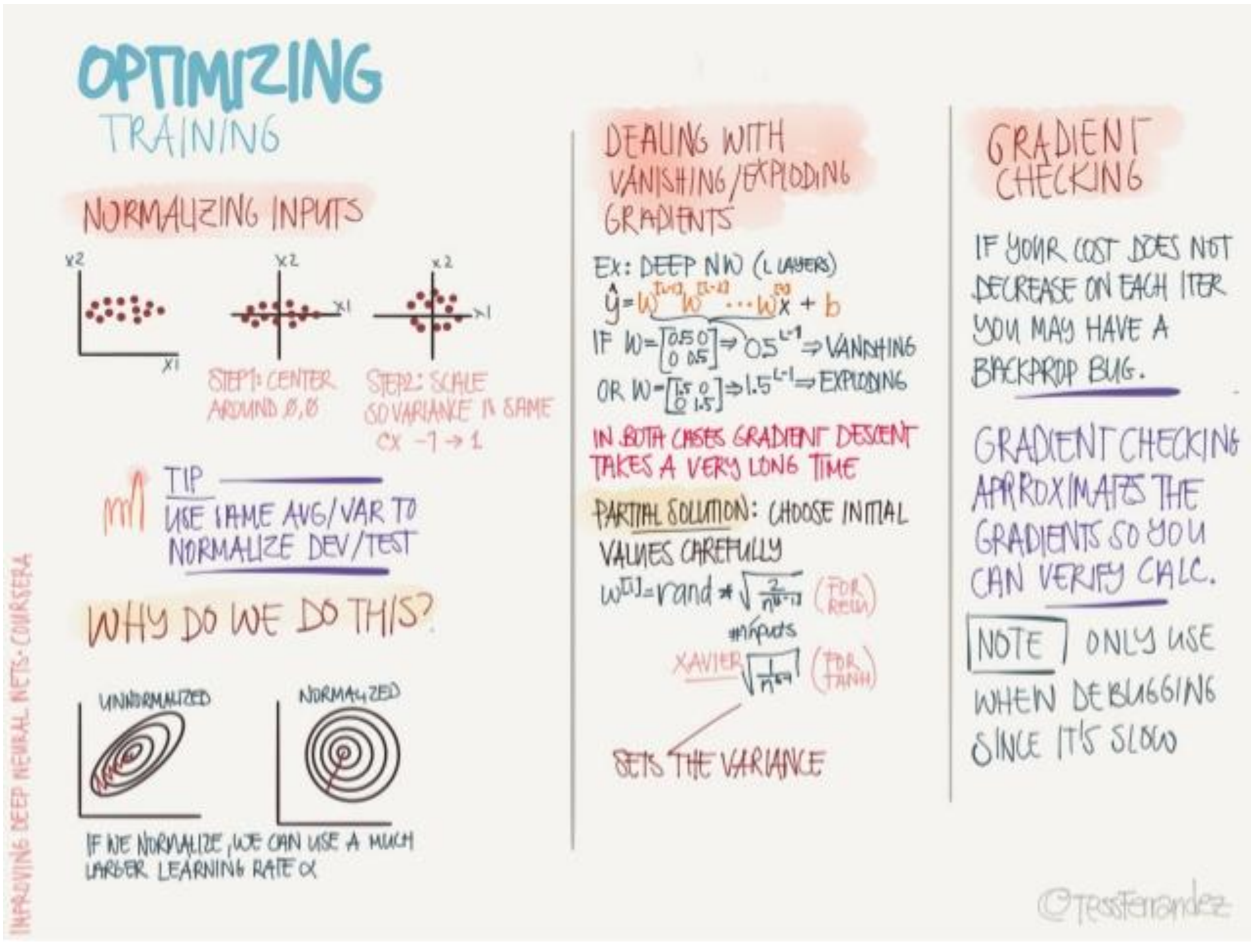
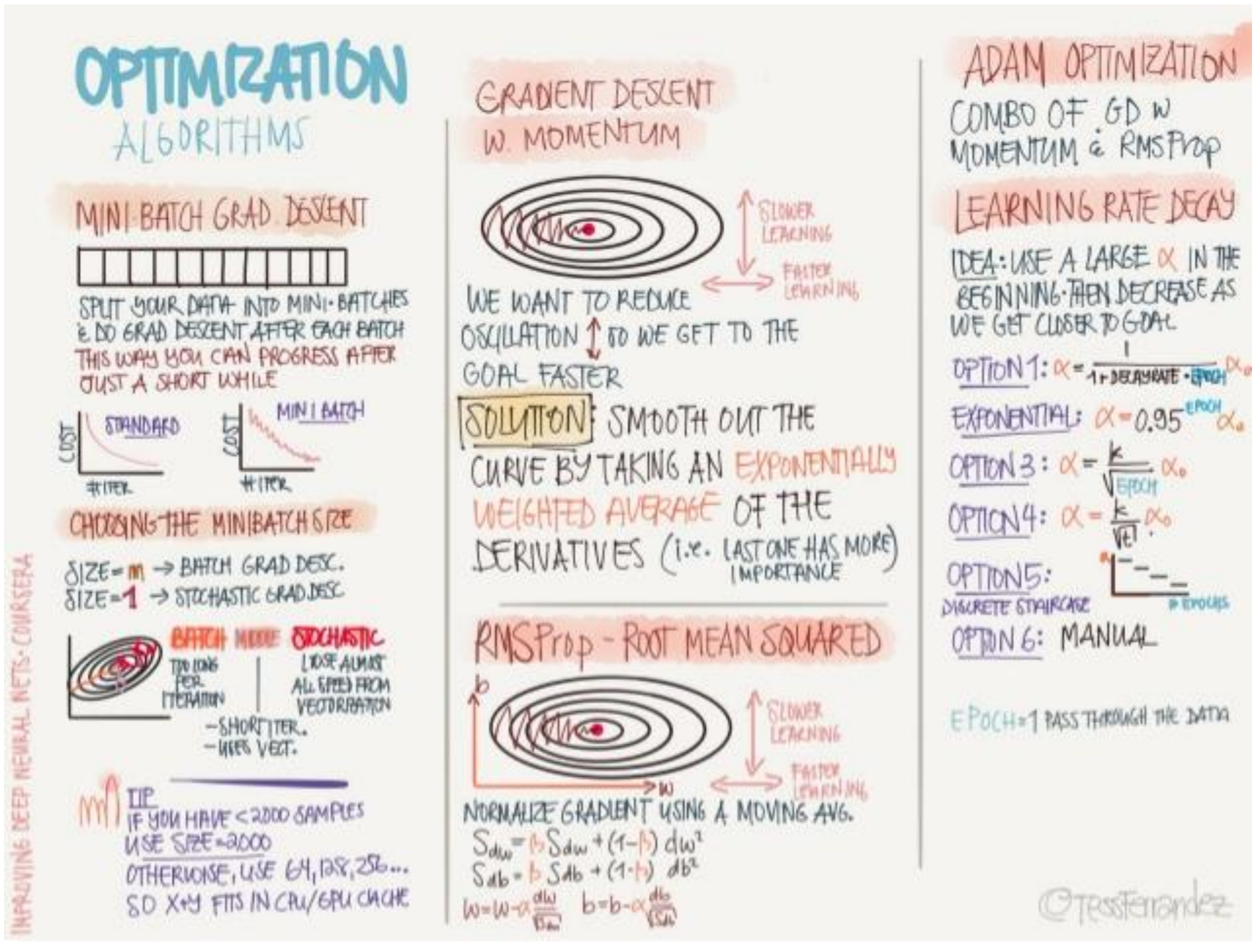
Optimization algorithms

