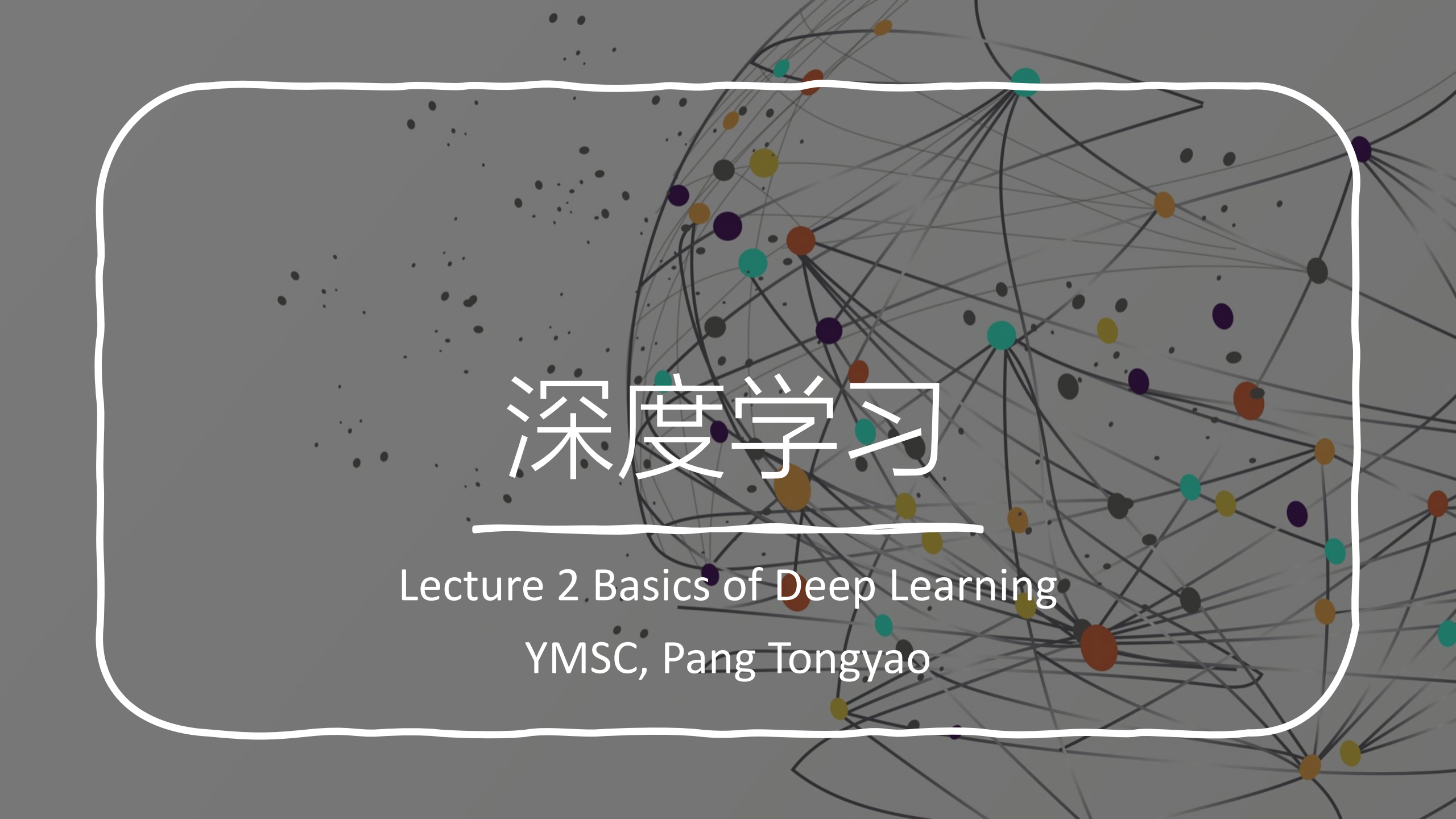


深度学习

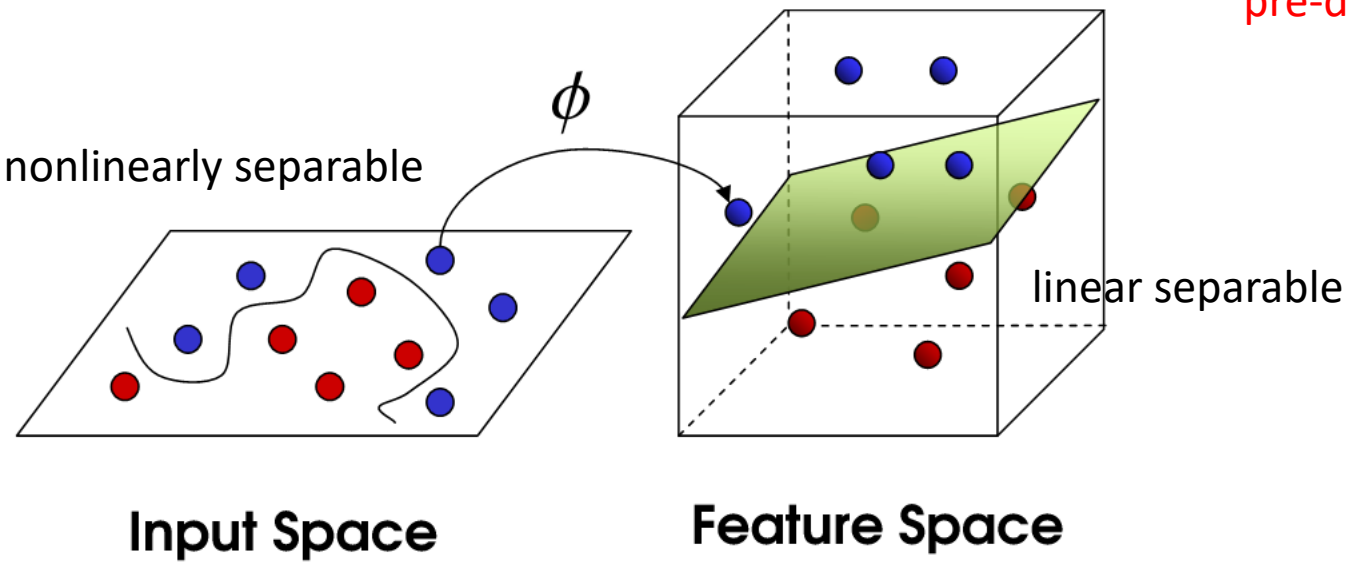
Lecture 2 Basics of Deep Learning

YMSC, Pang Tongyao

Recaps: Linear Models

- Hypothesis space: $\mathcal{H}_M = \left\{ f : f(x) = \sum_{j=0}^{M-1} w_j \phi_j(x) \right\}$

pre-defined feature mapping



Feature extraction is critical but challenging!

Deep Learning

- Manual feature extraction requires domain-specific expertise.
- Deep learning makes it possible to learn good feature mapping (or data representation) automatically from raw data using neural networks.

Deep learning

Yann LeCun ✉, Yoshua Bengio & Geoffrey Hinton

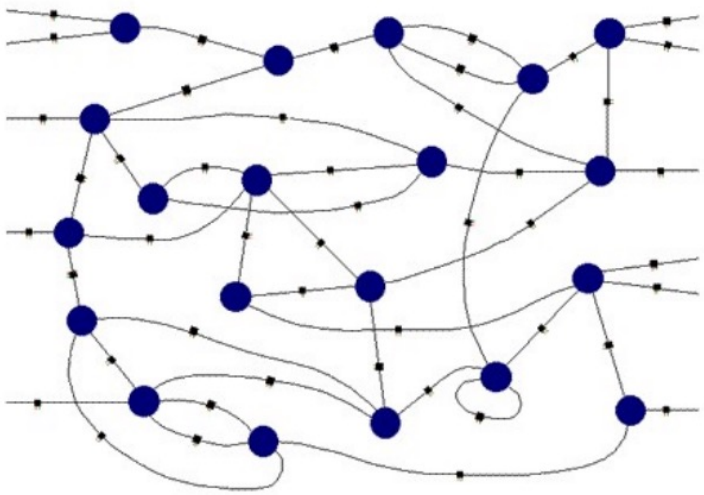
Nature **521**, 436–444 (2015) | Cite this article

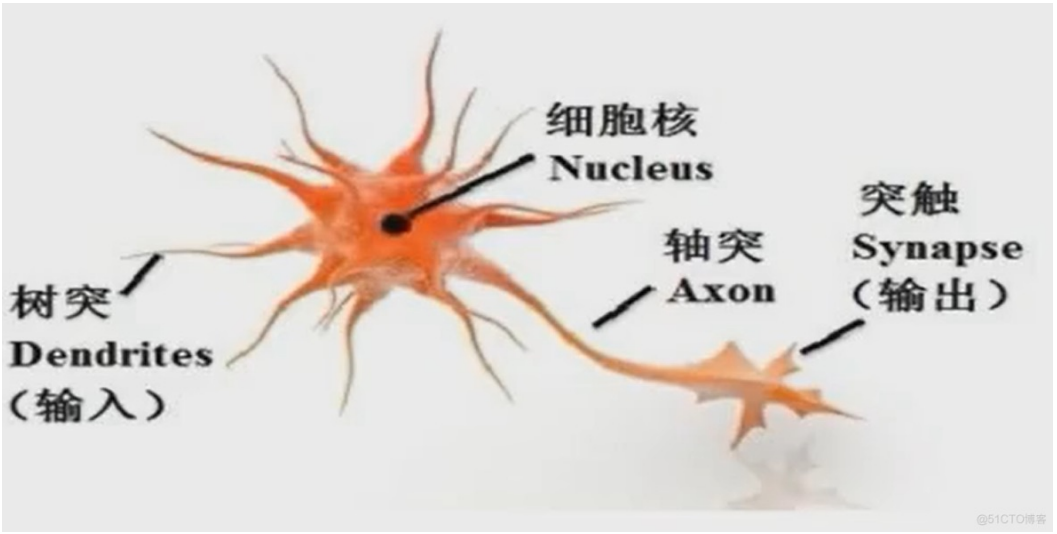
Abstract

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone light on sequential data such as text and speech.

Brain: Interconnected Neurons

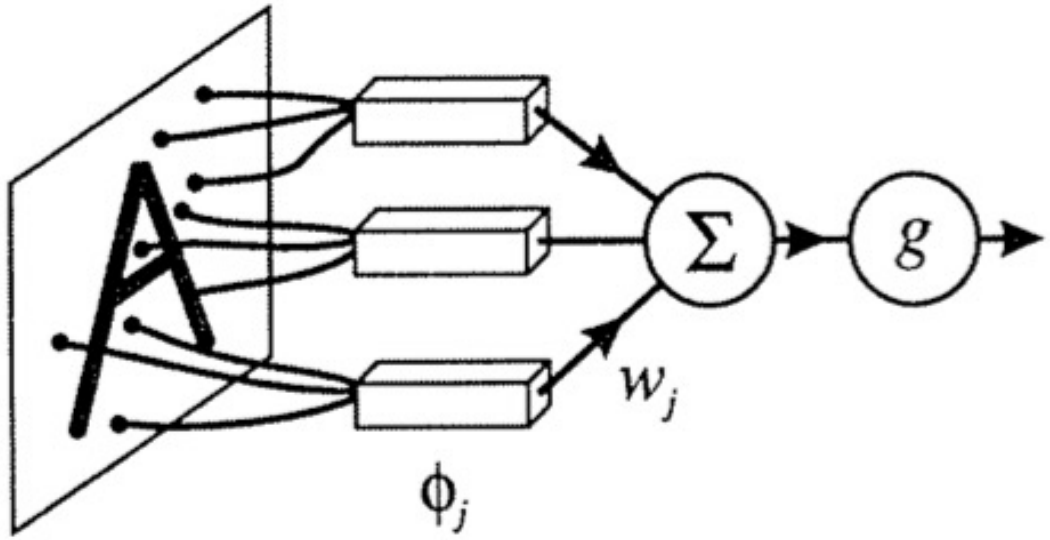
- The brain is a network of neurons
- The processing/capacity of the brain is a function of these connections
- All world knowledge is stored in the connections





Mathematical Model for Neurons

- Frank Rosenblatt's Perceptron,1958



$$f(x) = \text{sign}(\sum_{j=0}^{M-1} w_j \phi_j(x) + b)$$

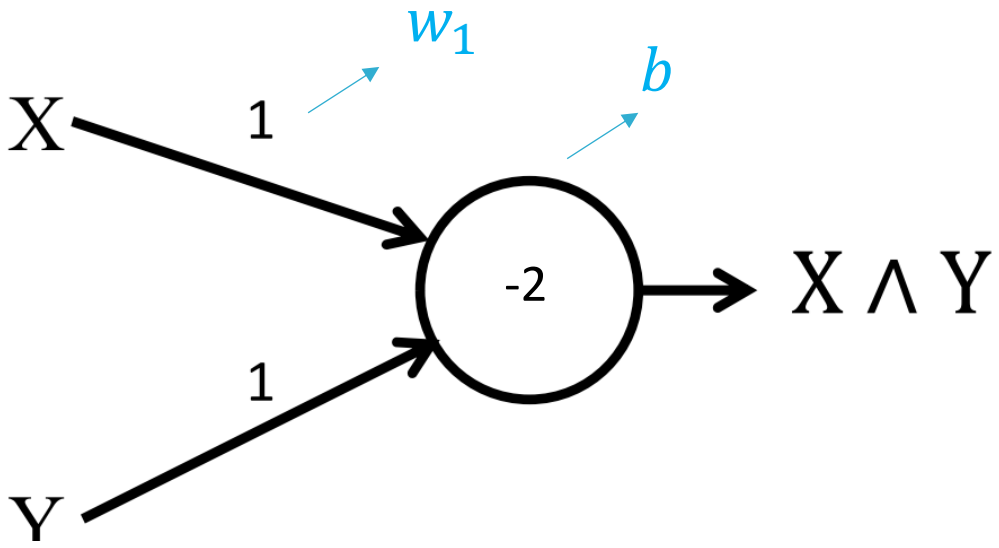
Also provided a learning algorithm

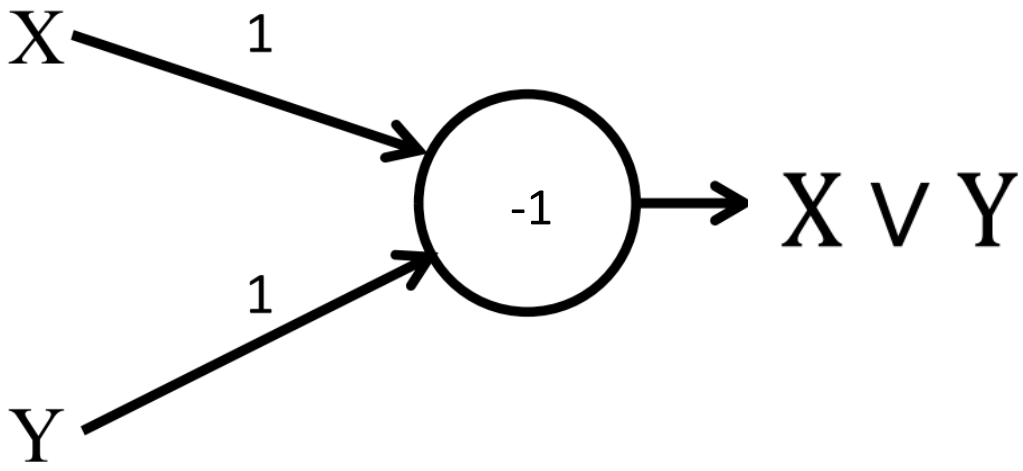
$$w_j = w_j + \eta\big(y - f(x)\big)\phi_j(x)$$

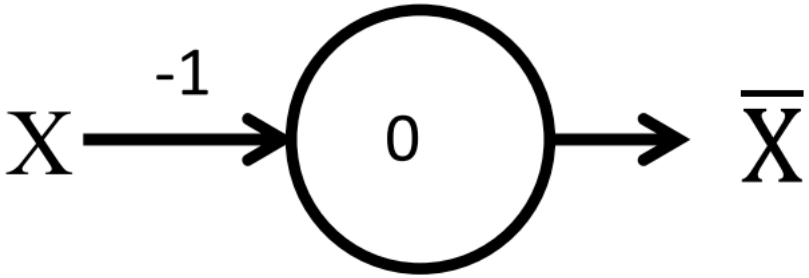
Gradient Descent:

- When $\sum w_j \phi_j > 0, f(x) = 1, y = -1,$ $w_j = w_j - 2\eta\phi_j(x).$
- When $\sum w_j \phi_j < 0, f(x) = -1, y = 1,$ $w_j = w_j + 2\eta\phi_j(x).$

Single Layer is not enough

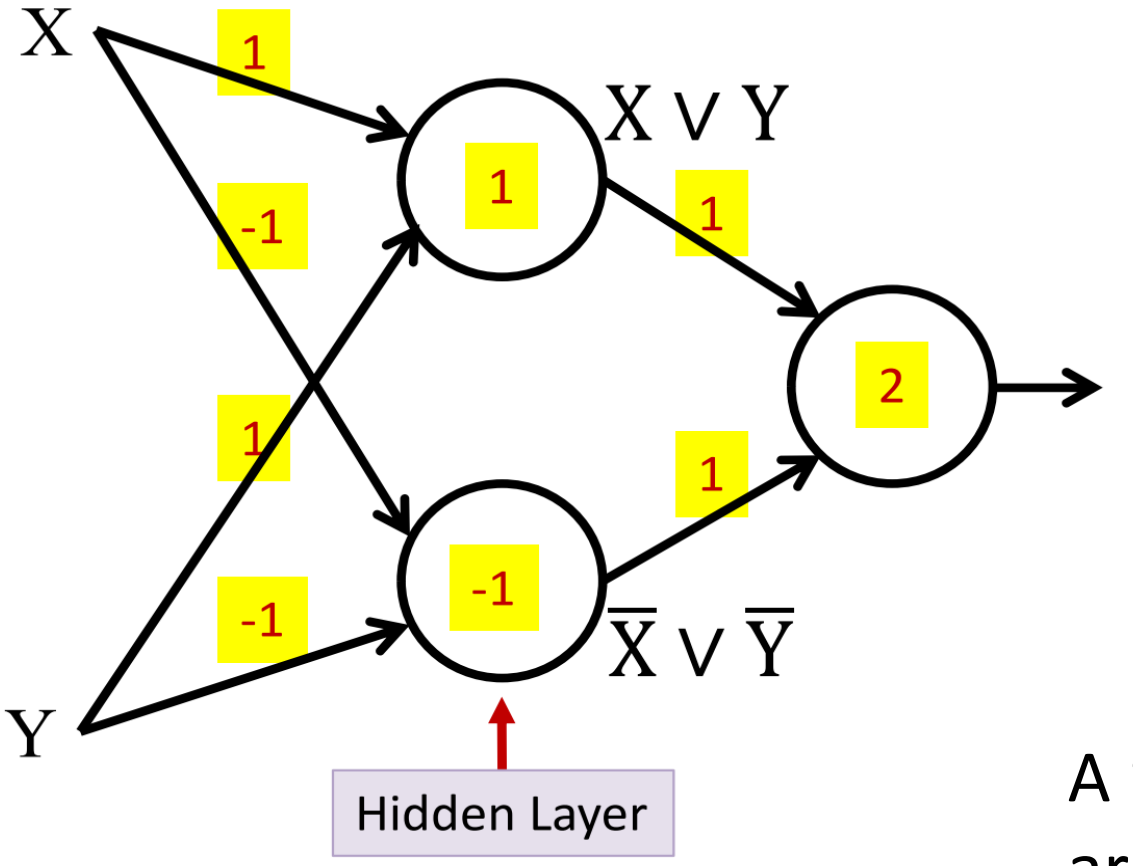


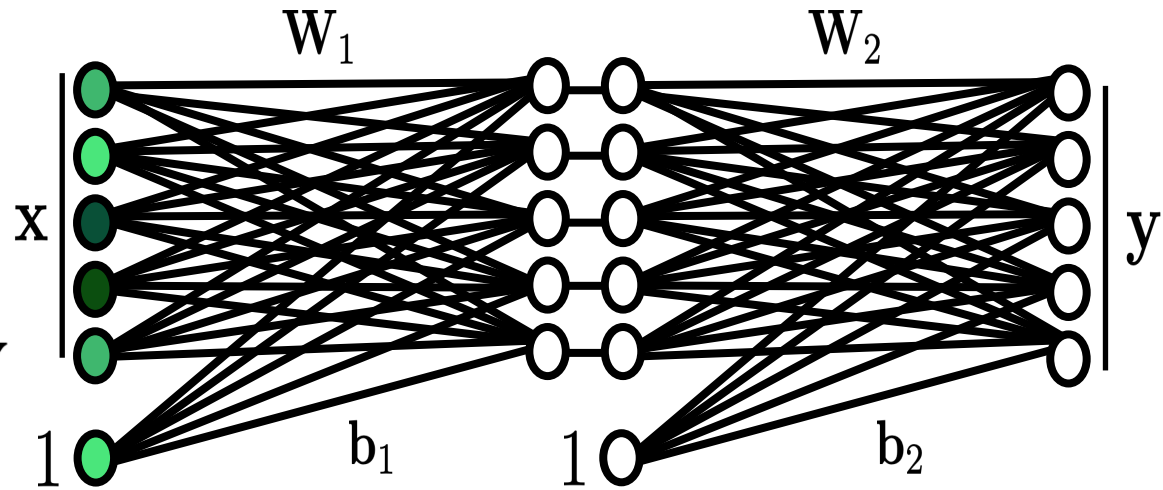




But cannot express XOR!

