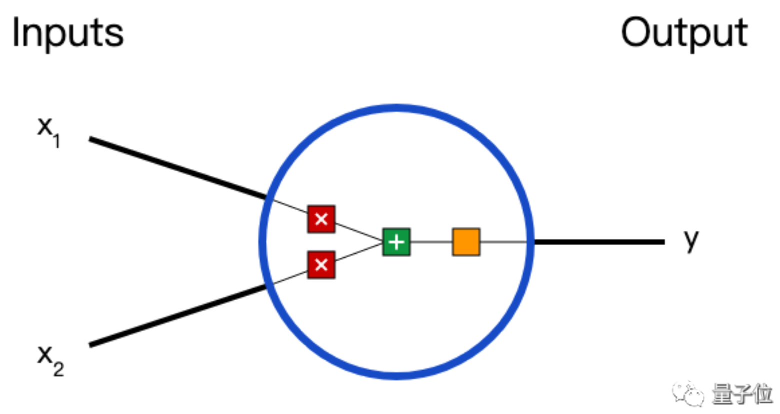
神经网络: 一步步搭建神经网络

> [如何自己从零实现一个*神经网络?*]()

（1）基本模块——神经元

一个2输入神经元的例子：



在这个神经元中，输入总共经历了3步数学运算，

先将两个输入乘以**权重**（weight）：

x1→x1 × w1
x2→x2 × w2

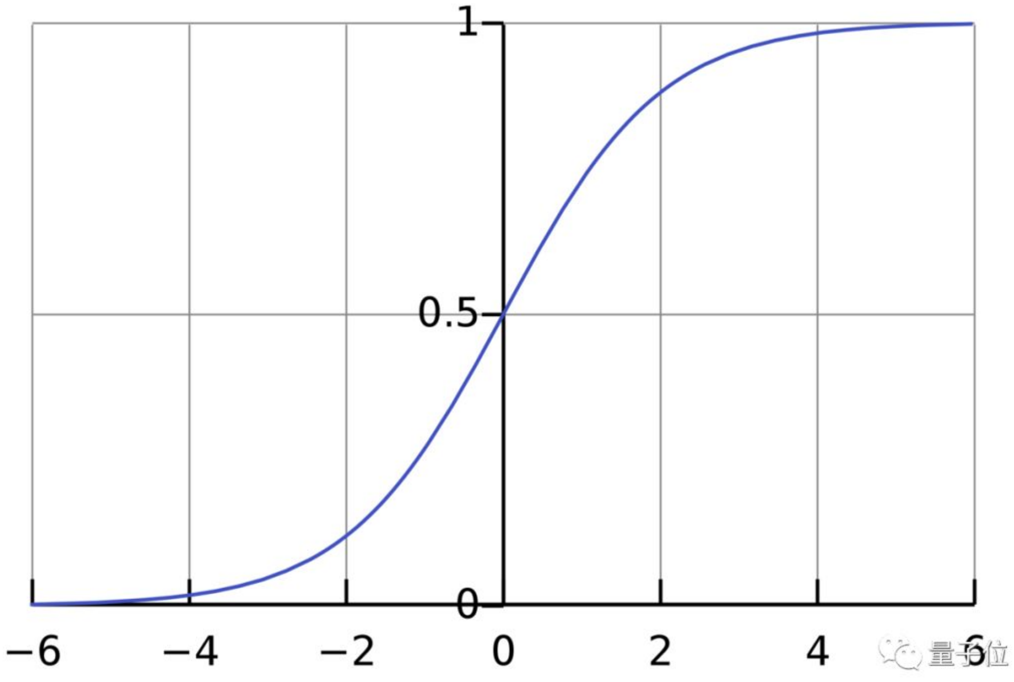
把两个结果想加，再加上一个**偏置**（bias）：

（x1 × w1）+（x2 × w2）+ b

最后将它们经过**激活函数**（activation function）处理得到输出：

y = f(x1 × w1 + x2 × w2 + b)

