CST31211
: Deep Learning

JING WEN

2024

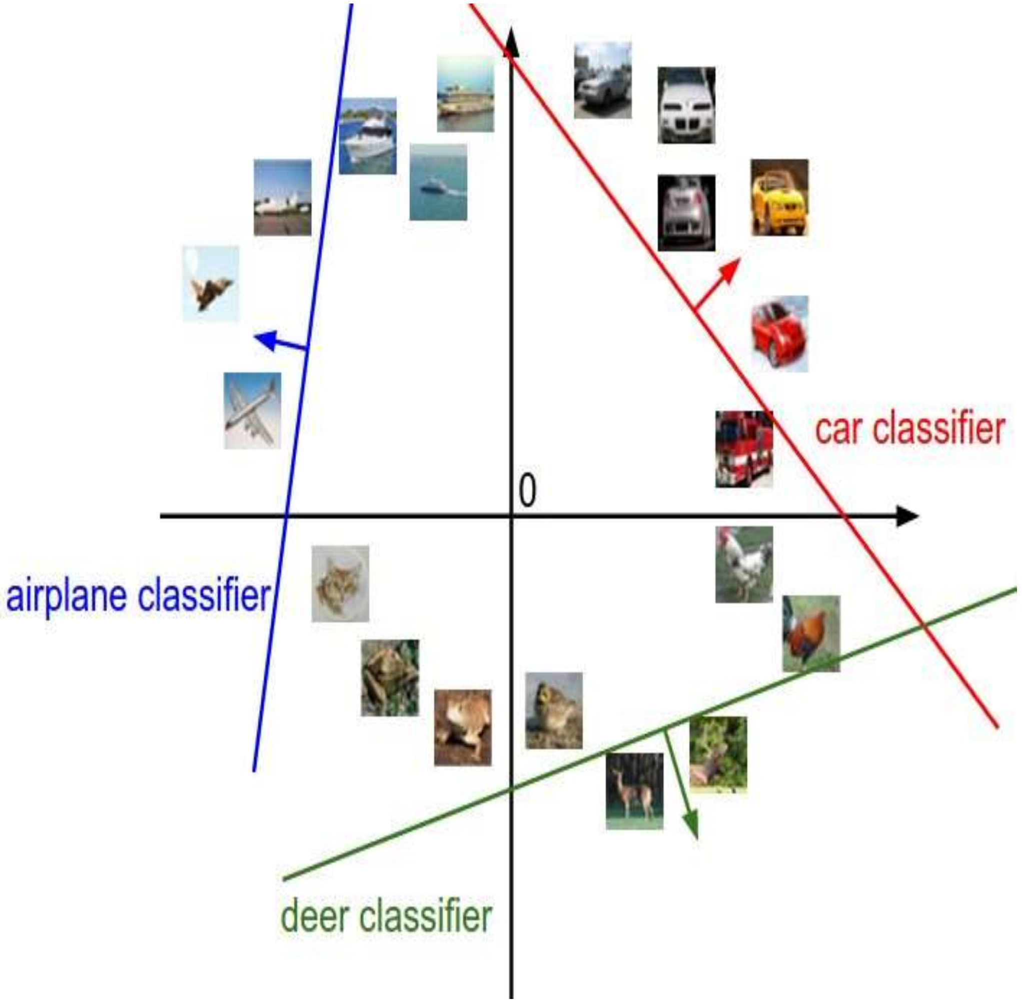
Lecture 2:Optimization

Last Time: Linear Classifier

$$f(x_i, W, b) = Wx_i + b$$

Example trained weights of a linear classifier trained on CIFAR-10:

Interpreting a Linear Classifier



$$f(x_i, W, b) = Wx_i + b$$

[32x32x3]
array of numbers 0...1
(3072 numbers total)

Last Time: Loss Functions

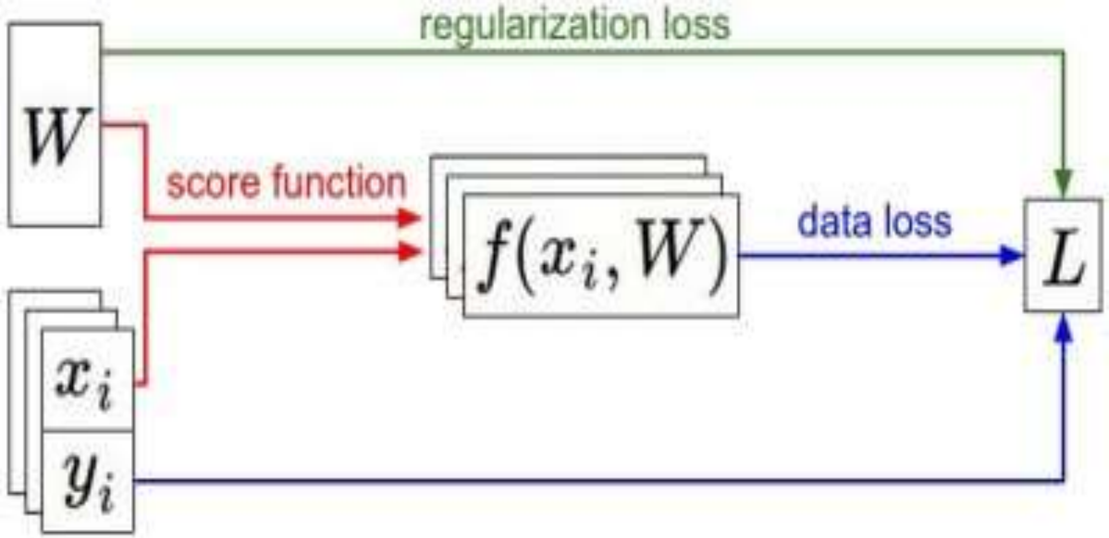
- We have some dataset of (x,y)
- We have a **score function:**
- We have a **loss function**:

e.g. $s = f(x; W) = Wx$

Softmax $L_i = -\log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$

SVM $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$

Full loss $L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$



Optimization

$$w^* = \arg\min_{w} L(w)$$

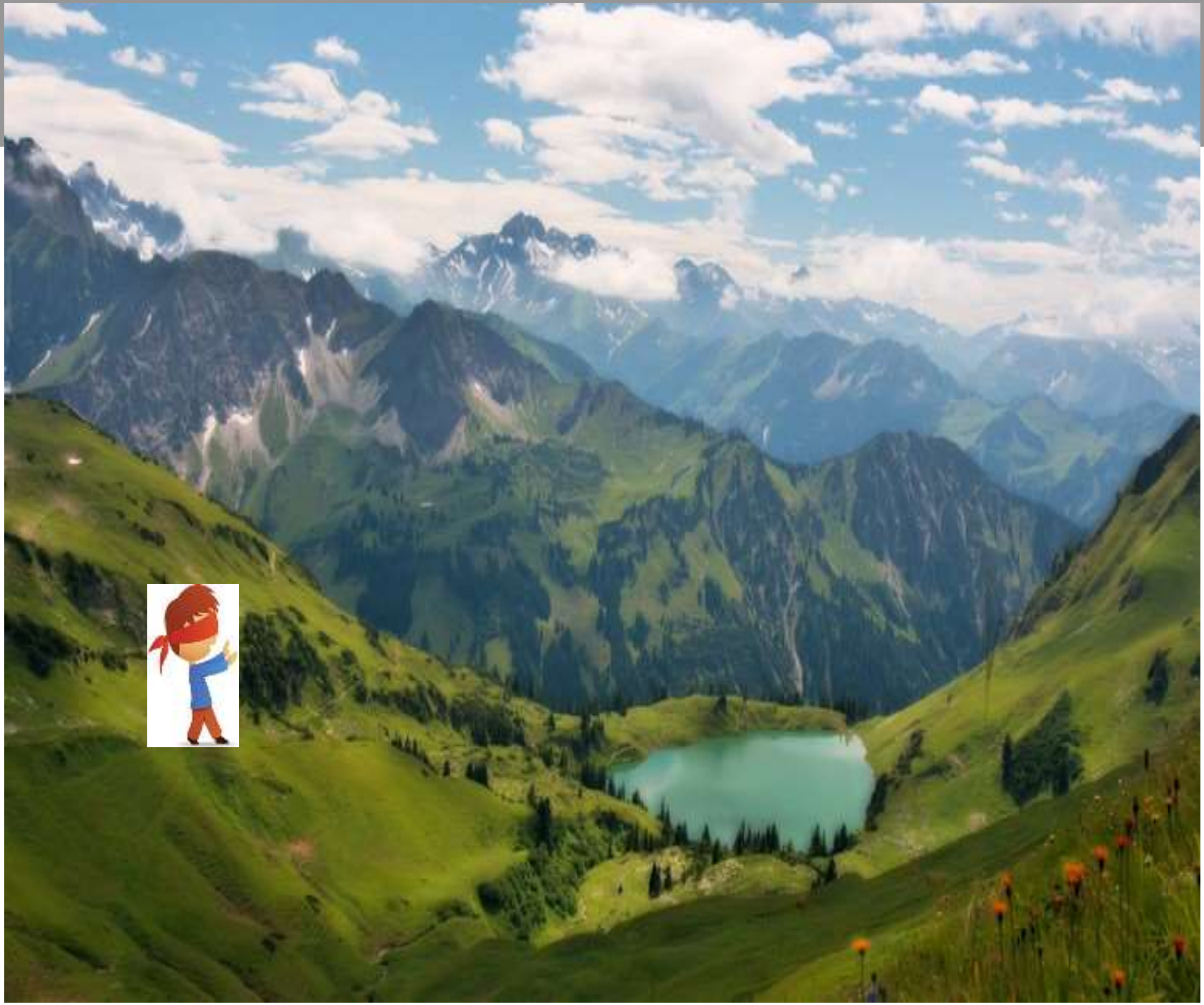
Strategy #1: A first very bad idea solution: **Random search**

```python
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in xrange(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training set
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (truncated: continues for 1000 lines)
```

Lets see how well this works on the test set...

```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

15.5% accuracy! not bad!



Strategy #2: **Follow the slope**

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives).

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^{k} + \lambda_k \boldsymbol{d}^{k}$$

$$\mathrm{x}^{k+1} = xk - \lambda \nabla f(xk)$$

current W:		gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] **loss 1.25347**		[?, ?, ?, ?, ?, ?, ?, ?, ?,…]

current W:	W + h (first dim):	gradient dW:
[0.34,	[0.34 + **0.0001**,	[?,
-1.11,	-1.11,	?,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25322**	

current W:	W + h (first dim):	gradient dW:
[0.34,	[0.34 + **0.0001**,	[**-2.5**,
-1.11,	-1.11,	?,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	
-3.1,	-3.1,	
-1.5,	-1.5,	
0.33,…]	0.33,…]	
loss 1.25347	**loss 1.25322**	

(1.25322 - 1.25347)/0.0001
= -2.5

$$\boxed{\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}}$$

current W:	**W + h** (second dim):	**gradient dW:**
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11 + **0.0001**,	?,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25353**	

current W:	W + h (second dim):	gradient dW:
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11 + **0.0001**,	**0.6**,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	
-1.5,	-1.5,	
0.33,…]	0.33,…]	
loss 1.25347	**loss 1.25353**	

(1.25353 - 1.25347)/0.0001
= 0.6

$$\boxed{\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}}$$

current W:	W + h (third dim):	gradient dW:
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11,	0.6,
0.78,	0.78 + **0.0001**,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25347**	

current W:	W + h (third dim):	gradient dW:
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11,	0.6,
0.78,	0.78 + **0.0001**,	**0**,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?...
-1.5,	-1.5,	
0.33,…]	0.33,…]	
loss 1.25347	**loss 1.25347**	

(1.25347 - 1.25347)/0.0001
= 0

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Evaluating the gradient numerically

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