- Mini-batch gradient descent
- Understanding mini-batch gradient descent
- Exponentially weighted averages
- Understanding exponentially weighted averages
- Bias correction in exponentially weighted averages
- Gradient descent with momentum
- RMSprop
- Adam optimization algorithm
- Learning rate decay
- The problem of local optima


OPTIMIZING TRAINING
NORMALIZING INPUTS
x2
x1
STEP1: CENTER AROUND 0,0
STEP2: SCALE SO VARIANCE IS SAME CX -1 → 1
TIP
USE SAME AVG/VAR TO NORMALIZE DEV/TEST
WHY DO WE DO THIS?
UNNORMALIZED
NORMALIZED
IF WE NORMALIZE, WE CAN USE A MUCH LARGER LEARNING RATE α
IMPROVING DEEP NEURAL NETS - COURSERA
DEALING WITH VANISHING/EXPLODING GRADIENTS
EX: DEEP NW (L LAYERS)
$\hat{y} = W^{[L]} W^{[L-1]} \cdots W^{[1]} x + b$
IF $W = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \Rightarrow 0.5^{L-1} \Rightarrow$ VANISHING
OR $W = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \Rightarrow 1.5^{L-1} \Rightarrow$ EXPLODING
IN BOTH CASES GRADIENT DESCENT TAKES A VERY LONG TIME
PARTIAL SOLUTION: CHOOSE INITIAL VALUES CAREFULLY
$W^{[l]} = \text{rand} * \sqrt{\frac{2}{n^{[l-1]}}}$ (FOR RELU)
#INPUTS
XAVIER $\sqrt{\frac{1}{n^{[l-1]}}}$ (FOR TANH)
SETS THE VARIANCE
GRADIENT CHECKING
IF YOUR COST DOES NOT DECREASE ON EACH ITER YOU MAY HAVE A BACKPROP BUG.
GRADIENT CHECKING APPROXIMATES THE GRADIENTS SO YOU CAN VERIFY CALC.
NOTE ONLY USE WHEN DEBUGGING SINCE IT'S SLOW
©TessFerrandez


OPTIMIZATION ALGORITHMS
MINI-BATCH GRAD DESCENT
SPLIT YOUR DATA INTO MINI-BATCHES & DO GRAD DESCENT AFTER EACH BATCH
THIS WAY YOU CAN PROGRESS AFTER JUST A SHORT WHILE
STANDARD
MINI BATCH
COST
# ITER
CHOOSING THE MINIBATCH SIZE
SIZE = m → BATCH GRAD DESC.
SIZE = 1 → STOCHASTIC GRAD DESC
BATCH MODE
STOCHASTIC
TOO LONG PER ITERATION
LOSE ALMOST ALL SPEED FROM VECTORIZATION
– SHORT ITER.
– USES VECT.
TIP
IF YOU HAVE < 2000 SAMPLES USE SIZE = 2000
OTHERWISE, USE 64, 128, 256...
SO X+Y FITS IN CPU/GPU CACHE
IMPROVING DEEP NEURAL NETS · COURSERA
GRADIENT DESCENT W. MOMENTUM
SLOWER LEARNING
FASTER LEARNING
WE WANT TO REDUCE OSCILLATION ↕ SO WE GET TO THE GOAL FASTER
SOLUTION: SMOOTH OUT THE CURVE BY TAKING AN EXPONENTIALLY WEIGHTED AVERAGE OF THE DERIVATIVES (i.e. LAST ONE HAS MORE IMPORTANCE)
RMSProp - ROOT MEAN SQUARED
b
w
SLOWER LEARNING
FASTER LEARNING
NORMALIZE GRADIENT USING A MOVING AVG.
$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$
$S_{db} = \beta S_{db} + (1-\beta) db^2$
$w = w - \alpha \frac{dw}{\sqrt{S_{dw}}}$ $b = b - \alpha \frac{db}{\sqrt{S_{db}}}$
ADAM OPTIMIZATION
COMBO OF GD W MOMENTUM & RMSProp
LEARNING RATE DECAY
IDEA: USE A LARGE α IN THE BEGINNING THEN DECREASE AS WE GET CLOSER TO GOAL
OPTION 1: $\alpha = \frac{1}{1 + \text{DECAYRATE} \cdot \text{EPOCH}} \alpha_0$
EXPONENTIAL: $\alpha = 0.95^{\text{EPOCH}} \alpha_0$
OPTION 3: $\alpha = \frac{k}{\sqrt{\text{EPOCH}}} \alpha_0$
OPTION 4: $\alpha = \frac{k}{\sqrt{t}} \alpha_0$
OPTION 5: DISCRETE STAIRCASE
α
# EPOCHS
OPTION 6: MANUAL
EPOCH = 1 PASS THROUGH THE DATA
@TessFerrandez

Optimization algorithms

Mini-batch gradient descent

- Training NN with a large data is slow. So to find an optimization algorithm that runs faster is a good idea.
- Suppose we have `m = 50 million`. To train this data it will take a huge processing time for one step.
  - because 50 million won't fit in the memory at once we need other processing to make such a thing.
- It turns out you can make a faster algorithm to make gradient descent process some of your items even before you finish the 50 million items.
- Suppose we have split m to **mini batches** of size 1000.
  - `X{1} = 0 ... 1000`
  - `X{2} = 1001 ... 2000`
  - `...`
  - `X{bs} = ...`
- We similarly split `X` & `Y`.

