Artificial Intelligence and Machine Learning

Gradient Descent Methods

Gradient Descent

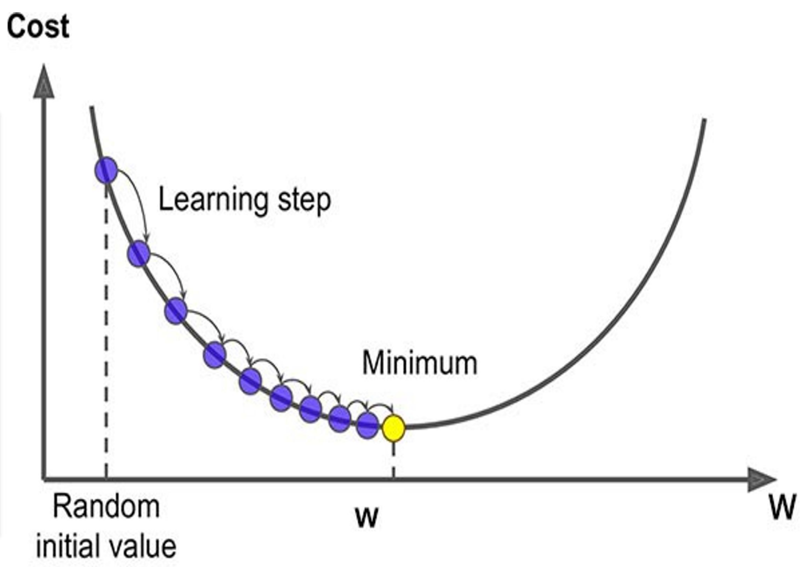
Want $\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

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

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

simultaneous update
for j = 0 ... d

Learning rate




```
# 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

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

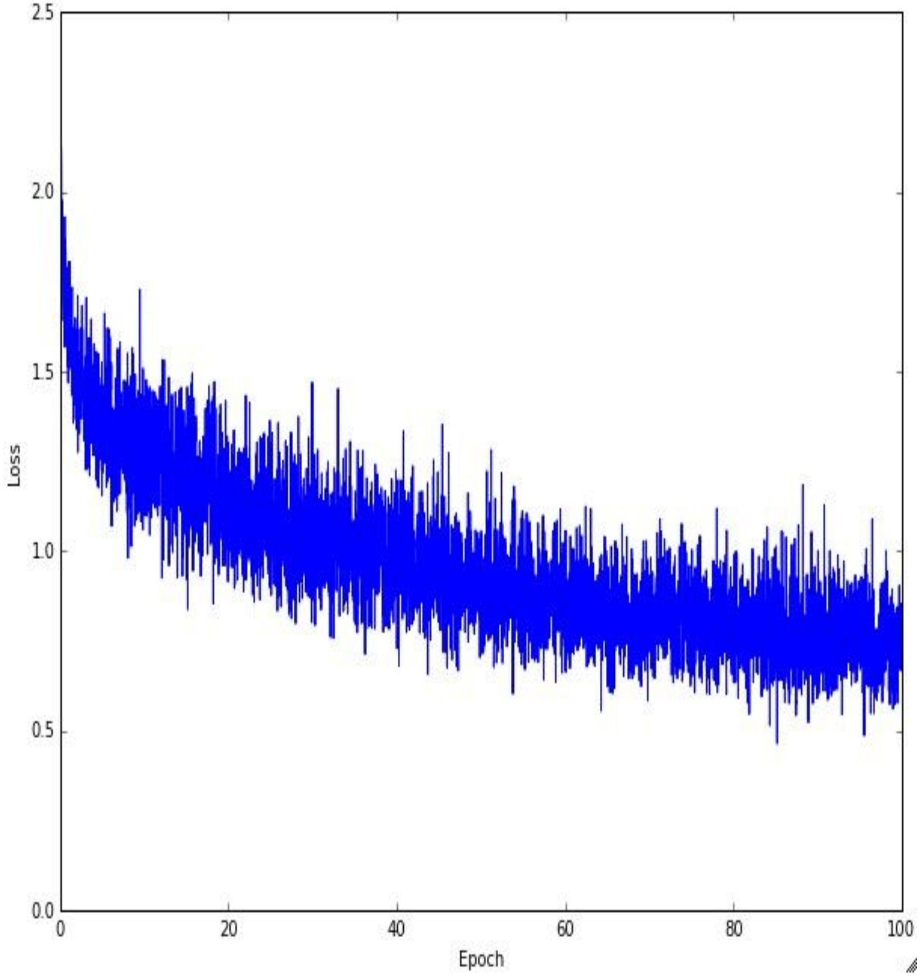
- only use a small portion of the training set to compute the gradient.