激活函数的作用是将无限制的输入转换为可预测形式的输出。一种常用的激活函数是sigmoid函数：



sigmoid函数的输出介于0和1，我们可以理解为它把 (−∞,+∞) 范围内的数压缩到 (0, 1)以内。正值越大输出越接近1，负向数值越大输出越接近0。

例子，上面神经元里的权重和偏置取如下数值：

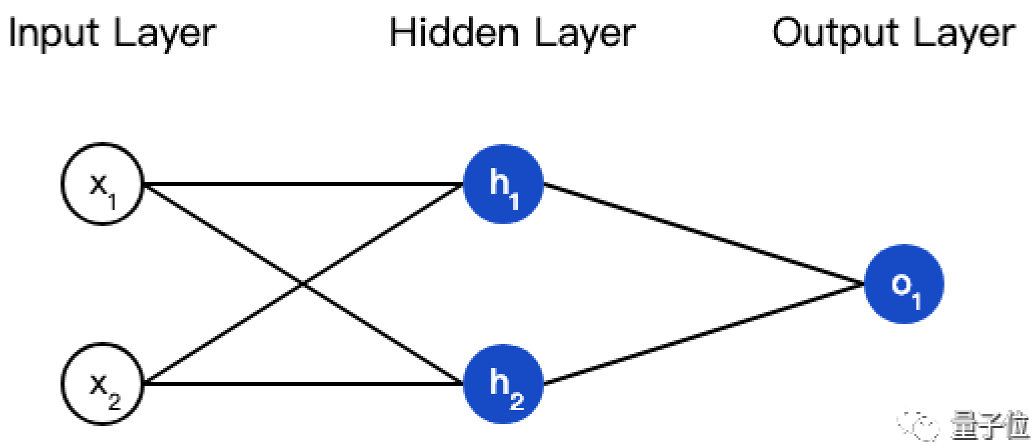
> w=[0,1]
> b = 4
>
> w=[0,1]是w1=0、w2=1的向量形式写法。给神经元一个输入x=[2,3]，可以用向量点积的形式把神经元的输出计算出来：
>
> w·x+b =（x1 × w1）+（x2 × w2）+ b = 0×2+1×3+4=7
> y=f(w·X+b)=f(7)=0.999

```
import numpy as np

def sigmoid(x):
  # Our activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

class Neuron:
  def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias

  def feedforward(self, inputs):
    # Weight inputs, add bias, then use the activation function
    total = np.dot(self.weights, inputs) + self.bias
    return sigmoid(total)

weights = np.array([0, 1]) # w1 = 0, w2 = 1
bias = 4                   # b = 4
n = Neuron(weights, bias)
```

```
x = np.array([2, 3])       # x1 = 2, x2 = 3
print(n.feedforward(x))    # 0.9990889488055994
```

（2）搭建神经网络

神经网络就是把一堆神经元连接在一起，下面是一个神经网络的简单举例：



这个网络有**2个输入**、一个包含2个神经元的隐藏层（h1和h2）、包含1个神经元的输出层o1。

隐藏层是夹在输入输入层和输出层之间的部分，一个神经网络可以有多个隐藏层。

把神经元的输入向前传递获得输出的过程称为**前馈**（feedforward）。

> 我们假设上面的网络里所有神经元都具有相同的权重w=[0,1]和偏置b=0，激活函数都是sigmoid
>
> h1=h2=f(w·x+b)=f((0×2)+(1×3)+0)
> =f(3)
> =0.9526
>
> o1=f(w·[h1,h2]+b)=f((0∗h1)+(1∗h2)+0)
> =f(0.9526)
> =0.7216

5. 训练神经网络

(1) 损失函数

现在我们已经学会了如何搭建神经网络，现在我们来学习如何训练它，其实这就是一个优化的过程。

假设有一个数据集，包含4个人的身高、体重和性别：

Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	[illegible]

现在我们的目标是训练一个网络，根据体重和身高来推测某人的性别。

为了简便起见，我们将每个人的身高、体重减去一个固定数值**（预处理）**，把性别男定义为0、性别女定义为1**（label）**。

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1

在训练神经网络之前，我们需要有一个标准定义它到底好不好，以便我们进行改进，这就是**损失**（loss）。

比如用**均方误差**（MSE）来定义损失：

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

> n是样本的数量，在上面的数据集中是4；
> y代表人的性别，男性是0，女性是1；
> ytrue是变量的真实值，ypred是变量的预测值。

顾名思义，均方误差就是所有数据方差的平均值，我们不妨就把它定义为损失函数。预测结果越好，损失就越低，**训练神经网络就是将损失最小化。**

如果上面网络的输出一直是0，也就是预测所有人都是男性，那么损失是：

Name	y_{true}	y_{pred}	$(y_{true} - y_{pred})^2$
Alice	1	0	1
Bob	0	0	0
Charlie	0	0	0
Diana	1	0	1

