

TRAINING NEURAL NETWORKS, PART 2

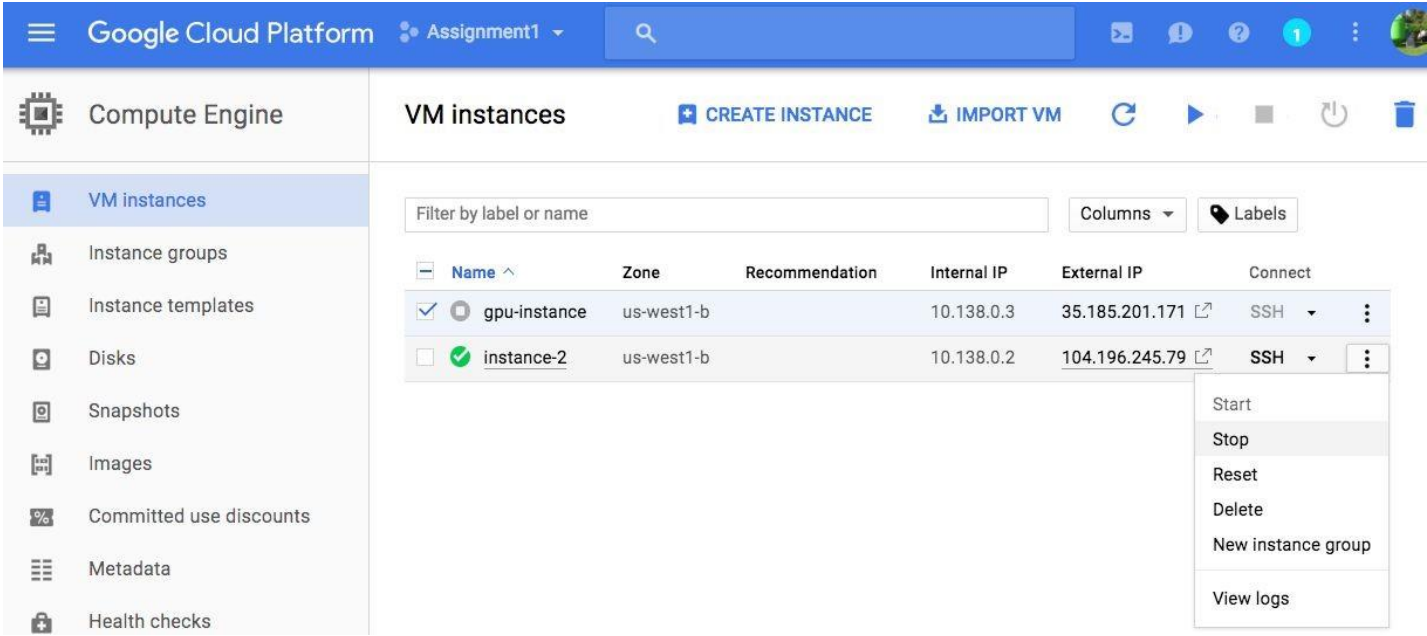
- ThS. Đoàn Chánh Thống
- ThS. Nguyễn Cường Phát
- ThS. Nguyễn Hữu Lợi
- ThS. Trương Quốc Dũng
- ThS. Nguyễn Thành Hiệp
- ThS. Võ Duy Nguyên
- ThS. Nguyễn Văn Toàn
- ThS. Lê Ngô Thục Vi
- TS. Nguyễn Duy Khánh
- TS. Nguyễn Tấn Trần Minh Khang

Administrative

- Assignment 1 is being graded, stay tuned
- Bài tập 1 đang được chấm điểm, các bạn chú ý theo dõi nhé
- Project proposals due today by 11:59pm
- Đề xuất dự án phải nộp trước 23:59 hôm nay
- Assignment 2 is out, due Thursday May 4 at 11:59pm
- Bài tập 2 đã hết, hạn nộp vào Thứ Năm ngày 4 tháng 5 lúc 11:59 tối

Administrative

- Administrative: Google Cloud.
- **STOP YOUR INSTANCES** when not in use!
- **DỪNG CÁC PHIÊN LÀM VIỆC** khi không sử dụng!



The screenshot shows the Google Cloud Platform interface for managing VM instances. The left sidebar lists various services, with 'Compute Engine' selected. Under 'Compute Engine', 'VM instances' is highlighted. The main panel shows a table of VM instances. Two instances are listed: 'gpu-instance' and 'instance-2'. The 'gpu-instance' is in the 'us-west1-b' zone, has an internal IP of 10.138.0.3, and an external IP of 35.185.201.171. The 'instance-2' is also in the 'us-west1-b' zone, has an internal IP of 10.138.0.2, and an external IP of 104.196.245.79. A context menu is open for 'instance-2', showing options: Start, Stop, Reset, Delete, New instance group, and View logs.

Name	Zone	Recommendation	Internal IP	External IP	Connect
<input checked="" type="checkbox"/> gpu-instance	us-west1-b		10.138.0.3	35.185.201.171	SSH
<input type="checkbox"/> instance-2	us-west1-b		10.138.0.2	104.196.245.79	SSH

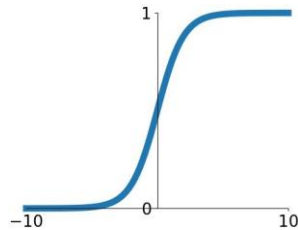
Administrative

- Keep track of your spending!
- Theo dõi chi phí của bạn!
- GPU instances are much more expensive than CPU instances - only use GPU instance when you need it (e.g. for A2 only on TensorFlow / PyTorch notebooks).
- Phiên làm việc trên GPU đắt hơn nhiều so với phiên làm việc trên CPU – chúng ta chỉ sử dụng phiên bản GPU khi cần thiết (ví dụ: chỉ dành cho A2 trên TensorFlow / PyTorch notebooks).

Last time: Activation function

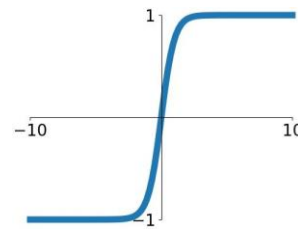
— Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



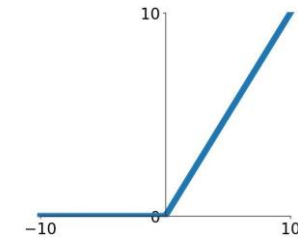
— tanh

$$\tanh(x)$$



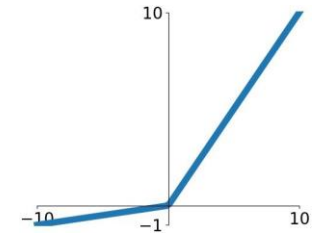
— ReLU

$$\max(0, x)$$



— Leaky ReLU

$$\max(0.001x, x)$$

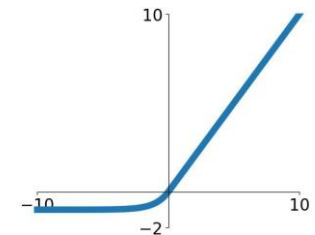


— Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

— ELU

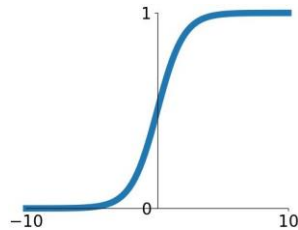
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation function

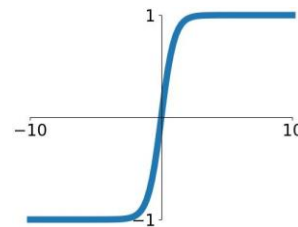
— Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



— tanh

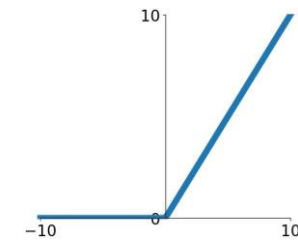
$$\tanh(x)$$



— ReLU

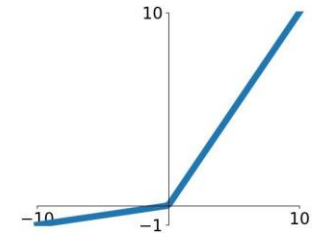
$$\max(0, x)$$

Good default choice



— Leaky ReLU

$$\max(0.001x, x)$$

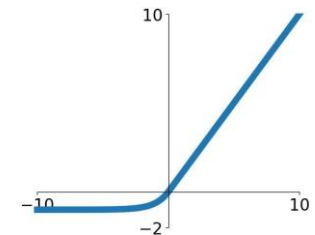


— Maxout

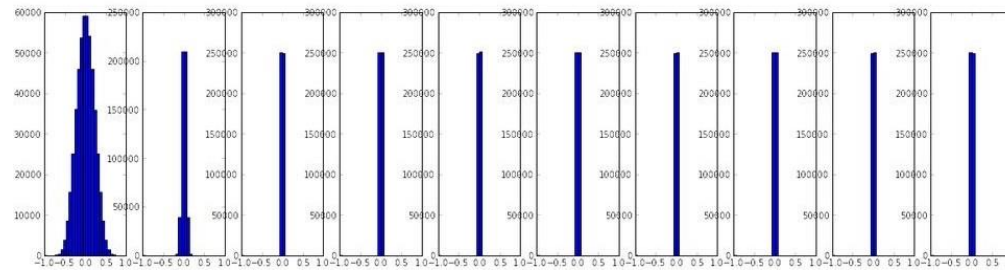
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

— ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

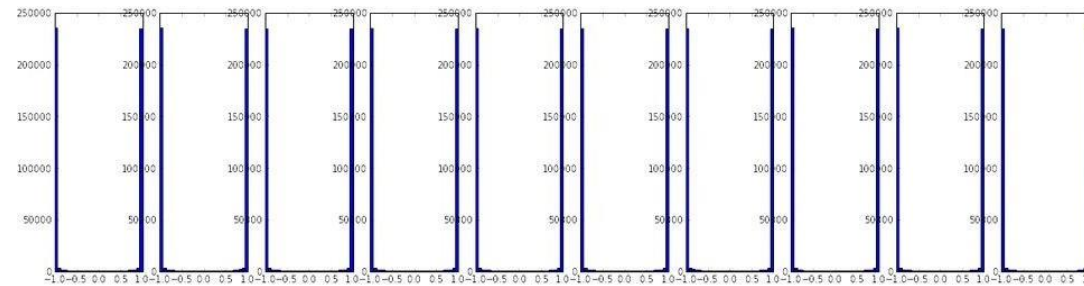


Last time: Weight Initialization



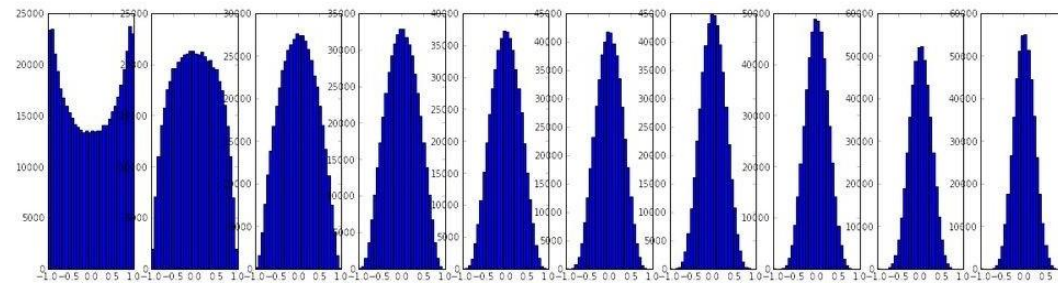
- Initialization too small: Activations go to zero, gradients also zero, No learning.
- Khởi tạo trọng số quá nhỏ: Các giá trị sau khi kích hoạt tiến đến 0, đạo hàm cũng tiến đến 0, không học được vì ma trận số không được cập nhật.

Last time: Weight Initialization



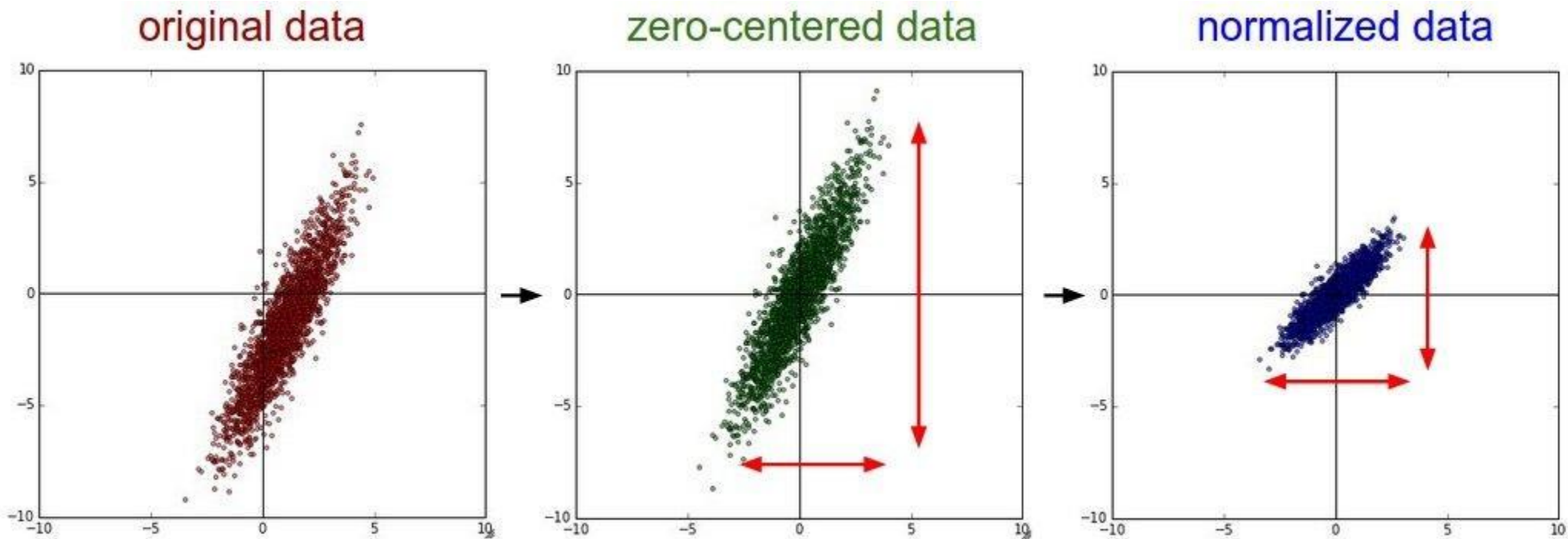
- Initialization too big: Activations saturate (for tanh), Gradients zero, no learning.
- Khởi tạo trọng số quá lớn: Các giá trị sau khi kích hoạt bão hòa (đối với hàm kích hoạt tanh), đạo hàm cũng tiến đến 0, không học được vì ma trận số không được cập nhật.

Last time: Weight Initialization



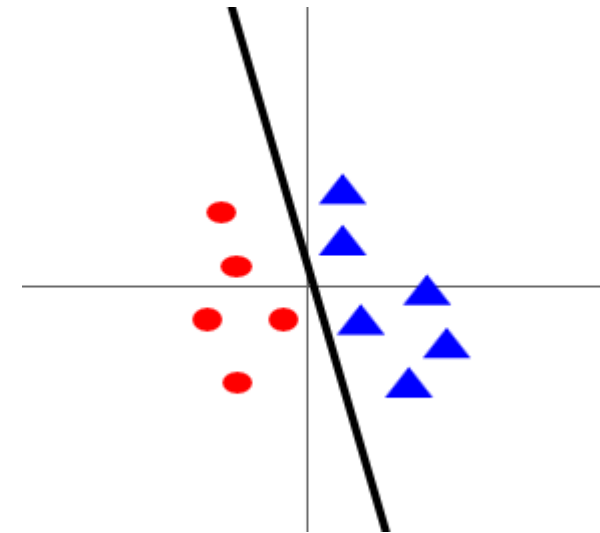
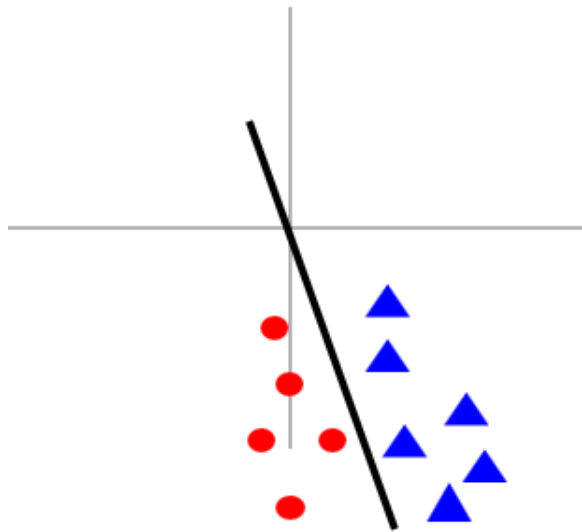
- Initialization just right: Nice distribution of activations at all layers, Learning proceeds nicely.
- Khởi tạo trọng số hợp lý: Các giá trị sau khi có phân phối tốt ở tất cả các lớp, quá trình học diễn ra tốt đẹp.

Last time: Data Preprocessing



Last time: Data Preprocessing

- Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize
- After normalization: less sensitive to small changes in weights; easier to optimize



Last time: Batch Normalization

— Input:

$$x: N \times D$$

— Learnable params:

$$\gamma, \beta : D$$

— Intermediates:

$$\mu, \sigma : D$$

$$\hat{x}: N \times D$$

— Output:

$$y: N \times D$$

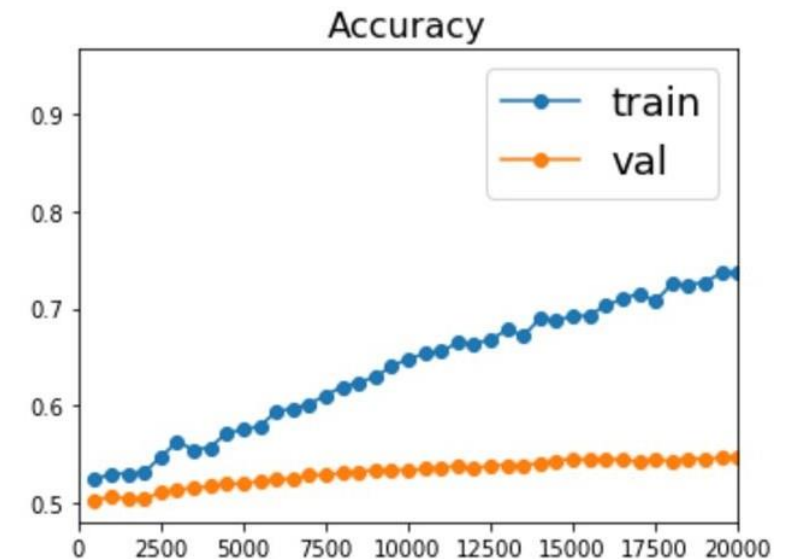
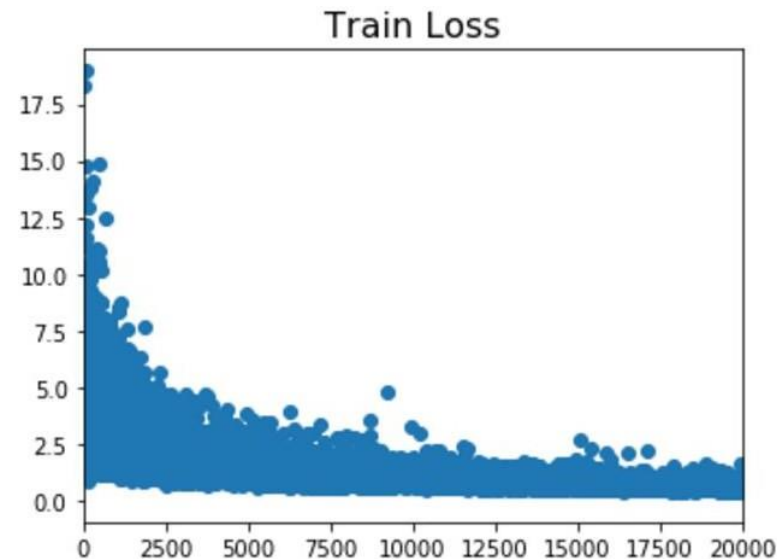
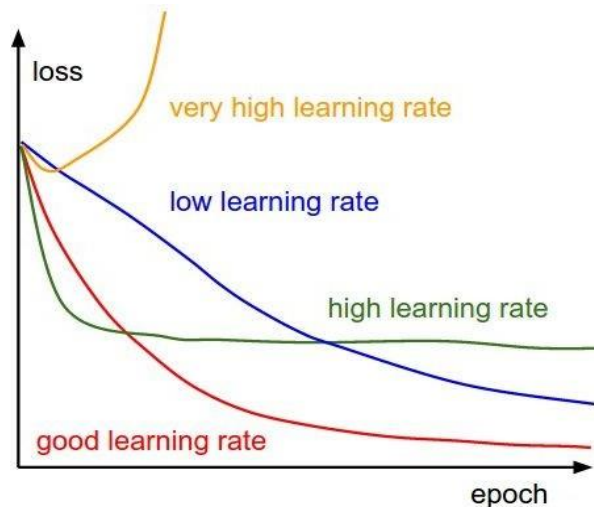
$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

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

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

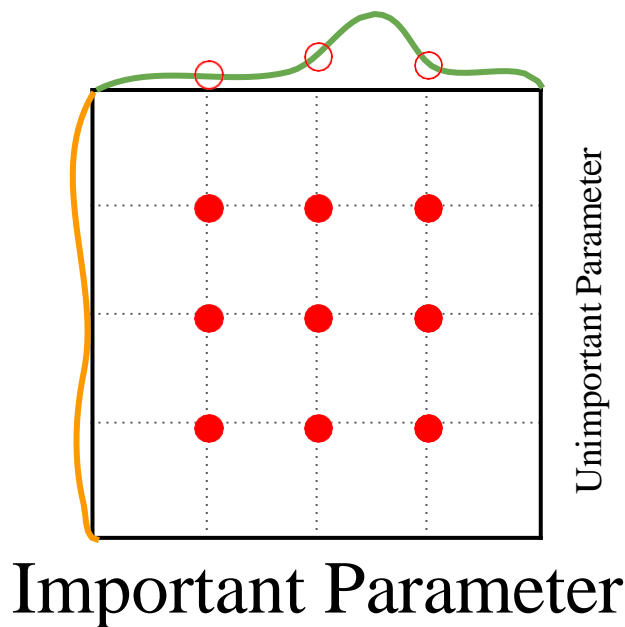
$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Last time: Babysitting Learning

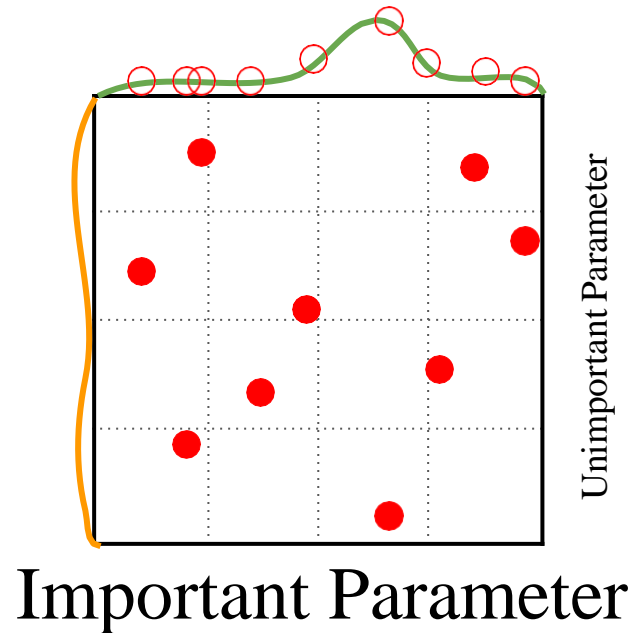


Last time: Hyperparameter Search

Grid Layout



Random Layout



Coarse to fine search

```

val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)

val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-01, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
    
```


Today

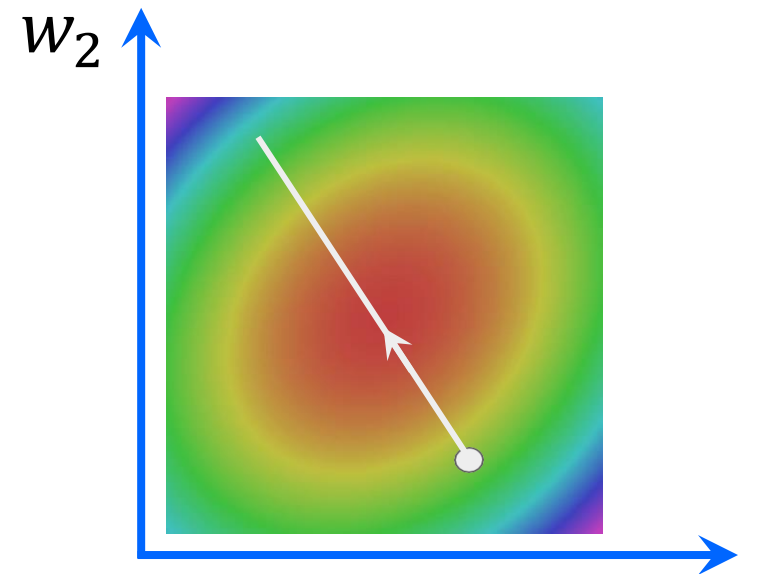
- Fancier optimization.
- Tối ưu hóa nâng cao.
- Regularization.
- Kỹ thuật chính quy hóa.
- Transfer Learning.
- Học chuyển giao – Học chuyển tiếp.

