An Introduction to Neural Network*

Laqudee Zhao†

2019-05-24

A simple explanation of how they work and how to implement one from scratch in Python.

*Victor Zhou's Blog

†E-mail:yunlongzhao1996@gmail.com

目录

1 Brief **3**

2 Building Blocks:Neurons **3**

2.1 Knowledge Bcakgroung 3

2.2 Neural and Others 3

3 Combining Neurons into a Neural Network **6**

3.1 An Exanple:Feedforward 7

3.2 Coding a Neural Network:Feedforward 8

4 Training a Network, Part 1 **9**

4.1 Data Preprocessing 9

4.2 Loss 10

4.3 An Example Loss Calculation 11

5 Training a Network, Part 2 **12**

5.1 Backprop 12

5.2 Example:Calculating the Partial Derivative 15

5.3 Traning:Stochastic Gradient Descent 17

5.4 Code:A Complete Neural Network 18

6 Now What? **23**

6.1 We Did 23

6.2 More We Will do 24

7 Feedforward Network and Backprop Algorithm **25**

7.1 Feedforward Network 25

7.2 Back-propagation Algorithm 29

7.3 Train of Network 31

1 Brief

Here's something that might surprise you:neral networks aren't that complicated! The term "neral network" gets used as a buzzword a lot, but in reality they're often much simpler than people imagine.

This post is intended for complete beginners and assumes ZERO prior knowledge of machine learning. We'll understand how neual networks work while implementing one from scratch in Python.

Let's get started!

2 Building Blocks:Neurons

2.1 Knowledge Bcakgroung

Deep Learning 是 Machine Learning 的一个分支，能够让机器自动学习良好的特征，从而免去人工选取过程。

Deep Learning 的基本思想来自神经生物学表明大脑神经系统处理是以分层的方式来处理信息的，即高层的特征是低层特征的组合，从低层到高层的特征表示越来越抽象，越来越能表现语义或者意图。深度学习借鉴这种分层的信息处理方式，其含有多个层，将当前层当作下一层的输入，通过这种方式实现对输入信息进行分级表达和特征提取。

2.2 Neural and Others

First, we have to talk about nerals, the basic unit of neural network. A neuron takes imputs, does some math with them, and produces one output. Here's what a 2-input neuron looks likes:

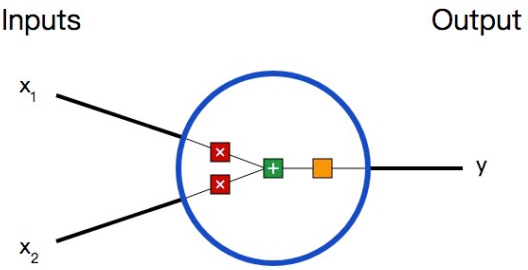

图 1: 神经元及其输出

3 things are happening here.First, each input is multiplied by a weight w:

$$x_1 \longrightarrow x_1 * w_1$$

$$x_2 \longrightarrow x_2 * w_2$$

Next, all the weighted inputs are added together with a bias b:

$$(x_1 * w_2) + (x_2 * w_2) + b$$

Finally, the sum is passed through an activation function:

$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$

The activation function is used to turn an unbounded input into an output that has a nice, predictable form. A commonly used activation function is the sigmoid function and tanh function:

The sigmoid function only outputs numbers in the range(0,1).The tanh function outputs munbers in the range(-1,1).sigmoid function 的导数 $f'(x) = f(x)(1 - f(x))$.

A Simple Example

Assume we have a 2-input neuron that uses the sigmoid activation function and has the following parameters:

$$w = [0, 1] \qquad b = 4$$

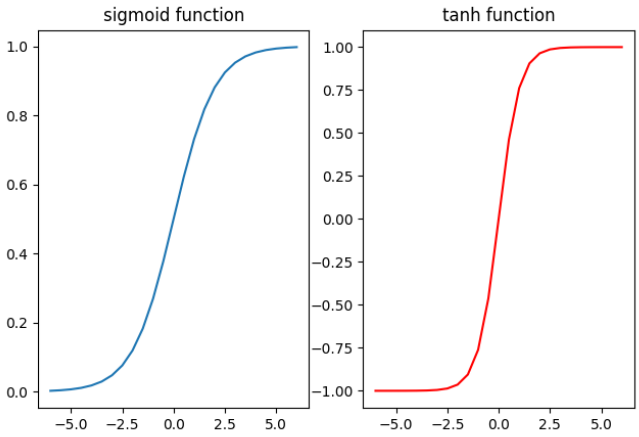

图 2: 激活函数

$w = [0, 1]$is just a way of writing $w_1 = 0, w_2 = 1$in vector form. Now, let's give the neuron an input of $x = [2, 3]$. We'll use the dot product to write things more concisely:

