



ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Huấn luyện mạng nơ-ron (*Phần 2*)

Hà Nội, 8/2021

Nội dung

1. Các kỹ thuật chuẩn hoá
2. Các giải thuật tối ưu cho mạng nơ-ron
3. Chiến lược thay đổi tốc độ học
4. Lựa chọn siêu tham số
5. Kỹ thuật kết hợp nhiều mô hình (ensemble)
6. Kỹ thuật học chuyển giao (transfer learning)

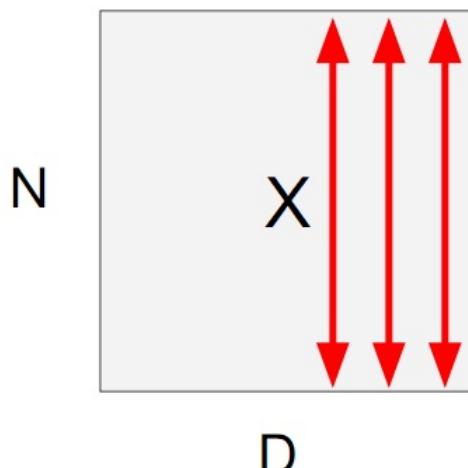
Các kỹ thuật chuẩn hóa

Batch Normalization

- Muốn hàm kích hoạt có phân bố đầu ra với trung bình bằng 0 và độ lệch chuẩn đơn vị? Hãy biến đổi theo ý tưởng

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Input: $x : N \times D$



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,
shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel var,
shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x,
Shape is N x D

Batch Normalization

- Ràng buộc kỳ vọng bằng 0 và độ lệch chuẩn đơn vị là quá chặt! Có thể khiến mô hình bị underfitting.
→ Nới lỏng cho mô hình, tạo lối thoát cho mô hình nếu nó không muốn bị ràng buộc.

Input: $x : N \times D$

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,
shape is D

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$,
 $\beta = \mu$ will recover the identity function!

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel var,
shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x,
Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output,
Shape is N x D

Batch Normalization

- Không thể tính kỳ vọng và phương sai theo lô dữ liệu (batch) lúc suy diễn

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$,
 $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j} \quad \text{Per-channel mean, shape is } D$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is } D$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \quad \text{Normalized x, Shape is } N \times D$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is } N \times D$$

Batch Normalization

- Lúc suy diễn, BN đơn giản là phép biến đổi tuyến tính. Có thể áp dụng phía sau lớp FC hoặc conv

Input: $x : N \times D$

μ_j = (Running) average of values seen during training

Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta : D$

σ_j^2 = (Running) average of values seen during training

Per-channel var, shape is D

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x,
Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output,
Shape is N x D

Batch Normalization

Batch Normalization for
fully-connected networks

\mathbf{x} : N × D

Normalize



μ, σ : 1 × D

γ, β : 1 × D

$$y = \gamma(x - \mu) / \sigma + \beta$$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

\mathbf{x} : N×C×H×W

Normalize

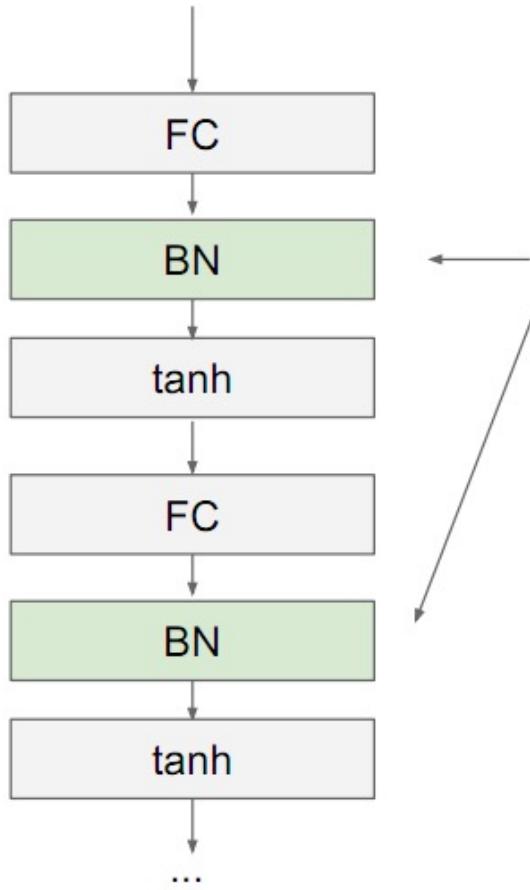


μ, σ : 1×C×1×1

γ, β : 1×C×1×1

$$y = \gamma(x - \mu) / \sigma + \beta$$

Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Ưu điểm của BN

- Dễ dàng hơn khi huấn luyện các mạng sâu!
- Cải thiện luồng gradient
- Cho phép huấn luyện với tốc độ học cao hơn, hội tụ nhanh hơn
- Mạng ổn định hơn, đỡ phụ thuộc hơn với khởi tạo trọng số
- Một kiểu ràng buộc khi huấn luyện
- Khi suy diễn không cần tính toán thêm, đơn giản là biến đổi tuyến tính
- **Khi huấn luyện và khi suy diễn làm việc khác nhau: đây là nguồn gốc gây ra nhiều lỗi!**

Chuẩn hóa theo lớp

Batch Normalization for
fully-connected networks

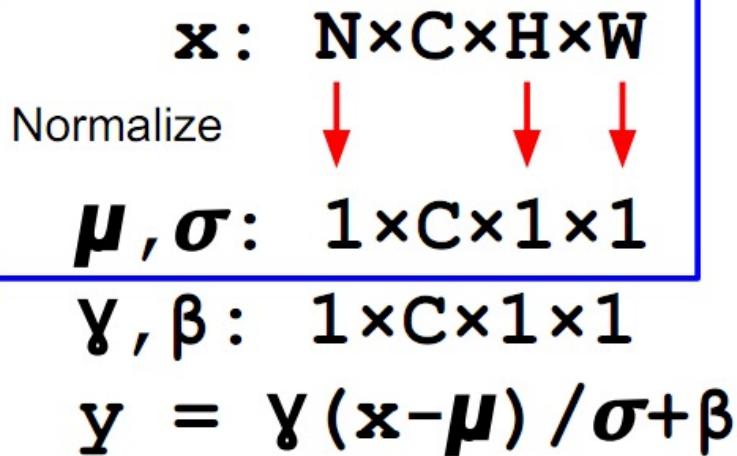
$$\begin{array}{l} \mathbf{x}: N \times D \\ \text{Normalize} \quad \downarrow \\ \boldsymbol{\mu}, \sigma: 1 \times D \\ \gamma, \beta: 1 \times D \\ \mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \beta \end{array}$$

Layer Normalization for
fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks

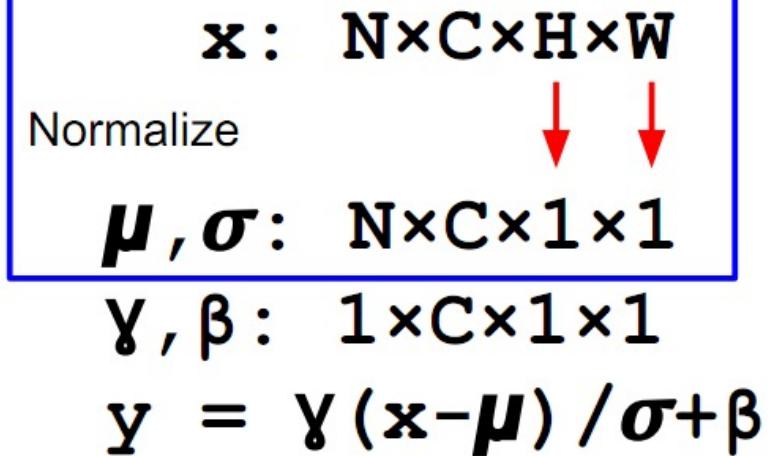
$$\begin{array}{l} \mathbf{x}: N \times D \\ \text{Normalize} \quad \downarrow \\ \boldsymbol{\mu}, \sigma: N \times 1 \\ \gamma, \beta: 1 \times D \\ \mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \beta \end{array}$$

Chuẩn hóa theo mẫu

Batch Normalization for convolutional networks

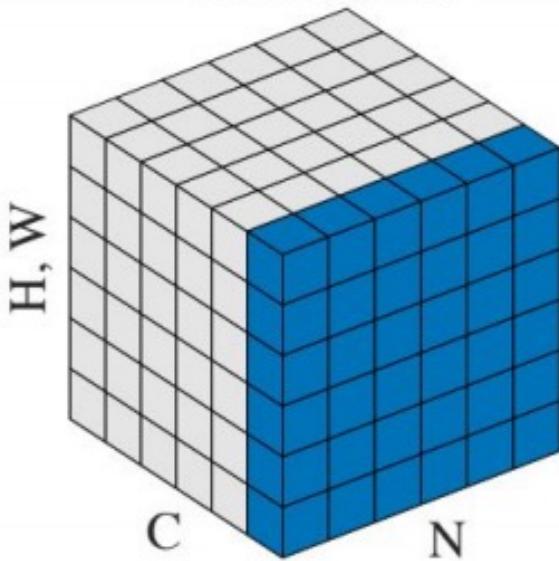


Instance Normalization for convolutional networks
Same behavior at train / test!

