



NUS | Computing
National University
of Singapore

IT5005 Artificial Intelligence

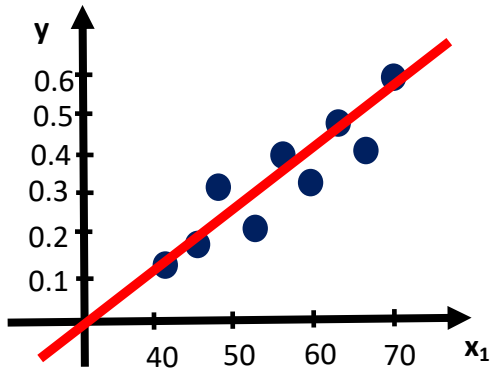
Sirigina Rajendra Prasad
AY2025/2026: Semester 1

MLPs: Miscellaneous

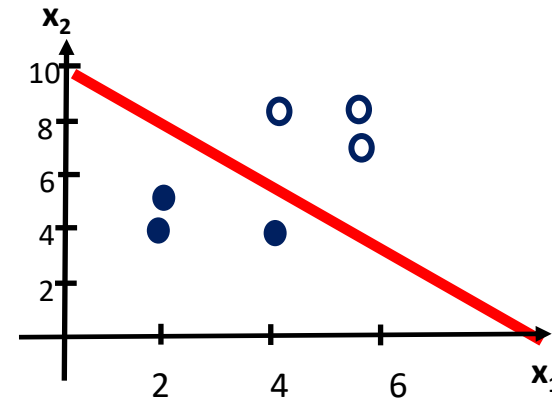
Supervised Learning

- Objective:
 - Given training data (x, y) , where x is the input and y is the output, find the mapping $h: x \rightarrow y$.
 - h is called model or hypothesis function
 - If you are given a new input x_{new} , predict the output using trained model h

Regression: y is continuous



Classification: y is discrete



Function Approximation using Neural Network

Objective: minimize the approximation error or loss (or to make $\hat{y} = h(x) = y$)

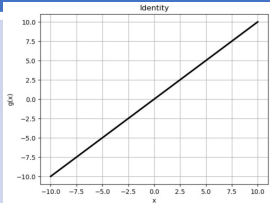
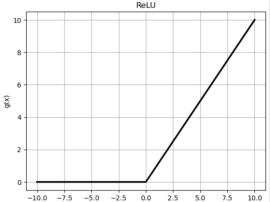
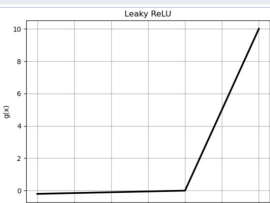
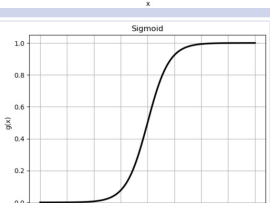
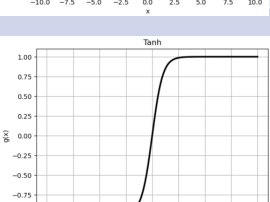
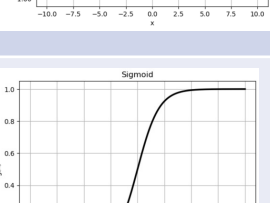


$$\hat{y} = \mathbf{g}^{[L]} \left(\mathbf{f}^{[L]} \left(\mathbf{g}^{[L-1]} \left(\dots \left(\mathbf{g}^{[l]} \left(\mathbf{f}^{[l]} \left(\mathbf{g}^{[l-1]} \left(\dots \left(\mathbf{g}^{[1]} \left(\mathbf{f}^{[1]}(x) \right) \right) \right) \right) \right) \right) \right) \right) \right)$$

Parameters

- Number of layers (L)
- Number of perceptrons per layer
- Activation function ($g(\cdot)$)
- Loss function ($L(\cdot)$)
- Backpropagation
 - Optimization Algorithm
 - Algorithm-related parameters
 - Learning Rate
 - Momentum, etc.
- Weights initialization
- Normalization
- Dropouts, etc.

