

程式人《十分鐘系列》



專為程式人寫的神經網路導論

(以反傳遞演算法為入門磚)

陳鍾誠

2017 年 7 月 4 日

最近我在學《深度學習技術》

主要是為了

- 要做機器翻譯的程式！

因為最近

- Google 和百度的機器翻譯
- 進展實在令人感到驚艷！

我原本以為

- 在我有生之年
- 機器翻譯技術應該無法成熟實用！

但是最近

- 深度學習技術的發展
- 讓我改變了觀點，或許機器翻譯有可能在幾年內成熟
- 高品質的機器翻譯是有可能達成的！

而那個背後的深度學習技術

- 其實說穿了，就是新一代的
《類神經網路》罷了！

所以要學深度學習

- 還是得從《神經網路》開始

而最經典的神經網路算法

- 莫過於《反傳遞演算法》了
(back-propagation algorithm)

在研究深度學習技術時

我看到了一篇文章

- 標題是：Hacker's guide to Neural Networks
 - 這是 ConvNetJS 的作者
Andrej Karpathy 寫的
 - 用 JavaScript 程式碼當範例

我發現寫得很好

不過是英文的

而且字有點多

為了讓大家比較輕鬆的

- 透過這篇文章理解神經網路

所以我把該文章改寫

- 變成這篇十分鐘系列！

希望會比較容易讀

- 也比較容易懂！

對於程式人而言

- 數學是有點恐怖的！

但是如果寫成程式

- 通常會變得親切一點！

Karpathy 說

My personal experience with Neural Networks is that everything became much clearer when I started ignoring full-page, dense derivations of backpropagation equations and just started writing code. Thus, this tutorial will contain **very little math** (I don't believe it is necessary and it can sometimes even obfuscate simple concepts). Since my background is in Computer Science and Physics, I will instead develop the topic from what I refer to as **hackers's perspective**. My exposition will center around code and physical intuitions instead of mathematical derivations. Basically, I will strive to present the algorithms in a way that I wish I had come across when I was starting out.

“...everything became much clearer when I started writing code.”

You might be eager to jump right in and learn about Neural Networks, backpropagation, how they can be applied to datasets in practice, etc. But before we get there, I'd like us to first forget about all that. Let's take a step back and understand what is really going on at the core. Lets first talk about real-valued circuits.

意思是這樣的

- 我們可以透過程式，很容易的理解那些數學，而不需要被《偏微分》等符號給嚇壞！

現在

- 就讓我們開始用程式理解數學吧！

對於學過數位邏輯的人而言

- 應該知道電腦可以由《邏輯閘》所建構出來！
- 這些邏輯閘的輸入不是 0 就是 1

例如

- $\text{and}(x, y)$, $\text{or}(x, y)$, $\text{not}(x)$, $\text{xor}(x, y)$,
...
- 其中的 x, y 只能是 0 或 1

但是對於神經網路而言

- 通常輸入為《實數》

- 像是這樣

- $f(x, y) = x * y$

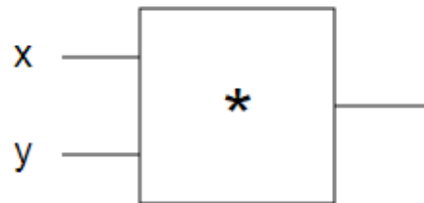
- $g(x, y, z) = (x + y) * z$

如果用電路的方式表示

- 就會是一種《實數值》電路

Base Case: Single Gate in the Circuit

Lets first consider a single, simple circuit with one gate. Here's an example:



The circuit takes two real-valued inputs `x` and `y` and computes `x * y` with the `*` gate. Javascript version of this would very simply look something like this:

```
var forwardMultiplyGate = function(x, y) {  
  return x * y;  
};  
forwardMultiplyGate(-2, 3); // returns -6. Exciting.
```

And in math form we can think of this gate as implementing the real-valued function:

$$f(x, y) = xy$$

As with this example, all of our gates will take one or two inputs and produce a **single** output value.

神經網路的問題

- 通常是要讓某個函數《最大化》或《最小化》
- 也就是一種《優化問題》（最佳化問題）

假如現在

- 我們想像自己在 $x=-2$, $y=3$ 這個點
- 那要怎麼做，才能讓 $f(x, y)=xy$ 的值，變得更大呢？

一個簡單的方法是

- 隨機調整一個變數試試看

The Goal

The problem we are interested in studying looks as follows:

1. We provide a given circuit some specific input values (e.g. $x = -2$, $y = 3$)
2. The circuit computes an output value (e.g. -6)
3. The core question then becomes: *How should one tweak the input slightly to increase the output?*

In this case, in what direction should we change x, y to get a number larger than -6 ? Note that, for example, $x = -1.99$ and $y = 2.99$ gives $x * y = -5.95$, which is higher than -6.0 . Don't get confused by this: -5.95 is better (higher) than -6.0 . It's an improvement of 0.05 , even though the *magnitude* of -5.95 (the distance from zero) happens to be lower.

如果調整後更好了

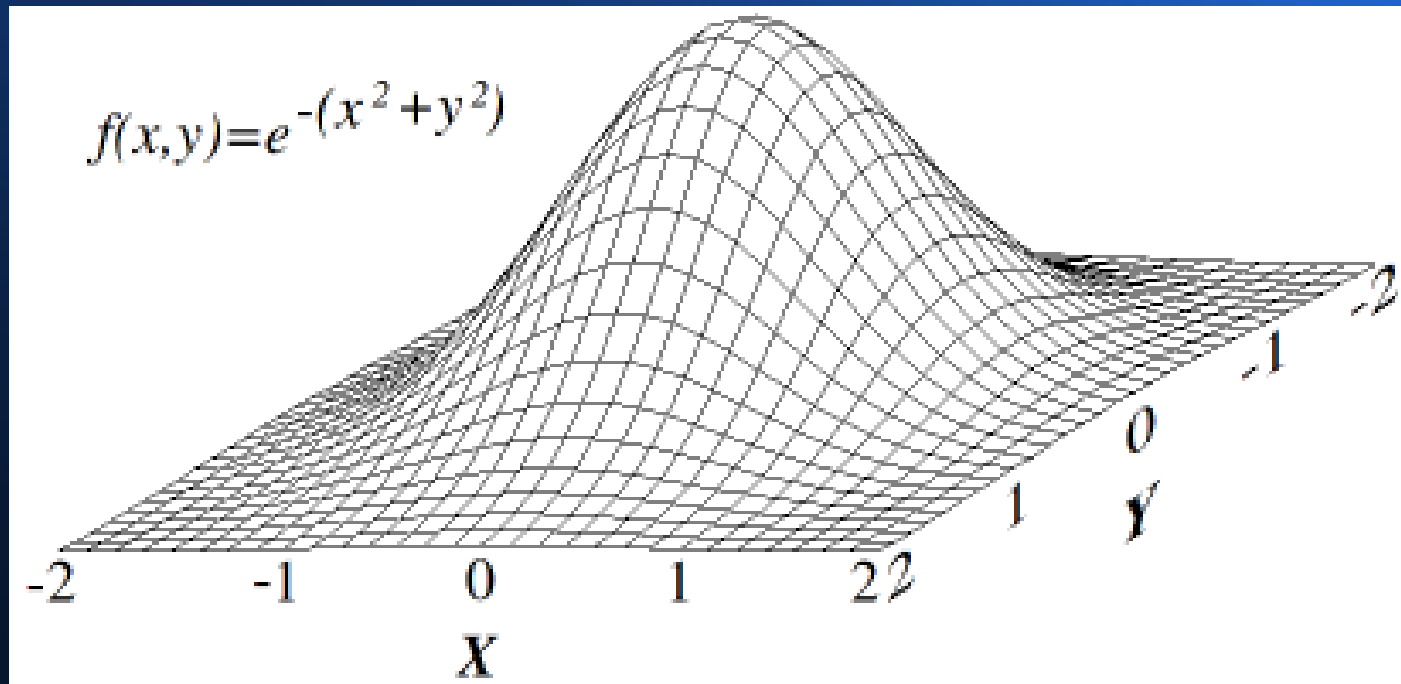
- 那就接受那個更好的值

```
// circuit with single gate for now
var forwardMultiplyGate = function(x, y) { return x * y; };
var x = -2, y = 3; // some input values

// try changing x,y randomly small amounts and keep track of what works best
var tweak_amount = 0.01;
var best_out = -Infinity;
var best_x = x, best_y = y;
for(var k = 0; k < 100; k++) {
  var x_try = x + tweak_amount * (Math.random() * 2 - 1); // tweak x a bit
  var y_try = y + tweak_amount * (Math.random() * 2 - 1); // tweak y a bit
  var out = forwardMultiplyGate(x_try, y_try);
  if(out > best_out) {
    // best improvement yet! Keep track of the x and y
    best_out = out;
    best_x = x_try, best_y = y_try;
  }
}
```

