

Contents

1	1. optimization algorithms	1
1.1	1.1. mini-batch gradient descent	1
1.2	1.2. understanding mini-batch gradient descent	1
1.3	1.3. exponentially weighted averages	4
1.4	1.4. understanding exponentially weighted averages	4
1.5	1.5. bias correction in exponentially weighted averages	7
1.6	1.6. gradient descent with momentum	7
1.7	1.7. RMSprop	7
1.8	1.8. Adam optimization algorithm	9
1.9	1.9. learning rate decay	10
1.10	1.10. the problem of local optima	10

contents

- 1. optimization algorithms
- 1.1. mini-batch gradient descent
- 1.2. understanding mini-batch gradient descent
- 1.3. exponentially weighted averages
- 1.4. understanding exponentially weighted averages
- 1.5. bias correction in exponentially weighted averages
- 1.6. gradient descent with momentum
- 1.7. RMSprop
- 1.8. Adam optimization algorithm
- 1.9. learning rate decay
- 1.10. the problem of local optima

1 1. optimization algorithms

1.1 1.1. mini-batch gradient descent

- batch gradient descent: 遍历所有训练样本
- mini-batch gradient descent: 一次处理一个 mini-batch

各符号的表示:

- $X^{(i)}$ 表示第 i 个训练样本
- $z^{[l]}$ 表示第 l 层的激活函数的输入
- $X^{\{t\}}, Y^{\{t\}}$ 表示第 t 个 mini-batch 的样本

假设有 5000000 个样本, 那么在一个 epoch 中, 遍历 5000 个 mini-batch, 每个 mini-batch (1000 个样本) 中 【一个 iteration】, 先前向计算, 然后算这 1000 个样本的 loss, 然后反向传播。

注意: 一个 mini-batch 就反向传播一次, 更新一次参数, 所以一个 epoch 就更新了 5000 次参数。

1.2 1.2. understanding mini-batch gradient descent

可见, 使用 mini-batch 的 cost 会有不小的抖动和噪音。

- batchsize=m: batch gradient descent。一般可以顺利到达 min。但耗时长。
- batchsize=1: stochastic gradient descent。很可能在 min 附近徘徊。而且失去了 vectorization 可以带来了加速
- batchsize between 1,m:
- 充分利用 vectorization
- make progress without processing entire training set

Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(1000)} & \dots & x^{(2000)} & \dots & x^{(m)} \end{bmatrix}$$

(n_x, m)

$$x^{\{t\}} \quad (n_x, 1000) \quad x^{\{2\}} \quad (n_x, 1000) \quad \dots \quad x^{\{5,000\}} \quad (n_x, 1000)$$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(1000)} & \dots & y^{(2000)} & \dots & y^{(m)} \end{bmatrix}$$

$(1, m)$

$$y^{\{t\}} \quad (1, 1000) \quad y^{\{2\}} \quad (1, 1000) \quad \dots \quad y^{\{5,000\}} \quad (1, 1000)$$

What if $m = 5,000,000$?
5,000 mini-batches of 1,000 each
Mini-batch t : $\underline{x^{\{t\}}, y^{\{t\}}}$

06:09 / 11:28

Andrew Ng

Figure 1: mini-batch-gradient-descent-introduction.png

Mini-batch gradient descent

repeat {
for $t = 1, \dots, 5000$ {

1 step of gradient descent
using $\underline{x^{\{t\}}, y^{\{t\}}}$
(as if $m=1000$)

Forward prop on $X^{\{t\}}$.

$$\begin{aligned} z^{\{t\}} &= w^{\{1\}} X^{\{t\}} + b^{\{1\}} \\ a^{\{t\}} &= g^{\{1\}}(z^{\{t\}}) \\ &\vdots \\ a^{\{T\}} &= g^{\{T\}}(z^{\{T\}}) \end{aligned}$$

Vectorized implementation (1000 examples)

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^1 l(y^{(i)}, a^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_i \|w^{(i)}\|_F^2$.

Backprop to compute gradients wrt $J^{\{t\}}$ (using $(x^{\{t\}}, y^{\{t\}})$)

$$w^{\{t\}} = w^{\{t\}} - \alpha \nabla J^{\{t\}}, \quad b^{\{t\}} = b^{\{t\}} - \alpha \nabla b^{\{t\}}$$

"1 epoch"
1 pass through training set.

