Artificial Intelligence and Machine Learning

Logistic Regression

Lecture 2: Outline

- Linear Regression (Review)
- Logistic Regression (Classification)
- Optimization

Recap

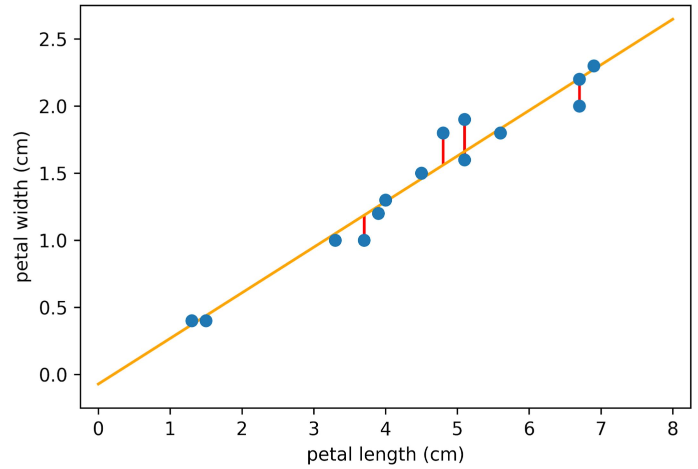
$$\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}, \mathbf{x}_i \in \mathbb{R}^n, y_i \in \mathbb{R}$$

Design your model

- Input scalar linear model (line fitting)
- Fitting polynomials (synthetically designing features from a one-dimensional input)

Design your loss function

- We used mean squared error loss throughout

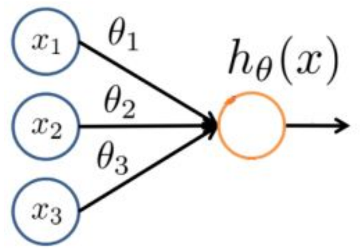


Finding optimal parameter fitting

- Closed form solution to the linear least squares?
- Why is it linear least squares?
- Solution is closed form

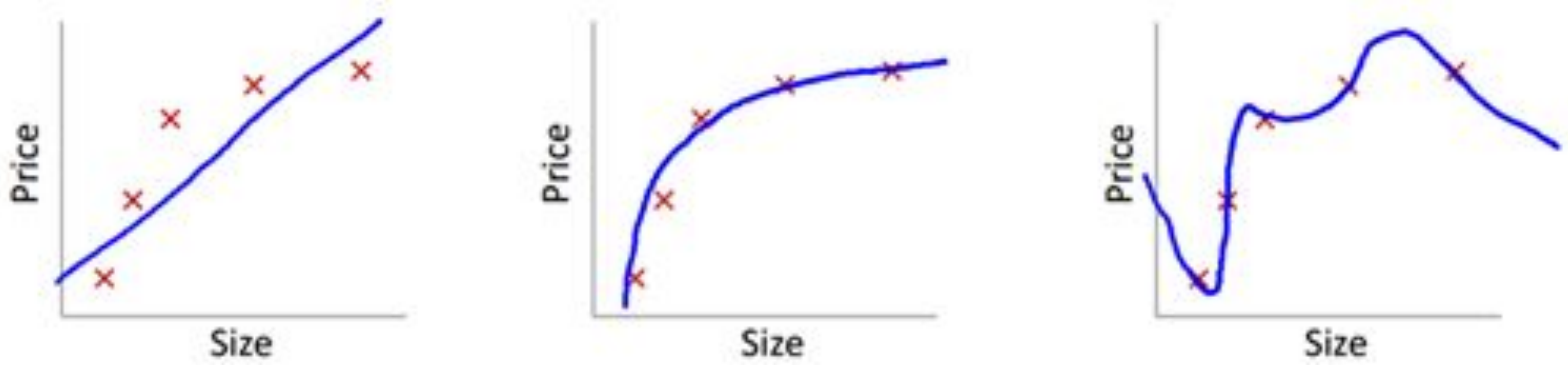
Logistic regression or Classification

Regression VS classification

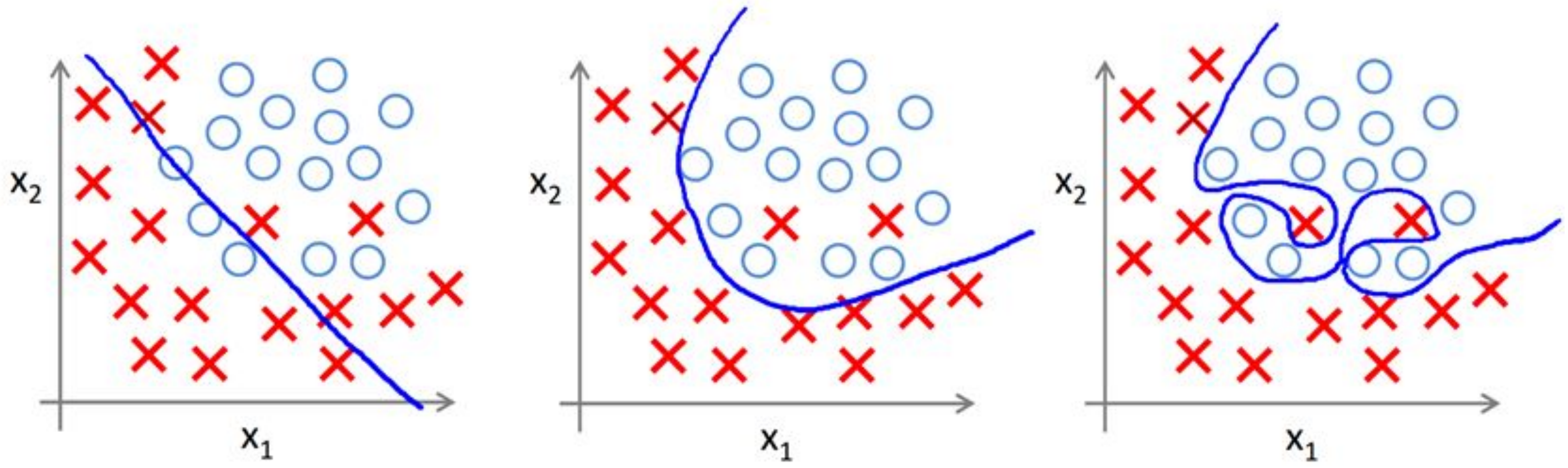


$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

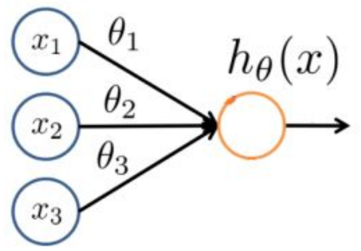
- Regression (linear and polynomial): for prediction



- Classification:



Regression VS classification



$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

Income prediction -> regression

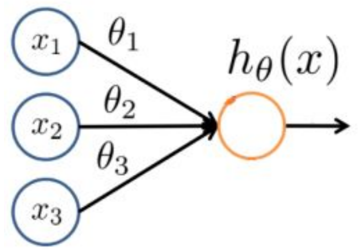
Male or Female -> classification

House price -> regression

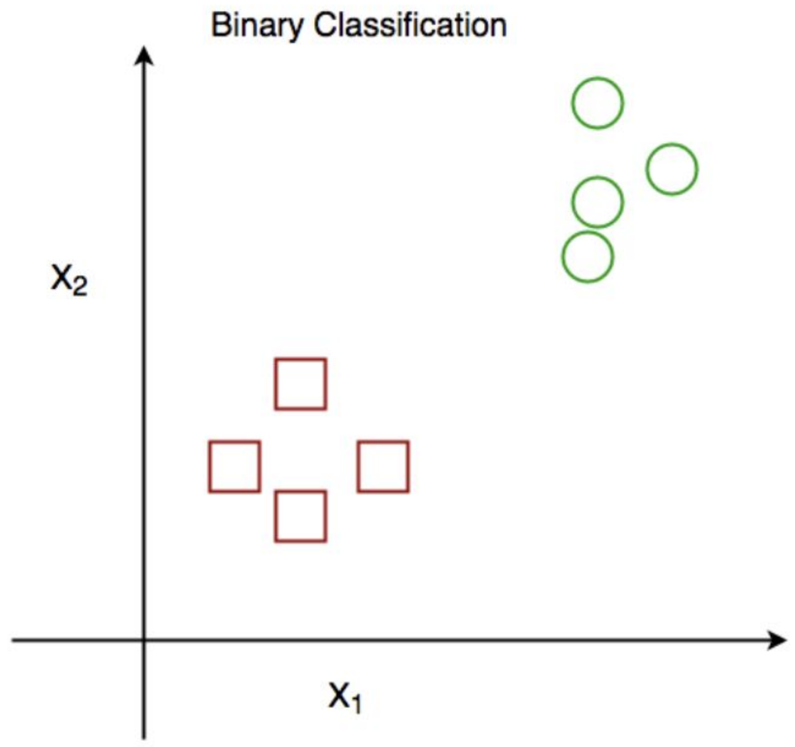
Spam detection -> classification

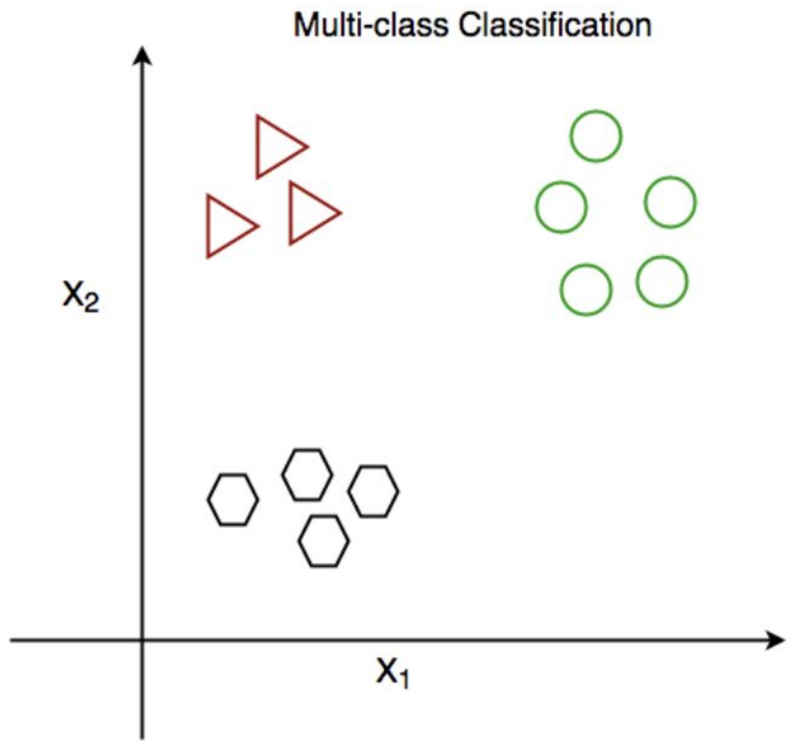
Image recognition -> classification

Classification



$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

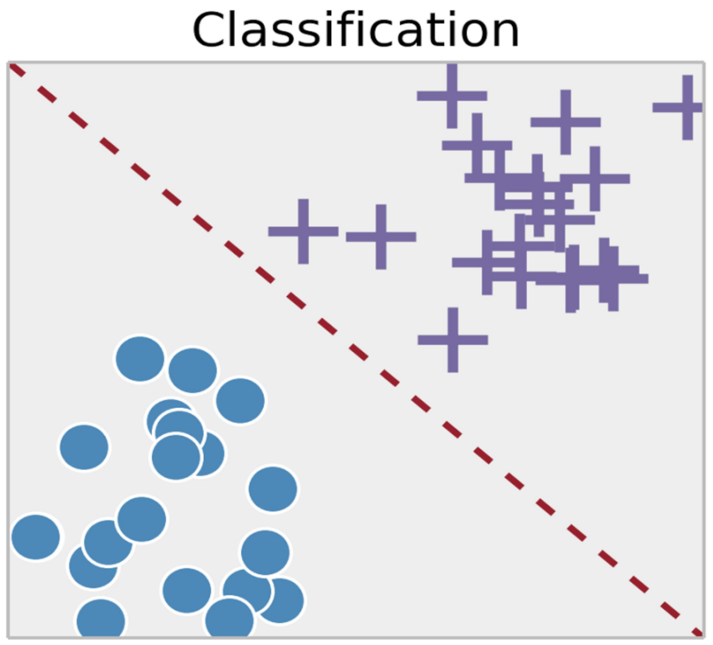




Logistic Regression

- Regular vs Fraudulent transaction
- Spam vs Non-spam emails
- Benign vs Malignant tumors
- Rising vs Falling stocks

$$\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}, \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{0, 1\}$$



Logistic Regression vs linear regression

Linear Regression	Logistic Regression
For Regression	For Classification
We predict the target value for any input value	We predict the probability that the input value belongs to the specific target
Target: Real Values, continuous values	Target: Discrete values
Graph: Straight Line	Graph: S-curve

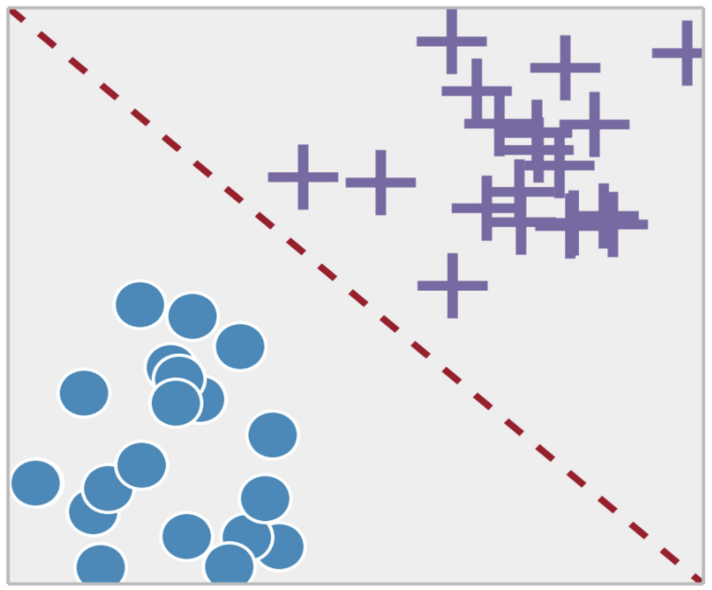
Logistic Regression

Despite the name, logistic regression is a

classification algorithm

Logistic Regression is a linear model with a "special function" that helps us use this linear model for classification

Classification



$$\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}, \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{0,1\}$$

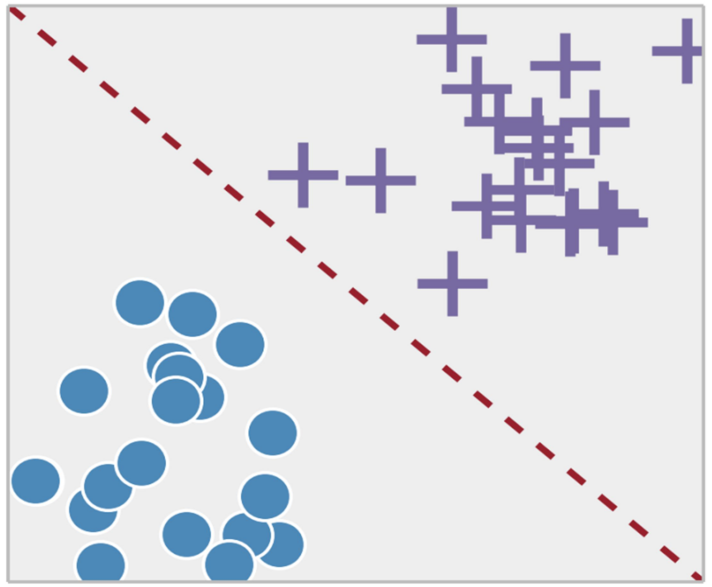
Linear Model in Disguise

$$\hat{y} = \mathbf{w}^T \mathbf{x}$$

$$\mathbf{w} = [w_0, w_1, \cdots, w_m]^T$$

$$\mathbf{x} = [1, x^1, \cdots, x^m]^T$$

Classification



$$\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}, \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{0, 1\}$$

Challenge:

$$\hat{y} = \mathbf{w}^T \mathbf{x}$$

- The above equation predicts continuous outputs.
- Unbounded output: There is no natural constraint on the output, and prediction could be any real number
- Intuitively, it also doesn’t make sense for $\hat{y}$ to take values larger than 1 or smaller than 0 when we know that y ∈ {0, 1}.

