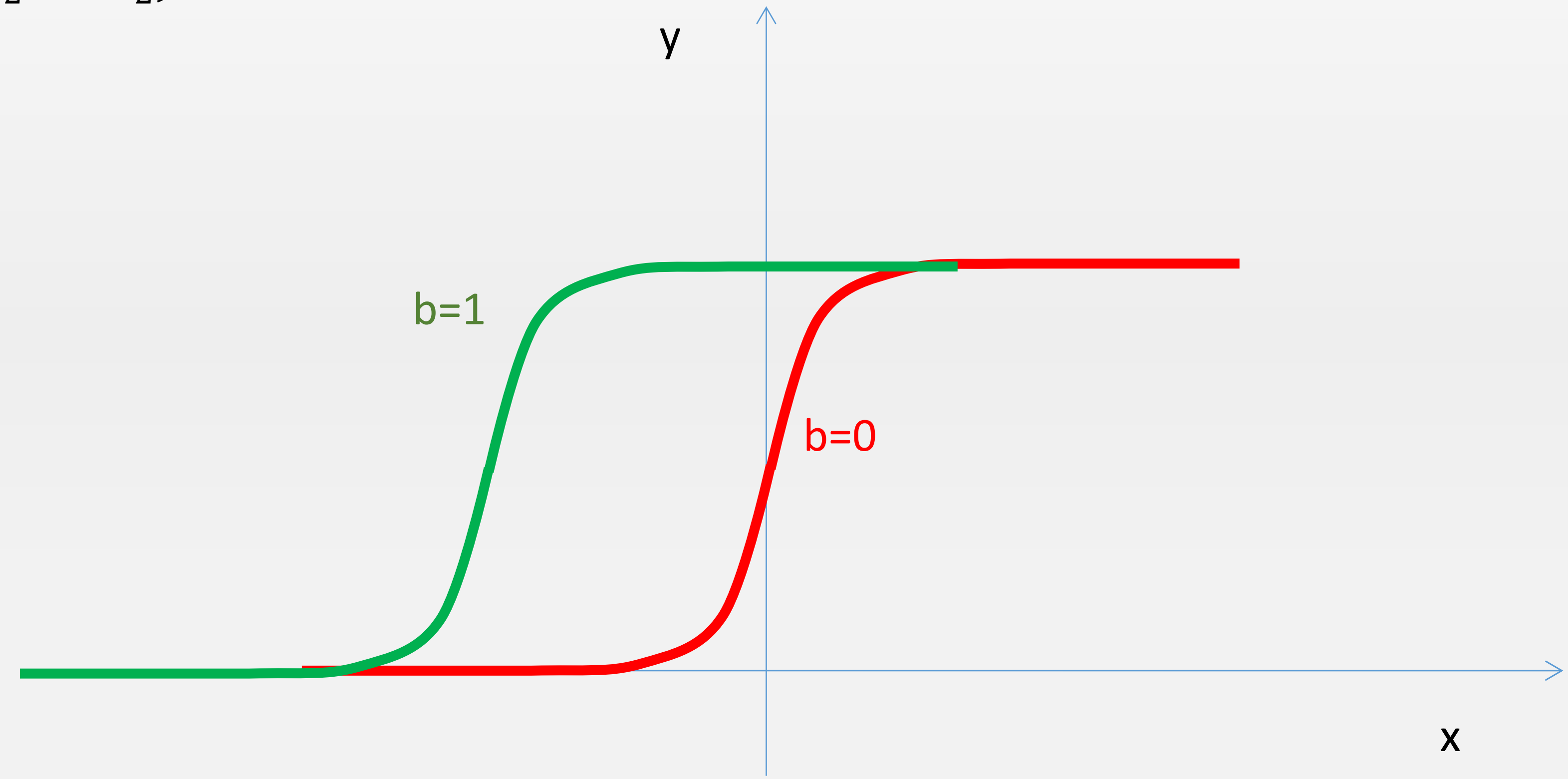
$$y = w_1'\sigma(w_1 x + b_1) + w_2'\sigma(w_2 x + b_2)$$

两个神经元



$$y = w_1'\sigma(w_1 x + b_1) + w_2'\sigma(w_2 x + b_2)$$

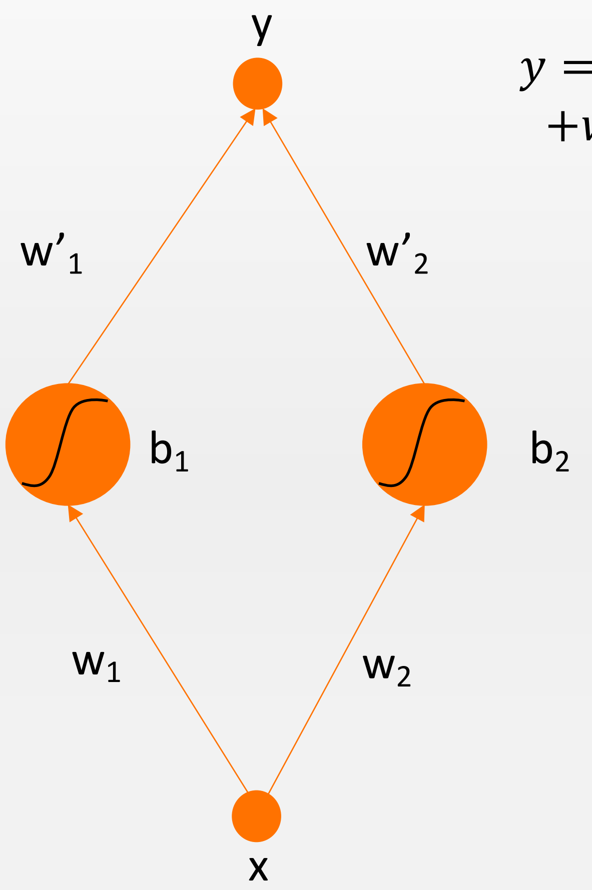
$$f_1(x) = \frac{1}{1 + \exp\big(-(x+1)\big)} \qquad f_2(x) = \frac{1}{1 + \exp(-x)}$$



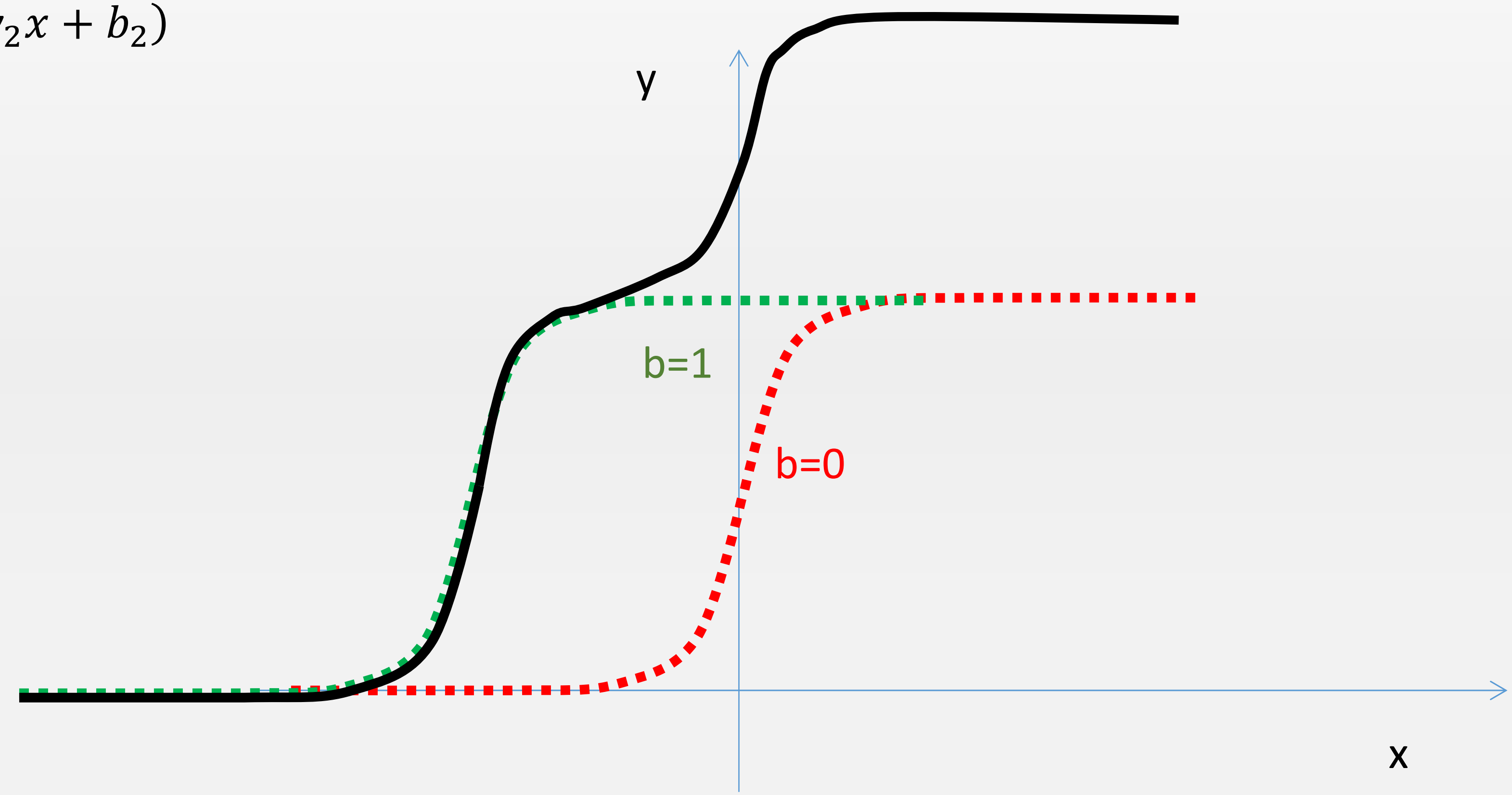
b控制着曲线的竖直位置

两个神经元

$$y=\frac{1}{1+\exp\big(-(x+1)\big)}+\frac{1}{1+\exp(-x)}$$

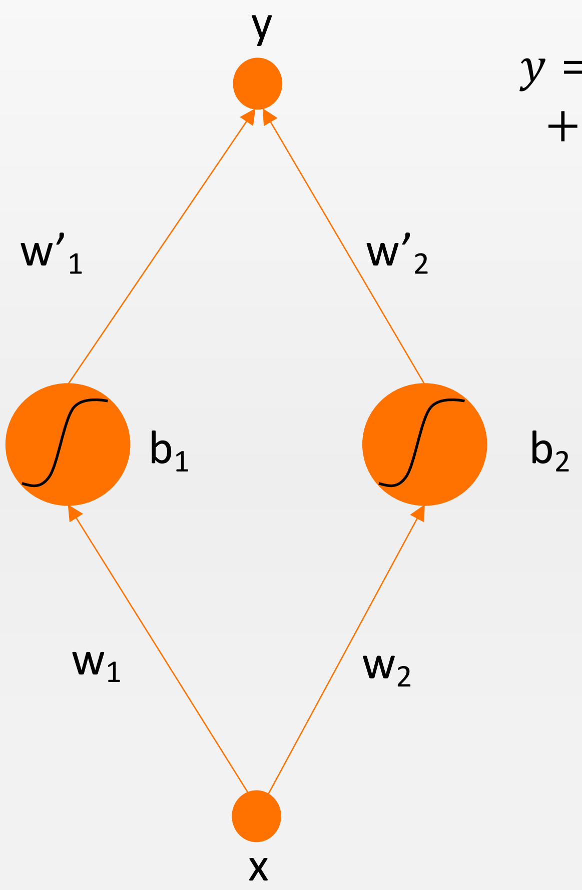


$$y=w_1'\sigma(w_1x+b_1)$$
$$+w_2'\sigma(w_2x+b_2)$$



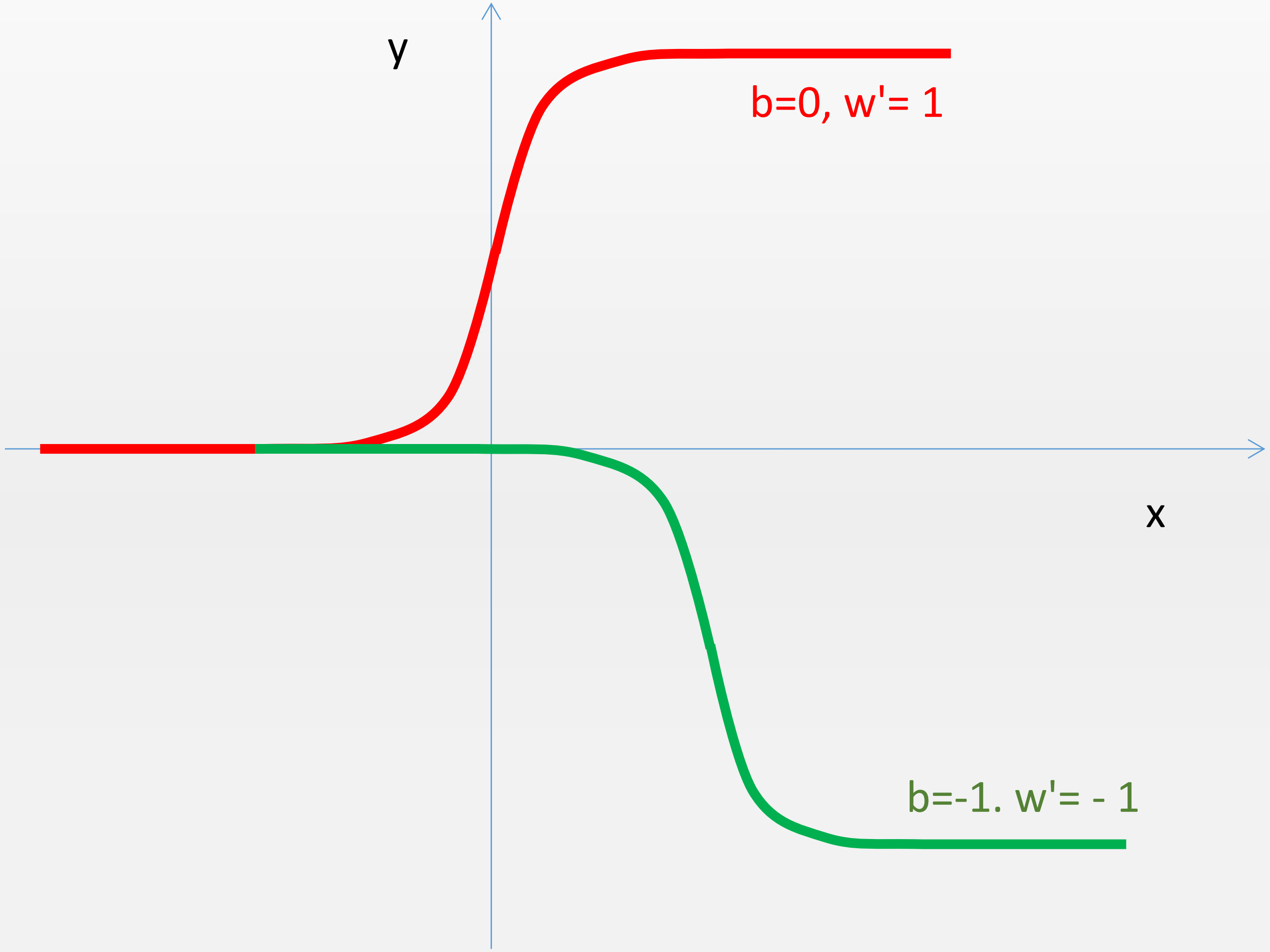
b控制着曲线的竖直位置

两个神经元



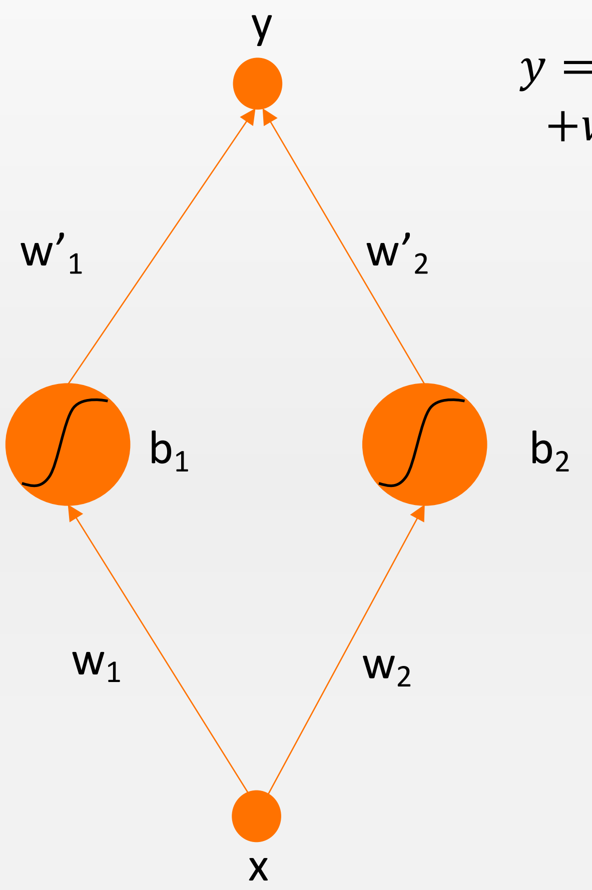
$$y = w_1'\sigma(w_1 x + b_1) + w_2'\sigma(w_2 x + b_2)$$