```
# 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
```

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

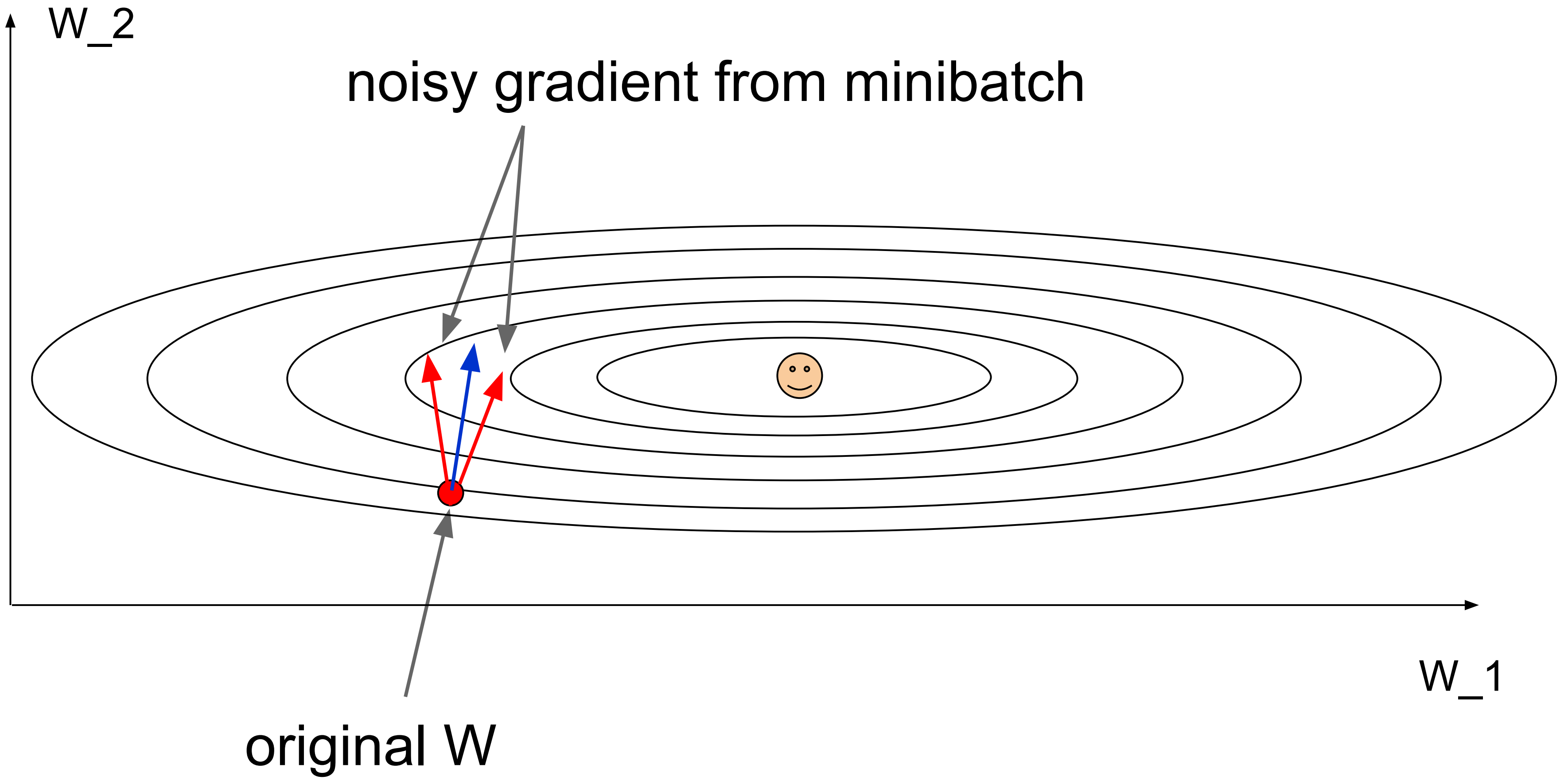
Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples



Example of optimization progress while training a neural network.

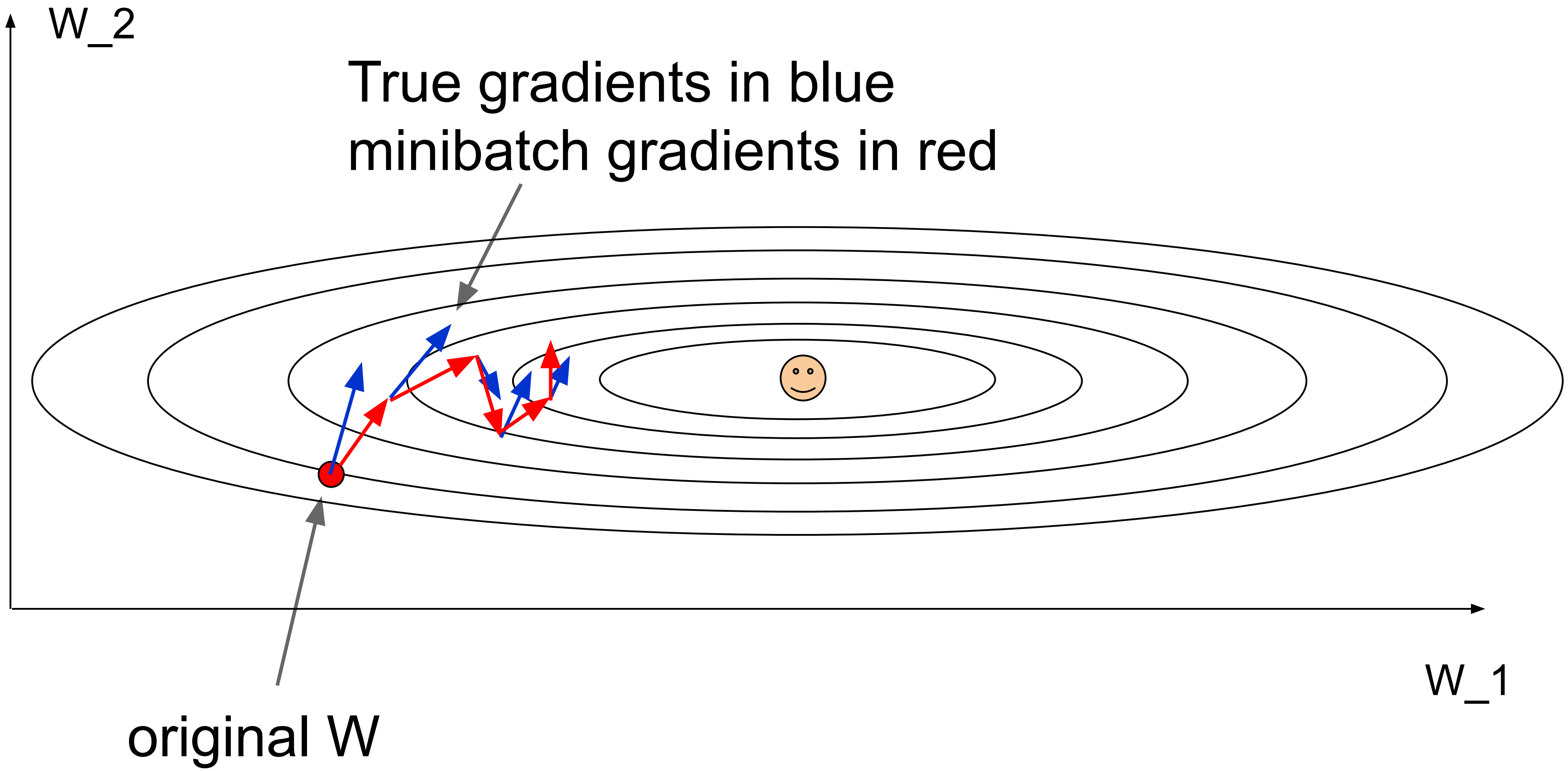
(Loss over mini-batches goes down over time.)

Stochastic Gradient



Slide based on CS294-129 by John Canny

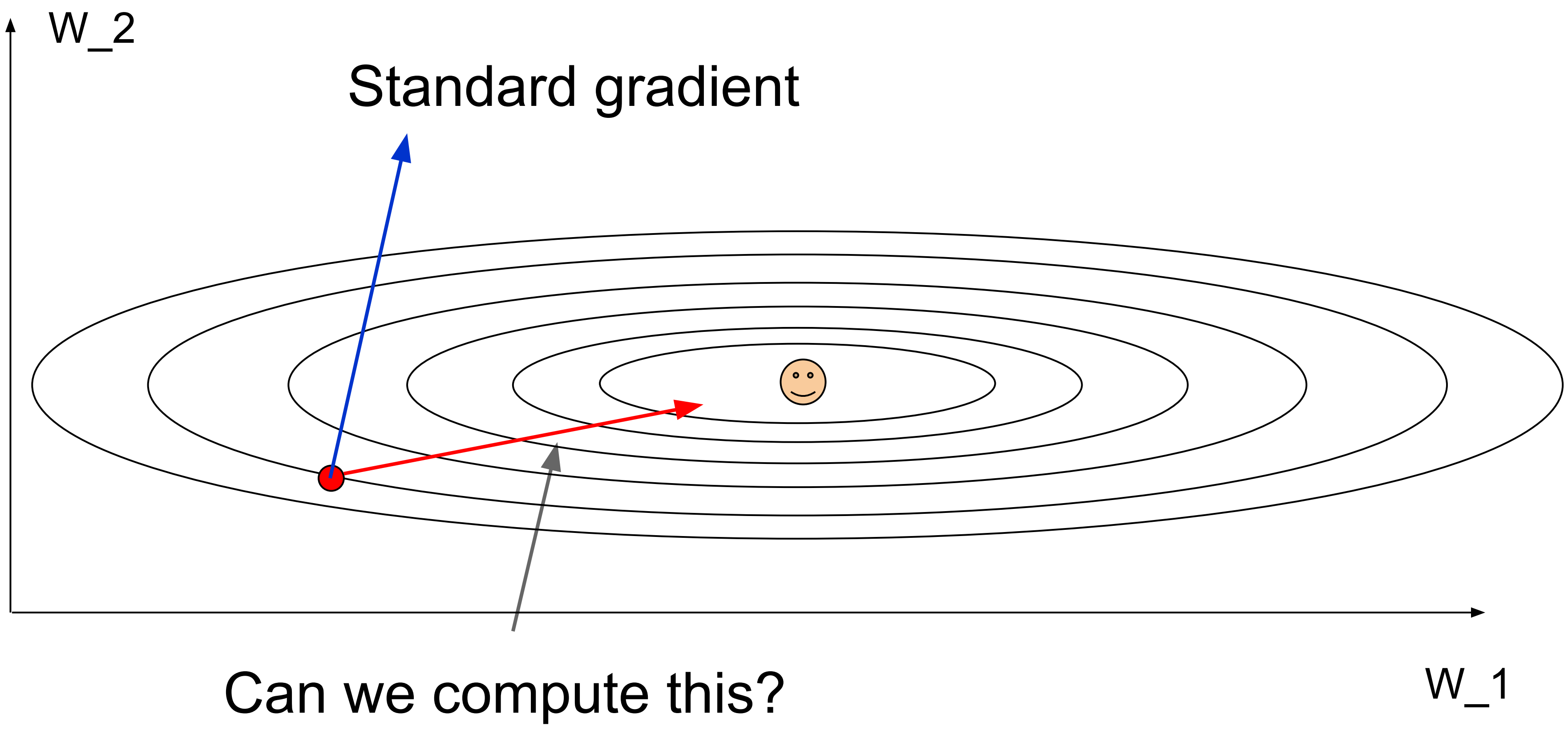
Stochastic Gradient Descent



Gradients are noisy but still make good progress on average

Slide based on CS294-129 by John Canny

You might be wondering…



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
```