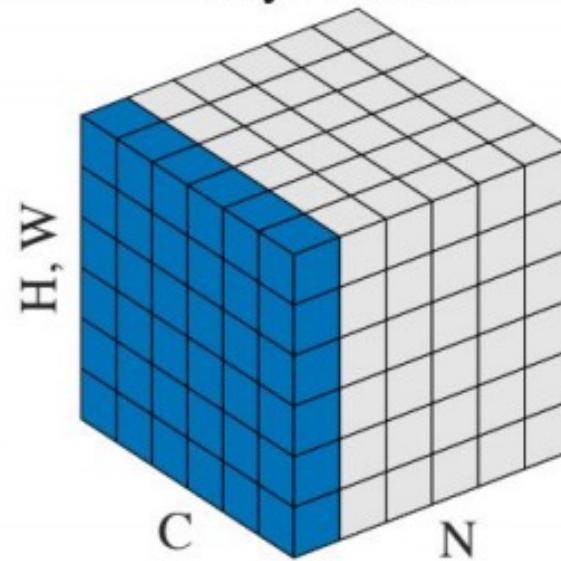


So sánh các phương pháp chuẩn hóa

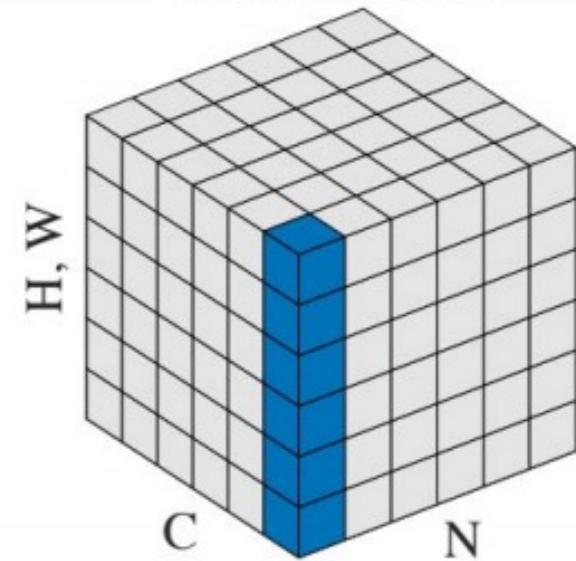
Batch Norm



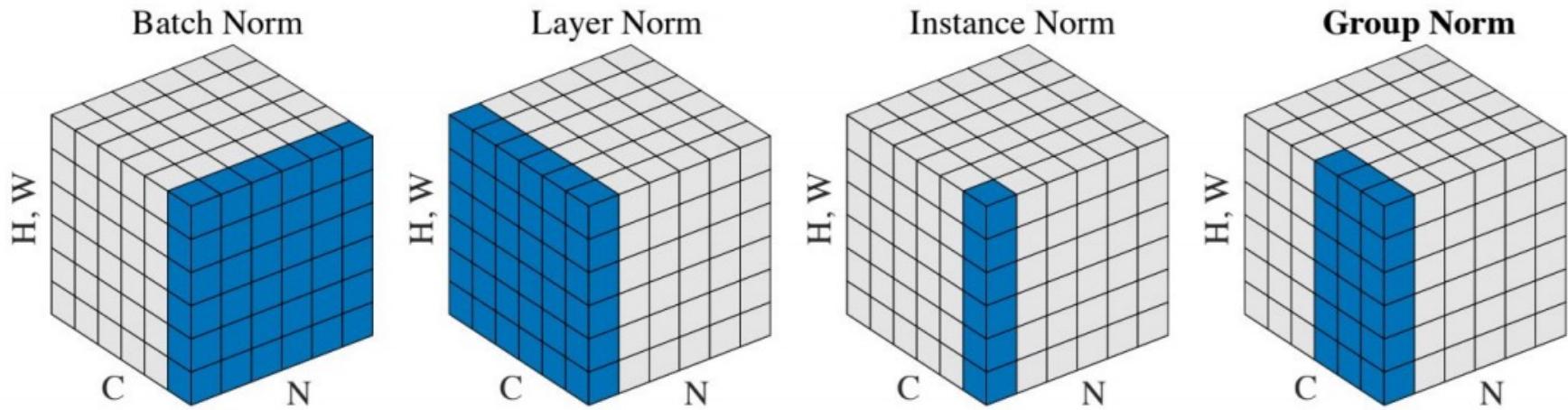
Layer Norm



Instance Norm



Chuẩn hóa theo nhóm



Các giải thuật tối ưu

Hàm mục tiêu

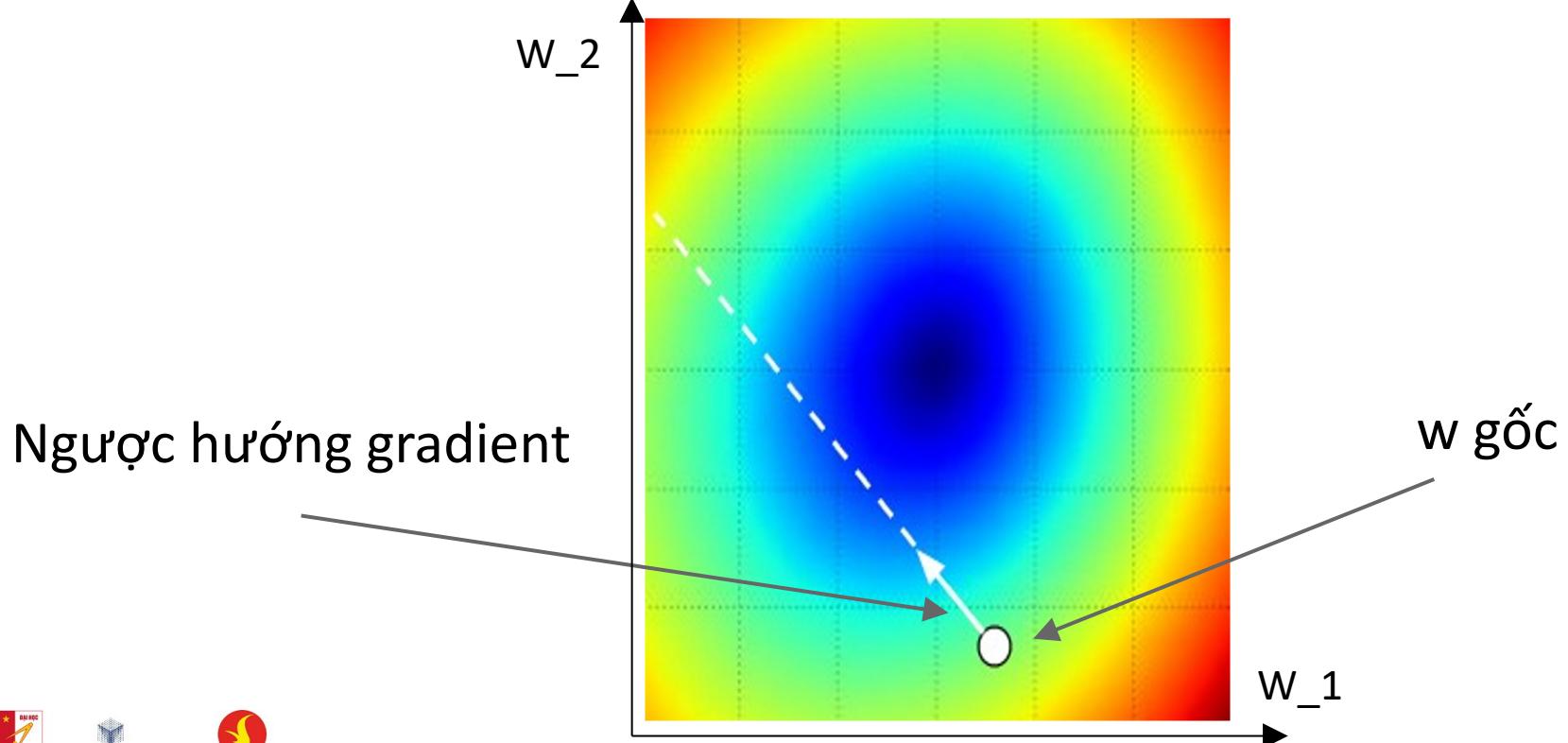


Phương pháp SGD

Vanilla Gradient Descent

while True:

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



SGD

- Chỉ dùng một phần nhỏ dữ liệu huấn luyện để xấp xỉ gradient

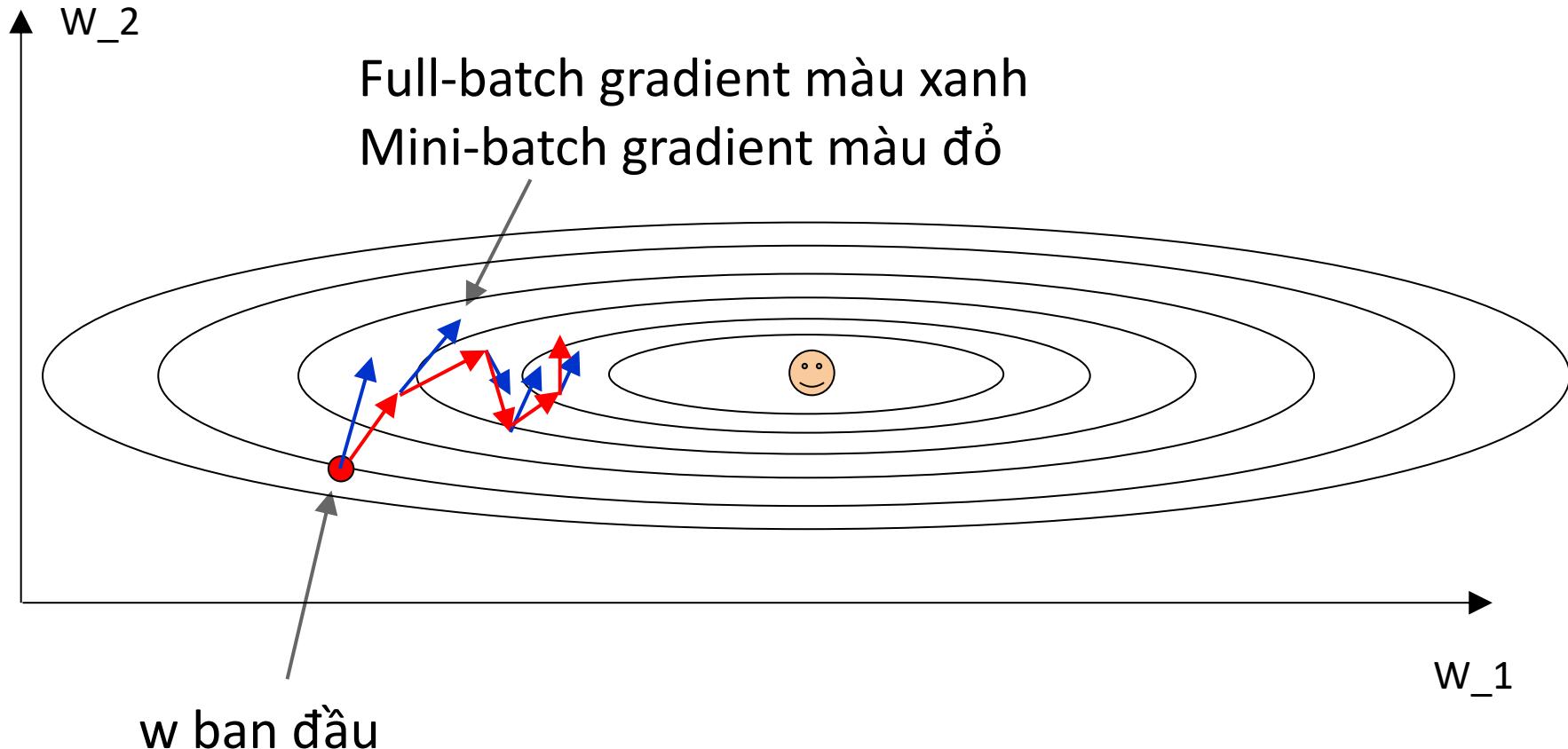
```
# Vanilla Minibatch Gradient Descent
```

```
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
```

