Applying Machine Learning is a highly empirical process.

Big Data → Require large networks → Training is slow

So, optimization algorithms can speed up training time and increase efficiency.

Mini-batch Gradient Descent

Recall Batch Gradient Descent and vectorization on m samples.

$X = [x^{(1)}\ x^{(2)} \cdots x^{(m)}]$, $X: (n_x, m)$ $\quad Y = [y^{(1)}\ y^{(2)} \cdots y^{(m)}]$, $Y: (1, m)$

It's impractical to read the entire training set into memory if m is large. E.g. $m = 5 \times 10^6$

It's just too slow to run over the entire traing set for one iteration in gradient descent.

It would be better and faster that the gradient descent can start making progress before finishing the whole training set.

Thus, we can select 1000 samples as a mini-batch. $m = 5 \cdot 10^6$

$$X = [\underbrace{x^{(1)}\ x^{(2)} \cdots x^{(1000)}}_{X^{\{1\}}} \mid \underbrace{x^{(1001)} \cdots x^{(2000)}}_{X^{\{2\}}} \mid \cdots \mid \underbrace{\cdots x^{(m)}}_{X^{\{5000\}}}] \qquad Y = [\underbrace{y^{(1)}\ y^{(2)} \cdots y^{(1000)}}_{y^{\{1\}}} \mid \underbrace{y^{(1001)} \cdots y^{(2000)}}_{y^{\{2\}}} \mid \cdots \mid \underbrace{\cdots y^{(m)}}_{y^{\{5000\}}}]$$

Mini-batch t: $X^{\{t\}}: (n_x, 1000)$, $Y^{\{t\}}: (1, 1000)$, $t: 1, \ldots, 5000$

Now, we can process mini-batch in one iteration of gradient descent.

One Epoch (One single pass through the training set)

for $t = 1, \ldots, 5000$

Forward Propagation on $X^{\{t\}}$

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]})$$

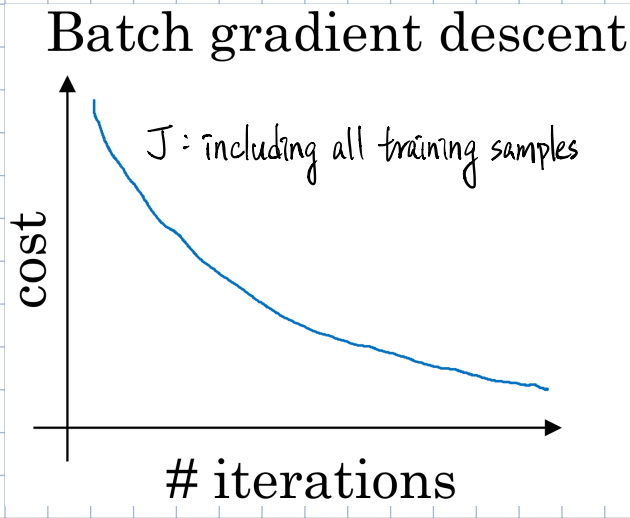
} Vectorization over 1000 samples

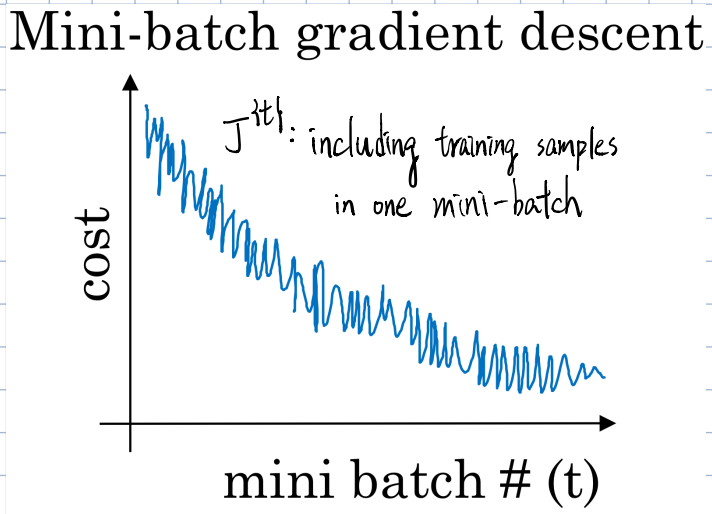
Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l=1}^{L} \|w^{[l]}\|_F^2$

Back Propagation to compute gradients w.r.t $J^{\{t\}}$ (using $X^{\{t\}}, Y^{\{t\}}$)

Now, with mini-batch, we can take 5000 gradient descent steps in one epoch, whereas with batch gradient descent, a single pass through the training set takes only one gradient descent step.

