

程式人《十分鐘系列》



用十分鐘快速理解 《深度學習技術》

陳鍾誠

2016 年 11 月 24 日

話說

- 最近深度學習技術很火熱！

幾個月前

- AlphaGo 和李世石的圍棋大戰

讓我開始關注深度學習這個主題！

最近幾天

- 當我發現 Google 翻譯改採深度學習技術之後，竟然進步如此之多！

這就讓我想起了

- 年輕時候的那個夢想！

那個夢想就是

- 我要創造出一種程式
- 會自動做英漢翻譯
- 這樣我們就不用學英文了！

於是我想

- 或許這次有機會能完成那個
夢想也說不定！

可怕的是

- 《深度學習》裏的那些數學
是連我這種教過微積分的人
都感到相當害怕的！

這種恐懼感

- 就像是面對吃人的巨人那樣
在心裡留下了深深的傷口 ..

還好

- 現在有很多開放原始碼

讓我可以

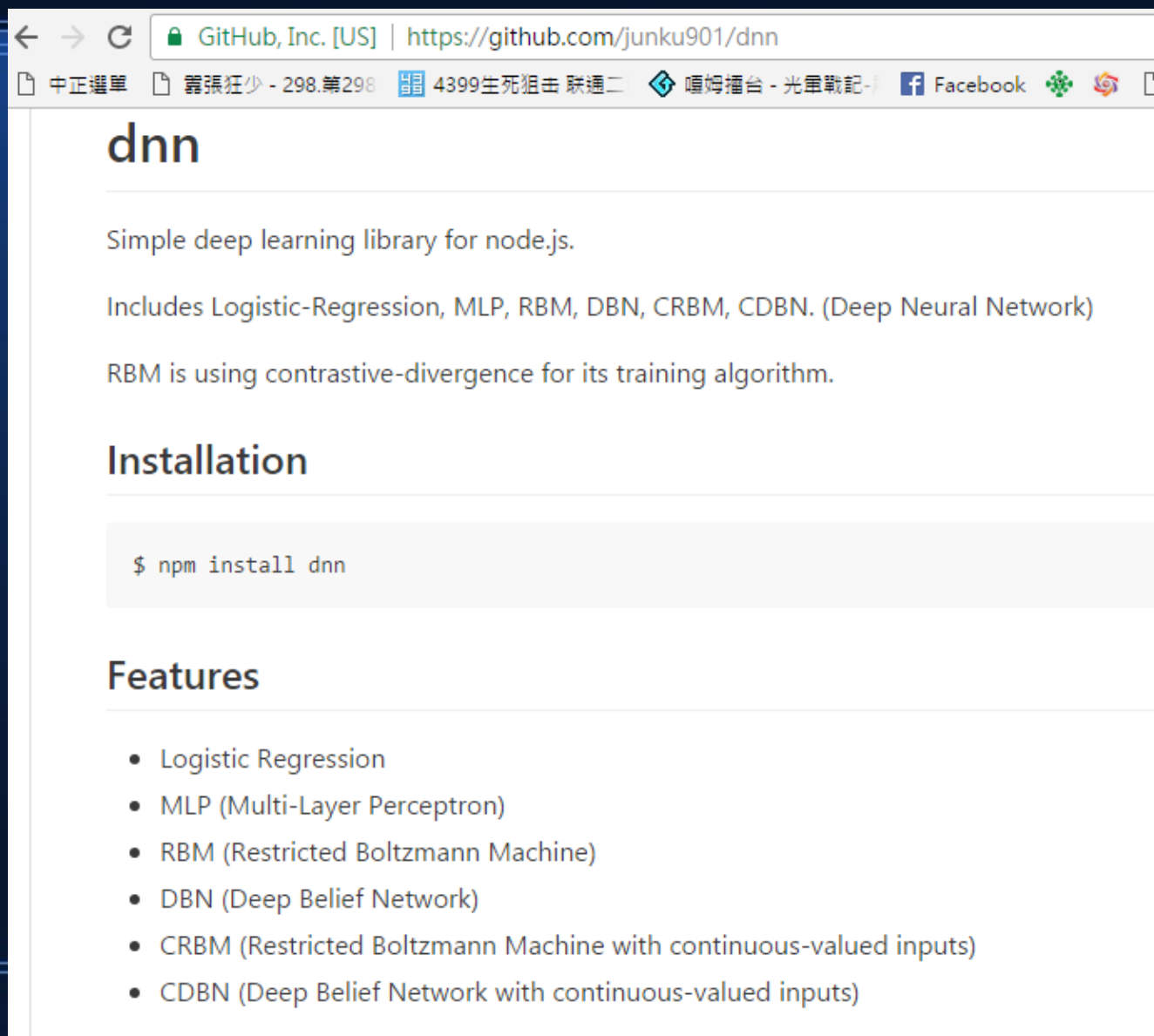
- 從程式中體會數學的意義！

現在

- 就讓我們搭配 dnn.js 這個專案，
來學習關於深度學習的那些事情吧！

<https://github.com/junku901/dnn>

雖然 dnn 一點都不強大



但卻實作了深度學習的核心算法

- 包含受限波茲曼機 RBM 和多層感知器 MLP

- Logistic Regression
- MLP (Multi-Layer Perceptron)
- RBM (Restricted Boltzmann Machine)
- DBN (Deep Belief Network)
- CRBM (Restricted Boltzmann Machine with continuous-valued inputs)
- CDBN (Deep Belief Network with continuous-valued inputs)

- 還有將兩者組合的《深信念網路 DBN》

以及《連續實數版》的 CRBM, CDBN 等等

雖然 dnn 只是個小程序式

- 也不像 Google 的 TensorFlow 那麼知名

但是

- 學習應該從簡單開始！
- 小蝦米還是不要一口就想吞鯨魚

今天

- 我們將透過 dnn.js 這個簡單的《深度學習程式範例》
- 由淺入深的，慢慢理解深度學習技術的那些事情！

首先

- 請您先安裝 node.js 環境

然後執行 `npm install dnn`

- 會看到下列訊息！

```
D:\>mkdir deepLearning
D:\>cd deepLearning
D:\deepLearning>npm install dnn
D:\deepLearning
`-- dnn@0.1.0

npm WARN enoent ENOENT: no such file or directory, open 'D:\deepLearning\package
.json'
npm WARN deepLearning No description
npm WARN deepLearning No repository field.
npm WARN deepLearning No README data
npm WARN deepLearning No license field.
```

那些警告訊息是告訴你，你沒有建立自己專案的 `package.json` 檔案，這並不影響執行！

切到 dnn 的範例資料夾

```
D:\deepLearning>ls
node_modules

D:\deepLearning>cd node_modules

D:\deepLearning\node_modules>cd dnn

D:\deepLearning\node_modules\dnn>ls
README.md  examples  lib  package.json

D:\deepLearning\node_modules\dnn>cd examples

D:\deepLearning\node_modules\dnn\examples>ls
cdbn.js  crbm.js  dbn.js  logistic_regression.js  mlp.js  rbm.js
```

接著就可以開始執行範例了

但執行範例前

- 讓我們先對 dnn.js 有個整體概念

MLP: 多層感知器 (二進位版)

RBM: 受限波茲曼機 (二進位版)

DBN: 深度信念網路 = RBM 訓練後, 再用 MLP 微調的方法

CRBM: 連續版本的 RBM (實數版)

CDBN: 連續版本的 DBN (實數版)

Logical Regression : 邏輯迴歸 (對照用)

對照用的範例是 Logistic Regression

```
var dnn = require('dnn');
var x = [[1,1,1,0,0,0],
        [1,0,1,0,0,0],
        [1,1,1,0,0,0],
        [0,0,1,1,1,0],
        [0,0,1,1,0,0],
        [0,0,1,1,1,0]];
var y = [[1, 0],
        [1, 0],
        [1, 0],
        [0, 1],
        [0, 1],
        [0, 1]];

var lrClassifier = new dnn.LogisticRegression({
  'input' : x,
  'label' : y,
  'n_in' : 6,
  'n_out' : 2
});
```

