Training Neural Networks Optimization (update rule)

M. Soleymani
Sharif University of Technology
Fall 2017

Some slides have been adapted from Fei Fei Li and colleagues lectures, cs231n, Stanford 2017, some from Andrew Ng lectures, “Deep Learning Specialization”, coursera, 2017, and few from Hinton lectures, “Neural Networks for Machine Learning”, coursera, 2015.

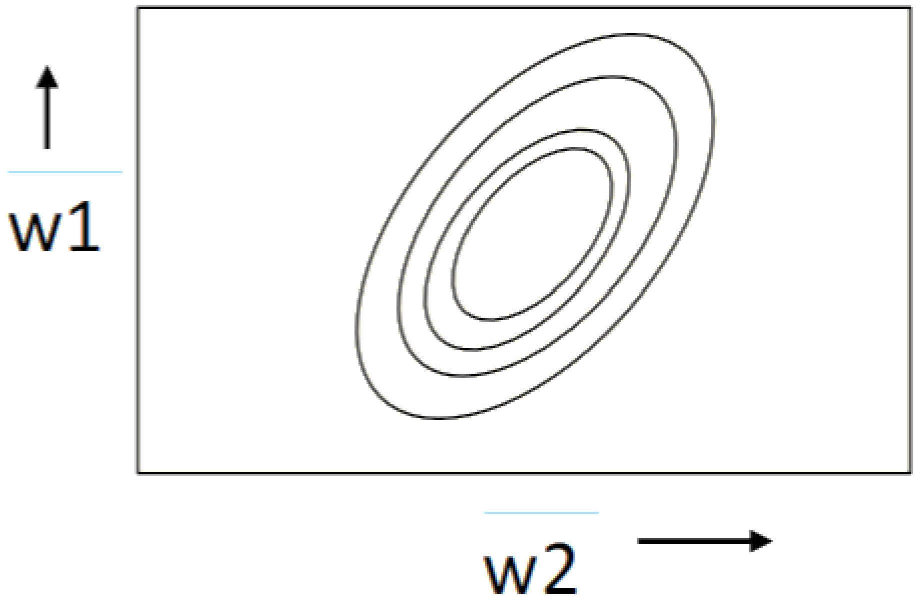
Outline

- Mini-batch gradient descent
- Gradient descent with momentum
- RMSprop
- Adam
- Learning rate and learning rate decay

- Batch normalization

Reminder: The error surface

- For a linear neuron with a squared error, it is a quadratic bowl.
  - Vertical cross-sections are parabolas.
  - Horizontal cross-sections are ellipses.
- For multi-layer nets the error surface is much more complicated.
  - But locally, a piece of a quadratic bowl is usually a very good approximation.



Mini-batch gradient descent

- Large datasets
  - Divide dataset into smaller batches containing one subset of the main training set
  - Weights are updated after seeing training data in each of these batches

- Vectorization provides efficiency

Gradient descent methods

Stochastic gradient

Batch size=1

Stochastic mini-batch gradient

e.g., Batch size= 32, 64, 128, 256

Batch gradient

Batch size=n
(the size of training set)

n: whole no of training data
bs: the size of batches
$m = \left\lceil \frac{n}{bs} \right\rceil$: the number of batches

Batch 1 $X^{\{1\}}, Y^{\{1\}}$	Batch 2 $X^{\{2\}}, Y^{\{2\}}$								Batch m $X^{\{m\}}, Y^{\{m\}}$

Mini-batch gradient descent

1 epoch: Single pass over all training samples

```
For epoch=1,…,k
    For t=1,…,m
        Forward propagation on $X^{\{t\}}$
        $J^{\{t\}} = \frac{1}{m}\sum_{n\in Batch_t} L\left(\hat{Y}_n^{\{t\}}, Y_n^{\{t\}}\right) + \lambda R(W)$
        Backpropagation on $J^{\{t\}}$ to compute gradients $dW$
        For $l = 1, \dots, L$
            $W^{[l]} = W^{[l]} - \alpha dW^{[l]}$
```

```
$A^{[0]} = X^{\{t\}}$
For $l = 1, \dots, L$
    $Z^{[l]} = W^{[l]}A^{[l-1]}$
    $A^{[l]} = f^{[l]}(Z^{[l]})$
$\hat{Y}_n^{\{t\}} = A_n^{[L]}$
```

Vectorized computaion

Batch 1 $X^{\{1\}}, Y^{\{1\}}$	Batch 2 $X^{\{2\}}, Y^{\{2\}}$								Batch m $X^{\{m\}}, Y^{\{m\}}$

Gradient descent methods

Stochastic gradient descent

Batch size=1

- Does not use vectorized form and thus not computationally efficient

Stochastic mini-batch gradient

e.g., Batch size= 32, 64, 128, 256

- Vectorization
- Fastest learning (for proper batch size)

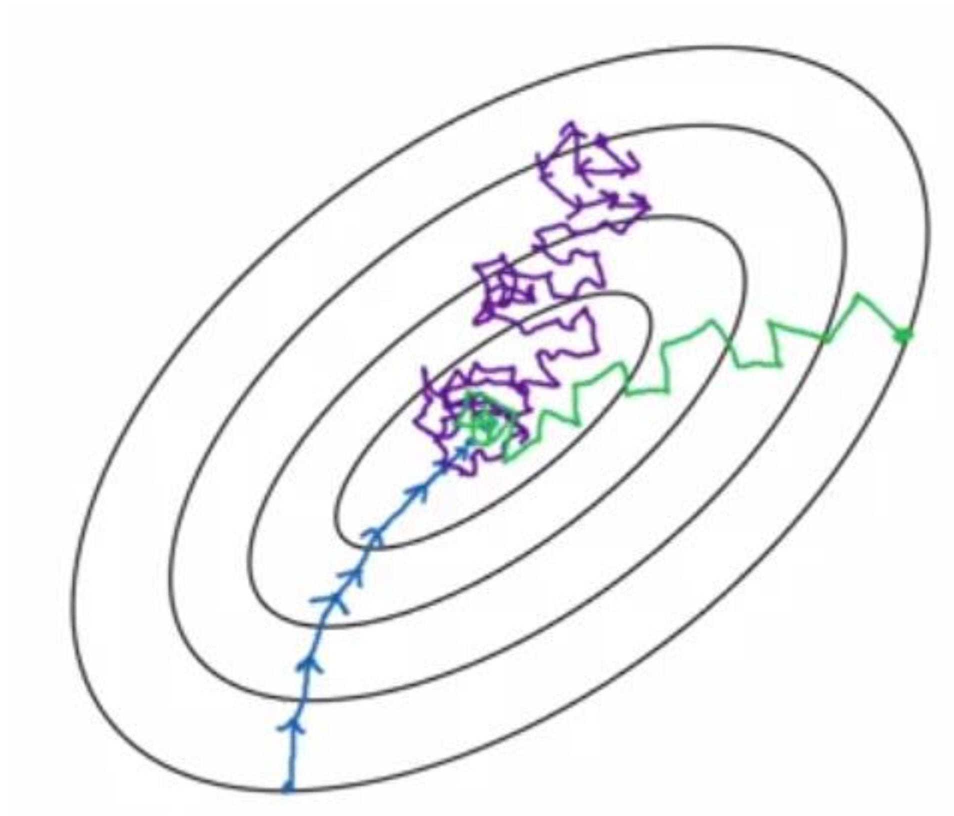
Batch gradient descent

Batch size=n
(the size of training set)

- Need to process whole training set for weight update

Batch size

- Full batch (batch size = N)
- SGB (batch size = 1)
- SGD (batch size = 128)

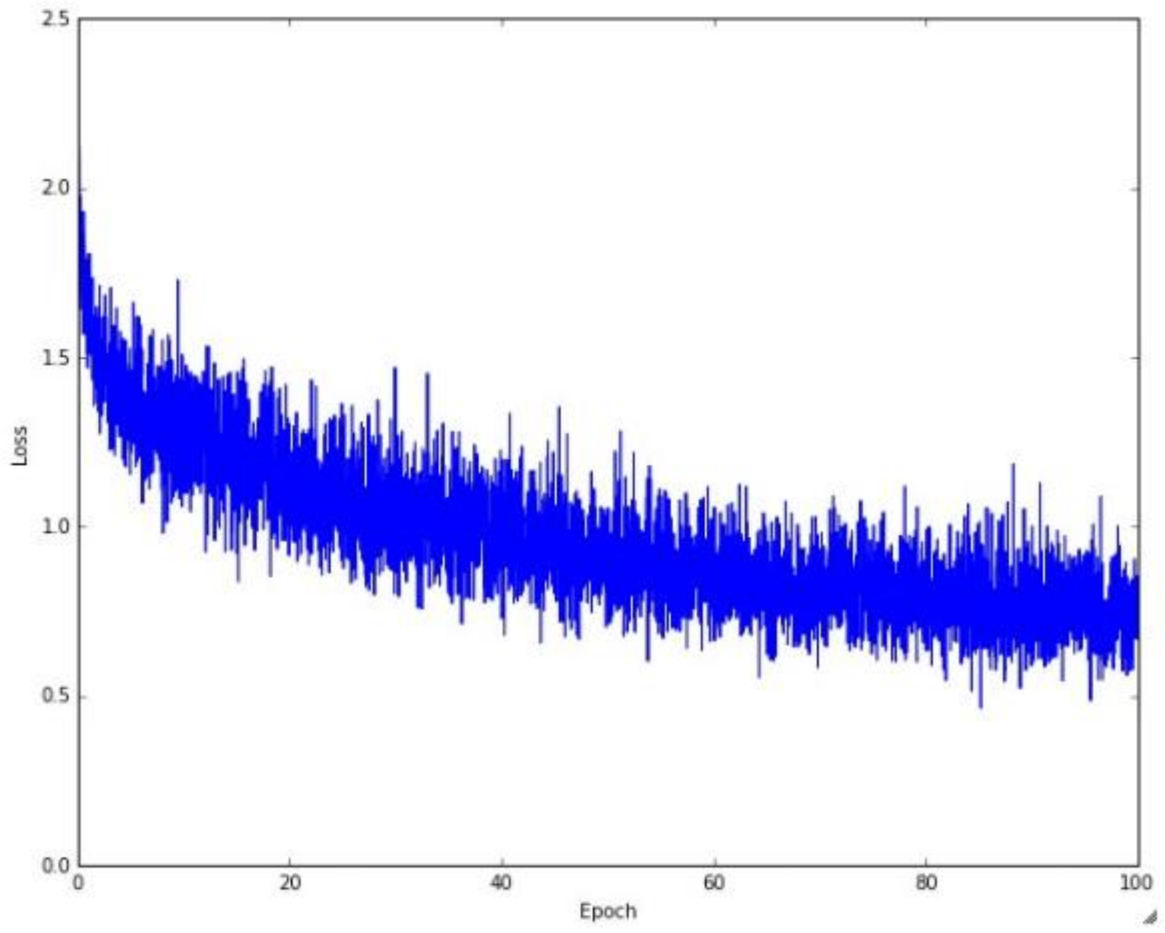


[Andrew Ng, Deep Learning Specialization, 2017]
© 2017 Coursera Inc.

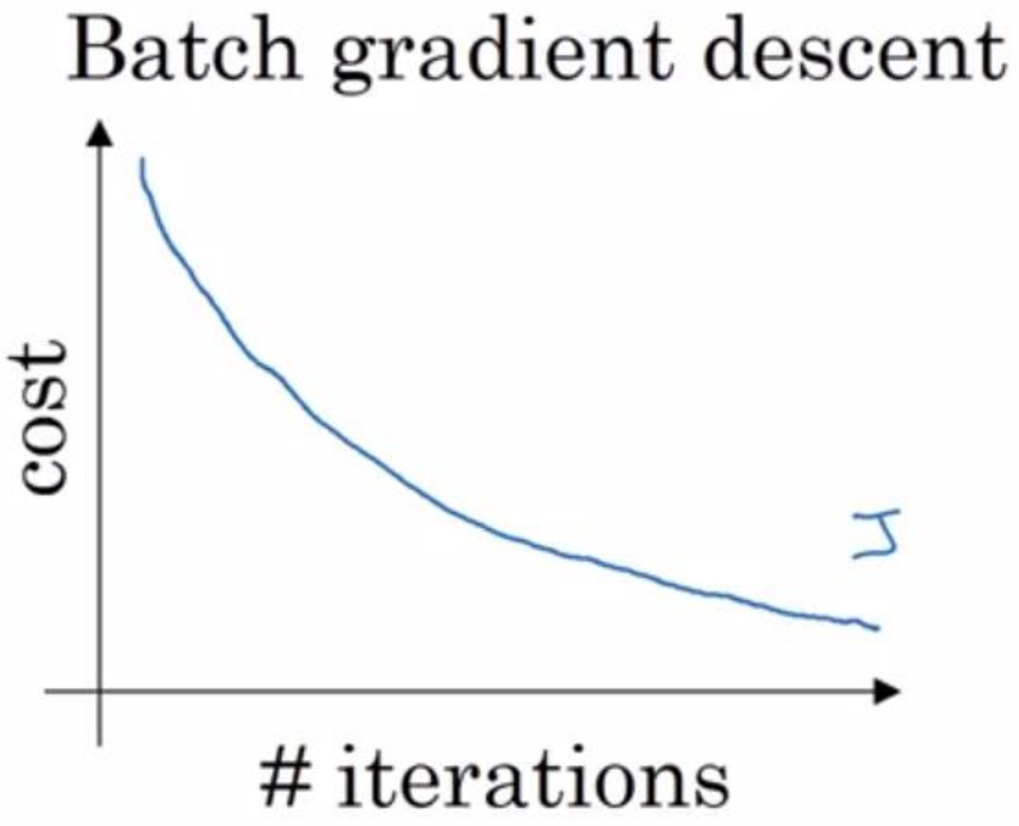
Choosing mini-batch size

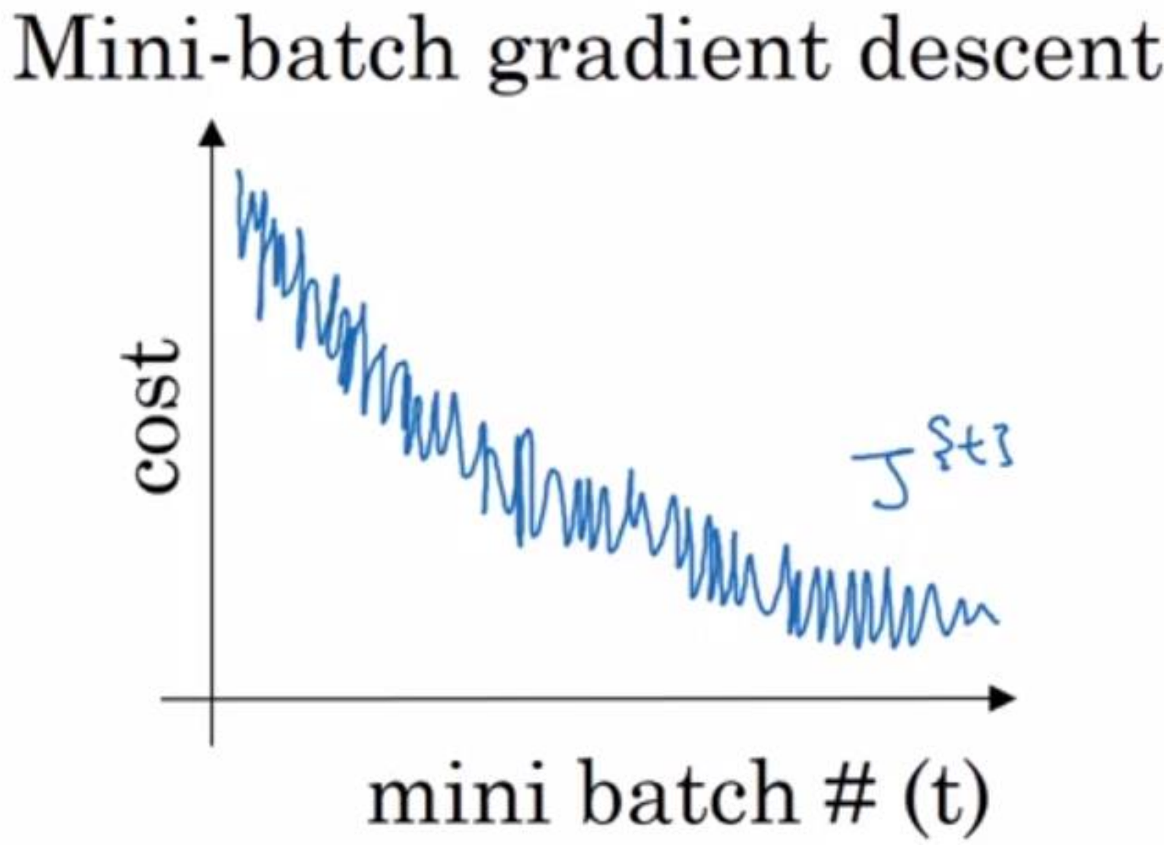
- For small training sets (e.g., n<2000) use full-batch gradient descent
- Typical mini-batch sizes for larger training sets:
  - 64, 128, 256, 512
- Make sure one batch of training data and the corresponding forward, backward required to be cached can fit in CPU memory

Mini-batch gradient descent: loss-#epoch curve



Mini-batch gradient descent vs. full-batch





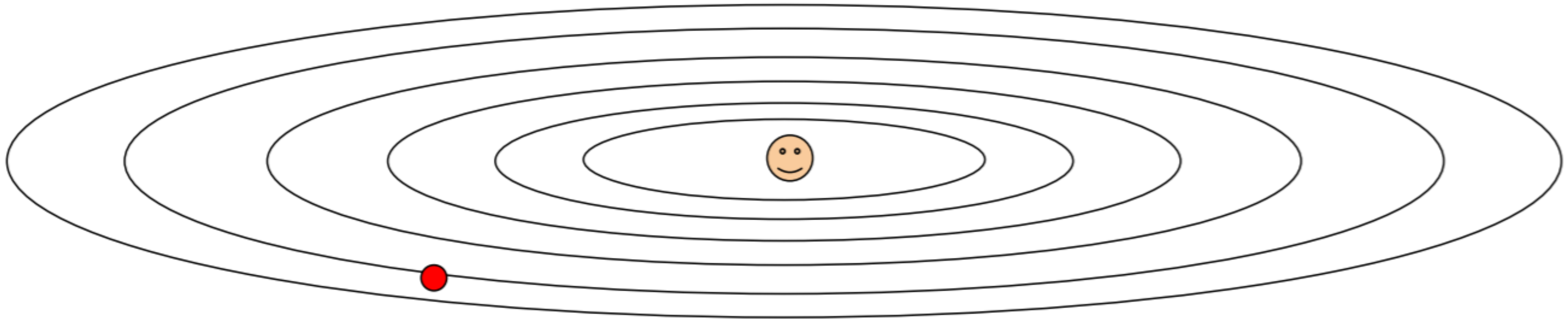
[Andrew Ng, Deep Learning Specialization, 2017]

Problems with gradient descent

- Poor conditioning
- Saddle points and local minima
- Noisy gradients in the SGD

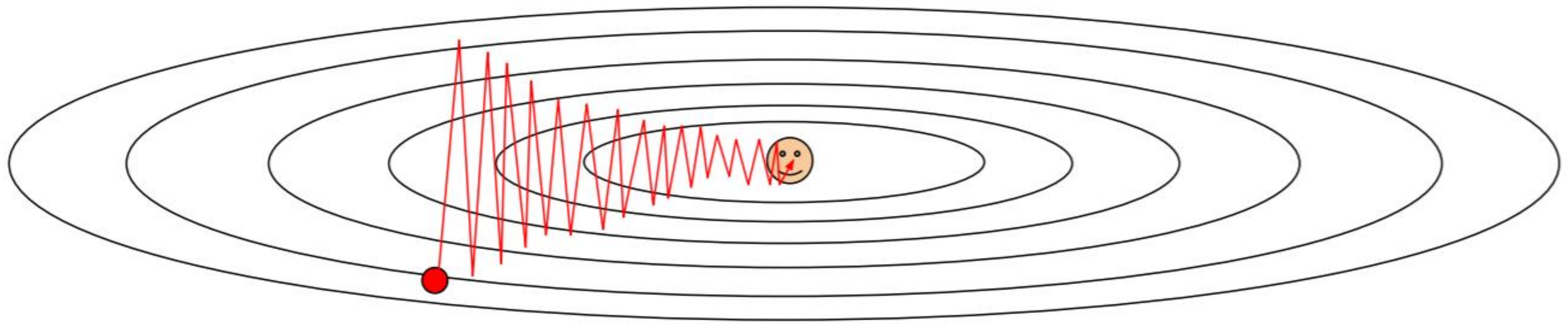
Problems with gradient descent

- What if loss changes quickly in one direction and slowly in another? What does gradient descent do?