Activation Functions

Activation Function	Range	Use case	
Identity: $g(x) = x$	$[-\infty, +\infty]$	Linear Regression	
ReLU: $g(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$	$[0, +\infty]$	Linear Regression Classification	
Leaky ReLU: $g(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{else} \end{cases}$	$[-\infty, +\infty]$	Linear Regression Classification	
Sigmoid: $g(x) = \sigma(x) = \frac{1}{1+\exp(-x)}$	$[0, 1]$	Linear Regression Classification	
Tanh: $g(x) = \tanh(x) = \frac{\exp(2x)-1}{\exp(2x)+1}$	$[-1, 1]$	Linear Regression Classification	
Softmax: $g(x) = \frac{\exp(-x_i)}{\sum_j \exp(-x_j)}$	$[0, 1]$	Multiclass Classification	

Loss Functions

Loss Function	Formula	Use case
Mean Square Error	$J(w) = \frac{1}{2m} \sum_{i=1}^m (\mathbf{w}^T \mathbf{x}^{(i)} - y^{(i)})^2$	Linear Regression
Mean Absolute Error	$J(w) = \frac{1}{2m} \sum_{i=1}^m \mathbf{w}^T \mathbf{x}^{(i)} - y^{(i)} $	Linear Regression
Huber Loss	$J_i(\mathbf{w}) = \begin{cases} \frac{1}{2} \sum_{i=1}^m (y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)})^2 & \text{if } y - \hat{y} < \delta \\ \delta \left(\sum_{i=1}^m y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)} - \frac{1}{2} \delta \right) & \text{Otherwise} \end{cases}$	Linear Regression
Tanh-Cos Loss	$J(w) = \sum_{i=1}^m \log(\cosh(y^{(i)} - \hat{y}^{(i)}))$	Linear Regression
Binary Cross Entropy Loss	$J(\mathbf{w}) = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log \sigma(\mathbf{w}^T \mathbf{x}^{(i)}) + (1 - y^{(i)}) \log (1 - \sigma(\mathbf{w}^T \mathbf{x}^{(i)}))$	Linear Regression Binary Classification
Multiclass Cross-Entropy Loss	$J(\mathbf{w}) = -\frac{1}{m} \sum_{i=1}^m \sum_{c=1}^C y_c^{(i)} \log(\hat{y}_c^{(i)})$	Multiclass Classification
Hinge Loss	$J(w) = \frac{1}{m} \sum_{i=1}^m \max(0, 1 - y^{(i)} \hat{y}^{(i)})$	Binary Classification