$$y = \frac{1}{1+\exp(-x)} + \frac{-1}{1+\exp(-x+1)}$$



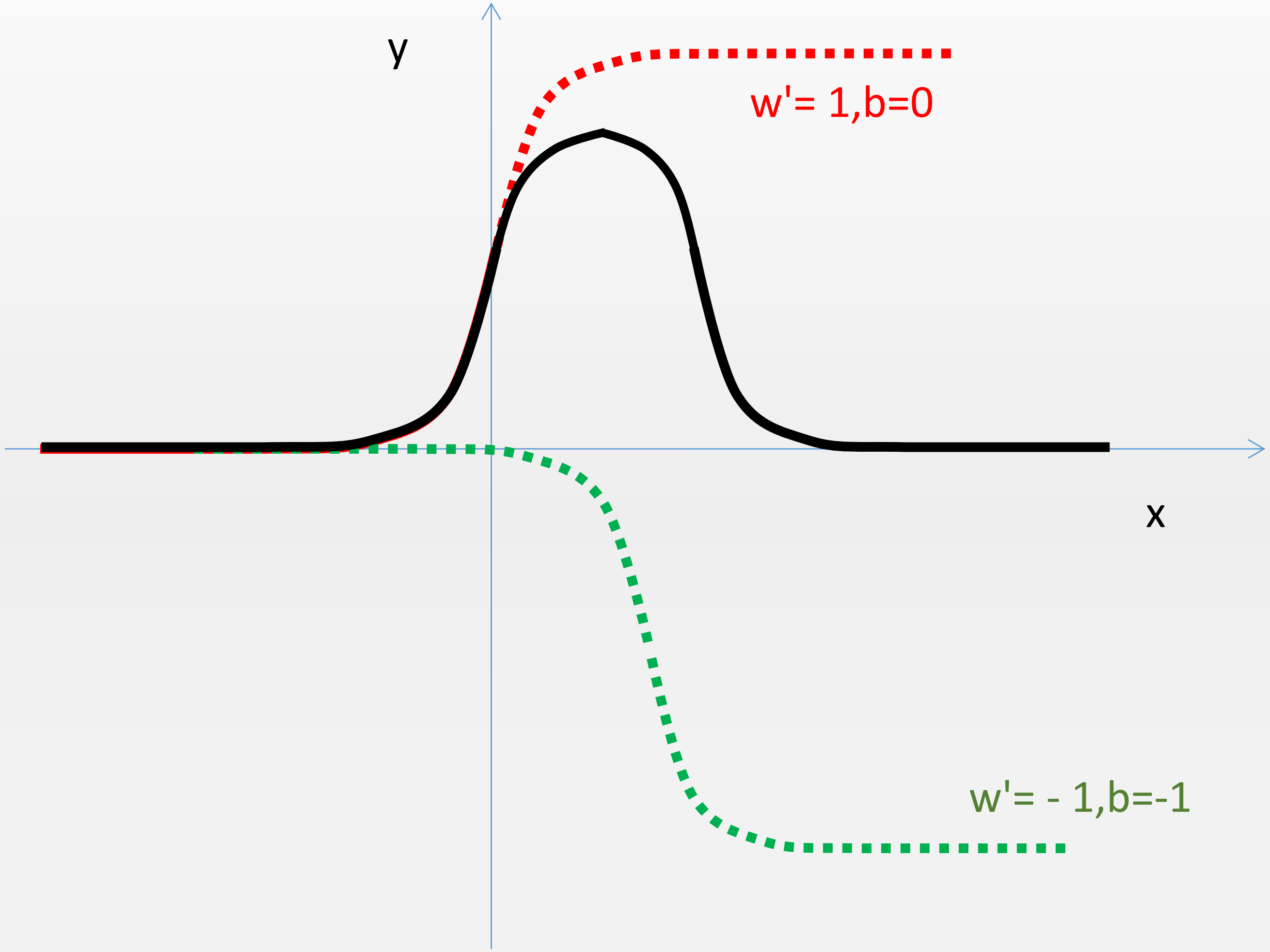
b控制着曲线的竖直位置

两个神经元



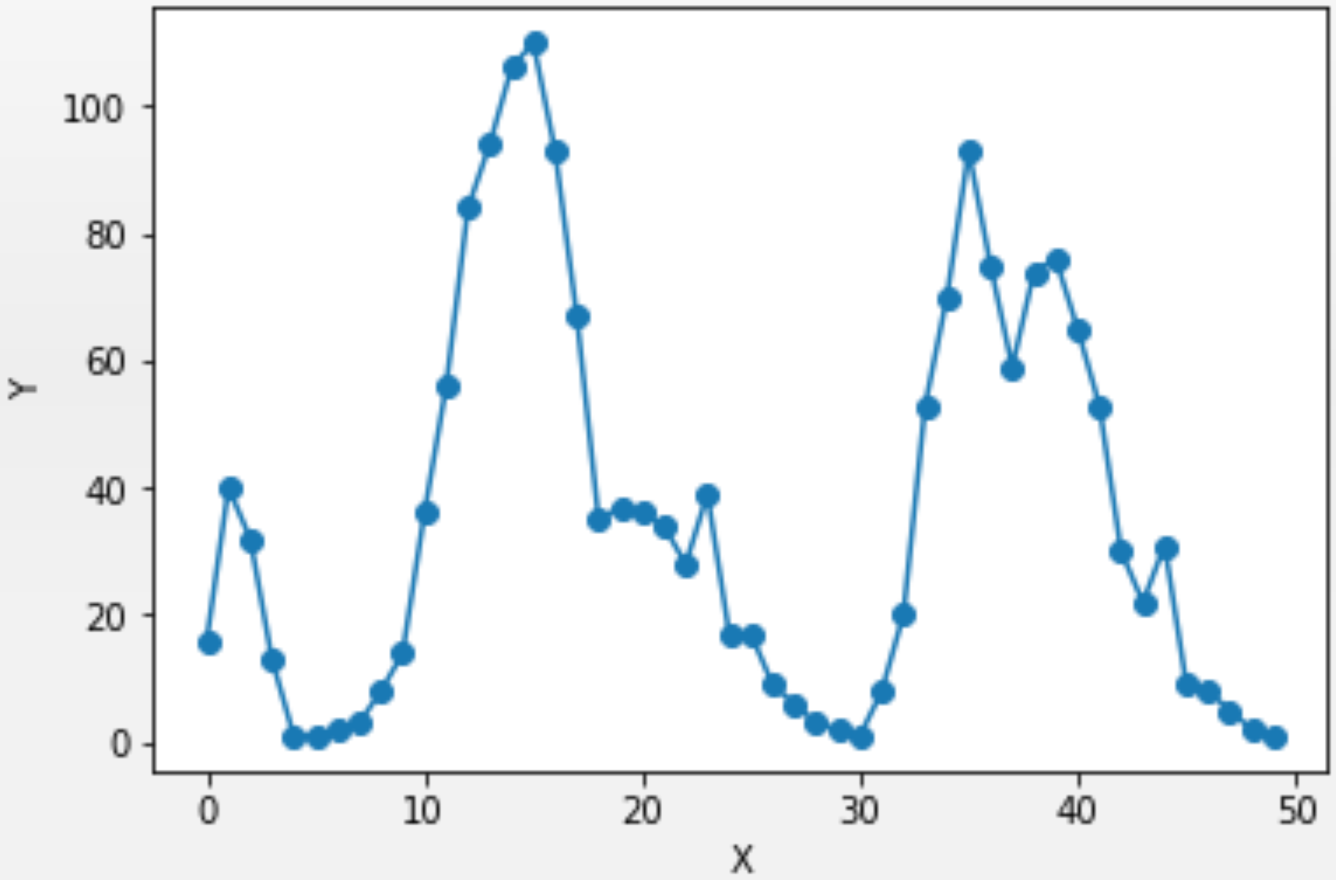
$$y = w_1'\sigma(w_1 x + b_1) + w_2'\sigma(w_2 x + b_2)$$

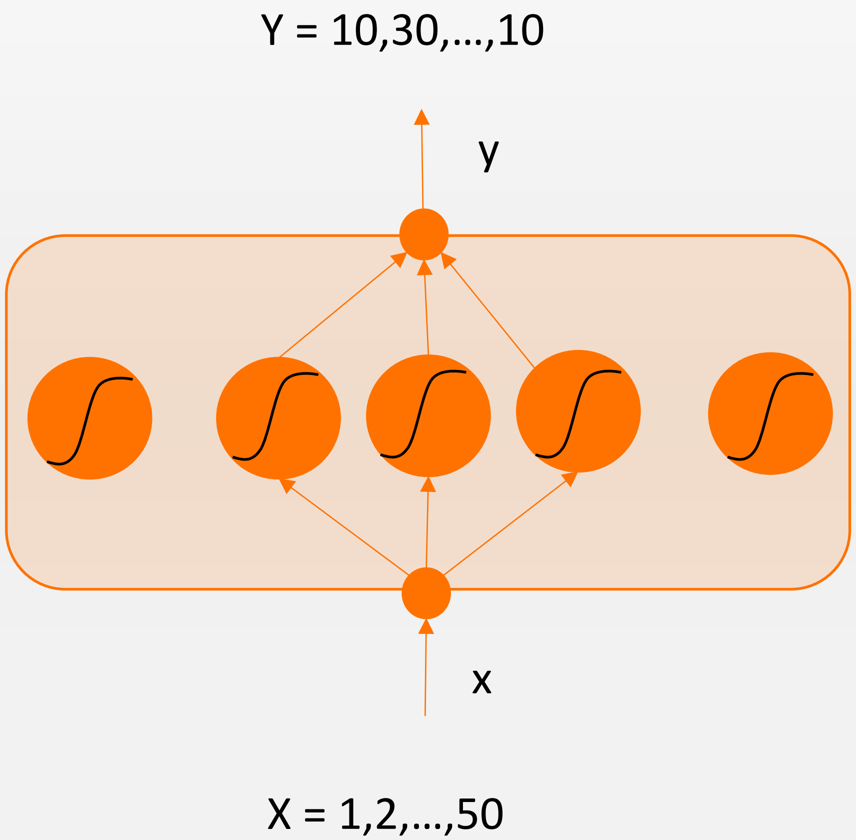
$$y = \frac{1}{1+\exp(-x)} + \frac{-1}{1+\exp(-x+1)}$$



b控制着曲线的竖直位置

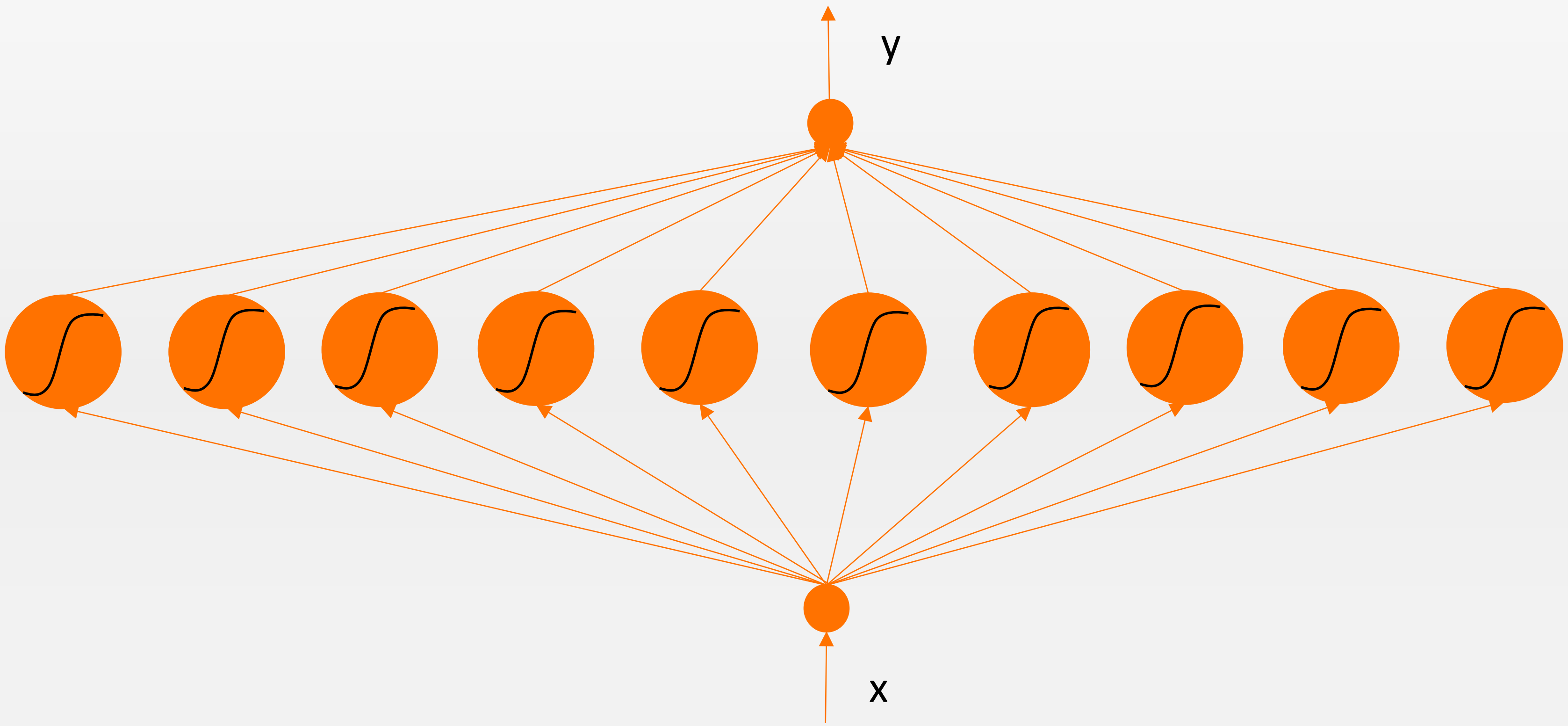
我们至少需要多少隐含层神经元?





第一个单隐含层神经网络

Y = 12,30,...,10



X = 1,2,...,50

导入需要的库

In [1]:

```
#导入需要使用的库
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch

# 让输出的图形直接在Notebook中显示
%matplotlib inline
```

导入、处理csv文件的库

有关Tensor操作的torch库
用于autograd自动求导变量的库

导入数据

In [2]:

```
#读取数据到内存中，rides为一个dataframe对象
data_path = 'Bike-Sharing-Dataset/hour.csv'
rides = pd.read_csv(data_path)
#看看数据长什么样子
rides.head()
```

导入、处理csv文件的库

显示前几条记录

Out[2]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt
0	1	2011-01-01	1	0	1	0	0	6	0	1	0.24	0.2879	0.81	0.0	3	13	16
1	2	2011-01-01	1	0	1	1	0	6	0	1	0.22	0.2727	0.80	0.0	8	32	40
2	3	2011-01-01	1	0	1	2	0	6	0	1	0.22	0.2727	0.80	0.0	5	27	32
3	4	2011-01-01	1	0	1	3	0	6	0	1	0.24	0.2879	0.75	0.0	3	10	13
4	5	2011-01-01	1	0	1	4	0	6	0	1	0.24	0.2879	0.75	0.0	0	1	1

初始化神经网络权重等变量

In [3]:

```
counts = rides['cnt'][:50]
x = torch.tensor(np.arange(len(counts)), dtype=torch.double, requires_grad = True)
y = torch.tensor(np.array(counts), dtype=torch.double, requires_grad = True)
sz = 10
# 初始化所有神经网络的权重（weights）和阈值（biases）
weights = torch.randn((1, sz), dtype = torch.double, requires_grad = True)
biases = torch.randn(sz, dtype = torch.double, requires_grad = True)
weights2 = torch.randn((sz, 1), dtype = torch.double, requires_grad = True)
```

提取数据库的cnt字段前50条记录

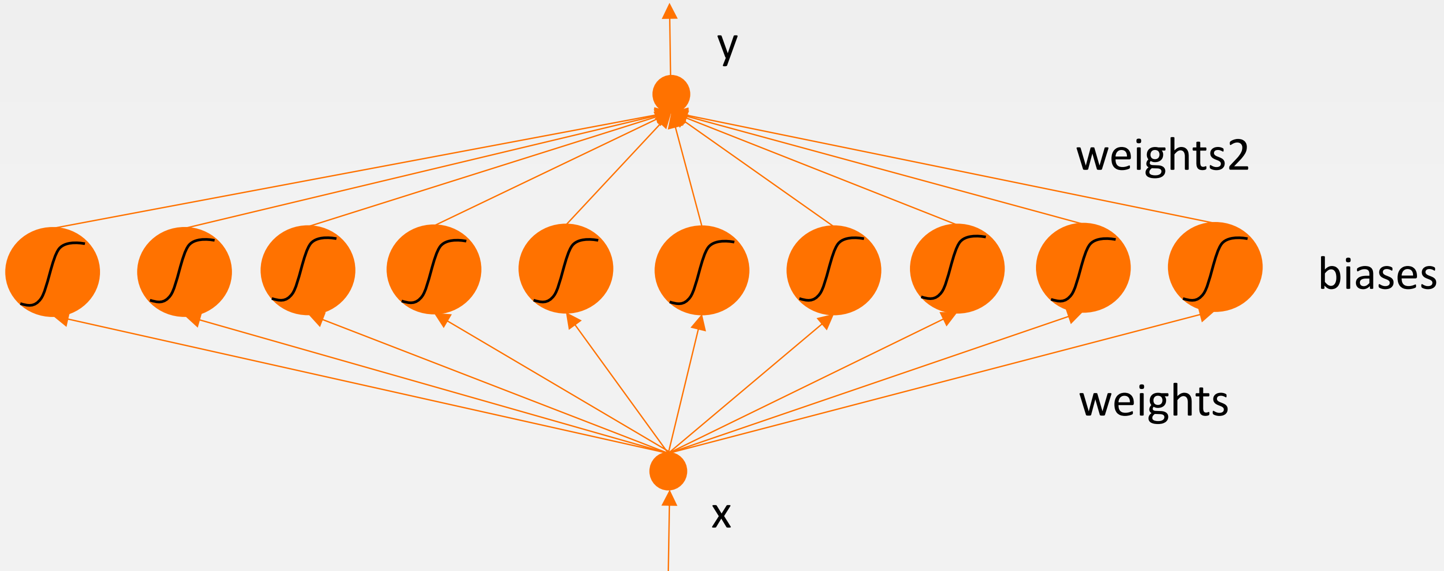
50行1列

50行1列

1行10列

10个元素的列向量

10行1列



神经网络梯度下降迭代

In [4]:

```
learning_rate = 0.0001 #设置学习率
losses = []
# 将 x 转换为(50,1)的维度，以便与维度为(1,10)的weights矩阵相乘
x = x.unsqueeze(1)
# 将 y 转换为(50,1)的维度
y = y.unsqueeze(1)
for i in range(1000000):
    # 从输入层到隐含层的计算
    hidden = x * weights+ biases
    # 将sigmoid函数作用在隐含层的每一个神经元上
    hidden = torch.sigmoid(hidden)
    # 隐含层输出到输出层，计算得到最终预测
    predictions = hidden.mm(weights2)
    # 通过与标签数据y比较，计算误差
    loss = torch.mean((predictions - y) ** 2)
    if i % 10000 == 0:
        print('loss:', loss)
    loss.backward() #对损失函数进行梯度反传
    #利用上一步计算中得到的weights，biases等梯度信息更新weights或biases中的data数值
    weights.data.add_(- learning_rate * weights.grad.data)
    biases.data.add_(- learning_rate * biases.grad.data)
    weights2.data.add_(- learning_rate * weights2.grad.data)
    # 梯度清空
    weights.grad.data.zero_()
    biases.grad.data.zero_()
    weights2.grad.data.zero_()
```

unsqueeze

In [4]:

```
learning_rate = 0.0001 #设置学习率
losses = []
# 将 x 转换为(50,1)的维度，以便与维度为(1,10)的weights矩阵相乘
x = x.unsqueeze(1)
# 将 y 转换为(50,1)的维度
y = y.unsqueeze(1)
for i in range(1000000):
   # 从输入层到隐含层的计算
   hidden = x * weights+ biases
   # 将sigmoid函数作用在隐含层的每一个神经元上
   hidden = torch.sigmoid(hidden)
   # 隐含层输出到输出层，计算得到最终预测
   predictions = hidden.mm(weights2)
   # 通过与标签数据y比较，计算误差
   loss = torch.mean((predictions - y) ** 2)
   if i % 10000 == 0:
      print('loss:', loss)
   loss.backward() #对损失函数进行梯度反传
   #利用上一步计算中得到的weights，biases等梯度信息更新weights或biases中的data数值
   weights.data.add_(- learning_rate * weights.grad.data)
   biases.data.add_(- learning_rate * biases.grad.data)
   weights2.data.add_(- learning_rate * weights2.grad.data)
   # 梯度清空
   weights.grad.data.zero_()
   biases.grad.data.zero_()
   weights2.grad.data.zero_()
```

x.size(): (50)

x.unsqueeze(1).size(): (50,1)
x.unsqueeze(0).size(): (1,50)