Problems with gradient descent

- What if loss changes quickly in one direction and slowly in another? What does gradient descent do?
- Very slow progress along shallow dimension, jitter along steep direction

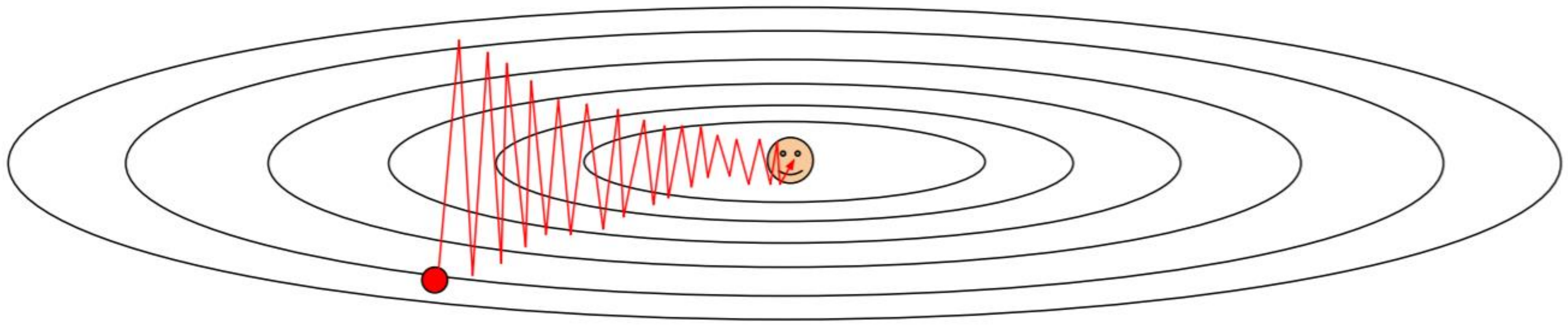


Poor conditioning

Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large

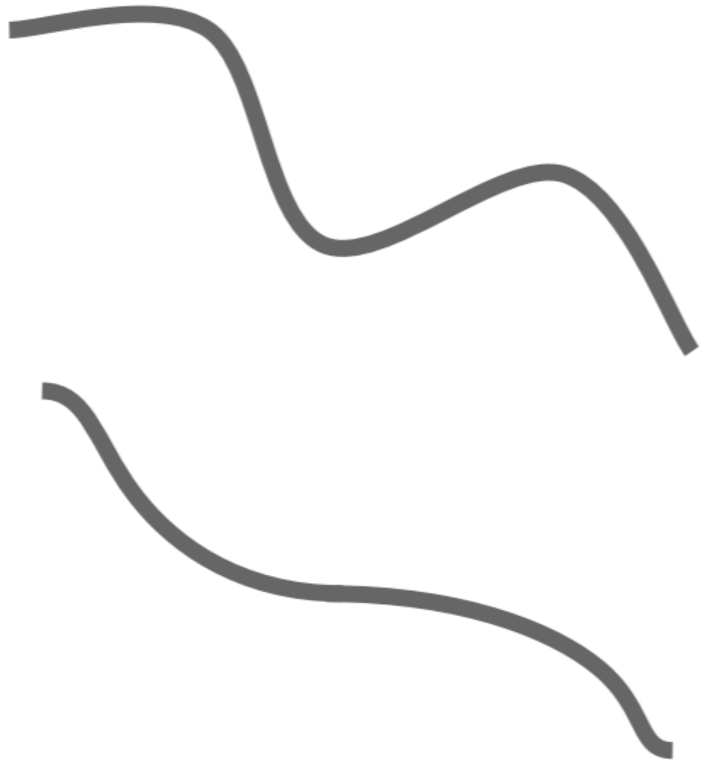
Convergence speed of gradient descent

- Learning rate
  - If the learning rate is big, the weights slosh to and fro across the ravine.
  - If the learning rate is too big, this oscillation diverges.
  - What we would like to achieve:
    - Move quickly in directions with small but consistent gradients.
    - Move slowly in directions with big but inconsistent gradients.
- How to train your neural networks much more quickly?



Optimization: Problems with SGD

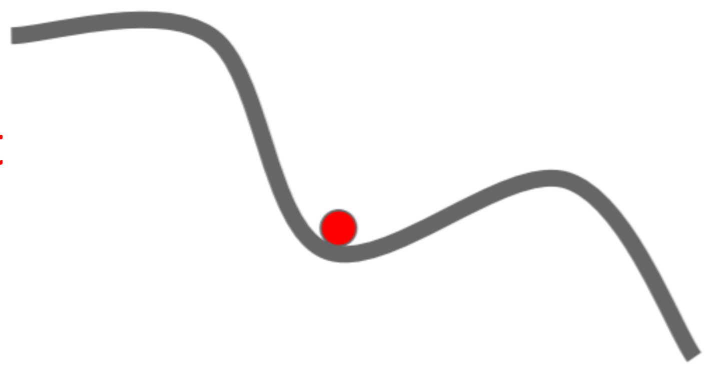
- What if the loss function has a local minima or saddle point?



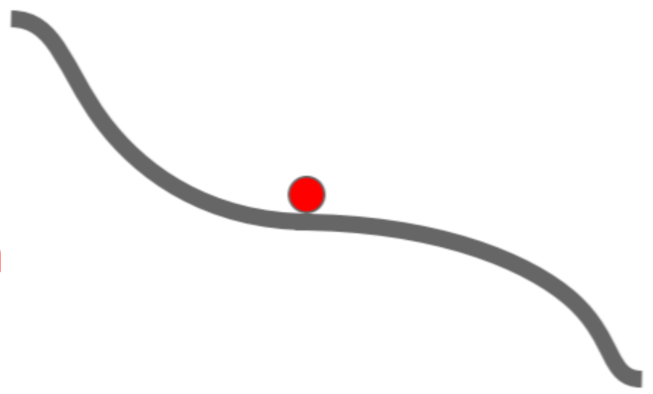
Optimization: Problems with SGD

- What if the loss function has a local minima or saddle point?

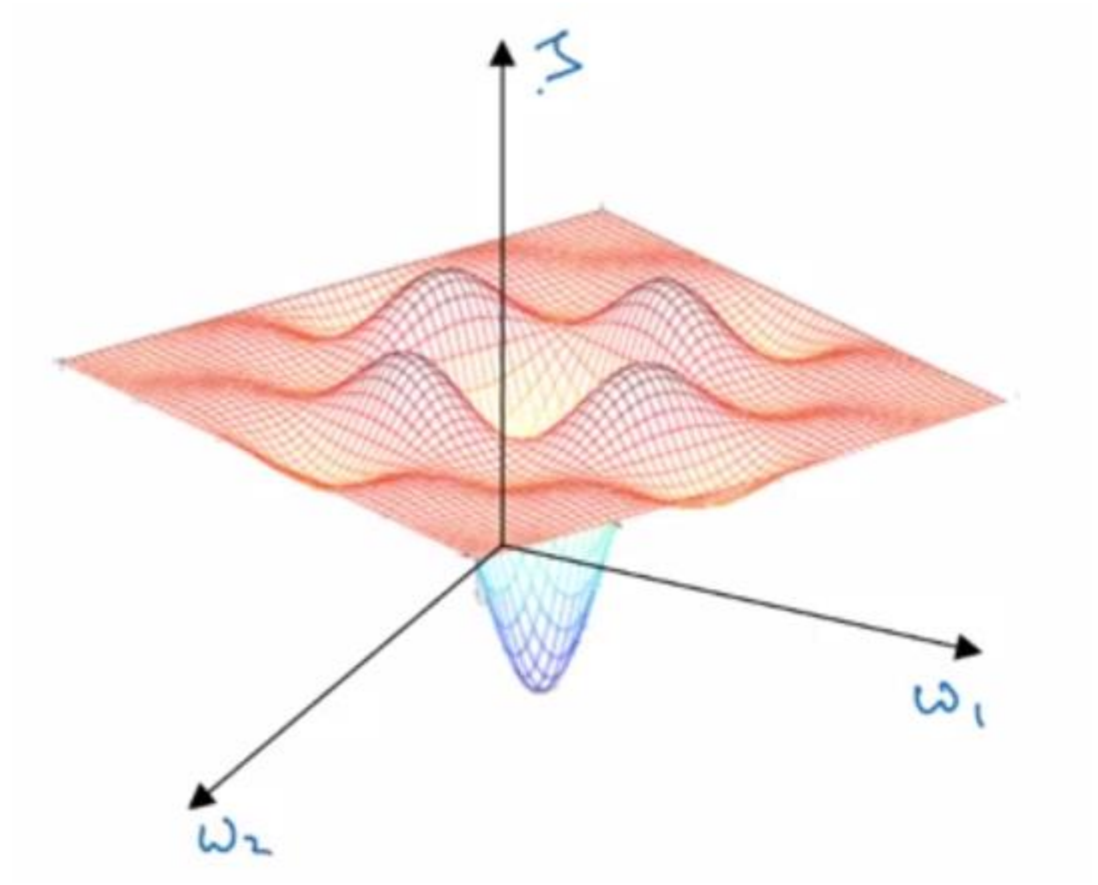
Zero gradient, gradient descent gets stuck



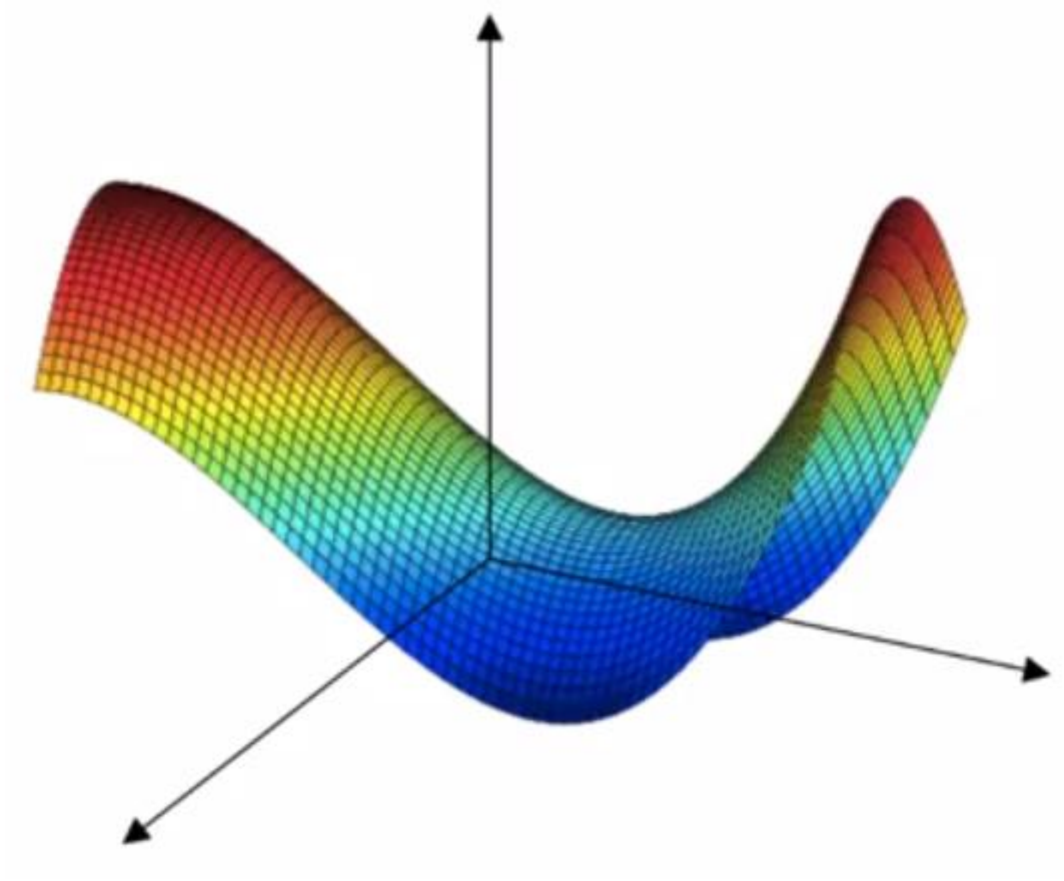
Saddle points much more common in high dimension



The problem of local optima



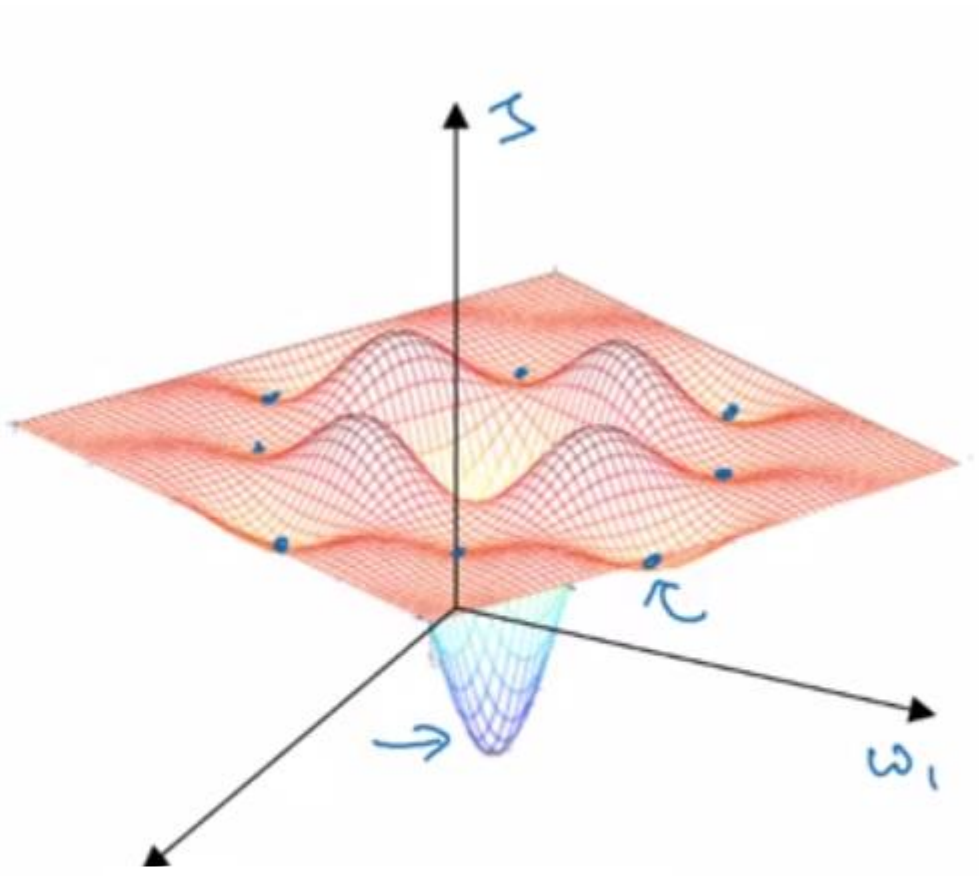
What about very high-dimensional spaces?

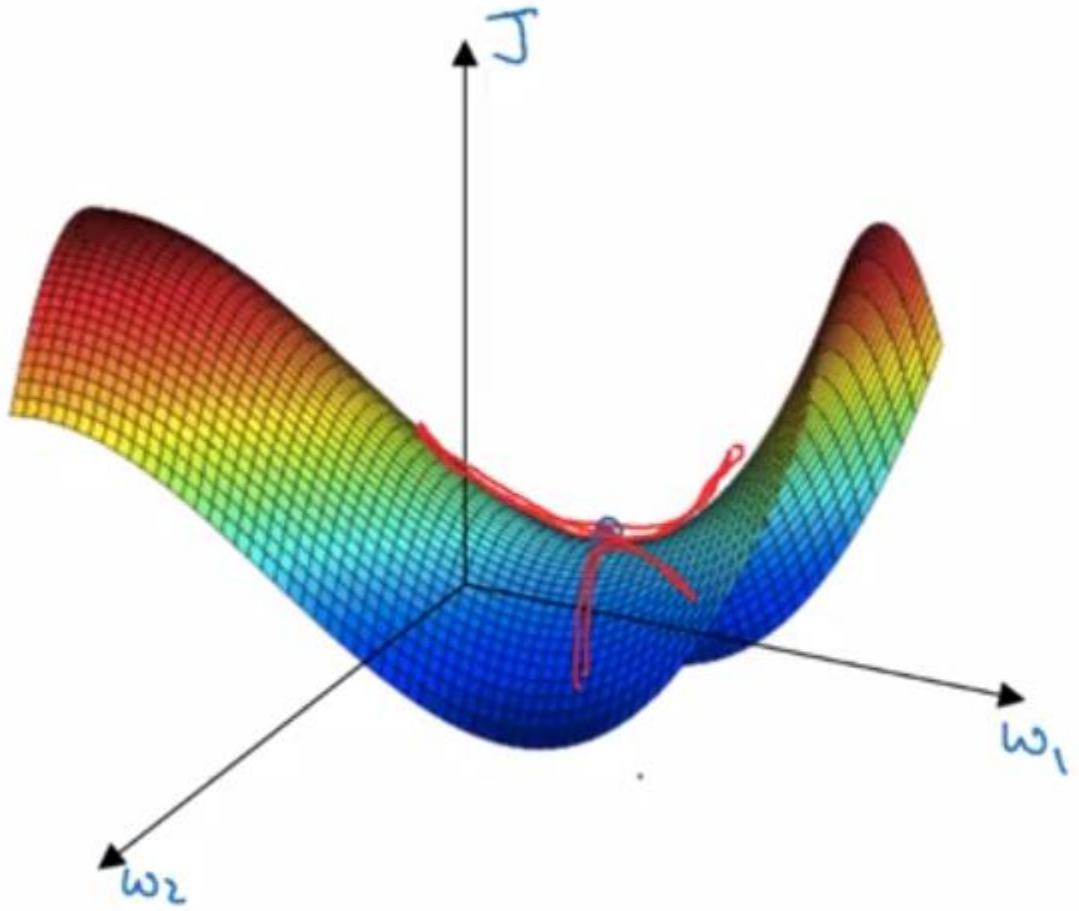


For high dimensional parameter space, most points of zero gradients are not local optima and are saddle points.

