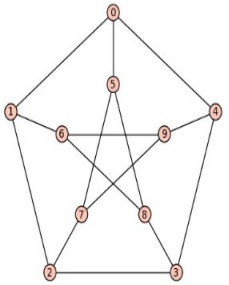
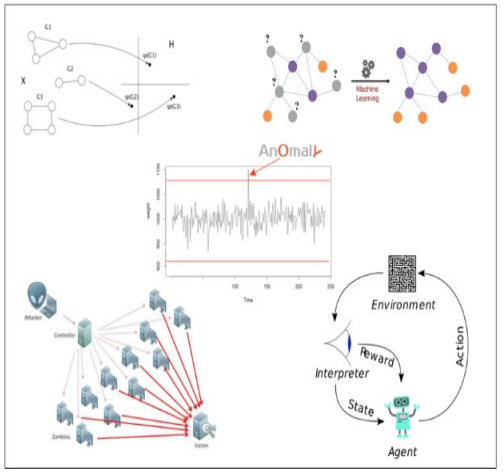
Optimizers

Pierre Pereira
Université Côte d'Azur / Inria Coati
pierre.pereira@inria.fr





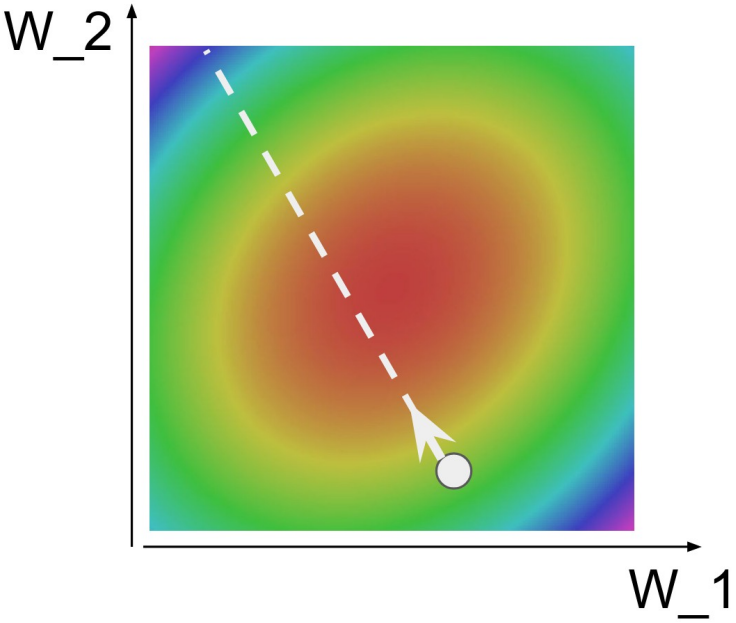
$$
\begin{aligned}
\min \quad & \sum_{e \in \mathcal{E}} y_e \\
\text{s.t.} \quad & \sum_{a \in A_i^+(u)} f_a^i - \sum_{a \in A_i^-(u)} f_a^i = \begin{cases} |V_i| - 1 & \text{if } u = s_i \\ -1 & \text{if } u \neq s_i \end{cases} && \forall u \in V_i,\ V_i \in C \\
& f_a^i \leq |V_i| \cdot x_a, && \forall V_i \in C, a \in A \\
& x_{(u,v)} \leq y_{uv}, && \forall uv \in \mathcal{E} \\
& x_{(v,u)} \leq y_{uv}, && \forall uv \in \mathcal{E}
\end{aligned}
$$