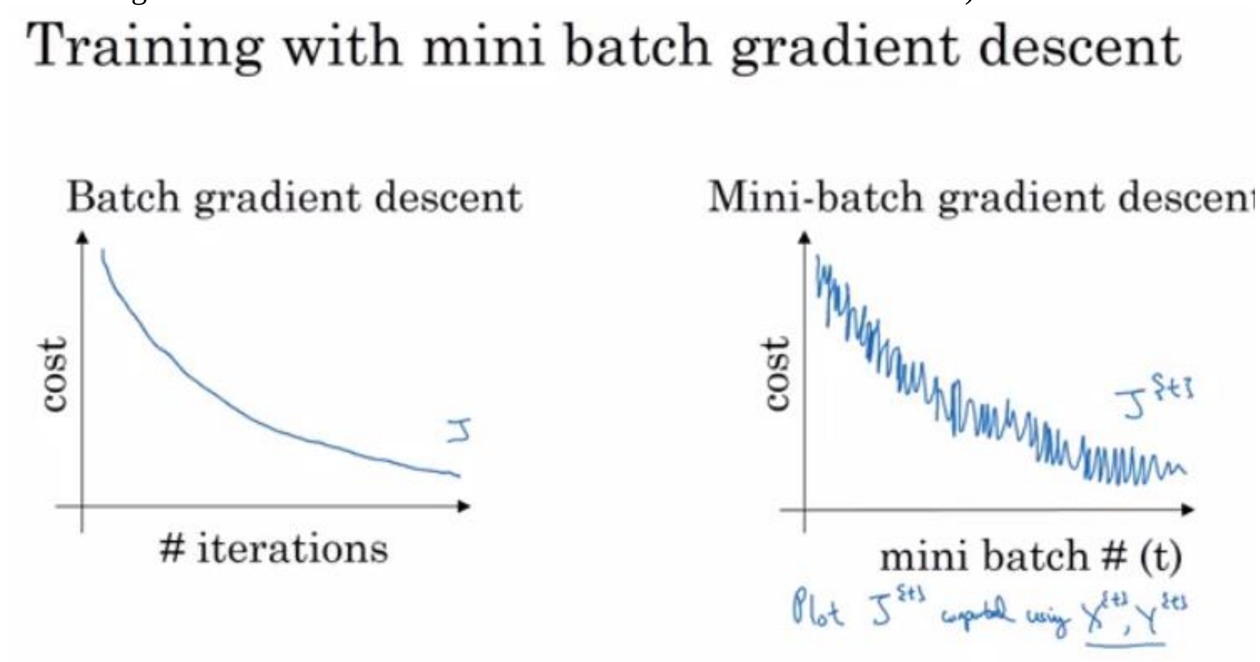
- So the definition of mini batches ==> `t: X{t}, Y{t}`
- In **Batch gradient descent** we run the gradient descent on the whole dataset.
- While in **Mini-Batch gradient descent** we run the gradient descent on the mini datasets.
- Mini-Batch algorithm pseudo code:

```
for t = 1:No_of_batches                         # this is called an epoch
  AL, caches = forward_prop(X{t}, Y{t})
  cost = compute_cost(AL, Y{t})
  grads = backward_prop(AL, caches)
  update_parameters(grads)
```

- The code inside an epoch should be vectorized.
- Mini-batch gradient descent works much faster in the large datasets.

Understanding mini-batch gradient descent

- In mini-batch algorithm, the cost won't go down with each step as it does in batch algorithm. It could contain some ups and downs but generally it has to go down (unlike the batch gradient descent where cost function descreases on each iteration).



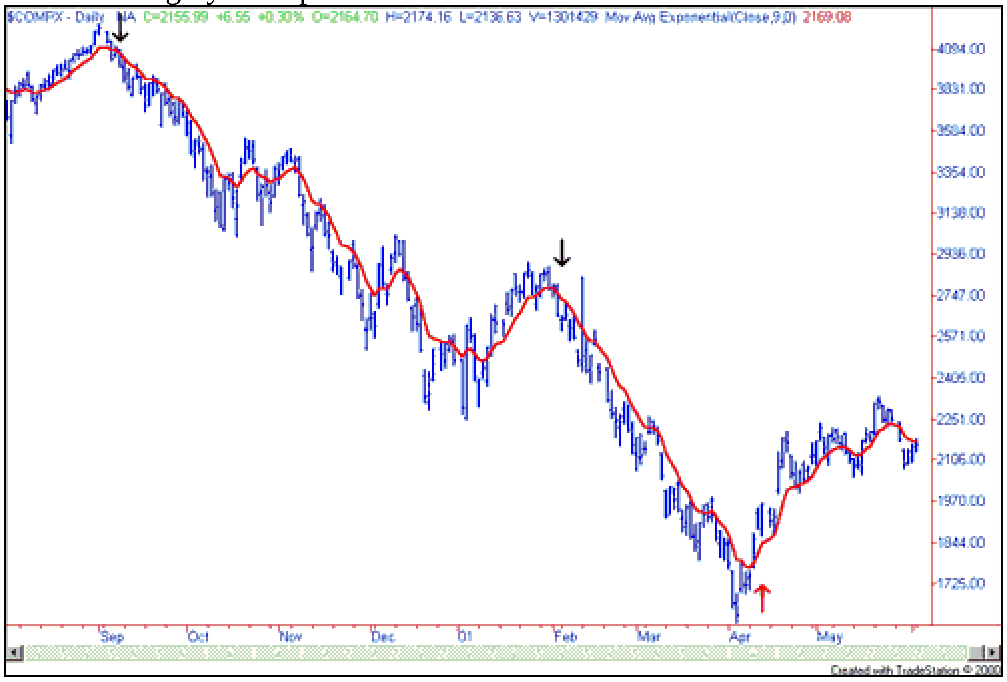
- Mini-batch size:
  - (`mini batch size = m`) ==> Batch gradient descent
  - (`mini batch size = 1`) ==> Stochastic gradient descent (SGD)
  - (`mini batch size = between 1 and m`) ==> Mini-batch gradient descent
- Batch gradient descent:
  - too long per iteration (epoch)
- Stochastic gradient descent:
  - too noisy regarding cost minimization (can be reduced by using smaller learning rate)
  - won't ever converge (reach the minimum cost)
  - lose speedup from vectorization

