

# Lecture 5: More Optimization

# Gradient Descent

Iteratively step in the direction of  
the negative gradient  
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
    dw = compute_gradient(loss_fn, data, w)
    w -= learning_rate * dw
```

## Hyperparameters:

- Weight initialization method
- Number of steps
- Learning rate

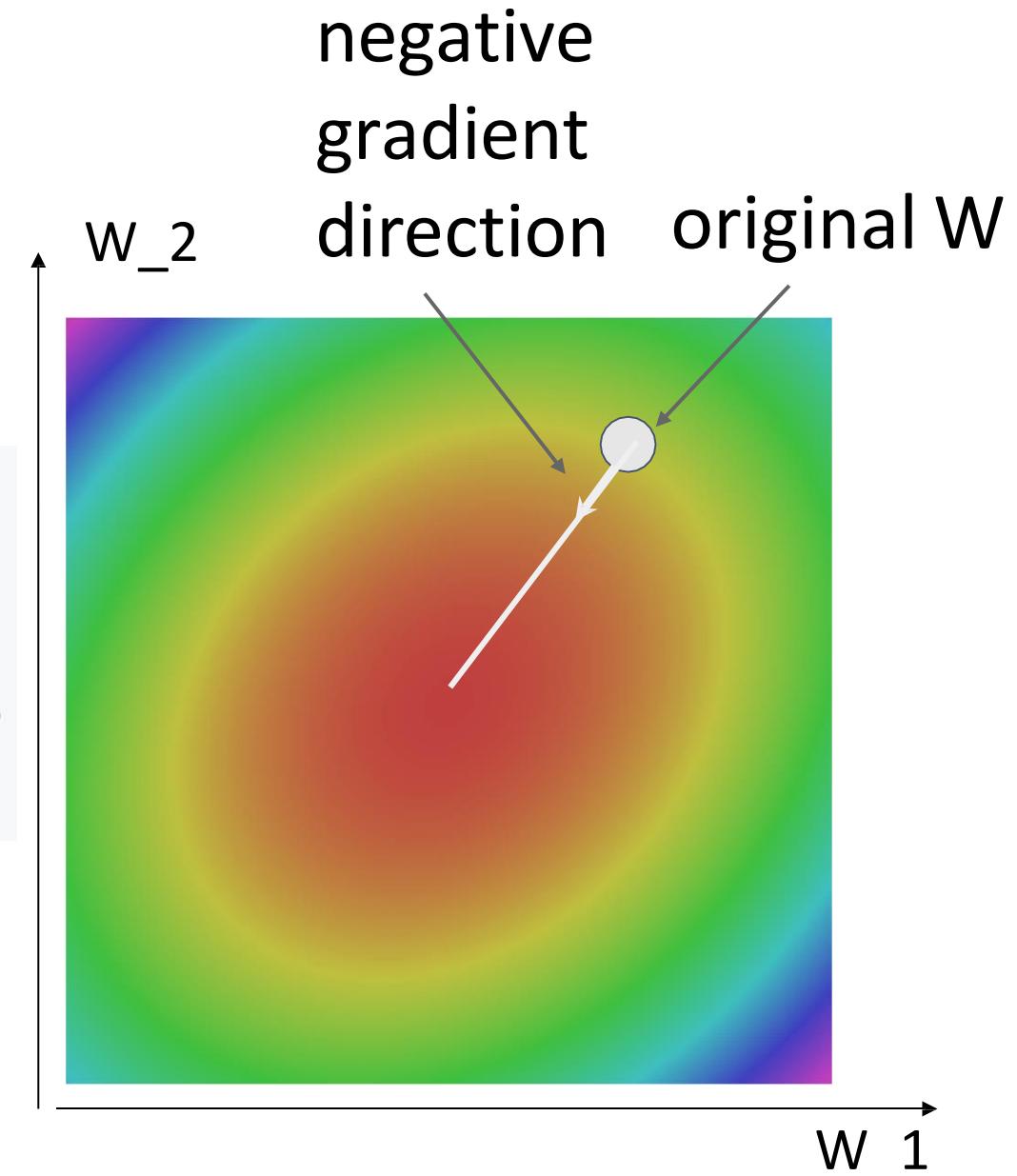
# Gradient Descent

Iteratively step in the direction of the negative gradient  
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
    dw = compute_gradient(loss_fn, data, w)
    w -= learning_rate * dw
```

## Hyperparameters:

- Weight initialization method
- Number of steps
- Learning rate



# Stochastic Gradient Descent (SGD)

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

Full sum expensive  
when N is large!

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

Approximate sum  
using a **minibatch** of  
examples  
32 / 64 / 128 common

```
# 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
```

# Stochastic Gradient Descent (SGD)

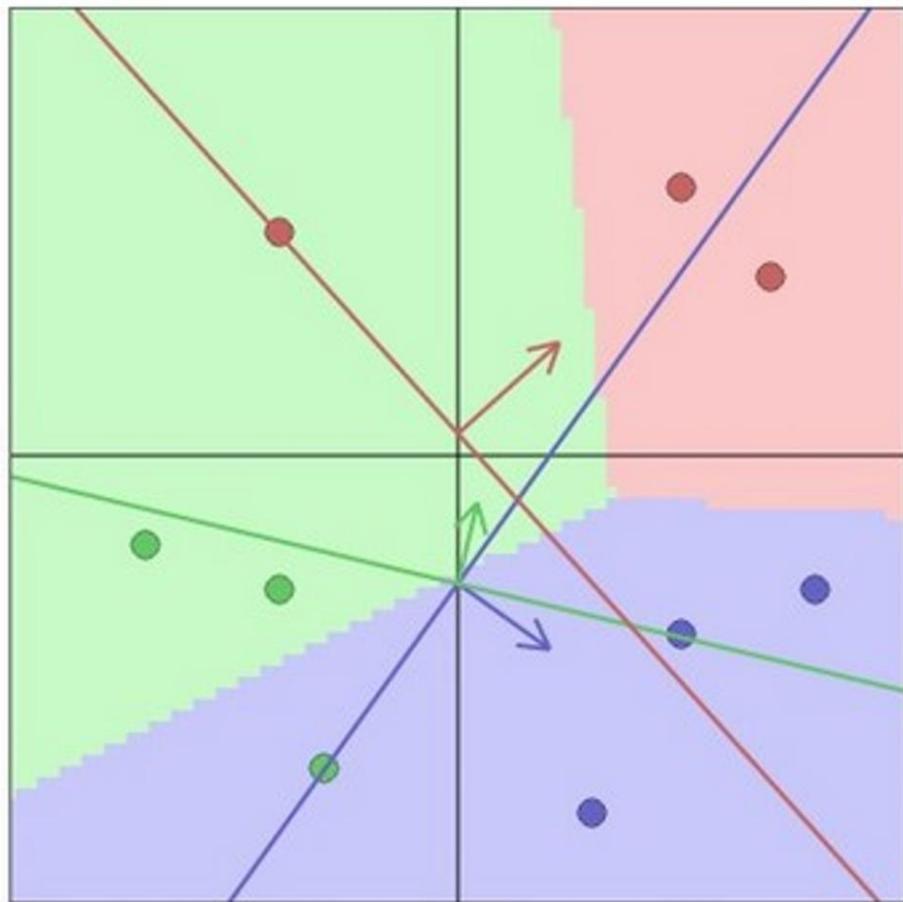
$$\begin{aligned} L(W) &= \mathbb{E}_{(x,y) \sim p_{data}}[L(x, y, W)] + \lambda R(W) \\ &\approx \frac{1}{N} \sum_{i=1}^N L(x_i, y_i, W) + \lambda R(W) \end{aligned}$$

Think of loss as an expectation over the full **data distribution**  $p_{data}$

Approximate expectation via sampling

$$\begin{aligned} \nabla_W L(W) &= \nabla_W \mathbb{E}_{(x,y) \sim p_{data}}[L(x, y, W)] + \lambda \nabla_W R(W) \\ &\approx \sum_{i=1}^N \nabla_w L_W(x_i, y_i, W) + \nabla_w R(W) \end{aligned}$$

# Interactive Web Demo



W[0,0]	W[0,1]	b[0]	x[0]	x[1]	y	s[0]	s[1]	s[2]	L
2.06 -0.12	1.48 0.12	-0.42 0.00	0.50	0.40	0	1.20	0.01	0.22	0.02
0.44 0.19	-1.82 -0.37	0.52 0.11	0.80	0.30	0	1.67	0.33	1.10	0.44
2.27 0.17	-2.04 0.14	-0.10 -0.11	0.30	0.80	0	1.38	-0.80	-1.05	0.00
			-0.40	0.30	1	-0.80	-0.20	-1.62	0.39
			-0.30	0.70	1	-0.01	-0.88	-2.21	1.87
			-0.70	0.20	1	-1.57	-0.15	-2.10	0.00
			0.70	-0.40	2	0.43	1.55	2.31	0.25
			0.50	-0.60	2	-0.28	1.83	2.26	0.57
			-0.40	-0.50	2	-1.98	1.26	0.01	2.24
mean:									0.64

Step size: 0.10000

Single parameter update

Start repeated update

Stop repeated update

Randomize parameters

Total data loss: 0.64

Regularization loss: 1.92

Total loss: 2.57

L2 Regularization strength: 0.10000

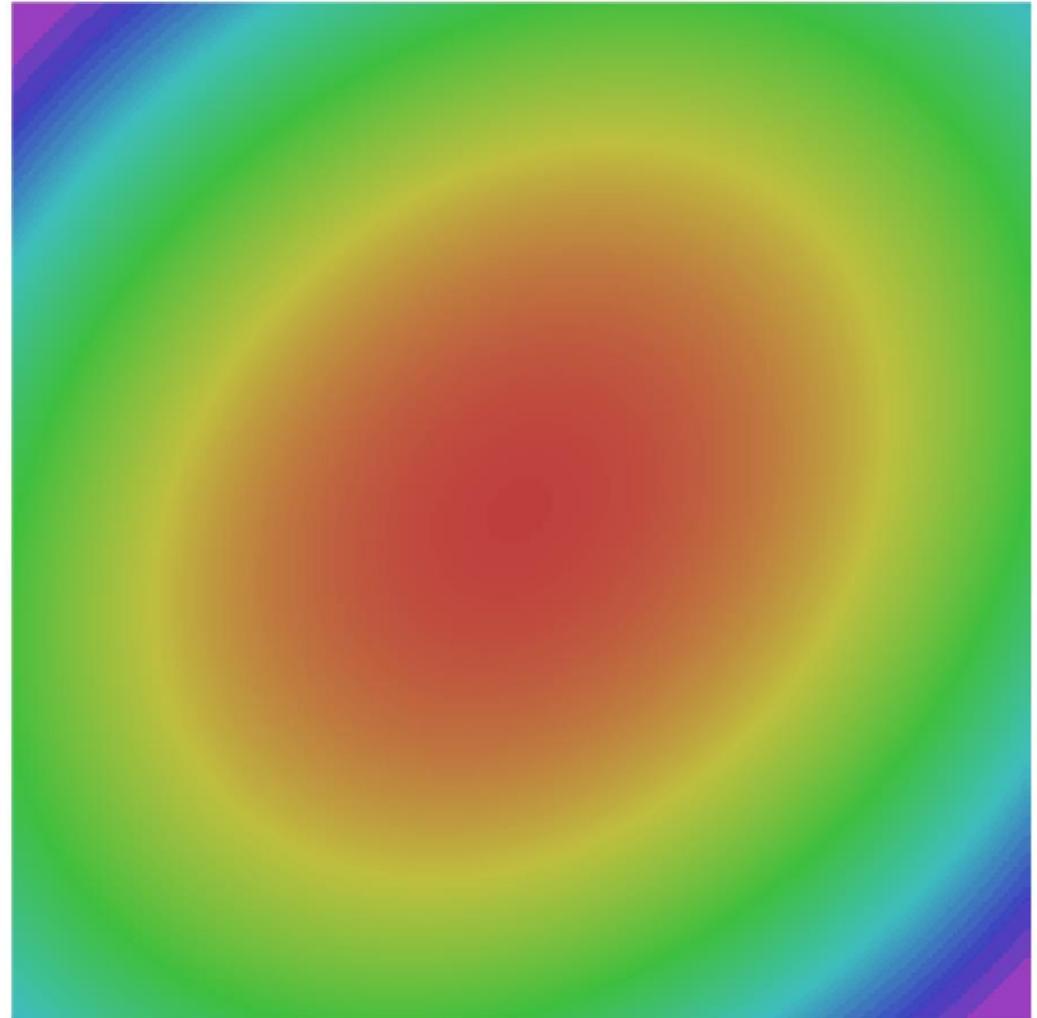
<http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/>

# Problems with SGD (I)

Our gradients come from minibatches  
so they can be noisy!