Solution

- In logistic regression, we define the problem as follows:
  - Instead of just predicting the class, give the probability of the instance being that class

$$\hat{y} = p(y \mid \boldsymbol{x})$$

- Thus we need a function that transforms the output into a probability distribution.

$$\hat{y} = \sigma\left(\mathbf{w}^T \mathbf{x}\right)$$

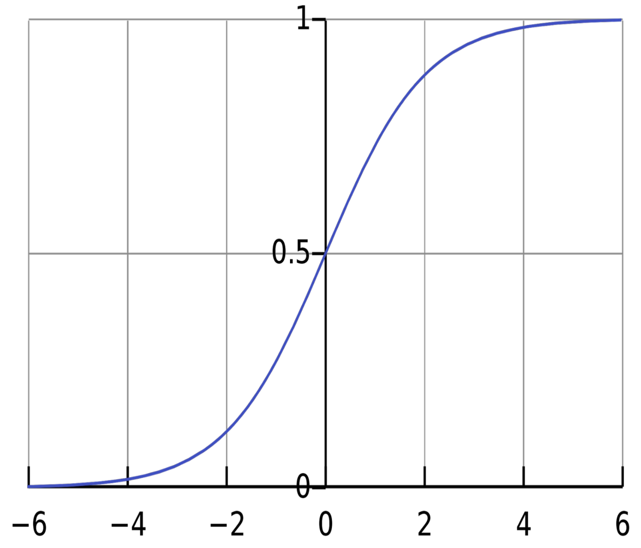
Sigmoid Function

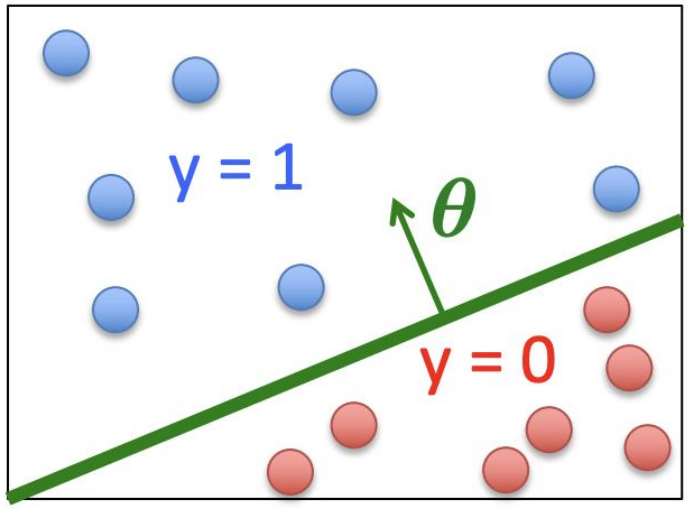
Widely used in classification

$$\sigma(z) = \frac{1}{1 + exp(-z)}$$

$$\lim_{z \to \infty} \sigma(z) = 1$$

$$\lim_{z \to -\infty} \sigma(z) = 0$$

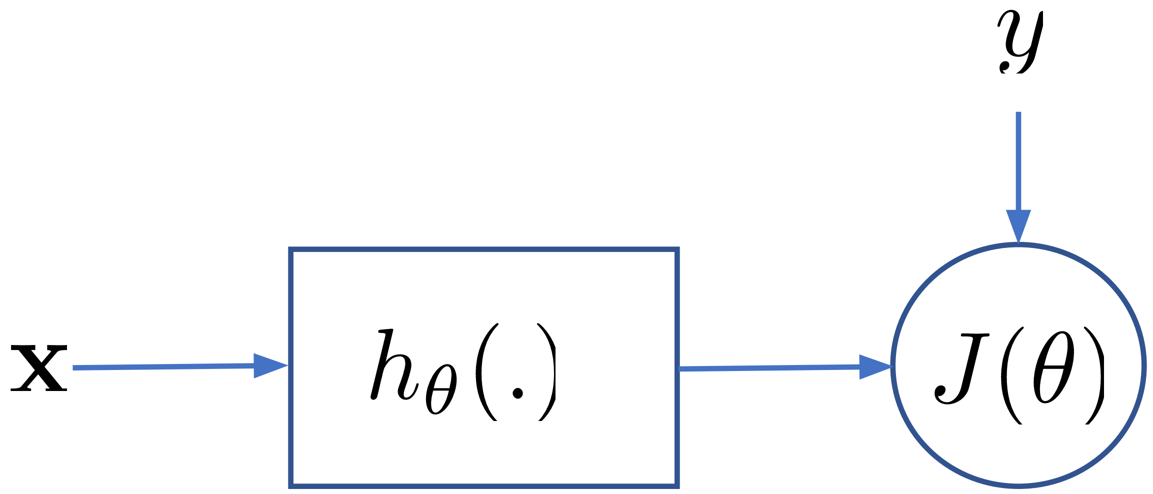




- Assume a threshold:
  - Predict y=1 if $\sigma(z)$ >= 0.5
  - Predict y=0 if $\sigma(z)$ < 0.5

Cost Function

- We want to minimize the discrepancy between our model hypothesis and the observed label.


$$\begin{tikzcd} & & y \arrow[d] \\ \mathbf{x} \arrow[r] & \boxed{h_\theta(.)} \arrow[r] & J(\theta) \end{tikzcd}$$

What type of loss to use?

MSE?

$$J = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- Not the best when it comes to classification
- Leads to suboptimal results
- Not ideal for probability output

Binary Cross Entropy Loss

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \mathrm{compare}(y_i, \sigma(\mathbf{w}^T\mathbf{x}_i))$$

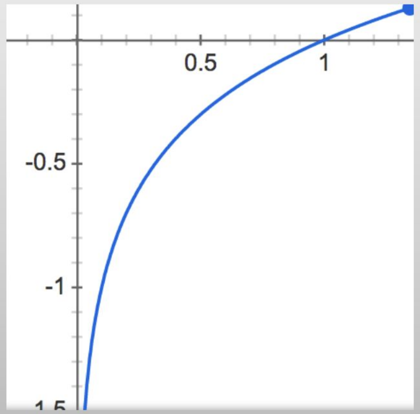
$$J(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\sigma(\mathbf{w}^T\mathbf{x}_i)) + (1 - y_i)\log(1 - \sigma(\mathbf{w}^T\mathbf{x}_i))$$

Intuition of Cost Function

- Cost of a single instance:

$$\text{cost}\left(\sigma(\mathbf{w}^T\mathbf{x}), y\right) = \begin{cases} -\log(\sigma(\mathbf{w}^T\mathbf{x})) & \text{if } y = 1 \\ -\log(1 - \sigma(\mathbf{w}^T\mathbf{x})) & \text{if } y = 0 \end{cases}$$

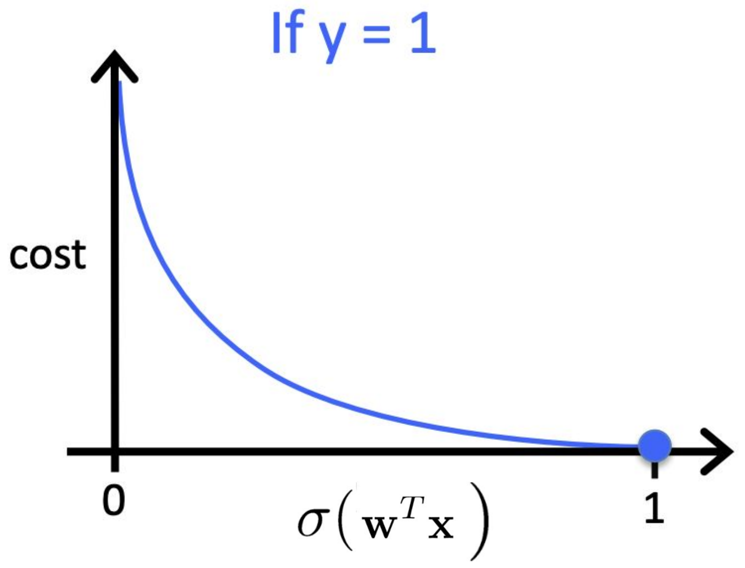
Aside: Recall the plot of log(z)



Intuition of Cost Function

- Cost of a single instance:

$$\text{cost}\,(\sigma(\mathrm{w}^T\mathbf{x}), y) = \begin{cases} \boxed{-\log(\sigma(\mathrm{w}^T\mathbf{x})) \quad \text{if } y = 1} \\ -\log(1 - \sigma(\mathrm{w}^T\mathbf{x})) \quad \text{if } y = 0 \end{cases}$$



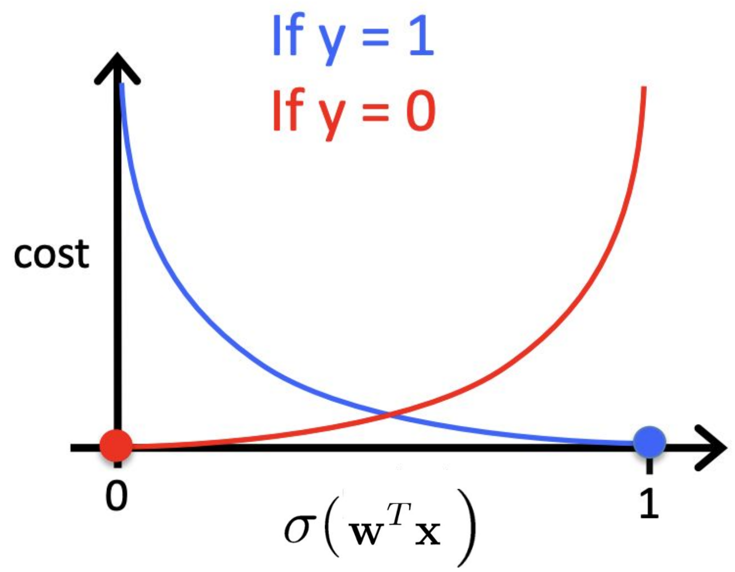
If y = 1

- Cost = 0 if prediction is correct
- As $\sigma(\mathrm{w}^T\mathbf{x}) \to 0, \text{cost} \to \infty$
- Captures intuition that larger mistakes should get larger penalties
  - e.g., predict $\sigma(\mathrm{w}^T\mathbf{x}) = 0$, but y = 1

Intuition of Cost Function

- Cost of a single instance:

$$\text{cost}\left(\sigma(\mathbf{w}^T\mathbf{x}), y\right) = \begin{cases} \boxed{-\log(\sigma(\mathbf{w}^T\mathbf{x})) \quad \text{if } y = 1} \\ \boxed{-\log(1 - \sigma(\mathbf{w}^T\mathbf{x})) \quad \text{if } y = 0} \end{cases}$$



If y = 0

- Cost = 0 if prediction is correct
- As $(1 - \sigma(\mathbf{w}^T\mathbf{x})) \to 0, \text{cost} \to \infty$
- Captures intuition that larger mistakes should get larger penalties

How to find optimal Parameters?

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\sigma(\mathbf{w}^T\mathbf{x}_i)) + (1 - y_i)\log(1 - \sigma(\mathbf{w}^T\mathbf{x}_i))$$

$$\frac{\partial}{\partial \mathrm{w}_j} J(\mathrm{w}) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma(\mathbf{w}^T\mathbf{x}_i) - y_i\right) x_j^{(i)}$$

Just like before, simply take

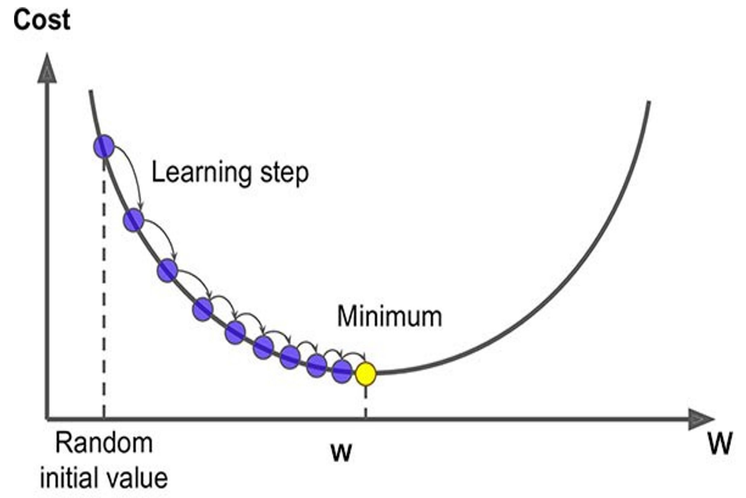
$$\nabla_{\mathbf{w}} J(\mathbf{w}) = 0$$

However, this does not have a nice closed solution.

Gradient Descent

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\sigma(\mathbf{w}^T\mathbf{x}_i)) + (1-y_i)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_i))$$

$$\boxed{\mathbf{w}^{k+1} = \mathbf{w}^{k} - \underbrace{\eta}_{\text{Learning rate}} \nabla_{\mathbf{w}} J(\mathbf{w}^{k})}$$



- We have a linear model for prediction
- For classification, we want to output a probability
- We map the prediction to probabilities with a sigmoid function
- We have a loss function (BCE) to compare models

Gradient Descent Algorithm

Gradient Descent

Want $\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

- Initialize $\boldsymbol{\theta}$
- Repeat until convergence

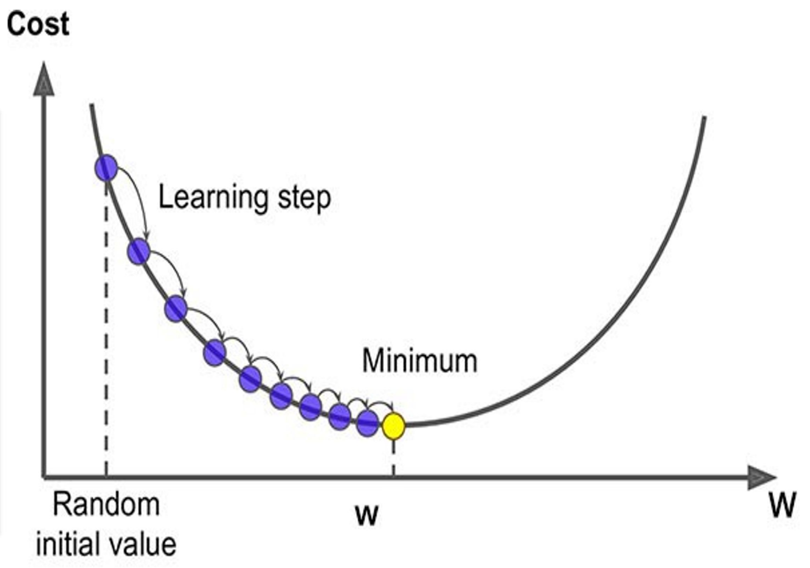
$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta})$$

Learning rate

simultaneous update for j = 0 ... d

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Direction of maximum increase and decrease for a function