- Mini-batch gradient descent:
    1. faster learning:
        - you have the vectorization advantage
        - make progress without waiting to process the entire training set
    2. doesn't always exactly converge (oscelates in a very small region, but you can reduce learning rate)
- Guidelines for choosing mini-batch size:
    0. If small training set (< 2000 examples) - use batch gradient descent.
    1. It has to be a power of 2 (because of the way computer memory is layed out and accessed, sometimes your code runs faster if your mini-batch size is a power of 2): `64, 128, 256, 512, 1024, ...`
    2. Make sure that mini-batch fits in CPU/GPU memory.
- Mini-batch size is a `hyperparameter`.

Exponentially weighted averages

- There are optimization algorithms that are better than **gradient descent**, but you should first learn about Exponentially weighted averages.
- If we have data like the temperature of day through the year it could be like this:
- `t(1) = 40`
- `t(2) = 49`
- `t(3) = 45`
- `...`
- `t(180) = 60`
- `...`
- This data is small in winter and big in summer. If we plot this data we will find it some noisy.
- Now lets compute the Exponentially weighted averages:
- `V0 = 0`
- `V1 = 0.9 * V0 + 0.1 * t(1) = 4          # 0.9 and 0.1 are hyperparameters`
- `V2 = 0.9 * V1 + 0.1 * t(2) = 8.5`
- `V3 = 0.9 * V2 + 0.1 * t(3) = 12.15`
- `...`
- General equation
- `V(t) = beta * v(t-1) + (1-beta) * theta(t)`
- If we plot this it will represent averages over `~ (1 / (1 - beta))` entries:
    - `beta = 0.9` will average last 10 entries
    - `beta = 0.98` will average last 50 entries
    - `beta = 0.5` will average last 2 entries
- Best beta average for our case is between 0.9 and 0.98

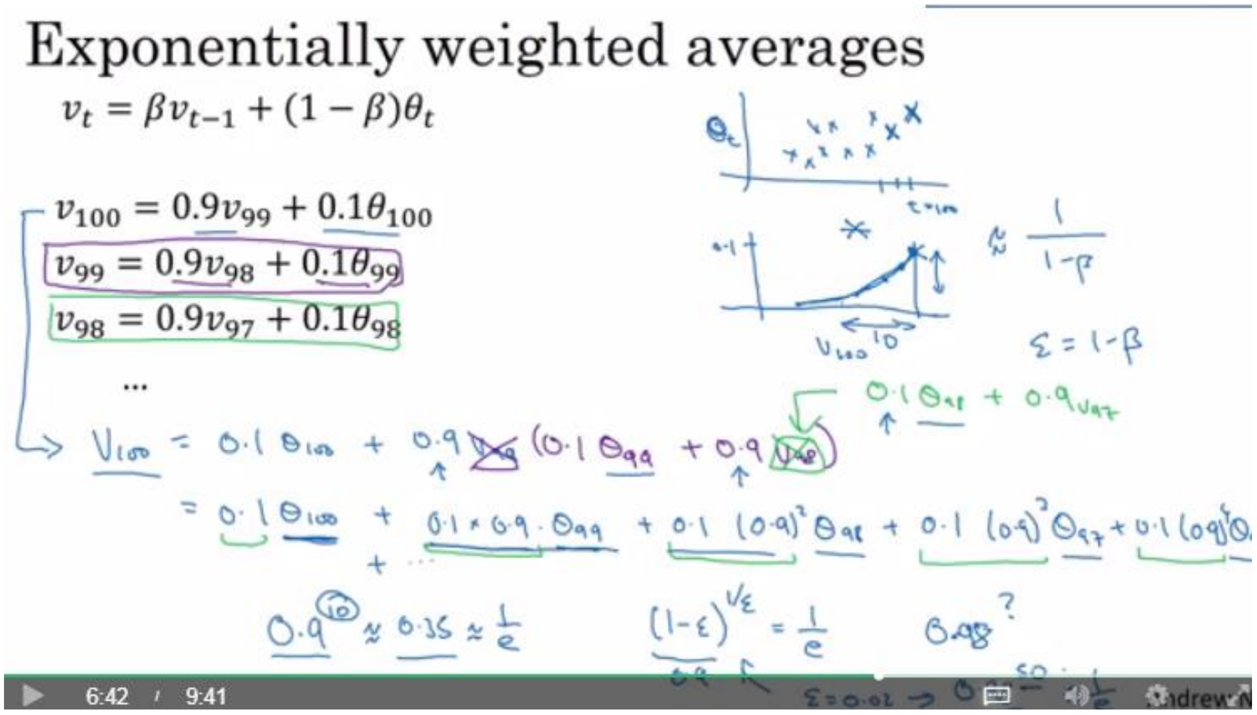
- Another imagery example:



(*taken from* [*investopedia.com*](investopedia.com))

Understanding exponentially weighted averages

- Intuitions:



Exponentially weighted averages

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

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

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

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

...

$$V_{100} = 0.1\,\theta_{100} + 0.9\,\cancel{v_{99}}\,(0.1\,\theta_{99} + 0.9\,\cancel{v_{98}})$$

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

$$= \underbrace{0.1}\,\theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1\,(0.9)^2\,\theta_{98} + 0.1\,(0.9)^3\,\theta_{97} + 0.1\,(0.9)^4\,\theta_{96} + \dots$$

$$0.9^{10} \approx 0.35 \approx \frac{1}{e} \qquad (1-\varepsilon)^{1/\varepsilon} = \frac{1}{e} \qquad 0.98\,?$$

$$\varepsilon = 0.02 \rightarrow$$

$$\approx \frac{1}{1-\beta} \qquad \varepsilon = 1-\beta$$

- We can implement this algorithm with more accurate results using a moving window. But the code is more efficient and faster using the exponentially weighted averages algorithm.
- Algorithm is very simple:

- `v = 0`
- `Repeat`
- `{`
- `  Get theta(t)`
- `  v = beta * v + (1-beta) * theta(t)`
- `}`

Bias correction in exponentially weighted averages

- The bias correction helps make the exponentially weighted averages more accurate.
- Because `v(0) = 0`, the bias of the weighted averages is shifted and the accuracy suffers at the start.
- To solve the bias issue we have to use this equation:
- `v(t) = (beta * v(t-1) + (1-beta) * theta(t)) / (1 - beta^t)`
- As t becomes larger the `(1 - beta^t)` becomes close to `1`

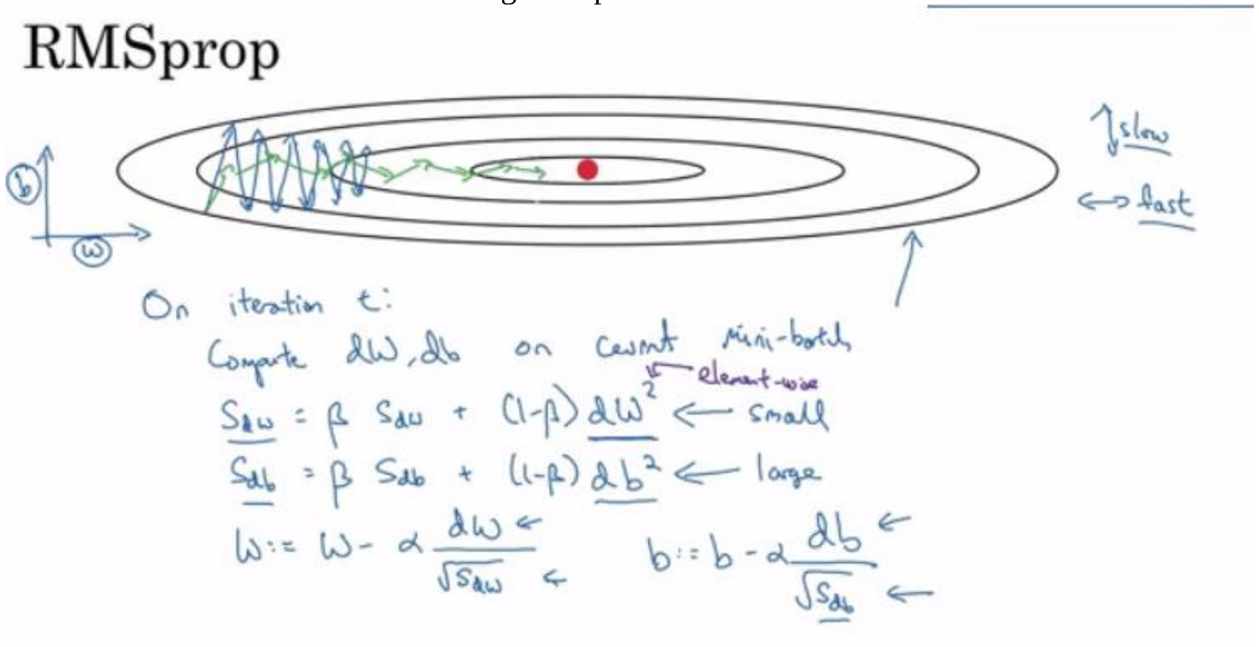
Gradient descent with momentum

- The momentum algorithm almost always works faster than standard gradient descent.
- The simple idea is to calculate the exponentially weighted averages for your gradients and then update your weights with the new values.
- Pseudo code:
- `vdW = 0, vdb = 0`
- `on iteration t:`
- `  # can be mini-batch or batch gradient descent`
- `  compute dw, db on current mini-batch`
-
- `  vdW = beta * vdW + (1 - beta) * dW`
- `  vdb = beta * vdb + (1 - beta) * db`
- `  W = W - learning_rate * vdW`
- `  b = b - learning_rate * vdb`
- Momentum helps the cost function to go to the minimum point in a more fast and consistent way.
- `beta` is another `hyperparameter`. `beta = 0.9` is very common and works very well in most cases.
- In practice people don't bother implementing **bias correction**.

RMSprop

- Stands for **Root mean square prop**.
- This algorithm speeds up the gradient descent.
- Pseudo code:
- `sdW = 0, sdb = 0`
- `on iteration t:`
- `  # can be mini-batch or batch gradient descent`
- `  compute dw, db on current mini-batch`
-

- `  sdW = (beta * sdW) + (1 - beta) * dW^2  # squaring is element-wise`
- `  sdb = (beta * sdb) + (1 - beta) * db^2  # squaring is element-wise`
- `  W = W - learning_rate * dW / sqrt(sdW)`
- `  b = B - learning_rate * db / sqrt(sdb)`
- RMSprop will make the cost function move slower on the vertical direction and faster on the horizontal direction in the following example:


RMSprop
b
W
slow
fast
On iteration t:
Compute dW, db on current mini-batch
element-wise
$S_{dW} = \beta S_{dW} + (1-\beta) dW^2 \leftarrow$ small
$S_{db} = \beta S_{db} + (1-\beta) db^2 \leftarrow$ large
$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}$
$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$

- Ensure that `sdW` is not zero by adding a small value `epsilon` (e.g. `epsilon = 10^-8`) to it:
  `W = W - learning_rate * dW / (sqrt(sdW) + epsilon)`
- With RMSprop you can increase your learning rate.
- Developed by Geoffrey Hinton and firstly introduced on Coursera.org course.

