Applied ML is a very iterative process.

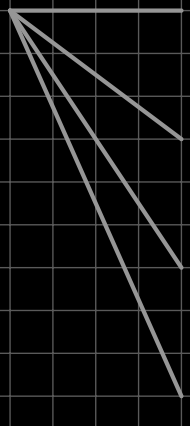
When doing with big data set, we can set our valid set. test set a much smaller percentage. Like we have 1000,000 data, we can use 1% as valid set, 1% as test set.

Make sure that the train set, test set come from the same distribution.

train / dev / test set

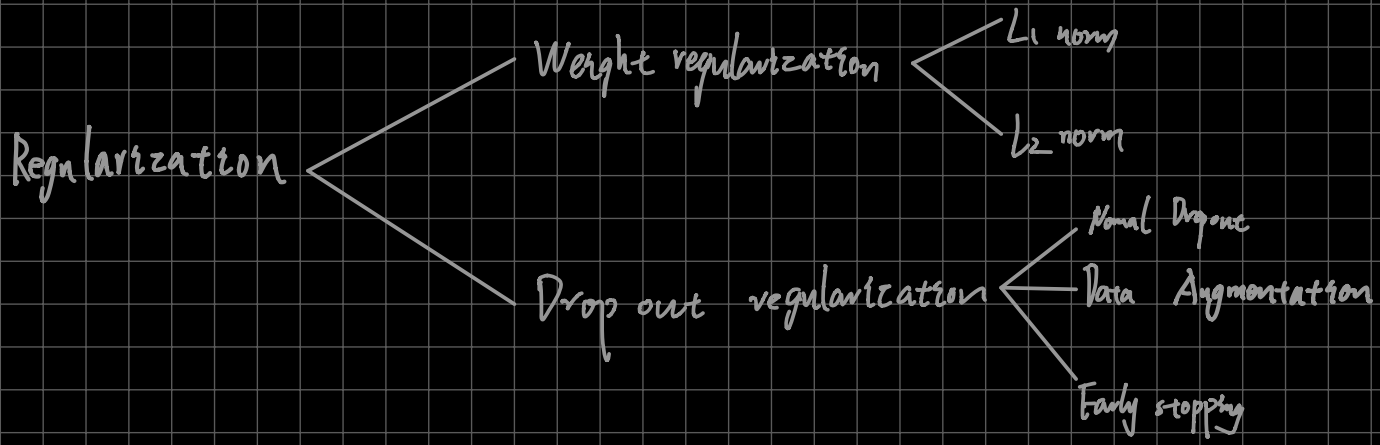
dev: development

Bias and variance:



	train error	dev error
high bias	15%	16%
high variance	1%	10%
high bias and high variance	15%	30%
just right	1%	2%

Basic recipe for ML (a systemetic way to do that)



Input normalization ⇒ ① 提高训练速度 ② 提高模型收敛速度（避免进入梯度饱和区） ③ 提高模型的稳定性（避免"蝴蝶效应"，降低模型的参数敏感度） ④ 提高模型的泛化性能（使得 model 能对不同分布的 input data 作出较好的适应）

Gradient descent problems:

Global method
- Gradient vanishing (梯度消失/饱和) ⇒
  RNN → LSTM ; sigmoid/tanh → relu ; batch normalization
- Gradient exploding (梯度爆炸) ⇒

Gradient clipping (梯度剪裁) algorithm
- 梯度上限剪裁
- 梯度同比收缩

Partial method : Weight Initialization

Idea: Init the weight matrix so that the gradient will not increase too quickly (>> 1) or decrease too quickly (<< 1).