向量矩阵乘法

In [4]:

```
learning_rate = 0.0001 #设置学习率
losses = []
for i in range(1000000):
    # 从输入层到隐含层的计算
    hidden = x * weights + biases
    # 将sigmoid函数作用在隐含层的每一个神经元上
    hidden = torch.sigmoid(hidden)
    # 隐含层输出到输出层，计算得到最终预测
    predictions = hidden.mm(weights2) + biases2.expand_as(y)
    # 通过 据y比较
    loss = t n((predictions - y) ** 2)
    losses. oss.data.numpy())
    if i % 1 ):
        print('loss:', loss)
    loss.backward() #对损失函数进行梯度反传
    #利用上一步计算中得到的weights，biases等梯度信息更新weights或biases中的data数值
    weights.data.add_(- learning_rate * weights.grad.data)
    biases.data.add_(- learning_rate * biases.grad.data)
    weights2.data.add_(- learning_rate * weights2.grad.data)
    biases2.data.add_(- learning_rate * biases2.grad.data)
```

$$x \quad \begin{pmatrix} 0.1 \\ 0.2 \\ \vdots \\ 1.0 \end{pmatrix} * (2.0, 1.1, 0.2, \dots, 0.1) \Rightarrow \begin{pmatrix} 0.2 & 0.11 & \dots & 0.01 \\ 0.4 & 0.22 & \cdots & 0.02 \\ \vdots & \vdots & \cdots & \vdots \\ 2.0 & 1.1 & \cdots & 0.1 \end{pmatrix}$$

向量矩阵乘法

In [4]:

```
learning_rate = 0.0001 #设置学习率
losses = []
for i in range(1000000):
    # 从输入层到隐含层的计算
    hidden = x * weights + biases
    # 将sigmoid函数作用在隐含层的每一个神经元上
    hidden = torch.sigmoid(hidden)
    # 隐含层输出到输出层，计算得到最终预测
    predictions = hidden.mm(weights2) + biases2.expand_as(y)
    # 通过与标签数
    los      ea      2)
    losses.append(lo
    if i % 10000 == 0:
        print('loss:', loss)
    loss.backward() #对损失函数进行梯度反传
    #利用上一步计算中得到的weights，biases等梯度信息更新weights或biases中的data数值
    weights.data.add_(- learning_rate * weights.grad.data)
    biases.data.add_(- learning_rate * biases.grad.data)
    weights2.data.add_(- learning_rate * weights2.grad.data)
    biases2.data.add_(- learning_rate * biases2.grad.data)
```

x*weights:

biases:

$$\begin{pmatrix} 0.2 & 0.11 \ldots & 0.01 \\ 0.4 & 0.22 \cdots & 0.02 \\ \vdots & \vdots \quad \cdots & \vdots \\ 2.0 & 1.1 \quad \cdots & 0.1 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.2 \\ \vdots \\ 1.0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0.3 & 0.21 \ldots & 0.11 \\ 0.6 & 0.42 \cdots & 0.22 \\ \vdots & \vdots \quad \cdots & \vdots \\ 3.0 & 2.1 \quad \cdots & 1.1 \end{pmatrix}$$

张量维度

In [4]:

```
learning_rate = 0.0001 #设置学习率
losses = []
for i in range(1000000):
    # 从输入层到隐含层的计算
    hidden = x * weights+ biases
    # 将sigmoid函数作用在隐含层的每一个神经元上
    hidden = torch.sigmoid(hidden)
    # 隐含层输出到输出层，计算得到最终预测
    predictions = hidden.mm(weights2)
    # 通过与标签数据y比较，计算误差
    loss = torch.mean((predictions - y) ** 2)
    if i % 10000 == 0:
        print('loss:', loss)
    loss.backward() #对损失函数进行梯度反传
    #利用上一步计算中得到的weights，biases等梯度信息更新weights或biases中的data数值
    weights.data.add_(- learning_rate * weights.grad.data)
    biases.data.add_(- learning_rate * biases.grad.data)
    weights2.data.add_(- learning_rate * weights2.grad.data)
    # 梯度清空
    weights.grad.data.zero_()
    biases.grad.data.zero_()
    weights2.grad.data.zero_()
```

hidden.size()=(50, 10)

神经网络梯度下降迭代

In [4]:

```
learning_rate = 0.0001 #设置学习率
losses = []
for i in range(1000000):
    # 从输入层到隐含层的计算
    hidden = torch.t(x.expand(sz, len(x))) * weights.expand(len(x), sz) + biases.expand(len(x), sz)
    # 将sigmoid函数作用在隐含层的每一个神经元上
    hidden = torch.sigmoid(hidden)
    # 隐含层输出到输出层，计算得到最终预测
    predictions = hidden.mm(weights2)
    # 通过与标签数据y比较，计算误差
    loss = torch.mean((predictions - y) ** 2)
    losses.append(loss.data.numpy())
    if i % 100
        print('
    loss.back
    #利用上 一步计算中得到的weights，biases等梯度信息 hts或 data数值
    weights.data.add_(- learning_rate * weights.grad.data)
    biases.data.add_(- learning_rate * biases.grad.data)
    weights2.data.add_(- learning_rate * weights2.grad.data)
    biases2.data.add_(- learning_rate * biases2.grad.data)
```

hidden: $\begin{pmatrix} 0.1 & 0.1 & 0.1 \cdots & 0.1 & 0.1 & 0.1 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \\ \vdots & \vdots & \vdots \cdots & \vdots & \vdots & \vdots \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{pmatrix}$ mm $\begin{pmatrix} 7.1 \\ -0.2 \\ 2 \\ 1.1 \\ \vdots \\ 1.0 \end{pmatrix}$ = $\begin{pmatrix} 0.71 \\ -0.02 \\ 0.2 \\ 0.11 \\ \vdots \\ 0.10 \end{pmatrix}$

张量维度

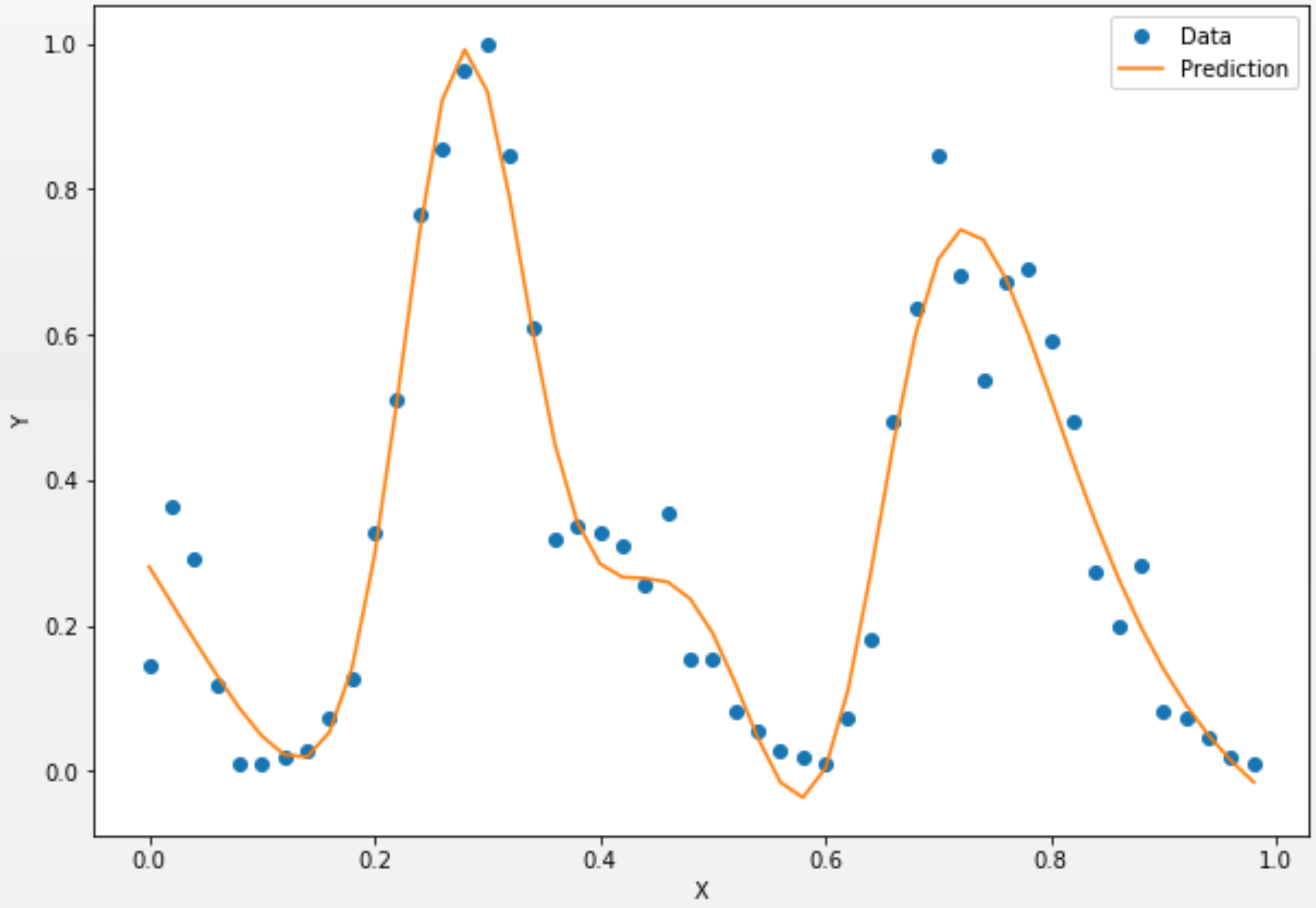
In [4]:

```
learning_rate = 0.0001 #设置学习率
losses = []
for i in range(1000000):
    # 从输入层到隐含层的计算
    hidden = x* weights + biases
    # 将sigmoid函数作用在隐含层的每一个神经元上
    hidden = torch.sigmoid(hidden)
    # 隐含层输出到输出层，计算得到最终预测
    predictions = hidden.mm(weights2)
    # 通过与标签数据y比较，计算误差
    loss = torch.mean((predictions - y) ** 2)
    if i % 10000 == 0:
        print('loss:', loss)
    loss.backward() #对损失函数进行梯度反传
    #利用上一步计算中得到的weights，biases等梯度信息更新weights或biases中的data数值
    weights.data.add_(- learning_rate * weights.grad.data)
    biases.data.add_(- learning_rate * biases.grad.data)
    weights2.data.add_(- learning_rate * weights2.grad.data)
    # 梯度清空
    weights.grad.data.zero_()
    biases.grad.data.zero_()
    weights2.grad.data.zero_()
```

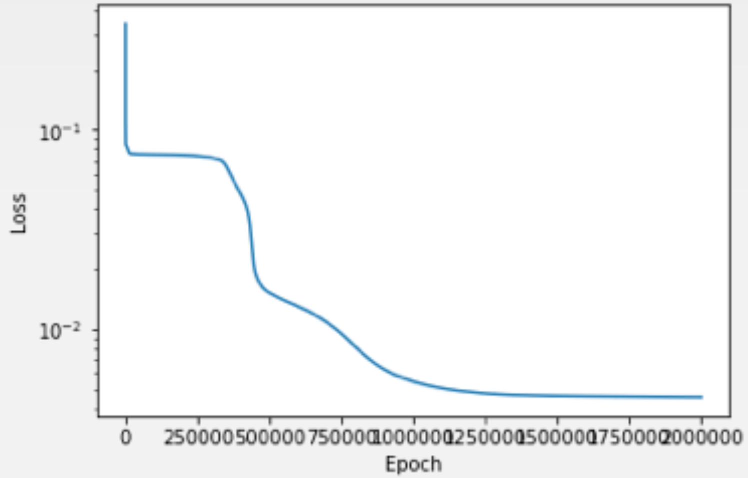
predictions.size()=(50, 1)

loss.size()=()

拟合结果



Data
Prediction
1.0
0.8
0.6
0.4
0.2
0.0
Y
0.0
0.2
0.4
0.6
0.8
1.0
X



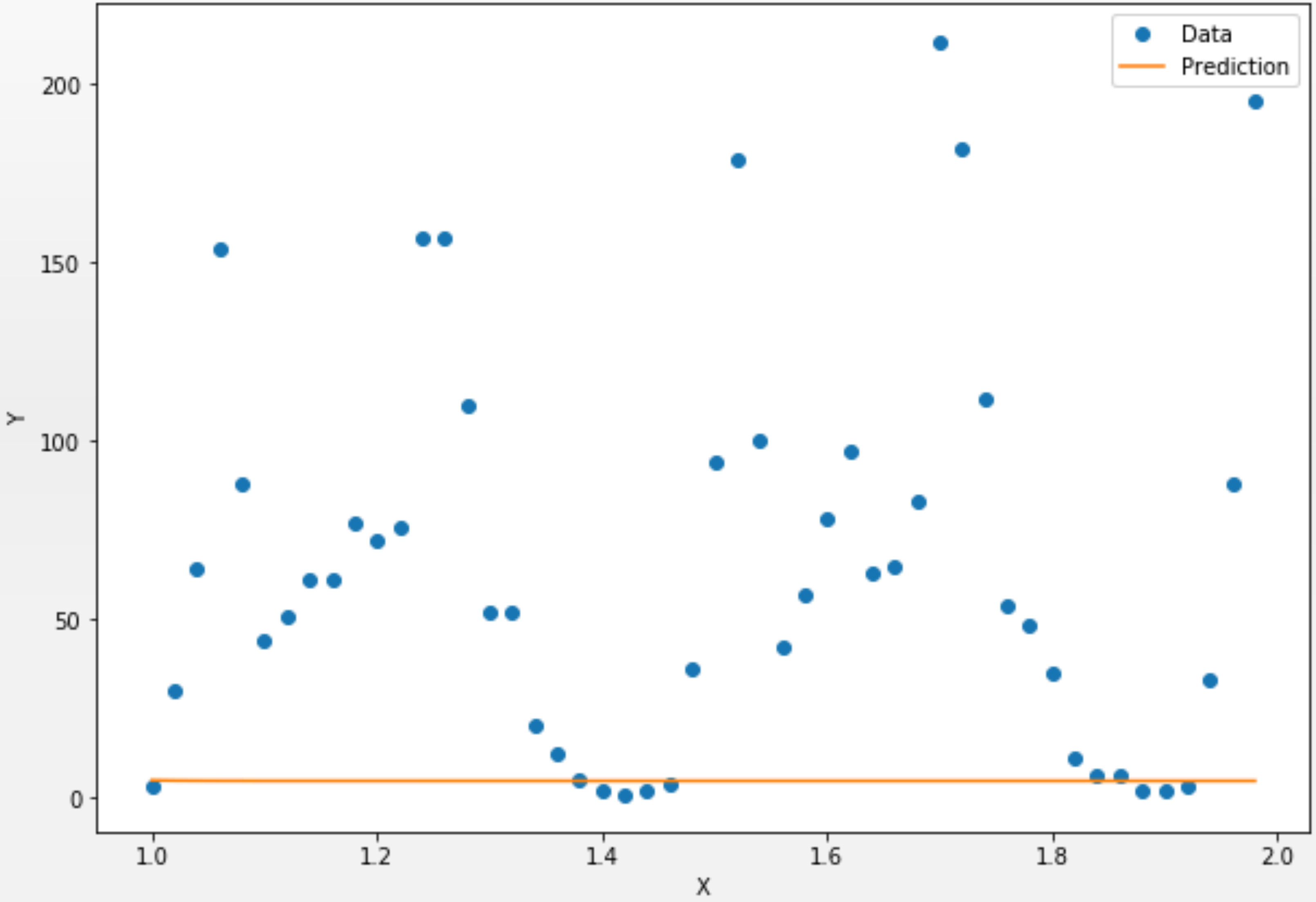
10^{-1}
10^{-2}
Loss
0
250000
500000
750000
1000000
1250000
1500000
1750000
2000000
Epoch

