Deep Learning

DN 的三個步驟

一、限定 Model 的範圍

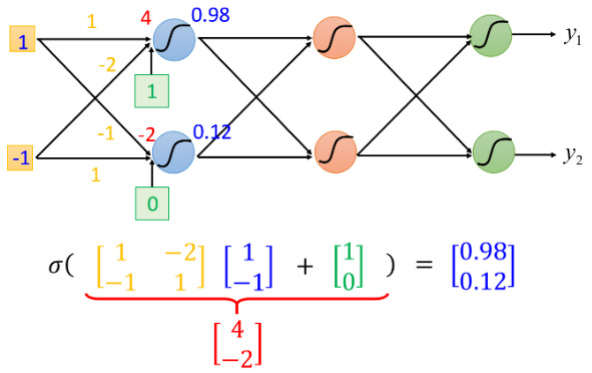
在使用 DL 時，實際上就是選擇 Neural Network 的 model 進行運用。
而依照 neural network 的類型又細分成一般常見疊很多層 layer 的 Deep Neural Network 以及特化用於影像分析類別的 Convolutional Neural Network。

可以將 neural network 想成由數個 Logistic Regression 前後 connect 在一起，然後把一個 Logistic Regression 稱之為 neuron。

每個 Logistic Regression 都有自己的 weight 和 bias，這些 weight 和 bias 集合起來，就是這個 network 的 parameter，我們用 θ 表示。

1. Matrix Operation

Network 的運作過程我們可以使用 matrix 的方式來表示，每一個 row 對應的是一個 neuron 的 weight，row 的數量就是 neuron 的個數；而 input x，bias b 和 output y 都是一個行向量，row 的數量就是 feature 的個數。



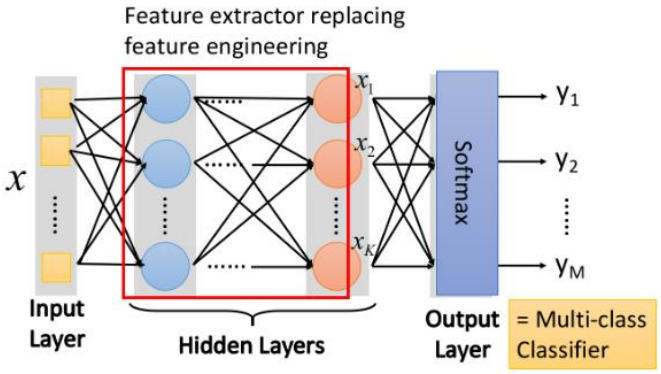
該方法使得 DNN 的運算得以透過矩陣運算完成，因此可使用 GPU 加速。

2. Feature Extractor

我們可以把 hidden layers 這部分，看作是一個 feature extractor，該處取代了做 ML 時手動進行 feature engineering、feature transformation 這些事情。
因此 DL 將原本 ML 從本來如何抽取 feature 的問題轉化成怎麼 design network structure。