3

11:03 / 11:28

Andrew Ng

Figure 2: mini-batch-gradient-descent.png

Training with mini batch gradient descent

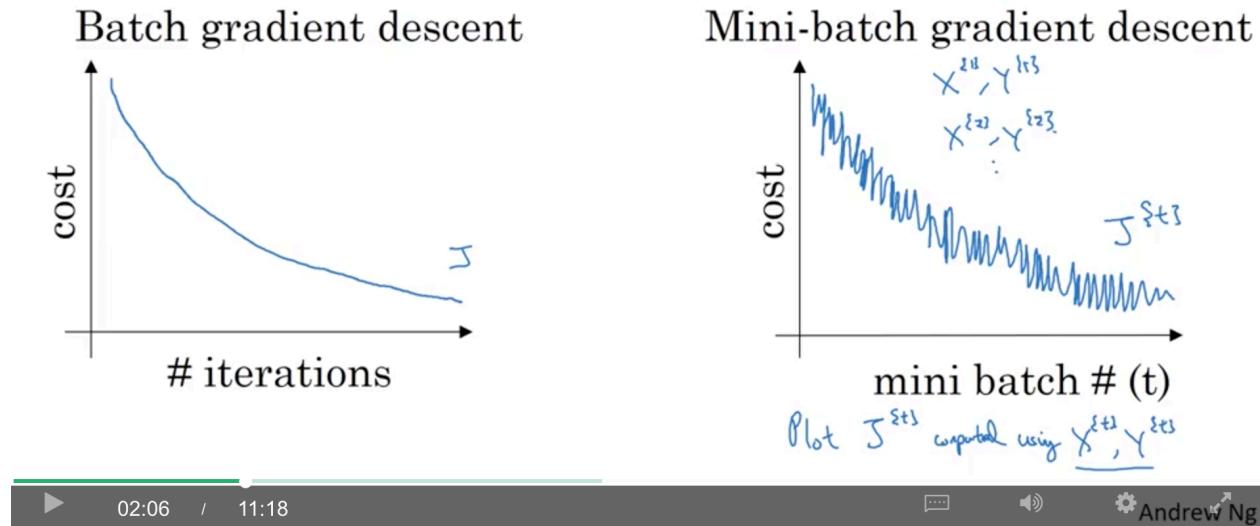


Figure 3: mini-batch-gradient-descent-cost.png

Choosing your mini-batch size

- If mini-batch size = m : Batch gradient descent. $(X^{(1)}, Y^{(1)}) = (X, Y)$.
- If mini-batch size = 1 : Stochastic gradient descent. Every example is its own $(X^{(1)}, Y^{(1)}) = (x^{(1)}, y^{(1)}) \dots (x^{(n)}, y^{(n)})$ mini-batch.

In practice: Somewhere in-between 1 and m

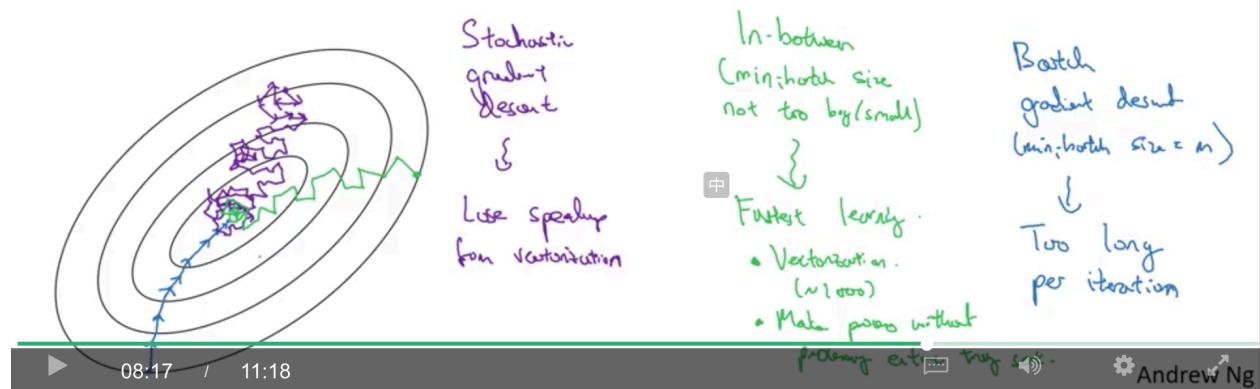


Figure 4: mini-batch-gradient-descent-choose-batchsize.png

- 训练集很小时 ($m \leq 2000$): 使用 batch gradient descent
- 其他情况下, 经典的 batchsize: 2 的指数 (64, 128, 256, 512), 一般较少用 1024, 非 2 的指数也比较少用 (如 1000)
- make sure mini-batch 里的 $X^{\{t\}}, Y^{\{t\}}$ 能够在 cpu/gpu 中被容纳下

Choosing your mini-batch size

If small tiny set: Use batch gradient descent.
($m \leq 2000$)

Typical mini-batch sizes:

$$\rightarrow 64, 128, 256, 512 \quad \frac{1024}{2^{10}}$$

Make sure minibatch fit in CPU/GPU memory.

$$X^{\{t\}}, Y^{\{t\}}$$



Figure 5: mini-batch-gradient-descent-choose-size.png

1.3 1.3. exponentially weighted averages

比 gradient descent 更快的算法大多使用指数平滑类的算法。通过图中 v_t 的计算, 可以得到图中的红线, 即 exponentially weighted (moving) average

$$v_t = \beta v_{t-1} + (1 - \beta)\theta_t$$

注意, β 对应的是 v_{t-1} , 而 $1 - \beta$ 对应的是 θ_t 。

而, 曲线描绘了过去大约 $\frac{1}{1-\beta}$ 天的变量取值的平均情况, 所以, β 越大, 窗口越大, 曲线越滞后, 对变量的变化也越不敏感。图中, 红线是 0.9[相当于过去 10 天], 绿线是 0.98[相当于过去 50 天], 黄线是 0.5[相当于过去 2 天]。还有一个常用的是 0.99[相当于过去 100 天]。