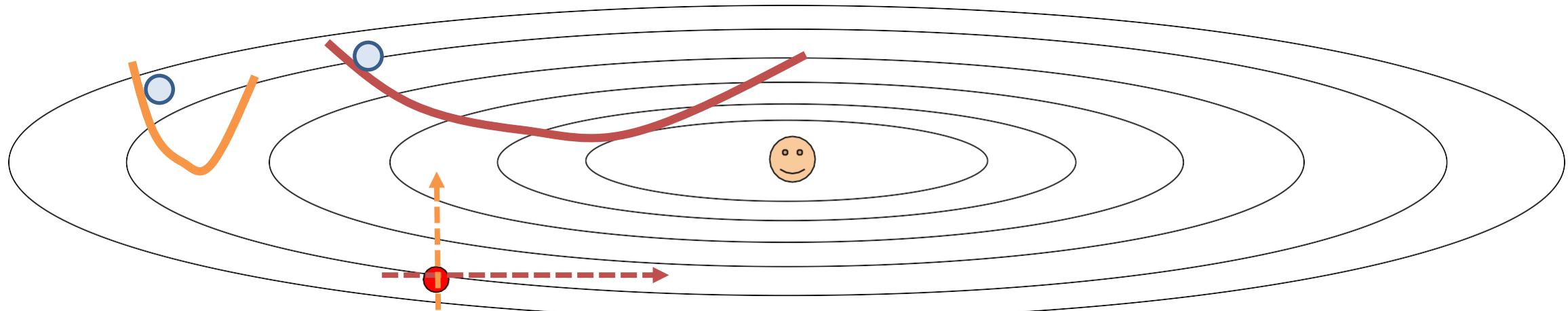
$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W) + \lambda R(W)$$

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



# Problems with SGD (II)

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



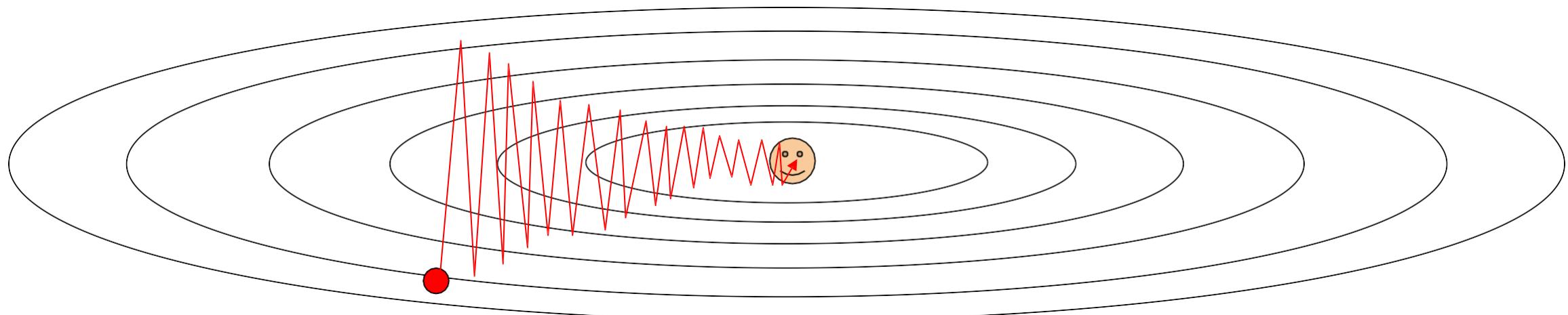
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

# Problems with SGD (II)

What if loss changes quickly in one direction and slowly in another?

What does gradient descent do?

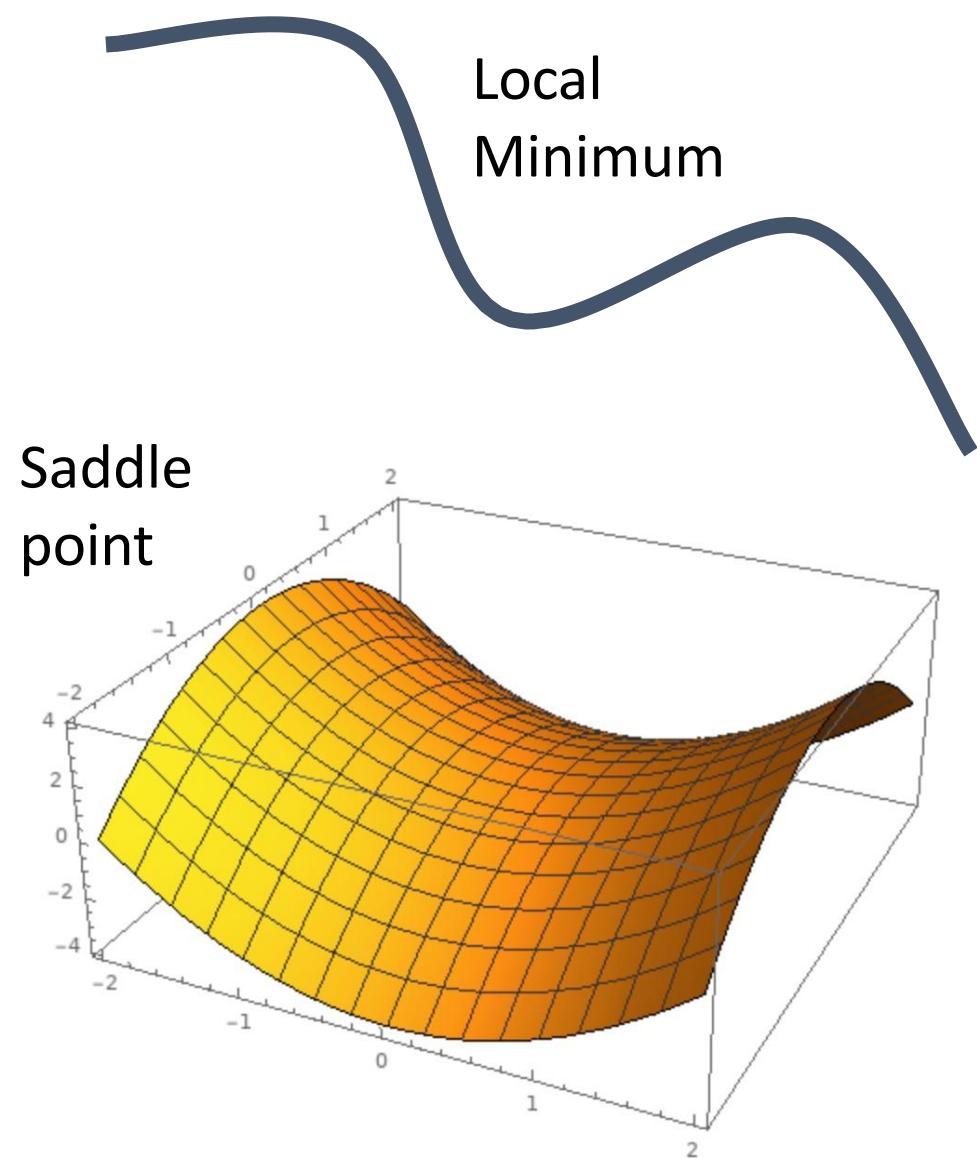
Very slow progress along shallow dimension, jitter along steep direction



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

# Problems with SGD (III)

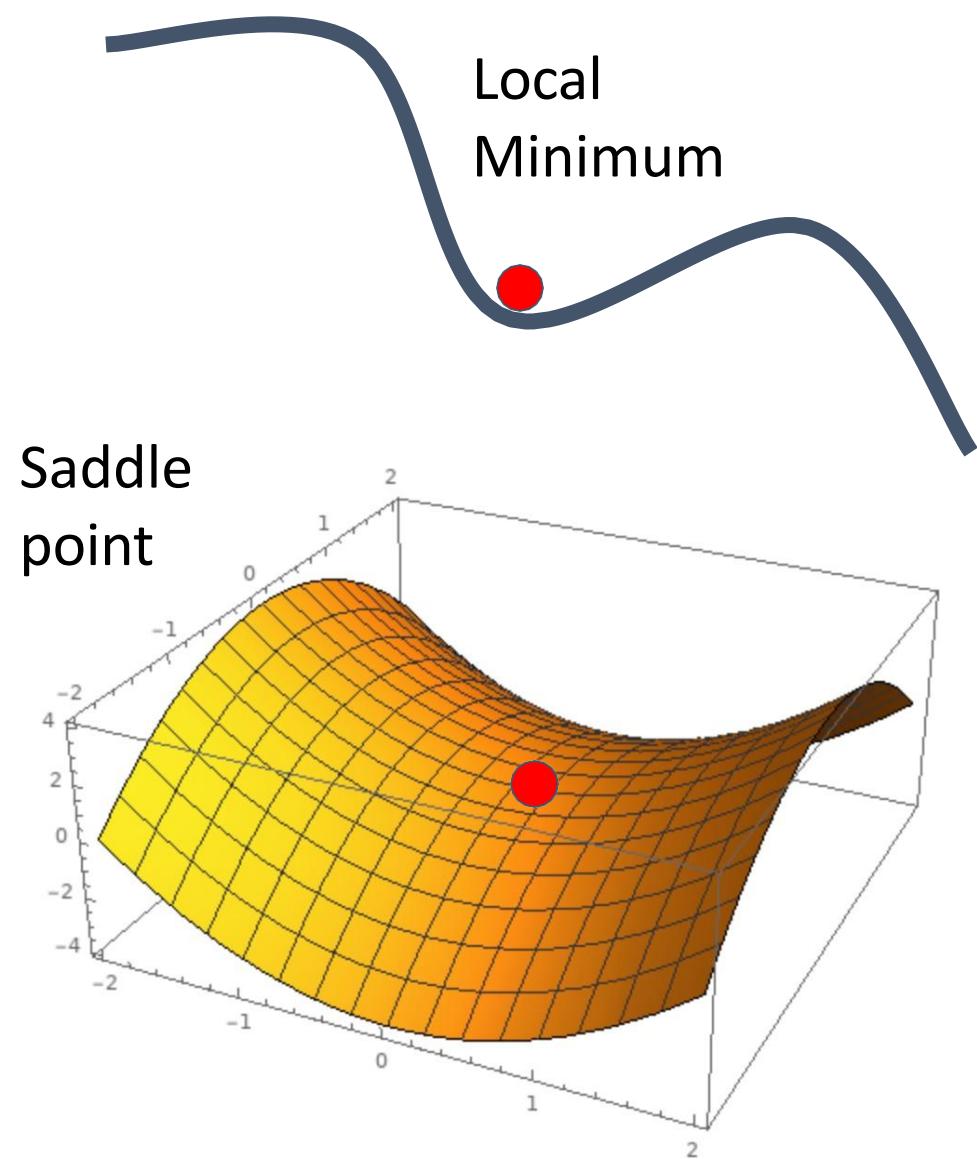
What if the loss function has a **local minimum** or **saddle point**?



# Problems with SGD (III)

What if the loss function has a **local minimum** or **saddle point**?

Zero gradient, gradient descent gets stuck



# SGD

SGD

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

```
for t in range(num_steps):
    dw = compute_gradient(w)
    w -= learning_rate * dw
```

# SGD + Momentum

## SGD

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

```
for t in range(num_steps):
    dw = compute_gradient(w)
    w -= learning_rate * dw
```

## SGD+Momentum

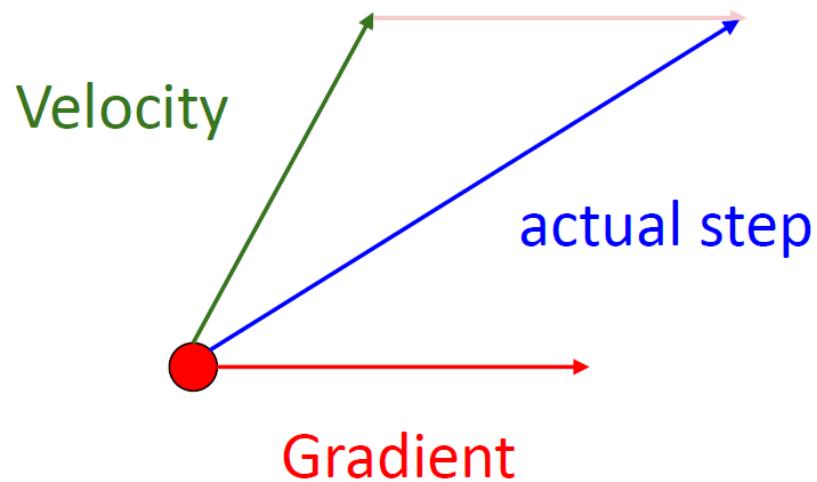
$$\begin{aligned} v_{t+1} &= \rho v_t + \nabla f(x_t) \\ x_{t+1} &= x_t - \alpha v_{t+1} \end{aligned}$$