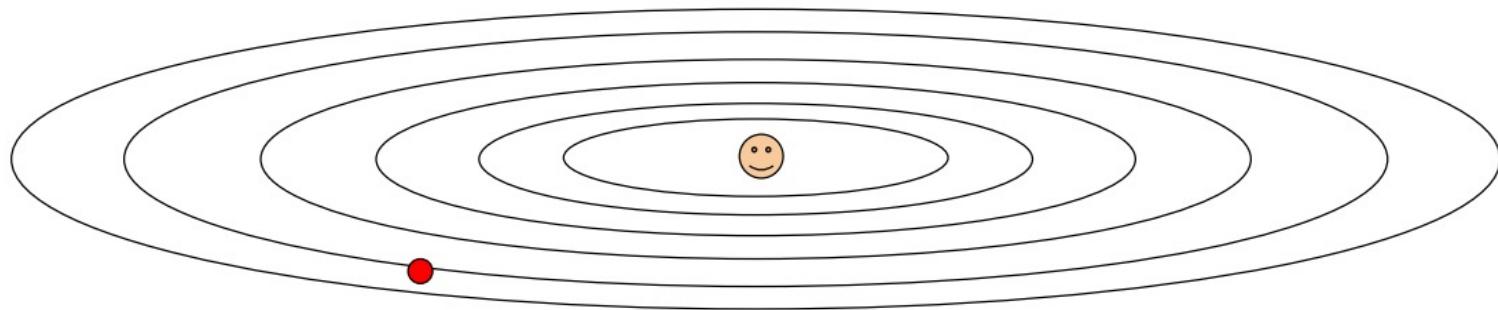
SGD

- Minh họa SGD



Vấn đề với SGD

- Điều gì sẽ xảy ra khi hàm mục tiêu thay đổi nhanh theo một chiều và thay đổi chậm theo chiều khác?
- Khi đó SGD sẽ làm việc như thế nào?

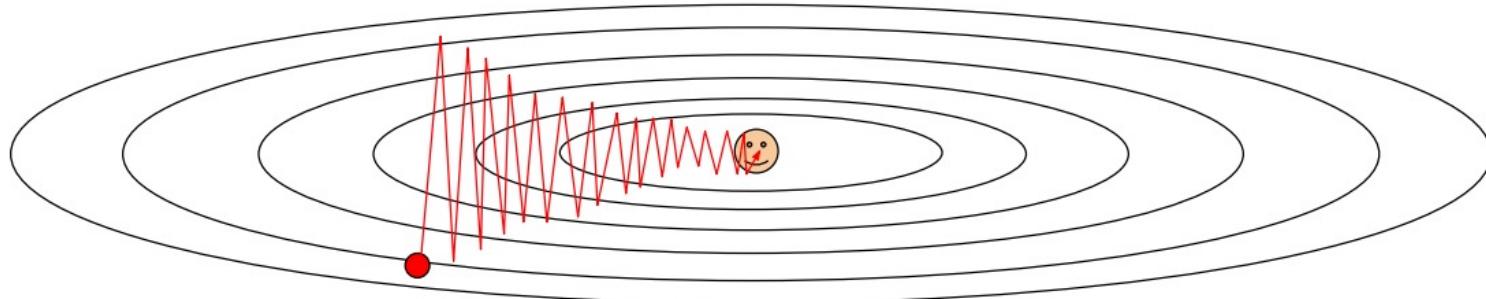


Hàm mục tiêu có **số điều kiện** lớn: tỉ lệ giữa giá trị riêng lớn nhất và giá trị riêng nhỏ nhất của ma trận Hessian là lớn.

Vấn đề với SGD

- Điều gì sẽ xảy ra khi hàm mục tiêu thay đổi nhanh theo một chiều và thay đổi chậm theo chiều khác?
- Khi đó SGD sẽ làm việc như thế nào?

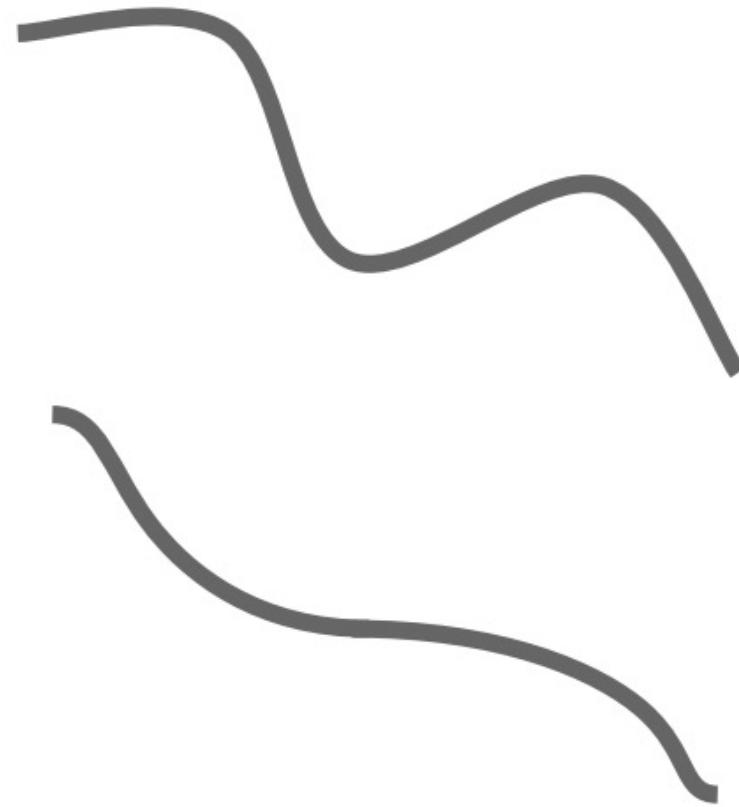
Thuật toán hội tụ rất chậm, nhảy từ bên này qua bên kia bề mặt hàm mục tiêu



Hàm mục tiêu có **số điều kiện** lớn: tỉ lệ giữa giá trị riêng lớn nhất và giá trị riêng nhỏ nhất của ma trận Hessian là lớn.

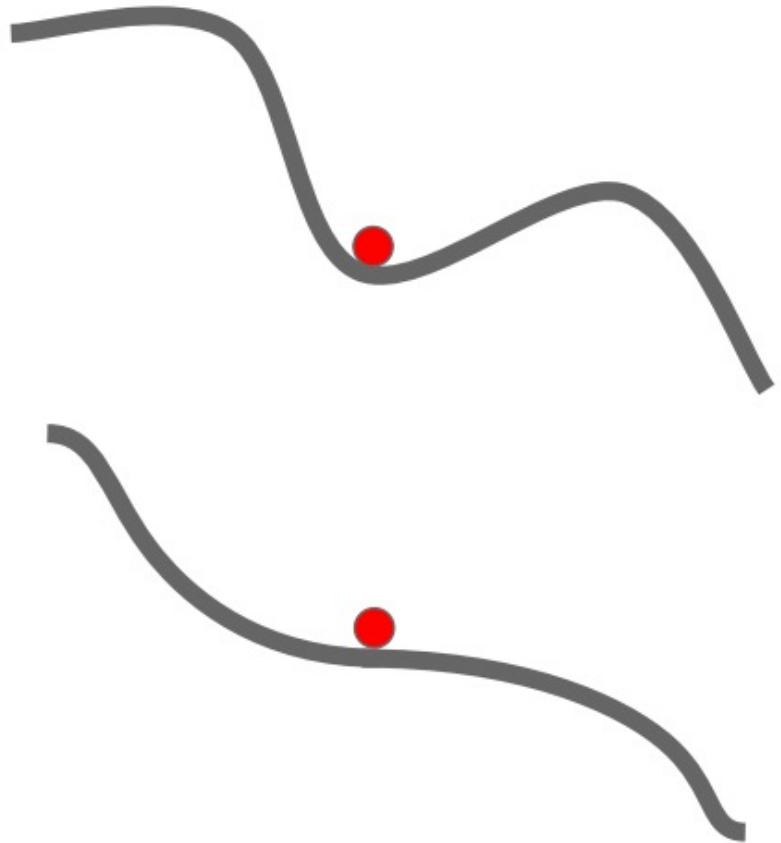
Vấn đề với SGD

- Chuyện gì xảy ra nếu hàm mục tiêu có cực tiểu địa phương hoặc điểm yên ngựa (saddle point)?



Vấn đề với SGD

- Chuyện gì xảy ra nếu hàm mục tiêu có cực tiểu địa phương hoặc điểm yên ngựa (saddle point)?
- Gradient bằng 0, thuật toán SGD bị tắc
- Điểm yên ngựa thường xuất hiện với các hàm mục tiêu nhiều biến

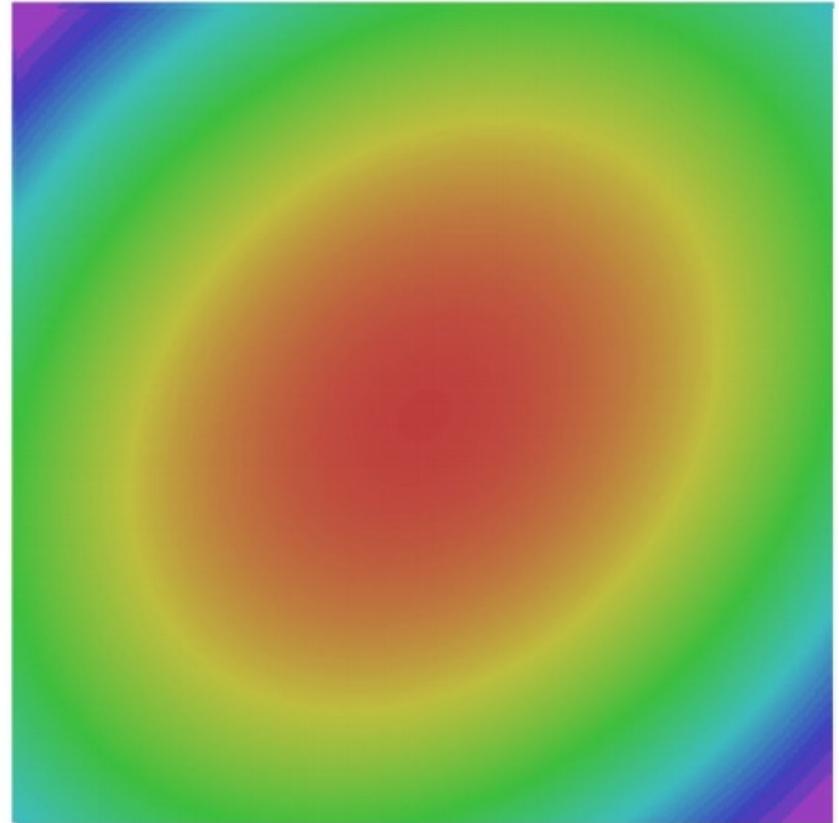


Vấn đề với SGD

- SGD xấp xỉ gradient theo từng lô dữ liệu nên thường rất nhiễu

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

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



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

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

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

- Xây dựng đại lượng “vận tốc” bằng trung bình dịch chuyển của gradients
- Lực ma sát rho thường bằng 0.9 hoặc 0.99.
- Tại thời điểm ban đầu rho có thể thấp hơn do hướng di chuyển chưa rõ ràng, ví dụ rho = 0.5

