mini-batch gradient descent

- Good optimization algorithms are key in the deep learning era as the models require large datasets to be trained

- batch vs. mini-batch
  - vectorization allows us to efficiently compute on m training examples without a explicit for loop.
    - $X = [x^{(1)}\ x^{(2)}\ x^{(3)} \dots x^{(m)}]$
      - (n_x, m)
    - $Y = [y^{(1)}\ y^{(2)}\ y^{(3)} \dots y^{(m)}]$
      - $(1, m)$

* what if m was really large? Vectorization can still lead to really slow training (as we have to process the vectors each time we are going to take a step of gradient descent)
  - we can split our training set into smaller mini-batches. Instead of vectorizing 5 million examples, each mini-batch can have 1k examples
    - $X = [\underbrace{x^{(1)}\ x^{(2)} \dots x^{(1000)}}_{X^{\{1\}}\ (n_x, 1000)}\ |\ \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}\ (n_x, 1000)}\ |\ \dots\ |\ \underbrace{\dots x^{(m)}}_{X^{\{5000\}}}]$

1000 training examples

    - Similarly Y would be split into 5000 mini-batches (1, 1000) for each mini batch

* $x^{(i)}$: ith training example, $X^{[l]}$ for layer #, $X^{\{t\}}$ mini-batch #

- Mini-batch gradient descent

for $t = 1, \ldots, 5000$ {

* inside this for-loop we implement one step of gradient descent using $X^{\{t\}}$ and $Y^{\{t\}}$ (as if we were to train our network on a set of 1000 examples)

forward prop on $X^{\{t\}}$

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

vectorized implementation of 1000 examples

Compute cost $J = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$ (from mini-batch $X^{\{t\}}, Y^{\{t\}}$)

$+ \frac{\lambda}{2 \cdot 1000} \sum_{i=1}^{1000} \|w^{[l]}\|_F^2$ (Frobenius norm)

Backprop to compute gradient $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

$$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$$
$$b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$$

}

* this is called doing one pass through the training set using mini batch gradient descent

- one epoch of training (single pass through training set)

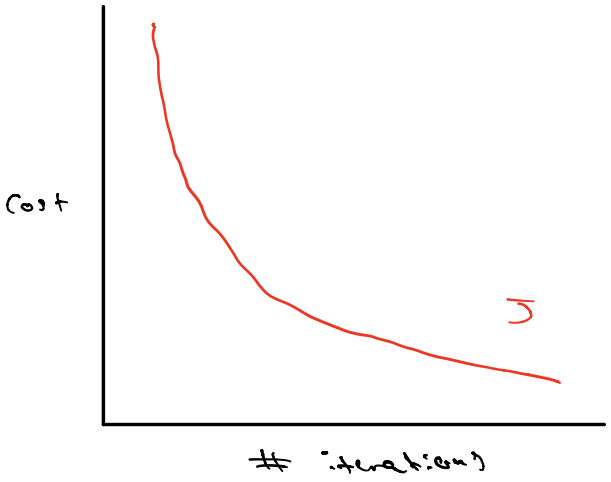
- with batch / pass means that we are taking one step of gradient descent, while / pass in mini batch means 5000 steps of gradient descent.

* we can repeat this # of times to do multiple passes on our training set.

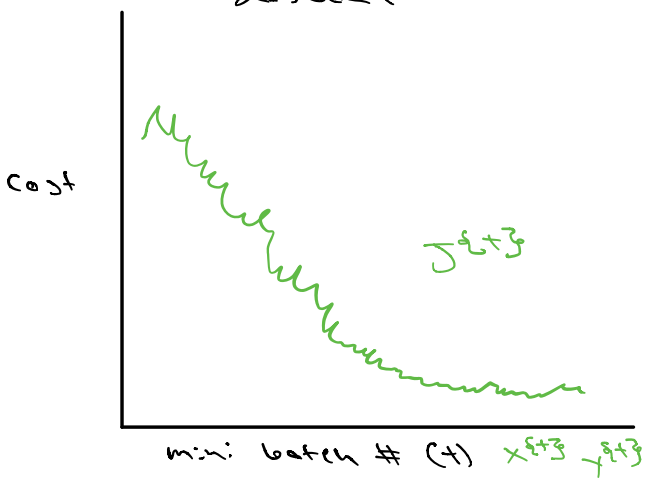
Understanding mini-batch gradient descent

- With mini-batch processing you start taking training steps before you finish processing the training set.