```
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

Evaluating the gradient numerically

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- approximate
- very slow to evaluate

```python
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

This is silly. The loss is just a function of W:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

This is silly. The loss is just a function of W:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

莱布尼茨

This is silly. The loss is just a function of W:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Calculus

This is silly. The loss is just a function of W:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

$$\nabla_W L = \ldots$$

$$\nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} \mathbb{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)\right) x_i$$

$$\nabla_{w_j} L_i = \mathbb{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

dW = ...
(some function
data and W)

gradient dW:

[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,…]

In summary:

- Numerical gradient（数值梯度）: approximate, slow, easy to write

- Analytic gradient（解析梯度）: exact, fast, error-prone

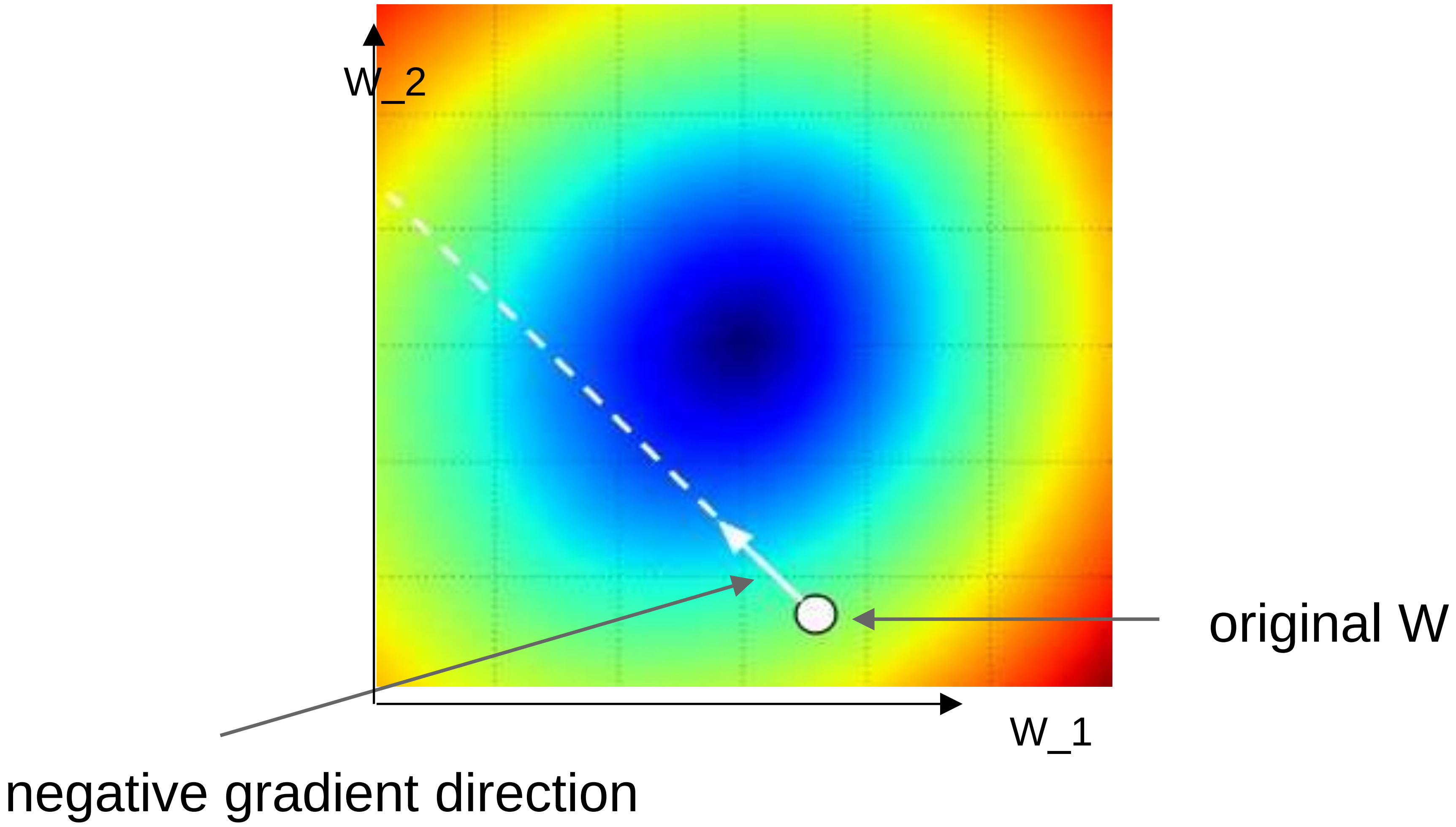
=>

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check.**

Gradient Descent

$$x^{k+1} = xk - \lambda \nabla f(xk)$$

```
# Vanilla Gradient Descent

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


W_2
W_1
original W
negative gradient direction

Gradient Descent

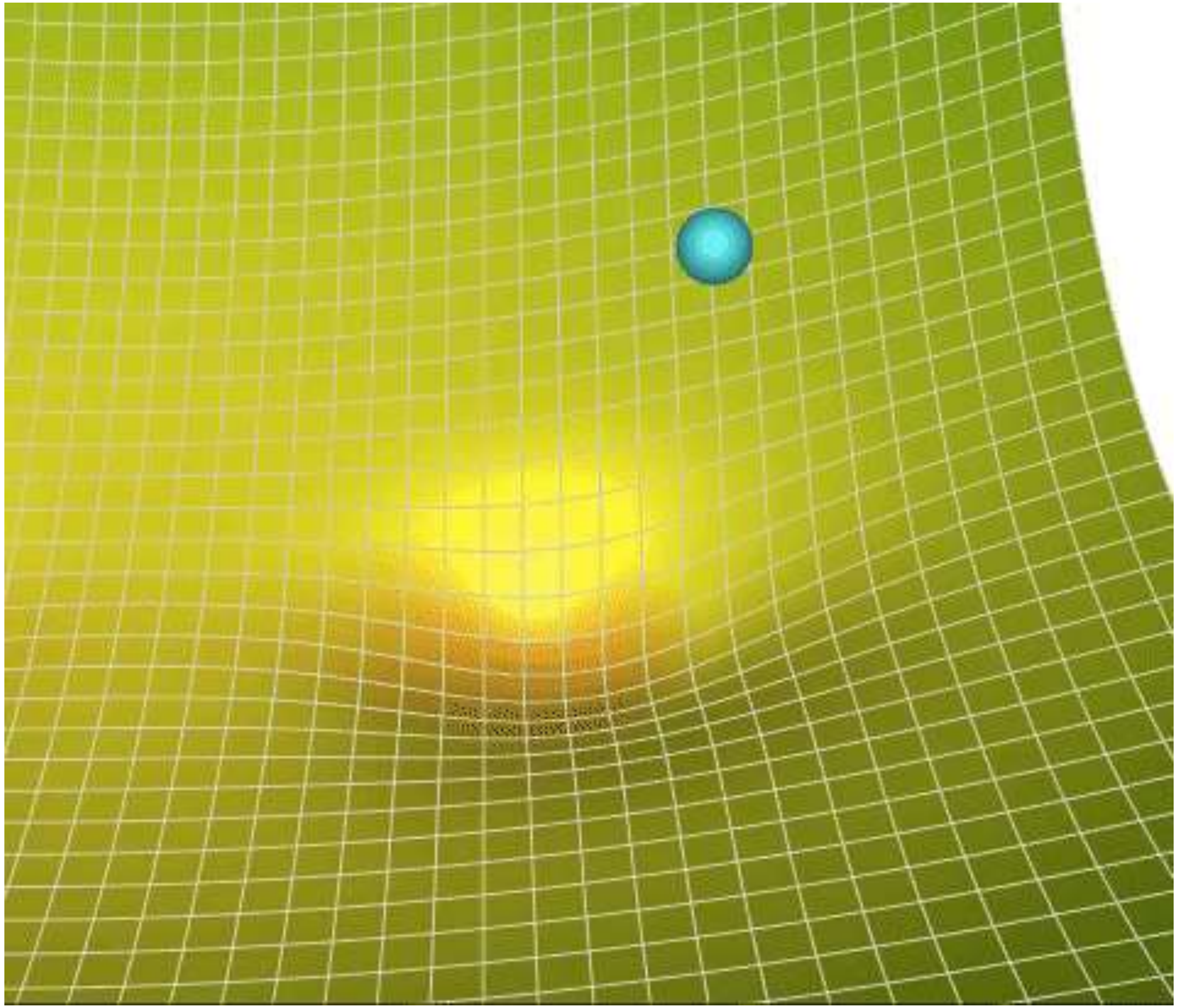
- SGD （随机梯度下降）
- MBGD （小批量梯度下降）
- BGD （批量梯度下降）

Stochastic Gradient Descent (SGD)

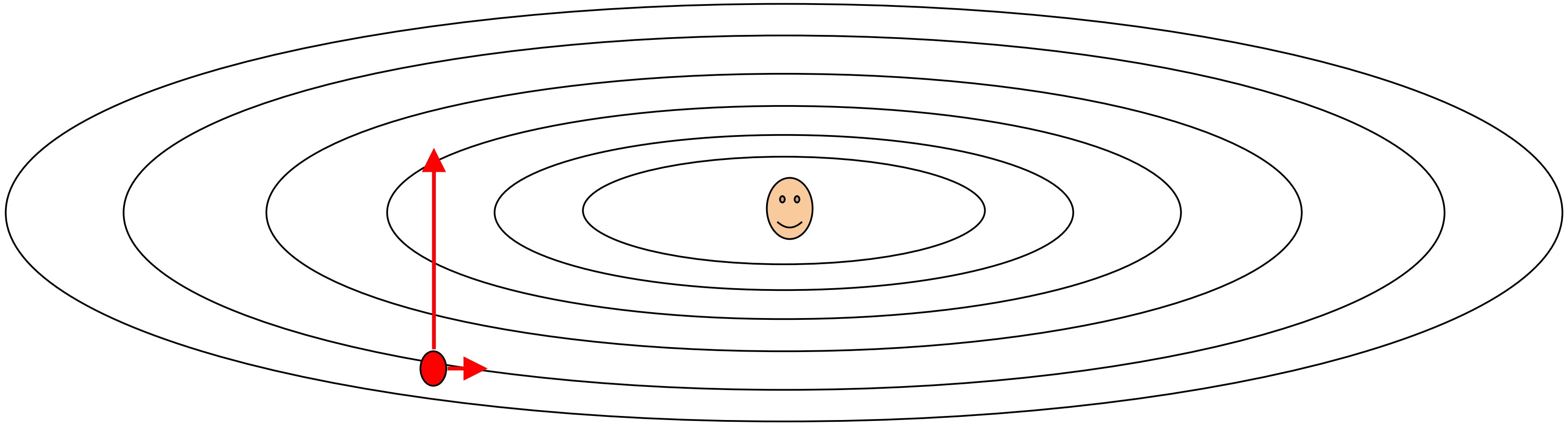
$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

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

SGD

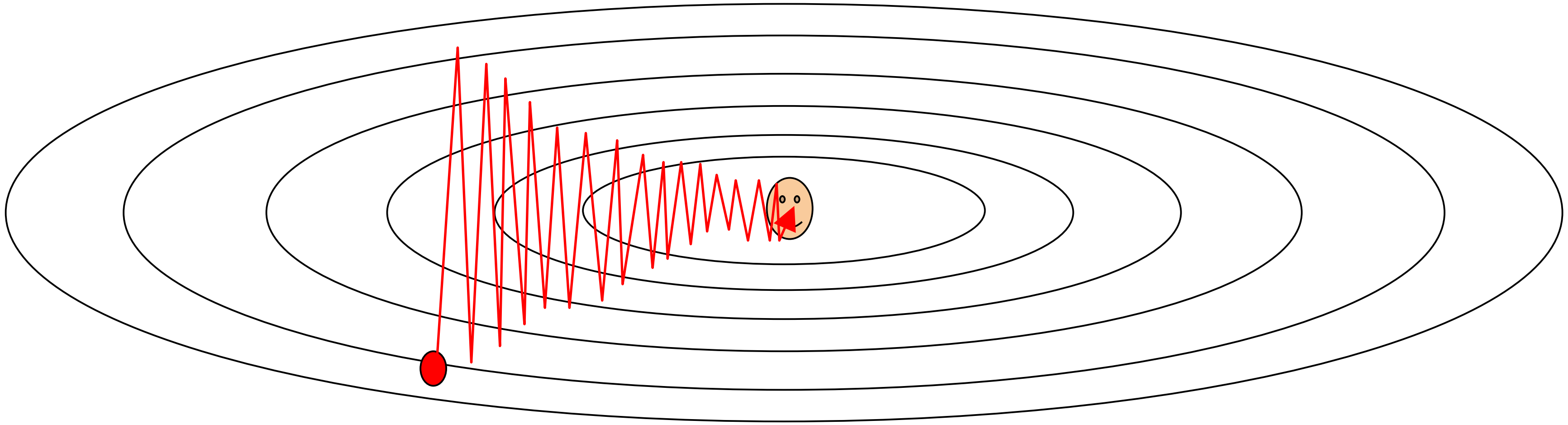


poor gradients:



Q: What is the trajectory along which we converge towards the minimum with SGD?

poor gradients:



Q: What is the trajectory along which we converge towards the minimum with SGD? very slow progress along flat direction, jitter along steep one

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

Mini-batch Gradient Descent

如何更新权重?
假设输入两个数据，权重为5个

数据1（x1）： Loss1 ， l1,gradients1=(1.5,-2.0,1.1,0.4,-0.9)
数据2（x2）： Loss2， l2， gradients2=(1.2,2.3,-1.1,-0.8,-0.7)

则更新权重时两种方法：

1. 求上面两个数据所求出来的梯度(gradients1， gradients2) 的平均值更新权重：

gradients_result = (gradients1， gradients2)/2 = (1.35， 0.15， 0， -0.2， -0.8)

2. 求两个损失的平均值计算梯度，再更新权重（tensorflow)

Mini-batch Gradient Descent

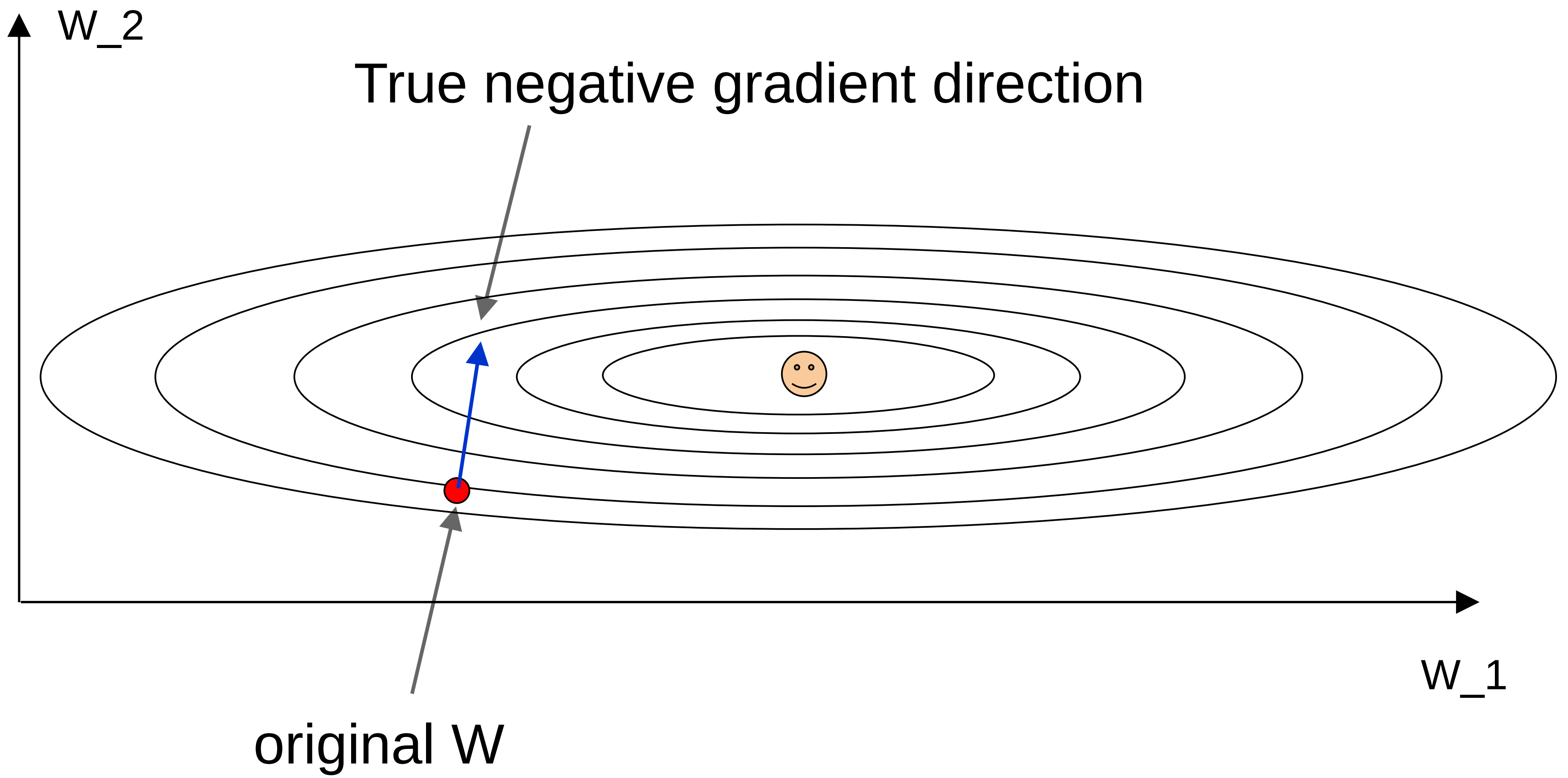
- only use a small portion of the training set to compute the gradient.

```
# 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
```

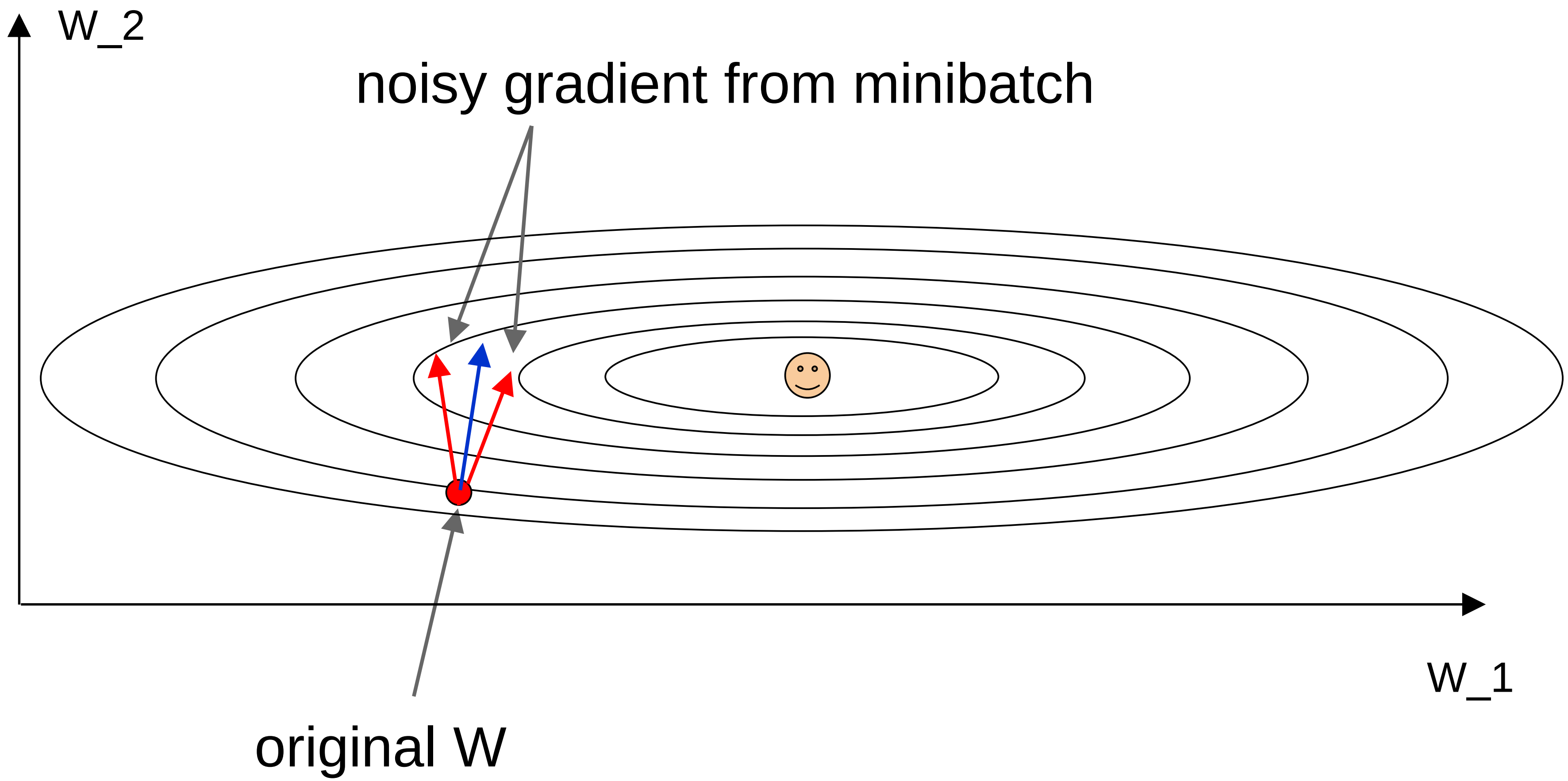
we will look at more fancy update formulas (momentum, Adagrad, RMSProp, Adam, …)