1.4 1.4. understanding exponentially weighted averages

$(1 - \epsilon)^{1/\epsilon}$ 约等于 $1/e$, 而 $\epsilon = 1 - \beta$ 。所以是 $\beta^{1/(1-\beta)}$, 所以前面提到, 大约是 $1/(1 - \beta)$ 天

好处:

- 计算简单
- 省内存 (只需要存一个变量 v 就行了)

Temperature in London

$$\theta_1 = 40^\circ\text{F} \quad 4^\circ\text{C} \leftarrow$$

$$\theta_2 = 49^\circ\text{F} \quad 9^\circ\text{C}$$

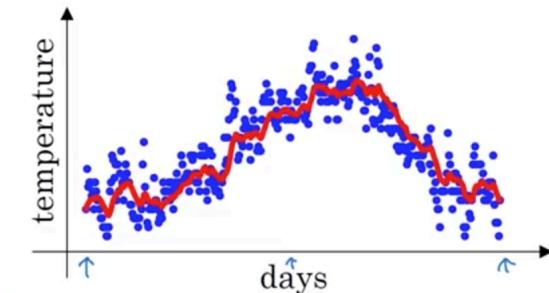
$$\theta_3 = 45^\circ\text{F} \quad \vdots$$

$$\vdots$$

$$\theta_{180} = 60^\circ\text{F} \quad 15^\circ\text{C}$$

$$\theta_{181} = 56^\circ\text{F} \quad \vdots$$

$$\vdots$$



$$V_0 = 0$$

$$V_t = 0.9 V_{t-1} + 0.1 \theta_t$$

$$V_2 = 0.9 V_1 + 0.1 \theta_2$$

$$V_3 = 0.9 V_2 + 0.1 \theta_3$$

$$\vdots$$

$$V_t = 0.9 V_{t-1} + 0.1 \theta_t$$



Figure 6: exponential-weight-average-introduction.png

Exponentially weighted ^{Moving} averages

$$V_t = \beta V_{t-1} + (1-\beta) \theta_t \leftarrow$$

$\beta = 0.9$: ≈ 10 days' tapers.
 $\beta = 0.98$: ≈ 50 days
 $\beta = 0.5$: ≈ 2 days

V_t is approximately
average over
 $\rightarrow \approx \frac{1}{1-\beta}$ days'
temperature.

$$\frac{1}{1-0.98} = 50$$

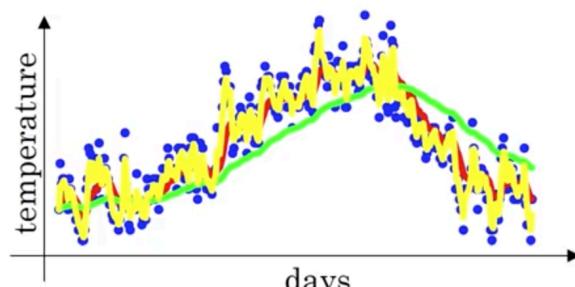


Figure 7: exponential-weight-average-formula.png

Exponentially weighted averages

$$v_t = \beta v_{t-1} + (1 - \beta) \theta_t$$

$$v_{100} = 0.9 v_{99} + 0.1 \theta_{100}$$

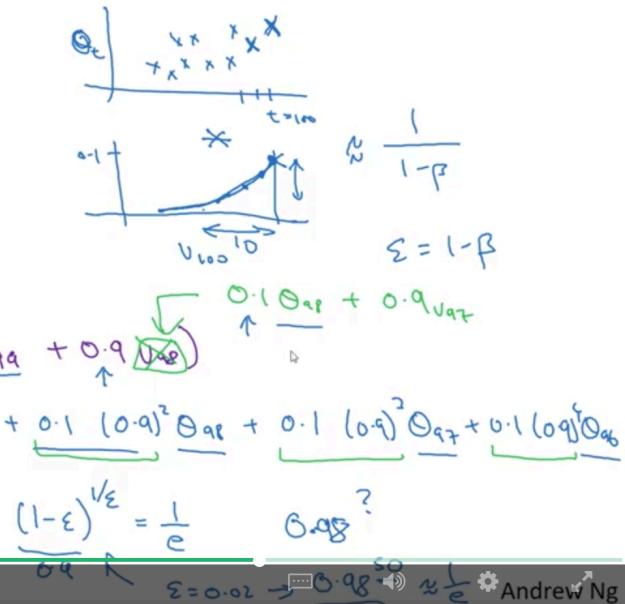
$$v_{99} = 0.9 v_{98} + 0.1 \theta_{99}$$

$$v_{98} = 0.9 v_{97} + 0.1 \theta_{98}$$

...