Tối ưu hóa nâng cao

FANCIER OPTIMIZATION

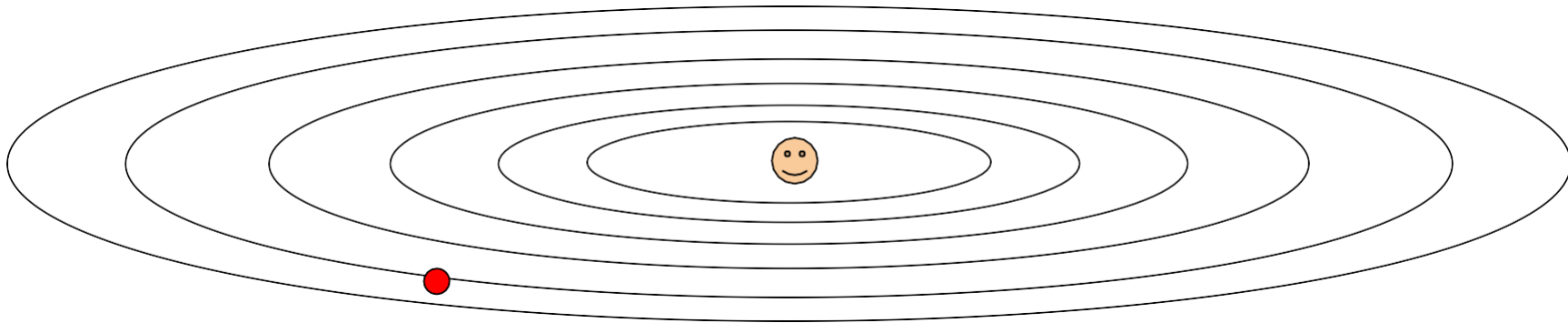
Optimization

```
1. # Vanilla Gradient Descent
2. while True:
3.     w_grad=evaluate_gradient(loss_fun,data,weights)
4.     weights += -step_size * w_grad
5.     # perform parameter update
```



Optimization: Problems with SGD

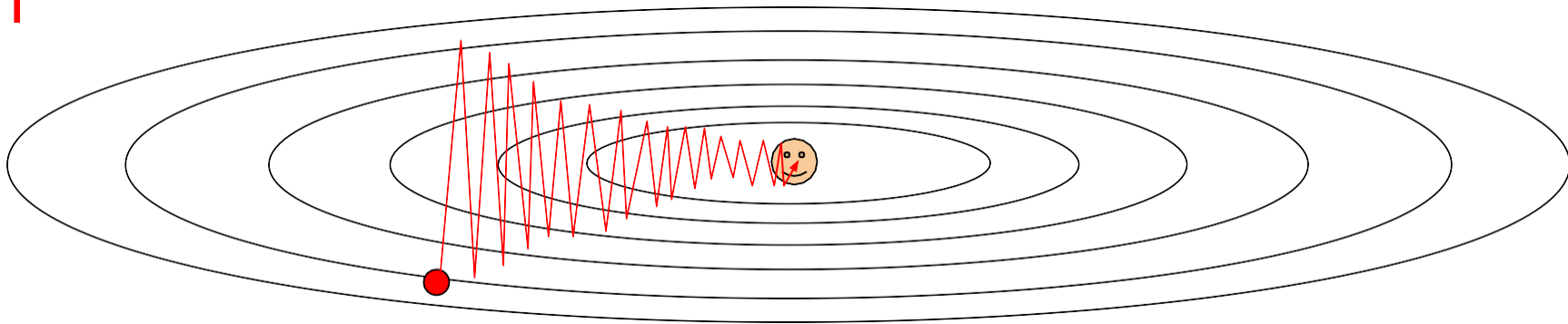
- What if loss changes quickly in one direction and slowly in another? What does gradient descent do?



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

Optimization: Problems with SGD

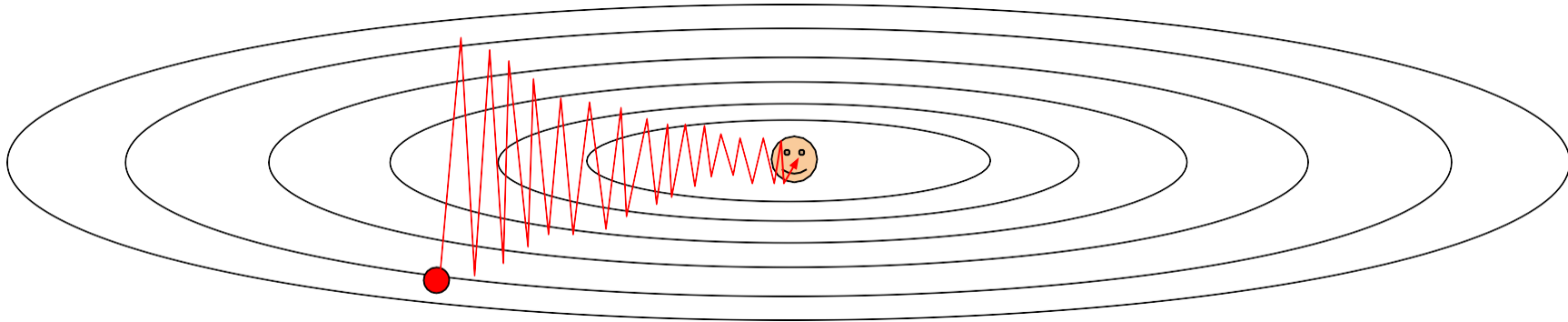
- What if loss changes quickly in one direction and slowly in another? What does gradient descent do?
- Very slow progress along shallow dimension, jitter along steep direction



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

Optimization: Problems with SGD

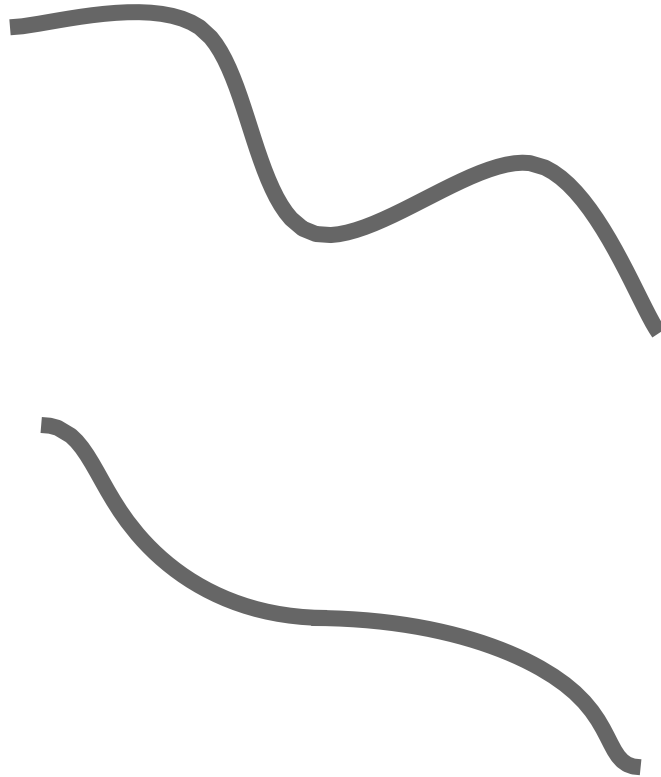
- Điều gì sẽ xảy ra nếu hàm mất mát thay đổi nhanh chóng theo hướng này và chậm rãi theo hướng khác? Giảm độ dốc làm gì?
- Tiến độ rất chậm dọc theo chiều nông, jitter dọc theo hướng dốc



- Hàm mất mát có số điều kiện cao: tỷ số giữa giá trị kỳ dị lớn nhất và nhỏ nhất của ma trận Hessian lớn.

Optimization: Problems with SGD

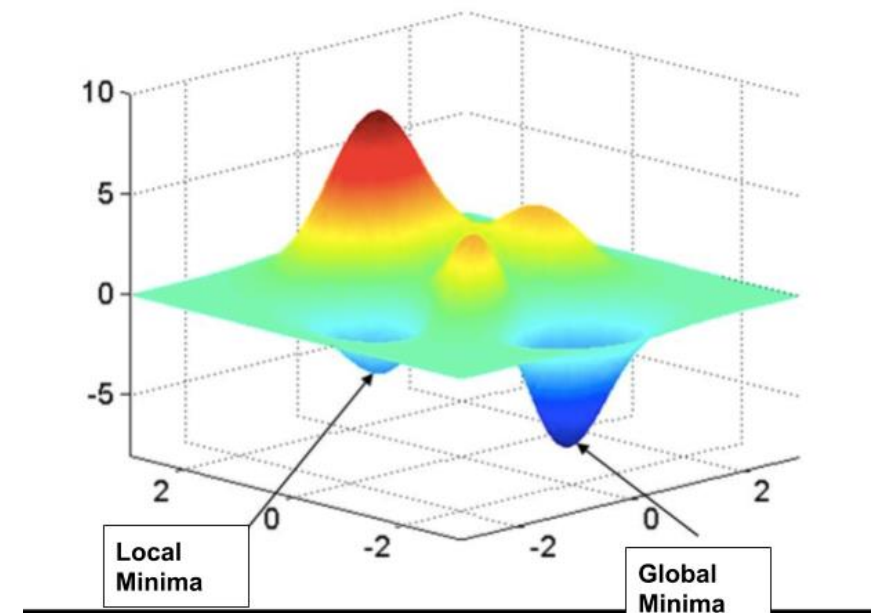
- What if the loss function has a local minima or saddle point?



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

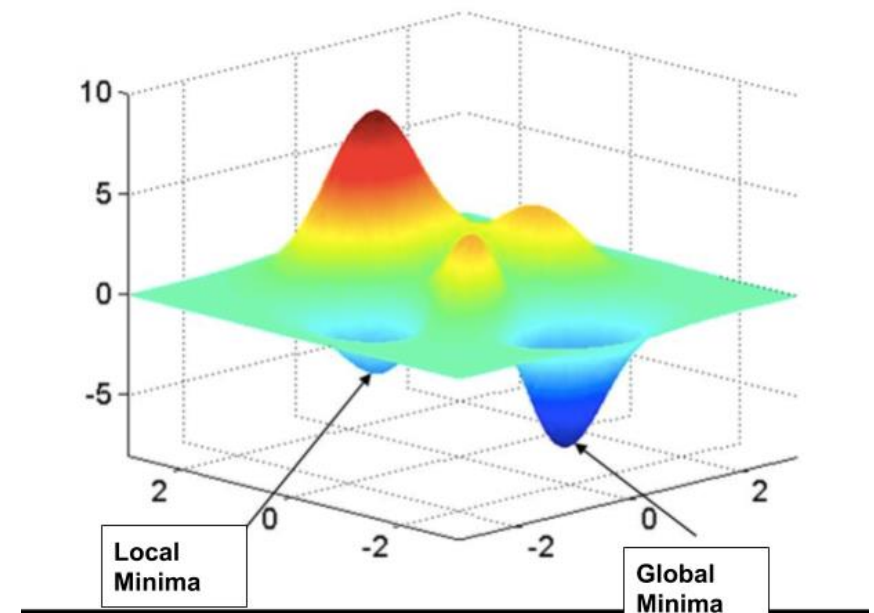
Local minima

- Local minima là một khái niệm quan trọng trong tối ưu hóa và học máy.
- Local minima là một điểm trong không gian các tham số của một hàm số mà giá trị của hàm tại đó nhỏ hơn hoặc bằng giá trị của hàm tại tất cả các điểm lân cận.



Local minima

- Định nghĩa: Local minima là một điểm trong miền xác định của hàm số mà giá trị của hàm tại đó nhỏ hơn hoặc bằng giá trị của hàm tại các điểm lân cận trong một khoảng cách nhất định.
- Nếu $f(x)$ là hàm số cần tối ưu hóa, thì $x = a$ là local minima nếu tồn tại một khoảng lân cận $\delta > 0$ sao cho $f(a) \leq f(x)$ với mọi x trong khoảng lân cận đó ($|x - a| < \delta$).



Local minima

— Ý nghĩa trong học máy:

- + Trong học máy, các mô hình thường được huấn luyện bằng cách tối ưu hóa một hàm mất mát.
- + Việc tìm kiếm giá trị nhỏ nhất của hàm mất mát giúp mô hình có thể dự đoán chính xác hơn.
- + Tuy nhiên, hàm mất mát có thể có nhiều điểm cực tiểu cục bộ (local minima) và một điểm cực tiểu toàn cục (global minimum).
- + Mục tiêu là tìm điểm cực tiểu toàn cục, nhưng trong thực tế, các thuật toán tối ưu hóa thường bị kẹt ở các điểm cực tiểu cục bộ.

Local minima

— Thách thức:

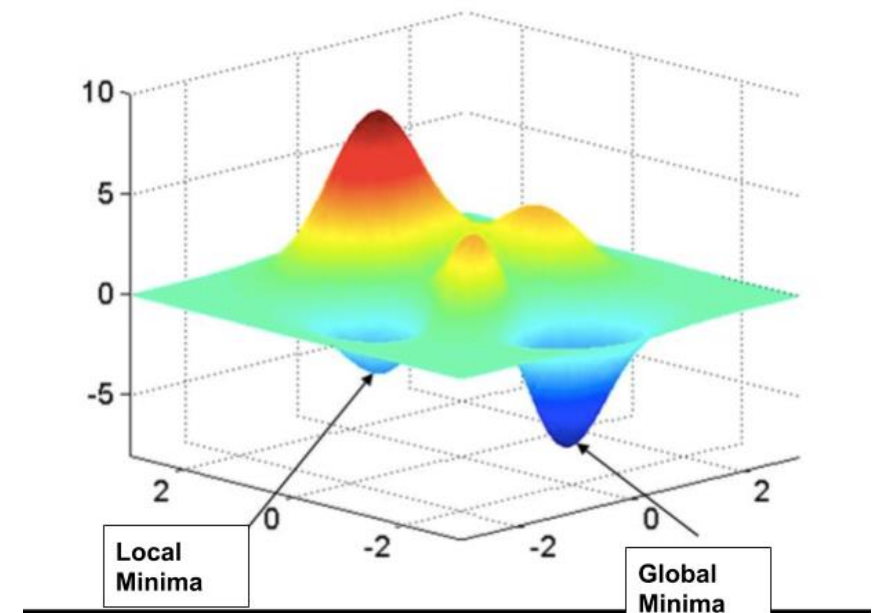
- + Việc gặp phải local minima có thể làm cho mô hình không đạt được hiệu suất tối ưu.
- + Đây là một thách thức lớn trong việc huấn luyện mô hình học sâu (deep learning), vì các hàm mất mát trong các mô hình này thường rất phức tạp và có nhiều local minima.

Local minima

- Giải pháp: Có nhiều kỹ thuật được phát triển để giúp tránh hoặc thoát khỏi local minima, chẳng hạn như:
 - + Khởi tạo ngẫu nhiên: Thử nhiều khởi tạo ngẫu nhiên khác nhau để tăng khả năng tìm thấy global minimum.
 - + Thuật toán tối ưu hóa nâng cao: Sử dụng các thuật toán như Adam, RMSprop, hoặc các biến thể của gradient descent có thể giúp vượt qua các local minima.
 - + Thêm nhiễu: Thêm nhiễu vào quá trình huấn luyện hoặc sử dụng các kỹ thuật như simulated annealing.

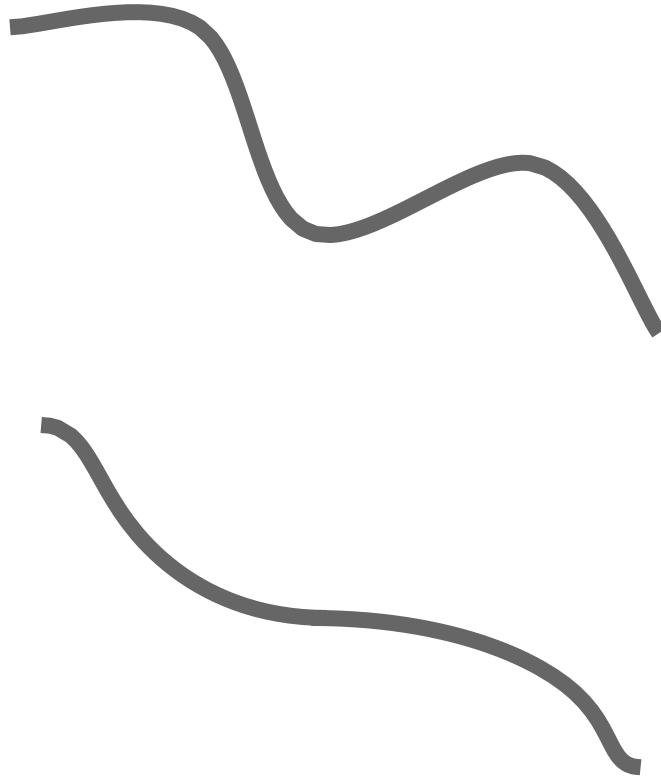
Local minima

- Tóm lại, local minima là một khái niệm quan trọng trong việc tối ưu hóa các mô hình học máy và việc hiểu rõ về nó giúp cải thiện hiệu suất của các thuật toán học máy.



Optimization: Problems with SGD

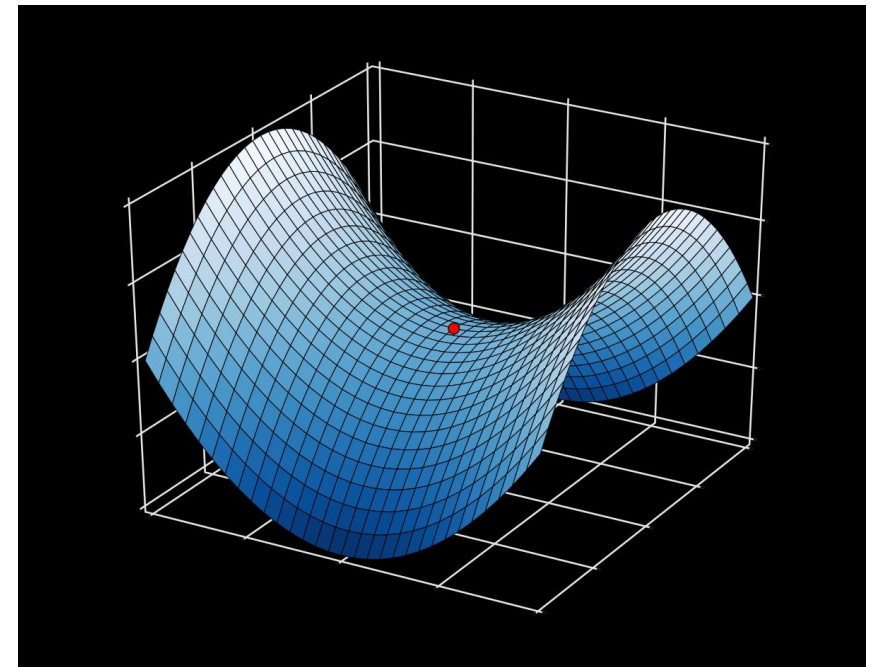
- What if the loss function has a local minima or saddle point?



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

Saddle point

- Saddle point (điểm yên ngựa) là một khái niệm trong toán học và tối ưu hóa, đặc biệt liên quan đến các hàm nhiều biến.
- Saddle point (điểm yên ngựa) là một điểm trên đồ thị của hàm số mà tại đó đạo hàm của hàm theo một số hướng là cực tiểu (local minimum) và đạo hàm theo các hướng khác là cực đại (local maximum).



Saddle point

— Đặc điểm của điểm yên ngựa:

- + Tại điểm yên ngựa, hàm số có giá trị nhỏ hơn giá trị của nó tại các điểm lân cận theo một số hướng và có giá trị lớn hơn giá trị của nó tại các điểm lân cận theo các hướng khác.
- + Ví dụ, trong không gian ba chiều, đồ thị của hàm số $f(x, y) = x^2 - y^2$ có một điểm yên ngựa tại $(0,0)$. Tại điểm này, $f(x, y)$ có giá trị cực tiểu theo hướng x và cực đại theo hướng y .

Saddle point

— Vai trò trong tối ưu hóa:

- + Trong tối ưu hóa, các thuật toán thường gặp khó khăn tại các điểm yên ngựa vì gradient (đạo hàm) tại những điểm này bằng không nhưng không phải là cực tiểu hay cực đại.
- + Các điểm yên ngựa có thể làm cho quá trình hội tụ của các thuật toán tối ưu hóa trở nên chậm chạp hoặc bị kẹt.

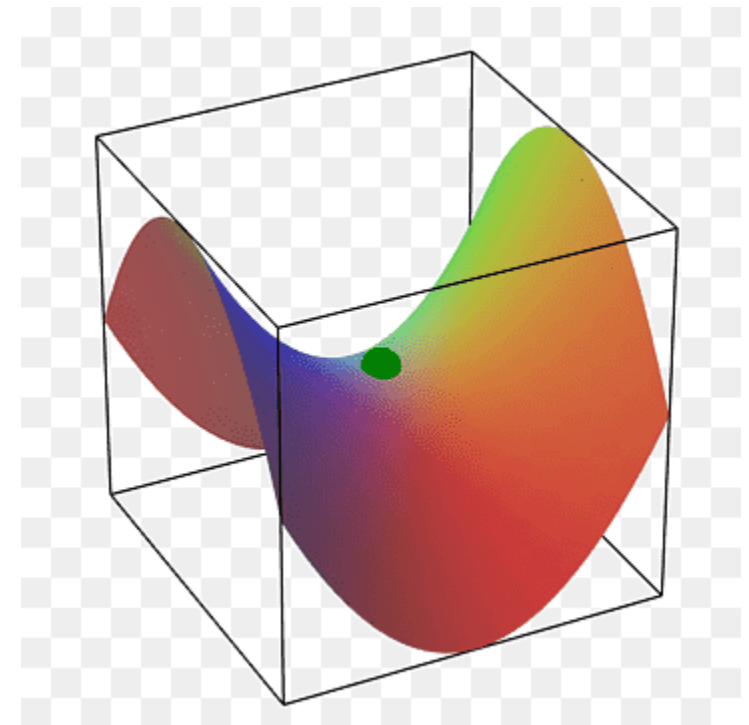
Saddle point

— Ứng dụng trong học máy:

- + Các điểm yên ngựa thường xuất hiện trong các hàm mất mát của các mô hình học sâu (deep learning). Do đó, việc phát hiện và vượt qua các điểm yên ngựa là một thách thức quan trọng.
- + Các thuật toán tối ưu hóa tiên tiến như Adam, RMSprop và những biến thể của gradient descent được thiết kế để giúp vượt qua các điểm yên ngựa.

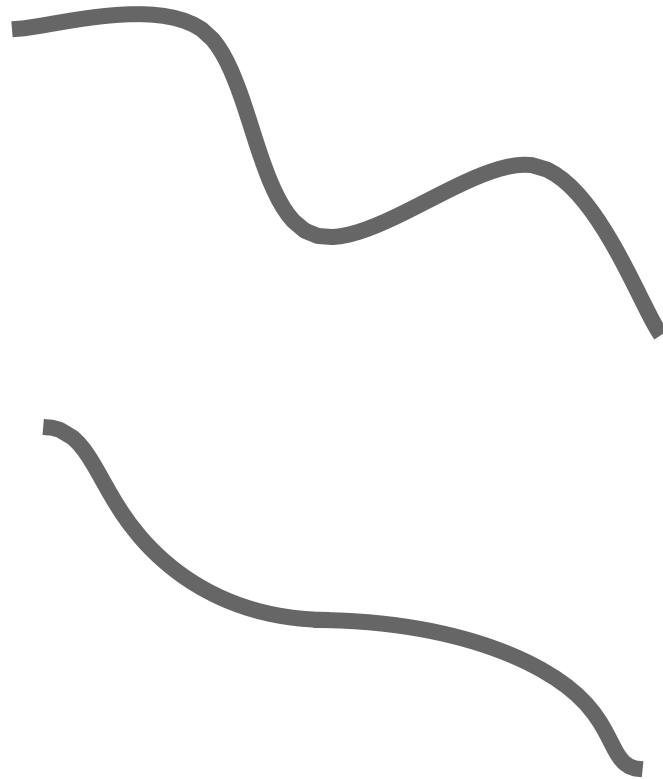
Saddle point

- Tóm lại, điểm yên ngựa là một khái niệm quan trọng trong toán học và tối ưu hóa, đặc biệt trong bối cảnh học máy và học sâu, vì nó ảnh hưởng đến hiệu quả và khả năng hội tụ của các thuật toán tối ưu hóa.