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

# SGD + Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

SGD+Momentum

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

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

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

# SGD + Momentum

## SGD+Momentum

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

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

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v - learning_rate * dw
    w += v
```

## SGD+Momentum

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

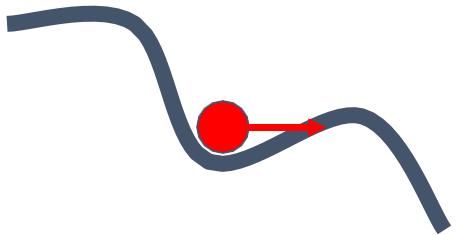
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```

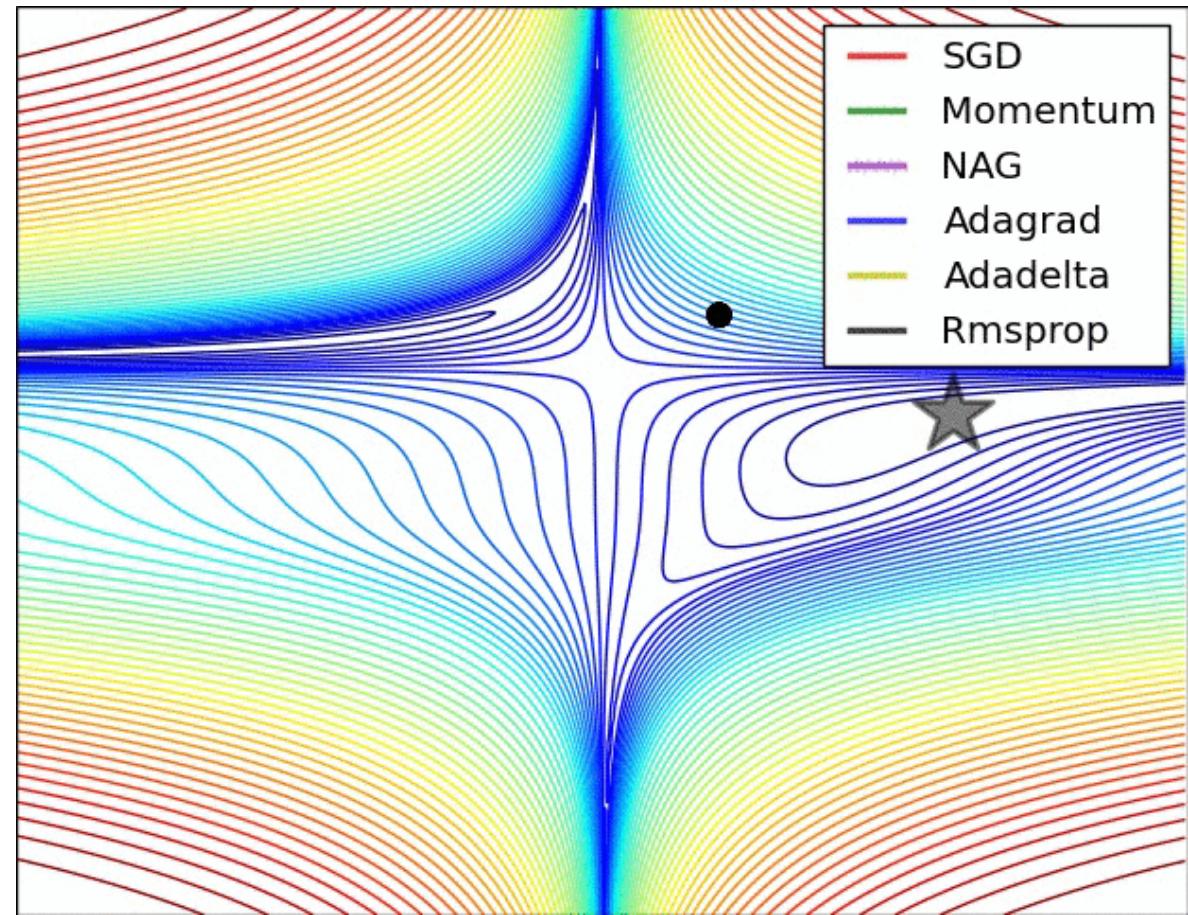
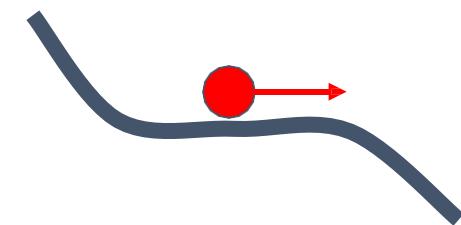
You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of  $x$

# SGD + Momentum

Local Minima

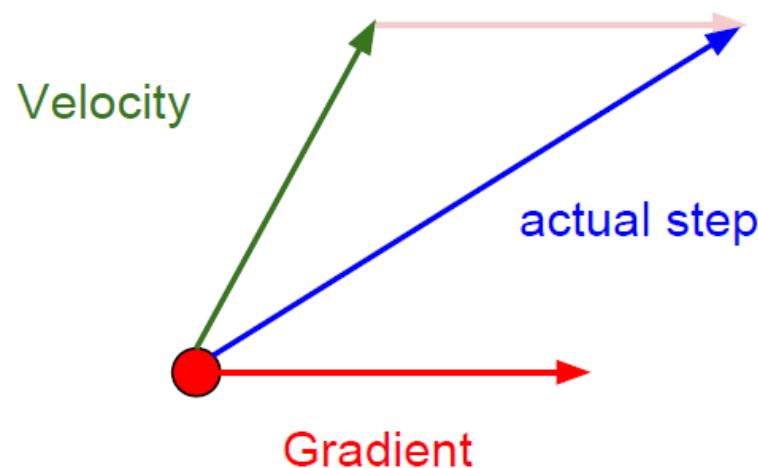


Saddle points



# SGD+Momentum

Momentum update:

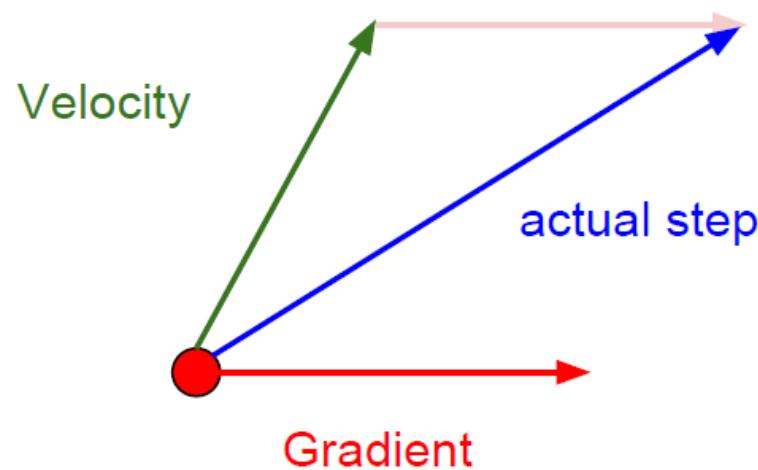


Combine gradient at current point with velocity to get step used to update weights

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 deep learning", ICML 2013

# Nesterov Momentum

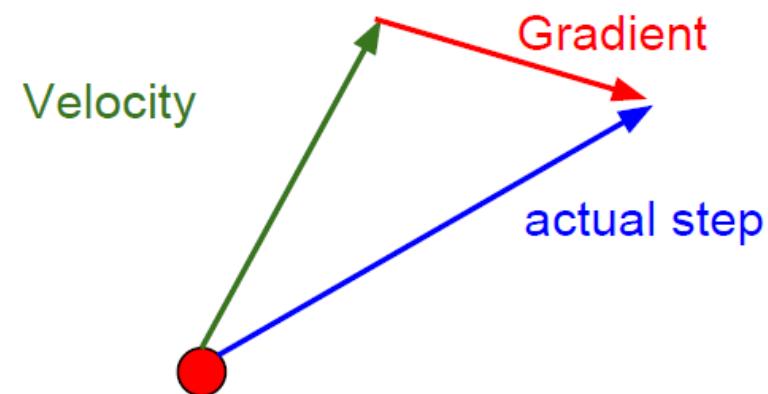
Momentum update:



Combine gradient at current point with velocity to get step used to update weights

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 deep learning", ICML 2013

Nesterov Momentum

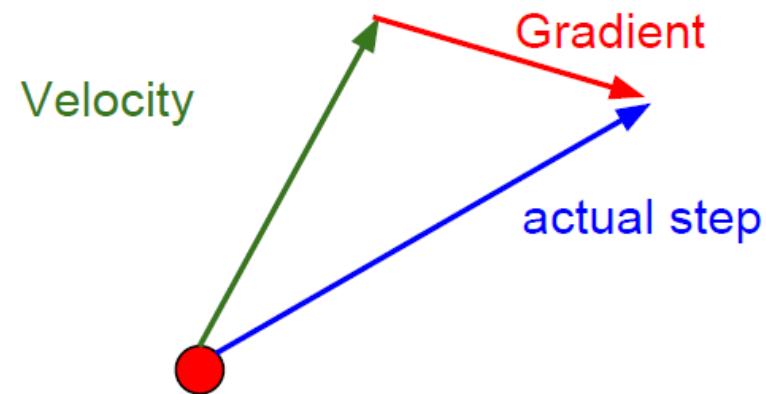


"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

# Nesterov Momentum

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

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



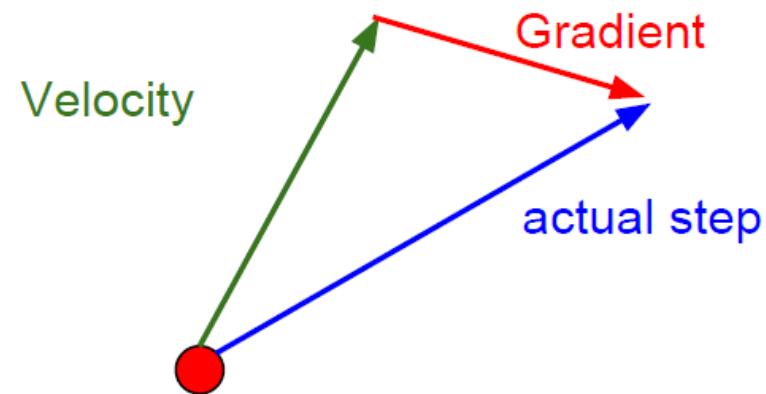
“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

# Nesterov Momentum

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

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

Annoying, usually we want update in terms of  $x_t, \nabla f(x_t)$



“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

# Nesterov Momentum

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

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

Annoying, usually we want update in terms of  $x_t, \nabla f(x_t)$

Change of variables  $\tilde{x}_t = x_t + \rho v_t$  and rearrange:

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

$$\begin{aligned}\tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)\end{aligned}$$