预测代码

In [5]:

```
counts_predict = rides['cnt'][50:100]#读取待预测的接下来的50个数据点

#首先对接下来的50个数据点进行选取，注意x应该取51，52，……，100，然后再归一化
x = torch.tensor((np.arange(len(counts_predict), dtype = float) + len(counts))/ len(counts_predict)
               , requires_grad = True)

#读取下50个点的y数值
y = torch.tensor(np.array(counts_predict, dtype = float), requires_grad = True)

x = x.unsqueeze(1)
y = y.unsqueeze(1)
# 从输入层到隐含层的计算
hidden = x * weights + biases

# 将sigmoid函数作用在隐含层的每一个神经元上
hidden = torch.sigmoid(hidden)

# 隐含层输出到输出层，计算得到最终预测
predictions = hidden.mm(weights2)

# 计算预测数据上的损失函数
loss = torch.mean((predictions - y) ** 2)
print(loss)
```

预测结果



Loss: 6710.35

存在问题

- 存在着严重的过拟合现象
- 采用单一属性（编号）预测未来单车数量效果太差
  - 事实上，单车使用数量和下标之间根本就没有关系！！！
- 运行速度缓慢
- 需要考虑其它可获得信息：是否工作日、天气情况：风速湿度等

接下来，你将会学到

- 如何设计一个多输入的神经网络
- 如何对数据进行预处理
- 如何以及为什么对数据分撮（Batch）
- 如何用pytorch简化神经网络构建流程

再来看看数据

http://capitalbikeshare.com/system-data

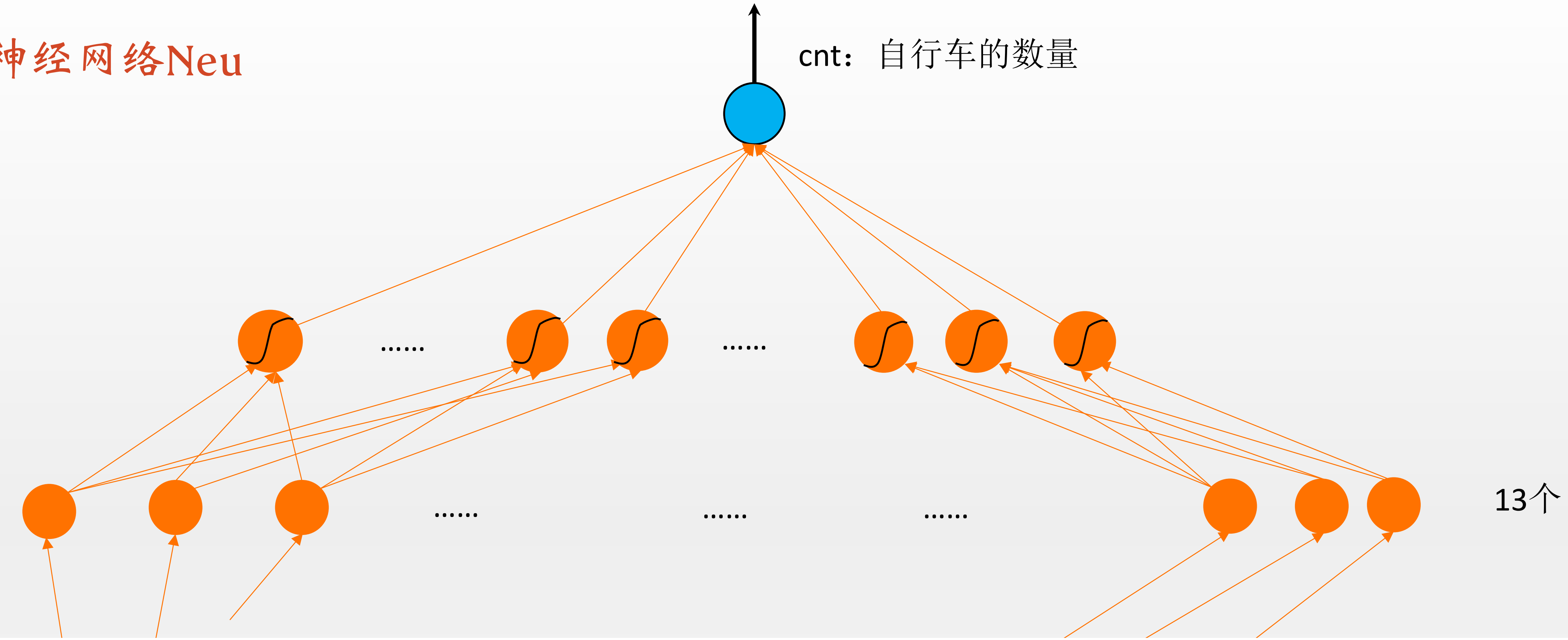
http://www.freemeteo.com

编号 季节 月 是否假期 是否工作日 温度 湿度 出行数量

	instant	dteday	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	cnt
0	1	2011/1/1	1	0	1	0	0	6	0	1	0.24	0.2879	0.81	0.0	16
1	2	2011/1/1	1	0	1	1	0	6	0	1	0.22	0.2727	0.80	0.0	40
2	3	2011/1/1	1	0	1	2	0	6	0	1	0.22	0.2727	0.80	0.0	32
3	4	2011/1/1	1	0	1	3	0	6	0	1	0.24	0.2879	0.75	0.0	13
4	5	2011/1/1	1	0	1	4	0	6	0	1	0.24	0.2879	0.75	0.0	1

日期 年 小时 星期几 天气，1晴，2雾…… 体表温度 风速

神经网络Neu



	instant	dteday	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt
0	1	2011-01-01	1	0	1	0	0	6	0	1	0.24	0.2879	0.81	0.0	3	13	16
1	2	2011-01-01	1	0	1	1	0	6	0	1	0.22	0.2727	0.80	0.0	8	32	40
2	3	2011-01-01	1	0	1	2	0	6	0	1	0.22	0.2727	0.80	0.0	5	27	32
3	4	2011-01-01	1	0	1	3	0	6	0	1	0.24	0.2879	0.75	0.0	3	10	13
4	5	2011-01-01	1	0	1	4	0	6	0	1	0.24	0.2879	0.75	0.0	0	1	1

数据预处理：类型变量

- 类型数据的处理
- Weekday：1，2，3，4，5，6，0

星期	类型变量	类型编码
星期日	0	1 0 0 0 0 0 0
星期一	1	0 1 0 0 0 0 0
星期二	2	0 0 1 0 0 0 0
星期三	3	0 0 0 1 0 0 0
星期四	4	0 0 0 0 1 0 0
星期五	5	0 0 0 0 0 1 0
星期六	6	0 0 0 0 0 0 1

数据预处理：类型变量

	holiday	weekday	workingday	weathersit	temp	atemp
1	0	6	0	2	0.344167	0.363625
1	0	0	0	2	0.363478	0.353739
1	0	1	1	1	0.196364	0.189405
1	0	2	1	1	0.2	0.212122
1	0	3	1	1	0.226957	0.22927
1	0	4	1	1	0.204348	0.233209

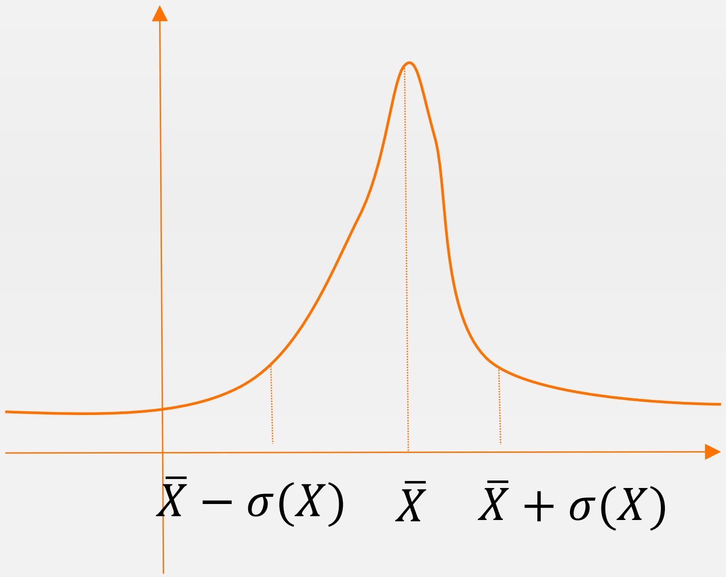
hr_23	weekday_0	weekday_1	weekday_2	weekday_3	weekday_4	weekday_5	weekday_6
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1

One-hot编码

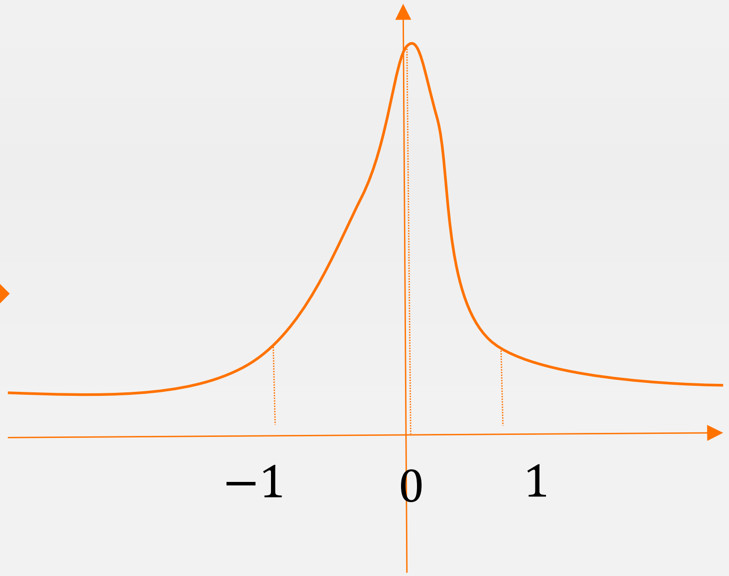
将一个变量扩展为n个变量，n为类型数目

数据预处理：数值类型变量归一化

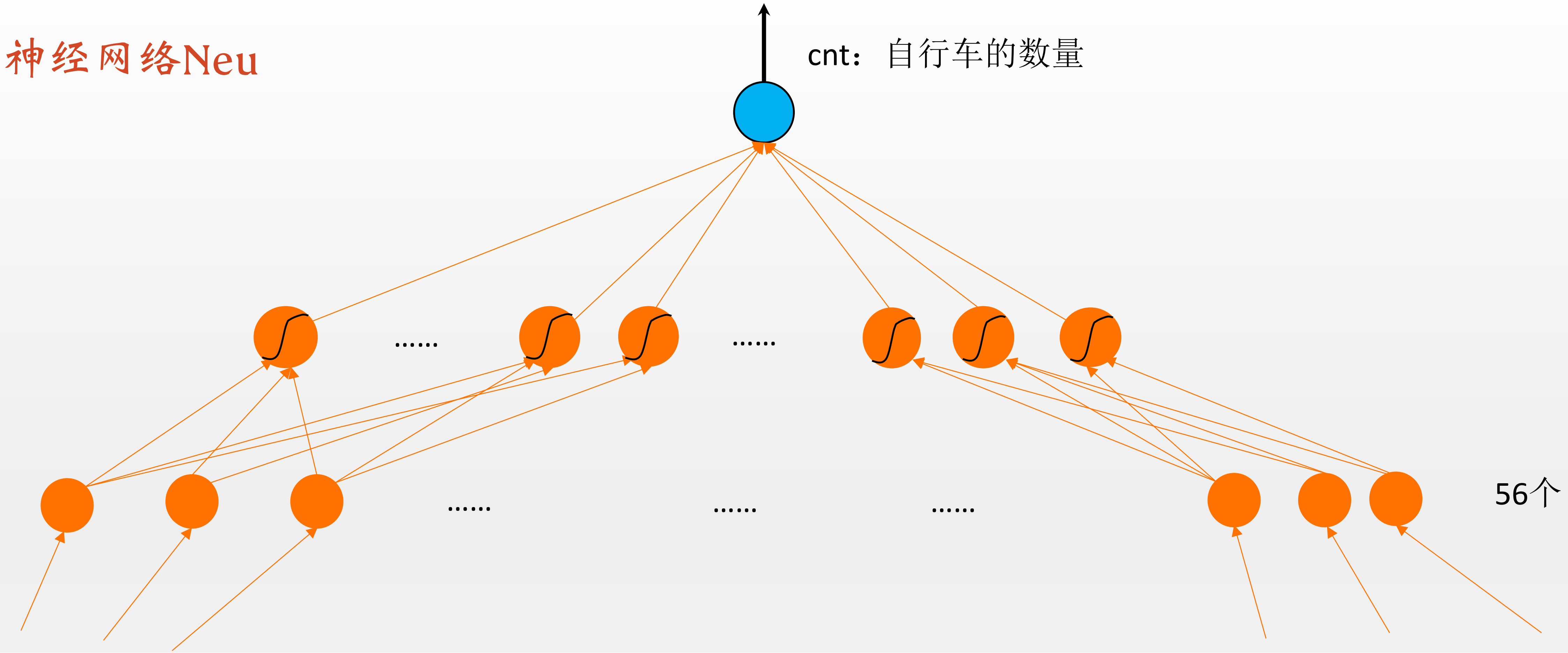
temp	atemp	hum
0.24	0.2879	0.81
0.22	0.2727	0.80
0.22	0.2727	0.80
0.24	0.2879	0.75
0.24	0.2879	0.75



$$\frac{X - \bar{X}}{\sigma(X)}$$

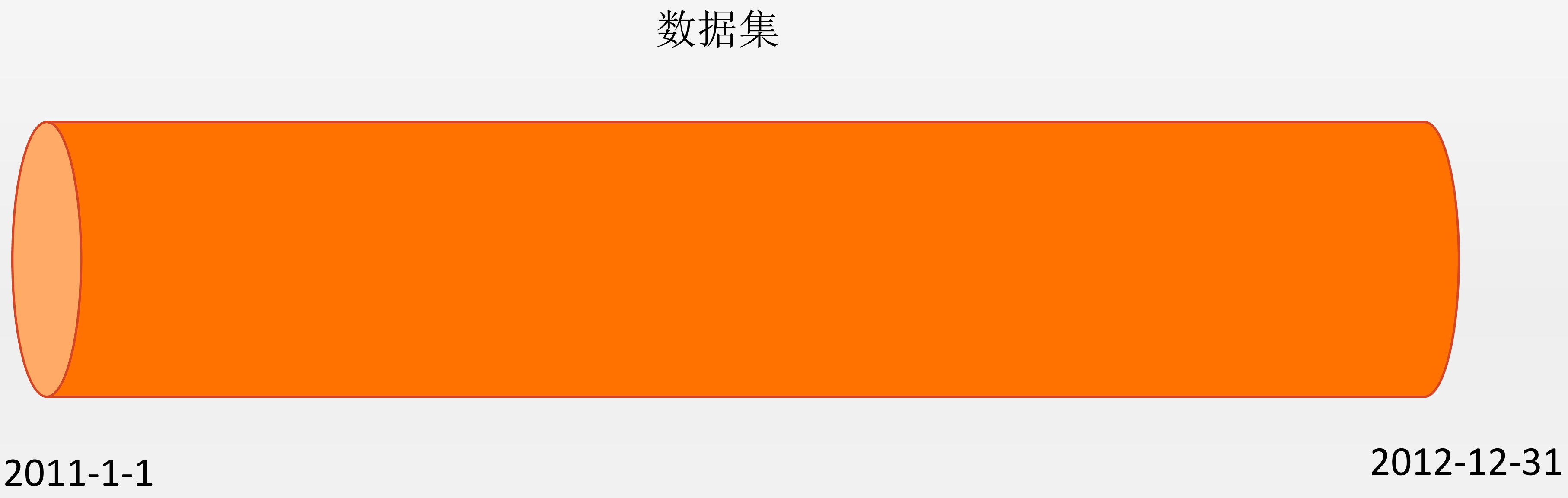


神经网络Neu

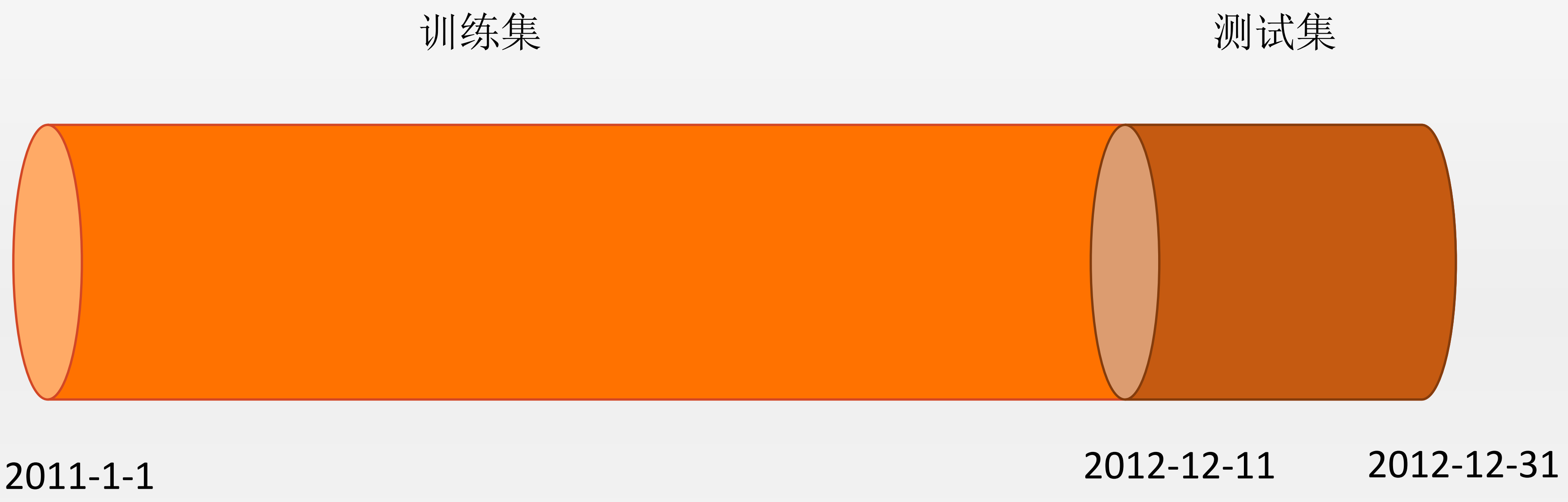


	yr	holiday	temp	hum	windspeed	casual	registered	cnt	season_1	season_2	...	hr_21	hr_22	hr_23	weekday_0	weekday_1	weekday_2	weekday_
0	0	0	0.24	0.81	0.0	3	13	16	1	0	...	0	0	0	0	0	0	
1	0	0	0.22	0.80	0.0	8	32	40	1	0	...	0	0	0	0	0	0	
2	0	0	0.22	0.80	0.0	5	27	32	1	0	...	0	0	0	0	0	0	
3	0	0	0.24	0.75	0.0	3	10	13	1	0	...	0	0	0	0	0	0	
4	0	0	0.24	0.75	0.0	0	1	1	1	0	...	0	0	0	0	0	0	