Optimization: Problems with SGD

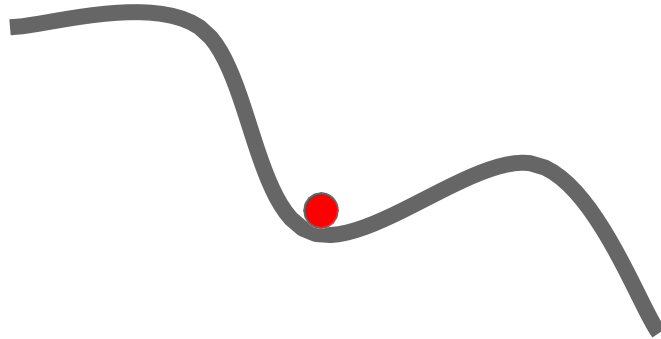
- What if the loss function has a local minima or saddle point?



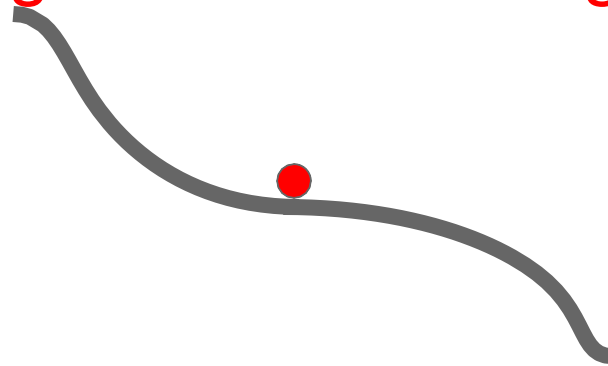
Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

Optimization: Problems with SGD

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



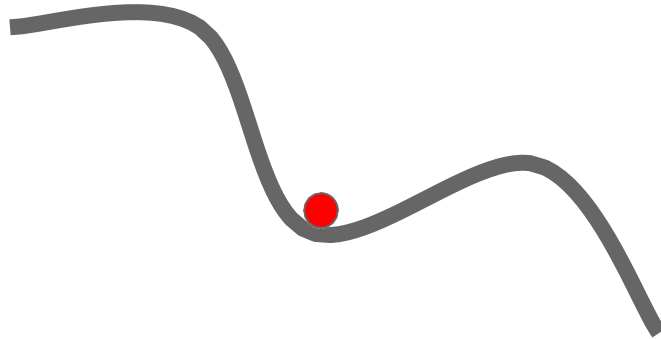
- Zero gradient, gradient descent gets stuck



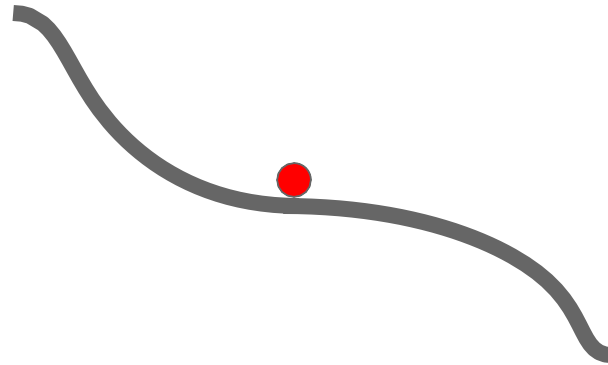
Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

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

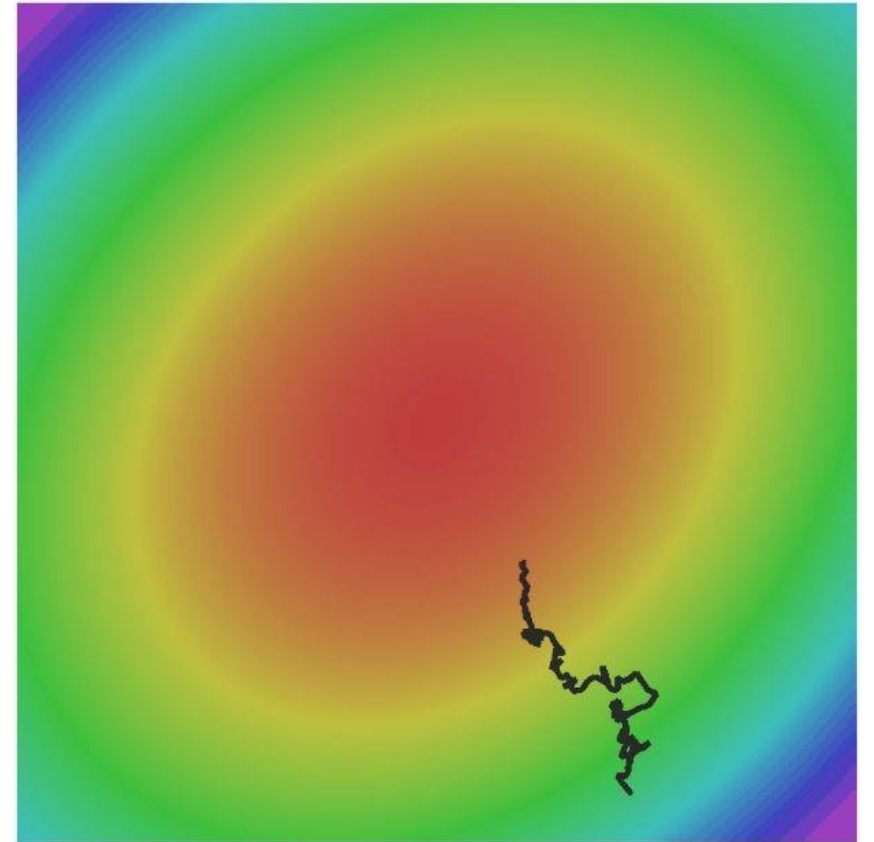
Optimization: Problems with SGD

- Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=0}^{N-1} L^{(i)}(x^{(i)}, y^{(i)}, W)$$

- Want:

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=0}^{N-1} \nabla_W L^{(i)}(x^{(i)}, y^{(i)}, W)$$



Momentum

- Trong học sâu (deep learning), "momentum" là một kỹ thuật tối ưu hóa được sử dụng để tăng tốc độ hội tụ của các thuật toán học, đặc biệt là khi làm việc với các gradient.
- Kỹ thuật momentum giúp tránh tình trạng kẹt trong các cực tiểu địa phương và cải thiện hiệu quả của quá trình huấn luyện mô hình.

Momentum

- Cách hoạt động của Momentum trong học sâu:
 - + Momentum được thêm vào quá trình cập nhật các trọng số của mô hình.
 - + Thay vì chỉ sử dụng gradient hiện tại để cập nhật trọng số, momentum còn sử dụng thông tin từ các gradient trước đó để điều chỉnh hướng cập nhật. Điều này giúp cập nhật trọng số mượt mà hơn và ít dao động hơn.

Momentum

- Công thức cập nhật với Momentum:
 - + Giả sử θ là các trọng số của mô hình,
 - + η là learning rate (tốc độ học),
 - + $\nabla J(\theta)$ là đạo hàm của hàm mất mát J theo trọng số θ .
- Momentum v được cập nhật và sử dụng như sau:

Momentum

— Momentum v được cập nhật và sử dụng như sau:

+ Cập nhật giá trị của momentum:

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta_t)$$

+ Cập nhật trọng số:

$$\theta_{t+1} = \theta_t - v_t$$

— Trong đó:

+ v_t : Là giá trị của momentum tại thời điểm t .

+ γ : Là hệ số momentum (thường có giá trị từ 0 đến 1).

Momentum

— Lợi ích của momentum:

- + **Tăng tốc độ hội tụ:** Momentum giúp tăng tốc độ hội tụ của thuật toán, đặc biệt là trong các thung lũng sâu và hẹp của không gian hàm mất mát.
- + **Giảm dao động:** Momentum giúp giảm dao động khi gradient thay đổi nhanh chóng giữa các epoch, đặc biệt là khi làm việc với các gradient nhỏ và không đồng đều.
- + **Vượt qua các cực tiểu địa phương:** Momentum giúp mô hình vượt qua các cực tiểu địa phương bằng cách duy trì hướng di chuyển dựa trên gradient trước đó.

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

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

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

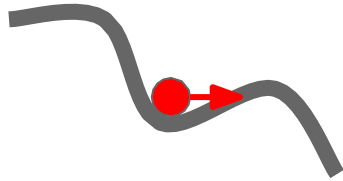
```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x += learning_rate * vx
```

Build up “velocity” as a running mean of gradients

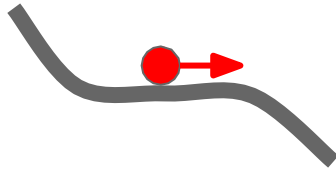
Rho gives “friction”; typically rho=0.9 or 0.99

SGD + Momentum

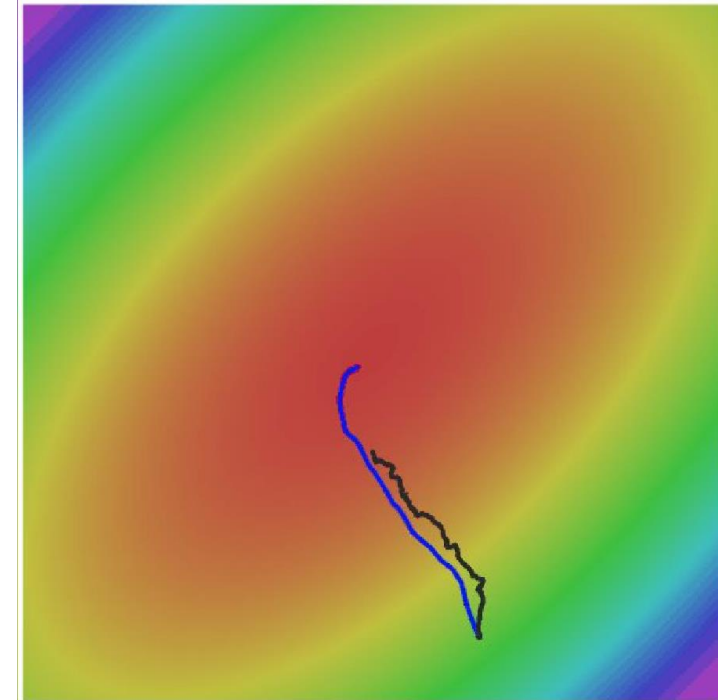
Local Minima



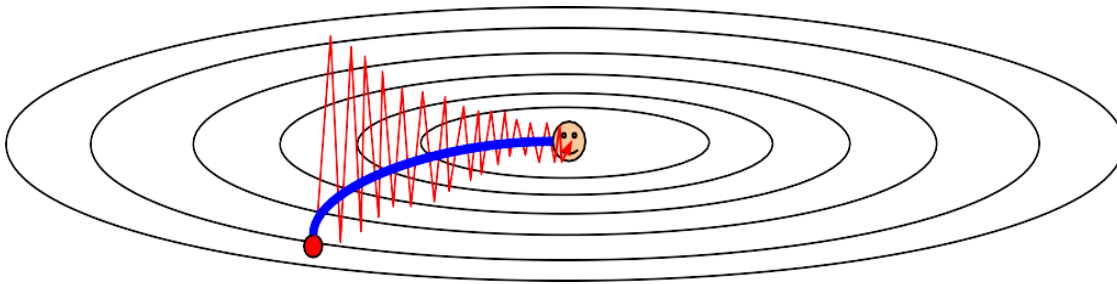
Saddle points



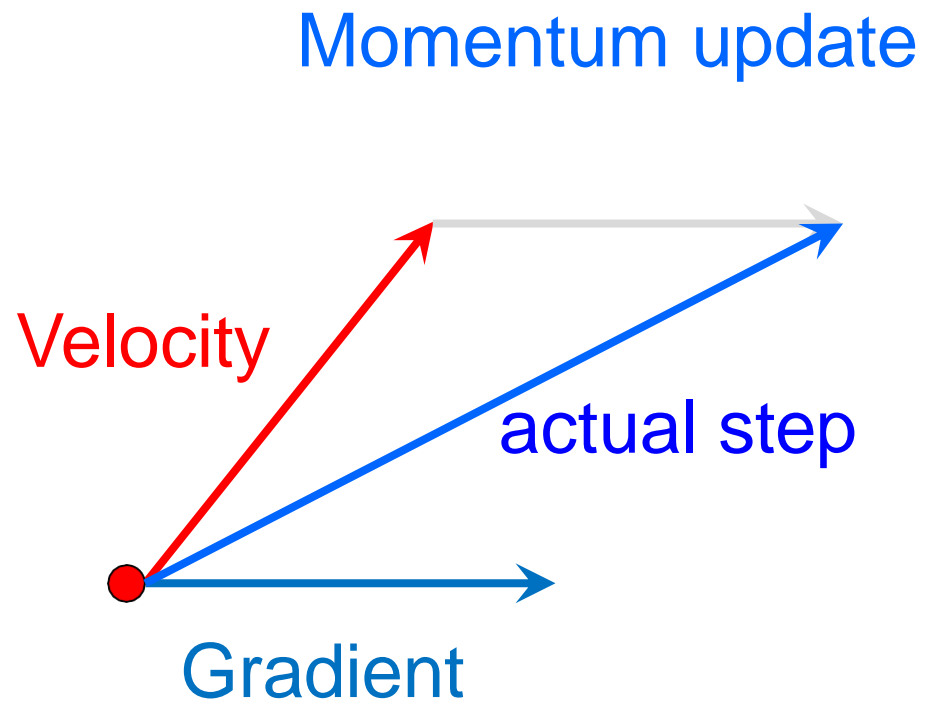
Gradient Noise



Poor Conditioning



SGD + Momentum



Nesterov Momentum

- Nesterov Momentum là một phương pháp tối ưu hóa được sử dụng để tăng tốc quá trình hội tụ của các thuật toán học máy, đặc biệt trong việc huấn luyện mạng nơ-ron sâu.
- Nesterov Momentum là một cải tiến của phương pháp Momentum thông thường, giúp tăng tốc độ hội tụ và ổn định quá trình huấn luyện.

Nesterov Momentum

— Ý tưởng cơ bản

- + Trong phương pháp Momentum thông thường, hướng đi của đạo hàm được điều chỉnh bằng cách thêm một thành phần động lực vào, giúp mô hình vượt qua các khu vực có độ dốc nhỏ và hội tụ nhanh hơn. Tuy nhiên, Momentum có thể dẫn đến việc mô hình vượt quá điểm cực tiểu.
- + Nesterov Momentum (NAG) khắc phục vấn đề mô hình vượt quá điểm cực tiểu bằng cách ước lượng đạo hàm tại vị trí mà ta sẽ đến, thay vì đạo hàm tại vị trí hiện tại. Điều này giúp thuật toán tránh việc vượt quá điểm cực tiểu và hội tụ ổn định hơn.

Nesterov Momentum

— Thuật toán Nesterov Momentum:

+ Cập nhật tốc độ (velocity): $v_t = \beta v_{t-1} + \alpha \nabla \mathcal{L}(\theta_{t-1} - \beta v_{t-1})$

+ Cập nhật các tham số mô hình: $\theta_t = \theta_{t-1} - v_t$

— Trong đó:

+ θ : là các tham số của mô hình.

+ $L(\theta)$: là hàm mất mát.

+ β : là hệ số động lực (momentum coefficient).

+ α : là tốc độ học (learning rate).

+ $\nabla \mathcal{L}$: là đạo hàm (gradient) của hàm mất mát.

Nesterov Momentum

— So sánh với Momentum thông thường

+ **Momentum thông thường:** Cập nhật các tham số dựa trên gradient tại vị trí hiện tại.

$$v_t = \beta v_{t-1} + \alpha \nabla \mathcal{L}(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} - v_t$$

+ **Nesterov Momentum:** Cập nhật các tham số dựa trên gradient tại vị trí ước lượng trước.

$$v_t = \beta v_{t-1} + \alpha \nabla \mathcal{L}(\theta_{t-1} - \beta v_{t-1})$$

$$\theta_t = \theta_{t-1} - v_t$$

Nesterov Momentum

— Lợi ích của Nesterov Momentum

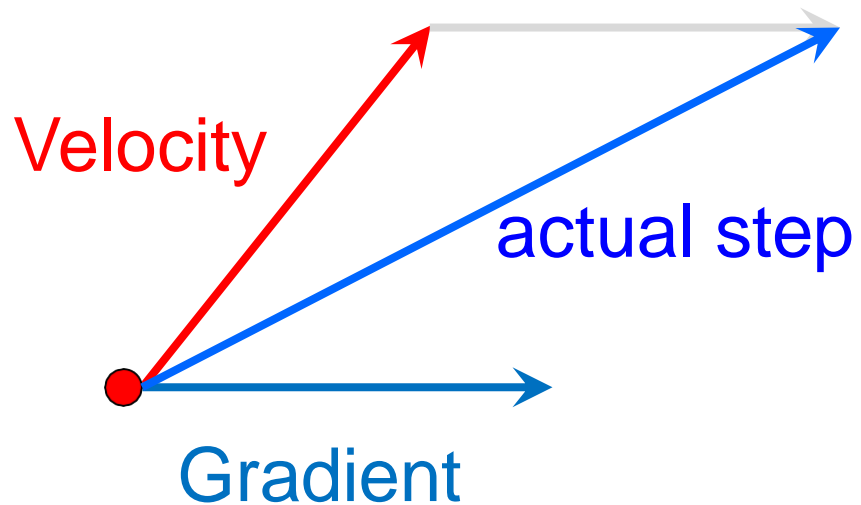
- + **Tăng tốc độ hội tụ:** Nhờ ước lượng gradient tại vị trí tiếp theo, Nesterov Momentum giúp mô hình cập nhật các tham số một cách chính xác hơn và nhanh hơn.
- + **Ổn định hơn:** Giảm thiểu nguy cơ vượt quá điểm cực tiểu và giúp hội tụ đến điểm cực tiểu nhanh hơn.

Nesterov Momentum

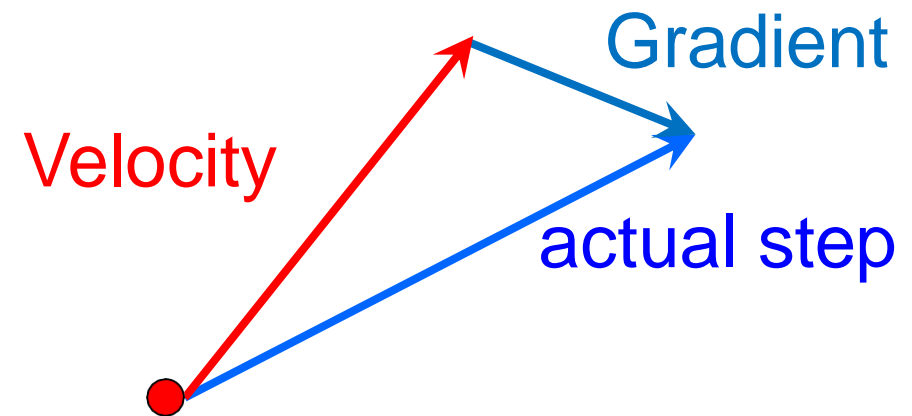
- Nesterov Momentum là một cải tiến hiệu quả của phương pháp Momentum thông thường, giúp tăng tốc và ổn định quá trình huấn luyện mô hình học máy.
- Phương pháp Nesterov Momentum thường được sử dụng trong việc huấn luyện các mạng nơ-ron sâu và các bài toán tối ưu hóa phức tạp.

Nesterov Momentum

Momentum update



Nesterov Momentum



Nesterov Momentum

— SGD + 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

— SGD + 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
 $x_t, \nabla f(x_t)$

Nesterov Momentum

— SGD + Momentum

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

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

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

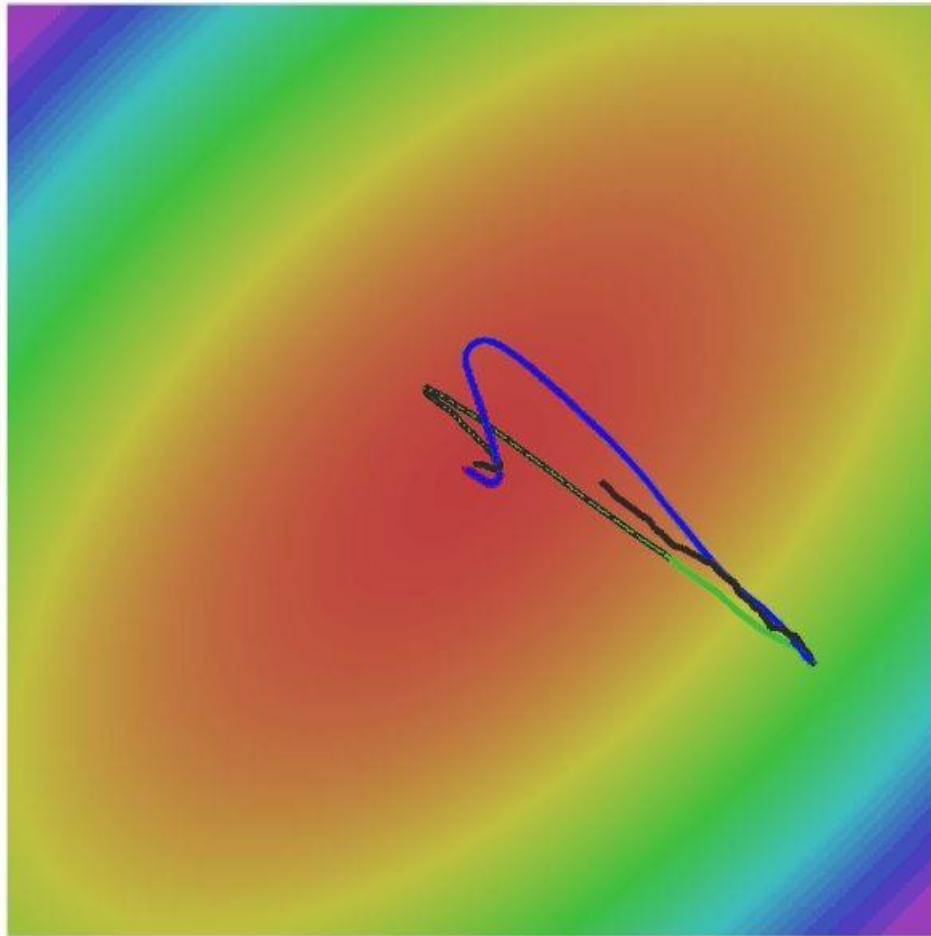
$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$

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

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

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

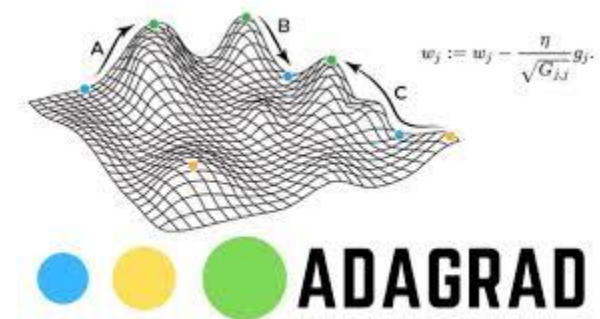
Nesterov Momentum



- SGD
- SGD+Momentum
- Nesterov

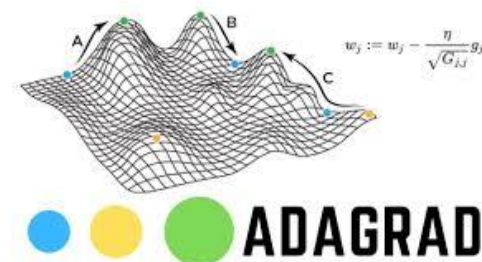
Thuật toán AdaGrad

- AdaGrad (Adaptive Gradient Algorithm) là một thuật toán tối ưu hóa được thiết kế để điều chỉnh tốc độ học (learning rate) cho từng tham số một cách tự động trong quá trình huấn luyện mô hình học máy.
- Thuật toán AdaGrad đặc biệt hữu ích khi làm việc với các dữ liệu có tần suất xuất hiện không đều (sparse data).