$$\begin{aligned} \rightarrow v_{100} &= 0.1 \theta_{100} + 0.9 (0.1 \theta_{99}) + 0.9 (0.9 \theta_{98}) \\ &= 0.1 \theta_{100} + 0.1 \times 0.9 \cdot 0.9 \theta_{99} + 0.1 (0.9)^2 \theta_{98} + 0.1 (0.9)^3 \theta_{97} + 0.1 (0.9)^4 \theta_{96} \end{aligned}$$

$$0.9^{\textcircled{10}} \approx 0.35 \approx \frac{1}{3}$$



6:51 / 9:41

Andrew Ng

Figure 8: exponential-weight-average-analyze.png

Implementing exponentially weighted averages

$$v_0 = 0$$

$$v_1 = \beta v_0 + (1 - \beta) \theta_1$$

$$v_2 = \beta v_1 + (1 - \beta) \theta_2$$

$$v_3 = \beta v_2 + (1 - \beta) \theta_3$$

...

$$\begin{aligned} v_0 &:= 0 \\ v_0 &:= \beta v + (1-\beta) \theta_1 \\ v_0 &:= \beta v + (1-\beta) \theta_2 \\ &\vdots \\ \rightarrow v_0 &= 0 \\ \text{Repeat } &\xi \\ \text{Get next } &\theta_t \\ v_0 &:= \beta v_0 + (1-\beta) \theta_t \end{aligned}$$

9:10 / 9:41

Andrew Ng

Figure 9: exponential-weight-average-implementation.png

1.5 1.5. bias correction in exponentially weighted averages

当 $\beta = 0.98$ 时理论上应该得到绿线，但实际得到的往往是紫线（因为 v 初始是 0，所以而 β 接近 1，所以起始的几个值都比较小，所以起点比较低），通过 bias correction 可以从紫线纠正到绿线。

$v_t = \frac{v_{t-1}}{1-\beta^t}$ ，因为 $0 < \beta < 1$ ，所以 t 越大， β^t 越小，分母越大，所以 v_t 越小。反之， t 越小， v_t 越大，就可以补上前面提到的 t 比较小时较低的起点。

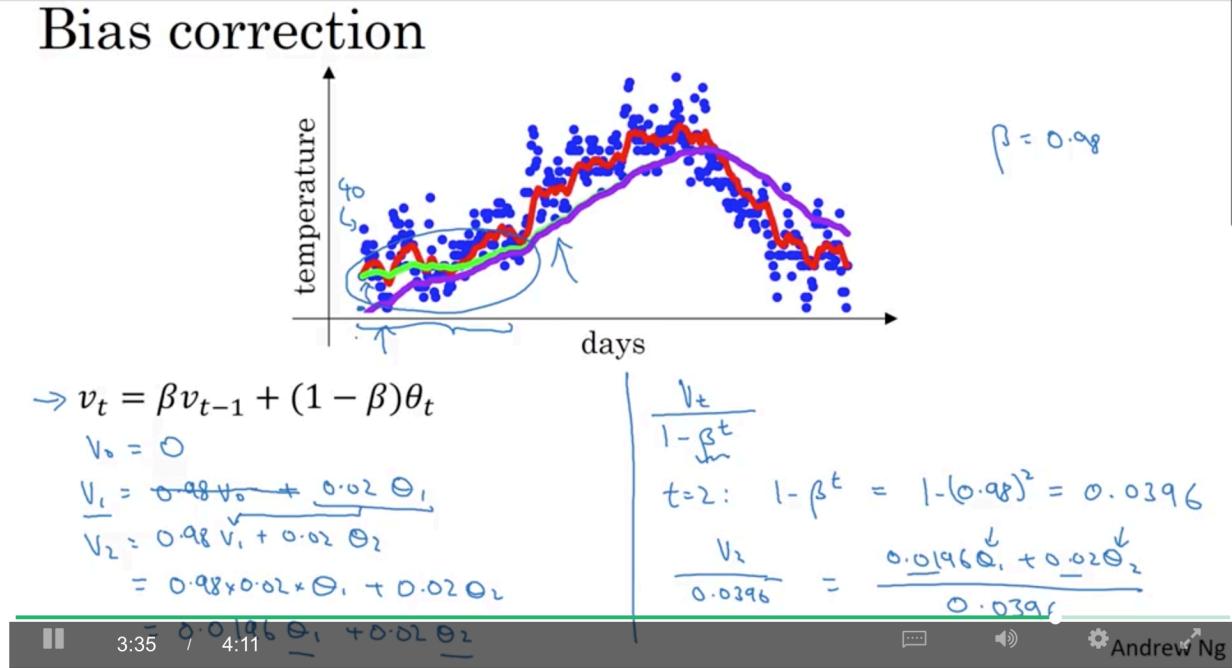


Figure 10: exponential-weight-average-bias-correction.png

1.6 1.6. gradient descent with momentum

如果 learning rate 太大，可能出现紫线的情况，波动太大 (oscillation，振动，波动)。所以，希望纵向 slower，而横向 faster。

