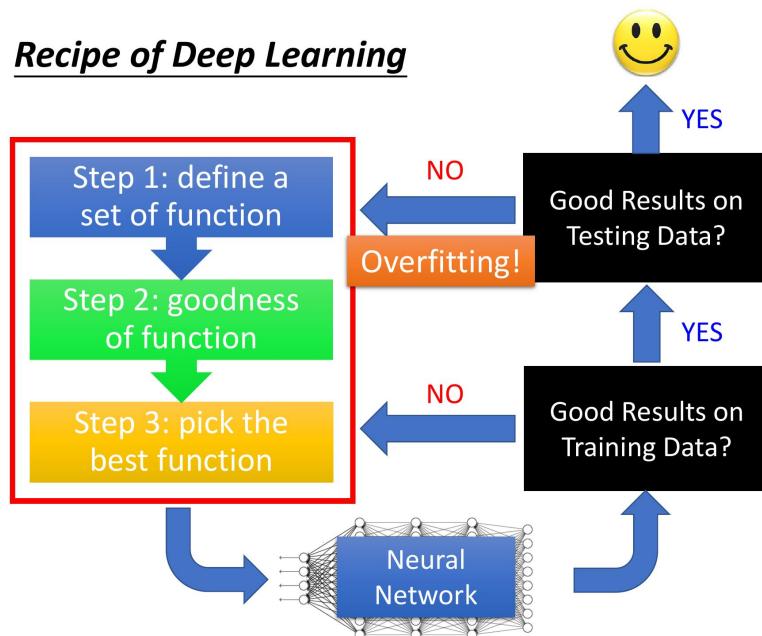


ML Lecture 9-1: Tips for Training DNN

臺灣大學人工智慧中心 科技部人工智慧技術暨全幅健康照護聯合研究中心 <http://aintu.tw>

這堂課要介紹的是幾個深度學習相關的技巧。在 CNN 那堂課程裡面，我們留下了兩個問題：第一個問題是，在 CNN 裡面，有 max pooling 架構，但是，max pooling 這樣的架構顯然不能微分，若將其放在一個神經網路裡面，當進行 Gradient Descent 時，必須如何處理微分問題？第二個問題是我們剛才提到的 L1 的 Regularization，我們還沒有詳細解釋。以上兩個問題都會在這份教學解釋。

在這份教學中最重要的一個觀念是所謂 Deep learning 的流程。訓練一個 deep learning 的 network 的流程，應該是由以下三個步驟組成：定義 function set 和網路架構、決定 loss function、用 Gradient Descent 進行 optimization。做完以上的流程以後，會得到一個訓練好的 neural network。



接下來，第一件要檢查這個 neural network 在 training set 上有沒有得到好的結果。注意並不是檢查 testing set，而是要先檢查的是這個 neural network 在 training set 上，有沒有得到好的結果。如果沒有的話就必須回頭去檢查，在這三個步驟之中是不是哪邊出了問題。必須思考應該做怎樣的修改，才能在 training set 上得到好的結果。

這邊提到先檢查 training set 的表現，其實是深度學習一個非常獨特的地方。如果今天使用的是其他的方法，比如說 k-nearest neighbor 或 decision tree，在做完以後，其實不太有必要去檢查 training set 的結果，因為在 training set 上的正確率就是 100%，沒有什麼好檢查的。

所以，有人會說 deep learning 的 model 裡面這麼多參數，感覺很容易 overfitting 的樣子。但其實 deep learning 的方法，並不容易 overfitting。所謂的 overfitting 就是在 training set 上表現很好，但 testing set 上表現沒有那麼好。而上面提到的 k-nearest neighbor 和 decision tree，它們的 training set 上正確率都是 100%，這樣才算是非常容易 overfitting。

而對 deep learning 來說，overfitting 往往不是第一個會遇到的問題。這邊並不是說 deep learning 沒有 overfitting 的問題，而是說第一個會遇到的問題，會發生在 training 的時候。它並不是像 k-nearest neighbor 這種方法一樣，一進行訓練就可以得到非常好的正確率。它有可能在 training set 上，根本沒有辦法得到不錯的正確率。所以，這時就要回頭去檢查，在前面的步驟裡面，要做什麼樣的修改，才能在 training set 上得到不錯的正確率。

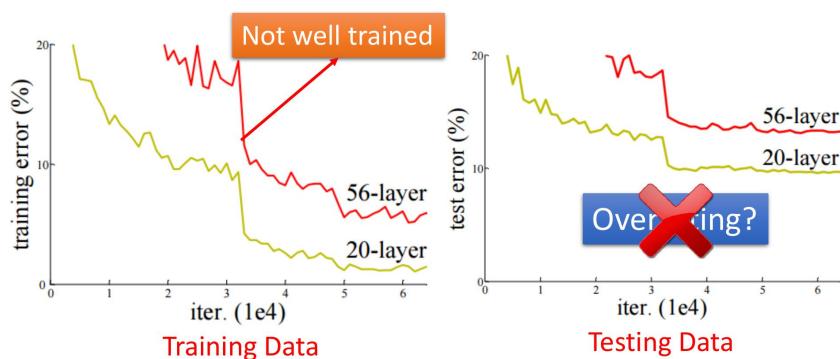
假設現在，已經在 training set 上得到好的表現，接下來才是把 network 套用在 testing set 上。其實要用 deep learning 在 training set 上得到 100% 的正確率沒有那麼容易，但可能你已經在 MNIST 上得到一個 99.8% 的正確率。而 testing set 上的表現才是大家最後真正關心的表現。

那把它套用到 testing set 上，這個神經網路會在 testing set 上有怎麼樣的表現呢？如果得到的是不好的結果，這種情形才是 Overfitting。在 training set 上得到好的結果，但在 testing set 上得到的是不好的結果，這種情形才能稱之為 Overfitting。

這時就要回頭用一些技巧試著去解決 overfitting 的問題。但有時想要用新的技巧去解決 overfitting 的問題時，反而會讓 training set 上的結果變壞。所以在做這一步的修改以後，還是要先檢查 training set 上的結果。如果 training set 上的結果變壞的話，就要從頭去對訓練的過程做一些調整。那如果同時在 training set 還有自己的 testing set 都得到好的結果的話，就可以把系統真正實際應用，這時就大功告成了。

這邊必須提到一個重點，不要看到所有不好的表現，就說是 overfitting。舉例來說，以下是文獻上的圖。

Do not always blame Overfitting



但在現實訓練中，也常常能夠觀察到類似的情況：右邊這張圖是在 testing set 上的結果。橫坐標是 model 在 Gradient Descent 時參數更新的次數；縱座標是 error rate，越低越好。在這張圖表中，20 層的 network 以黃線表示，而 56 層的 network 以紅線表示。可以觀察到 56 層的 network 的 error rate 比較高，它的表現比較差；而 20 層的 neural network，它的表現比較好。

有些人看到這張圖，就會得到一個結論：「56 層 model 的參數太多了。56 層果然沒有必要，這是 overfitting。」但是，真的能夠這樣說嗎？其實在說是 overfitting 之前，必須要先檢查一下在 training set 上的結果。對某些方法來說，不用檢查這件事，像剛剛提到的 k-nearest neighbor 或 decision tree。但是對 neural network 就必須檢查，因為有可能在 training set 上得到的結果是像左圖一樣。橫軸一樣是參數更新的次數，縱軸是 error rate。如果比較 20 層的 neural network 跟 56 層的 neural network 的話，會發現在 training set 上，20 層的 neural network 的表現本來就比 56 層好。56 層的 neural network 的表現其實是比較差的。

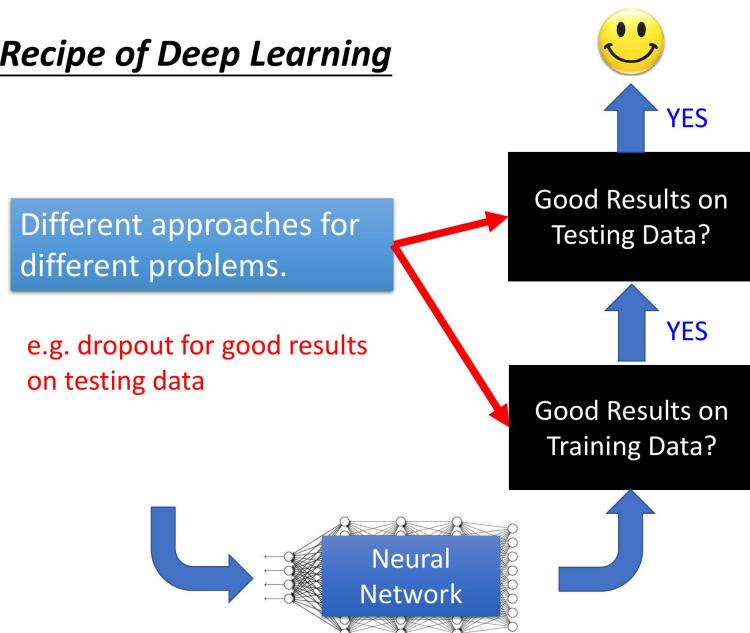
那為甚麼會發生這種情形呢？在訓練 neural network 的時候，有太多問題可能導致訓練結果不好。比如說，local minimum、saddle point，或者是 plateau 的問題。所以有可能這個 56 層的 neural network 在訓練的時候，它就卡在 local minimum，因此得到了一個較差的參數。這並不是 overfitting，而在訓練時就沒有訓練好。有人會說這種情況稱為 underfitting，但是這只是名詞定義的問題。我認為 underfitting 的意思應該是這個 model 的複雜度不足，或者說這個 model 的參數不夠多，所以它的能力不足以解出這個問題。

對這個 56 層的 neural network 來說，雖然它得到比較差的表現，但假如這個 56 層的 network 實際上是在 20 層的 network 後面另外加上 36 層的 network，那它的參數其實是比 20 層的 network 還多的。所以理論上，20 層的 network 可以做到的事情，56 層的 network 一定也可以做到。假設這個 56 層的 network，前面 20 層就做跟這個 20 層 network 完全相同，後面那 36 層就甚麼事都不做，當成是 identity matrix，那明明可以做到跟 20 層一樣的事情，為甚麼會做不到呢？

但是就是會有很多的問題使得這個 network 沒有辦法做到。56 層的 network 比 20 層的差，並不是因為它能力不夠，因為它只要前 20 層都跟 20 層的一樣，後面都是 identity 明明就可以跟 20 層一樣好，但它卻沒有得到這樣的結果。所以說它的能力其實是足夠的，所以我認為這不是 underfitting，就只是沒有訓練好，而我還不知道有沒有名詞專門指稱這個問題。

所以如果在 deep learning 的文獻上看到一個方法，永遠都必須要思考一下這個方法是要解決什麼樣的問題。因為在 deep learning 裡面有兩個問題：一個是 training set 上的表現不好，一個是 testing set 上的表現不好。當有一個方法被提出的時候，往往就是針對這兩個問題的其中一個處理。舉例來說，等一下會提到一個叫做 dropout 的方法。dropout 也許許多人或多或少都會知道，它是一個很有 deep learning 特色的方法。

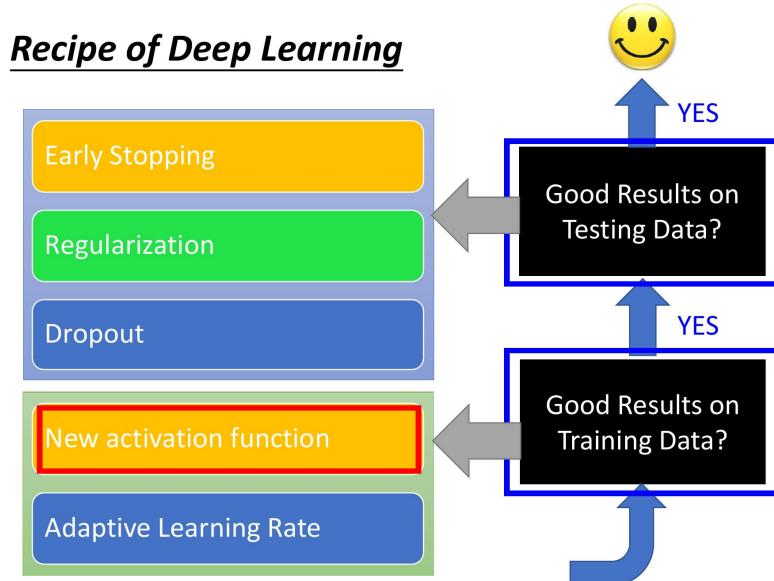
Recipe of Deep Learning



於是很多人就會說：「所以我今天只要看到 performance 不好，就可以決定要用 dropout。」但是，只要仔細思考一下 dropout 是甚麼時候用的，就會發現 dropout 是在 testing 的結果不好的時候才會使用的，而 testing data 結果好的時候是不會使用 dropout 的。所以如果今天問題是 training 的結果不好，而還是使用 dropout，只會越訓練越差而已。所以，不同的方法處理甚麼樣不同的問題，是必須要想清楚的。

臺灣大學人工智慧中心 科技部人工智慧技術暨全幅健康照護聯合研究中心 <http://aintu.tw>

我們剛才提到 deep learning 的流程裡面，在訓練的時候有兩個問題。所以接下來我們會對這兩個問題分開來討論，介紹在遇到這兩個問題的時候，有什麼解決方法。