Method:

Activations:

- He- initialization: relu ①
- Xavier/Glorot - initialization: tanh ②
- Lechun - initialization: tanh ③

① He : $W^{[L]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[L-1]}}\right)$

② Xavier : $W^{[L]} = np.random.randn(shape) * np.sqrt\left(\sqrt{\frac{1}{n^{[L-1]}}}\right)$

③ Lechun: $W^{[L]} = np.random.randn(shape) * np.sqrt\left(\sqrt{\frac{2}{n^{[L-1]} + n^{[L]}}}\right)$

△ Gradient Checking Notes:

- Remember regularization
- Only to debug, not used in training
- Don't work with dropout
- Run at random initialization

Mini-batch size choosing:

① If small train set, use batch gradient descent (size (mini-batch) = m)

② Use typical mini-batch size:

64, 128, 256 ...

while making sure that mini-batch fit in GPU/CPU memory.

Optimal Algorithms that are faster than gradient descent:

Exponentially weighted averages

$V_t = \beta \cdot V_{t-1} + (1-\beta) \cdot \theta_t$ ⇒ values for $\frac{1}{1-\beta}$ day's average.

加多数

$\beta \uparrow$, more smoother, adapts more slowly, rely more on the past.

why it called Exponentially Weighted Averages ?

$V_k = \beta \cdot V_{k-1} + (1-\beta) \cdot \theta_k$

$\Downarrow$

$\beta \cdot V_{k-2} + (1-\beta) \cdot \theta_{k-1}$

$\Downarrow$

$\vdots$

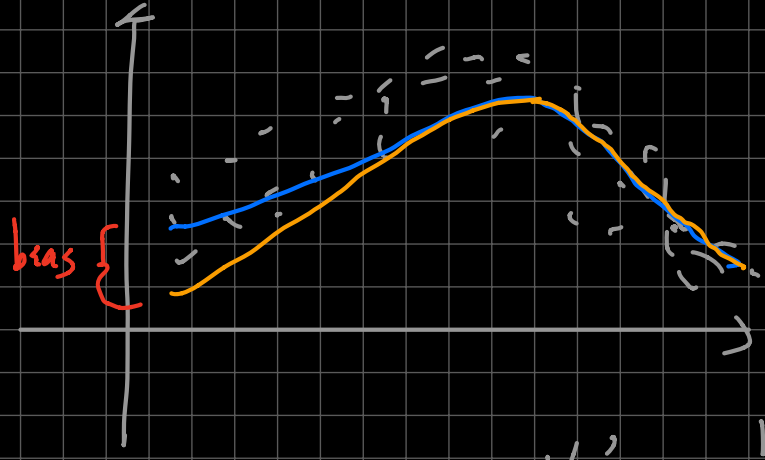
$V_k = (1-\beta)\theta_k + \beta \cdot (1-\beta) \cdot \theta_{k-1} + \beta \cdot (1-\beta)^2 \cdot \theta_{k-2} \cdots + \beta (1-\beta)^k \cdot \theta_0$

Bias correction ⇒ Solve the initialization problem

Originally, we init $V_0 = 0$, then we have:

$V_1 = \underbrace{\beta \cdot V_0}_{\Downarrow 0} + (1-\beta) \cdot \theta_1$, thus in the first few estimations, the value is usually too small.

That's why we introduce bias correction.



bias correction: $V_t' = \frac{V_t}{\underbrace{1-\beta^t}_{\text{added, when } t \text{ is large,}}} = \frac{\beta \cdot V_{t-1} + (1-\beta)\theta_t}{1-\beta^t}$

$1-\beta^t \to 1$, thus $V_t' = V_t$.

So after the 'warm-up', V_t curve and V_t' curve are quite overlapped

Momentum:

$$\begin{cases} \boxed{V_{dw}} = \beta \cdot V_{dw} + \boxed{(1-\beta) \cdot dw} \\ \boxed{V_{db}} = \beta \cdot V_{db} + \boxed{(1-\beta) \cdot db} \end{cases}$$

V_{dw} init to 0

$\Rightarrow$ accelerater

Gradient descent with momentum:

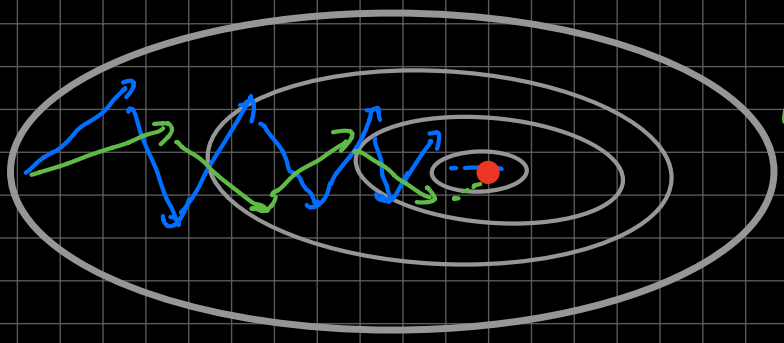
$w = w - \alpha \underline{\underline{V_{dw}}}$

$b = b - \alpha \cdot \underline{\underline{V_{db}}}$

original GD $\longrightarrow$ $w = w - \alpha \cdot dw$, $b = b - \alpha \cdot db$

Hyperparameters : α (learning rate), β (momentum factor)

Function of momentum => smooth and speed up the learning process.



GD with Momentum

GD

RMS$_{prop}$:

$$\begin{cases} S_{dw} = \beta \cdot S_{dw} + (1-\beta) \cdot dw^2 \\ \quad \rightarrow \text{init to } 0 \\ S_{db} = \beta \cdot S_{db} + (1-\beta) \cdot db^2 \end{cases}$$

$$w = w - \alpha \cdot \frac{dw}{\sqrt{S_{dw}}} \; ; \; b = b - \alpha \cdot \frac{db}{\sqrt{S_{db}}}$$

Intuition: when dw is big, S_{dw} is big, $\frac{1}{\sqrt{S_{dw}}}$ is small. so w's learning step is relative smaller.

Adam : Momentum + RMSprop

$\Rightarrow$ It's been proved to be useful in a wide range of neural networks.

$$\begin{cases} V_{dw} = \beta_1 \cdot V_{dw} + (1-\beta_1) \cdot dw, \quad V_{db} = \beta_1 \cdot V_{db} + (1-\beta_1) \cdot db \\ S_{dw} = \beta_2 \cdot S_{dw} + (1-\beta_2) \cdot dw^2, \quad S_{db} = \beta_2 \cdot S_{db} + (1-\beta_2) \cdot db^2 \\ V_{dw}^{corrected} = \dfrac{V_{dw}}{1-\beta_1^t} \; ; \quad V_{db} = \dfrac{V_{db}}{1-\beta_1^t} \\ S_{dw}^{corrected} = \dfrac{S_{dw}}{1-\beta_2^t} \; ; \quad S_{db} = \dfrac{S_{db}}{1-\beta_2^t} \end{cases}$$

$$w = w - \alpha \cdot \frac{V_{dw}}{\sqrt{S_{dw}^{corrected}} + \varepsilon}$$

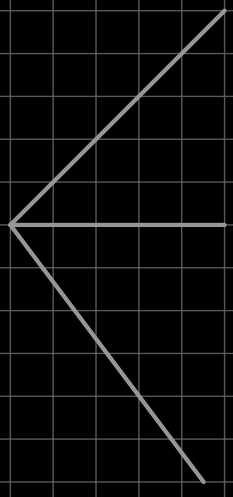
$$b = b - \alpha \cdot \frac{V_{db}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

Hyperparameters : $\beta_1, \beta_2, \varepsilon$

default : $0.9, 0.999, 10^{-8}$

Learning rate decay:

$$\alpha = \frac{\alpha_0}{1 + \underbrace{r}_{\text{decay rate}} * \#\text{epoch}}$$



$$\alpha = 0.95^{\#\text{epoch}} \cdot \alpha_0$$

$$\alpha = \frac{k\,\alpha_0}{\sqrt{\#\text{epoch}}}$$