$$\begin{aligned}\tilde{x}_{t+1} &= \underline{x_{t+1}} + \rho v_{t+1} \\ &= \underline{x_t} + \underline{v_{t+1}} + \rho v_{t+1} \\ &= \tilde{x}_t - \rho v_t + v_{t+1} + \rho v_{t+1}\end{aligned}$$

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

# Nesterov Momentum

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

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

Annoying, usually we want update in terms of  $x_t, \nabla f(x_t)$

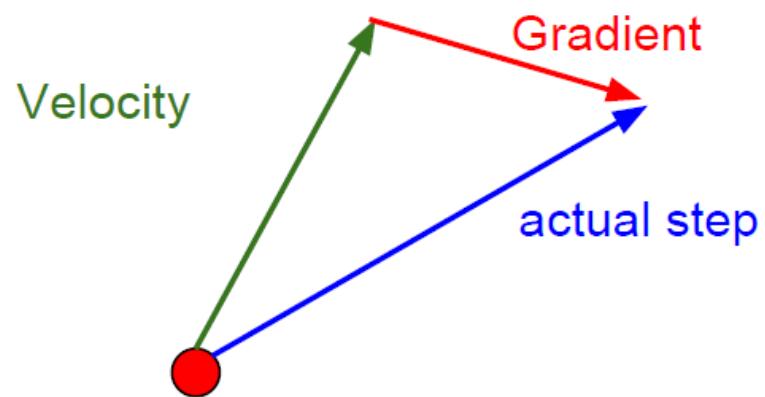
Change of variables  $\tilde{x}_t = x_t + \rho v_t$  and rearrange:

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

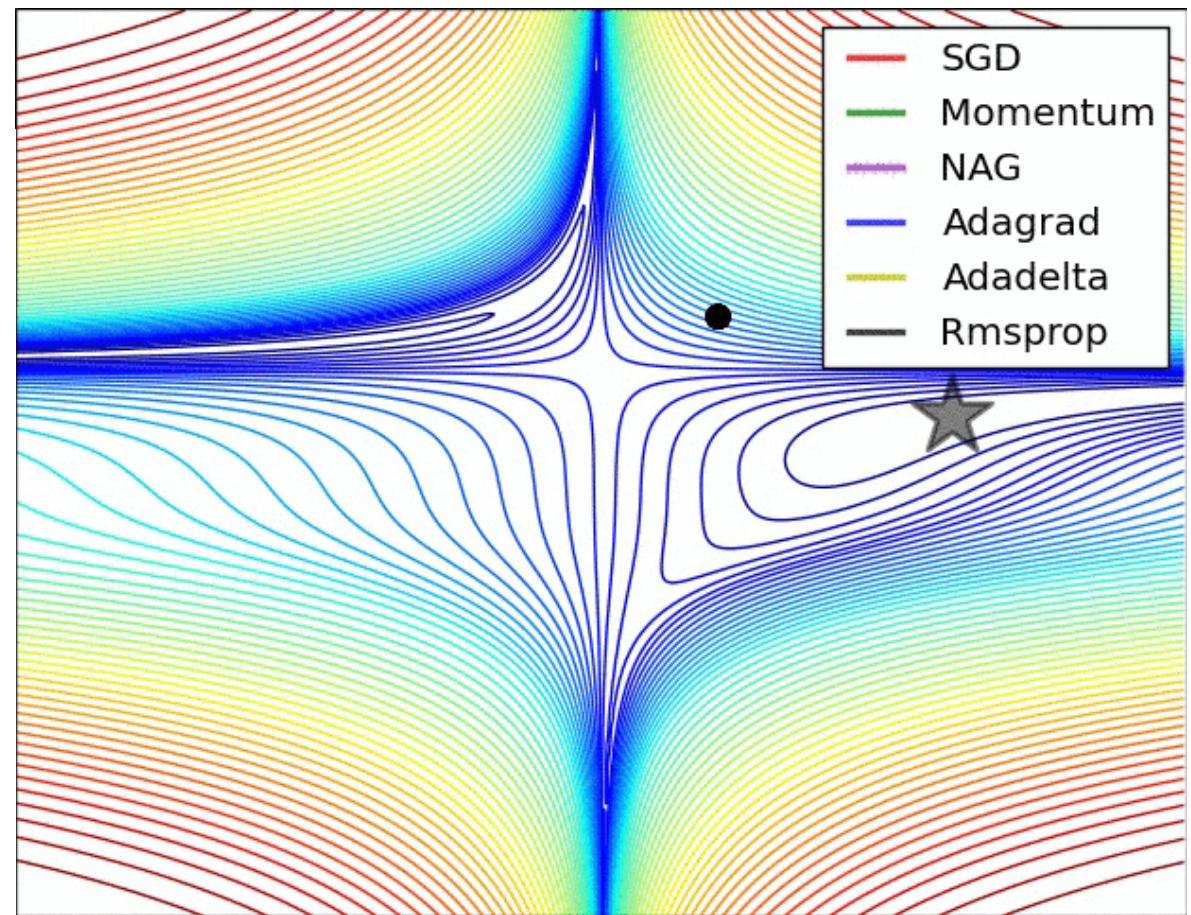
$$\begin{aligned}\tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)\end{aligned}$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

# Nesterov Momentum



“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

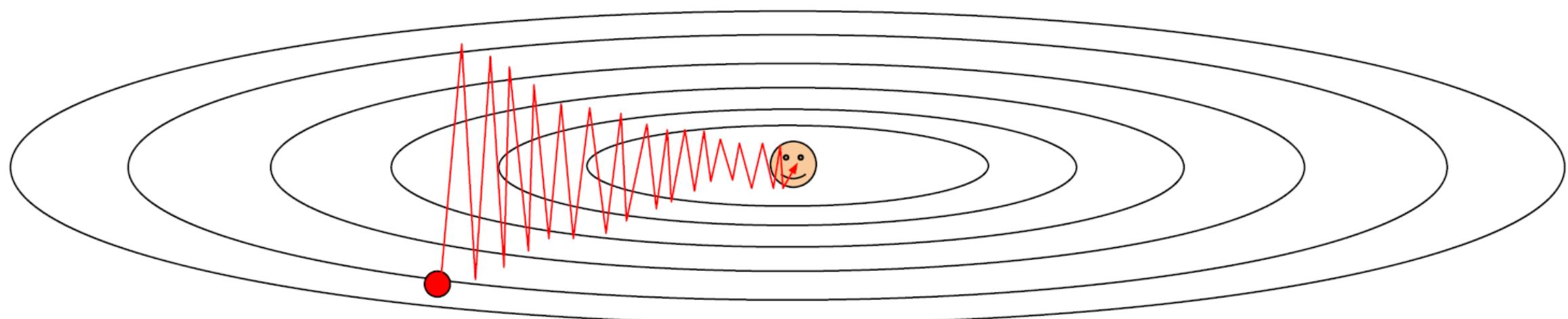


# Optimization: Problems with SGD

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

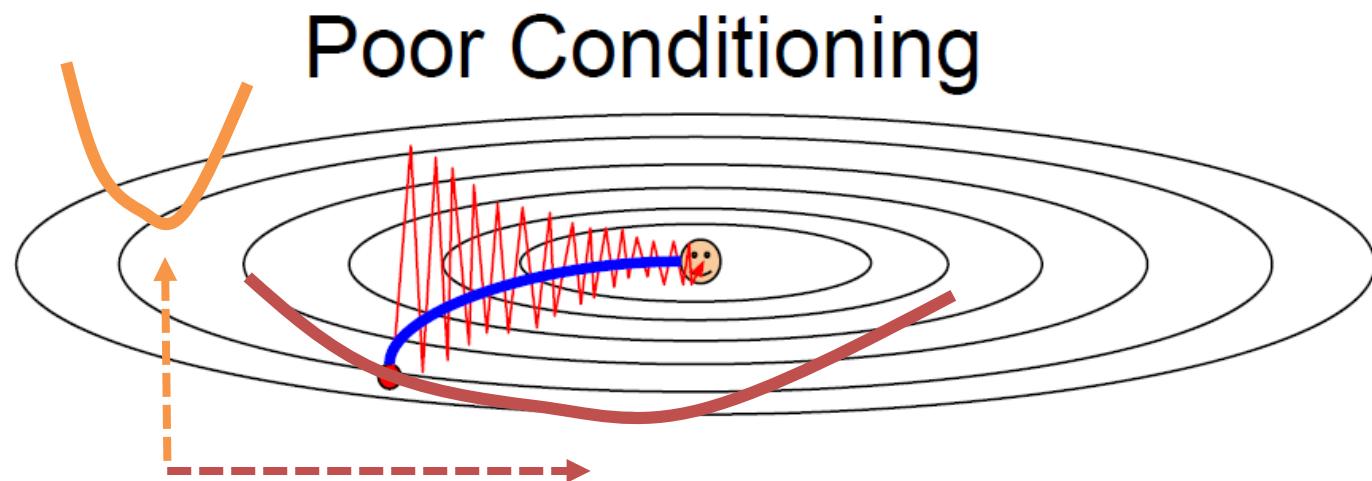


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

# AdaGrad

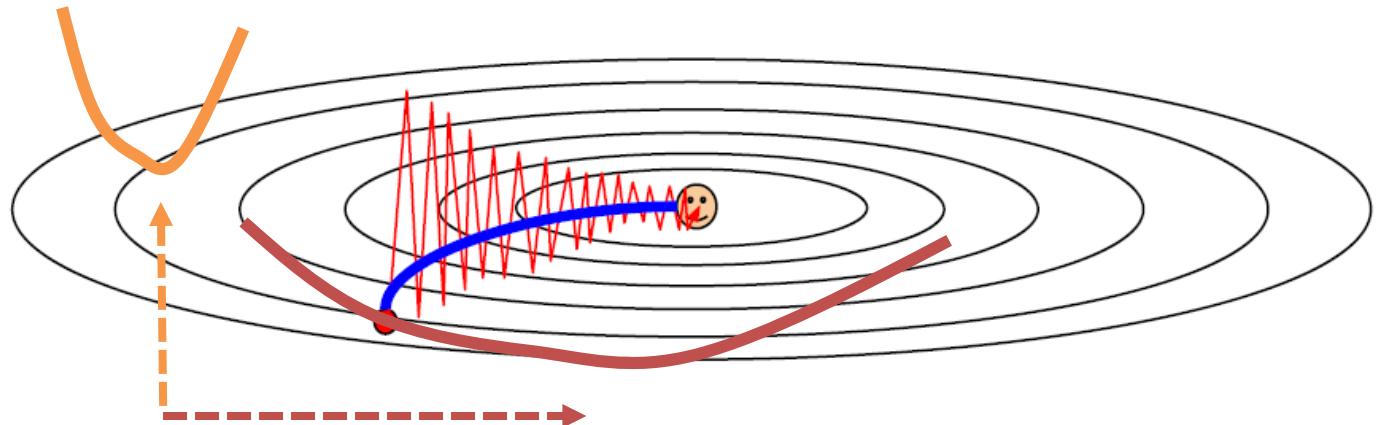
$$w^{(t+1)} = w^{(t)} - \alpha \nabla_w L(x; w)$$

$$ax^2 + bx + c$$



# AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

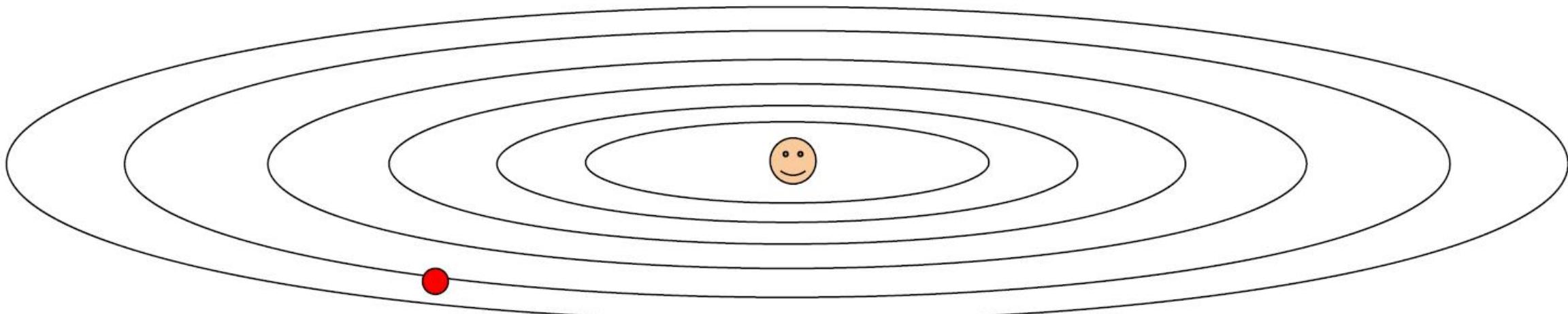


Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

“Per-parameter learning rates”  
or “adaptive learning rates”

# AdaGrad

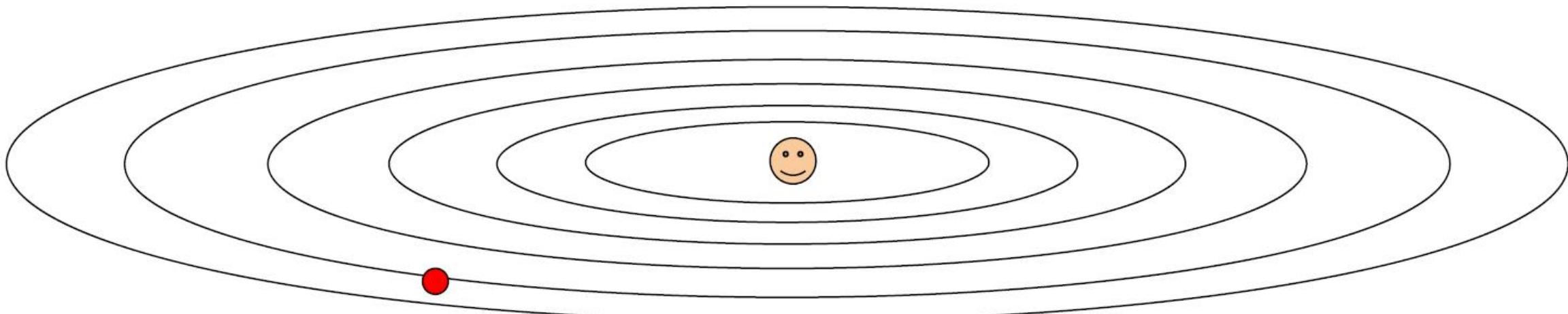
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad?

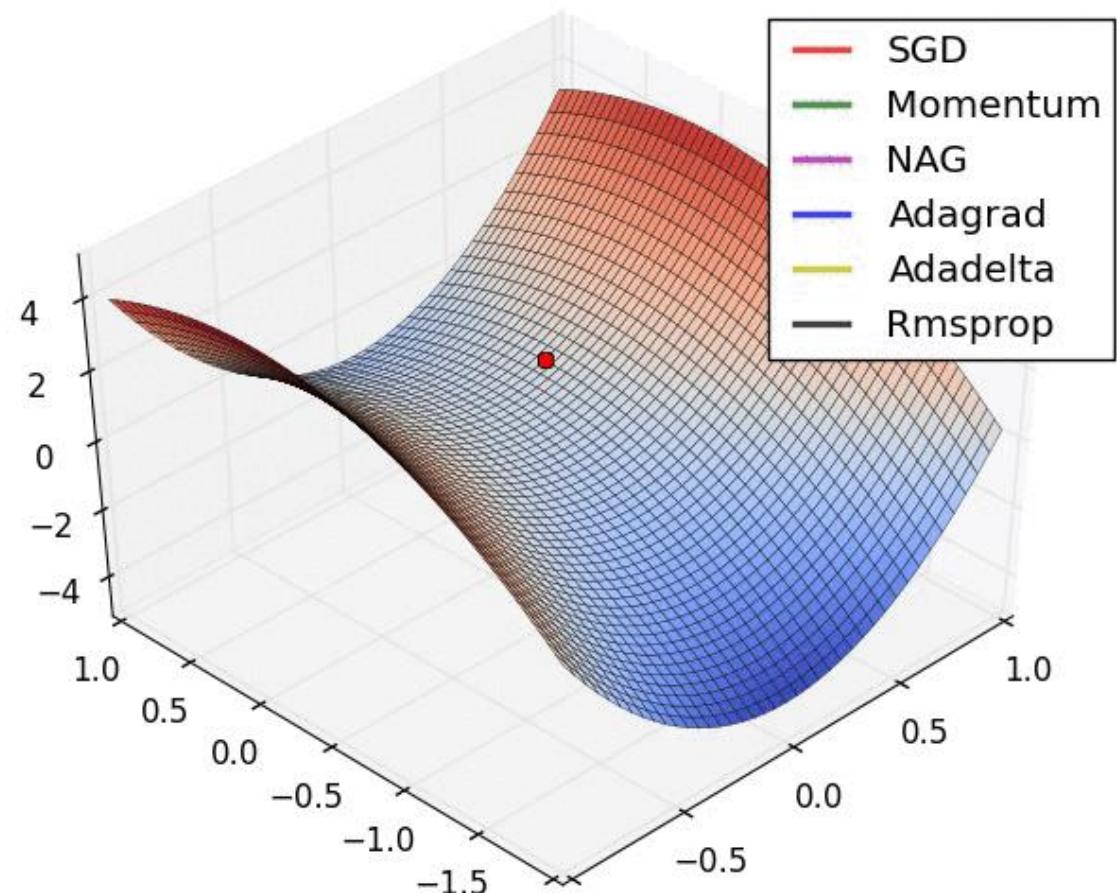
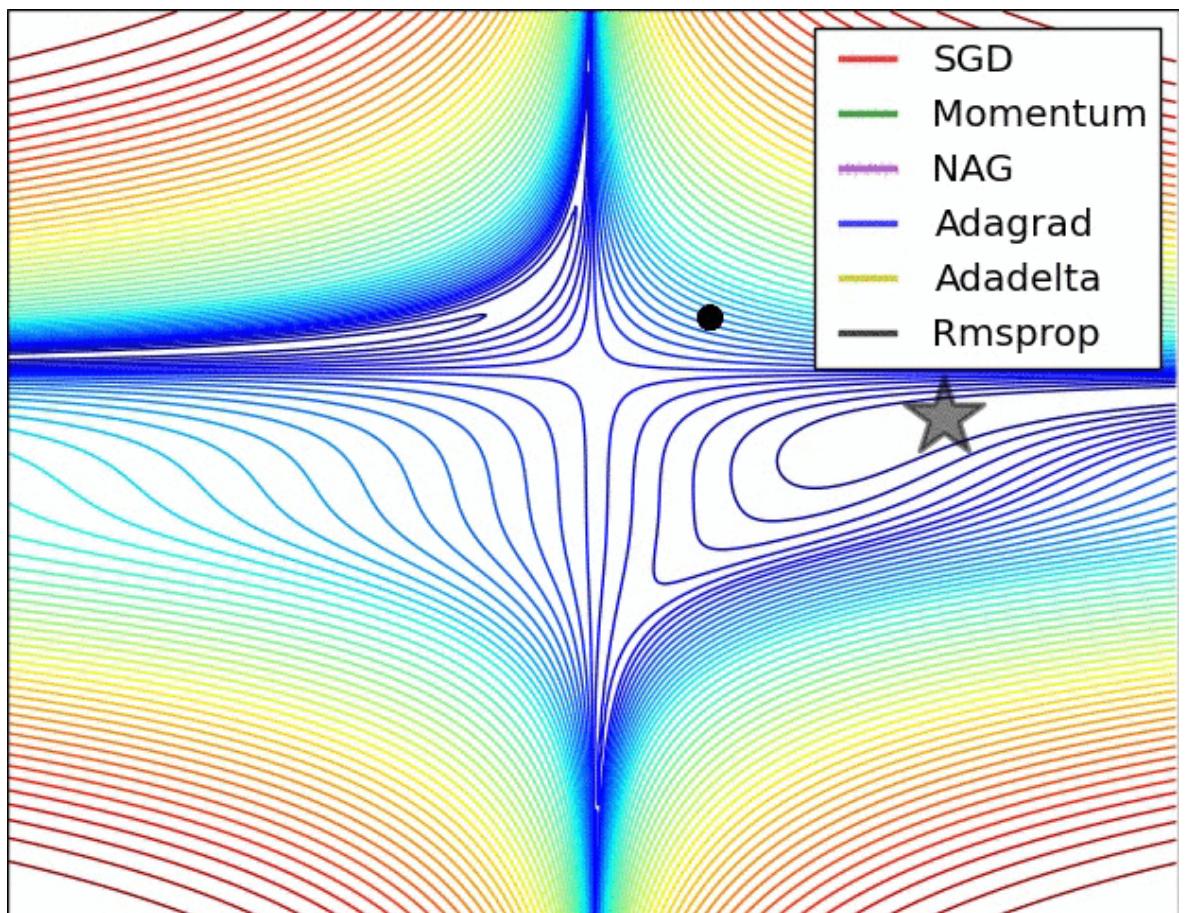
# AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



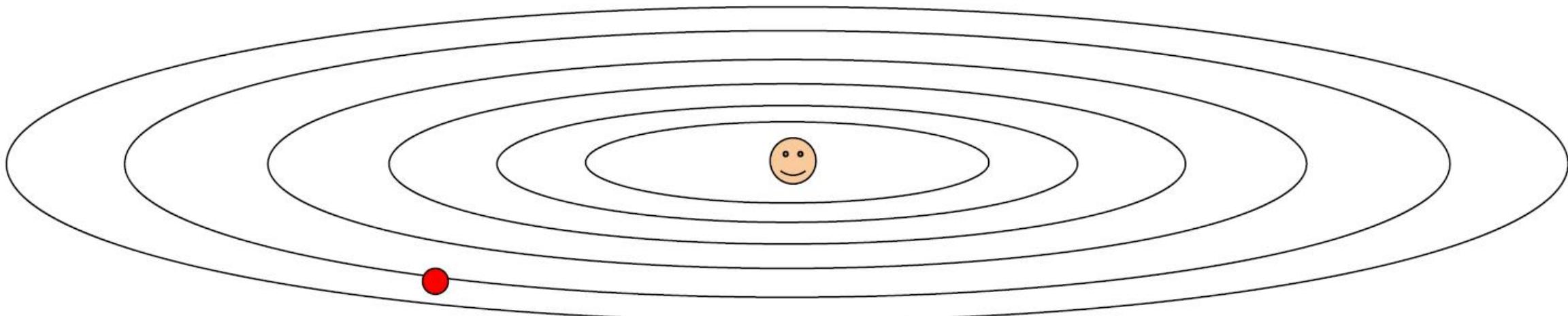
Q: What happens with AdaGrad?

Progress along “steep” directions is damped;  
progress along “flat” directions is accelerated



# AdaGrad

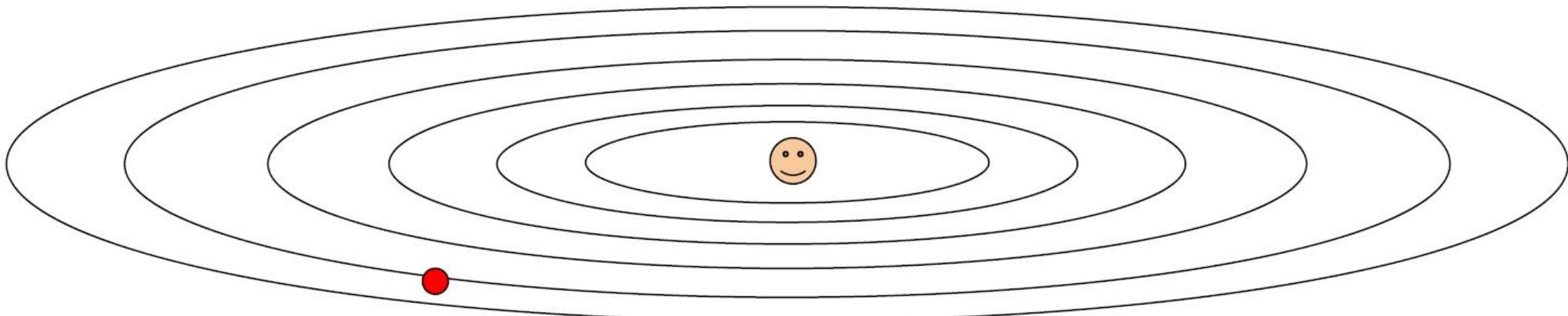
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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

# AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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

Decays to zero

# Exponentially weighted averages

$$\theta_1 = 40^{\circ}\text{F}$$

$$\theta_2 = 49^{\circ}\text{F}$$

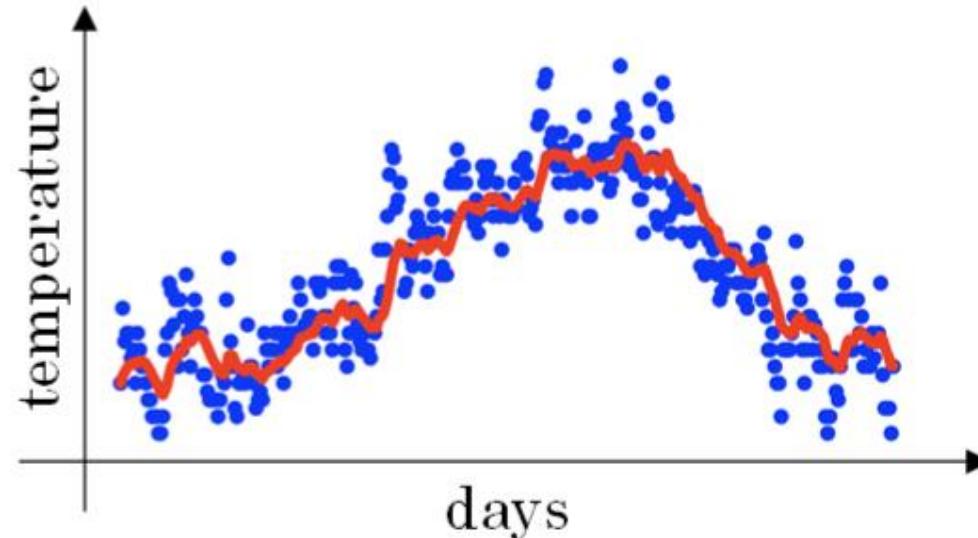
$$\theta_3 = 45^{\circ}\text{F}$$

⋮

$$\theta_{180} = 60^{\circ}\text{F}$$

$$\theta_{181} = 56^{\circ}\text{F}$$

⋮



$$V_0 = 0$$

$$V_1 = 0.9 * V_0 + 0.1 * \theta_1$$

$$V_2 = 0.9 * V_1 + 0.1 * \theta_2$$

$$V_3 = 0.9 * V_2 + 0.1 * \theta_3$$

⋮

⋮

⋮

$$V_t = \boxed{0.9} * V_{t-1} + \boxed{0.1} * \theta_t$$

$$\beta$$