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

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

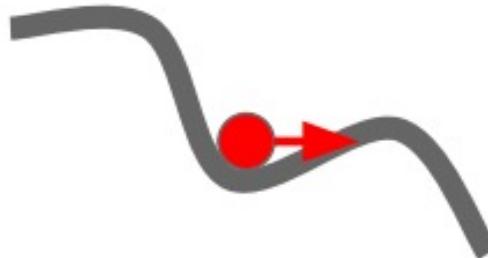
$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

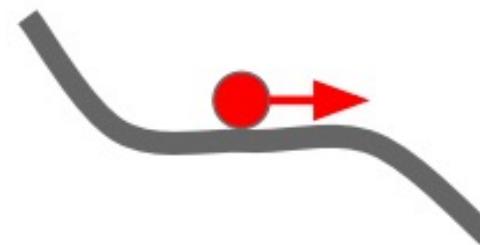
- SGD + momentum có thể phát triển theo nhiều cách khác nhau nhưng chúng tương đương nhau và đều đưa ra cùng một dãy x

SGD + momentum

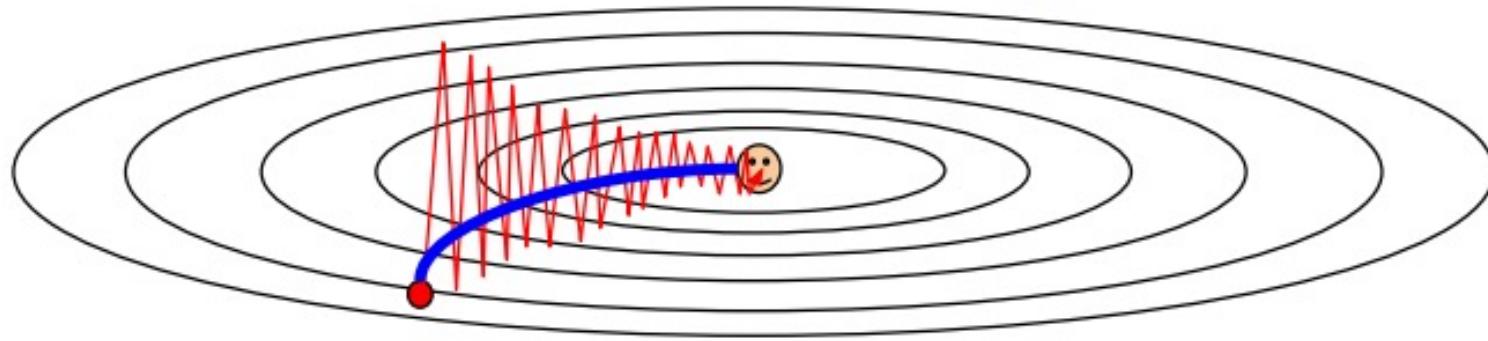
Local Minima



Saddle points



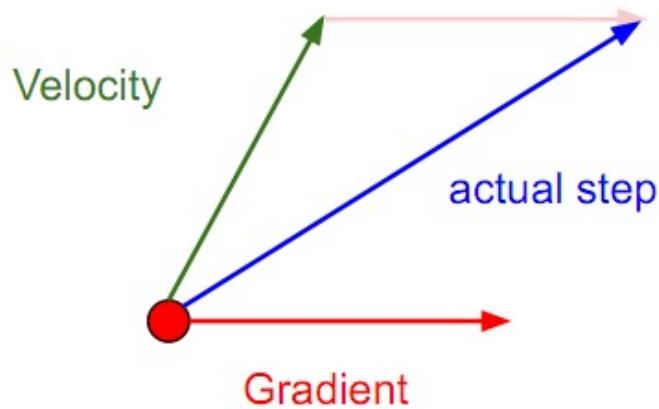
Poor Conditioning



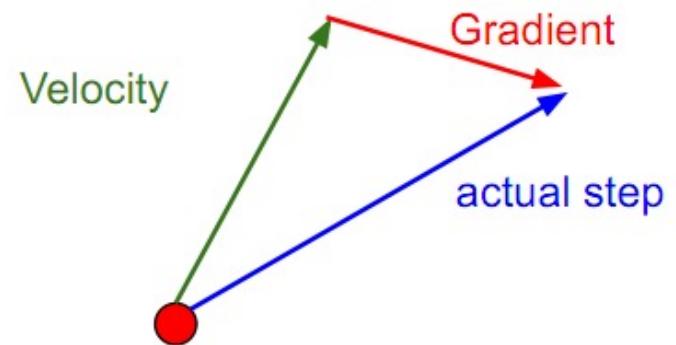
SGD+Momentum

Nesterov Momentum

Momentum update:



Nesterov Momentum



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

Nesterov Momentum

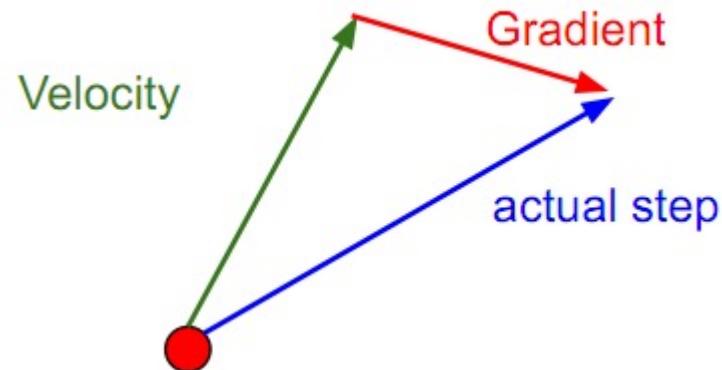
$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

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

- Thường người ta muốn tính theo $x_t, \nabla f(x_t)$
- Đặt $\tilde{x}_t = x_t + \rho v_t$ và chuyển về

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

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



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

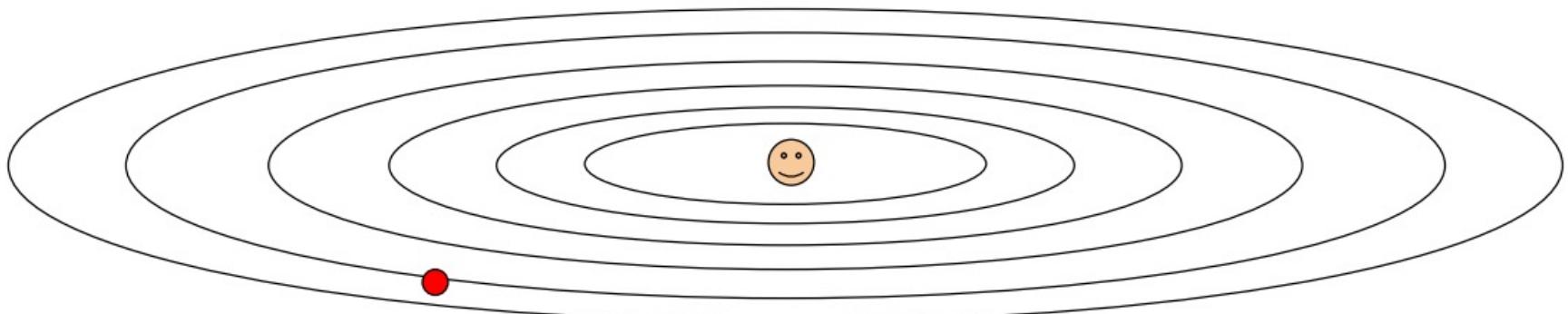
AdaGrad

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

- Mỗi trọng số có tốc độ học riêng: “Per-parameter learning rates” hoặc “adaptive learning rates”
- Tốc độ học của mỗi trọng số tỉ lệ nghịch với tổng bình phương độ lớn đạo hàm riêng của hàm mục tiêu đối với trọng số đó ở các bước trước

AdaGrad

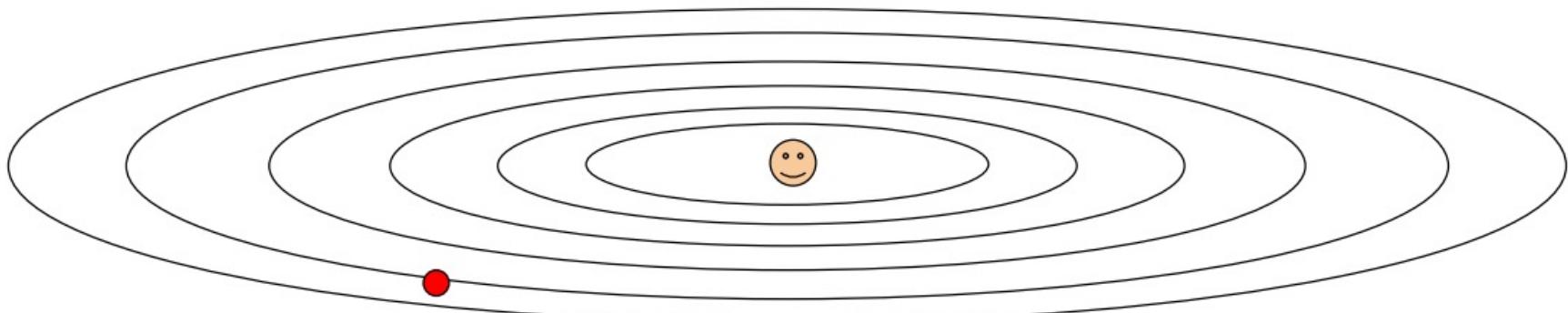
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