這個方法

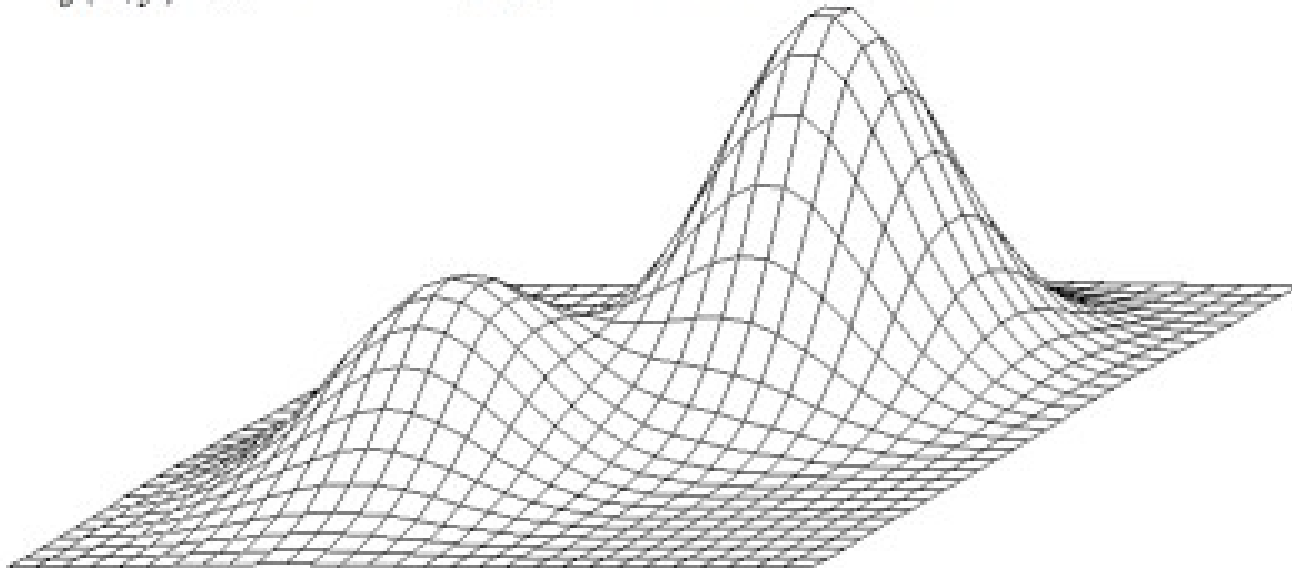
- 其實就是《爬山演算法》（隨機版）



當然

- 有時候會有好幾個山峰，爬山演算法會停在其中一個，但不見得是最高的那個

$$f(x,y)=e^{-(x^2+y^2)} + 2e^{-((x-1.7)^2+(y-1.7)^2)}$$



不過這種爬山算法

- 需要進行某些《嘗試》
- 每次《嘗試》都得重新計算函數值
- 如果函數很複雜，嘗試會花很多時間

還好

- 微積分裡的《導數》，可以給我們一些方向指引！

$$\frac{\partial f(x, y)}{\partial x} = \frac{f(x + h, y) - f(x, y)}{h}$$

我們可以透過計算導數

$$\frac{\partial f(x,y)}{\partial x} = \frac{f(x+h,y) - f(x,y)}{h}$$

- 瞭解到底該往哪個方向調整，才會讓函數值變好

```
var x = -2, y = 3;
var out = forwardMultiplyGate(x, y); // -6
var h = 0.0001;

// compute derivative with respect to x
var xph = x + h; // -1.9999
var out2 = forwardMultiplyGate(xph, y); // -5.9997
var x_derivative = (out2 - out) / h; // 3.0

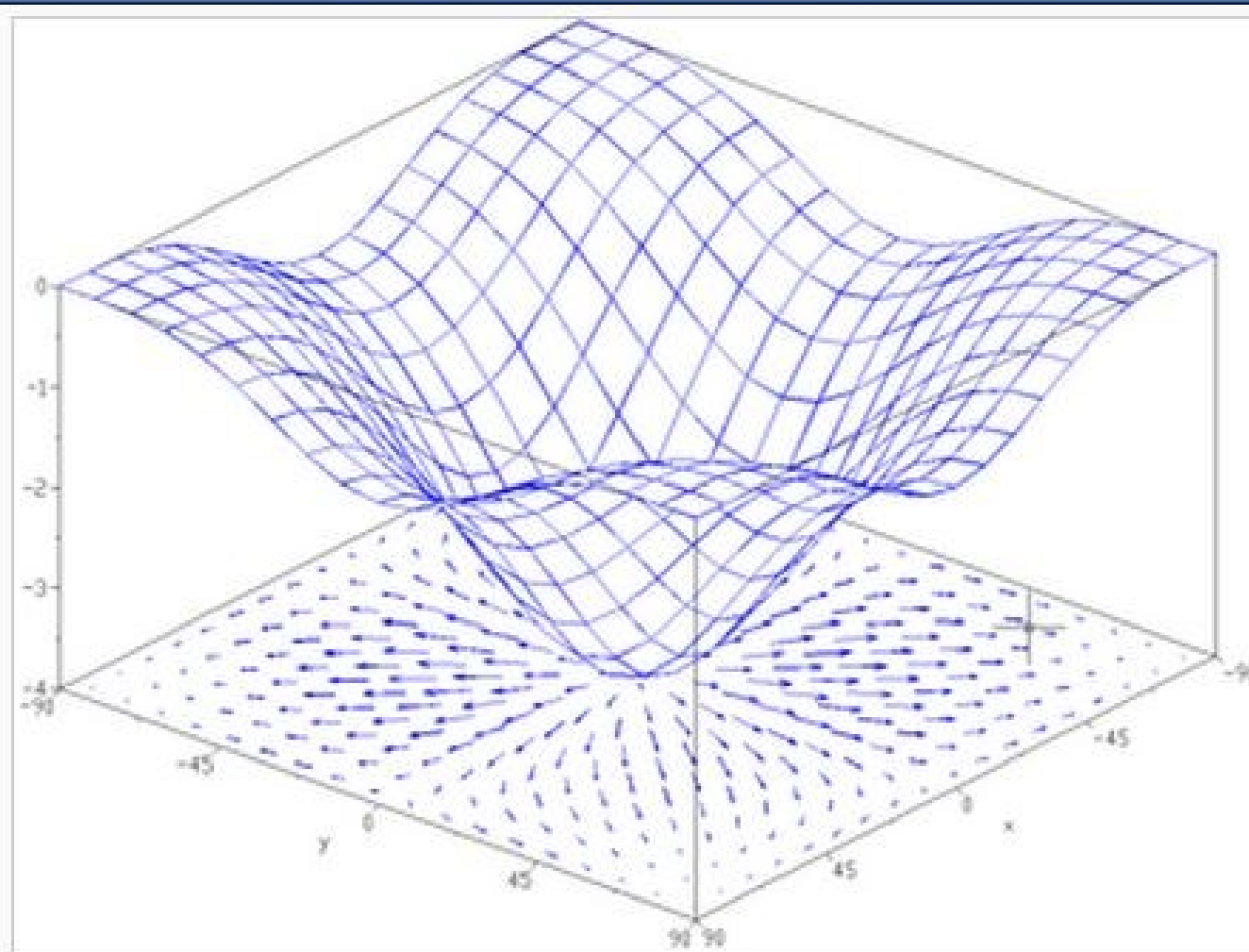
// compute derivative with respect to y
var yph = y + h; // 3.0001
var out3 = forwardMultiplyGate(x, yph); // -6.0002
var y_derivative = (out3 - out) / h; // -2.0
```

只要朝著梯度的方向調整

- 就能朝變化最快的方向前進

```
var step_size = 0.01;  
var out = forwardMultiplyGate(x, y); // before: -6  
x = x + step_size * x_derivative; // x becomes -1.97  
y = y + step_size * y_derivative; // y becomes 2.98  
var out_new = forwardMultiplyGate(x, y); // -5.87! exciting.
```

這就是所謂的《梯度下降法》



The gradient of the function $f(x, y) = -(\cos^2 x + \cos^2 y)^2$ depicted as a projected vector field on the bottom plane.