[Andrew Ng, Deep Learning Specialization, 2017]

The problem of local optima





Andrew Ng, Deep Learning Specialization, 2017]

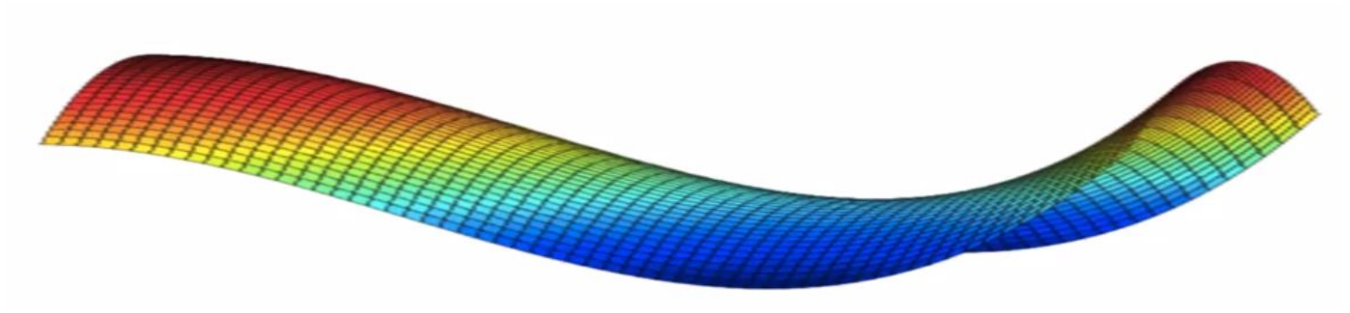
The problem of local optima

- A function of very high dimensional space:
  - if the gradient is zero, then in each direction it can either be a convex light function or a concave light function.
  - for it to be a local optima, all directions need to look like this.
    - And so the chance of that happening is maybe very small

Problem of plateaus

- The problem is near the saddle points

- Plateau is a region where the derivative is close to zero for a long time
  - Very very slow progress
  - Finally, algorithm can then find its way off the plateau.



[Andrew Ng, Deep Learning Specialization, 2017]

Problems with stochastic gradient descent

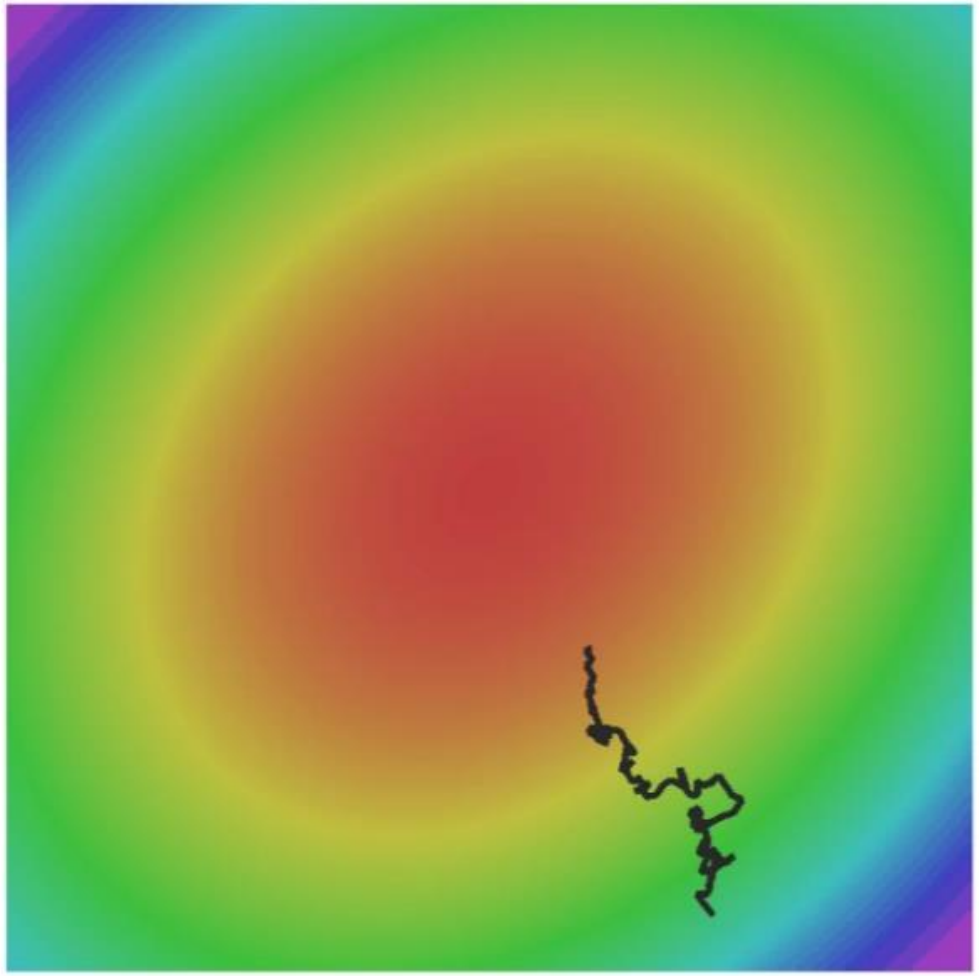
- Noisy gradients by SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$

- Might need long time



SGD + Momentum

$v_{dW} = 0$

On each iteration:

  Compute $\nabla_W J$ on the current mini-batch

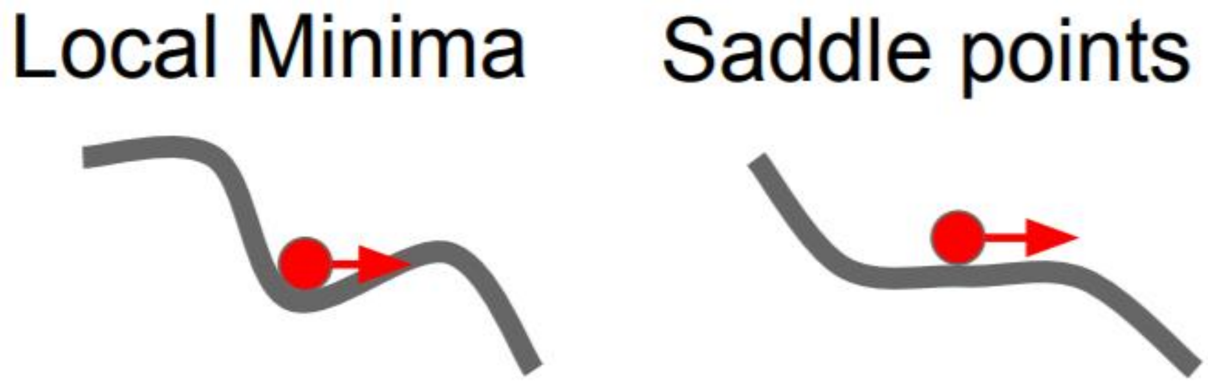
  $v_{dW} = \beta v_{dW} + (1-\beta)\nabla_W J$

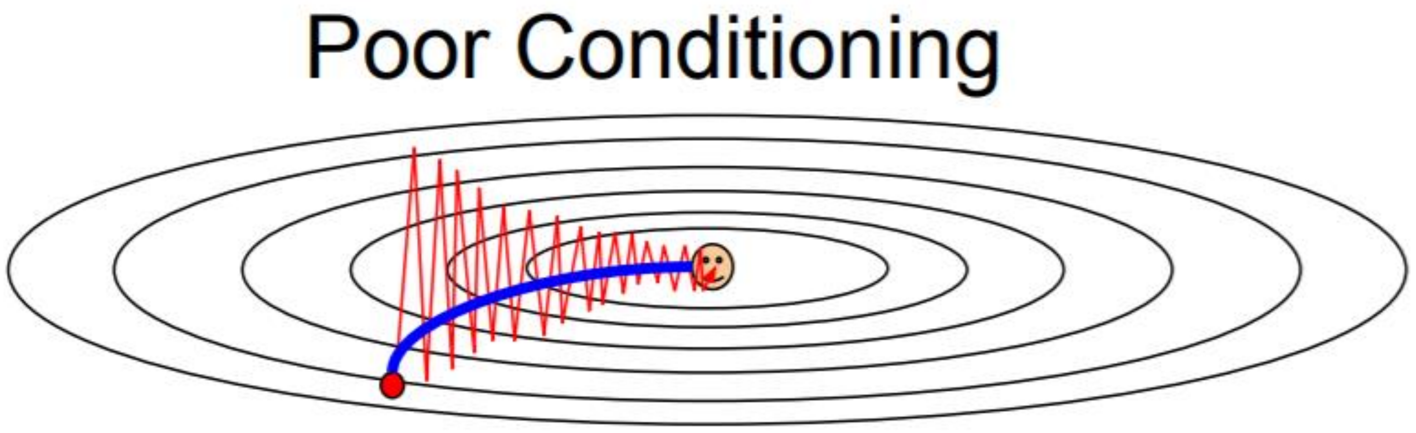
  $W = W - \alpha v_{dW}$

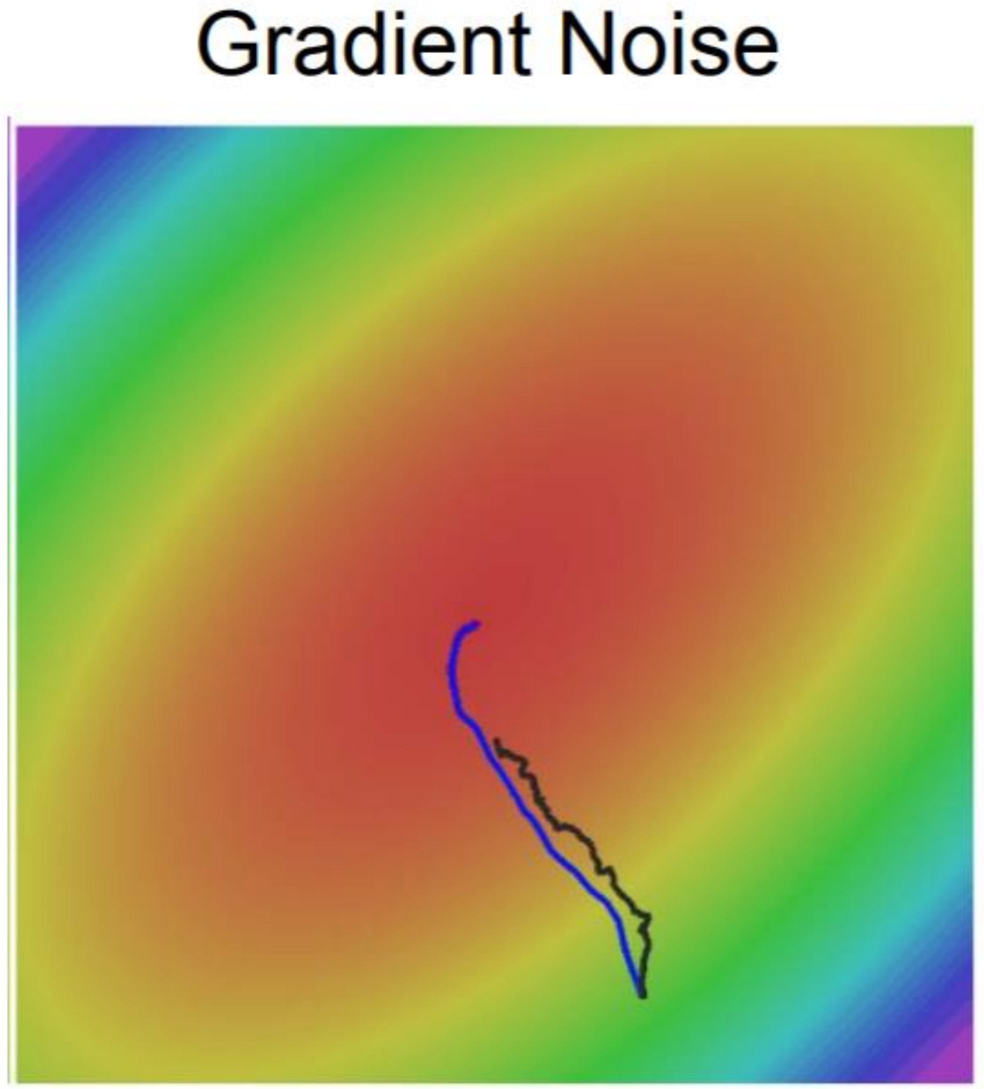
Build up “velocity” as a running mean of gradients
e.g., β=0.9 or 0.99

- We will show that v_{dW} is moving average of recent gradients with exponentially decaying weights

Momentum helps to reduce problems with SGD







Exponentially weighted averages

$\theta_1 = 40°F$

$\theta_2 = 49°F$

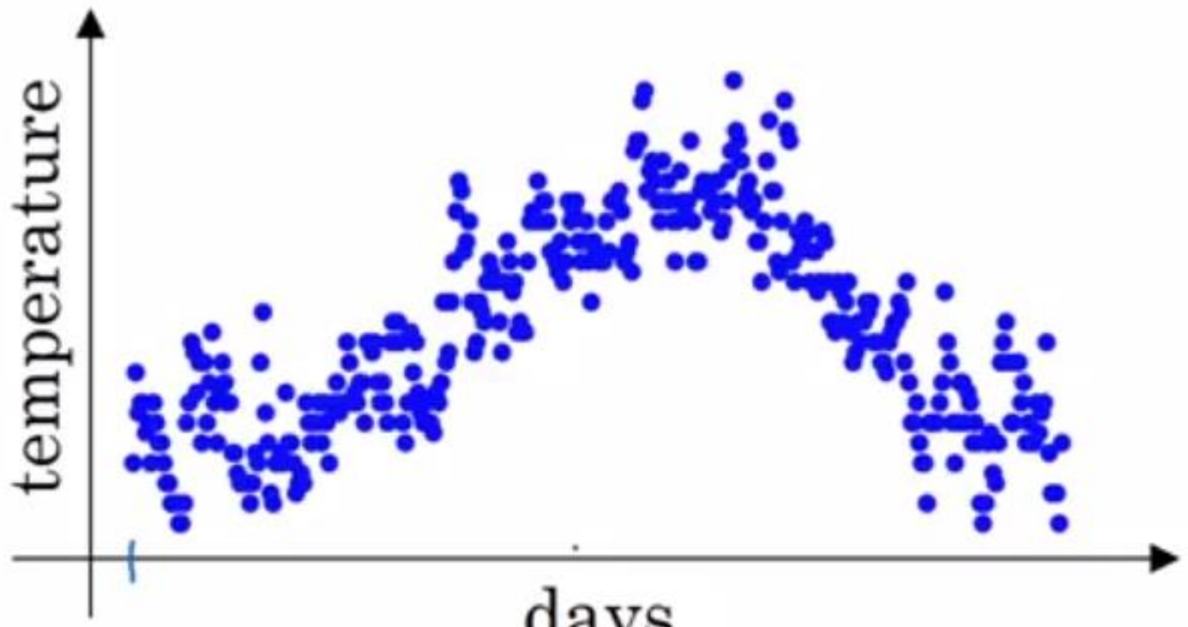
$\theta_3 = 45°F$

$\vdots$

$\theta_{180} = 60°F$

$\theta_{181} = 56°F$

$\vdots$



$v_0 = 0$

$v_1 = 0.9 v_0 + 0.1 \theta_1$

...