Multi-layer Perceptron (MLP)



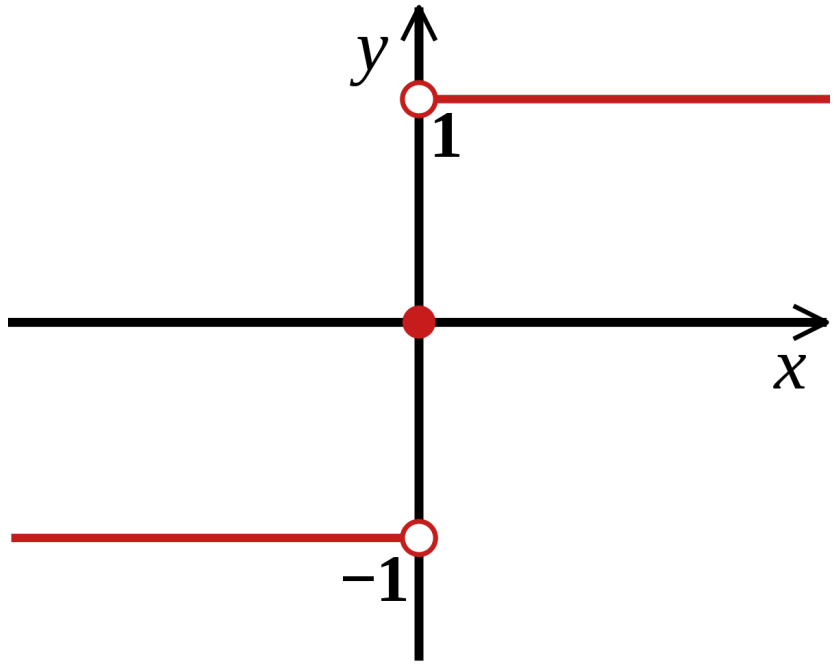


$$\mathbf{h} = g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \qquad \mathbf{y} = g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)$$

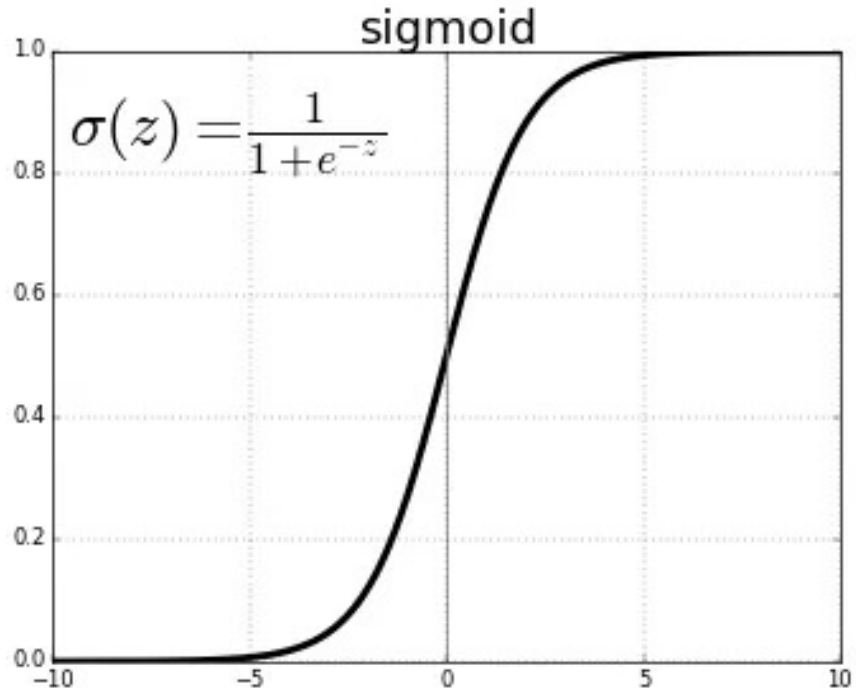
A “multi-layer” perceptron can compose arbitrarily complicated Boolean functions!

The perceptron with real inputs&outputs

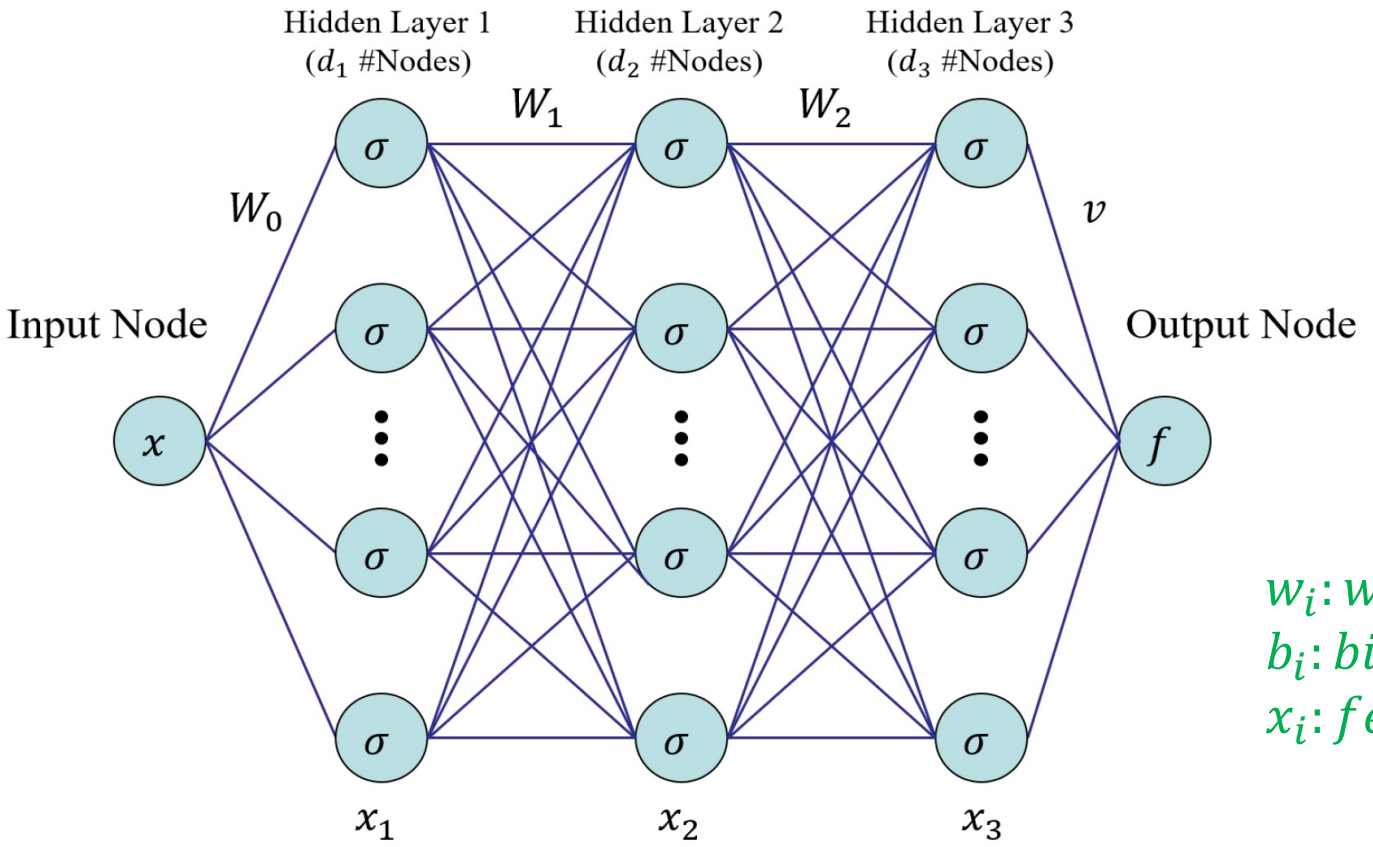
- Smooth activation function



Smoothing



MLP on Reals



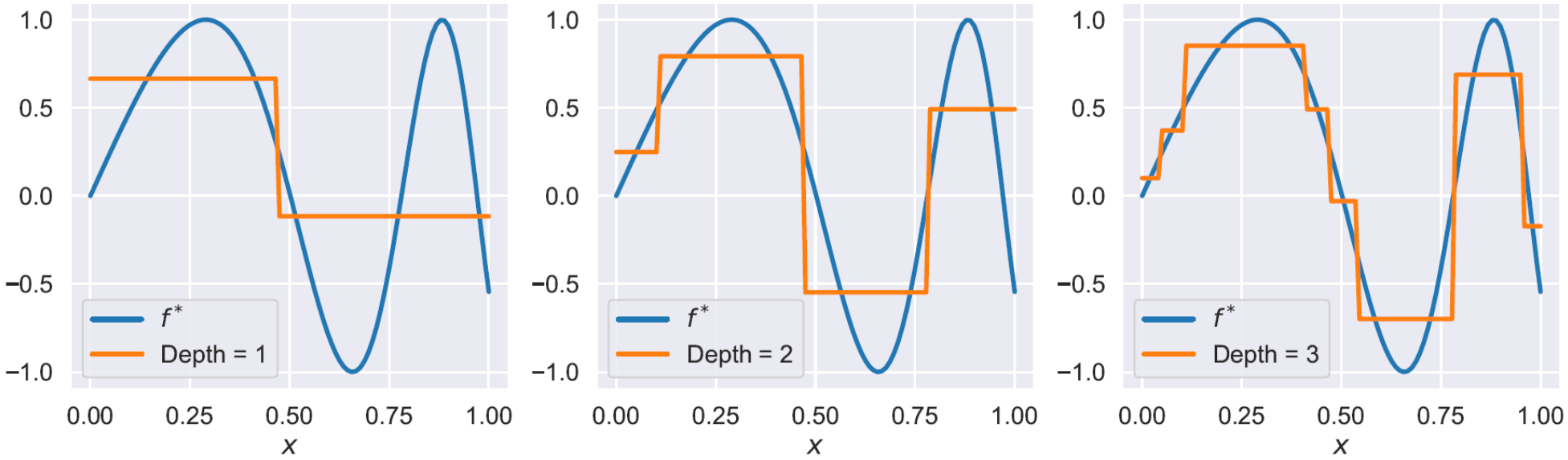
$$x_L = \sigma(W_{L-1}x_{L-1} + b_{L-1}),$$
$$x_{L-1} = \sigma(W_{L-2}x_{L-2} + b_{L-2}),$$
$$\vdots$$
$$x_1 = \sigma(W_0x_0 + +b_0)$$

w_i: *weights*
b_i: *bias*
x_i: *features*

σ: *nonlinear activation function*

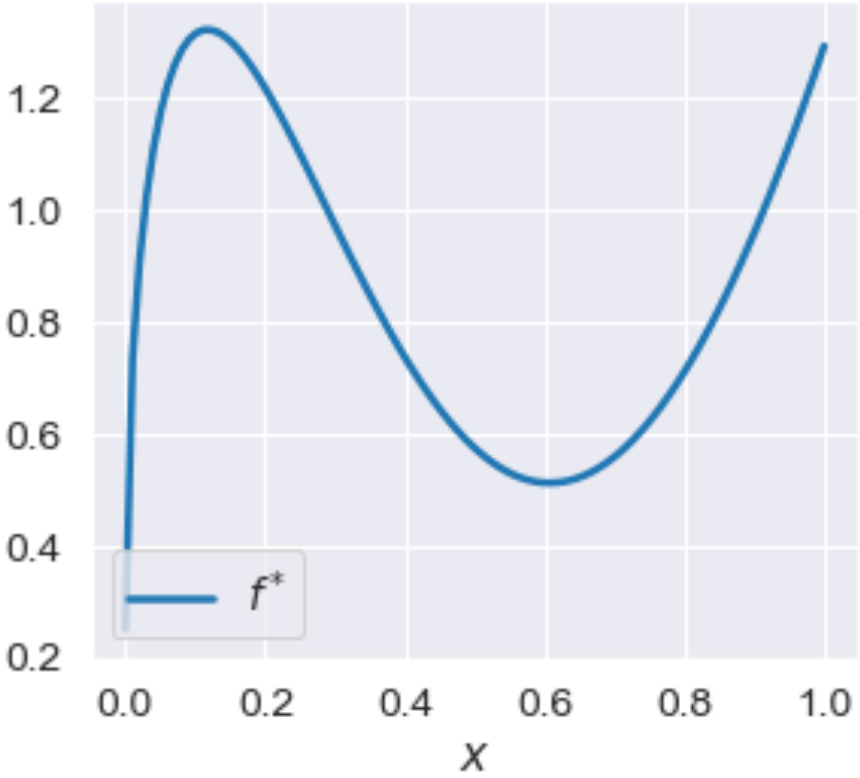
MLP for Function Approximation

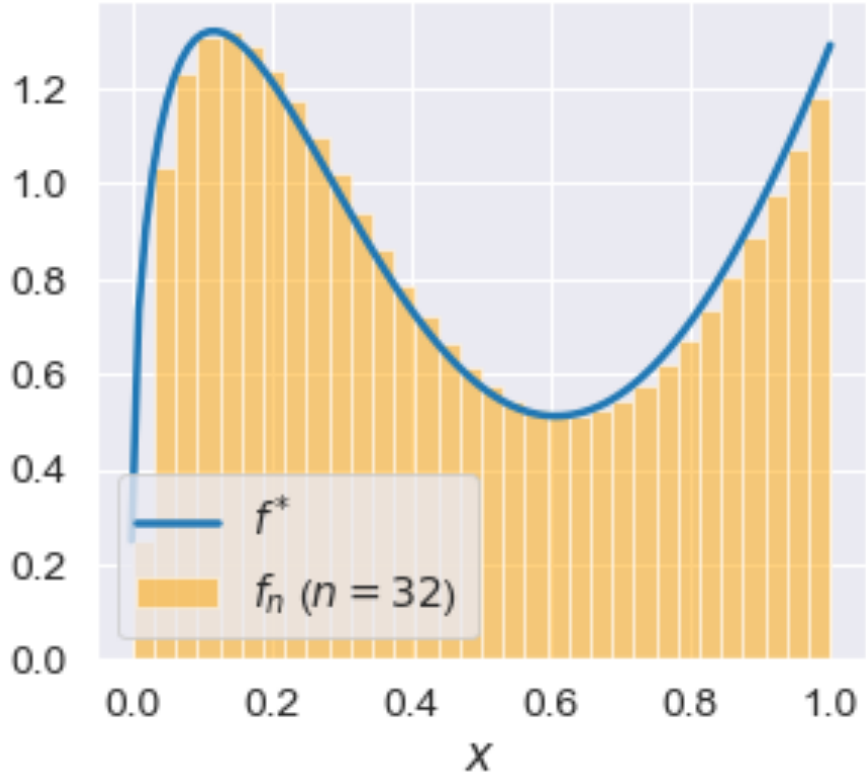
- Nonlinear approximation: piecewise approximation



MLP for Function Approximation

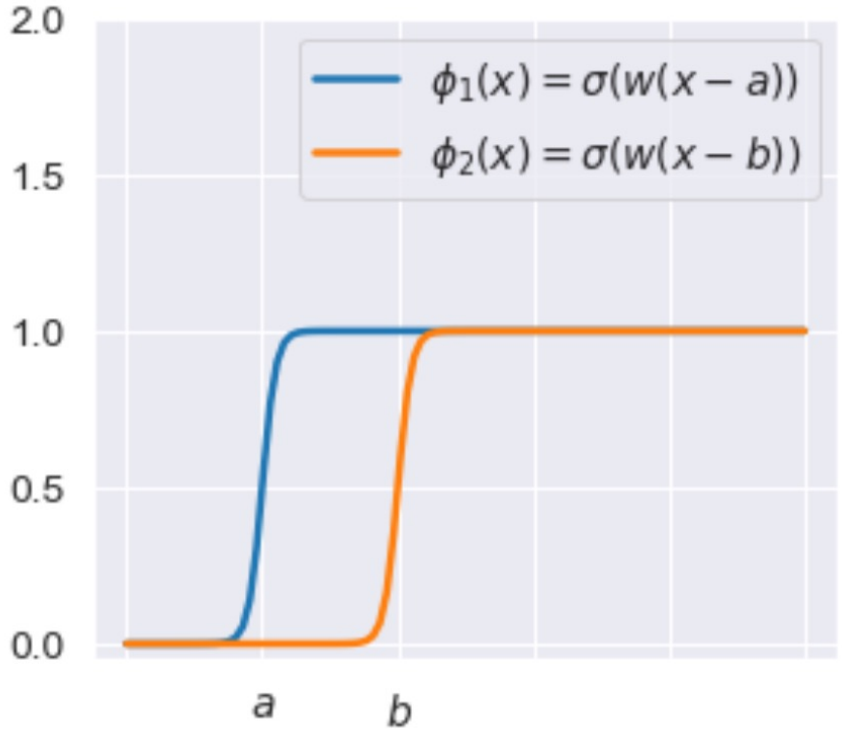
- Step function approximation

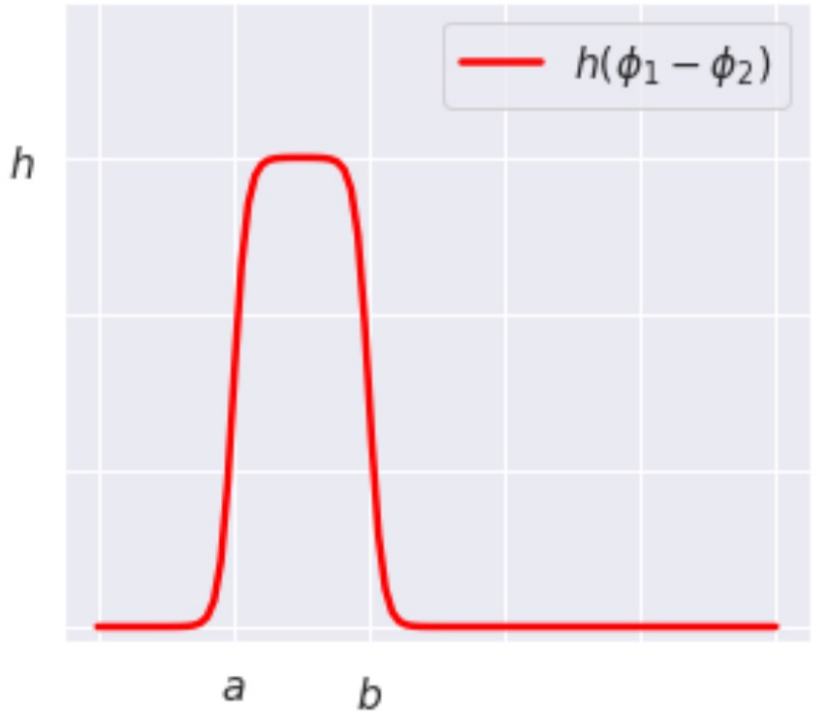




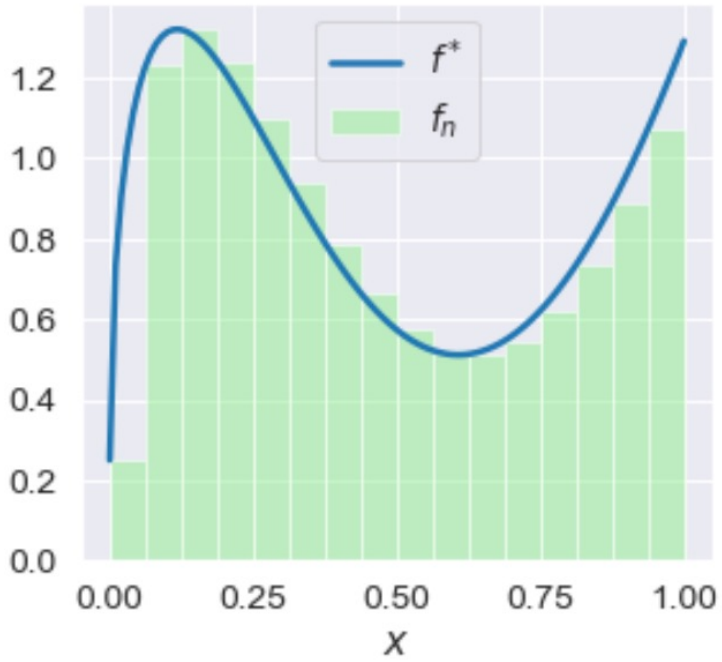
MLP for Function Approximation

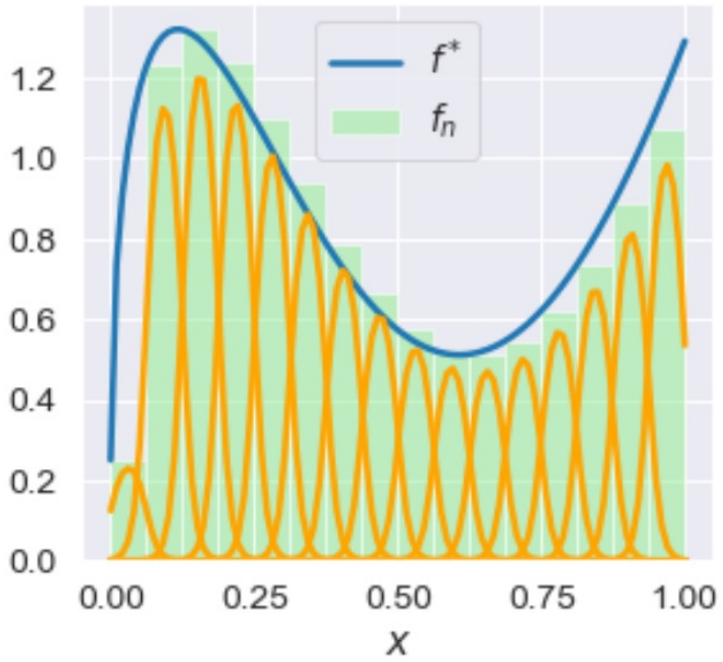
- Construct step functions using neural networks

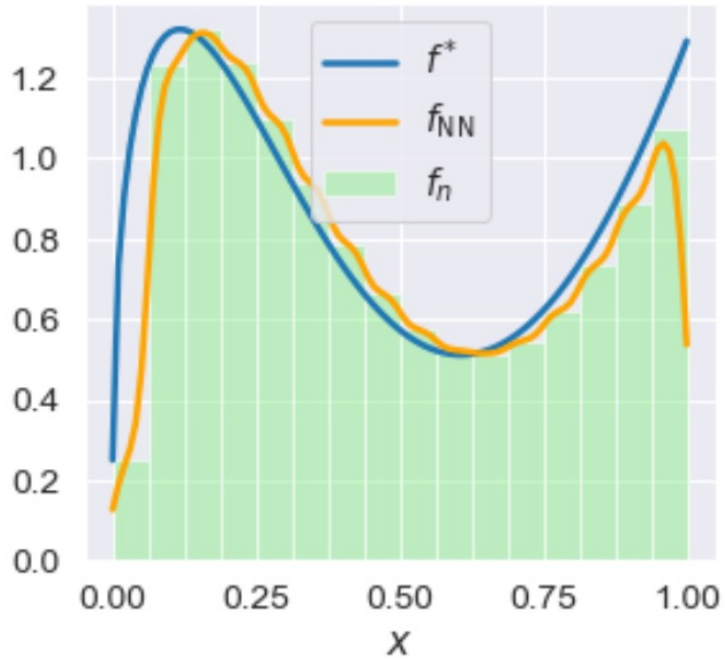




MLP for Function Approximation







Curse of Dimensionality

- The number of partitions grows exponentially with the dimension! If Each partition is approximated using fixed neurons, the number of neurons required also grows exponentially!
  - $d = 1$: $1/\epsilon$ partitions
  - $d = 2$: $1/\epsilon^2$ partitions
  - General d: $1/\epsilon^d$ partitions
- Can neural networks overcome the curse of dimensionality?