$$(1 - \beta)$$

# Exponentially weighted averages

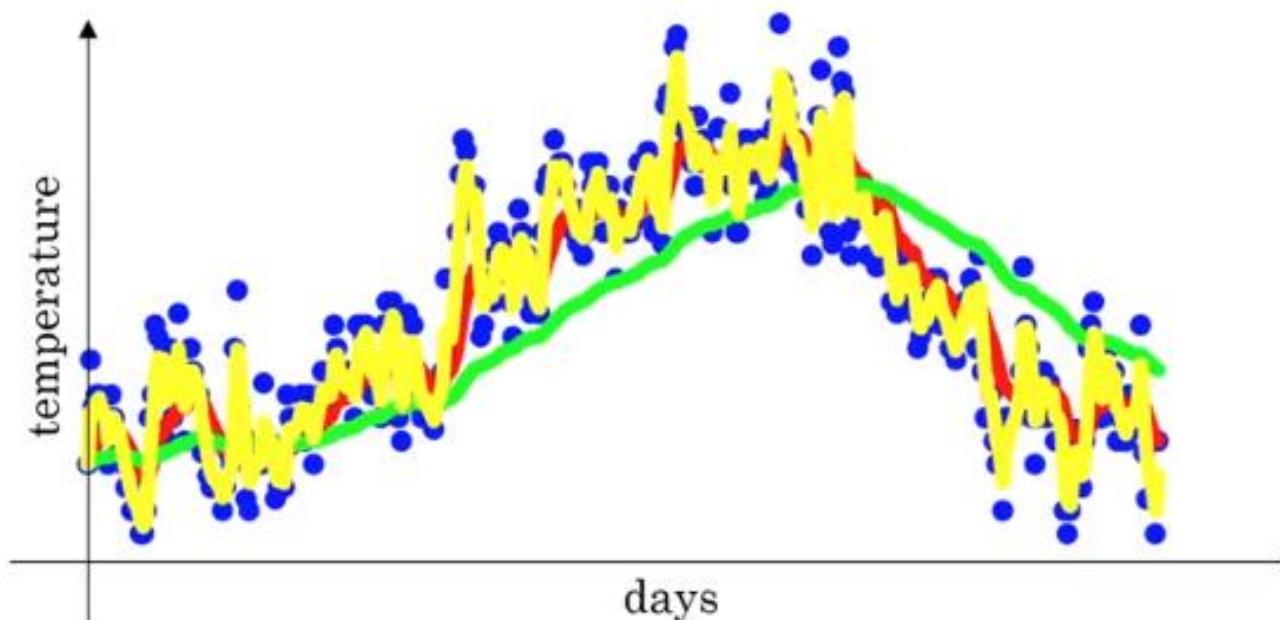
$$V_t = \beta * V_{t-1} + (1 - \beta) * \theta_t$$

$\beta = 0.9$  :  $\approx 10$  days

$V_t$  as approximately average  
over  $\approx \frac{1}{(1-\beta)}$  days' temperature

$\beta = 0.95$ :  $\approx 50$  days

$\beta = 0.5$  :  $\approx 2$  days



# RMSProp

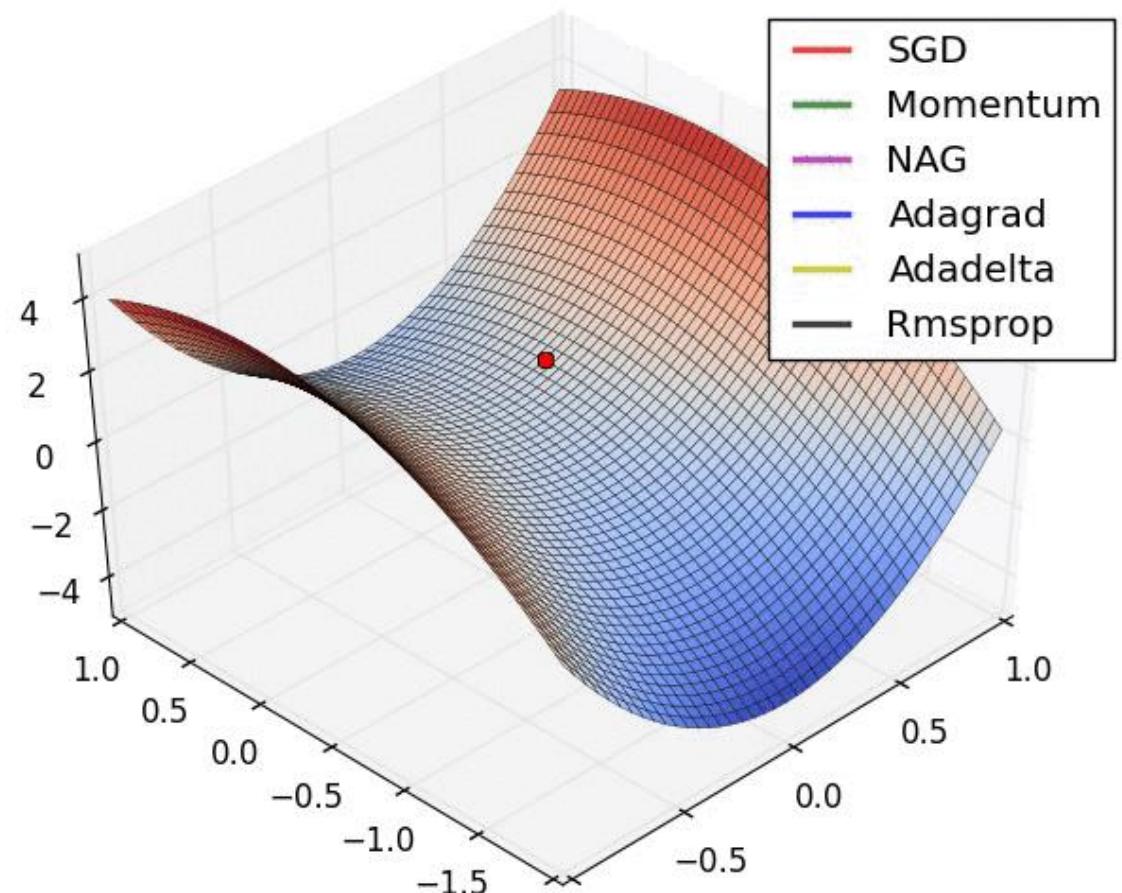
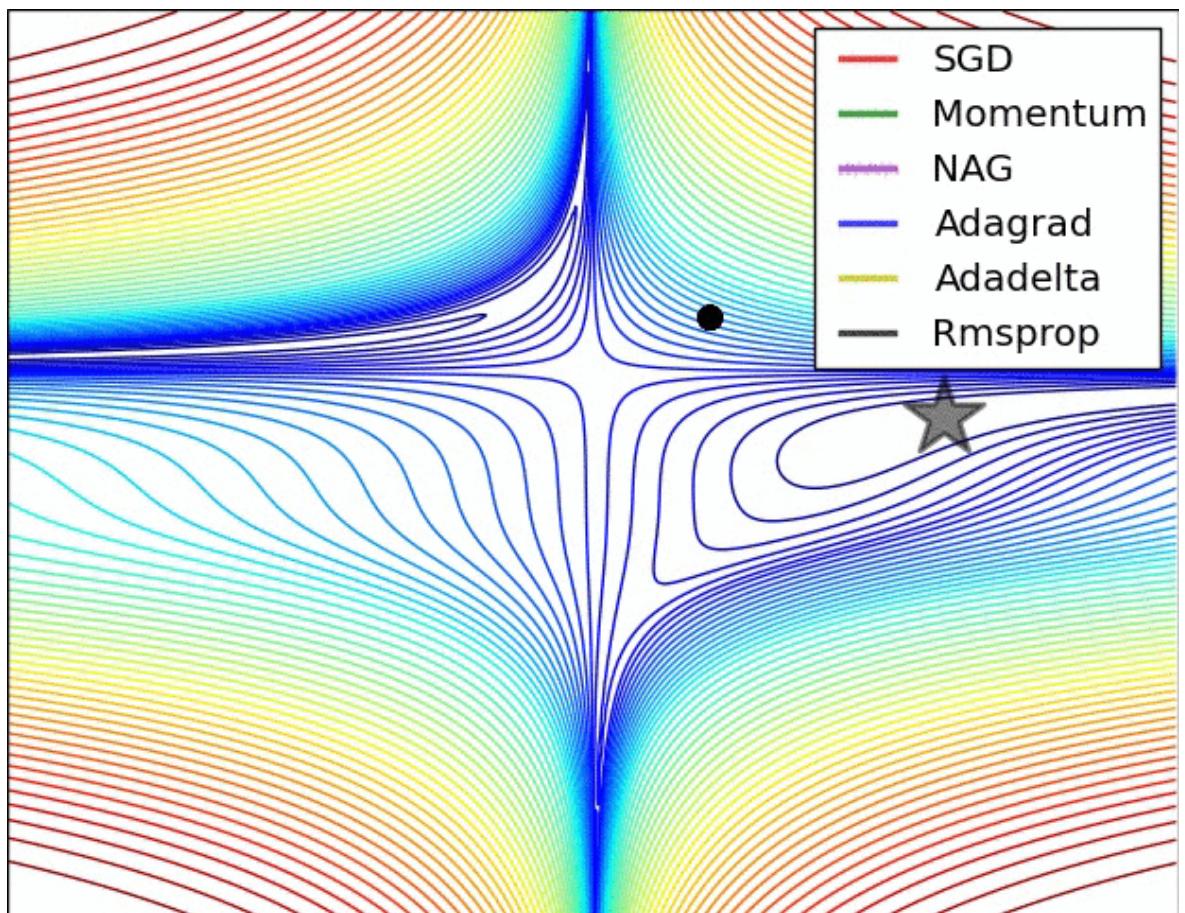
## AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



## RMSProp

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



# Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

# Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

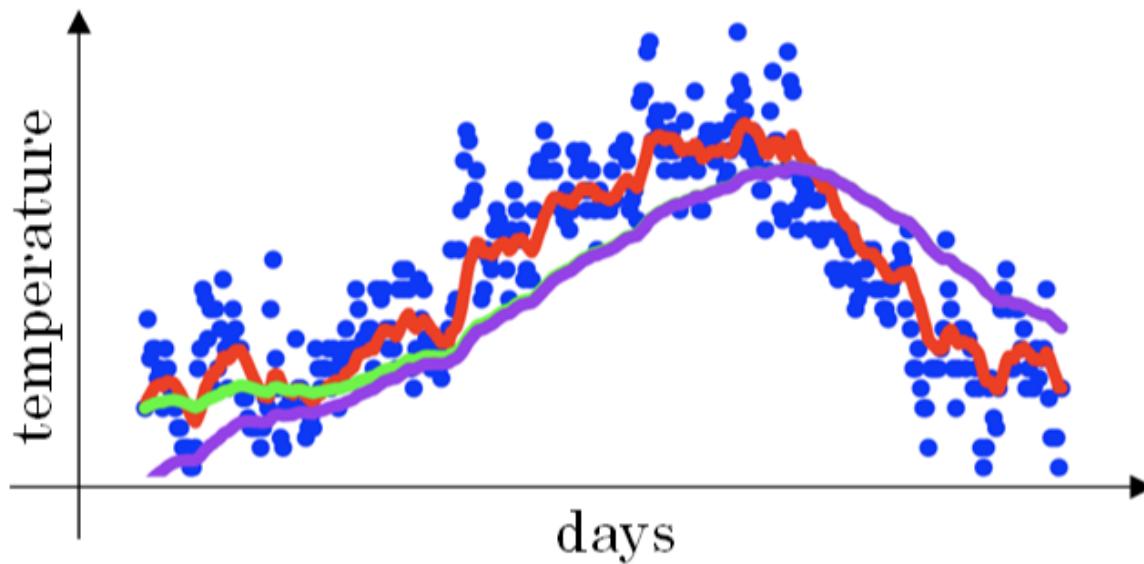
Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

# Bias Correction



$$\frac{V_t}{1 - \beta^t}$$

$$V_0 = 0$$

$$V_1 = \cancel{0.98 * V_0} + 0.02 * \theta_1$$

$$V_2 = 0.98 * V_1 + 0.02 * \theta_2$$

$$= 0.0196 * \theta_1 + 0.02 * \theta_2$$

$$t = 2: (1 - \beta^t) = 1 - 0.98^2 = 0.0396$$

$$\frac{V_2}{1 - \beta^2} = \frac{0.0196 * \theta_1 + 0.02 * \theta_2}{0.0396}$$

# Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that  
first and second moment  
estimates start at zero

# Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

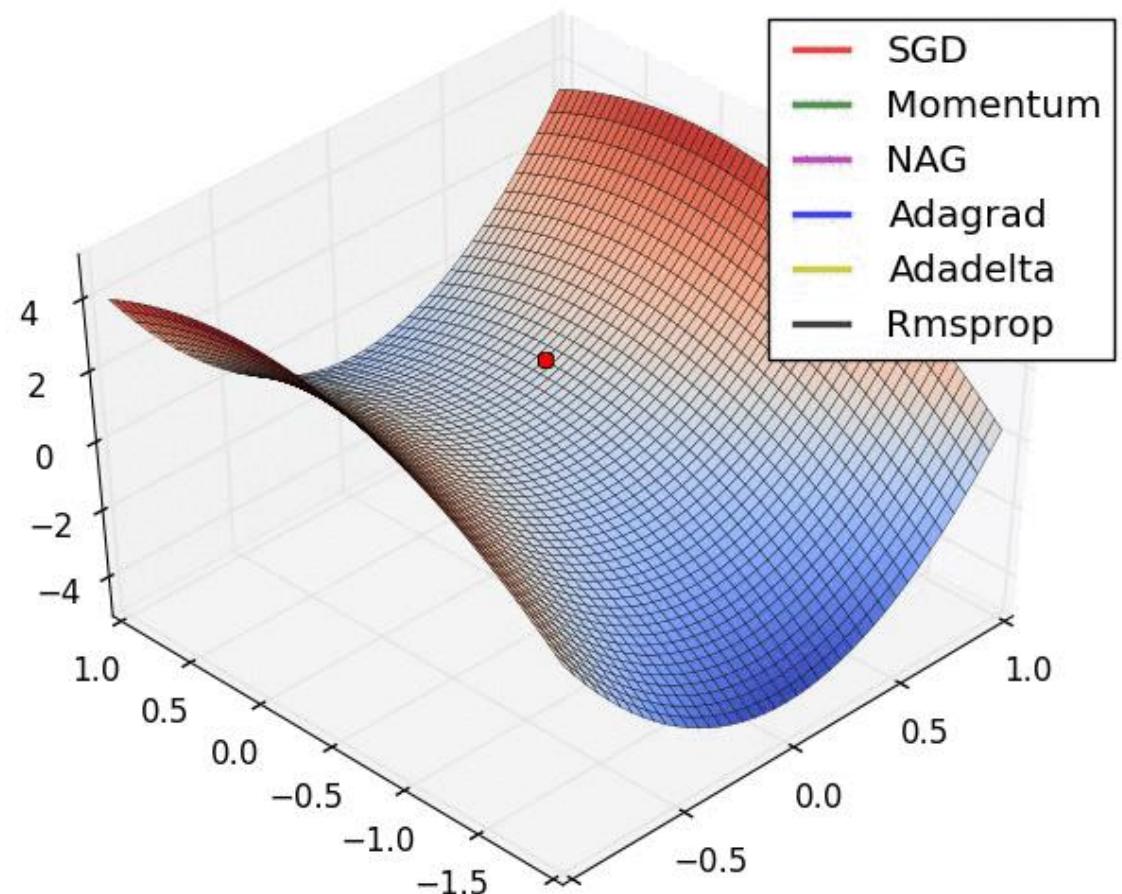
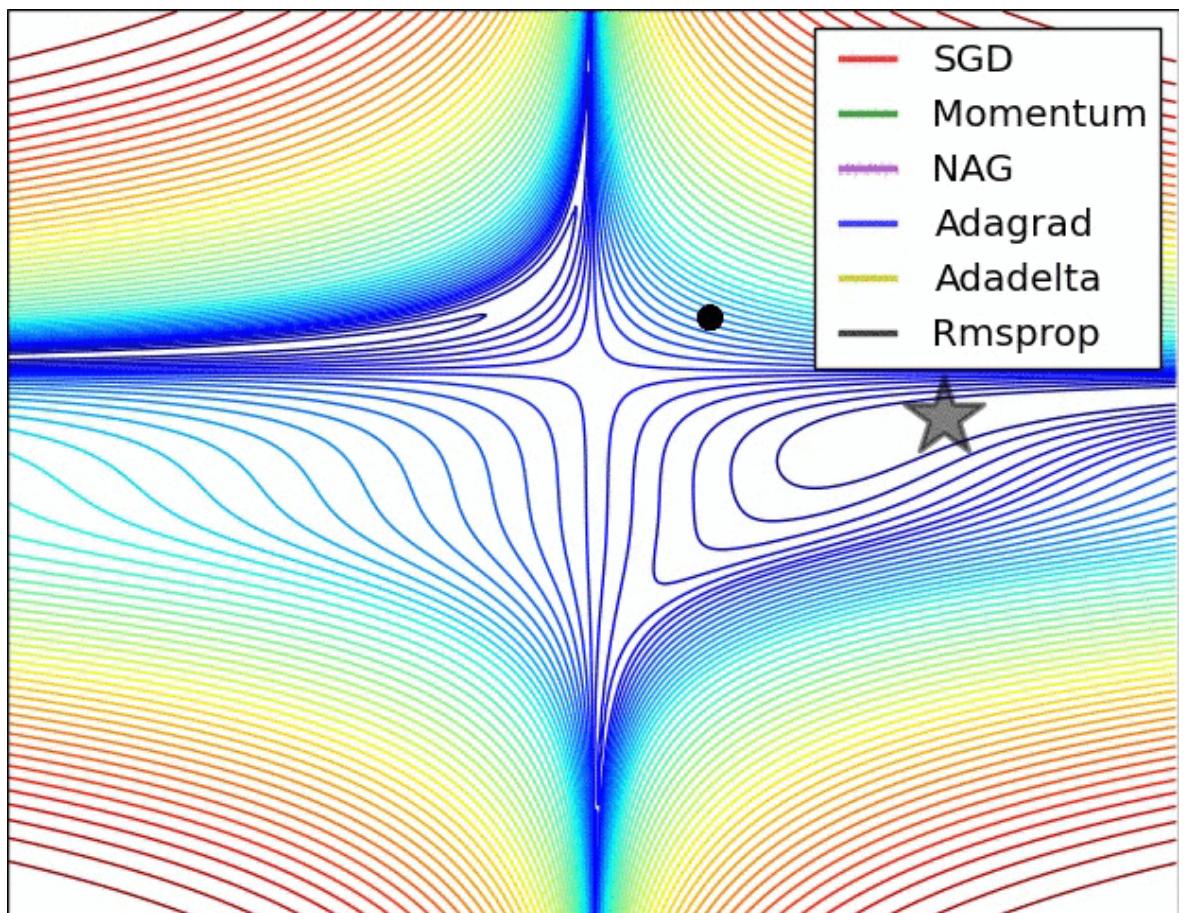
Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that  
first and second moment  
estimates start at zero

Adam with  $\beta_1 = 0.9$ ,  
 $\beta_2 = 0.999$ , and  $\text{learning\_rate} = 1e-3$  or  $5e-4$   
is a great starting point for many models!



# Adam: Very Common in Practice!

for input to the CNN; each colored pixel in the image yields a 7D one-hot vector. Following common practice, the network is trained end-to-end using stochastic gradient descent with the Adam optimizer [22]. We anneal the learning rate to 0 using a half cosine schedule without restarts [28].

Bakhtin, van der Maaten, Johnson, Gustafson, and Girshick, NeurIPS 2019

We train all models using Adam [23] with learning rate  $10^{-4}$  and batch size 32 for 1 million iterations; training takes about 3 days on a single Tesla P100. For each mini-batch we first update  $f$ , then update  $D_{img}$  and  $D_{obj}$ .

Johnson, Gupta, and Fei-Fei, CVPR 2018

ganized into three residual blocks. We train for 25 epochs using Adam [27] with learning rate  $10^{-4}$  and 32 images per batch on 8 Tesla V100 GPUs. We set the cubify thresh-

Gkioxari, Malik, and Johnson, ICCV 2019