Sources multiples, principalement
Roger Grosse, Toronto

Optimization

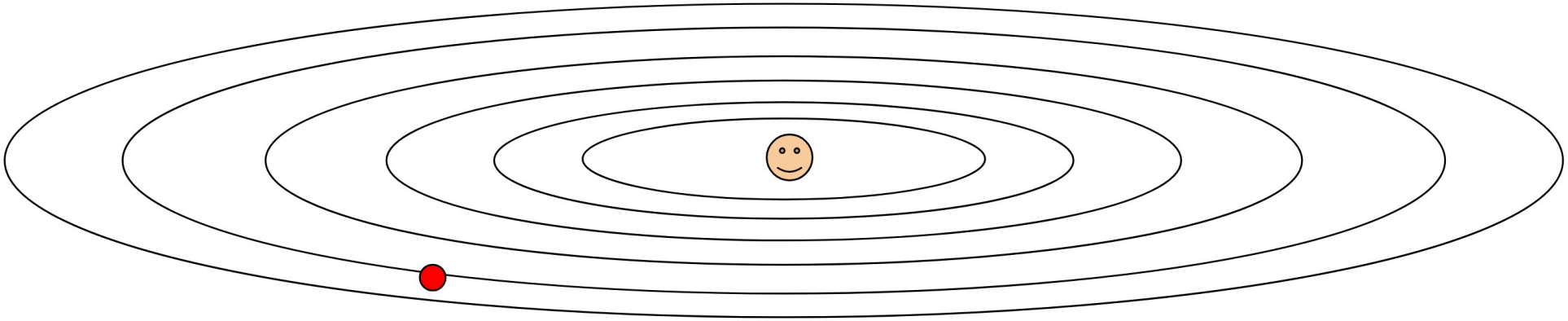
```
# Vanilla Gradient Descent

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



Optimization: Problems with SGD

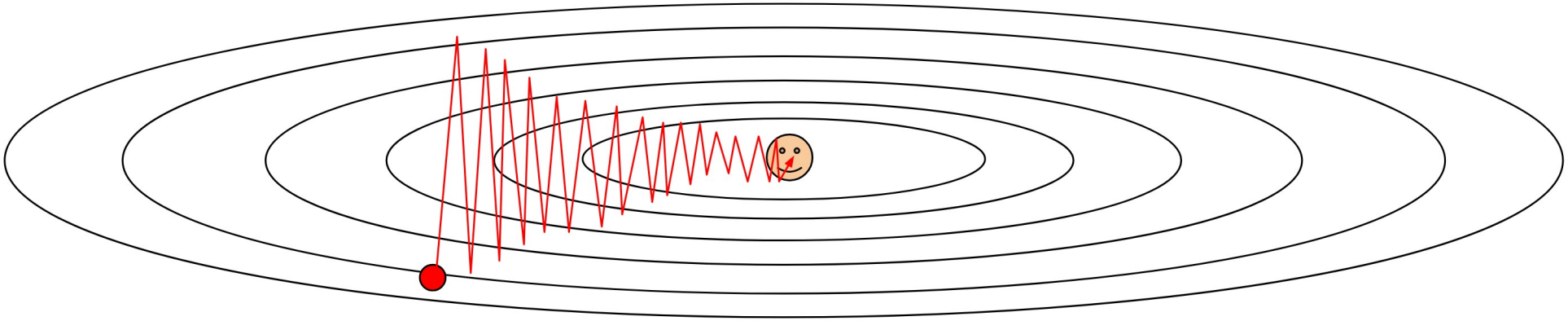
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?



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

Optimization: Problems with SGD

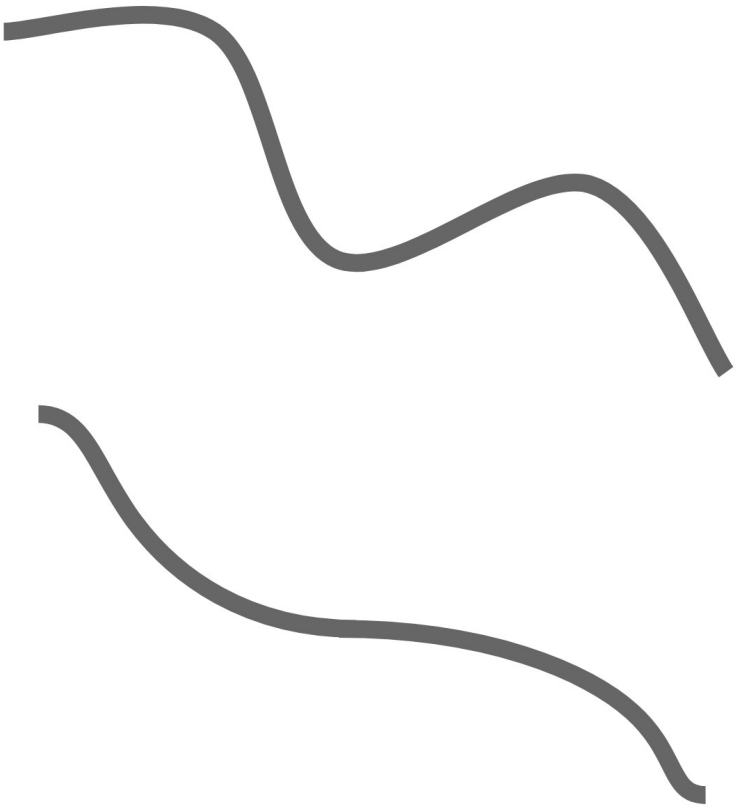
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



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

Optimization: Problems with SGD