邏輯迴歸程式！

```
lrClassifier.set('log level',1); // 0 : nothing, 1

var training_epochs = 800, lr = 0.01;

lrClassifier.train({
  'lr' : lr,
  'epochs' : training_epochs
});

x = [[1, 1, 0, 0, 0, 0],
     [0, 0, 0, 1, 1, 0],
     [1, 1, 1, 1, 1, 0]];

console.log("Result : ",lrClassifier.predict(x));
```

邏輯迴歸是多變量分析的方法

- 像是最小平方法也是一種多變量分析程式
- 但邏輯迴歸採用的是《最大似然估計》

邏輯分布公式 [\[編輯\]](#)

$$P(Y = 1|X = x) = \frac{e^{x'\beta}}{1 + e^{x'\beta}}.$$

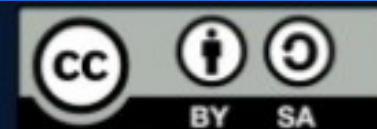
其中參數 β 常用[最大似然估計](#)。

邏輯迴歸通常採用

- 邏輯函數 $\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}}$ 進行分類
- 學過神經網路的人一定會覺得非常熟悉

昨天的十分鐘我們介紹過神經網路

程式人《十分鐘系列》



用十分鐘理解

神經網路發展史

陳鍾誠

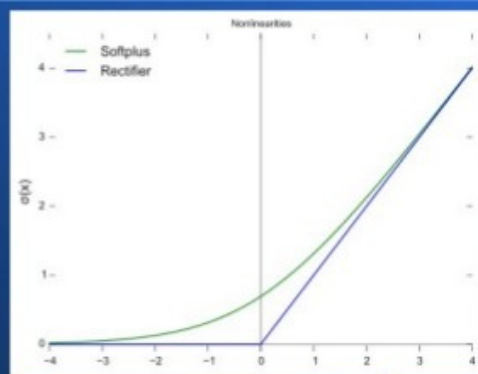
2016 年 11 月 23 日

裡面就有這類函數

另外還會加入 RELU 函數連接

線性整流函數 (Rectified Linear Unit, **ReLU**) ,又稱修正線性單元, 是一種人工神經網絡中常用的激活函數 (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絡廣泛使用於諸如圖像識別等**計算機視覺**^[1] 人工智慧領域。

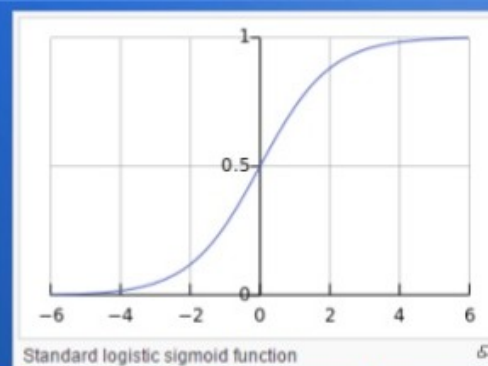


直線版 Rectifier

$$f(x) = \max(0, x)$$

平滑版 Softplus

$$f(x) = \ln(1 + e^x)$$



註：這和傳統神經網路使用 Sigmoid 的《邏輯 logistic》函數有所不同

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

意思是用神經網路
也可以做這種邏輯迴歸分類

- 方法是先加總 $t = \beta_0 + \beta_1 x$ 之後
計算 $\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}}$ 並進行分類
- 最終結果是：
$$F(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$
- 然後進行分類：
$$y = \begin{cases} 1 & \beta_0 + \beta_1 x + \varepsilon > 0 \\ 0 & \text{else} \end{cases}$$

當然變數可以不只一個

Multiple explanatory variables [\[edit \]](#)

If there are multiple explanatory variables, the above expression $\beta_0 + \beta_1 x$ can be revised to $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m$. Then when this is used in the equation relating the logged odds of a success to the values of the predictors, the linear regression will be a multiple regression with m explanators; the parameters β_j for all $j = 0, 1, 2, \dots, m$ are all estimated.

而且神經網路層數

- 可以不只一層！

讓我們回到剛剛的 dnn.js 程式範例 (Logistic Regression)

```
var dnn = require('dnn');
var x = [[1,1,1,0,0,0],
         [1,0,1,0,0,0],
         [1,1,1,0,0,0],
         [0,0,1,1,1,0],
         [0,0,1,1,0,0],
         [0,0,1,1,1,0]];
var y = [[1, 0],
         [1, 0],
         [1, 0],
         [0, 1],
         [0, 1],
         [0, 1]];

var lrClassifier = new dnn.LogisticRegression({
  'input' : x,
  'label' : y,
  'n_in' : 6,
  'n_out' : 2
});
```

邏輯迴歸程式！

```
lrClassifier.set('log level',1); // 0 : nothing, 1

var training_epochs = 800, lr = 0.01;

lrClassifier.train({
  'lr' : lr,
  'epochs' : training_epochs
});


x = [[1, 1, 0, 0, 0, 0],
     [0, 0, 0, 1, 1, 0],
     [1, 1, 1, 1, 1, 0]];

console.log("Result : ",lrClassifier.predict(x));
```

看看執行結果

```
D:\deepLearning\node_modules\dnn\examples>node logistic_regression.js  
LogisticRegression 1 % Completed.  
LogisticRegression 2 % Completed.  
LogisticRegression 3 % Completed.
```

```
LogisticRegression 97 % Completed.  
LogisticRegression 98 % Completed.  
LogisticRegression 99 % Completed.  
LogisticRegression Final Cross Entropy : 0.03819352709370631  
Result : [ [ 0.9913047066877619, 0.00869529331223797 ],  
           [ 0.00869529331223797, 0.9913047066877619 ],  
           [ 0.5, 0.5 ] ]
```



最小化後的交叉亂度 Cross Entropy 為 0.038

問題是

- 這個執行結果該怎麼解讀呢？

```
D:\deepLearning\node_modules\dnn\examples>node logistic_regression.js
LogisticRegression 1 % Completed.
LogisticRegression 2 % Completed.
LogisticRegression 3 % Completed.
```

```
LogisticRegression 97 % Completed.
LogisticRegression 98 % Completed.
LogisticRegression 99 % Completed.
LogisticRegression Final Cross Entropy : 0.03819352709370631
Result :  [ [ 0.9913047066877619, 0.00869529331223797 ],
            [ 0.00869529331223797, 0.9913047066877619 ],
            [ 0.5, 0.5 ] ]
```

首先看看輸入與輸出

- 用來訓練的標準答案希望輸入 x 時會輸出 y

```
x = [[1,1,1,0,0,0], y = [[1, 0],  
    [1,0,1,0,0,0],      [1, 0],  
    [1,1,1,0,0,0],      [1, 0],  
    [0,0,1,1,1,0],      [0, 1],  
    [0,0,1,1,0,0],      [0, 1],  
    [0,0,1,1,1,0]];     [0, 1]];
```

- 也就是輸入 111000 時會輸出 10

輸入 001110 時會輸出 01

...

在訓練完成後的預測結果

```
var training_epochs = 800, lr = 0.01;
```

```
lrClassifier.train({  
  'lr' : lr,  
  'epochs' : training_epochs  
});
```

```
x = [[1, 1, 0, 0, 0, 0],  
     [0, 0, 0, 1, 1, 0],  
     [1, 1, 1, 1, 1, 0]];
```

預測結果為

0.9913, 0.0086
0.0086, 0.9913
0.5, 0.5

```
console.log("Result : ",lrClassifier.predict(x));
```

該神經網路可以
對輸入 x 進行預測

```
Result :  [ [ 0.9913047066877619, 0.00869529331223797 ],  
           [ 0.00869529331223797, 0.9913047066877619 ],  
           [ 0.5, 0.5 ] ]
```

用分類函數的角度看

訓練資料大致分為兩塊

$x = \begin{bmatrix} [1, 1, 1, 0, 0, 0], \\ [1, 0, 1, 0, 0, 0], \\ [1, 1, 1, 0, 0, 0], \\ [0, 0, 1, 1, 1, 0], \\ [0, 0, 1, 1, 0, 0], \\ [0, 0, 1, 1, 1, 0] \end{bmatrix};$ $y = \begin{bmatrix} [1, 0], \\ [1, 0], \\ [1, 0], \\ [0, 1], \\ [0, 1], \\ [0, 1] \end{bmatrix};$