MSE= 1/4 (1+0+0+1)= 0.5

计算损失函数的代码如下：

```
import numpy as np

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

y_true = np.array([1, 0, 0, 1])
y_pred = np.array([0, 0, 0, 0])

print(mse_loss(y_true, y_pred)) # 0.5
```

###

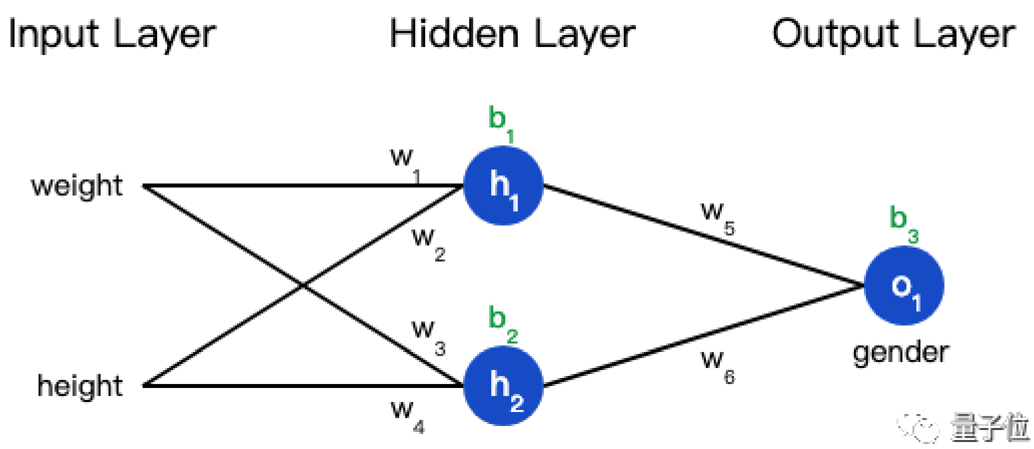
（2）减少损失函数 优化 反向传播

这个神经网络不够好，还要不断优化，尽量减少损失。我们知道，改变网络的权重和偏置可以影响预测值。

为了简单起见，我们把数据集缩减到只包含Alice一个人的数据。于是损失函数就剩下Alice一个人的方差：

$$\begin{aligned}\text{MSE} &= \frac{1}{1}\sum_{i=1}^{1}(y_{true} - y_{pred})^2 \\ &= (y_{true} - y_{pred})^2 \\ &= (1 - y_{pred})^2\end{aligned}$$

预测值是由一系列网络权重和偏置计算出来的：



所以**损失函数实际上是包含多个权重、偏置的多元函数**：

$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$

如果调整一下w1，损失函数是会变大还是变小？我们需要知道偏导数∂L/∂w1是正是负

根据链式求导法则：

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1}$$

而L=(1-ypred)^2，可以求得第一项偏导数：

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial (1 - y_{pred})^2}{\partial y_{pred}} = \boxed{-2(1 - y_{pred})}$$

接下来我们要想办法获得ypred和w1的关系，我们已经知道神经元h1、h2和o1的数学运算规则：

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

实际上只有神经元h1中包含权重w1，所以我们再次运用链式求导法则：

$$\frac{\partial y_{pred}}{\partial w_1} = \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{pred}}{\partial h_1} = \boxed{w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)}$$

然后**求∂h1/∂w1**

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = \boxed{x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)}$$

我们在上面的计算中遇到了2次**激活函数sigmoid的导数f'(x)**，sigmoid函数的导数很容易求得：

$$f(x) = \frac{1}{1+e^{-x}}$$

$$f'(x) = \frac{e^x}{(1+e^{-x})^2} = f(x) * (1 - f(x))$$

总的链式求导公式：

$$\boxed{\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}}$$

这种向后计算偏导数的系统称为**反向传播**（backpropagation）。

h1、h2和o1

h1=f(x1·w1+x2·w2+b1)=0.0474

h2=f(w3·x3+w4·x4+b2)=0.0474

o1=f(w5·h1+w6·h2+b3)=f(0.0474+0.0474+0)=f(0.0948)=0.524

神经网络的输出y=0.524，没有显示出强烈的是男（0）是女（1）的证据。现在的预测效果还很不好。

我们再计算一下当前网络的偏导数∂L/∂w1：

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\begin{aligned}\frac{\partial L}{\partial y_{pred}} &= -2(1 - y_{pred}) \\ &= -2(1 - 0.524) \\ &= -0.952\end{aligned}$$