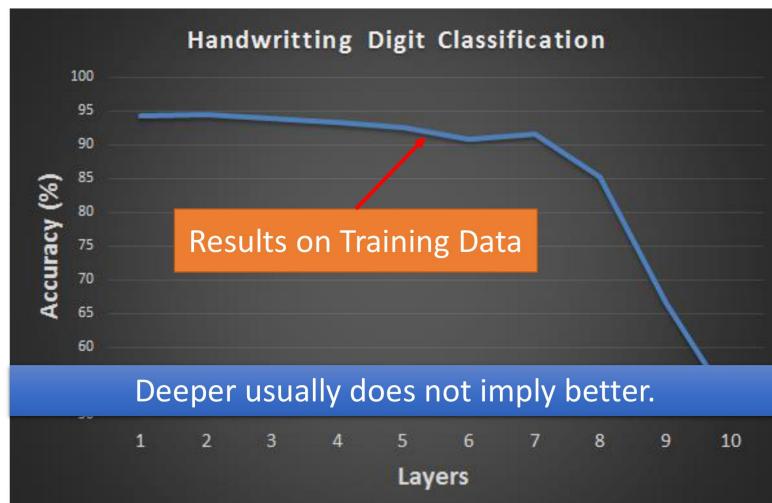


首先，如果在 training data 上的結果不好時，可以檢查一下是不是在 network 架構設計時設計得不好。舉例來說，可能 model 使用的 activation function 是比較不好的 activation function，或者說是對 training 比較不利的 activation function。可能可以透過換一些新的 activation function，得到比較好的結果。

我們知道，在 1980 年代的時候，比較常用的 activation function 是 sigmoid function。我們之前有稍微解釋為甚麼要使用 sigmoid function。今天如果我們使用 sigmoid function，其實你可能會發現越深的網路表現不一定比較好。

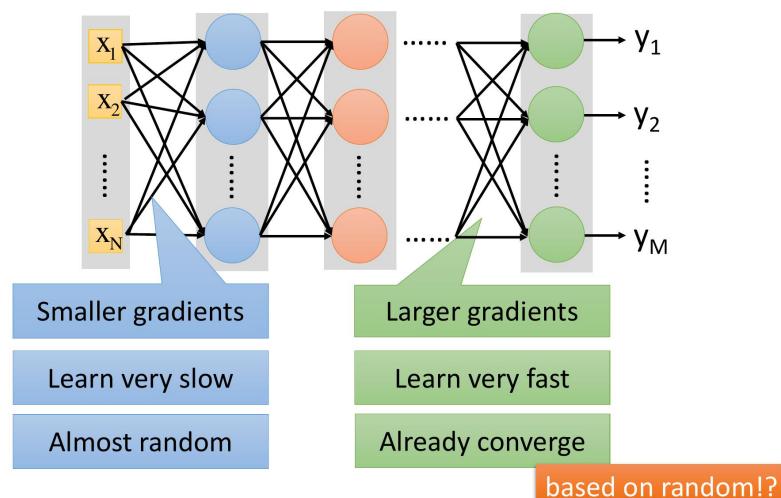
下圖是在 MNIST 手寫數字辨識上面的結果。當 layer 越來越多的時候，準確率一開始持平後來就變低了；當 layer 是 9 層、10 層時整個結果就崩潰了。有些人看到這張圖，就會說「9 層、10 層參數太多了，overfitting」如之前所說，這種情況並不是 overfitting。

Hard to get the power of Deep ...



為什麼呢？首先要檢查表現不好是不是來自於 overfitting 必須要看 training set 的結果。而這張圖表，是 training set 的結果，所以這並不是 overfitting。這個是訓練時，就訓練失敗了。其中一個原因叫做 Vanishing Gradient。當你把 network 疊得很深的時候，在最靠近 input 的幾個層的這些參數，對最後 loss function 的微分值會很小；而在比較靠近 output 的幾個層的微分值會很大。因此，當你設定同樣的 learning rate 時，會發現靠近 input 的地方參數更新的速度是很慢的；靠近 output 的地方參數更新的速度是很快的。

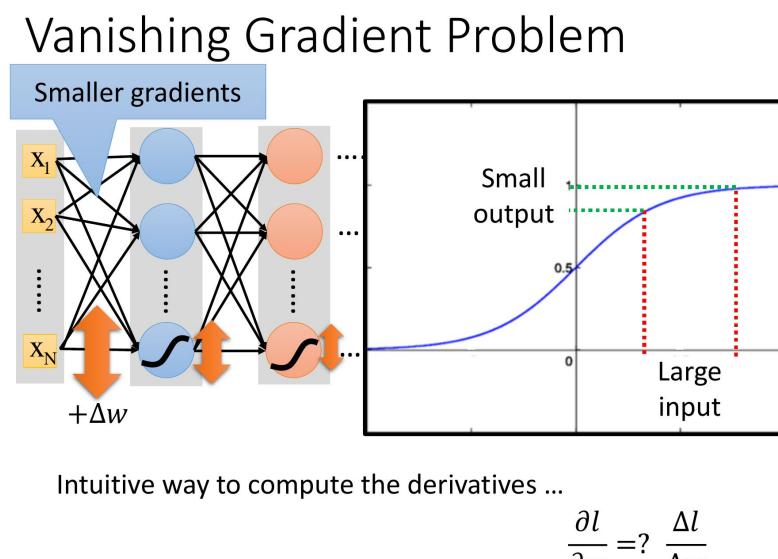
Vanishing Gradient Problem



所以，你會發現在 input 參數幾乎還是 random 的時候，output 就已經收斂了。在靠近 input 地方的這些參數還是 random 值時，output 的地方就已經根據這些 random 的結果找到一個 local minimum，然後就收斂了。這時你會發現 loss 下降的速度變得很慢，就覺得這個 model 參數卡在 local minimum 之類的，就傷心地把程式停掉了。此時你得到的結果其實是很差的，為什麼呢？因為這個收斂的狀態幾乎基於 random 的參數，所以得到的結果其實是很差的。

為甚麼會有這個現象發生呢？如果 Backpropagation 的式子寫出來的話便可以很輕易地發現 sigmoid function 會導致這種情況發生。但是就算不看 Backpropagation 的式子，從直覺上來想也可以了解為什麼這種情況會發生。用直覺來想，一個參數的 Gradient 的值應該是某一個參數 w 對 total cost C 的偏微分。也就是說，它直覺上的意思就是當我把某一個參數做小小的變化時，它對這個 cost 的影響。我們可以把一個參數做小小的變化，然後觀察它對 cost 的變化，而藉此來決定這個參數的 Gradient 的值有多大。

所以我們就把第一個 layer 裡面的某一個參數加上 Δw ，看看對 network 的 output 和 target 之間的 loss 有甚麼樣的影響。你會發現，如果今天這個 Δw 很大，在通過 sigmoid function 的時候是會變小的。也就是說，改變了某一個參數的 weight，對某一個 neuron 的 output 的值會有影響，但是這個影響會衰減。因為，假設用 sigmoid function，它的形狀長的如下圖：



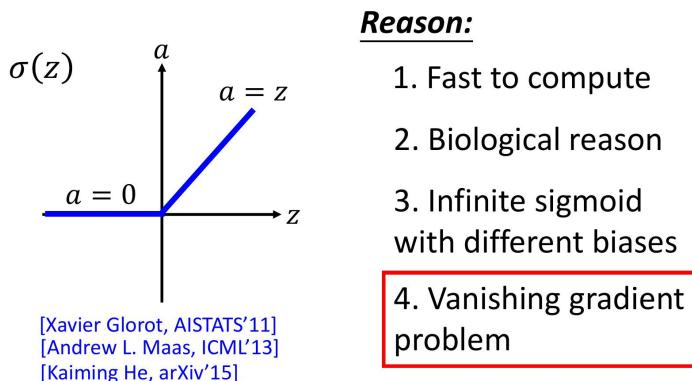
sigmoid function 會把負無窮大到正無窮大之間的值都硬壓到 0~1 之間。也就是說，如果有很大的 input 的變化，在通過 sigmoid function 以後，它對 output 的變化會很小。所以，就算今天這個 Δw 有很大的變化，造成 sigmoid function 的 input 有很大的變化，對 sigmoid function 來說，它的 output 的變化是會衰減的。而每通過一次 sigmoid function，變化就衰減一次。所以當 network 越深，它衰減的次數就越多，直到最後，它對 output 的影響是非常小的。換句話說，你在 input 的地方改一下你的參數，對它最後 output 的變化，其實是很小的，因此最後對 cost 的影響也很小。這樣會導致靠近 input 的那些 weight，它對 Gradient 的影響是小的。

那怎麼解決這個問題呢？比較早年的做法是去訓練 RBM，去做 layer-wise 的 training。也就是說，先訓練好一個 layer。因為如果把所有的這個 network 兜起來，那在做 Backpropagation 的時候，第一個 layer 幾乎沒有辦法被訓練到。所以，RBM 的精神就是：先把一個 layer train 好之後，再 train 第二個，再 train 第三個。最後在做 Backpropagation 的時候，雖然第一個 layer 幾乎沒有被 train 到也所謂，因為一開始在 pre-train 的時候，就把它 pre-train 好了。以上就是 RBM pre-train 為什麼可能有用的原因。

後來 Hinton 跟 Pengel 都幾乎在同樣的時間，不約而同地提出同樣的想法：改一下 activation function，可能就可以解決這個問題了。所以，現在比較常用的 activation function，叫做 Rectified Linear Unit，它的縮寫是 ReLU。這個 activation function 的函數圖形如下圖：

ReLU

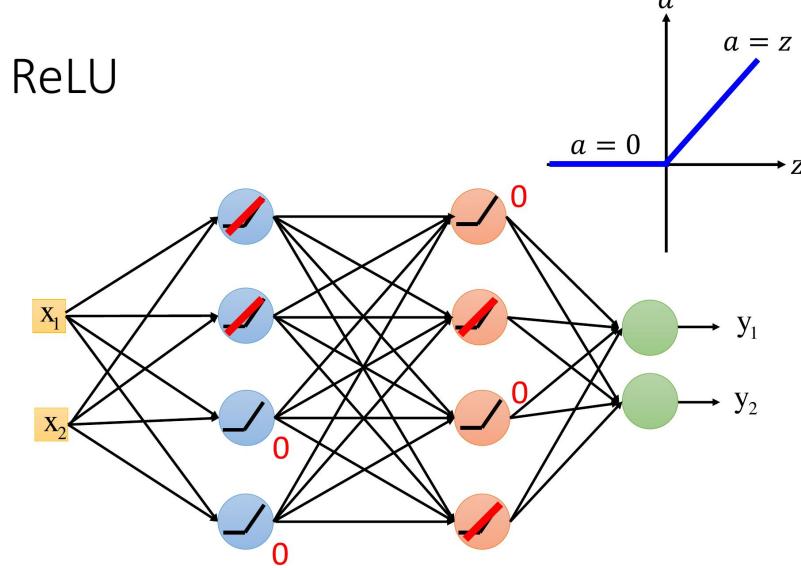
- Rectified Linear Unit (ReLU)



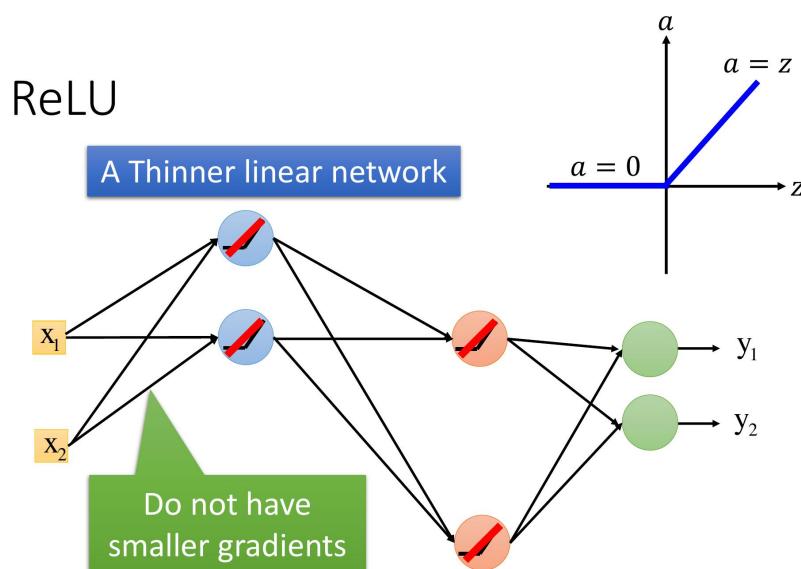
其中 z 是 activation function 的 input， a 是 activation function 的 output。如果 activation function 的 input 大於 0，input 就會等於 output；如果 activation function 的 input 小於 0，output 就是 0。

選擇這樣的 activation function 有甚麼好處呢？有以下幾個理由：第一個理由是它比較快，跟 sigmoid function 比起來，它的運算是快很多的。sigmoid function 裡面的 exponential 運算是很慢的，使用這個方法快得多。Pengel 的原始論文有提到這個 activation function 的想法其實有一些生命上的理由，而他把這樣的 activation 跟一些生物上的觀察結合在一起。

而 Hinton 則說過像 ReLU 這樣的 activation function 其實等同於無窮多的 sigmoid function 疊加的結果。那些無窮多的 sigmoid function 之中，它們的 bias 都不一樣，而疊加的結果會變成 ReLU 的 activation function。但它最重要的理由是它可以處理 Vanishing gradient 的問題。



