程式人

用十分鐘瞭解

《人工智慧的那些問題與方法》

（函數優化、爬山演算法與模擬退火法）

陳鍾誠

2016 年 3 月 14 日

《人工智慧》這個詞

- 聽起來就很高級！

所以、很多人會覺得

- 人工智慧的問題一定很難！

但是、真的那麼難嗎？

- 其實不一定！

很多人工智慧的問題

- 其實都很簡單！

但是、研究者為了讓它看起來高級一點

- 總是寫了一大堆數學！

這讓它看起來

- 好像真的很難！

其實、那些數學

- 我我大部分都看不懂！

那怎麼辦？

沒關係

- 我們有直覺！

所以

- 在這次的十分鐘系列中！

我要告訴大家

如何用直覺理解

- 人工智慧的那些理論！

首先、讓我們看看

- 到底人工智慧研究的
- 是那些問題？

基本上

- 就是要讓電腦具有和人類差不多的能力！

問題是

- 到底人類有哪些能力？

這個問題應該不難

因為你和我都是人類

我們人類

- 有五官：會《聽說讀寫》
- 有四肢：會《走唱跑跳》
- 有大腦：會《思考決策》

所以

- 電腦要有智慧，就要能模擬這些功能！

但是、這看起來很難！

不過、有時候很簡單

例如、你問一個小孩

- 請問你要一支棒棒糖還是兩支
  這件事情電腦應該也能回答！

但是、如果你問另一個小孩

- 請問你要被藤條打一下還是兩下？
- 這件事情應該也很容易決定！

這些問題

- 都有一個共同的特性

就是有好壞

人工智慧的核心問題

- 就是要判斷哪個好哪個壞？

換句話說

所有人工智慧的問題

- 都可以視為一種《優化問題》

尋找好的

- 放棄不好的！

最簡單的一種人工智慧問題

- 就是函數的優化問題！

舉例而言

如果你想找 17 的平方根

- 那你應該怎麼找呢？

其實、這個問題

- 我好像不知道怎麼解？

不過、如果你給我電腦

- 讓我寫個程式！

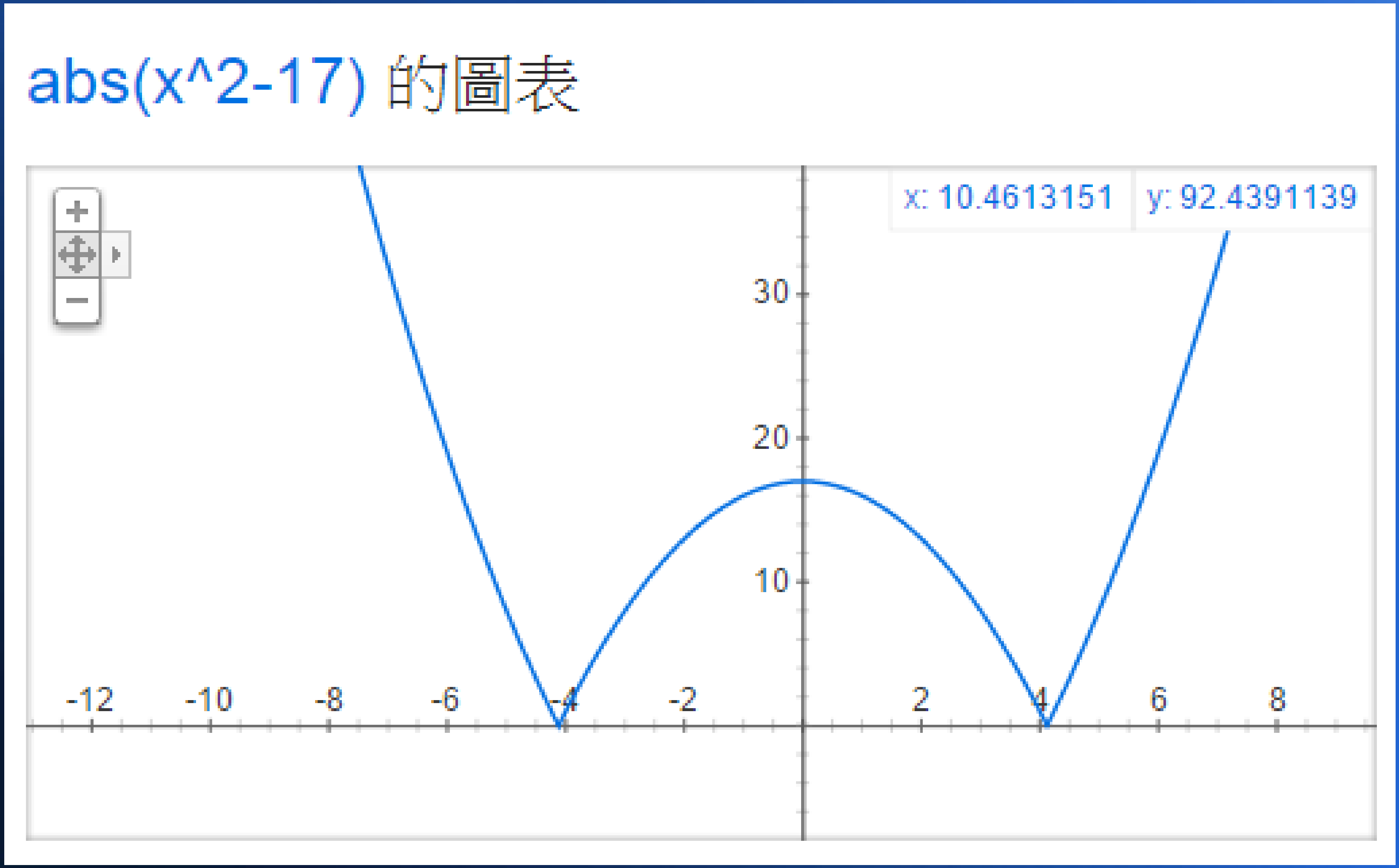
我就可以

- 輕易地給你解答！

怎麼解呢？

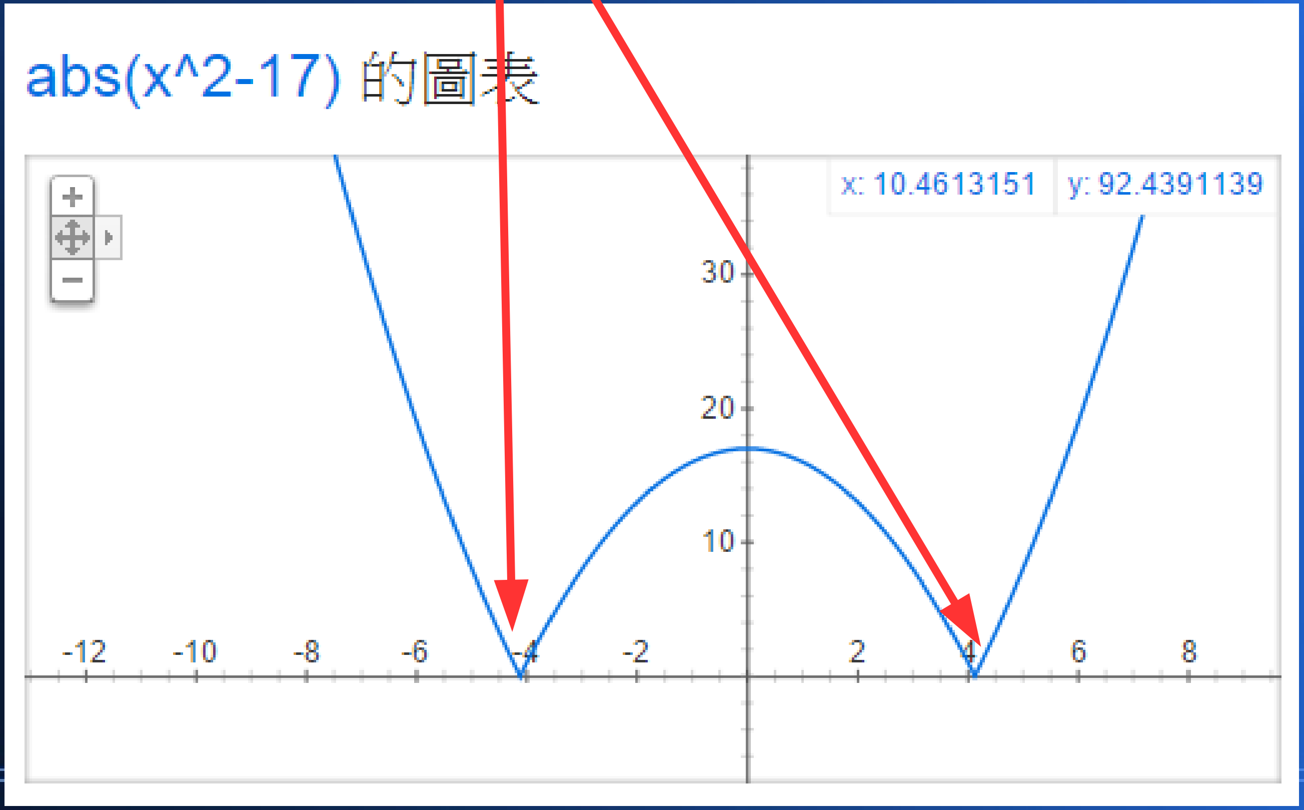
想法很簡單

讓我們先看看下列圖形



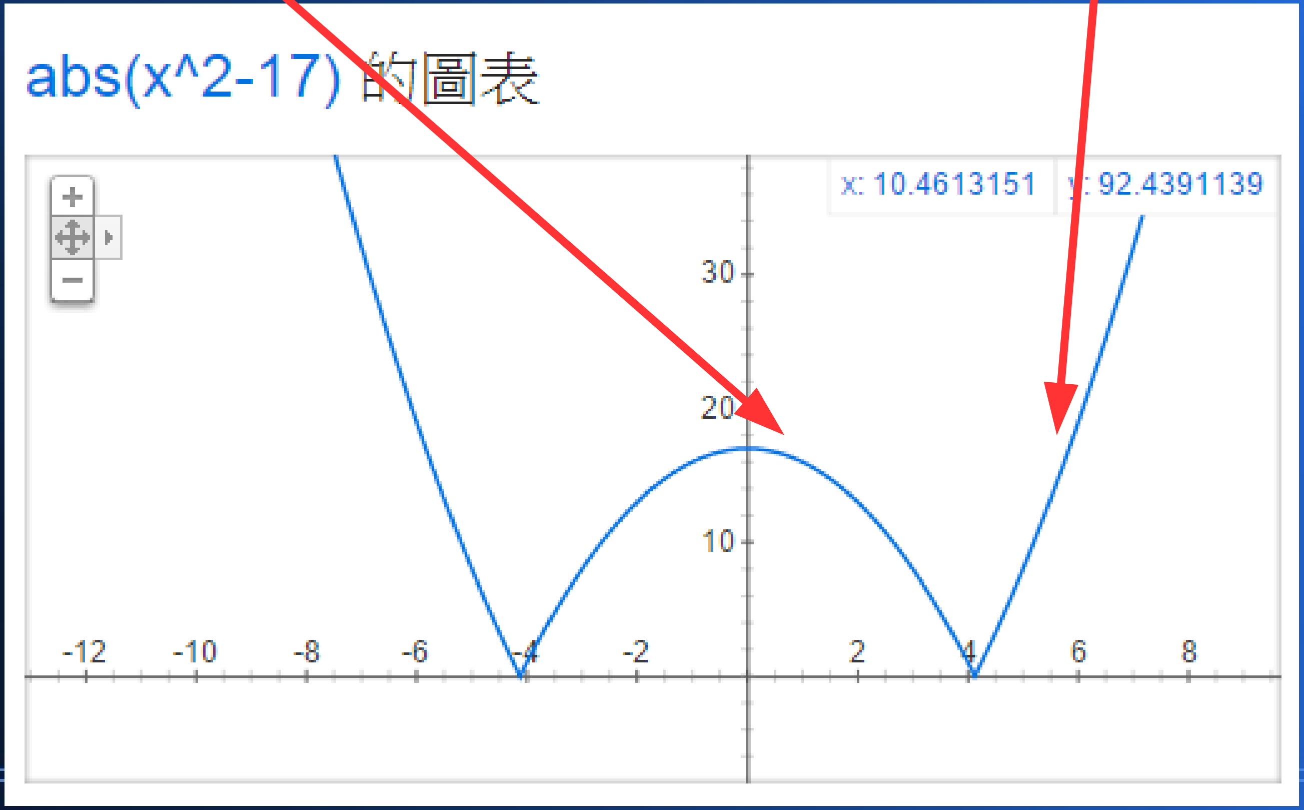
您會發現

abs(x^2-17) 這個函數，有兩個最低點，通常我們認為平方根是正的那個！



所以、如果我們從 x=0 開始

只要一直往右走，那麼將會發現，函數 f(x) = abs(x^2-17) 的值一開始會越來越小，等到過了最低點之後，就會變成越來越大！



於是、我們只要一直向右走

- 直到左右兩邊的函數值都比我大的時候，就代表了我們已經找到 17 的平方根了！

很多問題

- 表面上看起來
  - 並不是《優化問題》
  - 甚至不是《算分數》的問題
- 但是最後都可以轉為《算分數》的問題
  - 然後用電腦來解決！

像是、人工智慧中的

- 語音辨識、影像辨識、醫學診斷
- 電腦下棋、路徑規劃、機器人移動
- 甚至是機器翻譯、自然語言理解等等
- 最後都可以變成《計算分數的優化問題》，這樣才能夠用電腦進行計算並尋找解答！