$$\begin{aligned}(w \cdot x) + b &= ((w_1 * x_1) + (w_2 * x_2)) + b \\ &= 0 * 2 + 1 * 3 + 4 \\ &= 7\end{aligned}$$

$$y = f(w \cdot x + b) = f(7) = 0.999$$

The neuron outputs 0.999 given the inputs $x = [2, 3]$. That's it! This process of passing inputs forward to get an output is known as feedforward.

Coding a Neuron

Time to implement a neuron! We'll use Numpy, a popular and powerful computing library for Python, to help us do math:

```
import numpy as np

def sigmoid(x):
```

```
        return 1 / (1 + np.exp(-x))

    class Neuron:
        def __init__(self,weights,bias):
            self.weights = weights
            self.bias = bias
        def feedforward(self, inputs):
            total = np.dot(self.weights,inputs) + self.bias
            return sigmoid(total)
    weights = np.array([0, 1])
    bias = 4
    n = Neuron(weights,bias)

    x = np.array([2,3])
    print(n.feedforward(x)) # 0.99908...
```

Recognize those numbers? That's the example we just did! We get the same answer of 0.999.

3 Combining Neurons into a Neural Network

A neuron network is nothing more than a bunch of neurons connected together.

Here's what a simple neural network might look like:

这个神经网络有 2 个输入，一个隐藏层有两个神经元节点，一个输出，共同组成了神经网络（This network has 2 inputs, a hidden layer with 2 neurons (h_1 and h_2), and an output layer with 1 neuron (o_1). Notice that the inputs for $o_1$are the outputs from h_1 and h_2 - that's what makes this a network.）

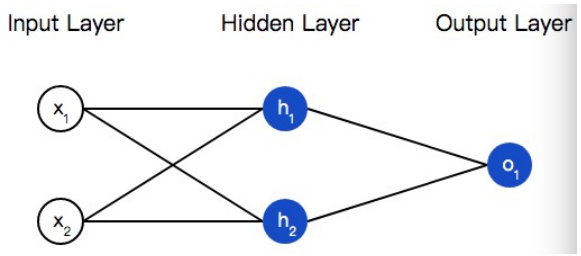

图 3: 简单的神经网络

Hidden Layer: is any layer between the input (first) layer and output(last) layer. There can be multiple hidden layers!

3.1 An Exanple:Feedforward

Let's use the network pictured above and assume all beurons have the same weights $w = [0, 1]$, the same bias $b = 0$, and the same sigmoid activation function. Let h_1, h_2, o_1, denote the outputs pf the neurons they represent.

What happens if we pass in the input $x = [2, 3]$?

$$\begin{aligned} h_1 = h_2 &= f(w \cdot x + b) \\ &= f((0 * 2) + (1 * 3) + 0) \\ &= f(3) \\ &= 0.9526 \end{aligned}$$

$$\begin{aligned} o_1 &= f(w \cdot [h_1, h_2] + b) \\ &= f((0 * h_1) + (1 * h_2) + 0) \\ &= f(0.0526) \\ &= 0.7216 \end{aligned}$$

The output of the neural network for input $x = [2, 3]$ is 0.7216.

Pretty simple,right?

A neural network can have any number of layers with any number of neurons in those layers. The basic idea stays the same: feed the input(s) forward through the neurons in the network to get the output(s) at the end. For simplicity, we' ll keep using the network pictured above for the rest of this post.