Optimization for Model Training

Gradient Descent

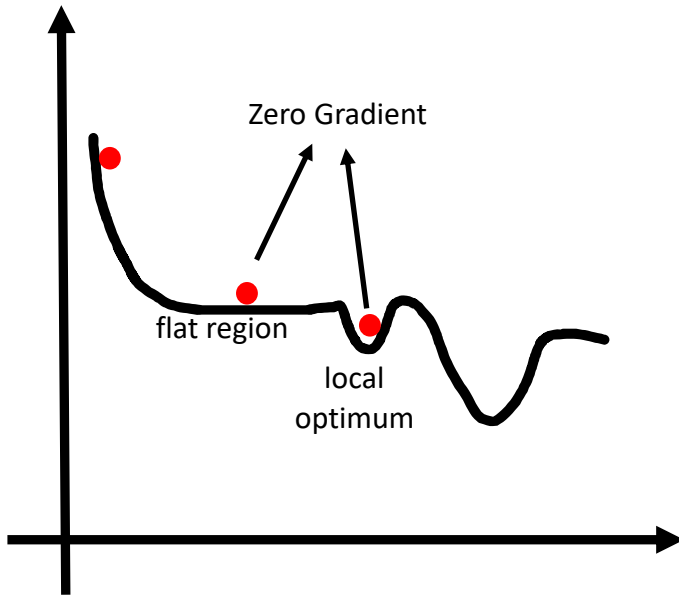
learning rate

Loss function

$$w_i := w_i - \alpha \frac{\partial J(\mathbf{w})}{\partial w_i}$$

simultaneously
for all w_i

Optimizers: Issues

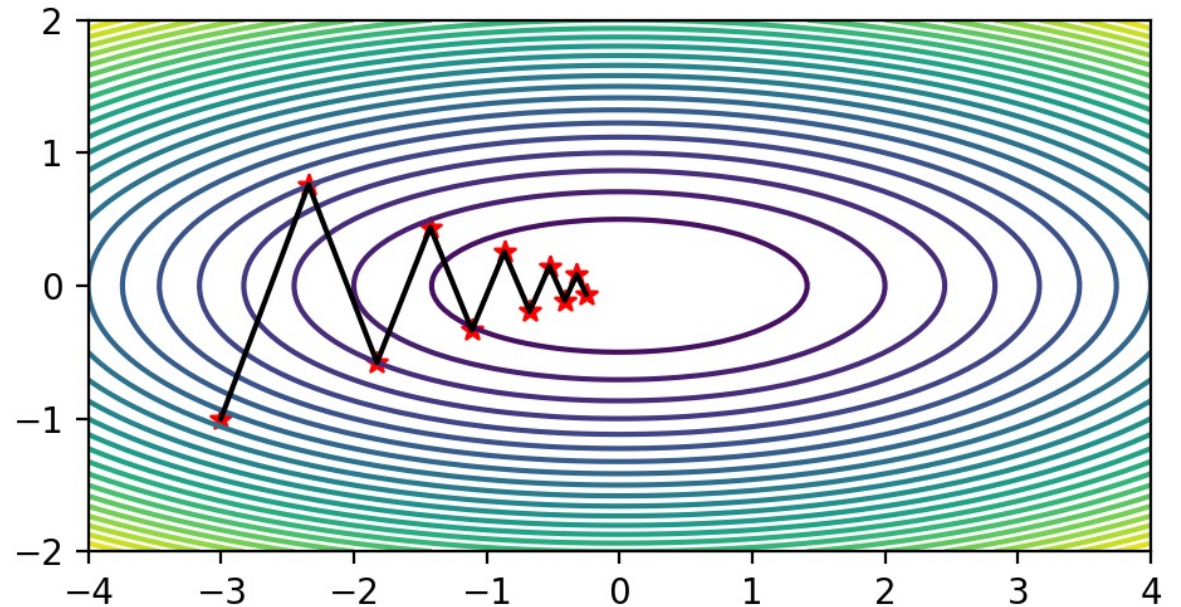


Issue:

Nonconvex loss function

Solution:

Momentum-based methods to overcome flat regions and local minima



Issue:

Slow convergence due to oscillations

Solution:

Adaptive and unique learning rate for each weight to damp oscillations

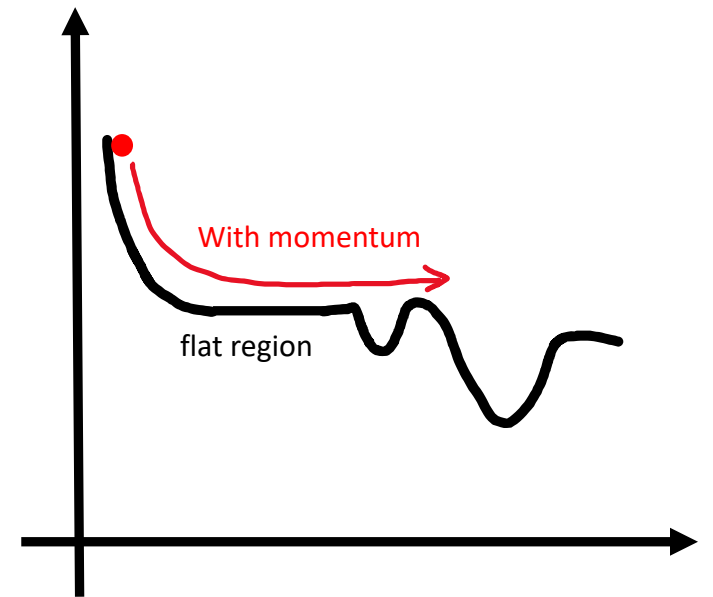
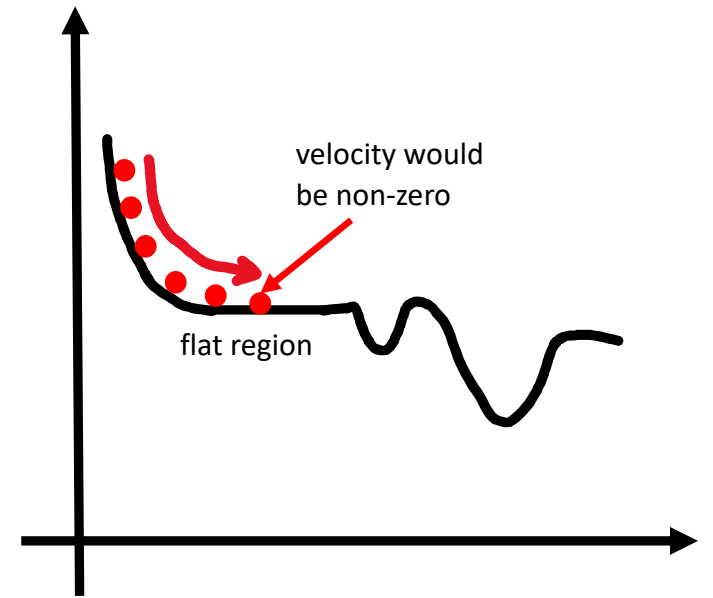
Momentum-based GD