Batch Gradient descent



mini-batch gradient descent



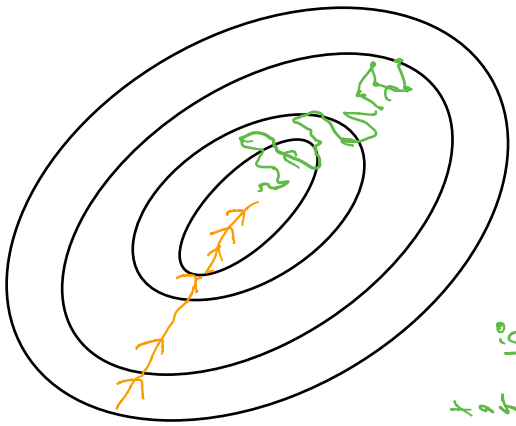
- with batch gradient descent, we'd expect the cost function to go down on every single iteration (if it goes up even in one iteration maybe that learning rate is too large)
- in mini-batch gd it is as if in every iteration we are training on a different training set, hence, we are more likely to get fluctuations with the general trend towards a lower cost.
  - the noisiness comes from the disparities between mini batches
- Choosing mini-batch size
  - if the mini-batch size = m: we would have batch gradient

descent $(x^{\{i:j\}}, y^{\{i:j\}}) = (x, y)$

↳ if mini_batch size = 1 : Stochastic gradient descent

↳ every example is its own mini batch

$(x^{\{i:j\}}, y^{\{i:j\}}) = (x^{(i)}, y^{(i)})$ ...



- Batch gradient descent will be able to take relatively low noise (large) steps
- Stochastic gradient descent is noisy, but will on average take you in a good direction (to minimize the cost function. Stochastic gradient never converges, it oscillates in the region near the minimum.

- In Practice, the mini-batch size used is somewhere in between 1 and m.

↳ Batch gradient descent (mini batch = m) makes us process a huge training set on every iteration taking too long for iteration (with a huge training set)

↳ With stochastic gradient descent we lose all the speed up from vectorization because we are processing each example at a time. This makes the way we process training examples inefficient.

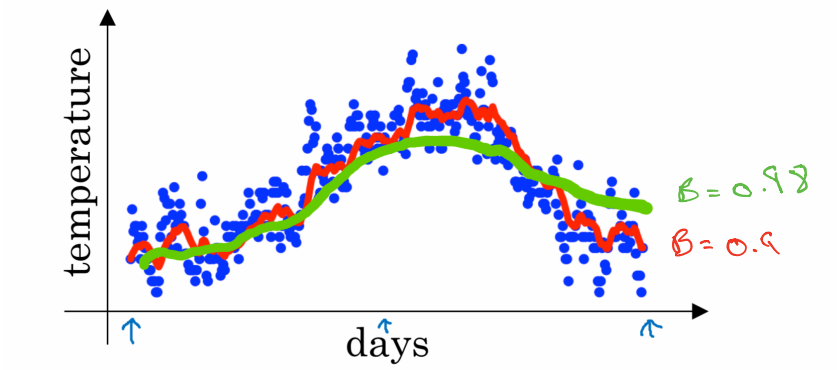
↳ with a medium size mini_batch we will get on average the fastest learning (through vectorization and by making progress without the extra time

required to process the whole train set)

- If we have a small train set: use batch gradient descent ($m \leq 2000$)
- typical mini-batch sizes: 64, 128, 256, 512
  - typically on the power of 2 because code sometimes runs faster.
  - we also need to make sure that the mini-batch fits in CPU/GPU memory

Exponentially Weighted Moving Averages

- way to denoise/filter data in order to get more smoothness in trends. $V_t = 0.9\, V_{t-1} + 0.1\, \theta_t$
  - V_{t-1}: value at previous day
  - θ_t: actual current day
- generalized: $V_t = \beta \cdot V_{t-1} + (1-\beta)\, \theta_t$
  - V_t can be thought as an averaging over $\frac{1}{1-\beta}$ past days
    - with the example above, we are averaging over the last 10 days (reducing the effect of short variations)
    - so we make β larger, the impact of any current value measurement is heavily outweighted by the previous measurements (and hence the curve smoothens)
- taking larger betas has the effect of shifting the curve further to the right as we are now averaging over larger windows (added latency to the formula)



Understanding Exponentially weighted Averages

- Implementing exponentially weighted averages

$v_\theta = 0$

$v_\theta := \beta \cdot v_\theta + (1-\beta)\theta_1$ } for loop over # of θs

$v_\theta := \beta \cdot v_\theta + (1-\beta)\theta_2$

$\vdots$

* This implementation takes little memory as only one value for v_θ is held at a time

This is a far more computationally efficient way to compute averages than storing values of the last n's in memory at a time