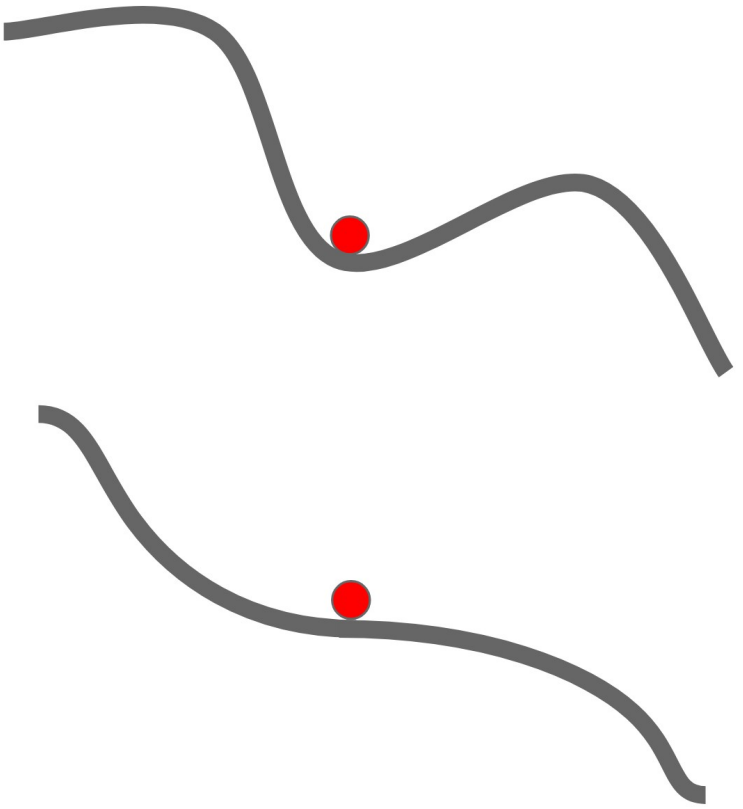
What if the loss function has a **local minima** or **saddle point**?



Optimization: Problems with SGD

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

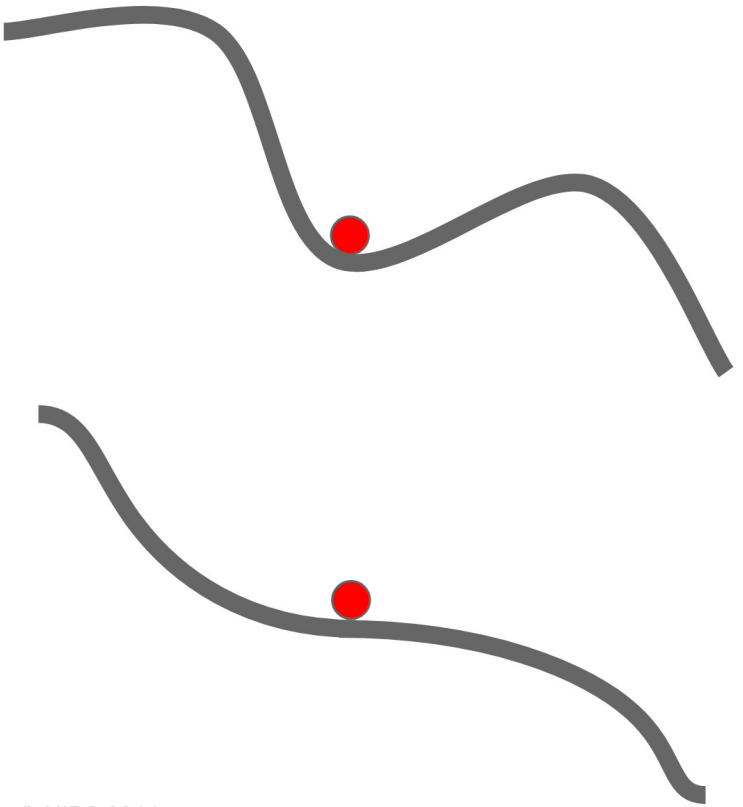
Zero gradient, gradient descent gets stuck



Optimization: Problems with SGD

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

Saddle points much more common in high dimension



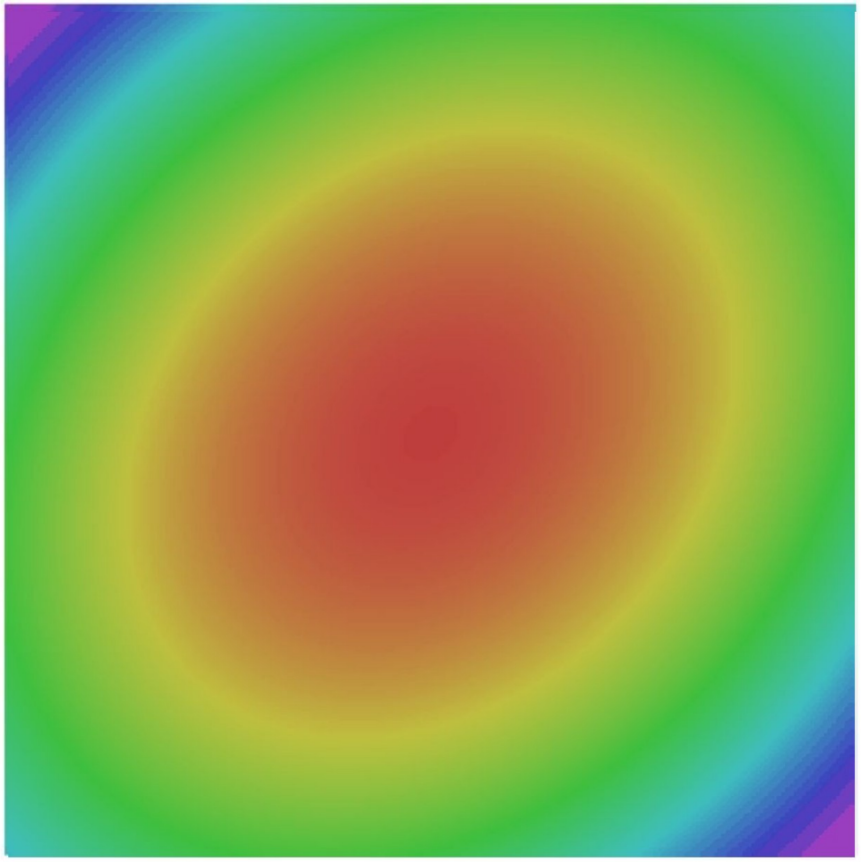
Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

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

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



SGD + Momentum

SGD

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

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

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

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

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

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

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

SGD+Momentum

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

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx - learning_rate * dx
  x += vx
```