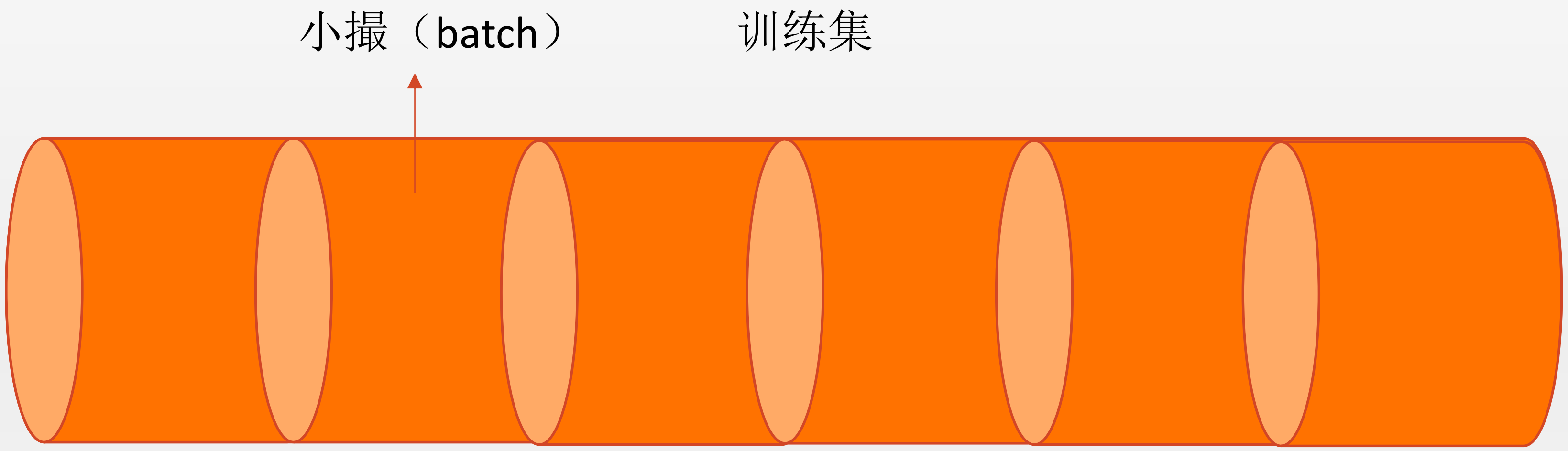
数据准备



数据准备



将数据分撮处理



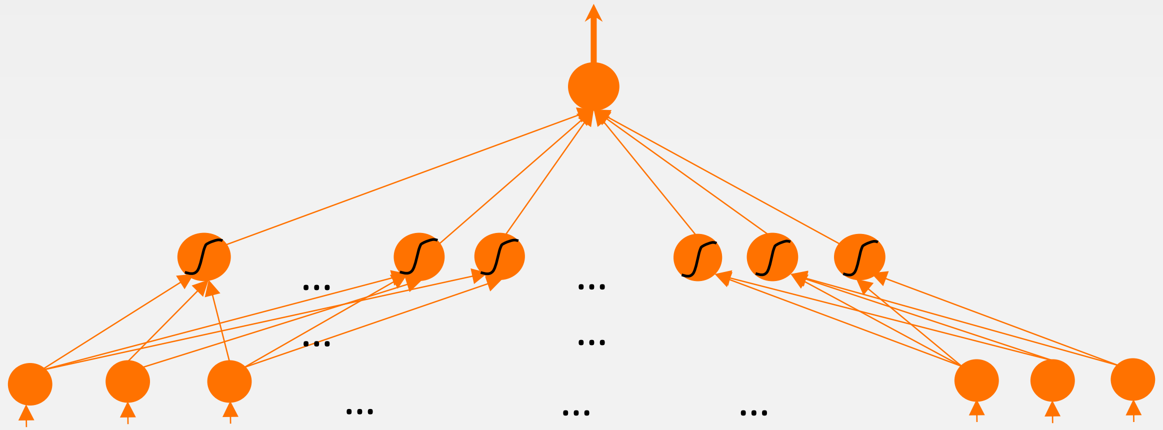
- 如何训练？
  - 将训练切割成小的撮（batch）
  - 对每一个小撮进行误差计算、反向传播，调整权重

建立神经网络

数据库预处理后的特征列数量

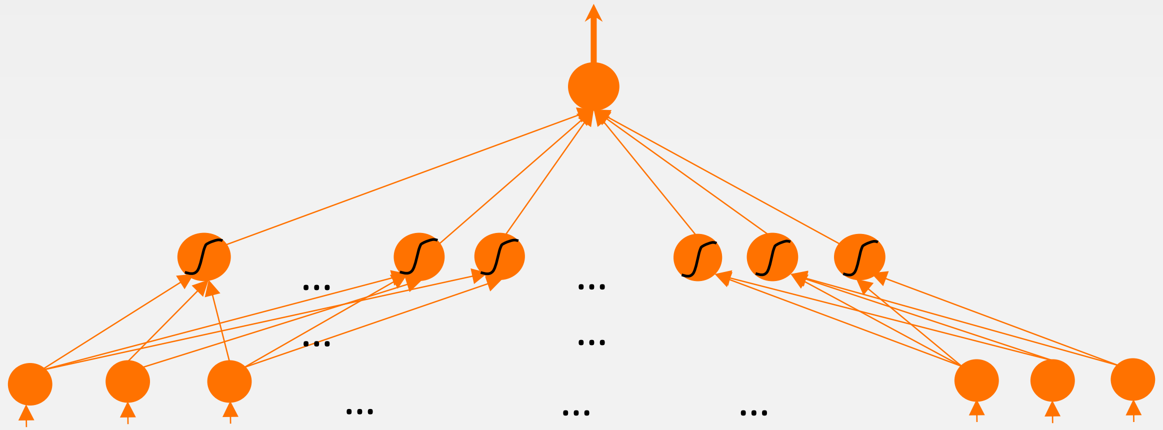
```
input_size = features.shape[1]
output_size = 1
weights1 = torch.randn([input_size, hidden_size], requires_grad = True)

biases = torch.randn([hidden_size], requires_grad = True)

weights2 = torch.randn([hidden_size, output_size], requires_grad = True)
```

- 定义一个input_size*hidden_size的第一层权重矩阵
- 定义一个hidden_size尺度的向量biases
- 定义一个hidden_size*1的第二层权重矩阵



建立神经网络

```
neu= torch.nn.Sequential(
    torch.nn.Linear(input_size, hidden_size),
    torch.nn.Sigmoid(),
    torch.nn.Linear(hidden_size, output_size),
)
```



- 建立一个多步操作的神经网络模型
- 第一步从输入层到隐含层节点为一个线性运算，输入维度input_size，隐含维度hidden_size
- 第二步为Sigmoid，作用到每一个隐含层神经元上
- 第三步又是一个线性元算，从隐含到输出，神经元个数分别为hidden_size和output_size
- 所有神经网络的参数都存储在neu.parameters()里面了

建立损失函数和优化器

```
cost = torch.nn.MSELoss()
optimizer = torch.optim.SGD(neu.parameters(), lr = 0.01)
```

- torch.nn.MSELoss() 等价于函数torch.mean((x-y)^2)
- torch.optim.SGD随机梯度下降算法
  - neu.parameters()返回神经网络neu的所有权重、偏置参数

主要训练循环

```
# 神经网络训练循环
for i in range(2000):
    batch_loss = [] #记录每一个撮的损失
    # 每128个样本点被划分为一个撮
    # start和end分别是提取一个batch数据的起始和终止下标
    for start in range(0, len(X), batch_size):
        end = start + batch_size if start + batch_size < len(X) else len(X)
        xx = torch.tensor(X[start:end], dtype = torch.float, requires_grad = True)
        yy = torch.tensor(Y[start:end], dtype = torch.float, requires_grad = True)
        predict = neu(xx) # 模型预测
        loss = cost(predict, yy) # 计算损失函数（均方误差）
        optimizer.zero_grad() # 将优化器存储的那些参数的梯度设置为0
        loss.backward() # 开始反向传播，计算所有梯度值
        optimizer.step() # 优化器开始运行一步，更新所有的参数
        batch_loss.append(loss.data.numpy())
    # 每隔100步输出一下损失值（loss）
    if i % 100==0:
        losses.append(np.mean(batch_loss))
        print(i, np.mean(batch_loss))
```

主要训练循环

```
# 神经网络训练循环
for i in range(2000):
    batch_loss = [] #记录每一个撮的损失
    # 每128个样本点被划分为一个撮
    # start和end分别是提取一个batch数据的起始和终止下标
    for start in range(0, len(X), batch_size):
        end = start + batch_size if start + batch_size < len(X) else len(X)
        xx = torch.tensor(X[start:end], dtype = torch.float, requires_grad = True)
        yy = torch.tensor(Y[start:end], dtype = torch.float, requires_grad = True)
        predict = neu(xx) # 模型
        loss = cost(predict, yy) #
        optimizer.zero_grad() #
        loss.backward() # 开始反向传播，计算所有梯度值
        optimizer.step() # 优化器开始运行一步，更新所有的参数
        batch_loss.append(loss.data.numpy())
    # 每隔100步输出一下损失值（loss）
    if i % 100==0:
        losses.append(np.mean(batch_loss))
        print(i, np.mean(batch_loss))
```

```
hidden = x* weights+ biases
hidden = torch.sigmoid(hidden)
predictions = hidden.mm(weights2)
```

主要训练循环

```
# 神经网络训练循环
for i in range(2000):
    batch_loss = [] #记录每一个撮的损失
    # 每128个样本点被划分为一个撮
    # start和end分别是提取一个batch数据的起始和终止下标
    for start in range(0, len(X), batch_size):
        end = start + batch_size if start + batch_size < len(X) else len(X)
        xx = torch.tensor(X[start:end], dtype = torch.float, requires_grad = True)
        yy = torch.tensor(Y[start:end], dtype = torch.float, requires_grad = True)
        predict = neu(xx) # 模型预测
        loss = cost(predict, yy) # 计算
        optimizer.zero_grad() # 将优
        loss.backward() # 开始反向传
        optimizer.step() # 优化器开始运行一步，更新所有的参数
        batch_loss.append(loss.data.numpy())
    # 每隔100步输出一下损失值（loss）
    if i % 100==0:
        losses.append(np.mean(batch_loss))
        print(i, np.mean(batch_loss))
```

```
weights.grad.data.zero_()
biases.grad.data.zero_()
weights2.grad.data.zero_()
```

主要训练循环

```
# 神经网络训练循环
for i in range(2000):
    batch_loss = [] #记录每一个撮的损失
    # 每128个样本点被划分为一个撮
    # start和end分别是提取一个batch数据的起始和终止下标
    for start in range(0, len(X), batch_size):
        end = start + batch_size if start + batch_size < len(X) else len(X)
        xx = torch.tensor(X[start:end], dtype = torch.float, requires_grad = True)
        yy = torch.tensor(Y[start:end], dtype = torch.float, requires_grad = True)
        predict = neu(xx) # 模型预测
        loss = cost(predict, yy) # 计算损失函数（均方误差）
        optimizer.zero_grad() # 将优化器存储的那些参数的梯度设置为0
        loss.backward() # 开始
        optimizer.step() # 优化
        batch_loss.append(loss
    # 每隔100步输出一下损
    if i % 100==0:
        losses.append(np.mean(batch_loss))
        print(i, np.mean(batch_loss))
```

```
weights.data.add_(- learning_rate * weights.grad.data)
biases.data.add_(- learning_rate * biases.grad.data)
weights2.data.add_(- learning_rate * weights2.grad.data)
```

主要训练循环

```python
# 神经网络训练循环
for i in range(2000):
    batch_loss = [] #记录每一个撮的损失
    # 每128个样本点被划分为一个撮
    # start和end分别是提取一个batch数据的起始和终止下标
    for start in range(0, len(X), batch_size):
        end = start + batch_size if start + batch_size < len(X) else len(X)
        xx = torch.tensor(X[start:end], dtype = torch.float, requires_grad = True)
        yy = torch.tensor(Y[start:end], dtype = torch.float, requires_grad = True)
        predict = neu(xx) # 模型预测
        loss = cost(predict, yy) # 计算损失函数（均方误差）
        optimizer.zero_grad() # 将优化器存储的那些参数的梯度设置为0
        loss.backward() # 开始反向传播，计算所有梯度值
        optimizer.step() # 优化器开始运行一步，更新所有的参数
        batch_loss.append(loss.data.numpy())
    # 每隔100步输出一下损失值（loss）
    if i % 100==0:
        losses.append(np.mean(batch_loss))
        print(i, np.mean(batch_loss))
```

xx.size()=(128,56)

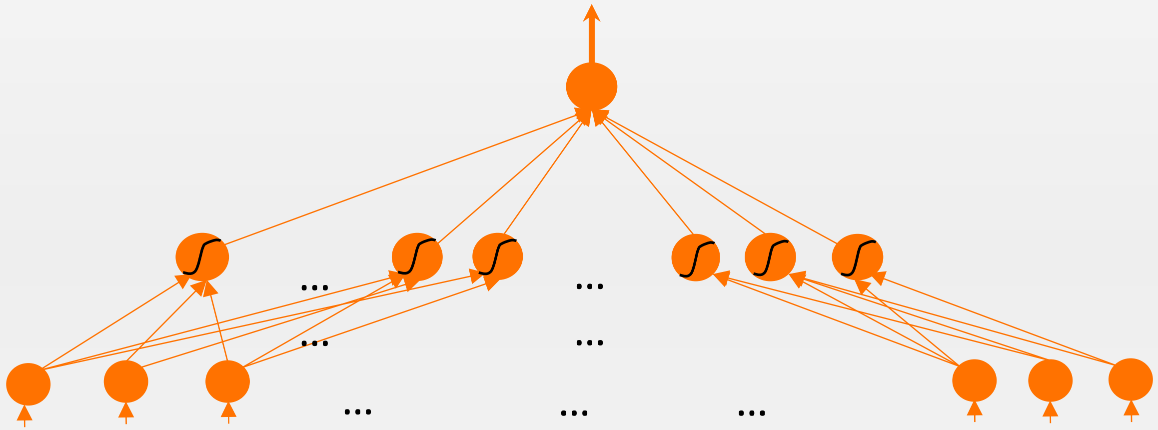
yy.size()=(128,1)

主要训练循环

```
# 神经网络训练循环
for i in range(2000):
    batch_loss = [] #记录每一个撮的损失
    # 每128个样本点被划分为一个撮
    # start和end分别是提取一个batch数据的起始和终止下标
    for start in range(0, len(X), batch_size):
        end = start + batch_size if start + batch_size < len(X) else len(X)
        xx = torch.tensor(X[start:end], dtype = torch.float, requires_grad = True)
        yy = torch.tensor(Y[start:end], dtype = torch.float, requires_grad = True)
        predict = neu(xx) # 模型预测
        loss = cost(predict, yy) # 计算损失函数（均方误差）
        optimizer.zero_grad() # 将优化器存储的那些参数的梯度设置为0
        loss.backward() # 开始反向传播，计算所有梯度值
        optimizer.step() # 优化器开始运行一步，更新所有的参数
        batch_loss.append(loss.data.numpy())
    # 每隔100步输出一下损失值（loss）
    if i % 100==0:
        losses.append(np.mean(batch_loss))
        print(i, np.mean(batch_loss))
```

predict.size()=(128,1)

神经网络的运算过程




```
neu= torch.nn.Sequential(
    torch.nn.Linear(input_size, hidden_size),
    torch.nn.Sigmoid(),
    torch.nn.Linear(hidden_size, output_size),
)
```

```
predict=neu(x)
```

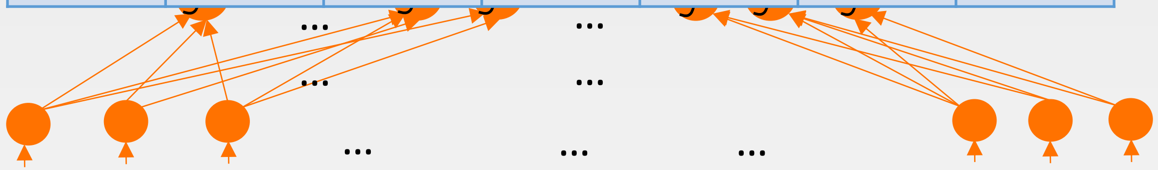
0.4	1.9	-0.1	0.2	1.1	3.2	0.3
-0.9	0.5	2.1	-0.9	-1.0	1.1	0.1
-0.9	0.5	2.1	-0.9	-1.0	1.1	0.1
0.4	1.9	-0.1	0.2	1.1	3.2	0.3

batch_size * input_size

神经网络的运算过程

0.4	1.9	-0.1	0.2	1.1	3.2	0.3
-0.9	0.5	2.1	-0.9	-1.0	1.1	0.1
-0.9	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.1
0.4	1.9	-0.1	0.2	1.1	3.2	0.3

batch_size * hidden_size




```
neu= torch.nn.Sequential(
    torch.nn.Linear(input_size, hidden_size),
    torch.nn.Sigmoid(),
    torch.nn.Linear(hidden_size, output_size),
)
```

```
predict=neu(x)
```