Bias Correction

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

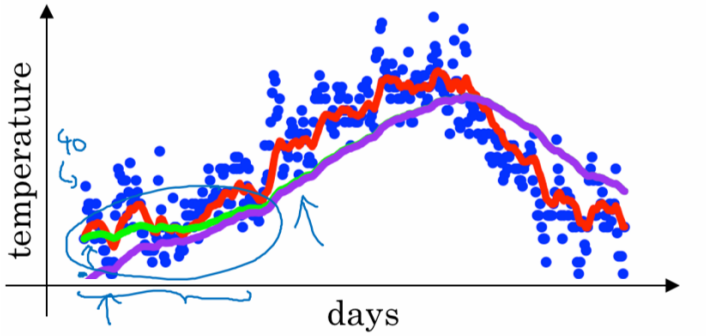
$V_0 = 0$

~~$V_1 = 0.98 V_0 + 0.02\,\theta_1$~~

$V_2 = 0.98\, V_1 + 0.02\,\theta_2$

$= 0.98 \times 0.02 \times \theta_1 + 0.02\,\theta_2$

$= 0.0196\,\theta_1 + 0.02\,\theta_2$



improve from purple line to the green line

- for low samples, the averaging starts flawed because there's not enough values accumulated in the V0 term
  - to fix this, we must employ bias correction.
    - $\frac{v_t}{1-\beta^t}$

denominator term

So for example for $(t=2, \beta=0.98) \rightarrow 1-(0.98)^2 = 0.0396$

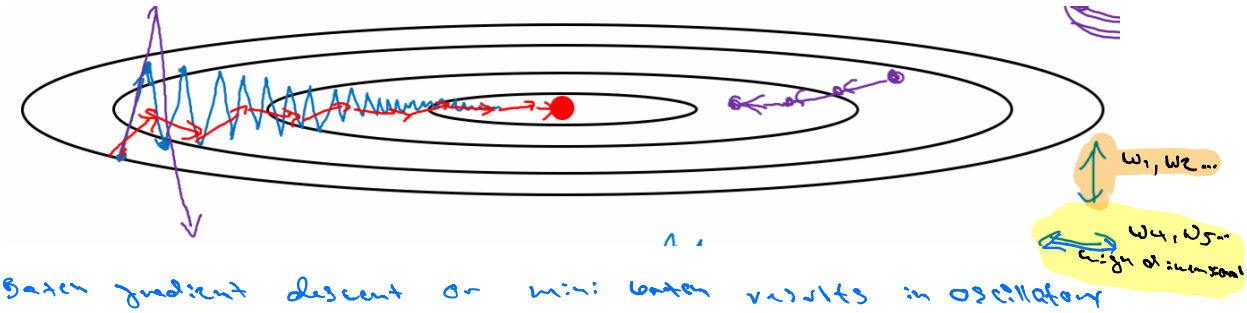
hence, $\frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$ numerator term = denominator term

* this operation takes t into account, removing the bias effect.
  - as t grows large, $\beta^t \approx 0$, hence $1-\beta^t \approx 1$

  having no role later on in the calculation.

Gradient descent with momentum

- almost always works faster than the standard gradient descent algorithm. Computing an exponentially weighted average of the gradients and the use that gradient to update the weights instead.



- Batch gradient descent or mini batch results in oscillatory behaviour as we move across the optimization contour

↳these up and down oscillations slows down gradient descent and inhibits us from using a larger learning rate.

↳(with a larger learning rate we might end up overshooting (so we are forced to use a learning rate which is not too large.

- we want faster learning on the horizontal direction of the contour and slower on the vertical.

- By using momentum, on each iteration (t):

  Compute dw, db on current mini-batch.

  $v_{dw} = \beta v_{dw} + (1-\beta) dw$ ← similar to previous moving average calculation

  $v_{db} = \beta \cdot v_{db} + (1-\beta) db$

  $w := w - \alpha \cdot v_{dw}$

  $b := b - \alpha \cdot v_{db}$

  ] this smooths out the steps of gradient descent by averaging the vertical oscillations. (oscillations are damped)

* Hence we end up traversing down the contour (optimization path) in a more effective way. (red on diagram)

* the most common β value is 0.9 (so that we average over the last 10 gradients)

* Bias correction is not really implemented as the derivative term is "warmed up" after a few iterations (hence not influencing the overarching performance).