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

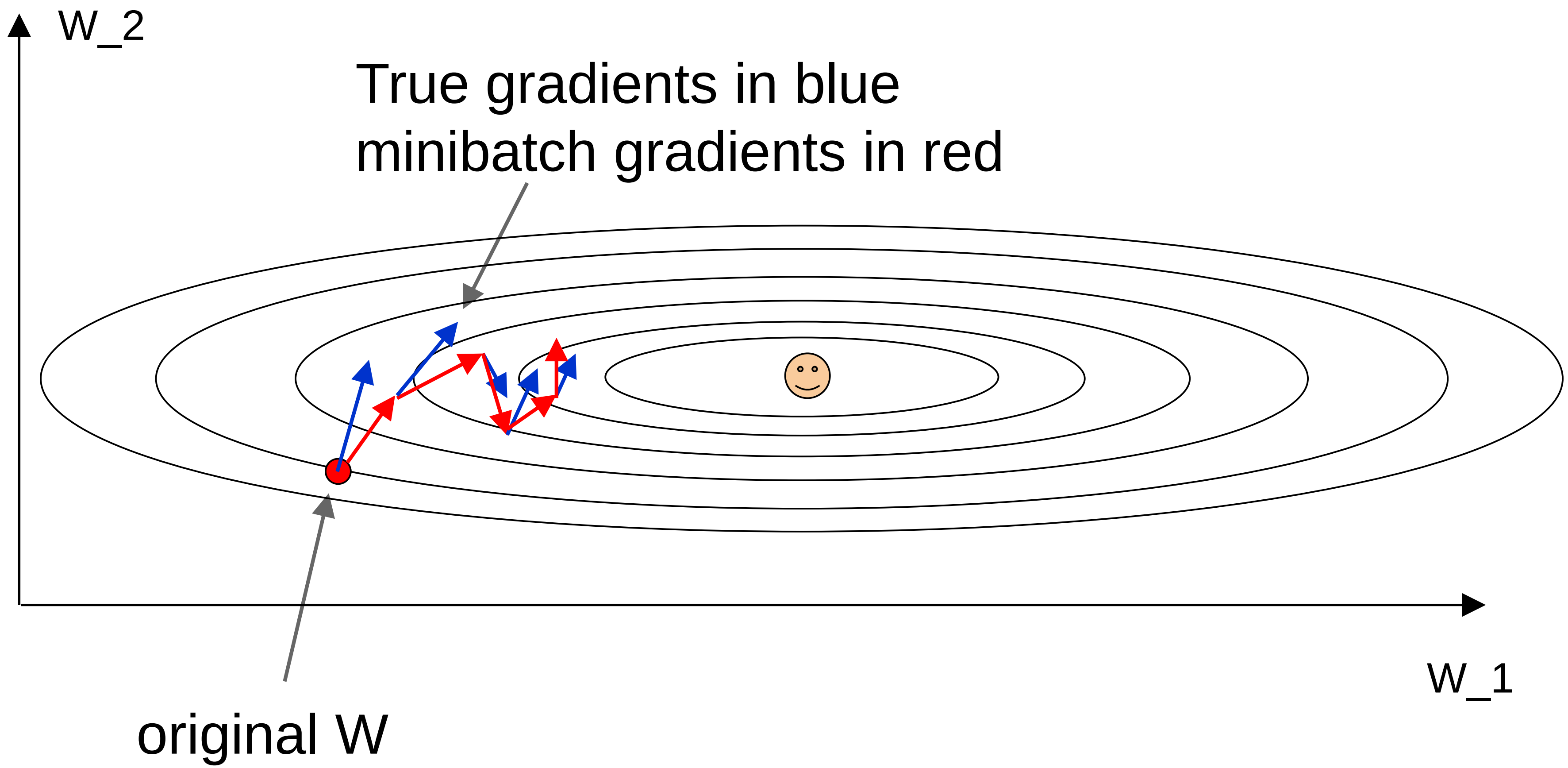
Minibatch updates



Mini-batch Gradient Descent

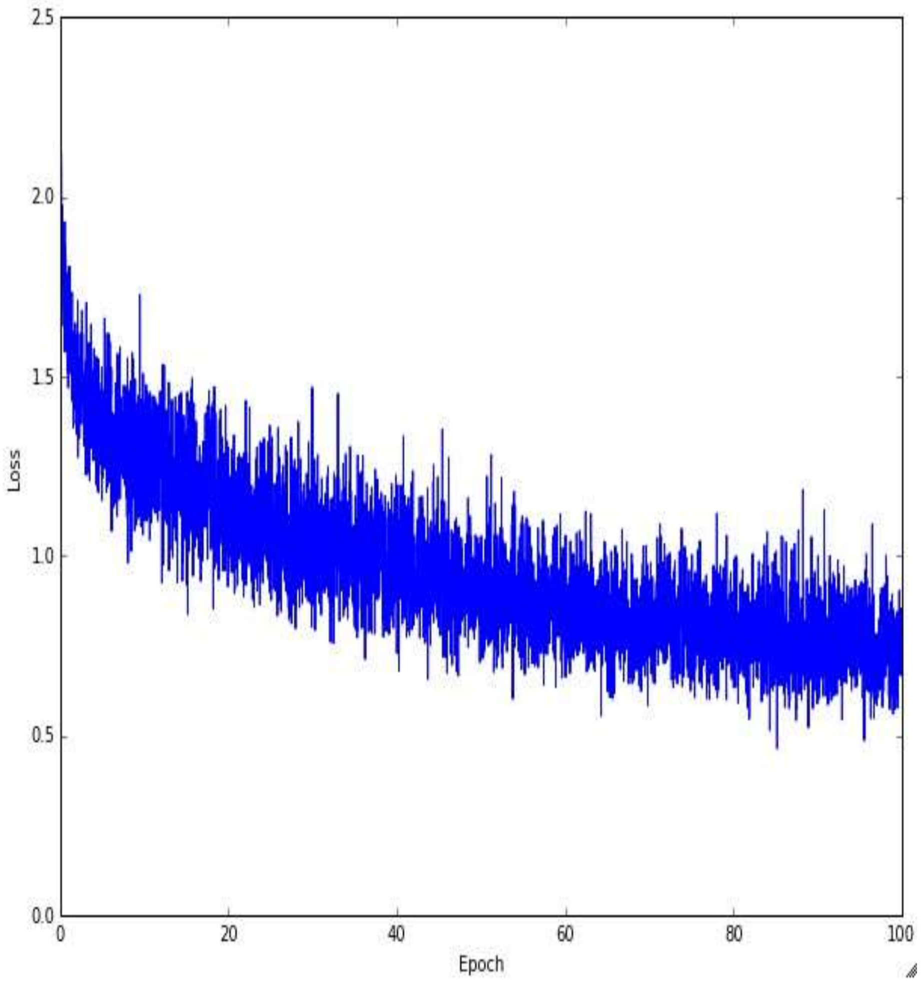


Mini-batch Gradient Descent



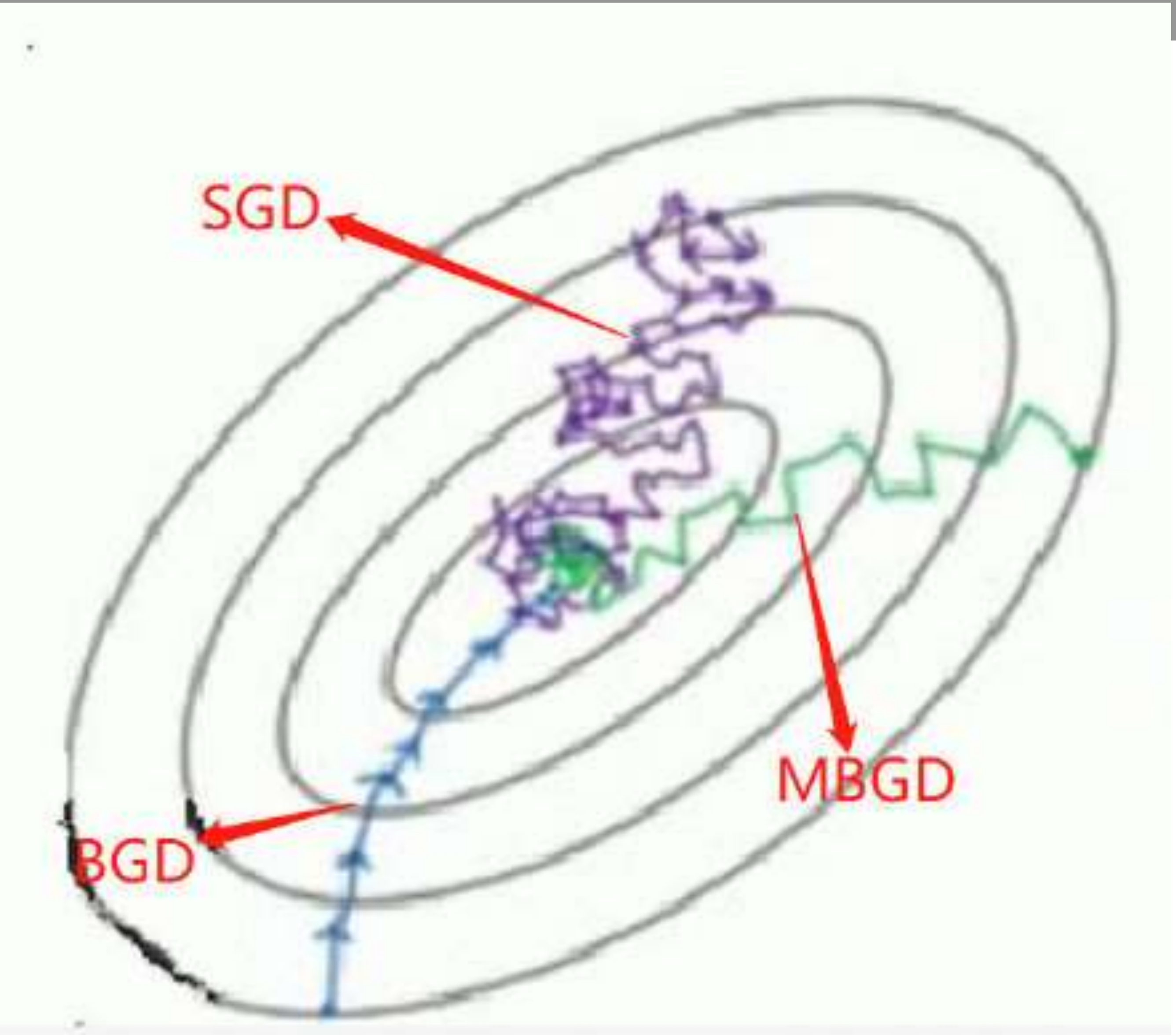
Gradients are noisy but still make good progress on average

Mini-batch Gradient Descent



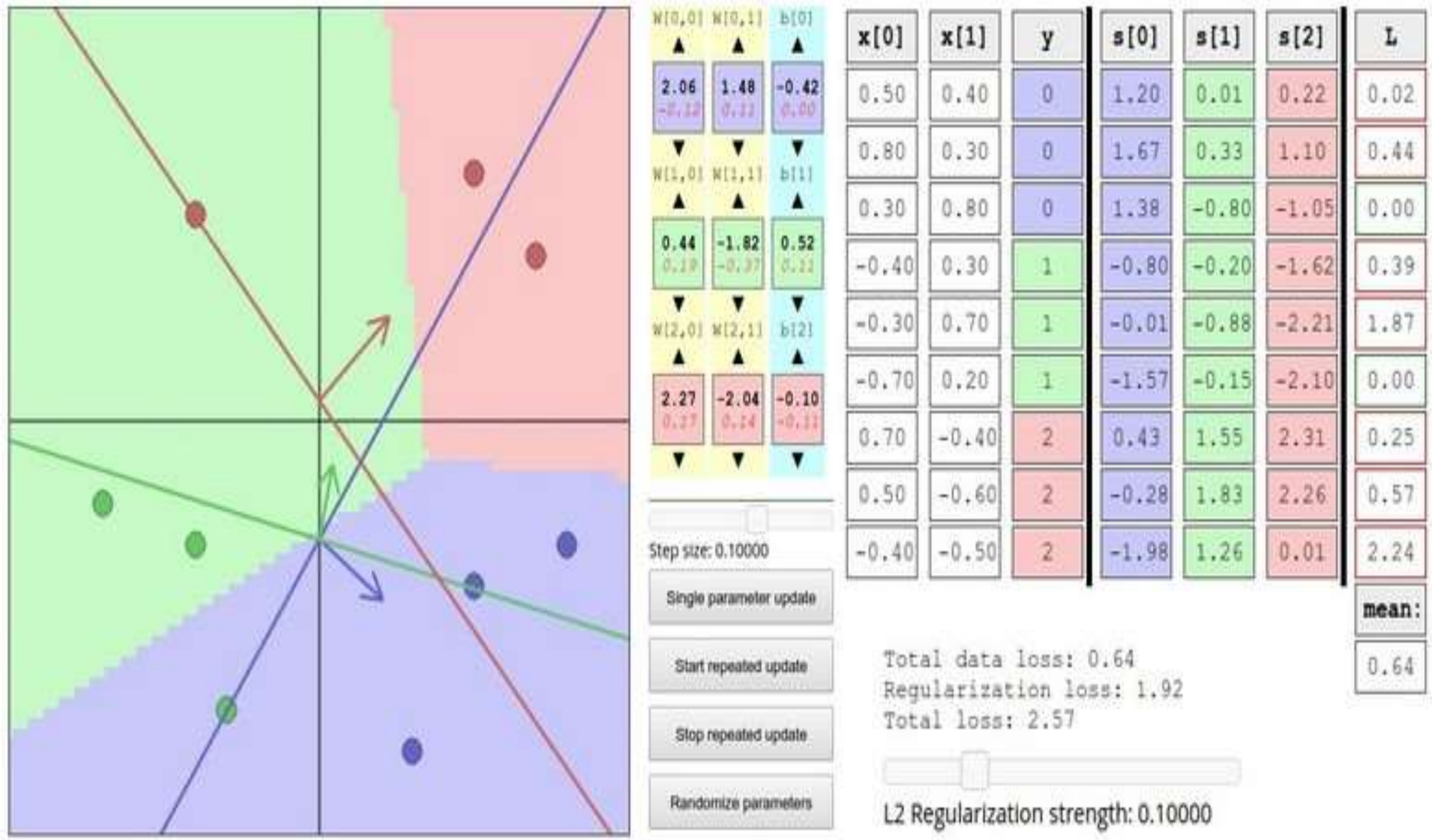
Example of optimization progress while training a neural network.

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


SGD
MBGD
BGD

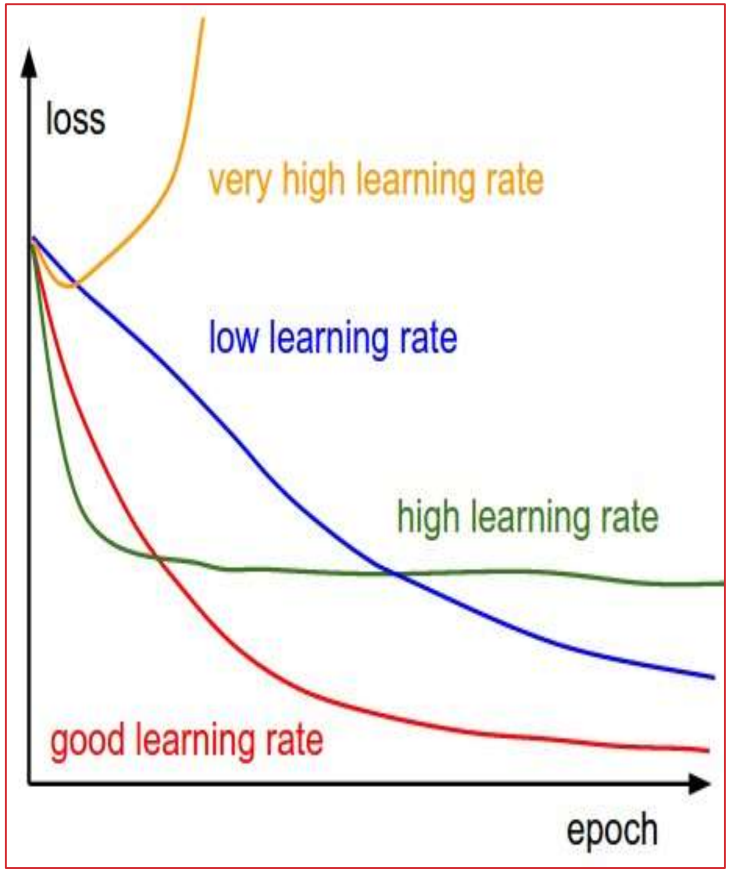
	BGD（批量）	SGD（随机）	MBGD（小批量）
优点	非凸函数可保证收敛至全局最优解	计算速度快	计算速度快，收敛稳定
缺点	计算速度缓慢，不允许新样本中途进入	计算结果不易收敛，可能会陷入局部最优解中	-

Interactive Web Demo time....