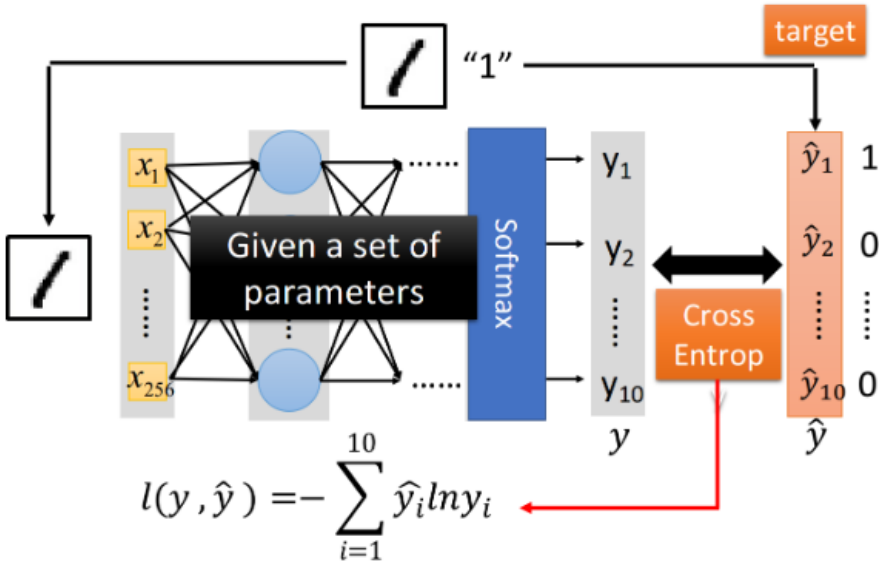
3. Multi-class Classifier

而 Output layer 做的事情，其實就是把它當作一個 Multi-class classifier。他會利用 feature extractor 分類出的 feature 對 input 進行分類，我們會在最後一個 layer 加上 softmax 用來預測每個 class 可能的機率。



二、從範圍內找出好的 function

以 Multi-class classification 為例，我們可將 softmax 預測的結果與 true label 得到的 target 進行 cross entropy（同 ML 的 classification 的問題）。



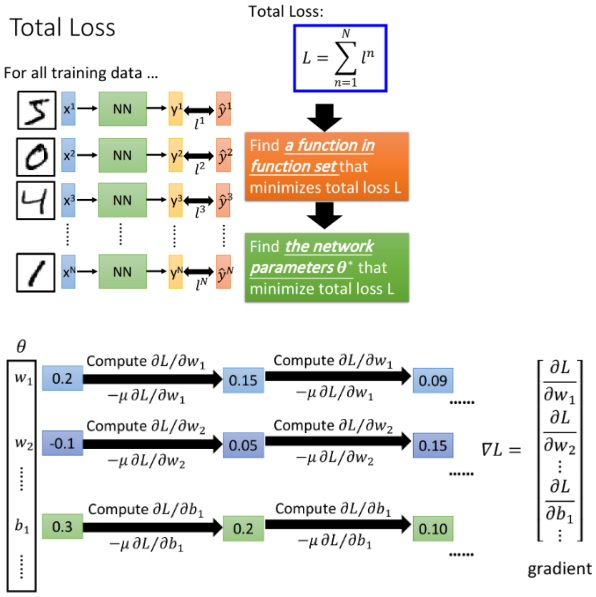
$$l(y,\hat{y}) = -\sum_{i=1}^{10} \hat{y}_i \ln y_i$$

$$L(f) = \sum_n l(f(x_i^n), \hat{y}^n)$$

$$l(f(x^n), \hat{y}^n) = -[\hat{y}^n \ln f(x^n) + (1-\hat{y}^n)\ln(1-f(x^n))]$$

三、找出最小的 Loss Function

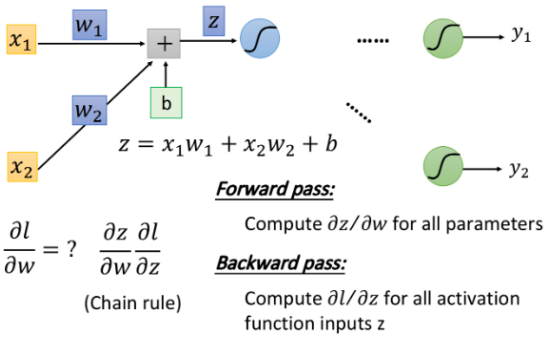
將所有 data 的 cross entropy 都加總起來就會得到 total loss，得到 loss function 之後找一組 network 的 parameters（θ*）使 total loss 最小，此處使用 gradient descent 即可（方法同 linear regression）。



1. Backpropagation

由於 NN 的 parameters 非常多，無法很有效率地一個個取其 gradient，因此可用 Backpropagation 增加運算效率。
因為 loss function 為所有 training data 的 cross entropy 的總和，因此只要考慮某一筆 data 的 cross entropy 對參數 w 的偏微分再加總，就可以把 total loss 對某一個參數 w 的偏微分給計算出來。