我們可以觀察以上這個 ReLU 的 neural network。它裡面的每一個 activation function 都是 ReLU 的 activation function。ReLU 的 activation function 作用在兩個不同的 region：一個 region 是當 activation function 的 input 大於 0 時，input 會等於 output；另外一個 region 是 activation function 的 input 小於 0 時，output 就是 0。所以，現在每一個 ReLU 的 activation function，都作用在以上提到的兩個不同的 region。當 $\text{input} = \text{output}$ 時，這個 activation function 其實就是 linear 的；那對那些 output 是 0 的 neuron 來說，它其實對整個 network 是一點影響都沒有的：因為它 output 是 0，所以它根本就不會影響最後 output 的值。假如有一個 neuron 它 output 是 0 的話，根本就可以把它從 network 裡面整個拿掉。當你把這些 output 是 0 的 network 拿掉，剩下的 neuron，就都是 input 等於 output，也就是 linear 時，整個 network 不就可以如下圖一樣，看成是一個很瘦長的linear network 嗎？



我們剛才提到：Gradient 會遞減，是因為通過 sigmoid function 的關係，原因是 sigmoid function 會把比較大的 input 變成比較小的 output。但是，如果是 linear network 的話，input 等於 output，就不會有所謂 activation function 遞減的問題了。

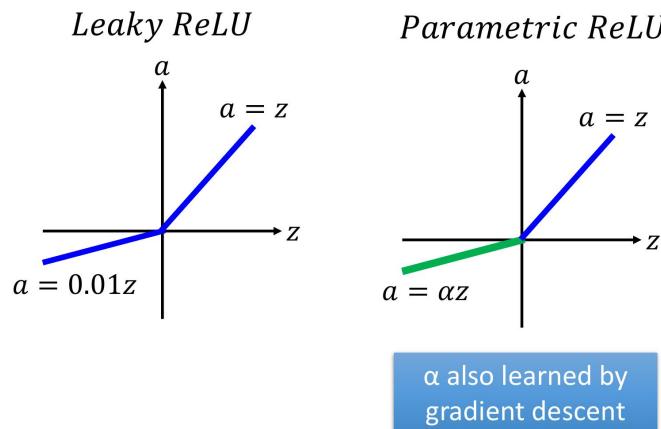
這邊有些人可能會有一個問題：如果用 ReLU 的話，整個 network 會變成 linear network，可是我們要的並不是一個 linear 的 network。我們之所以要用 deep learning，就是因為我們不想要我們的 function 是 linear 的，而是希望它是一個 non-linear、一個比較複雜的 function。當我們用 ReLU 的時候，它不就變成一個 linear 的 function 了嗎？這樣不是變得很弱嗎？

其實是因為這個 network 整體來說依舊是 non-linear 的。當每一個 neuron，它 operation 的 region 是一樣的時候，它是 linear 的，也就是說，如果對 input 做小小的改變，不去改變 neuron 的 operation 的 region 的話，它是一個 linear 的 function；但是如果對 input 做比較大的改變，改變了 neuron 的 operation region 的話，它就變成是 non-linear 的。

還有另外常被問到的問題：ReLU 不能微分。我們之前提到，在做 Gradient Descent 時，需要對 loss function 做微分。也就是說，neural network 要是一個可微的 function，但是 ReLU 不可微分，至少在原點是不可微的。那我們應該怎麼辦呢？

其實在實作上可以這樣解決：當 region 在原點右側的時候，gradient 就是 1；region 在原點左側的時候，微分就是 0。因為不可能會發生 input 正好是 0 的情況，就忽略它。

ReLU - variant



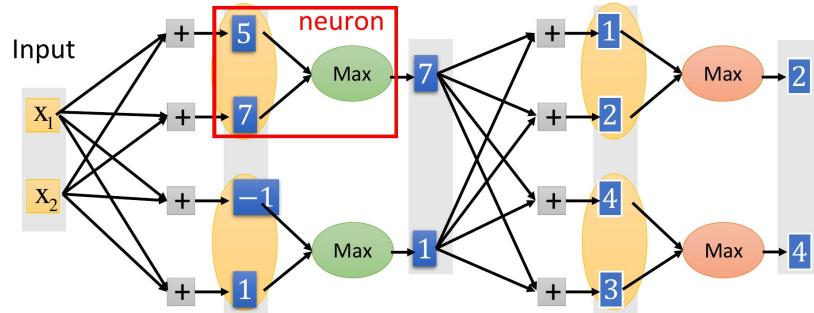
而 ReLU 其實還有各種變體。有人覺得原來的 ReLU 在 input 小於 0 的時候 output 會是 0，在這個時候微分是 0，就沒有辦法 update 參數了。所以，我們應該在 input 小於 0 的時候，output 還是有一點點的值，也就是說 input 小於 0 的時候，output 是 input 乘上 0.01，這個函數叫做 Leaky ReLU。

那此時，有人就會提出問題：為什麼是 0.01，而不是 0.07, 0.08 之類的數值呢？所以，就有人提出了 Parametric ReLU：在負的這側呢 $a = z \times \alpha$ 。 α 是一個 network 的參數，它可以透過 training data 被學出來，甚至每一個 neuron 都可以有不同的 α 的值。

Maxout

ReLU is a special cases of Maxout

- Learnable activation function [Ian J. Goodfellow, ICML'13]



You can have more than 2 elements in a group.

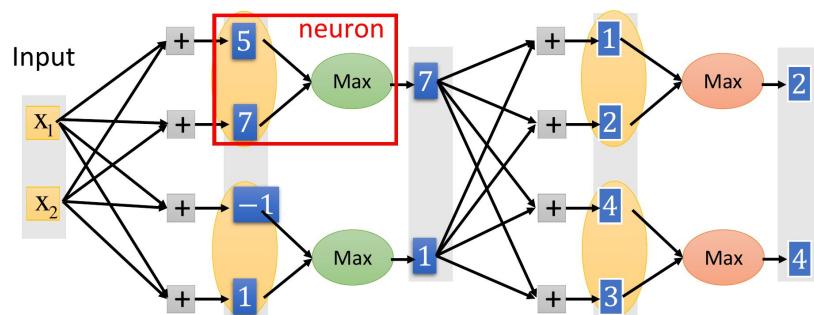
那又會有人問為甚麼一定要是 ReLU 這個樣子呢？所以，後來又有一個更進階的想法叫做 Maxout network。Maxout network 會讓 network 自動學它的 activation function。而因為現在 activation function 是自動學出來的，所以 ReLU 就只是 Maxout network 的一個 special case。也就是說 Maxout network 可以學出像 ReLU 這樣的 activation function，但是也可以學出其他的 activation function，training data 會決定現在的 activation function 應該要長甚麼樣子。

假設現在 input 是一個 2 dimension 的 vector $[x_1, x_2]$ 。然後把 $[x_1, x_2]$ 乘上不同的 weight 分別得到四個 value : 5, 7, -1, 1。本來這四個值應該要通過 activation function，不管是 sigmoid function 還是 ReLU，來得到另外一個 value。但是在 Maxout network 裡面我們會把這些 value group 起來，而哪些 value 應該被 group 起來這件事情是事先決定的。比如說，在這個例子中以上這兩個 value 是一組，以下這兩個 value 是一組，然後在同一個組裡面選一個值最大的當作 output。比如說，上面這個組就選 7，而下面這個組就選 1。

Maxout

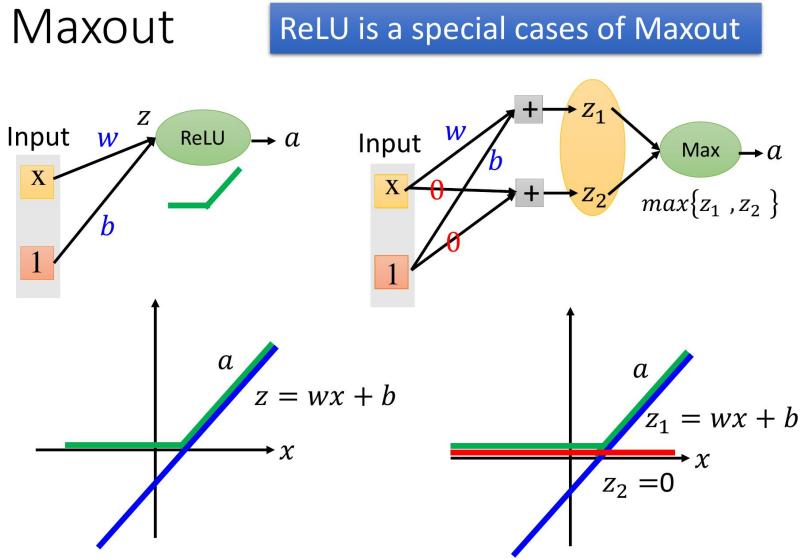
ReLU is a special cases of Maxout

- Learnable activation function [Ian J. Goodfellow, ICML'13]



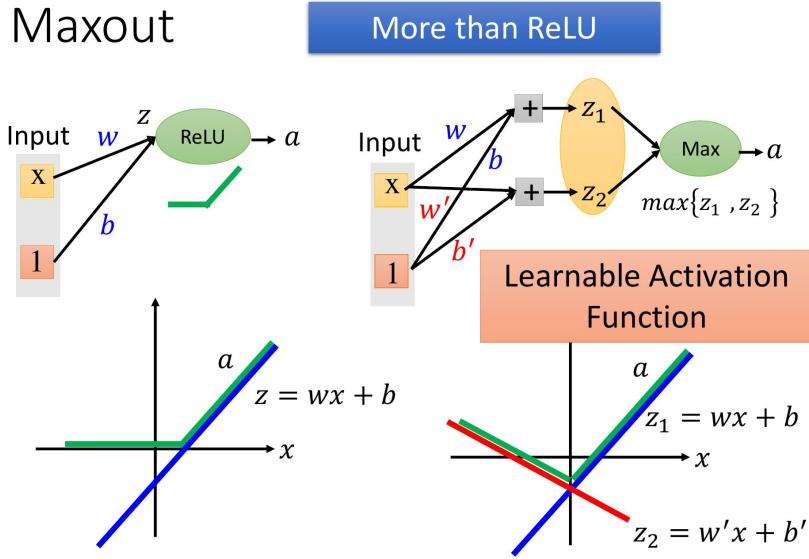
You can have more than 2 elements in a group.

這件事情其實就跟 Max Pooling 一樣，只是我們現在不是在 image 上做 Max Pooling，而是在一個 layer 上做 Max Pooling。我們把本來要放到 neuron 的 activation function 的這些 input 的值 group 起來，然後只選 max 當作 output，這樣就不用 activation function，得到的值是 7 跟 1。這個作法就是一個 neuron，只是它的 output 是一個 vector，而不是一個值。那接下來這兩個值乘上不同的 weight，就會得到另外一排不同的值，然後一樣把它們做 grouping。我們一樣從每個 group 裡面選最大的值：1 跟 2 就選 2，4 跟 3 就選 4。在實作上，幾個 element 要不要放在同一個 group 裡面，是你可以自己決定的。這就跟 network structure 一樣，是你自己需要調的參數。所以，你可以不是兩個 element 放一組，而是 3 個、4 個、5 個都可以。



我們剛剛提到 Maxout network 有辦法做到跟 ReLU 一模一樣的事情，可以模仿 ReLU 這個 activation function。怎麼做呢？我們知道說，假設我們這邊有一個 ReLU 的 neuron，它的 input 就一個 value x 。你會把 x 乘上這個 neuron 的 weight w 後再加上 bias b 最後通過 activation function ReLU 得到 a 。所以假設 x 是橫軸， z 是縱軸，這個圖形就是 $w \times x + b$ ， z 跟 x 之間的關係是 linear 的。而 a 跟 z 有甚麼樣的關係呢？因為現在通過的是 ReLU 的 activation function，所以如果 z 的值大過 0 的時候， $a = z$ ； z 的值小於 0 的時候， a 就是 0。所以在原點右側， $a = z$ ；在原點左側， $a = 0$ 。

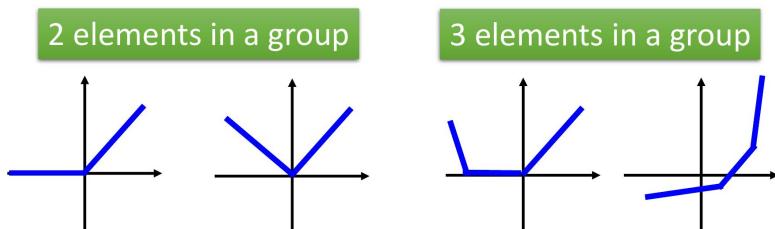
如果我們今天改用 Maxout network，我們會把 input x 乘上 weight w 再加上 bias b 得到 z_1 。再把 x 乘上另外一組 weight、加上另外一個 bias，得到 z_2 。假設另外一個 weight 跟另外一個 bias 都是 0，所以 $z_2 = 0$ 。然後 Max Pooling 會選 z_1 和 z_2 其中一個比較大的值當作 a 。如果觀察 z_1 跟 x 之間的關係，就是右圖的藍色線；如果觀察 z_2 跟 x 之間的關係，就是右圖的紅色線，因為 z_2 總是 0。Maxout 會選 z_1 和 z_2 之中選一個大的當作 output a 。所以，如果 x 是在紅藍線交叉點右側的 region 時， a 就會等於 z_1 ；如果 x 是在左側的 region 時， a 就會等於比較大的 z_2 。只要把 Maxout 中的 w 和 b ，等於 ReLU 中的 w 和 b ，就可以讓 ReLU 的 input 和 output 等於這個 Maxout network 的 input 和 output 的關係。由此可知，ReLU 是 Maxout network 可以做到的事情，只要它設定出正確的參數。