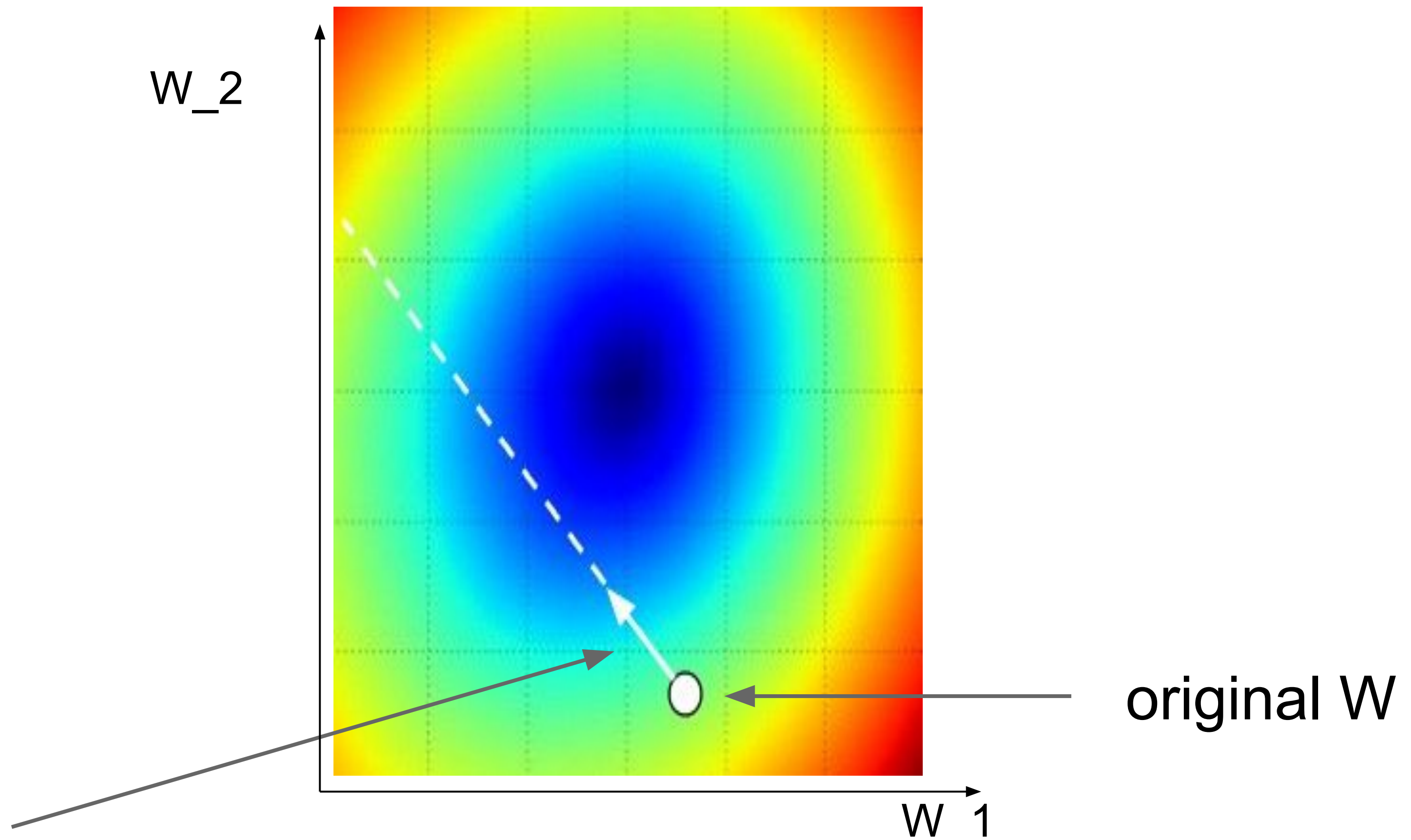
- Gradient direction is the direction of maximum increase for a function
- Negative gradient is the direction of maximum decrease for a function

Gradient Descent

- Initialize $\boldsymbol{\theta}$
- Repeat until convergence

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta})$$

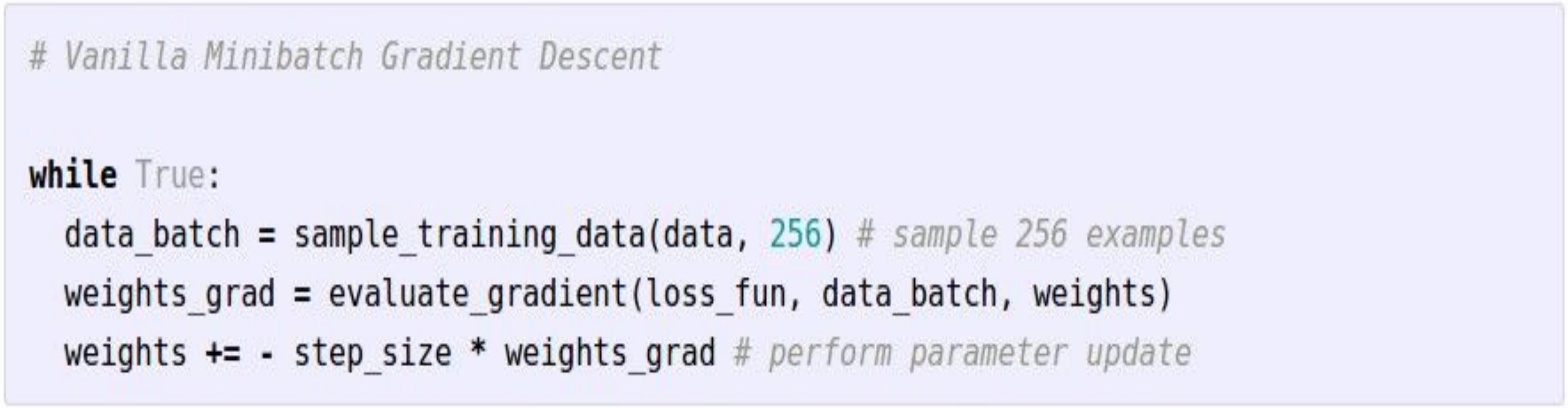
simultaneous update
for j = 0 ... d



negative gradient direction

Mini-batch (Stochastic) Gradient Descent

- only use a small portion of the training set to compute the gradient.



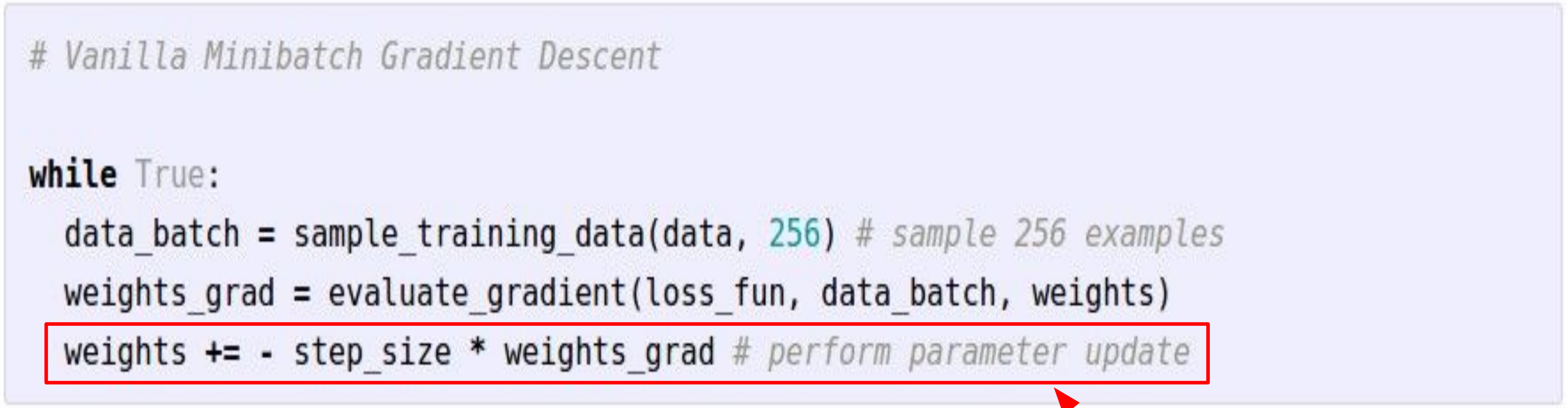
```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

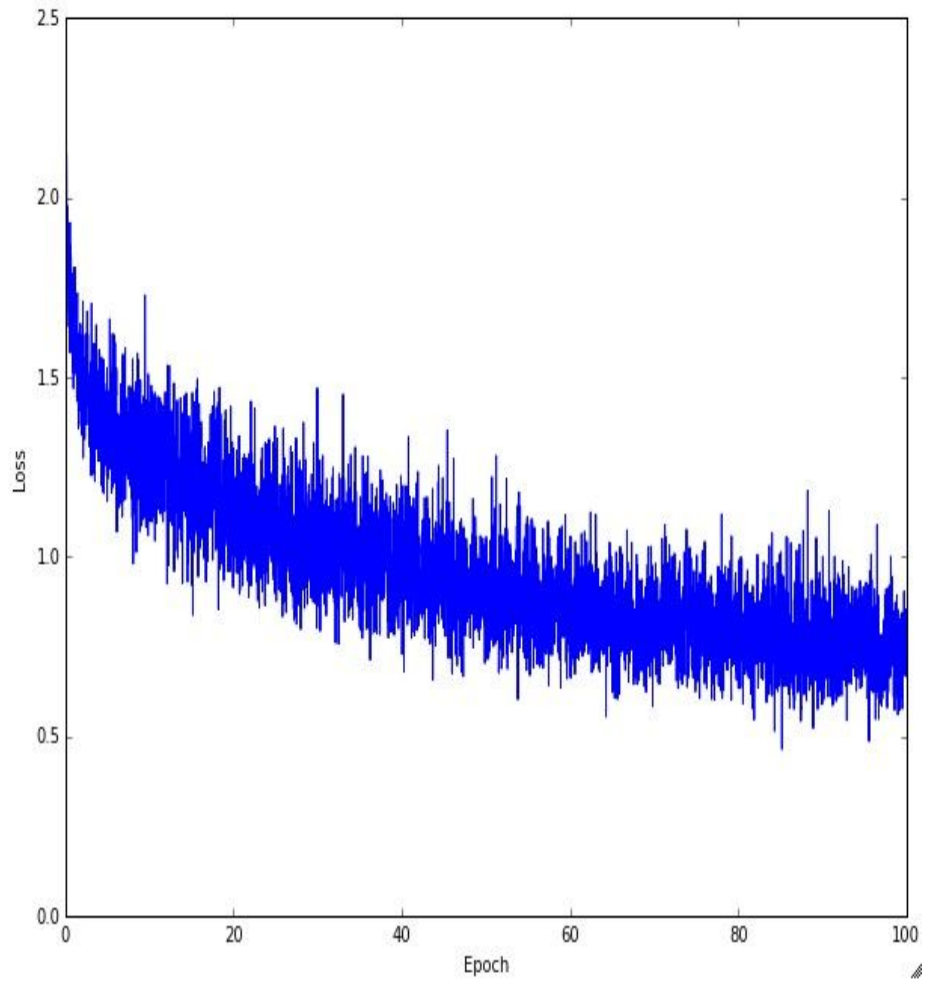


```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

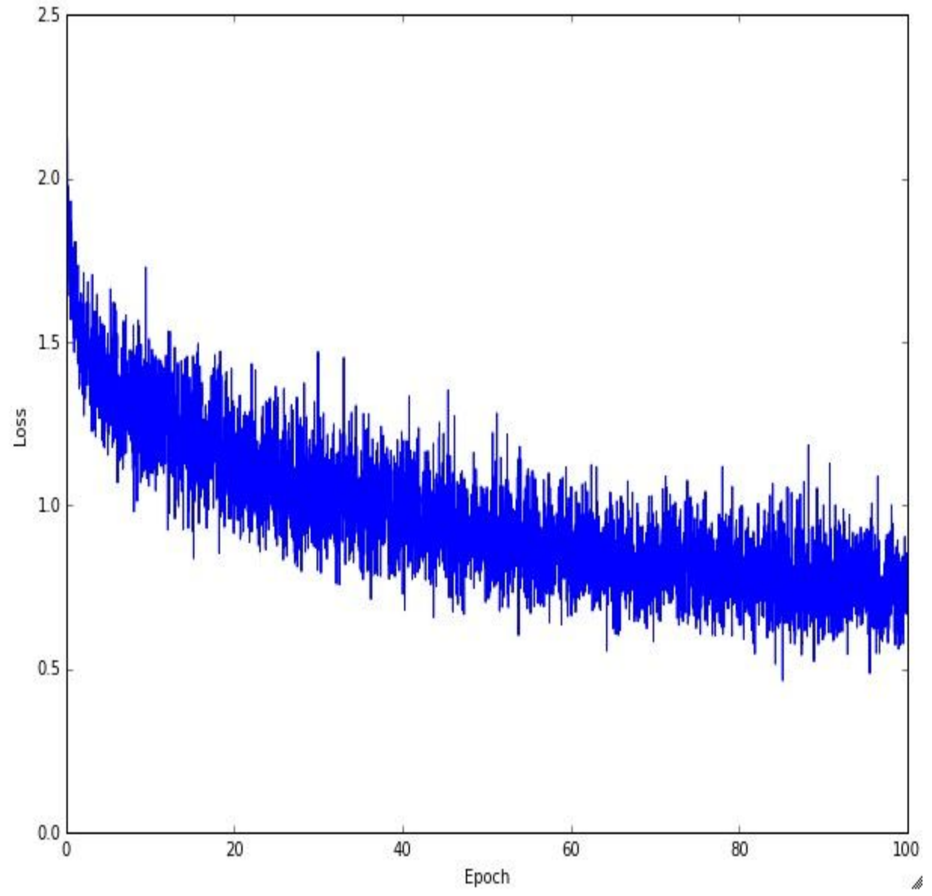
we will look at more fancy update formulas (momentum, Adagrad, RMSProp, Adam, …)

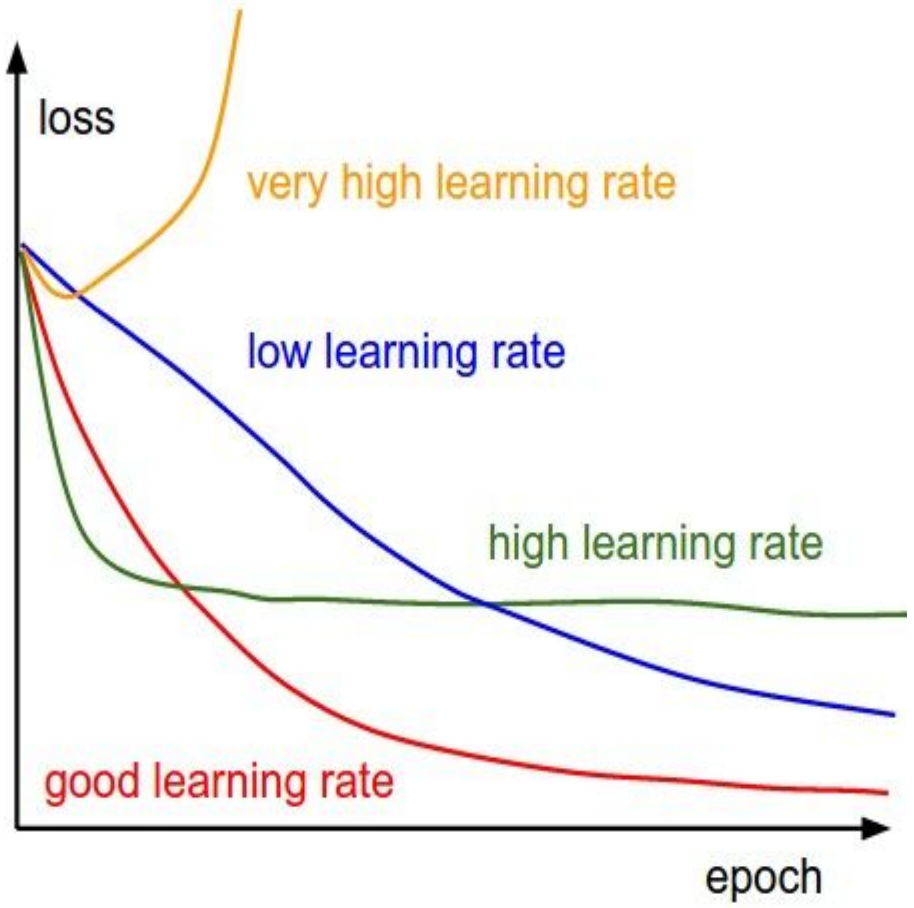


Example of optimization progress while training a neural network.