- Q1: Điều gì xảy ra với 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)
```



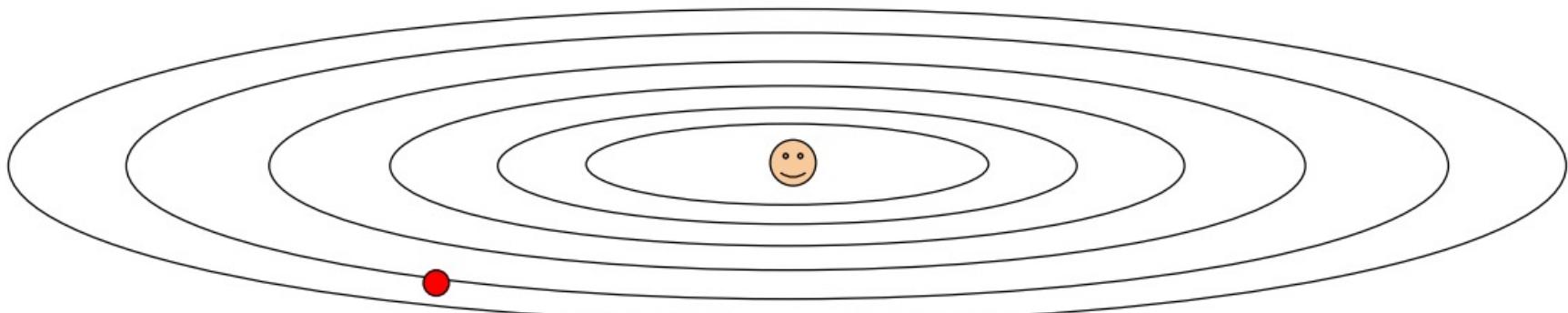
- Q1: Điều gì xảy ra với AdaGrad?

Tốc độ di chuyển theo hướng dốc được hãm dần

Tốc độ di chuyển theo hướng thoái được tăng tốc

AdaGrad

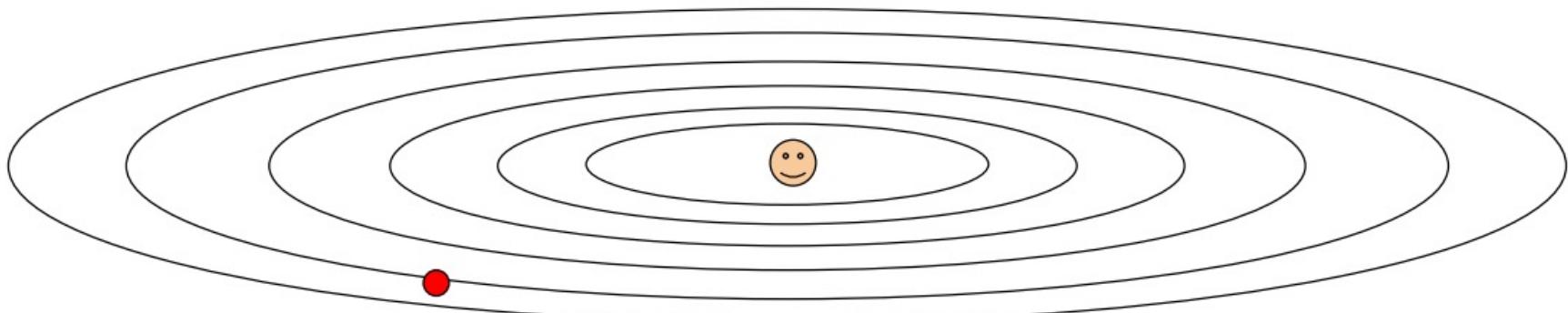
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



- Q2: Bước di chuyển thay đổi như thế nào khi số vòng lặp tăng dần?

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: Bước di chuyển thay đổi như thế nào khi số vòng lặp tăng dần?

Tiến tới 0

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 đơn giản

```
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

Có thể xem như là RMSProp + Momentum

Adam đầy đủ

```
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))
```

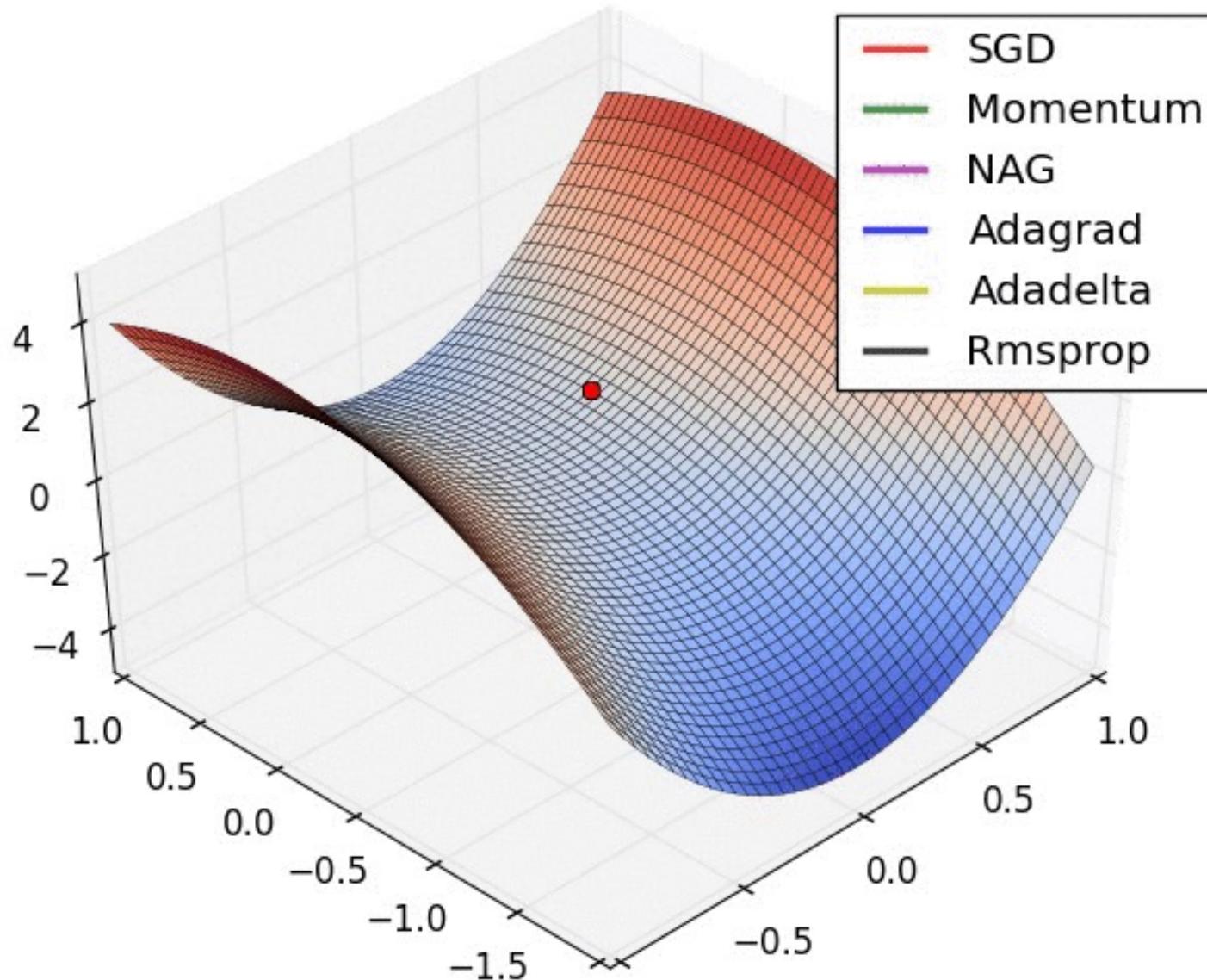
Momentum

Bias correction

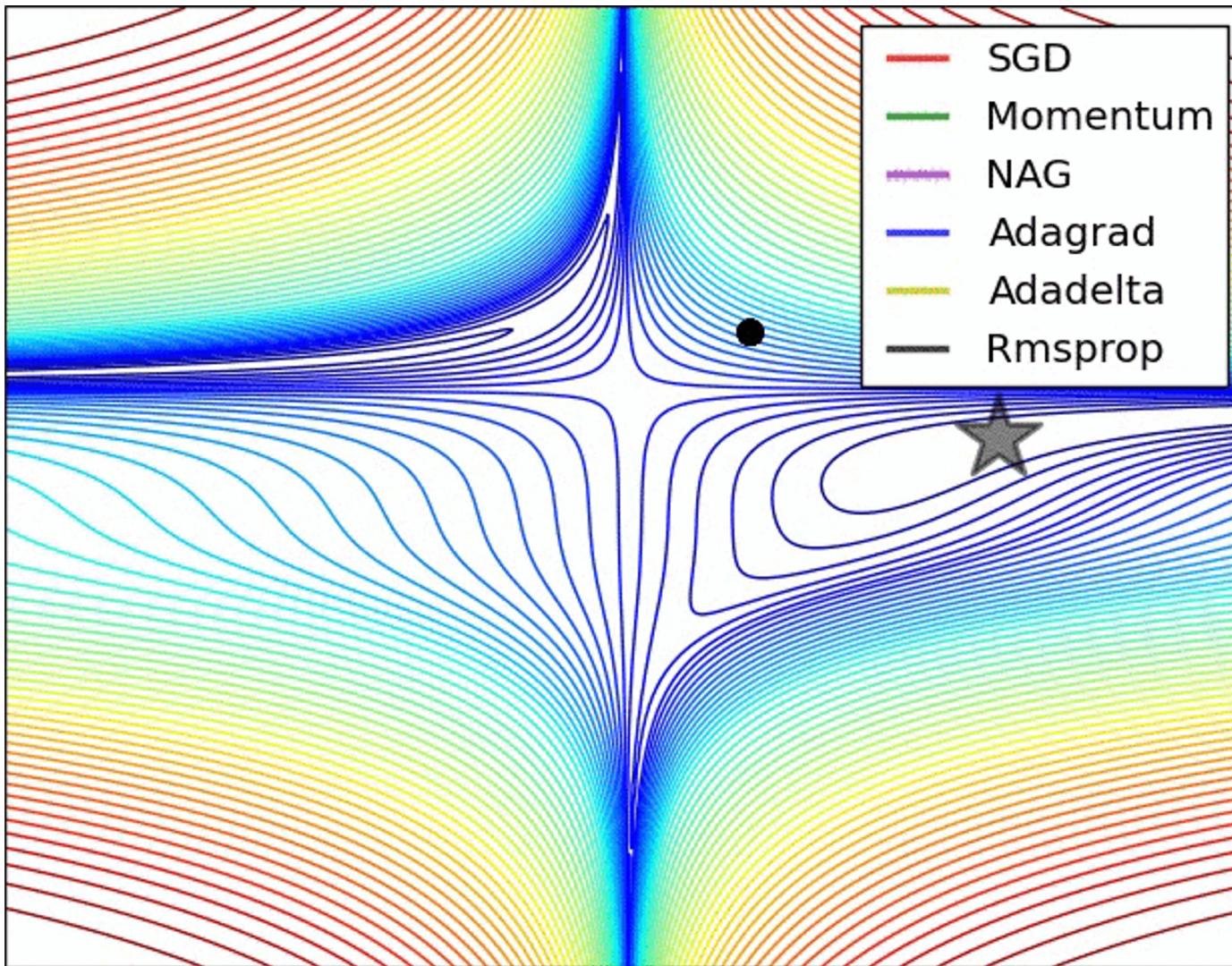
AdaGrad / RMSProp

- Hiệu chỉnh bias để thuật toán đỡ bị ảnh hưởng bởi giá trị của beta1 và beta2. Đồng thời giúp cho thuật toán ổn định hơn trong quá trình *warm up* tại một số bước đầu tiên khi cả hai moment đều khởi tạo bằng 0.
- Chứng minh chi tiết có thể tham khảo tại [Tài liệu tham khảo số 2](#) hoặc trong [bài báo gốc](#)
- Adam với beta1 = 0.9, beta2 = 0.999, và learning_rate = 1e-3 hoặc 5e-4 là tham số mặc định tốt cho nhiều mô hình!