爬山是往上爬

- 梯度卻是往下走

到底要走哪個方向

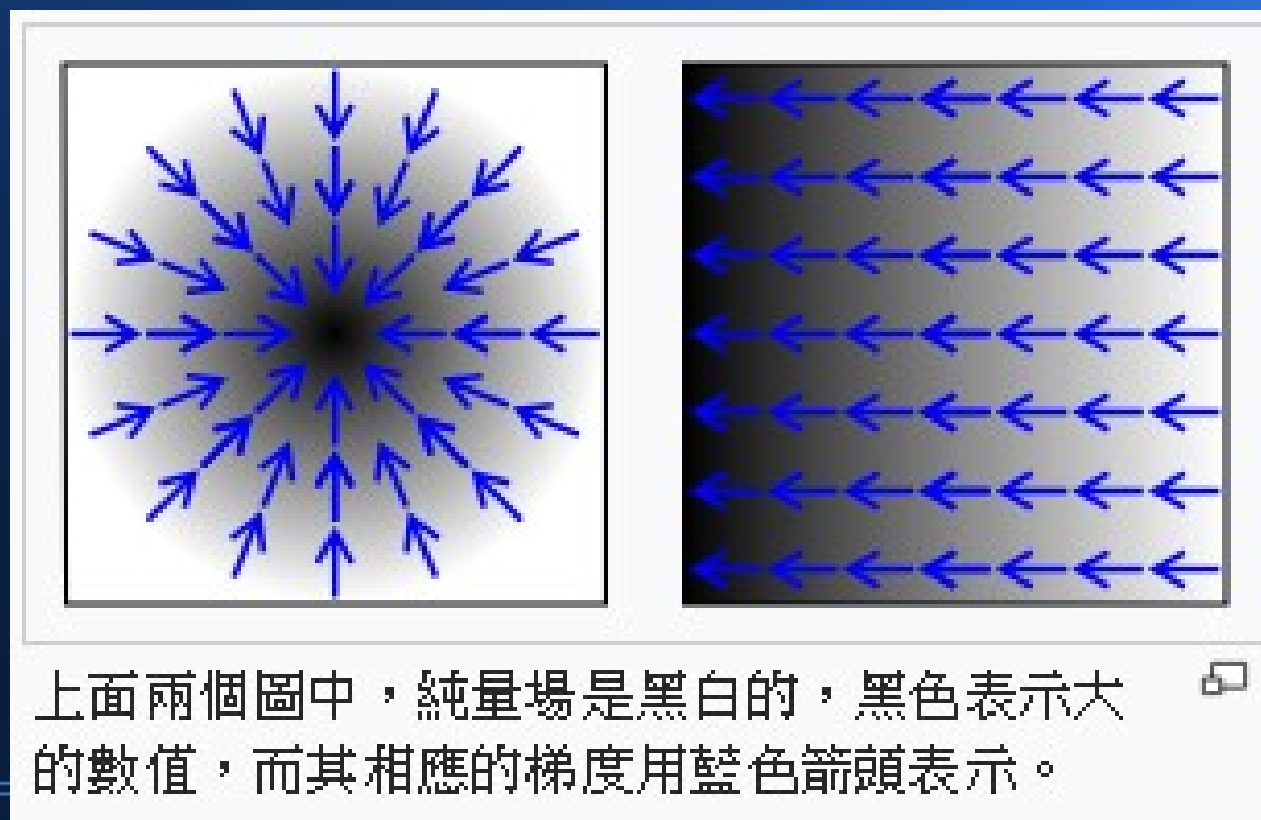
- 其實是看你要找《最大值》還是《最小值》而定的！
- 兩者只是方向相反而已。

只要能算出梯度

- 就能知道該走哪個方向了！

這種梯度概念

- 其實和物理學上的《力與場》的概念是相似的。



但是調整時

- 每一步通常不能調太大
- 否則可能會調過頭，而錯失了區域能量最低點（或最高點）！

但是上面那種計算梯度的方式

- 是數值微分法，還是得多算一次函數值

$$\frac{\partial f(x, y)}{\partial x} = \frac{f(x + h, y) - f(x, y)}{h}$$

- 如果有很多個變數調整 $(x+h, y+h, \dots)$ ，
就會計算很多次函數值

還好

- 微積分給了我們一個強大的工具
- 讓我們可以計算已知函數的微分式
- 而不需要用《數值微分》去計算！

以算式 $f(x, y) = xy$ 為例

- $f(x, y)$ 對 x 的偏導數為 y

$$\frac{\partial f(x, y)}{\partial x} = \frac{f(x + h, y) - f(x, y)}{h} = \frac{(x + h)y - xy}{h} = \frac{xy + hy - xy}{h} = \frac{hy}{h} = y$$

於是我們可以用 更快的方法朝梯度方向前進

數值微分（比較慢）

```
var x = -2, y = 3;
var out = forwardMultiplyGate(x, y); // -6
var h = 0.0001;

// compute derivative with respect to x
var xph = x + h; // -1.9999
var out2 = forwardMultiplyGate(xph, y); // -5.9998
var x_derivative = (out2 - out) / h; // 3

// compute derivative with respect to y
var yph = y + h; // 3.0001
var out3 = forwardMultiplyGate(x, yph); // -5.9998
var y_derivative = (out3 - out) / h; // -2.0
```

```
var step_size = 0.01;
var out = forwardMultiplyGate(x, y); // before: -6
x = x + step_size * x_derivative; // x becomes -1.97
y = y + step_size * y_derivative; // y becomes 2.98
var out_new = forwardMultiplyGate(x, y); // -5.87! exciting.
```

算式微分（比較快）

```
var x = -2, y = 3;
var out = forwardMultiplyGate(x, y); // -6
var x_gradient = y; // by our complex math
var y_gradient = x;

var step_size = 0.01;
x += step_size * x_gradient; // -1.97
y += step_size * y_gradient; // 2.98
var out_new = forwardMultiplyGate(x, y);
```

而且還會更精準

(因為數值微分只是逼近式而已)

但是

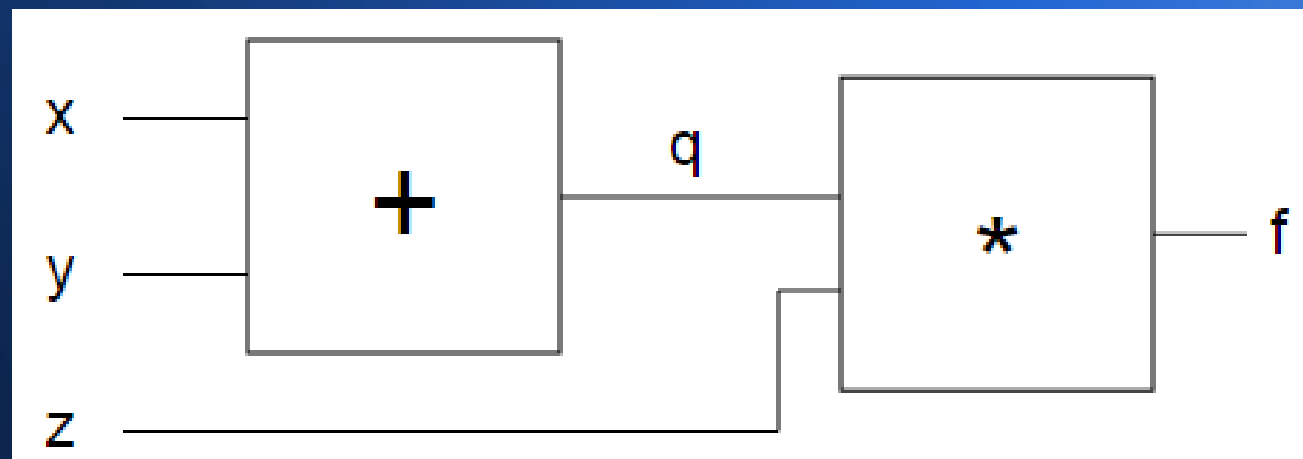
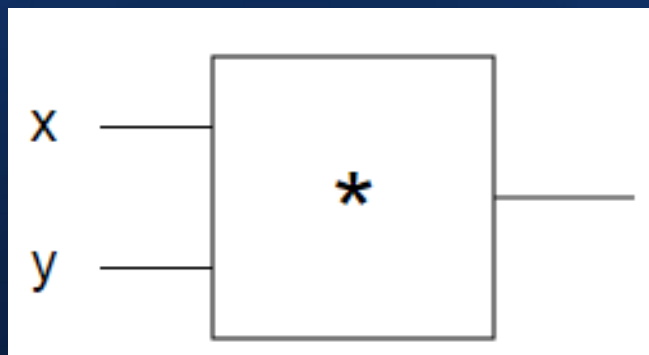
- 上述的 $f(x, y) = xy$ 範例
有點太簡單！