Thuật toán AdaGrad

- Ý tưởng chính: Thuật toán AdaGrad điều chỉnh tốc độ học cho mỗi tham số dựa trên tổng bình phương của tất cả gradient đã tính cho tham số đó. Điều này có nghĩa là những tham số thường có gradient lớn sẽ có tốc độ học giảm dần, trong khi những tham số có gradient nhỏ sẽ có tốc độ học lớn hơn. Kết quả là AdaGrad có thể đạt được sự hội tụ tốt hơn và ổn định hơn so với các thuật toán tối ưu hóa truyền thống.



Thuật toán AdaGrad

— Các bước cập nhật của thuật toán AdaGrad được định nghĩa như sau:

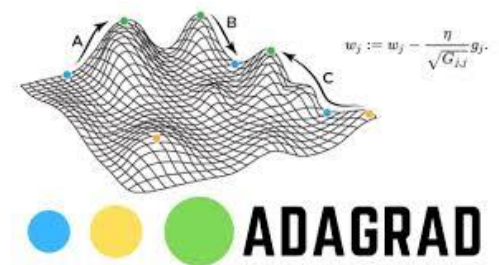
+ Tính gradient của hàm mất mát tại bước t

$$g_t = \nabla \theta(\theta_{t-1})$$

+ Cập nhật tổng bình phương của các gradient:

$$G_t = G_{t-1} + g_t \odot g_t$$

+ trong đó \odot là phép nhân từng phần tử (element-wise multiplication).



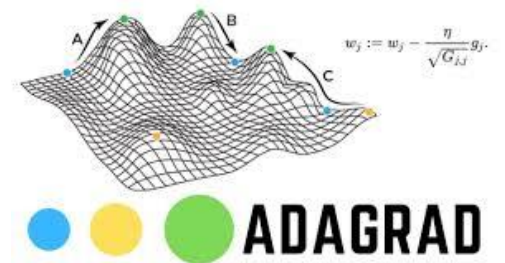
Thuật toán AdaGrad

+ Cập nhật các tham số mô hình:

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot g_t$$

+ Trong đó:

- α là tốc độ học ban đầu (learning rate).
- ϵ là một số rất nhỏ để tránh chia cho 0 (thường khoảng 10^{-8}).



Thuật toán AdaGrad

— Đặc điểm và lợi ích của AdaGrad

- + Điều chỉnh tốc độ học tự động: AdaGrad điều chỉnh tốc độ học cho từng tham số dựa trên lịch sử gradient của tham số đó, giúp mô hình học hiệu quả hơn.
- + Phù hợp với dữ liệu thưa (sparse data): AdaGrad hoạt động tốt với các dữ liệu thưa, nơi mà một số tham số có gradient lớn và các tham số khác có gradient nhỏ.
- + Đơn giản và dễ triển khai: Công thức của AdaGrad đơn giản và dễ hiểu, giúp nó trở thành lựa chọn phổ biến trong các ứng dụng thực tế.

Thuật toán AdaGrad

— Hạn chế của AdaGrad:

+ Mặc dù AdaGrad có nhiều ưu điểm, nhưng nó cũng có một số hạn chế: Tốc độ học giảm quá nhanh — Trong một số trường hợp, tốc độ học có thể giảm quá nhanh, dẫn đến mô hình hội tụ quá sớm và không đạt được tối ưu toàn cục.

Thuật toán AdaGrad

- Biến thể của AdaGrad: Để khắc phục một số hạn chế của AdaGrad, các biến thể khác như RMSProp và Adam đã được phát triển.
- Các biến thể RMSProp và Adam sử dụng trung bình trọng số theo cấp số nhân để điều chỉnh tốc độ học, giúp cải thiện hiệu suất và khả năng hội tụ của mô hình.

Thuật toán AdaGrad

— Kết luận:

- + AdaGrad là một thuật toán tối ưu hóa mạnh mẽ và linh hoạt, đặc biệt hữu ích trong các bài toán với dữ liệu thưa.
- + Bằng cách điều chỉnh tốc độ học tự động cho từng tham số, AdaGrad giúp cải thiện khả năng tổng quát hóa và hiệu suất của các mô hình học máy.
- + Tuy nhiên, cần lưu ý đến một số hạn chế của thuật toán AdaGrad và cân nhắc sử dụng các biến thể khác nếu cần thiết.

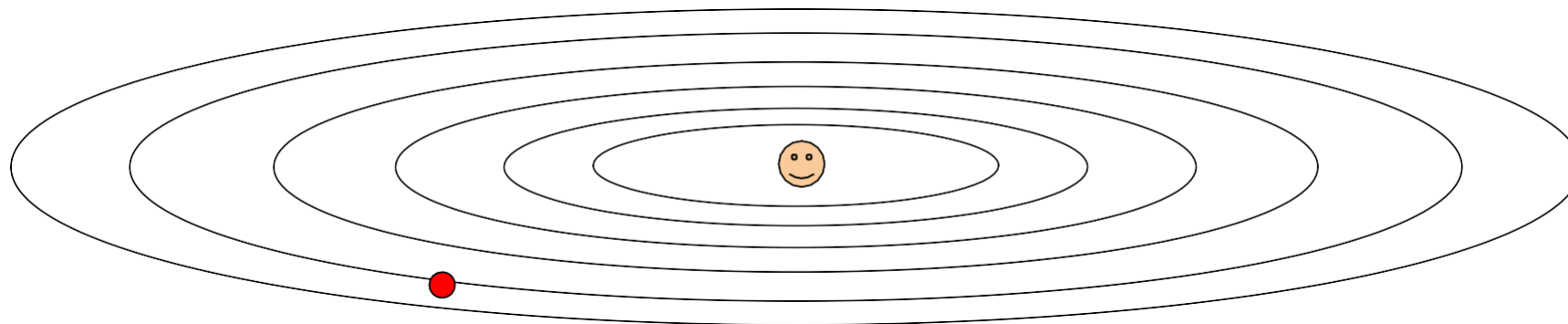
AdaGrad

```
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

AdaGrad

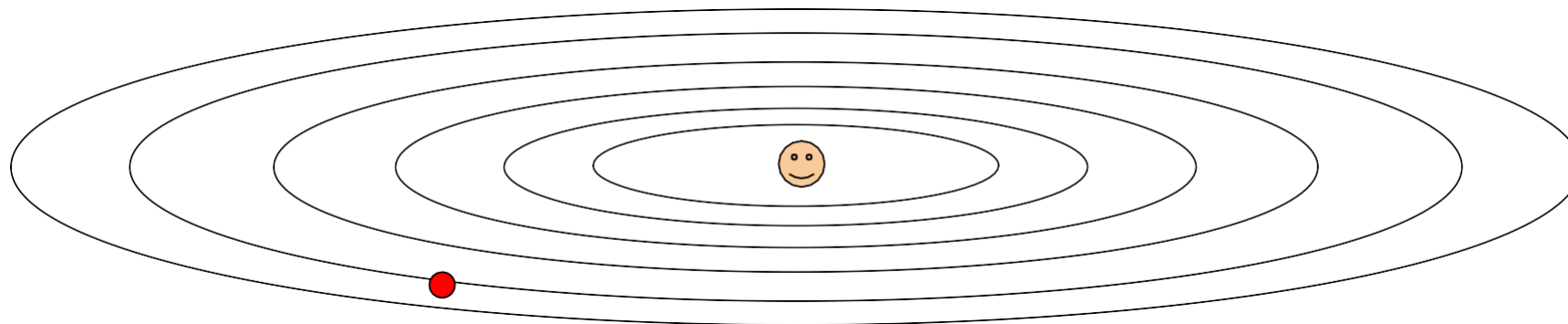
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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 to the step size over long time?

Thuật toán RMSProp

- RMSProp (Root Mean Square Propagation) là một thuật toán tối ưu hóa được phát triển để giải quyết một số hạn chế của AdaGrad.
- Đặc biệt, RMSProp điều chỉnh tốc độ học động, giúp khắc phục vấn đề tốc độ học giảm quá nhanh của AdaGrad và cải thiện hiệu quả huấn luyện của mô hình học sâu.



Thuật toán RMSProp

- Ý tưởng chính: RMSProp duy trì một giá trị trung bình động của bình phương đạo hàm và sử dụng giá trị này để điều chỉnh tốc độ học cho từng tham số.
- Bằng cách này, RMSProp có thể giữ cho tốc độ học ổn định trong suốt quá trình huấn luyện, giúp mô hình hội tụ nhanh hơn và hiệu quả hơn.



Thuật toán RMSProp

— Thuật toán của RMSProp:

+ Tính đạo hàm của hàm mất mát tại bước t :

$$g_t = \nabla \theta \mathcal{L}(\theta_{t-1})$$

+ Cập nhật giá trị trung bình động của bình phương gradient:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta) g_t^2$$

+ Trong đó:

- β là hệ số điều chỉnh (thông thường $\beta \in [0.9, 0.99]$)
- $E[g^2]_t$ là giá trị trung bình động của bình phương gradient tại bước t .

Thuật toán RMSProp

— Thuật toán của RMSProp:

+ ...

+ Cập nhật các tham số mô hình:

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

+ Trong đó:

- α là tốc độ học ban đầu (learning rate).
- ϵ là một số rất nhỏ để tránh chia cho 0 (thường khoảng 10^{-8}).

Thuật toán RMSProp

— Đặc điểm và lợi ích của RMSProp

- + Điều chỉnh tốc độ học động: RMSProp điều chỉnh tốc độ học dựa trên giá trị trung bình động của bình phương gradient, giúp tốc độ học không giảm quá nhanh và giữ cho quá trình huấn luyện ổn định hơn.
- + Phù hợp với dữ liệu thưa (sparse data): RMSProp hoạt động tốt với các dữ liệu thưa, tương tự như AdaGrad, nhưng khắc phục được nhược điểm tốc độ học giảm quá nhanh.

Thuật toán RMSProp

— Đặc điểm và lợi ích của RMSProp

+ ...

+ Tích hợp đơn giản và hiệu quả: RMSProp có công thức đơn giản và dễ triển khai, nhưng mang lại hiệu quả cao trong các ứng dụng thực tế.

Thuật toán RMSProp

— Kết luận:

- + RMSProp là một thuật toán tối ưu hóa mạnh mẽ và hiệu quả, giúp cải thiện quá trình huấn luyện mô hình học sâu bằng cách điều chỉnh tốc độ học động dựa trên giá trị trung bình động của bình phương gradient. Với khả năng khắc phục các hạn chế của AdaGrad và giữ cho tốc độ học ổn định, RMSProp đã trở thành một trong những thuật toán tối ưu hóa phổ biến và được sử dụng rộng rãi trong học máy và trí tuệ nhân tạo.

RMSProp

AdaGrad

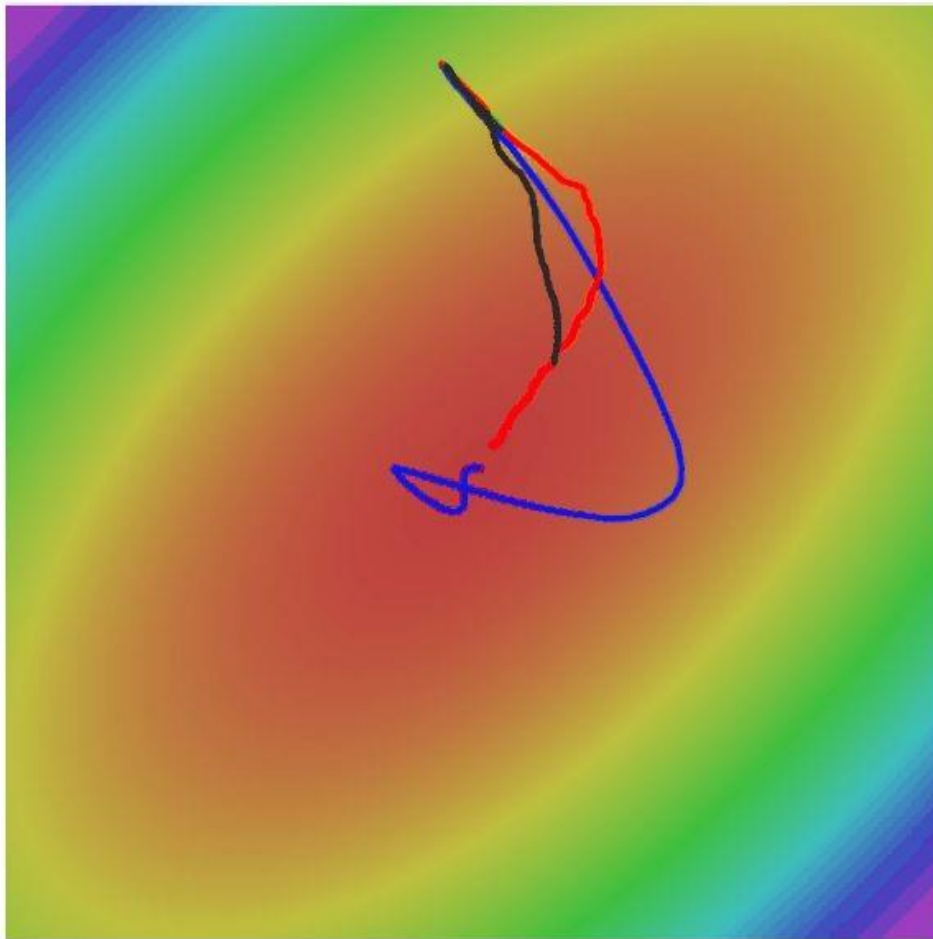
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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)
```

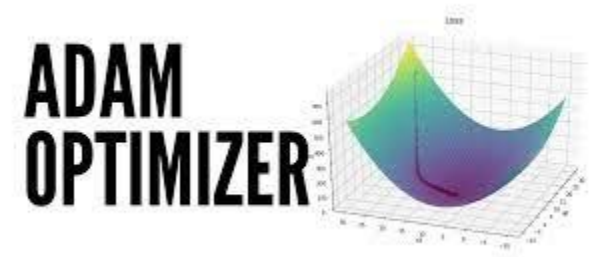
RMSProp



- SGD
- SGD + Momentum
- RMSProp

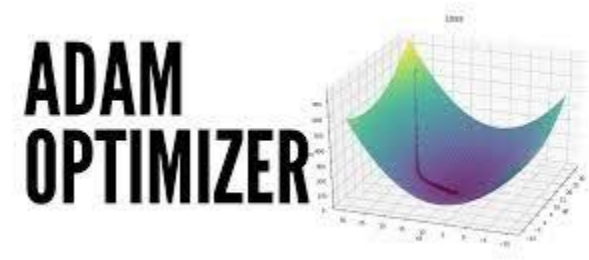
Thuật toán Adam

- Adam (Adaptive Moment Estimation) là một thuật toán tối ưu hóa kết hợp các ưu điểm của hai phương pháp phổ biến: RMSProp và Momentum.
- Adam được thiết kế để điều chỉnh tốc độ học cho từng tham số một cách tự động và động, giúp tăng tốc quá trình hội tụ và cải thiện hiệu quả huấn luyện của các mô hình học máy, đặc biệt là các mạng nơ-ron sâu.



Thuật toán Adam

- Ý tưởng chính: Adam duy trì cả hai giá trị trung bình động của đạo hàm (first moment) và bình phương đạo hàm (second moment), sau đó điều chỉnh các giá trị này để cập nhật các tham số của mô hình.
- Điều này giúp Adam có thể xử lý tốt hơn các đạo hàm không ổn định và hội tụ nhanh hơn.



Thuật toán Adam

— Thuật toán của Adam:

+ Tính đạo hàm của hàm mất mát tại bước t :

$$g_t = \nabla \theta \mathcal{L}(\theta_{t-1})$$

+ Cập nhật giá trị trung bình động của đạo hàm (first moment estimate):

$$m_t = \beta_1 m_{t-1} + (1 - \beta) g_t$$

+ Trong đó:

- β_1 là hệ số điều chỉnh của first moment (thường $\beta_1 \approx 0.9$)

Thuật toán Adam

— Thuật toán của RMSProp:

+ ...

+ Cập nhật giá trị trung bình động của đạo hàm bình phương (second moment estimate):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

+ Trong đó:

- β_2 là hệ số điều chỉnh của second moment (thường $\beta_2 \approx 0.999$)

Thuật toán Adam

— Thuật toán của RMSProp:

+ ...

+ Điều chỉnh giá trị trung bình động để giảm bias (bias- corrected estimate):

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Thuật toán Adam

— Thuật toán của RMSProp:

+ ...

+ Cập nhật các tham số mô hình:

$$\theta_t = \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

+ Trong đó:

- α là tốc độ học ban đầu (learning rate).
- ϵ là một số rất nhỏ để tránh chia cho 0 (thường khoảng 10^{-8}).

Thuật toán Adam

— Đặc điểm và lợi ích của Adam

- + **Điều chỉnh tốc độ học động:** Adam điều chỉnh tốc độ học cho từng tham số dựa trên cả giá trị trung bình động của gradient và bình phương gradient, giúp quá trình huấn luyện ổn định hơn.
- + **Phù hợp với dữ liệu thưa (sparse data):** Adam hoạt động tốt với các dữ liệu thưa và các gradient không ổn định, giúp mô hình học hiệu quả hơn.

Thuật toán Adam

— Đặc điểm và lợi ích của RMSProp

- + **Tốc độ hội tụ nhanh:** Adam kết hợp các ưu điểm của Momentum và RMSProp, giúp mô hình hội tụ nhanh hơn so với các phương pháp truyền thống.
- + **Đơn giản và hiệu quả:** Công thức của Adam đơn giản và dễ triển khai, nhưng mang lại hiệu quả cao trong các ứng dụng thực tế.

Thuật toán Adam

— Kết luận:

+ Adam là một trong những thuật toán tối ưu hóa mạnh mẽ và phổ biến nhất trong học máy và trí tuệ nhân tạo. Bằng cách kết hợp các ưu điểm của Momentum và RMSProp, Adam giúp cải thiện quá trình huấn luyện mô hình học sâu, đảm bảo tốc độ hội tụ nhanh và hiệu quả cao. Adam thường được sử dụng rộng rãi trong các bài toán học sâu và các mạng nơ-ron sâu phức tạp.

Adam (almost)

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

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (almost)

```
1. first_moment = 0
2. second_moment = 0
3. while True:
4.     dx = compute_gradient(x)
5.     first_moment = beta1 * first_moment + (1 - beta1) * dx           Momentum
6.     second_moment = beta2 * second_moment + (1 - beta2) * dx * dx   AdaGrad/RMSProp
7.     x -= learning_rate*first_moment/(np.sqrt(second_moment)+1e-7)   AdaGrad/RMSProp
```

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (full form)

```
1. first_moment = 0
2. second_moment = 0
3. for t in range(num_iterations):
4.     dx = compute_gradient(x)
5.     first_moment = beta1 * first_moment + (1 - beta1) * dx           Momentum
6.     second_moment = beta2 * second_moment + (1 - beta2) * dx * dx   AdaGrad/RMSProp
7.     first_unbias = first_moment / (1 - beta1 ** t)                   Bias correction
8.     second_unbias = second_moment / (1 - beta2 ** t)                 Bias correction
9.     x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7) AdaGrad/RMSProp
```

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

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam (full form)

```

1. first_moment = 0
2. second_moment = 0
3. for t in range(1, num_iterations):
4.     dx = compute_gradient(x)
5.     first_moment = beta1 * first_moment + (1 - beta1) * dx           Momentum
6.     second_moment = beta2 * second_moment + (1 - beta2) * dx * dx   AdaGrad/RMSProp
7.     first_unbias = first_moment / (1 - beta1 ** t)                   Bias correction
8.     second_unbias = second_moment / (1 - beta2 ** t)                 Bias correction
9.     x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7) AdaGrad/RMSProp

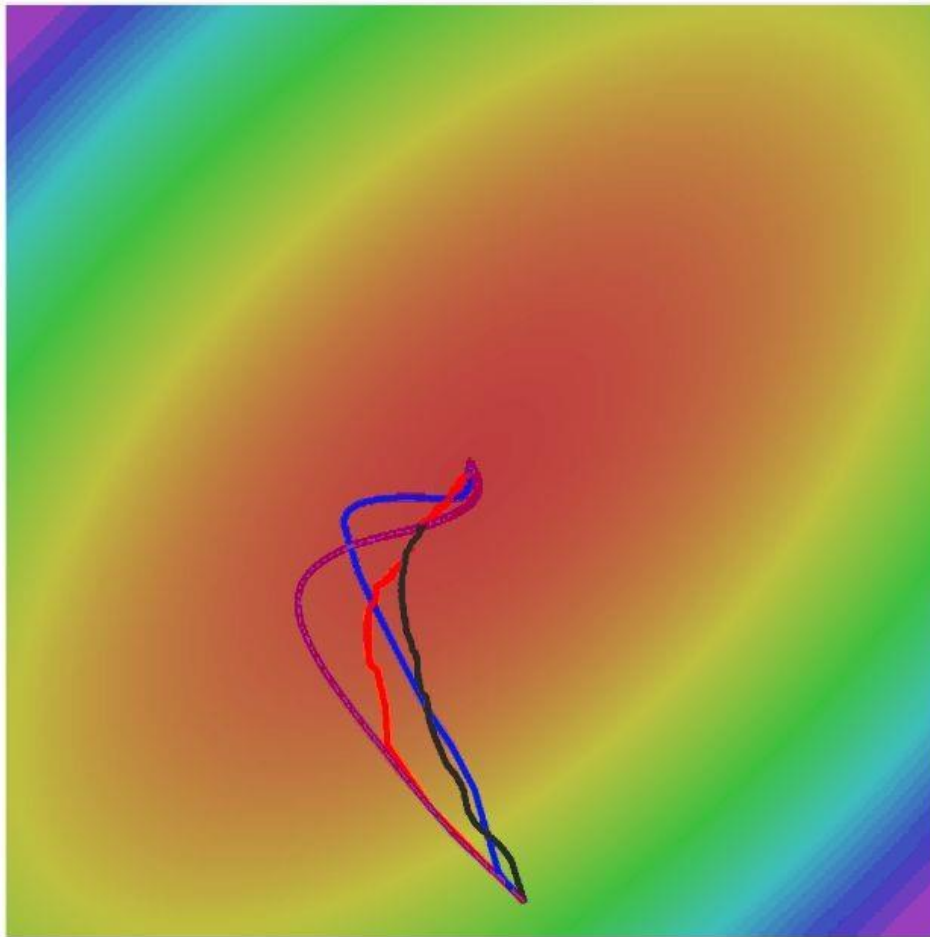
```

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!