3.2 Coding a Neural Network:Feedforward

Let's implement feedforward for our neural network. Here's the image of the network again for refernce:

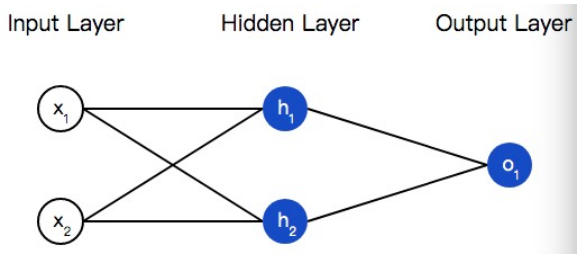

图 4: 简单的神经网络

```
import numpy as np

class OurNeuralNetwork:
    '''
    A neural network with:
        - 2 inputs
        - a hidden layer with 2 neurons (h1, h2)
        - an output layer with 1 neuron (o1)
    Each neuron has the same weights and bias:
```

```
        - w = [0,1]
        - b = 0
    '''
    def __init__(self):
        weights = np.array([0,1])
        bias = 0
        # The Neuon class here is from the previous section
        self.h1 = Neuron(weights, bias)
        self.h2 = Neuron(weights, bias)
        self.o1 = Neuron(weights, bias)

    def feedforward(self,x):
        out_h1 = self.h1.feedforward(x)
        out_h2 = self.h2.feedforward(x)
        out_o1 = self.o1.feedforward(np.array([out_h1, out_h2]))
        return out_o1

network = OurNeuralNetwork()
x = np.array([2,3])
print(network.feedforward(x)) #0.72163
```

We got 0.7216 again! Looks like it works.

4 Training a Network, Part 1

4.1 Data Preprocessing

Say we have the following measurements:

Name	Weight	Height	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F

Let's train our network to predict someone's gender given their weight and height:

We'll represent Male with a 0 and Female with a 1, and we'll also

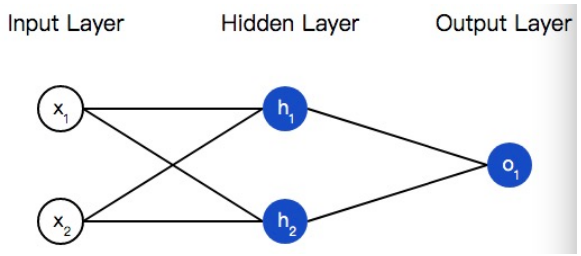

图 5: 训练和预测使用的神经网络

shift the data to make it easier to use:

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

I arbitrarily chose the shift amounts (135 and 66) to make the numbers look nice. Normally, you' d shift by the mean.

4.2 Loss

Before we train our network, we first need a way to quantify how "good" it's doing so that it can try to do "better". That's what the Loss is.

We'll use the mean squared error(MSE, 均方误差)loss:

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

Let's break this down:

- n **is the number of samples, which is 4(Alice, Bob, Charlie, Diana)**
- y **represents the variable being predicted, which is Gender.**
- y_{true} **is the true value of variable(the "correct answer"). For example,** y_{true} **for Alice would be 1(Female).**
- y_{pred} **is the predicted value of the variable. It's whatever our network outputs.**

$(y_{true} - y_{pred})^2$ **is known as the squared error. Our loss function is simply taking the average over all squared errors(hence the name mean squared error). The better our predictions are, the lower our loss will be!**

Better predictions = Lower Loss.
Traning a network = trying to minimize its loss.

4.3 An Example Loss Calculation

Let' s say our network always outputs 0 - in other words, it' s confident all humans are Male. What would our loss be?

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**

$$MSE = \frac{1}{4}(1 + 0 + 0 + 1) = 0.5$$

Code:MSE Loss

Here's some code to calculate loss for us:

```
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
```

Nice. Onwards!

5 Training a Network, Part 2

5.1 Backprop

We now have a clear goal: minimize the loss of the neural network. We know we can change the network's weights(w) and biases(b) to influence its predictions, but how do we do so in way

that decreases loss?

This section uses a bit of multivariable calculus. If you' re not comfortable with calculus, feel free to skip over the math parts. For simplicity, let's pretend we only have Alice in our dataset:

Name	Weight(minus135)	Height(minus66)	Gender
Alice	-2	-1	1

Then the mean squared error loss is just Alice's squared error:

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

Another way to think about loss is as a function of weights and biases. Let's label each weight and bias in our network:

Then, we can write loss as a multivariable function:

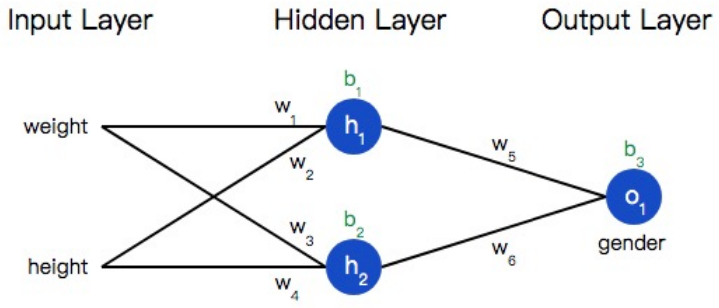

图 6: 信息的正向传播：参数传递

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

Imagine we wanted to tweak w_1. How would loss L change if we changed w_1? That's a question the partial derivation(偏微分)$\frac{\partial L}{\partial w_1}$

can answer. How do we calculate it?
Here' s where the math starts to get more complex. Don' t be discouraged! I recommend getting a pen and paper to follow along - it' ll help you understand.

To start, let's rewrite the partial derivative in terms of $\frac{\partial y_{pred}}{\partial w_1}$ instead:

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

注；偏微分的运算：链式法制

We can calculate $\frac{\partial L}{\partial y_{pred}}$ because we computed $L = (1 - y_{pred]})^2$ above:

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

Now, let's figure out what to do with $\frac{\partial y_{pred}}{\partial w_1}$. Just like before, let h_1, h_2, o_1 be the outputs of the neurons they represent. Then

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

注：f is sigmoid function
Since w_1 only affects h_1 (not h_2), we can write

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

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

We do same thing for $\frac{\partial h_1}{\partial w_1}$:

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

x_1 here is weight, and x_2 is height. This the second time we've seen $f'(x)$(the derivate of the sigmoid function) now! Let' s derive

it:

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

We'll use this nice form for $f'(x)$ later.
We're done! We've managed to break down $\frac{\partial L}{\partial w_1}$ into several parts we can calculate:

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

This system of calculating partial derivatives by working backwards is known as backpropagation, or "backprop".
Phew. That was a lot of symbols - it' s alright if you' re still a bit confused. Let' s do an example to see this in action!

5.2 Example:Calculating the Partial Derivative

We're going to continue pretending only Alice is our dataset:

Name	Weight(minus135)	Height(minus66)	Gender
Alice	-2	-1	1

Let's initialize all the weights to 1 and all the biases to 0. if we do a feedforward pass throught the network, we get:

$$\begin{aligned} h_1 &= f(w_1 x_1 + w_2 x_2 + b_1) \\ &= f(-2 + -1 + 0) \\ &= 0.0474 \end{aligned}$$

$$h_2 = f(w_3 x_1 + w_4 x_2 + b_2) = 0.0474$$

$$\begin{aligned} o_1 &= f(w_5 h_1 + w_6 h_2 + b_3) \\ &= f(0.0474 + 0.0474 + 0) \\ &= 0.524 \end{aligned}$$

The network outputs $y_{pred} = 0.524$, which doesn't strongly favor Male(0) or Female(1).Let's calculate $\frac{\partial L}{\partial w_1}$:

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

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

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

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

$$\frac{\partial L}{\partial w_1} = -0.952 * 0.249 * (-0.0904) = 0.0214$$

Reminder: we derived $f'(x) = f(x) * (1 - f(x))f$for our sigmoid activation function earlier.

We did it! This tells us that if we were to uncrease w_1, L would uncrease a tiny bit as a result.

5.3 Traning:Stochastic Gradient Descent

We have all the tools we need to train a neural network now!
We'll use an optimization algorithm called stochastic gradient descent (SGD, 随机梯度
that tells us how to change our weights and biases to minimize loss.
It's basically just this update equation:

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

其中，η 是 learning rate(学习率)，用于控制训练的速度.
doing is subtracting $\eta \frac{\partial L}{\partial w_1}$ from w_1:

- **If $\frac{\partial L}{\partial w_1}$ is postitive, w_1 will decrease, which makes L decrease.**
- **If $\frac{\partial L}{\partial w_1}$ is negative, w_1 will increase, which makes L decrease.**

If we do this for every weight and bias in the network, the loss will slowly decrease and our network will improve.

Our training process will look like this:

1、Choose one sample from our dataset. This is what makes it stochastic gradient descent - we only operate on one sample at a time.