神经网络的运算过程

0.4	1.9	-0.1	0.2	1.1	3.2	0.3
-0.9	0.5	2.1	-0.9	-1.0	1.1	0.1
-0.9	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.1
0.4	1.9	-0.1	0.2	1.1	3.2	0.3

batch_size * hidden_size

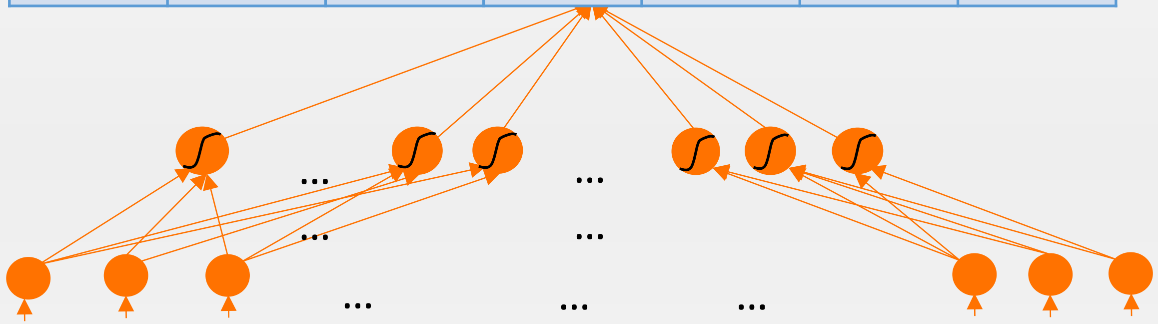

```
neu= torch.nn.Sequential(
    torch.nn.Linear(input_size, hidden_size),
    torch.nn.Sigmoid(),
    torch.nn.Linear(hidden_size, output_size),
)
```

```
predict=neu(x)
```

神经网络的运算过程

0.4	1.9	-0.1	0.2	1.1	3.2	0.3
-0.9	0.5	2.1	-0.9	-1.0	1.1	0.1
-0.9	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.1
0.4	1.9	-0.1	0.2	1.1	3.2	0.3

batch_size * output_size



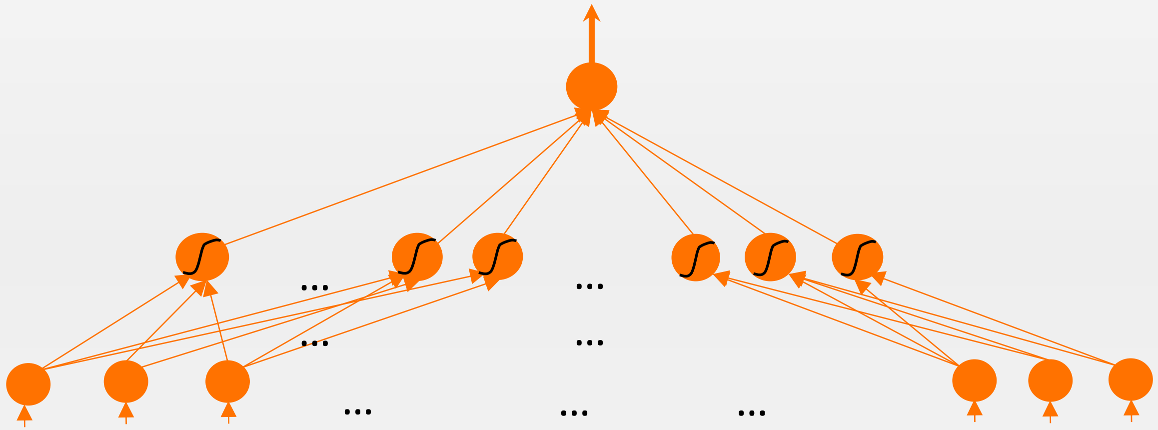
```
neu= torch.nn.Sequential(
    torch.nn.Linear(input_size, hidden_size),
    torch.nn.Sigmoid(),
    torch.nn.Linear(hidden_size, output_size),
)
```

```
predict=neu(x)
```

神经网络的运算过程

0.4	1.9	-0.1	0.2	1.1	3.2	0.3
-0.9	0.5	2.1	-0.9	-1.0	1.1	0.1
-0.9	0.5	[illegible]	[illegible]	[illegible]	1.1	0.1
0.4	1.9	-0.1	0.2	1.1	3.2	0.3

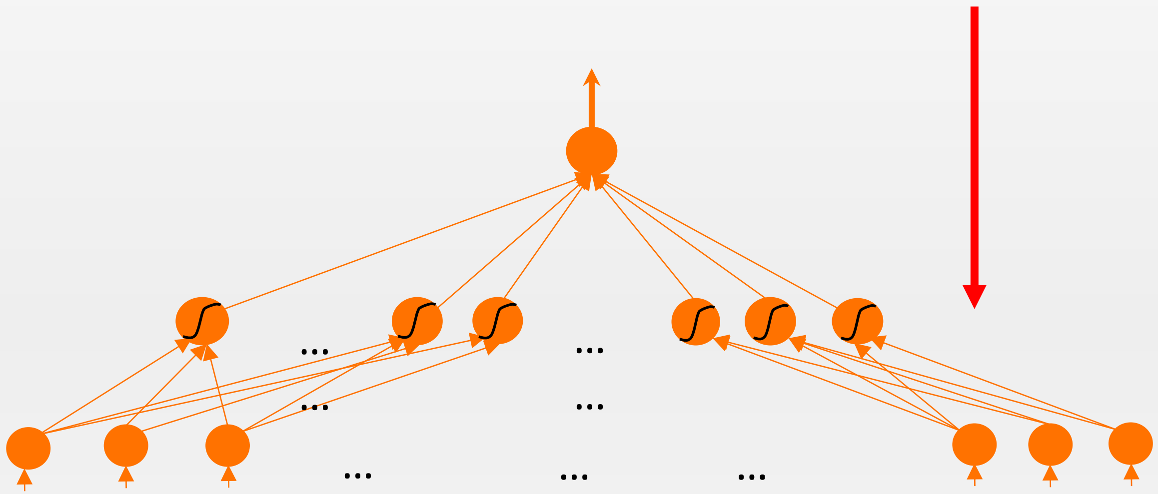
batch_size * output_size




```
neu= torch.nn.Sequential(
    torch.nn.Linear(input_size, hidden_size),
    torch.nn.Sigmoid(),
    torch.nn.Linear(hidden_size, output_size),
)
```

```
predict=neu(x)
```

神经网络的运算过程




```
neu= torch.nn.Sequential(
    torch.nn.Linear(input_size, hidden_size),
    torch.nn.Sigmoid(),
    torch.nn.Linear(hidden_size, output_size),
)
```

```
loss.backward()
```

注意：每一个批次，都进行一次梯度反传并更新所有**Linear**里面的权重和**biases**

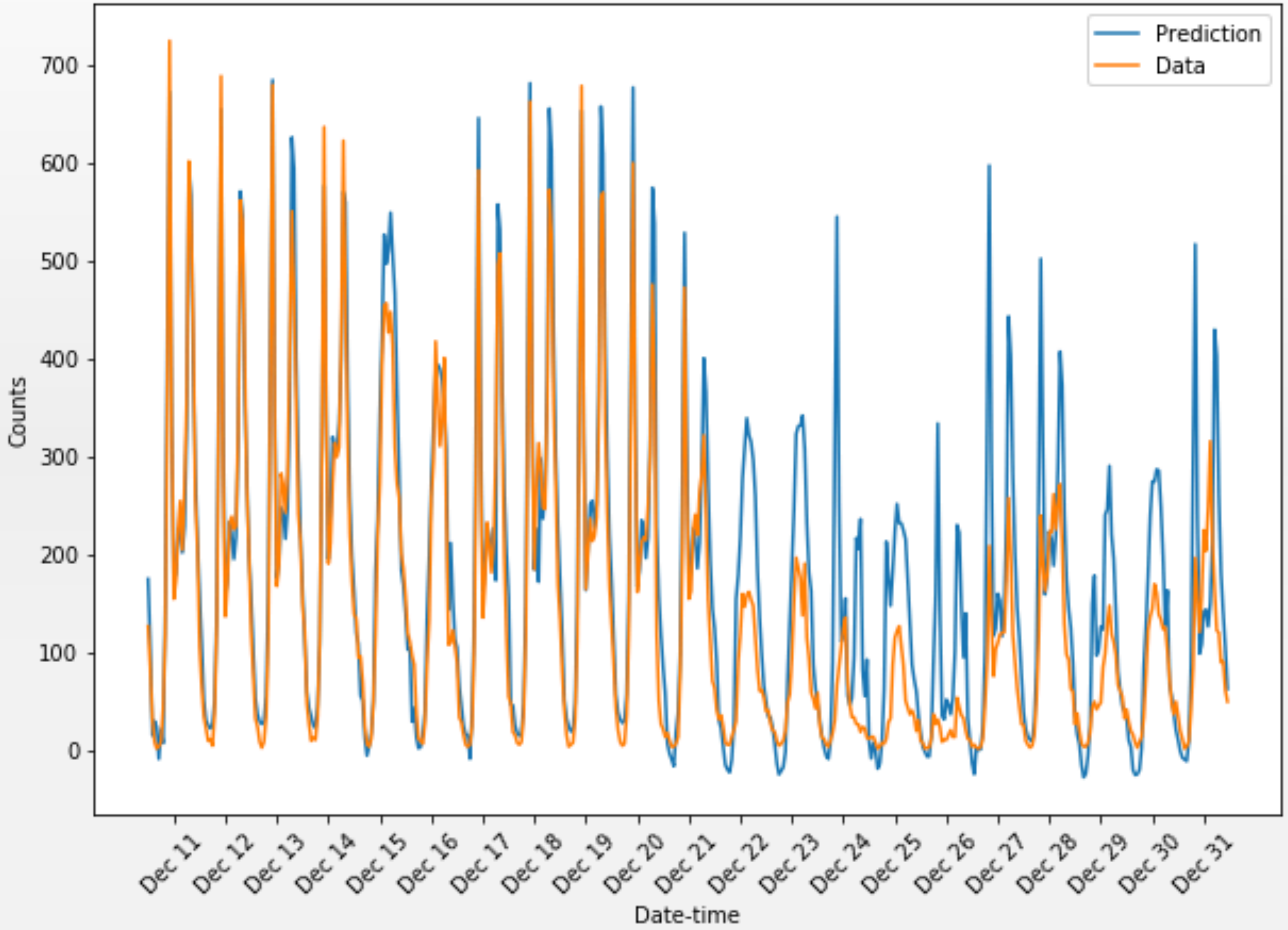
为什么可以分批次梯度下降?

$$L = \sum_{i=1}^{N} L_i = \sum_{i \in B_1} L_i + \sum_{i \in B_2} L_i + \cdots + \sum_{i \in B_M} L_i$$

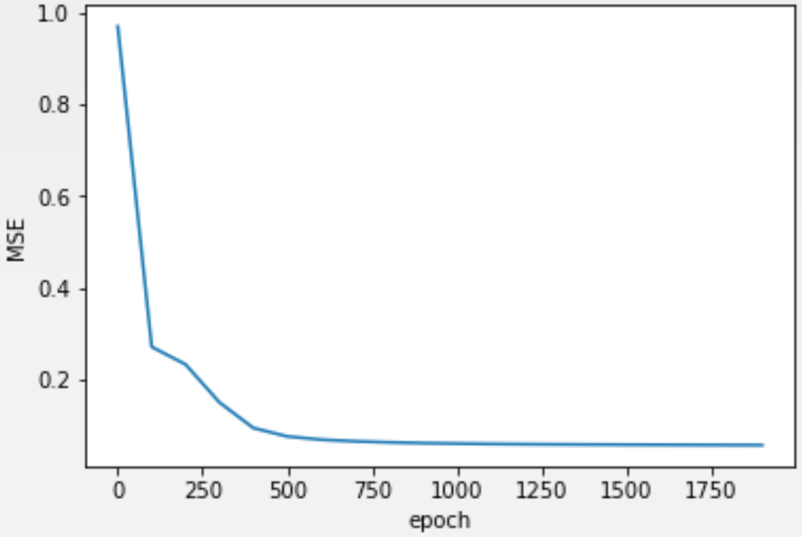
$$\partial L / \partial X = \partial \sum_{i=1}^{N} L_i \,/ \partial X = \partial \sum_{i \in B_1} L_i \,/ \partial X + \partial \sum_{i \in B_2} L_i \,/ \partial X + \cdots + \partial \sum_{i \in B_M} L_i \,/ \partial X$$

只要**Loss**函数是线性可分的，都可以分批次梯度下降

运行结果

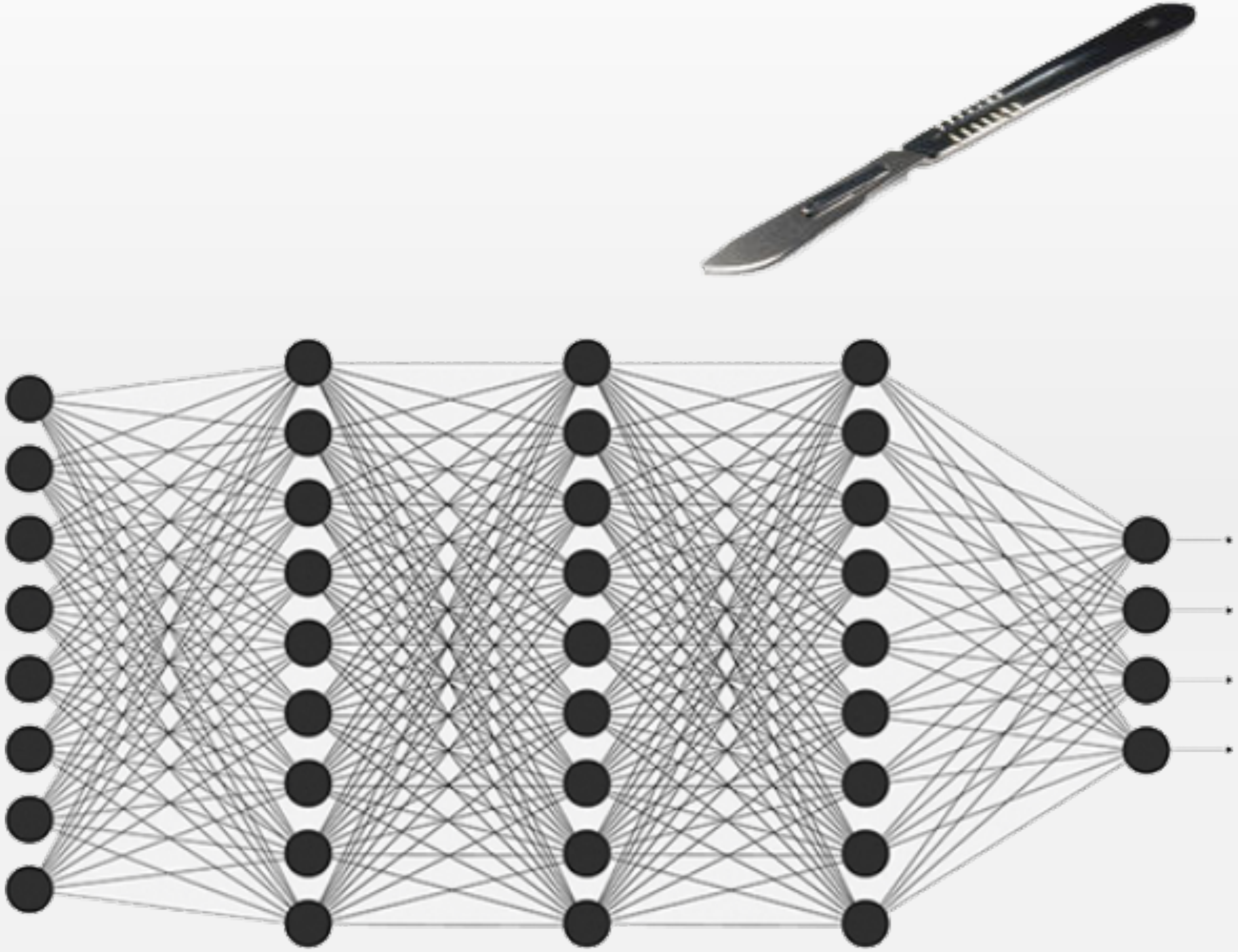


Prediction
Data
Counts
700
600
500
400
300
200
100
0
Dec 11
Dec 12
Dec 13
Dec 14
Dec 15
Dec 16
Dec 17
Dec 18
Dec 19
Dec 20
Dec 21
Dec 22
Dec 23
Dec 24
Dec 25
Dec 26
Dec 27
Dec 28
Dec 29
Dec 30
Dec 31
Date-time