所以、要學習人工智慧

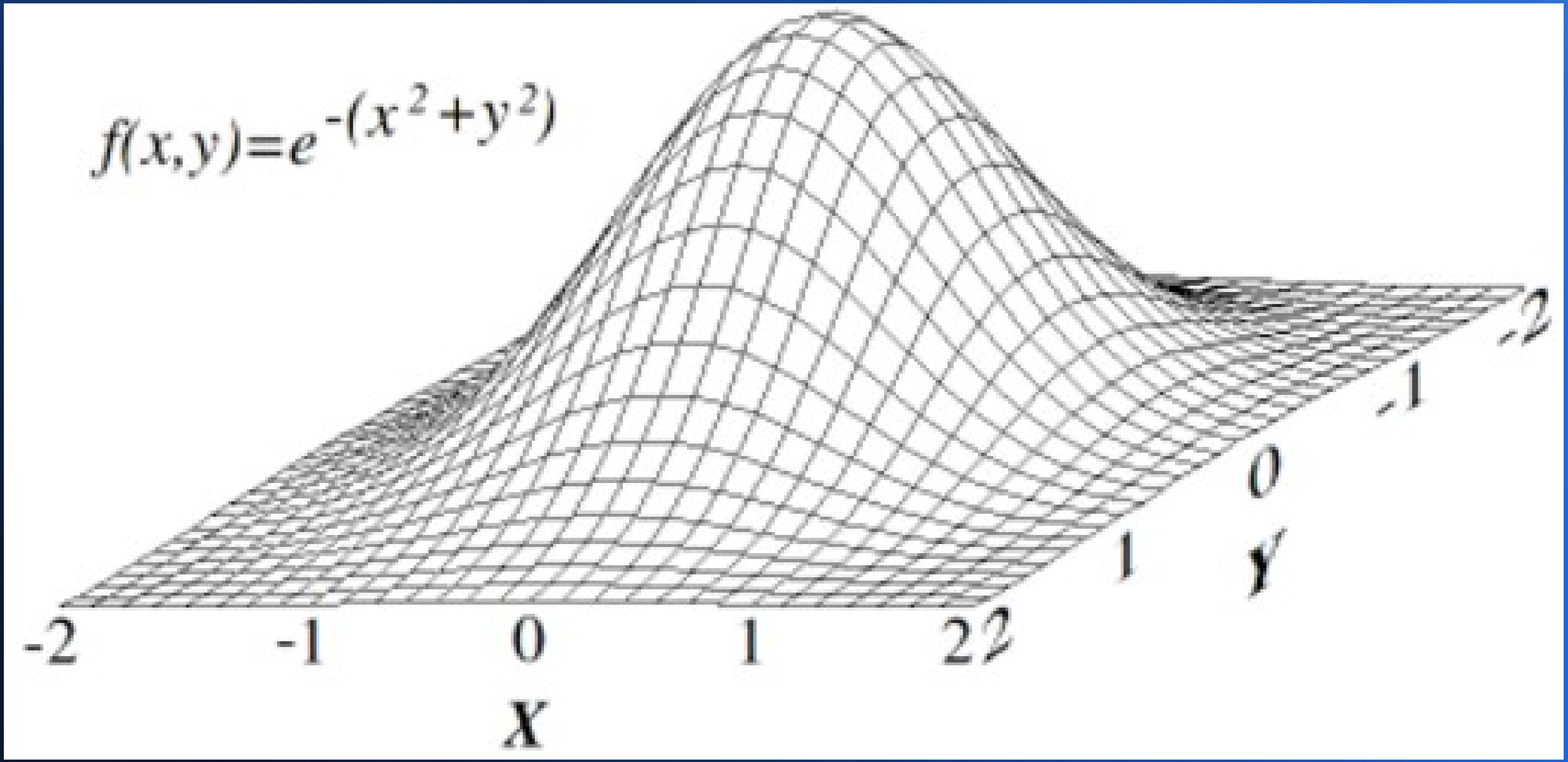
- 首先要能處理《優化問題》

處理優化問題的方法中

- 最簡單應該就是《爬山演算法》了！

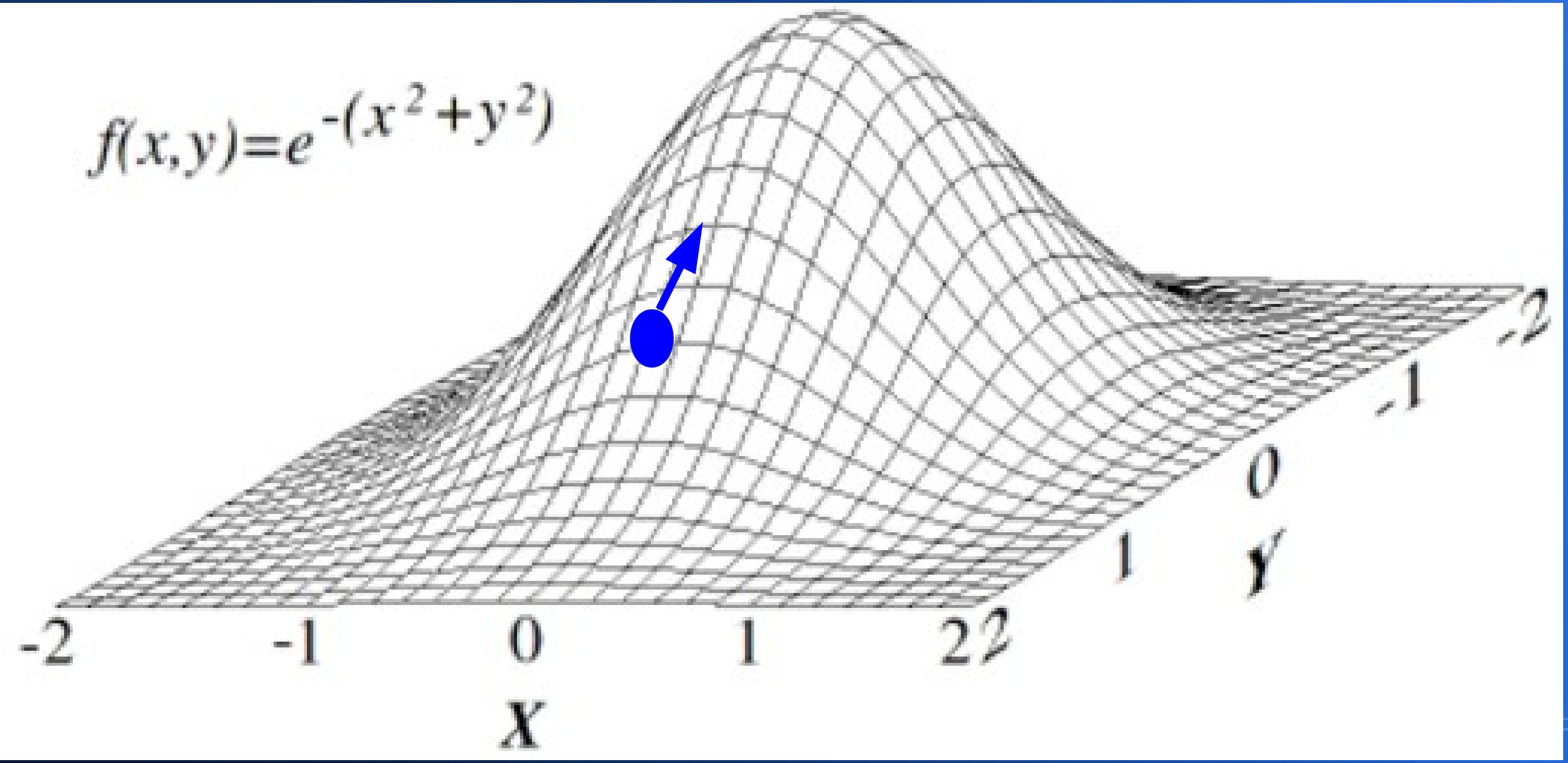
所謂的爬山演算法

- 就是讓程式一直往上爬的方法



你只要看到旁邊的點比現在的位置高

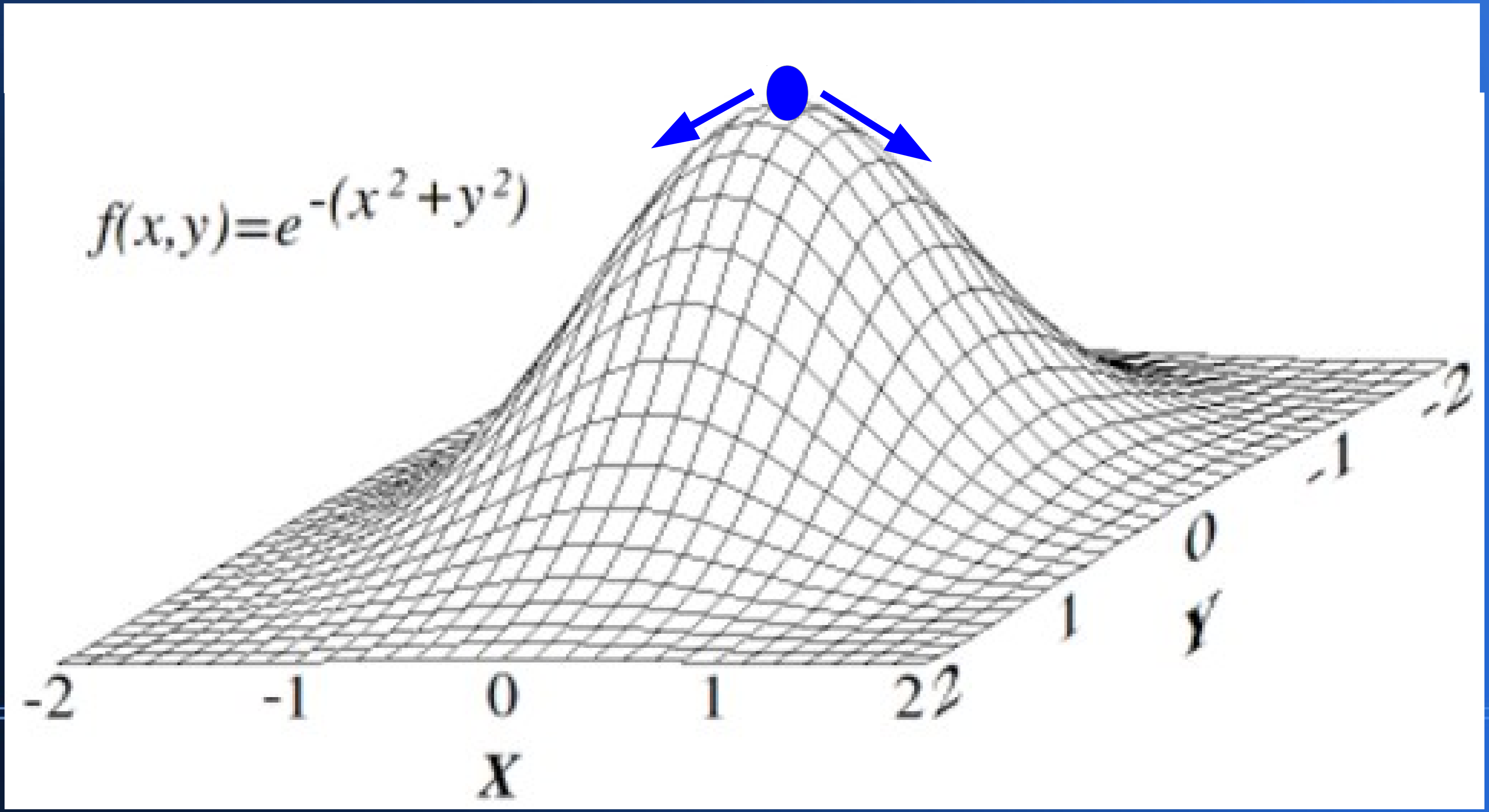
- 就往那邊爬 …



直到你發現

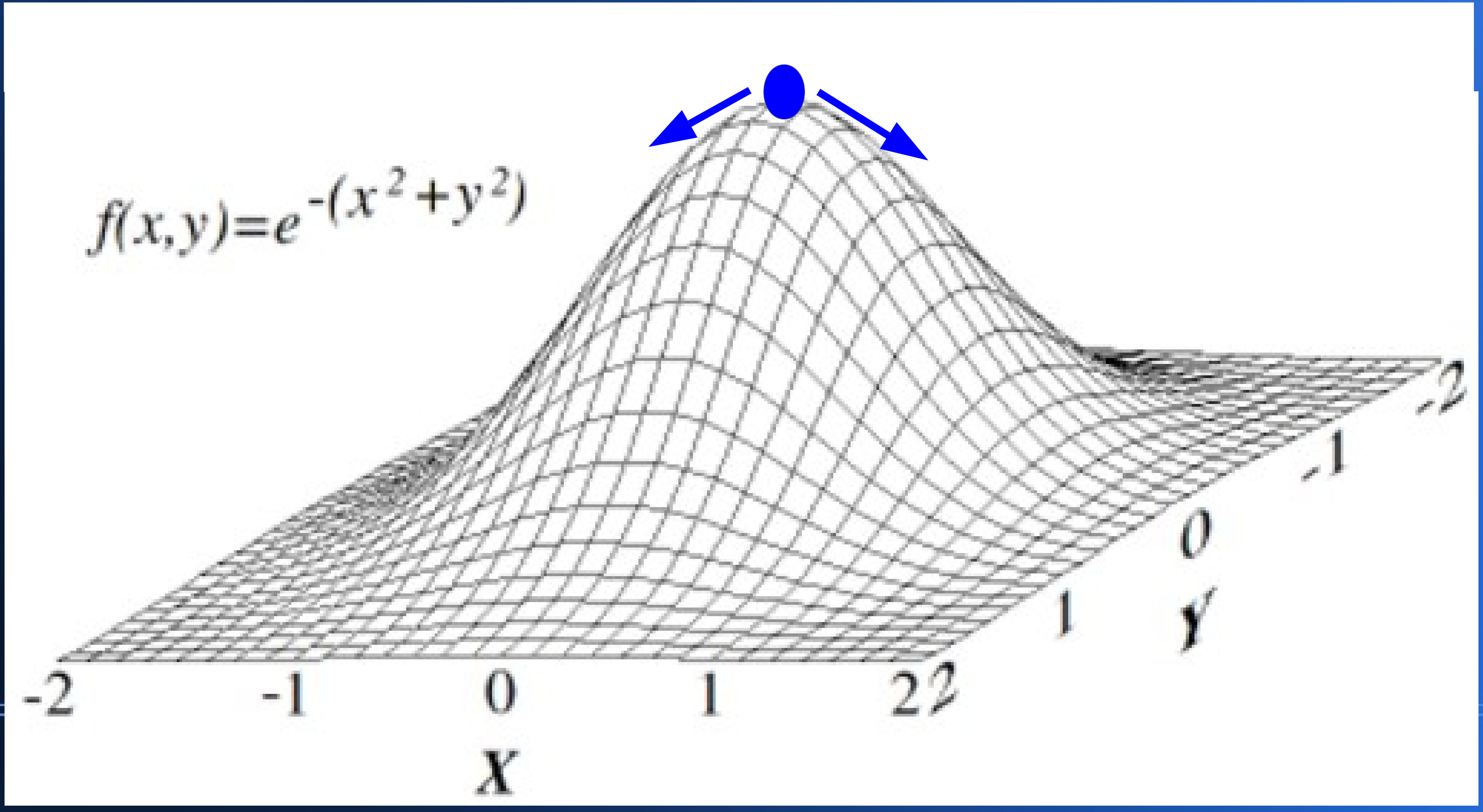
- 四面八方都比你低，再也爬不上去了

  此時代表你已經位於山頂上了



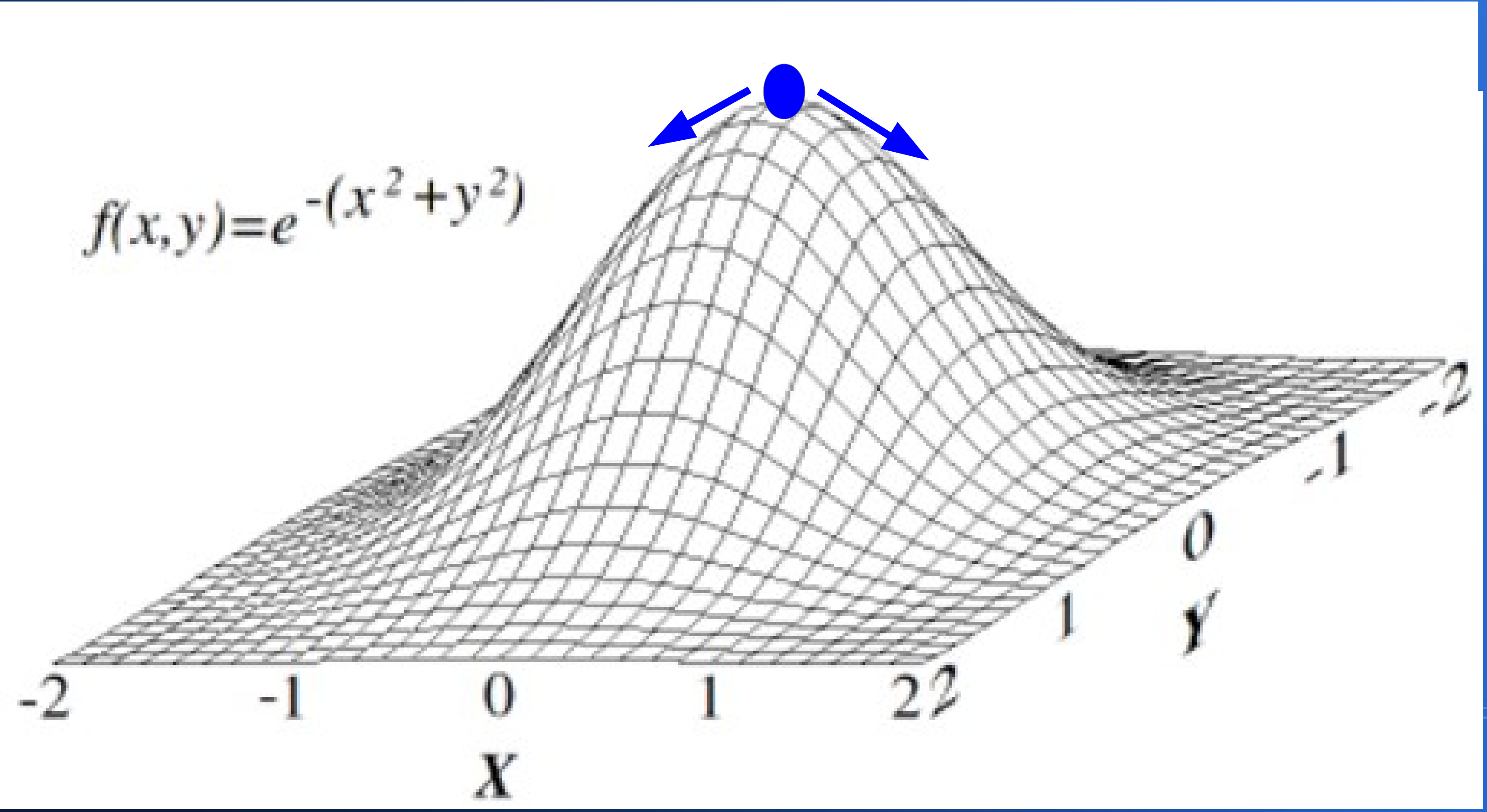
於是乎、你找到了一個不錯的點

- 雖然不是世界最高，但是也算本地最高點了！
- 這就是所謂的區域最佳解。



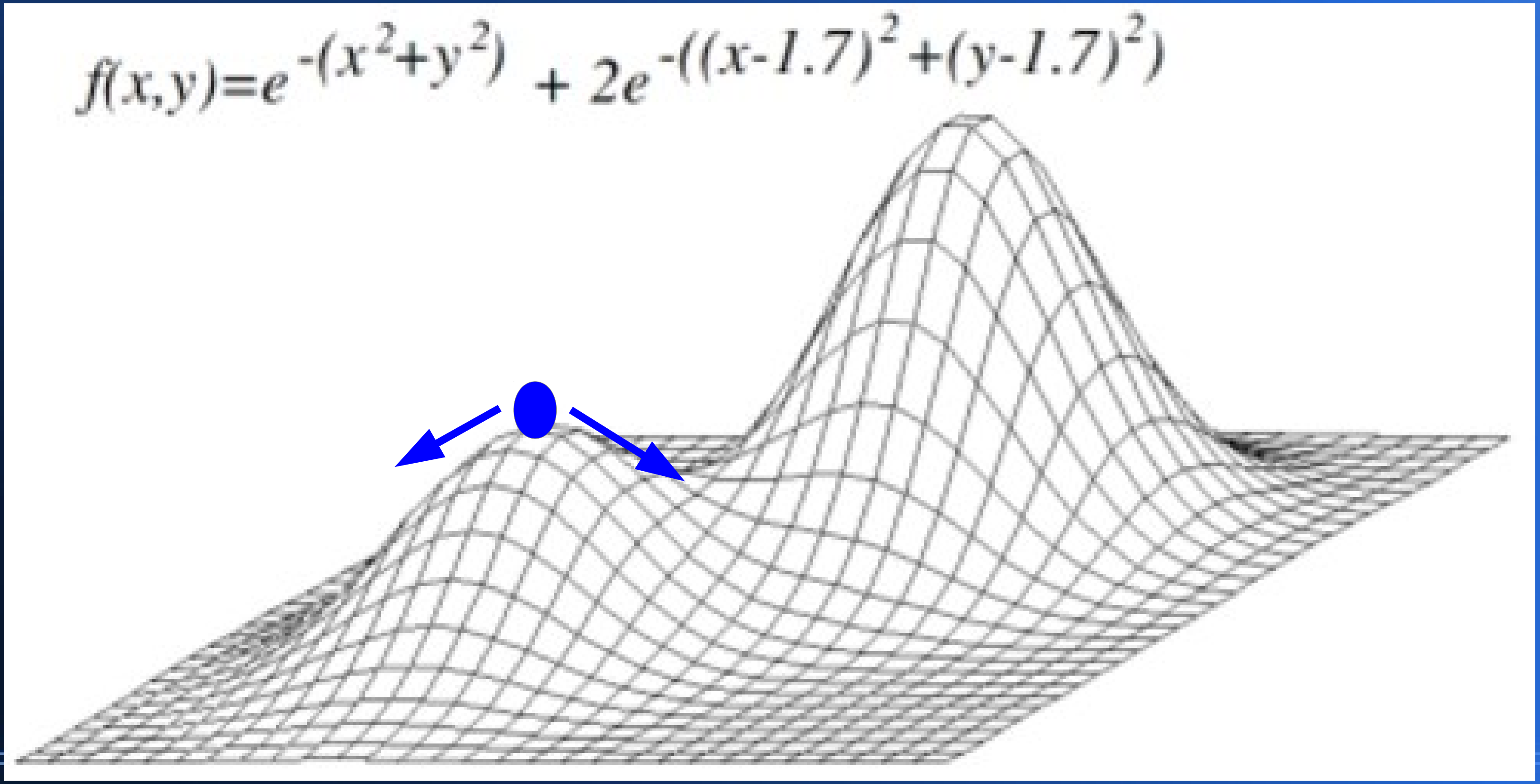
奇怪

- 圖中的那點不就是《世界最高點》嗎？



有可能、但是如果看遠一點

- 也有可能會像這樣

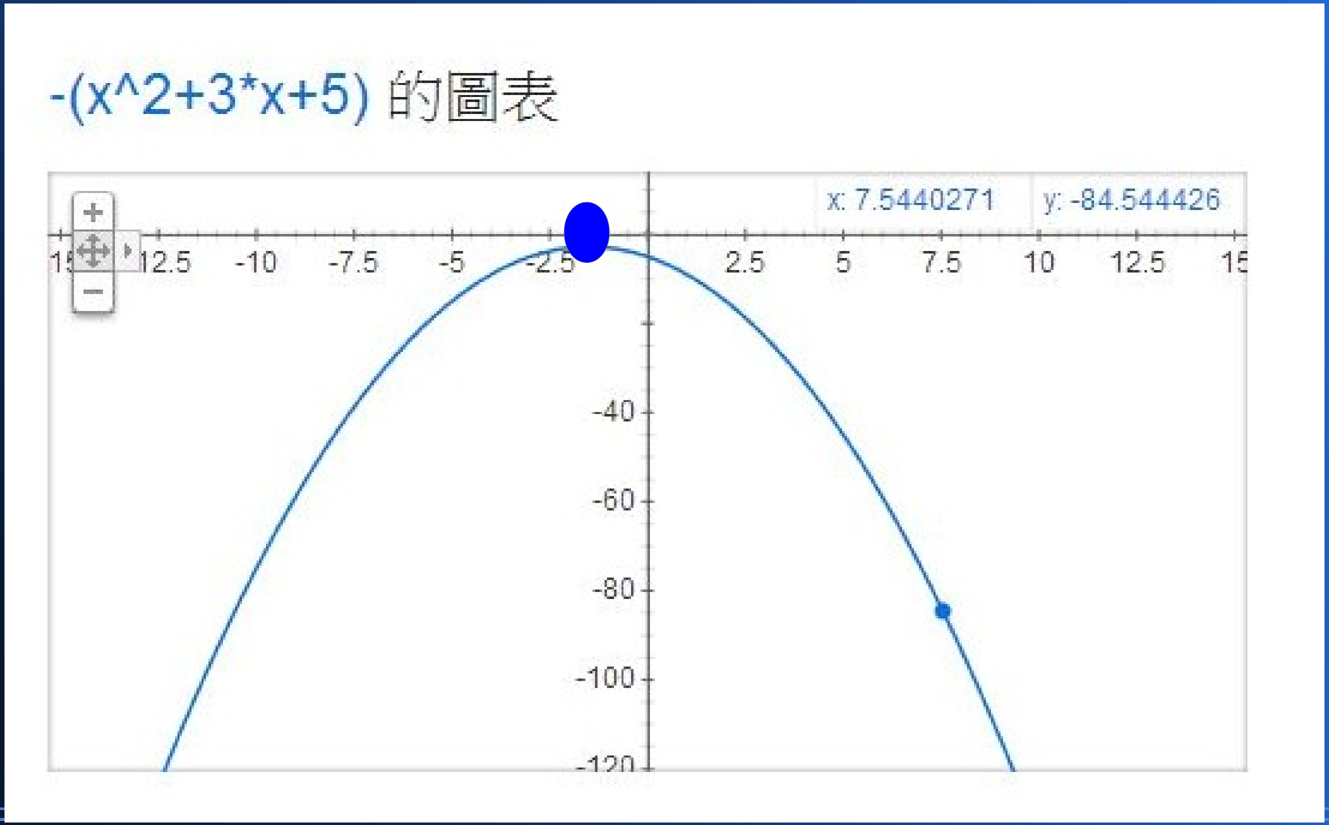


這就叫做

- 一山還有一山高囉！

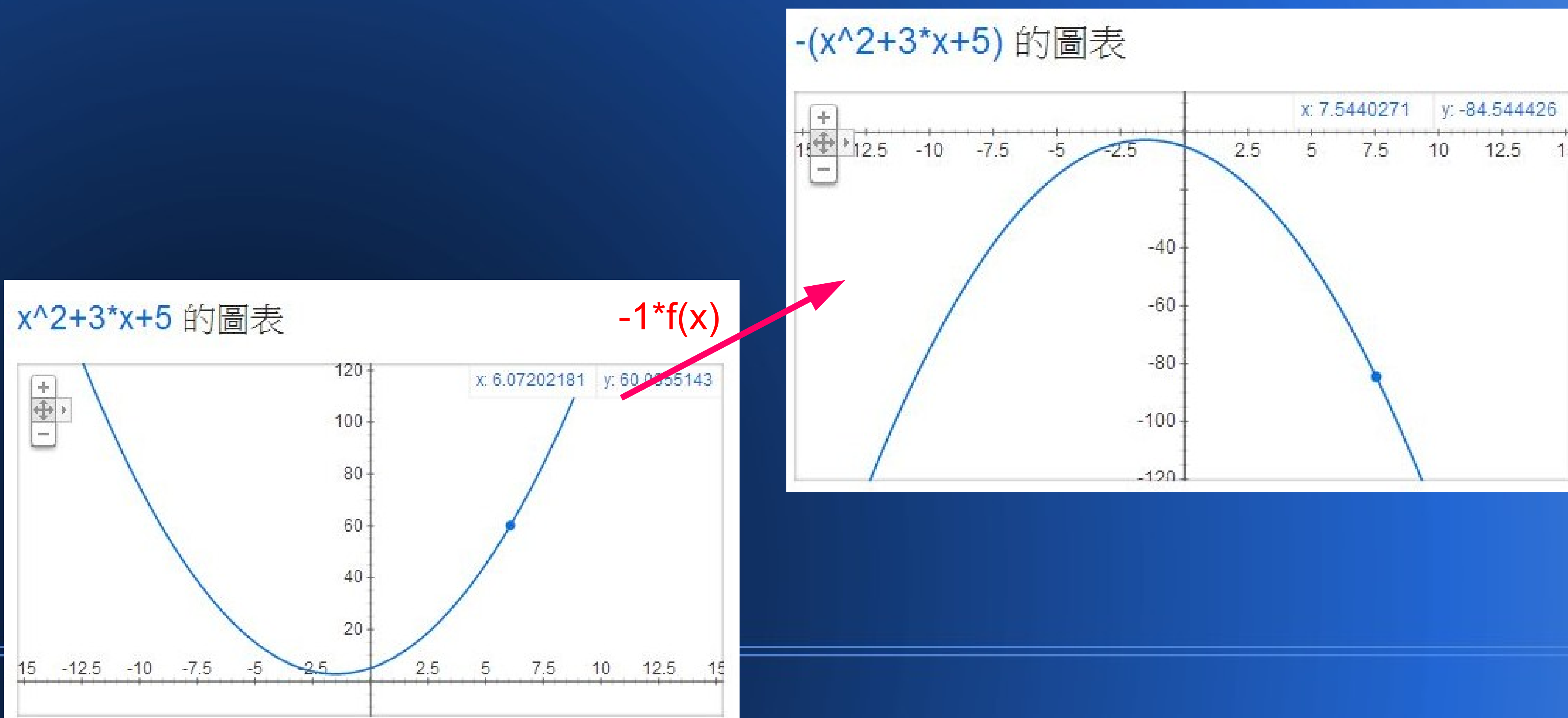
歐！對了

- 這種方法，不只可以用來找最高點！



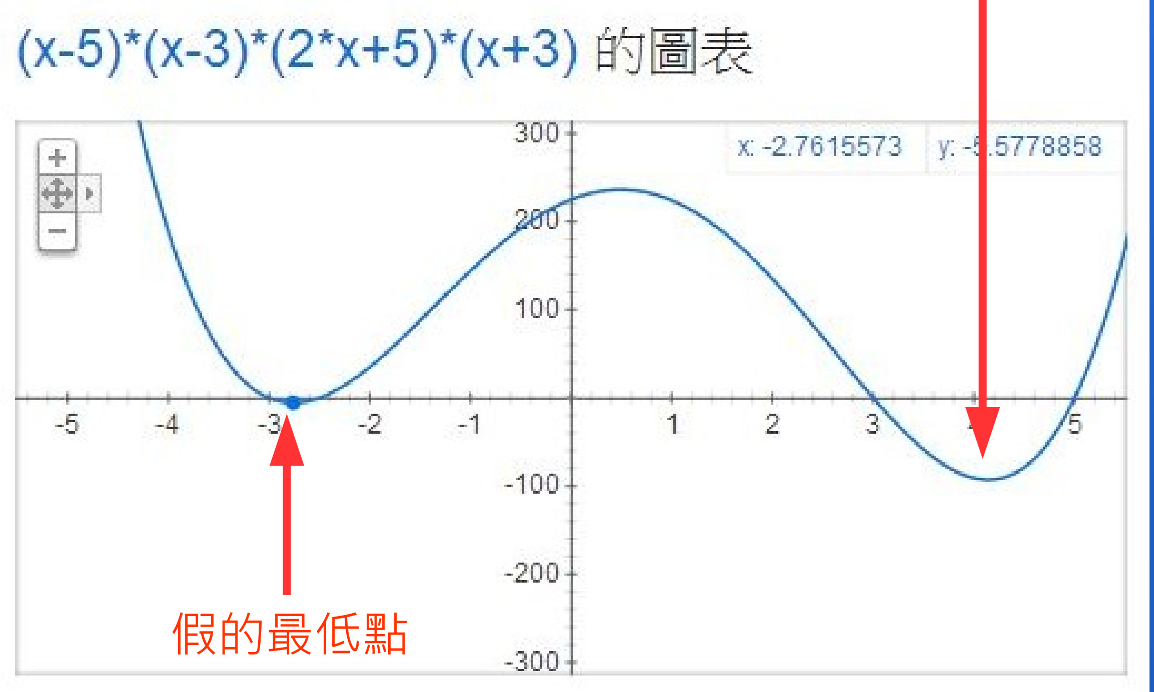
也可以用來找最低點

- 只要你將目標函數乘上 -1 就行了！



當然、這種方法

- 有時找不到真正的最高點（或最低點）！

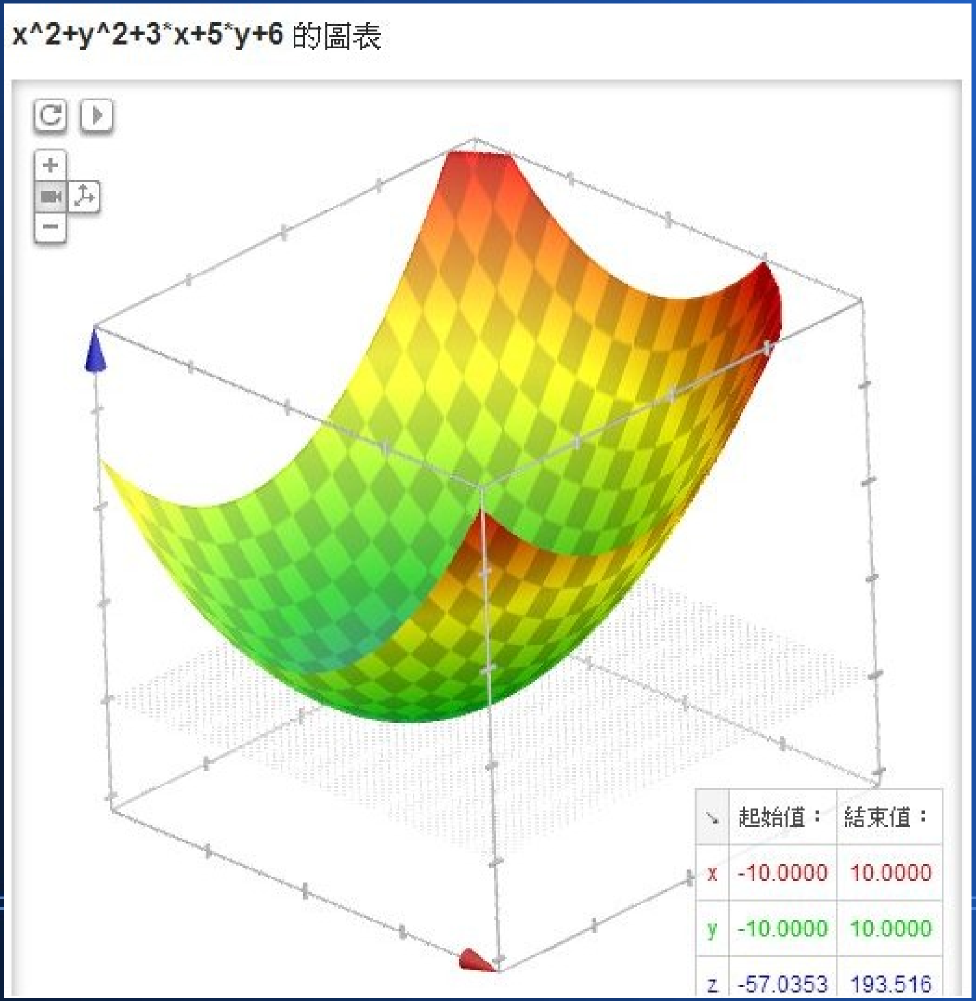


圖、兩個山谷的情況，如何找到最低的山谷呢？

不過如果多試幾次，從不同地方開始，那麼就有可能找到更好的區域低點，甚至是最低點！

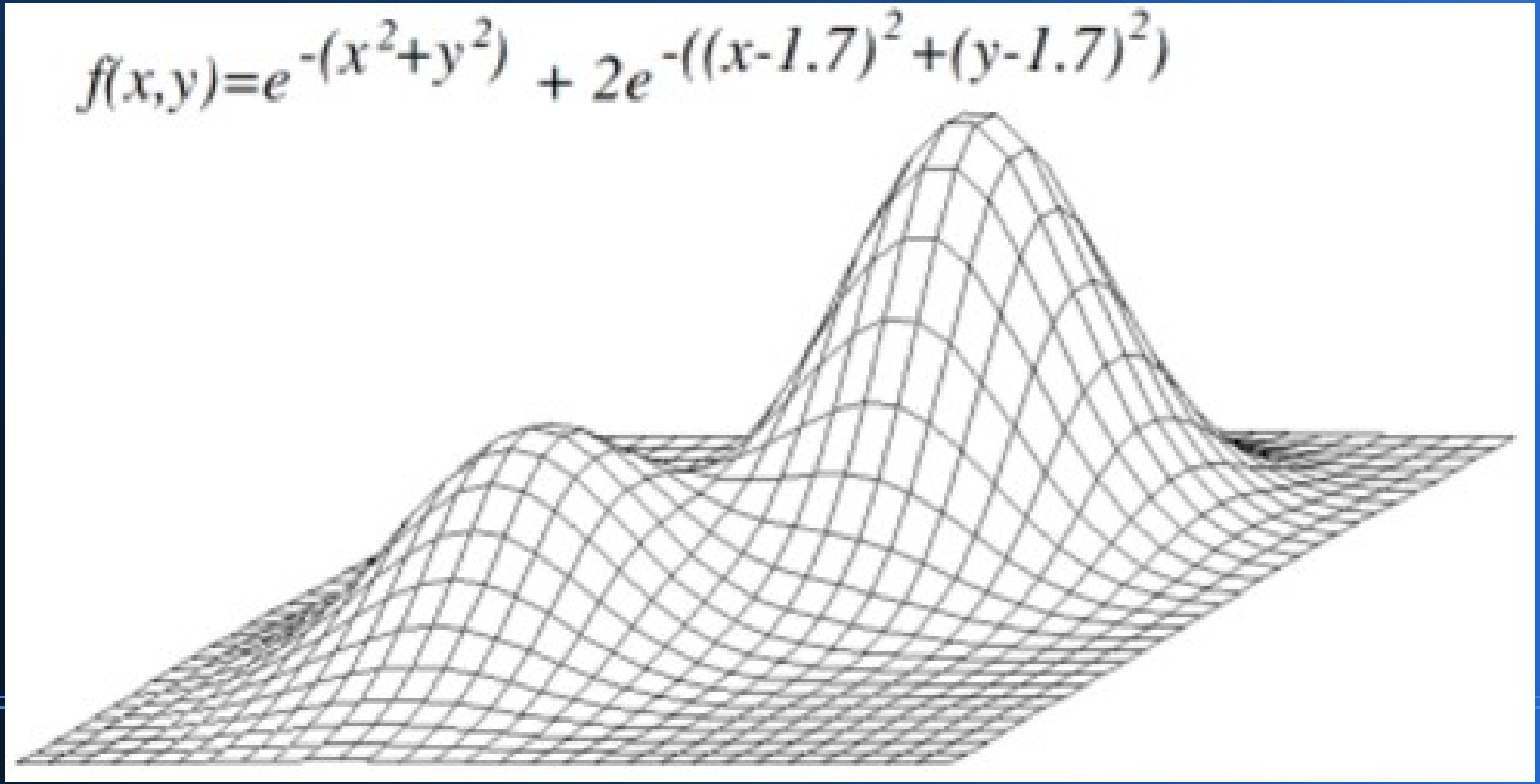
對於那種只有一個最低點的連續函數而言

- 不管函數有幾個維度，都可以輕易地找到最高點（或最低點）



但是如果有很多座山

- 那就不一定找得到最高點了



雖然如此

- 但是沒關係！
- 因為其他方法通常也沒辦法找到最高點，特別在函數非常複雜的時候！

而且

- 區域最佳解的表現，通常也已經很不錯了！

問題是

- 我要怎麼把爬山演算法寫成程式呢？

這個問題並不難

以下是爬山演算法的《算法》

```
Algorithm HillClimbing(f, x)
  x = 隨意設定一個解。
  while (x 有鄰居 x' 比 x 更高)
    x = x';
  end
  return x;
end
```

如果寫成 JavaScript 程式，會像這樣

```
var dx = 0.01;
function hillClimbing(f, x) {
  while (true) {
    console.log("f(%s)=%s", x.toFixed(4), f(x).toFixed(4));
    if (f(x+dx) >= f(x))
      x = x+dx;
    else if (f(x-dx) >= f(x))
      x = x-dx;
    else
    break;
  }
}

function f(x) { return -1*(x*x+3*x+5); }