Cross Entropy 為 0.038

- 預測能力也不錯：預測結果為

$\begin{bmatrix} [1, 1, 0, 0, 0, 0] \\ [0, 0, 0, 1, 1, 0] \\ [1, 1, 1, 1, 1, 0] \end{bmatrix}$

0.9913, 0.0086

0.0086, 0.9913

0.5, 0.5

解讀

第一類

第二類

難以分類

dnn.js 的邏輯迴歸法採用 $y = \text{SoftMax}(xW + b)$
透過梯度下降的方式調整 W, b 以逼近 y

```
for(i=0;i<epochs;i++) {  
    var probYgivenX = math.softmaxMat(math.addMatVec(math.mulMat(self.  
x,self.W),self.b));  
    var deltaY = math.minusMat(self.y,probYgivenX);  
  
    var deltaW = math.mulMat(math.transpose(self.x),deltaY);  
    var deltaB = math.meanMatAxis(deltaY,0);  
  
    self.W = math.addMat(self.W,math.mulMatScalar(deltaW,lr));  
    self.b = math.addVec(self.b,math.mulVecScalar(deltaB,lr));  
    if(self.settings['log level'] > 0) {  
        var progress = (1.*i/epochs)*100;  
        if(progress > currentProgress) {  
            console.log("LogisticRegression",progress.toFixed(0),"%  
Completed.");  
            currentProgress++;  
        }  
    }  
}
```

但上述的是邏輯迴歸法

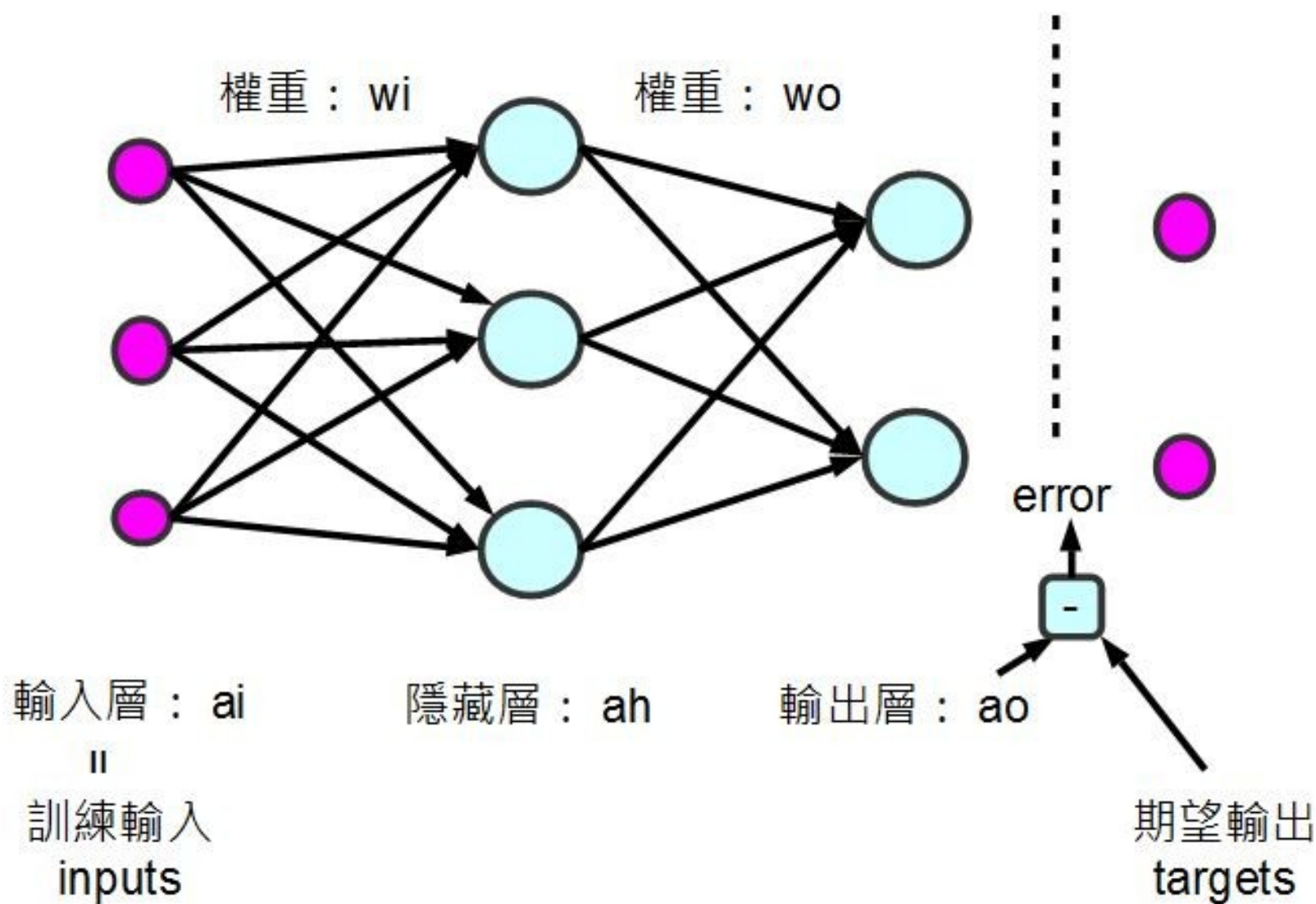
- 只是拿來當深度學習的《對照組》而已！

dnn 裏的深度學習模組

- 是由
 - 受限波茲曼 RBM
 - 多層感知器 MLP
- 所組成的
 - 深度信念網路 DBN

現在就讓我們先執行看看吧！

先瞭解一下多層感知器 MLP 模型



再執行多層感知器範例 mlp.js

```
D:\deepLearning\node_modules\dnn\examples>node mlp
MLP 1 % Completed.
MLP 9 % Completed.
MLP 17 % Completed.
MLP 25 % Completed.
MLP 33 % Completed.
MLP 41 % Completed.
MLP 49 % Completed.
MLP 57 % Completed.
MLP 65 % Completed.
MLP 73 % Completed.
MLP 81 % Completed.
MLP 89 % Completed.
MLP 97 % Completed.
MLP Final Cross Entropy : 0.019417566867119228
[ [ 0.9922553104522456, 0.007631463388737132 ],
  [ 0.010082947515195171, 0.9900558264665876 ],
  [ 0.337180747545254, 0.6627139249988923 ] ]
```

mlp 的主程式重點

```
x = [[0.4, 0.5, 0.5, 0., 0., 0.],  
      [0.5, 0.3, 0.5, 0., 0., 0.],  
      [0.4, 0.5, 0.5, 0., 0., 0.],  
      [0., 0., 0.5, 0.3, 0.5, 0.],  
      [0., 0., 0.5, 0.4, 0.5, 0.],  
      [0., 0., 0.5, 0.5, 0.5, 0.]];
```

```
y = [[1, 0],  
      [1, 0],  
      [1, 0],  
      [0, 1],  
      [0, 1],  
      [0, 1]];
```

```
var mlp = new dnn.MLP({  
  'input' : x,  
  'label' : y,  
  'n_ins' : 6,  
  'n_outs' : 2,  
  'hidden_layer_sizes' : [4,4,5]  
});
```

```
mlp.train({  
  'lr' : 0.6,  
  'epochs' : 20000  
});  
  
a = [[0.5, 0.5, 0., 0., 0., 0.],  
      [0., 0., 0., 0.5, 0.5, 0.],  
      [0.5, 0.5, 0.5, 0.5, 0.5, 0.]];  
  
console.log(mlp.predict(a));
```

訓練完後對 a 進行預測

```
x = [[0.4, 0.5, 0.5, 0., 0., 0.], y = [[1, 0],  
      [0.5, 0.3, 0.5, 0., 0., 0.]      [1, 0],  
      [0.4, 0.5, 0.5, 0., 0., 0.],      [1, 0],  
      [0., 0., 0.5, 0.3, 0.5, 0.],      [0, 1],  
      [0., 0., 0.5, 0.4, 0.5, 0.],      [0, 1],  
      [0., 0., 0.5, 0.5, 0.5, 0.]]      [0, 1]];
```

```
a = [[0.5, 0.5, 0., 0., 0., 0.],  
      [0., 0., 0., 0.5, 0.5, 0.],  
      [0.5, 0.5, 0.5, 0.5, 0.5, 0.]];
```

```
console.log(mlp.predict(a));
```

MLP 97 % Completed.