RMSprop (Root mean squared prop)

On iteration t:

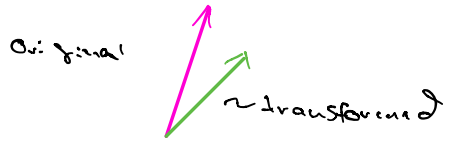
Compute dw, db on current mini-batch

$S_{dw} = \beta_2 \cdot S_{dw} + (1-\beta)\, dw^2$ ~ element wise squaring operation ~ small

$S_{db} = \beta_2 \cdot S_{db} + (1-\beta) \cdot db^2$ ~ large

$w := w - \alpha \cdot \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$ ~ we want to update the vert. by dividing a relatively small #

$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$ ~ we want to limit the update to b by dividing by a large #.



* if we go back to the contour diagram, we see that similar to momentum, the gradient doesn't oscillate as much and travels faster towards the minimum (as this methodology enables an increase in α).

* $+\varepsilon = 10^{-8}$ is added so that we never divide the derivatives by zero or extremely small #s (numerical stability)

Adam optimization

- Basically taking momentum and RMSprop and putting them together
- Programmatic implementation:

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

On iteration t:

Compute dw, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\, dw$ →

$$v_{dw} = \beta_1 v_{dw} + (1-\beta_1)\, dw$$
$$v_{db} = \beta_1 v_{db} + (1-\beta_1)\, db$$

} momentum implementation

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

} RMSprop update

* in the typical implementation of adam we add bias correction

$$v_{dw}^{corrected} = v_{dw} / (1-\beta_1^t)$$
$$v_{db}^{corrected} = v_{db} / (1-\beta_1^t)$$
$$s_{dw}^{corrected} = s_{dw} / (1-\beta_2^t)$$
$$s_{db}^{corrected} = s_{db} / (1-\beta_2^t)$$

update terms

$$w := w - \alpha \frac{v_{dw}^{corrected}}{\sqrt{s_{dw}^{corrected}} + \varepsilon}$$
$$b := b - \alpha \frac{v_{db}^{corrected}}{\sqrt{s_{db}^{corrected}} + \varepsilon}$$

- This algorithm combines the effect of momentum with gradient descent with RMSprop to create a effective learning algorithm which works on a wide variety of architectures.

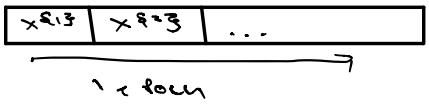
- hyperparameter choices
  - α: need to be tuned
  - β_1: 0.9 (dw)
  - β_2: 0.999 (dw^2)
  - ε: 10^{-8}

- adam: adaptive moment estimation

Learning rate decay

- Slowly reducing the learning rate over time helps speed up the learning algorithm (learning rate decay).
  - We oscillate in a tighter region near the minimum than with an otherwise large learning rate.

- applying learning rate decay:
  - 1 epoch = 1 pass through the data

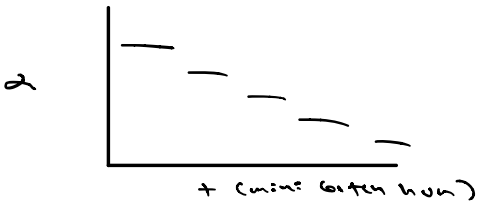


We want to set the learning rate to,

$$\alpha = \frac{1}{1 + \text{decay\_rate} * \text{epoch\_number}} * \alpha_0$$ ~ initial decay rate

* note that the decay_rate might be another hyperparameter that we would have to tune.

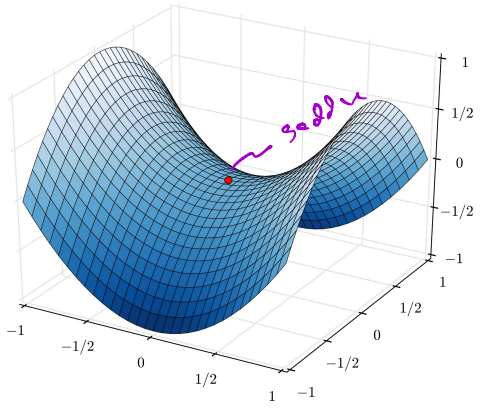
- Other learning rate decay methods
  - exponential decay
    - $\alpha = 0.95^{\text{epoch\_num}} * \alpha_0$
  - $\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} * \alpha_0$ ~ some #
  - discrete staircase



*learning rate decay can help training, but it's often a little bit down the list when it comes to things that we can do to improve the model
- try choosing an adequate learning rate first

The problem of local optima

- In the early days of DL there was a fear of algorithms getting trapped in local optima (this notion has changed)
- in high dimensional space, a local optima requires all features to have 0 gradient at one specific point (which is unlikely when we are operating in 20000 dimensions) (concave or convex shape)
  - we are much more likely to run into saddle points.



- the real problems arise from plateaus (regions where the derivative is close to zero, slowing down learning)
  - RMSprop, adam, momentum can all help in this type of scenario.