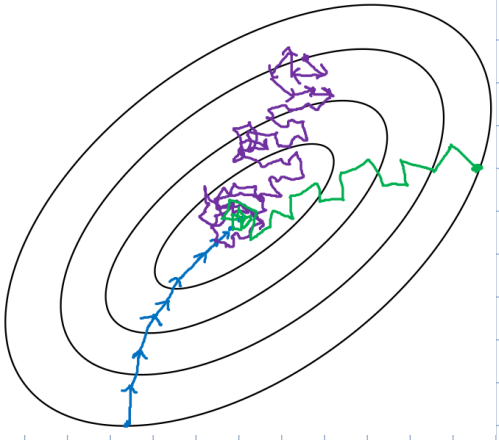
Understanding Mini-batch Gradient Descent





mini-batch size = m : Batch Gradient Descent

mini-batch size = 1 : Stochastic Gradient Descent (SGD)



SGD: Lose speed up from vectorization
Never hit the minimum and stay there.
Wander around, sometimes the wrong direction.

Batch Gradient Descent: Too long per iteration

The Best Solution: mini-batch size In-Between (eg: 1000)
Faster learing in one gradient descent step
Vectorization
Make progress without processing the entire training set

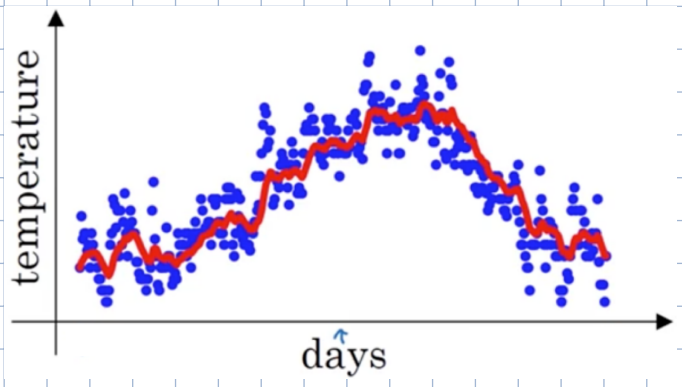
Practical Tips for choosing mini-batch size.

Small training set: Just use Batch Gradient Descent. Eg: $m \leq 1000$

Typical mini-batch size: 64, 128, 256, 512. Remember to use 2^n.

Fit in CPU/GPU memory.

Exponentially Weighted (Moving) Averages:



Day 1: $\theta_1 = 40°F$
Day 2: $\theta_2 = 49°F$
⋮
Day 365: $\theta_{365} = 40°F$

Exponentially weighted moving averages:
$V_0 = 0$
$V_1 = 0.9V_0 + 0.1\theta_1$
$V_2 = 0.9V_1 + 0.1\theta_2$
⋮
$V_t = 0.9V_{t-1} + 0.1\theta_t$

Equation: $V_t = \beta V_{t-1} + (1-\beta)\theta_t$ V_t : Approximately averaging over $\approx \frac{1}{1-\beta}$ days' temperature.

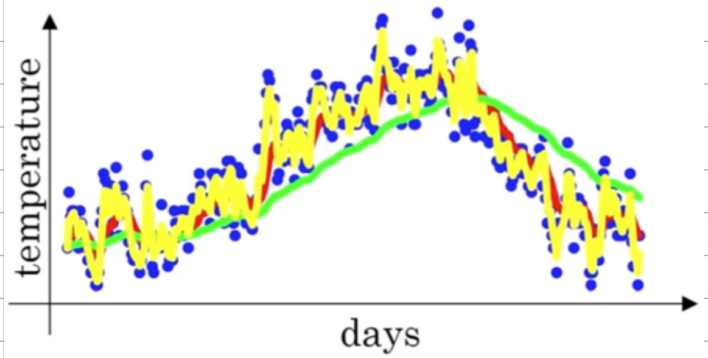
Eg: $\beta = 0.9$. $\frac{1}{1-\beta} = 10$, ∴ The temperature at day 10 is the average temperature from 1st to 10th days.