sampled with each bit drawn uniformly at random. For gradient descent, we use Adam [29] with a learning rate of  $10^{-3}$  and default hyperparameters. All models are trained with batch size 12. Models are trained for 200 epochs, or 400 epochs if being trained on multiple noise layers.

Zhu, Kaplan, Johnson, and Fei-Fei, ECCV 2018

16 dimensional vectors. We iteratively train the Generator and Discriminator with a batch size of 64 for 200 epochs using Adam [22] with an initial learning rate of 0.001.

Gupta, Johnson, et al, CVPR 2018

Adam with `beta1 = 0.9`,  
`beta2 = 0.999`, and `learning_rate = 1e-3, 5e-4, 1e-4` is a great starting point for many models!

# Optimization Algorithm Comparison

Algorithm	Tracks first moments (Momentum)	Tracks second moments (Adaptive learning rates)	Leaky second moments	Bias correction for moment estimates
SGD	✗	✗	✗	✗
SGD+Momentum	✓	✗	✗	✗
Nesterov	✓	✗	✗	✗
AdaGrad	✗	✓	✗	✗
RMSProp	✗	✓	✓	✗
Adam	✓	✓	✓	✓

# L2 Regularization vs Weight Decay

## Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

# L2 Regularization vs Weight Decay

## Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

## L2 Regularization

$$L(w) = L_{data}(w) + \lambda |w|^2$$

$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

# L2 Regularization vs Weight Decay

## Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

## L2 Regularization

$$L(w) = L_{data}(w) + \lambda |w|^2$$

$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

## Weight Decay

$$L(w) = L_{data}(w)$$

$$g_t = \nabla L_{data}(w_t)$$

$$s_t = \text{optimizer}(g_t) + 2\lambda w_t$$

$$w_{t+1} = w_t - \alpha s_t$$

# L2 Regularization vs Weight Decay

## Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

## L2 Regularization

$$L(w) = L_{data}(w) + \lambda |w|^2$$

$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

## Weight Decay

$$L(w) = L_{data}(w)$$

$$g_t = \nabla L_{data}(w_t)$$

$$s_t = \text{optimizer}(g_t) + 2\lambda w_t$$

$$w_{t+1} = w_t - \alpha s_t$$

# L2 Regularization vs Weight Decay

## Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

But they are not the same for adaptive methods (AdaGrad, RMSProp, Adam, etc)

## L2 Regularization

$$L(w) = L_{data}(w) + \lambda |w|^2$$

$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

## Weight Decay

$$L(w) = L_{data}(w)$$

$$g_t = \nabla L_{data}(w_t)$$

$$s_t = \text{optimizer}(g_t) + 2\lambda w_t$$

$$w_{t+1} = w_t - \alpha s_t$$

# AdamW: Decoupled Weight Decay

**Algorithm 2** Adam with L<sub>2</sub> regularization and Adam with decoupled weight decay (AdamW)

---