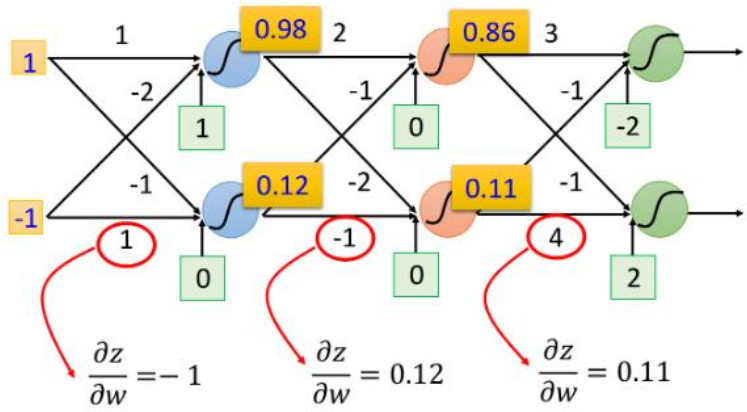
先考慮某一個假設只有兩個 input 的 neuron 經過 activation function 從這個 neuron 中 output 出來，接著再經過數個 neuron 得到最終的 output：y_1, y_2。
而該他們與 target 的 cross entropy 對 w 的偏微分可分作 Forward pass 與 Backward pass 討論。



(1) Forward Pass

即$\frac{\partial z}{\partial w}$，該項可透過上圖得知$\frac{\partial z}{\partial w_1}$與$\frac{\partial z}{\partial w_2}$分別為$x_1$與$x_2$。故知 w 前面連接的 input 是什麼，那微分後的值就是什麼。

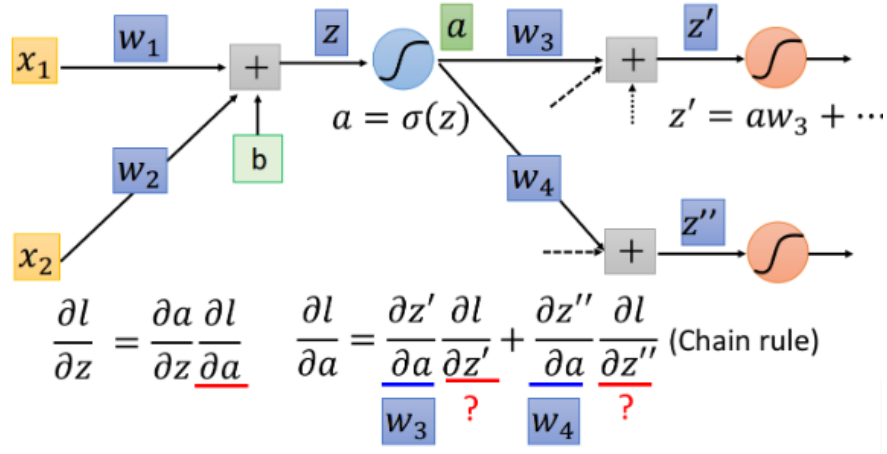
Compute $\partial z/\partial w$ for all parameters



(2) Backward Pass

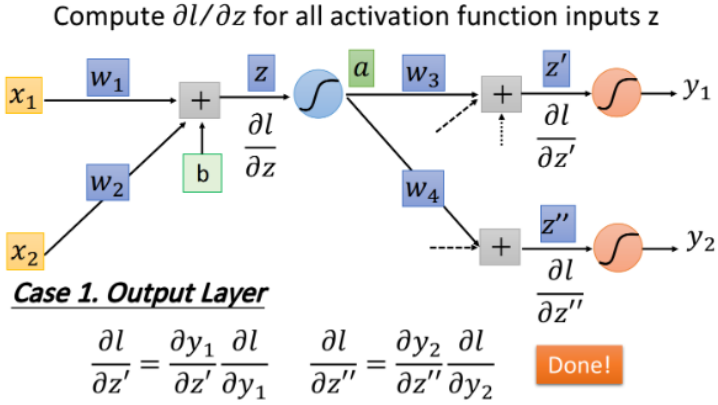
此處假設 activation function 是 sigmoid function，透過 chain rule 可將$\frac{\partial l}{\partial z}$不斷轉換直到下圖最後的式子，此時只要得知$\frac{\partial l}{\partial z'}$與$\frac{\partial l}{\partial z''}$即可。

Compute $\partial l/\partial z$ for all activation function inputs z



而$\frac{\partial l}{\partial z'}$與$\frac{\partial l}{\partial z''}$又可分作兩個 case 討論：