Adam optimization algorithm

- Stands for **Adaptive Moment Estimation**.
- Adam optimization and RMSprop are among the optimization algorithms that worked very well with a lot of NN architectures.
- Adam optimization simply puts RMSprop and momentum together!
- Pseudo code:
- `vdW = 0, vdW = 0`
- `sdW = 0, sdb = 0`
- `on iteration t:`
- `  # can be mini-batch or batch gradient descent`
- `  compute dw, db on current mini-batch`
-
- `  vdW = (beta1 * vdW) + (1 - beta1) * dW     # momentum`
- `  vdb = (beta1 * vdb) + (1 - beta1) * db     # momentum`
-
- `  sdW = (beta2 * sdW) + (1 - beta2) * dW^2   # RMSprop`
- `  sdb = (beta2 * sdb) + (1 - beta2) * db^2   # RMSprop`

-
- `vdW = vdW / (1 - beta1^t)      # fixing bias`
- `vdb = vdb / (1 - beta1^t)      # fixing bias`
-
- `sdW = sdW / (1 - beta2^t)      # fixing bias`
- `sdb = sdb / (1 - beta2^t)      # fixing bias`
-
- `W = W - learning_rate * vdW / (sqrt(sdW) + epsilon)`
- `b = B - learning_rate * vdb / (sqrt(sdb) + epsilon)`
- Hyperparameters for Adam:
  - Learning rate: needed to be tuned.
  - `beta1`: parameter of the momentum - `0.9` is recommended by default.
  - `beta2`: parameter of the RMSprop - `0.999` is recommended by default.
  - `epsilon`: `10^-8` is recommended by default.

Learning rate decay

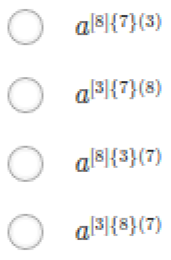
- Slowly reduce learning rate.
- As mentioned before mini-batch gradient descent won't reach the optimum point (converge). But by making the learning rate decay with iterations it will be much closer to it because the steps (and possible oscillations) near the optimum are smaller.
- One technique equations is`learning_rate = (1 / (1 + decay_rate * epoch_num)) * learning_rate_0`
  - `epoch_num` is over all data (not a single mini-batch).
- Other learning rate decay methods (continuous):
  - `learning_rate = (0.95 ^ epoch_num) * learning_rate_0`
  - `learning_rate = (k / sqrt(epoch_num)) * learning_rate_0`
- Some people perform learning rate decay discretely - repeatedly decrease after some number of epochs.
- Some people are making changes to the learning rate manually.
- `decay_rate` is another `hyperparameter`.
- For Andrew Ng, learning rate decay has less priority.

The problem of local optima

- The normal local optima is not likely to appear in a deep neural network because data is usually high dimensional. For point to be a local optima it has to be a local optima for each of the dimensions which is highly unlikely.
- It's unlikely to get stuck in a bad local optima in high dimensions, it is much more likely to get to the saddle point rather to the local optima, which is not a problem.
- Plateaus can make learning slow:
  - Plateau is a region where the derivative is close to zero for a long time.
  - This is where algorithms like momentum, RMSprop or Adam can help.

Week 2 Quiz - Optimization algorithms

1. Which notation would you use to denote the 3rd layer's activations when the input is the 7th example from the 8th minibatch?



- $a^{[8]\{7\}(3)}$
- $a^{[3]\{7\}(8)}$
- $a^{[8]\{3\}(7)}$
- $a^{[3]\{8\}(7)}$

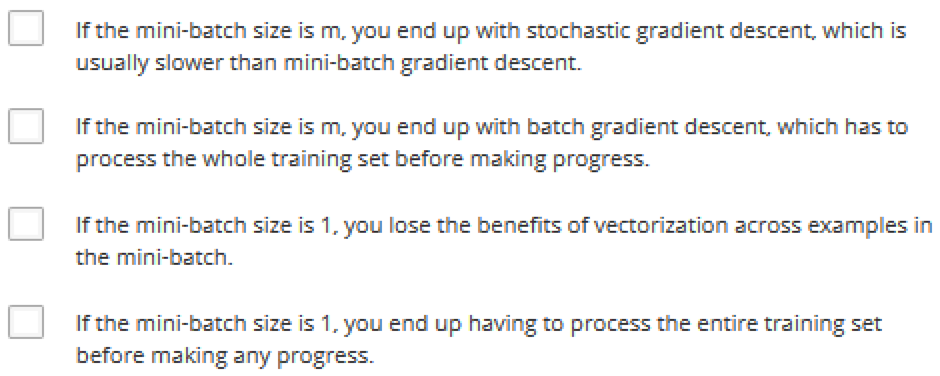
   - a^[3]{8}(7)

   Note: **[i]{j}(k)** superscript means **i-th layer**, **j-th minibatch**, **k-th example**

2. Which of these statements about mini-batch gradient descent do you agree with?
   - ☐ You should implement mini-batch gradient descent without an explicit for-loop over different mini-batches, so that the algorithm processes all mini-batches at the same time (vectorization).
   - ☐ Training one epoch (one pass through the training set) using mini-batch gradient descent is faster than training one epoch using batch gradient descent.
   - ✓ One iteration of mini-batch gradient descent (computing on a single mini-batch) is faster than one iteration of batch gradient descent.

   Note: Vectorization is not for computing several mini-batches in the same time.

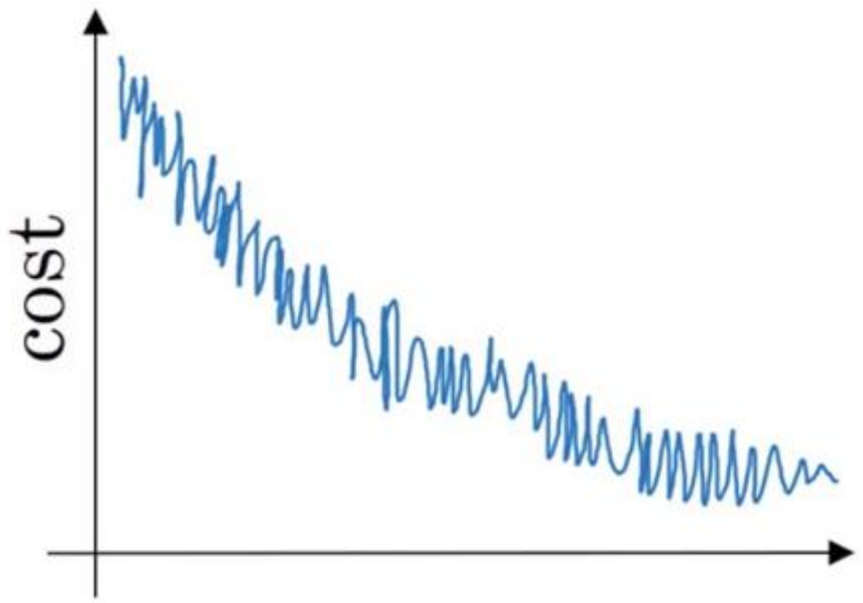
3. Why is the best mini-batch size usually not 1 and not m, but instead something in-between?