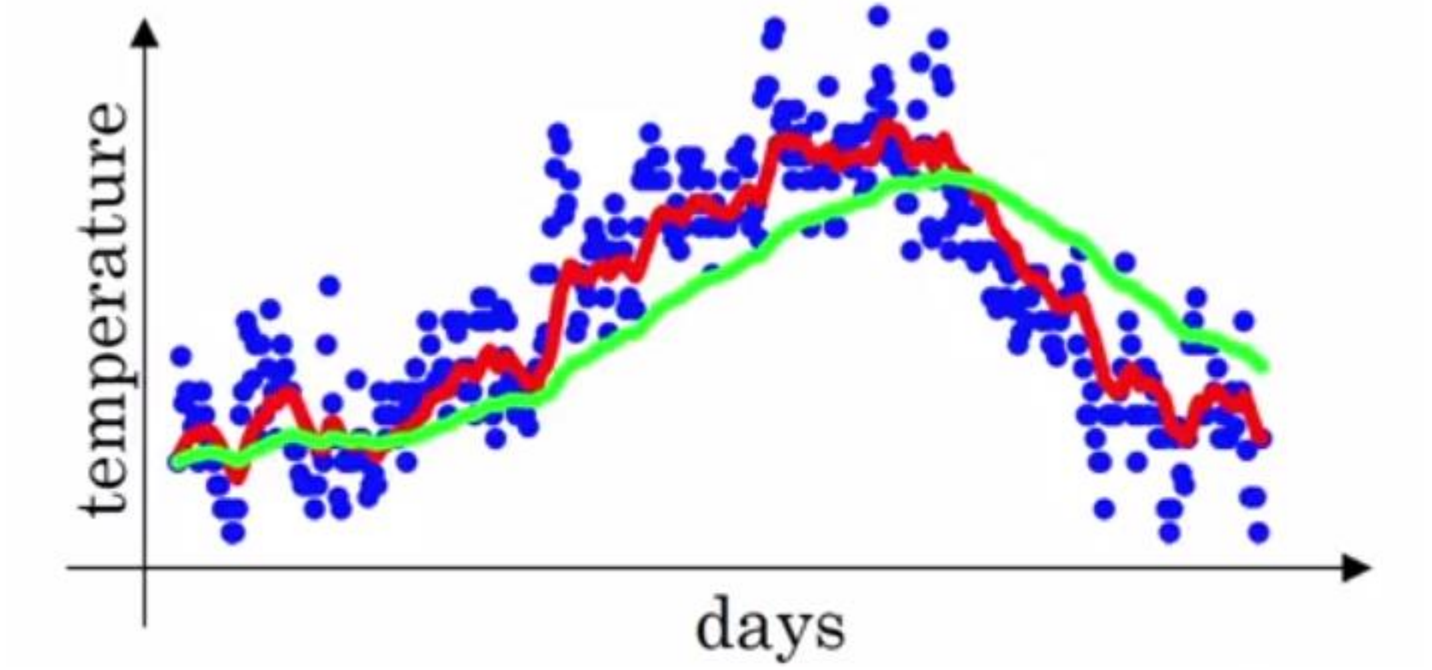
$v_t = 0.9 v_{t-1} + 0.1 \theta_t$

Exponentially weighted averages

- $v_0 = 0$
- $v_t = \beta v_{t-1} + (1 - \beta)\theta_t$

$\beta = 0.9$

$\beta = 0.98$



Exponentially weighted averages

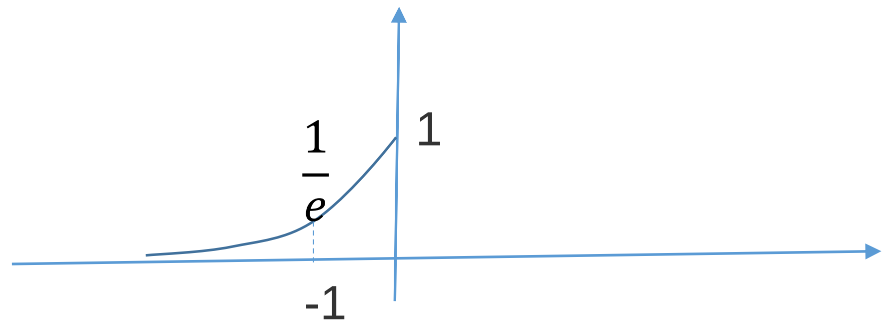
- $v_0 = 0$
- $v_t = \beta v_{t-1} + (1-\beta)\theta_t$

$$v_t = (1-\beta)(\theta_t + \beta\theta_{t-1} + \beta^2\theta_{t-2} + \cdots + \beta^{t-1}\theta_1)$$

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

$$0.9^{10} = \frac{1}{e}$$

$$0.98^{50} = \frac{1}{e}$$



Bias correction

- Computation of these averages more accurately:

$$v_t^c = \frac{v_t}{1-\beta^t}$$

- Example:

$$v_0 = 0$$
$$v_1 = 0.98 v_0 + 0.02\theta_1 = 0.02\theta_1$$
$$v_2 = 0.98 v_1 + 0.02\theta_2 = 0.0196\theta_1 + 0.02\theta_2$$

$$v_2^c = \frac{0.0196\theta_1 + 0.02\theta_2}{1-0.98^2} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

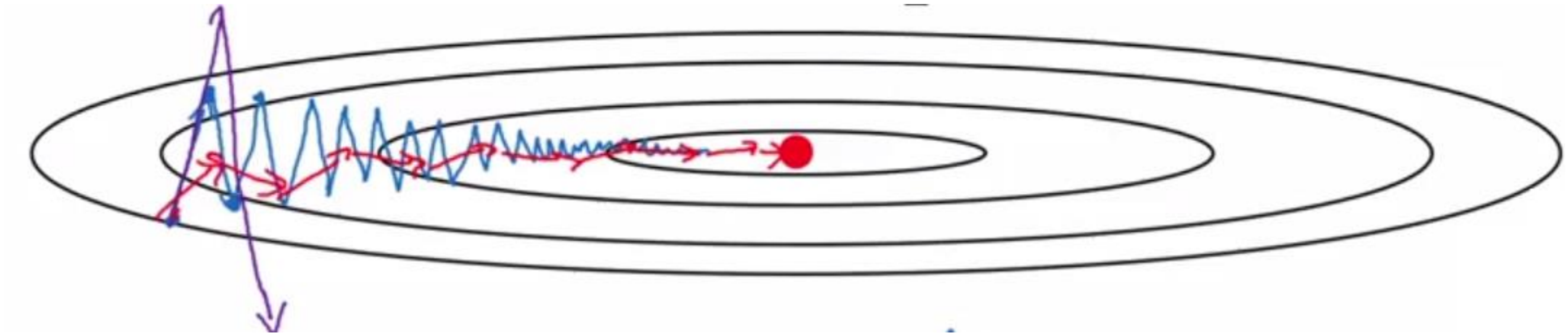
[Andrew Ng, Deep Learning Specialization, 2017]

Bias correction

$$v_t^c = \frac{v_t}{1 - \beta^t}$$

- when you're still warming up your estimates, the bias correction can help
- when t is large enough, the bias correction makes almost no difference
- people don't often bother to implement bias corrections.
  - rather just wait that initial period and have a slightly more biased estimate and go from there.

Gradient descent with momentum



$v_{dW} = 0$

On each iteration:

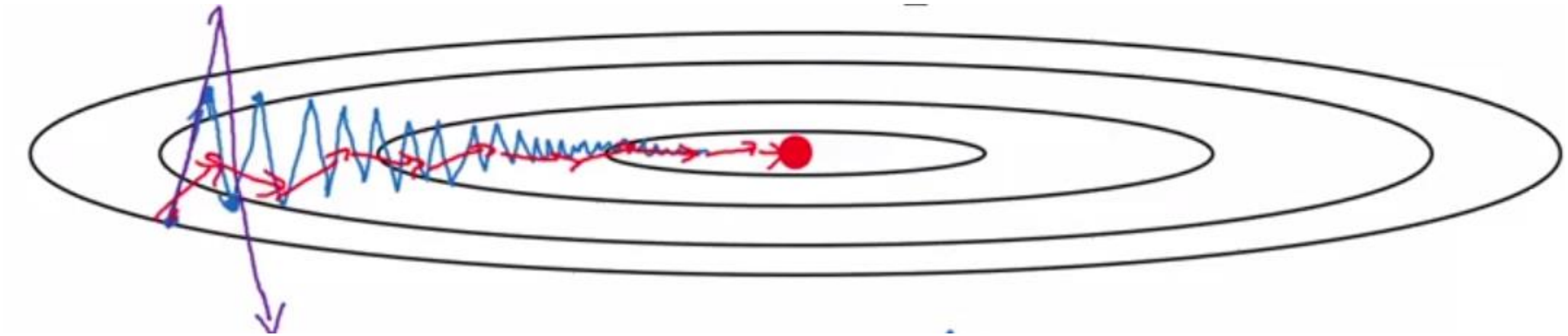
  Compute dW on the current mini-batch

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

  $W = W - \alpha v_{dW}$

- Hyper-parameters β and α $(\beta = 0.9)$

Gradient descent with momentum



$v_{dW} = 0$

On each iteration:

Compute dW on the current mini-batch

$v_{dW} = \beta v_{dW} + (1-\beta) dW$ $\longrightarrow$ $v_{dW} = \beta v_{dW} + dW$

$W = W - \alpha v_{dW}$

learning rate would need to be tuned differently for these two different versions

- Hyper-parameters β and α ($\beta = 0.9$)

Gradient descent with momentum

$v_{dW} = 0$

On each iteration:

Compute dW on the current mini-batch

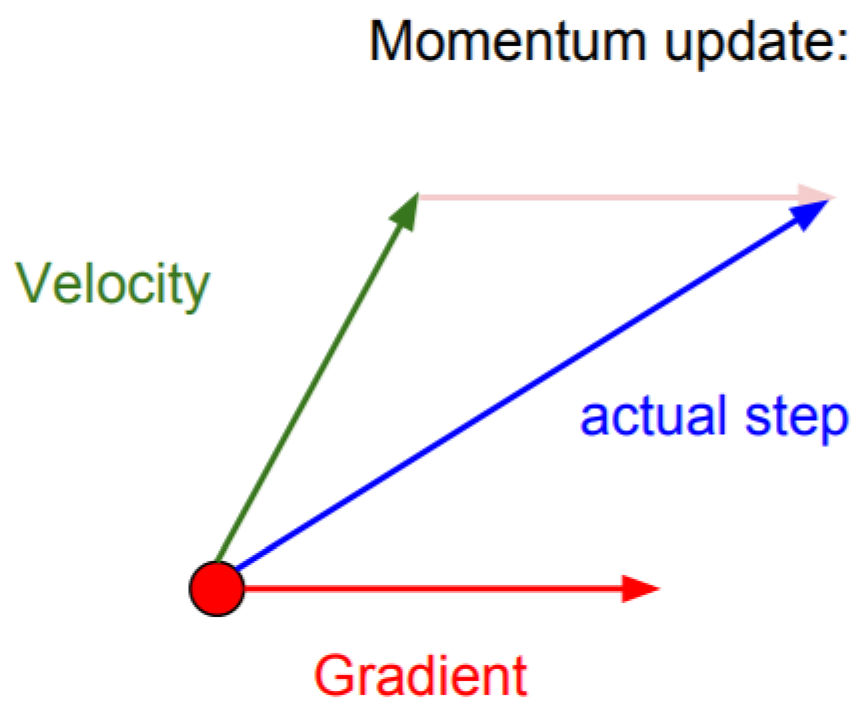
$v_{dW} = \beta v_{dW} + dW$

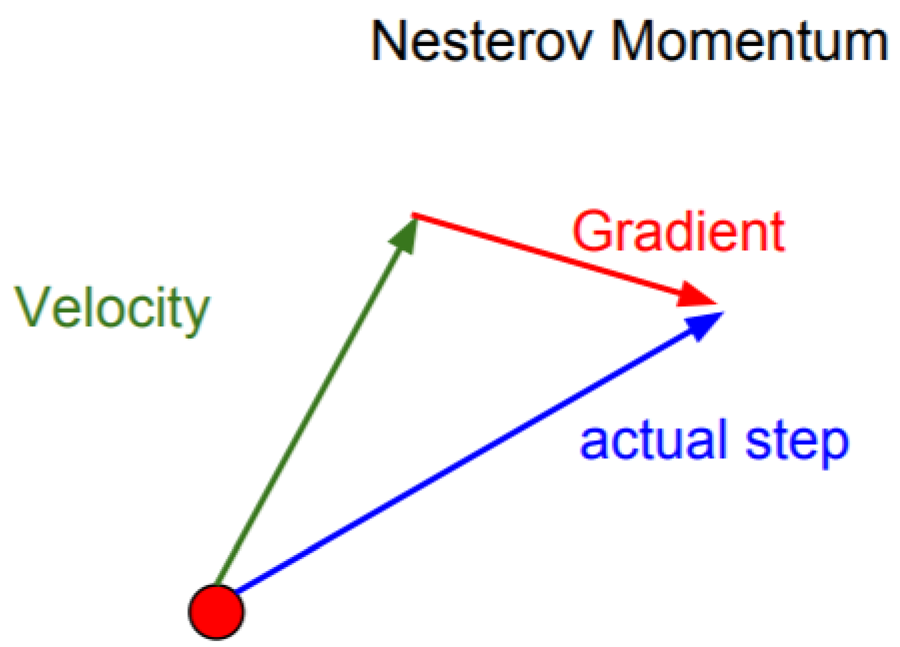
$W = W - \alpha v_{dW}$

Nesterov Momentum

- **Nesterov Momentum** is a slightly different version of the momentum update
  - It enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistently works slightly better than standard momentum

Nesterov Momentum





Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deel learning”, ICML 2013

Nesterov Momentum

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

Simple update rule

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

Nesterov

Nesterov Accelerate Gradient (NAG)

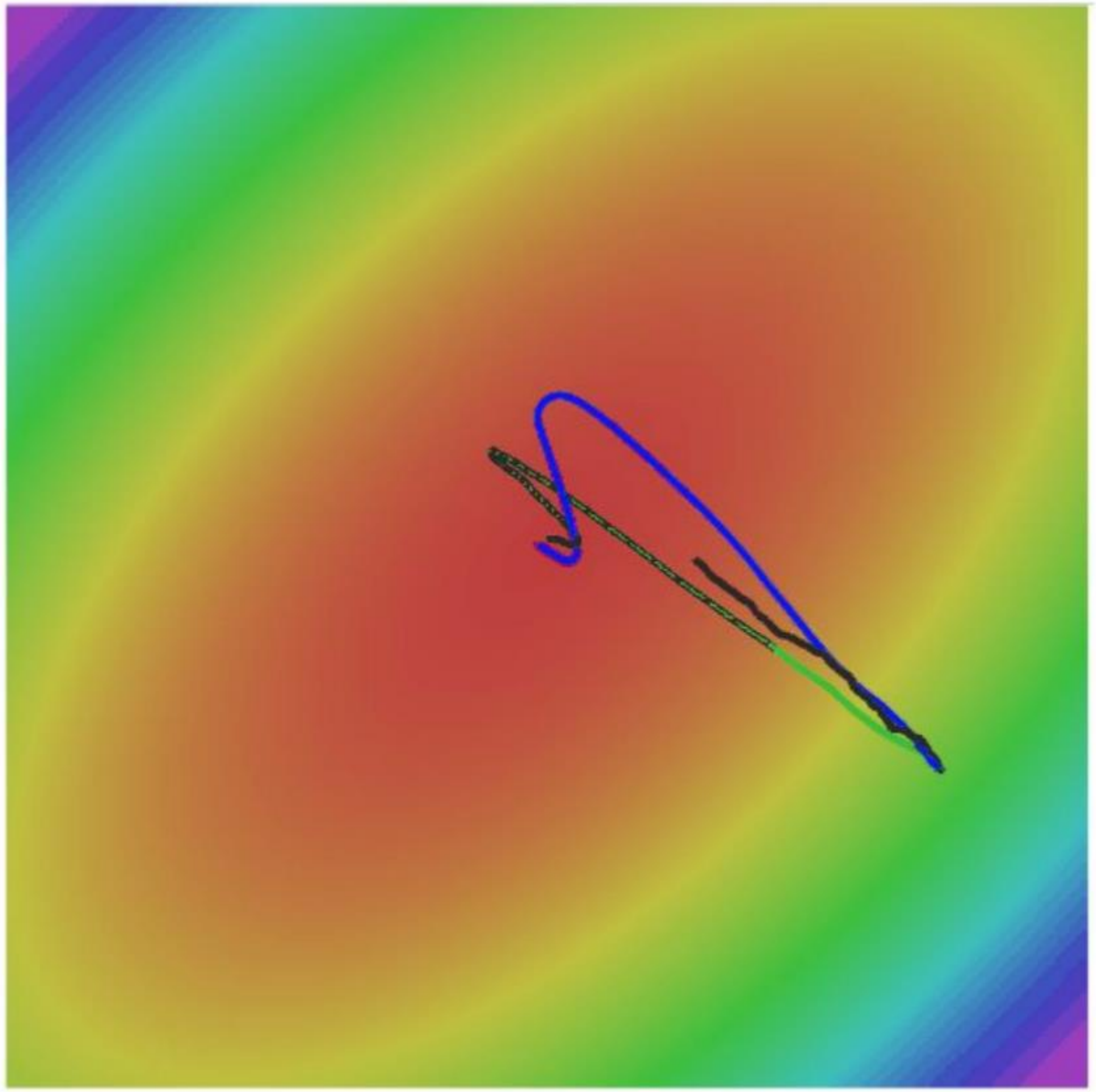
- Nestrov accelerated gradient
  - a little is annoying (requires gradient at $x_t + \rho v_t$ instead of at x_t)
    - You like to evaluate gradient and loss in the same point

$$\boxed{\begin{aligned} v_{t+1} &= \rho v_t - \alpha \nabla f(\boxed{x_t + \rho v_t}) \\ x_{t+1} &= x_t + v_{t+1} \end{aligned}}$$

  - With change of variables, it is resolved:

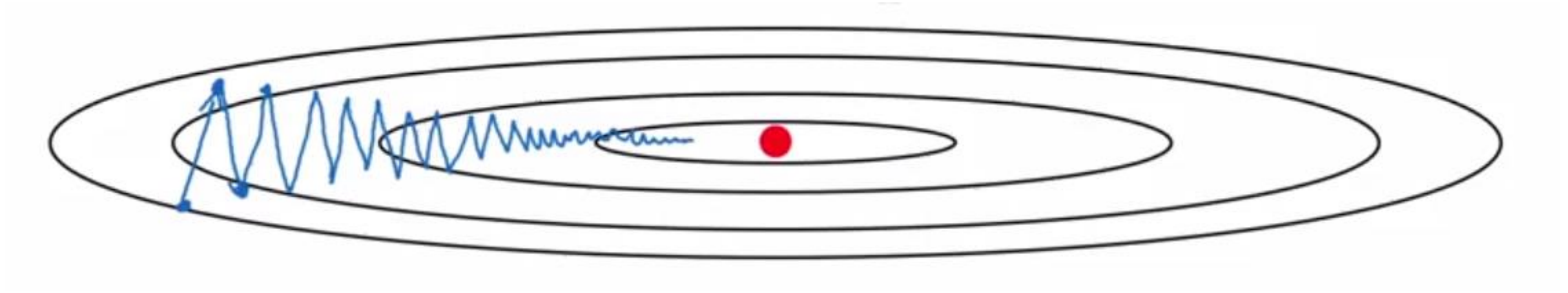
$$\tilde{x}_t = x_t + \rho v_t \qquad \boxed{\begin{aligned} v_{t+1} &= \rho v_t - \alpha \nabla f(\tilde{x}_t) \\ \tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t) \end{aligned}}$$

Nesterov Momentum



SGD
SGD+Momentum
Nesterov

RMSprop (Root Mean Square prop)



- How to adjust the scale of each dimension of dW?