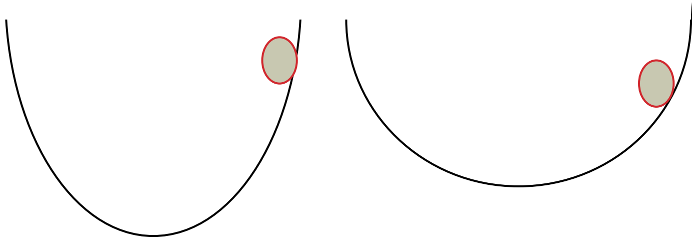
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- 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

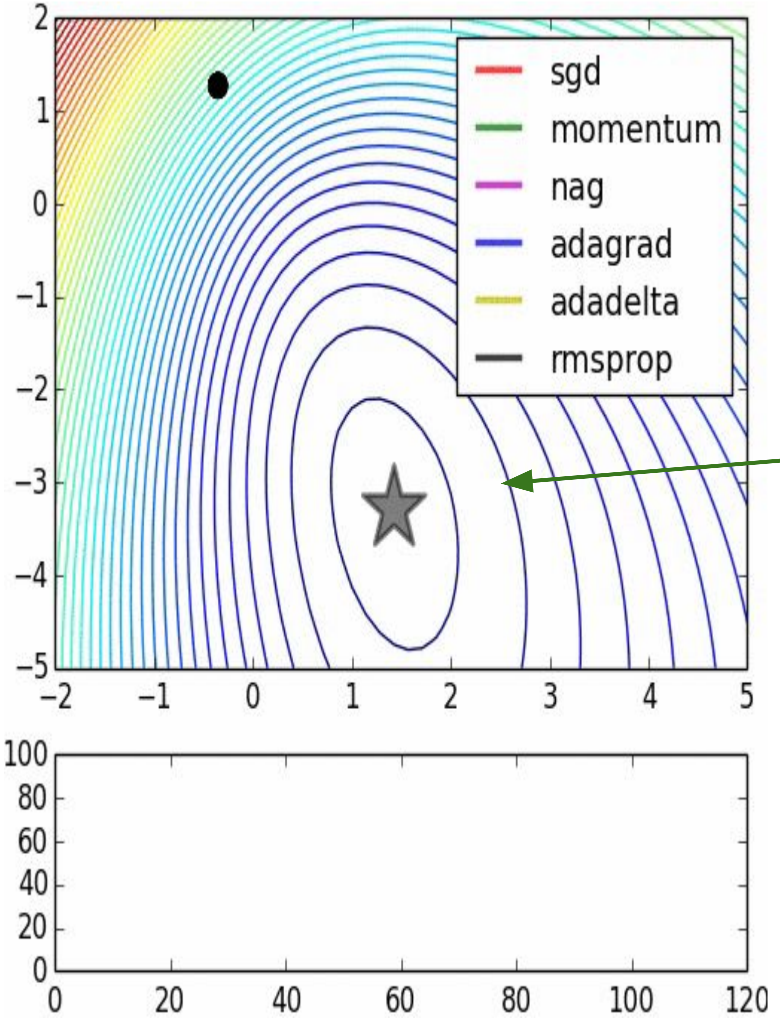
```
# Gradient descent update
x += - learning_rate * dx
```

```
# 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.

AdaGrad update

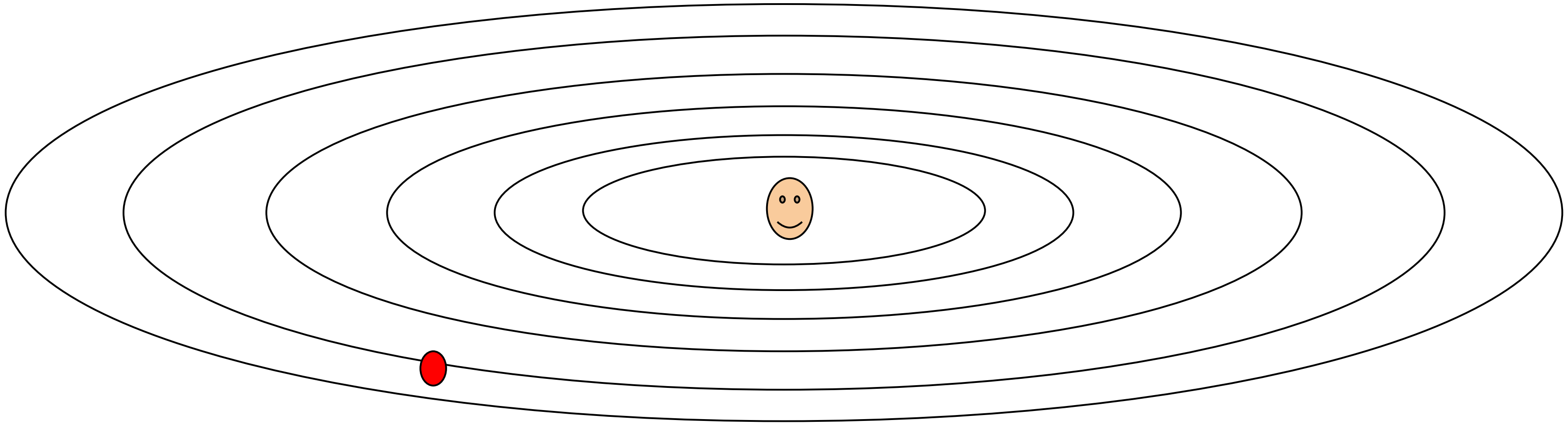
[Duchi et al., 2011]