hillClimbing(f, 0.0);
```

以下是該程式的執行結果

求解：$-(x^2+3x+5)$ 的最高點，也就是 x^2+3x+5 的最低點。

```
D:\Dropbox\Public\web\ai\code\optimize>node hillClimbingSimple
f(0.0000)=-5.0000
f(-0.0100)=-4.9701
f(-0.0200)=-4.9404
f(-0.0300)=-4.9109
f(-0.0400)=-4.8816
f(-0.0500)=-4.8525
...
f(-1.4500)=-2.7525
f(-1.4600)=-2.7516
f(-1.4700)=-2.7509
f(-1.4800)=-2.7504
f(-1.4900)=-2.7501
f(-1.5000)=-2.7500
```

如果把函數換掉，也可以順利執行。

```
function f(x) { return -1*Math.abs(x*x-4); }
```

那麼就可以用來求解 $|x^2 - 4|$ 的最低點，也就是尋找 4 的平方根，以下是執行結果：

```
D:\Dropbox\Public\web\ai\code\optimize>node hillClimbingSimple
f(0.0000)=-4.0000
f(0.0100)=-3.9999
f(0.0200)=-3.9996
f(0.0300)=-3.9991
f(0.0400)=-3.9984
f(0.0500)=-3.9975
...
f(1.9500)=-0.1975
f(1.9600)=-0.1584
f(1.9700)=-0.1191
f(1.9800)=-0.0796
f(1.9900)=-0.0399
f(2.0000)=-0.0000
```

不過上述程式只能處理單變數函數

- 對於多變數函數，必須處理多變數（維度）的選擇問題
- 可以用隨機的方式從 n 個變數中取出一個，進行微小調整後，看看是否能變得更好。
- 如果更好就接受，沒有更好就放棄！
- 當我們連續嘗試很多很多次（例如一萬次）都沒有變得更好時，就認為已經達到《山頂》，於是輸出解答！

以下是一個通用的
《爬山演算法程式架構》

```
var hillClimbing = function() {} // 爬山演算法的物件模版 (類別)

hillClimbing.prototype.run = function(s, maxGens, maxFails) { // 爬山演算法的主體函數
  console.log("s=%s", s); // 印出初始解
  var fails = 0;          // 失敗次數設為 0
  // 當代數 gen<maxGen，且連續失敗次數 fails < maxFails 時，就持續嘗試尋找更好的解。
  for (var gens=0; gens<maxGens && fails < maxFails; gens++) {
    var snew = s.neighbor();           // 取得鄰近的解
    var sheight = s.height();          // sheight=目前解的高度
    var nheight = snew.height();       // nheight=鄰近解的高度
    if (nheight >= sheight) {          // 如果鄰近解比目前解更好