所谓 momentum，就是对 dW 和 db ，分别使用 exponentially weighted average，计算对应的 v_{dW} 和 v_{db} ，更新时，用这两个 v 替换掉原来的 dW 和 db 。

这样，能够在横向保持方向不变，而在纵向上，有『平滑』的效果，减少波动。

公式中， dW 相当于加速度， v_t 相当于速度 (velocity)， β 相当于摩擦力 (friction)

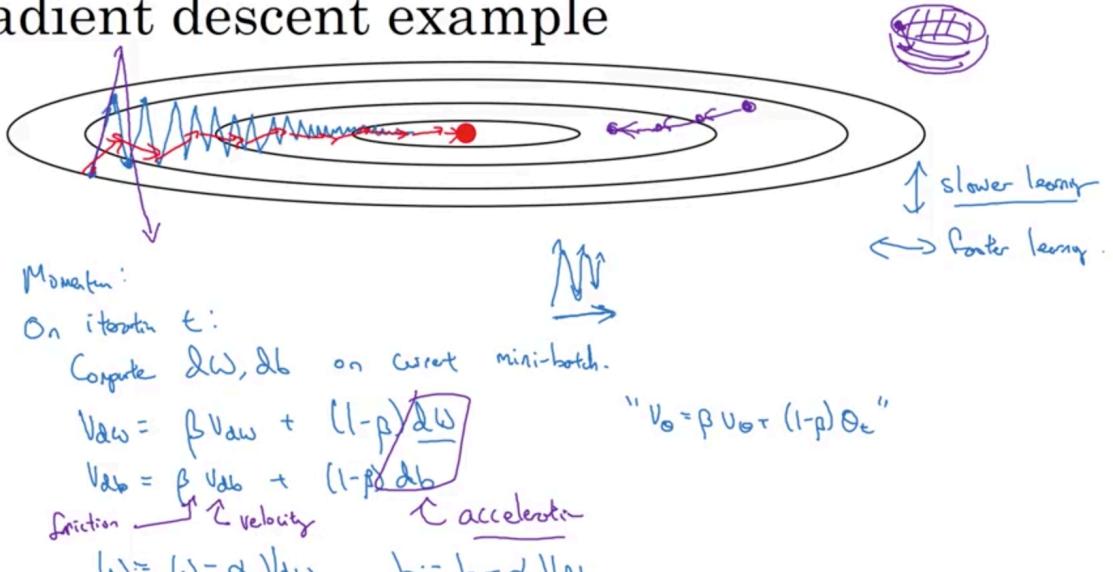
实现时的几个注意点：

- 最常取的 $\beta = 0.9$ ，也就是相当于最近 10 个 iterations 的 gradients。
- 而且不一定会用 bias correction ($\frac{v_{dW}}{1-\beta^t}$)，因为 10 个 iteration 左右，这个 bias 就差不多没了。
- 有的实现是去掉 $(1 - \beta)$ (图中的紫色部分，《深度学习》P182 的动量方法就和这个方法类似)，而这相当于 v_{dW} 缩小了 $(1 - \beta)$ 倍，所以更新时， α 相当于要根据 $\frac{1}{1-\beta}$ 进行相应的变化。这两种方法其实都行，但会影响到如何选择最佳的 α 。而这种方法可能有一些 less intuitive 的就是，最后要调整 β 时，会影响到 v_{dW} 和 v_{db} 的 scaling，从而，可能还要再修改学习率 α 。

1.7 1.7. RMSprop

其中 $(dW)^2$ 指的是 element-wise squaring。

Gradient descent example



▶ 6:14 / 9:20 ⏺ 🔍 Andrew Ng

Figure 11: momentum-introduction.png

Implementation details

$$v_{dw} = 0, \quad v_{db} = 0$$

On iteration t :

Compute dW, db on the current mini-batch

$$\begin{aligned} \rightarrow v_{dw} &= \beta v_{dw} + (1-\beta) \cancel{dW} \\ \rightarrow v_{db} &= \beta v_{db} + (1-\beta) \cancel{db} \end{aligned} \quad \left| \begin{array}{l} \cancel{v_{dw}} = \beta \cancel{v_{dw}} + \cancel{dW} \leftarrow \\ \cancel{v_{db}} = \beta \cancel{v_{db}} + \cancel{db} \end{array} \right.$$

$$W = W - \underbrace{\alpha v_{dw}}, \quad b = \cancel{b} - \underbrace{\alpha v_{db}}$$

Hyperparameters: α, β

$$\beta = 0.9$$

average over time α gradients

▶ 8:49 / 9:20 ⏺ 🔍 Andrew Ng

Figure 12: momentum-implementatioin.png

$$s_{dW} = \beta s_{dW} + (1 - \beta_2)(dW)^2$$

$$s_{db} = \beta s_{db} + (1 - \beta_2)(db)^2$$

$$w = w - \alpha \frac{dW}{\sqrt{s_{dW}}}$$

$$b = b - \alpha \frac{db}{\sqrt{s_{db}}}$$

例如图中的例子，横轴是 W ，纵轴是 b ，希望横轴 fast，纵轴 slow，也就是，希望 s_{dW} 小一些， w 就减少得快一点，同理， s_{db} 大一些， b 就减小得慢一点。