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{V}_{t+1}$$

$$\mathbf{V}_0 = \mathbf{0}$$

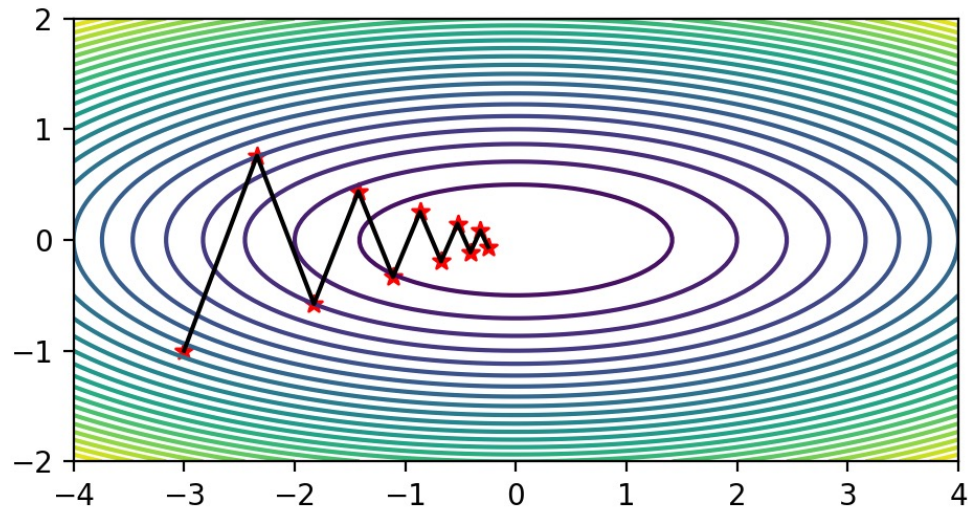
$$\mathbf{V}_{t+1} = \beta \mathbf{V}_t - \alpha \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} \right)_{\mathbf{w}=\mathbf{w}_t}$$

where $\beta \in (0,1)$ is momentum coefficient
 α is the learning rate

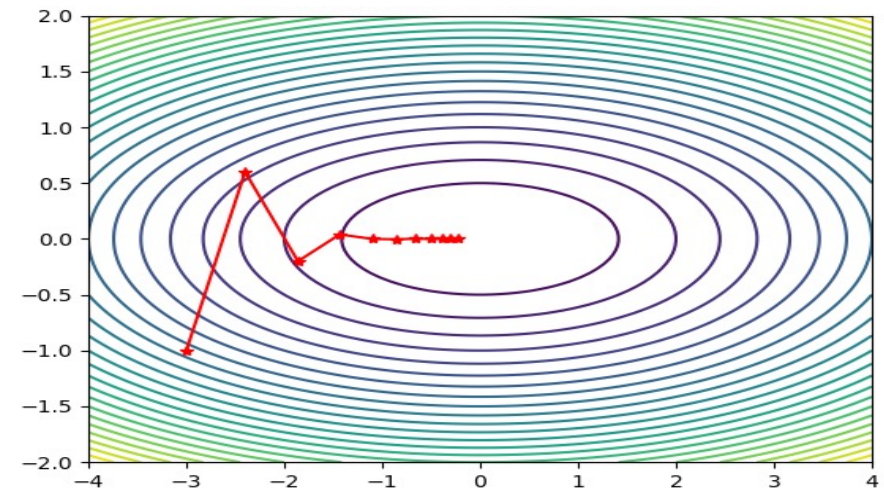


Steepest Vs Momentum-based Gradient Descent

Steepest Descent



Momentum-based GD



AdaGrad Algorithm

- The update for i -th variable:

$$w_{i,t+1} = w_{i,t} - \frac{\alpha}{\sqrt{A_i + \epsilon}} \left(\frac{\partial J(w)}{\partial w_{i,t}} \right)$$