Deep Neural Network

- Deep means a lot! (function composition)

Theorem 1.1. *Given* $f \in C([0,1]^d)$, *for any* $L \in \mathbb{N}^+$, $N \in \mathbb{N}^+$, *and* $p \in [1,\infty]$, *there exists a function* ϕ *implemented by a ReLU FNN with width* $C_1 \max\{d\lfloor N^{1/d}\rfloor, N+1\}$ *and depth* $12L + C_2$ *such that*

$$\|f - \phi\|_{L^p([0,1]^d)} \le 19\sqrt{d}\,\omega_f\left(N^{-2/d}L^{-2/d}\right),$$

where $C_1 = 12$ *and* $C_2 = 14$ *if* $p \in [1,\infty)$; $C_1 = 3^{d+3}$ *and* $C_2 = 14 + 2d$ *if* $p = \infty$.

N: $width$, L: $depth$

$NL \sim (1/\sqrt{\epsilon})^d$

computational complexity: $O(N^2L)$

Reference: Deep Network Approximation Characterized by Number of Neurons

Deep Neural Network

Lemma 3.5. *For any $L \in \mathbb{N}^+$, there exists a function ϕ in*

$$\mathcal{NN}(\#\text{input} = 2;\ \text{width} \le 7;\ \text{depth} \le 2L+1;\ \#\text{output} = 1)$$

such that, for any $\theta_1, \theta_2, \cdots, \theta_L \in \{0,1\}$, we have

$$\phi(\text{bin}\,0.\theta_1\theta_2\cdots\theta_L,\ \ell) = \sum_{j=1}^{\ell} \theta_j, \quad \text{for } \ell = 1, 2, \cdots, L.$$

← Bit extraction

Lemma 3.6. *For any $N, L \in \mathbb{N}^+$, any $\theta_{m,\ell} \in \{0,1\}$ for $m = 0, 1, \cdots, M-1$ and $\ell = 0, 1, \cdots, L-1$, where $M = N^2L$, there exists a function ϕ implemented by a ReLU FNN with width $4N+3$ and depth $3L+3$ such that*

$$\phi(m, \ell) = \sum_{j=0}^{\ell} \theta_{m,j}, \quad \text{for } m = 0, 1, \cdots, M-1 \text{ and } \ell = 0, 1, \cdots, L-1.$$

← A neural network with width O(N) and depth O(L) can create N^2L^2 breakpoints.

Reference: Deep Network Approximation Characterized by Number of Neurons

Training

Back Propagation

- Gradient Descent

$$J(\theta) = \sum_{i=1}^{N} L(f_\theta(x_i), y_i)$$

$(Empirical\ loss)$

$$\theta^{t+1} = \theta^t - \eta_t \left. \frac{\partial J(\theta)}{\partial \theta} \right|_{\theta=\theta^t}$$

Letter | Published: 09 October 1986

Learning representations by back-propagating errors

David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams

Nature **323**, 533–536 (1986) | Cite this article

191k Accesses | **18k** Citations | **702** Altmetric | Metrics

Before their success, the AI winter has lasted for decades.
Making your idea work is important!

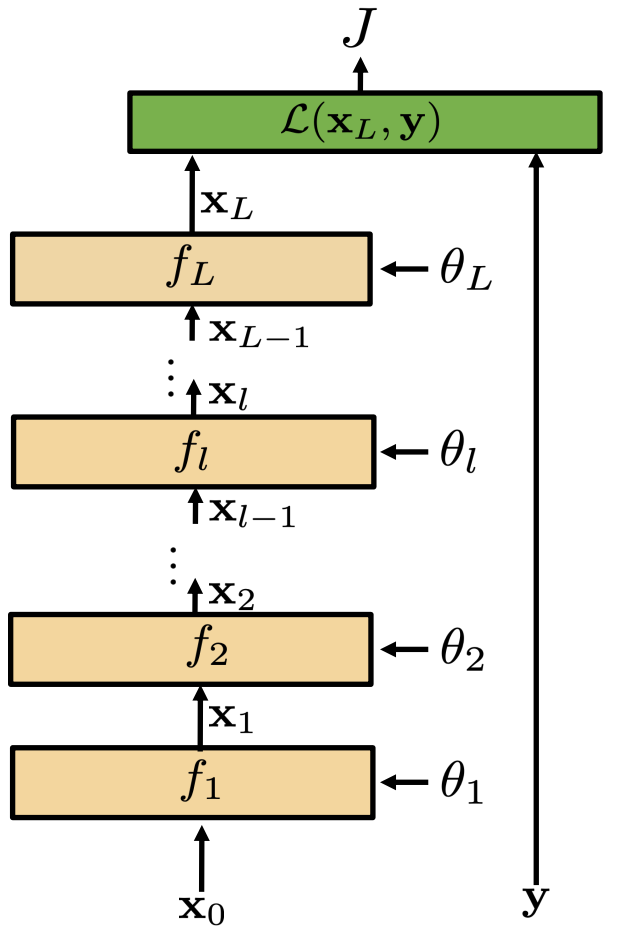
Back Propagation

- MLP

$$x_L = f_{\theta_L}(x_{L-1}),$$
$$x_{L-1} = f_{\theta_{L-1}}(x_{L-2}),$$
$$\vdots$$
$$x_1 = f_{\theta_1}(x_0)$$

- Chain Rule

$$\frac{\partial \mathcal{J}}{\partial \theta_L} = \frac{\partial \mathcal{J}}{\partial x_L}\frac{\partial x_L}{\partial \theta_L}$$



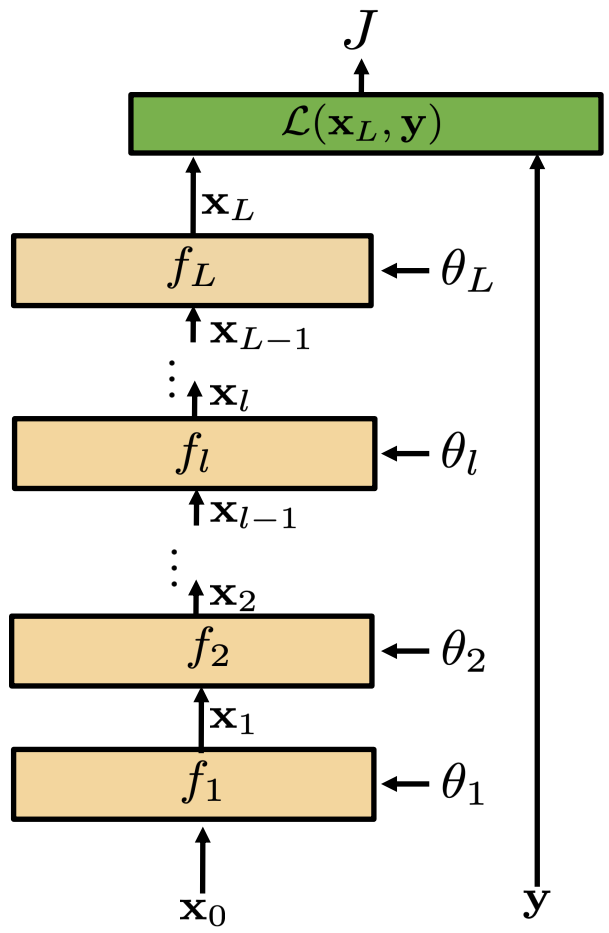
$$\frac{\partial \mathcal{J}}{\partial \theta_{L-1}} = \frac{\partial \mathcal{J}}{\partial x_L}\frac{\partial x_L}{\partial x_{L-1}}\frac{\partial x_{L-1}}{\partial \theta_{L-1}}$$

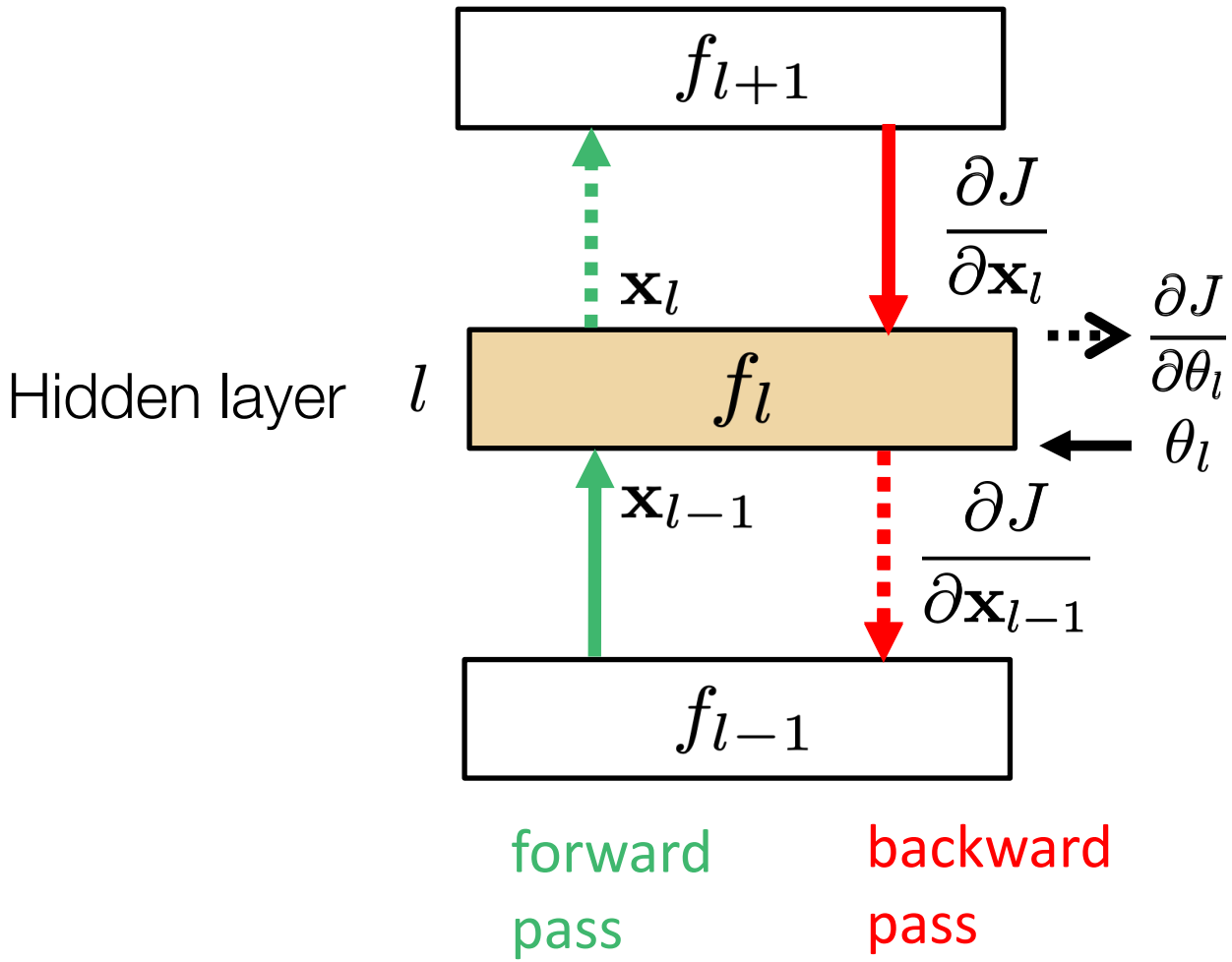
$$\vdots$$

$$\frac{\partial \mathcal{J}}{\partial \theta_2} = \frac{\partial \mathcal{J}}{\partial x_L}\frac{\partial x_L}{\partial x_{L-1}}\cdots\frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial \theta_2}$$

$$\frac{\partial \mathcal{J}}{\partial \theta_1} = \frac{\partial \mathcal{J}}{\partial x_L}\frac{\partial x_L}{\partial x_{L-1}}\cdots\frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial x_1}\frac{\partial x_1}{\partial \theta_1}$$

The red terms are used repeatedly. So they are restored.

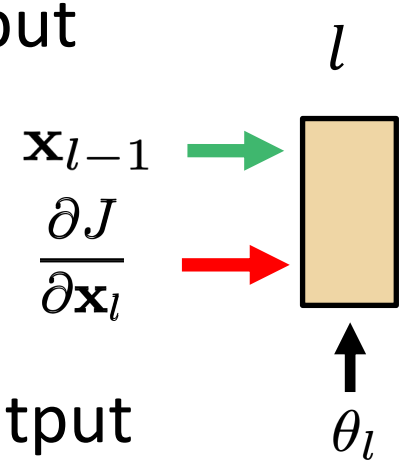




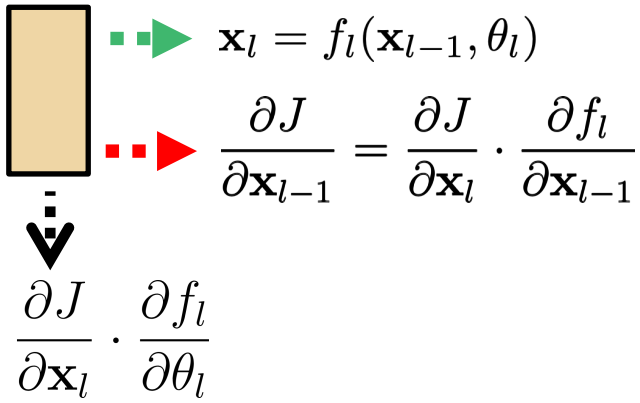
--Sara Beery, MIT

Training Process

- Input



- Output



- Computation

$$f_l(x_{l-1}, \theta_l), \frac{\partial f_l}{\partial x_{l-1}}, \frac{\partial f_l}{\partial \theta_l}$$

Back Propagation

- Forward Process

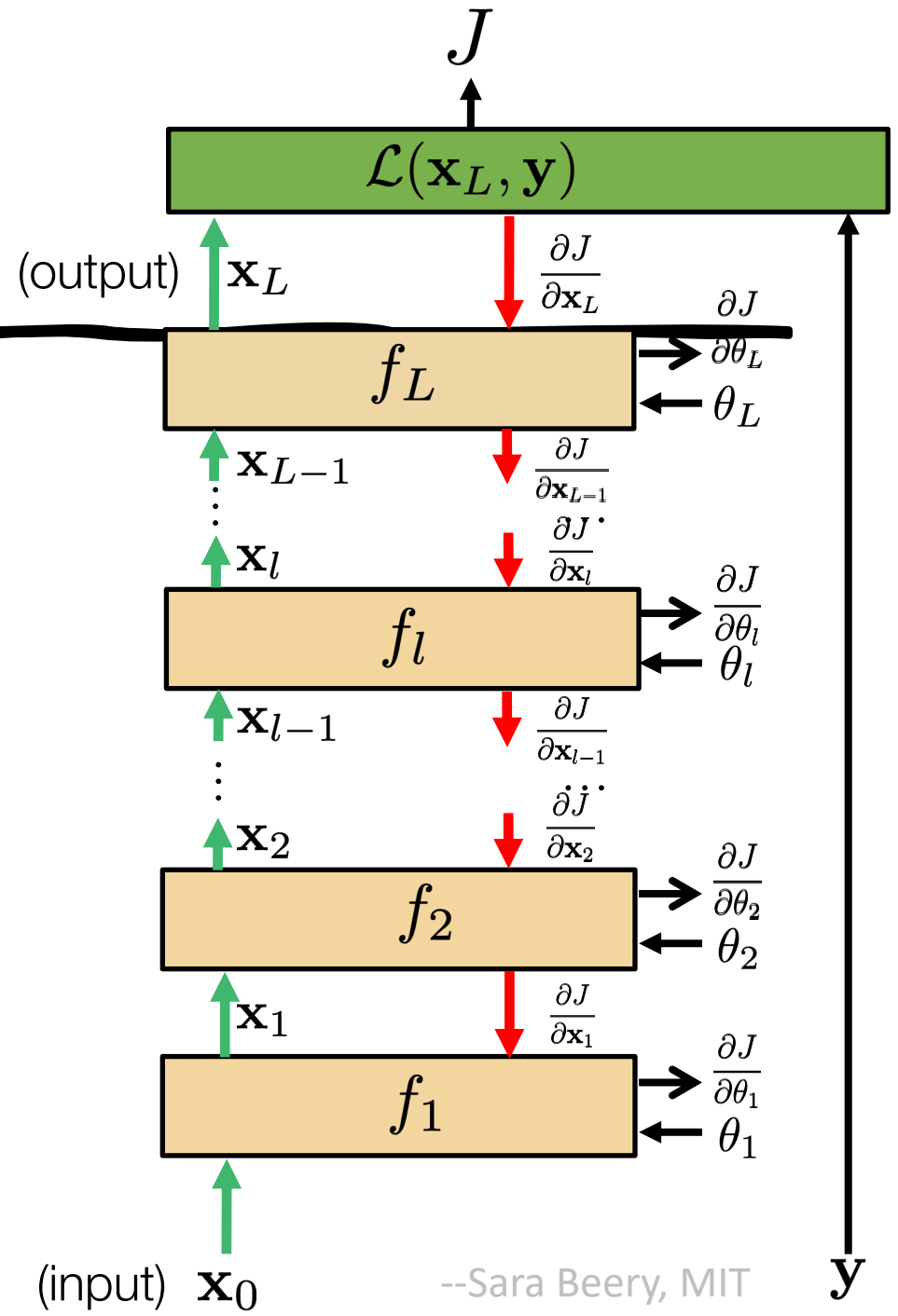
$$\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \theta_l)$$

- Backward Process

$$\frac{\partial J}{\partial \mathbf{x}_{l-1}} = \frac{\partial J}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l}{\partial \mathbf{x}_{l-1}}$$

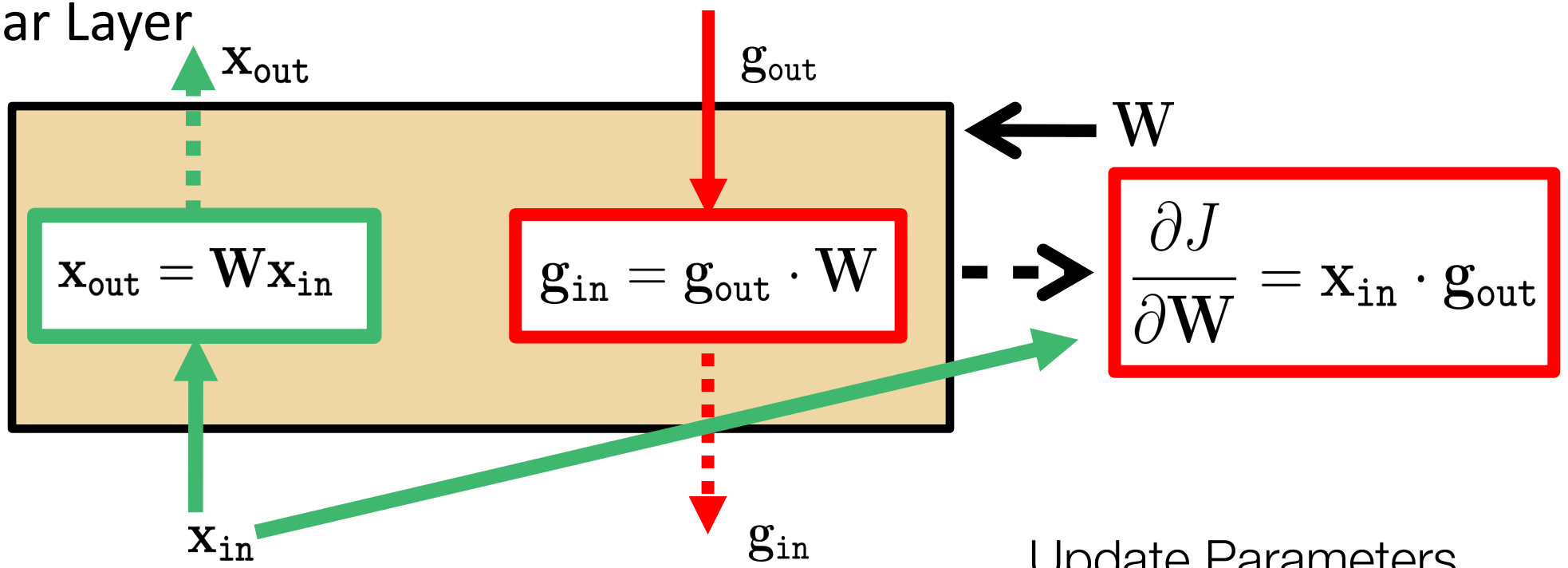
- Update parameters

$$\frac{\partial J}{\partial \theta_l} = \frac{\partial J}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l}{\partial \theta_l} \qquad \theta_l' = \theta_l - \eta \frac{\partial \mathcal{J}}{\partial \theta_l}$$