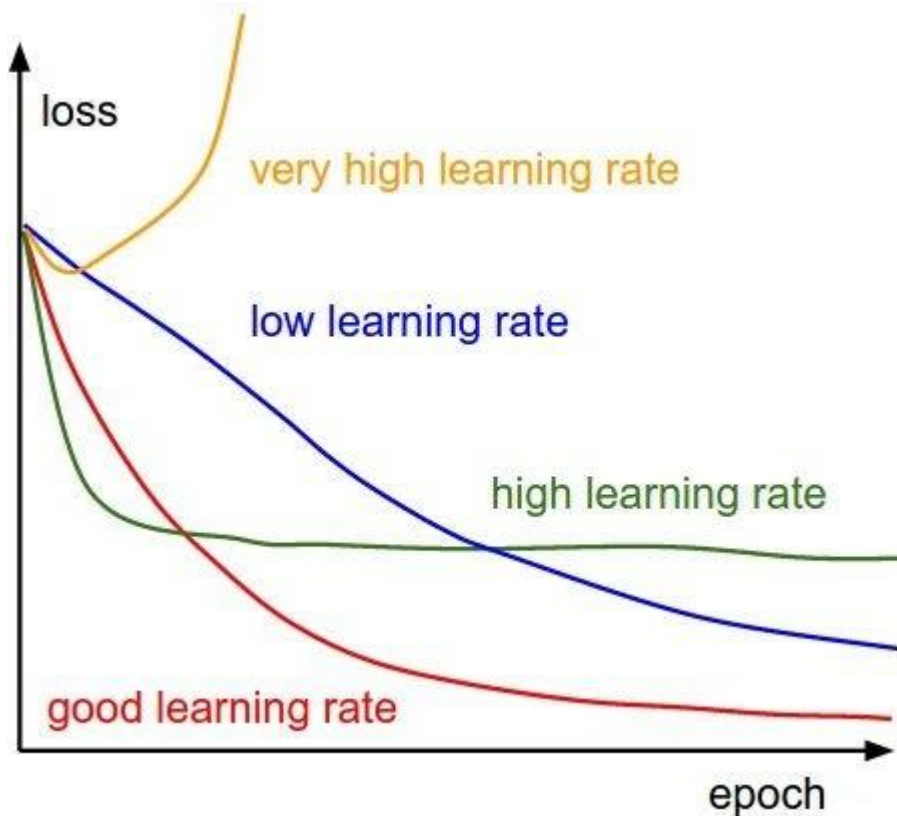
Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam



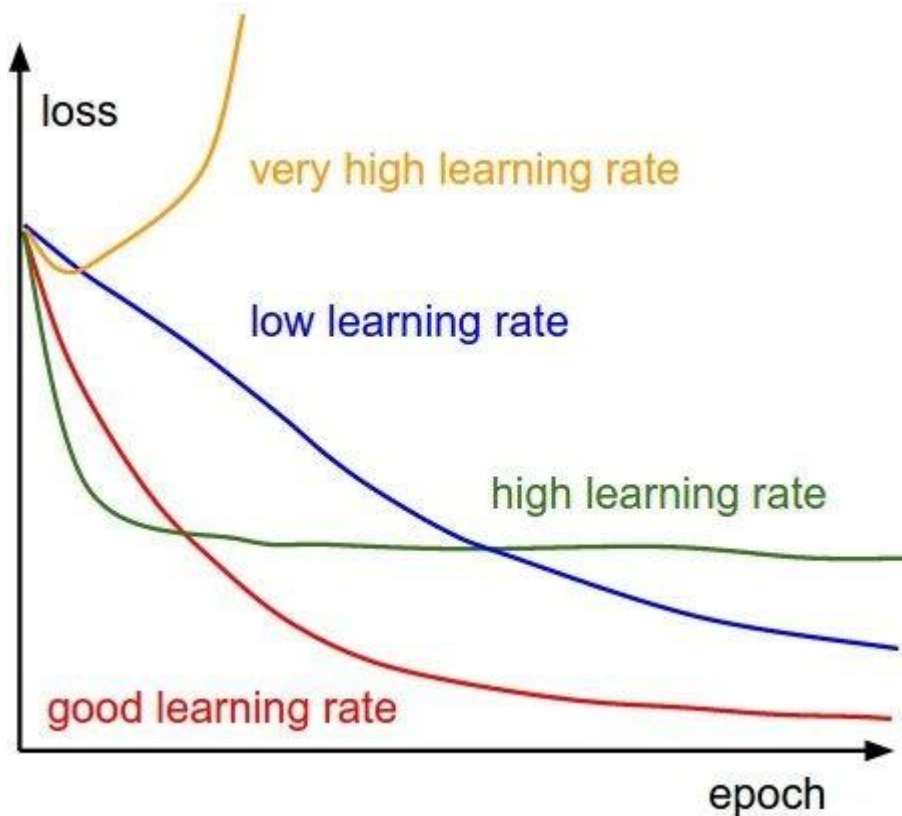
- SGD
- SGD + Momentum
- RMSProp
- Adam

Learning rate as a hyperparameter



- SGD, SGD + Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.
- Q: Which one of these learning rates is best to use?

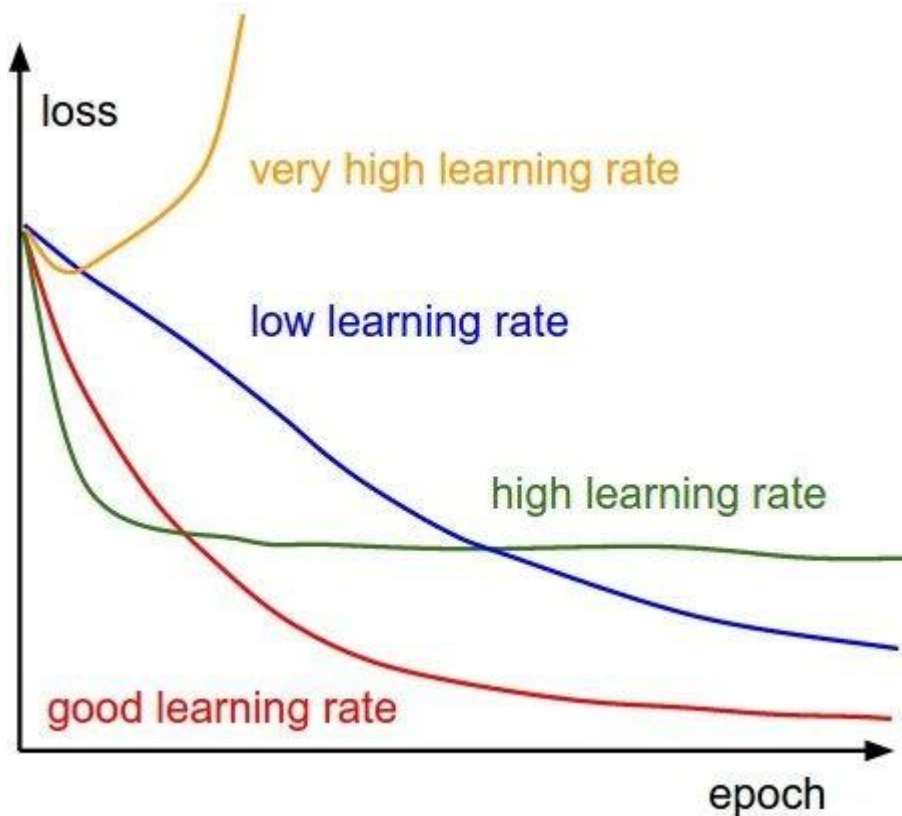
Learning rate as a hyperparameter



⇒ Learning rate decay over time!

Tốc độ học tập giảm dần theo thời gian!

Learning rate as a hyperparameter

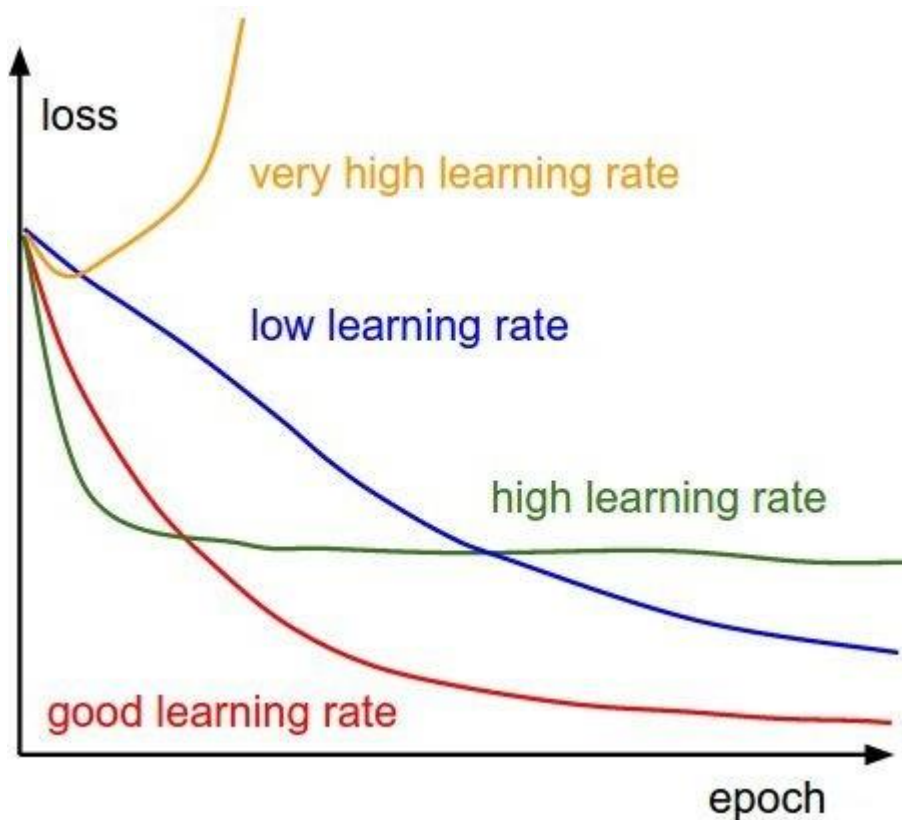


⇒ Learning rate decay over time!

Tốc độ học tập giảm dần theo thời gian!

- **Step decay:** e.g. decay learning rate by half every few epochs.
- **Step Decay:** Giảm tốc độ học theo các bước cố định.

Learning rate as a hyperparameter

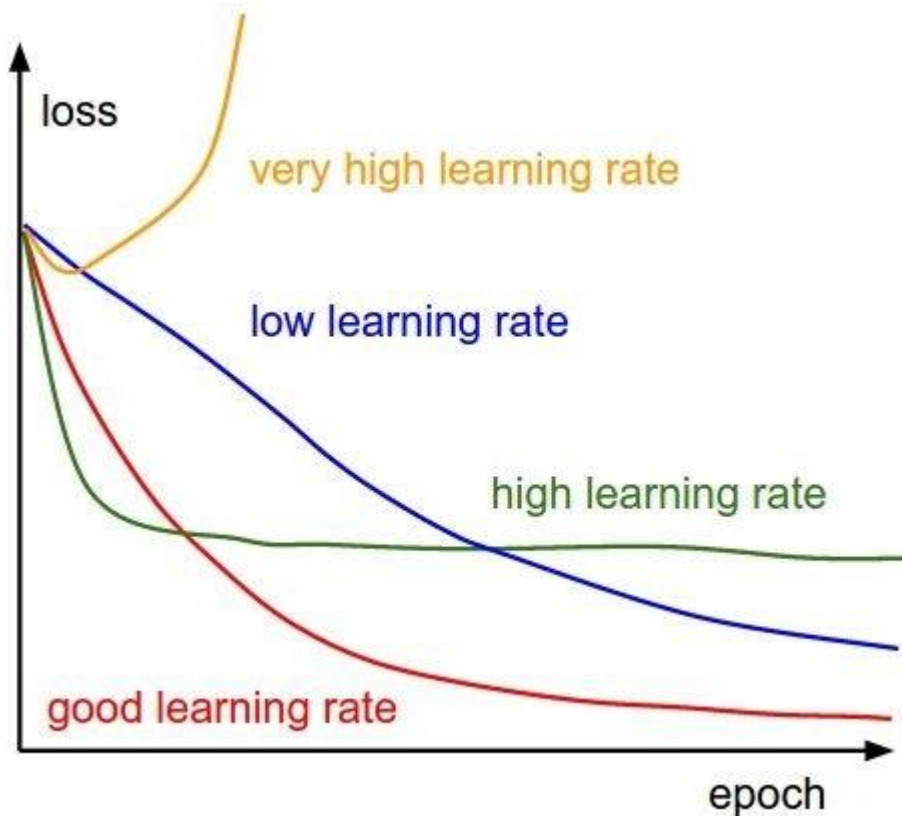


⇒ Learning rate decay over time!

Tốc độ học tập giảm dần theo thời gian!

- **Exponential decay:** $\alpha = \alpha_0 e^{-kt}$
- **Exponential Decay:** Giảm tốc độ học theo hàm mũ của số epoch.

Learning rate as a hyperparameter



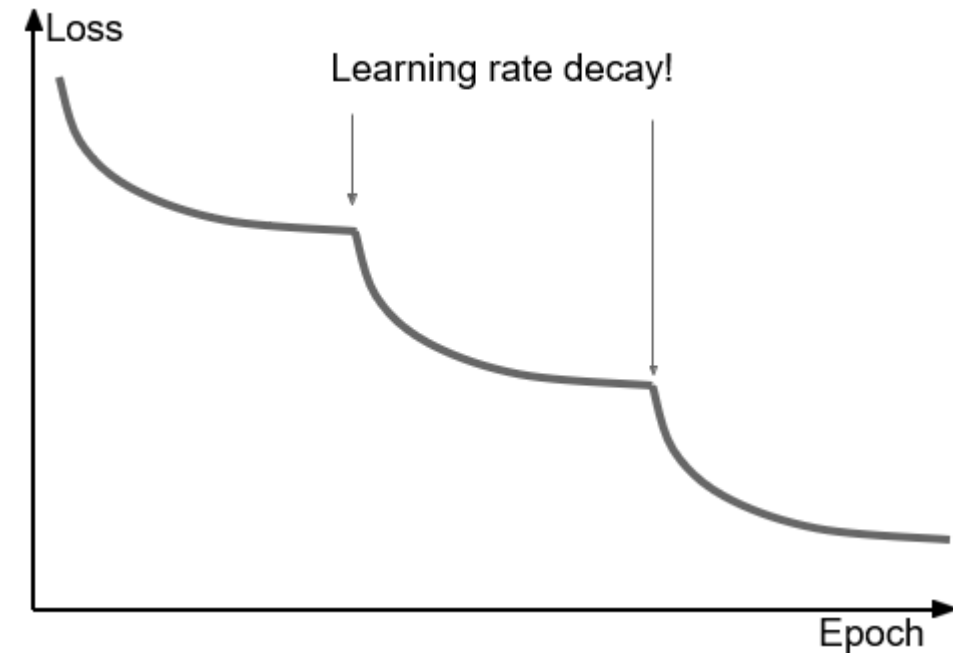
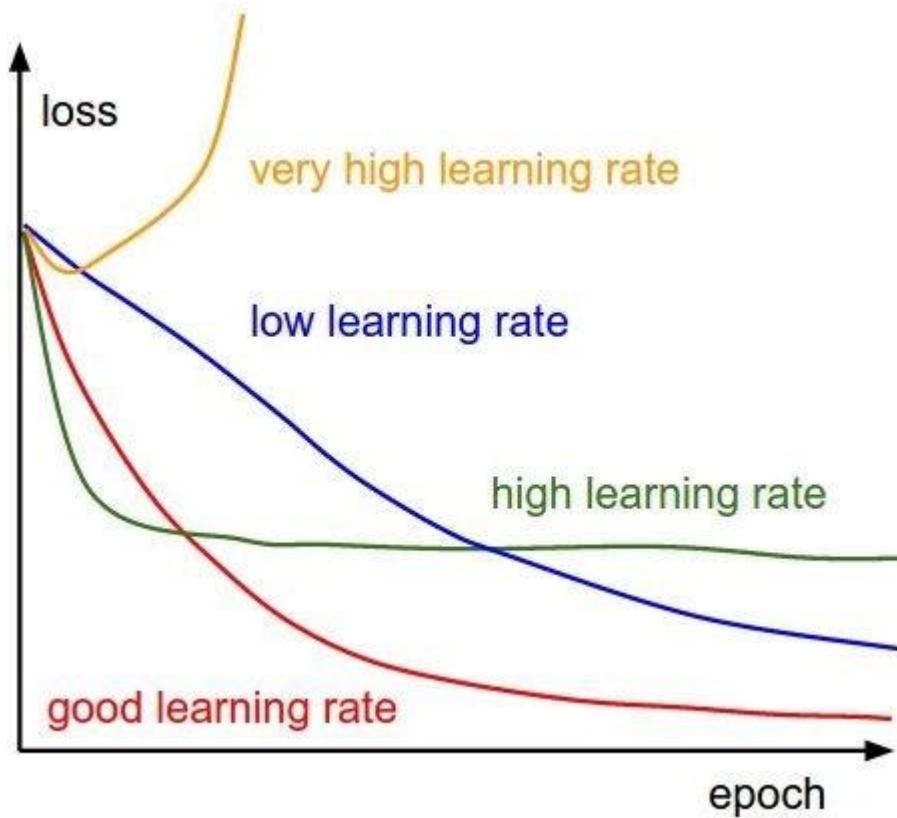
⇒ Learning rate decay over time!

Tốc độ học tập giảm dần theo thời gian!

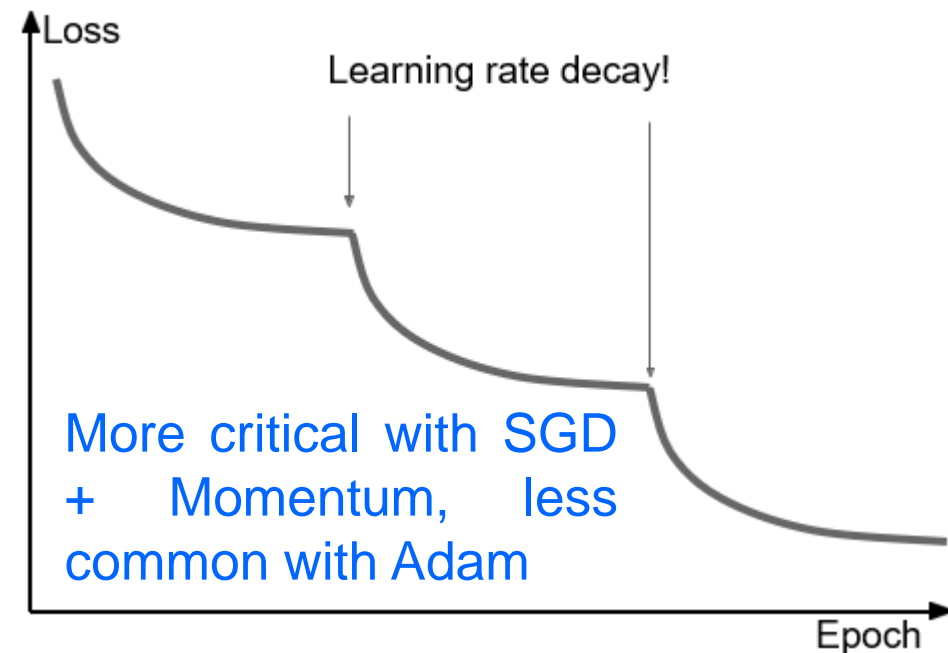
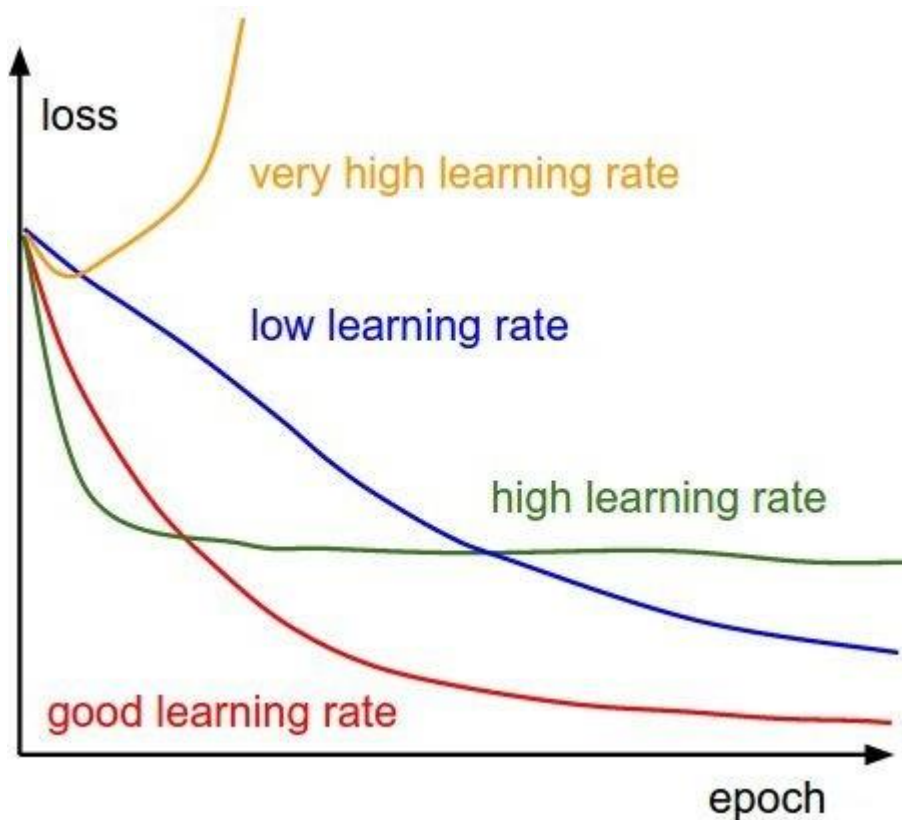
– $\frac{1}{t}$ decay: $\alpha = \frac{\alpha_0}{1+kt}$

– $\frac{1}{t}$ decay: Giảm tốc độ học tỉ lệ nghịch với số epoch.