RMSprop

$S_{dW} = 0$

On iteration t:

Compute dW on the current mini-batch

$S_{dW} = \beta S_{dW} + (1-\beta) dW^2$ $\longrightarrow$ Element-wise: $dW_i^2 = dW_i * dW_i$

$W = W - \alpha \frac{dW}{\sqrt{S_{dW}}}$

S_{dW}: an exponentially weighted average of the squares of the derivatives (second order moment)

Adagrad

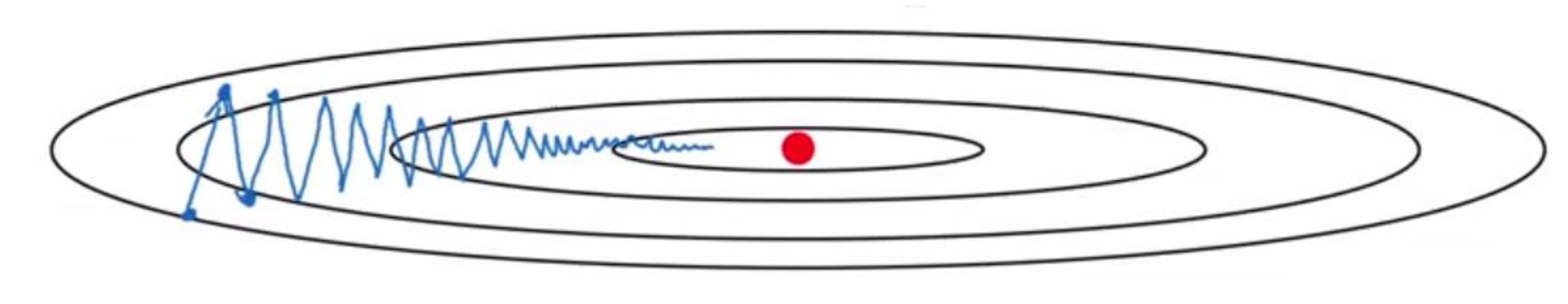
$S_{dW} = S_{dW} + dW^2$

(S_{dW}: Sum of the squares of derivatives)

Step size will be very small during this algorithm

RMSprop

- Updates in the vertical direction is divided by a much larger number while in the horizontal direction by a smaller number.



- Thus, you can therefore use a larger learning rate alpha

RMSprop

$S_{dW} = 0$

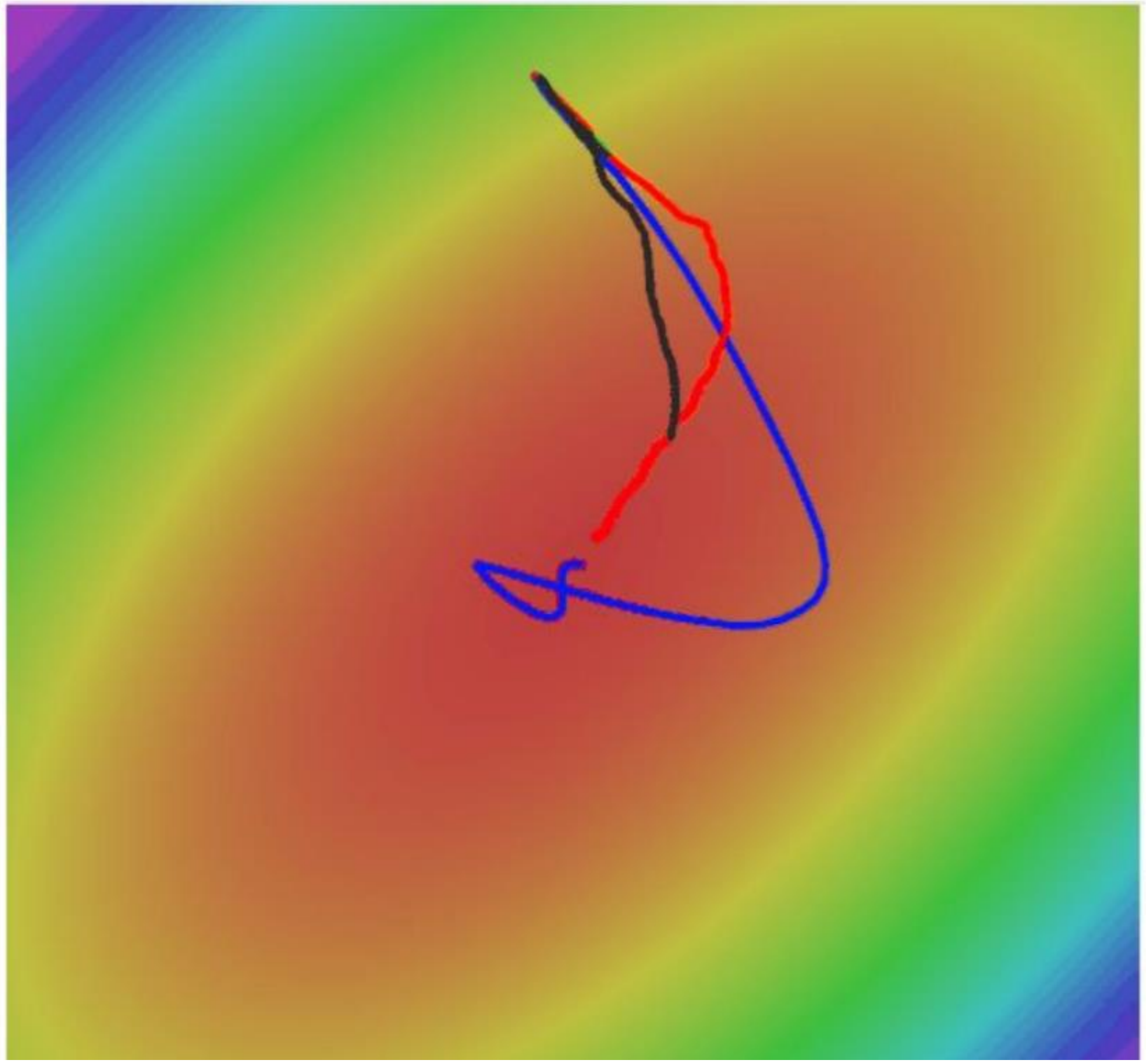
On iteration t:

Compute dW on the current mini-batch

$S_{dw} = \beta S_{dw} + (1-\beta) dW^2$

$W = W - \alpha \frac{dW}{\sqrt{S_{dw} + \epsilon}}$ $\epsilon = 10^{-8}$

RMSprop



▬	SGD
▬	SGD+Momentum
▬	RMSProp

Adam (Adaptive Moment Estimation)

- Takes RMSprop and momentum idea together

$v_{dW} = 0, S_{dW} = 0$

On each iteration:

Compute dW on the current mini-batch

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

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

$W = W - \alpha \frac{V_{dW}}{\sqrt{S_{dW} + \epsilon}}$

Problem: may cause very large steps at the beginning

Solution: Bias correction to prevent first and second moment estimates start at zero

Adam (full form)

- Takes RMSprop and momentum idea together

$v_{dW} = 0, S_{dW} = 0$

On each iteration:

Compute dW on the current mini-batch

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

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

$V_{dW}^c = \frac{V_{dW}}{1-\beta_1^t}$

$S_{dW}^c = \frac{S_{dW}}{1-\beta_2^t}$

$W = W - \alpha \frac{V_{dW}^c}{\sqrt{S_{dW}^c + \epsilon}}$

Adam

- Takes RMSprop and momentum idea together

$v_{dW} = 0, S_{dW} = 0$

On each iteration:

Compute dW on the current mini-batch

$$\boxed{V_{dW} = \beta_1 V_{dW} + (1-\beta_1) dW}$$ Momentum

$$\boxed{S_{dW} = \beta_2 S_{dW} + (1-\beta_2) dW^2}$$ RMSprop

$$V_{dW}^c = \frac{V_{dW}}{1-\beta_1^t}$$

$$S_{dW}^c = \frac{S_{dW}}{1-\beta_2^t}$$

Bias correction

$$W = W - \alpha \frac{V_{dW}^c}{\sqrt{S_{dW}^c + \epsilon}}$$

Hyperparameter Choice:

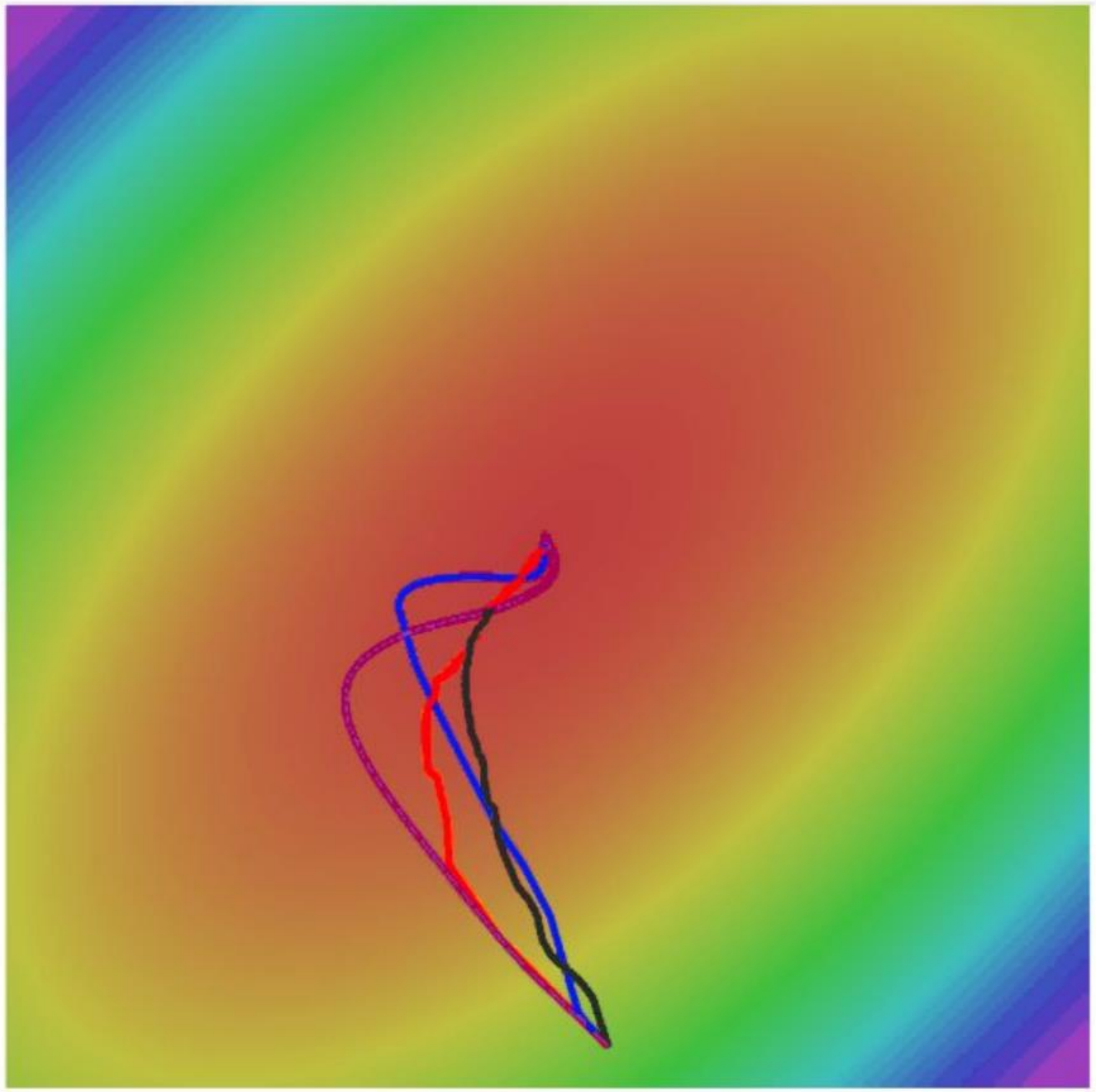
α: need to bee tuned

$\beta_1 = 0.9$

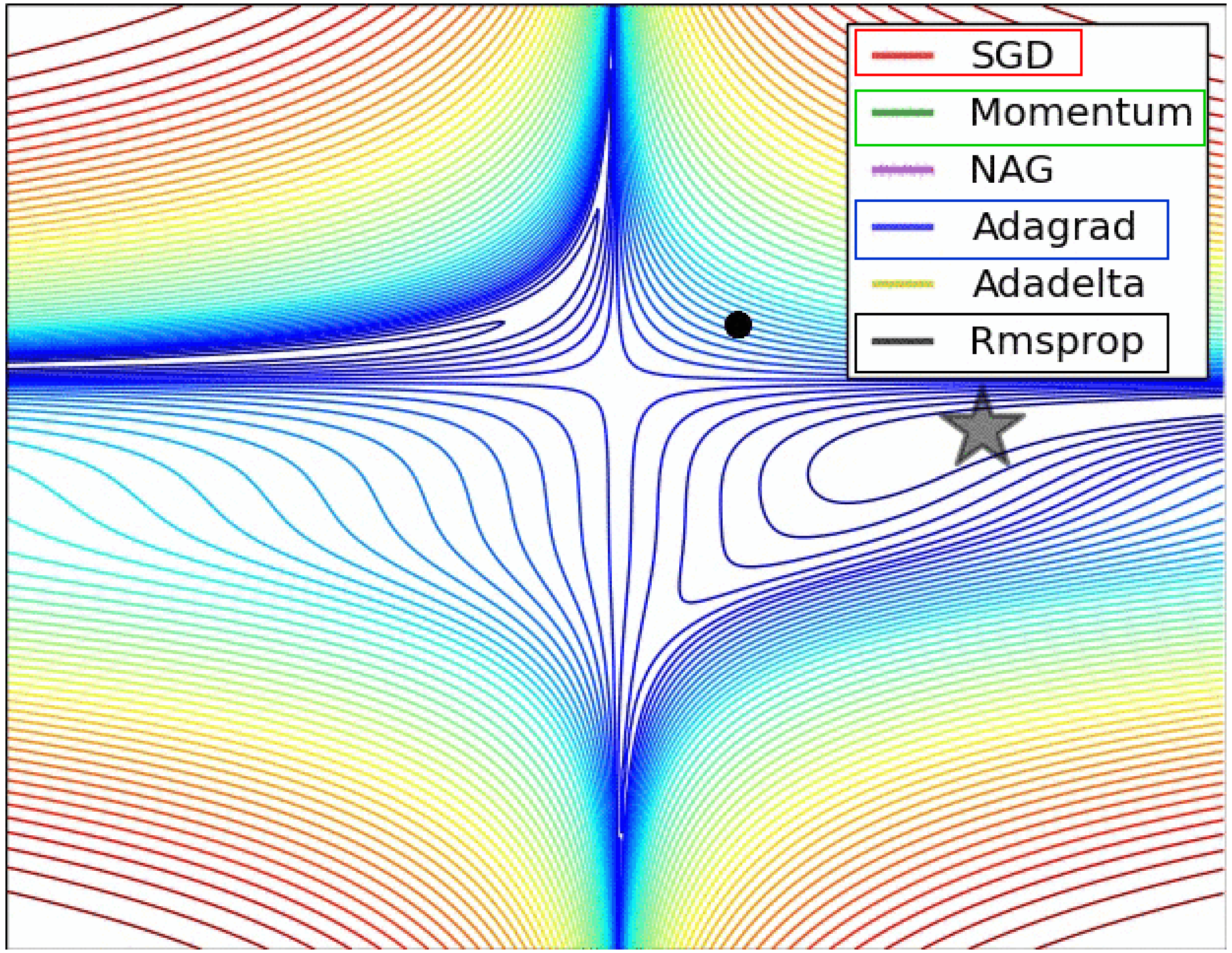
$\beta_2 = 0.999$

$\epsilon = 10^{-8}$

Adam



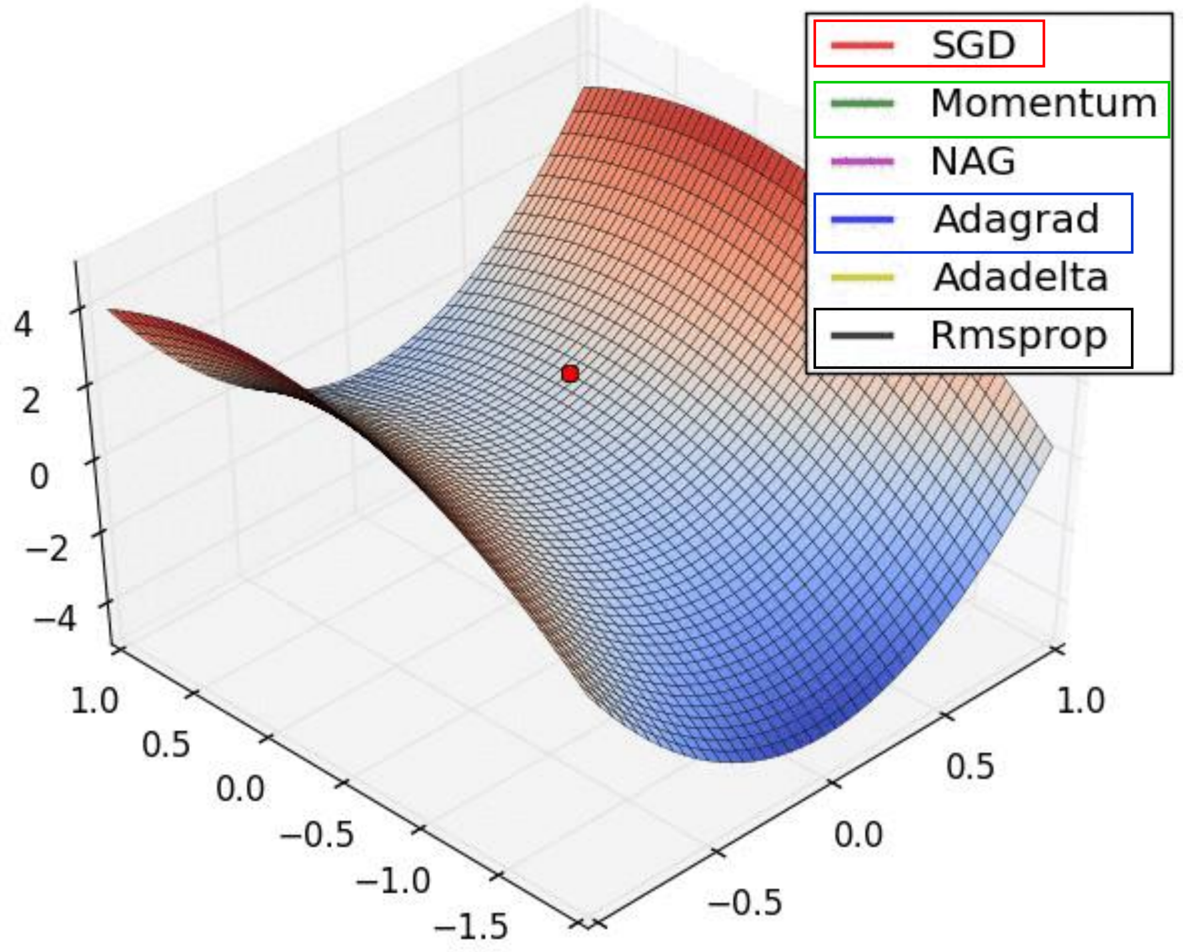
Example



RMSprop, AdaGrad, and Adam can adaptively tune the update vector size (per parameter)
Known also as methods with adaptive learning rates