      s = snew;                        //   就移動過去
      console.log("%d: %s", gens, s);  //   印出新的解
      fails = 0;                       //   移動成功，將連續失敗次數歸零
    } else                             // 否則
      fails++;                         //   將連續失敗次數加一
  }
  console.log("solution: %s", s);     // 印出最後找到的那個解
  return s;                            //   然後傳回。
}
```

還有通用的《解物件》(Solution)之定義

```
var Solution = function(v) { // 解答的物件模版 (類別)
  this.v = v;              // 參數 v 為解答的資料結構
}

Solution.prototype.step = 0.01;          // 每一小步預設走的距離

Solution.prototype.height = function() { // 爬山演算法的高度函數
  return -1*this.energy();              // 高度 = -1 * 能量
}
```

通常若是尋找高點，我們會用高度 height() 代表，
但若尋找低點，我們會用能量 energy() 代表！

有了這個爬山演算法的通用程式架構

- 我們就可以套用在任何需要優化的問題上
- 只要你能定義出《高度》或《能量函數》就行了！

以下是尋找平方根的範例

```
var Solution = require("./solution");          // 引入解答類別

Solution.prototype.neighbor = function() {     // 單變數解答的鄰居函數。
  var x = this.v, dx=this.step;                // x:解答 , dx : 移動步伐大小
  var xnew = (Math.random() > 0.5)?x+dx:x-dx;  // 用亂數決定向左或向右移動
  return new Solution(xnew);                   // 建立新解答並傳回。
}

Solution.prototype.energy = function() {       // 能量函數
  var x = this.v;                              // x:解答
  return Math.abs(x*x-4);                      // 能量函數為 |x^2-4|
}

Solution.prototype.toString = function() {     // 將解答轉為字串，以供印出觀察。
  return "energy("+this.v.toFixed(3)+")="+this.energy().toFixed(3);
}

module.exports = Solution;                     // 將解答類別匯出。
```

```