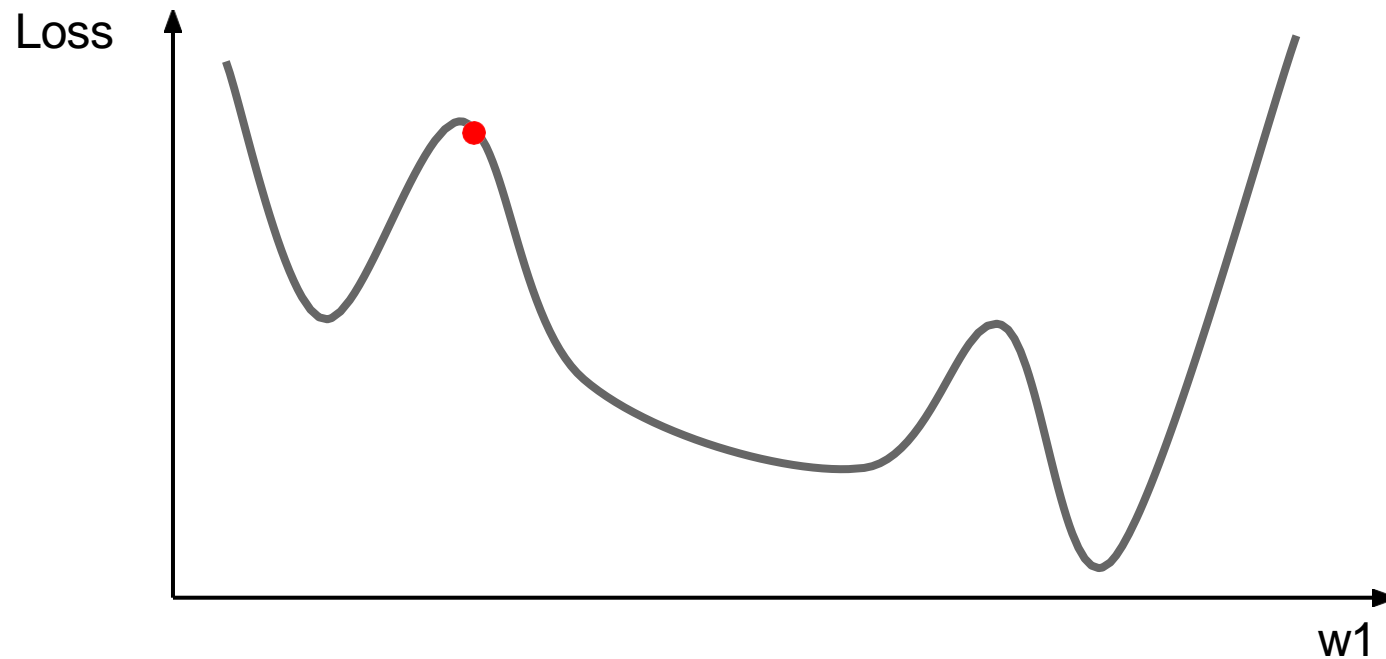
Learning rate as a hyperparameter



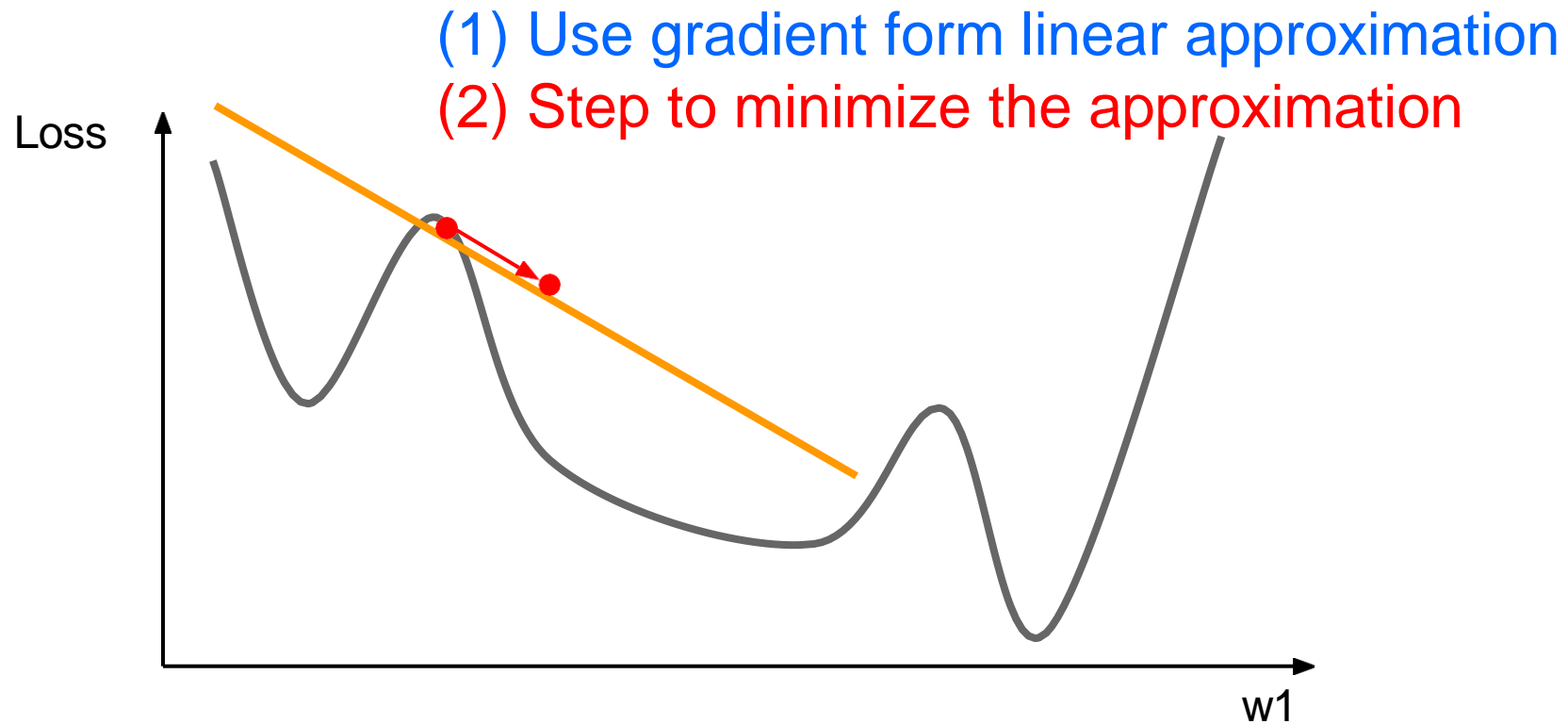
Learning rate as a hyperparameter



First-Order Optimization

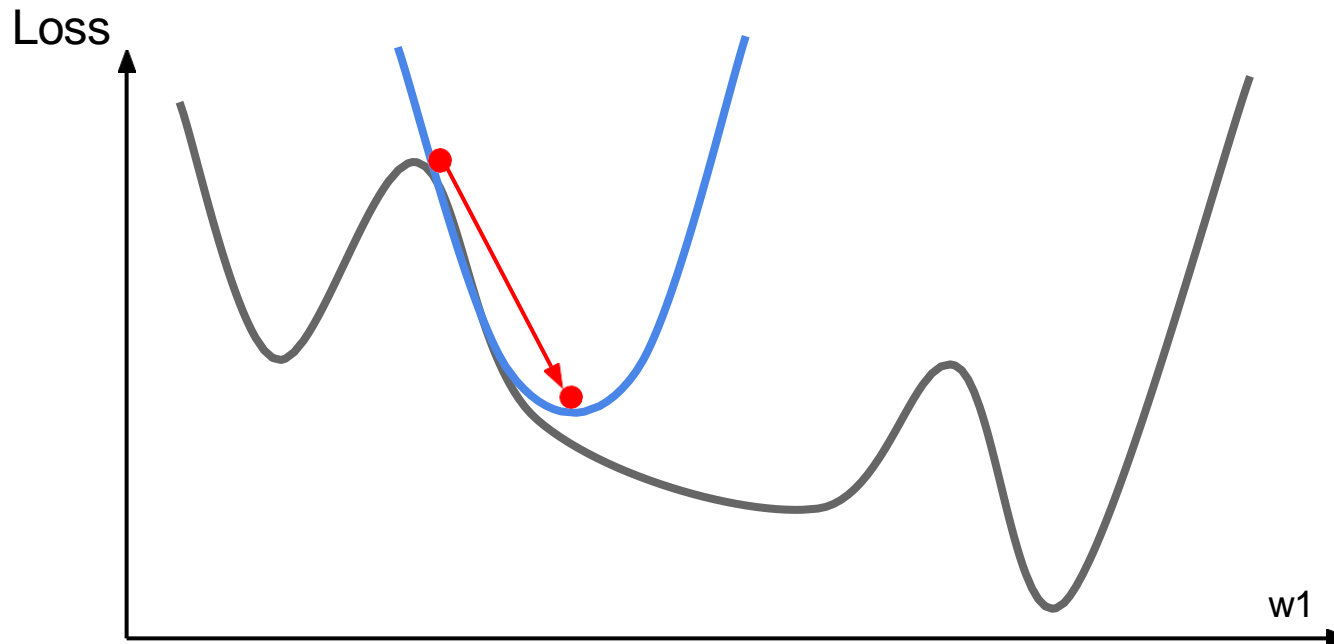


First-Order Optimization



Second-Order Optimization

- (1) Use gradient and Hessian to form quadratic approximation
- (2) Step to the minima of the approximation



Second-Order Optimization

- second-order Taylor expansion:

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_{\theta} J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^T H (\theta - \theta_0)$$

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

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0)$$

- Q1: What is nice about this update?

Second-Order Optimization

- second-order Taylor expansion:

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_{\theta} J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^T H (\theta - \theta_0)$$

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

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0)$$

No hyperparameters!

No learning rate!

- Q1: What is nice about this update?

Second-Order Optimization

- second-order Taylor expansion:

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_{\theta} J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^T H (\theta - \theta_0)$$

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

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0)$$

Hessian has $O(N^2)$ elements

Inverting takes $O(N^3)$

N = (Tens or Hundreds of) Millions

- Q2: Why is this bad for deep learning?

Second-Order Optimization

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0)$$

- 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.

L-BFGS

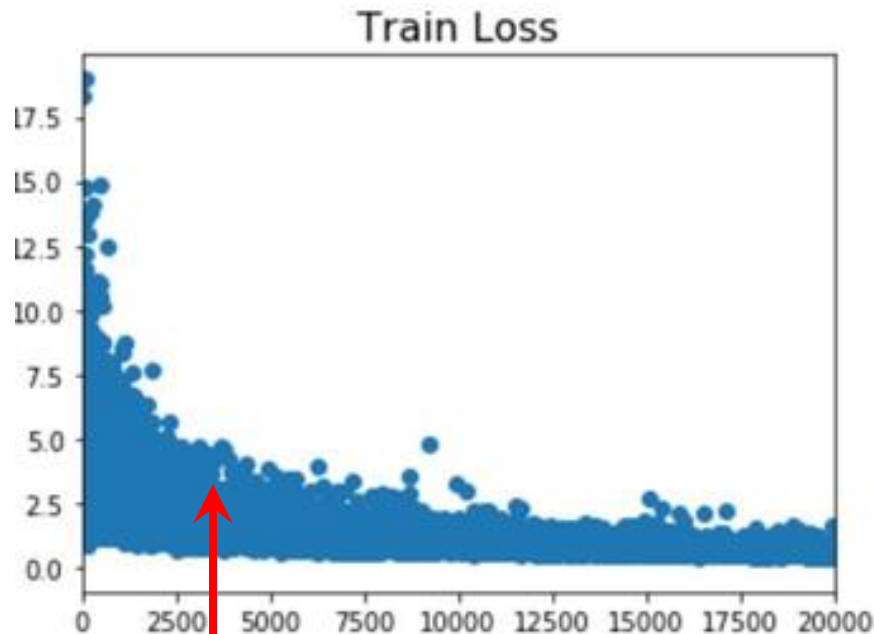
- Usually works very well in full batch, deterministic mode i.e. if you have a single, deterministic $f(x)$ then L-BFGS will probably work very nicely
- Does not transfer very well to mini-batch setting. Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

Le et al, “On optimization methods for deep learning, ICML 2011”

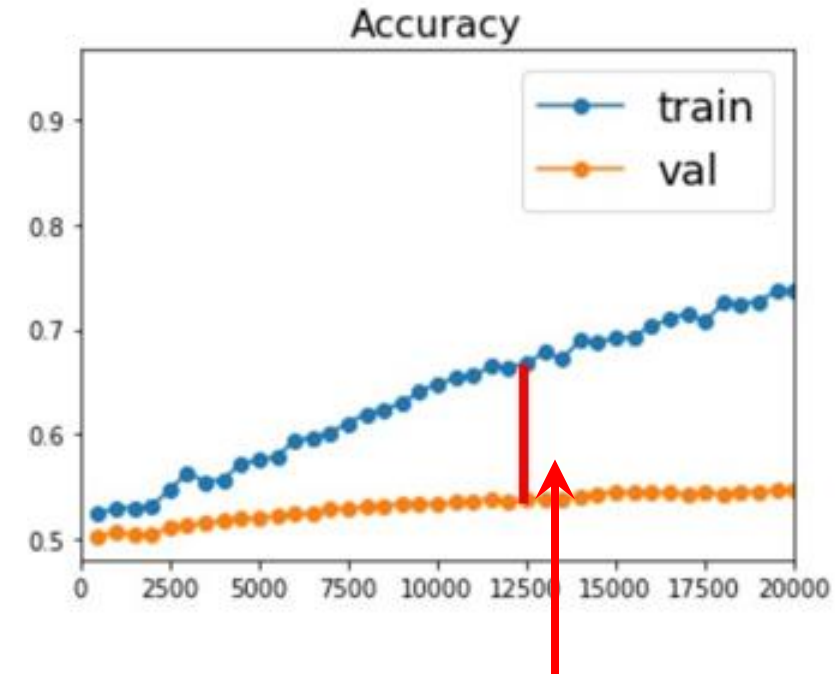
In practice

- Adam is a good default choice in most cases.
- Adam là lựa chọn mặc định tốt trong hầu hết các trường hợp.
- If you can afford to do full batch updates then try out L-BFGS (and don't forget to disable all sources of noise).
- Nếu có khả năng thực hiện cập nhật hàng loạt đầy đủ thì hãy thử L-BFGS (và đừng quên tắt tất cả các nguồn gây nhiễu).

Beyond Training Error



— Better optimization algorithms help reduce training loss



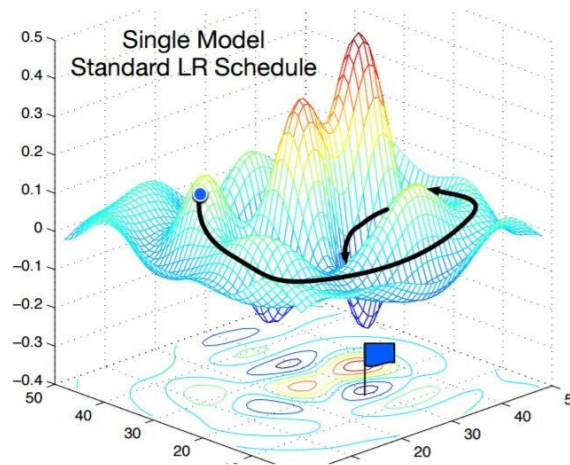
— But we really care about error on new data – how to reduce the gap?

Model Ensembles

1. Train multiple independent models
 2. At test time average their results
- Enjoy 2% extra performance

Model Ensembles: Tips and Tricks

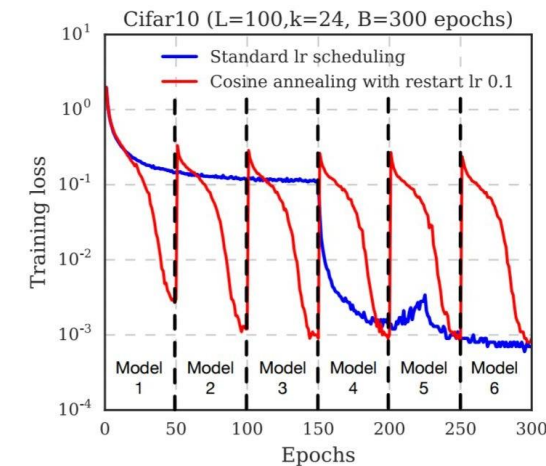
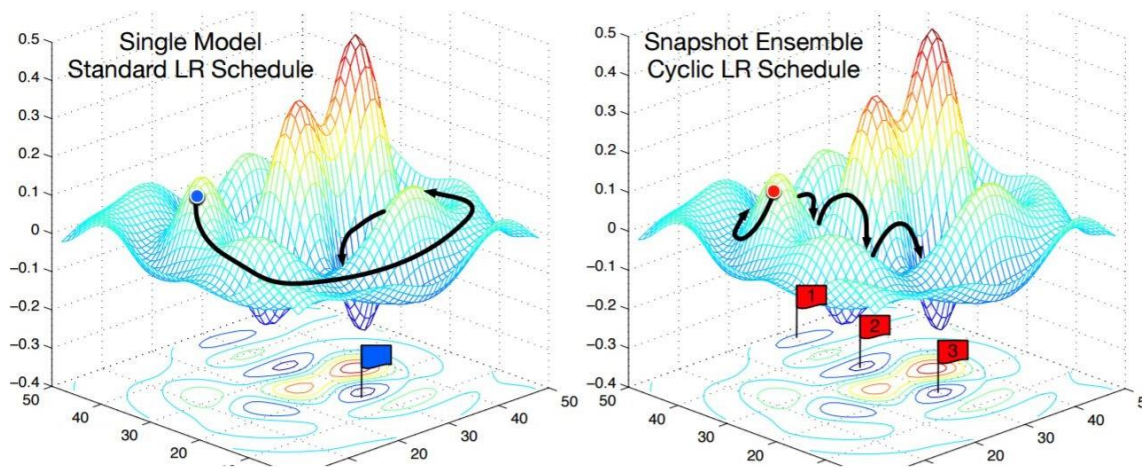
- Instead of training independent models, use multiple snapshots of a single model during training!



Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016 Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017 Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission

Model Ensembles: Tips and Tricks

- Instead of training independent models, use multiple snapshots of a single model during training!



Cyclic learning rate schedules can make this work even better!

Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016 Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017 Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission

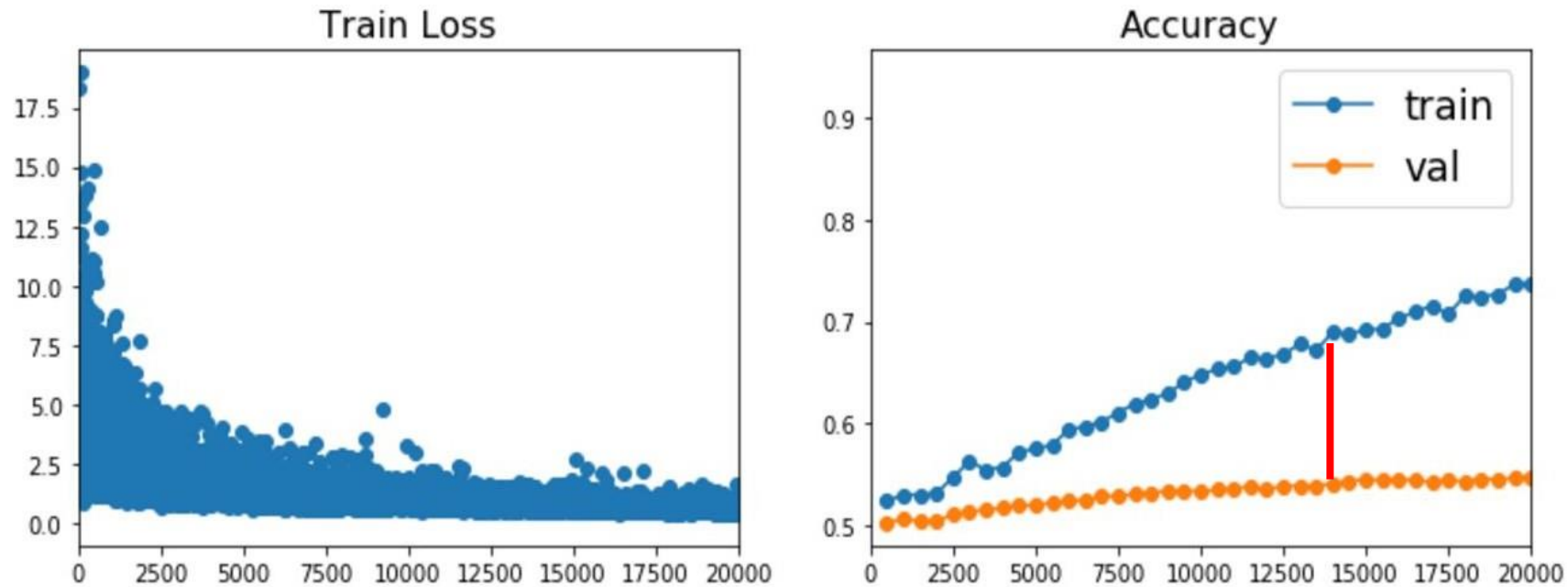
Model Ensembles: Tips and Tricks

- Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging).

```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
    x_test = 0.995*x_test + 0.005*x # use for test set
```

Polyak and Juditsky, “Acceleration of stochastic approximation by averaging”, SIAM Journal on Control and Optimization, 1992.

How to improve single-model performance?



Regularization

Regularization: Add term to loss

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y^{(i)}} \max \left(0, f(x^{(i)}; W)_j^{(i)} - f(x^{(i)}; W)_{y^{(i)}}^{(i)} + 1 \right) + \boxed{\lambda R(W)}$$

— λ : regularization strength (hyperparameter)

— In common use:

1. L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$
2. L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$
3. Elastic net (L1 + L2): $R(W) = \sum_k \sum_l (\beta W_{k,l}^2 + |W_{k,l}|)$
4. ...

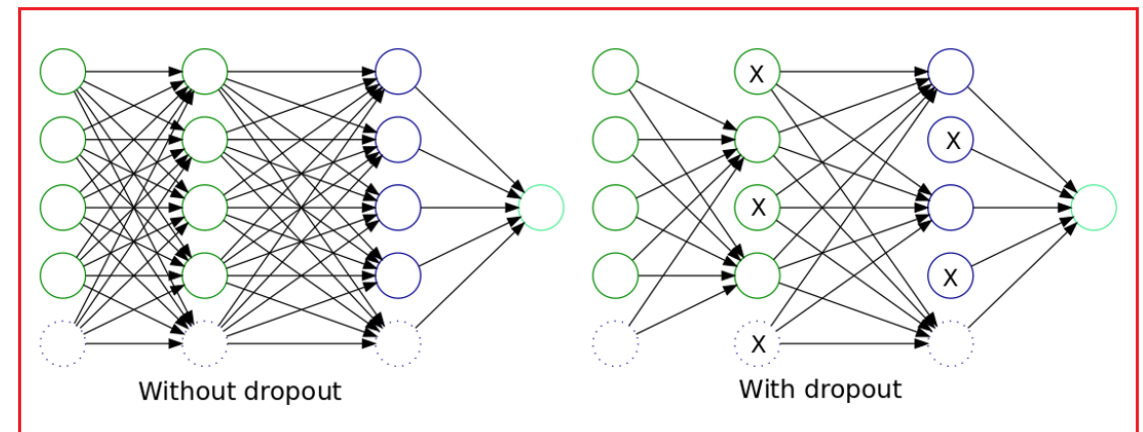
Regularization: Add term to loss

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y^{(i)}} \max \left(0, f(x^{(i)}; W)_j^{(i)} - f(x^{(i)}; W)_{y^{(i)}}^{(i)} + 1 \right) + \boxed{\lambda R(W)}$$

- λ : regularization strength (hyperparameter)
- In common use:
 4. Max norm regularization (might see later)
 5. Dropout (will see later)
 6. Fancier: Batch normalization, stochastic depth

Dropout

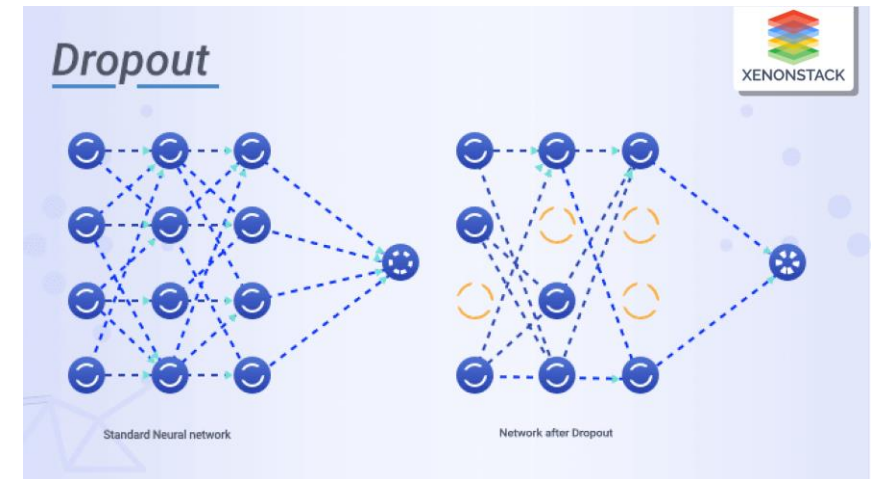
- Dropout là một kỹ thuật phổ biến được sử dụng trong huấn luyện mạng nơ-ron nhân tạo nhằm giảm thiểu tình trạng overfitting (quá khớp) và cải thiện khả năng tổng quát của mô hình.
- Kỹ thuật này được giới thiệu bởi Geoffrey Hinton và các cộng sự vào năm 2012.



Dropout

— Cách hoạt động của Dropout:

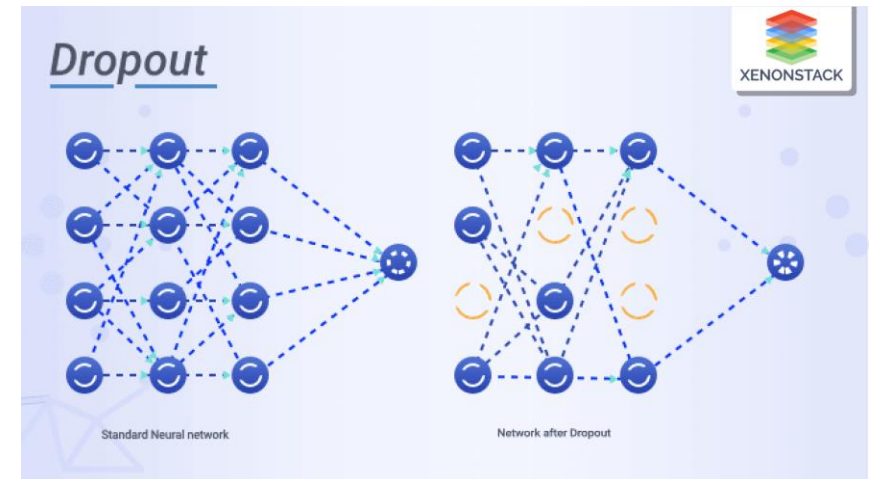
- + **Quy trình:** Trong quá trình huấn luyện, tại mỗi bước huấn luyện, dropout sẽ ngẫu nhiên "loại bỏ" (drop out) một tỷ lệ các neuron trong mạng. Điều này có nghĩa là một số neuron sẽ không được kích hoạt và không tham gia vào quá trình lan truyền xuôi hoặc lan truyền ngược trong bước đó.



Dropout

— Cách hoạt động của Dropout:

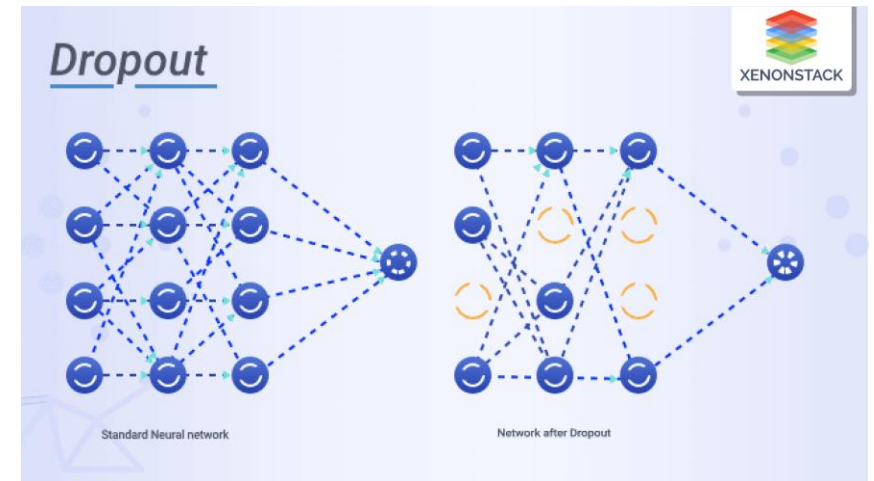
- + **Mục đích:** Bằng cách ngẫu nhiên loại bỏ các neuron, dropout làm giảm sự phụ thuộc giữa các neuron. Điều này giúp mạng nơ-ron học được các đặc trưng tổng quát hơn thay vì học thuộc các đặc trưng cụ thể của dữ liệu huấn luyện, từ đó giảm thiểu tình trạng overfitting.



Dropout

— Cách hoạt động của Dropout:

- + Trong giai đoạn suy luận: Khi mạng đã được huấn luyện xong và bước vào giai đoạn suy luận (inference), tất cả các neuron đều được kích hoạt, nhưng đầu ra của mỗi neuron được nhân với tỷ lệ dropout đã sử dụng trong quá trình huấn luyện. Điều này giúp đảm bảo rằng đầu ra của mạng được chuẩn hóa.