2、Calculate all the partial derivatives of loss with respect to weights or biases (e.g.$\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}$,etc)

3、Use the update equation to update each weight and bias.

4、Go back to step 1.

Let's see it in action!

5.4 Code:A Complete Neural Network

It's finally time to implement a complete neural network:

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

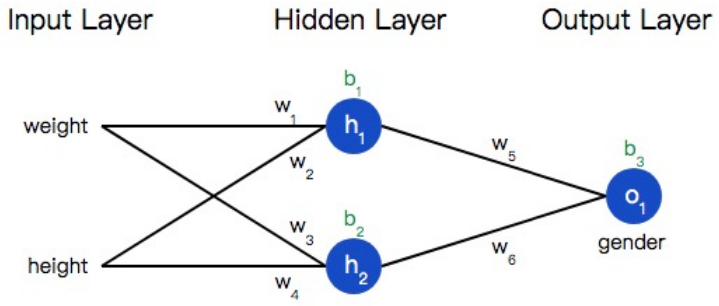

图 7: 神经网络

```
    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)
```

Our loss steadily decreases as the network learns:
We can now use the nework to predict genders: //

```
 # Make some predictions
emily = np.array([-7, -3]) # 128 pounds, 63 inches
```

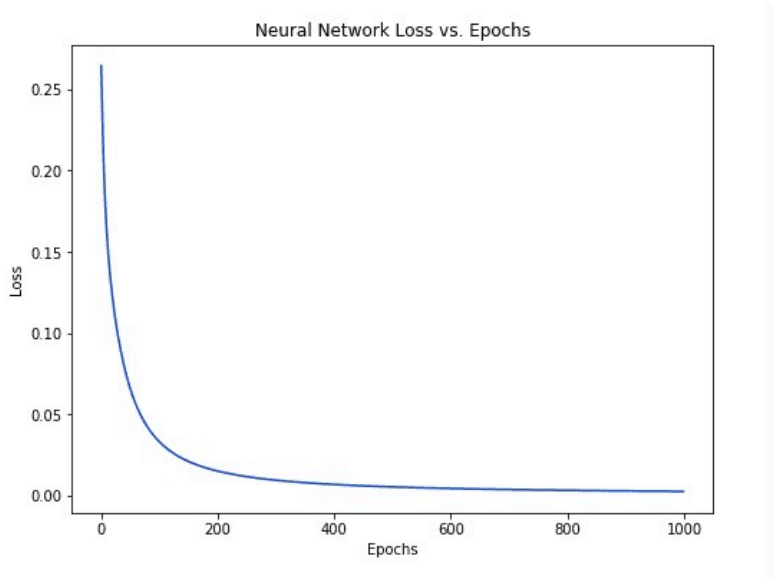

图 8: Epochs

```
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 Now What?

6.1 We Did

- **Introduced neurons, the building blocks of neural networks.**
- **Used the sigmoid activation function in our neurons.**
- **Saw that neural networks are just neurons connected together.**
- **Created a dataset with Weight and Height as inputs (or features) and Gender as the output (or label).**
- **Learned about loss functions and the mean squared error (MSE) loss.**

- **Realized that training a network is just minimizing its loss.**
- **Used backpropagation to calculate partial derivatives.**
- **Used stochastic gradient descent (SGD) to train our network.**

6.2 More We Will do

- **Experiment with bigger/better neural networks using proper machine learning libraries like Tensorflow, Keras, PyTorch.**
- **Tinker with a neural network in your brower.**
- **Discover other activation functions besides sigmoid.**
- **Discover other optimizers besides SGD.**
- **Read CNNsConvolutional Neural Networks)**
- **Leran about Recurrent Neural Networks(回归神经网络), often used for Natural Langage Processing(NLP，自然语言处理).**

7 Feedforward Network and Backprop Algorithm

本章节主要介绍前馈神经网络和反向传播算法，这是贯穿整个神经网络的基调！也是作为前六章的补充，加深对神经网络的认识，为以后学习其他神经网络算法打基础！

7.1 Feedforward Network

前面已经介绍过“神经元”、“激活函数”等概念，这一章节不再叙述。
sigmoid 和 tanh 两个激活函数的关系：$tanh(x) = 2\sigma(2z) - 1$.
所谓神经网络就是将许多个单一神经元结合在一起，一个神经元的输出就可以是另一个神经元的输入。下图就是一个简单的神经网络：