Source: http://cs231n.github.io/neural-networks-3/

Example

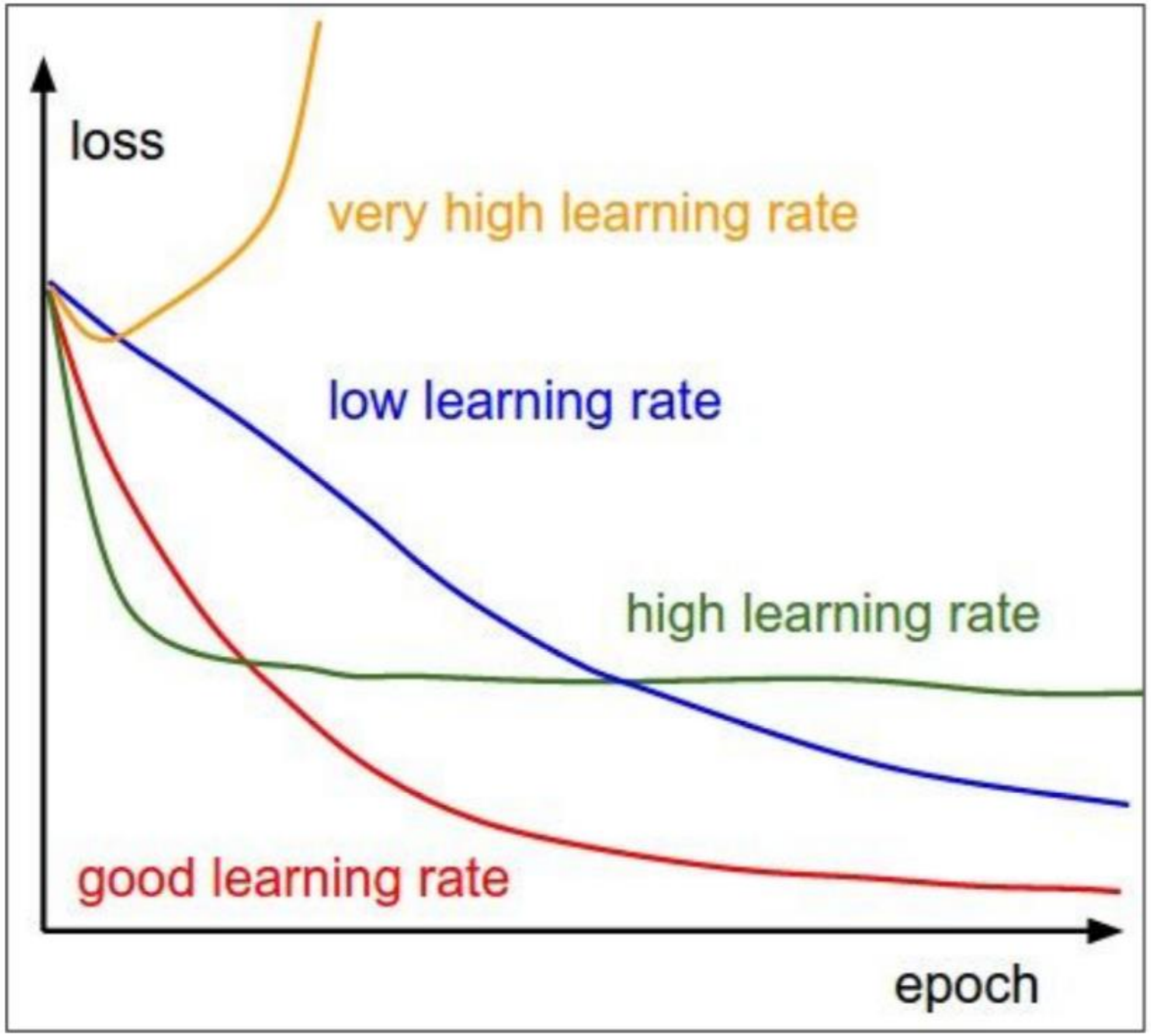


Source: http://cs231n.github.io/neural-networks-3/

Learning rate

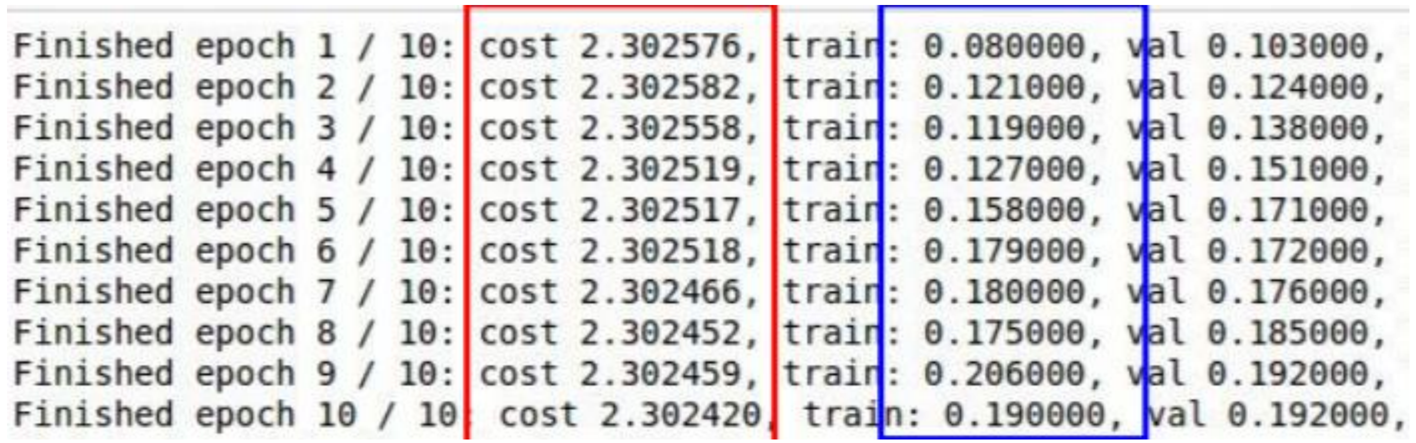
- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.
- Learning rate is an important hyperparameter that usually adjust first

Which one of these learning rates is best to use?



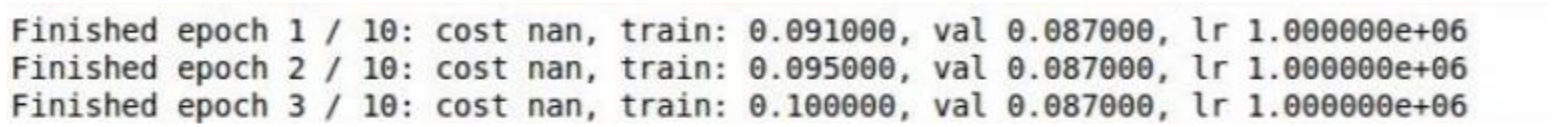
Choosing learning rate parameter

- loss not going down: learning rate too low



```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000,
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000,
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000,
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000,
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000,
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000,
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000,
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000,
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000,
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000,
```