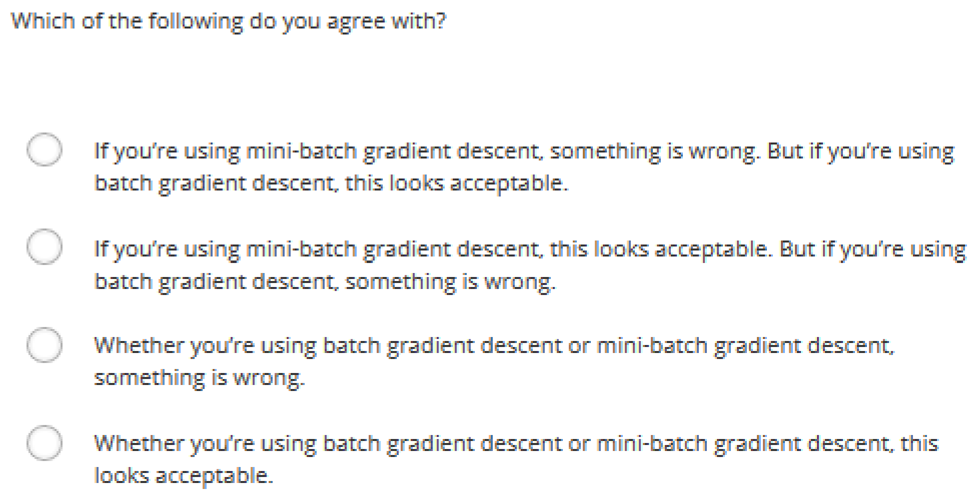
- ☐ If the mini-batch size is m, you end up with stochastic gradient descent, which is usually slower than mini-batch gradient descent.
- ☐ If the mini-batch size is m, you end up with batch gradient descent, which has to process the whole training set before making progress.
- ☐ If the mini-batch size is 1, you lose the benefits of vectorization across examples in the mini-batch.
- ☐ If the mini-batch size is 1, you end up having to process the entire training set before making any progress.

- If the mini-batch size is 1, you lose the benefits of vectorization across examples in the mini-batch.
- If the mini-batch size is m, you end up with batch gradient descent, which has to process the whole training set before making progress.

4. Suppose your learning algorithm's cost ***J***, plotted as a function of the number of iterations, looks like this:



Which of the following do you agree with?



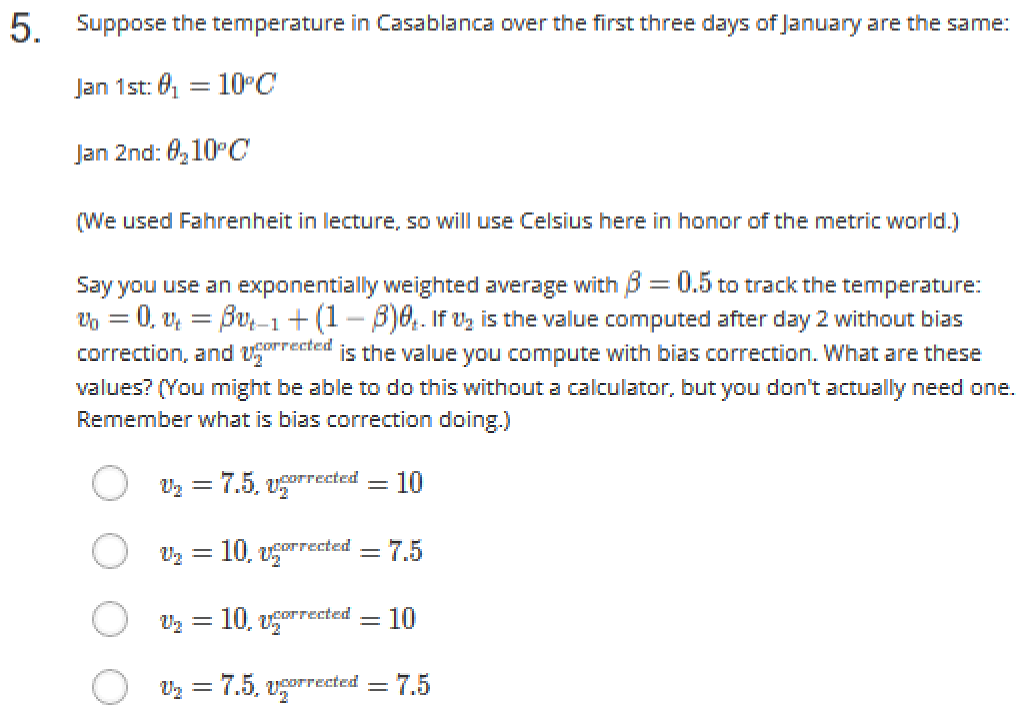
Which of the following do you agree with?

- ○ If you're using mini-batch gradient descent, something is wrong. But if you're using batch gradient descent, this looks acceptable.
- ○ If you're using mini-batch gradient descent, this looks acceptable. But if you're using batch gradient descent, something is wrong.
- ○ Whether you're using batch gradient descent or mini-batch gradient descent, something is wrong.
- ○ Whether you're using batch gradient descent or mini-batch gradient descent, this looks acceptable.

- If you're using mini-batch gradient descent, this looks acceptable. But if you're using batch gradient descent, something is wrong.

Note: There will be some oscillations when you're using mini-batch gradient descent since there could be some noisy data example in batches. However batch gradient descent always guarantees a lower ***J*** before reaching the optimal.

5. Suppose the temperature in Casablanca over the first three days of January are the same:



5. Suppose the temperature in Casablanca over the first three days of January are the same:

Jan 1st: $\theta_1 = 10^o C$

Jan 2nd: $\theta_2 10^o C$

(We used Fahrenheit in lecture, so will use Celsius here in honor of the metric world.)

Say you use an exponentially weighted average with $\beta = 0.5$ to track the temperature: $v_0 = 0$, $v_t = \beta v_{t-1} + (1 - \beta)\theta_t$. If v_2 is the value computed after day 2 without bias correction, and $v_2^{corrected}$ is the value you compute with bias correction. What are these values? (You might be able to do this without a calculator, but you don't actually need one. Remember what is bias correction doing.)