Back Propagation

- Linear Layer

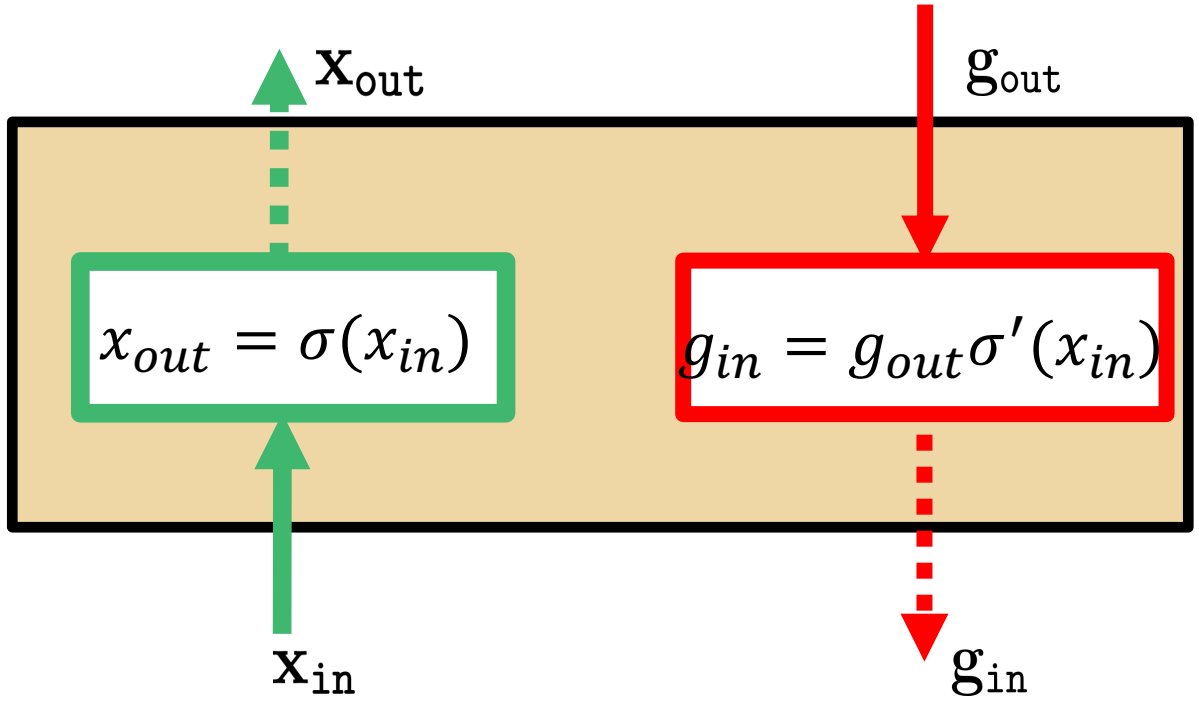


Update Parameters

$$W^{k+1} = W^k - \frac{\eta \partial J}{\partial W}$$

Back Propagation

- Non-linear Layer



Sigmoid function has no parameters

Learning Rate

- Proper learning rates are important

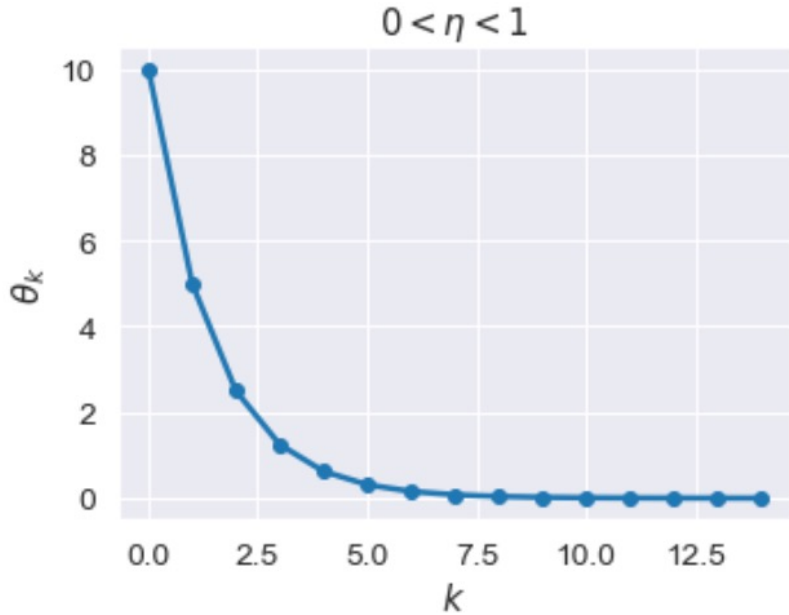
$$\theta_{k+1} = \theta_k - \eta \nabla \mathcal{J}(\theta_k)$$

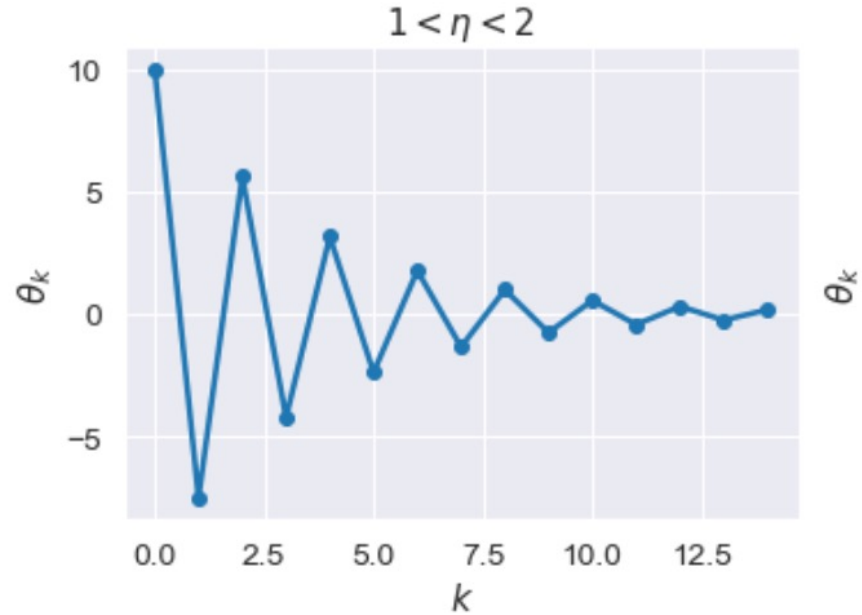
  - Small η: low convergence
  - Large η: non-convergent

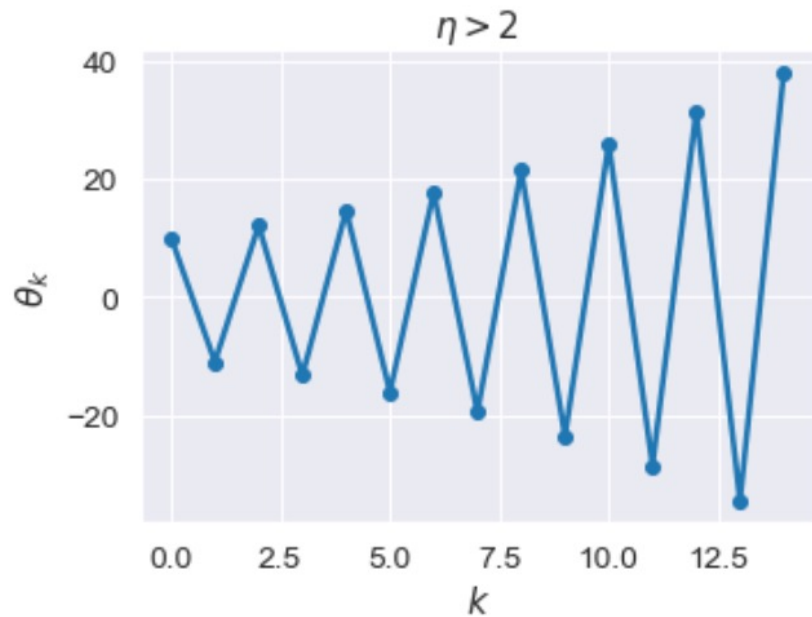
- Example: $\mathcal{J}(\theta) = \frac{1}{2}\theta^2, \quad \nabla \mathcal{J}(\theta) = \theta$

$$\theta^{k+1} = \theta^k - \eta\theta^k = (1-\eta)\theta^k$$

$$\theta^k = (1-\eta)^k \theta^0$$

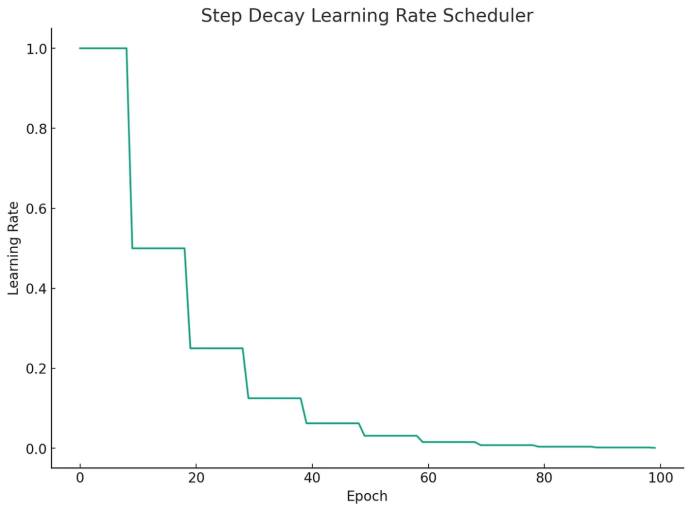

$0 < \eta < 1$
10
8
6
4
2
0
θ_k
0.0
2.5
5.0
7.5
10.0
12.5
k


$1 < \eta < 2$
10
5
0
−5
θ_k
0.0
2.5
5.0
7.5
10.0
12.5
k
θ_k

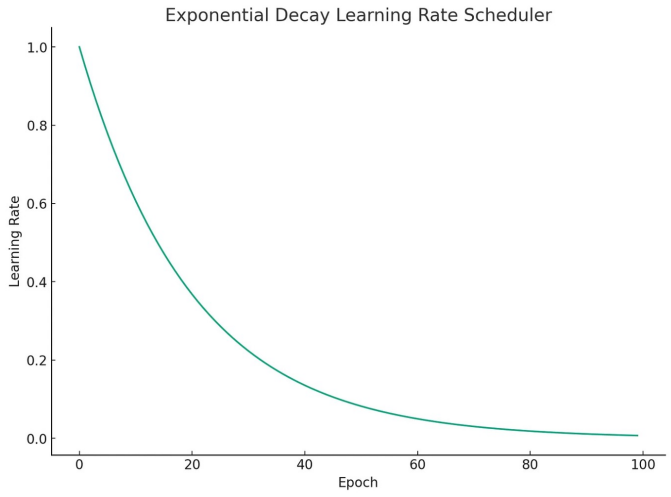

$\eta > 2$
40
20
0
−20
θ_k
0.0
2.5
5.0
7.5
10.0
12.5
k

Learning Rate

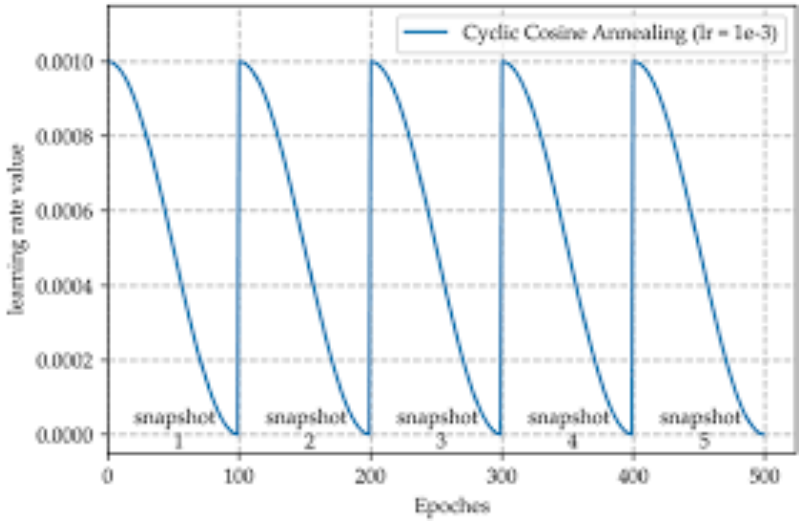
- To converge stably and quickly, learning rate is usually set to decay over time



Step Decay



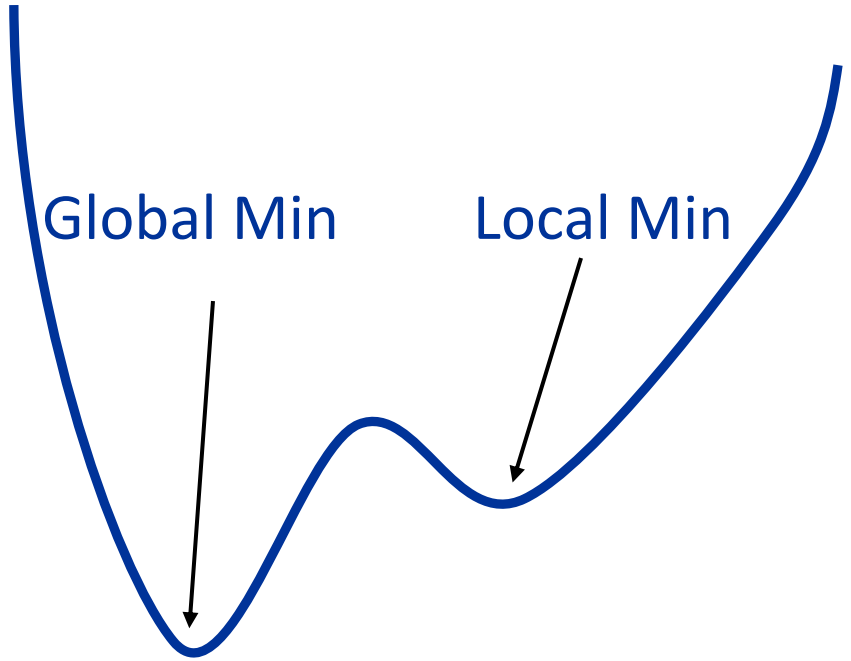
Exponential Decay



Cosine Annealing

Non-convex Optimization

- For convex function, gradient descent algorithm converges to a global minimum, while for non-convex function, there is no guarantee.



The iteration may stuck in the local minimum for non-convex optimization

Stochastic Gradient Descent

- Empirical Loss：

$$\mathcal{J}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{J}_i(\theta)\,,\quad \mathcal{J}_i(\theta) = L(f_\theta(y_i), x_i)$$

Stochastic Gradient Descent: only compute gradients over a mini batch for each iteration

$$\theta_{k+1} = \theta_k - \eta\frac{1}{B}\sum_{i\in I_B}\nabla\mathcal{J}_i(\theta_k)$$

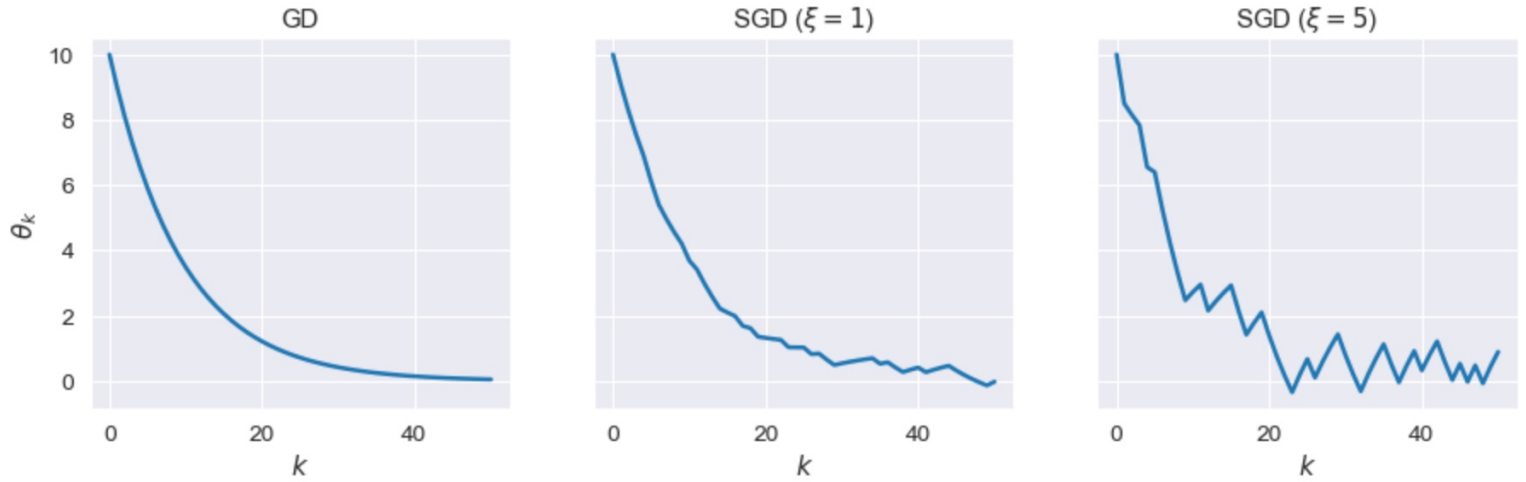
I_B(mini batch) are randomly sampled from the training data indexs $\{1,2,\ldots,N\}$

Stochastic Gradient Descent

- The gradient on the mini-batch can be seen as an approximation of the total gradient.

Example: $\mathcal{J}_1(\theta) = \frac{1}{2}(\theta - \xi)^2 \quad \mathcal{J}_2(\theta) = \frac{1}{2}(\theta + \xi)^2$

$$\mathcal{J}(\theta) = \mathcal{J}_1(\theta) + \mathcal{J}_2(\theta) = \frac{1}{2}(\theta^2 + \xi^2)$$



Stochastic Gradient Descent

- The gradient is computed on a mini-batch each time, which greatly reduces the computational cost.
- The randomness can help avoid getting stuck in local optima to some extent.
- It has good generalization properties and tends to converge to a flatter solution.

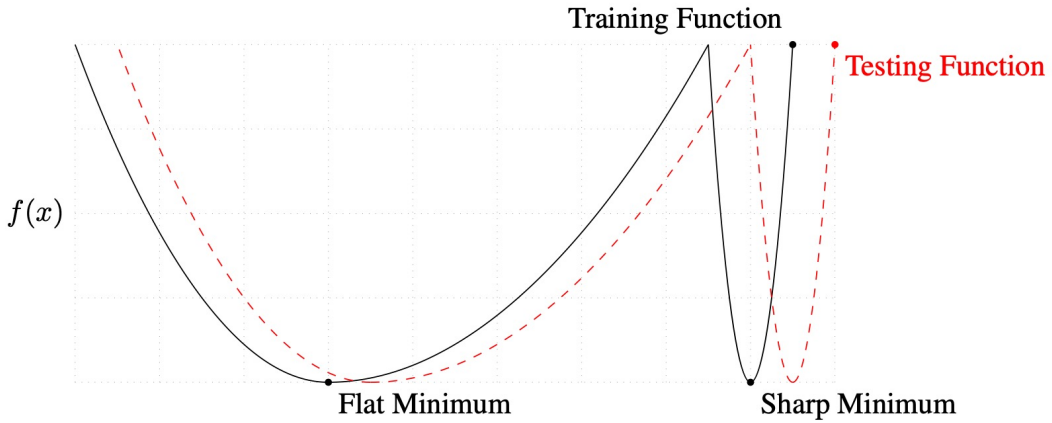


Figure 1: A Conceptual Sketch of Flat and Sharp Minima. The Y-axis indicates value of the loss function and the X-axis the variables (parameters)

-- https://arxiv.org/pdf/1609.04836.pdf

SGD with Momentum

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

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

v_t: Momentum helps accelerate convergence by accumulating the gradients of past steps and smoothing out the oscillations

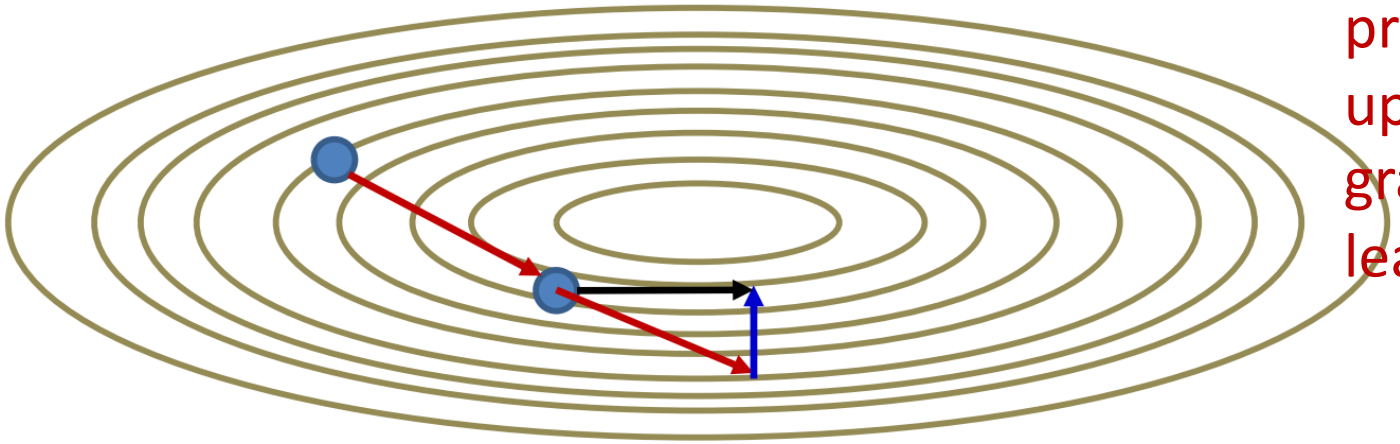
$\rho = 0.9 \; or \; 0.99$: control the friction

Nesterov Momentum

- A “lookahead” step

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

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



Standard momentum moves in the direction of the past gradient, while Nesterov momentum first makes a prediction (lookahead) about the next update and then computes the gradient at this predicted location, leading to faster convergence.

Adaptive Learning Rate

- AdaGrad

$$r_t = r_{t-1} + \nabla f(x_t) \odot \nabla f(x_t)$$

$$x_{t+1} = x_t - \frac{\alpha}{\sqrt{r_t + \epsilon}} \odot \nabla f(x_t)$$

For frequent features, the updates will be smaller, and for rare features, the updates will be larger.