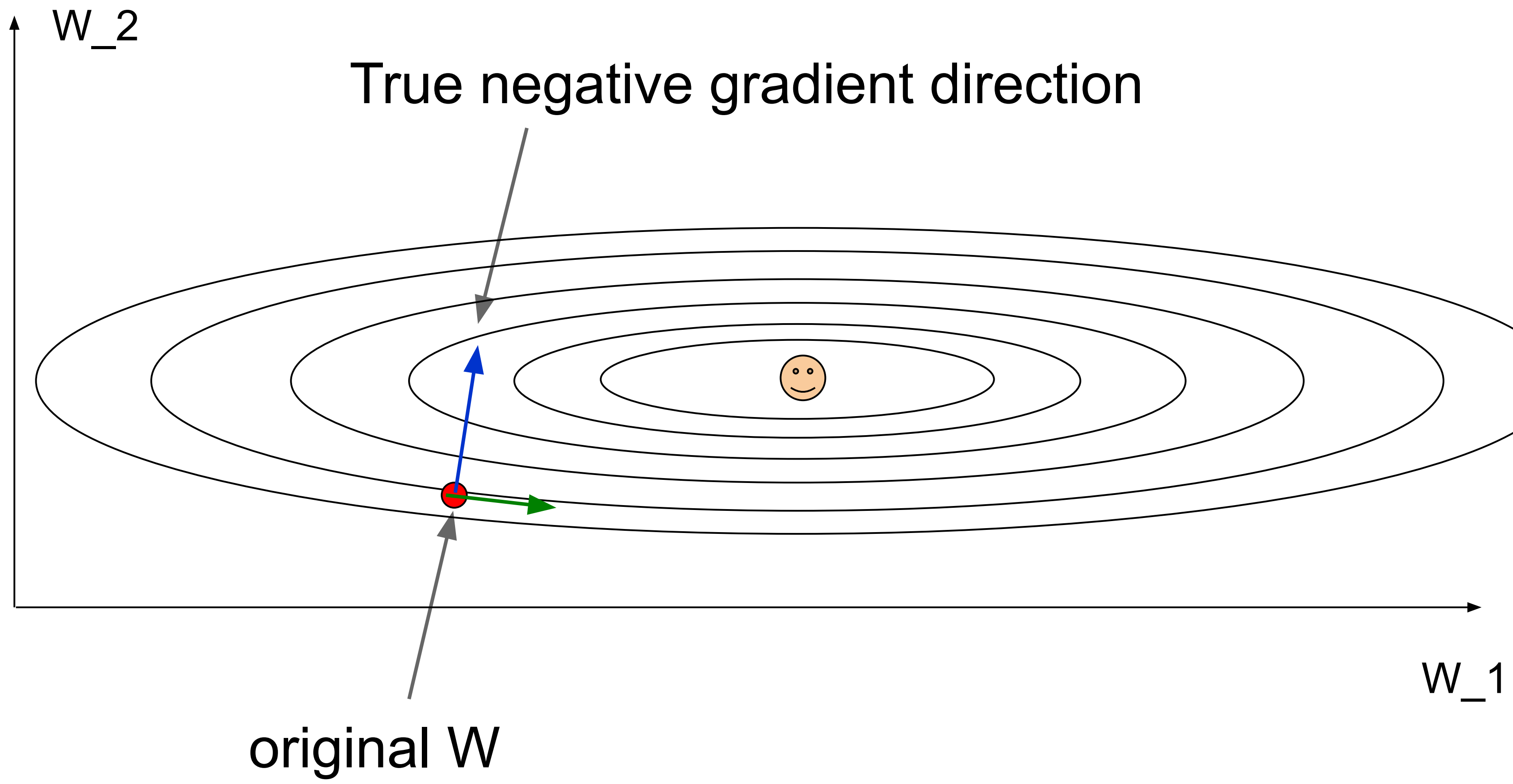
(Loss over mini-batches goes down over time.)

The effects of step size (or “learning rate”)



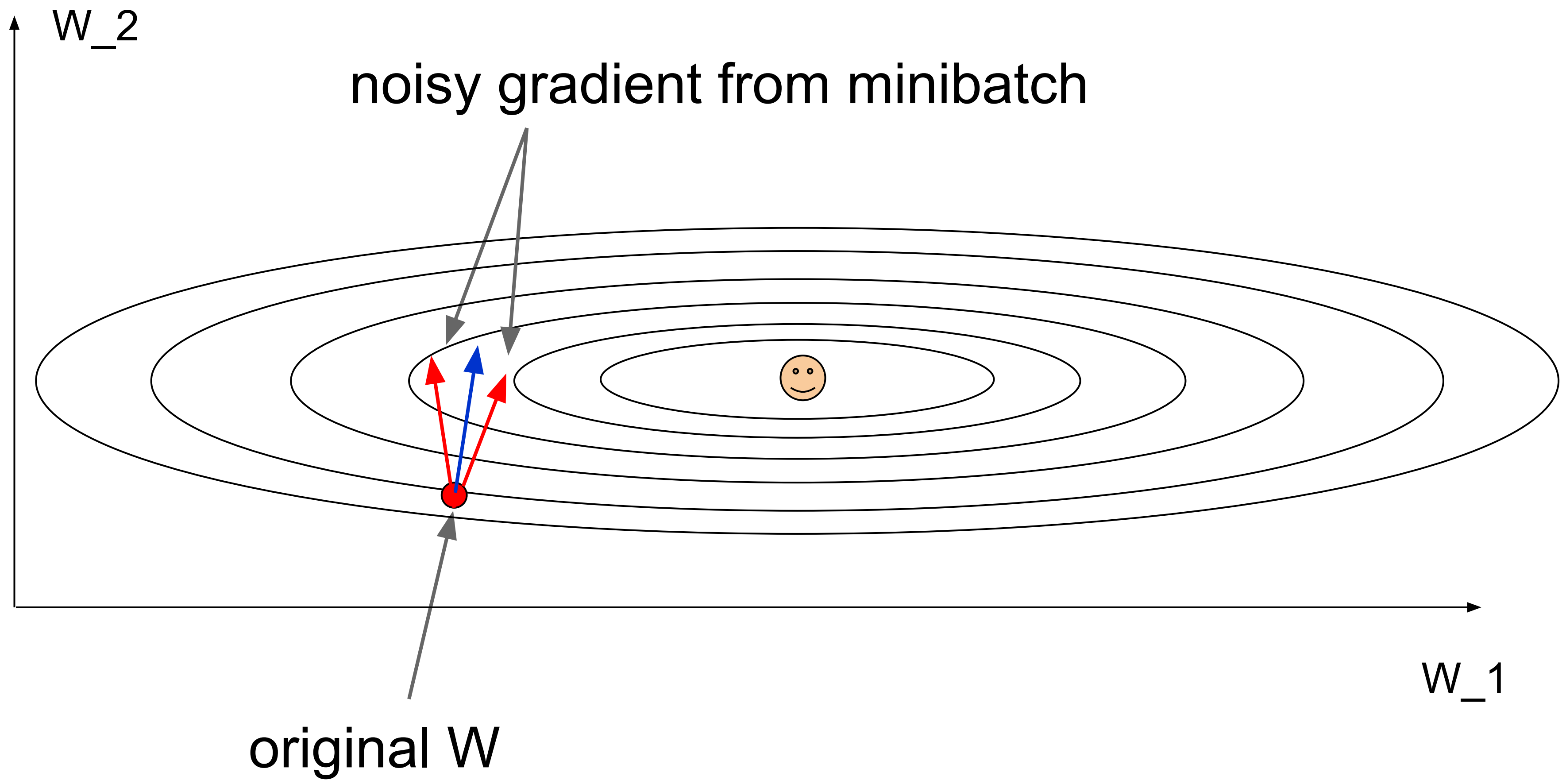


Minibatch updates

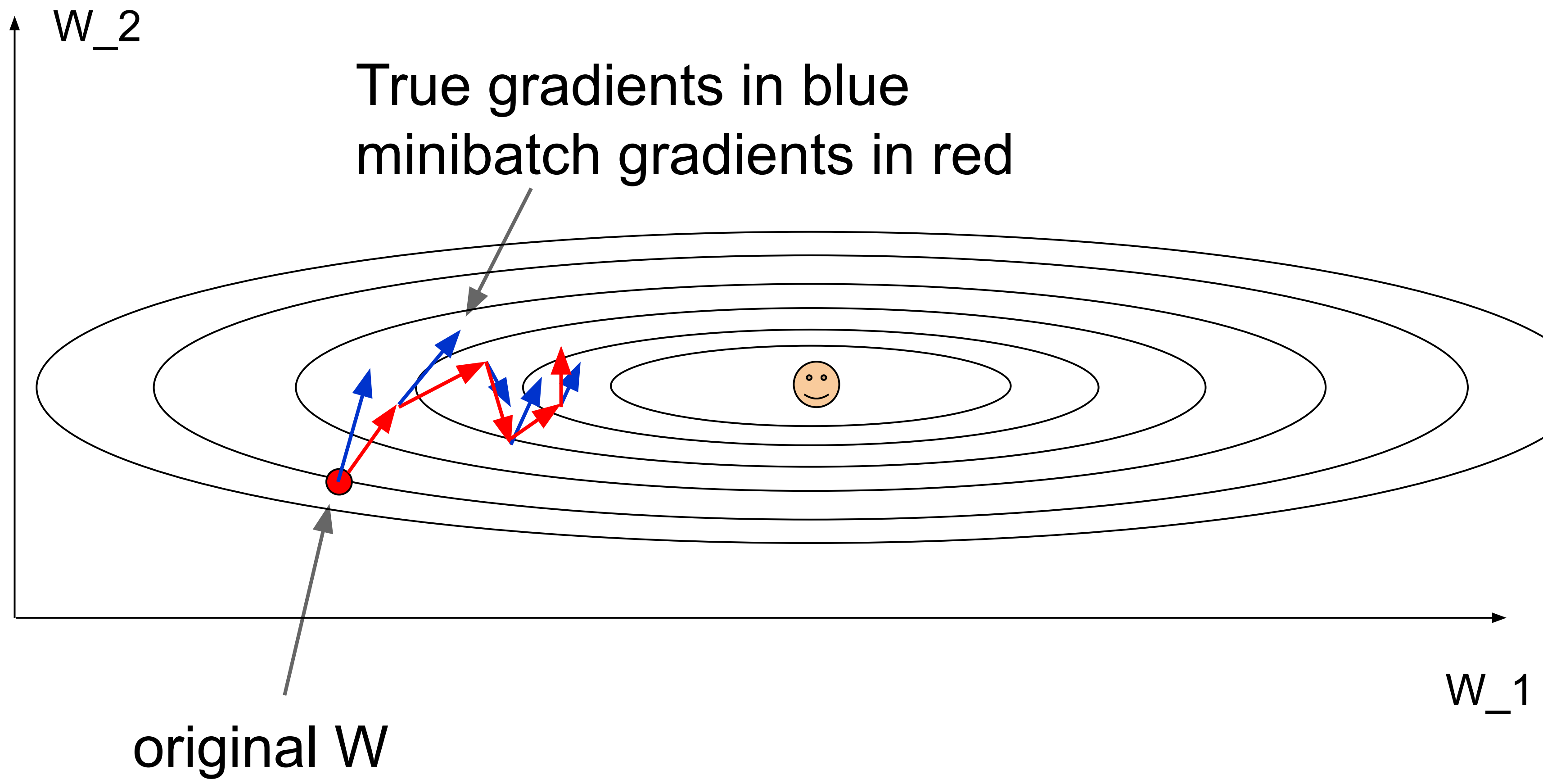


Slide based on CS294-129 by John Canny

Stochastic Gradient



Stochastic Gradient Descent



Gradients are noisy but still make good progress on average

Slide based on CS294-129 by John Canny

A Slight Detour: A Look at Optimization Tools

Optimization

Unconstrained Optimization

$$\underset{x}{\text{minimize}} \quad f(x)$$

Constrained Optimization

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & h_i(x) = 0, \;\; i = 1, \ldots, p \\ & f_i(x) \leq 0, \;\; i = 1, \ldots, m \end{array}$$

Line Search Framework for Unconstrained Minimization

$$\underset{x}{\text{minimize}}\ f(x)$$

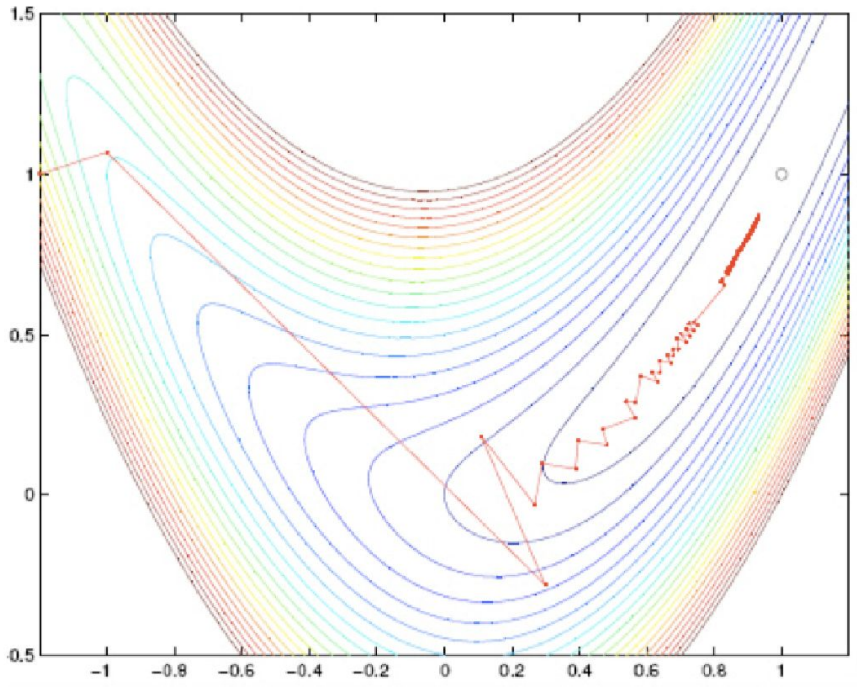
Solution Template

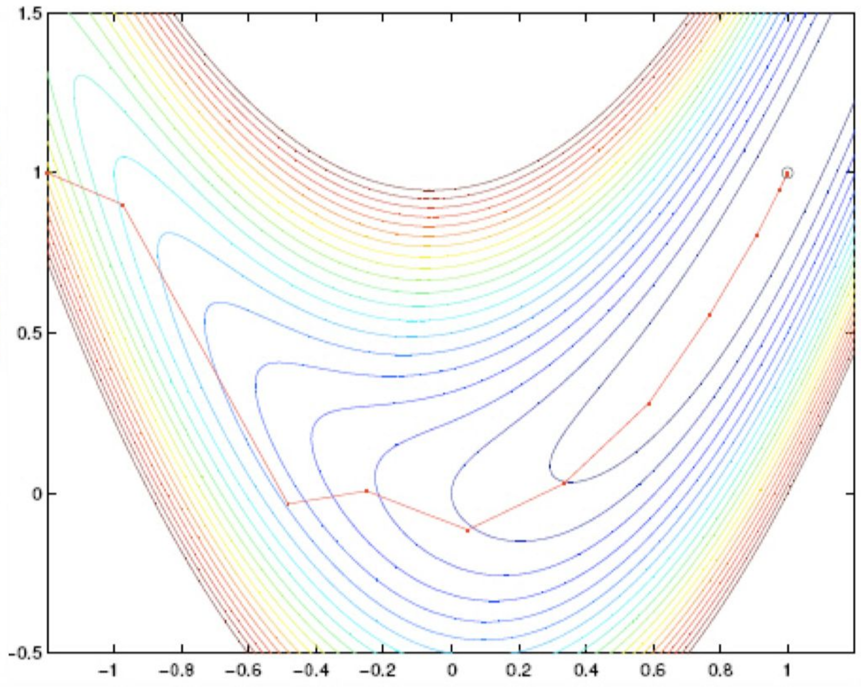
```
$k = 0$
choose a starting point, $x^0$
while (not converged)
    choose a search direction, $p^k$
    choose a step size in the search direction,  $x^{k+1} = x^k + t\, p^k$
    $k = k + 1$
```

Backtracking Line Search

- Simple and effective strategy for line search
- Reduce t incrementally: $t = \beta t$
- Termination condition: $f(x^k + tp^k) \leq f(x^k) + \alpha t \nabla f(x^k)^t p^k$
- Curvature condition automatically satisfied
- Algorithm parameters: α and β