但是 Maxout network 也可以做出更多的、不同的 activation function。比如說，現在假設這兩個 weight 不是 0，而是 w' 和 b' ，就會得到藍色這條線 z_1 和紅色這條線 z_2 。因為 w' 和 b' 是不一樣的值，所以它可能是另外一條斜直線，如右圖。接下來 Max Pooling 的時候會在 z_1 和 z_2 之中選一個大的。所以，在交點的右側會選 z_1 ；在交點左側會選 z_2 ，於是就得到了一個不一樣的 activation function。而這個 activation function 長甚麼樣子是由 network 的參數 w, b, w', b' 決定，所以這個 activation function 是一個 Learnable 的 activation function。也就是說它是一個可以根據 data 去 generate 出來的 activation function。每一個 neuron 根據不同的 weight，可以有不同的 activation function。也就是說，它可以做出任何 piecewise 的 linear 的 convex activation function，如果觀察一下它的性質，便不難理解這件事情。

Maxout

- Learnable activation function [Ian J. Goodfellow, ICML'13]
 - Activation function in maxout network can be any piecewise linear convex function
 - How many pieces depending on how many elements in a group

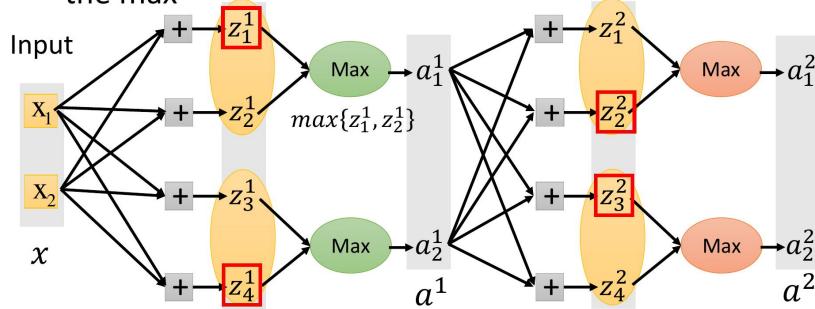


至於這個 piecewise 的 linear function 裡有多少個 piece，取決於我們把多少個 element 放在一個 group。假如說兩個 element 一個 group，可能可以有長的像第一張圖的 activation function，也就是 ReLU；也可以有一個 activation function 的作用就是取絕對值。假設是 3 個 element 一個 group，可以有第三張圖的 activation function；也可以有第四張圖的 activation

function 等等。

Maxout - Training

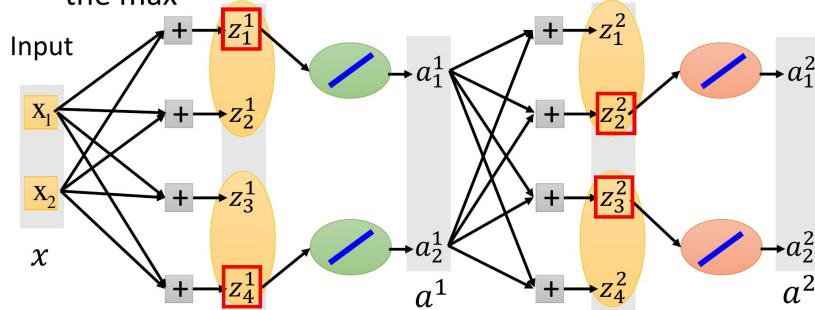
- Given a training data x , we know which z would be the max



接下來的另外一個問題，就是 Maxout 要怎麼 train。因為 Maxout 裡面有個 Max 函數，而它不能被微分。我們把這個 group 裡面比較大的值，用紅框表示。假設 z_1^1 和 z_2^1 這兩個值中比較大的是 z_1^1 ，那這個比較大的值就會等於這個 max operation 的 output。所以， z_1^1 等於 a_1^1 ， z_4^1 等於 a_2^1 ， z_2^2 等於 a_1^2 ， z_3^2 等於 a_2^2 。

Maxout - Training

- Given a training data x , we know which z would be the max



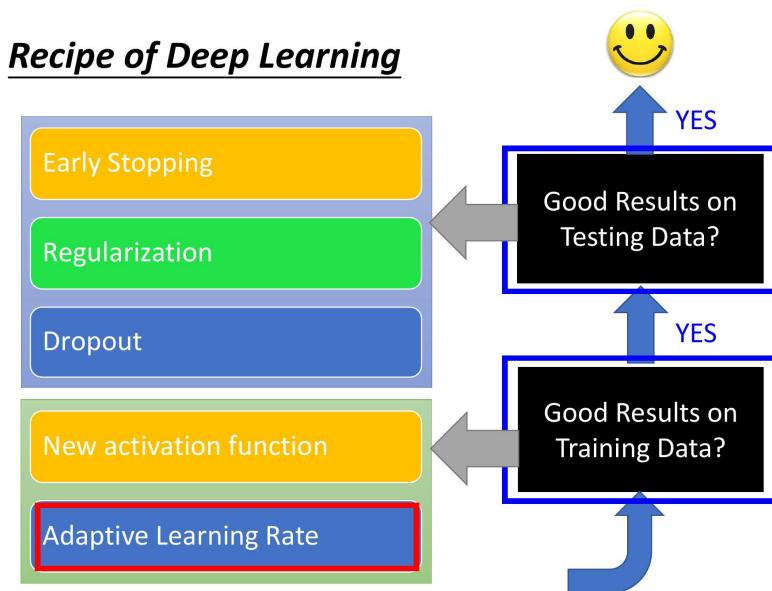
- Train this thin and linear network

Different thin and linear network for different examples

所以 max operation 其實在這邊就是一個 linear 的 operation，只是它只接給前面這個 group 裡面的某一個 element。也就是說其實這些沒有被接到的 element，它就不會影響 network 的 output，所以就可以把它們從 network 裡拿掉。其實給 Maxout 一個 input 時，也只是得到一個比較細長的 linear network。所以在 train 的時候就是在 train 這個比較細長的 linear network 裡面的參數，也就是連到紅框 element 的這些參數。如果是一個這樣子的 linear network，我們用 Backpropagation train 就能 train。

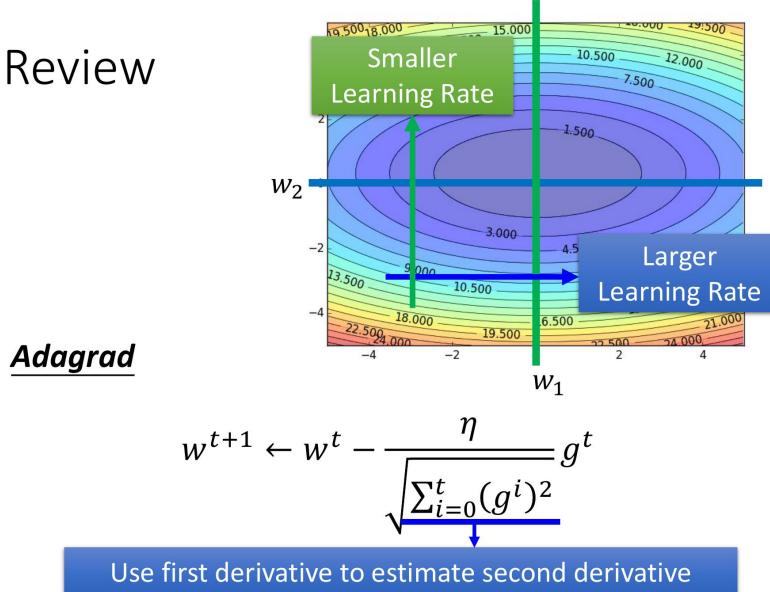
這個時候呢就會有一個問題：沒被 train 到的 element 怎麼辦呢？如果某一個 element 不是最大的值，那它連接的那些 weight，就不會被 train 到了嗎？在做 Backpropagation 的時候，只會 train 在這個圖上連到紅框 element 的這些實線，不會 train 到其他的 weight。那這些 weight 不就沒被 train 到了嗎？

這看起來表面上是一個問題，但實作上並不是一個問題，因為當你給它不同 input 的時候，得到的這些 z 的值是不一樣的：你給它不同 input 的時候， \max 的值是不一樣的。所以，每一次你給它不同的 input 的時候這個 network 的 structure 都是不一樣的因為我們有很多很多筆 training data，而 network 的 structure 不斷地變換，所以最後每一個 weight 在實際上都會被 train 到。所以回到 Max Pooling 來討論，Max Pooling 跟 Maxout 是一模一樣的 operation，你會 train Maxout，你就會 train Max Pooling，兩者是一模一樣的作法。



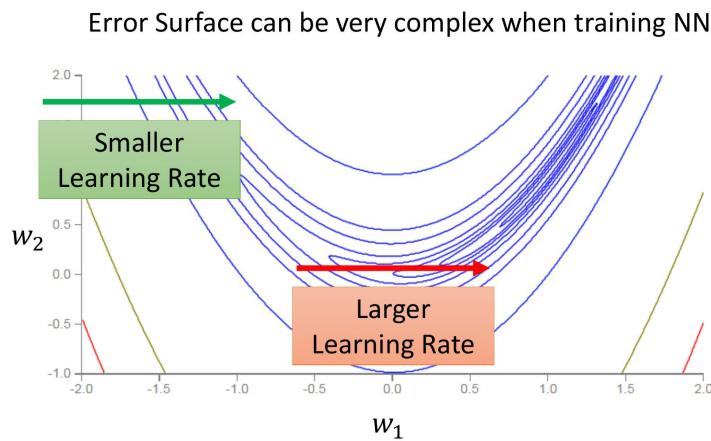
另外一個我們要介紹的是 adaptive 的 learning rate。其實 adaptive 的 learning rate，之前已經有介紹過：之前提到過的 Adagrad，做法就是每一個 parameter 都要有不同的 learning rate。於是我們就把一個固定的 learning rate η 除掉這一個參數過去所有 gradient 值的平方和開根號，就得到新的 parameter。

Review



而 Adagrad 的精神就是假設我們今天考慮兩個參數 w_1 和 w_2 ， w_1 是在水平的方向上，它平常 gradient 都比較小，那它是比較平坦的，給它比較大的 learning rate。反過來說，在垂直方向上平常 gradient 都是比較大的，所以它是比較陡峭的，給它比較小的 learning rate。但是實際上我們面對的問題有可能是比 Adagrad 可以處理的問題更加複雜的。也就是說，我們之前在做這個 Linear Regression 的時候，我們觀察到的 optimization 的 loss function 是像上圖的 convex 的形狀。但實際上，當我們在做 deep learning 的時候，這個 loss function 可以是任何形狀。比如說，它可以像下圖一樣是怪異的月形的形狀。

RMSProp



如果 error surface 是這個形狀，那有個問題是就算是同一個方向上，learning rate 也必須要能夠快速地變動。我們剛才在做 convex function 的時候，如果某個方向很平坦，就一直很平坦；某個方向很陡峭，就一直很陡峭。但是，如果今天在處理更複雜的問題的時候，可能你考慮 w_1 變改是在水平方向，在某個區域它很平坦，所以它需要比較小的 learning rate；但是到了另外一個區域，它又突然變得很陡峭，這個時候，它需要比較大的 learning rate。所以，真正要處理 deep learning 的問題，用 Adagrad 可能是不夠的，需要有更動態的調整 learning rate 的方

法。

RMSProp

$$\begin{aligned} w^1 &\leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 & \sigma^0 &= g^0 \\ w^2 &\leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 & \sigma^1 &= \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2} \\ w^3 &\leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 & \sigma^2 &= \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2} \\ &\vdots && \\ w^{t+1} &\leftarrow w^t - \frac{\eta}{\sigma^t} g^t & \sigma^t &= \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2} \end{aligned}$$

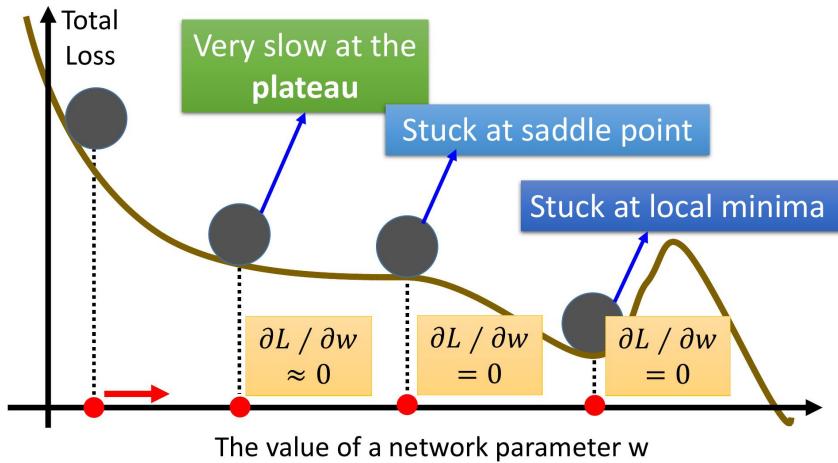
Root Mean Square of the gradients
with previous gradients being decayed