```
# 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

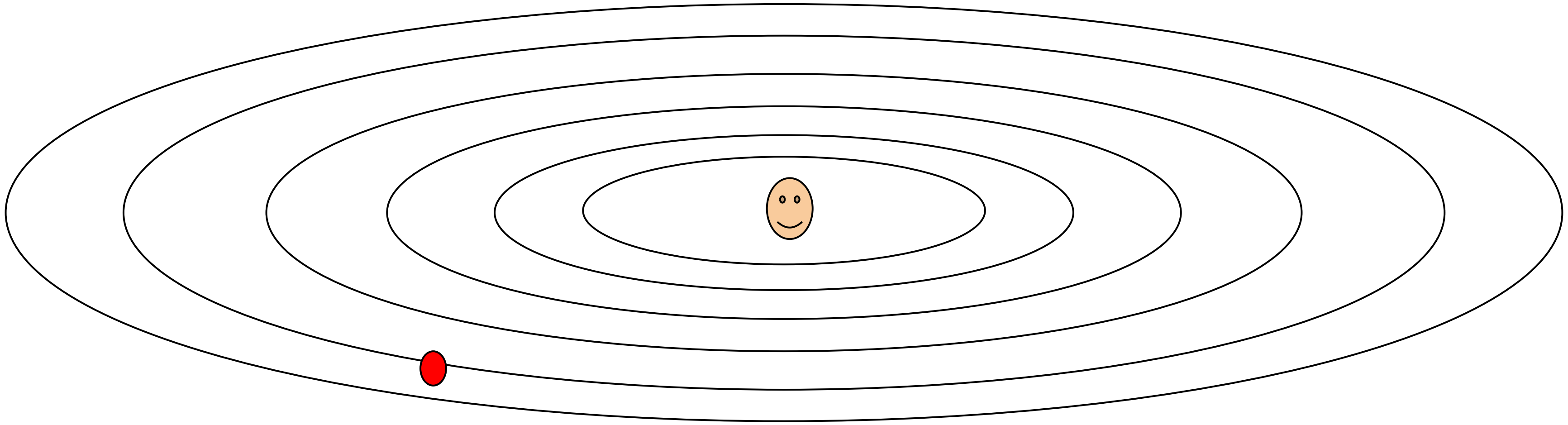
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