Case I Output Layer

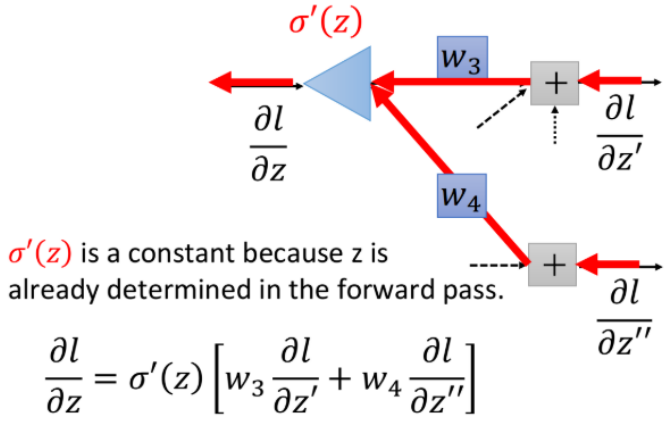


$\frac{\partial y_1}{\partial z'}$與$\frac{\partial y_2}{\partial z''}$只要知道前面紅色的 activation function 就可以得出其值。

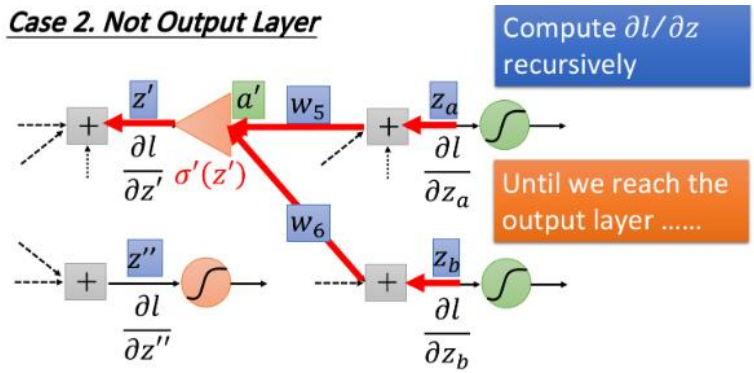
$\frac{\partial l}{\partial y_1}$與$\frac{\partial l}{\partial y_2}$取決於 loss function 是怎麼定義的（例如 cross entropy）。

Case II Not Output Layer

這裡我們可以另一個觀點來看待這個式子，想像一個新的 neuron 為下列形式。



將其進一步推廣至原本的 network 中，反覆運算下去直到抵達 output layer 便可適用 Case I。

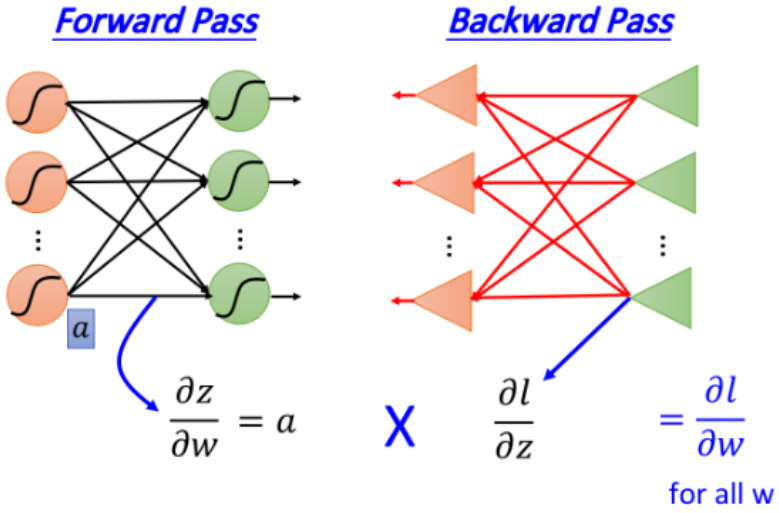


Summary

Forward pass：每個 neuron 的 activation function 的 output，就是它所連接的 weight 的$\frac{\partial z}{\partial w}$。