var hillClimbing = require("./hillClimbing");       // 引入爬山演算法類別
var solutionNumber = require("./solutionNumber");   // 引入平方根解答類別

var hc = new hillClimbing();                        // 建立爬山演算法物件
// 執行爬山演算法 (從「解答=0.0」開始尋找, 最多十萬代、失敗一千次就跳出。
hc.run(new solutionNumber(0.0), 100000, 1000);
```

```
$ node hillClimbingNumber.js
s=energy(0.000)=4.000
0: energy(-0.010)=4.000
2: energy(-0.020)=4.000
3: energy(-0.030)=3.999
10: energy(-0.040)=3.998
12: energy(-0.050)=3.998
....
366: energy(-1.910)=0.352
371: energy(-1.920)=0.314
375: energy(-1.930)=0.275
380: energy(-1.940)=0.236
382: energy(-1.950)=0.197
388: energy(-1.960)=0.158
389: energy(-1.970)=0.119
391: energy(-1.980)=0.080
392: energy(-1.990)=0.040
394: energy(-2.000)=0.000
solution: energy(-2.000)=0.000
```

當然

- 這個程式也可以用來處理更複雜的問題
- 只要你能寫出《能量函數》 energy() 與《鄰居函數》 neighbor() 就行了！

完整的程式位於 http://ccc.nqu.edu.tw/db/ai/hillClimbingFramework.html

其實、在大部分的情況下

- 爬山演算法就已經夠好用了！

不過

- 為了克服《超小山丘》的那種問題，你也可以做一點點修改，讓爬山演算法可以有機會離開那個《無言的山丘》。
- 這個改良版就稱為《模擬退火法》。

為何稱為

- 模擬退火法呢？

這是因為有人發現

- 打鐵煉鋼的時候，如果溫度降得太快，煉出來的鐵就會脆脆的不堅固。
- 要能煉得好的秘訣是溫度要慢慢降，然後一直敲一直打，這樣打出來的鐵才會夠堅固，成為寶刀或寶劍

如果模仿這種想法

- 在爬山演算法的相反版，也就是《下山演算法》當中，用《能量》的概念來取代《高度》，那麼整個爬山的過程就反過來變成在尋找能量的最低點。
- 這時如果加入溫度的概念，讓這些鐵原子在溫度高的時候可以比較自由的亂動，等到溫度慢慢降低之後才逐漸固定下來，這樣打出來的鐵，原子的排列就會比較整齊，也就會比較堅固。

模仿這種鐵原子慢慢降溫穩定過程的機器稱為《波茲曼機》

- 而模仿單一鐵原子震動或移動，然後慢慢隨溫度下降而減少變動，逐漸固定下來的行為，就是《模擬退火法了》

以下是《模擬退火法》的演算法

```
Algorithm SimulatedAnnealing(s)
  while (溫度還不夠低，或還可以找到比 s 更好的解 s’ 的時候)
    根據能量差與溫度，用機率的方式決定是否要移動到新解 s’。
    將溫度降低一些
  end
end
```