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

$\beta = 0.5$. $\frac{1}{1-\beta} = 2$

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

$$v_0 = 0$$
$$v_1 = \beta v_0 + (1-\beta)\theta_1$$
$$v_2 = \beta v_1 + (1-\beta)\theta_2$$
$$v_3 = \beta v_2 + (1-\beta)\theta_3$$
$$\cdots$$



If $\beta = 0.9$
$V_{100} = 0.1\theta_{100} + 0.1\cdot 0.9\cdot\theta_{99} + 0.1\cdot 0.9^2\theta_{98} + \cdots$

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$
$0.98^{50} \approx \frac{1}{e}$

$\varepsilon = 1-\beta$
$(1-\varepsilon)^{1/\varepsilon} = \frac{1}{e}$

When the percentage drops to $\frac{1}{e}$, its effect becomes minor.

Implementing exponentially weighted averages

$V_\theta = 0$
Repeat {
  Get next θ_t
  $V_\theta = \beta V_\theta + (1-\beta)\theta_t$
} It's fast and memory efficient.

Now I fully understand why exponentially weighted average is better than using window-based average. In window-based average, it needs an array to store values in a given window when calculating the average of each element, which is slow and memory consuming.

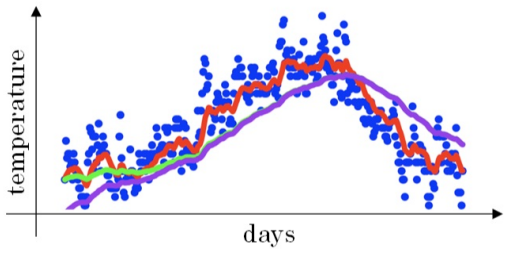
Bias Correction in Exponentially Weighted Averages:

If using the aforementioned implementation to implement Exponentially Weighted Averages when $\beta = 0.98$, we won't get the green curve, but the purple curve as shown in the right image. What? Why? I'm fooled by Andrew.

If looking closer: $V_0 = 0$, $V_1 = 0.98V_0 + 0.02\theta_1$, because the initial value is 0, so the starting few values are heavily influenced by the initial value $V_0 = 0$.

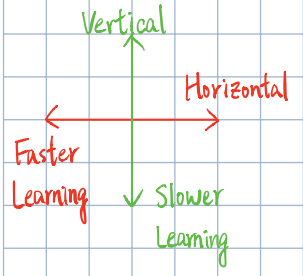
The fix : $V_t = \frac{\beta V_{t-1} + (1-\beta)\theta_t}{1-\beta^t}$. If $t=1$, $1-\beta^t = 0.02$
If $t=2$, $1-\beta^t = 0.0396$

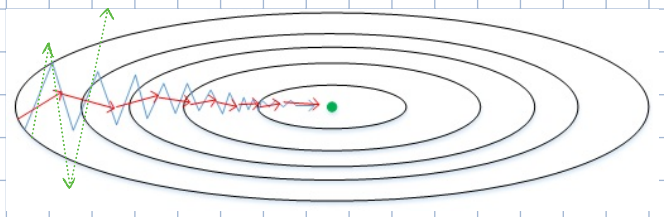
When t becomes larger, β^t approaches zero and has lesser influence.
Hence, we can get the green curve after applying Bias Correction.



Eg: $\theta_1 = 30$, $\theta_2 = 15$, $\beta = 0.5$, $V_0 = 0$.
$V_1 = 15$
$V_2 = 15$, $V_2^{\text{corrected}} = \frac{0.5\times 15 + 0.5\times 15}{1-0.5^2} = 20$

Gradient Descent with Momentum: It computes an exponentially weighted average of gradients, and use that gradient to update the weights.





In order to prevent oscillation, overshooting and potential divergence, the learning rate can't be too large.

Quiz: Given the blue and the red line on the left, which one has bigger β? The Red line.

On the vertical axis, we want the learning to be a bit slower.
On the horizontal axis, we want the learning to be faster.

On iteration t: Initial $V_{dw}=0, V_{db}=0$
Compute dw and db on the current mini-batch

$V_{dw} = \beta V_{dw} + (1-\beta)\, dw$ "Recall $V_t = \beta V_{t-1} + (1-\beta)\theta_t$"

$V_{db} = \beta V_{db} + (1-\beta)\, db$

friction, velocity, acceleration

Analogy for mechanical engineer

$W = W - \alpha V_{dw}$, $b = b - \alpha V_{db}$

Default: $\beta = 0.9$, Average out last 10 gradients.

Remember to use bias correction $\frac{V_t}{1-\beta^t}$

α: the Learning Rate

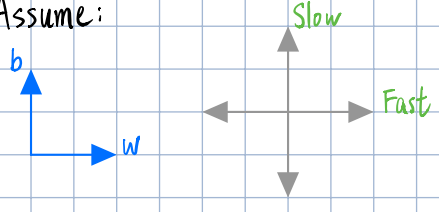
Like the temperature is smoothed out in exponentially weighted average, the gradient is also smoothed out.