所以有一個 Adagrad 的進階版叫做 RMSProp。RMSProp 是一個滿神奇的方法，因為大家都找不到它的 paper。這個方法在 Hinton 的線上課程裡被提出，所以大家要 cite 的時候，要 cite 那個線上課程的連結。而 RMSProp 還真的有用，它是這樣做的：把這個固定的 learning rate 除以 σ 。在第一個時間點， σ 就是第一個算出來的 gradient 的值 g^0 ；在第二個時間點算出一個新的 gradient 的值 g^1 ，此時新的 σ 的值 σ^1 ，就是原來的 σ 值的平方，乘上 α 後再加上新的 g 的值 $(g^1)^2$ 乘上 $(1-\alpha)$ 後開根號，而這個 α 的值是可以自由去調整的。在下一個時間點，我們又算出 g^2 ，而 σ^2 是把原來的 σ^1 取平方乘上 α 再加上 $(1-\alpha)$ 乘上 $(g^2)^2$ 後再開根號。

RMSProp 和 Adagrad 不一樣的地方是 Adagrad 在分母放的值就是把 g^0, g^1, g^2 都取平方和開根號。但是在 RMSProp 裡面，這個 σ^1 包含了 g^0 跟 g^1 ，而 σ^2 的右邊這項包含了 g^2 。所以 σ^2 根號裡面也同樣包含了 g^0, g^1, g^2 ，就跟 Adagrad 一樣。但是在 RMSProp 可以給它乘上 weight α 或者是 $(1-\alpha)$ 。所以可以調整這個 α 的值，而這個 α 的值就也是像 learning rate，是我們要手動調整的值，可以設個 0.9 之類的數值。如果把這個 α 的值設的小一點，那就代表傾向於相信新的 gradient 所告訴我們的 error surface 平滑或陡峭的程度，比較無視舊的 gradient 提供給你的 information。

在第 t 個時間點， σ 就是把 $(\sigma^{t-1})^2$ 乘上 α 加上 $(1-\alpha)$ 乘上在第 t 個時間點算出來的 gradient 的平方。所以做 RMSProp 時一樣是在算 gradient 的 zooming square。但是，我們可以給現在已經看到的 gradient 比較大的 weight，而給過去看到的 gradient 比較小的 weight。

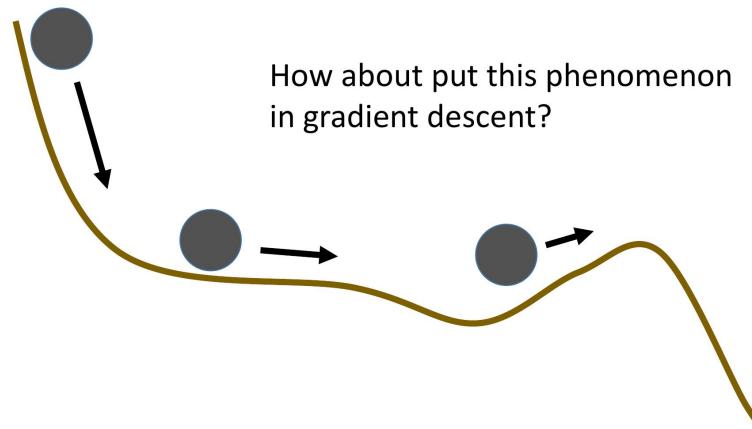
Hard to find optimal network parameters



除了 learning rate 的問題以外，我們知道在做 deep learning 的時候會卡在 local minimum，之前也有提過不見得是卡在 local minimum，也有可能卡在 saddle point，甚至是卡在 plateau 的地方。大家對這個問題都非常的擔心，覺得 deep learning 是非常困難的，因為可能胡亂做一下就產生很多問題。其實 Yann LeCun 在 2007 年的時候，提出一個滿特別的說法：不用擔心 local minimum 的問題。其實在 error surface 上沒有太多 local minimum，因為要是一個 local minimum，必須在每一個 dimension 都要是一個谷底的形狀。假設山谷的谷底出現的機率是 p ，因為 network 有非常多非常多的參數，所以假設有 1000 個參數，每一個參數都要是山谷的谷底的機率就是 p^{1000} 。network 越大、參數越多，出現的機率就越低。所以 local minimum 在一個很大的 neural network 裡面，其實沒有想像中的那麼多。一個很大的 neural network，它看起來其實搞不好是很平滑的，根本沒有太多 local minimum。所以當走到一個你覺得是 local minimum 的地方而卡住的時候，它八成就是 local minimum，或是很接近 local minimum。

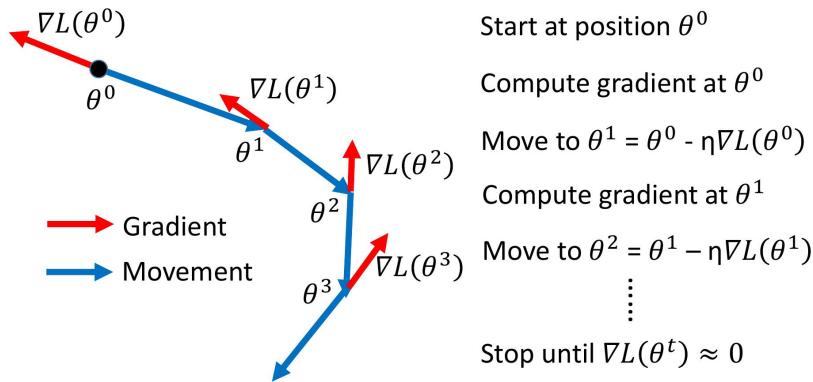
In physical world

- Momentum



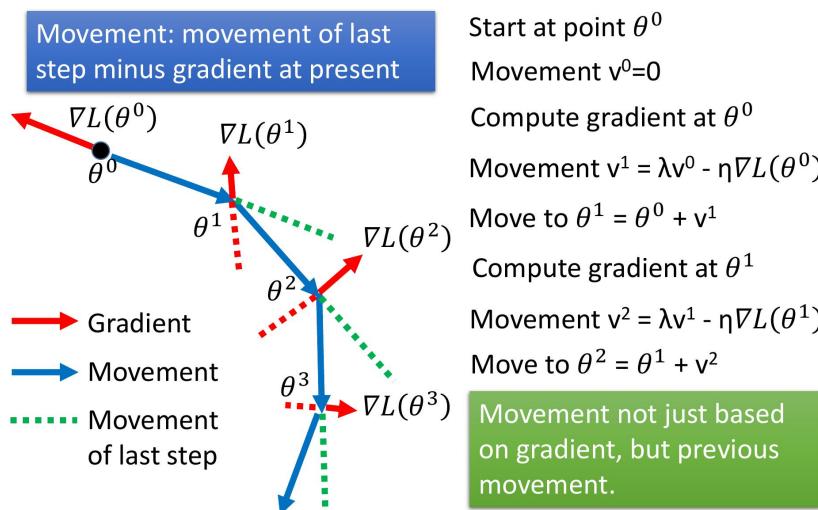
有一個 heuristic 的方法，可以稍微處理上述說的 local minimum 還有 plateau 的問題，這個方法可以說是從真實的世界得到的一些靈感。我們知道在真實的世界裡面，把一個球從上圖的左上角讓它滾下來，在球滾到 plateau 的地方時，因為有慣性，所以它不會停下來，而是會繼續往前。就算是走到上坡的地方，假設這個坡沒有很陡，因為慣性的關係，可能還是可以翻過這個山坡，結果它就走到了比這個 local minimum 還要好的地方。

Review: Vanilla Gradient Descent



所以我們就把這個慣性的特性加到 Gradient Descent 裡面去，這就叫做 momentum。我們先很快地複習一下一般的 Gradient Descent 是怎麼做的：選一個初始的值並計算一下它的 gradient，假設它的 gradient 是左上角紅色箭頭的方向，那我們就走 gradient 的反方向乘上一個 learning rate η 得到 θ^1 ，再算 gradient 之後再走一個新的方向，以此類推。直到 gradient = 0 的時候，或 gradient 趨近 0 的時候就停止。

Momentum



當加上 momentum 的時候，每一次移動的方向不再是只有考慮 gradient，而是現在的 gradient 加上在前一個時間點移動的方向。聽起來可能很抽象，所以我們實際用圖來看一下，它是怎麼運作的：一樣選一個初始值 θ^0 ，然後我們用 v 去記錄在前一個時間點移動的方向。因為是初始值，之前沒有移動過，所以前一個時間點移動的方向是 0。接下來計算在 θ^0 地方的 gradient，算出來是紅色這個箭頭。然後要移動的方向並不是紅色箭頭的方向，而是前一個時間點的 movement v^0 再加上 negative 的 gradient，得到現在要移動的方向 v^1 。所以就像慣性一樣，如果我們之前走的方向是 v^0 ，那有一個新的 gradient，並不會讓你參數 update 的方向完全轉向，而是只會改變你的方向：因為有慣性的關係，所以原來走的方向還是有一定程度的影響。

我們再觀察下一個例子，會對 moment 的概念比較清楚。上一個時間點移動的方向是 v^1 。接下來再計算一下 gradient 就是第三個紅色箭頭，於是我們要決定，在第二個時間點我們要走的方向。而第二個時間點要走的方向是過去走的方向 v^1 減掉 leaning rate 乘上 gradient。如果我們觀察上圖，gradient 會告訴我們說要走紅色虛線方向，也就是說負的 η 乘上 gradient，要紅色虛線方向。但是，前面的 movement 是綠色的箭頭，我們會把這個 movement 乘上一個 λ ，而這個 λ 實際也是一個跟 learning rate 一樣，是要手動調整的參數。這個參數代表慣性的影響力有多大： λ 大的話，慣性影響力就大； λ 小的話，慣性影響力就小。總之，慣性告訴我們走綠色方向，gradient 告訴我們走紅色方向。這兩個合起來呢，就走了一個新的方向，就是藍色箭頭方向，而這就是 v_2 ，以此類推。新的 gradient 告訴我們走紅色虛線方向，慣性告訴我們走綠色虛線方向，合起來最後就是走藍色方向。然後 update 參數以後，gradient 告訴我們走紅色虛線方向，慣性告訴我們走綠色虛線方向，合起來就是走藍色的方向。

Momentum

Movement: movement of last step minus gradient at present

v^i is actually the weighted sum of all the previous gradient:
 $\nabla L(\theta^0), \nabla L(\theta^1), \dots \nabla L(\theta^{i-1})$

$$v^0 = 0$$

$$v^1 = -\eta \nabla L(\theta^0)$$

$$v^2 = -\lambda \eta \nabla L(\theta^0) - \eta \nabla L(\theta^1)$$

⋮

Start at point θ^0

Movement $v^0=0$

Compute gradient at θ^0

$$\text{Movement } v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$$

$$\text{Move to } \theta^1 = \theta^0 + v^1$$

Compute gradient at θ^1

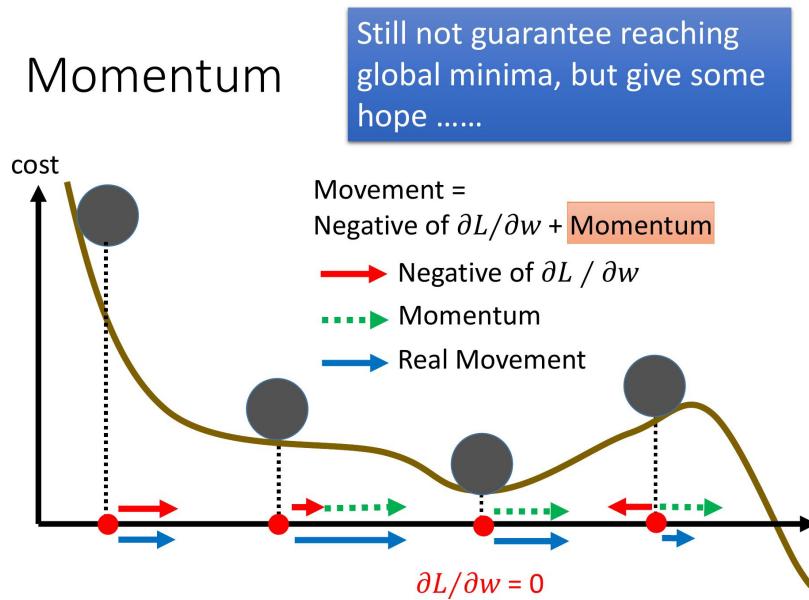
$$\text{Movement } v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$$

$$\text{Move to } \theta^2 = \theta^1 + v^2$$

Movement not just based on gradient, but previous movement

我們可以用另外一個方法來理解這件事情。其實在第 i 個時間點移動的量、方向 v^i ，其實就是過去所有算出來的 gradient 的總和。我們知道 $v^0 = 0$ ， v^1 是 $\lambda v^0 - \nabla L(\eta \theta^0)$ ，所以 $v^1 = -\nabla L(\eta \theta^0)$ 。而 v^2 就把 v^1 是 $-\nabla L(\eta \theta^0)$ 代進去。我們把 v^1 代到 v^2 這邊，乘上 λ ，再減掉 η 乘以 θ^1 的 gradient，得到結果就是把 θ^0 的地方的 gradient 乘上 $-\lambda \eta$ ，再減掉 η 乘上 θ^1 的 gradient，得到 v^2 。

所以 v^2 裡面同時有在 θ^0 算出來的 gradient，也同時有在 θ^1 的地方算出來的 gradient，只是這兩個 gradient，它的 weight 是不一樣的。如果 λ 都設小於 0 的值的話，越之前的 gradient，它的 weight 就越小，就越不去理它，會越在意現在的 gradient，但是過去的 gradient 也會對現在要 update 的方向有一定程度的影響力。這就是 momentum。