Benefits: Efficient for sparse information

Drawbacks: Aggressive Learning Rate Decay

RMSProp

- Exponential Moving Averaging (EMA)

$$r_t = \beta r_{t-1} + (1-\beta)\nabla f(x_t) \odot \nabla f(x_t)$$

$$x_{t+1} = x_t - \frac{\alpha}{\sqrt{r_t + \epsilon}} \odot \nabla f(x_t)$$

- RMSProp uses a moving average, avoiding overly aggressive decay compared with AdaGrad
- Typical values: $\beta = 0.9$

Adam

- Adam (Adaptive Moment Estimation) = RMSprop + Momentum:

$$g_t = \nabla f(x_t)$$

$$v_t = \beta_1 v_{t-1} + (1-\beta_1) g_t, \qquad r_t = \beta_2 r_{t-1} + (1-\beta_2) g_t \odot g_t$$

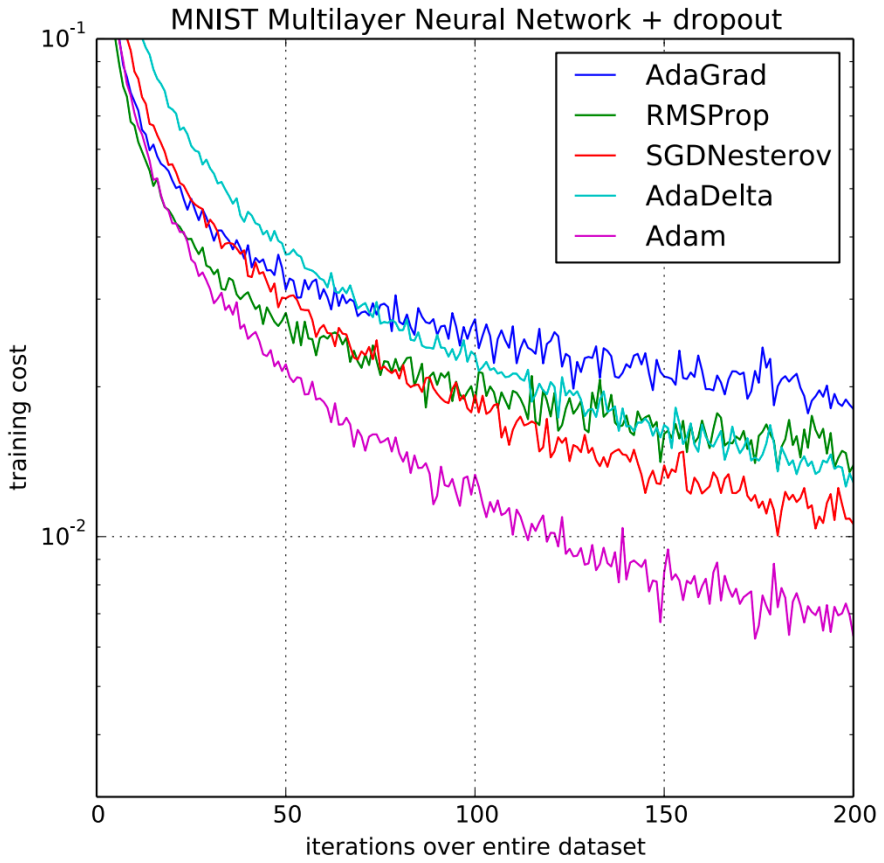
$$v_t = \frac{v_t}{1-\beta_1^t}, \gamma_t = \frac{\gamma_t}{1-\beta_2^t}$$

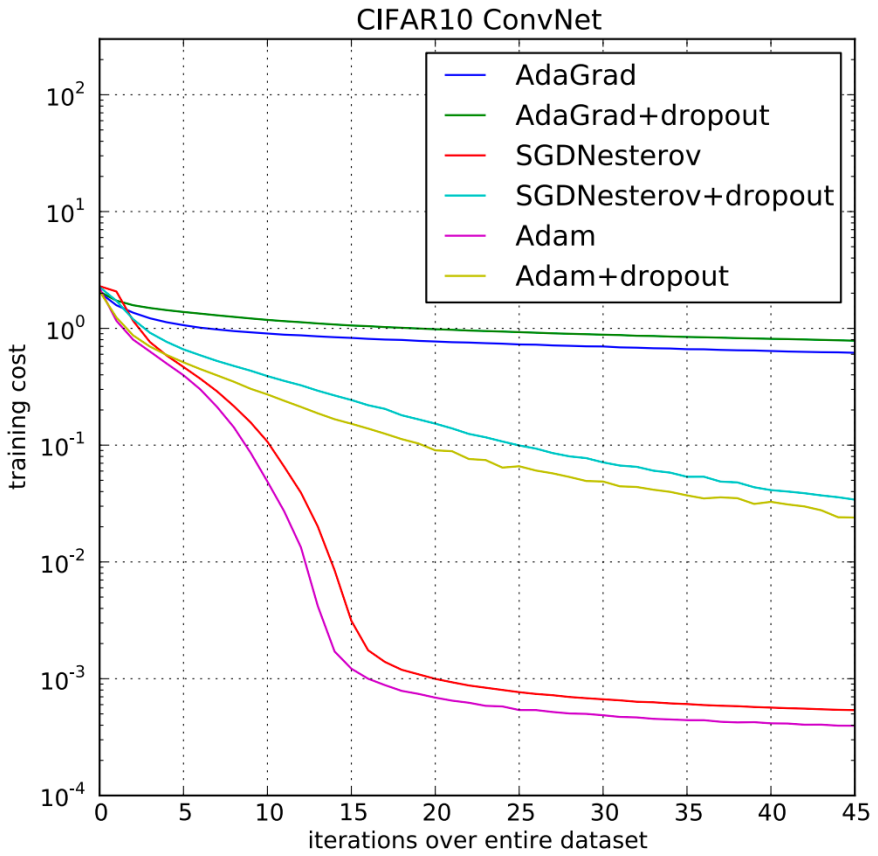
Bias correction

$$x_{t+1} = x_t - \frac{\alpha}{\sqrt{r_t + \epsilon}} \odot v_t$$

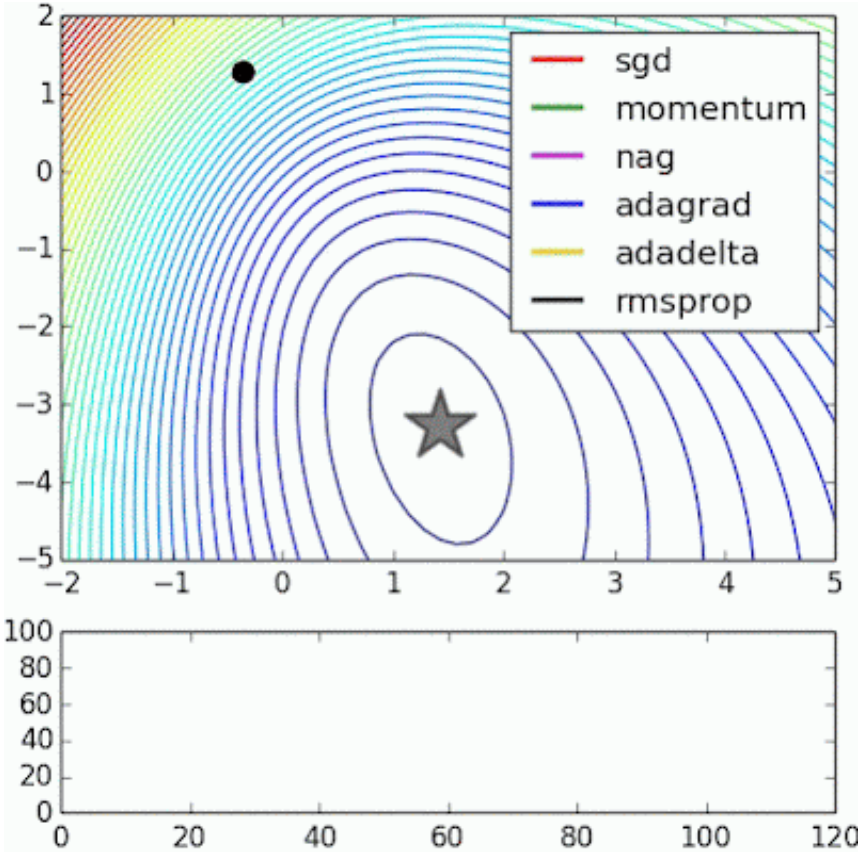
ensure $\beta_1 v_{t-1}$ and $\beta_2 r_{t-1}$ do not dominate in early iterations

Comparison

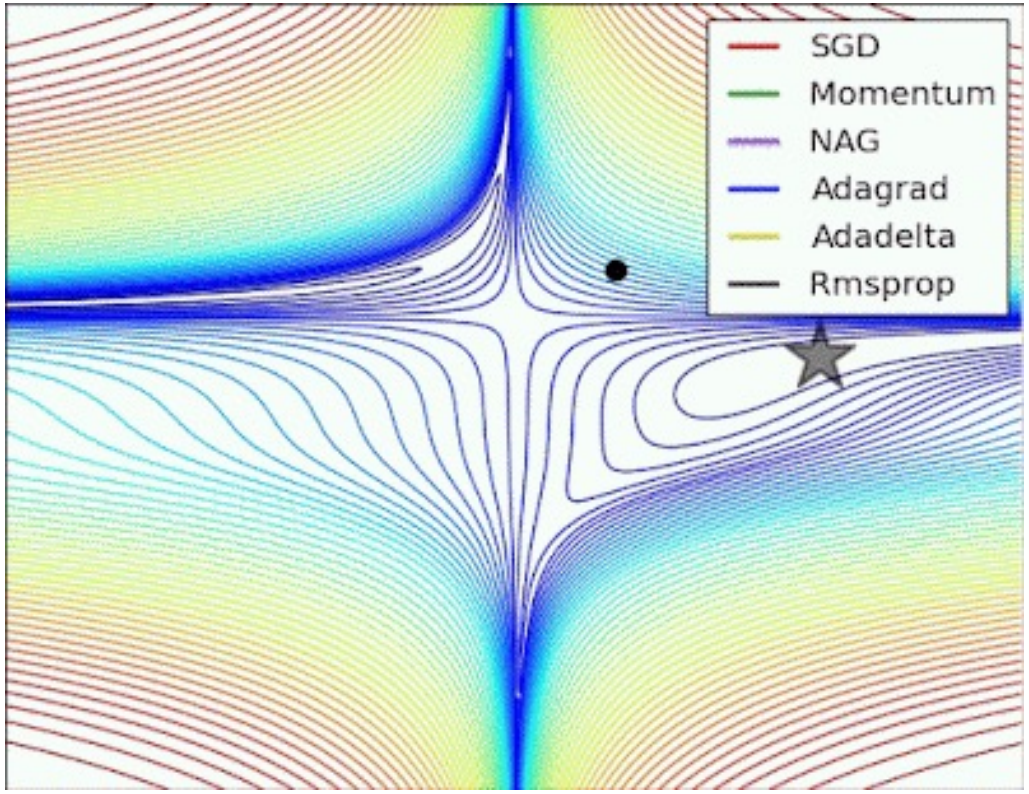

MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
training cost
iterations over entire dataset


CIFAR10 ConvNet
AdaGrad
AdaGrad+dropout
SGDNesterov
SGDNesterov+dropout
Adam
Adam+dropout
training cost
iterations over entire dataset

Visualization



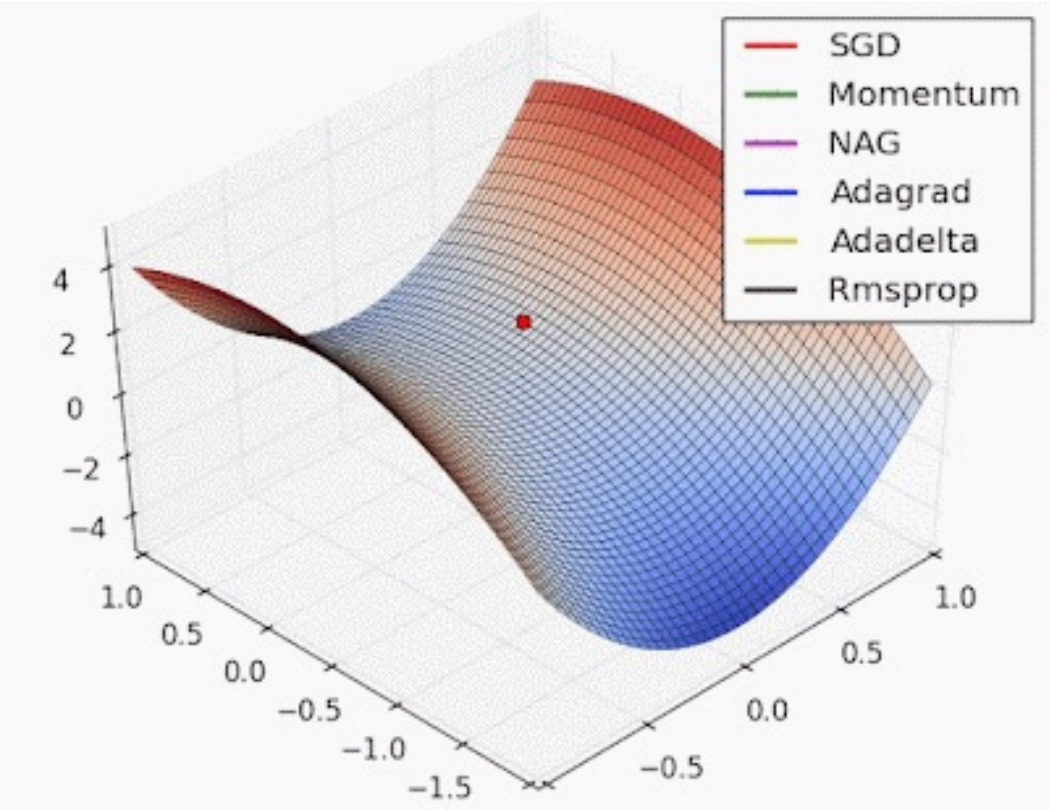
Logistic regression on noisy moons dataset



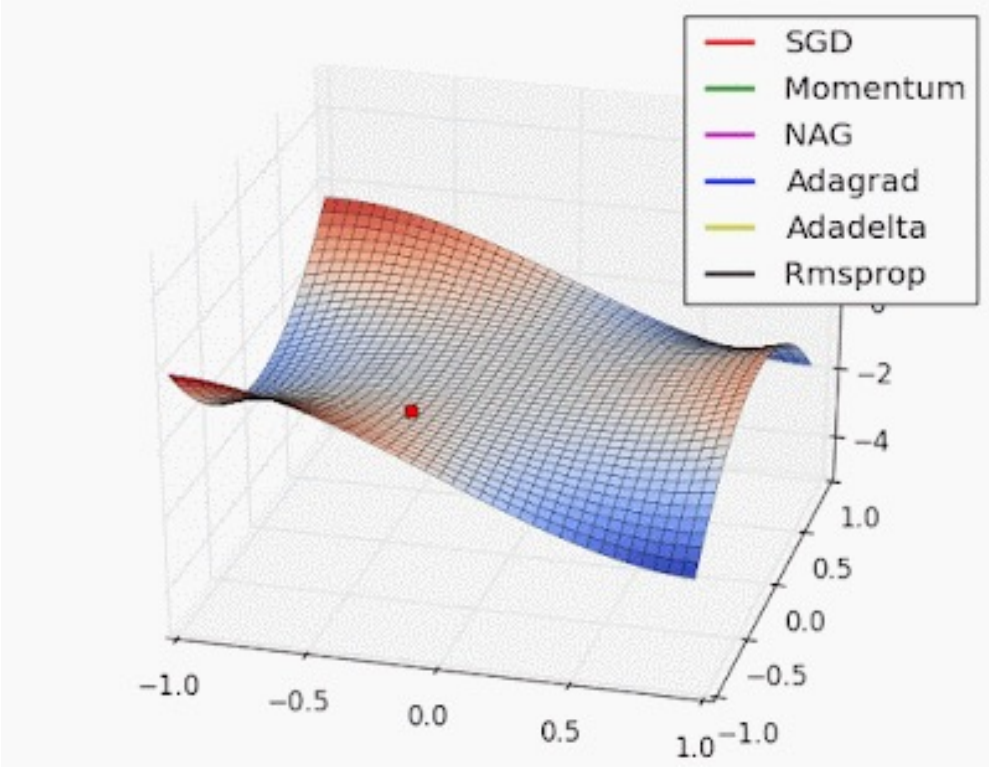
Velocity based techniques shoot off and bounce around; RMSProp is more stable

http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Visualization



Algorithms that scale step size based on the gradient quickly break symmetry and begin descent



Behaviour around a saddle point.

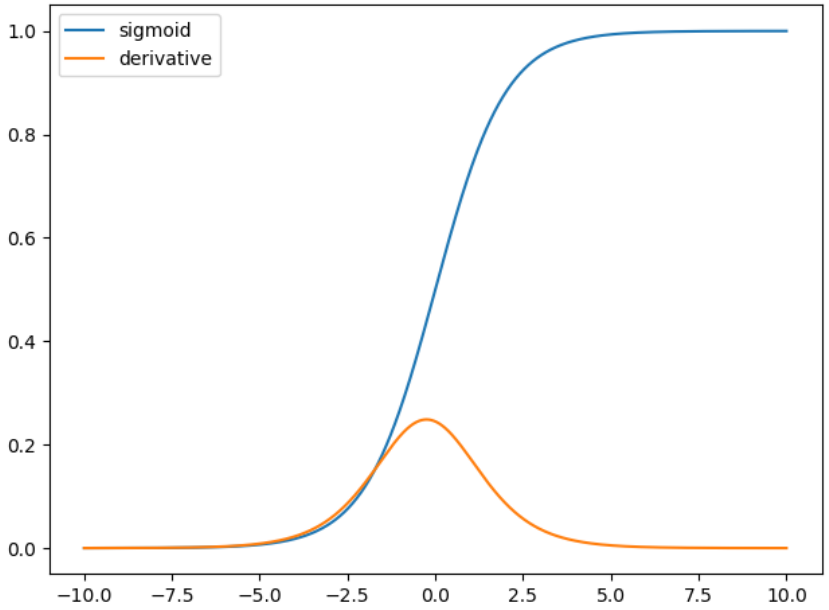
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Vanishing or Exploding Gradient

- Chain Rule

$$\frac{\partial \mathcal{J}}{\partial \theta_1} = \frac{\partial \mathcal{J}}{\partial x_L} \frac{\partial x_L}{\partial x_{L-1}} \cdots \frac{\partial x_3}{\partial x_2} \frac{\partial x_2}{\partial x_1} \frac{\partial x_1}{\partial \theta_1}$$

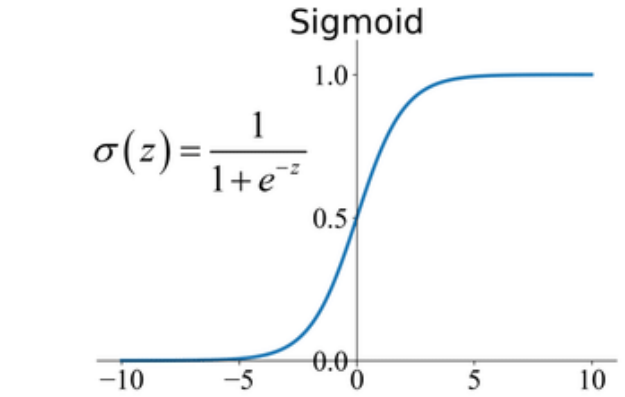
The gradient of the Sigmoid function is very small most of the time, leading to vanishing gradients!



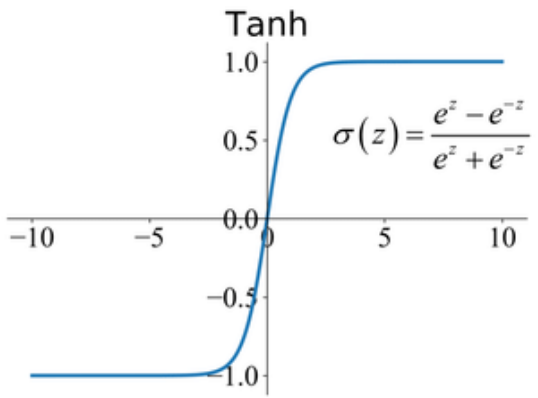
Vanishing or Exploding Gradient

More Activation Functions

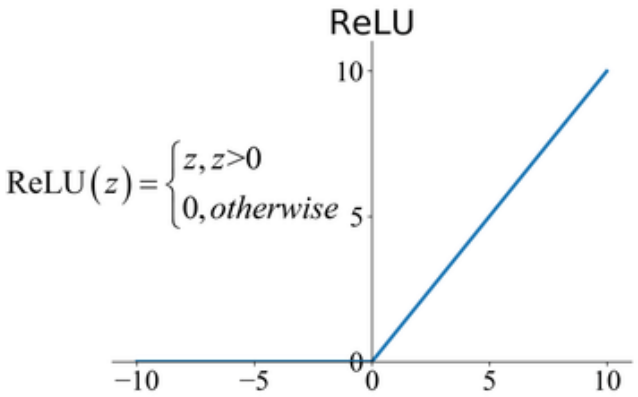
- Tanh: The maximum derivative is 1, but it can still vanish.
- ReLU: The gradient neither vanishes nor explodes; it is computationally fast, but some neurons may not be activated.
- LeakyReLU: Solves the issue with ReLU and is the most commonly used.