$$\begin{aligned}\frac{\partial y_{pred}}{\partial h_1} &= w_5 * f'(w_5 h_1 + w_6 h_2 + b_3) \\ &= 1 * f'(0.0474 + 0.0474 + 0) \\ &= f(0.0948) * (1 - f(0.0948)) \\ &= 0.249\end{aligned}$$

$$\begin{aligned}\frac{\partial h_1}{\partial w_1} &= x_1 * f'(w_1 x_1 + w_2 x_2 + b_1) \\ &= -2 * f'(-2 + -1 + 0) \\ &= -2 * f(-3) * (1 - f(-3)) \\ &= -0.0904\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial w_1} &= -0.952 * 0.249 * -0.0904 \\ &= \boxed{0.0214}\end{aligned}$$

这个结果告诉我们：如果增大w1，损失函数L会有一个非常小的增长。

- **随机梯度下降**

下面将使用一种称为**随机梯度下降**（SGD）的优化算法，来训练网络。

经过前面的运算，我们已经有了训练神经网络所有数据。但是该如何操作？SGD定义了改变权重和偏置的方法：

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

η是一个常数，称为**学习率**（learning rate），它决定了我们训练网络速率的快慢。将w1减去η·∂L/∂w1，就等到了新的权重w1。

当∂L/∂w1是正数时，w1会变小；当∂L/∂w1是负数 时，w1会变大。

如果我们用这种方法去逐步改变网络的权重w和偏置b，损失函数会缓慢地降低，从而改进我们的神经网络。

训练流程如下：

1、从数据集中选择一个样本；
2、计算损失函数对所有权重和偏置的偏导数；
3、使用更新公式更新每个权重和偏置；
4、回到第1步。

我们用Python代码实现这个过程：

```python
import numpy as np

def sigmoid(x):
  # Sigmoid activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

def deriv_sigmoid(x):
  # Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
  fx = sigmoid(x)
  return fx * (1 - fx)

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

class OurNeuralNetwork:
  '''
  A neural network with:
    - 2 inputs
    - a hidden layer with 2 neurons (h1, h2)
    - an output layer with 1 neuron (o1)

  *** DISCLAIMER ***:
  The code below is intended to be simple and educational, NOT optimal.
  Real neural net code looks nothing like this. DO NOT use this code.
  Instead, read/run it to understand how this specific network works.
  '''
  def __init__(self):
    # Weights
    self.w1 = np.random.normal()
    self.w2 = np.random.normal()
    self.w3 = np.random.normal()
    self.w4 = np.random.normal()
    self.w5 = np.random.normal()
    self.w6 = np.random.normal()

    # Biases
    self.b1 = np.random.normal()
    self.b2 = np.random.normal()
    self.b3 = np.random.normal()

  def feedforward(self, x):
    # x is a numpy array with 2 elements.
    h1 = sigmoid(self.w1 * x[0] + self.w2 * x[1] + self.b1)
```

```
    h2 = sigmoid(self.w3 * x[0] + self.w4 * x[1] + self.b2)
    o1 = sigmoid(self.w5 * h1 + self.w6 * h2 + self.b3)
    return o1

  def train(self, data, all_y_trues):
    '''
    - data is a (n x 2) numpy array, n = # of samples in the dataset.
    - all_y_trues is a numpy array with n elements.
      Elements in all_y_trues correspond to those in data.
    '''
    learn_rate = 0.1
    epochs = 1000 # number of times to loop through the entire dataset

    for epoch in range(epochs):
      for x, y_true in zip(data, all_y_trues):
        # --- Do a feedforward (we'll need these values later)
        sum_h1 = self.w1 * x[0] + self.w2 * x[1] + self.b1
        h1 = sigmoid(sum_h1)

        sum_h2 = self.w3 * x[0] + self.w4 * x[1] + self.b2
        h2 = sigmoid(sum_h2)

        sum_o1 = self.w5 * h1 + self.w6 * h2 + self.b3
        o1 = sigmoid(sum_o1)
        y_pred = o1

        # --- Calculate partial derivatives.
        # --- Naming: d_L_d_w1 represents "partial L / partial w1"
        d_L_d_ypred = -2 * (y_true - y_pred)

        # Neuron o1
        d_ypred_d_w5 = h1 * deriv_sigmoid(sum_o1)
        d_ypred_d_w6 = h2 * deriv_sigmoid(sum_o1)
        d_ypred_d_b3 = deriv_sigmoid(sum_o1)

        d_ypred_d_h1 = self.w5 * deriv_sigmoid(sum_o1)
        d_ypred_d_h2 = self.w6 * deriv_sigmoid(sum_o1)

        # Neuron h1
        d_h1_d_w1 = x[0] * deriv_sigmoid(sum_h1)
        d_h1_d_w2 = x[1] * deriv_sigmoid(sum_h1)
        d_h1_d_b1 = deriv_sigmoid(sum_h1)

        # Neuron h2
        d_h2_d_w3 = x[0] * deriv_sigmoid(sum_h2)
        d_h2_d_w4 = x[1] * deriv_sigmoid(sum_h2)
        d_h2_d_b2 = deriv_sigmoid(sum_h2)

        # --- Update weights and biases
        # Neuron h1
        self.w1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w1
        self.w2 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w2
        self.b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_b1

        # Neuron h2
        self.w3 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w3
        self.w4 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w4
        self.b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_b2
```

```
        # Neuron o1
        self.w5 -= learn_rate * d_L_d_ypred * d_ypred_d_w5
        self.w6 -= learn_rate * d_L_d_ypred * d_ypred_d_w6
        self.b3 -= learn_rate * d_L_d_ypred * d_ypred_d_b3

      # --- Calculate total loss at the end of each epoch
      if epoch % 10 == 0:
        y_preds = np.apply_along_axis(self.feedforward, 1, data)
        loss = mse_loss(all_y_trues, y_preds)
        print("Epoch %d loss: %.3f" % (epoch, loss))

# Define dataset
data = np.array([
  [-2, -1],  # Alice
  [25, 6],   # Bob
  [17, 4],   # Charlie
  [-15, -6], # Diana
])
all_y_trues = np.array([
  1, # Alice
  0, # Bob
  0, # Charlie
  1, # Diana
])

# Train our neural network!
network = OurNeuralNetwork()
network.train(data, all_y_trues)
```