SGD+Momentum

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

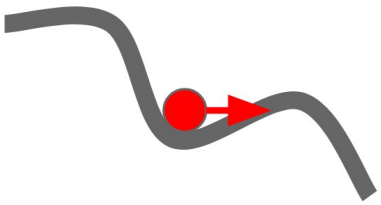
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

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

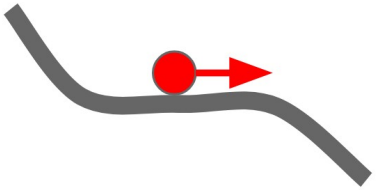
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

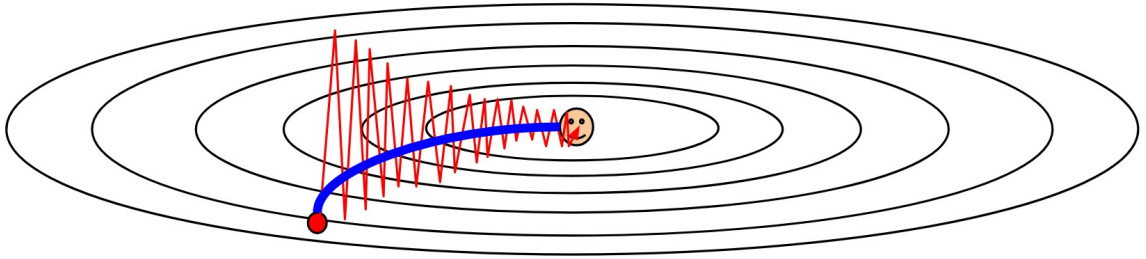
Local Minima



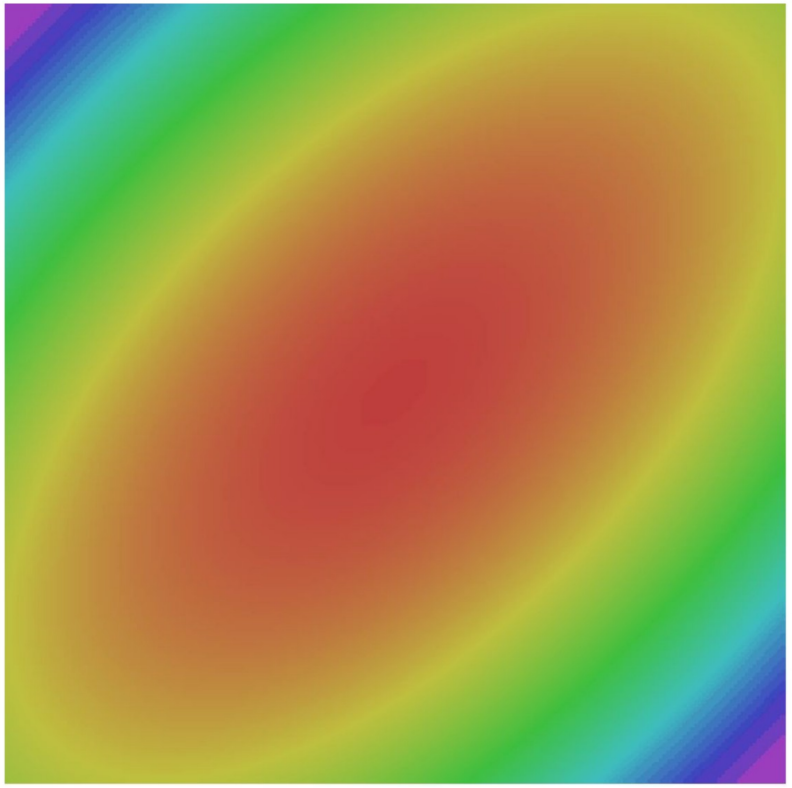
Saddle points



Poor Conditioning



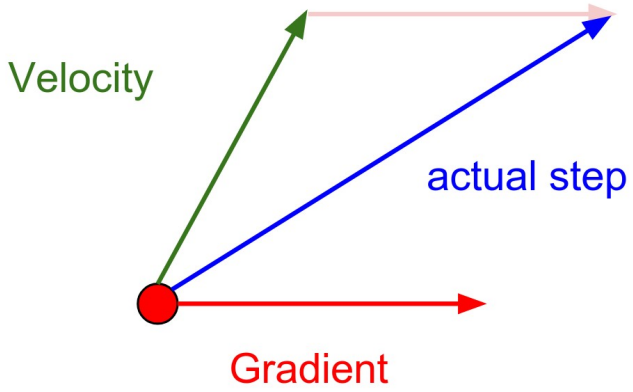
Gradient Noise



SGD SGD+Momentum

SGD+Momentum

Momentum update:

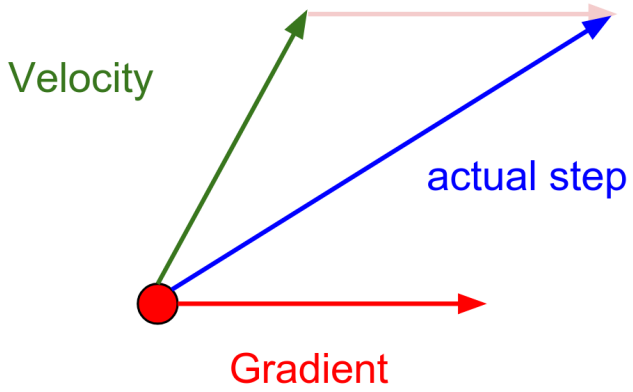


Combine gradient at current point with velocity to get step used to update weights

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

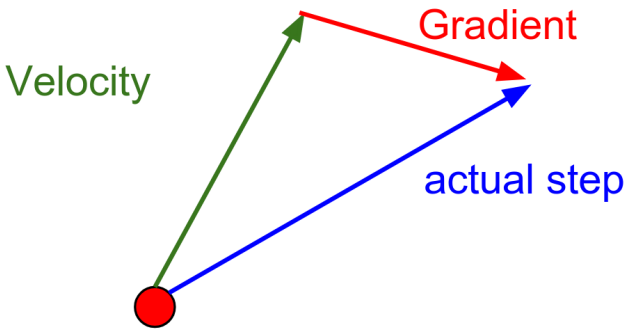
Nesterov Momentum

Momentum update:



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

Nesterov Momentum

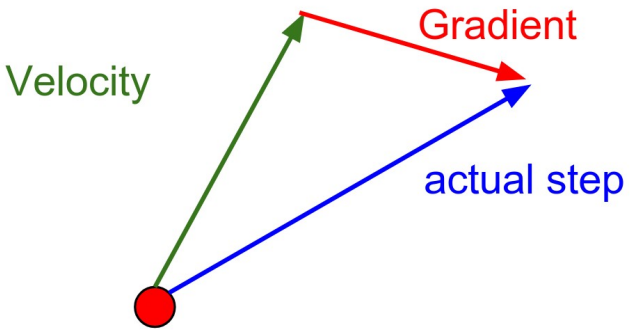


"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov, "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

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

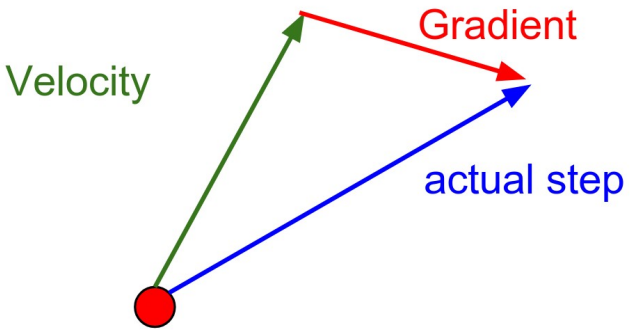


"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

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 $x_t, \nabla f(x_t)$

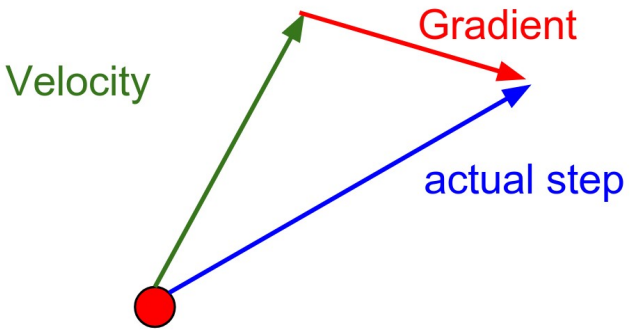


"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

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:

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

"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 $x_t, \nabla f(x_t)$

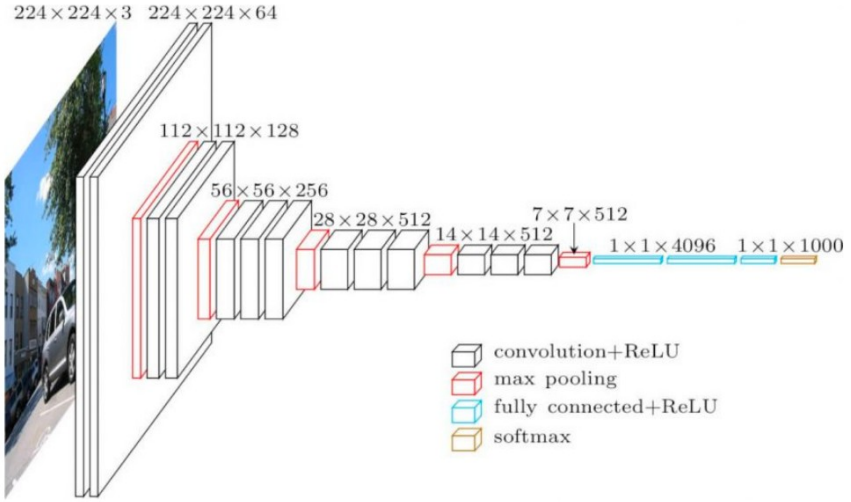
Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

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

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

Adaptive methods

The magnitude of the gradients often varies highly between layers due to, so a global learning rate may not work well.