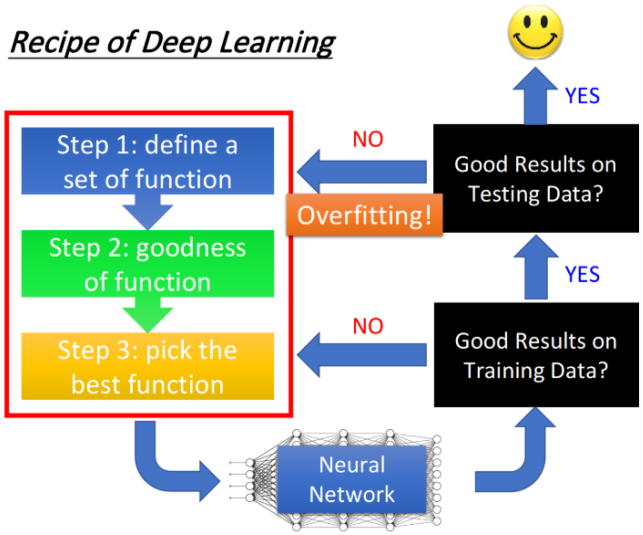
Backward pass，建一個與原來方向相反的 neural network，它的三角形 neuron 的 output 就是$\frac{\partial l}{\partial z}$。

兩者相乘即為所求：$\frac{\partial l}{\partial w} = \frac{\partial z}{\partial w}|_{forward\ pass} \cdot \frac{\partial l}{\partial z}|_{backward\ pass}$



四、Tips for Deep Learning

再結束上述三個步驟後，為使 NN 的表現更好，我們通常會再檢查 model 在 training data 與 testing data 的表現。

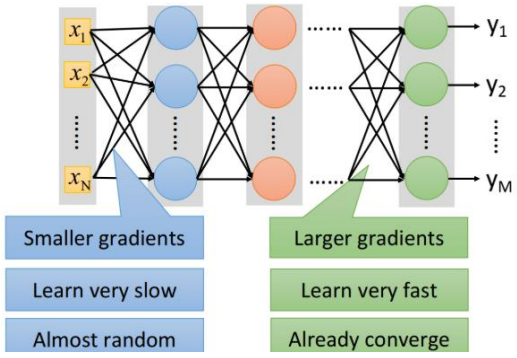


1. Performance in Training Data

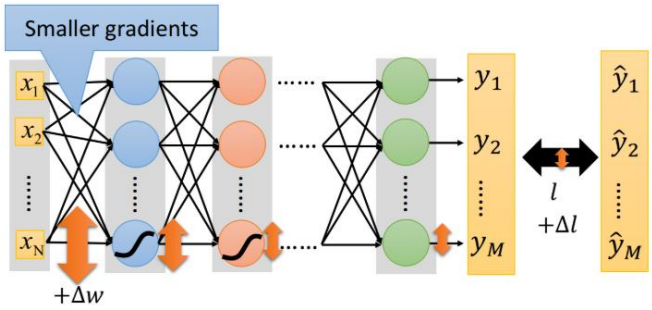
此處為 DL 一項特別的地方，若非 DL 方法（如 k-nearest neighbor 或 decision tree）其在 training set 上的 performance 正確率就是 100，並無檢查必要。也因如此非 DL 方法非常容易 overfitting，而對 DL 來說 overfitting 往往不會是遇到的第一個問題。以下有兩個方式可以增加其在在 training set 上的 performance。

(1) New Activation Function

使用 sigmoid function 時 network 疊得很深的時候準確率反而會下降，該處稱 Vanishing Gradient Problem。
意即在靠近 input 的地方，這些參數的 gradient（對 loss function 的微分）是比較小的；而在比較靠近 output 的地方，它對 loss 的微分值會是比較大的。
因此設定同樣 learning rate 的時候，靠近 input 的地方，它參數的 update 是很慢的；而靠近 output 的地方，它參數的 update 是比較快的。

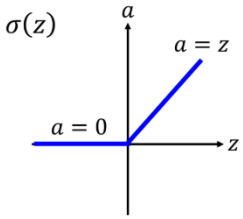


原因在於 sigmoid function 會不斷縮小參數的變化，導致 input 對 loss 的影響會比較小，於是靠近 input 的那些 weight 對 loss 的 gradient 遠小於靠近 output 的 gradient。