Sample Search Paths



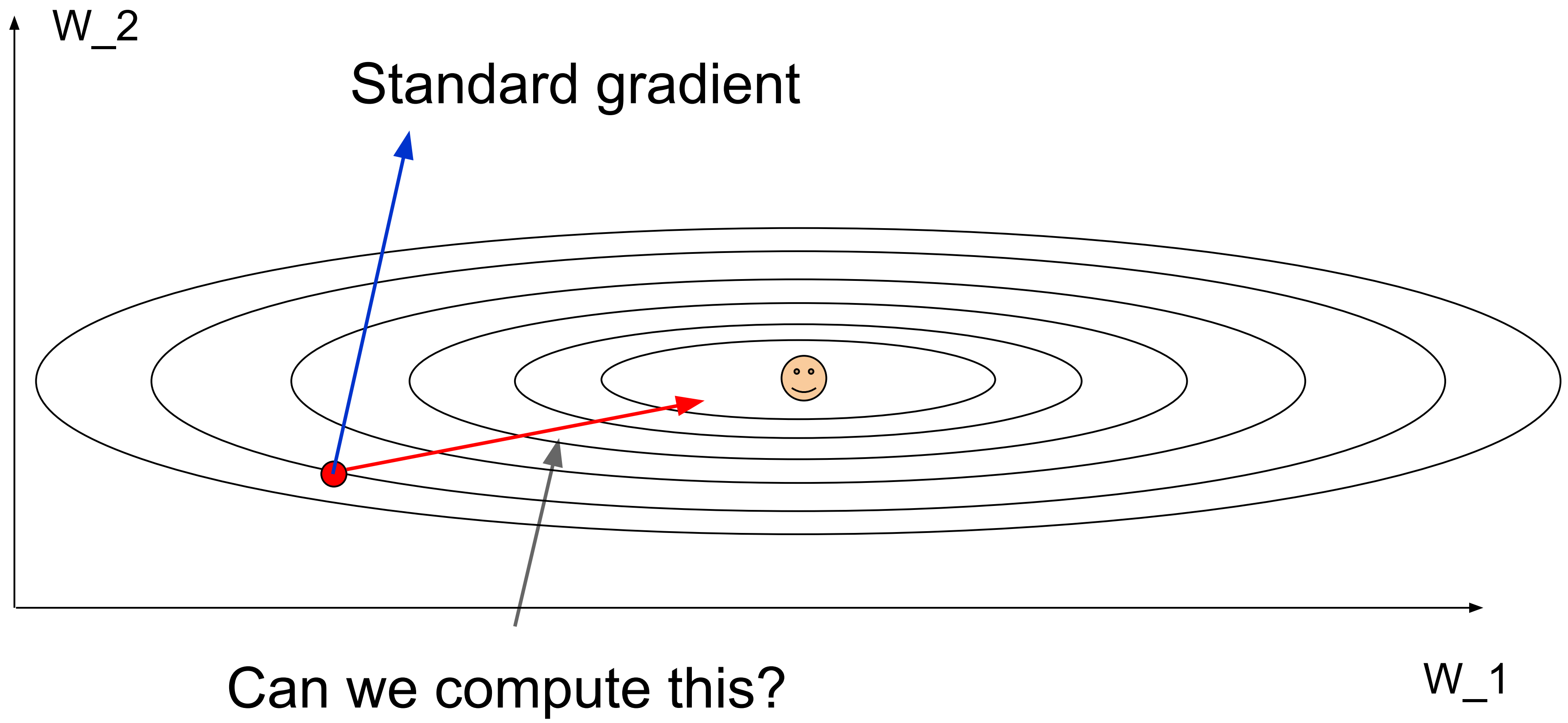


Steepest Descent with Backtracking in Matlab

```
function t = backtrackLineSearch(f, gk, pk, xk)
  a = 0.1; b = 0.8;  % α and β parameters
  t = 1;
  while ( f(xk+t*pk) > f(xk) + a*t*gk'*pk )
  t = b * t;
  end
```

```
function [x, hist] = steepestDescentBT(f, grad, x0)
  x = x0; hist = x0; tol = 1e-5;
  while (norm(grad(x)) > tol)
  p = -grad(x);
  t = backtrackLineSearch(f, grad(x), p, x);
  x = x + t * p;
  hist = [hist x];
  end
```

You might be wondering…



Taylor Series approximation of function

- *Let's have a look at the Taylor series approximation of function of single and multiple variables:*

$$f(x) = f(x^* + \Delta x) = f(x^*) + f'(x^*)\,\Delta x + \frac{1}{2}f''(x^*)\,\Delta x^2 + \cdots$$

$$\begin{aligned} f(x) = f(x^* + \Delta x) &= f(x^*) + \nabla f(x^*)^t\,\Delta x + \frac{1}{2}\Delta x^t\,\nabla^2 f(x^*)\,\Delta x + \cdots \\ &= f^* + \nabla f^{*t}\,\Delta x + \frac{1}{2}\,\Delta x^t\,\nabla^2 f^*\,\Delta x + \cdots \end{aligned}$$

Newton's method for zeros of a function

Based on the Taylor Series for $f(x+h)$:

$$f(x+h) = f(x) + hf'(x) + O(h^2)$$

To find a zero of f, assume $f(x+h) = 0$, so

$$h \approx -\frac{f(x)}{f'(x)}$$

And as an iteration:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Slide based on CS294-129 by John Canny

Newton's method for optima

For zeros of $f(x)$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

At a local optima, $f'(x) = 0$, so we use:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

If $f''(x)$ is constant (f is quadratic), then Newton's method finds the optimum in ***one step***.

More generally, Newton's method has ***quadratic converge.***

Slide based on CS294-129 by John Canny

Newton's method for gradients:

To find an optimum of a function $f(x)$ for high-dimensional x, we want zeros of its gradient: $\nabla f(x) = 0$

For zeros of $\nabla f(x)$ with a vector displacement h, Taylor's expansion is:

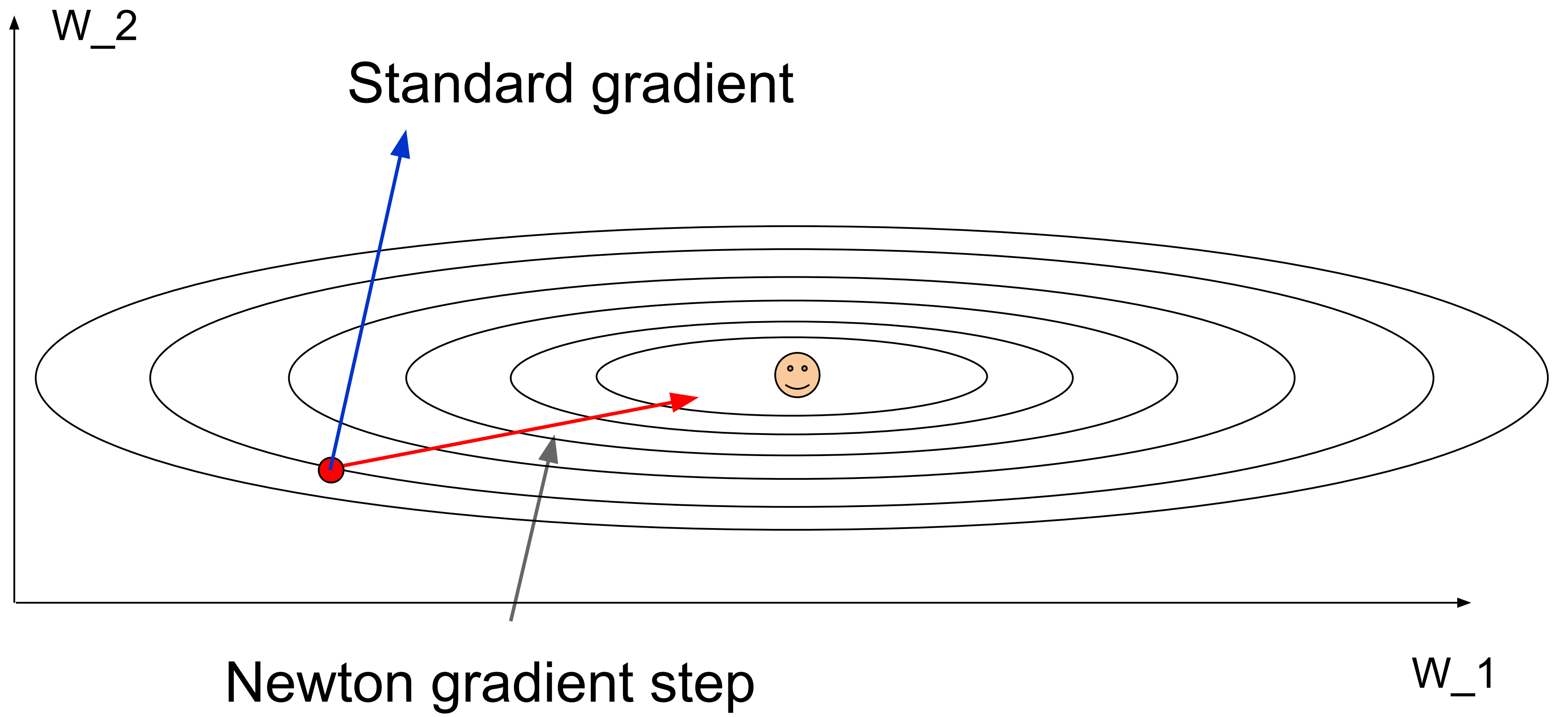
$$\nabla f(x+h) = \nabla f(x) + h^T H_f(x) + O(||h||^2)$$

Where H_f is the Hessian matrix of second derivatives of f. The update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

Slide based on CS294-129 by John Canny

Newton step



Slide based on CS294-129 by John Canny

Newton’s method for gradients:

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

Slide based on CS294-129 by John Canny

Newton’s method for gradients:

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1}\nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

Too expensive: if x_n has dimension M, the Hessian $H_f(x_n)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Slide based on CS294-129 by John Canny

Newton’s method for gradients:

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

Too expensive: if x_n has dimension M, the Hessian $H_f(x_n)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Too unstable: it involves a high-dimensional matrix inverse, which has poor numerical stability. The Hessian may even be singular.

Slide based on CS294-129 by John Canny

L-BFGS

There is an approximate Newton method that addresses these issues called L-BGFS, (Limited memory BFGS). BFGS is Broyden-Fletcher-Goldfarb-Shanno.

Idea: compute a low-dimensional approximation of $H_f(x_n)^{-1}$ directly.

Expense: use a k-dimensional approximation of $H_f(x_n)^{-1}$, Size is O(kM), cost is O(k^2 M).

Stability: much better. Depends on largest singular values of $H_f(x_n)$.

Convergence Nomenclature

Quadratic Convergence: error decreases quadratically (Newton's Method):

$$\epsilon_{n+1} \propto \epsilon_n^2 \quad \text{where} \quad \epsilon_n = x_{opt} - x_n$$

Linearly Convergence: error decreases linearly:

$$\epsilon_{n+1} \leq \mu \, \epsilon_n$$ where μ is the *rate of convergence*.

SGD: ??

Slide based on CS294-129 by John Canny

Convergence Behavior

Quadratic Convergence: error decreases quadratically

$$\epsilon_{n+1} \propto \epsilon_n^2 \quad \text{so} \quad \epsilon_n \text{ is } O\left(\mu^{2^n}\right)$$

Linearly Convergence: error decreases linearly:

$$\epsilon_{n+1} \leq \mu \, \epsilon_n \quad \text{so} \quad \epsilon_n \text{ is } O(\mu^n).$$

SGD: If learning rate is adjusted as 1/n, then (Nemirofski)

$$\epsilon_n \quad \text{is} \quad O(1/n).$$

Slide based on CS294-129 by John Canny

Convergence Comparison

Quadratic Convergence: ϵ_n is $O\left(\mu^{2^n}\right)$, time $\log(\log(\epsilon))$

Linearly Convergence: ϵ_n is $O(\mu^n)$, time $\log(\epsilon)$

SGD: ϵ_n is O(1/n), time $1/\epsilon$

SGD is terrible compared to the others. Why is it used?

Slide based on CS294-129 by John Canny

Convergence Comparison

Quadratic Convergence: ϵ_n is $O\left(\mu^{2^n}\right)$, time $\log(\log(\epsilon))$

Linearly Convergence: ϵ_n is $O(\mu^n)$, time $\log(\epsilon)$

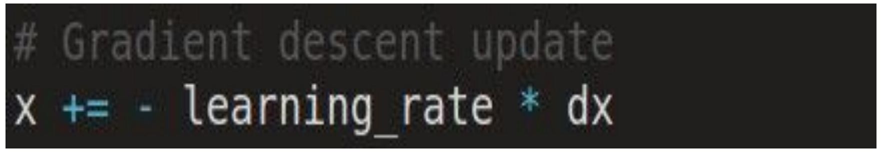
SGD: ϵ_n is O(1/n), time $1/\epsilon$

SGD is ***good enough*** for machine learning applications.
Remember "n" for SGD is minibatch count.

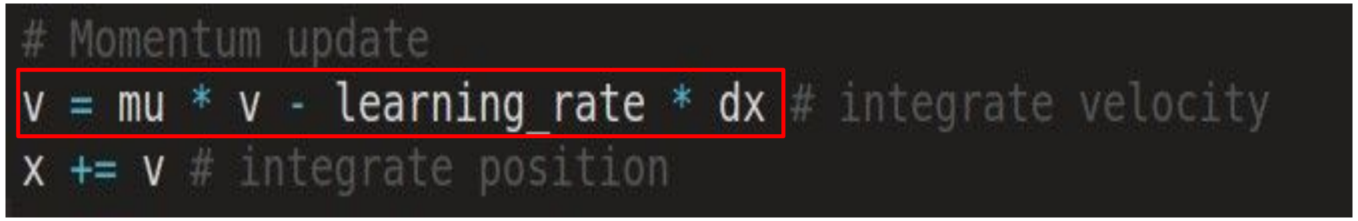
After 1 million updates, ϵ is order O(1/n) which is approaching floating point single precision.

Slide based on CS294-129 by John Canny

Momentum update



```
# Gradient descent update
x += - learning_rate * dx
```



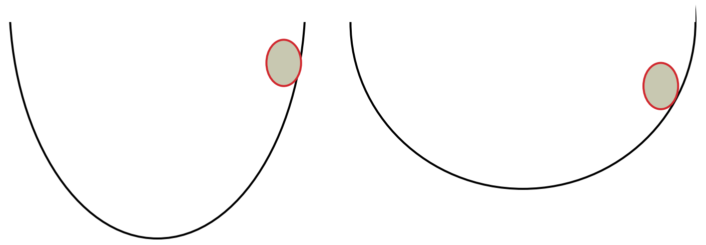
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
- mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

Momentum update

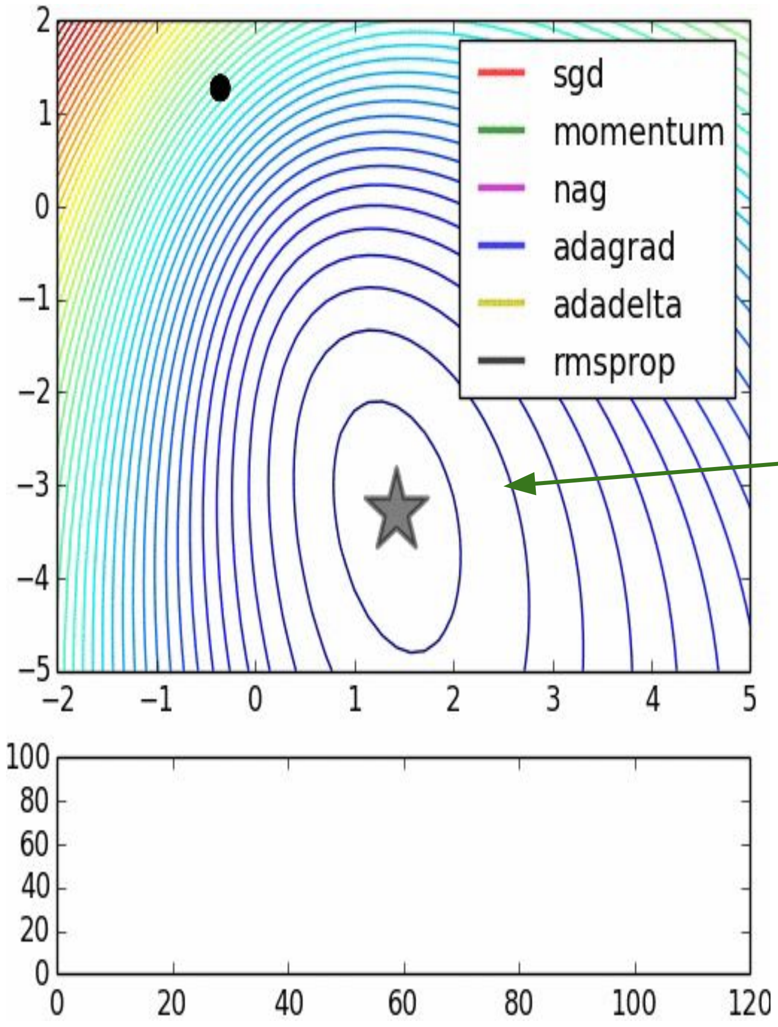
```
# Gradient descent update
x += - learning_rate * dx
```

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```



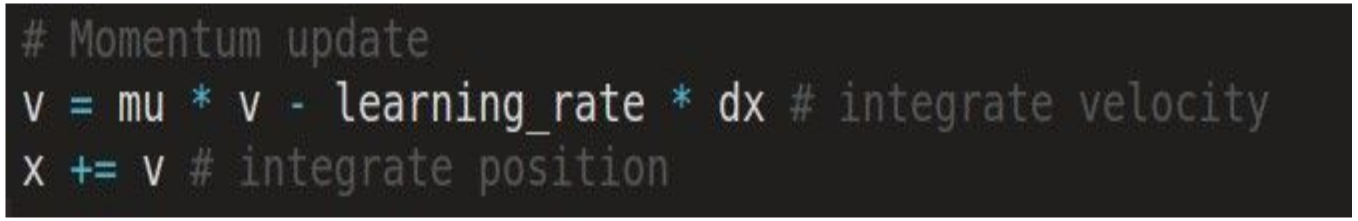
- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

SGD
vs
Momentum



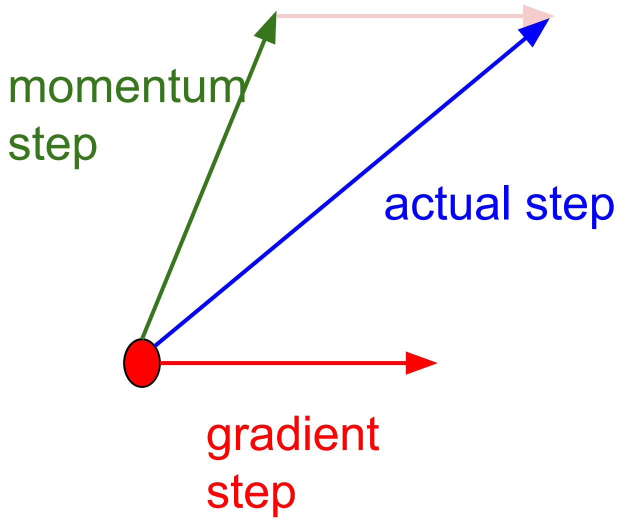
notice momentum overshooting the target, but overall getting to the minimum much faster than vanilla SGD.