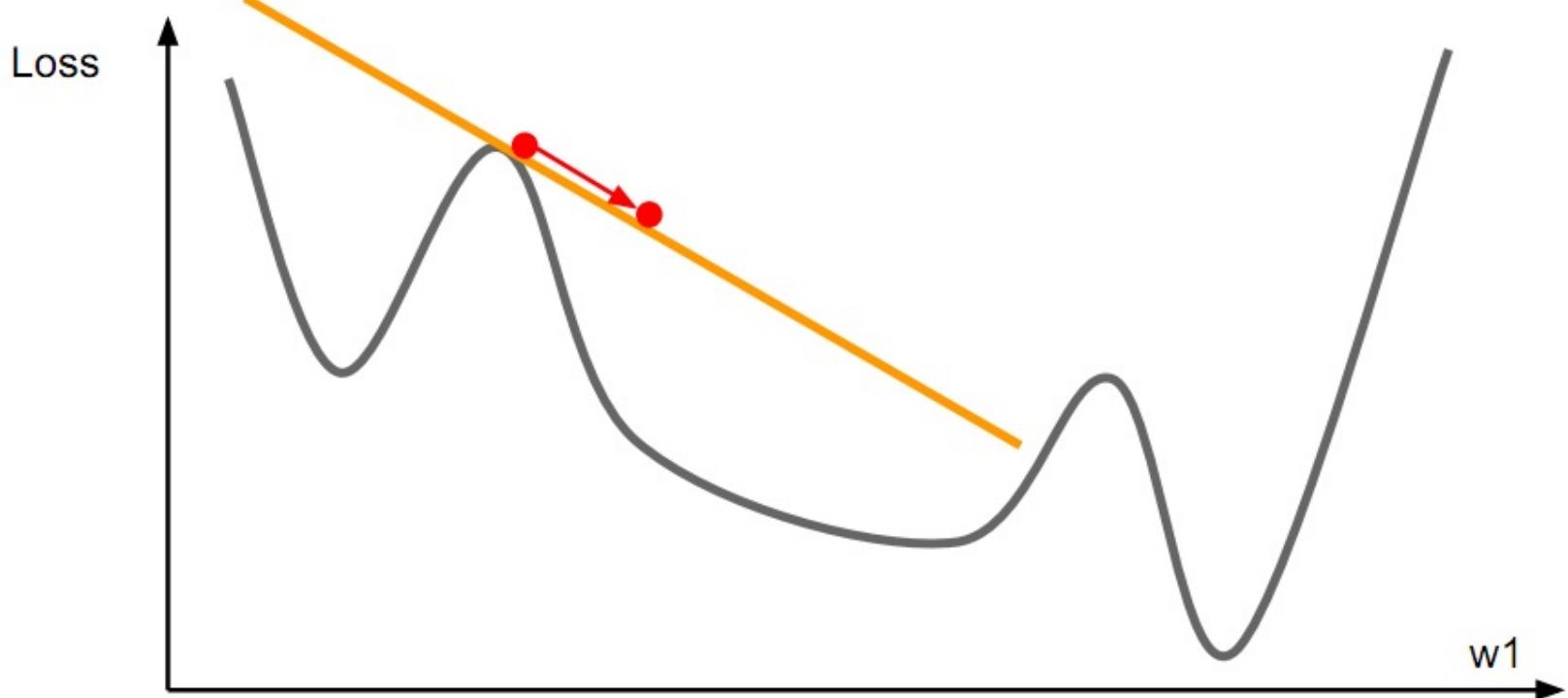
So sánh các phương pháp tối ưu



So sánh các phương pháp tối ưu

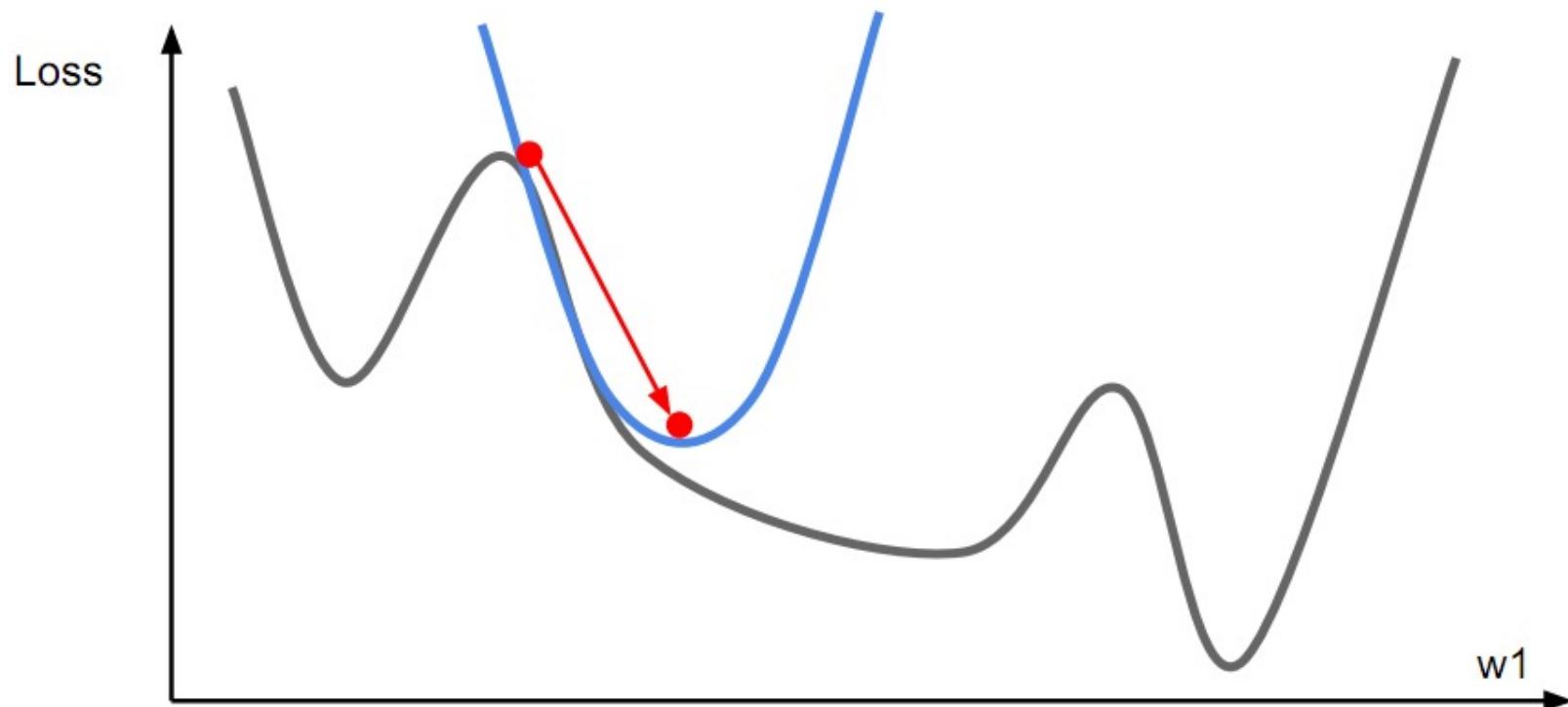


First-order optimization



Second-order optimization

- Sử dụng ma trận Hessian



Second-order optimization

- Khai triển Taylor

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

- Điểm cực tiểu:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

- Không tốt cho DL (do độ phức tạp tính nghịch đảo là $O(n^3)$)
- Quasi-Newton (BGFS)

SOTA optimizers

- NAdam = Adam + NAG
- RAdam (Rectified Adam)
- LookAhead
- Ranger = RAdam + LookAhead

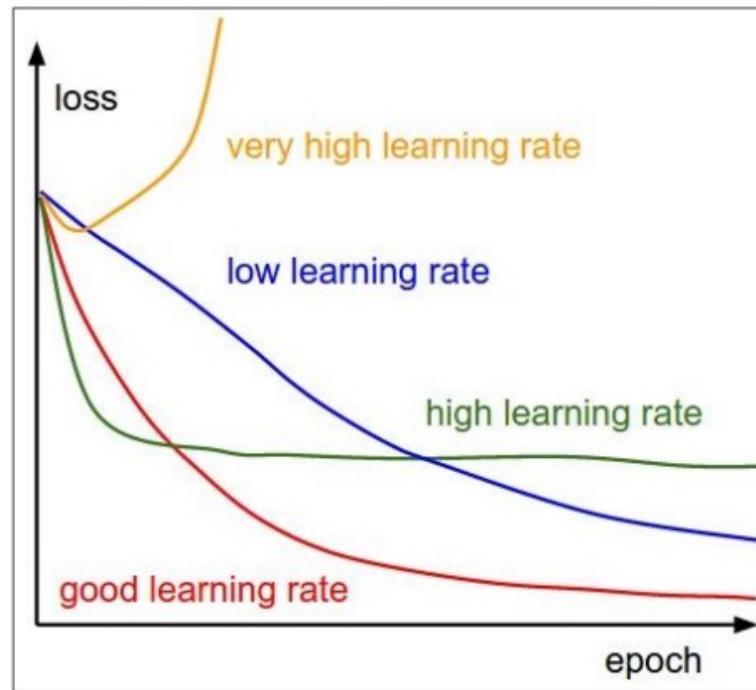
Trong thực tế

- Adam là lựa chọn mặc định tốt trong nhiều trường hợp
- SGD+Momentum thường tốt hơn Adam nhưng cần phải tinh chỉnh tốc độ học và lên chiến lược thay đổi tốc độ học hợp lý