MLP Final Cross Entropy : 0.019417566867119228

[[0.9922553104522456, 0.007631463388737132 1,

[0.010082947515195171, 0.9900558264665876 1,

[0.337180747545254, 0.6627139249988923]]

0.9922, 0,0076

0.0100, 0,9900

0.3371, 0.6627

解讀

第一類

第二類

難以分類

上述的多層感知器 MLP 採用的

- 是傳統經典的反傳遞學習法 backpropagation

```
initialize network weights (often small random values)
do
  forEach training example named ex
    prediction = neural-net-output(network, ex) // forward pass
    actual = teacher-output(ex)
    compute error (prediction - actual) at the output units
    compute  $\Delta w_h$  for all weights from hidden layer to output layer // backward pass
    compute  $\Delta w_i$  for all weights from input layer to hidden layer // backward pass
    update network weights // input layer not modified by error estimate
  until all examples classified correctly or another stopping criterion satisfied
return the network
```

MLP 的關鍵程式碼（正向傳遞）

```
for(epoch=0 ; epoch < epochs ; epoch++) {  
  
    // Feed Forward  
    var i;  
    var layerInput = [];  
    layerInput.push(self.x);  
    for(i=0; i<self.nLayers+1 ; i++) {  
        layerInput.push(self.sigmoidLayers[i].output(layerInput[i]));  
    }  
    var output = layerInput[self.nLayers+1];  
}
```

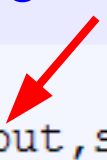
每一層都向前算

$\text{layerInput}[i+1] = \text{output}(\text{layerInput}[i])$



```
HiddenLayer.prototype.output = function(input) {  
    var self = this;  
    if(typeof input !== 'undefined') self.input = input;  
    var linearOutput = math.addMatVec(math.mulMat(self.input, self.W), self.b);  
    return math.activateMat(linearOutput, self.activation);  
};
```

$\text{output} = \text{sigmoid}(\text{input} * W + b)$



MLP 的關鍵程式碼（反向傳遞）

```
// Back Propagation
var delta = new Array(self.nLayers + 1);
delta[self.nLayers] = m.mulMatElementWise(m.minusMat(self.y, output),
    m.activateMat(self.sigmoidLayers[self.nLayers].linearOutput(layerInput[self.nLayers]), m.dSigmoid));

/*
self.nLayers = 3 (3 hidden layers)
delta[3] : output layer
delta[2] : 3rd hidden layer, delta[0] : 1st hidden layer
*/
for(i = self.nLayers - 1; i >= 0; i--) {
    delta[i] = m.mulMatElementWise(self.sigmoidLayers[i+1].backPropagate(delta[i+1]),
        m.activateMat(self.sigmoidLayers[i].linearOutput(layerInput[i]), m.dSigmoid));
}
```

m.dSigmoid = function(x) {
 a = m.sigmoid(x);
 return a * (1. - a);
};


// 計算輸出層的誤差值
 $\text{delta}[n] = (y - \text{sigmoid}(o)) * d\text{Sigmoid}(o)$

// 第 i+1 層的誤差值反饋給第 i 層
 $\text{delta}[i] = \text{layer}[i+1].\text{backPropagate}(\text{delta}[i+1])$

```
HiddenLayer.prototype.backPropagate = function (input) { // example+num * n_out matrix
    var self = this;
    if(typeof input === 'undefined') throw new Error("No BackPropagation Input.")
    var linearOutput = math.mulMat(input, m.transpose(self.W));
    return linearOutput;
}
```

$\text{backPropagate} = \text{input} * W^T$

MLP 的關鍵程式碼（更新權重）



```
// Update Weight, Bias
for(var i=0; i<self.nLayers+1 ; i++) {
    var deltaW = m.activateMat(m.mulMat(m.transpose(layerInput[i]),
    delta[i]),function(x){return 1. * x / self.x.length;})
    var deltaB = m.meanMatAxis(delta[i],0);
    self.sigmoidLayers[i].W = m.addMat(self.sigmoidLayers[i].W,
    deltaW);
    self.sigmoidLayers[i].b = m.addVec(self.sigmoidLayers[i].b,
    deltaB);
}
```

$\text{deltaW} = \text{layerInput}[i]^T * \text{delta}[i]$; $\text{deltaB} = \text{colMean}(\text{delta}[i])$

$W = W + \text{deltaW}$

; $b = b + \text{deltaB}$

MLP 的關鍵程式碼（計算輸出）

- 和前向傳遞的方法相同

```
MLP.prototype.predict = function(x) {  
    var self = this;  
    var output = x;  
    for(i=0; i<self.nLayers+1 ; i++) {  
        output = self.sigmoidLayers[i].output(output);  
    }  
    return output;  
};
```

這就是著名的多層感知器

- 還有《反向傳遞學習算法》
的實作方式了！

接著看《受限波茲曼機 RBM》

- 還有其《對比散度》學習法
(Contrast Divergence, CD)

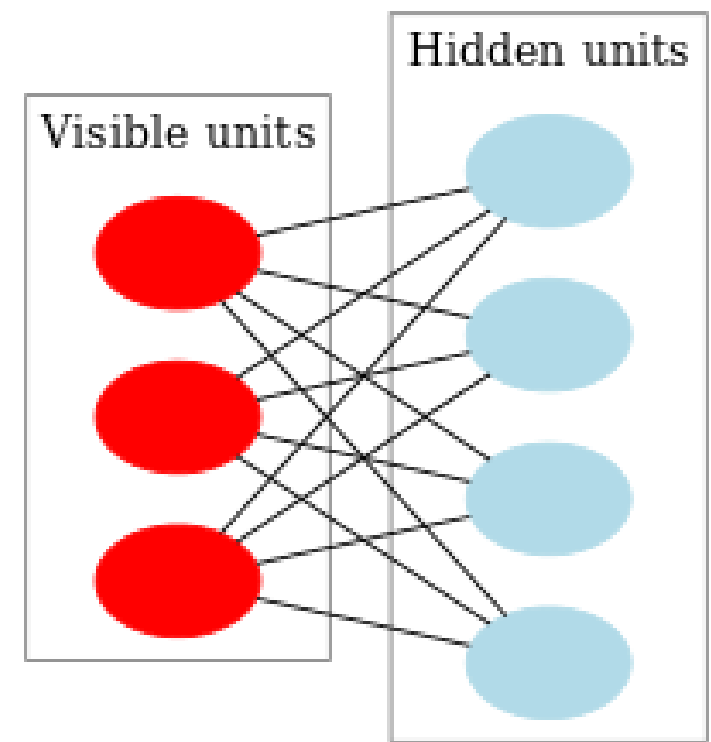
對比散度 CD 學習法的關鍵 (是用 v 預測 h' ，再用 h' 預測 v')

由於RBM為一個二分圖，層內沒有邊相連，因而隱層是否激活在給定可見層節點取值的情況下是條件獨立的。類似地，可見層節點的激活狀態在給定隱層取值的情況下也條件獨立^[6]。亦即，對 m 個可見層節點和 n 個隱層節點，可見層的配置 v 對於隱層配置 h 的條件機率如下：

$$P(v|h) = \prod_{i=1}^m P(v_i|h).$$

類似地， h 對於 v 的條件機率為

$$P(h|v) = \prod_{j=1}^n P(h_j|v).$$



包含三個可見單元和四個隱單元的受限玻爾茲曼機示意圖（不包含偏置節點）

該方法的數學式如下

- 用顯層值 v 估計隱層機率 h'

$$P(h|v) = \prod_{j=1}^n P(h_j|v)$$

實作程式： $h1_mean = \text{propup}(v) = \text{sigmoid}(v*W + \text{hbias})$

- 再用隱層 h' 估計顯層機率 v'

$$P(v|h) = \prod_{i=1}^m P(v_i|h)$$

實作程式： $v1_mean = \text{propdwn}(h) = \text{sigmoid}(h*W^T + \text{vbias})$

- 然後用下列算式去更新權重

$$\Delta w_{i,j} = \epsilon(vh^T - v'h'^T)$$

以下是單步對比散度算法 CD-1

基本的針對一個樣本的單步對比分歧（CD-1）步驟可被總結如下：

1. 取一個訓練樣本 v ，計算隱層節點的機率，在此基礎上從這一機率分布中獲取一個隱層節點激活向量的樣本；
2. 計算 v 和 h 的外積，稱為「正梯度」；
3. 從 h 獲取一個重構的可見層節點的激活向量樣本 v' ，此後從 v' 再次獲得一個隱層節點的激活向量樣本 h' ；
4. 計算 v' 和 h' 的外積，稱為「負梯度」；
5. 使用正梯度和負梯度的差以一定的學習率更新權重 $w_{i,j}$ ：
$$\Delta w_{i,j} = \epsilon(vh^T - v'h'^T)。$$