Dropout

— Ưu điểm của Dropout:

- + **Giảm overfitting:** Bằng cách ngẫu nhiên loại bỏ các neuron, dropout giúp mạng không phụ thuộc quá nhiều vào bất kỳ tập hợp con nào của các neuron và do đó giảm thiểu tình trạng overfitting.
- + **Cải thiện khả năng tổng quát:** Dropout khuyến khích mạng học được các đặc trưng tổng quát hơn từ dữ liệu.

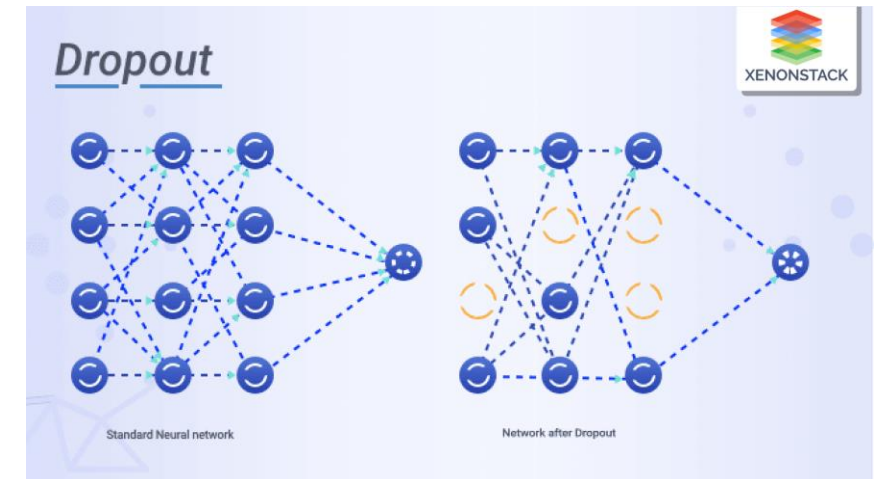
Dropout

— Nhược điểm của Dropout:

- + **Thời gian huấn luyện lâu hơn:** Dropout có thể làm tăng thời gian huấn luyện vì mạng cần học để làm việc với các tập hợp con khác nhau của các neuron tại mỗi bước huấn luyện.
- + **Không phù hợp với tất cả các mô hình:** Dropout chủ yếu được sử dụng trong các mạng nơ-ron sâu và có thể không phù hợp với các loại mô hình khác.

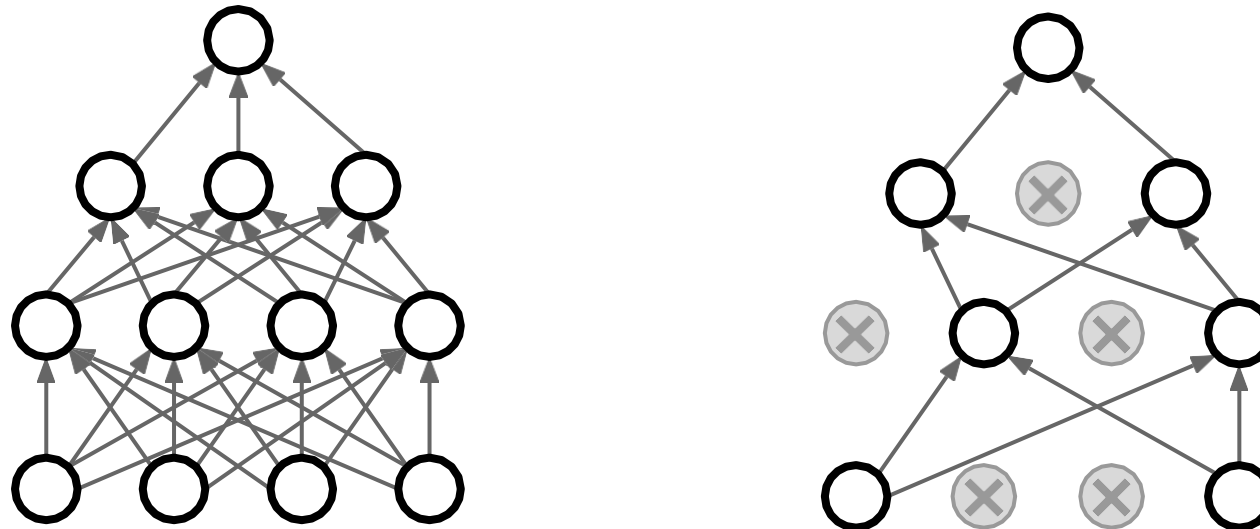
Dropout

- Tóm lại, dropout là một kỹ thuật mạnh mẽ để giảm thiểu tình trạng overfitting và cải thiện khả năng tổng quát của mạng nơ-ron nhân tạo, được sử dụng rộng rãi trong học sâu.



Regularization: Dropout

- In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common.



Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Regularization: Dropout

```
p = 0.5 # probability of keeping a unit active. higher = less dropout
```

```
def train_step(X):
```

```
    """ X contains the data """
```

```
    # forward pass for example 3-layer neural network
```

```
    H1 = np.maximum(0, np.dot(W1, X) + b1)
```

```
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
```

```
    H1 *= U1 # drop!
```

```
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
```

```
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
```

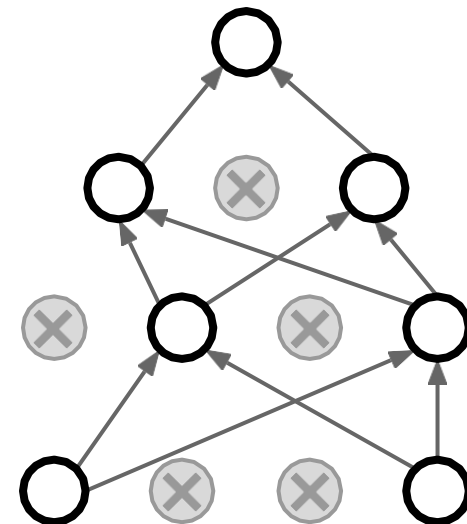
```
    H2 *= U2 # drop!
```

```
    out = np.dot(W3, H2) + b3
```

```
    # backward pass: compute gradients... (not shown)
```

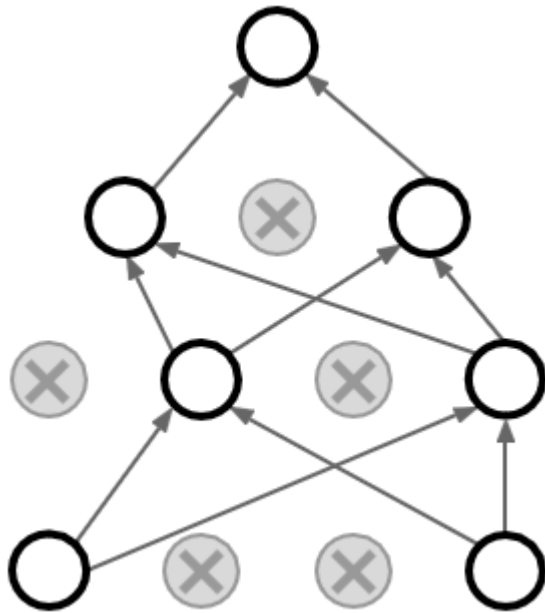
```
    # perform parameter update... (not shown)
```

— Example forward pass with a 3-layer network using dropout.

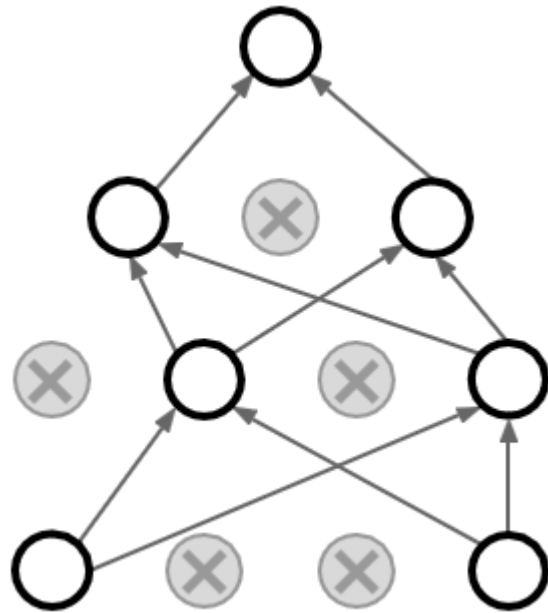


Regularization: Dropout

- How can this possibly be a good idea?
- Forces the network to have a redundant representation; Prevents co-adaptation of features



Regularization: Dropout



- How can this possibly be a good idea?
- Another interpretation:
- Dropout is training a large ensemble of models (that share parameters).
- Each binary mask is one model
- An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
- Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

- Dropout makes our output random!

$$y = f_W(x, z)$$

- + y : output (label).
- + x : input (image).
- + z : random mask.
- Want to “average out” the randomness at test-time

$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

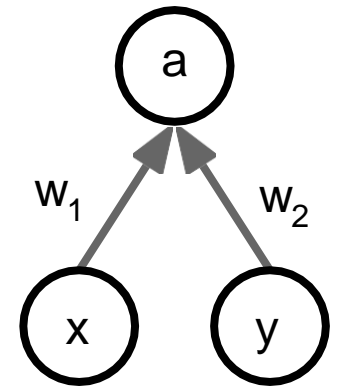
- But this integral seems hard ...

Dropout: Test time

- Want to approximate the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

- Consider a single neuron.



+ At test time we have: $E(a) = w_1x + w_2y$

+ During training we have: $E(a) = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(w_10 + w_2y) = \frac{1}{2}(w_1x + w_2y)$

- At test time, multiply by dropout probability.

Dropout: Test time

```
def predict(X):  
    # ensembled forward pass  
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations  
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations  
    out = np.dot(W3, H2) + b3
```

— At test time all neurons are active always

⇒ We must scale the activations so that for each neuron:
output at test time = expected output at training time.

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

test time is unchanged!

Regularization: A common pattern

- Training: Add some kind of randomness

$$y = f_W(x, z)$$

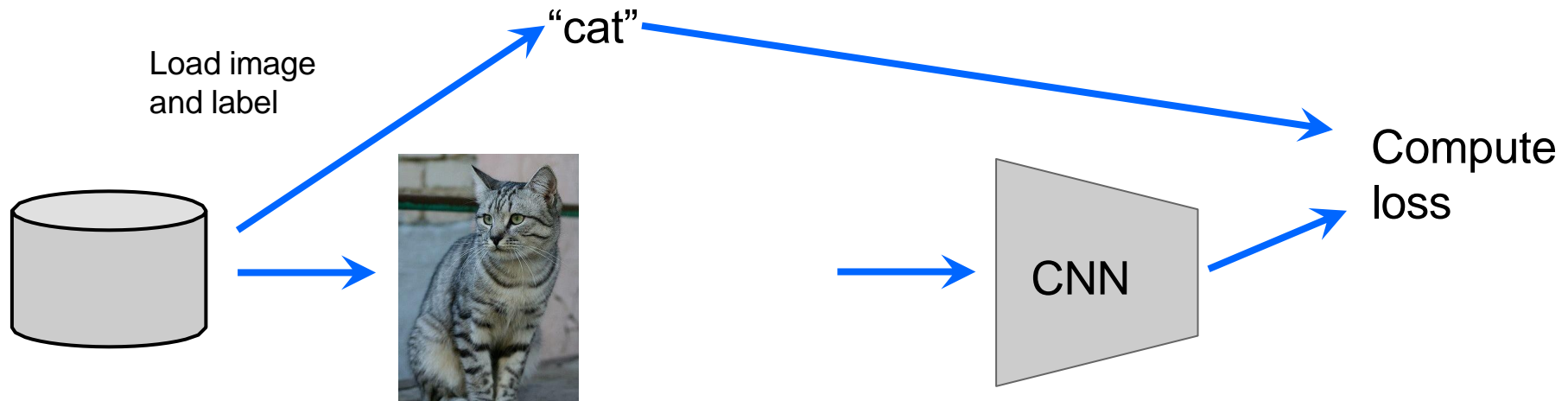
- Testing: Average out randomness
(sometimes approximate)

$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

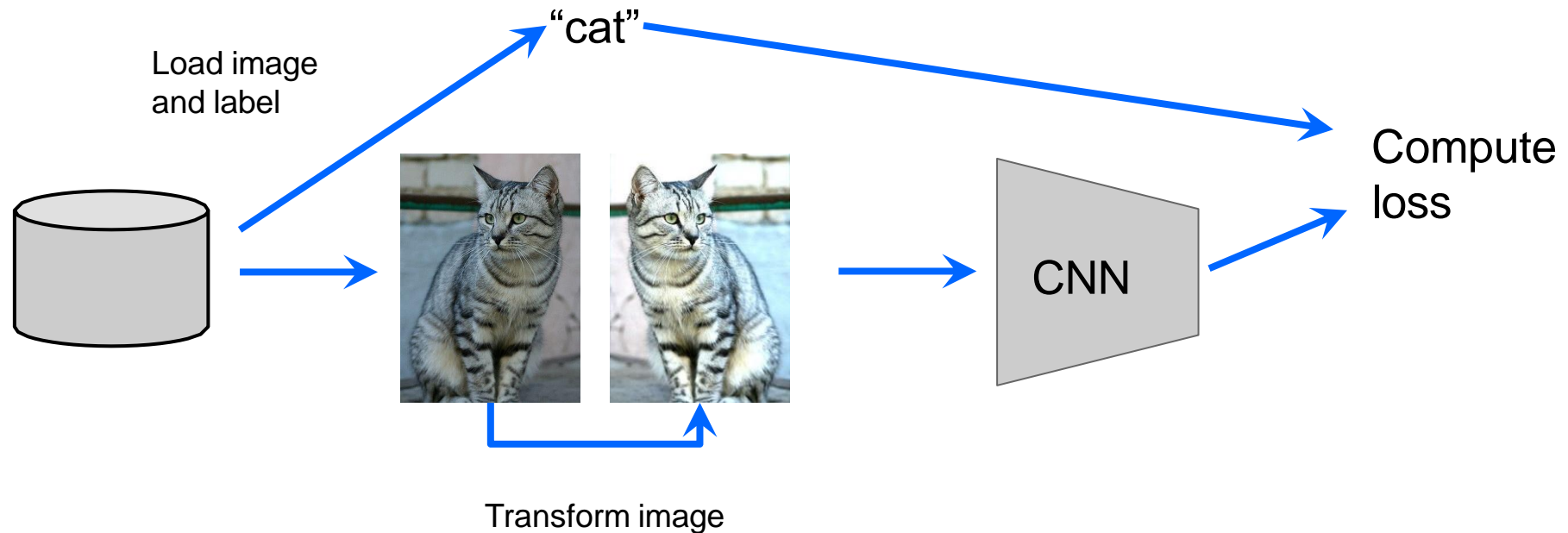
Regularization: A common pattern

- Training: Add some kind of randomness
 $y = f_W(x, z)$
- Example: Batch Normalization.
- Testing: Average out randomness (sometimes approximate)
 $y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$
- Training: Normalize using stats from random minibatches.
- Testing: Use fixed stats to normalize.

Regularization: Data Augmentation

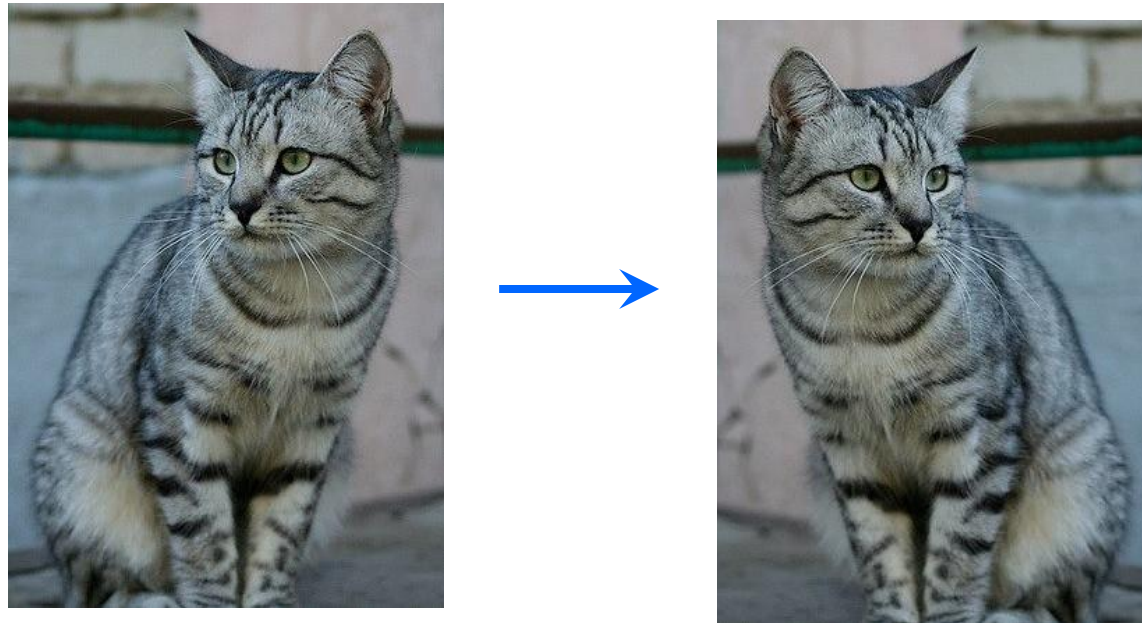


Regularization: Data Augmentation



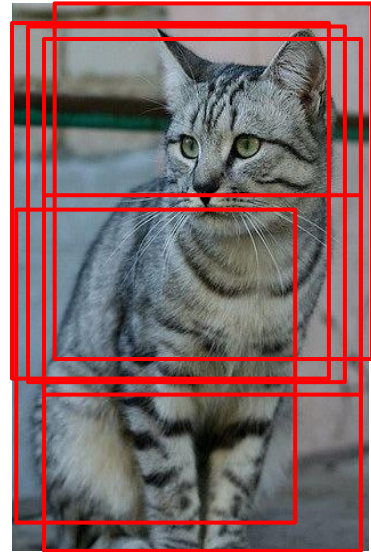
Data Augmentation

— Horizontal Flips



Data Augmentation

- Random crops and scales
 - + Training: sample random crops / scales
 - + ResNet:
 1. Pick random L in range $[256, 480]$
 2. Resize training image, short side = L
 3. Sample random 224×224 patch



Data Augmentation

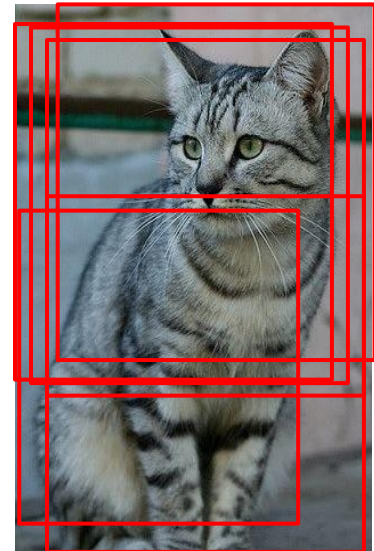
— Random crops and scales

+ ...

+ Testing: average a fixed set of crops

+ ResNet:

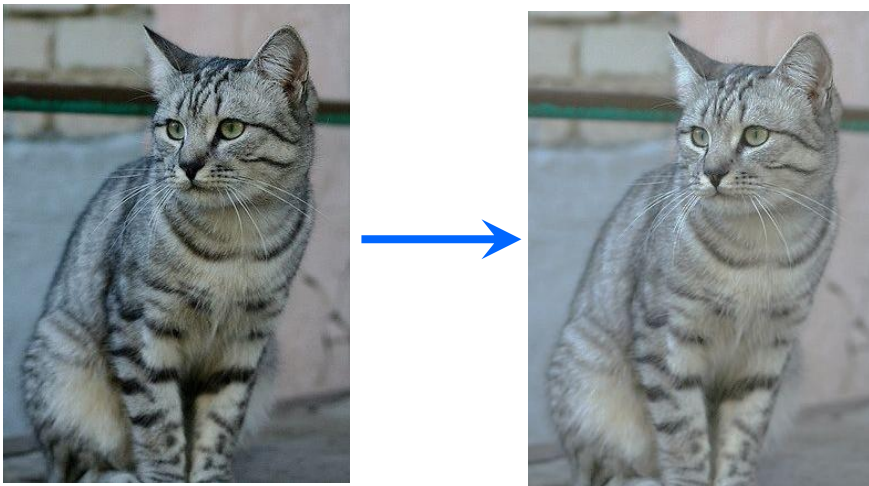
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224×224 crops: 4 corners + center, + flips.



Data Augmentation

— Color Jitter

+ Simple: Randomize contrast and brightness

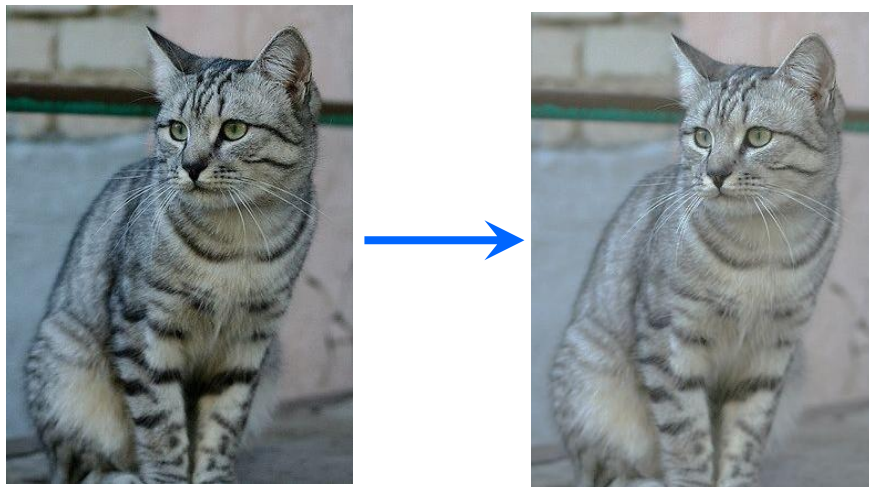


Krizhevsky et al. 2012

Data Augmentation

— Color Jitter

+ Simple: Randomize contrast and brightness



— More Complex:

1. Apply PCA to all [R, G, B] pixels in training set.
2. Sample a “color offset” along principal component directions.
3. Add offset to all pixels of a training image.