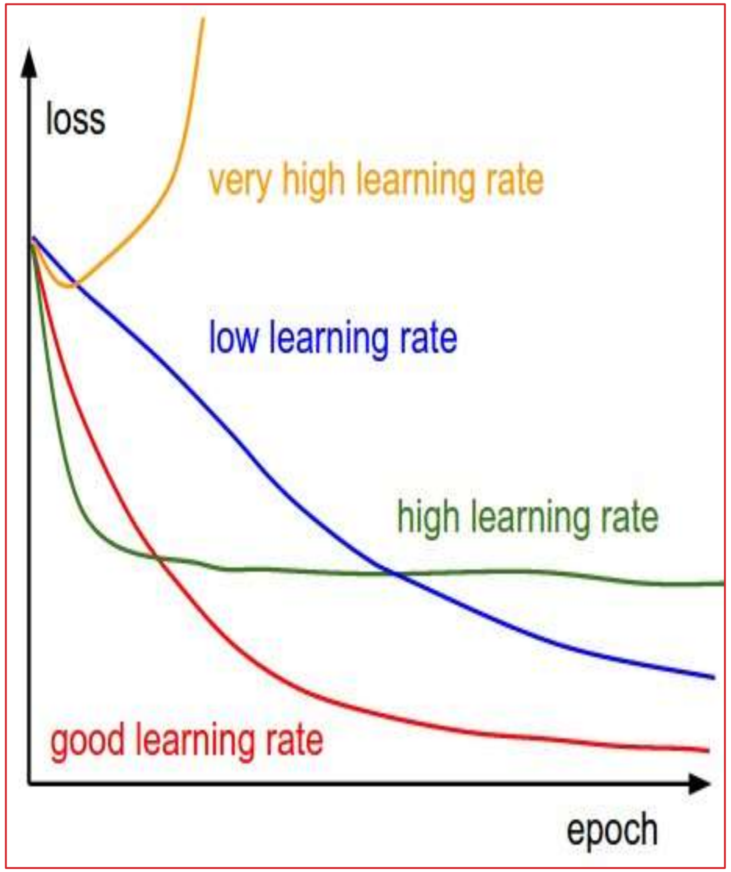
http://vision.stanford.edu/teaching/cs231n/linear-classify-demo/

learning rate as a hyperparameter.



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

learning rate as a hyperparameter.



=> Learning rate decay over time!

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

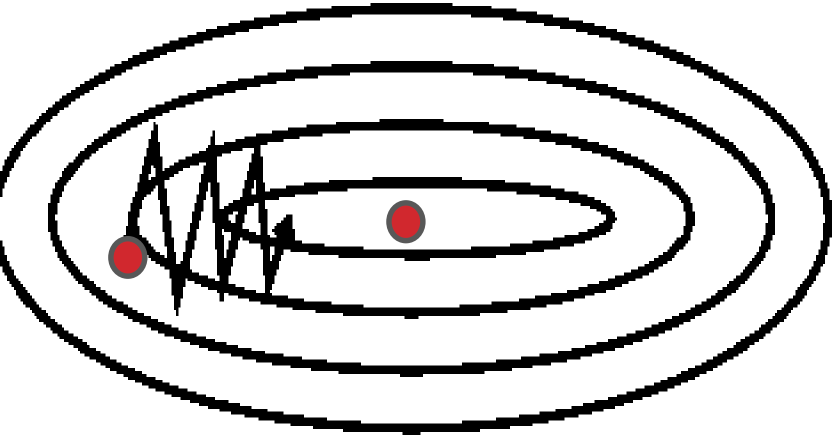
exponential decay:

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

1/t decay:

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

Momentum update



SGD

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

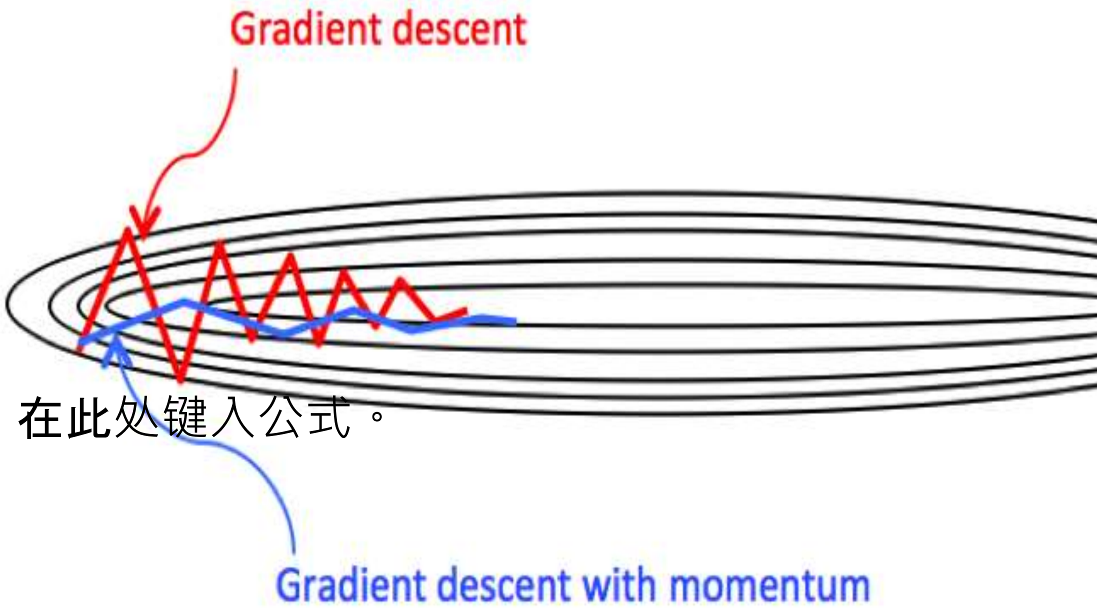
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- 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
```



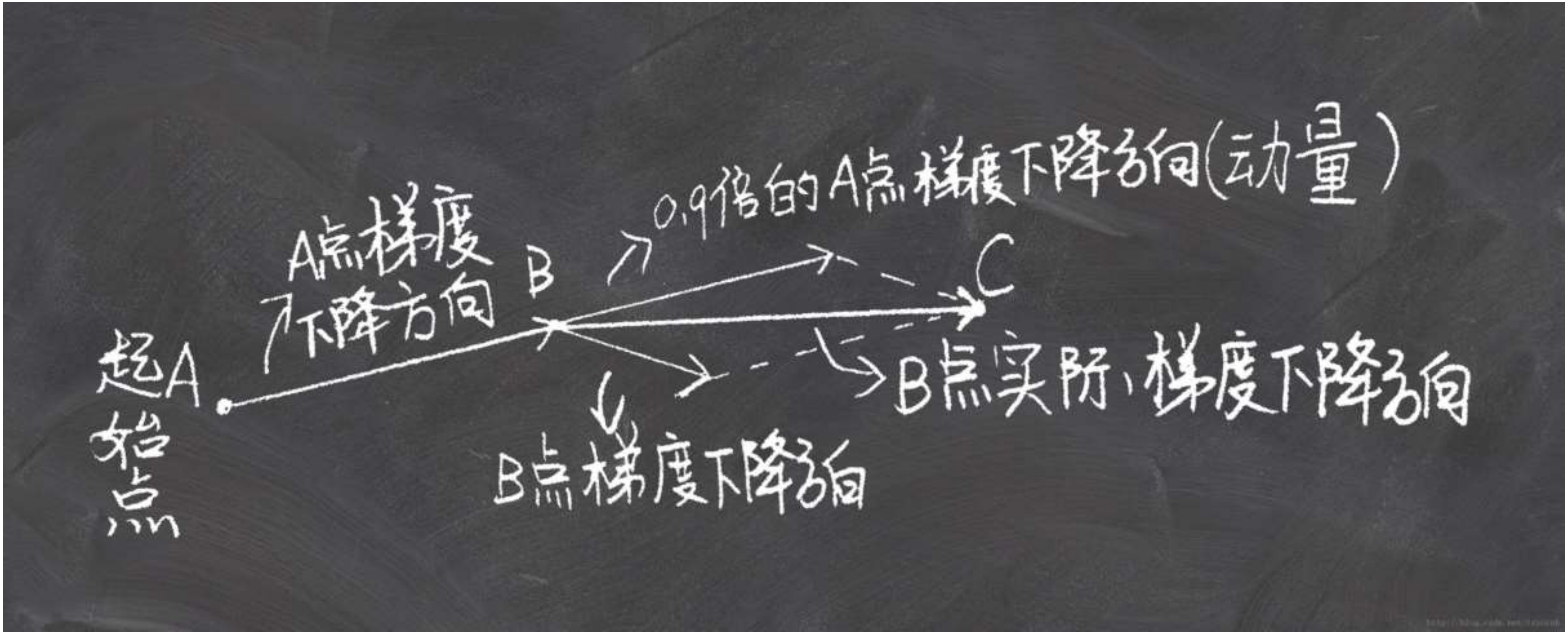
在此处键入公式。

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

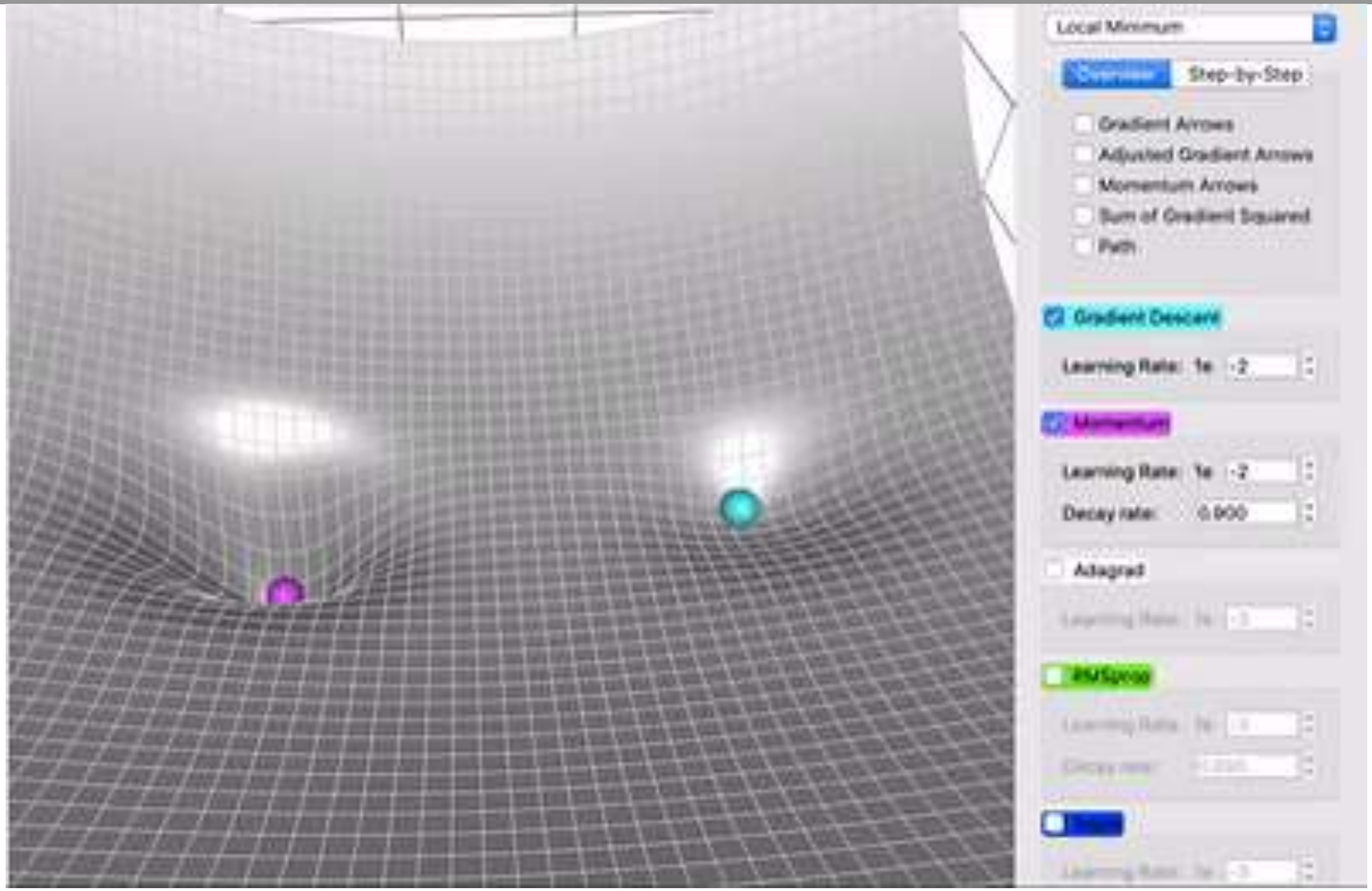
$$V_t = \mu * V_{t-1} - (1-\mu) * l * \nabla f(W_{t-1})$$

$$W_t = W_{t-1} + V_t$$

Momentum update



Momentum update



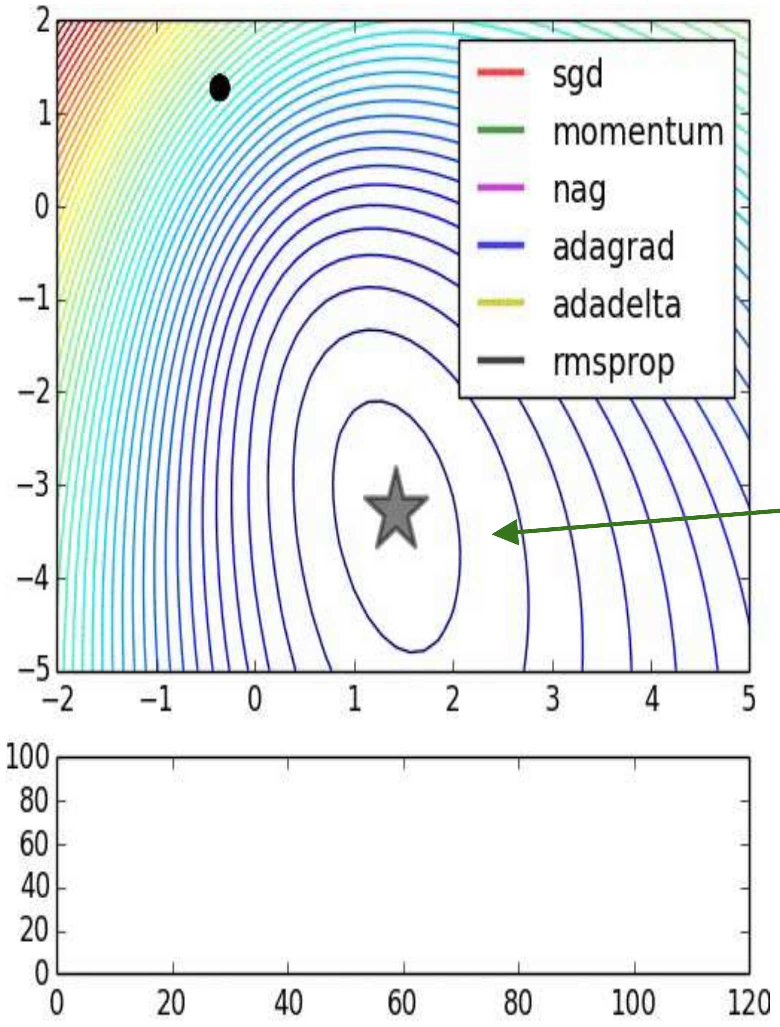
优点：
1. 动量移动得更快(因为它积累的所有动量)
2. 动量有机会逃脱局部极小值(因为动量可能推动它脱离局部极小值)。

Momentum update

SGD

vs

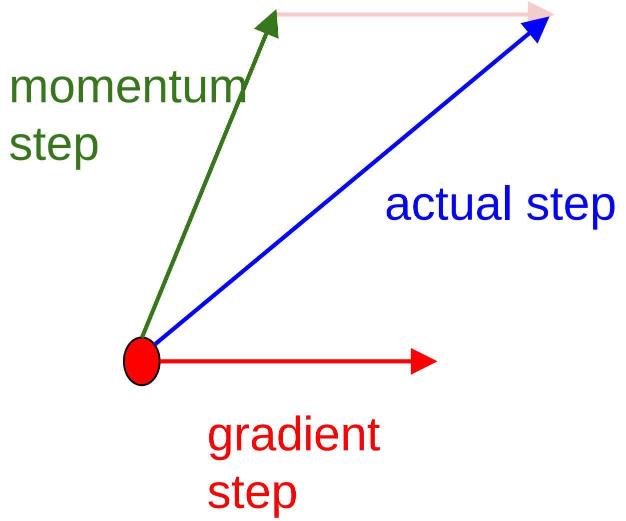
Momentum



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

Nesterov Momentum update

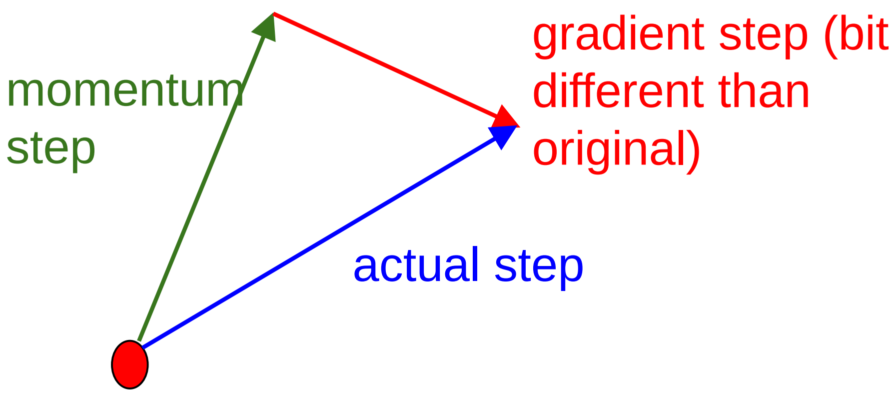
Momentum update



$$V_t = \mu * V_{t-1} - (1-\mu) * l * \nabla f(W_{t-1})$$

$$W_t = W_{t-1} + \alpha V_t$$

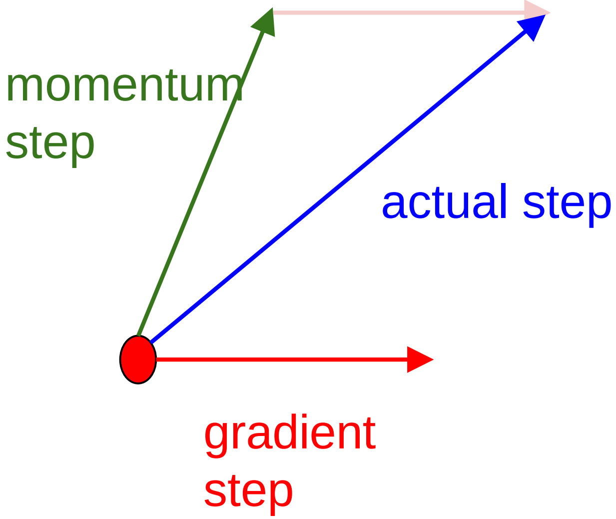
Nesterov momentum update

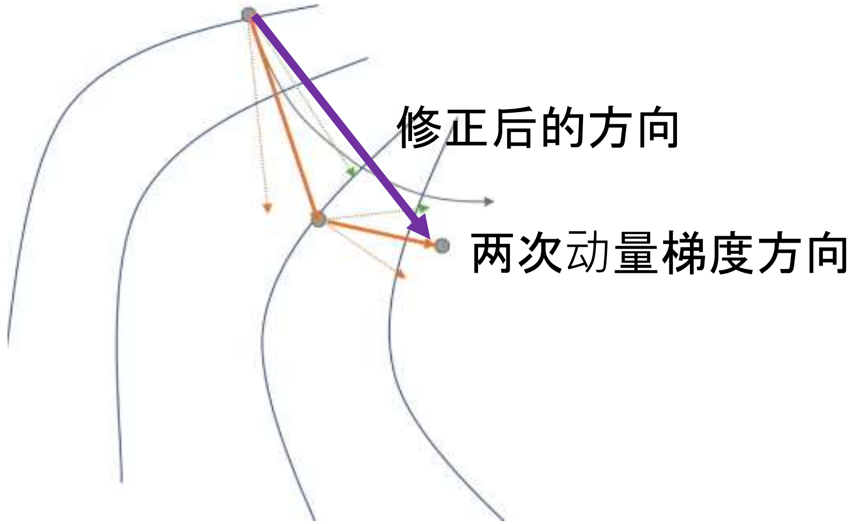


Nesterov: the only difference...

Nesterov Momentum(牛顿动量) update（NAG）

Ordinary momentum update:





$$V_t = \mu * V_{t-1} - (1-\mu) * l * \nabla f(W_{t-1})$$

$$V_t = \mu * V_{t-1} - (1-\mu) * l * \nabla f\boxed{(W_{t-1} \boxed{+ \gamma V_{t-1}})}$$

$$W_t = W_{t-1} + \alpha V_t$$

迭代公式，即点的移动

Nesterov Momentum update

$$V_t = \mu * V_{t-1} - \varepsilon * \nabla f\left(\boxed{W_{t-1} + \mu V_{t-1}}\right)$$

$$W_t = W_{t-1} + V_t$$

Slightly inconvenient…
usually we have :

Variable transform and rearranging saves the day:

$$\emptyset_{t-1} = W_{t-1} + \mu V_{t-1}$$

Replace, rearrange and obtain:

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

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

令$\emptyset_t = W_t$

$$\boxed{W_t = W_{t-1} - \mu v_{t-1} + (1 + \mu) v_t}$$

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

Nesterov Momentum update

证明：

$$V_{t+1} = \beta V_t - \alpha \nabla_{\theta_t} L(\theta_t + \beta V_t)$$

NAG=
Nesterov Accelerated Gradient

$$\theta_{t+1} = \theta_t + V_{t+1}$$

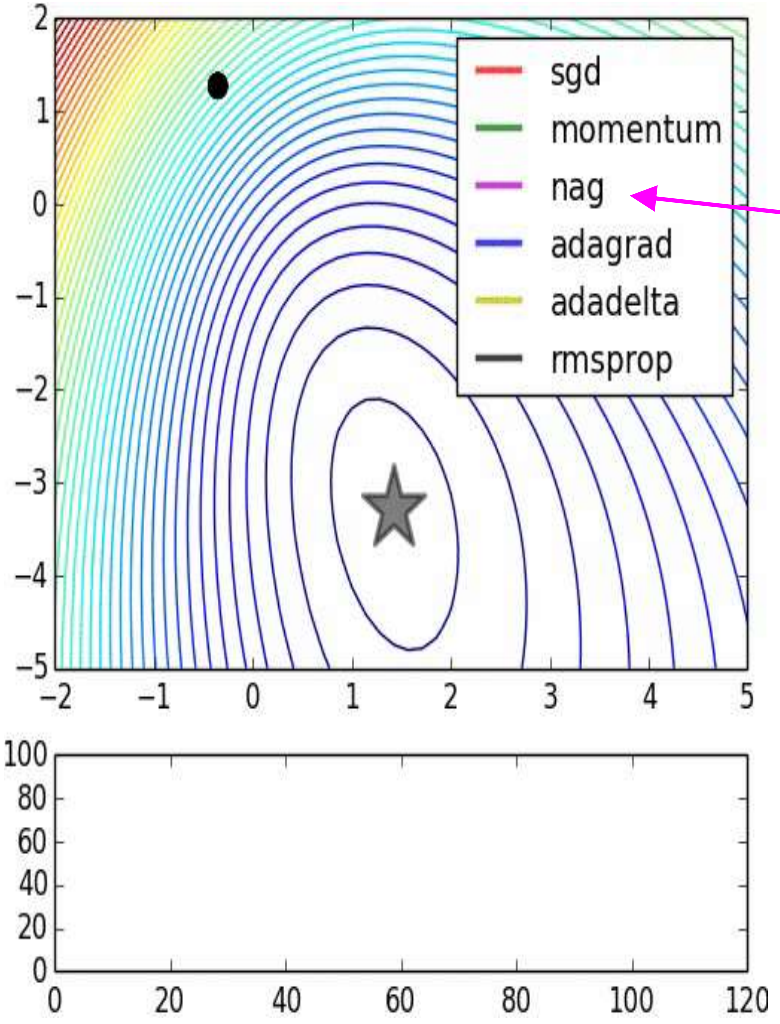
令 $\theta_t' = \theta_t + \beta V_t$，则 $V_{t+1} = \beta V_t - \alpha \nabla_{\theta_t} L(\theta_t')$