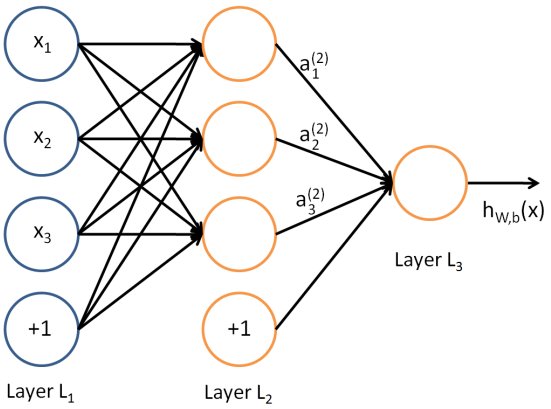

图 9: 一个简单的神经网络

In this figure, we have used circles to also denote the inputs to the network. The circle labeled ”+1” are called bias units, and correspond to the intercept term. The leftmost layer of the network is called input layer, and the rightmost layer the output layer(which, in this example, has only one node). The middle layer of nodes is called the hidden layer, because its values are not observed in the

training set.

We also say that our example neural network has 3 input units (not counting the bias unit), 3 hidden unit, and 1 output unit.

- n_l **: the number of layers in our network.**
- l **: label as** L_l**, so** L_1 **is the input layer,**L_l **is output.**
- (W, b) **: parameters.write** W_{ij}^l **to denote the para associated with the connection between unit** j **in layer** l**,and unit** i **in the layer**l_1**.** b_i^l **is the bias associated with unit in layer** $l+1$**.**
- $W \in R^{m\times n}$**,** $b \in R^{1\times n}$**.**

信息的正向传播 在本例中 $W^{(1)} \in R^{3\times 3}, W^2 \in R^{1\times 3}$.
以 $a_i^{(l)}$ 表示第 i 层单元的激活值（输出值）。本例神经网络的计算步骤如下：

$$
\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b^{(1)_1}) \\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b^{(1)_2}) \\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b^{(1)_3})
\end{aligned}
$$

以 $z_i^{(l)}$ 表示第 l 层第 i 个单元的输入，即 $z_i^{(l)} = \sum_j W_{ij}^{(l-1)}x_j + b^{(l-1)_i}$，则此单元的输出为 $a^{(l)_i} = f(z_i^{(l)})$，其中 f 为激活函数。

可将标量形式的激活函数 $f(\cdot)$ 扩展为向量形式表示，即 $f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$，则神经网络计算可简洁地表示为

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^1 \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

以上的计算过程就是信息的正向传播。给定第 l 层的激活值 $a^{(l)}$ 后，第 l+1 层的激活值 $a^{(l+1)}$ 可按以下步骤进行计算：

$$z^{(l+1)} = W^{(l)}a^{l} + b^{(l)}$$
$$a^{(l+1)} = f(z^{(l+1)})$$

神经网络可以有多个隐藏层和多个输出单元，如下图展示：

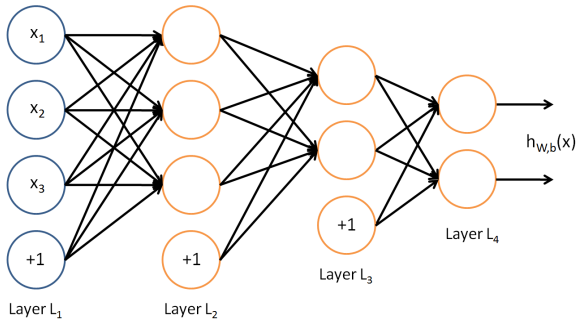

图 10: 含有两个隐藏层的神经网络

注：隐藏层的激活函数不能是线性函数，因为第 l 层的输出 $a^{(l)} = f(z^{(l)})$; 网络的第 l+1 层的净输入为

$$\begin{aligned} z^{(l+1)} &= W^{(l)}a^{l} + b^{(l)} \\ &= W^{(l)}f(z^{(l)}) + b^{(l+1)} \end{aligned}$$