讓我們多加一層看看！

$$f(x, y) = xy$$



$$f(x, y, z) = (x + y)z$$

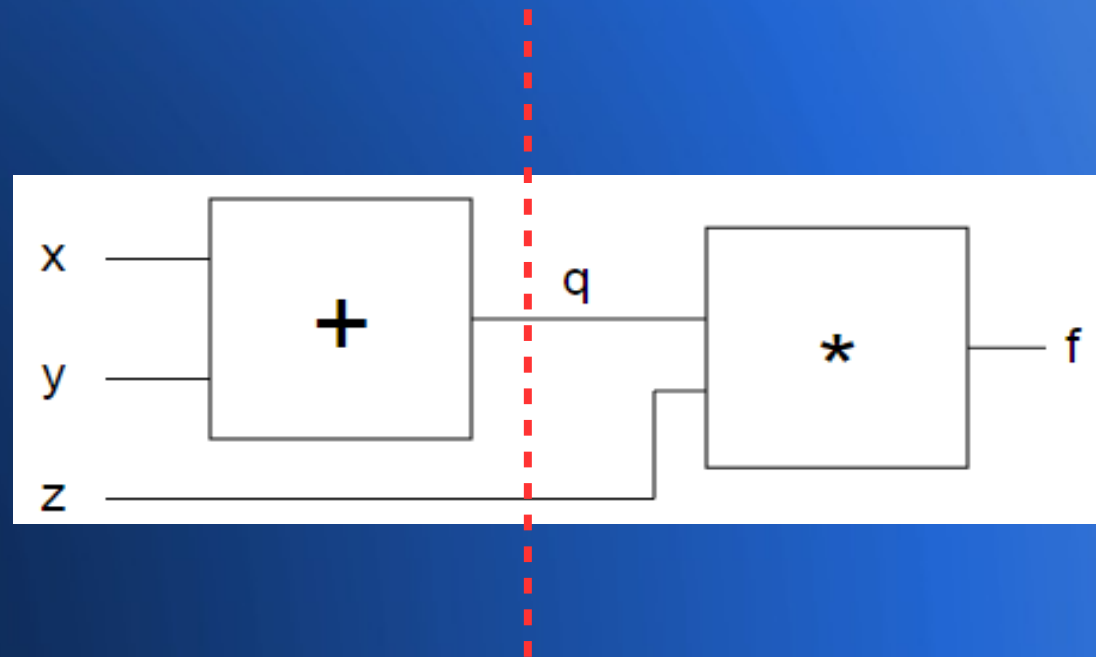


其計算程式會變成這樣

```
var forwardMultiplyGate = function(a, b) {  
  return a * b;  
};  
var forwardAddGate = function(a, b) {  
  return a + b;  
};  
var forwardCircuit = function(x,y,z) {  
  var q = forwardAddGate(x, y);  
  var f = forwardMultiplyGate(q, z);  
  return f;  
};  
  
var x = -2, y = 5, z = -4;  
var f = forwardCircuit(x, y, z); // output is -12
```

如果我們把他拆為兩層

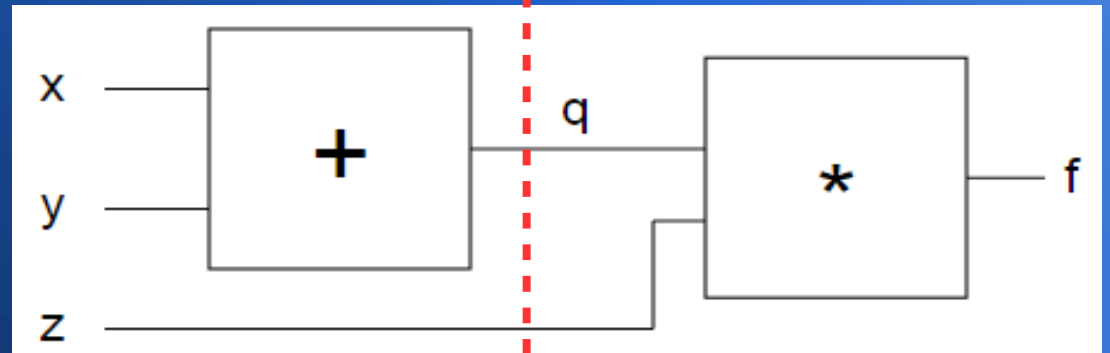
$$f(x, y, z) = (x + y)z$$



那麼偏導數的計算也可以分兩段

$$f(x, y, z) = (x + y)z$$

$$\frac{\partial f(q, z)}{\partial x} = \frac{\partial q(x, y)}{\partial x} \frac{\partial f(q, z)}{\partial q}$$



$$f(q, z) = qz$$

$$\Rightarrow$$

$$\frac{\partial f(q, z)}{\partial q} = z,$$

$$\frac{\partial f(q, z)}{\partial z} = q$$

$$q(x, y) = x + y$$

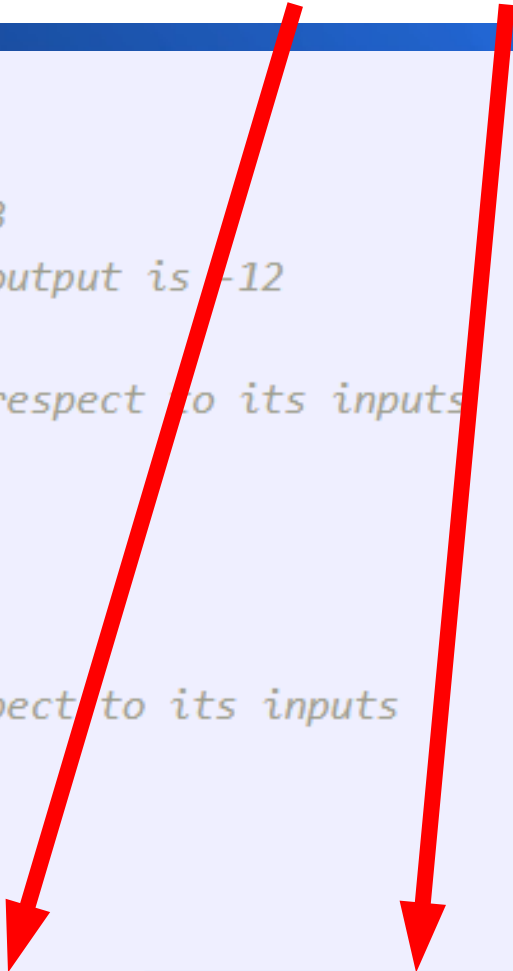
$$\Rightarrow$$

$$\frac{\partial q(x, y)}{\partial x} = 1,$$

$$\frac{\partial q(x, y)}{\partial y} = 1$$

寫成程式就會變這樣

$$f(x, y, z) = (x + y)z$$

$$\frac{\partial f(q, z)}{\partial x} = \frac{\partial q(x, y)}{\partial x} \frac{\partial f(q, z)}{\partial q}$$


```
// initial conditions
var x = -2, y = 5, z = -4;
var q = forwardAddGate(x, y); // q is 3
var f = forwardMultiplyGate(q, z); // output is -12

// gradient of the MULTIPLY gate with respect to its inputs
// wrt is short for "with respect to"
var derivative_f_wrt_z = q; // 3
var derivative_f_wrt_q = z; // -4

// derivative of the ADD gate with respect to its inputs
var derivative_q_wrt_x = 1.0;
var derivative_q_wrt_y = 1.0;

// chain rule
var derivative_f_wrt_x = derivative_q_wrt_x * derivative_f_wrt_q; // -4
var derivative_f_wrt_y = derivative_q_wrt_y * derivative_f_wrt_q; // -4
```