因此改用其他的 activation function 就可以解決該問題。

a. ReLU



與 sigmoid function 相比他有下列好處：

－可以處理 Vanishing gradient 的問題。

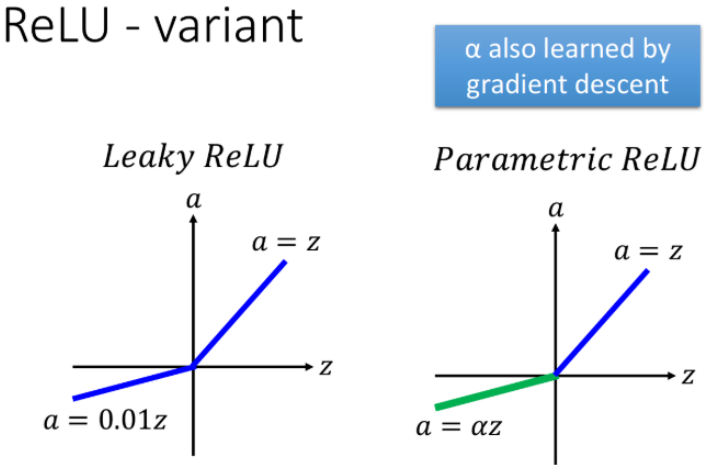
－ReLU 的想法結合了生物上的觀察（Pengel 的 paper)。

－運算速度較快。

－無窮多 bias 不同的 sigmoid function 疊加的結果會變成 ReLU。

b. Leaky ReLU 與 Parametric ReLU

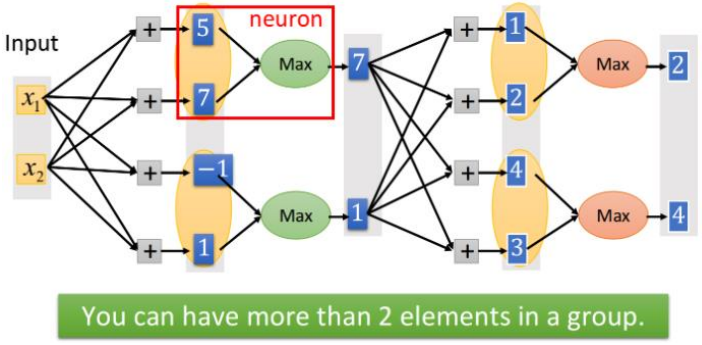
使用 ReLU 時當 input<0，output=0，此時微分值 gradient 也為 0，如此便無法更新參數，故提出了其他形式的 ReLU 解決該問題。



c. Maxout

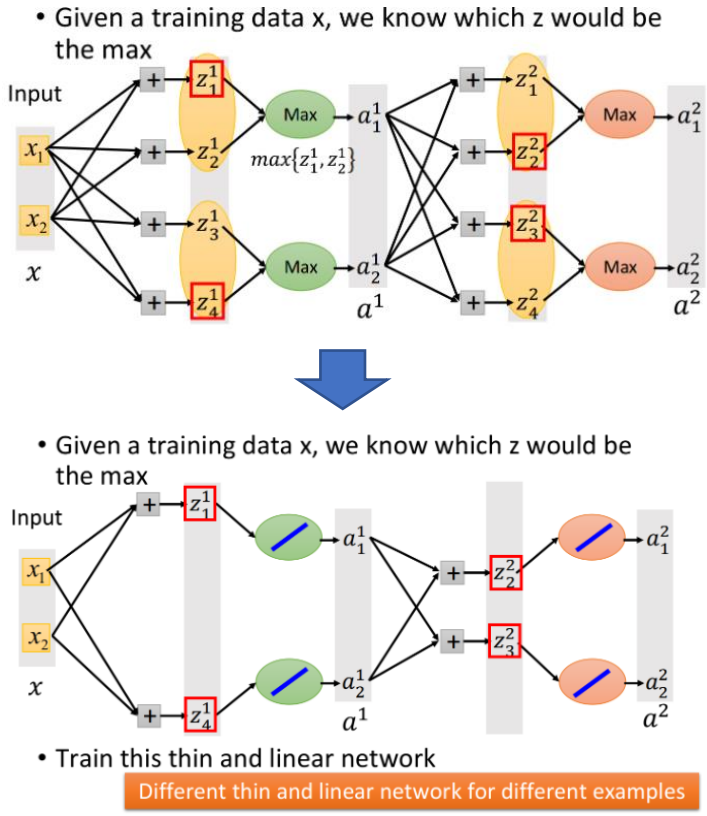
Maxout 的想法是，讓 network 自動去學習它的 activation function，一切都是由 training data 來決定的。