- loss exploding: learning rate too high



```
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

cost: NaN almost always means high learning rate...

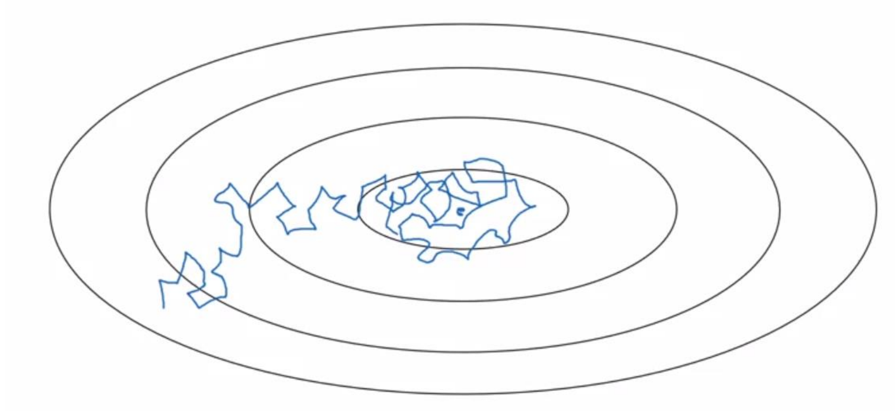
- Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]

Learning rate decay

- Maybe during the initial steps of learning, you could afford to take much bigger steps.
- But then as learning approaches converges, then having a slower learning rate allows you to take smaller steps.

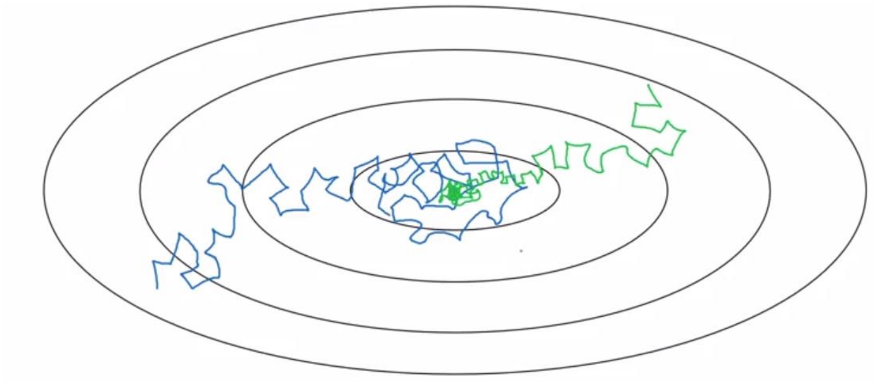
Learning rate decay

- Mini-batch gradient descent



it won't exactly converge

- Slowly reduce learning rate over the time



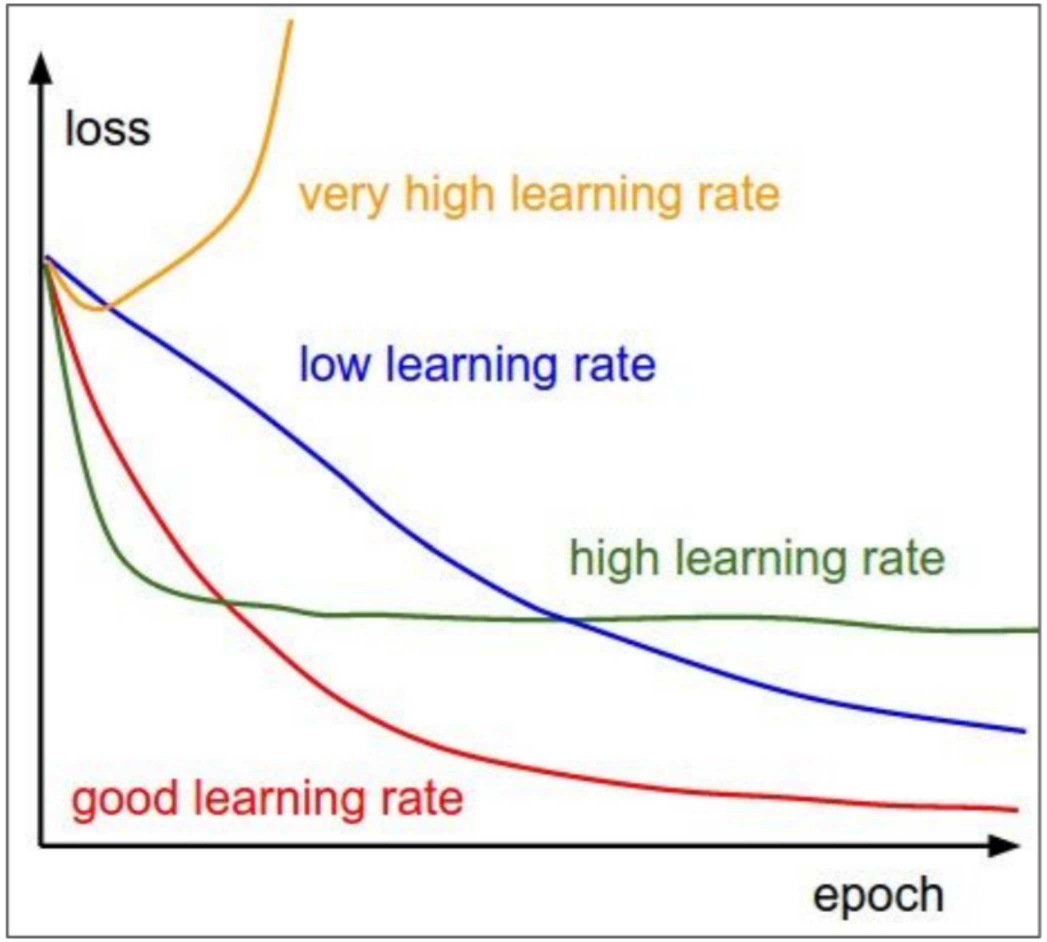
oscillating in a tighter region around this minimum

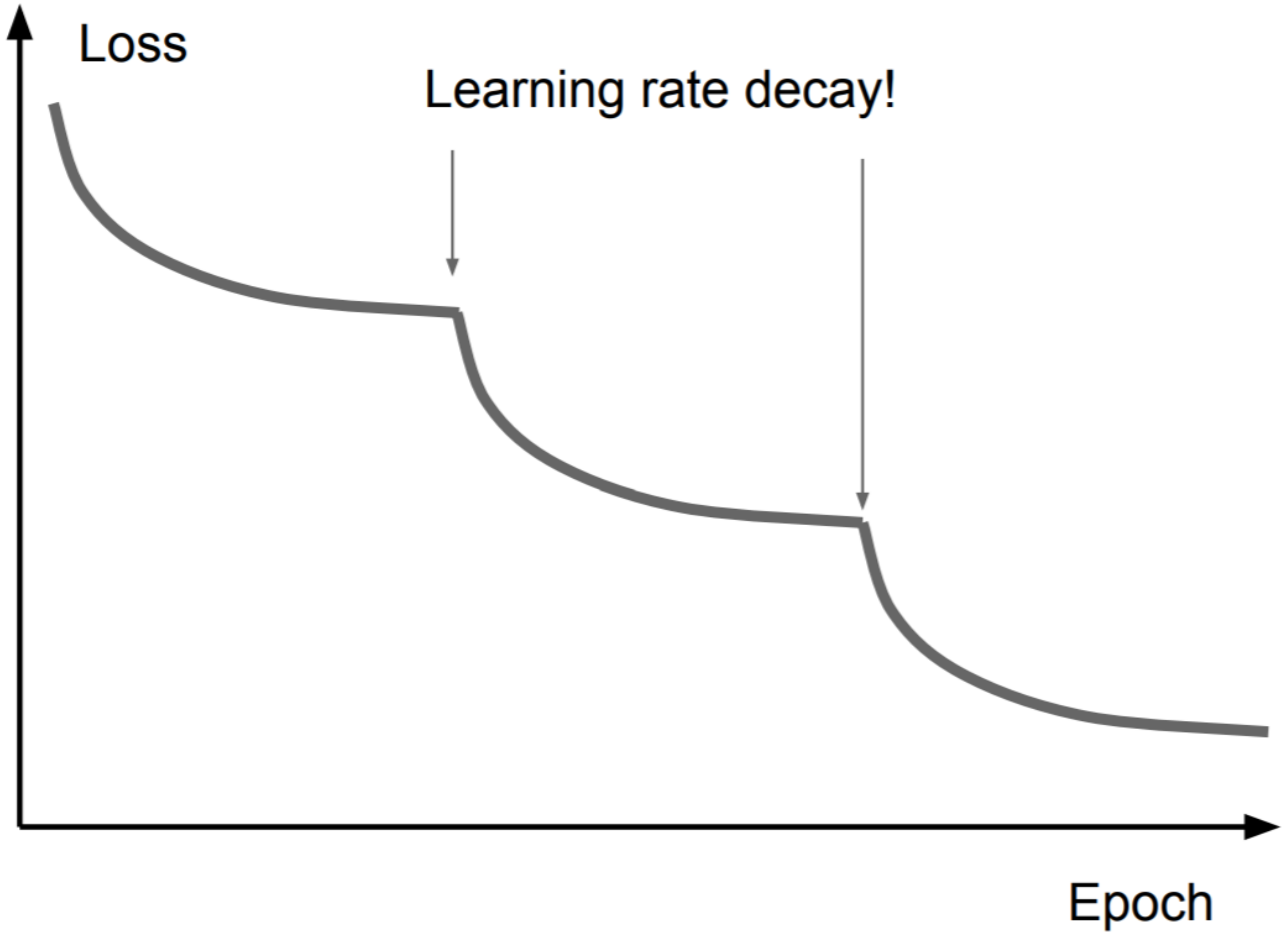
[Andrew Ng, Deep Learning Specialization, 2017]

Learning rate decay over time

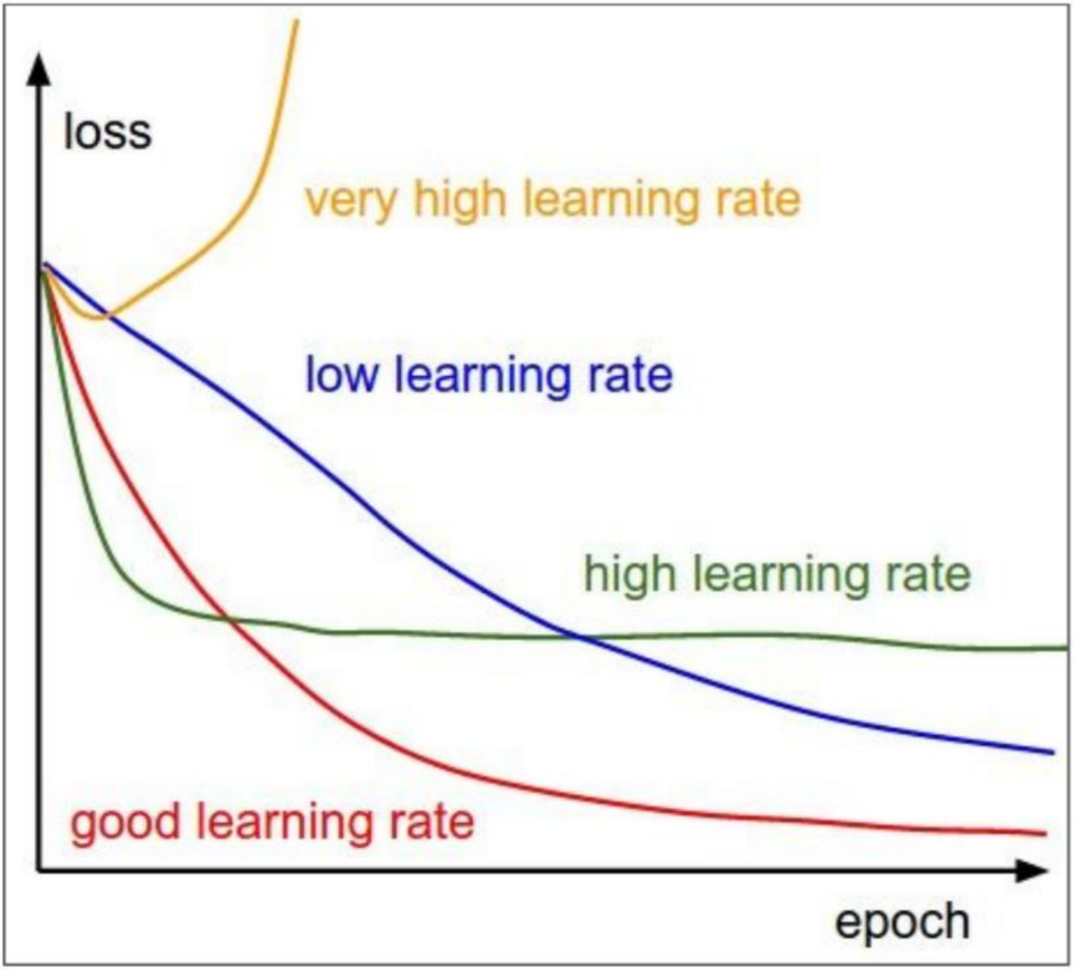
- Step decay: decay learning rate by half every few epochs.
- Exponential decay:
  - $\alpha = \alpha_0 e^{-\text{decay\_rate} \times \text{epoch\_num}}$ $\left(\alpha = \alpha_0 0.95^{\text{epoch\_num}}\right)$
- 1/t decay:
  - $\alpha = \frac{\alpha_0}{1 + \text{decay\_rate} \times \text{epoch\_num}}$

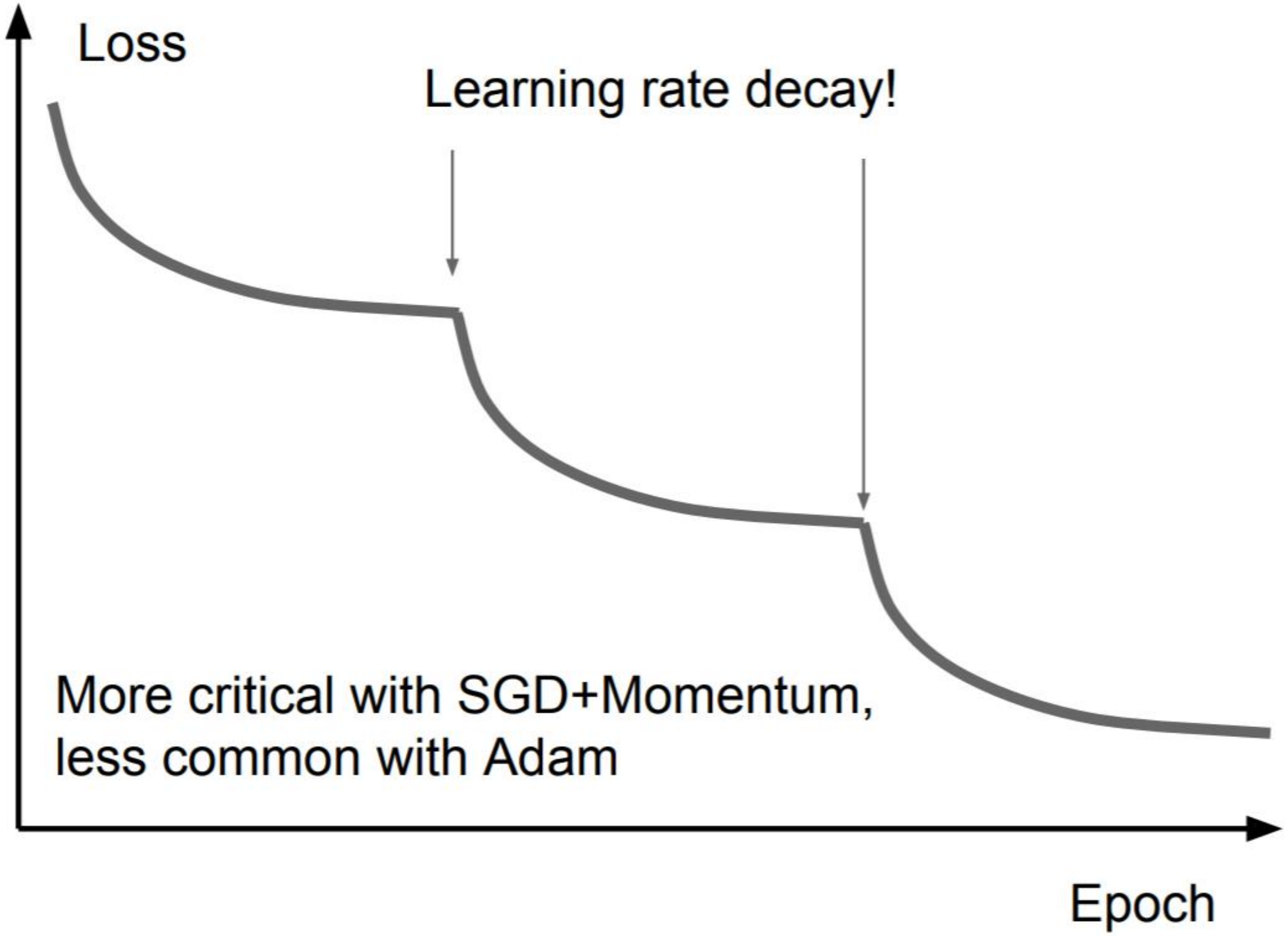
Learning rate





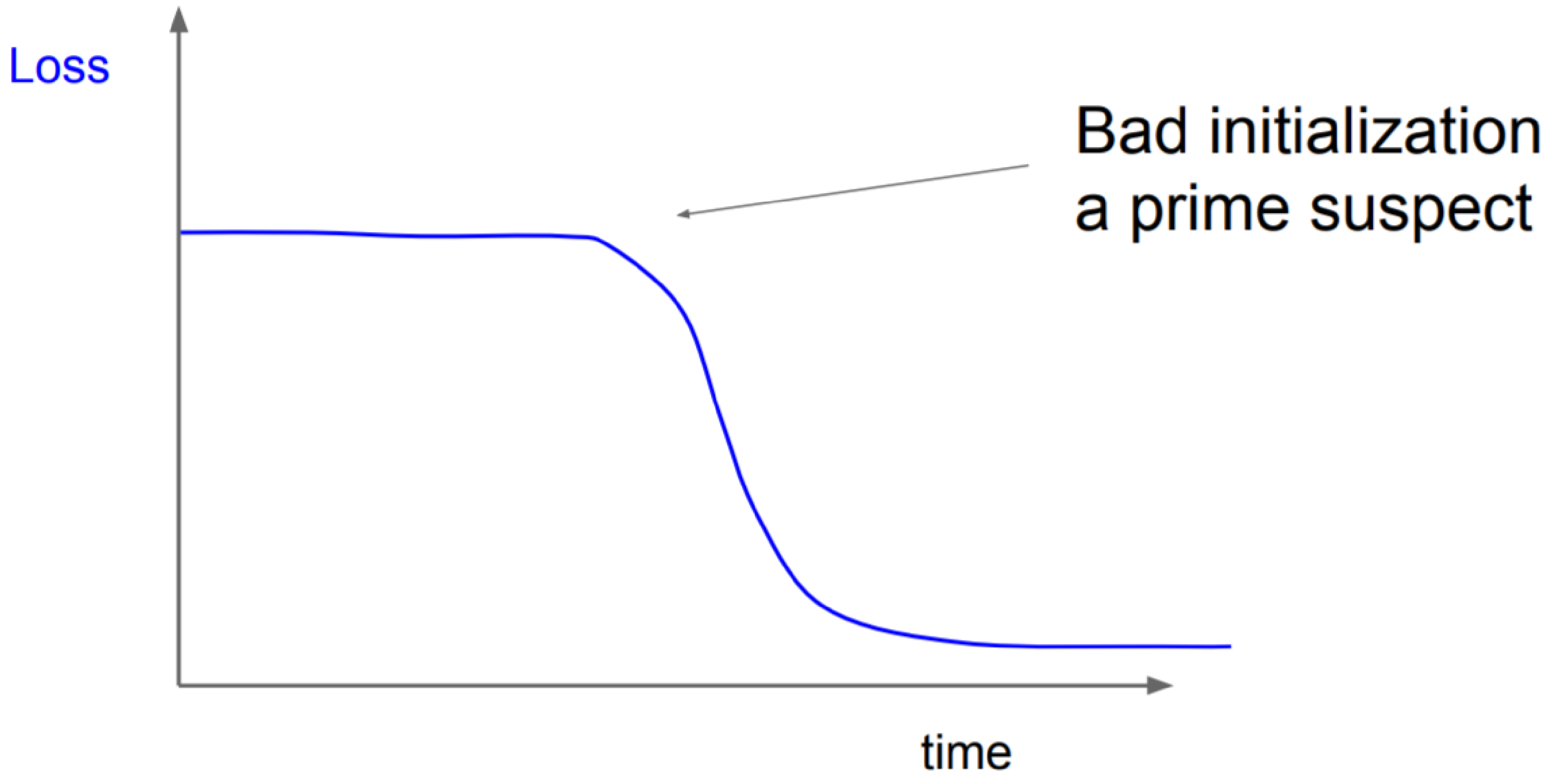
Learning rate



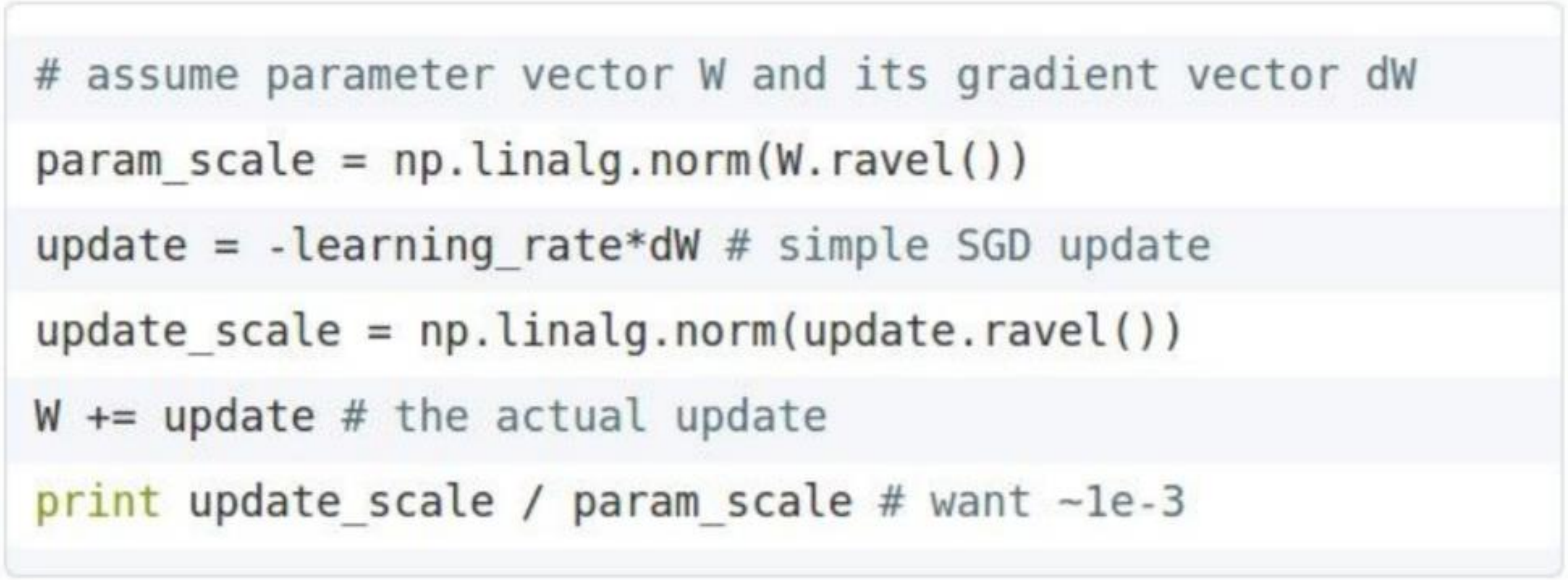


More critical with SGD+Momentum,
less common with Adam

Monitoring loss function during iterations



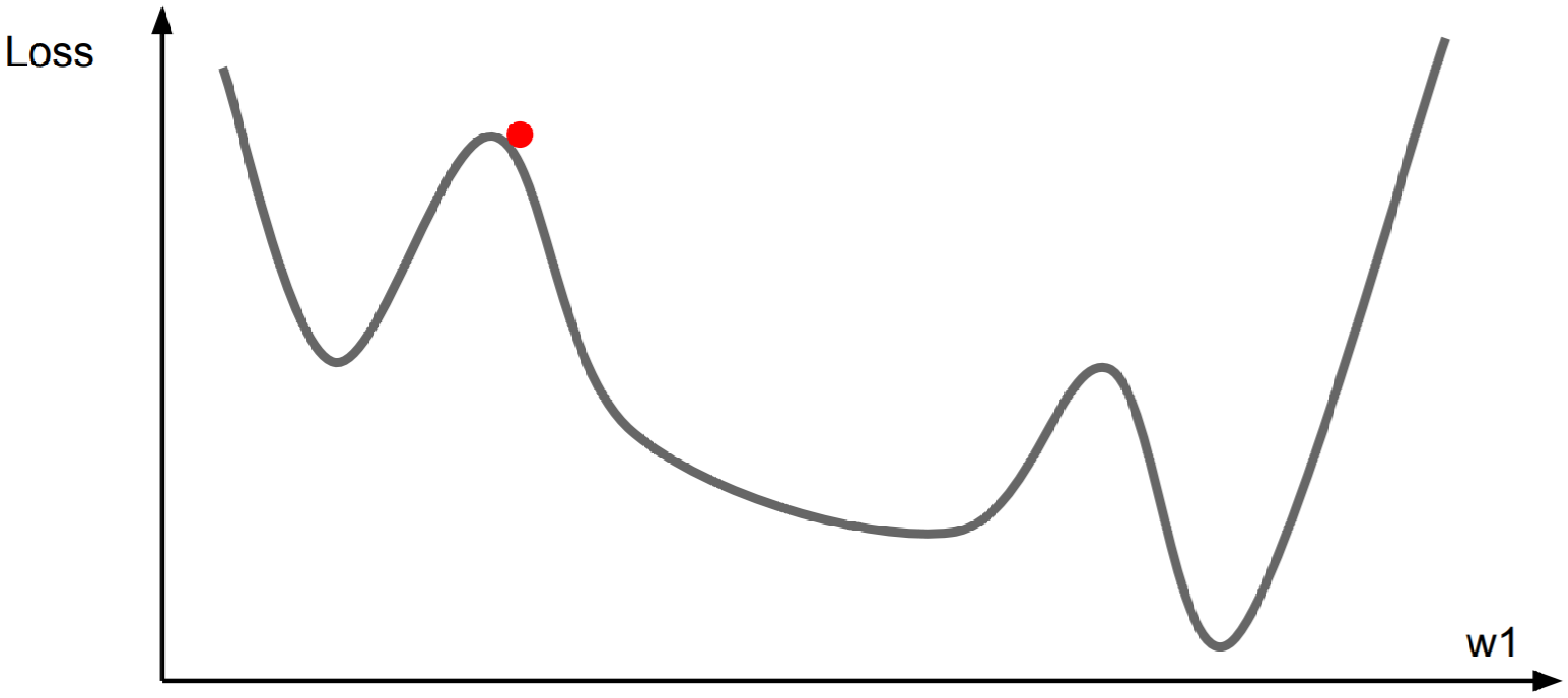
Track the ratio of weight updates / weight magnitudes:



```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

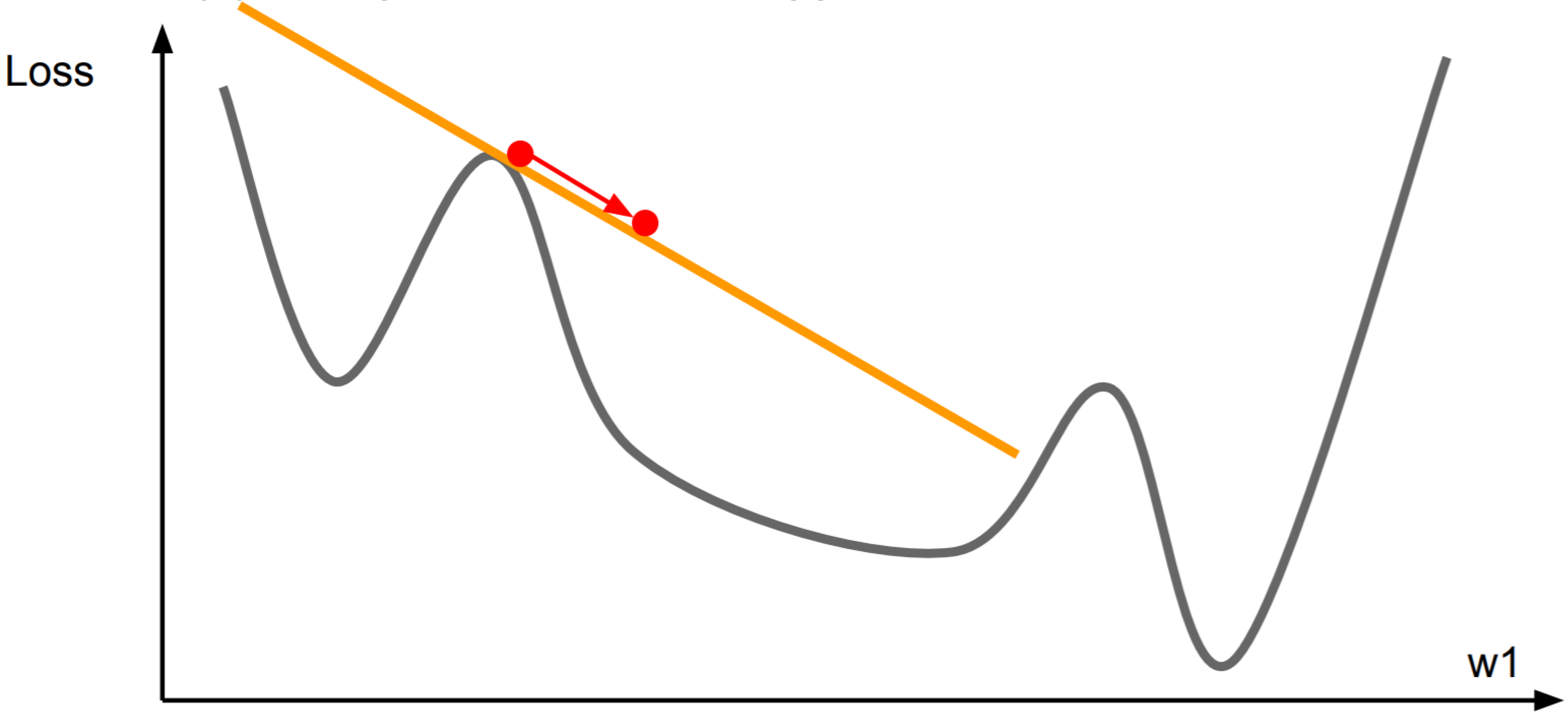
ratio between the updates and values: e.g., 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so