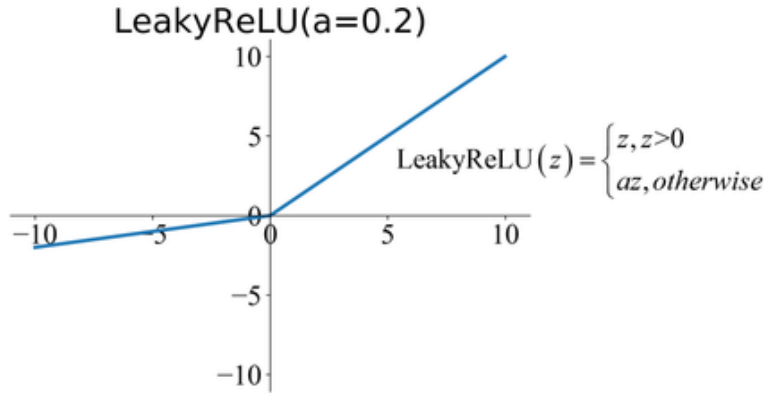
(a)



(b)



(c)



(d)

Initialization

- Consider the back propagation $\frac{\partial \mathcal{J}}{\partial x} = \frac{\partial \mathcal{J}}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial \mathcal{J}}{\partial y}W$

$$Var\left(\frac{\partial \mathcal{J}}{\partial x}\right) = nVar(w)Var\left(\frac{\partial \mathcal{J}}{\partial y}\right)$$

Cumulate after multi-layers

$$Var\left(\frac{\partial \mathcal{J}}{\partial x_1}\right) = \prod_l n_l Var\left(w^{(l)}\right)Var\left(\frac{\partial \mathcal{J}}{\partial x_L}\right)$$

To avoid gradient vanishment or exploding, we should set

$$n_l Var\left(w^{(l)}\right) \sim 1$$

Initialization

Consider the linear layer $y = Wx$

$$Var(y_i) = Var\left(\sum_j W_{ij} x_j\right)$$

$$Var(y_i) = \sum_j (Var(W_{ij} x_j))$$

The components of x (and w) are independent.

$$Var(y_i) = \sum_j (Var(W_{ij}) Var(x_j))$$

w_{ij} and x_j are independent, and of zero mean.

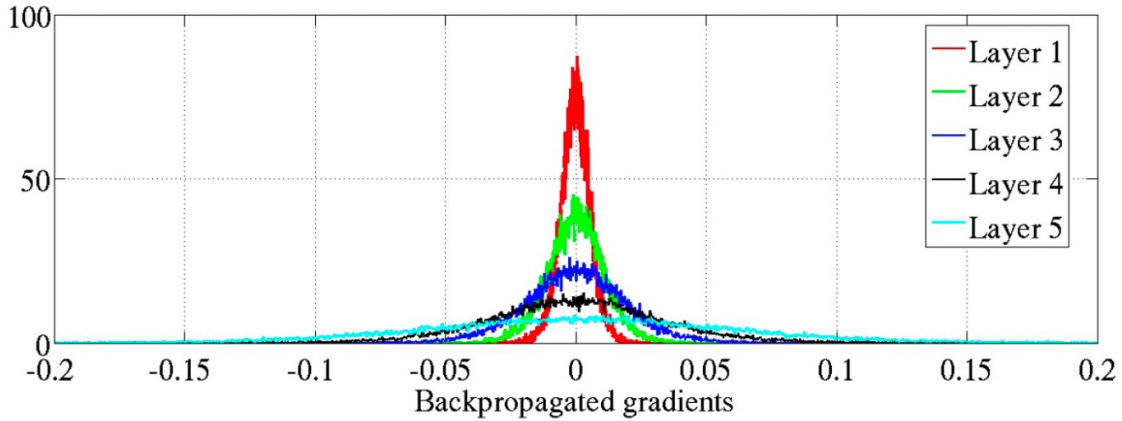
$$Var(y) = nVar(w)Var(x)$$

The components of x (and w) are identically distributed.

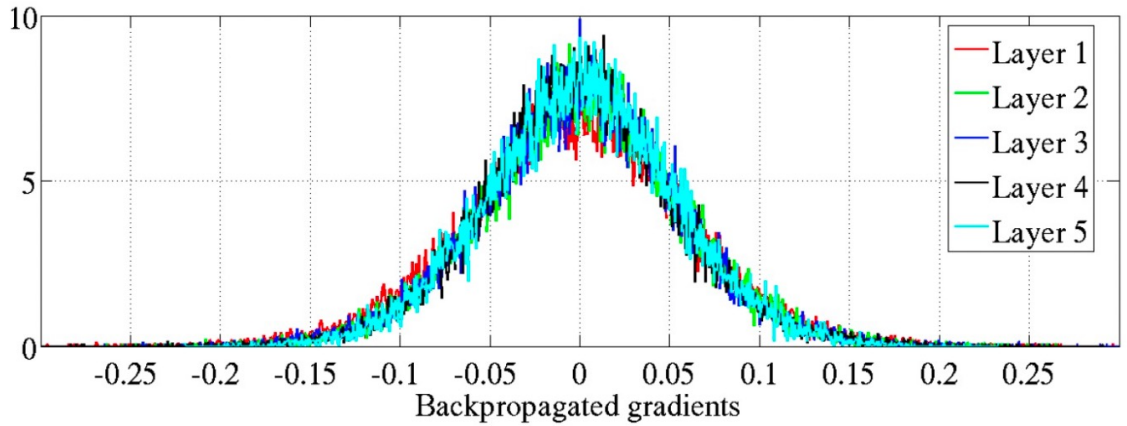
Initialization

- Xavier Initialization

Gaussian $N\left(0, \frac{1}{n}\right)$, Uniform: $U[-\sqrt{\frac{3}{n}}, \sqrt{\frac{3}{n}}]$



poor initialization:
earlier layer has smaller gradients



Xavier initialization:
all layers have similar gradient scale

Glorot & Bengio 2010 “Understanding the difficulty of training deep feedforward neural networks”

Initialization

- Consider Relu activation: $x' = ReLU(x), y = Wx'$

$$E(x'^2) = \frac{1}{2}Var(x)$$

Assume x is symmetric and of zero mean.

$$Var(y_i) = \sum_j (Var(W_{ij}x'_j))$$

The components of x' (and w) are independent.

$\mathrm{Var}[wx']$
$= E[(wx')^2] - (E[wx'])^2$
$= E[(wx')^2] - (E[w]E[x])^2$
$= E[w^2]E[x'^2] = \mathrm{Var}[w]E[x'^2]$

$$Var(y_i) = \sum_j (Var(W_{ij})E(x'^2_j))$$

w_{ij} and x_j are independent; w_{ij} are of zero mean.

$$Var(y) = nVar(w)E(x'^2) = \frac{1}{2}nVar(w)Var(x)$$

The components of x (and w) are identically distributed.

Initialization

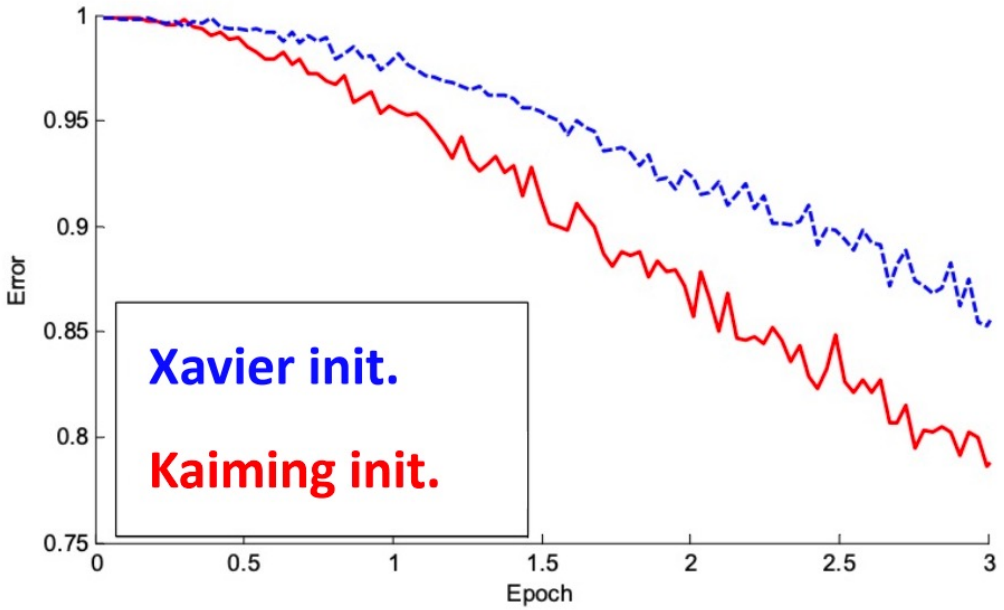
- Kaiming Initialization

$$Var\left(\frac{\partial \mathcal{J}}{\partial x_1}\right) = \prod_l \frac{1}{2} n_l Var\left(w^{(l)}\right) Var\left(\frac{\partial \mathcal{J}}{\partial x_L}\right)$$

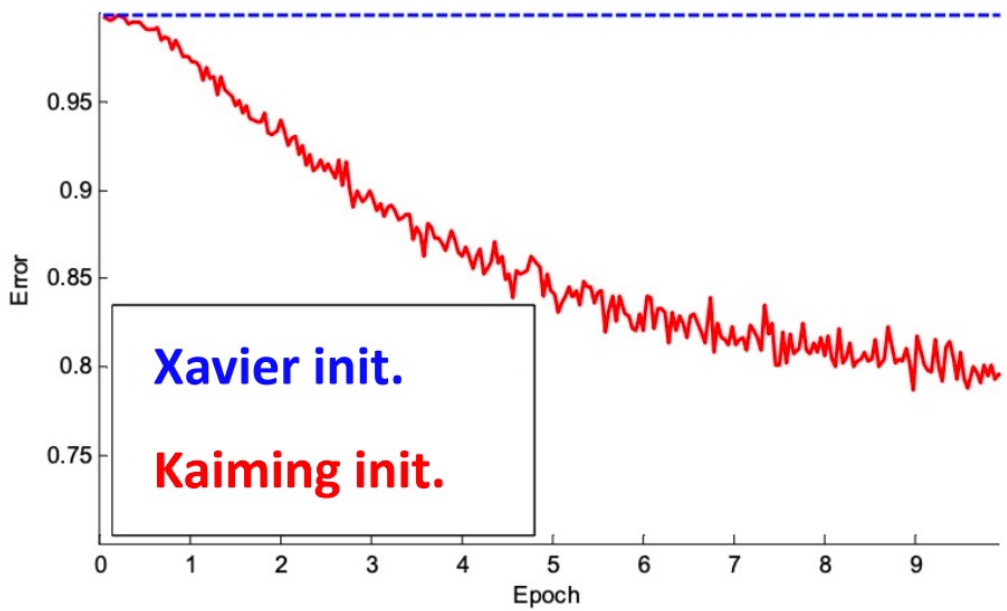
$$\frac{1}{2} n_l Var\left(w^{(l)}\right) \sim 1$$

$$\text{Gaussian: } N\left(0, \frac{2}{n}\right), \text{Uniform: } U[-\sqrt{\frac{6}{n}}, \sqrt{\frac{6}{n}}]$$

Initialization



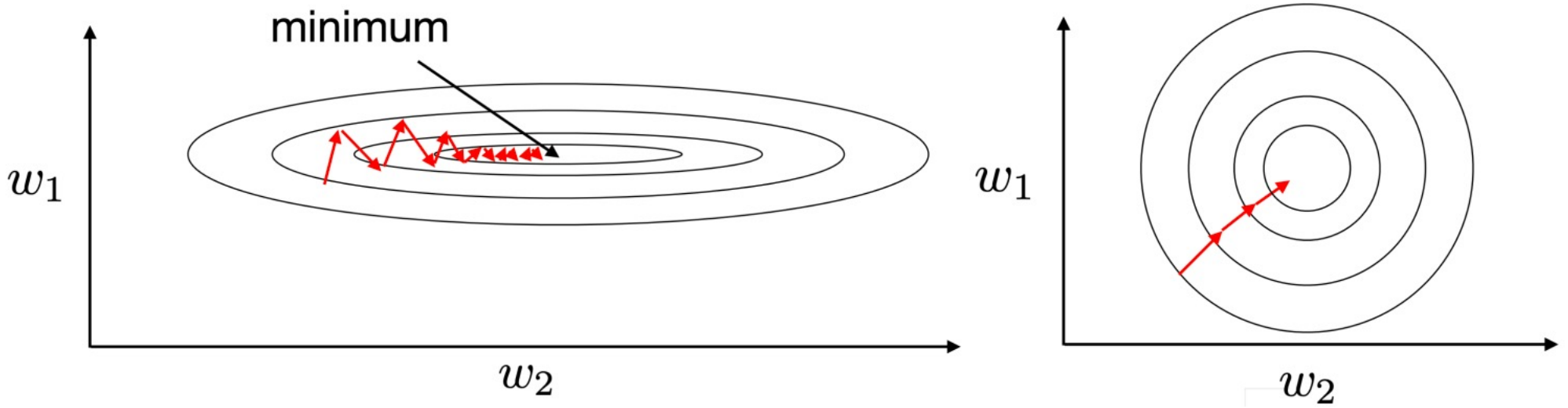
22-layer VGG



30-layer VGG

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. ICCV 2015.

Normalization



- When same learning rate applied to all weights, large weights dominate updates and small weights oscillate (or diverge)
- After normalizing their variances, the updating becomes more steady and efficient.

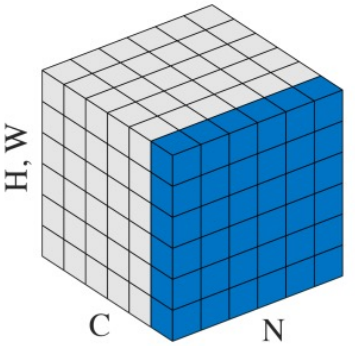
Batch Normalization

- To avoid variance becoming too small (vanishing gradient) or too large (exploding gradient) in deep layers, normalize features in the network

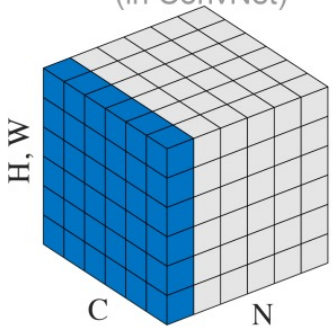
$$\hat{x}_i = \frac{x_i - \mathbb{E}x_i}{\sqrt{var(x_i)}}$$

- **Batch** Normalization: normalize features across samples within each batch.
- Other normalization variants: e.g. Layer normalization, instance normalizations, differ in support sets of calculating the mean and variance.

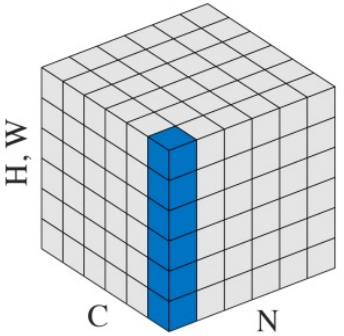
BatchNorm



LayerNorm
(in ConvNet)



InstanceNorm



Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

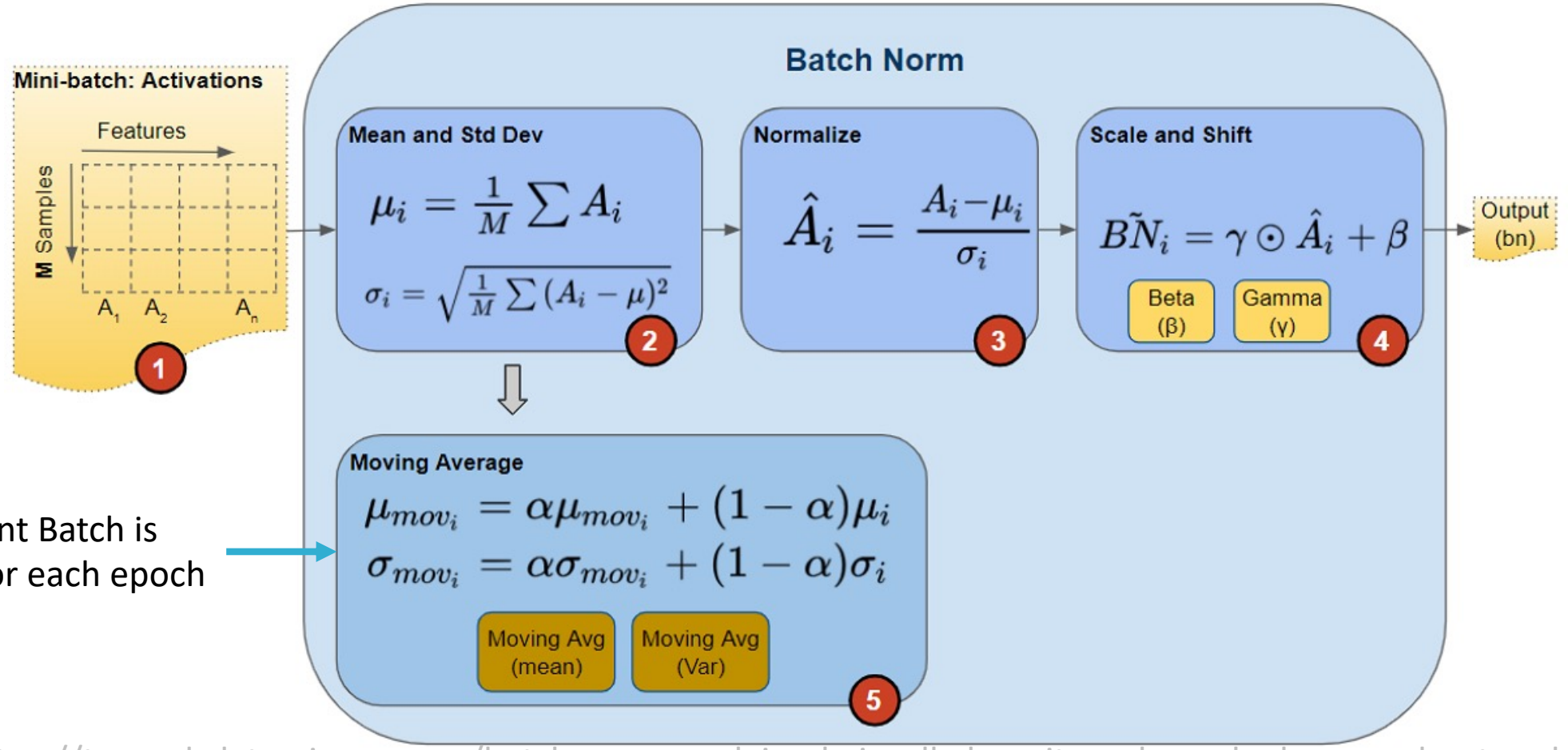
Scale and shift after BN

$$y_i = \gamma\widehat{x_i} + \beta$$

γ, β are learnable scalars

Ioffe & Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". ICML 2015

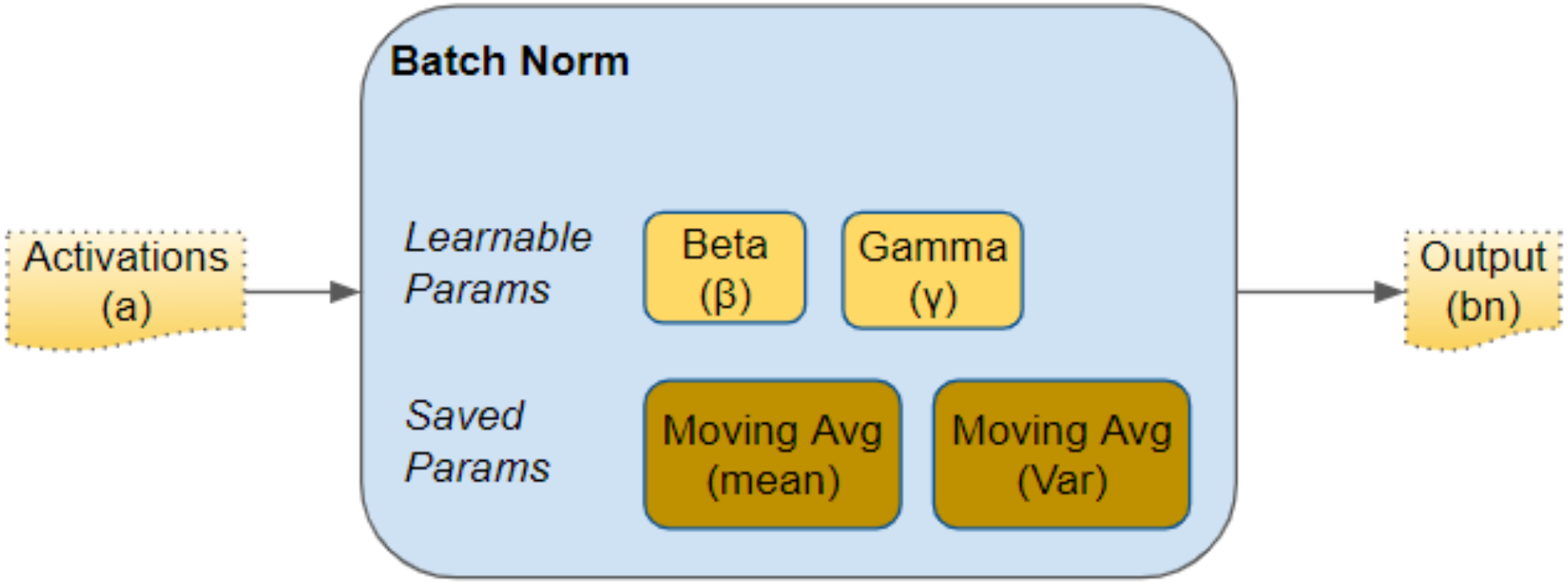
Batch Normalization



Different Batch is used for each epoch

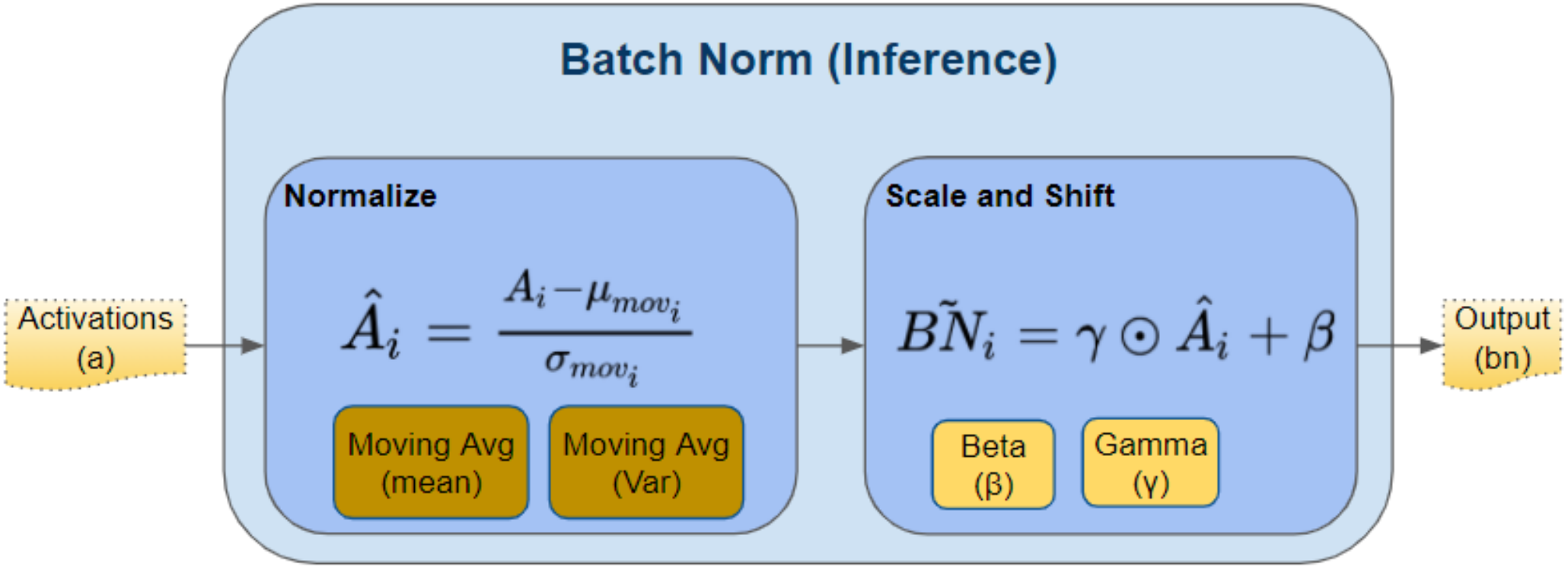
https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