$$\begin{aligned}
\theta_{t+1}' &= \theta_{t+1} + \beta V_{t+1} \\
&= \theta_t + V_{t+1} + \beta V_{t+1} \\
&= \theta_t + (1+\beta) V_{t+1} \\
&\boxed{= \theta_t' - \beta V_t + (1+\beta)[\beta V_t - \alpha \nabla_{\theta_t'} L(\theta_t')]} \\
&= \theta_t' + \beta^2 V_t - (1+\beta)\alpha \nabla_{\theta_t'} L(\theta_t')
\end{aligned}$$

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

令 $\theta_t = \theta_t'$，则上述公式为：$\theta_{t+1} = \theta_t + \beta^2 V_t - (1+\beta)\alpha \nabla_{\theta_t} L(\theta_t)$

Nesterov Momentum update



nag = Nesterov Accelerated Gradient

AdaGrad （Adaptive Gradient） update

[Duchi et al., 2011]

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

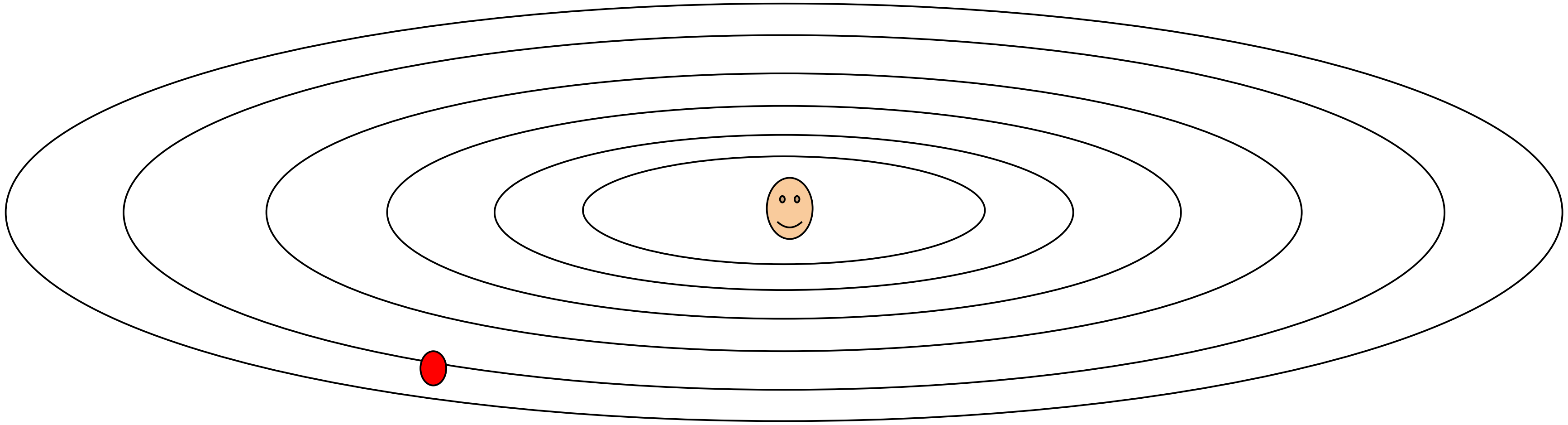
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

$$S_t = S_{t-1} + \nabla W \cdot \nabla W$$

$$W_t = W_{t-1} - \frac{l}{sqrt(S_t) + \varepsilon} \nabla W$$

AdaGrad update

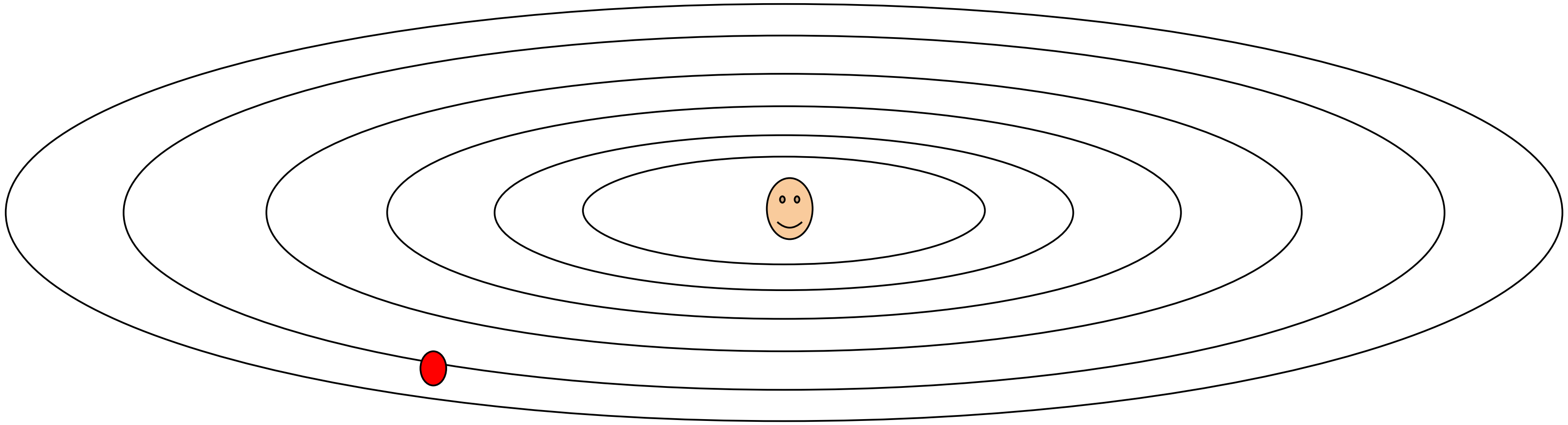
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Q: What happens with AdaGrad?

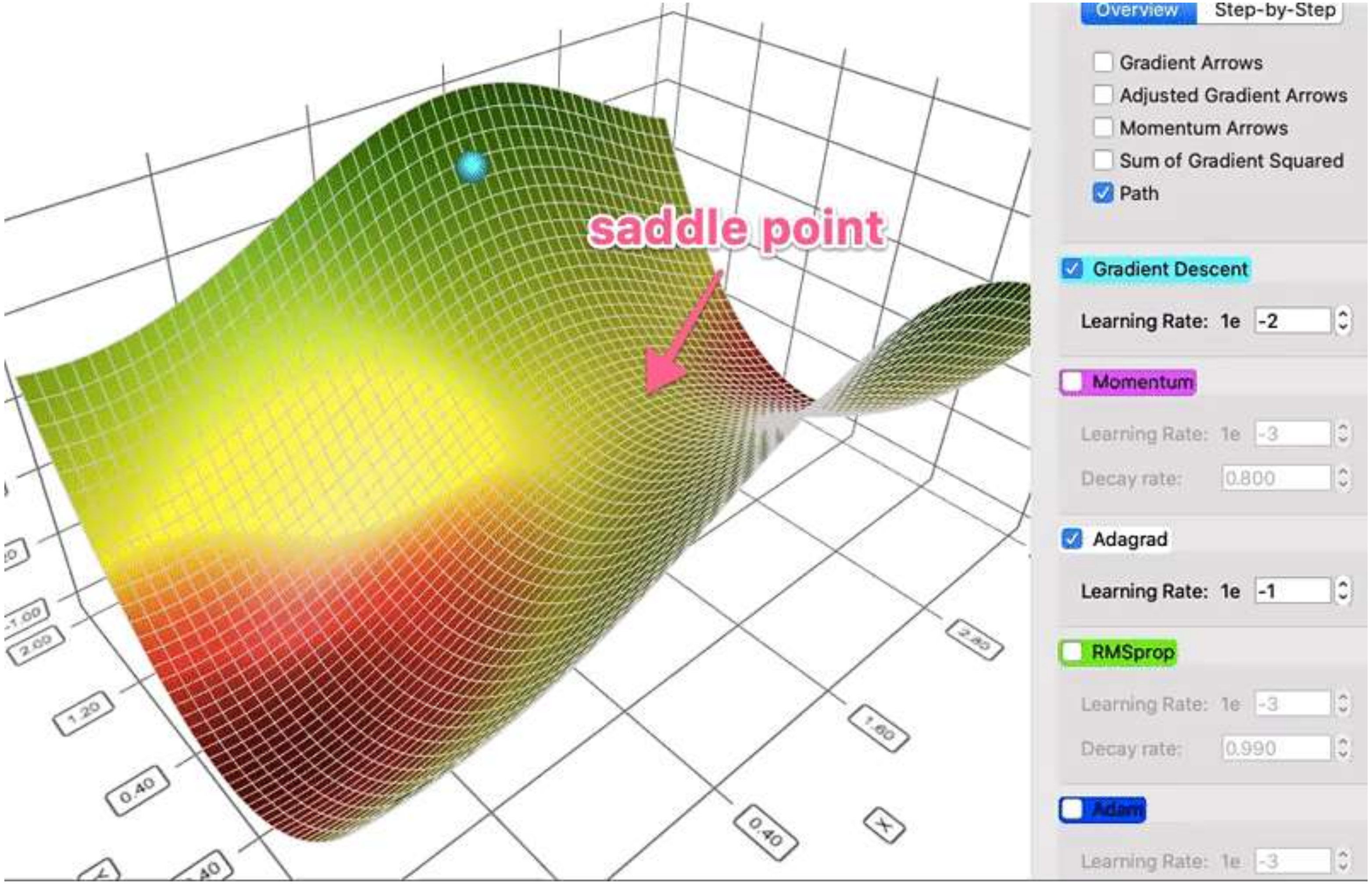
AdaGrad update

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

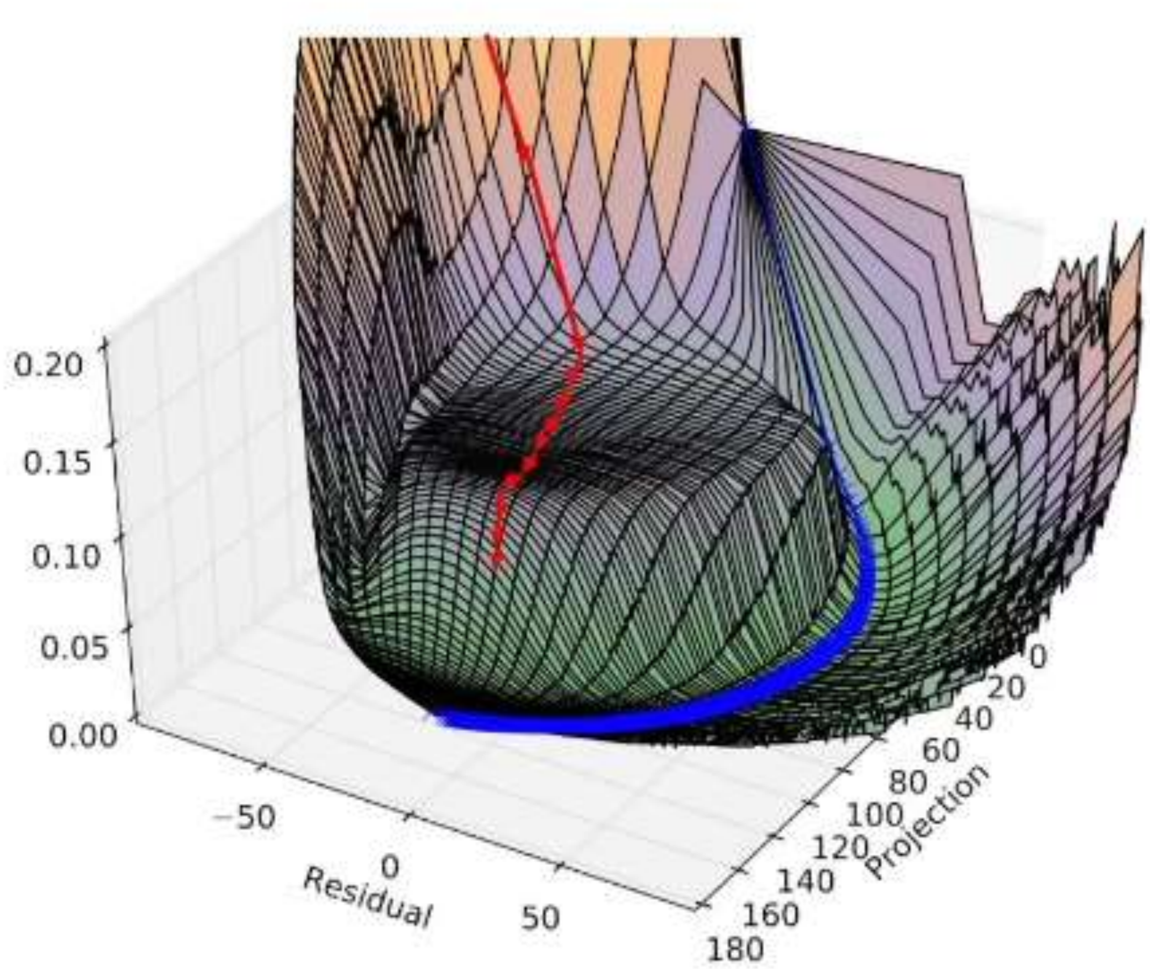


Q2: What happens to the step size over long time?

中后期，分母上梯度累加的平方和会越来越大，使得参数更新量趋近于0，使得训练提前结束，无法学习



http://github.com/lilipads/gradient_descent_viz


0.20
0.15
0.10
0.05
0.00
−50
0
Residual
50
0
20
40
60
80
100
120
140
160
180
Projection

red:adagrad

RMSProp update

[Tieleman and Hinton, 2012]

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

$$S_t = S_{t-1} + \nabla W \cdot \nabla W$$

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

$$S_t = \beta S_{t-1} + (1-\beta)\ \nabla W \cdot \nabla W$$

$$W_t = W_{t-1} - \frac{l}{sqrt(S_t) + \varepsilon} \nabla W$$

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

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

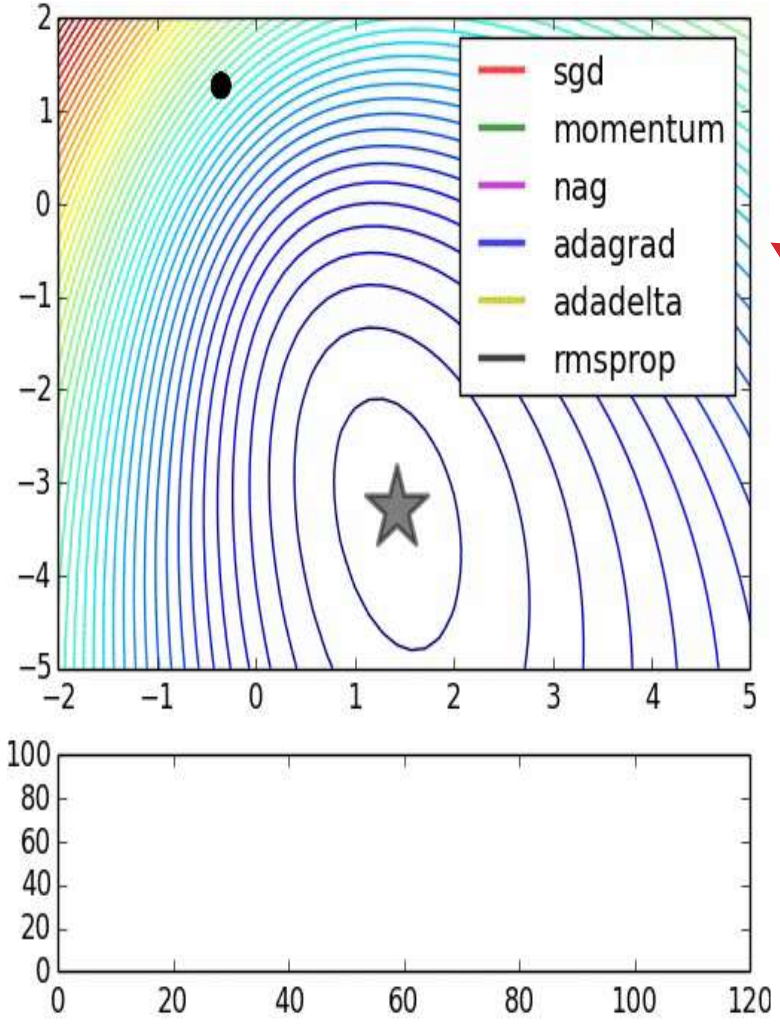
$$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t-1) + 0.1\left(\partial E/\partial w\,(t)\right)^2$$

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

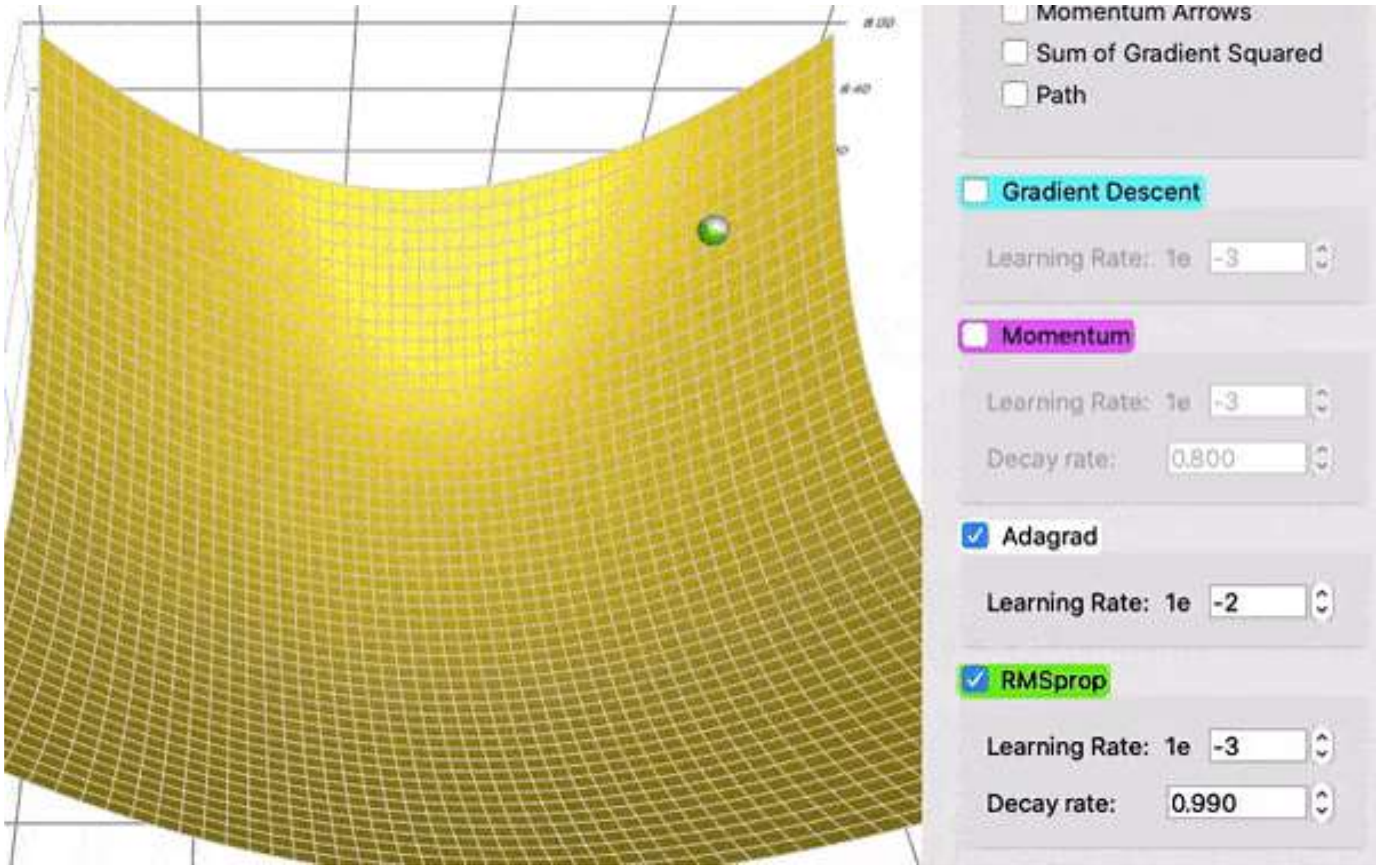
Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

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)
```