如果激活函数是线性函数，则第 l 层和第 l+1 层将是线性关系。得到的网络最终输出 $a^{(L)}$ 和输入层 $a^{(1)} = x$ 之间也是线性关系，网络将退化成简单的线性分类器。

网络参数学习方法

目标函数

在之前章节中介绍的 **Loss function** 是 **MSE**（均方误差），这里使用的方法是相似的!
为了得到最优的参数矩阵，将使用批量梯度下降法来训练（学习）神经网络参数 W, b。具体来说，对于单个样本 (x, y)，其代价函数为：

$$J(W, b; x, y) = \frac{1}{2}||h_{W,b}(x) - y||_2^2$$

样本集上的整体代价函数（**cost function**）：

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m}\sum_{i=1}^{m} J\left(W, b; x^{(i)}, y^{(i)}\right)\right] + \frac{\lambda}{2}\sum_{l}\left\|W^{(l)}\right\|_2^2 \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}\left(x^{(i)}\right) - y^{(i)}\right\|_2^2\right)\right] + \frac{\lambda}{2}\sum_{l}\left\|W^{(l)}\right\|_2^2 \end{aligned}$$

上式第一项为均方误差项（**MSE**），第二项为一个正则化项，用于减小权重的幅度，防止过拟合。以上的代价函数经常被用于分类和回归问题。

梯度算法

梯度下降法中每次迭代都按照以下公式对参数 W, b 进行更新：

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \lambda\nabla_{W_{ij}^l} J(W, b)$$

$$b_i^{(l)} = b_i^{(i)} - \lambda\nabla_{b_i^{(l)}} J(W, b)$$

其中，λ 为学习效率（**learn rate**），实现上述梯度算法的关键是如何计算偏导数。$\lambda\nabla_{W_{ij}^l} J(W, b)$ 和 $\lambda\nabla_{b_i^{(l)}} J(W, b)$ 是整体代价函数 $J(W, b)$ 的偏函数，其定义为：

$$\begin{aligned} \nabla_{W_{ij}^{(l)}} J(W, b) &= \left[\frac{1}{m}\sum_{i=1}^{m}\nabla_{W_{ij}^{(l)}} J\left(W, b; x^{(i)}, y^{(i)}\right)\right] + \lambda W_{ij}^{(l)} \\ \nabla_{b_i^{(l)}} J(W, b) &= \frac{1}{m}\sum_{i=1}^{m}\nabla_{b_i^{(l)}} J\left(W, b; x^{(i)}, y^{(i)}\right) \end{aligned}$$

其中，$\lambda\nabla_{W_{ij}^l} J(W, b; x, y)$ 和 $\lambda\nabla_{b_i^{(l)}} J(W, b; x, y)$ 是单个样本 (x, y) 的代价函数 $J(W, b; x, y)$ 的偏导数，它们可通过反向传播算法 求出。

7.2 Back-propagation Algorithm

基本关系和量

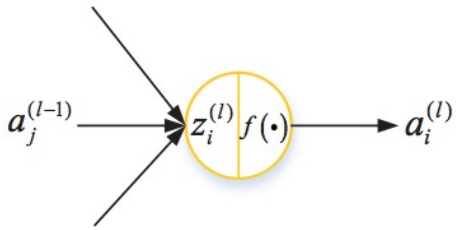

图 11: 第 l 层的第 i 个神经元

第 l 层的第 i 个神经元的净输入

$$z_i^{(l)} = \sum_j W_{ij}^{(l-1)} a_j^{(l-1)} + b^{(l-1)}$$

第 l 层的第 i 个神经元的输出（激活值）

$$a_i^{(l)} = f(z_i^{(l)})$$

第 l 层的第 i 个神经元的净输入 $z_i^{(l)}$ 相对于代价函数 $J(W,b;x,y)$ 的变化率，称为残差

$$\delta_i^{(l)} = \frac{\partial J(W,b;x,y)}{\partial z_i^{(l)}}$$

第 l-1 层网络参数 $W_{ij}^{(l-1)}$ 相对于代价函数 $J(W,b;x,y)$ 的变化率

$$\begin{aligned}
\nabla_{W_{ij}^{(l-1)}} J(W,b;x,y) &= \frac{\partial J(W,b;x,y)}{\partial W_{ij}^{(l-1)}} = \frac{\partial J(W,b;x,y)}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial W_{ij}^{(l-1)}} \\