Chiến lược thay đổi tốc độ học

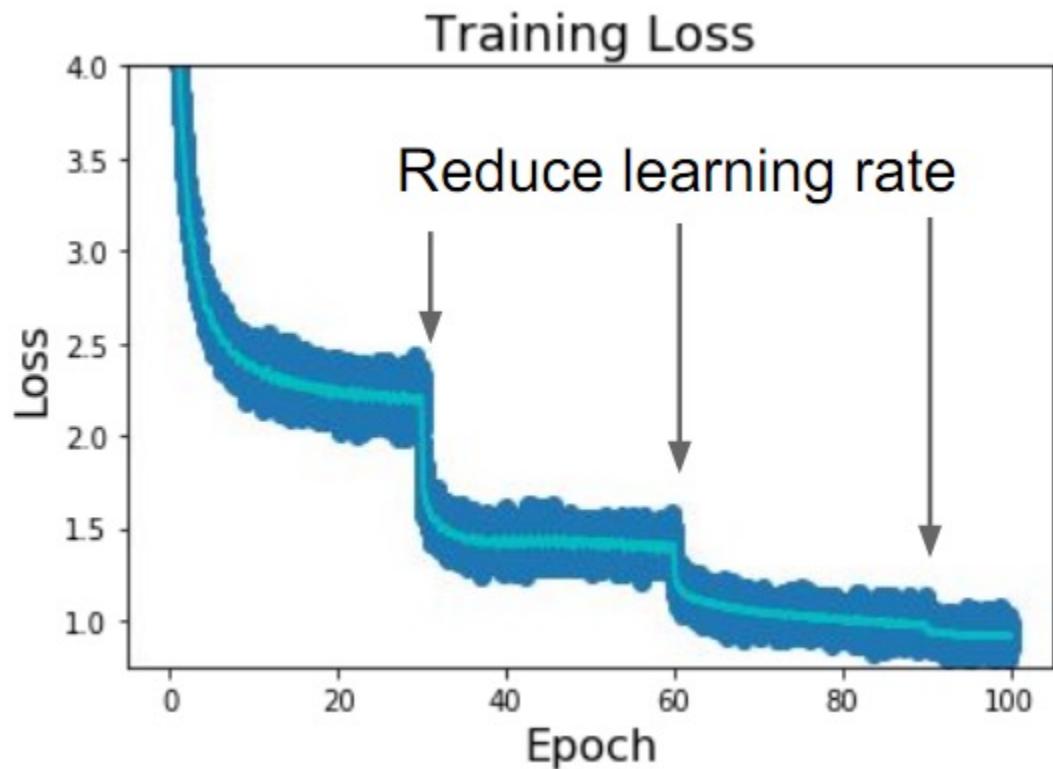
Tốc độ học

- Tốc độ học là siêu tham số (hyperparameter) của tất cả các thuật toán tối ưu SGD, SGD+Momentum, Adagrad, RMSProp, Adam...
- Thường bắt đầu với giá trị lớn và giảm dần theo thời gian



Chiến lược thay đổi tốc độ học

- Step: Thay đổi tốc độ học tại một số thời điểm cố định.
- Ví dụ: với ResNets có thể giảm lr 10 lần tại các epochs 30, 60 và 90.



Chiến lược thay đổi tốc độ học

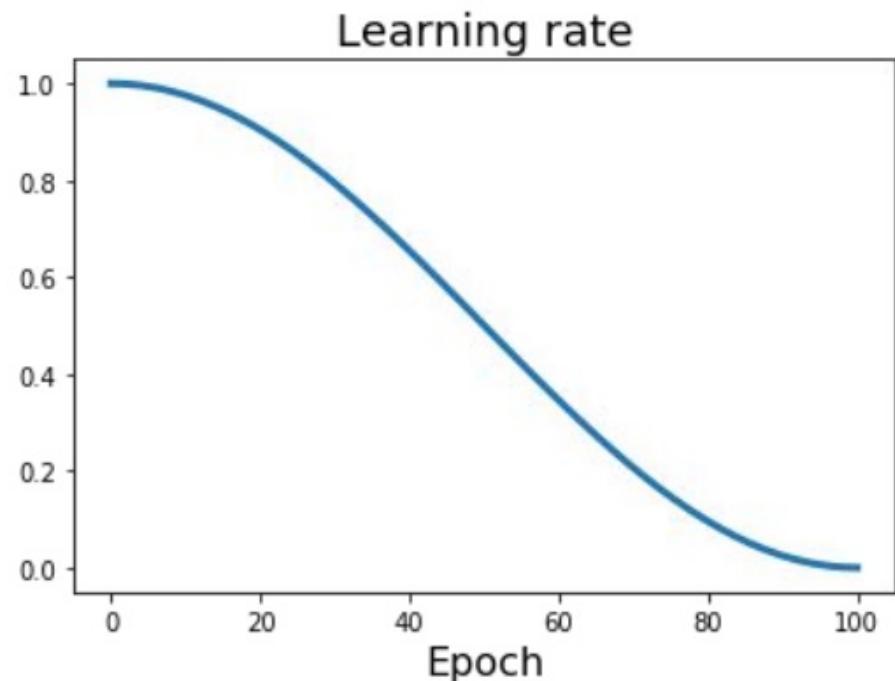
- Giảm theo cosin

$$\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$$

α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs



Chiến lược thay đổi tốc độ học

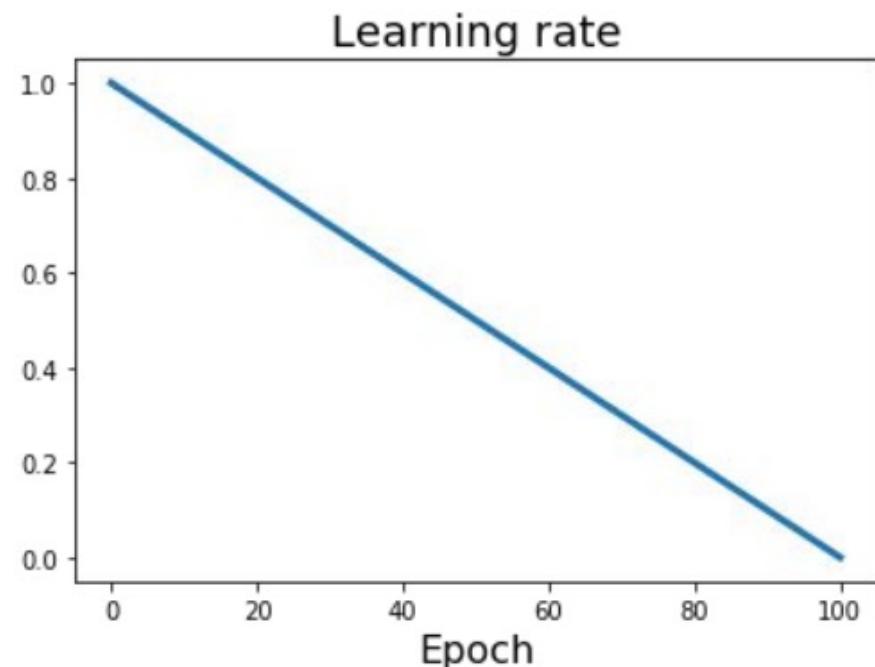
- Giảm tuyến tính

$$\alpha_t = \alpha_0(1 - t/T)$$

α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs



Chiến lược thay đổi tốc độ học

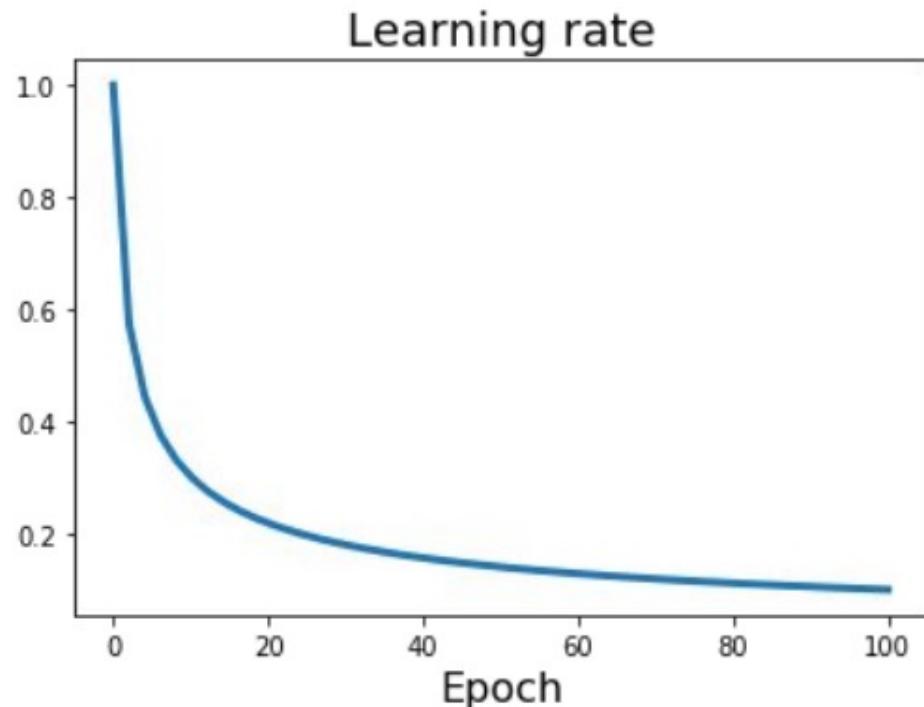
- Tỉ lệ nghịch căn bậc hai số epoch:

$$\alpha_t = \alpha_0 / \sqrt{t}$$

α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

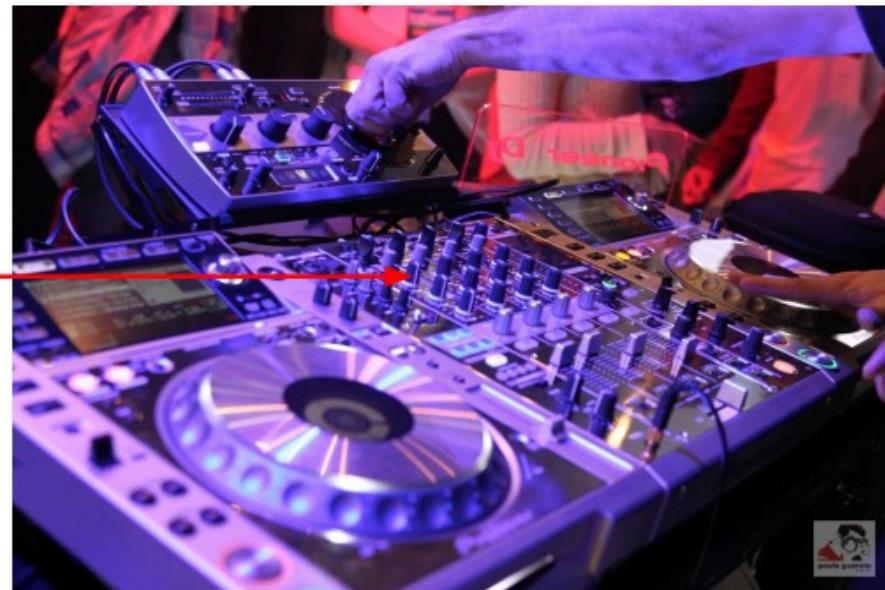


Lựa chọn siêu tham số

Siêu tham số

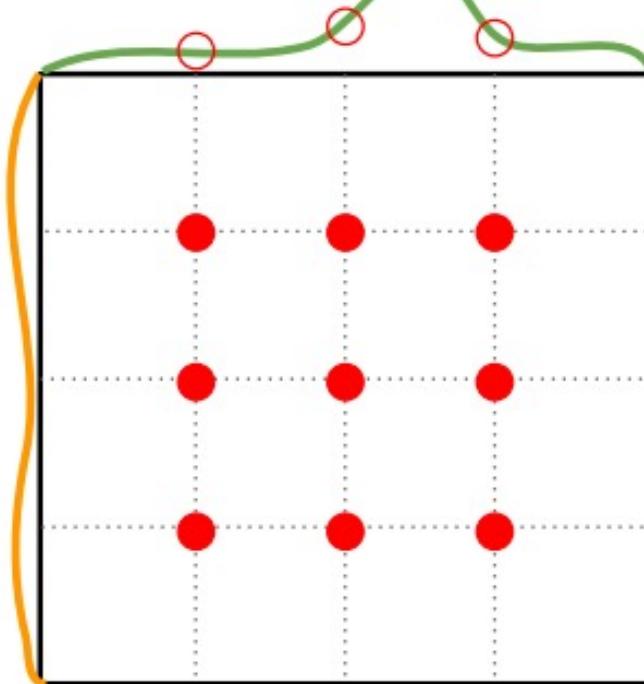
- Kiến trúc mạng
- Tốc độ học, tham số trong chiến lược thay đổi tốc độ học, thuật toán tối ưu
- Các hệ số điều khiển (L2 weight decay, drop rate)

neural networks practitioner
music = loss function



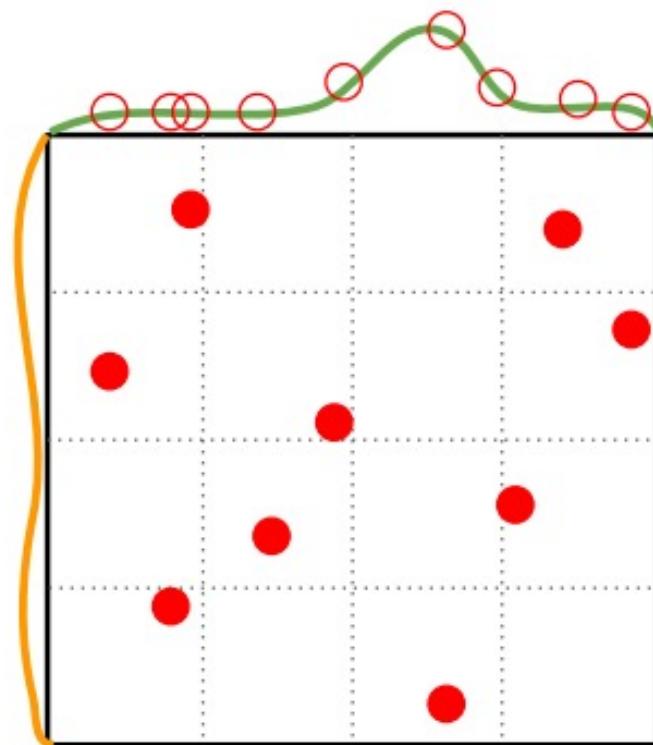
Random Search vs Grid Search

Grid Layout



Important Parameter

Random Layout



Important Parameter

Unimportant Parameter

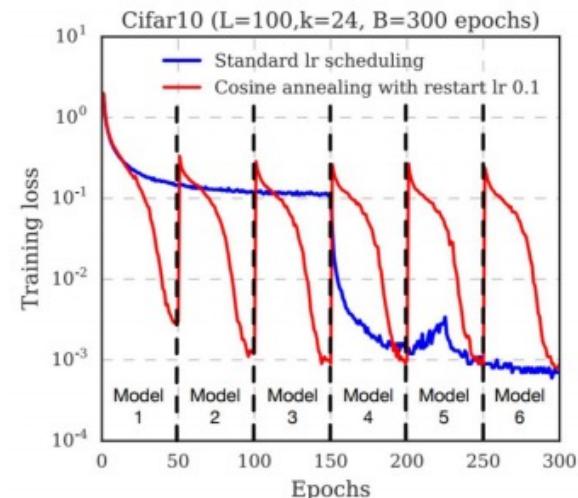
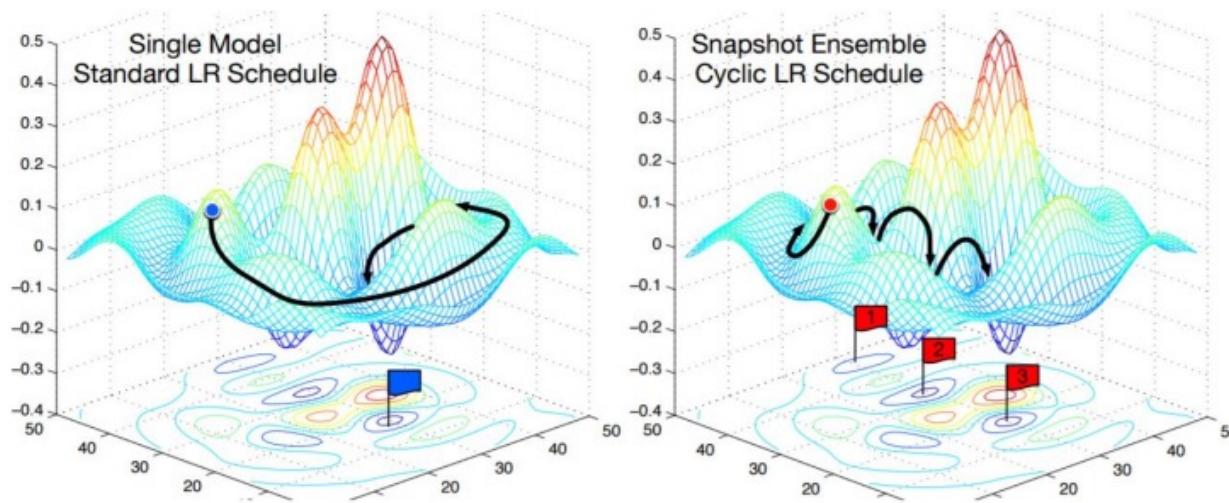
Kỹ thuật kết hợp nhiều mô hình (model ensemble)

Model Ensembles

- Huấn luyện nhiều mô hình độc lập
- Khi test kết hợp kết quả nhiều mô hình
- Độ chính xác thường tăng 2%

Model Ensembles

- Thay vì huấn luyện nhiều mô hình độc lập, có thể dùng nhiều snapshot của cùng một mô hình trong quá trình huấn luyện

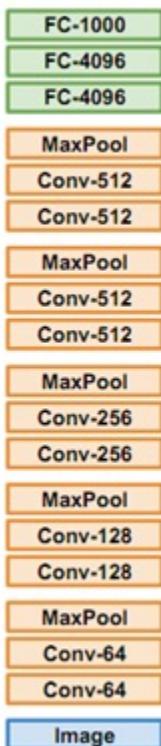


Kỹ thuật học chuyển giao (transfer learning)

Học chuyển giao

Huấn luyện mạng trên một tập dữ liệu lớn có sẵn, sau đó huấn luyện tiếp với tập dữ liệu của mình

1. Train on Imagenet



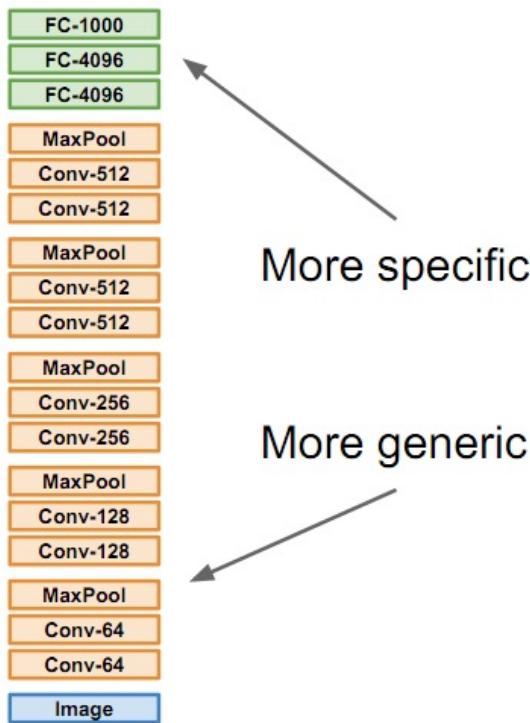
2. Small Dataset (C classes)



3. Bigger dataset



Học chuyển giao



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

More tips and tricks

- Machine Learning Yearning by Andrew Ng

<https://d2wvfoqc9gyqzf.cloudfront.net/content/uploads/2018/09/Ng-MLY01-13.pdf>

Tài liệu tham khảo

1. Bài giảng biên soạn dựa trên khóa cs231n của Stanford, bài giảng số 7-8:

<http://cs231n.stanford.edu>

2. Adam:

<https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c>

3. Stanford lecture note:

<http://cs231n.github.io/neural-networks-3/>



25
YEARS ANNIVERSARY
SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

**Thank you
for your
attentions!**