如果不太喜歡數學式子的話，我們從直覺上來觀察一下 Momentum 是怎麼運作的。在加入 Momentum 以後每一次移動的方向是 negative 的 gradient 加上 momentum 建議我們要走的方向。Momentum 實際上就是上一個時間點的 movement。所以，假設初始的參數是在這個位置，gradient 建議我們往右走，所以就往右移動。那如果說之後移到這個位置，gradient 建議我們往右走，而 momentum 也會建議我們往右走，因為我們上次是從左邊移過來的。所以前一個步伐我們是向右，如果考慮 momentum 的話，我們也會向右。把 gradient 建議我們走的方向跟 momentum 建議我們走的方向合起來，就會得到藍色的線，所以我們會繼續向右。如果我們今天走到 local minimum 的地方 gradient 是 0。所以 gradient 會告訴你說就停在這裡；但是，momentum 會告訴你，之前是從右邊走過來的，所以你仍然應該要繼續往右走，也就是綠色箭頭的方向。最後你參數 update 的方向，仍然會繼續向右。甚至你可以樂觀地期待如果今天在往右的時候，考慮 gradient 的話參數應該往左移動。但是，momentum 建議我們繼續向右走，因為你是從左邊過來的，所以 momentum 建議你繼續向右走。如果今天 momentum 實際上比較強的話，最後就還是會向右走。所以，有一定的可能性可以跳出 local minimum，如果這個 local minimum 不深的話，有可能藉由慣性的力量跳出 local minimum，然後走到比較低的 local minimum。

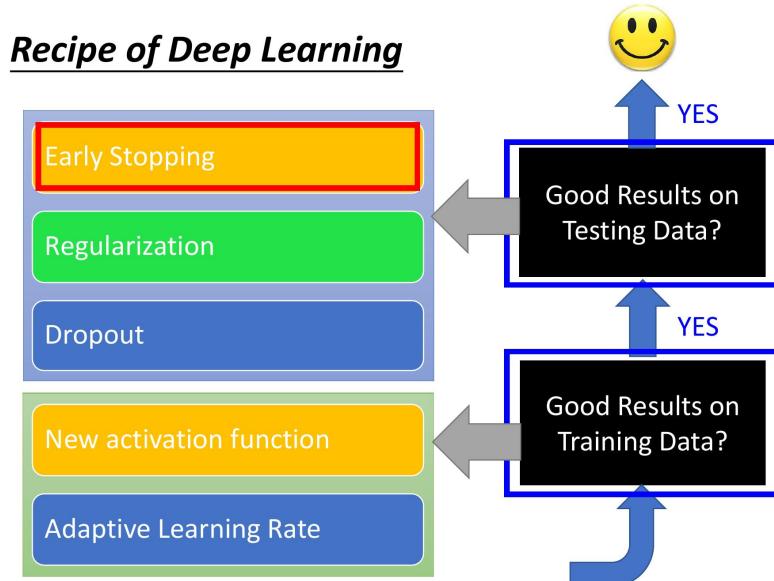
Adam

RMSProp + Momentum

Algorithm 1: Adam, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

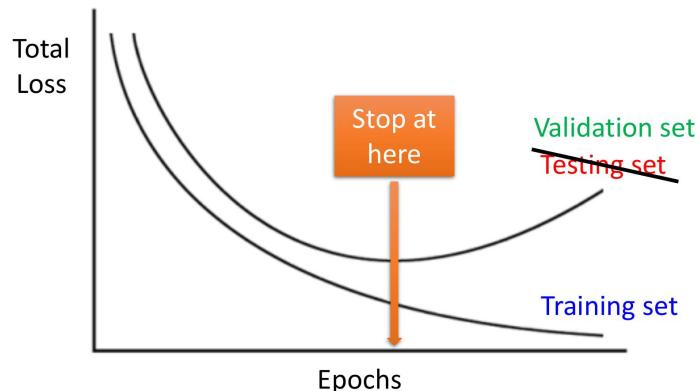
```
Require:  $\alpha$ : Stepsize  
Require:  $\beta_1, \beta_2 \in [0, 1]$ : Exponential decay rates for the moment estimates  
Require:  $f(\theta)$ : Stochastic objective function with parameters  $\theta$   
Require:  $\theta_0$ : Initial parameter vector  
 $m_0 \leftarrow 0$  (Initialize 1st moment vector) → for momentum  
 $v_0 \leftarrow 0$  (Initialize 2nd moment vector) → for RMSprop  
 $t \leftarrow 0$  (Initialize timestep)  
while  $\theta_t$  not converged do  
     $t \leftarrow t + 1$   
     $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )  
     $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)  
     $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)  
     $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)  
     $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)  
     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)  
end while  
return  $\theta_t$  (Resulting parameters)
```

如果把 RMSProp 加上 momentum 的話，其實你就會得到 Adam 這樣子的算法。我們快速地觀察一下上圖的式子。在這個式子裡，一開始要先初始一個東西叫做 m_0 ，也就是 momentum，前一個時間點的 movement。這邊有另外一個值叫做 v_0 ，就是我們剛才在 RMSProp 裡面看到的那個 σ ，就是之前 gradient 的 zooming square，也就是之前算出來的 gradient 的平方和。它先算一下 gradient g_t ，然後根據 g_t 就可以算出 m_t ，也就是現在要走的方向，是考慮過去要走的方向再加上 gradient。接下來，算一下要放在分母的地方的 v_t ，是過去、前一個時間點的 v_t 加上 gradient 的平方，後面要開根號。此外它做了一個原來 RMSProp 跟 momentum 裡面沒有的東西，叫做 bias correction。它會把 m_t 跟 v_t 都除上一個值，這個值一開始會比較小，後來會越來越接近 1，至於為什麼這麼做，原始論文裡面有提出它的理由。最後在 update 的時候，把 momentum 建議的方向 \hat{m}_t 乘上 learning rate α ，再除掉 RMSProp normalize 以後，建議的 learning rate。最後，得到 update 的方向。



我們剛才所討論的都是如果在 training data 上的結果不好的話怎麼辦。接下來要討論的則是如果今天已經在 training data 上得到夠好的結果，但是在 testing data 上的結果仍然不好，那有甚麼可行的方法。接下來會很快介紹 3 個方法：Early Stopping、Regularization 跟 Dropout。Early Stopping 跟 Regularization 是很 typical 的作法，他們不是 specific design for deep learning 的，是一個很傳統、typical 的作法。而 Dropout 是一個滿有 deep learning 特色的做法，在介紹 deep learning 的時候，需要討論一下。

Early Stopping



Keras: <http://keras.io/getting-started/faq/#how-can-i-interrupt-training-when-the-validation-loss-isnt-decreasing-anymore>

首先我們來介紹一下 Early Stopping，Early Stopping 是甚麼意思呢？我們知道，隨著你的 training，如果 learning rate 調的對的話，total loss 通常會越來越小；那如果 rate 沒有設好，loss 變大也是有可能的。想像一下如果 learning rate 調的很好的話，那在 training set 上的 loss 應該是逐漸變小的。但是因為 training set 跟 testing set 他們的 distribution 並不完全一樣，所以有可能當 training 的 loss 逐漸減小的時候，testing data 的 loss 却反而上升了。

所以理想上，假如知道 testing data 的 loss 的變化，我們應該停在不是 training set 的 loss 最小、而是 testing set 的 loss 最小的地方。在 train 的時候，不要一直 train 下去，可能 train 到中間這個地方的時候，就要停下來了。但是實際上，我們不知道 testing set，根本不知道 testing set 的 error 是甚麼。所以我們其實會用 validation set 來 verify 這件事情。這邊的 testing set，並不是指真正的 testing set。它指的是有 label data 的 testing set。比如說，如果你今天是在做作業的時候這邊的 testing set 可能指的是 Kaggle 上的 public set；或者是，你自己切出來的 validation set。但是我們不會知道真正的 testing set 的變化所以其實我們會用 validation set 模擬 testing set 來看甚麼時候是 validation set 的 loss 最小的時候，並且把 training 停下來。

Regularization

- New loss function to be minimized
 - Find a set of weight not only minimizing original cost but also close to zero

$$L'(\theta) = \underline{L(\theta)} + \lambda \frac{1}{2} \|\theta\|_2 \rightarrow \text{Regularization term}$$

↓
 Original loss
 (e.g. minimize square error, cross entropy ...) $\theta = \{w_1, w_2, \dots\}$
L2 regularization:
 $\|\theta\|_2 = (w_1)^2 + (w_2)^2 + \dots$
 (usually not consider biases)

那 Regularization 是甚麼呢？我們重新定義了那個我們要去 minimize 的 loss function。我們原來要 minimize 的 loss function 是 define 在你的 training data 上的，比如說要 minimize square error 或 minimize cross entropy。那在做 Regularization 的時候我們會加另外一個 Regularization 的 term，比如說，它可以是你的參數的 L2 norm。假設現在我們的參數 θ 裡面，它是一群參數， w_1, w_2 等等有一大堆的參數。那這個 θ 的 L2 norm 就是 model 裡面的每一個參數都取平方然後加起來，也就是這個 $\|\theta\|_2$ 。因為現在用 L2 norm 來做 Regularization，所以我們稱之為 L2 的 Regularization。

我們之前有提過，在做 Regularization 的時候一般是不會考慮 bias 這項。因為加 Regularization 的目的是為了要讓我們的 function 更平滑，而 bias 通常跟 function 的平滑程度是沒有關係的。所以，通常在算 Regularization 的時候不會把 bias 考慮進來。

L2 regularization:
 Regularization $\|\theta\|_2 = (w_1)^2 + (w_2)^2 + \dots$

- New loss function to be minimized

$$L'(\theta) = L(\theta) + \lambda \frac{1}{2} \|\theta\|_2 \quad \text{Gradient: } \frac{\partial L'}{\partial w} = \frac{\partial L}{\partial w} + \lambda w$$

Update: $w^{t+1} \rightarrow w^t - \eta \frac{\partial L'}{\partial w} = w^t - \eta \left(\frac{\partial L}{\partial w} + \lambda w^t \right)$

$$= (1 - \eta \lambda) w^t - \eta \frac{\partial L}{\partial w}$$

Weight Decay

↓
Closer to zero

如果我們把這個新的 objective function L' ，也就是 L 加上 parameter 的 L2 norm 做 gradient 的話，我們會得到 L' 對某一個參數 w 的偏微分，會等於 L 對某個參數的偏微分加上 λ 乘上某一個參數。因為 $\|\theta\|_2$ 是所有參數的平方和，所以把這項對某個參數 w 做偏微分得到的結果就是 w 。所以，你現在 update 參數的式子會變成下方這樣。本來我們 update 的式子是把原來的參數減掉 η 乘上 loss function 對 w 的偏微分，得到新的參數。現在這個新的 loss function 所造成的 $\frac{\partial L'}{\partial w}$ 可以換成 $\frac{\partial L}{\partial w} + \lambda w$ 。那你會發現可以把這幾項整理在一起，變成 $(1 - \eta\lambda)w^t$ 再減掉你的參數對原來的 loss function 的 gradient。

所以如果根據這個式子，你會發現其實在每一次在 update 之前，都會把參數先乘 $1 - \eta\lambda$ 。通常你這邊的 η 就是你的 learning rate，是一個很小的值；這個 λ 通常會設一個很小的值，比如說 0.001 之類的。所以， $\eta\lambda$ 就是一個很小的值，而 $1 - \eta\lambda$ 通常是一個接近 1 的值，比如說 0.99。不管原來的 loss function 怎麼寫只看這個 update 式子的話，等於在 update 參數的時候，你做的事情是每次 update 參數之前，就不分青紅皂白，先乘個 0.99，也就是說，你每次都會讓你的參數越來越接近 0。不一定是越來越小，因為如果今天 w 是負的，乘上 0.99 它就變大了，它就接近 0。

所以，今天每一個參數在 update 之前，都乘上一個小於 1 的值，使得它每次都越來越靠近 0。有人會問如果越來越靠近 0，最後不就通通變 0 嗎？這聽起來就不 make sense，不會最後所有的參數都變 0，因為還有後面這一項。沒有後面這一項，每一次 update 參數就越來越小，最後通通變 0，但是還有從微分那邊得到這一項。最後這項跟前面會取得平衡，所以並不會最後所有的參數都變成 0。因為如果使用 L2 的 Regularization 的時候，每次都會讓 weight 變小一點，所以這招叫做 Weight Decay。

其實在 deep learning 裡面 Regularization 雖然有幫助，但是它的 importance 跟其他方法，比如說 SVM 比起來，並沒有那麼高，也就是說 Regularization 幫助往往沒有那麼顯著。我覺得有一個可能的原因是如果你看前面的 Early Stopping，我們可以決定甚麼時候 training 應該要被停下來。因為我們在做這個 neural network 的時候，通常我們在初始參數時都是給它一個很小的、接近 0 的值。那你在做 update 的時候，通常就是讓參數離 0 越來越遠、越來越遠。而做 Regularization 這件事情，它要達到的目的就是希望我們的參數不要離 0 太遠。而參數不要離 0 太遠，也就是加上 Regularization 所造成的效果，跟減少 update 次數所造成的效果，其實可能是很像的。因為 Early Stopping 減少 update 次數 其實也會避免參數離那些接近 0 的值太遠，跟 Regularization 做的事情可能是很接近的。所以在 neural network 裡面，Regularization 雖然有幫助，但沒有那麼重要，沒有重要到說，比如說像 SVM，它是 explicitly 把 Regularization 寫在 objective function 裡面。因為在做 SVM 的時候，它其實是要解一個 convex optimization problem。所以實際上它解的時候，並不一定會有 iteration 的過程，而是一步就解出那個最好的結果了，不像 deep learning 裡面有 Early Stopping 這件事。所以 SVM 沒有辦法用 Early Stopping 防止它離你太遠，必須要把 Regularization explicitly 加到 loss function 裡面去。