該方法將某幾個“neuron”的 input 分為一個 group，然後在這個 group 裡選取一個最大值作為 output。



在具體的實踐上，我們先根據 data 把 max 函數轉化為某個具體的函數，意即取走沒有用到的 weight，再對這個轉化後的 linear network 進行微分。

因此當你 input 不同 data 的時候，得到的 network structure 是不同的，留在 network 裡面的參數也是不同的。



(2) Adaptive Learning Rate

在做 deep learning 的時候，這個 loss function 可視化之後可以是任何形狀，不便於尋找 optimal，因此出現下列兩項方式改善該問題。

a. RMSProp

$$w^{t+1} = w^t - \frac{\eta}{\sigma^t} g^t$$

$$\sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

Adagrad 的進階版，跟 Adagrad 不同之處在於，Adagrad 的分母是對過程中所有的 gradient 取平方和開根號，也就是說 Adagrad 考慮的是整個過程平均的 gradient 訊息；RMSProp 雖然也是對所有的 gradient 進行平方和開根號，但是它用一個 α 來調整對不同 gradient 的使用程度。

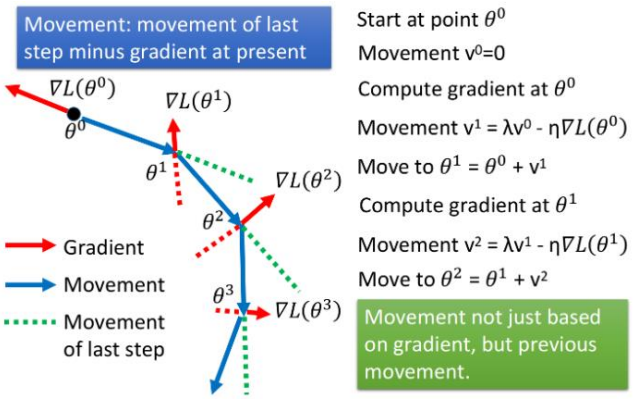
優：可透過調整 α 改變傾向相信新的 gradient 所告訴你的 error surface 的平滑或陡峭程度或舊的 gradient 所提供的 information。

b. Momentum

在做 deep learning 的時候，由於 error surface 極為複雜，因此有卡在 local minimum、saddle point 或是 plateau 的疑慮。此處便可利用慣性的概念去改良。

$$\theta^{t+1} = \theta^t + v^{t+1} \qquad v^{t+1} = \lambda v^t - \eta \nabla L(\theta^t)$$

在 gradient descent 裡加上 Momentum 的時候，每一次 update 的方向，不再只考慮 gradient 的方向，還要考慮上一次 update 的方向，那這裡我們就用一個變量 v 去記錄前一個時間點 update 的方向。



c. Adam

綜合 RMSProp 與 Momentum 的方法。

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: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_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 θ_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)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

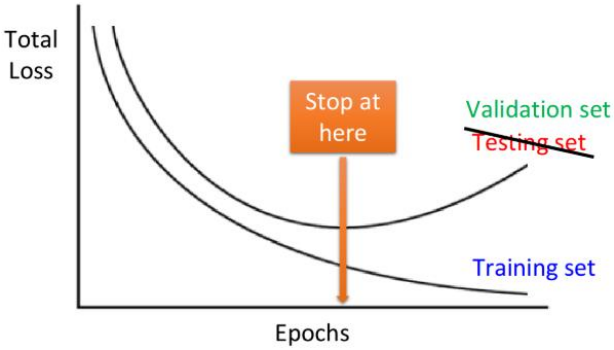
2. Performance in Testing Data

意即解決 overfitting 的問題。

(1) Early Stopping

理想上假如知道 testing data 上的 loss 變化情況，在 testing set 的 loss 最小的時候停下來，而不是在 training set 的 loss 最小的時候停下來時，就可讓 testing set 的表現變好。