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)
```



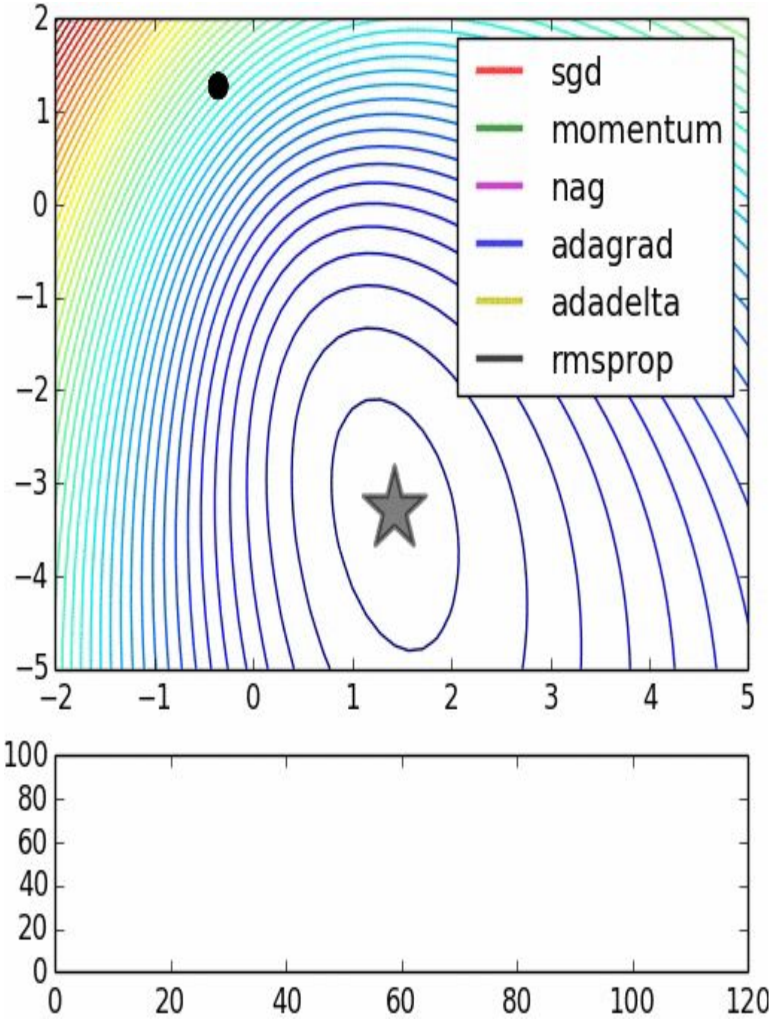
Q2: What happens to the step size over long time?

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)
```



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]

أكاديمية كاوست
KAUST ACADEMY

```
# 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)
```

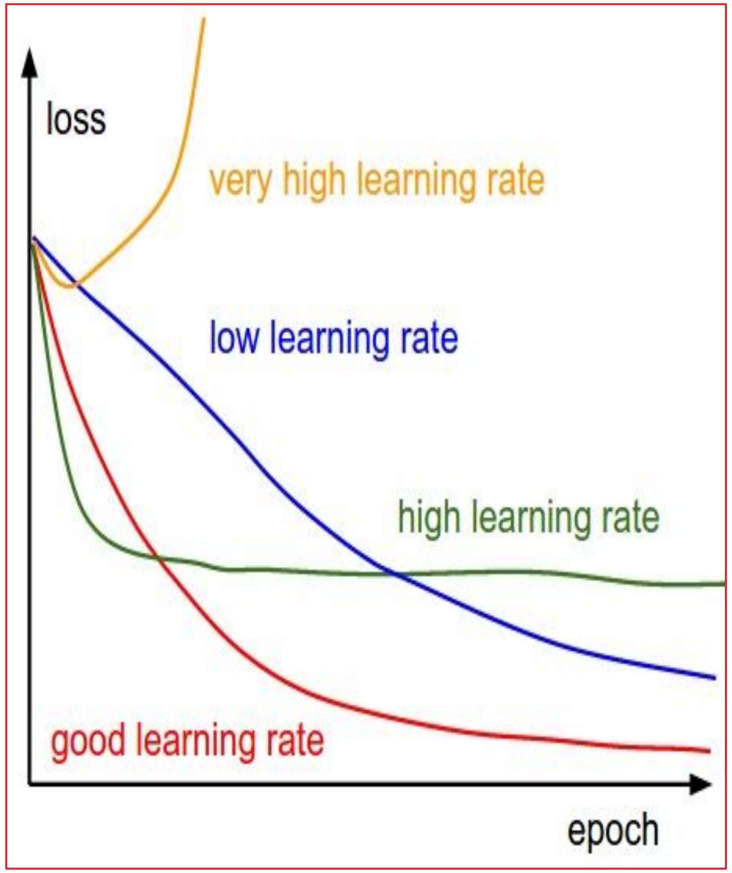
momentum

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.