&= \delta_i^{(l)} \frac{\partial \left(\sum_j W_{ij}^{(l-1)} a_j^{(l-1)} + b^{(l-1)} \right)}{\partial W_{ij}^{(l-1)}} \\
&= \delta_i^{(l)} a_j^{(l-1)}
\end{aligned}$$

即

$$\nabla_{W_{ij}^{(l-1)}} J(W,b;x,y) = \delta_i^{(l)} a_j^{(l-1)}$$

注：上述推导中使用了链式法则。

如此可见如果求得了残差 $\lambda_i^{(l)}$ 即可容易求得上述梯度。关于各层参数的偏导数

$$\nabla_{W_{ij}^{(l)}} J(W,b;x,y) = a_j^{(l)} \delta_i^{(l+1)}$$
$$\nabla_{b_i^{(l)}} J(W,b;x,y) = \delta_i^{(l+1)}$$

反向传播算法的基本思想

首先进行“前向传播”运算，计算出网络中所有神经元的激活值，包括 $h_{W,b}(x)$ 的输出值。

然后，针对第 l 层的每一个神经元（节点）i，计算其残差 $\lambda_i^{(l)}$，次残差反映了此节点对最终输出值的误差产生了多少影响。对于最终的输出节点，可以直接算出网络产生的输出值于实际值之间的差距，将这个差距定义为 $\lambda_i^{(n_l)}$（第 n_l 层表示输出层）。第 l 层节点的残差 $\lambda_i^{(l)}$ 可依据第 l+1 层的残差 $\lambda_i^{(l+1)}$ 来计算。

计算残差

假设依据前向传播算法得到每个层的所有神经元的输出 $a_i^{(i)}, \ldots, a_i^{(l)}, \ldots, (n_l)_i$，其中 i 表示每个层的第 i 个节点（神经元）。

1) 计算输出层（第 n_l 层）的残差 $\lambda_i^{(n_l)}$

$$\begin{aligned}
\delta_i^{(n_l)} &= \frac{\partial}{\partial z_i^{(n_l)}} J(W,b;x,y) \\
&= \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|_2^2 \\
&= \frac{1}{2} \frac{\partial}{\partial z_i^{(n_l)}} \sum_j \left(y_j - a_j^{(n_l)} \right)^2 \\
&= \frac{1}{2} \frac{\partial}{\partial z_i^{(n_l)}} \sum_j \left(y_j - f\left(z_j^{(n_l)} \right) \right)^2 \\
&= -\left(y_i - a_i^{(n_l)} \right) f'\left(z_i^{(n_l)} \right) \\
&= -\left(y_i - a_i^{(n_l)} \right) f'\left(z_i^{(n_l)} \right)
\end{aligned}$$

可简记为

$$\delta_i^{(n_l)} = -(y_i - a_i^{n_l}) f'(z_i^{n_l})$$

计算隐藏层的残差

$$\delta_i^{(l)} = \left(\sum_{j=1}^{N_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z^{(l)_i})$$

其中 N_{l+1} 为第 l+1 层的节点数。上式中 $(\cdot)$ 部分的含意是第 l+1 层中所有与第 l 层中第 i 个节点相连接的节点的残差的加权和。其中权值就是连接的网络加权参数，如下图 12 所示，这就是反向传播。

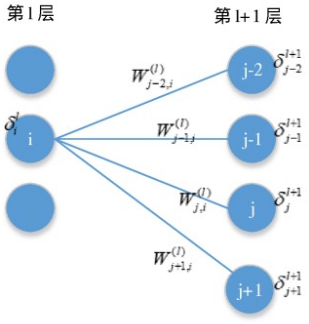

图 12: 残差的反向传播

7.3 Train of Network

前馈神经网的训练包含如下过程：

1）依据当前参数计算前向传播过程中每一层上的输出值；

2）依据反向传播算法计算每一层参数的梯度，并实现参数更新；

3）重复上述两个过程，直到收敛。

注：注意梯度扩散（gradient diffusion）现象。

appendix

- **Thanks for Victor Zhou, This notebook faked from him.**
- **Subscribe Victor Zhou at https://victorzhou.com/nlog.**
- **我会结合自己的学习方式来更新完善 Neural Networks 的学习!**
- **Subscribe me: https://github.com/ZhaoYLong/.**
- **You can follow me at Twitter, My name @laqudee1.**