Adaptive learning rate

Same as steepest descent

$$\text{where } A_i \Leftarrow A_i + \left(\frac{\partial J(w)}{\partial w_{i,t}} \right)^2$$

- Derivative is scaled inversely with $\sqrt{A_i}$
 - Parameter A_i keeps track of the aggregated squared magnitude of the partial derivative with respect to each parameter
 - If the gradient of a component is inconsistent and varying wildly (for example, between +100 and -100), gradient of that component will be penalized more
- Aggressive decaying of learning rate
 - Because A_i keeps accumulating with iterations
 - No update in weights if number of iterations are high

RMSProp Algorithm

- The update for i -th variable:

$$w_{i,t+1} = w_{i,t} - \frac{\alpha}{\sqrt{A_i + \epsilon}} \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)$$

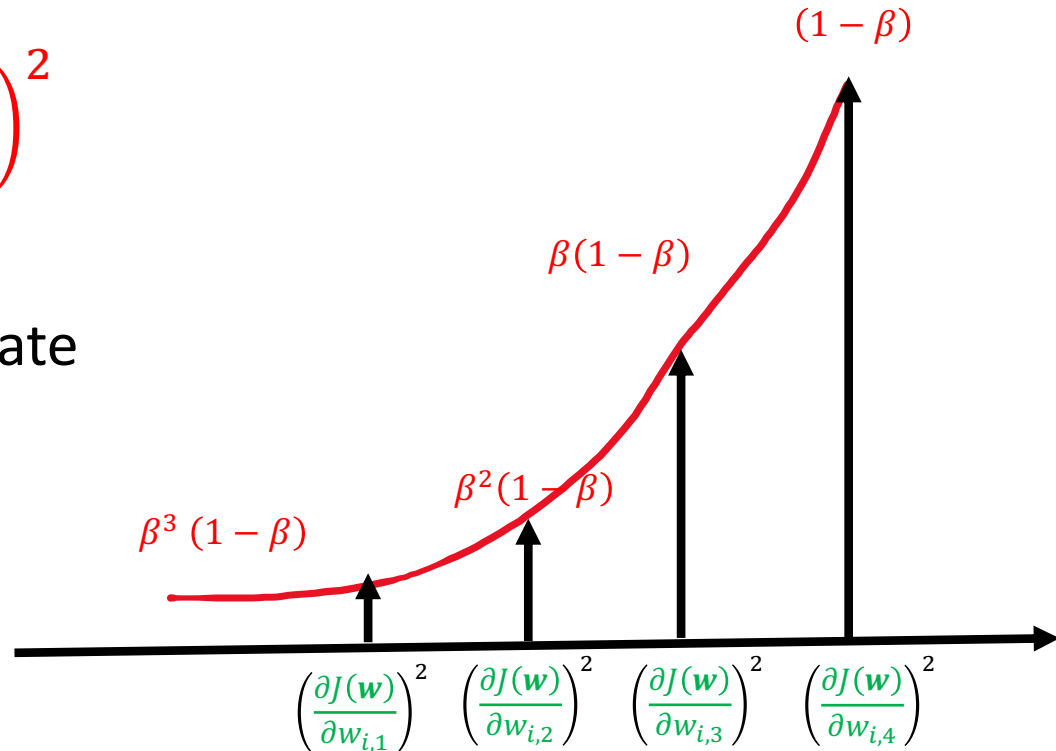
where

$$A_i \Leftarrow \beta A_i + (1 - \beta) \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)^2$$