有人會問說，為什麼 Regularization 一定是平方，能不能用別的？當然可以用別的，比如說可以用 L1 的 Regularization，也就是把 Regularization 換成參數的 1 norm，換成參數的集合裡面每一個參數的絕對值的和。如果我們把 Regularization 這項從 2 norm 換成 1 norm 的話會發生什麼事呢？第一個問題可能是絕對值不能微分。一個簡單的回答是這個東西 implement 在 Keras 和 TensorFlow 都沒有問題，所以顯然是可以微分的。

Regularization

L1 regularization:

$$\|\theta\|_1 = |w_1| + |w_2| + \dots$$

- New loss function to be minimized

$$L'(\theta) = L(\theta) + \lambda \frac{1}{2} \|\theta\|_1 \quad \frac{\partial L'}{\partial w} = \frac{\partial L}{\partial w} + \lambda \text{sgn}(w)$$

Update:

$$\begin{aligned} w^{t+1} &\rightarrow w^t - \eta \frac{\partial L'}{\partial w} = w^t - \eta \left(\frac{\partial L}{\partial w} + \lambda \text{sgn}(w^t) \right) \\ &= w^t - \eta \frac{\partial L}{\partial w} - \underline{\eta \lambda \text{sgn}(w^t)} \quad \text{Always delete} \\ &= (1 - \eta \lambda) w^t - \eta \frac{\partial L}{\partial w} \quad \dots \text{L2} \end{aligned}$$

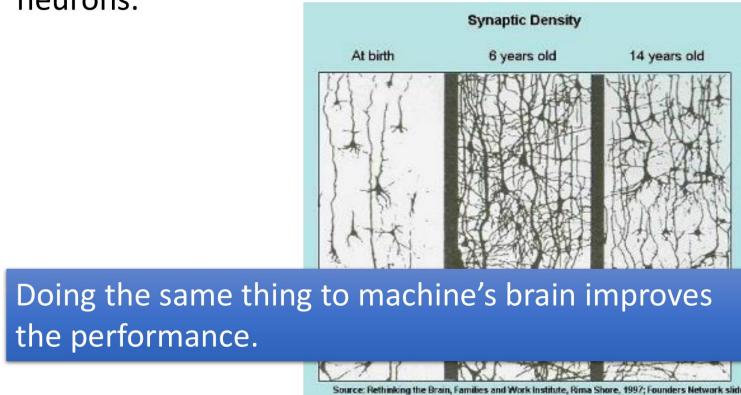
實際的回答是取絕對值，input 和 output 的關係是一個 V 的形狀，在 V 的一邊微分值是 1，在另外一邊微分值是 -1，不能微的地方，其實只有在 0 的地方而已，就忽略這種情況。真的出現 0 的時候你就胡亂給它一個值，比如說，給它 0 就好了。所以說如果把 $\|\theta\|_1$ 對 w 做微分的時候得到的結果是：如果今天 w 是正數，那微分出來就是 +1； w 是負數，微分出來就是 -1。所以，我們這邊寫了一個 w 的 sign function，也就是說，如果 w 是正數的話這個 function output 就是 +1； w 是負數的話，這個 function output 就是 -1。這個式子告訴我們，每一次在 update 參數的時候，我們就不管三七二十一，都一定要去減一個 $\eta \lambda$ ，再乘一個 w 的 sign，也就是說，如果今天 w 是正的， w 的 sign 就是 +1，所以就變成是減一個 positive 的值，會讓參數變小；如果 w 是負的， w 的 sign 就是 -1，那就變成是加一個值，會讓參數變大。也就是說，只要參數是正的就減掉一些，只要參數是負的就加上一些，不管那個參數原來的值是多少。

所以如果把這個 L1 跟 L2 做一下比較的話，他們同樣是讓參數變小，但是他們做的事情是略有不同的。因為用 L1 的時候，每次都減掉固定的值；用 L2 的時候，你是每一次都乘上一個小於 1 固定的值。所以今天 w 是一個很正的值，比如說一百萬，那乘上一個 0.99，其實是把 w 減掉一個很大的值。但是，對 L1 來說，它每次減掉的值都是固定的，不管 w 是一百萬還是 0.1， w 減掉的值都是固定的。所以對 L2 來說，只要 w 有出現很大的值，這個很大的 w 會下降很快，很快就會變得很小。但是如果 L1 的話，那就不一樣了。如果 w 有很大的值，它的下降速度跟其他很小的 w 是一樣的。所以，透過 L1 的 training 以後有可能 learn 出來的 model 裡面還是有一些很大很大的值。但是如果我們考慮很小的值的話，比如說 0.1, 0.01，對 L2 來說它的下降速度就很慢。所以，在 L2 裡面，它 train 出來的結果會保留很多接近 0 的值；L1 裡面不會保留很多很小的值。所以用 L1 做 training，得到的結果是會比較 sparse，也就是說 train 出來的參數裡面有很多接近 0 的值，但是會有很大的值。不像是 L2，train 出來的結果平均值的都比較小。所以他們 train 出來的結果是略有差異的。

我們剛才在討論 CNN 的時候，有提到 L1 就是要產生一個 image 的時候，有產生 L1。在剛才那個 task 裡面，L1 是比較適合的，因為我想要看到 sparse 的結果。我有試過用 L2，但是結果就沒有 L1 看起來那麼明顯，雖然 L1 看起來也沒有很明顯，但 L1 看起來的結果，還比較像是一個 digit。

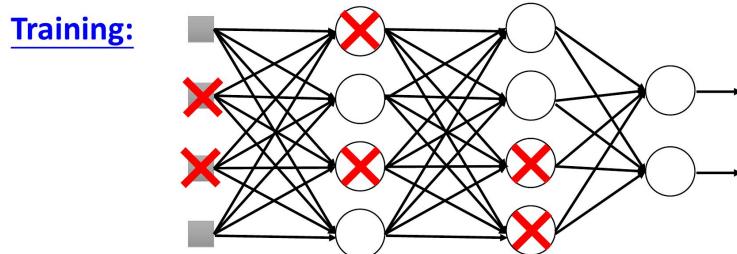
Regularization - Weight Decay

- Our brain prunes out the useless link between neurons.



那這邊就胡扯一個東西，Weight Decay。我們在人腦裡面也會做 Weight Decay。我記得龍騰的生物課本上有這張圖。左邊這個是剛出生的時候，嬰兒的神經長的樣子；6 歲的時候，有很多很多的神經；但是，到 14 歲的時候，神經間的連結又減少了，所以 neural network 也會跟我們人有一些很類似的事情，如果有一些 weight 都沒有去 update 它，那它每次都會越來越小，最後就接近 0 不見了，這跟人腦的運作，是有異曲同工之妙。

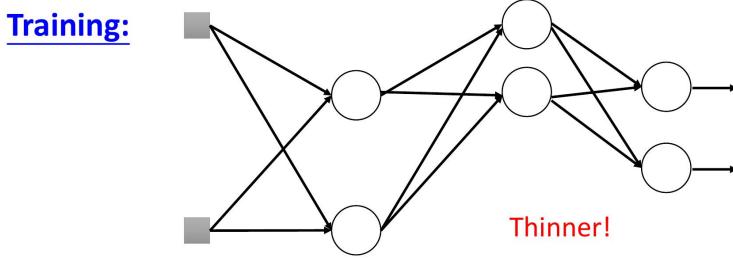
Dropout



- Each time before updating the parameters
 - Each neuron has p% to dropout

最後我們要介紹一下 dropout。我們先介紹 dropout 是怎麼做的，然後才說明為什麼這樣做。dropout 是怎麼做的呢？它是這樣，在 training 的時候，每一次我們要 update 參數之前，我們都對每一個 neuron，包括 input layer 裡面的每一個 element 做 sampling，那這個 sampling 是決定這個 neuron 要不要被丟掉，每個 neuron 有 p% 的機率會被丟掉。

Dropout



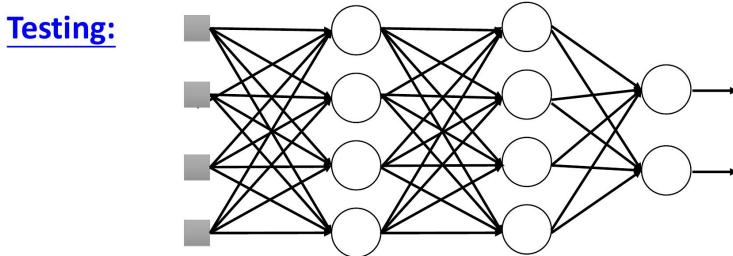
➤ Each time before updating the parameters

- Each neuron has p% to dropout
- **The structure of the network is changed.**
- Using the new network for training

For each mini-batch, we resample the dropout neurons

那如果一個 neuron 被 sample 到要丟掉的時候，跟它相連的 weight 也失去作用，所以就變上圖這樣。所以，做完這個 sample 以後，network 的 structure 就變瘦了，變得比較細長，然後再去 train 這個比較細長的 network。這邊要注意一下，所謂的 sampling 是每次 update 參數之前都要做一次，每一次 update 參數的時候 training 的那個 network structure 是不一樣的。當你在 training 使用 dropout 的時候，performance 會變差，因為本來如果你不要 dropout 好好的做的話，在 MNIST 上，可以把正確率做到 100%。但是如果加 dropout，因為神經元在 train 時有時候莫名其妙就會不見，所以你在 training 時有時候 performance 會變差，本來可以 train 到 100%，就會變成只剩下 98%。

Dropout



- No dropout
- If the dropout rate at training is p%,
all the weights times 1-p%
 - Assume that the dropout rate is 50%.
If a weight w = 1 by training, set w = 0.5 for testing.

所以，當你加了 dropout 的時候，在 training 上會看到結果變差。dropout 它真正要做的事情是就是要讓 training 的結果變差，但是 testing 的結果是變好的。也就是說，如果你今天遇到的問題是 training 做得不夠好，你再加 dropout，就是越做越差那在 testing 的時候怎麼做呢？在 testing 的時候要注意兩件事：第一件事情就是 testing 的時候所有的 neuron 都要用，不做 dropout；另外一件事情是，假設在 training 時，dropout rate 是 p%，那在 testing 的時候，

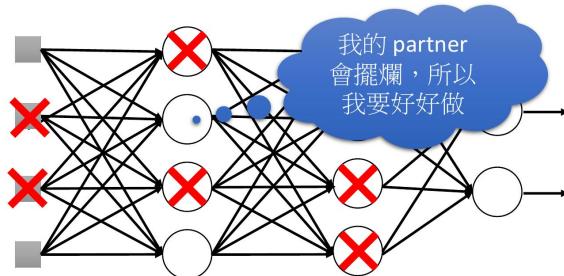
所有 weight 都要乘 $(1 - p\%)$ 。也就是說，假設現在 dropout rate 是 50%，在 training 時 learn 出來的 weight 等於 1，那 testing 的時候，你要把那個 weight 設成 0.5。

Dropout - Intuitive Reason



那為甚麼 dropout 有用，直覺的想法是這樣子：training 的時候，會丟掉一些 neuron，就好像是你要練輕功的時候，你會在腳上綁一些重物；然後，實際上 testing 的時候，是沒有 dropout 的實際上 test 的時候，你就把重物拿下來，所以就會變得很強。這個是小李，他平常都綁一個重物，只有在要貫徹自己的忍道的時候，他才會拿下來。

Dropout - Intuitive Reason



- When teams up, if everyone expect the partner will do the work, nothing will be done finally.
- However, if you know your partner will dropout, you will do better.
- When testing, no one dropout actually, so obtaining good results eventually.

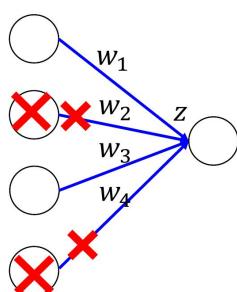
另外一個直覺的理由是：一個 neural network 裡面的每一個 neuron 就是一個學生，大家被連結在一起。大家聽到要做 final project。在一個團隊裡面，總是有人會擺爛，就是它是會 dropout 的。所以假設你覺得你的隊友其實會擺爛，這個時候你就會想要好好做，會想要去 carry 他。但是，實際上最後在 testing 的時候，大家都會好好做，沒有人需要被 carry，因為每個人都做是更有利，所以結果是更好的。因此在 testing 的時候，不用 dropout。

Dropout - Intuitive Reason

- Why the weights should multiply $(1-p)\%$ (dropout rate) when testing?