```
1: given  $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ ,  $\epsilon = 10^{-8}$ ,  $\lambda \in \mathbb{R}$ 
2: initialize time step  $t \leftarrow 0$ , parameter vector  $\theta_{t=0} \in \mathbb{R}^n$ , first moment vector  $m_{t=0} \leftarrow \theta$ , second moment
   vector  $v_{t=0} \leftarrow \theta$ , schedule multiplier  $\eta_{t=0} \in \mathbb{R}$ 
3: repeat
4:    $t \leftarrow t + 1$ 
5:    $\nabla f_t(\theta_{t-1}) \leftarrow \text{SelectBatch}(\theta_{t-1})$                                  $\triangleright$  select batch and return the corresponding gradient
6:    $\mathbf{g}_t \leftarrow \nabla f_t(\theta_{t-1}) + \lambda \theta_{t-1}$ 
7:    $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t$                                  $\triangleright$  here and below all operations are element-wise
8:    $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2$ 
9:    $\hat{\mathbf{m}}_t \leftarrow \mathbf{m}_t / (1 - \beta_1^t)$                                           $\triangleright \beta_1$  is taken to the power of  $t$ 
10:   $\hat{\mathbf{v}}_t \leftarrow \mathbf{v}_t / (1 - \beta_2^t)$                                           $\triangleright \beta_2$  is taken to the power of  $t$ 
11:   $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$                                  $\triangleright$  can be fixed, decay, or also be used for warm restarts
12:   $\theta_t \leftarrow \theta_{t-1} - \eta_t \left( \alpha \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \epsilon) + \lambda \theta_{t-1} \right)$ 
13: until stopping criterion is met
14: return optimized parameters  $\theta_t$ 
```

---

# AdamW: Decoupled Weight Decay

**Algorithm 2** Adam with L<sub>2</sub> regularization and Adam with decoupled weight decay (AdamW)

- 1: **given**  $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
  - 2: **initialize** time step  $t \leftarrow 0$ , parameter vector  $\theta_{t=0} \in \mathbb{R}^n$ , first moment vector  $m_{t=0} \leftarrow \mathbf{0}$ , second moment vector  $v_{t=0} \leftarrow \mathbf{0}$ , schedule multiplier  $n_{t=0} \in \mathbb{R}$

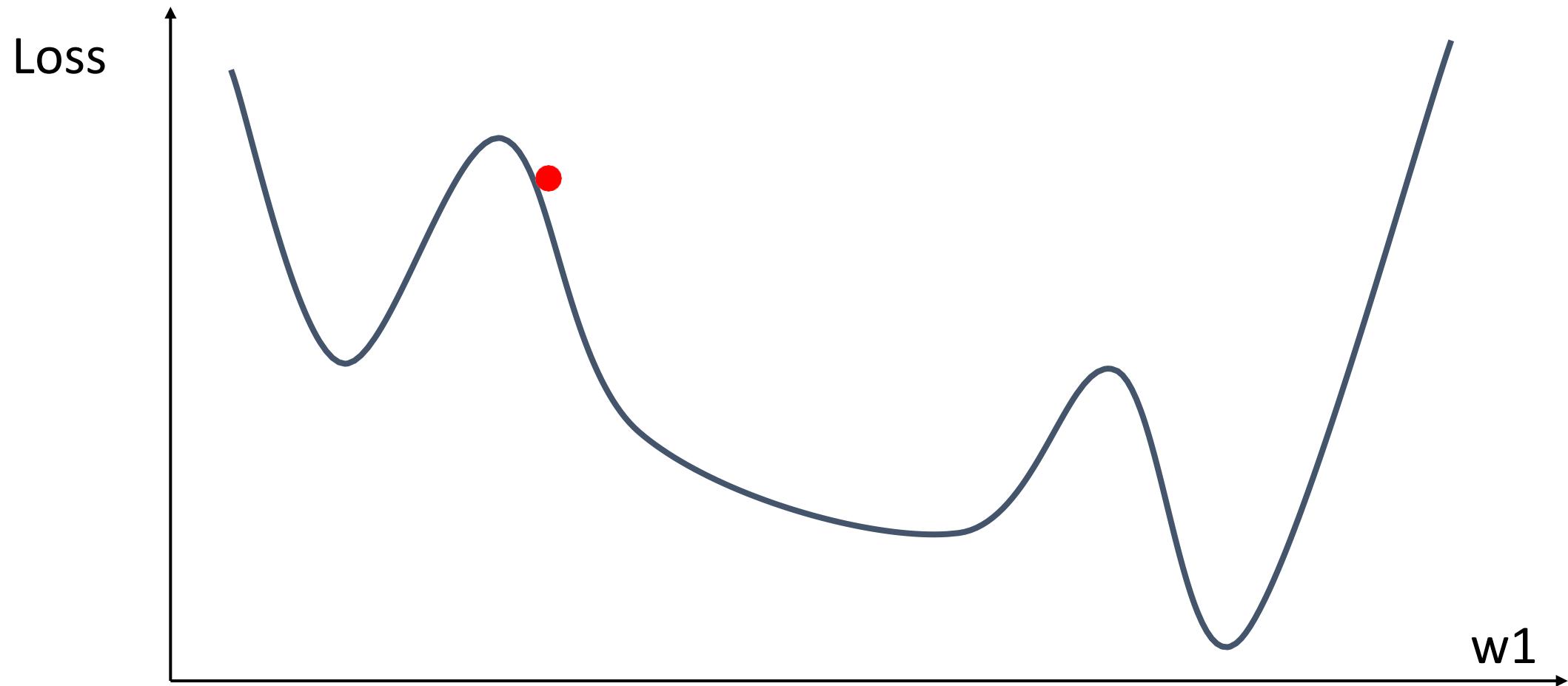
AdamW should probably be your  
“default” optimizer for new problems