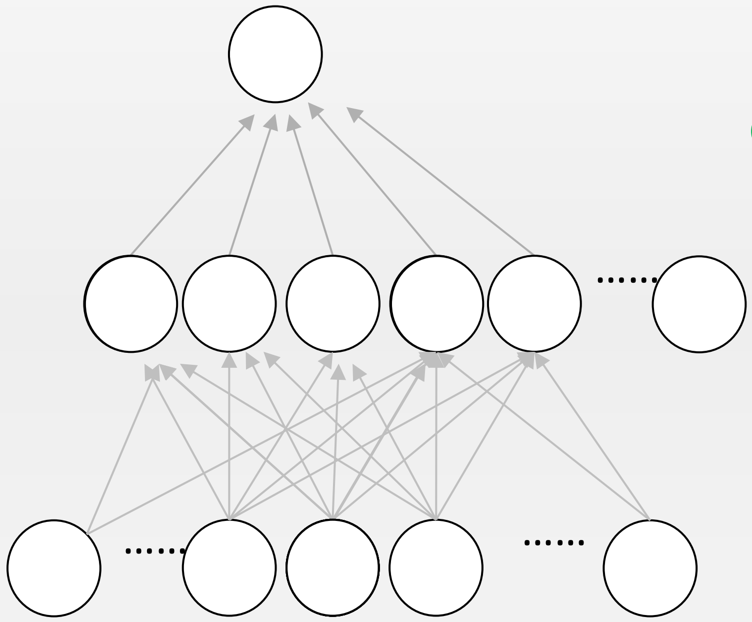


MSE
1.0
0.8
0.6
0.4
0.2
0
250
500
750
1000
1250
1500
1750
epoch

解剖神经网

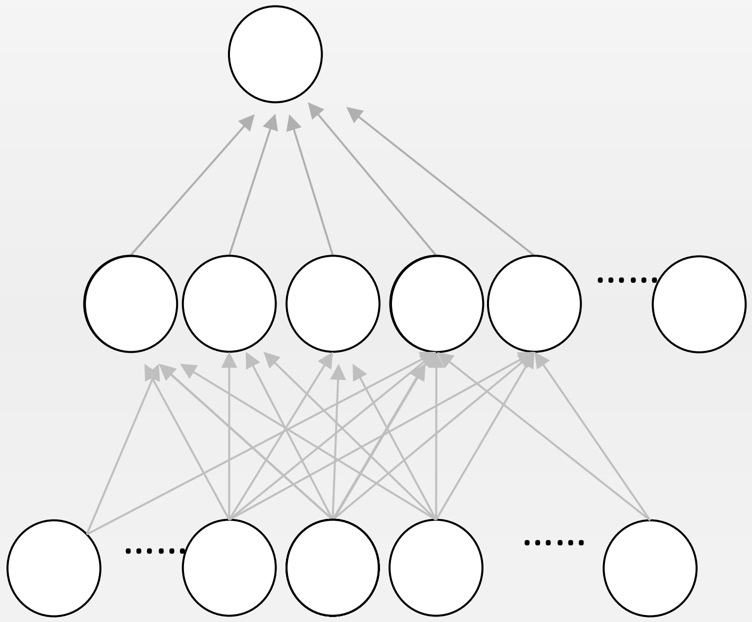


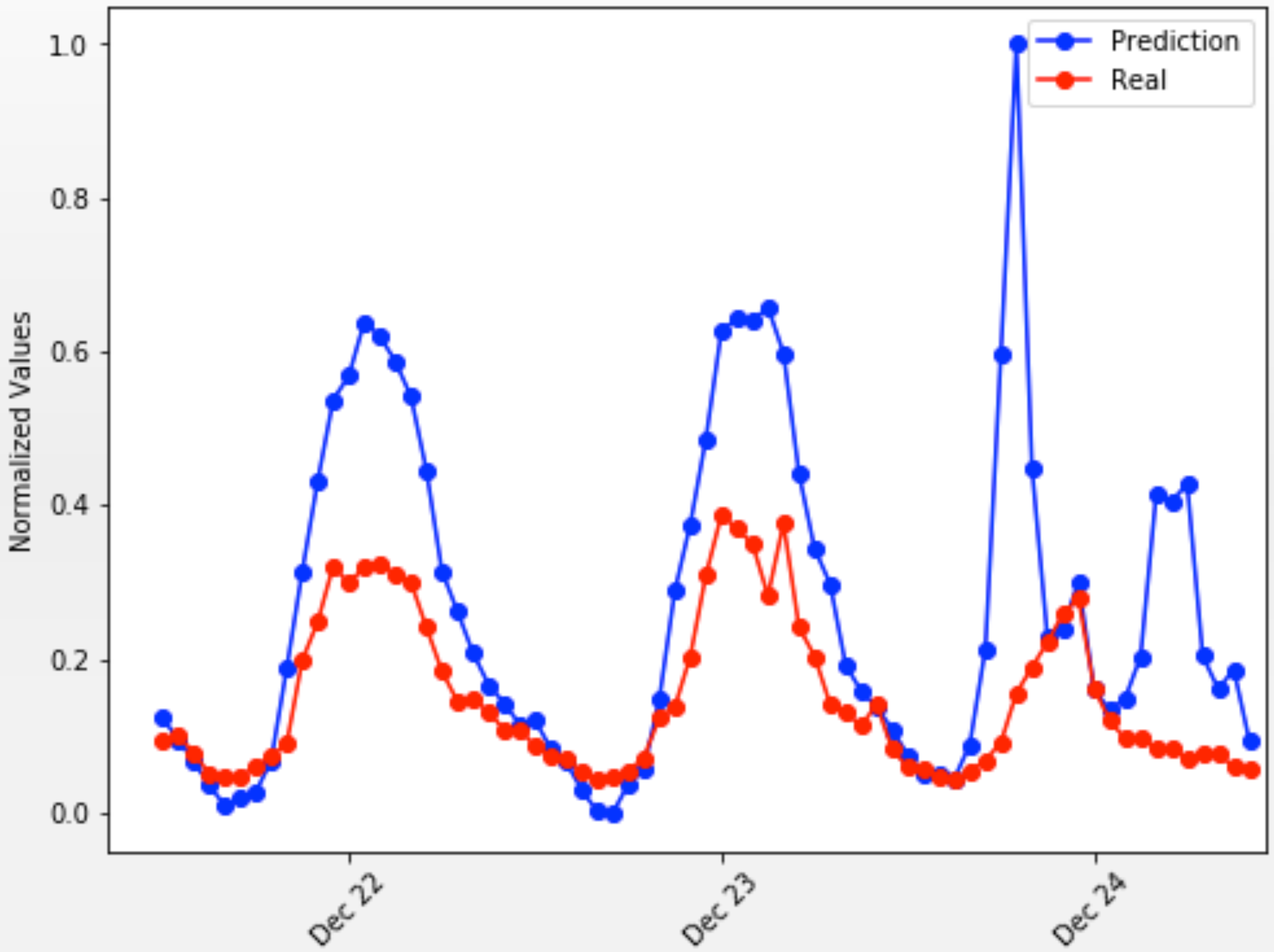
对Neu进行诊断




```
def feature(X, net):
    X = torch.tensor(X, dtype = torch.float, requires_grad = False)
    dic = dict(net.named_parameters())
    weights = dic['0.weight']
    biases = dic['0.bias']
    h = torch.sigmoid(X.mm(weights.t()) + biases.expand([len(X), len(biases)]))
    return h
```

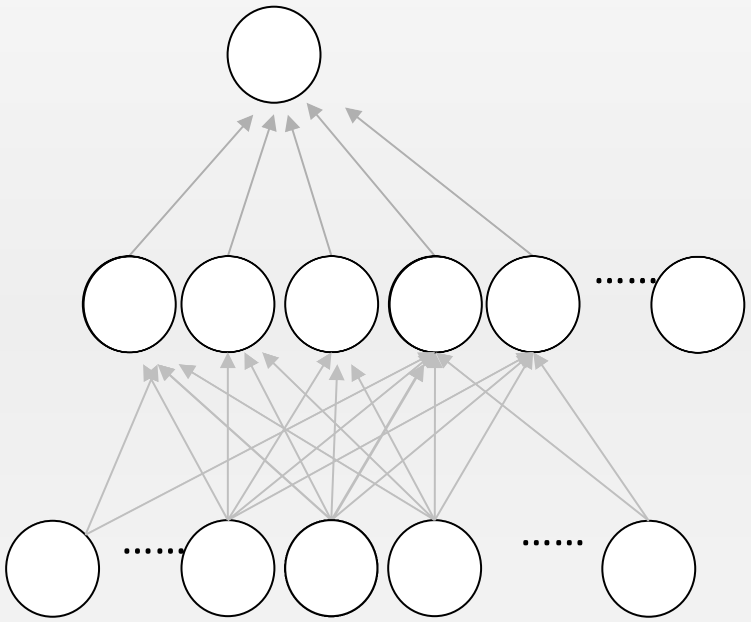
对Neu进行诊断

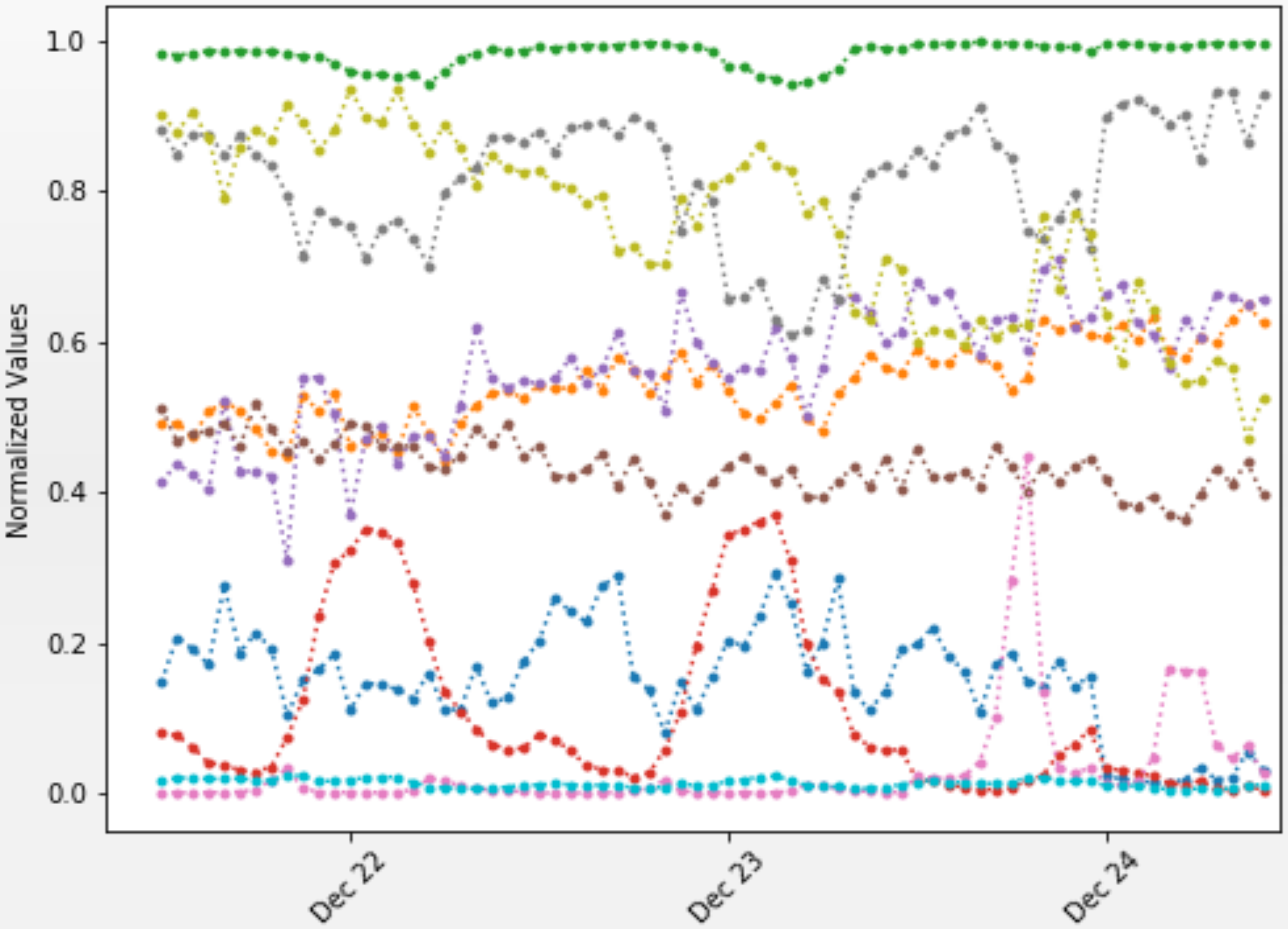




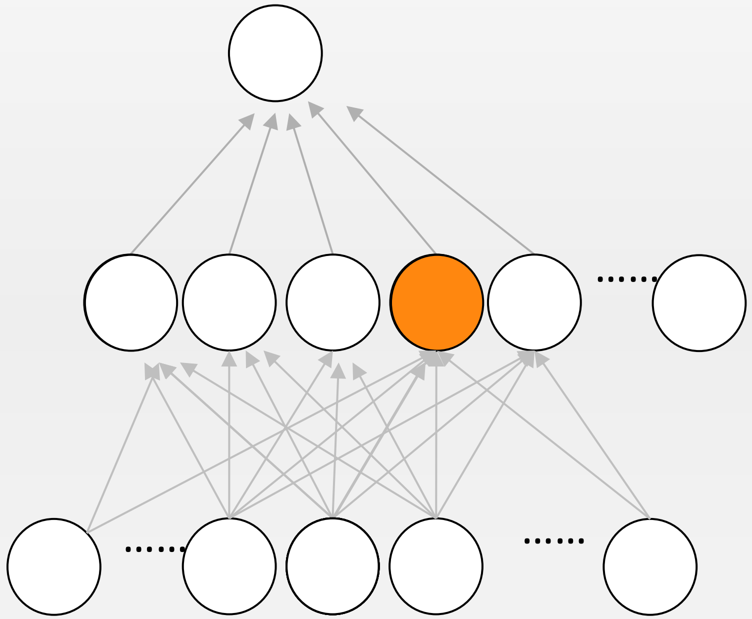
Prediction
Real
1.0
0.8
0.6
0.4
0.2
0.0
Normalized Values
Dec 22
Dec 23
Dec 24