First-order optimization



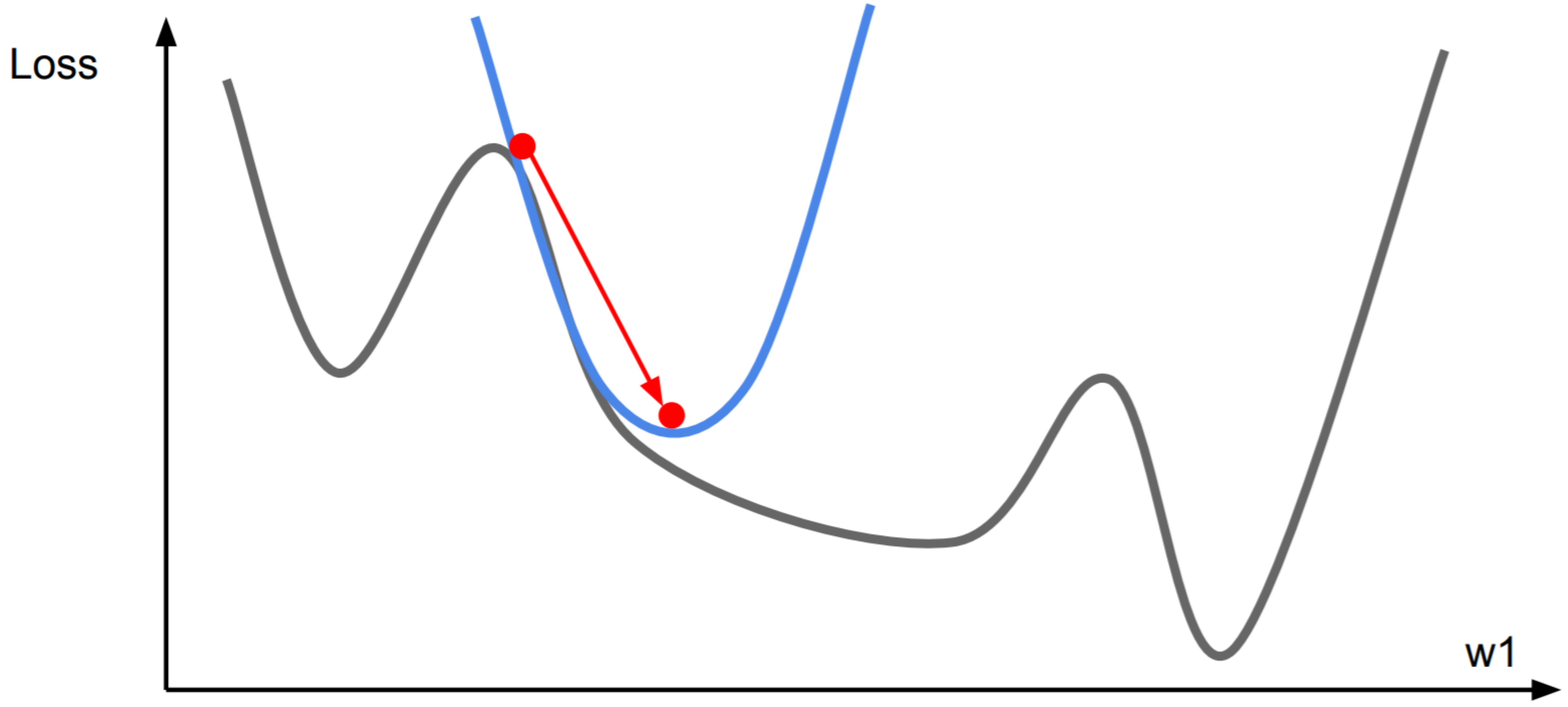
First-order optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation



Second-order optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation



Second-order optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

What is nice about this update?

No hyperparameters!
No learning rate!

Second-order optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Why is this bad for deep learning?

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Second-order optimization: L-BFGS

- Quasi-Newton methods (BGFS most popular):
  - instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).
- L-BFGS (Limited memory BFGS):
  - Does not form/store the full inverse Hessian.
- L-BFGS usually works very well in full batch, deterministic mode
  - i.e. work very well when you have a single, deterministic cost function
- But does not transfer very well to mini-batch setting.
  - Gives bad results.
  - Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

In practice

- Adam is a good default choice in most cases

- If you can afford to do full batch updates then try out L-BFGS (and don’t forget to disable all sources of noise)

Before going to the generalization in the next lecture

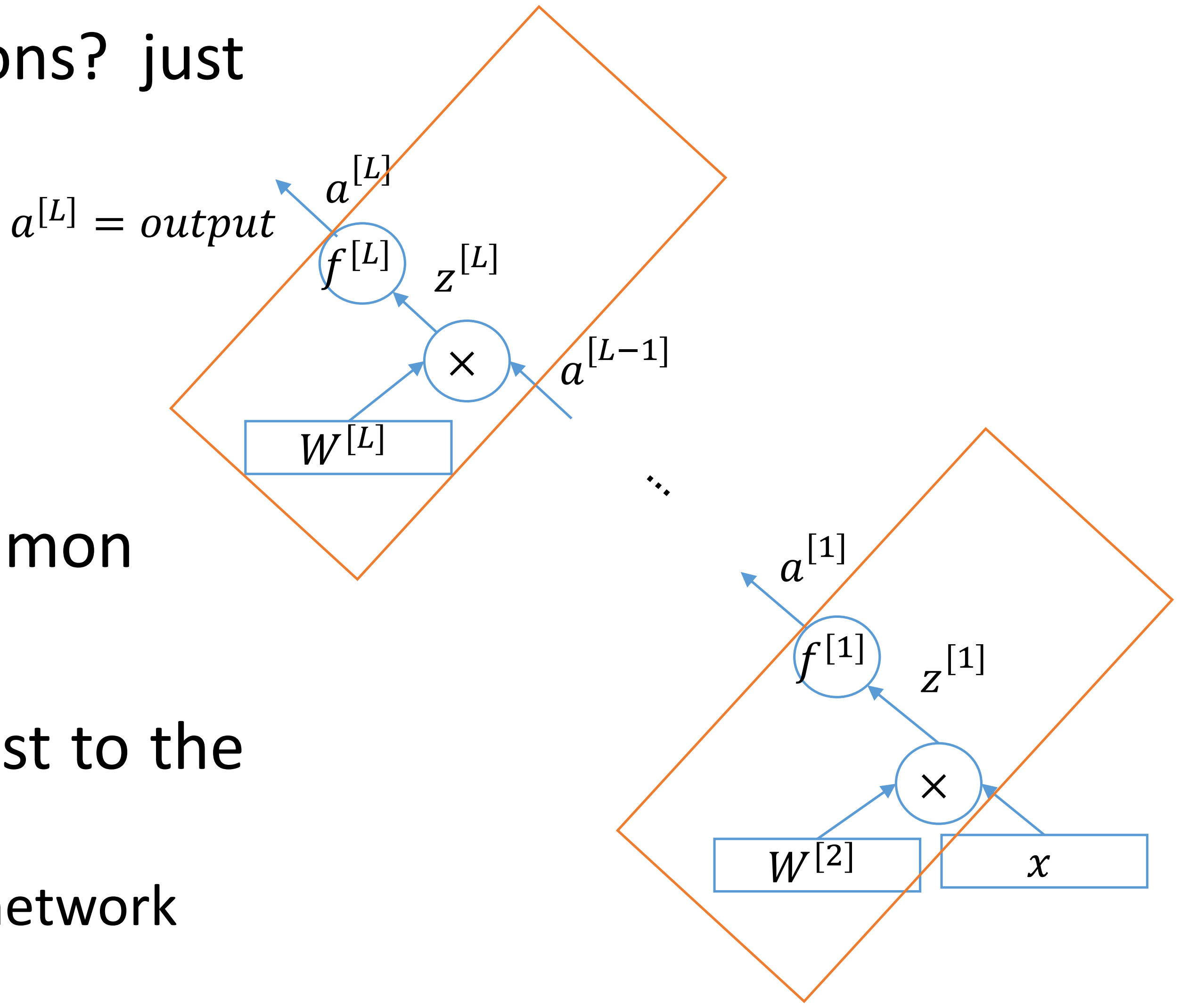
We now see an important technique that helps to speed up training while also acts as a form of regularization

Recall: Normalizing inputs

- On the training set compute mean of each input (feature)
  - $\mu_i = \frac{\sum_{n=1}^{N} x_i^{(n)}}{N}$
  - $\sigma_i^2 = \frac{\sum_{n=1}^{N}\left(x_i^{(n)} - \mu_i\right)^2}{N}$
- **Remove mean**: from each of the input mean of the corresponding input
  - $x_i \leftarrow x_i - \mu_i$
- **Normalize variance**:
  - $x_i \leftarrow \frac{x_i}{\sigma_i}$

Batch normalization

- "you want unit gaussian activations? just make them so."
- Can we normalize $a^{[l]}$ too?
- Normalizing $z^{[l]}$ is much more common
- Makes neural networks more robust to the range of hyperparameters
  - Much more easily train a very deep network



Batch normalization

[Ioffe and Szegedy, 2015]

- Consider a batch of activations at some layer.
- To make each dimension unit gaussian, apply:

$$\mu^{[l]} = \sum_i \frac{z^{(i)[l]}}{b}$$

$$\sigma^{2^{[l]}} = \sum_i \frac{\left(z^{(i)[l]} - \mu^{[l]}\right)^2}{b}$$

$$z_{norm}^{(i)[l]} = \frac{z^{(i)[l]} - \mu^{[l]}}{\sqrt{\sigma^{2^{[l]}} + \epsilon}}$$

Batch normalization

- To make each dimension unit gaussian, apply:

$$\mu^{[l]} = \sum_i \frac{z^{(i)[l]}}{b}$$

$$\sigma^{2[l]} = \sum_{i=1}^{b} \frac{\left(z^{(i)[l]} - \mu^{[l]}\right)^2}{b}$$

$$z_{norm}^{(i)[l]} = \frac{z^{(i)[l]} - \mu^{[l]}}{\sqrt{\sigma^{2[l]} + \epsilon}}$$

- Problem: do we necessarily want a unit gaussian input to an activation layer?

$$\hat{z}^{(n)[l]} = \gamma^{[l]} z_{norm}^{(i)[l]} + \beta^{[l]}$$

$\gamma^{[l]}$ and $\beta^{[l]}$ are learnable parameters of the model

Batch normalization

[Ioffe and Szegedy, 2015]

$$z_{norm}^{(i)[l]} = \frac{z^{(i)[l]} - \mu^{[l]}}{\sqrt{\sigma^{2[l]} + \epsilon}}$$

In the linear area of tanh

- Allow the network to squash the range if it wants to:

$$\hat{z}^{(i)[l]} = \gamma^{[l]} z_{norm}^{(i)[l]} + \beta^{[l]}$$

The network can learn to recover the identity:

$$\gamma^{[l]} = \sqrt{\sigma^{2[l]} + \epsilon}$$
$$\beta^{[l]} = \mu^{[l]}$$
$$\Rightarrow \hat{z}^{(i)[l]} = z^{(i)[l]}$$

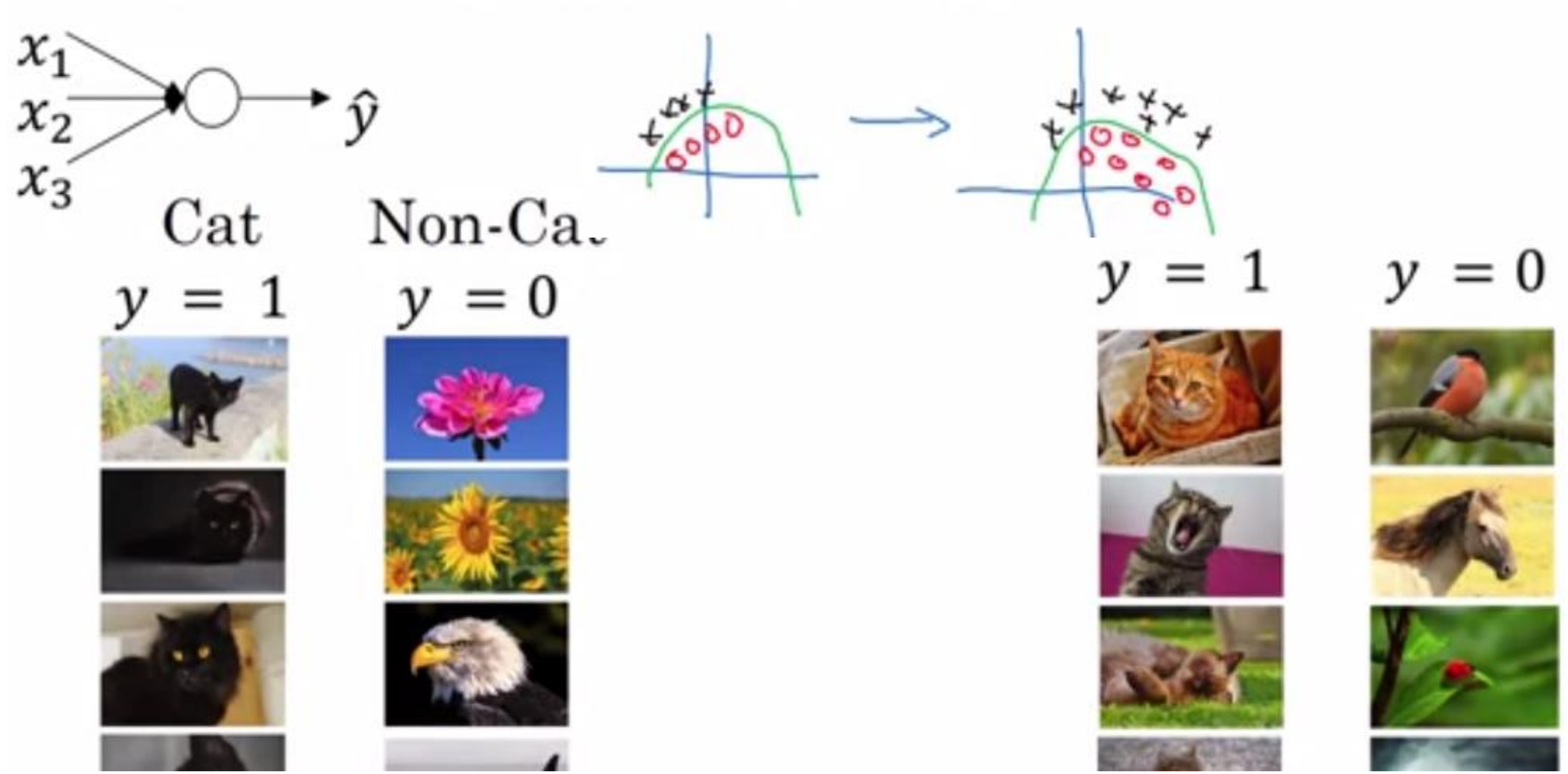
Batch normalization at test time

- At test time BatchNorm layer functions differently:
  - The mean/std are not computed based on the batch.
  - Instead, a single fixed empirical mean of activations during training is used. (e.g. can be estimated during training with running averages)

```
$\mu^{[l]} = 0$
$for\ i=1...m$
    $\mu^{[l]} = \beta\mu^{[l]} + (1-\beta)\frac{\sum_{j=1}^{b} X_j^{\{i\}}}{b}$
```

How does Batch Norm work?

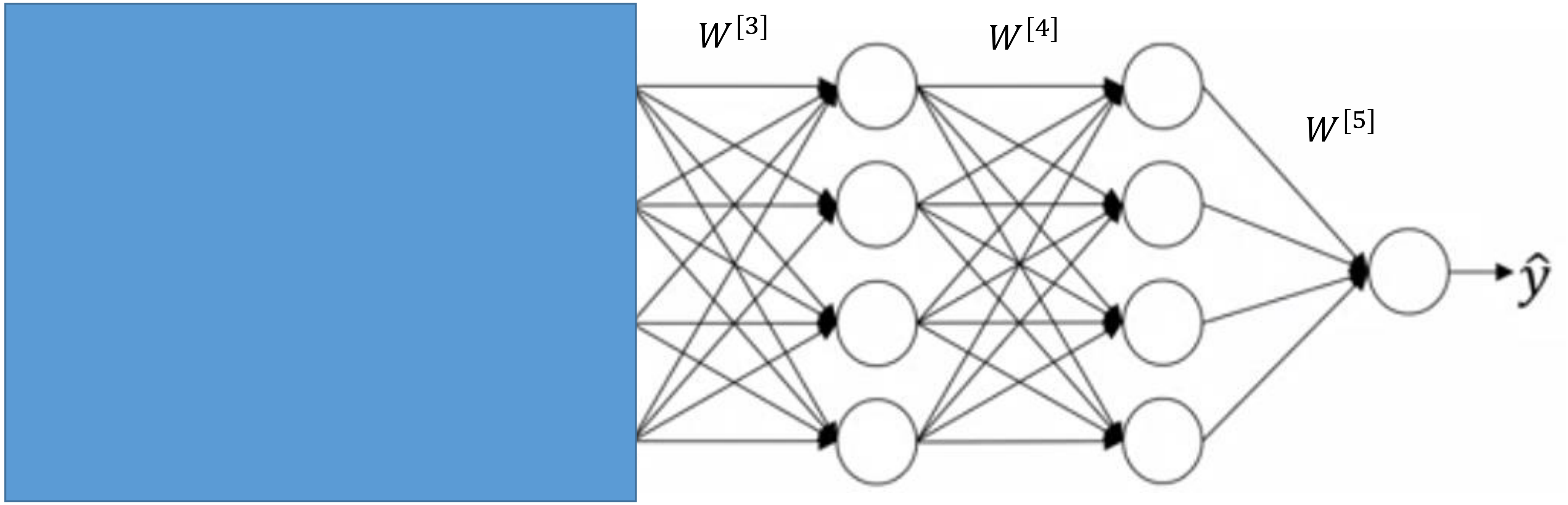
- Learning on shifting input distributions



[Andrew Ng, Deep Learning Specialization, 2017]

Why this is a problem with neural networks?

Limits the amount to which updating weights in earlier layers can affect the distribution of values that the subsequent layers see



The values become more stable to stand later layers on

Batch normalization: summary

- Speedup the learning process
  - Weaken the coupling between the earlier layers and later layers parameters
- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way (as we will see)

Resources

- Deep Learning Book, Chapter 8.
- Please see the following note:
  - http://cs231n.github.io/neural-networks-3/