- ```

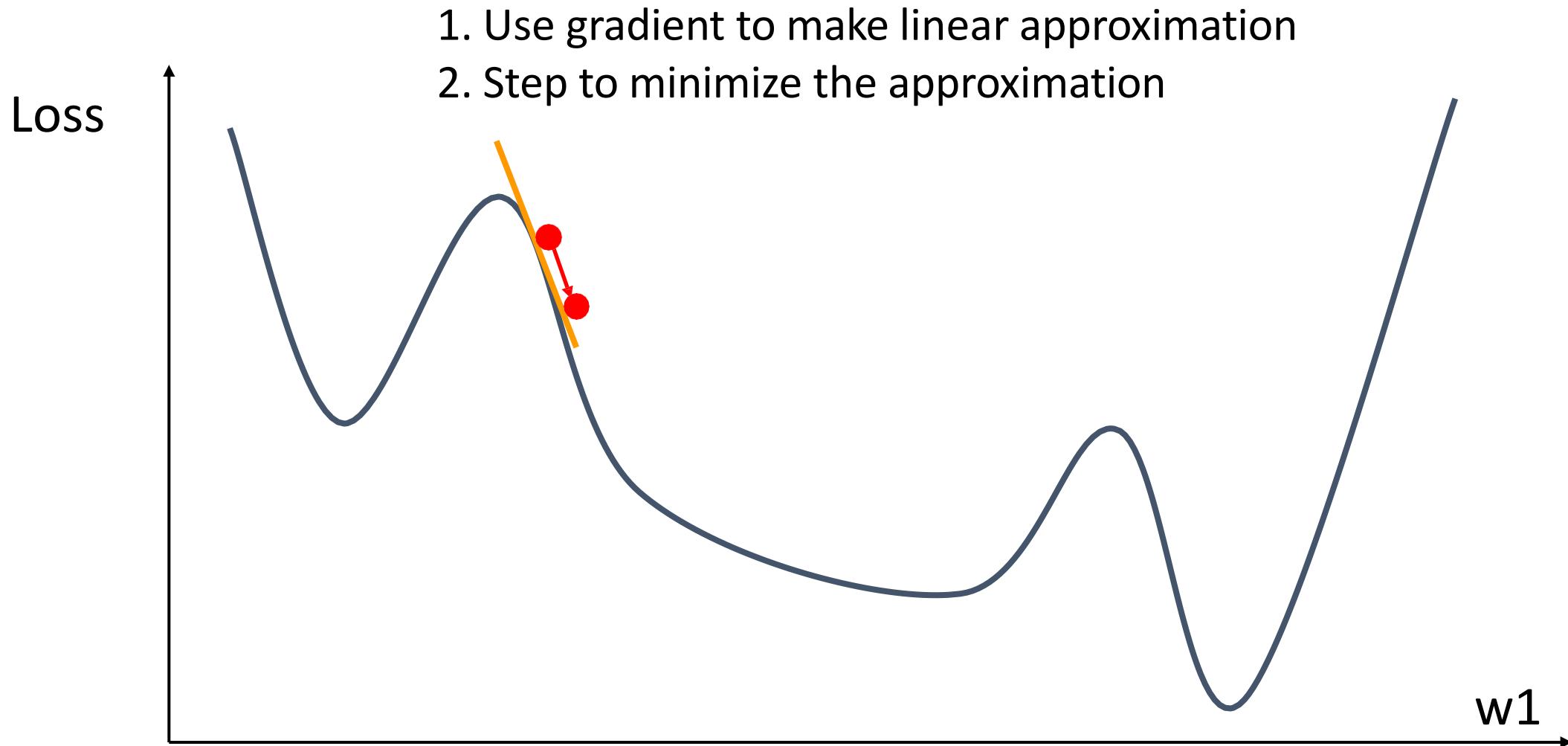
11:    $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$             $\triangleright$  can be fixed, decay, or also be used for warm restarts
12:    $\theta_t \leftarrow \theta_{t-1} - \eta_t \left( \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) + \lambda \theta_{t-1} \right)$ 
13: until stopping criterion is met
14: return optimized parameters  $\theta_t$ 

```

# So far: First-Order Optimization

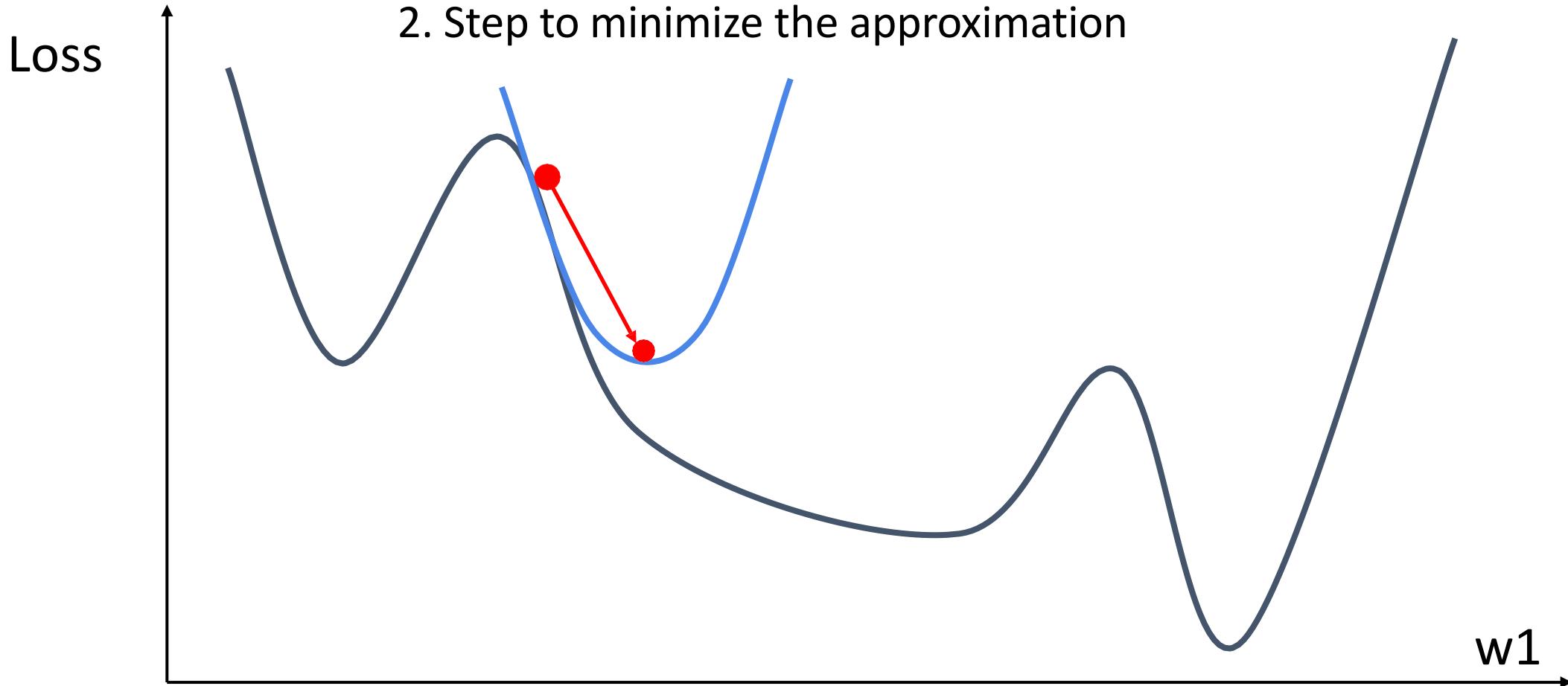


# So far: First-Order Optimization



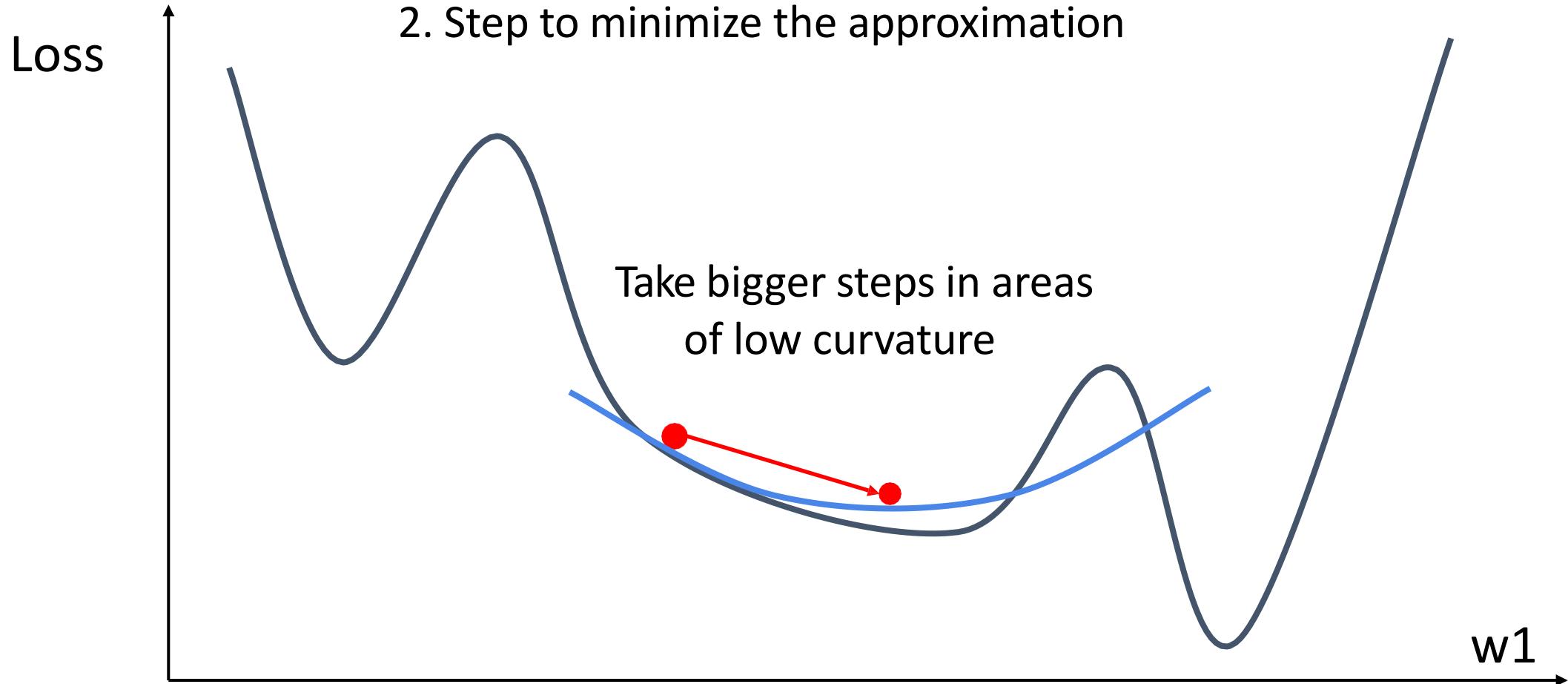
# Second-Order Optimization

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation



# Second-Order Optimization

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation



# Second-Order Optimization

Second-Order Taylor Expansion:

$$L(w) \approx L(w_0) + (w - w_0)^\top \nabla_w L(w_0) + \frac{1}{2}(w - w_0)^\top \mathbf{H}_w L(w_0)(w - w_0)$$

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

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

# Second-Order Optimization

Second-Order Taylor Expansion:

$$L(w) \approx L(w_0) + (w - w_0)^\top \nabla_w L(w_0) + \frac{1}{2}(w - w_0)^\top \mathbf{H}_w L(w_0)(w - w_0)$$

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

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

Q: Why is this impractical?

# Second-Order Optimization

Second-Order Taylor Expansion:

$$L(w) \approx L(w_0) + (w - w_0)^\top \nabla_w L(w_0) + \frac{1}{2}(w - w_0)^\top \mathbf{H}_w L(w_0)(w - w_0)$$

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

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

Q: Why is this impractical?

Hessian has  $O(N^2)$  elements  
Inverting takes  $O(N^3)$   
 $N = (\text{Tens or Hundreds of}) \text{ Millions}$

# Second-Order Optimization

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

- 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.*

# Second-Order Optimization: L-BFGS

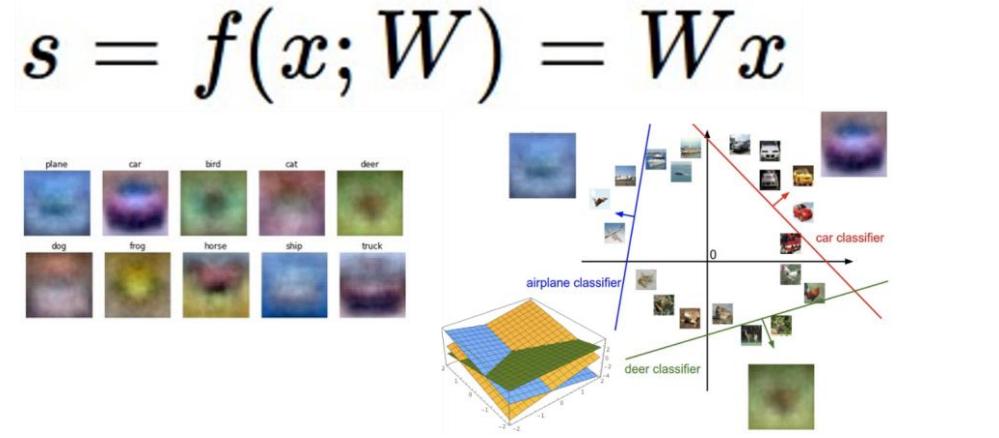
- **Usually works very well in full batch, deterministic mode**  
i.e. if you have a single, deterministic  $f(x)$  then L-BFGS will probably work very nicely
- **Does not transfer very well to mini-batch setting.** Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

## In practice:

- **Adam** is a good default choice in many cases  
**SGD+Momentum** can outperform Adam but may require more tuning
- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

# Summary

1. Use **Linear Models** for image classification problems
2. Use **Loss Functions** to express preferences over different choices of weights
3. Use **Regularization** to prevent overfitting to training data
4. Use **AdamW** to minimize our loss functions and train the model



$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{sj}}\right)$$
 Softmax

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

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

SVM