- Less aggressive decaying of learning rate

Different learning rate
for each variable w_i

Same as steepest descent

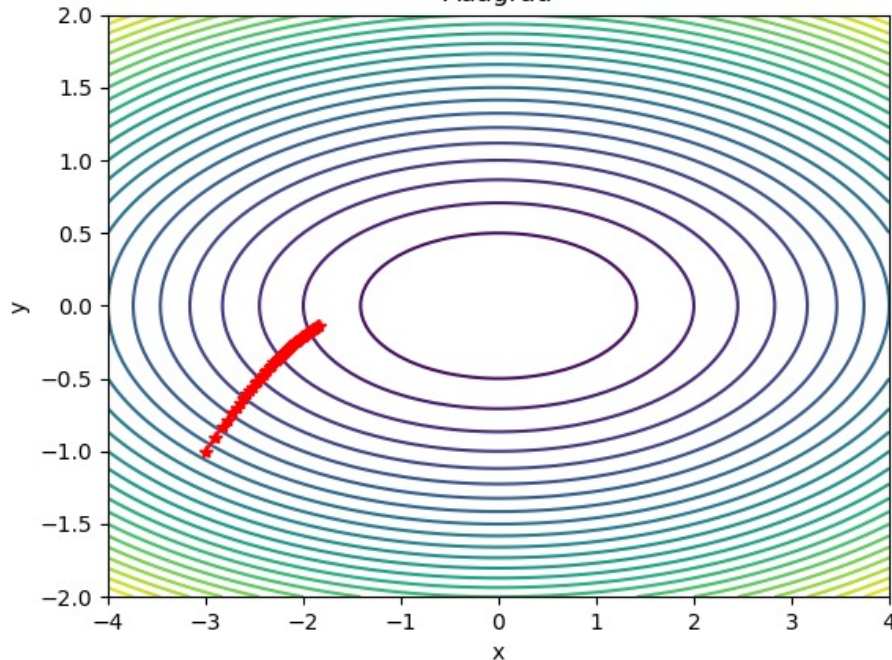


AdaGrad vs RMSProp Algorithm

$$w_{i,t+1} = w_{i,t} - \frac{\alpha}{\sqrt{A_i + \epsilon}} \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)$$

$$A_i \leftarrow A_i + \left(\frac{\partial f(x)}{\partial x_{i,t}} \right)^2$$

Adagrad

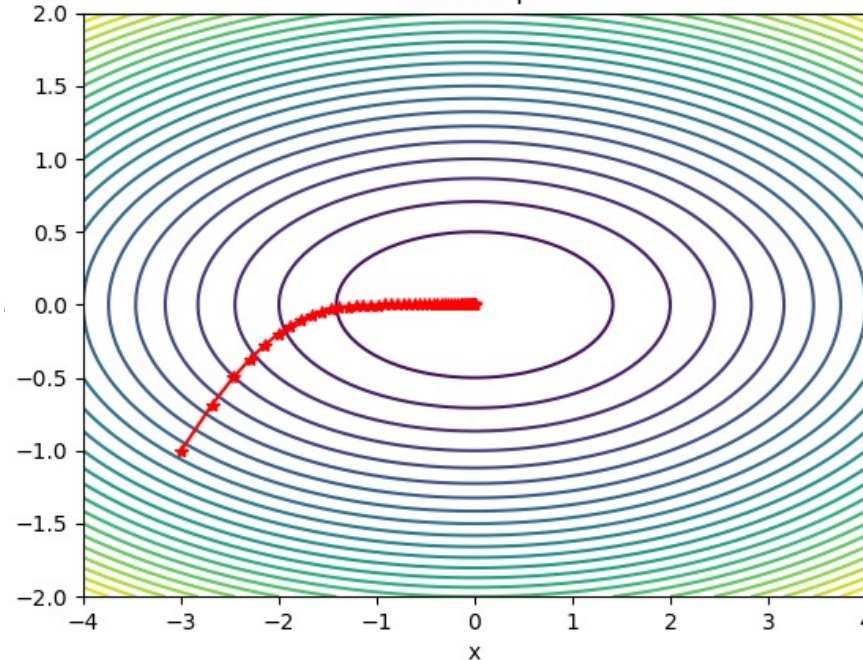


Accumulation of A_i leads to aggressive decay of $\frac{\alpha}{\sqrt{A_i + \epsilon}}$

$$w_{i,t+1} = w_{i,t} - \frac{\alpha}{\sqrt{A_i + \epsilon}} \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)$$

$$A_i \leftarrow \beta A_i + (1 - \beta) \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)^2$$

RMSProp



Exponential averaging of A_i leads to less aggressive decay of $\frac{\alpha}{\sqrt{A_i + \epsilon}}$

Adam Algorithm (Momentum+RMSProp)

- The update for i -th variable:

$$w_{i,t+1} = w_{i,t} - \frac{\alpha_t}{\sqrt{A_i + \epsilon}} F_i$$

Adaptive learning rate α_t and Velocity F_i are indicated by arrows pointing to their respective terms in the equation.