偏置 a 和 b 也可以使用類似的方法更新。

多步的對比散度演算法

初始化: 令可见层单元的初始状态 $\mathbf{v}_1 = \mathbf{x}_0$; W 、 \mathbf{a} 和 \mathbf{b} 为随机的较小数值。

For $t = 1, 2, \dots, T$

For $j = 1, 2, \dots, m$ (对所有隐单元)

计算 $P(\mathbf{h}_{1j} = 1|\mathbf{v}_1)$, 即 $P(\mathbf{h}_{1j} = 1|\mathbf{v}_1) = \sigma(b_j + \sum_i v_{1i} W_{ij})$;

从条件分布 $P(\mathbf{h}_{1j}|\mathbf{v}_1)$ 中抽取 $\mathbf{h}_{1j} \in \{0, 1\}$.

EndFor

For $i = 1, 2, \dots, n$ (对所有可见单元)

计算 $P(\mathbf{v}_{2i} = 1|\mathbf{h}_1)$, 即 $P(\mathbf{v}_{2i} = 1|\mathbf{h}_1) = \sigma(a_i + \sum_j W_{ij} h_{1j})$;

从条件分布 $P(\mathbf{v}_{2i}|\mathbf{h}_1)$ 中抽取 $\mathbf{v}_{2i} \in \{0, 1\}$.

EndFor

For $j = 1, 2, \dots, m$ (对所有隐单元)

计算 $P(\mathbf{h}_{2j} = 1|\mathbf{v}_2)$, 即 $P(\mathbf{h}_{2j} = 1|\mathbf{v}_2) = \sigma(b_j + \sum_i v_{2i} W_{ij})$;

EndFor

按下式更新各个参数

– $W \leftarrow W + \epsilon(P(\mathbf{h}_{1.} = 1|\mathbf{v}_1)\mathbf{v}_1^T - P(\mathbf{h}_{2.} = 1|\mathbf{v}_2)\mathbf{v}_2^T)$;

– $\mathbf{a} \leftarrow \mathbf{a} + \epsilon(\mathbf{v}_1 - \mathbf{v}_2)$;

– $\mathbf{b} \leftarrow \mathbf{b} + \epsilon(P(\mathbf{h}_{1.} = 1|\mathbf{v}_1) - P(\mathbf{h}_{2.} = 1|\mathbf{v}_2))$;

理解了原理之後

- 讓我們真正來執行受限波茲曼機的 rbm 程式
- 以及看看真正的對比散度學習法程式碼！

請執行 node rbm

```
RBM 81 % Completed.  
RBM 89 % Completed.  
RBM 97 % Completed.  
RBM Final Cross Entropy : 0.6616376645777886  
[ [ 0.994075177436916,  
    0.5810711201955319,  
    0.996782288311315,  
    0.008527218463050252,  
    0.0077485873160137,  
    0.002799400912007968 ],  
  [ 0.005620274338682828,  
    0.0027108379759264835,  
    0.9955310337095183,  
    0.9977488822307009,  
    0.7024036612316482,  
    0.002642187157021731 ] ]  
[ [ 0.000044298398175647186, 0.998052280010676 ],  
  [ 0.9988097875852785, 0.00003900626380158672 ] ]
```

rbm 是用來大致分群的

- 所以不需要標準答案

```
var data = [[1,1,1,0,0,0],  
            [1,0,1,0,0,0],  
            [1,1,1,0,0,0],  
            [0,0,1,1,1,0],  
            [0,0,1,1,0,0],  
            [0,0,1,1,1,0]];
```

結果的解讀

```
console.log(rbm.reconstruct(v));
```

```
var data = [[1,1,1,0,0,0],  
            [1,0,1,0,0,0],  
            [1,1,1,0,0,0],  
            [0,0,1,1,1,0],  
            [0,0,1,1,0,0],  
            [0,0,1,1,1,0]];
```

```
[ 0.994075177436916,  
  0.5810711201955319, 第1群  
  0.996782288311315,  
  0.008527218463050252,  
  0.0077485873160137,  
  0.002799400912007968 1,  
 [ 0.005620274338682828,  
  0.0027108379759264835,  
  0.9955310337095183,  
  0.9977488822307009, 第2群  
  0.7024036612316482,  
  0.002642187157021731 1 1
```

第2群觸發
隱層節點 1

```
v = [[1, 1, 0, 0, 0, 0],  
      [0, 0, 0, 1, 1, 0]];
```

第1群觸發
隱層節點 2

```
[ 0.000044298398175647186, 0.998052280010676 1,  
 [ 0.9988097875852785, 0.00003900626380158672 1
```

所以隱層節點 h

- 確實學到了如何資料進行分群
- 只要你給出 v 並用 $\text{sampleHgivenV}(v)$ 就能觀察到隱層節點 h 的關聯性！
- 而用 $\text{reconstruct}(v)$ 則能用 $v \Rightarrow h' \Rightarrow v'$ 的方式觀察重建的 v' 值。

接著看看 RBM 的程式碼

```
for(i=0;i<epochs;i++) {  
    /* CD - k . Contrastive Divergence */  
    var ph = self.sampleHgivenV(self.input);  
    var phMean = ph[0], phSample = ph[1];  
    var chainStart = phSample;  
    var nvMeans, nvSamples, nhMeans, nhSamples;  
  
    for(j=0 ; j<k ; j++) {  
        if (j==0) {  
            var gibbsVH = self.gibbsHVH(chainStart);  
            nvMeans = gibbsVH[0], nvSamples = gibbsVH[1], nhMeans = gibbsVH[2], nhSamples = gibbsVH[3];  
        } else {  
            var gibbsVH = self.gibbsSHVH(nhSamples);  
            nvMeans = gibbsVH[0], nvSamples = gibbsVH[1], nhMeans = gibbsVH[2], nhSamples = gibbsVH[3];  
        }  
    }  
}
```

估計第一層的 h

用 h 估計 v' , 再用 v' 估計 h'

對於每一層都這樣做！

完成後更新權重

```
var deltaW = math.mulMatScalar(math.minusMat(math.mulMat(math.transpose(self.vh), self.h), self.v), self.h);  
var deltaVbias = math.meanMatAxis(math.minusMat(self.input, nvSamples), 0);  
var deltaHbias = math.meanMatAxis(math.minusMat(phSample, nhMeans), 0);  
  
self.W = math.addMat(self.W, math.mulMatScalar(deltaW, lr));  
self.vbias = math.addVec(self.vbias, math.mulVecScalar(deltaVbias, lr));  
self.hbias = math.addVec(self.hbias, math.mulVecScalar(deltaHbias, lr));
```

$$\Delta w_{i,j} = \epsilon(vh^T - v'h'^T)$$

其中關鍵的 gibbsHVH 函數

```
RBM.prototype.sampleHgivenV = function(v0_sample) {  
    var self = this;  
    var h1_mean = self.propup(v0_sample);  
    var h1_sample = math.probToBinaryMat(h1_mean);  
    return [h1_mean, h1_sample];  
};  
  
RBM.prototype.sampleVgivenH = function(h0_sample) {  
    var self = this;  
    var v1_mean = self.propdown(h0_sample);  
    var v1_sample = math.probToBinaryMat(v1_mean);  
    return [v1_mean, v1_sample];  
};  
  
RBM.prototype.gibbsHVH = function(h0_sample) {  
    var self = this;  
    var v1 = self.sampleVgivenH(h0_sample);  
    var h1 = self.sampleHgivenV(v1[1]);  
    return [v1[0], v1[1], h1[0], h1[1]];  
};
```