然後我們就可以知道 x, y, z
各應該調多少才會朝梯度方向前進

```
// final gradient, from above: [-4, -4, 3]
var gradient_f_wrt_xyz = [derivative_f_wrt_x, derivative_f_wrt_y, derivative_f_wrt_z]

// let the inputs respond to the force/tug:
var step_size = 0.01;
x = x + step_size * derivative_f_wrt_x; // -2.04
y = y + step_size * derivative_f_wrt_y; // 4.96
z = z + step_size * derivative_f_wrt_z; // -3.97

// Our circuit now better give higher output:
var q = forwardAddGate(x, y); // q becomes 2.92
var f = forwardMultiplyGate(q, z); // output is -11.59, up from -12! Nice!
```

```
// chain rule
var derivative_f_wrt_x = derivative_q_wrt_x * derivative_f_wrt_q; // -4
var derivative_f_wrt_y = derivative_q_wrt_y * derivative_f_wrt_q; // -4
```

換個方式說


- 我們先算 q 應該調多少才會讓 f 朝梯度方向前進

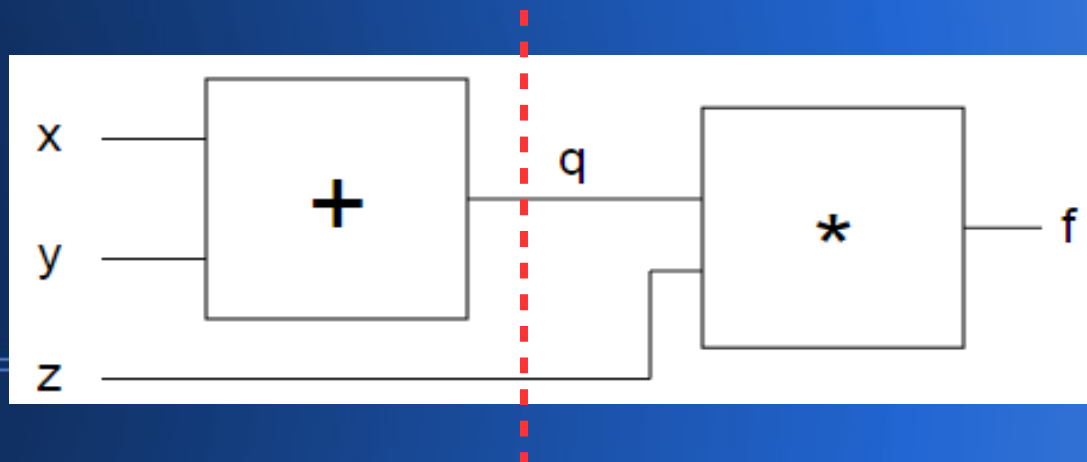
$$\frac{\partial f(q, z)}{\partial q}$$

- 然後算 x 應該調多少才會讓 q 朝梯度方向前進

$$\frac{\partial q(x, y)}{\partial x}$$

- 兩者相乘得到 x 應該調多少才會讓 f 朝梯度方向前進

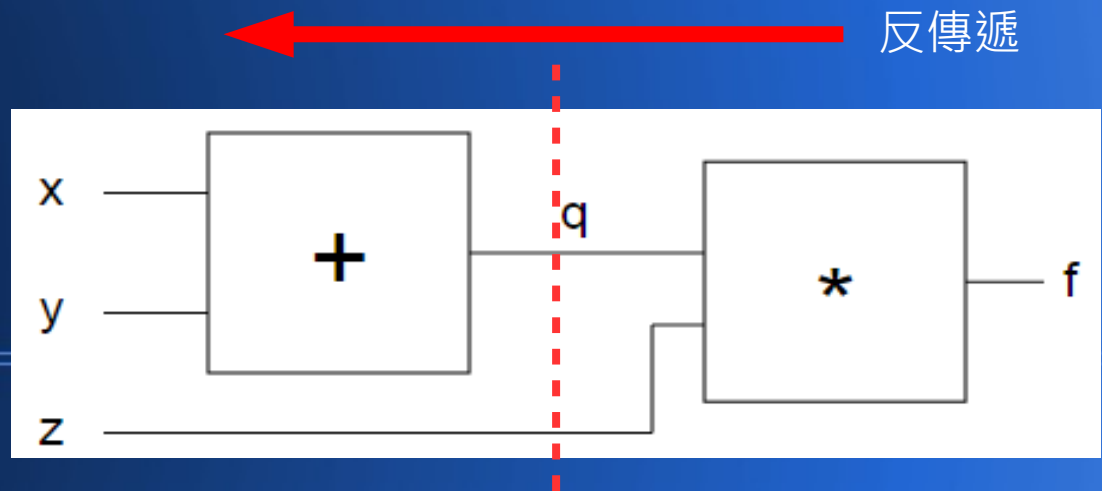
$$\frac{\partial f(q, z)}{\partial x} = \frac{\partial q(x, y)}{\partial x} \frac{\partial f(q, z)}{\partial q}$$




因此在調整的時候

- 要從輸出層 f 開始反向計算
- 然後透過鏈鎖規則一層一層算回來
- 這就是反傳遞演算法的原理了

$$\frac{\partial f(q, z)}{\partial x} = \frac{\partial q(x, y)}{\partial x} \frac{\partial f(q, z)}{\partial q}$$



在反傳遞時

- 如果某層要調很多（拉得很用力）
- 哪麼反傳遞力量很大，也會造成前面的輸入要用力調很大！

如果我們對微分式沒把握

- 其實可以用數值微分驗證一遍

```
// initial conditions
var x = -2, y = 5, z = -4;

// numerical gradient check
var h = 0.0001;
var x_derivative = (forwardCircuit(x+h,y,z) - forwardCircuit(x,y,z)) / h; // -4
var y_derivative = (forwardCircuit(x,y+h,z) - forwardCircuit(x,y,z)) / h; // -4
var z_derivative = (forwardCircuit(x,y,z+h) - forwardCircuit(x,y,z)) / h; // 3
```

and we get `[-4, -4, 3]`, as computed with backprop. phew! :)

$$f(x, y, z) = (x + y)z$$

$$f(q, z) = qz$$

$$q(x, y) = x + y$$

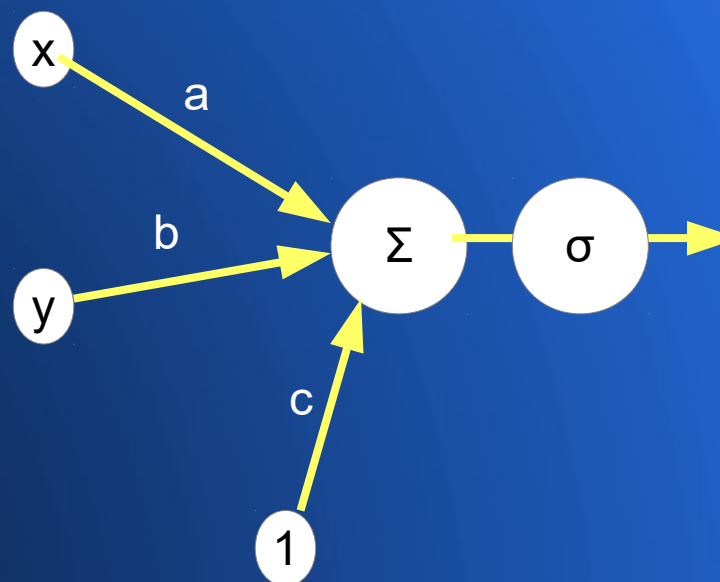
$$\frac{\partial f(q, z)}{\partial x} = \frac{\partial q(x, y)}{\partial x} \frac{\partial f(q, z)}{\partial q} = 1 * -4 = -4$$

在理解了反傳遞法之後

- 讓我們來看看真正《神經網路》神經元的正向計算與反向傳遞

對於一個有兩個輸入的神經元

$$f(x, y, a, b, c) = \sigma(ax + by + c)$$



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

神經網路需要兩個值

- 一個是前向 forward 計算使用的 value
- 一個是反向 backward 傳遞使用的 grad（梯度）

```
// every Unit corresponds to a wire in the diagrams  
var Unit = function(value, grad) {  
  // value computed in the forward pass  
  this.value = value;  
  // the derivative of circuit output w.r.t this unit, computed in backward pass  
  this.grad = grad;  
}
```