Nesterov Momentum update

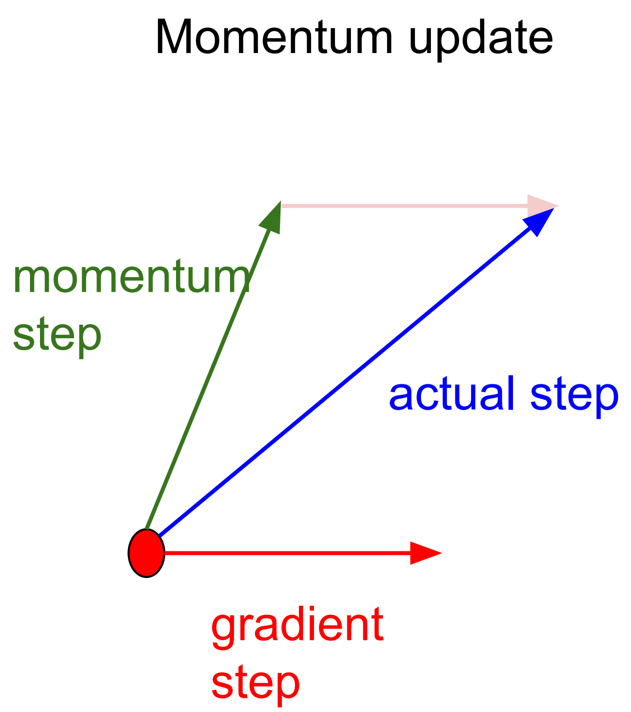


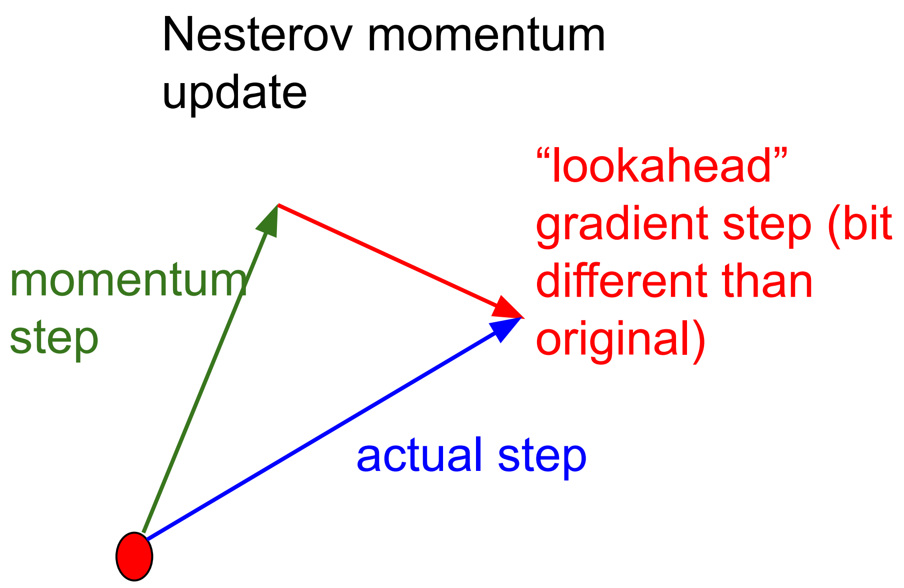
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Ordinary momentum update:

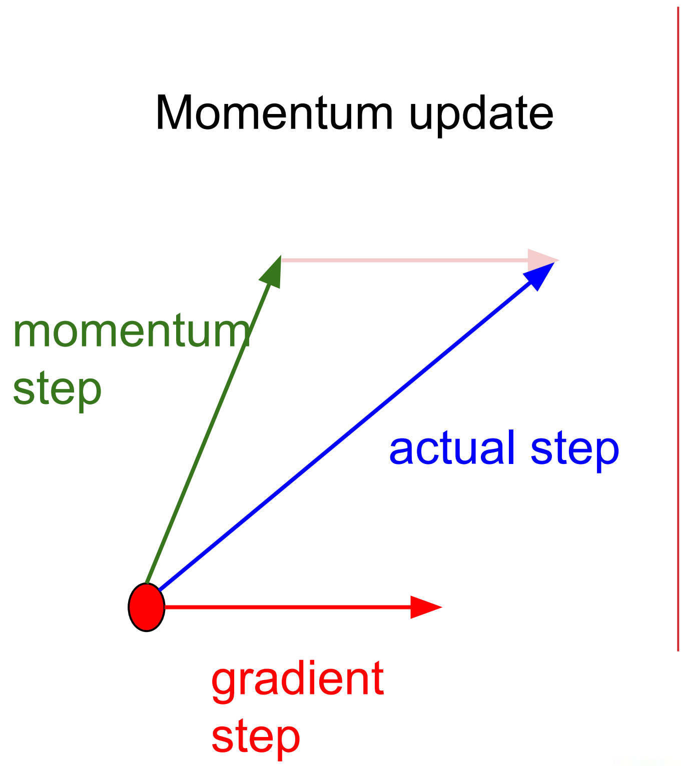


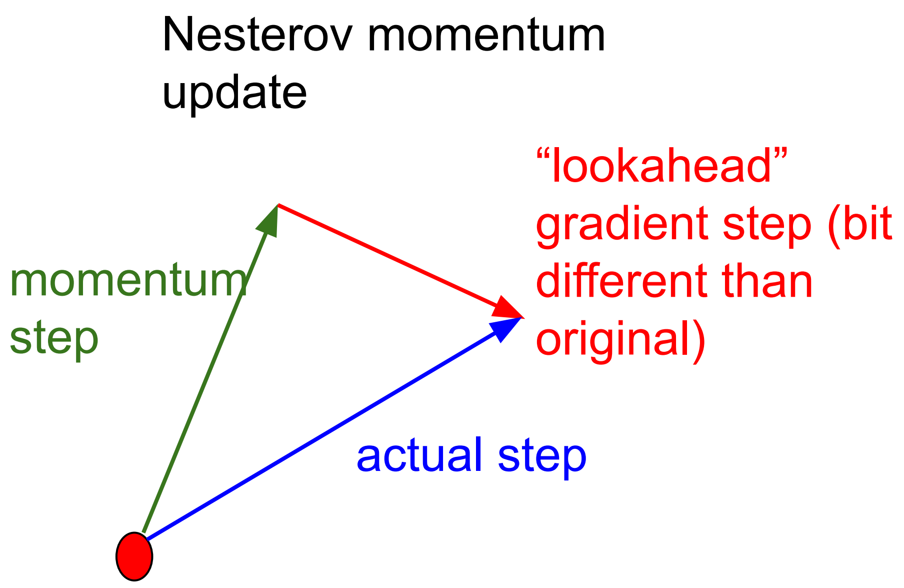
Nesterov Momentum update





Nesterov Momentum update





Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} \boxed{+ \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$

Nesterov Momentum update

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\boxed{\theta_{t-1} + \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Nesterov Momentum update

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\boxed{\theta_{t-1} + \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Variable transform and rearranging saves the day:

$$\boxed{\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}}$$

Nesterov Momentum update

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\boxed{\theta_{t-1} + \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

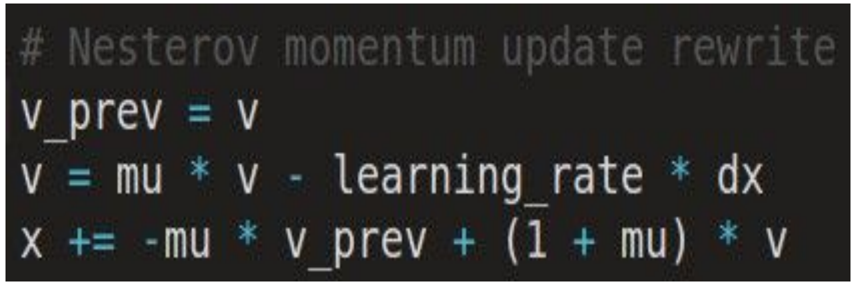
Variable transform and rearranging saves the day:

$$\boxed{\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}}$$

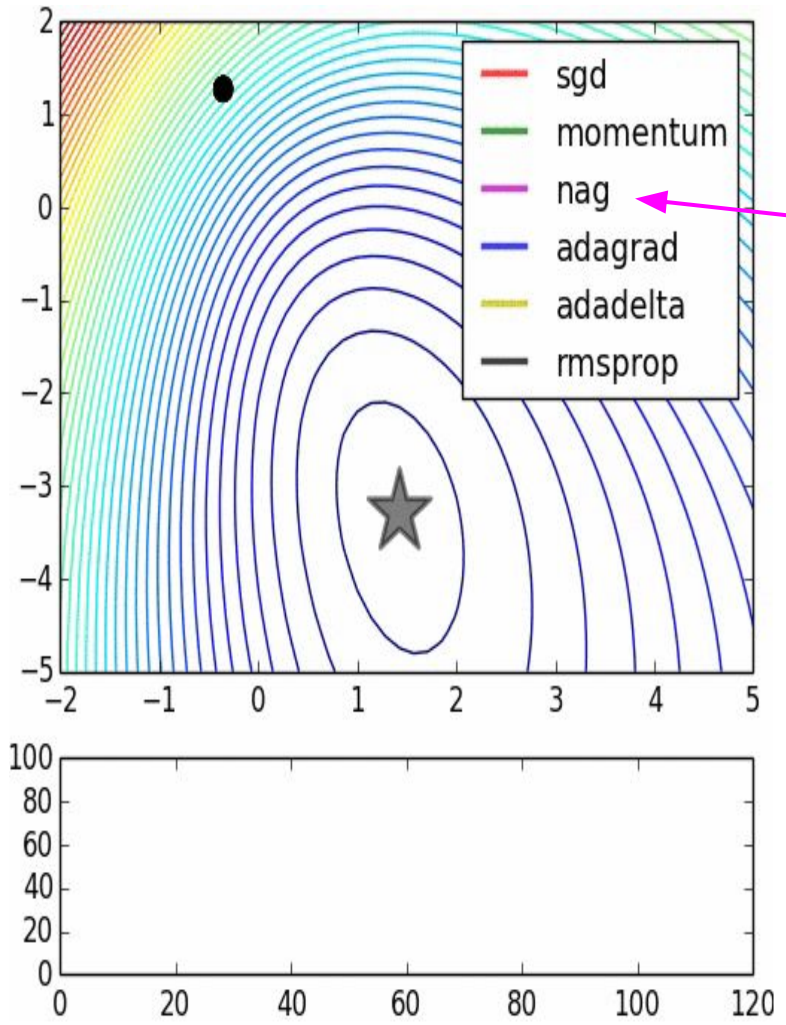
Replace all thetas with phis, rearrange and obtain:

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\phi_{t-1})$$

$$\phi_t = \phi_{t-1} - \mu v_{t-1} + (1 + \mu) v_t$$



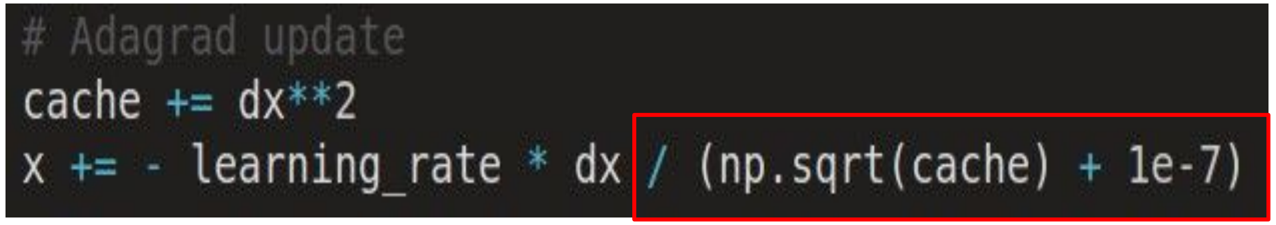
```
# Nesterov momentum update rewrite
v_prev = v
v = mu * v - learning_rate * dx
x += -mu * v_prev + (1 + mu) * v
```



nag = Nesterov Accelerated Gradient

AdaGrad update

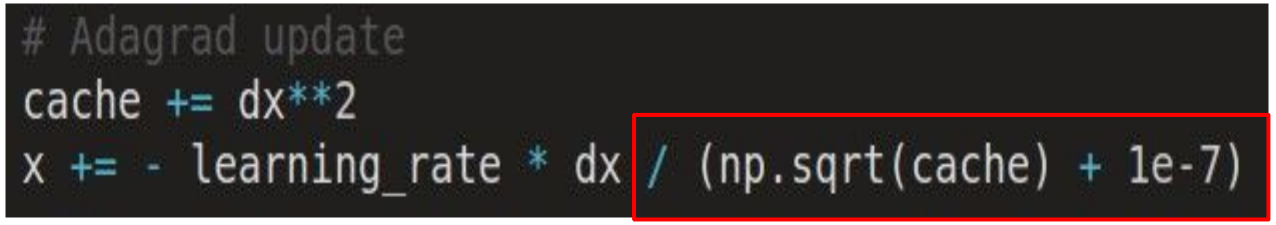
[Duchi et al., 2011]



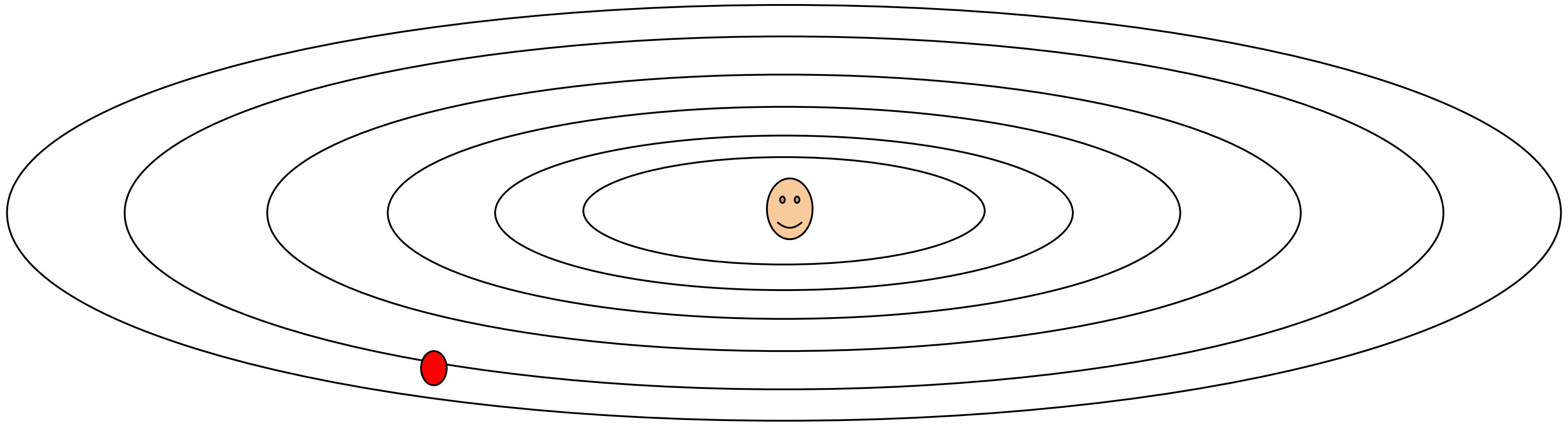
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