而事实上，图中蓝线部分，也就是每一轮 loss 的变化方向，可见，横轴方向的分量比较小，也就是 dW 比较小，所以 s_{dW} 也比较小，就满足了上述希望，而正好是比较小的 dW 除以一个比较小的 $\sqrt{s_{dW}}$ ，实现了『自适应』，结果如图中的绿线所示。

因此，可以使用一个比较大的学习率，而不会在纵轴上 diverge(偏离，也就是上面说的抖动的振幅)太多。

实际上，不只是 b 和 W ，是很多维的参数 vector。

另外，这里用了 β_2 是为了和 momentum 的 β 区分开，后面要讲的 Adam 就是结合了 momentum 和 RMSprop 的方法。

另外，分母的 ϵ 是为了防止当分母很小，趋近于 0 时，不被 0 除，一般 ϵ 约等于 10^{-8} 。

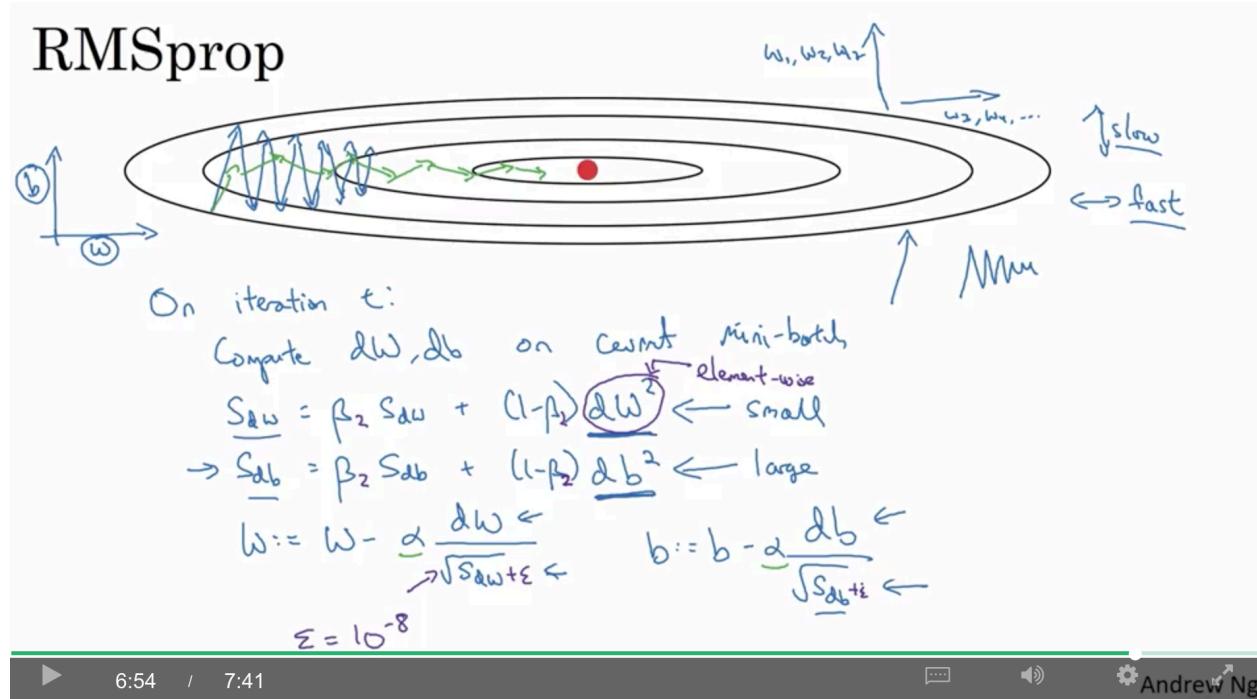


Figure 13: rmsprop.png

1.8 1.8. Adam optimization algorithm

超参：

- α 需要调
- β_1 : momentum 的参数，一般设为 0.9
- β_2 : RMSprop 的参数，一般设为 0.999

Adam optimization algorithm

$$V_{dw} = 0, S_{dw} = 0. \quad V_{db} = 0, S_{db} = 0$$

On iteration t :

Compute dW, db using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dW, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1) db \leftarrow \text{"moment"} \beta_1$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dW^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db \leftarrow \text{"RMSprop"} \beta_2$$

$$V_{dw}^{corrected} = V_{dw} / (1 - \beta_1^t), \quad V_{db}^{corrected} = V_{db} / (1 - \beta_1^t)$$

$$S_{dw}^{corrected} = S_{dw} / (1 - \beta_2^t), \quad S_{db}^{corrected} = S_{db} / (1 - \beta_2^t)$$

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}} \quad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}}$$



Figure 14: adam.png

- ϵ : 一般不调整, 设为 10^{-8}

Adam 的由来:

- Adaptive Moment estimation
- β_1 用来计算 dW , 称为第一矩 (first moment)
- β_2 用来计算 dW^2 的指数加权平均, 称为第二矩 (second moment)

1.9 1.9. learning rate decay

如果使用固定的 learning rate, 在 mini-batch gradient descent 中, 因为各个 batch 不一样, 可能最后是在 Minimum 附近徘徊, 不一定会收敛到 Minimum。

而如果在快到达 Minimum 的时候, 使用较小的学习率, 就可以在 minimum 附近以更小的振幅徘徊, 从而更接近 minimum。而这样的话, 在前面可以使用更大的学习率。

方法 1:

其他方法:

- $\alpha = 0.95^{epoch_num} \alpha_0$: exponentially decay
- $\alpha = \frac{k}{\sqrt{epoch_num}} \alpha_0$ 或者 $\alpha = \frac{k}{\sqrt{t} \alpha_0}$ 其中的 t 是第 t 个 mini-batch
- discrete staircase: 随 epoch 或者 t 的变化, 每次学习率变为之前的一半

1.10 1.10. the problem of local optima

在训练神经网络时, 大部分 loss function 的梯度为 0 的点不是 local optima, 而大多是 saddle point(鞍点), 即, 当维度很高时, 可能在某些维度是坡谷, 而在另一些维度是坡峰 (例如, 有 $2w$ 维, 在这 $2w$ 维全是坡谷的概率就是 $\frac{1}{2^{20000}}$, 很低的概率)

Hyperparameters choice:

- α : needs to be tune
- β_1 : 0.9 $\rightarrow (\underline{dw})$
- β_2 : 0.999 $\rightarrow (\underline{dw^2})$
- ϵ : 10^{-8}

Adam: Adaptive moment estimation



Figure 15: adam-hyperparameter.png

Learning rate decay

1 epoch = 1 pass through data.

$$\alpha = \frac{1}{1 + \underbrace{\text{decay-rate} * \text{epoch-num}}_{\downarrow}} \alpha_0$$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
:	:

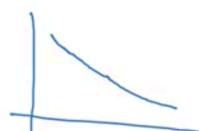
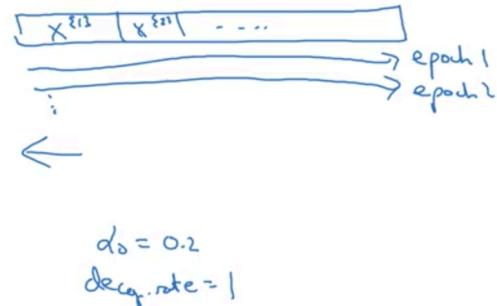
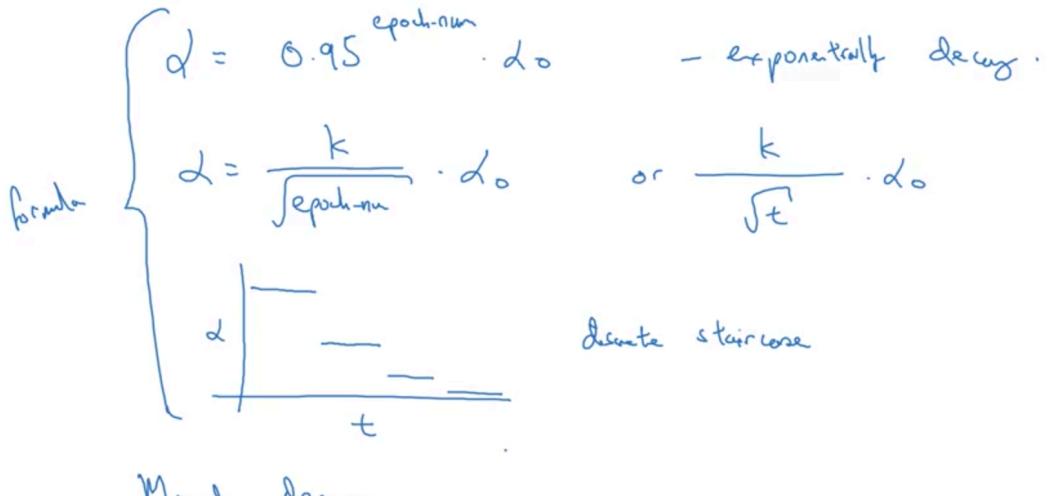