- $v_2 = 7.5$, $v_2^{corrected} = 10$
- $v_2 = 10$, $v_2^{corrected} = 7.5$
- $v_2 = 10$, $v_2^{corrected} = 10$
- $v_2 = 7.5$, $v_2^{corrected} = 7.5$

Jan 1st: θ_1 = 10

Jan 2nd: θ_2 * 10

Say you use an exponentially weighted average with β = 0.5 to track the temperature: v_0 = 0, v_t = βv_t−1 + (1 − β)θ_t. If v_2 is the value computed after day 2 without bias correction, and v^corrected_2 is the value you compute with bias correction. What are these values?

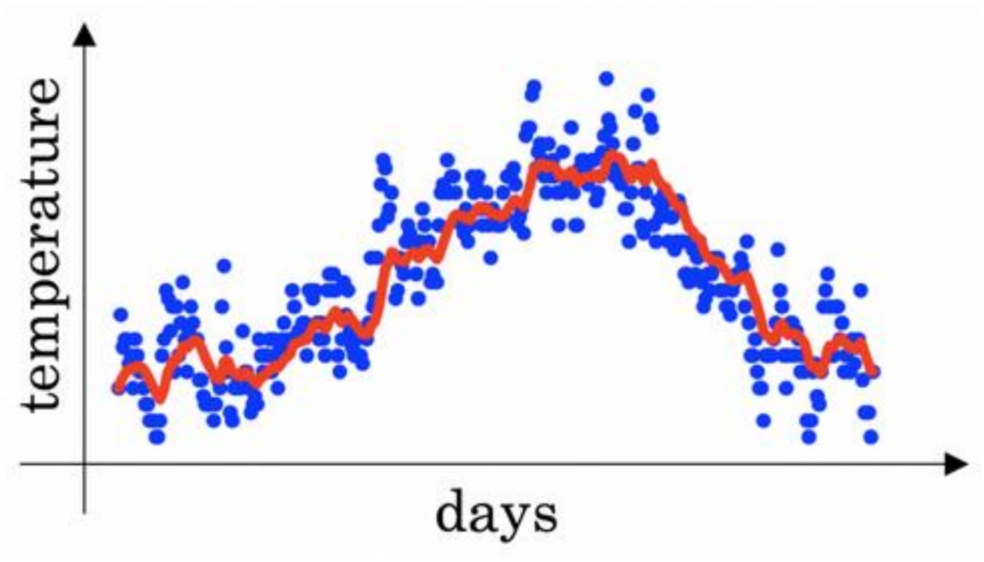
   - v_2 = 7.5, v^corrected_2 = 10

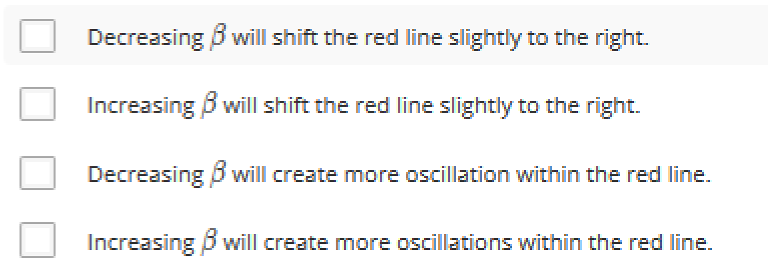
6. Which of these is NOT a good learning rate decay scheme? Here, t is the epoch number.
   - α = e^t * α_0

   Note: This will explode the learning rate rather than decay it.

7. You use an exponentially weighted average on the London temperature dataset. You use the following to track the temperature: v_t = βv_t−1 + (1 − β)θ_t. The red line below was

computed using β = 0.9. What would happen to your red curve as you vary β? (Check the two that apply)

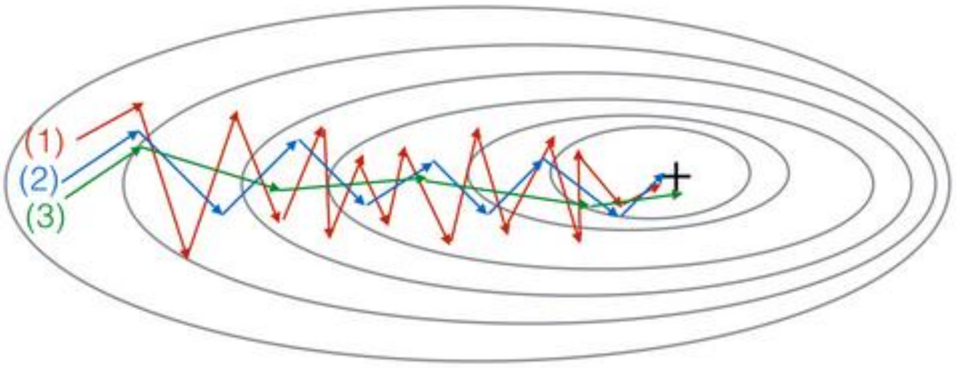




- [ ] Decreasing β will shift the red line slightly to the right.
- [ ] Increasing β will shift the red line slightly to the right.
- [ ] Decreasing β will create more oscillation within the red line.
- [ ] Increasing β will create more oscillations within the red line.

   - Increasing β will shift the red line slightly to the right.
   - Decreasing β will create more oscillation within the red line.

8. Consider this figure:



These plots were generated with gradient descent; with gradient descent with momentum (β = 0.5) and gradient descent with momentum (β = 0.9). Which curve corresponds to which algorithm?

(1) is gradient descent. (2) is gradient descent with momentum (small β). (3) is gradient descent with momentum (large β)

9. Suppose batch gradient descent in a deep network is taking excessively long to find a value of the parameters that achieves a small value for the cost function J(W[1],b[1],...,W[L],b[L]). Which of the following techniques could help find parameter values that attain a small value forJ? (Check all that apply)
   - ✓ Try using Adam
   - ✓ Try better random initialization for the weights
   - ✓ Try tuning the learning rate α
   - ✓ Try mini-batch gradient descent
   - ☐ Try initializing all the weights to zero
10. Which of the following statements about Adam is False?
   - Adam should be used with batch gradient computations, not with mini-batches.

   Note: Adam could be used with both.