AdaGrad update

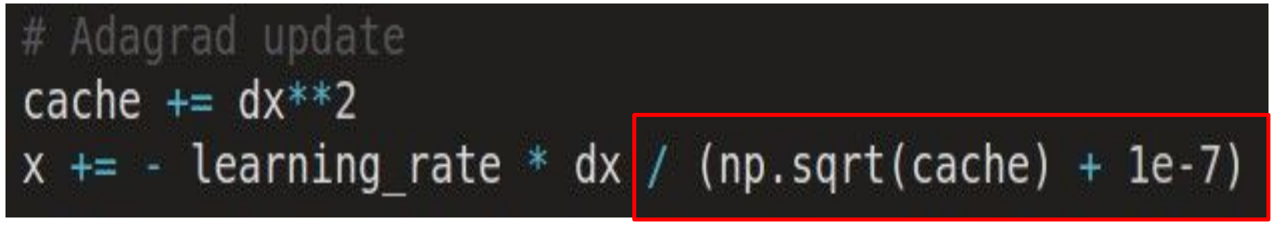


```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

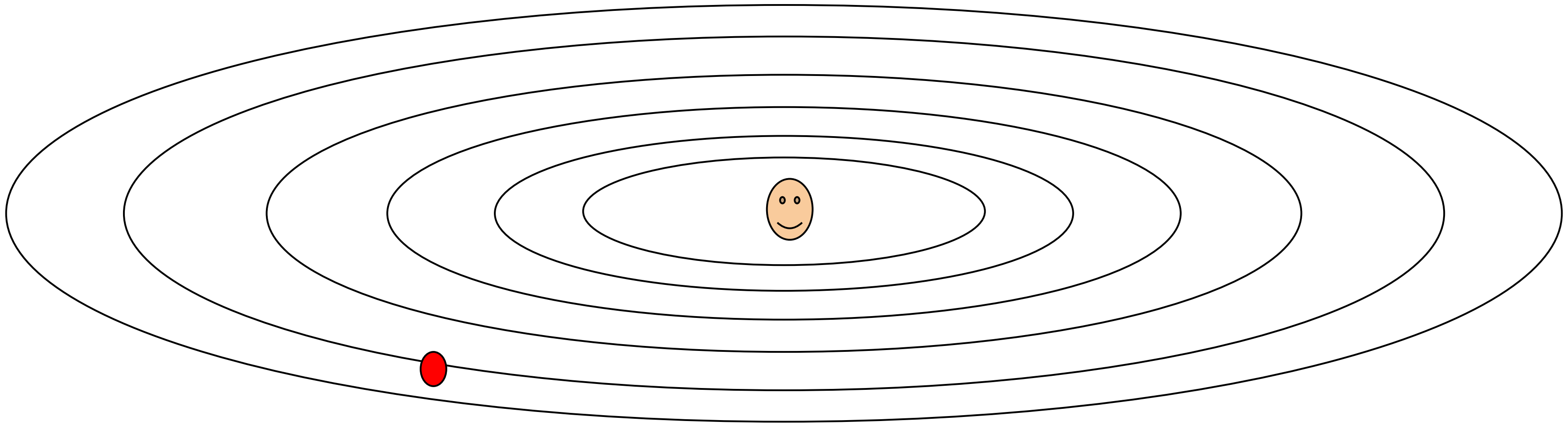


Q: What happens with AdaGrad?

AdaGrad update

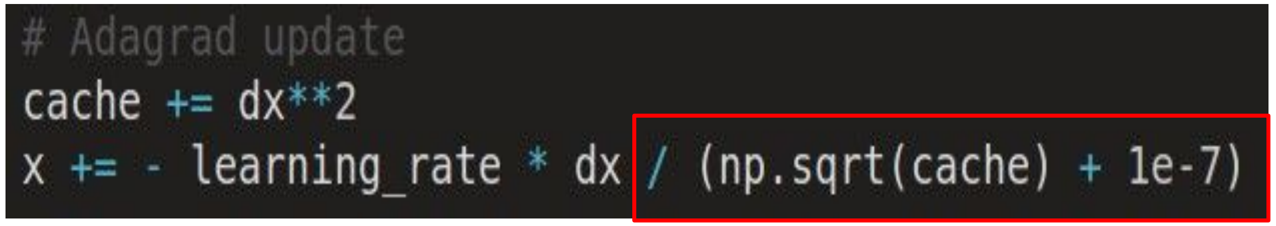


```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

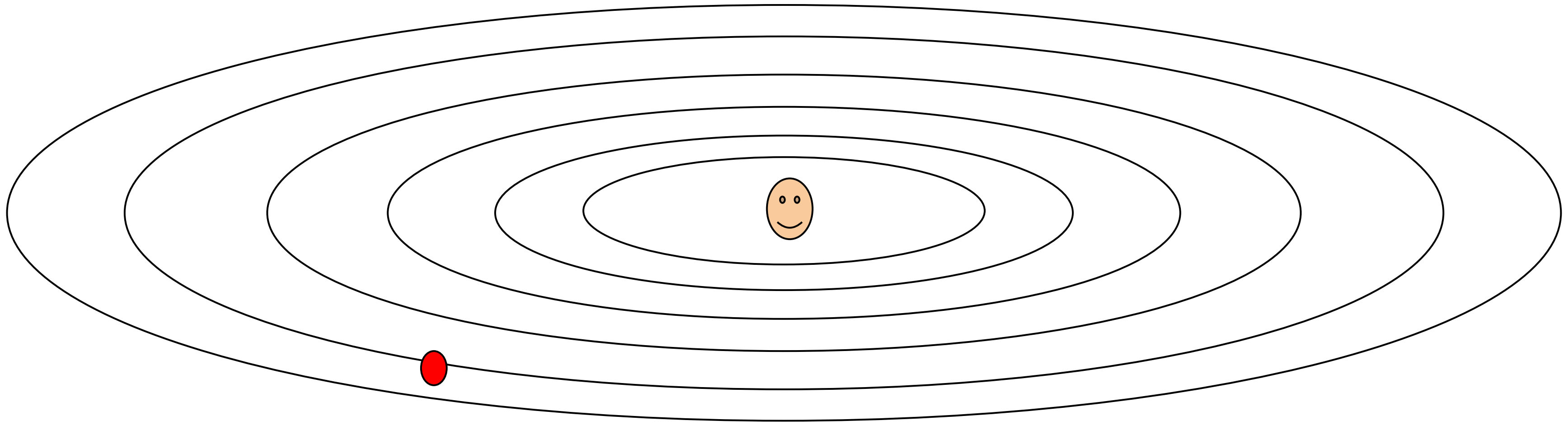


Q2: What happens to the step size over long time?

AdaGrad update



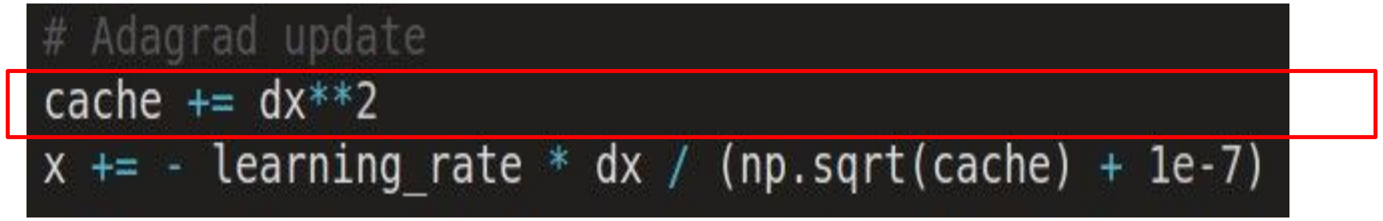
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



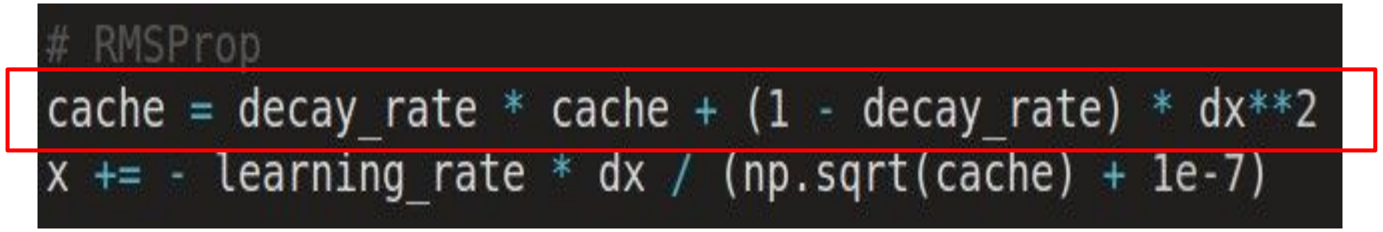
Q2: What happens to the step size over long time?

RMSProp update

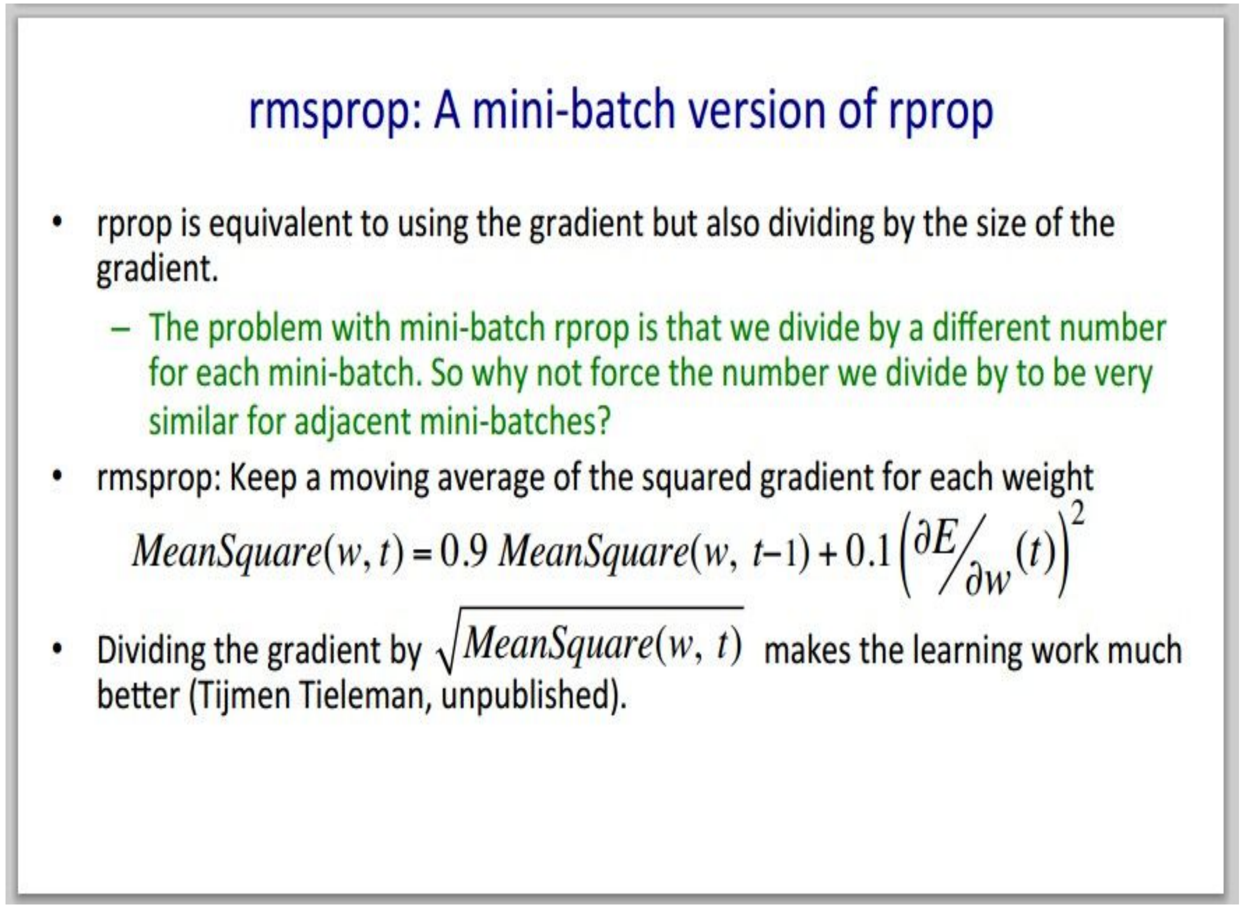
[Tieleman and Hinton, 2012]



```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

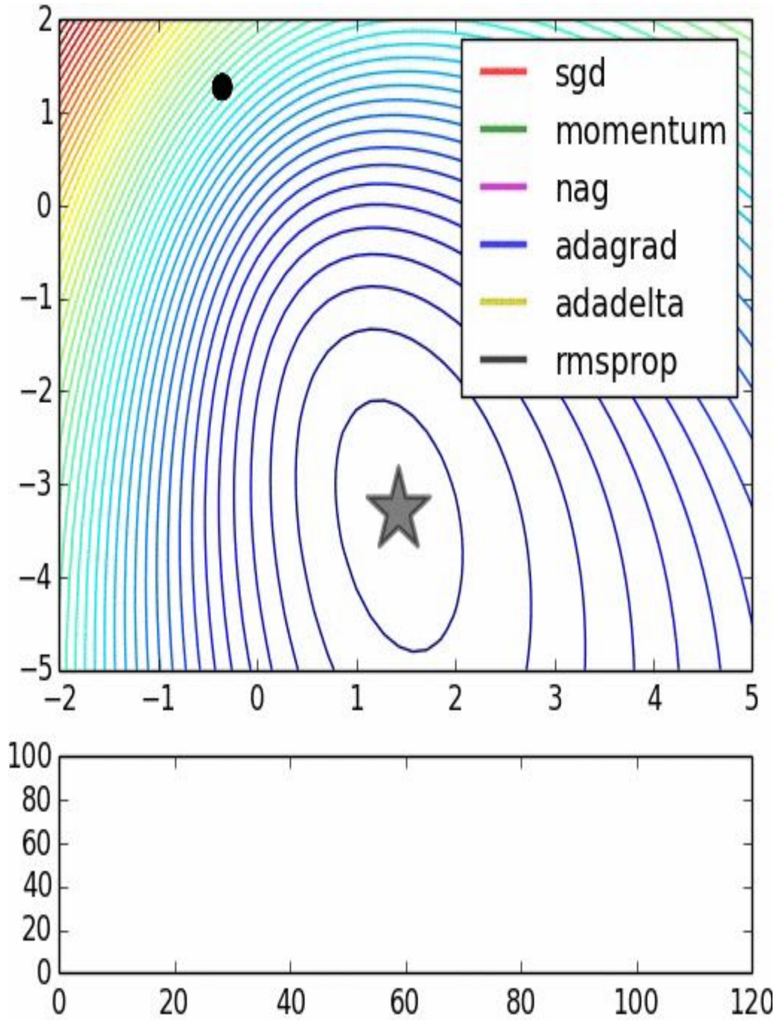
$$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t{-}1) + 0.1\left(\partial E/\partial w\,(t)\right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

Cited by several papers as:

[52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.

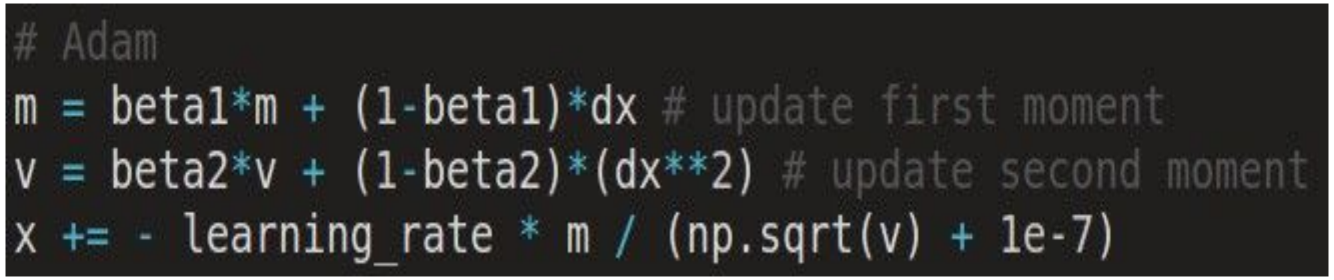


adagrad
rmsprop

Adam update

[Kingma and Ba, 2014]

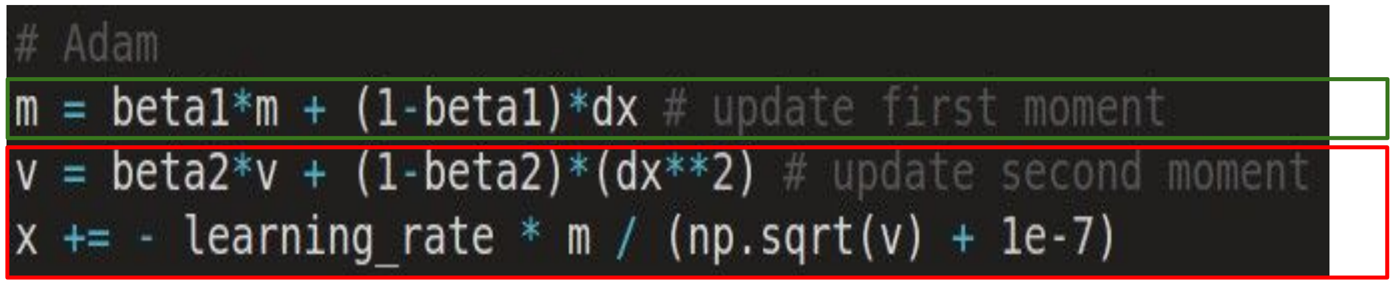
(incomplete, but close)