對於一個乘法單元

- 前後向的更新分別如下

```
var multiplyGate = function(){ };
multiplyGate.prototype = {
  forward: function(u0, u1) {
    // store pointers to input Units u0 and u1 and output unit utop
    this.u0 = u0;
    this.u1 = u1;
    this.utop = new Unit(u0.value * u1.value, 0.0);
    return this.utop;
  },
  backward: function() {
    // take the gradient in output unit and chain it with the
    // local gradients, which we derived for multiply gate before
    // then write those gradients to those Units.
    this.u0.grad += this.u1.value * this.utop.grad;
    this.u1.grad += this.u0.value * this.utop.grad;
  }
}
```

$$f(q, z) = qz \quad \implies \quad \frac{\partial f(q, z)}{\partial q} = z, \quad \frac{\partial f(q, z)}{\partial z} = q$$

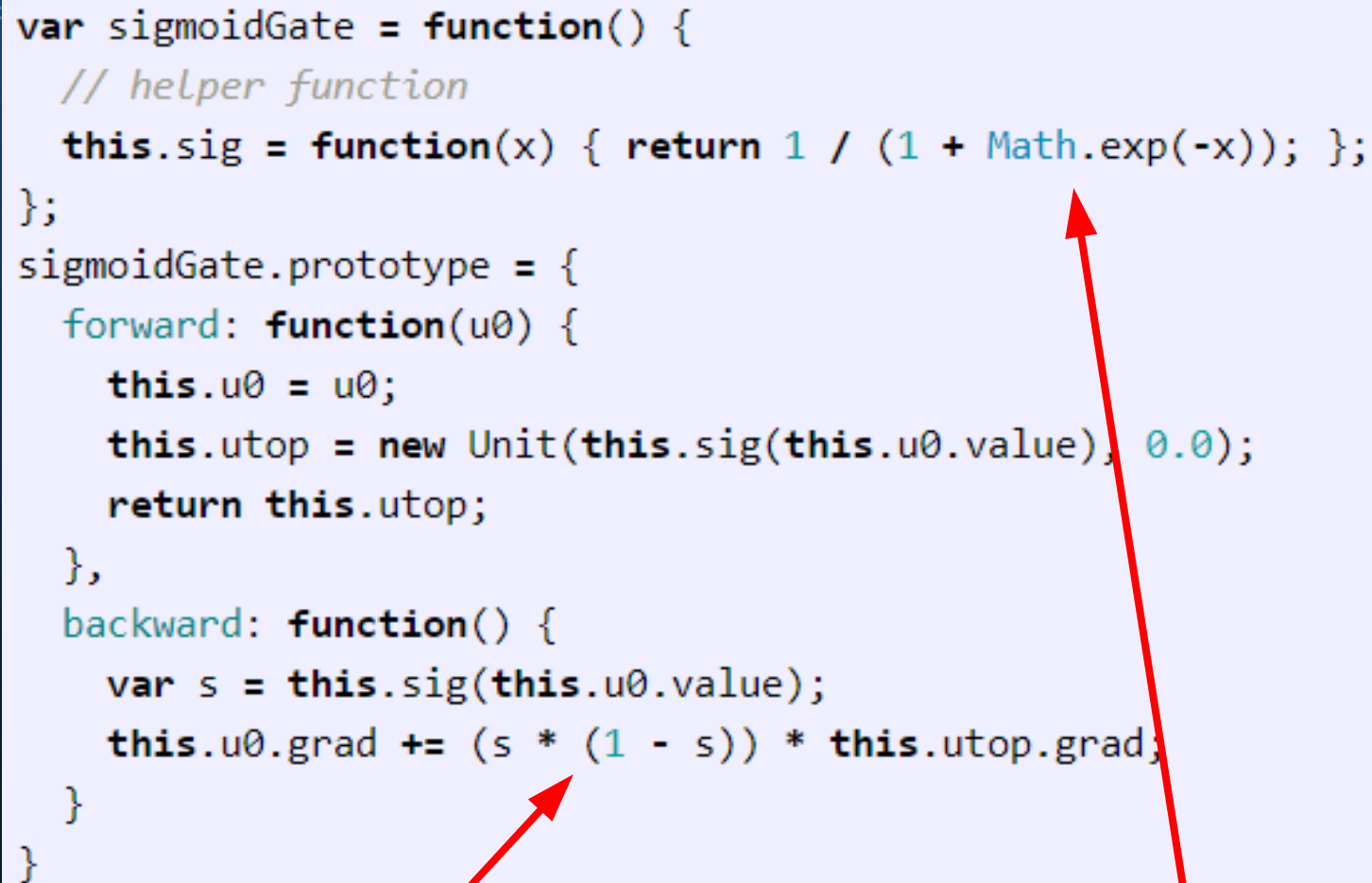
對於加法單元則如下

```
var addGate = function(){ };
addGate.prototype = {
  forward: function(u0, u1) {
    this.u0 = u0;
    this.u1 = u1; // store pointers to input units
    this.utop = new Unit(u0.value + u1.value, 0.0);
    return this.utop;
  },
  backward: function() {
    // add gate. derivative wrt both inputs is 1
    this.u0.grad += 1 * this.utop.grad;
    this.u1.grad += 1 * this.utop.grad;
  }
}
```

$$q(x, y) = x + y \quad \implies \quad \frac{\partial q(x, y)}{\partial x} = 1, \quad \frac{\partial q(x, y)}{\partial y} = 1$$

然後 sigmoid 單元的算式如下

```
var sigmoidGate = function() {  
  // helper function  
  this.sig = function(x) { return 1 / (1 + Math.exp(-x)); };  
};  
sigmoidGate.prototype = {  
  forward: function(u0) {  
    this.u0 = u0;  
    this.utop = new Unit(this.sig(this.u0.value), 0.0);  
    return this.utop;  
  },  
  backward: function() {  
    var s = this.sig(this.u0.value);  
    this.u0.grad += (s * (1 - s)) * this.utop.grad;  
  }  
}
```



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

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

最後這些 單元可以 組合起來

```
// create input units
var a = new Unit(1.0, 0.0);
var b = new Unit(2.0, 0.0);
var c = new Unit(-3.0, 0.0);
var x = new Unit(-1.0, 0.0);
var y = new Unit(3.0, 0.0);

// create the gates
var mulg0 = new multiplyGate();
var mulg1 = new multiplyGate();
var addg0 = new addGate();
var addg1 = new addGate();
var sg0 = new sigmoidGate();

// do the forward pass
var forwardNeuron = function() {
  ax = mulg0.forward(a, x); //  $a*x = -1$ 
  by = mulg1.forward(b, y); //  $b*y = 6$ 
  axpby = addg0.forward(ax, by); //  $a*x + b*y = 5$ 
  axpbypc = addg1.forward(axpby, c); //  $a*x + b*y + c = 2$ 
  s = sg0.forward(axpbypc); //  $\text{sig}(a*x + b*y + c) = 0.8808$ 
};
forwardNeuron();

console.log('circuit output: ' + s.value); // prints 0.8808
```

```
s.grad = 1.0;
sg0.backward(); //
addg1.backward();
addg0.backward();
mulg1.backward();
mulg0.backward();
```

前向計算

反向傳遞

反向傳遞梯度後 程式會根據步伐大小更新其值

```
var step_size = 0.01;
a.value += step_size * a.grad; // a.grad is -0.105
b.value += step_size * b.grad; // b.grad is 0.315
c.value += step_size * c.grad; // c.grad is 0.105
x.value += step_size * x.grad; // x.grad is 0.105
y.value += step_size * y.grad; // y.grad is 0.210

forwardNeuron();
console.log('circuit output after one backprop: ' + s.value); // prints 0.8825
```

更新完之後的結果應該會更好： $0.8825 > 0.8808$

```
var forwardNeuron = function() {
  ax = mulg0.forward(a, x); //  $a*x = -1$ 
  by = mulg1.forward(b, y); //  $b*y = 6$ 
  axpby = addg0.forward(ax, by); //  $a*x + b*y = 5$ 
  axpbypc = addg1.forward(axpby, c); //  $a*x + b*y + c = 2$ 
  s = sg0.forward(axpbypc); //  $\text{sig}(a*x + b*y + c) = 0.8808$ 
};

forwardNeuron();

console.log('circuit output: ' + s.value); // prints 0.8808
```