- **使用训练好的模型做预测**

现在我们可以用它来推测出每个人的性别了：

```
# Make some predictions
emily = np.array([-7, -3]) # 128 pounds, 63 inches
frank = np.array([20, 2])  # 155 pounds, 68 inches
print("Emily: %.3f" % network.feedforward(emily)) # 0.951 - F
print("Frank: %.3f" % network.feedforward(frank)) # 0.039 - M
```

6. pytorch 深度学习框架的实现过程

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt


class Net(nn.Module):
```

```python
    def __init__(self, input_dim=2, out_put_dim=1):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(in_features=input_dim, out_features=2)
        self.fc2 = nn.Linear(in_features=2, out_features=out_put_dim)

    def forward(self, x):

        x = F.relu(self.fc1(x))
        outputs = self.fc2(x)

        return outputs

# --- 实例化一个网络 --- #
my_net = Net()
print("Network:", my_net)
# parameters
for name, params in my_net.named_parameters():
    print(name, params)
print('Total parameters:', sum(param.numel() for param in my_net.parameters()))
# --- optimizer --- #
optimizer = optim.Adam(params=my_net.parameters(), lr=0.01)

# --- data --- #
# Define dataset
data = np.array([
  [-2, -1],  # Alice
  [25, 6],   # Bob
  [17, 4],   # Charlie
  [-15, -6], # Diana
])

all_y_trues = np.array([
  1, # Alice
  0, # Bob
  0, # Charlie
  1, # Diana
])

# --- train --- #
loss_list = list()
for i in range(1000):
    # forward
    sample_index = np.random.randint(0, 4)
    x_train = data[sample_index]
    y_label = all_y_trues[sample_index]
    # numpy to tensor
    x_train = torch.tensor(x_train).to(dtype=torch.float32)
    y_label = torch.tensor(y_label).to(dtype=torch.float32).unsqueeze(dim=0)
    y_pred = my_net(x_train)
    # loss
    loss = F.mse_loss(y_pred, y_label)
    # grad zero
    optimizer.zero_grad()
    # back propagation
    loss.backward()
    # update weight
    optimizer.step()
    print('Epoch: %d Loss : %f'%(i, loss.item()))
```

```
    loss_list.append(loss.item())
plt.plot(loss_list)
plt.show()
```

神经网络结构

http://playground.tensorflow.org/

反向传播的概念

http://colah.github.io/posts/2015-08-Backprop/

神经网络理解

http://colah.github.io/

激活函数的解释

https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0