Batch Normalization



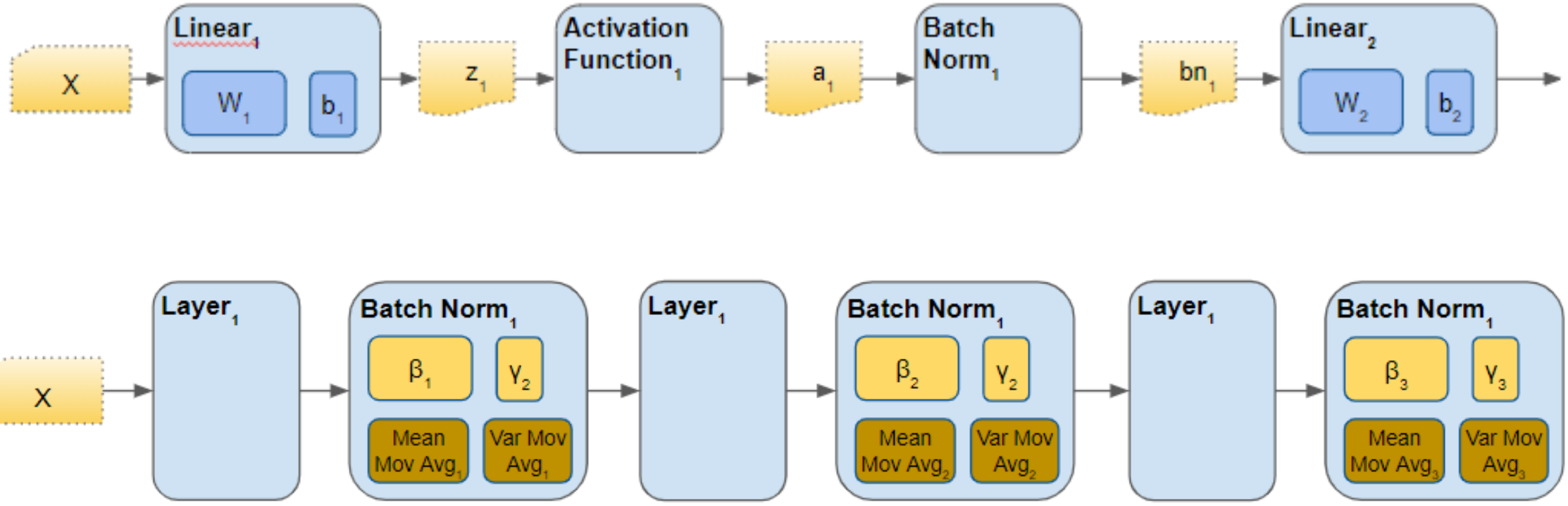
https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

Batch Normalization



https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

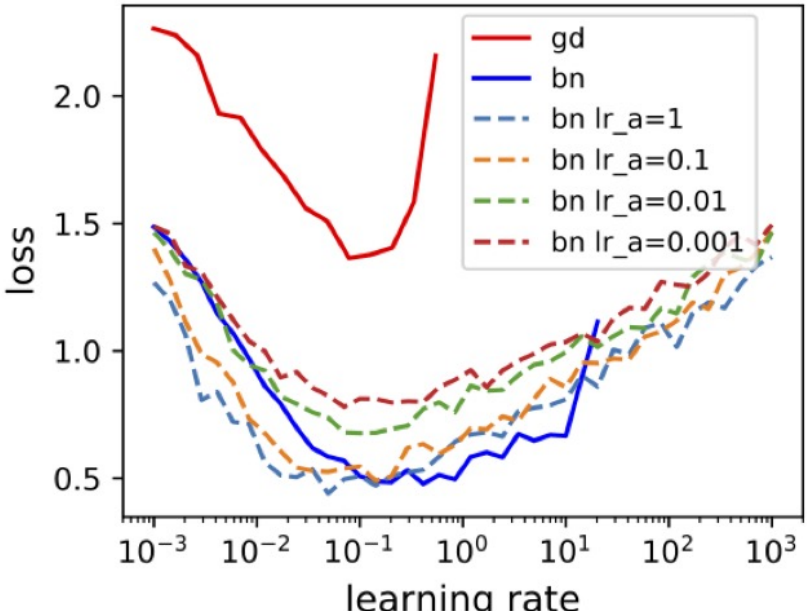
Batch Normalization

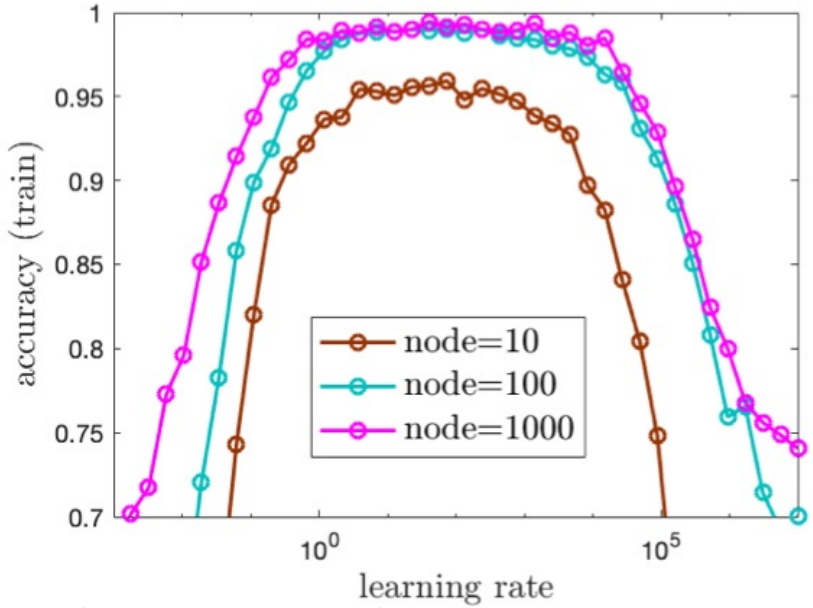


https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

Batch Normalization

- BN is less sensitive to step size:
  - In the linear case, It is proven that BN (Batch Normalization) + GD (Gradient Descent) converges for any step size.
  - The higher the dimension, the larger the maximum convergence step size.
- More benefits: stabilizes training provides a regularization effect, ensures faster convergence……

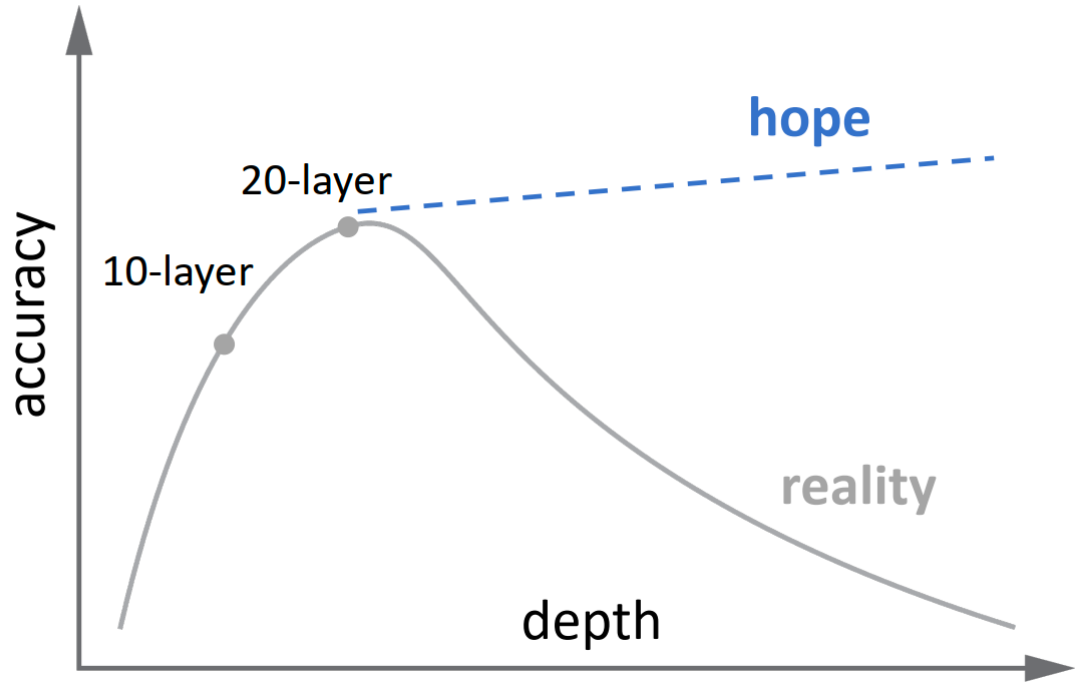




Reference： A Quantitative Analysis of the Effect of Batch Normalization on Gradient Descent

Residual Network (ResNet)

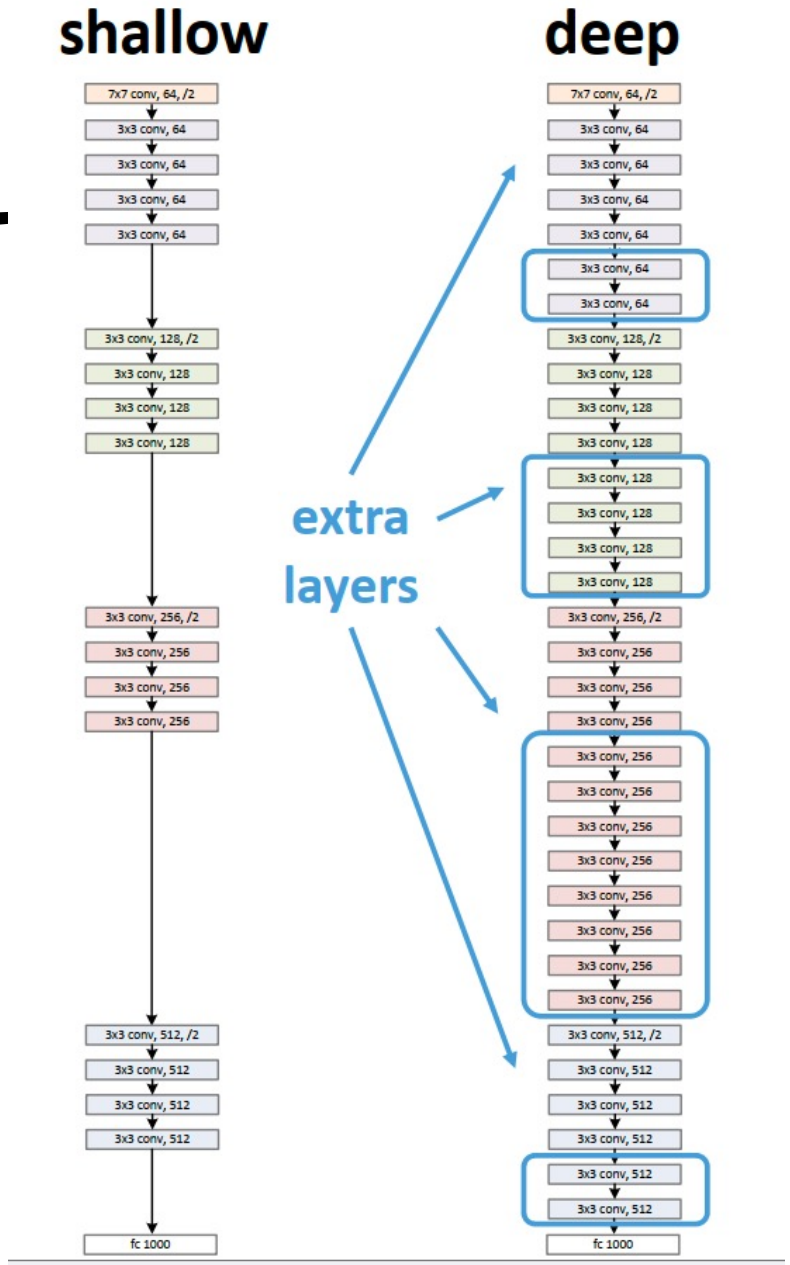
Even with good initialization and BN, stacking more than 20 layers will make the network very hard to train.



-- By Kaiming He

A thought experiment

- Deeper nets are more expressive; the best approximation cannot be worse.
- Bad optimization is the problem.
- Find the solution by construction
  - take a good shallow net
  - copy trained layers
  - let extra layers be identity mapping
  - no worse training accuracy

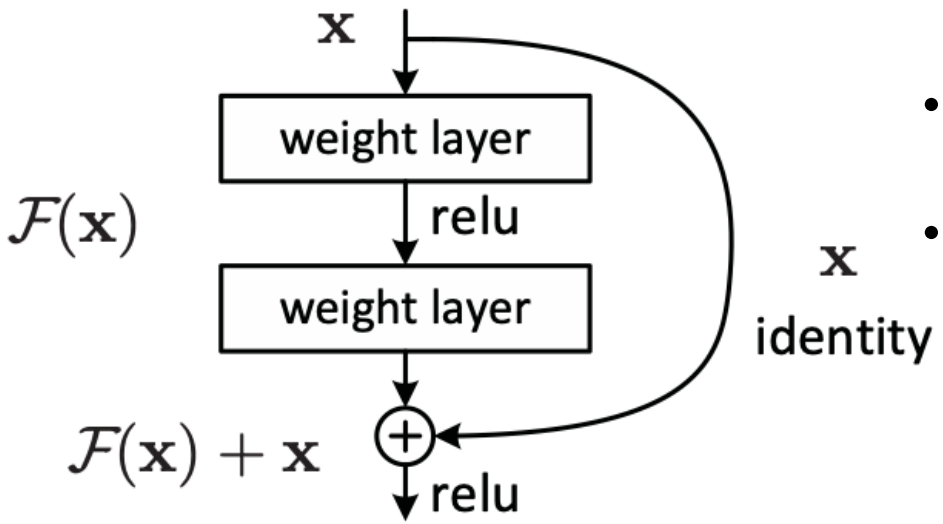


Adapted from Kaiming He's slides

ResNet

ResNet-152

Residual Connection (a.k.a. shortcut, skip connection)

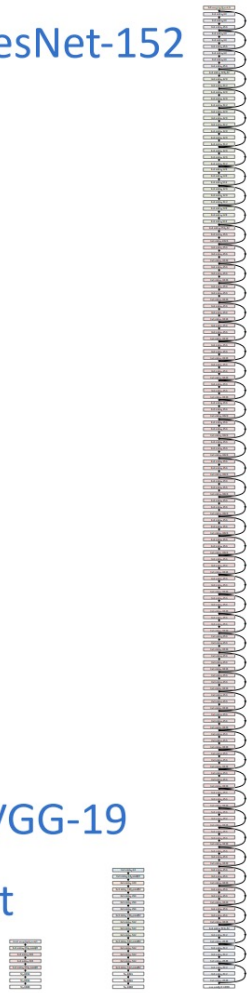


$$H(x) = F(x) + x$$

- Small progressive changes in each block ($H(x)$ is near identity)
- Small weights (then small or zero weight initialization is preferred.)
- Enable deep stacking.

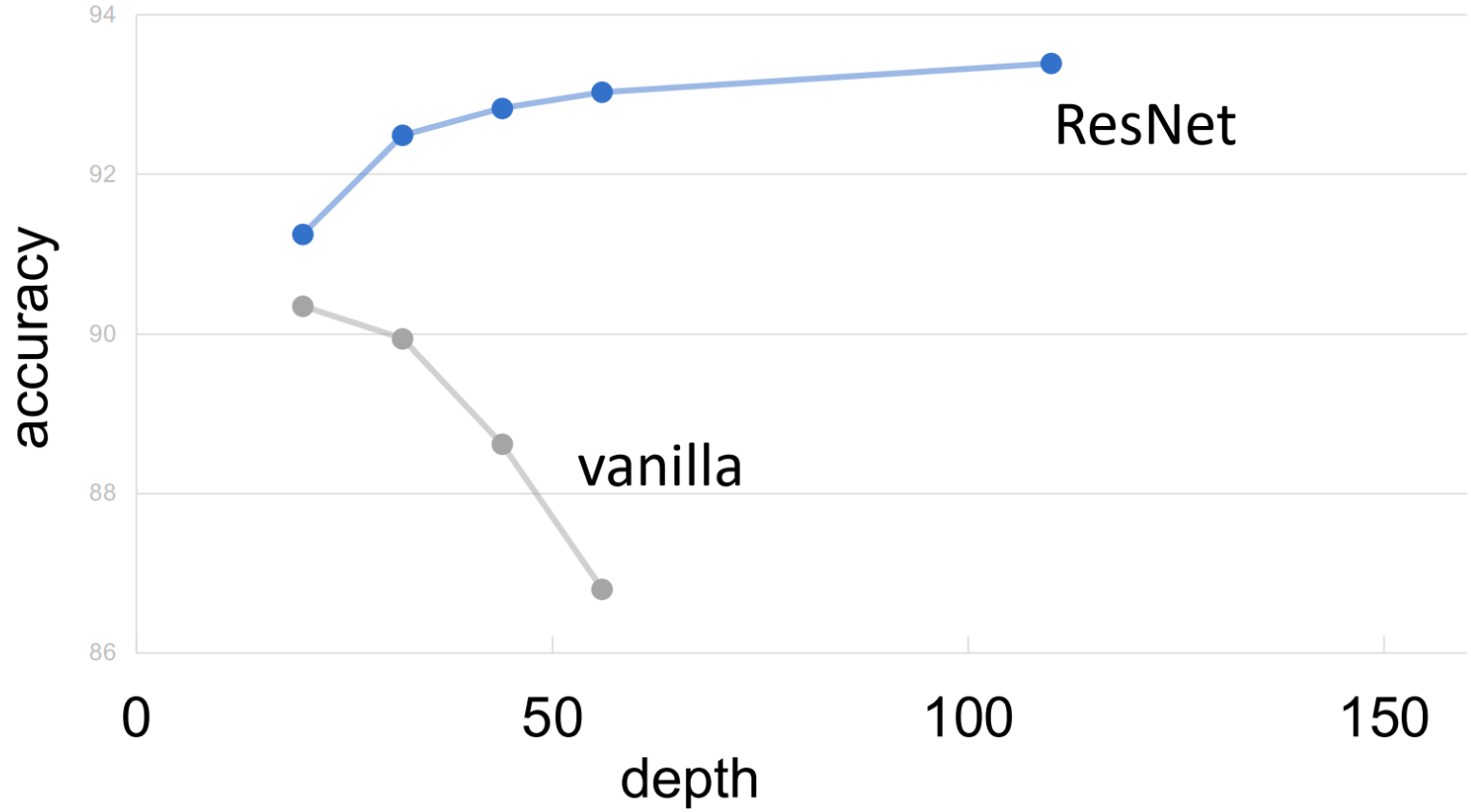
VGG-19

AlexNet



Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. “Deep Residual Learning for Image Recognition”. arXiv 2015, CVPR 2016

ResNet



CIFAR10 Classfication

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. “Deep Residual Learning for Image Recognition”. arXiv 2015, CVPR 2016

ResNet

Plain Network

$$x_{l+1} = W_l x_l$$

$$x_L = \left(\prod W_i\right) x_l$$

$$\frac{\partial \mathcal{E}}{\partial x_l} = \frac{\partial \mathcal{E}}{\partial x_L}\left(\prod W_i\right)$$

Vanishing gradients

Residual Network

$$x_{l+1} = x_l + F_l(x_l)$$

$$x_L = x_l + \sum F_i(x_i)$$

$$\frac{\partial \mathcal{E}}{\partial x_l} = \frac{\partial \mathcal{E}}{\partial x_L}\left(1 + \frac{\partial}{\partial x_l}\sum F_i(x_i)\right)$$

No vanishing

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. “Identity Mappings in Deep Residual Networks”, ECCV 2016

ResNet

- Consider non-identity shortcut $= \lambda x$:

$$x_{l+1} = \lambda x_l + F_l(x_l)$$

$$x_L = \lambda^{L-l} x_l + \sum \hat{F}_i(x_i)$$

$$\frac{\partial \mathcal{E}}{\partial x_l} = \frac{\partial \mathcal{E}}{\partial x_L}\left(\lambda^{L-l} + \frac{\partial}{\partial x_l}\sum \hat{F}_i(x_i)\right)$$