再追下去 propup, proppdown 等函數

```
RBM.prototype.propup = function(v) {  
    var self = this;  
    var preSigmoidActivation = math.addMatVec(math.  
mulMat(v,self.W),self.hbias);  
    return math.activateMat(preSigmoidActivation, m.  
sigmoid);  
};
```

propup = sigmoid($v \cdot W + \text{hbias}$)

```
RBM.prototype.proppdown = function(h) {  
    var self = this;  
    var preSigmoidActivation = math.addMatVec(math.  
mulMat(h,math.transpose(self.W)),self.vbias);  
    return math.activateMat(preSigmoidActivation, m.  
sigmoid);  
};
```

proppdown = sigmoid($h \cdot W^T + \text{vbias}$)

所以 sampleHgivenV 函數

```
RBM.prototype.sampleHgivenV = function(v0_sample) {  
    var self = this;  
    var h1_mean = self.propup(v0_sample);  
    var h1_sample = math.probToBinaryMat(h1_mean);  
    return [h1_mean, h1_sample];  
};
```

是用 v 估計 h : $h1_mean = propup(v) = \text{sigmoid}(v * W + h\text{bias})$

$h1_sample$ 則是將 $h1_mean$ 轉成 0 或 1

sampleVgivenH 函數則反過來

```
RBM.prototype.sampleVgivenH = function(h0_sample) {  
    var self = this;  
    var v1_mean = self.propdown(h0_sample);  
    var v1_sample = math.probToBinaryMat(v1_mean);  
    return [v1_mean, v1_sample];  
};
```

用 h 估計 v : $v1_mean = propdown(h) = \text{sigmoid}(h * W^T + vbias)$

$v1_sample$ 同樣是轉為 0 或 1 的布林值

在開始學習之前

```
if(typeof settings['W'] === 'undefined') {  
    var a = 1. / self.nVisible;  
    settings['W'] = math.randMat(self.nVisible,  
self.nHidden,-a,a);  
}  
  
if(typeof settings['hbias'] === 'undefined')  
    settings['hbias'] = math.zeroVec(self.nHidden  
);  
  
if(typeof settings['vbias'] === 'undefined')  
    settings['vbias'] = math.zeroVec(self.  
nVisible);
```

W 用亂數設定

hbias 和 vbias
都設為零

再重新看一次《對比散度算法》

```
for(i=0;i<epochs;i++) {  
    /* CD - k . Contrastive Divergence */  
    var ph = self.sampleHgivenV(self.input);  
    var phMean = ph[0], phSample = ph[1];  
    var chainStart = phSample;  
    var nvMeans, nvSamples, nhMeans, nhSamples;  
  
    for(j=0 ; j<k ; j++) {  
        if (j==0) {  
            var gibbsVH = self.gibbsHVH(  
                chainStart);  
            nvMeans = gibbsVH[0], nvSamples =  
                gibbsVH[1], nhMeans = gibbsVH[2],  
            nhSamples = gibbsVH[3];  
        } else {  
            var gibbsVH = self.gibbsHVH(nhSamples  
            );  
            nvMeans = gibbsVH[0], nvSamples =  
                gibbsVH[1], nhMeans = gibbsVH[2],  
            nhSamples = gibbsVH[3];  
        }  
    }  
}
```

用 v_0 估計 h_0

先用 h_i 估計 v_{i+1}
再用 v_{i+1} 估計 h_{i+1}

$$\begin{aligned} h_0 &\sim P(h|v_0), & v_1 &\sim P(v|h_0), \\ h_1 &\sim P(h|v_1), & v_2 &\sim P(v|h_1), \\ & \dots\dots\dots, & v_{k+1} &\sim P(v|h_k). \end{aligned}$$

估計完後的更新動作

$$\Delta W_{ij} = \epsilon(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{recon}}),$$

$$\Delta a_i = \epsilon(\langle v_i \rangle_{\text{data}} - \langle v_i \rangle_{\text{recon}}),$$

$$\Delta b_j = \epsilon(\langle h_j \rangle_{\text{data}} - \langle h_j \rangle_{\text{recon}}),$$

```
var deltaW = math.mulMatScalar(math.minusMat(
  math.mulMat(math.transpose(self.input), phMean),
  math.mulMat(math.transpose(nvSamples), nhMeans)), 1. / self.input.length);
var deltaVbias = math.meanMatAxis(math.minusMat(self.input, nvSamples), 0);
var deltaHbias = math.meanMatAxis(math.minusMat(phSample, nhMeans), 0);
```

```
self.W = math.addMat(self.W, math.mulMatScalar(deltaW, lr));
self.vbias = math.addVec(self.vbias, math.mulVecScalar(deltaVbias, lr));
self.hbias = math.addVec(self.hbias, math.mulVecScalar(deltaHbias, lr));
```

計算更新量

$$\Delta W = (v^t h - v'^t h') / n$$

$$\Delta V = \text{normalize}(v - v')$$

$$\Delta H = \text{normalize}(h - h')$$

更新 W, V, H

$$W = W + \Delta W$$

$$V = V + \Delta V$$

$$H = H + \Delta H$$

以上就是受限波茲曼機 RBM

- 還有其對比散度學習法 CD 的程式碼了

有了 RBM 與 MLP 兩種神經網路

- 就可以多層疊合
 - 先用 RBM 粗略分類之後
 - 再用 MLP 進行細部調整
- 形成《深度信念網路 DBN》了！

讓我們先執行 dbn 範例

```
D:\deepLearning\node_modules\dnn\examples>node dbn
DBN RBM 0 th Layer Final Cross Entropy: 0.4722572164357061
DBN RBM 0 th Layer Pre-Training Completed.
DBN RBM 1 th Layer Final Cross Entropy: 0.426721702657097
DBN RBM 1 th Layer Pre-Training Completed.
DBN RBM 2 th Layer Final Cross Entropy: 0.8359339428296436
DBN RBM 2 th Layer Pre-Training Completed.
DBN RBM 3 th Layer Final Cross Entropy: 0.35228608447551735
DBN RBM 3 th Layer Pre-Training Completed.
DBN RBM 4 th Layer Final Cross Entropy: 0.021801203784731008
DBN RBM 4 th Layer Pre-Training Completed.
DBN RBM 5 th Layer Final Cross Entropy: 0.026244201715647394
DBN RBM 5 th Layer Pre-Training Completed.
DBN Pre-Training Completed.
MLP 1 % Completed.
MLP 9 % Completed.
MLP 17 % Completed.
MLP 25 % Completed.
MLP 33 % Completed.
MLP 41 % Completed.
MLP 49 % Completed.
MLP 57 % Completed.
MLP 65 % Completed.
MLP 73 % Completed.
MLP 81 % Completed.
MLP 89 % Completed.
MLP 97 % Completed.
MLP Final Cross Entropy : 0.07341316913065739
[ [ 0.9687244317837319, 0.03130581054294937 ],
  [ 0.0407481362652578, 0.9592121700413977 ],
  [ 0.4292659887808886, 0.6205155753047139 ] ]
```

先用 RBM 粗略分類之後

再用 MLP 進行細部調整

完成學習之後，就能對
任意輸入取得輸出值！

dbn 範例用了六個中間層

```
var dbn = new dnn.DBN({  
  'input' : x,  
  'label' : y,  
  'n_ins' : 6,  
  'n_outs' : 2,  
  'hidden_layer_sizes' : [10,12,11,8,6,4]  
});
```

每個中間層都有 《受限波茲曼機 RBM+ 感知器》

```
// Constructing Deep Neural Network
var i;
for(i=0 ; i<self.nLayers ; i++) {
    var inputSize, layerInput;
    if(i == 0) inputSize = settings['n_ins'];
    else inputSize = settings['hidden_layer_sizes'][i-1];
    if(i == 0) layerInput = self.x;
    else layerInput = self.sigmoidLayers[self.sigmoidLayers.length-1].
        sampleHgivenV();

    var sigmoidLayer = new HiddenLayer({
        'input' : layerInput,
        'n_in' : inputSize,
        'n_out' : settings['hidden_layer_sizes'][i],
        'activation' : math.sigmoid
    });
    self.sigmoidLayers.push(sigmoidLayer);

    var rbmLayer = new RBM({
        'input' : layerInput,
        'n_visible' : inputSize,
        'n_hidden' : settings['hidden_layer_sizes'][i]
    });
    self.rbmLayers.push(rbmLayer);
}
```