=> Learning rate decay over time!

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

exponential decay:

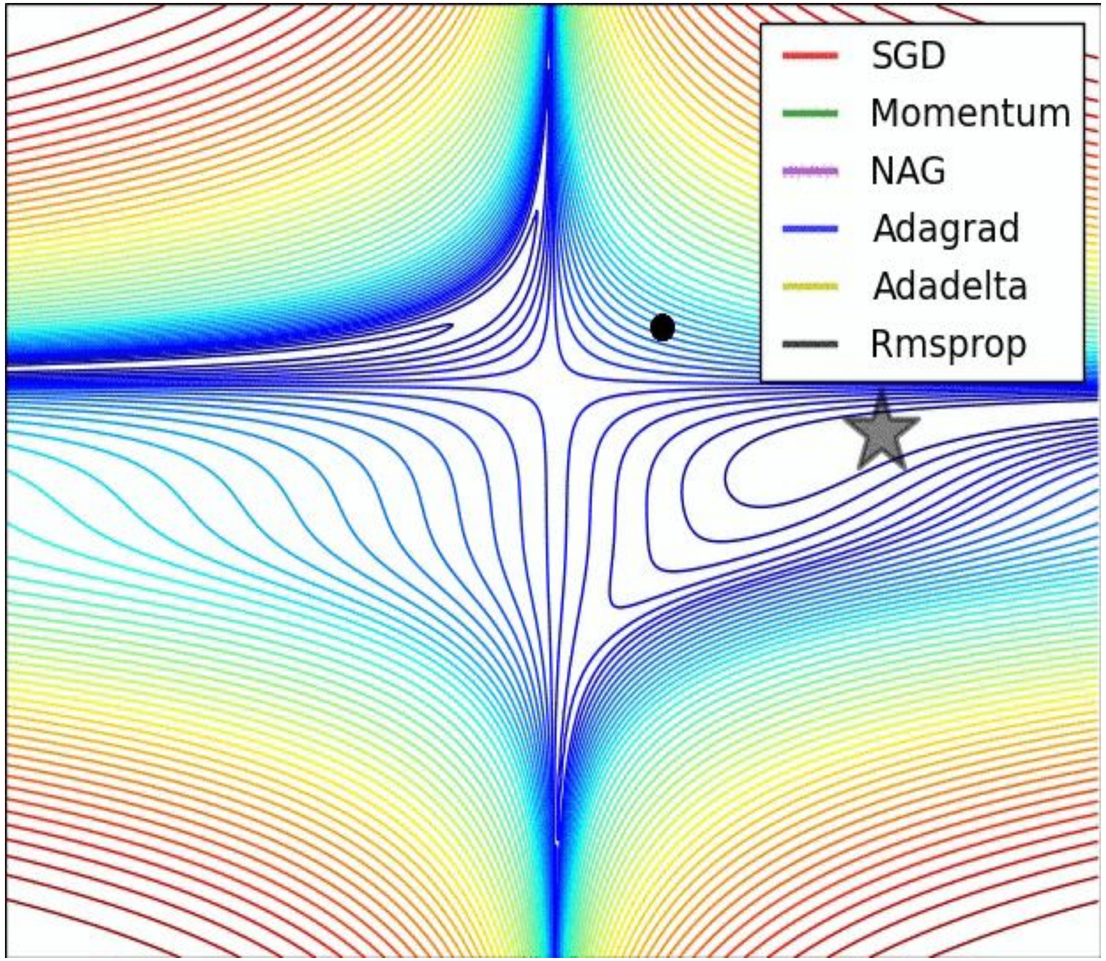
$$\alpha = \alpha_0 e^{-kt}$$

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.
- **Momentum:** is another method to produce better effective gradients.
- ADAGRAD, RMSprop scale the gradient.
- ADAM scales and applies momentum.



(image credits to Alec Radford)

Questions?