$\lambda < 1$, vanishing gradients

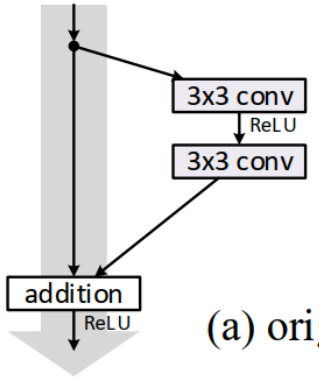
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. "Identity Mappings in Deep Residual Networks", ECCV 2016

ResNet

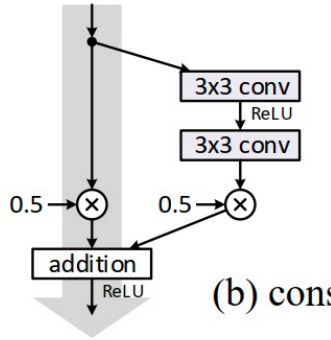
ResNet-110 on CIFAR-10

$h(x) = x$

error: **6.6%**



(a) original



(b) constant scaling

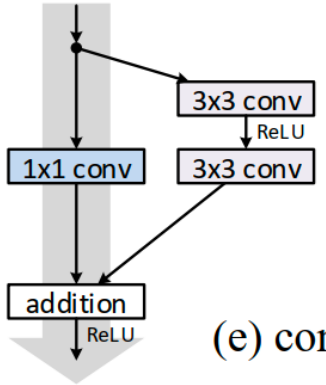
$h(x) = 0.5x$

error: **12.4%**

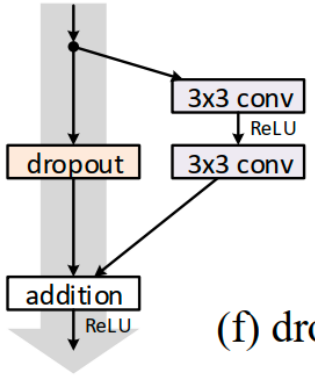
Shortcut≠identity, blocked signals

$h(x) = \text{conv}(x)$

error: **12.2%**



(e) conv shortcut



(f) dropout shortcut

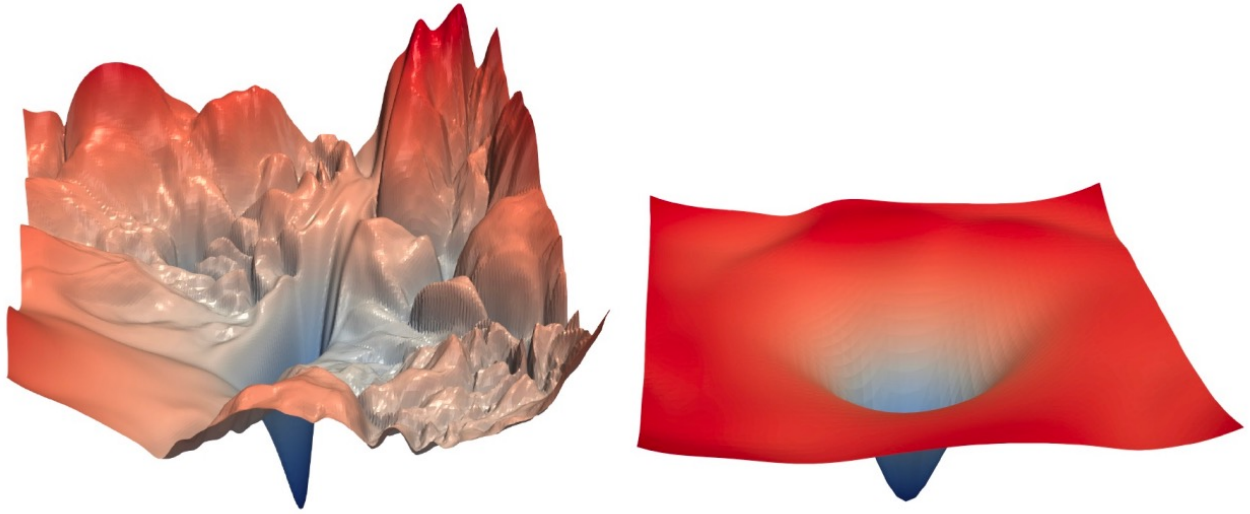
$h(x) = \text{dropout}(x)$

error: **> 20%**

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. “Identity Mappings in Deep Residual Networks”, ECCV 2016

ResNet

- The landscape of residual network is smoother.



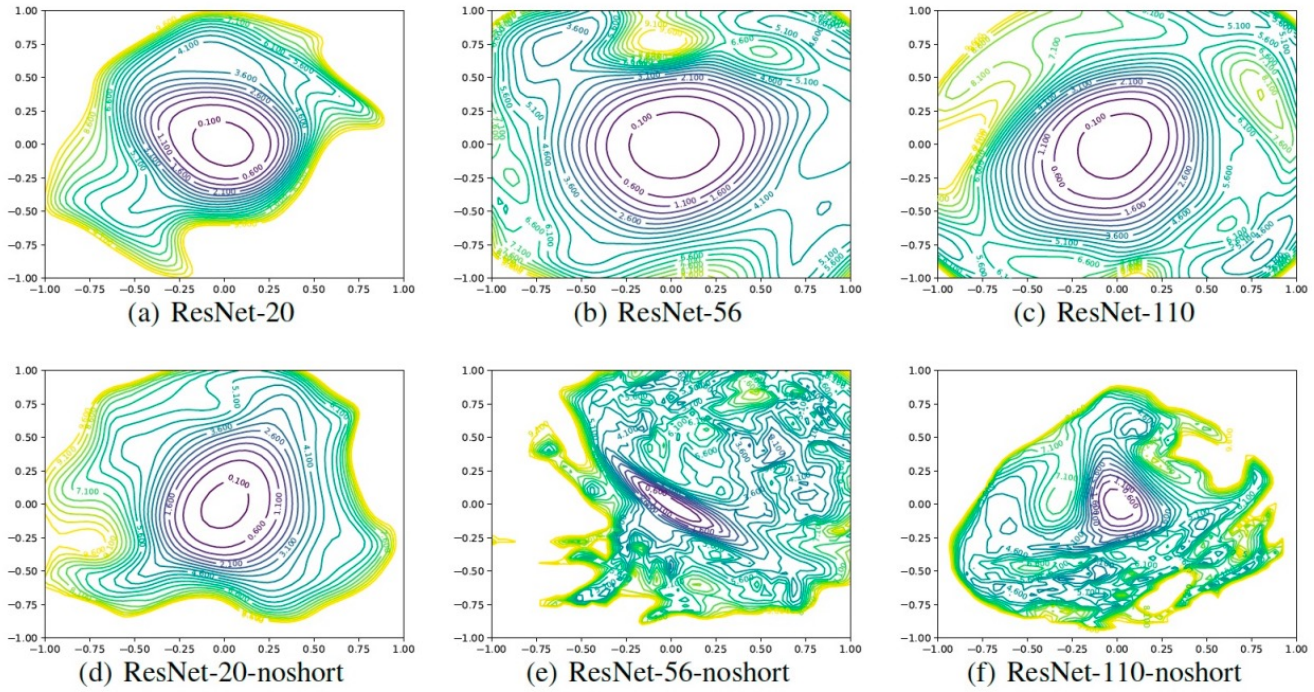
(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The vertical axis is logarithmic to show dynamic range. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

Li, et al. “Visualizing the Loss Landscape of Neural Nets”, NeurIPS 2018

ResNet

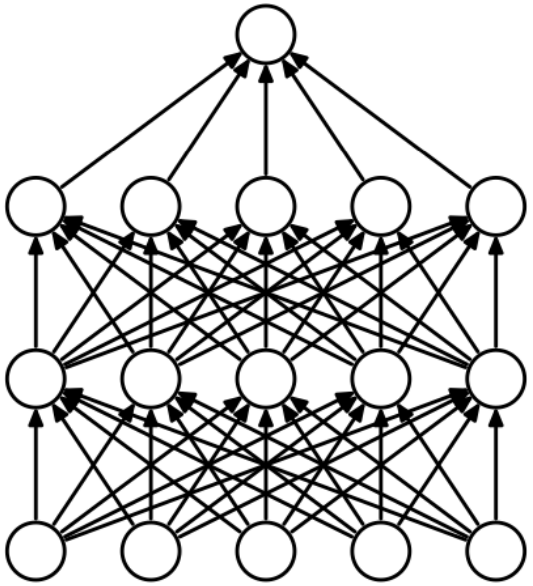
- Without residual connections, deeper nets have more chaotic landscapes



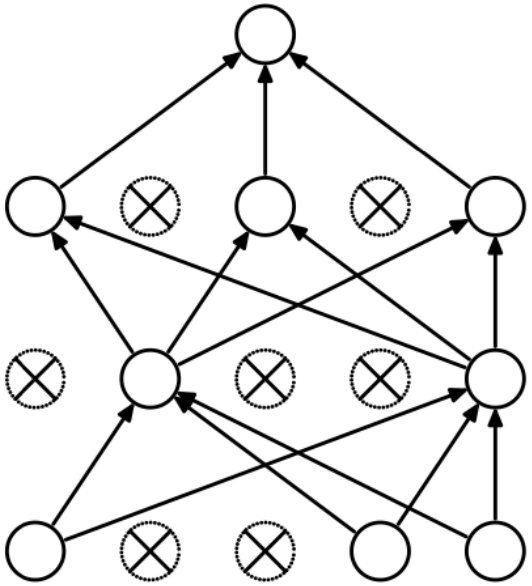
Li, et al. “Visualizing the Loss Landscape of Neural Nets”, NeurIPS 2018

Generalization: Dropout

- Dropout: drop parameters randomly during training
- Avoid overfitting: force the representation to be redundant.



(a) Standard Neural Net

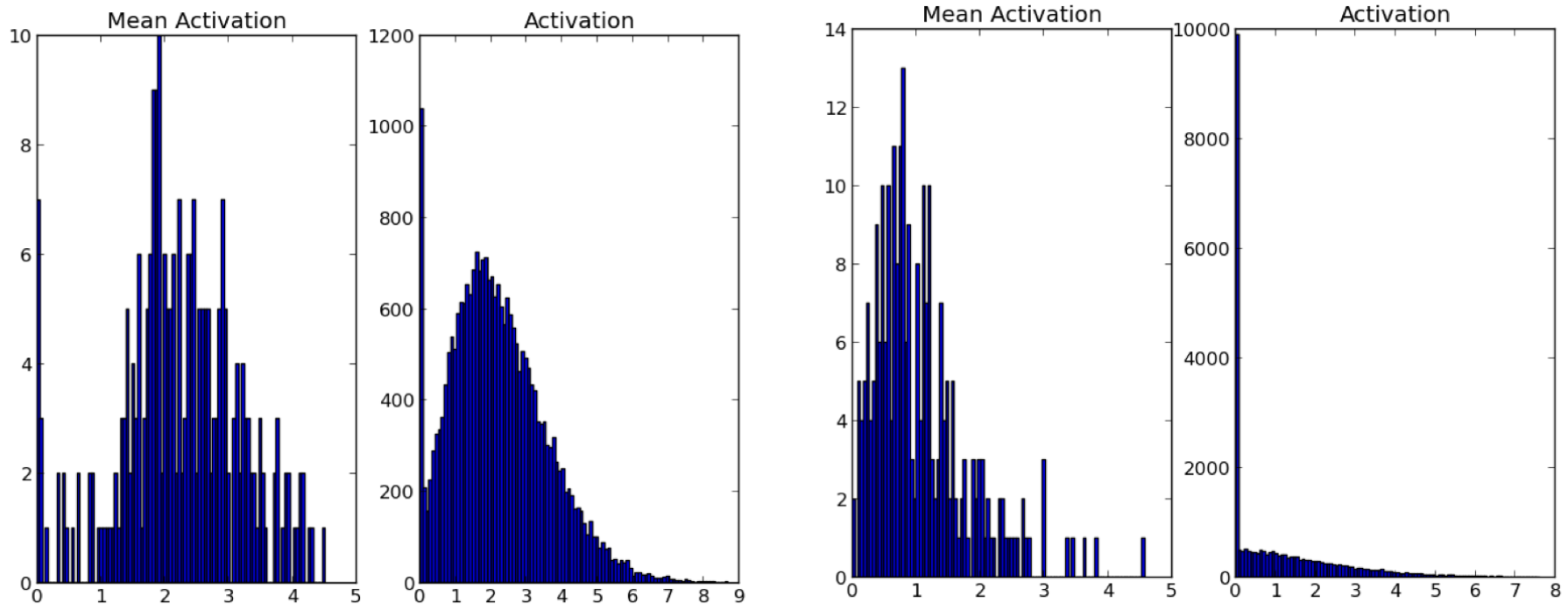


(b) After applying dropout.

$$\mathrm{w} = \begin{cases} 0, p = 0.5 \\ 1, p = 0.5 \end{cases}$$

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

Generalization: Dropout

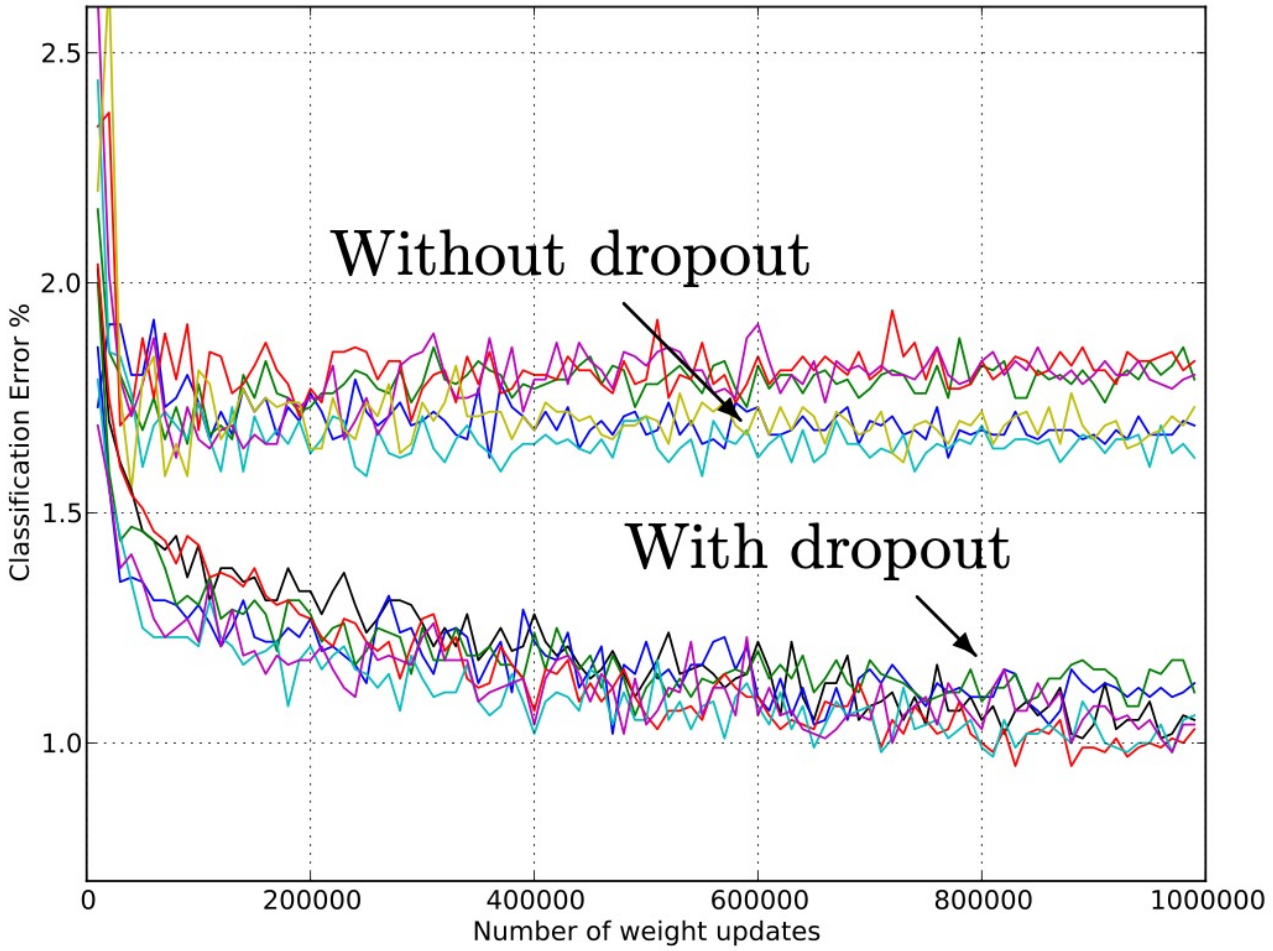


(a) Without dropout (b) Dropout with $p = 0.5$.

For each model, the histogram on the left shows the distribution of mean activations of hidden units across the minibatch. The histogram on the right shows the distribution of activations of the hidden units. **The activations of the hidden units become sparse with dropout.**

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

Generalization: Dropout

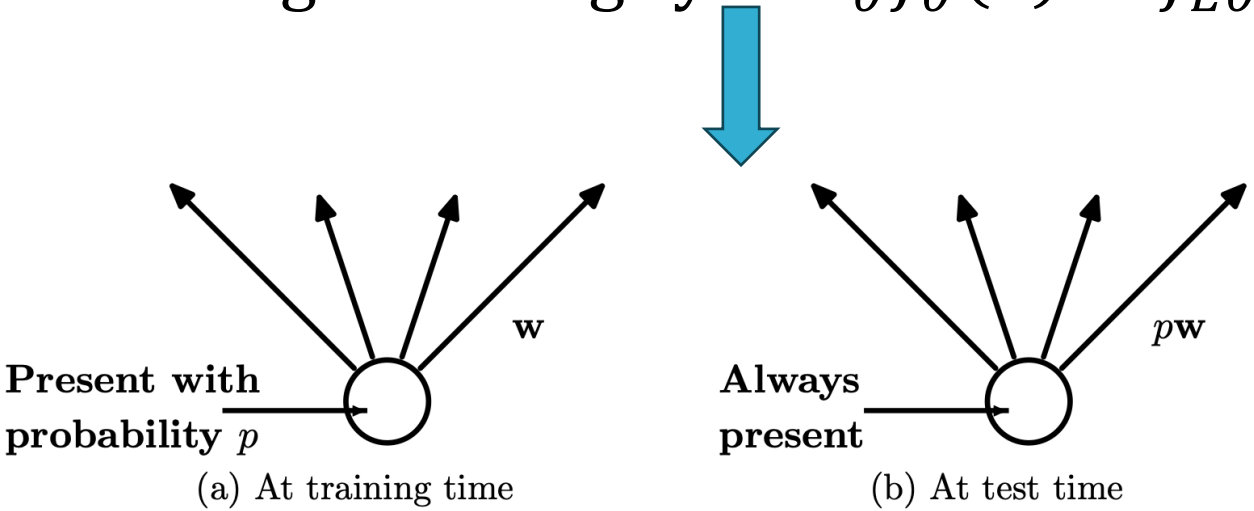


Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

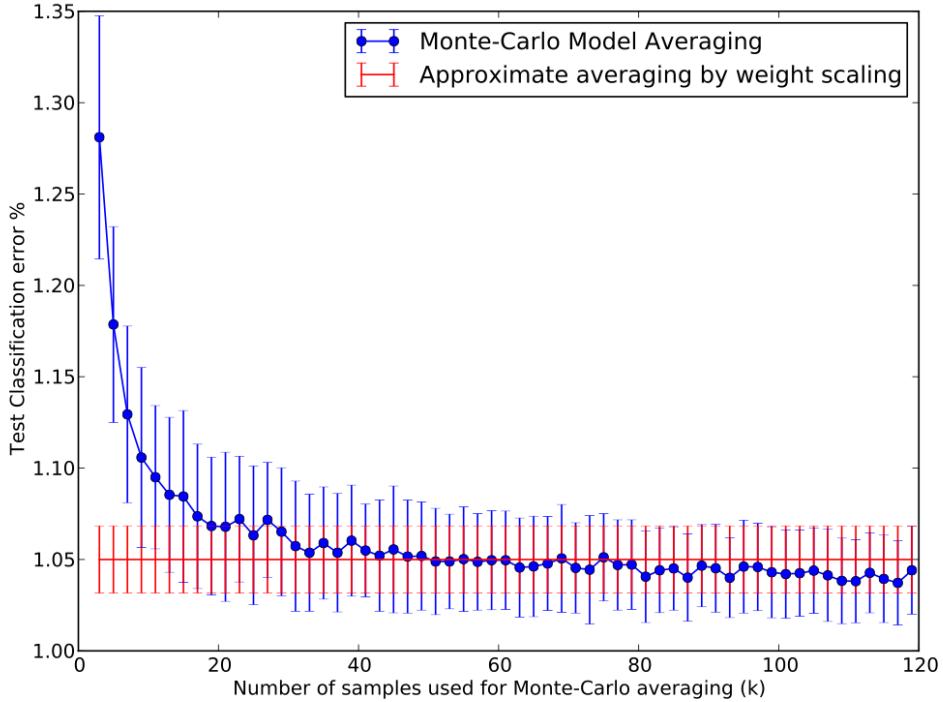
Generalization: Dropout

For testing :

1. Monte Carlo Averaging: $y = \int p(\theta) f_\theta(x) = \frac{1}{N}\sum_i f_{\theta_i}(x), \theta_i \sim p(\theta)$;
2. Weight Scaling: $y = E_\theta f_\theta(x) \approx f_{E\theta}(x)$



(a) At training time (b) At test time



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

Summary

- MLP

$$x_L = f_{\theta_L}\left(f_{\theta_{L-1}}\left(\cdots f_{\theta_1}(x_0)\right)\right)$$

$$x_i = f_{\theta_i}(x_{i-1}) = \sigma(w_i x_{i-1} + b_i)$$

- Universal Approximation Theory
- Activation function: Sign → Sigmoid → ReLU (LeakyReLU)
- Optimization: SGD,Adam , Initialization, Batch Normalization
- Generalization: Dropout
- Advanced architectures: ResNet, CNN, Transformer (to come…)