感知器

受限波茲曼機

每一層先用 RBM 粗略分類

```
for(i=0; i<self.nLayers ; i++) {  
    var layerInput ,rbm;  
    if (i==0)  
        layerInput = self.x;  
    else  
        layerInput = self.sigmoidLayers[i-1].sampleHgivenV(layerInput);  
    rbm = self.rbmlayers[i];  
    rbm.set('log level',0);  
    rbm.train({  
        'lr' : lr,  
        'k' : k,  
        'input' : layerInput,  
        'epochs' : epochs  
    });  
}
```

然後再用《感知器》進行細部優化

```
for(i=0; i<self.nLayers ; i++) {
    pretrainedWArray.push(self.sigmoidLayers[i].W);
    pretrainedBArray.push(self.sigmoidLayers[i].b);
}
// W,b of Final Output Layer are not involved in pret
var mlp = new MLP({
    'input' : self.x,
    'label' : self.y,
    'n_ins' : self.nIns,
    'n_outs' : self.nOuts,
    'hidden_layer_sizes' : self.hiddenLayerSizes,
    'w_array' : pretrainedWArray,
    'b_array' : pretrainedBArray
});
mlp.set('log level',self.settings['log level']);
mlp.train({
    'lr' : lr,
    'epochs' : epochs
});
```

這種多層次的神經網路架構

- 融合了《受限波茲曼機 RBM》
與《多層感知器 MLP》
- 透過 RBM 粗略分群後，再用 MLP 細部學習的方式
- 就是所謂的《深度信念網路 DBN》了！

看到這裏

- 我想大家應該對深度學習的技術，還有 RBM 、 MLP 、 DBN 有個大致的概念了！

接下來讓我們看看

- 當輸入不是 0 與 1，而是實數的 (0-1) 之間的機率值時
- 就必須修改 RBM 的程式，變成 CRBM

(Continuous RBM, 連續受限波茲曼機)

CRBM 測試程式還是差不多

```
var dnn = require('dnn');
var data = [[0.4, 0.5, 0.5, 0., 0., 0.7],
            [0.5, 0.3, 0.5, 0., 1, 0.6],
            [0.4, 0.5, 0.5, 0., 1, 0.9],
            [0., 0., 0., 0.3, 0.5, 0.],
            [0., 0., 0., 0.4, 0.5, 0.],
            [0., 0., 0., 0.5, 0.5, 0.]];

var crbm = new dnn.CRBM({
  input : data,
  n_visible : 6,
  n_hidden : 5
});

crbm.set('log level',1); // 0 : nothing, 1 : info, 2 : warning.

crbm.train({
  lr : 0.6,
  k : 1, // CD-k.
  epochs : 1500
});

var v = [[0.5, 0.5, 0., 0., 0., 0.],
        [0., 0., 0., 0.5, 0.5, 0.]];

console.log(crbm.reconstruct(v));
console.log(crbm.sampleHgivenV(v)[0]); // get hidden layer prob
```


執行結果也類似

```
D:\deepLearning\node_modules\dnn\examples>node crbm
RBM 1 % Completed.
RBM 9 % Completed.
```

```
RBM Final Cross Entropy : 2.384549076343718
[ [ 0.4461881527489191,
    0.42094250816345347,
    0.4515421155308401,
    0.03679130227115817,
    0.6899733919408773,
    0.6903995351404129 1,
  [ 0.014984907133223547,
    0.014982060367267812,
    0.014680865683228135,
    0.35502257986158037,
    0.579398810021863,
    0.01489374671054267 1 1
[ [ 0.0003664580033086354,
    0.00022612244448464125,
    0.0003219573522749086,
    0.0005350213339647141,
    0.0004091341935104166 1,
  [ 0.9995580986178153,
    0.9995619113234891,
    0.999637248557945,
    0.9995236254140974,
    0.9992506397241046 1 1
```

但是 CRBM 更複雜一些

(以下是兩者的 sampleVgivenH 的程式碼)

```
RBM.prototype.sampleVgivenH =  
function(h0_sample) {  
    var self = this;  
    var v1_mean = self.propdown(  
        h0_sample);  
    var v1_sample = math.  
        probToBinaryMat(v1_mean);  
    return [v1_mean,v1_sample];  
};
```

```
CRBM.prototype.sampleVgivenH = function(h0_sample) {  
    var self = this;  
    var a_h = self.propdown(h0_sample);  
    var a = math.activateMat(a_h,function(x) {  
        return 1. / (1-Math.exp(-x)) ; });  
    var b = math.activateMat(a_h,function(x) {  
        return 1./x ; });  
    var v1_mean = math.minusMat(a,b);  
    var U = math.randMat(math.shape(v1_mean)[0],  
        math.shape(v1_mean)[1],0,1);  
    var c = math.activateMat(a_h,function(x) {  
        return 1 - Math.exp(x); });  
    var d = math.activateMat(math.mulMatElementWise  
        (U,c),function(x) {return 1-x; });  
    var v1_sample = math.activateTwoMat(math.  
        activateMat(d,Math.log),a_h,function(x,y) {  
            if(y==0) y += 1e-14; // Javascript Float  
            Precision Problem.. This is a limit of  
            javascript.  
            return x/y;  
        })  
    return [v1_mean,v1_sample];  
};
```

仔細看一下

```
CRBM.prototype.sampleVgivenH = function(h0_sample) {  
  var self = this;  
  var a_h = self.propdown(h0_sample);  
  var a = math.activateMat(a_h, function(x) {  
    return 1. / (1-Math.exp(-x)) ; });  
  var b = math.activateMat(a_h, function(x) {  
    return 1./x ; });  
  var v1_mean = math.minusMat(a,b);  
  var U = math.randMat(math.shape(v1_mean)[0],  
    math.shape(v1_mean)[1],0,1);  
  var c = math.activateMat(a_h, function(x) {  
    return 1 - Math.exp(x) ; });  
  var d = math.activateMat(math.mulMatElementWise  
    (U,c), function(x) {return 1-x;});  
  var v1_sample = math.activateTwoMat(math.  
    activateMat(d,Math.log), a_h, function(x,y) {  
    if(y==0) y += 1e-14; // Javascript Float  
    Precision Problem.. This is a limit of  
    javascript.  
    return x/y;  
  })  
  return [v1_mean,v1_sample];  
};
```

$a_h = v' = \text{propdown}(h)$

$a = 1/(1-e^{-v'})$

猜測：平滑化

$b = 1/v'$

$v1_mean = a - b = 1/(1-e^{-v'}) - 1/v'$

$U = \text{radnom}(m,n)$

$c = 1 - e^{v'}$

猜測：隨機取樣

$d = 1 - U * c$

$v1_sample = \log(d)/v'$

說明：目前這段我還不太懂

有了 CRBM 之後

- 就可以和 MLP 搭配，成為 CDBN
(連續深度信念網路)

CDBN 的執行結果還是差不多

```
DBN RBM 0 th Layer Final Cross Entropy: 2.1300125183901746
DBN RBM 0 th Layer Pre-Training Completed.
DBN RBM 1 th Layer Final Cross Entropy: 0.0029572011202460744
DBN RBM 1 th Layer Pre-Training Completed.
DBN RBM 2 th Layer Final Cross Entropy: 0.0028045364288642252
DBN RBM 2 th Layer Pre-Training Completed.
DBN RBM 3 th Layer Final Cross Entropy: 0.0032232537527731825
DBN RBM 3 th Layer Pre-Training Completed.
DBN RBM 4 th Layer Final Cross Entropy: 0.0029422574447065413
DBN RBM 4 th Layer Pre-Training Completed.
DBN RBM 5 th Layer Final Cross Entropy: 0.0036081190375711774
DBN RBM 5 th Layer Pre-Training Completed.
DBN Pre-Training Completed.
MLP 1 % Completed.
MLP 9 % Completed.
MLP 17 % Completed.
MLP 25 % Completed.
MLP 33 % Completed.
MLP 41 % Completed.
MLP 49 % Completed.
MLP 57 % Completed.
MLP 65 % Completed.
MLP 73 % Completed.
MLP 81 % Completed.
MLP 89 % Completed.
MLP 97 % Completed.
MLP Final Cross Entropy : 0.016932193011390807
[ [ 0.9919539190169472, 0.008045932445991862 ],
  [ 0.008814648516340307, 0.9911855196717525 ],
  [ 0.9919539023407488, 0.008045947397979588 ] ]
```