在上述演算法中，所謂的機率的方式，是採用 $\exp(\frac{e-e'}{T})$ 這個機率公式，去判斷是否要從 s 移動到 s'，其中 e 是 s 的能量值，而 e' 是 s' 的能量值。

如果寫成程式，就會像這樣

```
simulatedAnnealing.prototype.run = function(s, maxGens) { // 模擬退火法的主要函數
  var sbest = s;                                      // sbest:到目前為止的最佳解
  var ebest = s.energy();                             // ebest:到目前為止的最低能量
  var T     = 100;                                    // 從 100 度開始降溫
  for (var gens=0; gens<maxGens; gens++) {            // 迴圈，最多作 maxGens 這麼多代。
    var snew = s.neighbor();                          // 取得鄰居解
    var e    = s.energy();                            // e    : 目前解的能量
    var enew = snew.energy();                         // enew : 鄰居解的能量
    T  = T * 0.999;                                   // 每次降低一些溫度
    if (this.P(e, enew, T) > Math.random()) {         // 根據溫度與能量差擲骰子，若通過
      s = snew;                                       // 則移動到新的鄰居解
      console.log("%d T=%s %s", gens, T.toFixed(3), s.toString()); // 印出觀察
    }
    if (enew < ebest) {                               // 如果新解的能量比最佳解好，則更新最佳解。
      sbest = snew;
      ebest = enew;
    }
  }
  console.log("solution: %s", sbest.toString()); // 印出最佳解
  return sbest;                                   // 傳回最佳解
}
```

當我們用上述《模擬退火法》模組

- 尋找函數 $x^2 + 3y^2 + z^2 - 4x - 3y - 5z + 8$ 的最低點時，可以寫出下列主程式。

```
var simulatedAnnealing = require("./simulatedAnnealing"); // 引入模擬退火法類別
var solutionArray = require("./solutionArray");          // 引入多變數解答類別 (x
^2+3y^2+z^2-4x-3y-5z+8)

var sa = new simulatedAnnealing();                        // 建立模擬退火法物件
// 執行模擬退火法 (從「解答(x,y,z)=(1,1,1)」開始尋找, 最多執行 2 萬代。
sa.run(new solutionArray([1,1,1]), 20000);
```

完整的程式位於： http://ccc.nqu.edu.tw/db/ai/simulatedAnnealing.html

其執行結果如下

- 您會發現解答(x=2, y=0.5, z=2.5)

  正是函數 $x^2+3y^2+z^2-4x-3y-5z+8$

  的最低點！

- 其能量值為 -3 ，

  也就是函數 f(x, y, z) 的值。

```
0 T=99.900 energy( 1.000   1.000   0.990 )=1.030
1 T=99.800 energy( 1.000   0.990   0.990 )=1.000
2 T=99.700 energy( 1.000   0.980   0.990 )=0.971
3 T=99.601 energy( 0.990   0.980   0.990 )=0.991
4 T=99.501 energy( 0.990   0.990   0.990 )=1.021
5 T=99.401 energy( 1.000   0.990   0.990 )=1.000
6 T=99.302 energy( 1.000   0.990   1.000 )=0.970
...
5985 T=0.251 energy( 0.870   1.260   1.770 )=0.543
5986 T=0.250 energy( 0.870   1.250   1.770 )=0.497
```

```
5989 T=0.250 energy( 0.870   1.250   1.760 )=0.512
5990 T=0.249 energy( 0.860   1.250   1.760 )=0.535
...
15036 T=0.000 energy( 2.000   0.500   2.510 )=-3.000
15038 T=0.000 energy( 2.000   0.500   2.500 )=-3.000
15173 T=0.000 energy( 2.010   0.500   2.500 )=-3.000
15174 T=0.000 energy( 2.000   0.500   2.500 )=-3.000
15261 T=0.000 energy( 2.000   0.500   2.490 )=-3.000
15265 T=0.000 energy( 2.000   0.500   2.500 )=-3.000
solution: energy( 2.000   0.500   2.500 )=-3.000
```

這種優化方法

- 看來好像只能解單一函數的優化
- 但事實上，爬山演算法和模擬退火法
  - 連《方程組的優化》也通常能解！

舉例而言、如果你想解下列方程組

$$\begin{cases} 2x + y = 8 \\ x + y = 6 \end{cases}$$

- 除了用國中時所學的消去法之外，也可以用上述的《爬山演算法》或《模擬退火法》來解。
- 只要把能量函數設為下列函數就行了。
  - $(2x+y-8)^2 + (x+y-6)^2$

同樣的、這種方法也能求解更複雜的方程組

- 包含線性方程組

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \quad &\quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m. \end{aligned}$$

$$\begin{aligned} x + 3y - 2z &= 5 \\ 3x + 5y + 6z &= 7 \\ 2x + 4y + 3z &= 8 \end{aligned}$$

- 與非線性方程組

$$\begin{cases} s^3 + 4c^3 - 3c &= 0 \\ s^2 + c^2 - 1 &= 0 \end{cases}$$

$$\begin{cases} t^3 - t = 0 \\ x = \frac{t^2+2t-1}{3t^2-1} \\ y = \frac{t^2-2t-1}{3t^2-1} \end{cases}$$

甚至也能用於求解《微分方程組》

$$P_1(x)Q_1(y) + P_2(x)Q_2(y)\,\frac{dy}{dx} = 0$$

$$P_1(x)Q_1(y)\,dx + P_2(x)Q_2(y)\,dy = 0$$

- 只是不一定有辦法找到完全符合條件的解答而已！

當然、尋找這些函數的最佳解或方程組的解答

- 感覺並不太像是《人工智慧》的問題
- 反而比較像是《數值分析》或《科學計算》的問題！

但是、天底下的知識

- 幾乎都是相關聯的
- 而且常常能夠一通百通！
- 只要學會一招，通常就夠用了。

舉例而言

- 對於演算法中的《最小擴展樹》或《旅行推銷員》等問題，其實也都是在找某些數值最小
- 因此當然也能用《爬山演算法》或《模擬退火法》來求解！
- 只是在這些問題上對於《鄰居》的定義不一樣，還有《能量函數》也有所不同而已！

甚至

- 對於《手寫辨識》等人工智慧上的問題，只要你能定義出《兩個手寫字相似程度》的數學函數。
- 那麼就可以用《爬山演算法》來求解此類問題，只是找到的解不一定會是最好的而已！

甚至、對於《影像和語音辨識》問題

- 只要能定義兩個影像或語音的相似程度，也能夠用《爬山演算法來求解》找出最相似的語音（通常就是正確解答）。
- 不過這種相似函數很難直接用人腦定義出來，通常必須要先進行某些特徵抽取後才有辦法計算相似度。

而對於《機器翻譯》的問題

- 只要你能定義《中文語句》和《英文語句》之間的意義相似度
- 那麼給定某個《英文語句》，你只要能從《所有可能的中文語句》裏，挑出與該英文語句意義最相似的一句話出來，這樣就完成了翻譯動作。
- 只是《中文語句通常有無限多》，不過我們可以先用《逐字對譯》的方式，取出所有可能的候選中文字詞，然後再進行排列組合，這樣就不會因為《中文語句無限多》而無法列舉了！

還有電腦下棋的問題

- 其實也只是在尋找一個有效評估盤面好壞的《盤面評估函數》
- 然後每次下子時，都是從所有可能的下法當中，尋找對方最不利的下法，讓對方難以得勝，讓我方盡可能獲勝而已！

不過

- 雖然人工智慧的問題都是優化問題

但是每一種方法的適用性卻有所不同

有些方法在某些問題上會表現得比較好

- 因此我們必須選擇解決該問題的適當方法。

除了爬山演算法

- 還有模擬退火法之外

還有很多其他方法

像是

- 模仿鳥類的《粒子群演算法》
- 模仿螞蟻的《蟻群演算法》
- 模仿 DNA 兩性生殖的《遺傳演算法》
- 純粹用亂數統計的《蒙地卡羅法》

以上這些從大自然模仿而來的方法

- 通常被歸類為《軟計算》方法
- 因為這些方法可以到處套用，很有彈性，所以很軟 …

這些軟計算方法

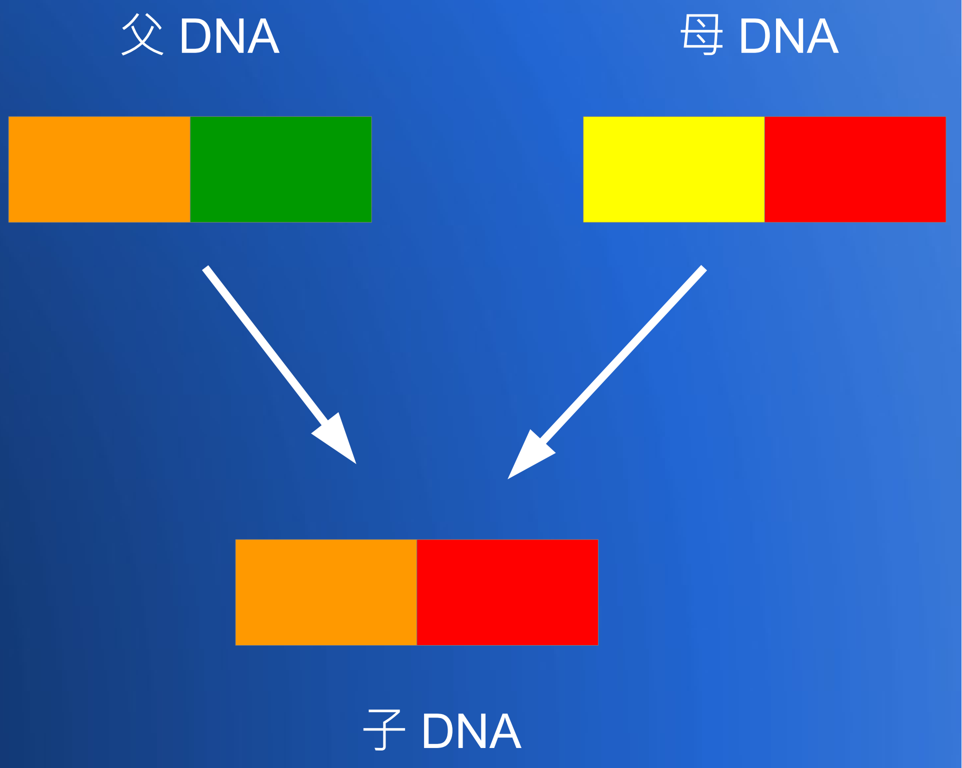
- 和爬山演算法之間的差異

  除了粒子比較多之外

  通常《鄰居》的定義也不太一樣

像是遺傳演算法 GA

- 下一代的 DNA 就是由父母交配後的結果
- 其鄰居的搜索空間很大
- GA 適用在兩個好的父母會生出好的子女之問題上，也就是要有《好的片段會組成好的整體》之特性