Training of Dropout

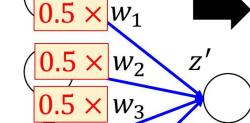
Assume dropout rate is 50%



Testing of Dropout

No dropout

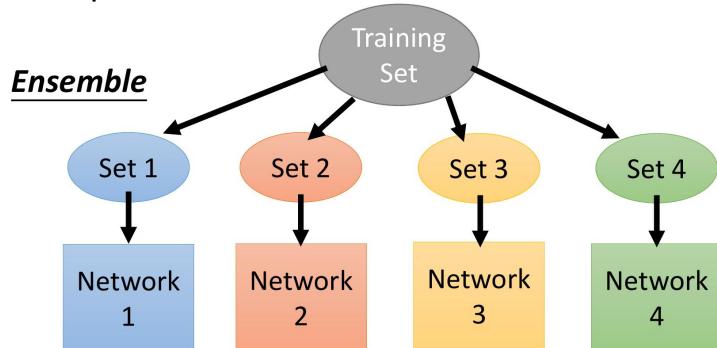
$$0.5 \times w_1 \quad \rightarrow z' \approx 2z$$



$$\text{Weights multiply } 1-p\% \quad \rightarrow z' \approx z$$

另外想解釋的就是為甚麼 dropout rate 50% 的時候，testing 的 weight 就要乘 0.5？為甚麼 training 跟 testing 的 weight 是不一樣的呢？因為照理說 training 用甚麼 weight 就要用在 testing 上，training 跟 testing 的時候居然是用不同的 weight，為甚麼這樣呢？直覺的理由是：假設現在 dropout rate 是 50%，那在 training 的時候期望總是會丟掉一半的 neuron。對每一個 neuron 來說，總是期望說它有一半的 neuron 是不見的，是沒有 input 的。所以假設在這個情況下，learn 好一組 weight，但是在 testing 的時候，是沒有 dropout 的。對同一組 weight 來說，假如你在這邊用這組 weight 得到 z ，跟在這邊用這組 weight 得到 z' ，它們得到的值，其實是會差兩倍。因為在左邊這個情況下，總是會有一半的 input 不見；在右邊這個情況下，所有的 input 都會在。而用同一組 weight 的話，變成 z' 就是 z 的兩倍了，這樣變成 training 跟 testing 不 match，performance 反而會變差。所以把所有 weight 都乘 0.5，做一下 normalization，這樣 z 就會等於 z' 。把這個 weight 乘上一個值以後，反而會讓 training 跟 testing 是比較 match 的。這個是比較直觀上的結果，如果要更正式的話，其實 dropout 有很多理由。

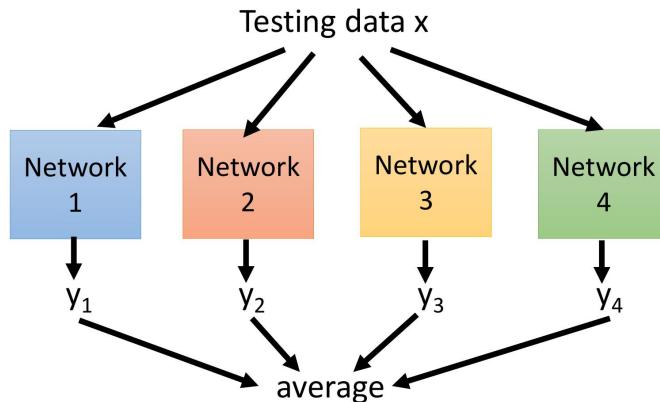
Dropout is a kind of ensemble.



Train a bunch of networks with different structures

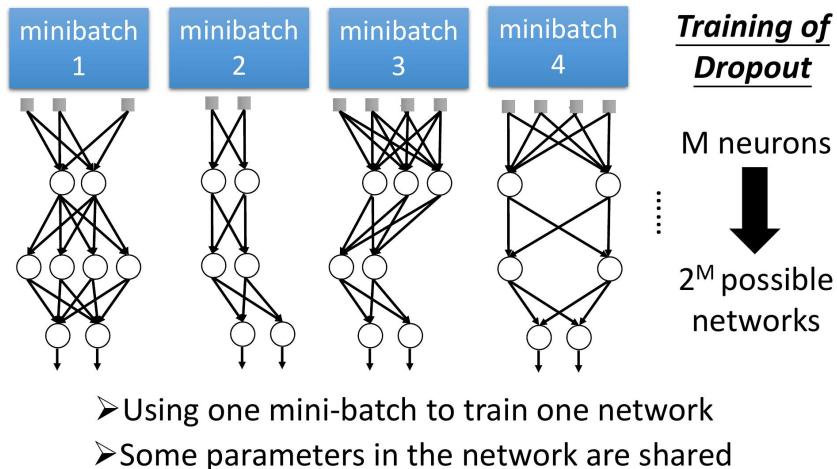
Dropout is a kind of ensemble.

Ensemble



dropout 還是一個可以探討的問題，在文獻上找到很多不同的觀點來解釋為什麼 dropout 會 work。那我覺得我比較能接受的是 dropout 是一種終極的 ensemble 的方法。甚麼是 ensemble 的方法呢？ensemble 的方法在比賽的時候常用，意思是說我們有一個很大的 training set，每次從 training set 裡面只 sample 一部分的 data 出來。記得我們在說明 bias 跟 variance 的 trade off 的時候有說過，打靶有兩種狀況，一種是你的 bias 大，所以你打不準一種是你的 variance 很大，所以你打得準。如果今天有一個很複雜很笨重、很大的 model 的時候，它往往是 bias 準，但 variance 很大。但是，如果你這個笨重的 model 有很多個雖然它 variance 很大，最後平均起來結果就很準。所以今天 ensemble 做的事情，其實就是要利用這個特性：我們把原來的 training data 裡面 sample 出很多 subset，然後 train 很多個 model，每一個 model 甚至可以是 structure 不一樣。雖然說，每一個 model 他們可能 variance 很大，但是如果他們都是很複雜的 model 的時候，平均起來這個 bias 就很小。所以你 train 了一把 model，然後在 testing 的時候丟一筆 training data 進來，它通過所有的 model 得到一大堆的結果，再把這一大堆的結果平均起來當作我們最後的結果。如果你的 model 很複雜的話，這一招往往有用。random forest 也是實踐這個精神的一個方法。如果你用一個 decision tree，它就很弱，胡亂做它就會 overfitting，那如果你用 random forest 就沒有那麼容易 overfitting。

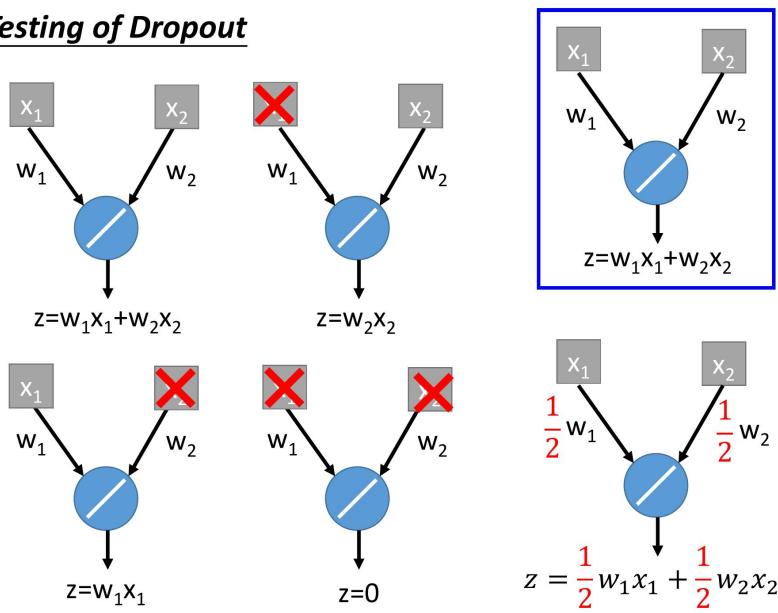
Dropout is a kind of ensemble.



那為什麼說 dropout 是一個終極的 ensemble 的方法呢？我們知道在做 dropout 的時候，我們每一次要 update 參數的時候就拿一個 minibatch 出來，要 update 參數的時候都會做一次 sample。所以，你拿第一個 minibatch 的時候，你 train 的 network 長得像第一張圖；你拿第二個 minibatch 的時候，你 train 的 network 可能長得像第二張圖；你拿第三個長得像第三張，你拿第四個長得像第四張。所以 dropout 等於是一個終極的 ensemble 的方式。假設你有 M 個 neuron，每一個 neuron 可以 drop 或不 drop，所以你可能的 neuron 的數目有 2^M 個。當你在做 dropout 的時候，你等於是用這 2^M 個 neuron 去 train 這個 minibatch。你每次都只用一個 minibatch 的 data 去 train 一個 network，你可能就用這個 minibatch 裡面的 100 筆 data 去 train 這些 network，總共有 2^M 個可能的 network。當然因為你最後 update 的次數是有限的，你可能沒有辦法把 2^M 個 network 每個都 train 一遍。但是你可能就 train 了好多好多的參數，好多好多的 network。你有做幾次 update 參數，你就 train 幾次 network。但是每個 network 就只用一個 batch 來 train，這可能會讓人覺得很不安：一個 batch 才 100 筆 data，怎麼 train 一個 network 呢。沒有關係，因為這些不同的 network 之間的參數是 shared，也就是說這一個 network 的這一個參數就是這個 network 的這個參數，就是圖上的這 4 個同位置的參數其實是同一個參數。所以雖然說一個 network 的 structure，它只用一個 batch train，但是一個 weight，它可能用好多個 batch 來 train。比如說，這個 weight，它在這 4 個 batch 做 dropout 的時候，都沒有把這個 weight 丟掉。那這個 weight，就是拿這 4 個 batch 合起來 train 的結果。所以，當你做 dropout 的時候，你就是 train 了一大把的 network structure。

理論上，每一次 update 參數的時候，你都 train 了一個 network 出來。那 testing 的時候呢？按照 ensemble 這個方法的邏輯應該就是，你把那一大把的 network 通通拿出來，然後你把 testing data 丟到那一把 network 裡面去，每一個 network 都給你吐一個結果，然後把所有的結果平均起來就是最終的結果。但是，在實作上你沒辦法這麼做，因為這一把 network 實在太多了，你沒有辦法每一個都丟一個 input 進去去看它 output 是什麼，再平均起來，這樣運算量太大。所以 dropout 最神奇的地方是告訴你：當你把一個完整的 network 不做 dropout，但是把它的 weight 乘上 $(1 - p\%)$ ，然後把 training data 丟進去得到它的 output 的時候，這個 ensemble 的結果，跟把 weight 乘上 $(1 - p\%)$ 的結果，是可以 approximate 的。

Testing of Dropout



何以見得呢？我們舉一個例子。我們來 train 一個很簡單的 network，它就只有一個 neuron。它的 activation function 是 linear 的，不考慮 bias。這邊有一個 neuron，它的 input 是 x_1, x_2 ，經過 dropout training 以後算出來的 weight 是 w_1, w_2 ，所以它的 output 就是 $w_1x_1 + w_2x_2$ 。這邊 neuron 沒有 activation function 或 activation function 是 linear 的。

我們做 dropout 的時候，不會 drop 那個 output 的 neuron，只會 drop hidden layer 跟 input 的 neuron。那這邊每一個 neuron，它可以被 drop 或不被 drop，所以我們總共有 4 種 structure：一個是通通沒被 drop、一個是 drop x_1 、一個是 drop x_2 ，一個是 x_1, x_2 都被 drop 掉。假設你 input x_1, x_2 ，左上角 network 紿我們的就是 $w_1x_1 + w_2x_2$ 。同樣的 input，但是 x_1 被 drop 掉，得到的 output 是 w_2x_2 ，左下角是 w_1x_1 ，右下角的 output 是 0。我們要做 ensemble，所以要把這 4 個 network 的 output 通通都 average 起來。這邊有 4 個值，然後 w_1x_1 出現兩次、 w_2x_2 出現兩次，所以得到的結果是 $\frac{1}{2}w_1x_1 + \frac{1}{2}w_2x_2$ 。我們現在把這兩個 weight 都乘 $\frac{1}{2}$ ，也就是把 w_1 乘 $\frac{1}{2}$ 、把 w_2 乘 $\frac{1}{2}$ ，同樣 input x_1, x_2 ，得到的 output 也同樣是 $\frac{1}{2}w_1x_1 + \frac{1}{2}w_2x_2$ 。所以，這邊想要呈現的是，在這個最簡單的 case 裡面用不同的 network structure 做 ensemble 這件事情，跟我們把 weight multiply 一個值而不做 ensemble 所得到的 output，其實是一樣的。

你可能會說，這個例子這麼簡單，所以這個例子上會 work，也是很直覺的。大概小學生都知道，這個是 equivalent。但是假如說，activation function 是 sigmoid function；或是它是很多個 layer，它還會 work 嗎？結論就是不會 equivalent，只有是 linear 的 network ensemble 才會等於 multiply 一個 weight。左邊跟右邊要相等的前提是你的 network 要是 linear 的。但是 network 不是 linear 的，所以他們其實不 equivalent。

這個就是 dropout 最後一個很神奇的地方，雖然不 equivalent，但是最後結果還是會 work。所以根據這個結論，有一個想法是說：既然 dropout 在 linear 的 network 上，ensemble 才會等於乘一個 weight。所以今天如果 network 很接近 linear 的話，應該 dropout performance 會比較好，比如說用 ReLU，或者用 Maxout network，相對於 sigmoid，他們是很接近 linear 的。所以 dropout 確實在用 ReLU 或 Maxout network 的時候，它的 performance 是確實比較

好的。如果去看 Maxout network 的論文的話，它裡面也有提到這一點，它的Maxout 跟 dropout 加起來的記錄量是比 sigmoid function 還要大的，這也是作者相當自豪的一點。

臺灣大學人工智慧中心 科技部人工智慧技術暨全幅健康照護聯合研究中心 <http://aintu.tw>