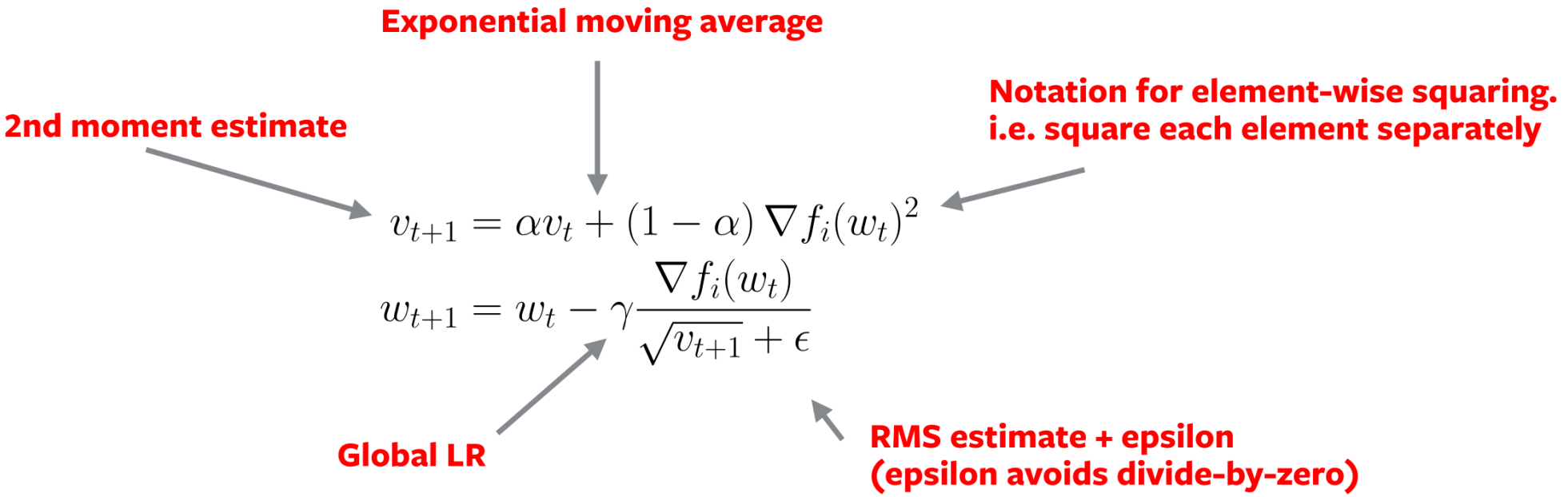


General **IDEA**: Instead of using the same learning rate for every weight in our network, **maintain an estimate of a better rate separately for each weight**.

The exact way of adapting to the learning rates varies between algorithms,
But most methods either **adapt** to the **variance** of the weights, or to the l**ocal curvature** of the problem.

RMSprop

Key **IDEA**: normalize by the **root-mean-square** of the gradient



Most adaptive methods use a moving average, it's a simple way to estimate a noisy quantity that changes over time.

If the expected value of gradient is small, this is similar to dividing by the standard deviation (i.e. whitening)

Adam: RMSprop with a kind of momentum

"Adaptive Moment Estimation"

Presented here without **bias-correction** (i.e. steady state Adam)

RMSProp

$$v_{t+1} = \alpha v_t + (1-\alpha)\nabla f_i(w_t)^2$$
$$w_{t+1} = w_t - \gamma \frac{\nabla f_i(w_t)}{\sqrt{v_{t+1}} + \epsilon}$$

Momentum (Exponential moving average)

2nd moment estimate (same as RMSProp)

