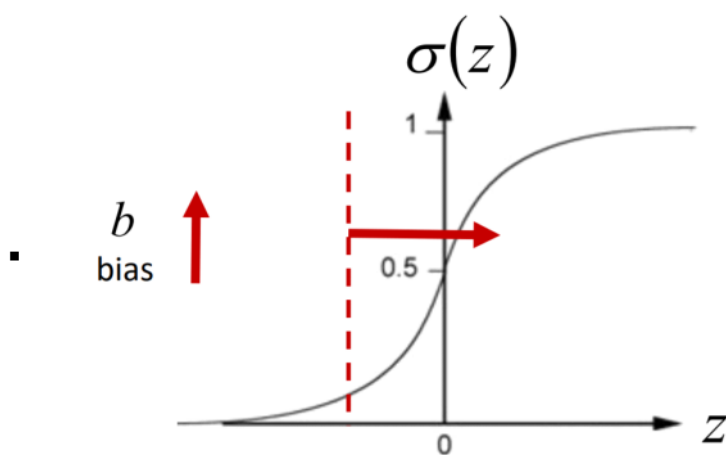


- function越複雜越難訓練
 - function無限多種要嘗試
 - 系統化評估好的function? model長怎樣?怎麼找好的function?
- Model architecture
 - 二分類/是否independent或相依/yes no
 - 不同維度上面的轉換 (M-dim input, N-dim output), 目前還沒辦法讓model學到 hyperparameters, 目前還需要人工(有研究再做)
 - 學到兩組之間的mapping (weighted-sum)
 - 每個神經元都只會做一件小事情 EX: bias=>一定會讓這個數值加進去
 - 有bias可以讓輸出必定至少有一定的數值, bias越大可以讓af越靠右邊, 類似一種prior, 因為他可能更常出現, 所以可以調整z一開始的value讓值盡可能再sigmoid是接近1的, 如果有domain knowledge就可以調整af靠0或靠右



The bias term gives a class prior

- 同一層裡面的各neuron會是獨立的, 但其實應該各種label都有dependency, 所以一層是沒用的(沒有辦法互相share information)
 - 想要implement XOR的話就是要function的組合, 簡單形狀的組合來模擬(hidden), 越多層情境越可以模擬(overfit?), 參數量越多越複雜
 - DNN: 很多層的MLP (FC), 多深要依據資料量、task難度而有不同情況
- 兩個layer的weight會以matrix來表示
- af:

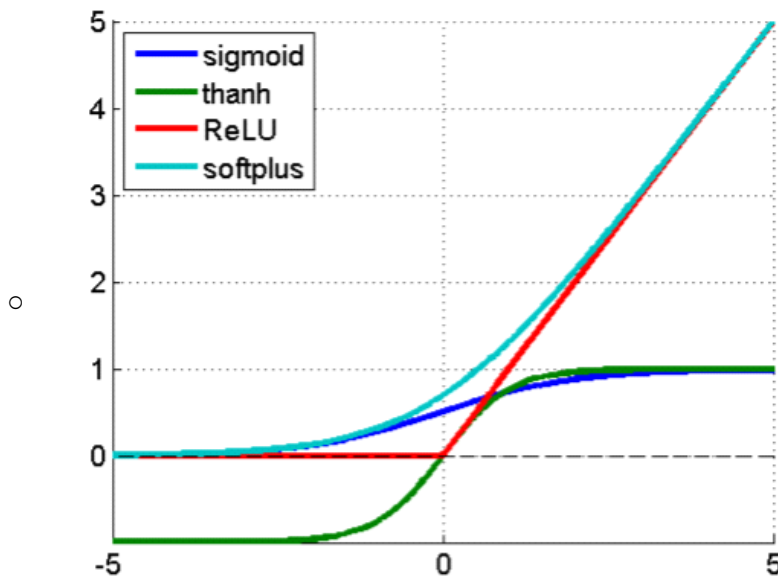
Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

boolean

linear

non-linear

- ReLU也是unbounded function, 沒有理論說哪個好, 但有常用的:



tanh = tanh (-1~1)

- 選non-linear的很重要，否則跟一層的linear transform會是一樣的，跌了很多層的linear會沒意義
- Loss function design 好的函式
 - maximize reward, minimize cost, 希望可以接近真正y這個數值
 - 預測出來的y要跟真正的y越近越好, distribution要接近越好
 - cross entropy: output distribution越低越好, 要跟真實的越近越好
 - square-loss: output是一個value的話要去相減算平方和