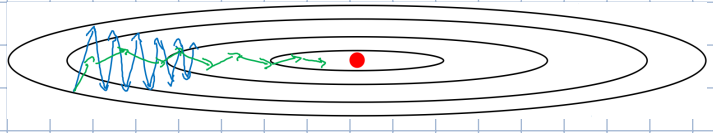
Eg:  The vertical direction got cancelled out, but the horizontal direction keeps being strong.

Sometimes, some literatures use $V_{dw} = \beta V_{dw} + (1-\beta) dW = \beta V_{dw} + dW$ as the formula.

RMSprop: Root Mean Squared Propagation

Assume:





Goal: We want to slow down the learning vertically and speed up (at least not slowing down) horizontally.

On iteration t:

Try to explain the equation. If not, rewatch video.

Compute dw, db on the current mini-batch

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dW^2$ element-wise

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

β_2 is used to differ itself from β in momentum above.

$$W = W - \alpha \frac{dW}{\sqrt{S_{dw}} + \varepsilon}, \quad b = b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

ε is a small number (e.g. 10^{-8}) to ensure numerical stability. ($S_{d*} = 0$)

Adam Optimization Algorithm

Research AdamW

Adam = Momentum + RMSprop

On iteration t: $V_{dW} = 0$, $S_{dW} = 0$, $V_{db} = 0$, $S_{db} = 0$

Compute dw, db on the current mini-batch

$$V_{dW} = \beta_1 V_{dW} + (1-\beta_1)\, dW \quad , \quad V_{db} = \beta_1 V_{db} + (1-\beta_1)\, db$$

$$S_{dW} = \beta_2 S_{dW} + (1-\beta_2) dW^2 \quad , \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$$

$$V_{dW}^{corrected} = \frac{V_{dW}}{1-\beta_1^t} \quad , \quad V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$$

$$S_{dW}^{corrected} = \frac{S_{dW}}{1-\beta_2^t} \quad , \quad S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$$

$$W = W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \varepsilon} \quad , \quad b = b - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \varepsilon}$$

Hyperparameters choice:

α: needs to be tuned (Learning Rate)

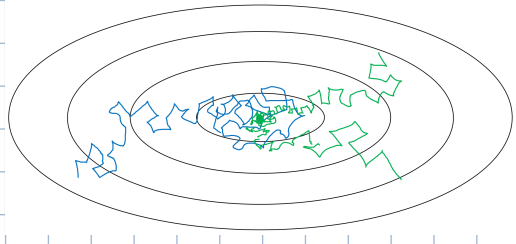
β_1: 0.9 rarely tuned

β_2: 0.999 rarely tuned

ε: 10^{-8} never tuned

Learning Rate Decay

Blue line represents the learning rate is constant, so it keeps wandering around and never converges.



Green line represents the learning rate is decaying, so it can reach the minimum.

One Epoch: One pass through the entire training set.

$$\alpha = \frac{1}{1 + DR \cdot EN}\alpha_0$$

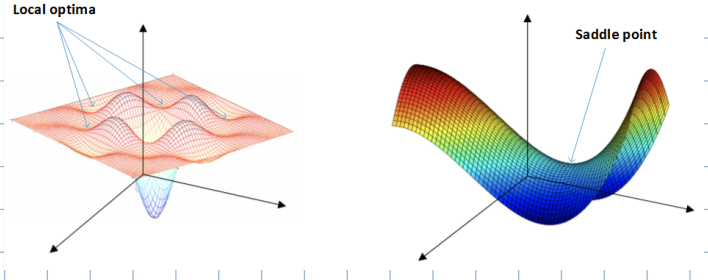
α_0: initial learning rate

DR: Decay Rate

EN: Epoch Number

$$, \quad \alpha = 0.95^{EN}\alpha_0 \quad , \quad \alpha = \frac{k}{\sqrt{EN}}\alpha_0 \quad , \quad \alpha = \frac{k}{\sqrt{t}}\alpha_0 \quad , \quad t: \text{mini-batch \#}$$

The Problem of Local OPtima



For a long time, people think the local optima appears quite often and will become a problem in the training phase if the input feature is high dimensional vector. But it turns out it's highly unlikely the case. In fact, it will most likely be a saddle point. Why?

Given a feature space with 20000 dimensions, each element will be either convex $\cup$ or concave $\cap$. If it's a local optima, then the probability is $\frac{1}{2^{20000}}$, which is almost impossible. Thus, it mostly will be a saddle point.