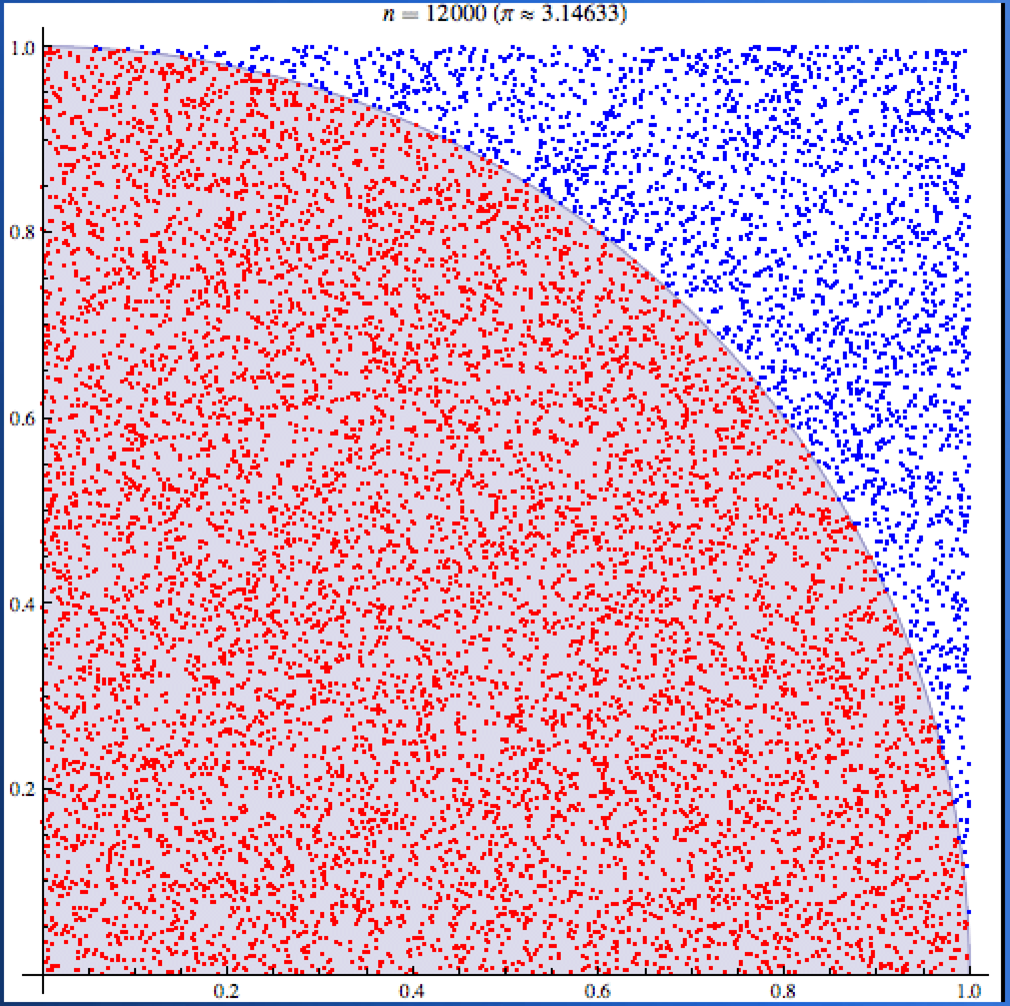


而蒙地卡羅法

- 則是用亂數統計來估計某函數的一種方法

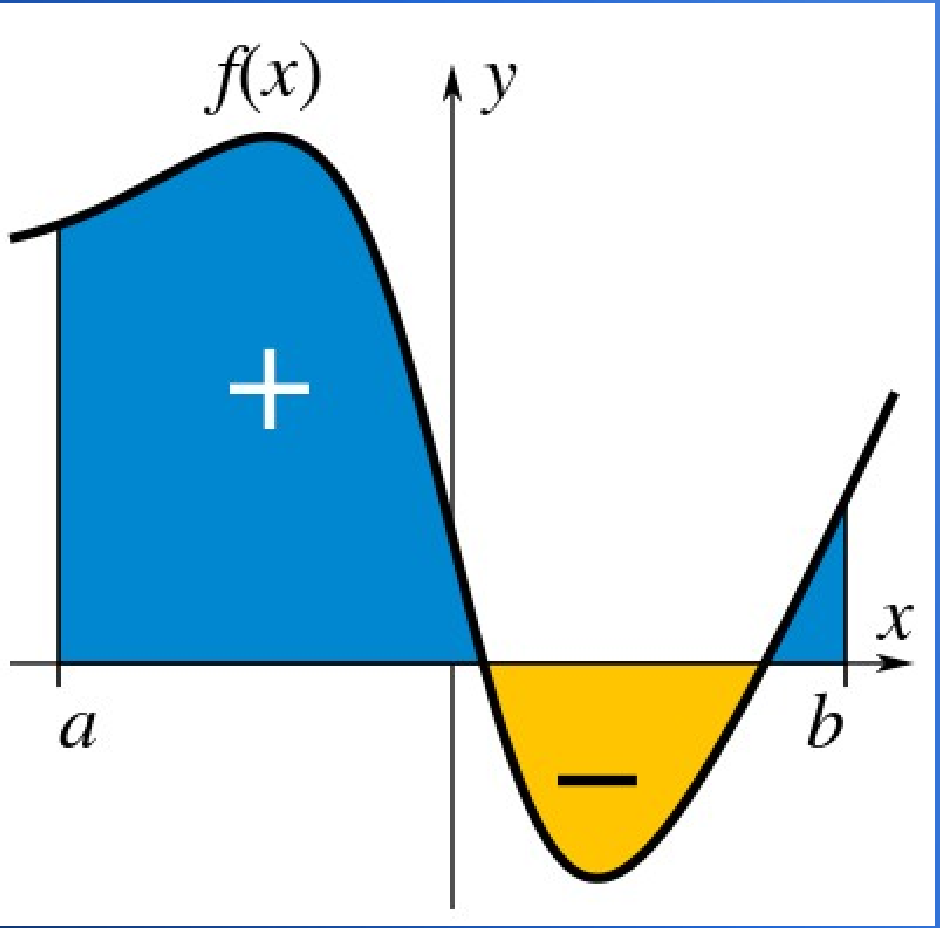
例如你想計算圓面積或 π 值

- 那麼可以用大量的亂數，經由統計計算出圓與方形的比例，進而計算圓面積。



當然、蒙地卡羅法

- 也可以用來計算微積分中的函數面積

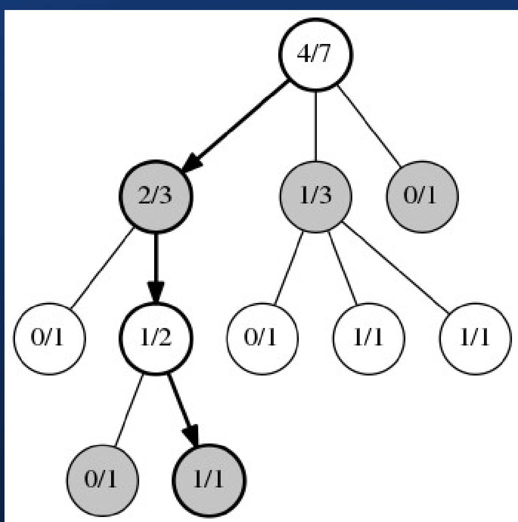


甚至在下電腦圍棋的 AlphaGo 程式上

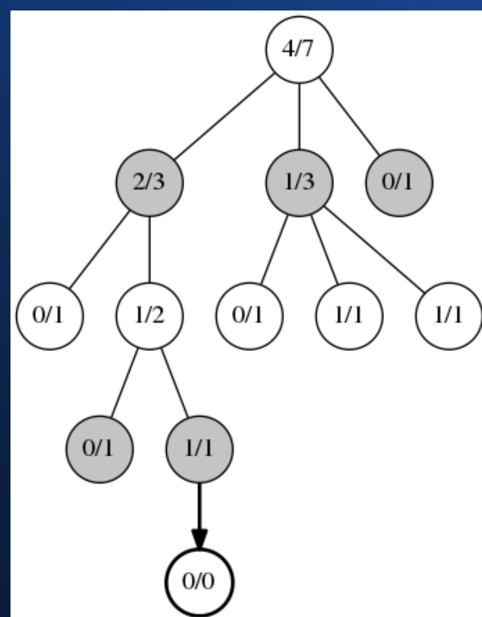
- 也用了蒙地卡羅樹狀搜尋法來尋找下一子圍棋的好下法

以下是《蒙地卡羅對局搜尋法》(MCTS)的一個搜尋擴展範例

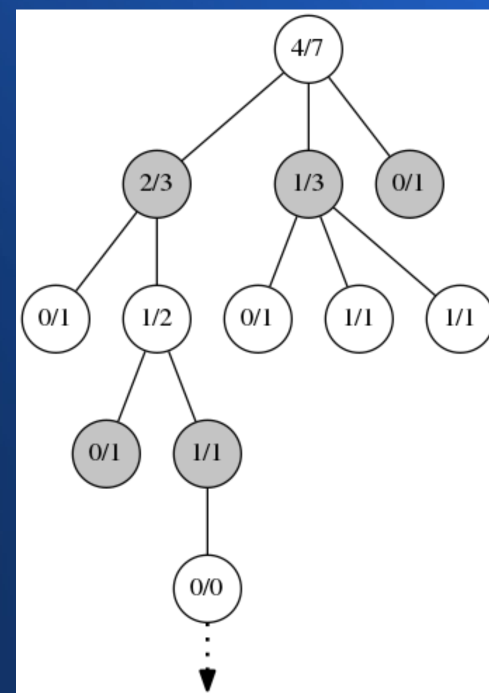
1. 選擇上界 UCB 最高的一條路直到末端節點



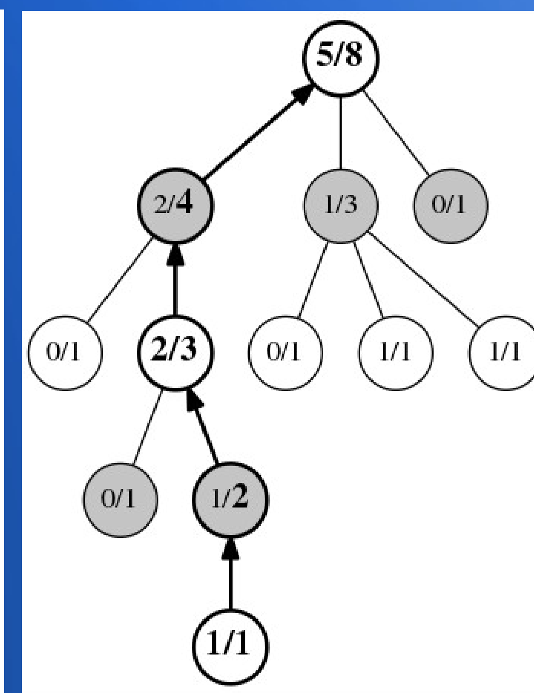
2. 對該末端節點進行探索 (隨機對下，自我對局)