```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)



```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

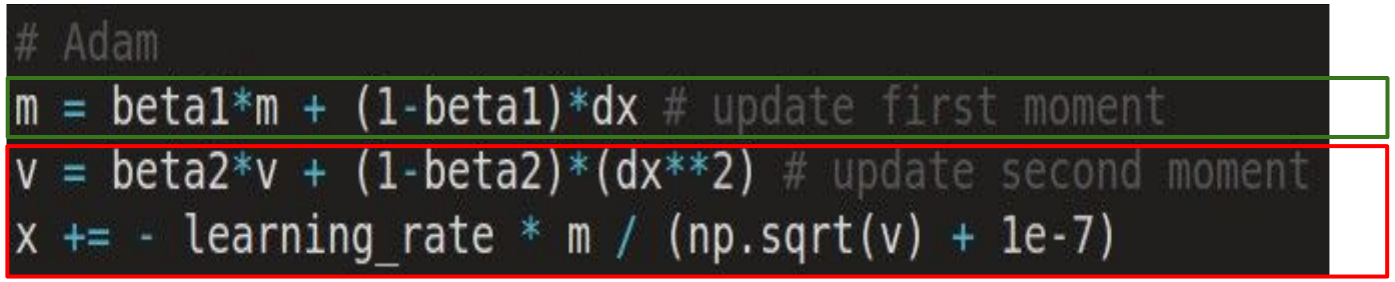
RMSProp-like

Looks a bit like RMSProp with momentum

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

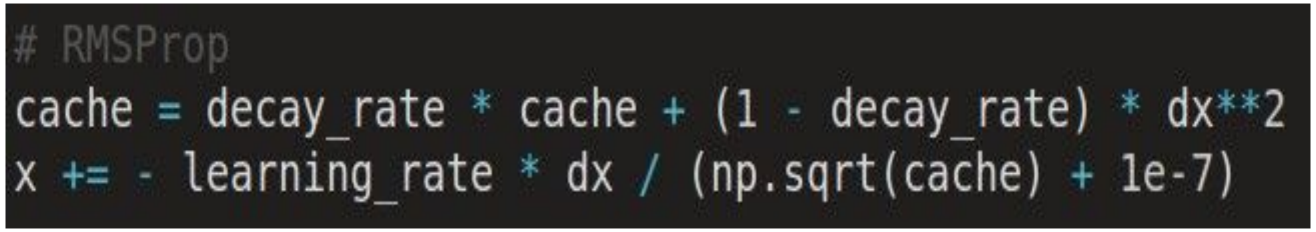


```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

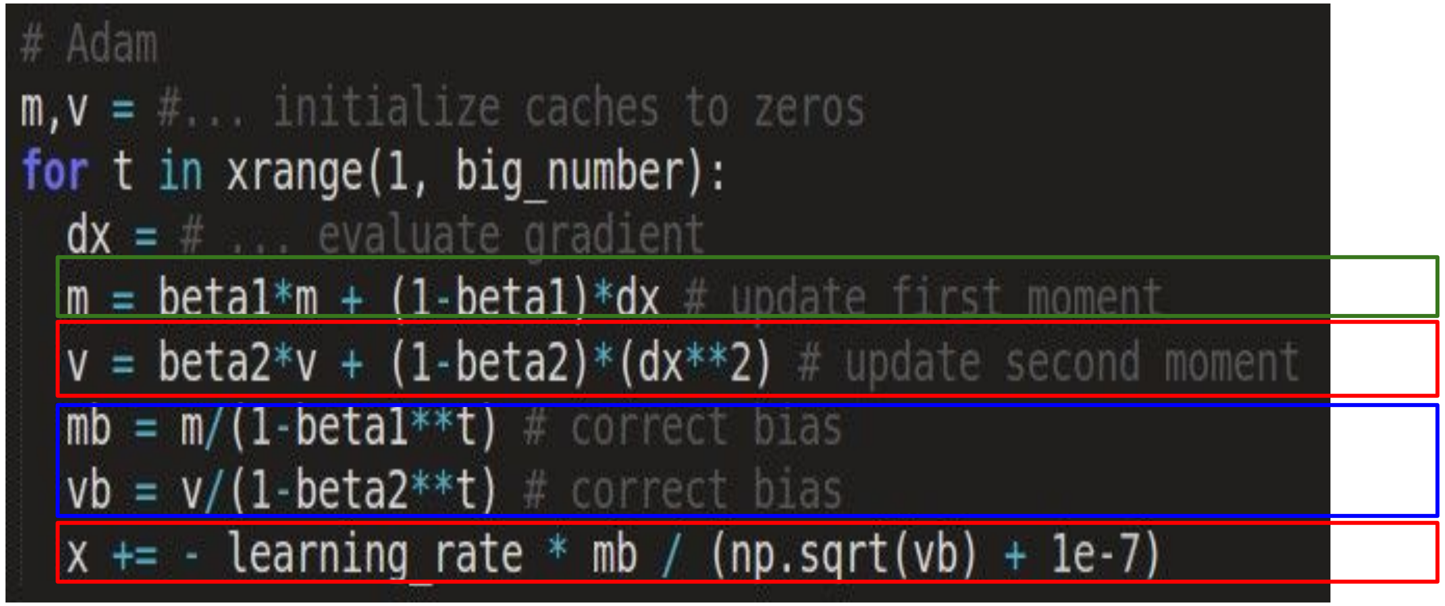
Looks a bit like RMSProp with momentum



```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Adam update

[Kingma and Ba, 2014]



```
# Adam
m,v = #... initialize caches to zeros
for t in xrange(1, big_number):
  dx = # ... evaluate gradient
  m = beta1*m + (1-beta1)*dx # update first moment
  v = beta2*v + (1-beta2)*(dx**2) # update second moment
  mb = m/(1-beta1**t) # correct bias
  vb = v/(1-beta2**t) # correct bias
  x += - learning_rate * mb / (np.sqrt(vb) + 1e-7)
```

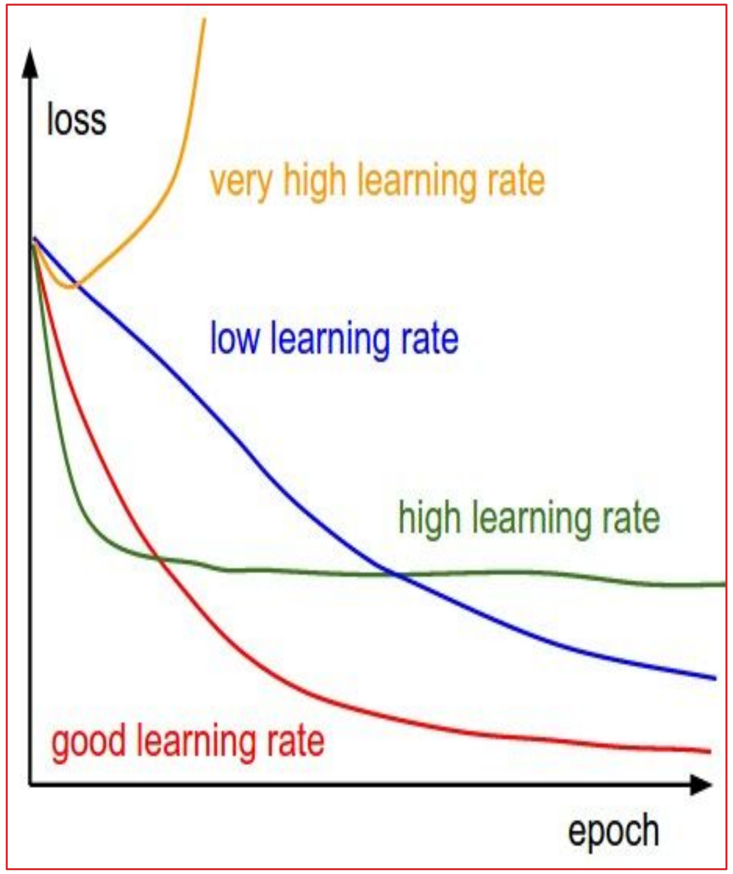
momentum

bias correction
(only relevant in first few iterations when t is small)

RMSProp-like

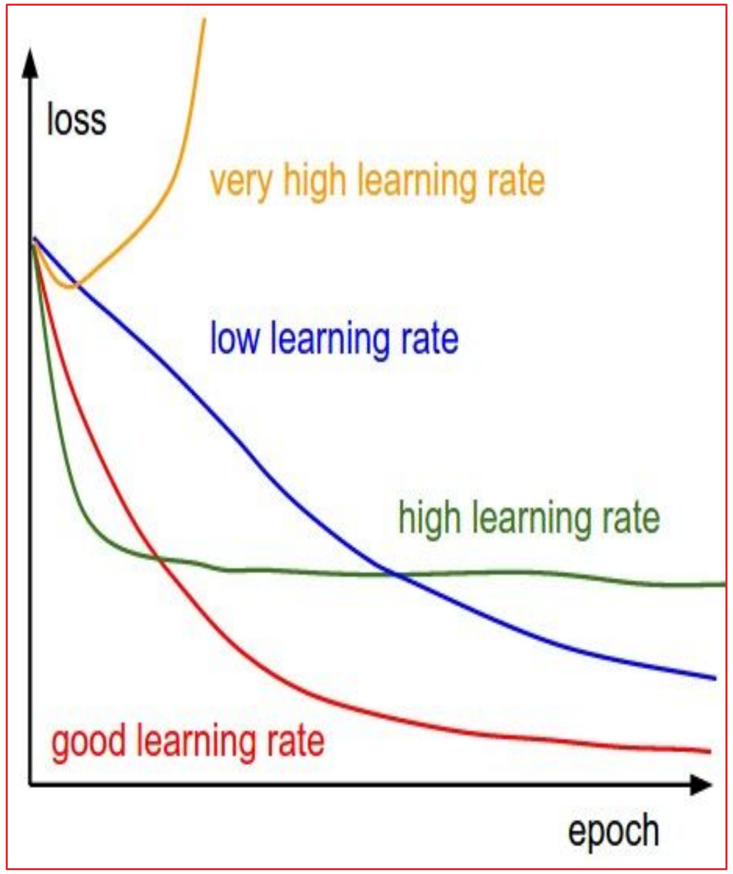
The bias correction compensates for the fact that m,v are initialized at zero and need some time to “warm up”.

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

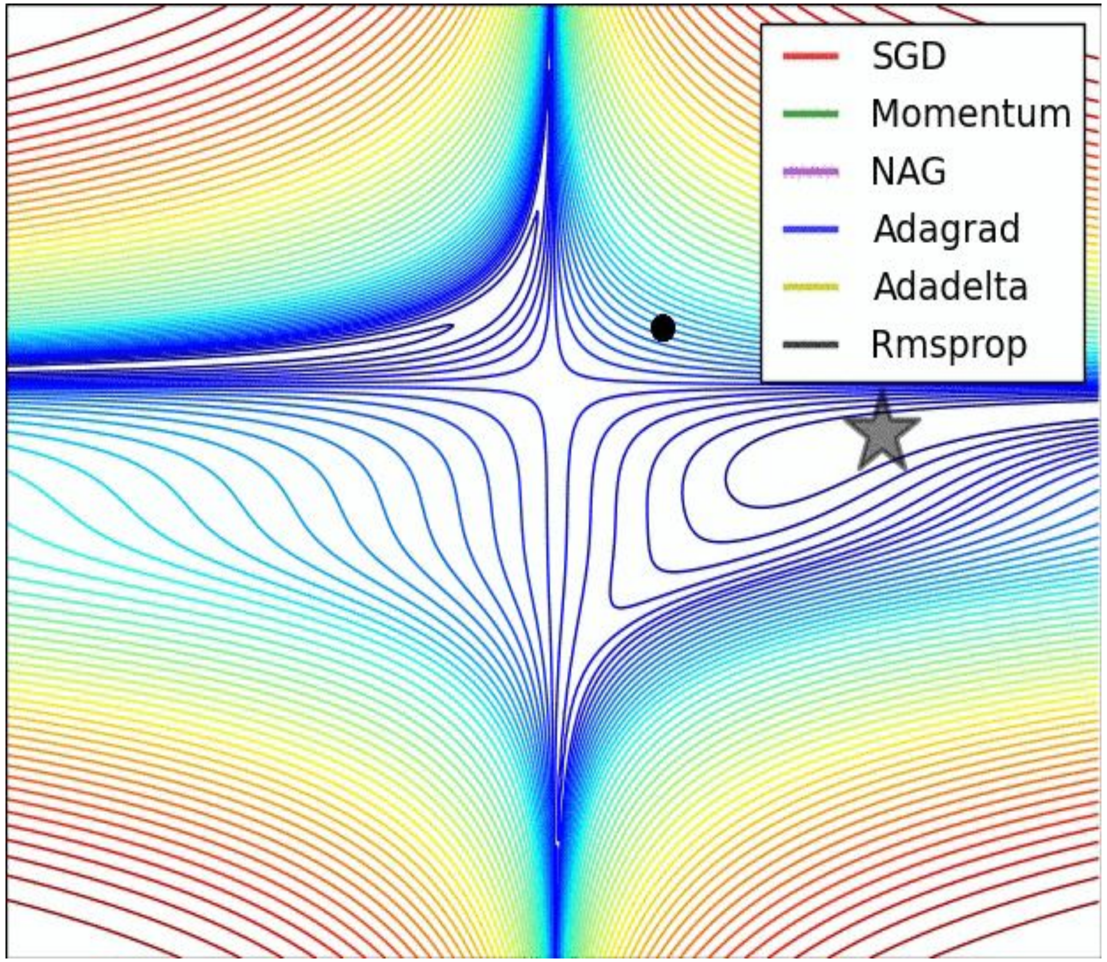
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Summary

- **Simple Gradient Methods** like SGD can make adequate progress to an optimum when used on minibatches of data.
- **Second-order** methods make much better progress toward the goal, but are more expensive and unstable.
- **Convergence rates:** quadratic, linear, O(1/n).
- **Momentum:** is another method to produce better effective gradients.
- ADAGRAD, RMSprop diagonally scale the gradient. ADAM scales and applies momentum.



(image credits to Alec Radford)

Questions?