這樣

- 我們就已經看完 dnn.js 的所有
《深度學習相關程式》了！

稍微回顧一下

dnn.js 的深度神經網路，包含：

MLP: 多層感知器（二進位版）

RBM: 受限波茲曼機（二進位版）

DBN: 深度信念網路 = RBM 訓練後，再用 MLP 微調的方法

CRBM: 連續版本的 RBM（實數版）

CDBN: 連續版本的 DBN（實數版）

Logical Regression : 邏輯迴歸（對照用）

DBN 深度信念網路

- 是由

- RBM 受限波茲曼機 + CD 對比散度學習法
- MLP 多層感知機 + BP 反傳遞學習法

在每一層都串接所組成的！

兩種基本網路

- RBM 負責初步的分群權重設定
- MLP 負責細部有答案的學習微調
- RBM+MLP 可多層套疊，形成

《深度信念神經網路 DBN》！

兩種學習演算法

- RBM 的對比散度學習法
 - 是用機率模型，透過 v , h , v' , h'
 - 計算出 w , v , h 的調整量
- MLP 的反傳遞學習法
 - 是用偏微分的梯度下降法
 - 計算出 w , v , h 的調整公式

透過 dnn.js 的介紹

- 希望您已經大致了解深度學習這個技術領域了！

但是

- 今天的介紹還有不少盲點

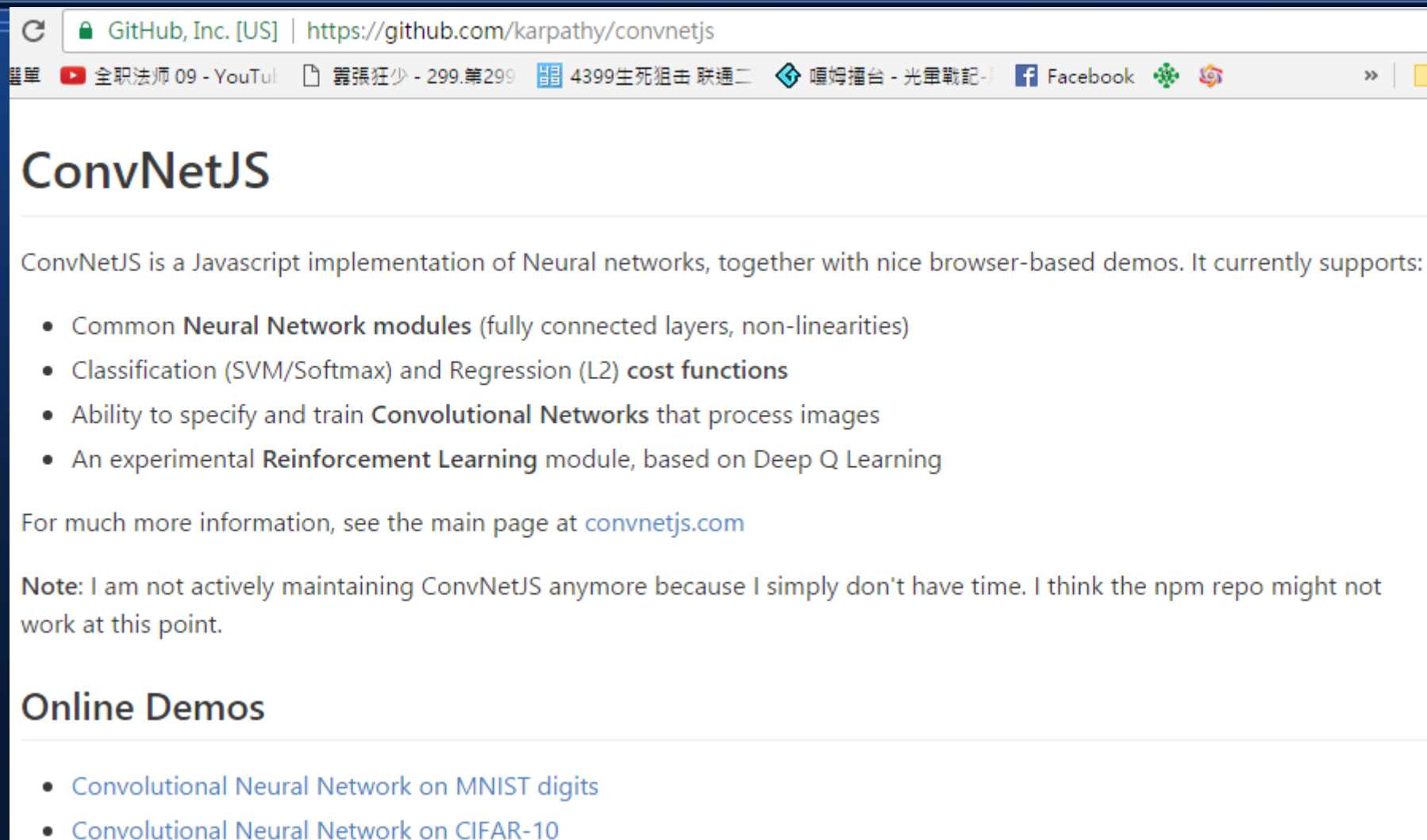
首先是

- dnn.js 的測試案例太過陽春，
無法反映出深度信念網路的能力！
- 另外 dnn.js 沒有加入捲積神經網路
的模型，也沒有影像處理的範例！

所以

- 我們將在下一篇的十分鐘系列中

介紹 ConvNetJS 這個專案



The screenshot shows a web browser window with the address bar displaying "GitHub, Inc. [US] | https://github.com/karpathy/convnetjs". The browser's taskbar at the top shows several open applications, including YouTube, a document editor, and a game. The main content of the page is the repository page for "ConvNetJS". It features a large heading "ConvNetJS" followed by a description: "ConvNetJS is a Javascript implementation of Neural networks, together with nice browser-based demos. It currently supports:". Below this is a bulleted list of features: "Common Neural Network modules (fully connected layers, non-linearities)", "Classification (SVM/Softmax) and Regression (L2) cost functions", "Ability to specify and train Convolutional Networks that process images", and "An experimental Reinforcement Learning module, based on Deep Q Learning". A paragraph follows, stating "For much more information, see the main page at [convnetjs.com](\"http://convnetjs.com\")". A "Note" section then says "I am not actively maintaining ConvNetJS anymore because I simply don't have time. I think the npm repo might not work at this point." The page concludes with a section titled "Online Demos" containing two links: "Convolutional Neural Network on MNIST digits" and "Convolutional Neural Network on CIFAR-10".

GitHub, Inc. [US] | https://github.com/karpathy/convnetjs

ConvNetJS

ConvNetJS is a Javascript implementation of Neural networks, together with nice browser-based demos. It currently supports:

- Common **Neural Network modules** (fully connected layers, non-linearities)
- Classification (SVM/Softmax) and Regression (L2) **cost functions**
- Ability to specify and train **Convolutional Networks** that process images
- An experimental **Reinforcement Learning** module, based on Deep Q Learning

For much more information, see the main page at convnetjs.com

Note: I am not actively maintaining ConvNetJS anymore because I simply don't have time. I think the npm repo might not work at this point.

Online Demos

- [Convolutional Neural Network on MNIST digits](#)
- [Convolutional Neural Network on CIFAR-10](#)

ConvNetJS

是一個以影像辨識為主的深度學習專案

- 主要的技術就是採用《深捲積神經網路》



這樣我們才能

- 更全面的瞭解深度學習技術！

這就是我們今天的

十分鐘系列

我們下回見！

Bye Bye!