3. 透過自我對局，直到得出本次對局的勝負結果



4. 用這次的對局結果，更新路徑上的勝負統計次數！

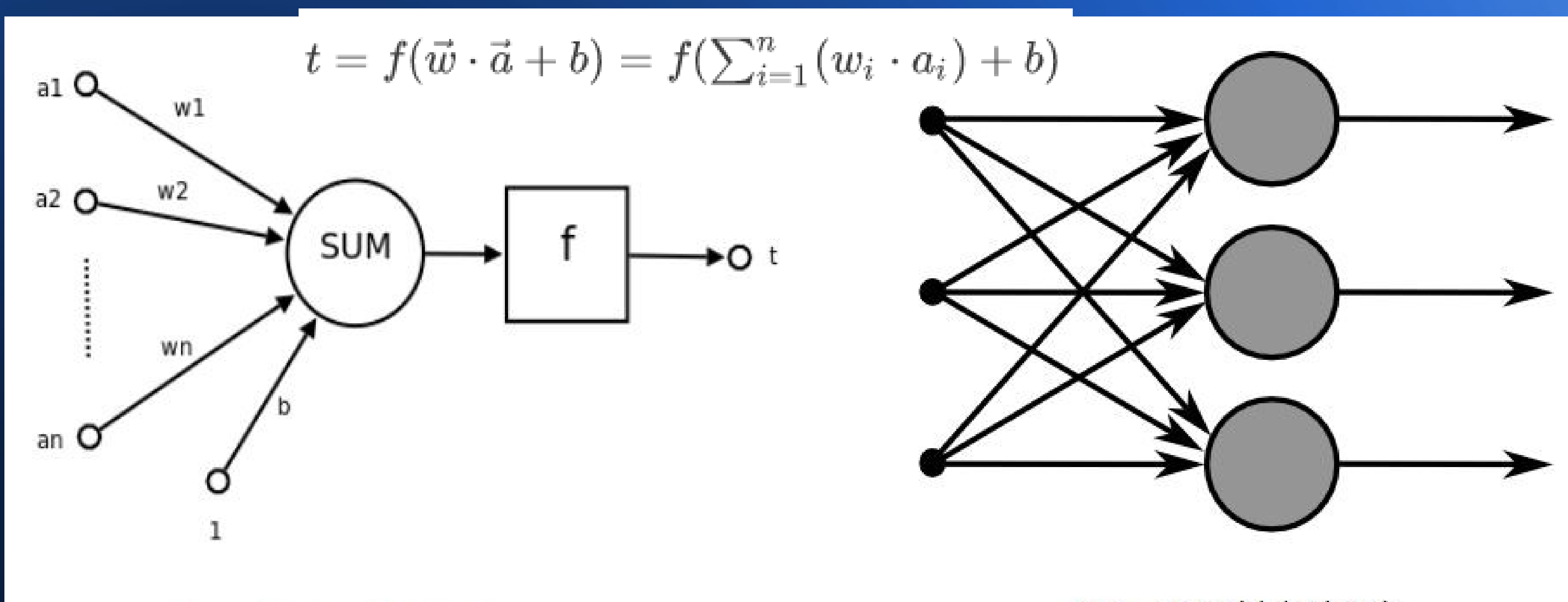


說明：上圖中白色節點為我方下子時的《得勝次數 / 總次數》之統計數據，

灰色的為對方下子的數據，本次自我對局結束後，得勝次數與總次數都會更新！

另外、人工智慧領域裡常用的《神經網路》學習模型

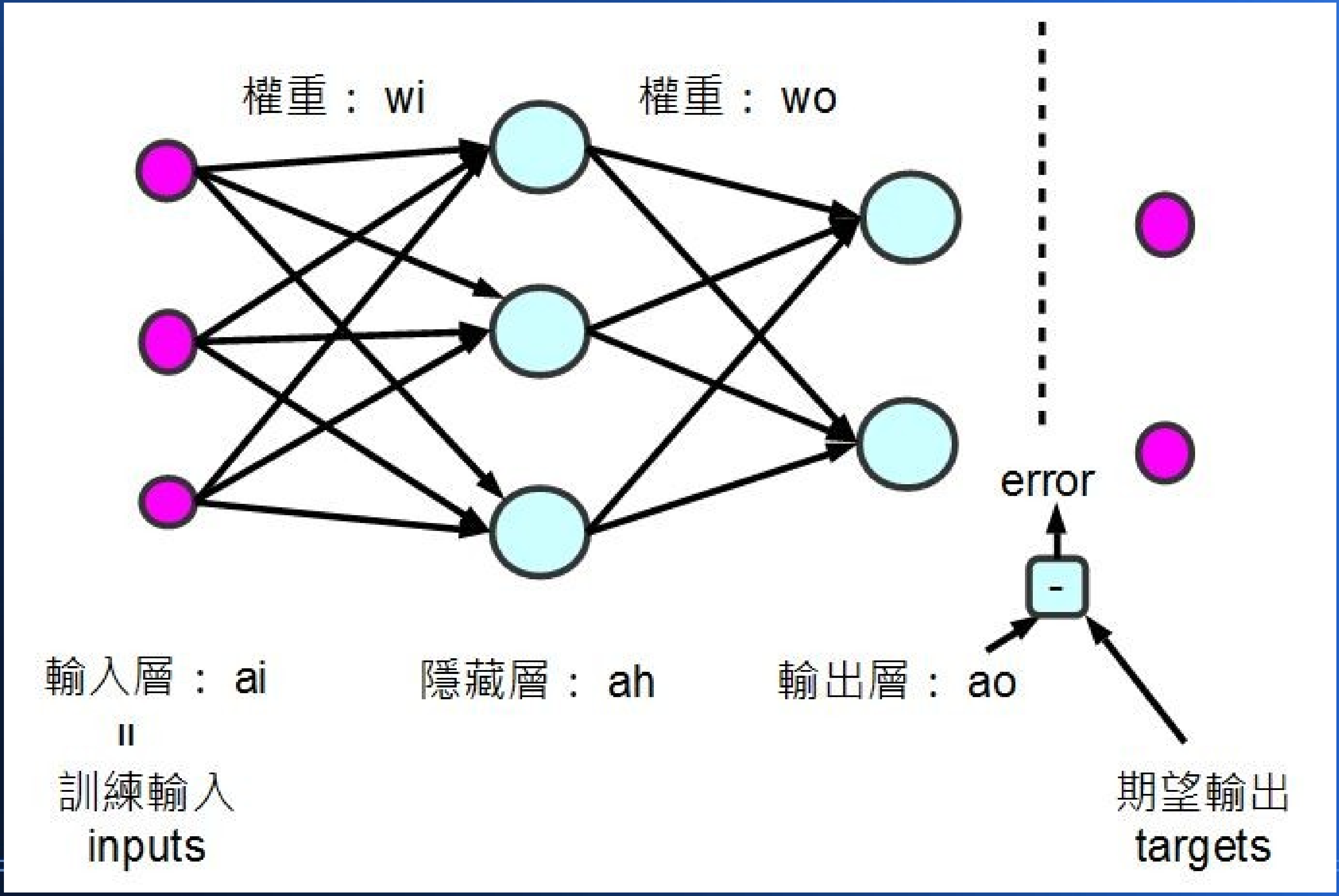
- 也只是在優化《錯誤率》這個《能量函數》而已。



(a) 單一神經元的模型

(b) 單層神經網路

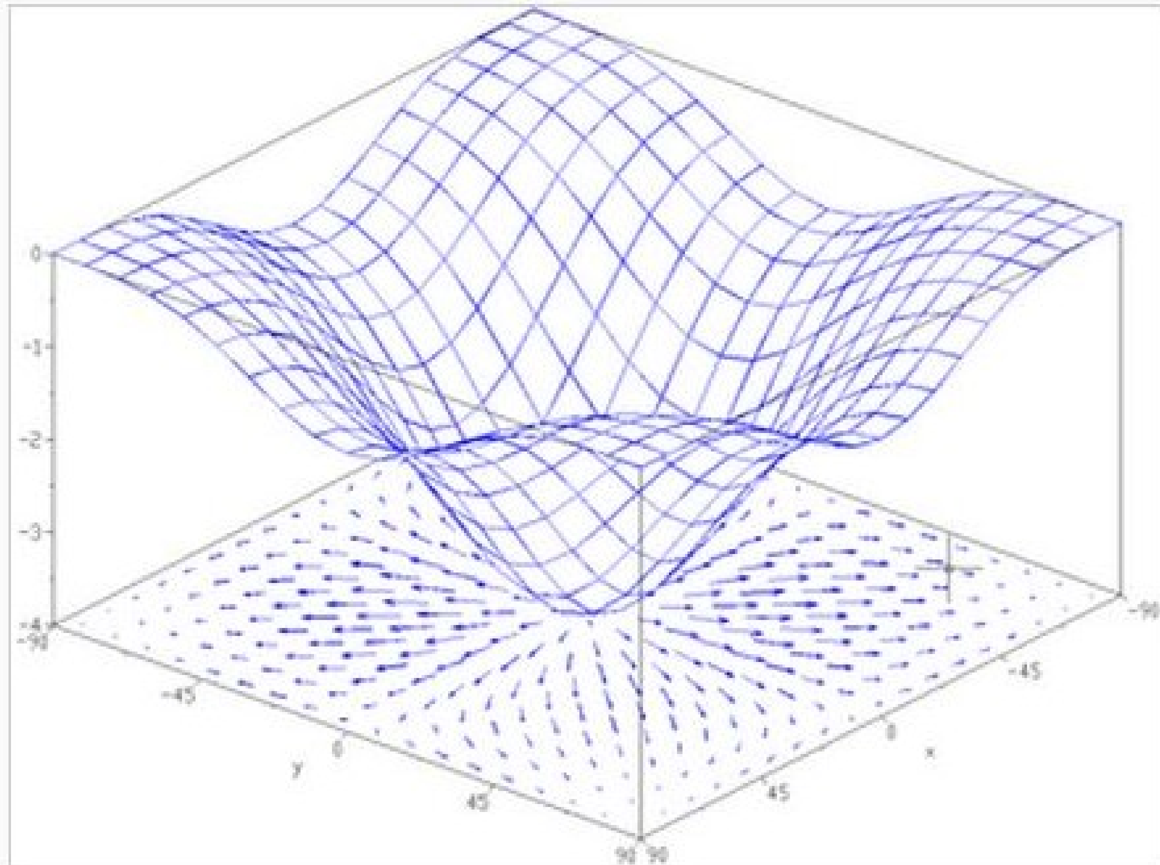
而神經網路中著名的《反傳遞學習算法》



也只是用《梯度下降法》

- 在尋找《降低錯誤能量》的神經權重之組合而已

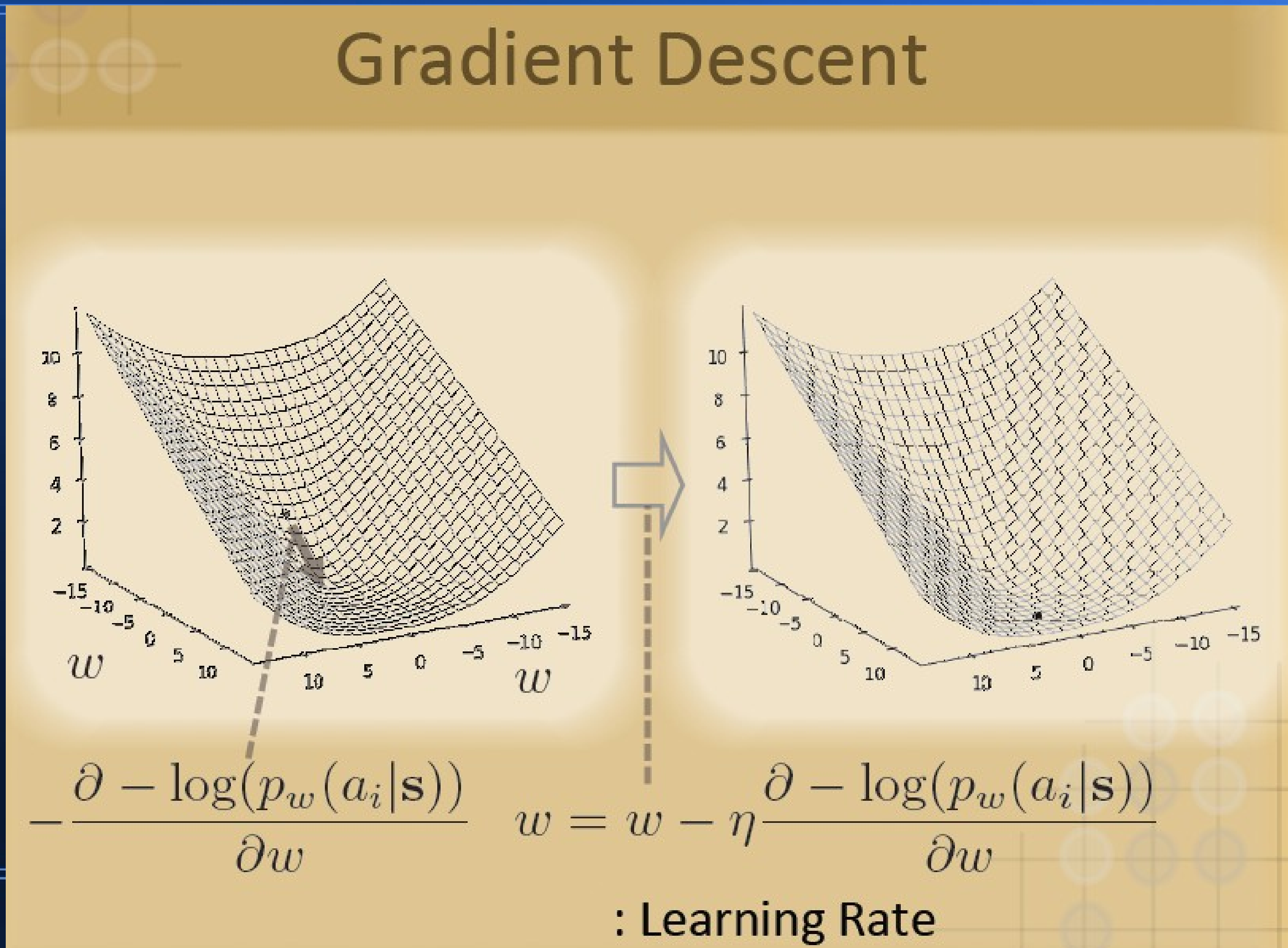
$$\nabla f = \frac{\partial f}{\partial x_1}\overrightarrow{e_1} + \cdots + \frac{\partial f}{\partial x_n}\overrightarrow{e_n}$$



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.

圖、曲面與每一點的梯度向量

那個梯度，就是斜率最大的方向指引



最近很紅的《深度學習》技術
Deep Learning

- 其中所使用的《捲積神經網路》
  - (Convolutional Neural Network)
  - 也只不過是將《反傳遞神經網路》的中間層稍微改變了一些而已！

有關神經網路的議題

- 以及最近因為 AlphaGo 大戰《李世石》引發大家對《捲積神經網路》與《深度學習》強烈好奇的問題，就讓我們留待下次的《十分鐘系列》再來探討了！

以上

- 就是我們今天的十分鐘系列！

希望

- 您已經學會了
  - 爬山演算法
  - 各種優化算法
  - 還有關於人工智慧的基本概念

我們下次見囉！

Bye bye!