如果我們改用數值微分計算

- 結果應該是一樣的

```
var forwardCircuitFast = function(a,b,c,x,y) {  
    return 1/(1 + Math.exp( - (a*x + b*y + c)));  
};  
var a = 1, b = 2, c = -3, x = -1, y = 3;  
var h = 0.0001;  
var a_grad = (forwardCircuitFast(a+h,b,c,x,y) - forwardCircuitFast(a,b,c,x,y))/h;  
var b_grad = (forwardCircuitFast(a,b+h,c,x,y) - forwardCircuitFast(a,b,c,x,y))/h;  
var c_grad = (forwardCircuitFast(a,b,c+h,x,y) - forwardCircuitFast(a,b,c,x,y))/h;  
var x_grad = (forwardCircuitFast(a,b,c,x+h,y) - forwardCircuitFast(a,b,c,x,y))/h;  
var y_grad = (forwardCircuitFast(a,b,c,x,y+h) - forwardCircuitFast(a,b,c,x,y))/h;
```

現在你應該會發現

- 其實寫神經網路程式沒那麼困難

接著

- Karpathy 開始簡化反傳遞的概念
- 企圖讓我們可以成為《反傳遞忍者》

Becoming a Backprop Ninja

Over time you will become much more efficient in writing the backward pass, even for complicated circuits and all at once. Lets practice backprop a bit with a few examples. In what follows, lets not worry about Unit, Circuit classes because they obfuscate things a bit, and lets just use variables such as `a, b, c, x`, and refer to their gradients as `da, db, dc, dx` respectively. Again, we think of the variables as the “forward flow” and their gradients as “backward flow” along every wire. Our first example was the `*` gate:

```
var x = a * b;  
// and given gradient on x (dx), we saw that in backprop we would compute:  
var da = b * dx;  
var db = a * dx;
```

但是、上述的微積分符號太複雜

- 所以 Karpathy 用更簡單的符號 dx , dy , da , db 來代表 x , y , a , b 變數的《梯度》
- 如右邊的算式所示 \Rightarrow

$$dx = \frac{\partial f}{\partial x}$$

$$da = \frac{\partial f}{\partial a}$$

$$db = \frac{\partial f}{\partial b}$$

所以對於乘法單元

- 我們可以用下列梯度計算公式

```
var x = a * b;
```

```
// and given gradient on x (dx), we saw that in backprop we would compute:
```

```
var da = b * dx;
```

```
var db = a * dx;
```

$$\frac{\partial f}{\partial a} = \frac{\partial x}{\partial a} \frac{\partial f}{\partial x} = b * \frac{\partial f}{\partial x}$$

必須注意的是：這裡的 **da**, **db**, **dx** 通通都代表該變數的梯度，也就是 $\frac{\partial f}{\partial a}$ ，而非微分符號 $\frac{da}{dx}$ 當中的概念。

所以下列的方程式

```
var x = a * b;  
// and given gradient on x (dx), we saw that in backprop we would compute:  
var da = b * dx;  
var db = a * dx;
```

- 代表當 $x = a*b$ 時
 - a 的梯度 $da = b*dx$
 - b 的梯度 $db = a*dx$
- 其中 dx 代表已知上一層的 x 之梯度

這種寫法會比上面那些程式更簡潔

```
var x = a * b;
```

// and given gradient on x (dx), we saw that in backprop we would compute:

```
var da = b * dx;
```

```
var db = a * dx;
```

```
var multiplyGate = function(){ };
```

```
multiplyGate.prototype = {
```

```
  forward: function(u0, u1) {
```

```
    // store pointers to input Units u0 and u1 and output unit utop
```

```
    this.u0 = u0; // u0 = a
```

```
    this.u1 = u1; // u1 = b
```

```
    this.utop = new Unit(u0.value * u1.value, 0.0);
```

```
    return this.utop;
```

```
  },
```

```
  backward: function() {
```

```
    // take the gradient in output unit and chain it with the
```

```
    // local gradients, which we derived for multiply gate before
```

```
    // then write those gradients to those Units.
```

```
    this.u0.grad += this.u1.value * this.utop.grad; // da = b * dx
```

```
    this.u1.grad += this.u0.value * this.utop.grad; // db = a * dx
```

```
  }
```

```
}
```

於是我們可以很快推出梯度公式

- 成為梯度忍者

加法單元的梯度計算方式

```
var x = a + b;  
// ->  
var da = 1.0 * dx;  
var db = 1.0 * dx;
```

三個數字相加的梯度忍術


```
// lets compute  $x = a + b + c$  in two steps:  
var q = a + b; // gate 1  
var x = q + c; // gate 2  
  
// backward pass:  
dc = 1.0 * dx; // backprop gate 2  
dq = 1.0 * dx;  
da = 1.0 * dq; // backprop gate 1  
db = 1.0 * dq;
```

線性轉換的梯度忍術

```
var x = a * b + c;  
// given dx, backprop in-one-sweep would be =>  
da = b * dx;  
db = a * dx;  
dc = 1.0 * dx;
```

Sigmoid 單元的梯度忍術

```
// Lets do our neuron in two steps:  
var q = a*x + b*y + c;  
var f = sig(q); // sig is the sigmoid function  
// and now backward pass, we are given df, and:  
var df = 1;  
var dq = (f * (1 - f)) * df;  
// and now we chain it to the inputs  
var da = x * dq;  
var dx = a * dq;  
var dy = b * dq;  
var db = y * dq;  
var dc = 1.0 * dq;
```



平方式的梯度忍術

可以套用乘法單元的公式

```
var x = a * a;  
var da = ///???
```

```
var x = a * b;  
// and given gra  
var da = b * dx;  
var db = a * dx;
```

```
var da = a * dx; // gradient into a from first branch  
da += a * dx; // and add on the gradient from the second branch  
  
// short form instead is:  
var da = 2 * a * dx;
```

這和微積分裡的結果是一樣的

$$f(a) = a^2$$

$$\frac{\partial f(a)}{\partial a} = 2a$$

更複雜的平方總和式

- 也不會太難！

```
var x = a*a + b*b + c*c;  
// we get:  
var da = 2*a*dx;  
var db = 2*b*dx;  
var dc = 2*c*dx;
```

如果算式更複雜一些

```
var x = Math.pow(((a * b + c) * d), 2);
```


也只要按這上述方法一步一步來

```
var x = Math.pow(((a * b + c) * d), 2);
```

```
var x1 = a * b + c;  
var x2 = x1 * d;  
var x = x2 * x2; // this is identical to the above expression for x  
// and now in backprop we go backwards:  
var dx2 = 2 * x2 * dx; // backprop into x2  
var dd = x1 * dx2; // backprop into d  
var dx1 = d * dx2; // backprop into x1  
var da = b * dx1;  
var db = a * dx1;  
var dc = 1.0 * dx1; // done!
```

就可以算出梯度值， 成為梯度忍者 ...

如果有除法

- 那麼也是按公式來

```
var x = 1.0/a; // division  
var da = -1.0/(a*a);
```

包含除法的更複雜算式

```
var x = (a + b)/(c + d);  
// lets decompose it in steps:  
var x1 = a + b;  
var x2 = c + d;  
var x3 = 1.0 / x2;  
var x = x1 * x3; // equivalent to above  
// and now backprop, again in reverse order:  
var dx1 = x3 * dx;  
var dx3 = x1 * dx;  
var dx2 = (-1.0/(x2*x2)) * dx3; // local gradient as shown above, and chain rule  
var da = 1.0 * dx1; // and finally into the original variables  
var db = 1.0 * dx1;  
var dc = 1.0 * dx2;  
var dd = 1.0 * dx2;
```

也可以按部就班的反向傳遞計算

萬一有像 $\max(a, b)$ 這樣的取大值算式

- 也只要按照定義計算導數

```
var x = Math.max(a, b);  
var da = a === x ? 1.0 * dx : 0.0;  
var db = b === x ? 1.0 * dx : 0.0;
```