Figure 16: learning-rate-decay-formula1.png

Other learning rate decay methods



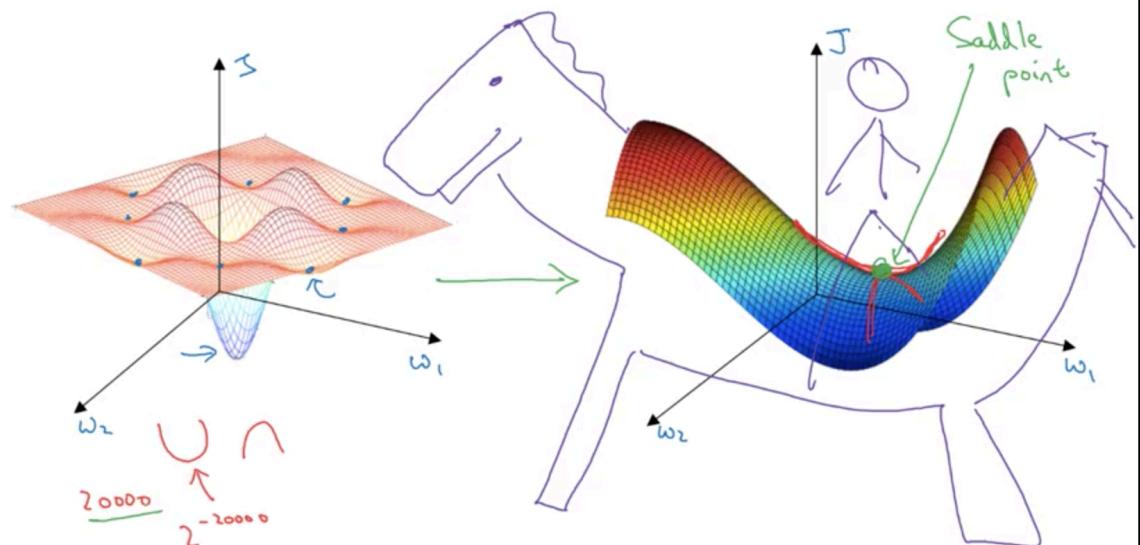
Manual decay.

5:37 / 6:44

Andrew Ng

Figure 17: learning-rate-decay-other-methods.png

Local optima in neural networks



3:17 / 5:23

Andrew Ng

Figure 18: local-optima-in-nn.png

可见，鞍点并不是 problem，而 problem 在于 plateaus，也就是梯度在很长一段时间里都很接近 0。例如图中的蓝线，可能这段路比较平坦，所以可能会走很久才到达下面那个蓝点，之后才可以比较快地下降（红线）。所以，momentum/RMSprop/Adam 就是针对这个问题进行的优化。

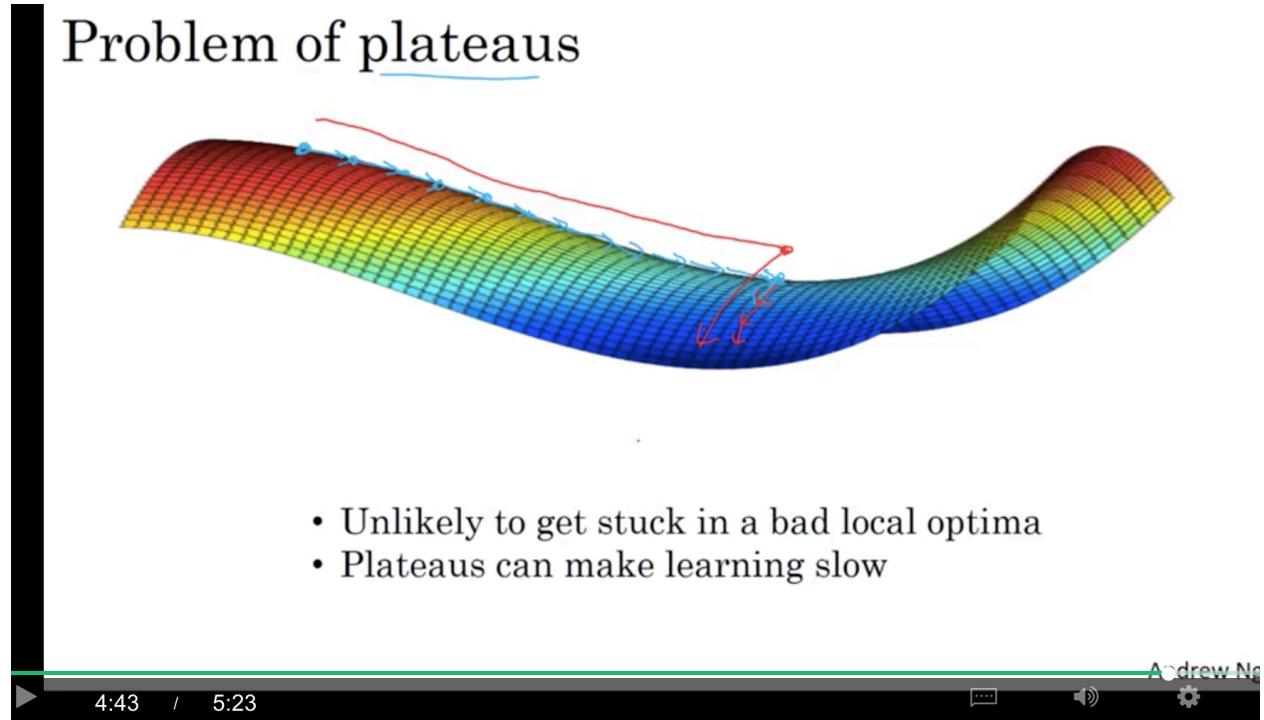


Figure 19: plateaus.png