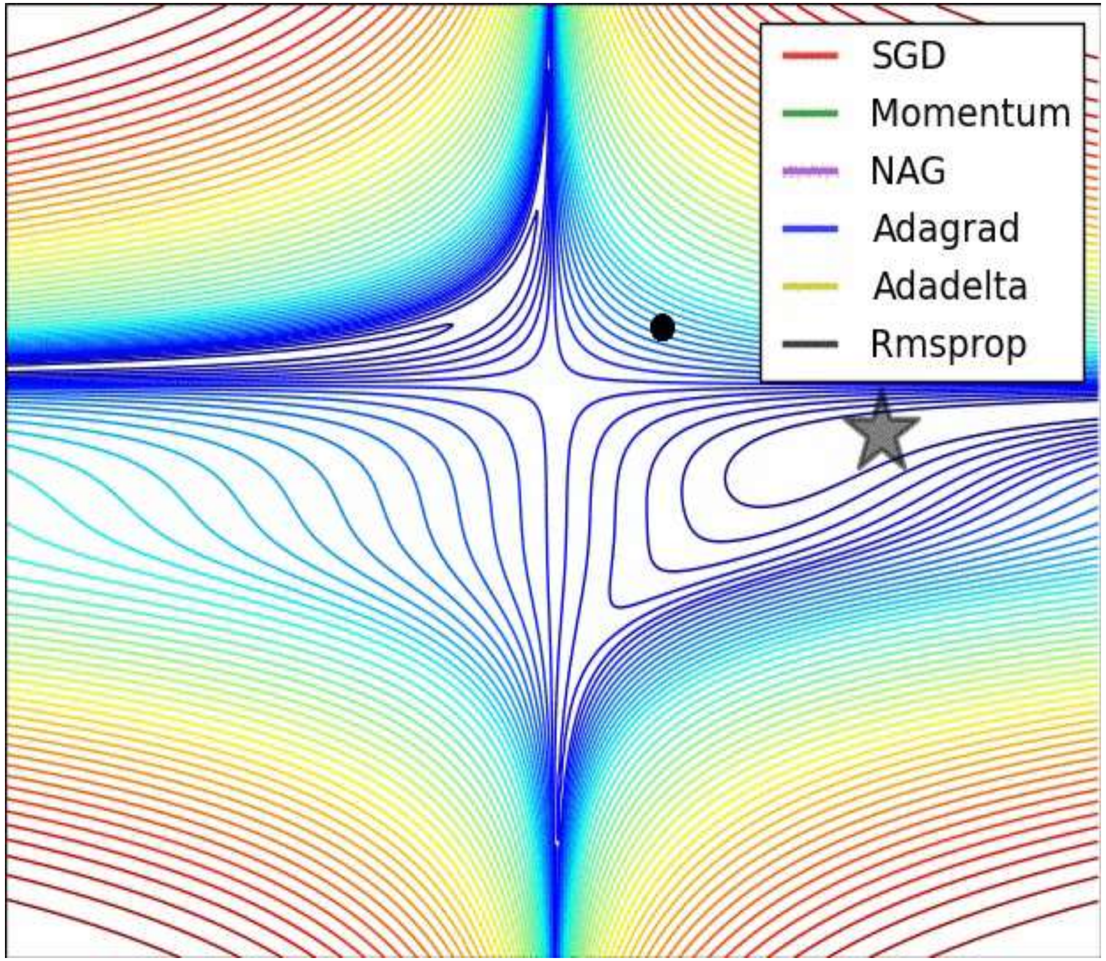
momentum

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".

The effects of different update formulas



(image credits to Alec Radford)

Gradient Descent Visualization

https://github.com/lilipads/gradient_descent_viz

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.