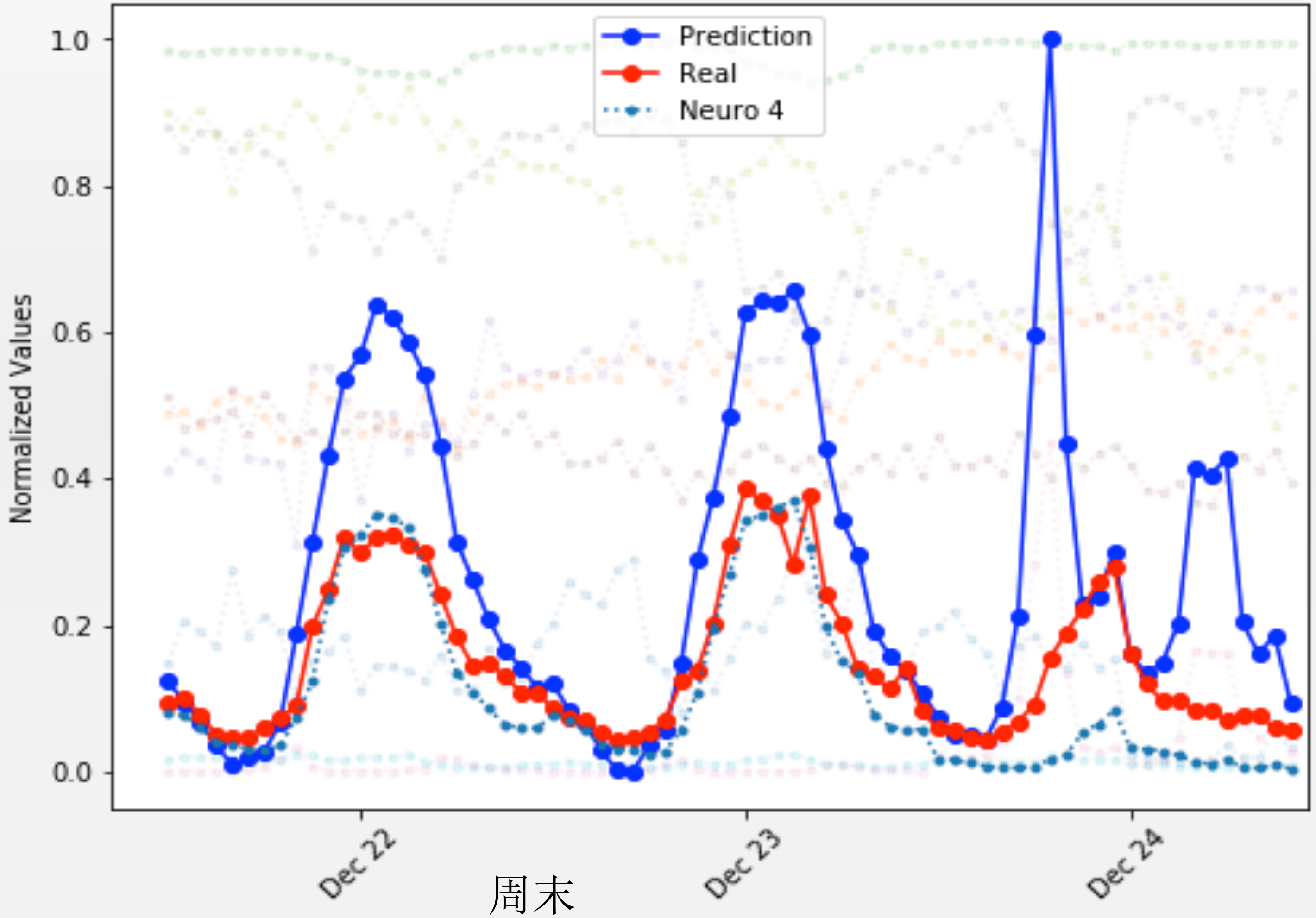
解剖Neu





解剖Neu

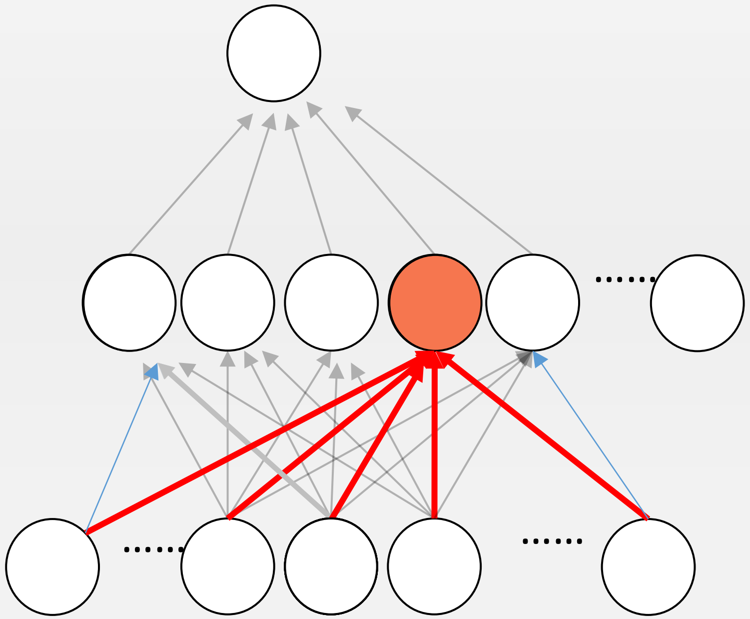


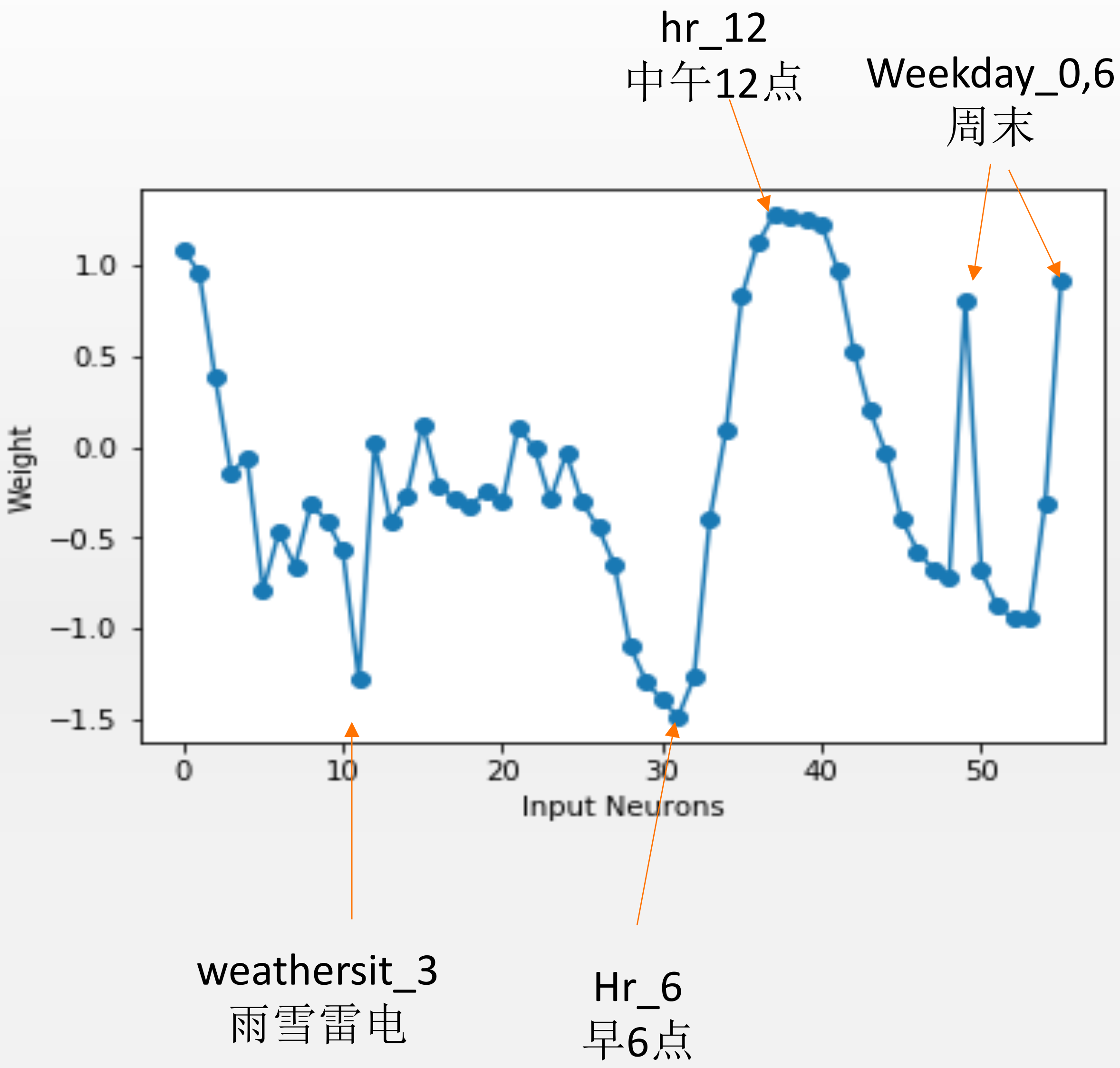


Prediction
Real
Neuro 4
Normalized Values
1.0
0.8
0.6
0.4
0.2
0.0
Dec 22
Dec 23
Dec 24

周末

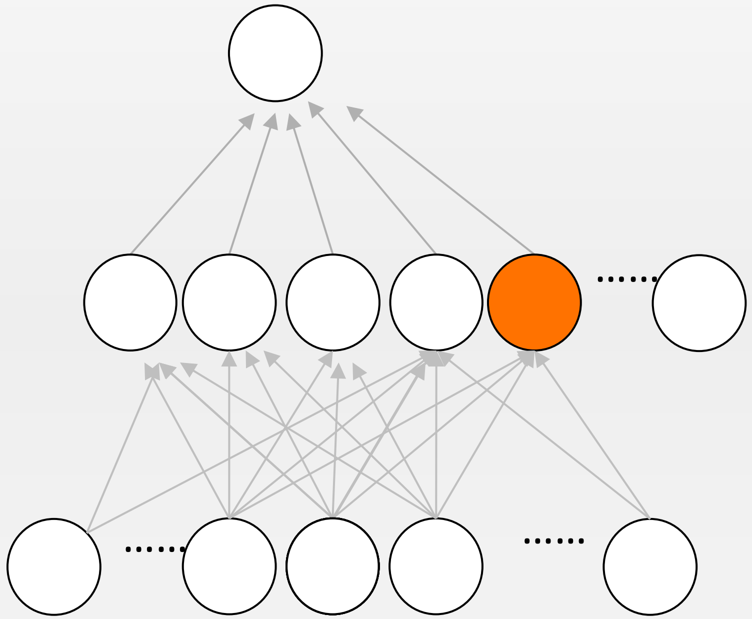
解剖Neu

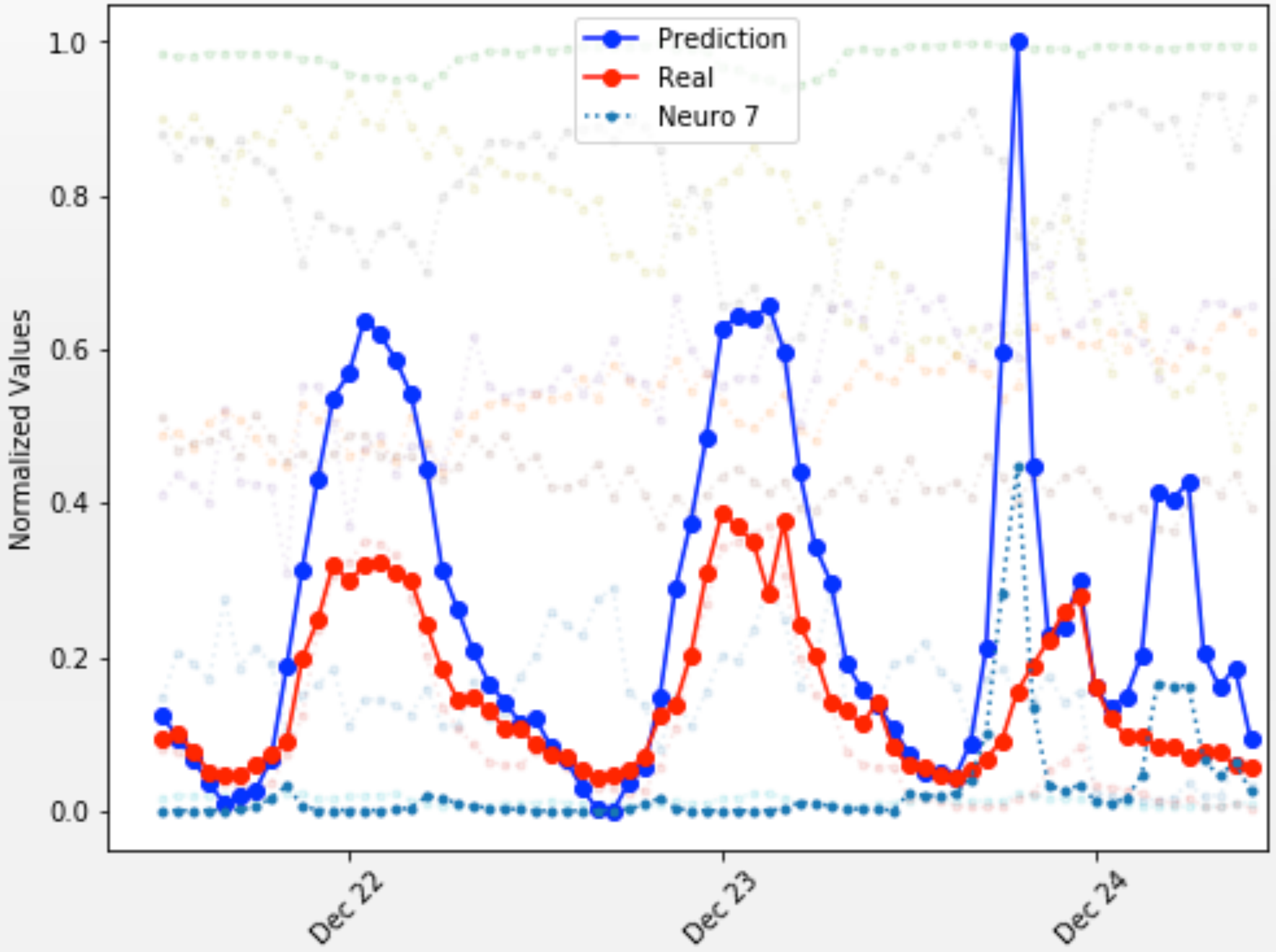




hr_12
中午12点
Weekday_0,6
周末
1.0
0.5
0.0
−0.5
−1.0
−1.5
Weight
0
10
20
30
40
50
Input Neurons
weathersit_3
雨雪雷电
Hr_6
早6点

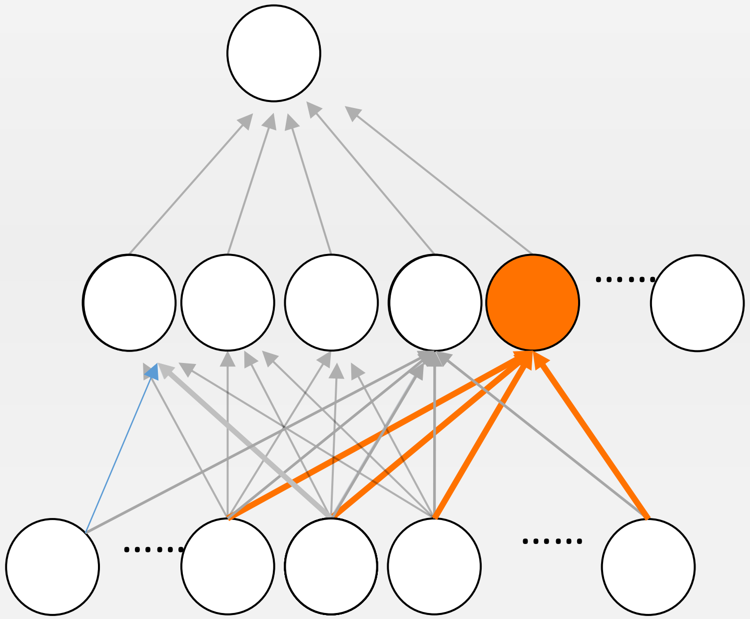
解剖Neu

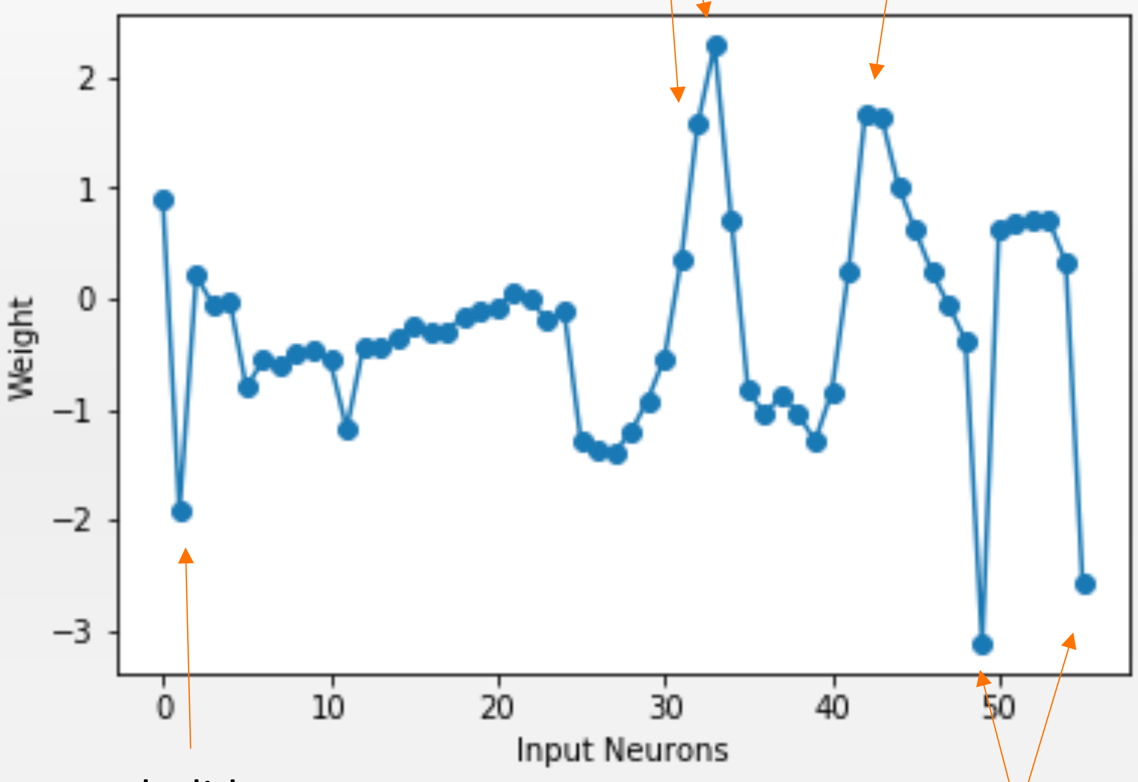




Prediction
Real
Neuro 7
Normalized Values
1.0
0.8
0.6
0.4
0.2
0.0
Dec 22
Dec 23
Dec 24

解剖Neu

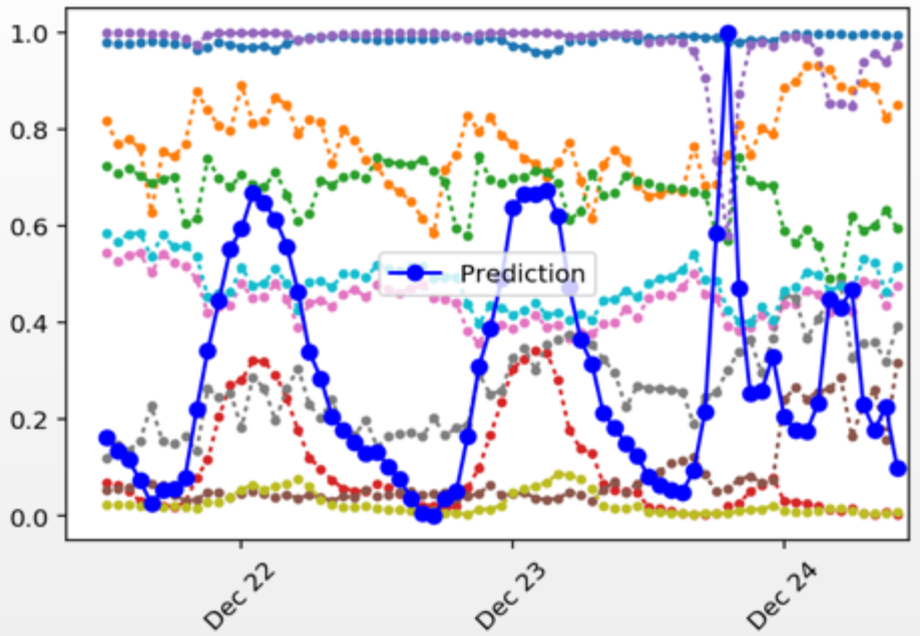


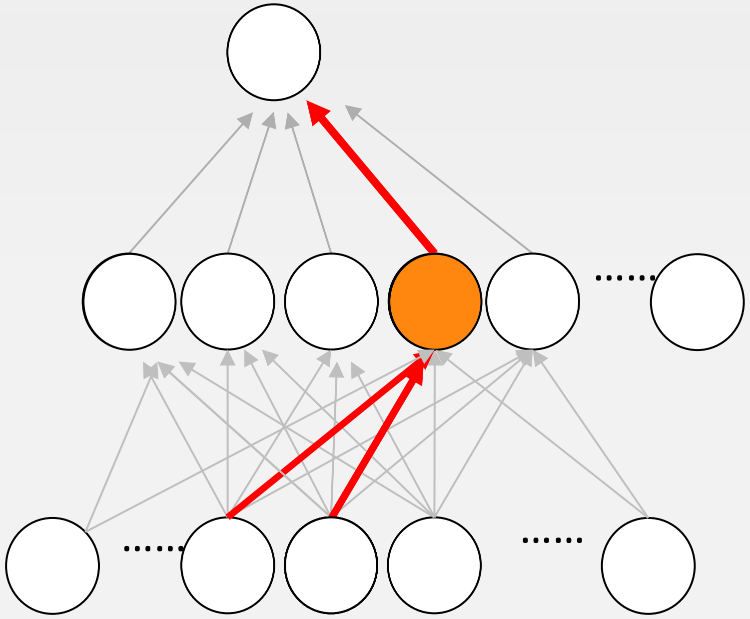


hr_7,8
早高峰
hr_17,18
晚高峰
Weight
Input Neurons
holiday
是否节假日
Weekday6,0
周末

结论

- 预测不准是因为圣诞节假期的反常模式
- 在24号预测值偏高是因为对节假日抑制单元的抑制不够
- 由于圣诞节的缘故，22、23这两天的午高峰出行较少，甚至比一般节假日还少
- 解决：特殊日期的训练需要提供更多数据，或者手工调整权重





今日回顾

- 人工神经网络的工作原理
- 如何用人工神经网络来做预测
- 数据处理方法：类型变量、归一化、分批次训练
- 如何分析一个训练好的人工神经网络
- 运用神经网络进行分类的基本原理

- PyTorch中构建序列化神经网络的方法

练习与作业

- 练习（不需要上交）：
  - 实现一个三分类网络，对自行车预测数据进行高、中、低这三个类别的划分