但 testing set 實際上是未知的東西，所以我們需要用 validation set 來替代它去做這件事情。



(2) Regularization

在原來的 loss function 上額外增加幾個 term 使得 function 更平滑，新加入的 term 可分作 L1 與 L2 兩種。

L1

Regularization

L1 regularization:

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

- New loss function to be minimized

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

Update:

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

$$= w^t - \eta \frac{\partial L}{\partial w} - \eta\lambda \,\mathrm{sgn}(w^t)$$ Always delete

每次 update 的時候，都要減去$\eta\lambda sgn(w^t)$，使參數的絕對值減小至接近於 0。

L2

Regularization

L2 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

$(1-\eta\lambda)$: Closer to zero

參數在每次 update 之前，都會乘上一個$(1-\eta\lambda)$，而η與λ通常會假設的很小，所以$(1-\eta\lambda)$是一個接近於 1 的值（例如 0.99），這意味著，隨著 update 次數增加，參數會越來越接近於 0。
使用 L2 regularization 可以讓 weight 每次都變得更小一點，這就叫做 Weight Decay。

L1 vs. L2

L1 和 L2，雖然它們同樣是讓參數的絕對值變小，但它們做的事情其實略有不同：
－L1 使參數絕對值變小的方式是每次 update 減掉一個固定的值
－L2 使參數絕對值變小的方式是每次 update 乘上一個小於 1 的固定值

當參數 w 的絕對值比較大的時候，L2 會讓 w 下降得更快，而 L1 每次 update 只讓 w 減去一個固定的值，train 完以後可能還會有很多比較大的參數。

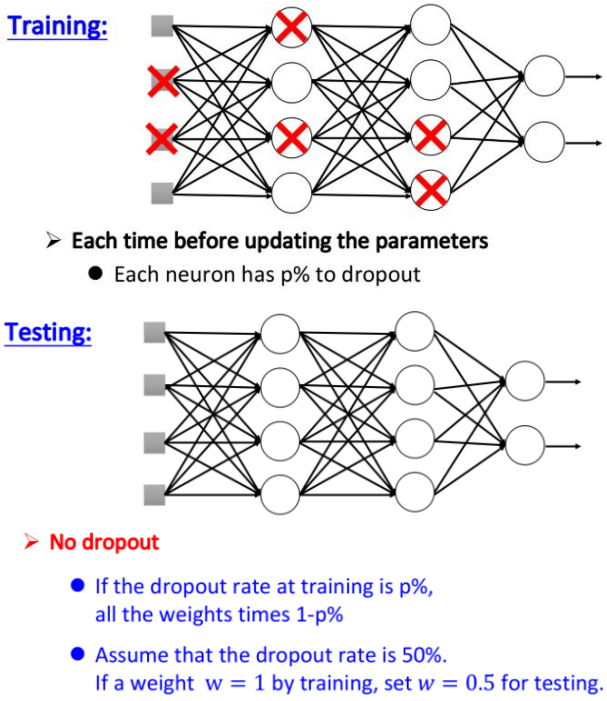
當參數 w 的絕對值比較小的時候，L2 的下降速度就會變得很慢，train 出來的參數平均都是比較小的，而 L1 每次下降一個固定的 value，train 出來的參數是比較 sparse 的，這些參數有很多是接近 0 的值，也會有很大的值。

(3) Dropout

在 training 的時候，每次 update 參數之前，我們對每一個 neuron（也包括 input layer 的“neuron”）做抽樣，每個 neuron 都有 p%的機率會被丟掉，如果某個 neuron 被丟掉的話，跟它相連的 weight 也都要被丟掉。

意即每次 update 參數之前都通過抽樣只保留 network 中的一部分 neuron 來做訓練。

而在做 testing 的時候，每個 neuron 都要用到，但所有的 weight 都要乘上(1-p%)才能被當做 testing 的 weight 使用。



綜上所述，Dropout 會讓 training set 上的結果變差，但是在 testing set 上的結果是變好的。（因為某些 neuron 在 training 的時候莫名其妙就會消失不見）