捲積神經網路裏常用的 ReLU 函數

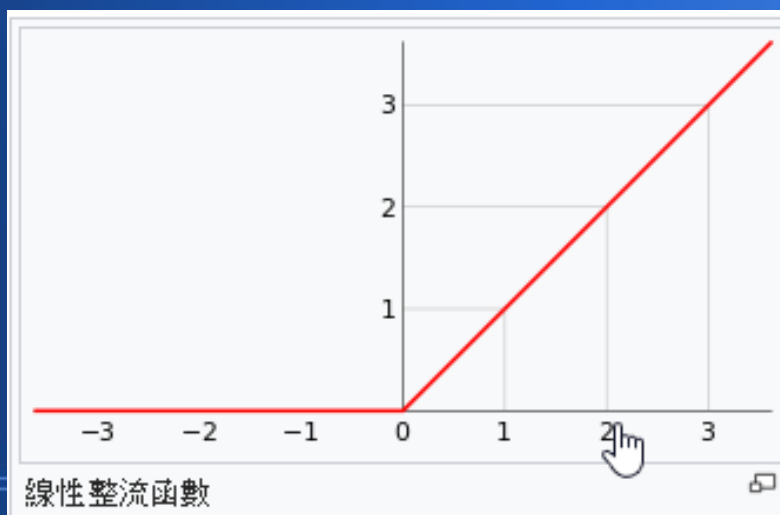
- 其實就是 $\max(a, 0)$ 的算式

```
var x = Math.max(a, 0)
// backprop through this gate will then be:
var da = a > 0 ? 1.0 * dx : 0.0;
```

線性整流函數 (Rectified Linear Unit, **ReLU**) ,又稱修正線性單元, 是一種人工神經網路中常用的激活函數 (activation function) , 通常指代以斜坡函數及其變種為代表的非線性函數。

比較常用的線性整流函數有斜坡函數 $f(x) = \max(0, x)$, 以及帶泄露整流函數 (Leaky ReLU) , 其中 x 為神經元 (Neuron) 的輸入。線性整流被認為有一定的生物學原理^[1] , 並且由於在實踐中通常有著比其他常用激活函數 (譬如邏輯函數) 更好的效果, 而被如今的深度神經網路廣泛使用於諸如圖像識別等計算機視覺^[1] 人工智慧領域。

線性整



看到這裡

- 您應該會有所感受

只要透過模組化的設計方式

- 那些可怕的微分式
- 其實都會變得很簡單！

甚至

- 我們可以透過這種梯度忍術
- 撰寫出《自動微分》的程式

Karpathy 在 recurrent.js 專案裡

- 就展示了這種
自動微分技術

```
sigmoid: function(m) {  
  // sigmoid nonlinearity  
  var out = new Mat(m.n, m.d);  
  var n = m.w.length;  
  for(var i=0; i<n; i++) {  
    out.w[i] = sig(m.w[i]);  
  }  
  
  if(this.needs_backprop) {  
    var backward = function() {  
      for(var i=0; i<n; i++) {  
        // grad for z = tanh(x) is (1 - z^2)  
        var mwi = out.w[i];  
        m.dw[i] += mwi * (1.0 - mwi) * out.dw[i];  
      }  
    }  
    this.backprop.push(backward);  
  }  
  return out;  
},
```

另外像 dritchie 在 adnn 專案裏

- 運用自動微分 ad (Automatic differentiation)
- 建構了整套神經網路 nn 工具
- 所以該專案才會叫做 adnn

以下是 adnn 的程式碼片段

```
d.neg = makeUnaryDerivatives('-1');
d.add = makeBinaryDerivatives('1', '1');
d.sub = makeBinaryDerivatives('1', '-1');
d.mul = makeBinaryDerivatives('y', 'x');
d.div = makeBinaryDerivatives('1/y', '-x/(y*y)');
d.sqrt = makeUnaryDerivatives('1/(2*out)');
d.exp = makeUnaryDerivatives('out');
d.log = makeUnaryDerivatives('1/x');
d.pow = makeBinaryDerivatives('y*Math.pow(x,y-1)', 'Math.log(x)*out');
d.sin = makeUnaryDerivatives('Math.cos(x)');
d.cos = makeUnaryDerivatives('-Math.sin(x)');
d.tan = makeUnaryDerivatives('1 + out*out');
d.asin = makeUnaryDerivatives('1 / Math.sqrt(1 - x*x)');
d.acos = makeUnaryDerivatives('-1 / Math.sqrt(1 - x*x)');
d.atan = makeUnaryDerivatives('1 / (1 + x*x)');
d.atan2 = makeBinaryDerivatives('y/(x*x + y*y)', '-x/(x*x + y*y)');
d.sinh = makeUnaryDerivatives('Math.cosh(x)');
d.cosh = makeUnaryDerivatives('Math.sinh(x)');
d.tanh = makeUnaryDerivatives('1 - out*out');
d.asinh = makeUnaryDerivatives('1 / Math.sqrt(x*x + 1)');
d.acosh = makeUnaryDerivatives('1 / Math.sqrt(x*x - 1)');
d.atanh = makeUnaryDerivatives('1 / (1 - x*x)');
```

而 ehaas 在 js-autodiff 專案裏

- 則結合了 uglify-js 與 vm 等套件，
用編譯器技術，直接建構出系統程
式等級的 JavaScript 自動微分技術

以下是 js-autodiff 的程式碼片段

```
var AD = {
  neg: function(x) {
    return Dual(-x.a, -x.b);
  },
  incr: function(x) {
    ++x.a;
  },
  decr: function(x) {
    --x.a;
  },
  gt: function(x0, x1) {
    return x0.a > x1.a;
  },
  gte: function(x0, x1) {
    return x0.a >= x1.a;
  },
  lt: function(x0, x1) {
    return x0.a < x1.a;
  },
  lte: function(x0, x1) {
    return x0.a <= x1.a;
  },
```

```
  add: function(x0, x1) {
    return Dual(x0.a + x1.a, x0.b + x1.b);
  },
  sub: function(x0, x1) {
    return Dual(x0.a - x1.a, x0.b - x1.b);
  },
  mul: function(x0, x1) {
    return Dual(x0.a * x1.a, x0.a * x1.b + x1.a * x0.b);
  },
  div: function(x0, x1) {
    return Dual(x0.a / x1.a, (x1.a * x0.b - x0.a * x1.b) / (x1.a * x1.a));
  },
  addAssign: function(x0, x1) {
    var sum = AD.add(x0, x1);
    x0.a = sum.a;
    x0.b = sum.b;
  },
  subAssign: function(x0, x1) {
    var diff = AD.sub(x0, x1);
    x0.a = diff.a;
    x0.b = diff.b;
  },
```

有了自動微分之後

- 要建構神經網路與反傳遞功能
就會變得相當簡單了...

像是 adnn 裏的多層感知器 MLP

只用了幾行程式碼就做完了

```
var sequence = require('../composition.js').sequence;
var linear = require('./linear.js').linear;

// Convenience function for defining multilayer perceptrons
function mlp(nIn, layerdefs, optname, optDebug) {
  optname = optname || 'mlp';
  var nets = [];
  for (var i = 0; i < layerdefs.length; i++) {
    var ldef = layerdefs[i];
    nets.push(linear(nIn, ldef.nOut, optname+'_layer'+i));
    if (ldef.activation) {
      nets.push(ldef.activation);
    }
    nIn = ldef.nOut;
  }
  return sequence(nets, optname || 'multiLayerPerceptron', optDebug);
}
```

只要您看完 Karpathy 這篇

- Hacker's guide to Neural Networks
- 就能成為《反傳遞演算法忍者》，再也不用害怕那些惱人的偏微分符號了 ...

在我看完這篇文章後

立馬就動手寫了一個叫做 nn6 的《自動微分+神經網路套件》！



雖然不像 tensorflow 那麼強大

但至少可以在上課時教學用



The screenshot shows a web browser displaying a GitHub repository page. The address bar shows the URL: <https://github.com/cccnqu/ai106b/tree/master/example/03-neuralnet/nn6>. The page title is "README.md". The main heading is "nn6 -- 神經網路套件". Below it is the section "安裝" (Installation) with the command `$ npm install nn6`. The next section is "範例 1 -- mlp7seg.js" (Example 1 -- mlp7seg.js) with the subtitle "學習七段顯示器函數" (Learning 7-segment display function). Below this is a code block showing a neural network configuration for digit recognition.

```
// A B C D E F G
[1,1,1,1,1,1,0], // 0
[0,1,1,0,0,0,0], // 1
```

<https://github.com/cccnqu/ai106b/tree/master/example/03-neuralnet/nn6>

這就是 Andrej Karpathy

- 在 Hacker's guide to Neural Networks 這篇網誌裡
- 講給我們程式人聽的《神經網路原理》！

如果您還覺得有疑問

- 建議看看原文：

<http://karpathy.github.io/neuralnets/>

這就是我們今天的

十分鐘系列

希望您會喜歡！

我們下回見！

Bye Bye!