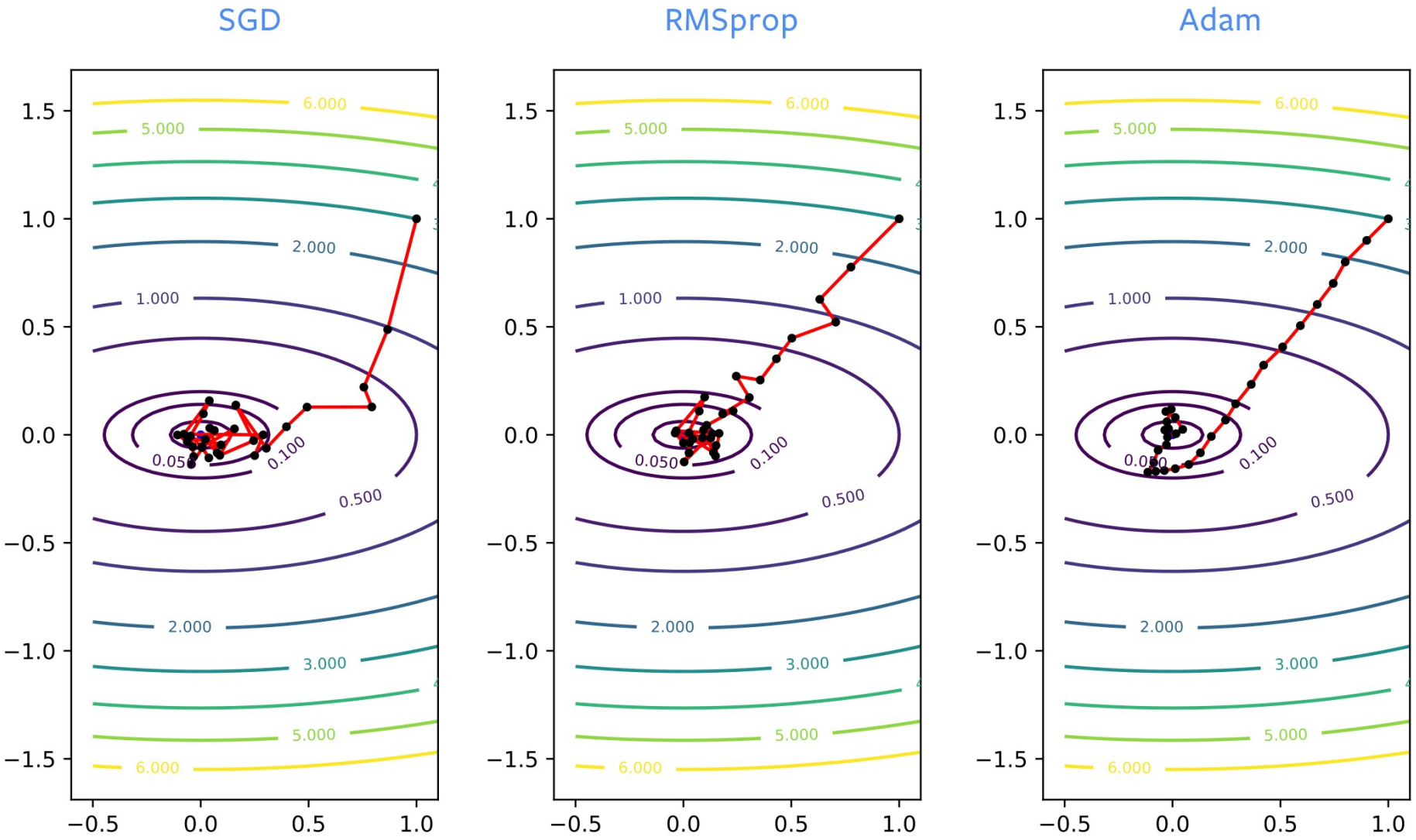
$$m_{t+1} = \beta v_t + (1-\beta)\nabla f_i(w_t)$$
$$v_{t+1} = \alpha v_t + (1-\alpha)\nabla f_i(w_t)^2$$
$$w_{t+1} = w_t - \gamma \frac{m_t}{\sqrt{v_{t+1}} + \epsilon}$$

Use m instead of the gradient

Just as momentum improves SGD, it improves RMSProp as well.
The exponential-moving-average method of updating momentum is equivalent to the standard form under rescaling. Nothing mysterious.

The full version of Adam has **bias-correction** as well, which just keeps the moving averages **unbiased** during early iterations. The algorithm quickly approaches the above steady state form.

Diederik, Kingma; Ba, Jimmy (2014). "Adam: A method for stochastic optimization"


SGD
RMSprop
Adam
6.000
5.000
2.000
1.000
0.500
0.100
0.050
3.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−0.5
0.0
0.5
1.0

Practical side

For poorly conditioned problems, Adam is often **much** better than SGD.
I recommend using Adam over RMSprop due to the clear advantages of momentum

BUT, Adam is poorly understood theoretically, and has known disadvantages:

- Does not converge at all on some simple example problems!
- Gives worse generalization error on many computer vision problems (i.e. ImageNet)
- Requires more memory than SGD
- Has 2 momentum parameters, so some tuning may be needed

It's a good idea to try both SGD + momentum and Adam with a sweep of different learning rates, and use whichever works better on held out data