Square loss

$$C(\theta) = (1 - \hat{y}f(x; \theta))^2$$

Hinge loss

$$C(\theta) = \max(0, 1 - \hat{y}f(x; \theta))$$

- Logistic loss

$$C(\theta) = -\hat{y} \log(f(x; \theta))$$

Cross entropy loss

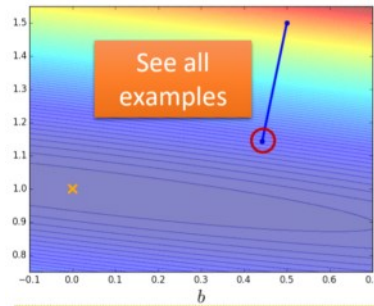
$$C(\theta) = -\sum \hat{y} \log(f(x; \theta))$$

Others: large margin, etc.

- Optimization
 - DL常用GD
 - 可以用暴力法，但是用篇微分=0比較快，但其實會不知道整個space太大了不一定找的到，所以才用GD(全部的error加起來取平均)。但有些缺點所以有BP
 - 看全部的数据才去update會很慢，所以才有SGD
 - 看一個sample就先小update一次，每一次只找一個sample去看loss用篇微分update (假設所有sample取到的機率一樣uniform)，所有變化的平均，比GD更快速達到想要的點

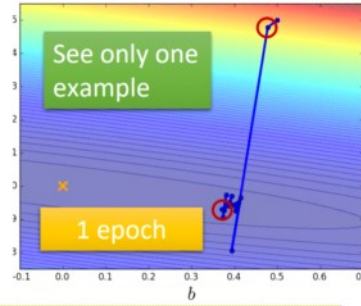
Gradient Descent

Update after seeing all examples



Stochastic Gradient Descent

If there are 20 examples, update 20 times in one epoch.



SGD approaches to the target point faster than gradient descent

- 每次挑一個batch去做update: mini-batch最快, SGD>GD

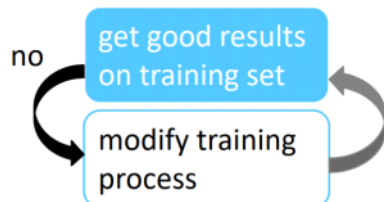
- SGD acc最高, >minibatch>GD

Handwriting Digit Classification



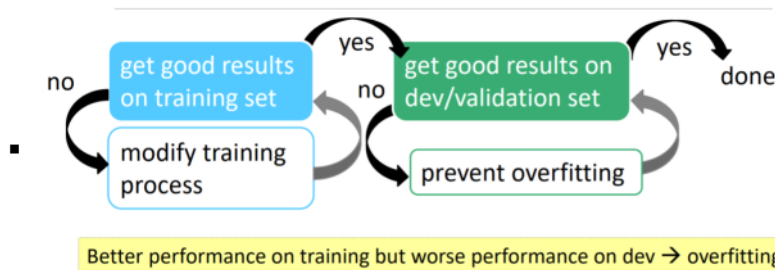
- 實作tips:

- 每次都從不同的點開始, random從不同開始的點找相對低點 (initialization要random) 才可以到global minima
- lr: 離目標遠的時候大步一點, 後期要小一點 才可以到local minima, adaptive lr自動調整
- 每個epoch都要shuffle一次, 才可以學到data真正的distribution, 讓NN不是用背的順序
- 每個epoch的batch size要固定, implement的時候訓練才會快
- validation是為了模擬testing情境, 去反應真的再使用之表現如何, 根據validation去知道model performance



Possible reasons

- no good function exists: bad hypothesis function set
→ reconstruct the model architecture
- cannot find a good function: local optima
→ change the training strategy



- Overfitting
 - 要去收集更多training data，加入dropout

HW tips:

- FastText、gensim、GloVe、w2v可以用來訓練embedding
- 前處理: SpaCy、NLTK
- 利用special token來分開這句話是誰說的，或是用不同speaker train不同的embedding再去concat
- RNN 可以最後變成一個vector輸出，或是用mean/max pooling (不然用個XGB看哪個維度重要變成weighted sum)
- 經過attention以後的vector可以把attention前的vector進行互動 EX: element-wise product跟element-wise 相減，再去concatenate然後在丟到下一層
- 比較兩個vector: inner product、MLP、cos-similarity
- loss: binary crossentropy、softmax不一定好
- SpaCy會去做一些前處理的function可以關掉以加速，其中 1:4/1:9的negative sampling比較好