Krizhevsky et al. 2012

Data Augmentation

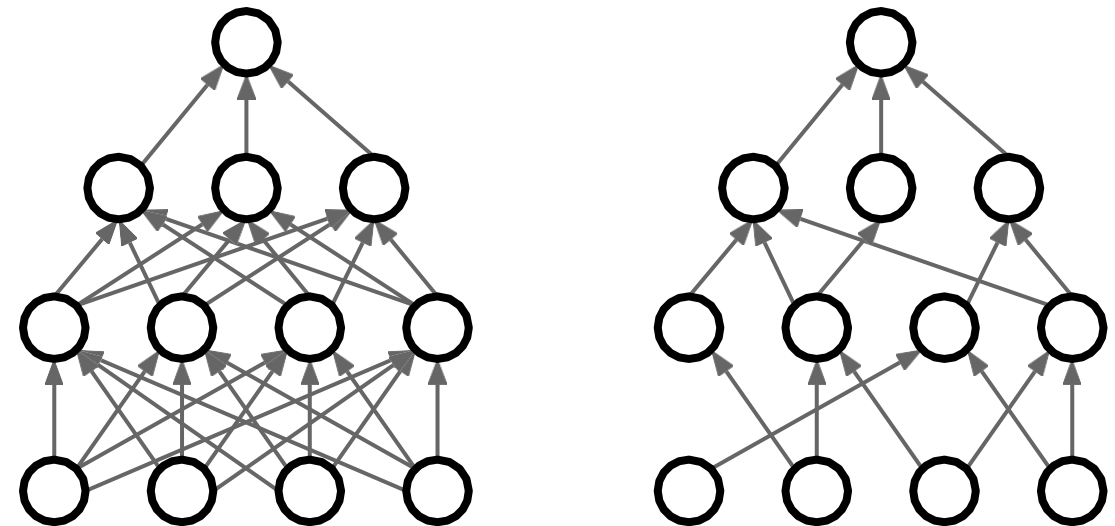
- Get creative for your problem!
- Random mix/combinations of:
 - + translation
 - + rotation
 - + stretching
 - + shearing,
 - + lens distortions, ... (go crazy)

Regularization: A common pattern

- Training: Add random noise.
- Testing: Marginalize over the noise.
- Examples:
 - + Dropout
 - + Batch Normalization
 - + Data Augmentation

Regularization: A common pattern

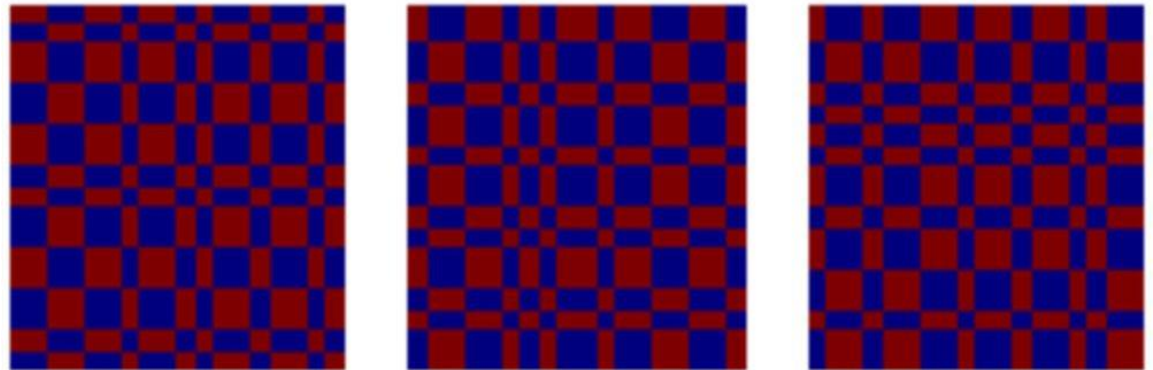
- Training: Add random noise.
- Testing: Marginalize over the noise.
- Examples:
 - + Dropout
 - + Batch Normalization
 - + Data Augmentation
 - + DropConnect



Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

Regularization: A common pattern

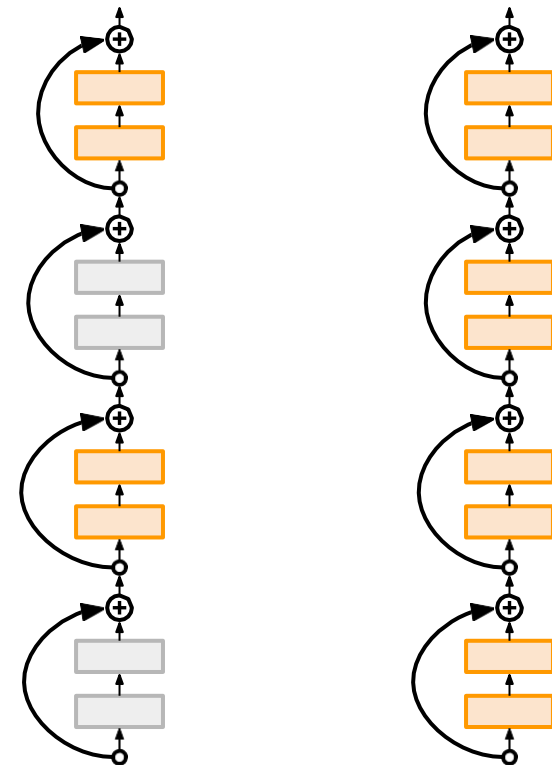
- Training: Add random noise.
- Testing: Marginalize over the noise.
- Examples:
 - + Dropout
 - + Batch Normalization
 - + Data Augmentation
 - + DropConnect
 - + Fractional Max Pooling



Graham, "Fractional Max Pooling", arXiv 2014

Regularization: A common pattern

- Training: Add random noise.
- Testing: Marginalize over the noise.
- Examples:
 - + Dropout
 - + Batch Normalization
 - + Data Augmentation
 - + DropConnect
 - + Fractional Max Pooling
 - + Stochastic Depth



Huang et al, “Deep Networks with Stochastic Depth”, ECCV 2016

TRANSFER LEARNING

Transfer learning

“You need a lot of a data
if you want to train/use CNNs”

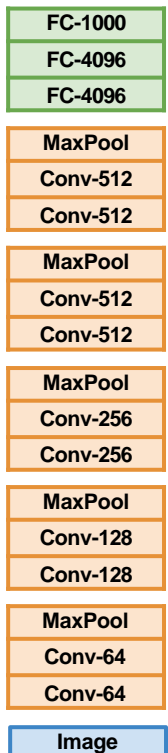
Transfer learning

“You need a lot of a data
if you want to train/use CNNs”

BUSTED

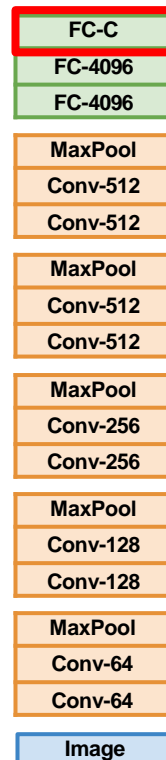
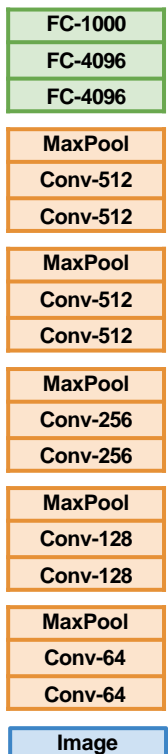
Transfer Learning with CNNs

1. Train on Imagenet



Transfer Learning with CNNs

1. Train on Imagenet
2. Small Dataset (C classes)

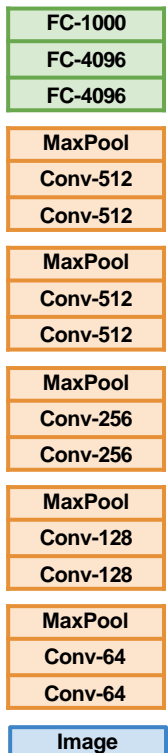


Reinitialize
this and train

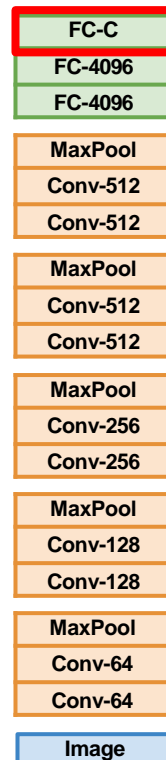
Freeze
these

Transfer Learning with CNNs

1. Train on Imagenet



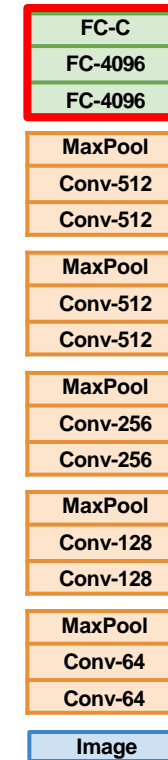
2. Small Dataset (C classes)



Reinitialize
this and train

Freeze
these

3. Bigger dataset



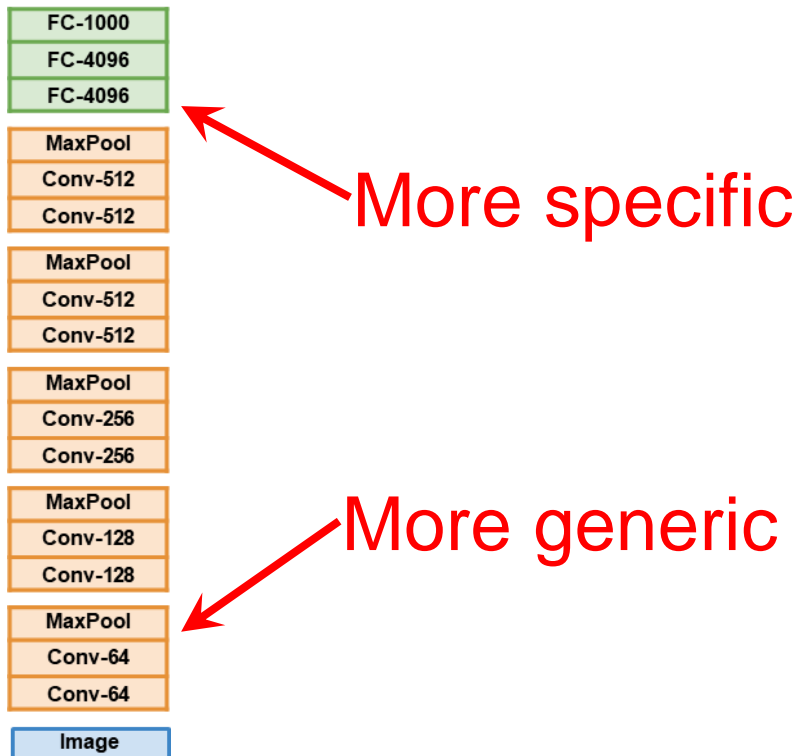
Train
these

With bigger dataset,
train more layers

Freeze
these

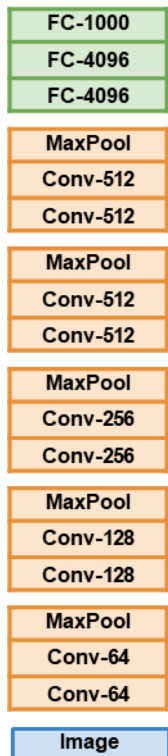
Lower learning rate
when finetuning;
1/10 of original LR is
good starting point

Transfer Learning with CNNs



	Very similar dataset	Very different dataset
Very little data	?	?
Quite a lot of data	?	?

Transfer Learning with CNNs

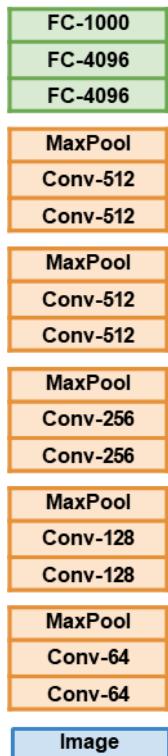


More specific

More generic

	Very similar dataset	Very different dataset
Very little data	Use Linear Classifier on top layer	?
Quite a lot of data	Finetune a few layers	?

Transfer Learning with CNNs



More specific

More generic

	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

Transfer learning with CNNs is pervasive...

Object Detection (Fast R-CNN)

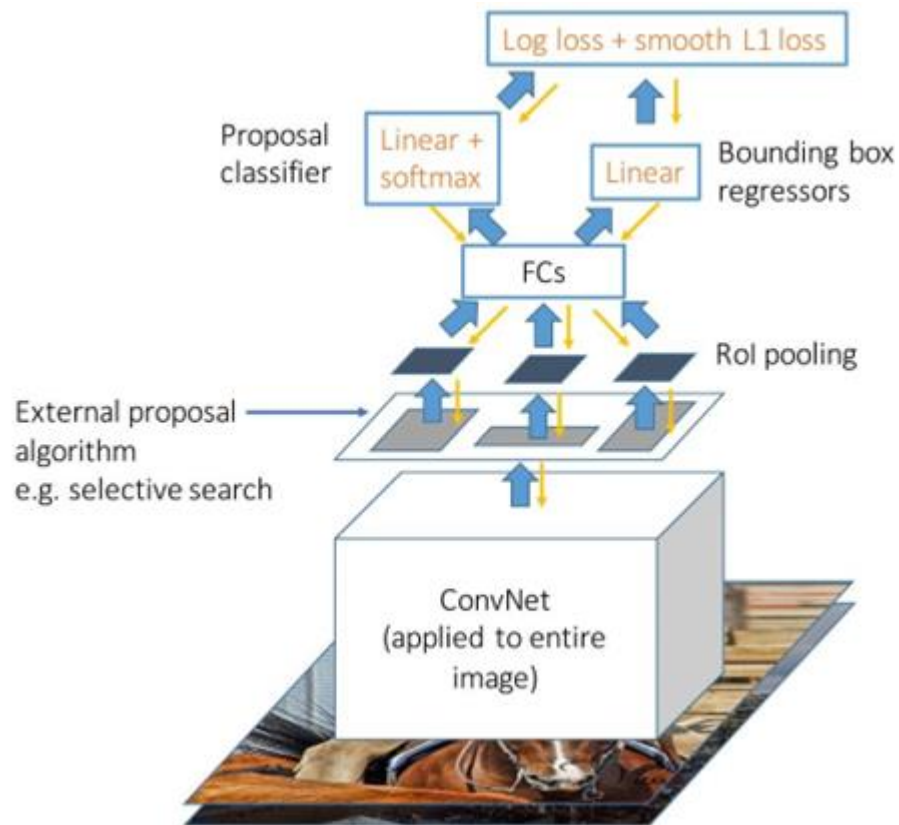
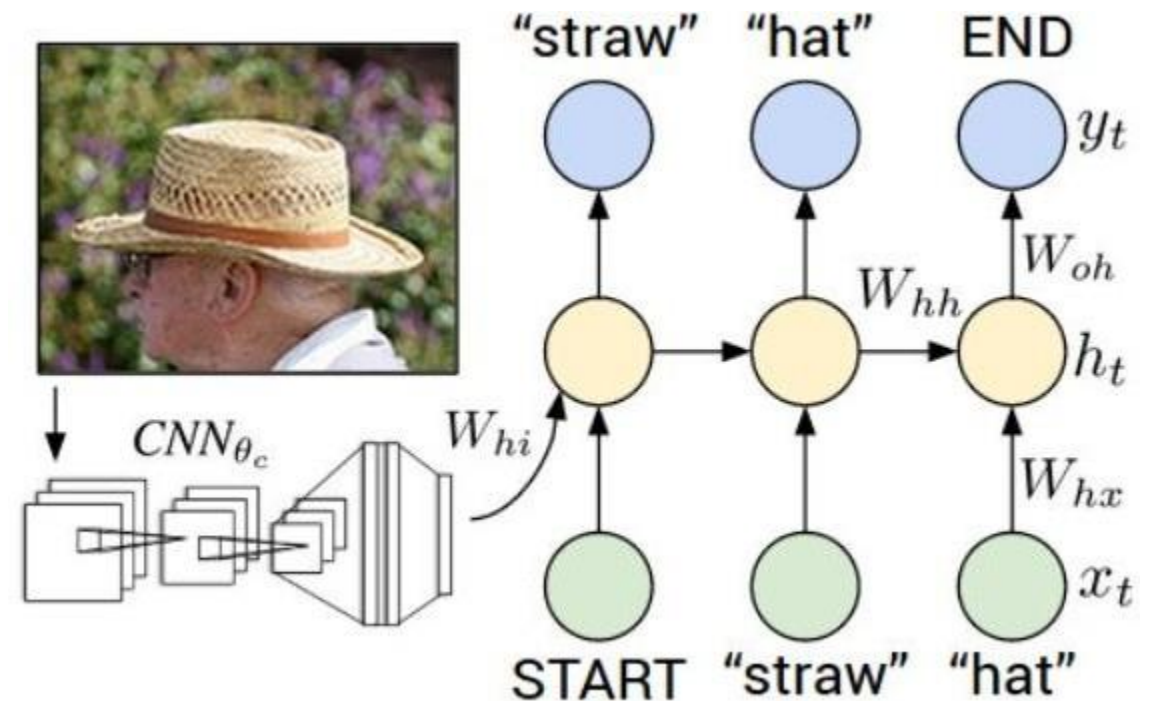


Image Captioning: CNN + RNN



Transfer learning with CNNs is pervasive...

Object Detection (Fast R-CNN)

CNN pretrained
on ImageNet

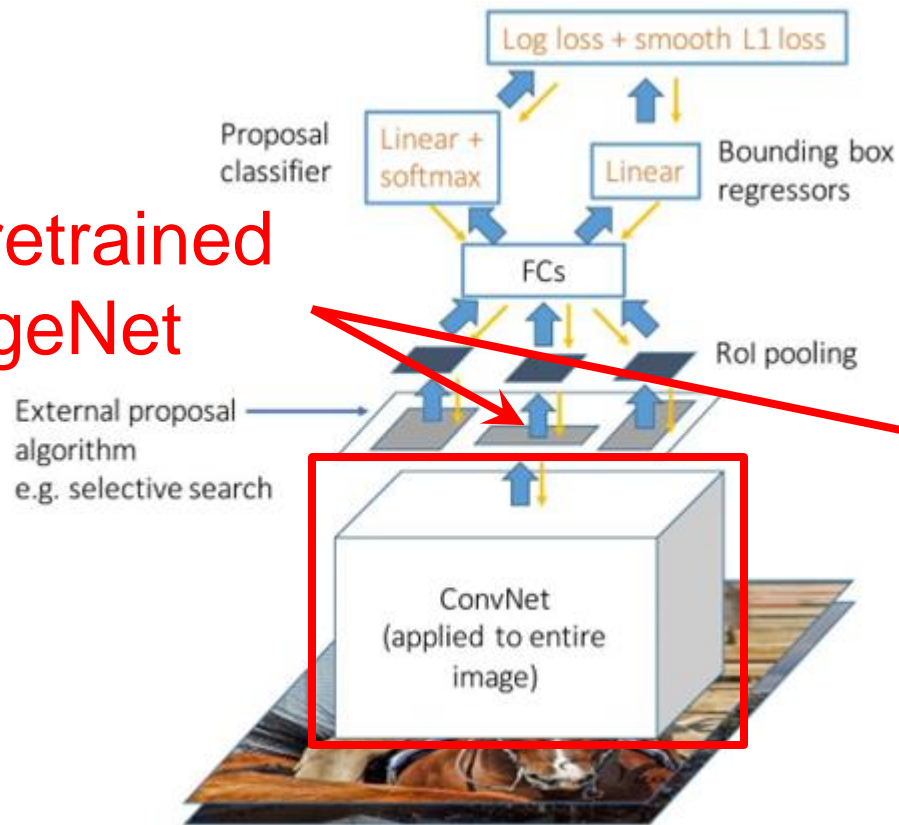
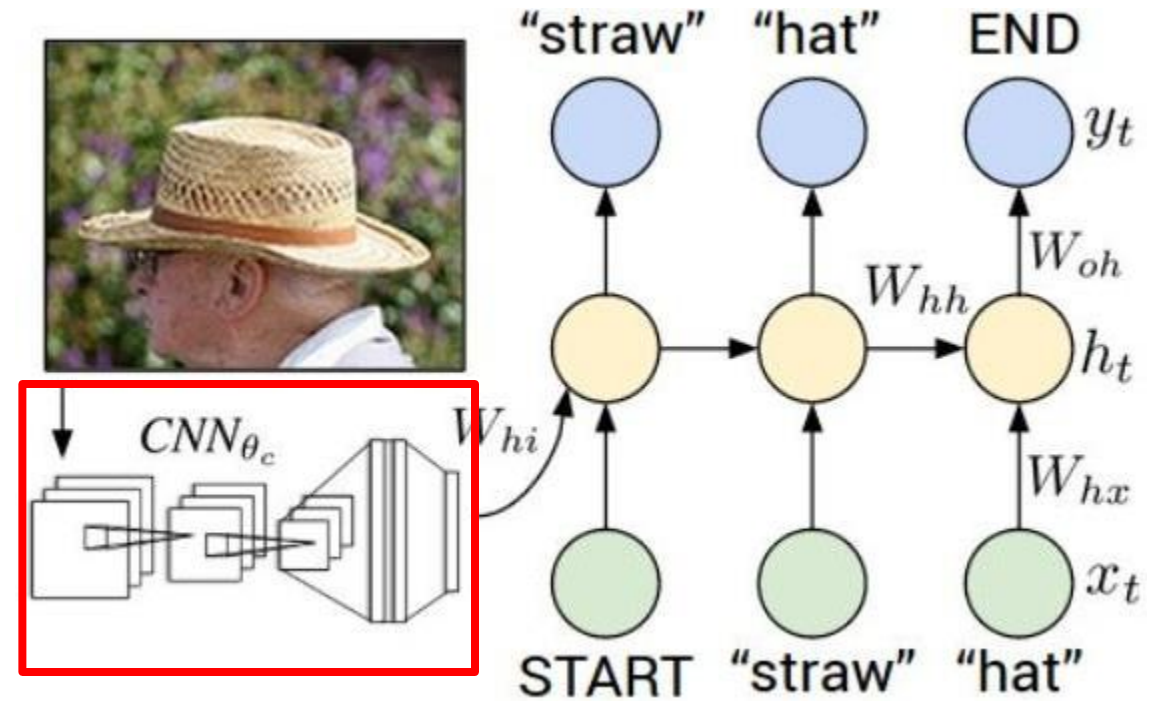


Image Captioning: CNN + RNN



Transfer learning with CNNs is pervasive...

Object Detection (Fast R-CNN)

CNN pretrained on ImageNet

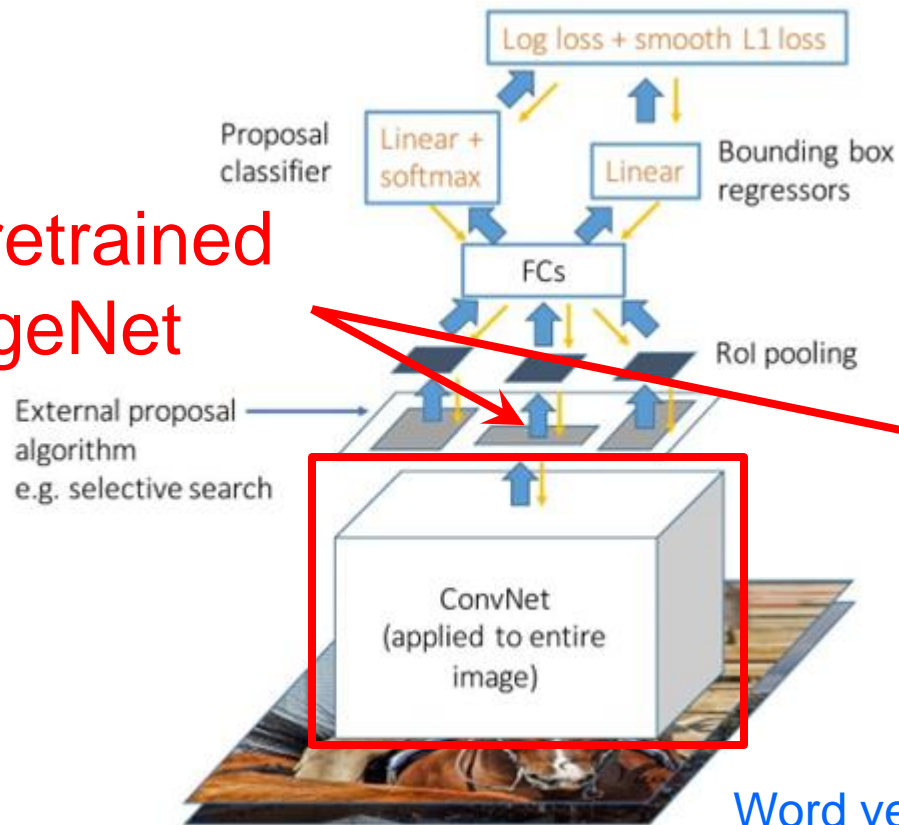
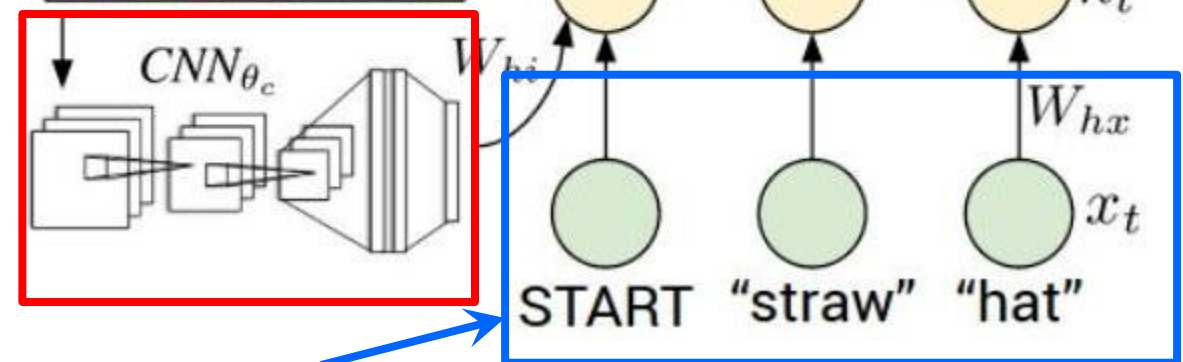


Image Captioning: CNN + RNN



Word vectors pretrained with word2vec

Takeaway for your projects and beyond:

- Have some dataset of interest but it has $< \sim 1\text{M}$ images?
 1. Find a very large dataset that has similar data, train a big ConvNet there
 2. Transfer learn to your dataset
 - + Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own
 - + Caffe: <https://github.com/BVLC/caffe/wiki/Model-Zoo>
 - + TensorFlow: <https://github.com/tensorflow/models>
 - + PyTorch: <https://github.com/pytorch/vision>

Summary

- Optimization
 - + Momentum, RMSProp, Adam, etc
- Regularization
 - + Dropout, etc
- Transfer learning
 - + Use this for your projects!

Next time: Deep Learning Software!

Chúc các bạn học tốt
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN TP.HCM

Nhóm UIT-Together
TS. Nguyễn Tấn Trần Minh Khang