where $F_i \leftarrow \rho_f F_i + (1 - \rho_f) \frac{\partial J(w)}{\partial w_{i,t}}$ ← Momentum

$$A_i \leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial J(w)}{\partial w_{i,t}} \right)^2$$

← RMSProp

$$\alpha_t = \alpha \left(\frac{\sqrt{1 - \rho_f^t}}{1 - \rho_f^t} \right)$$

Bias term to offset the effects of initialization of F_i and A_i

- Penalizes inconsistent components with $\sqrt{A_i + \epsilon}$ term (similar to RMSProp algorithm)
- Includes momentum in the update to overcome flat regions and local minima

PyTorch:

Betas = (β_1, β_2)

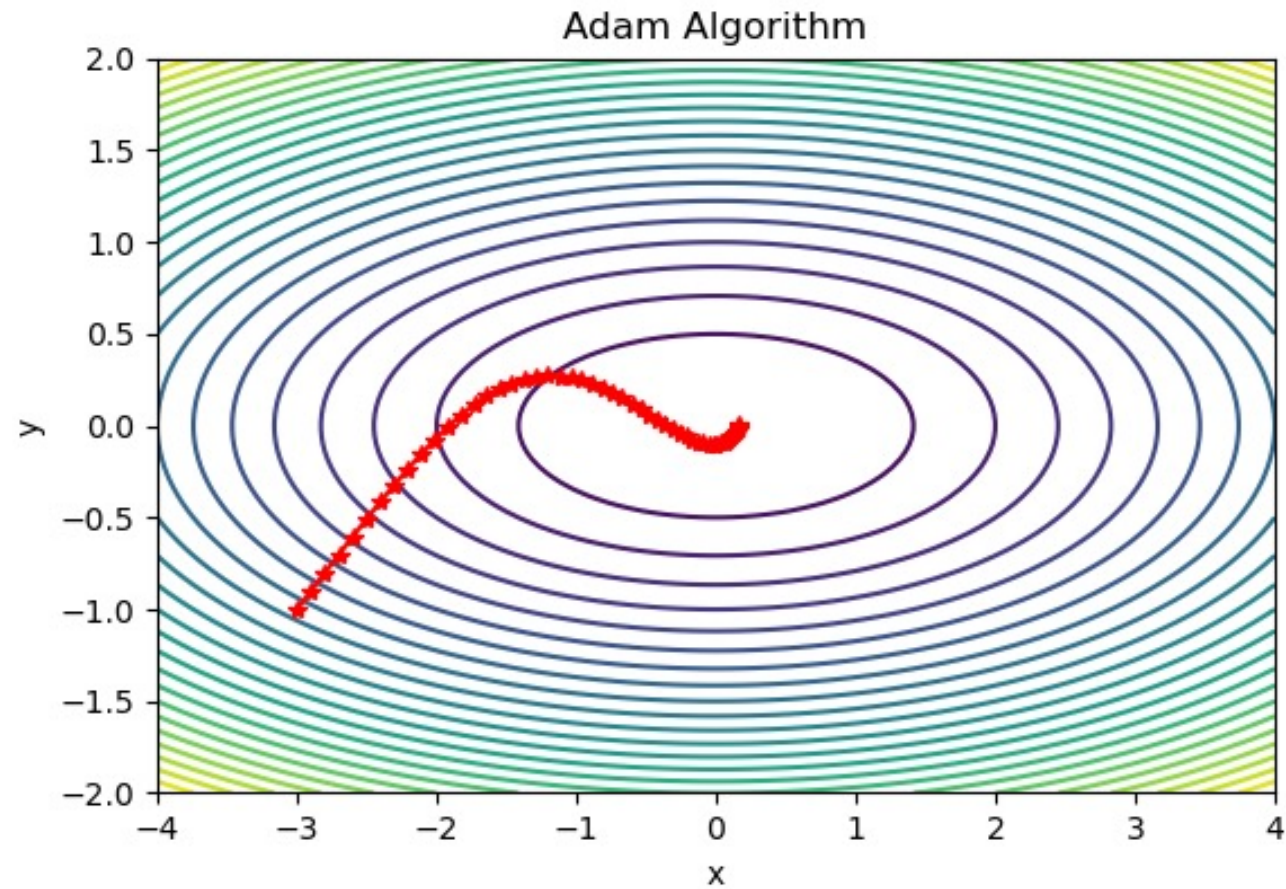
ρ_f : β_1

ρ : β_2

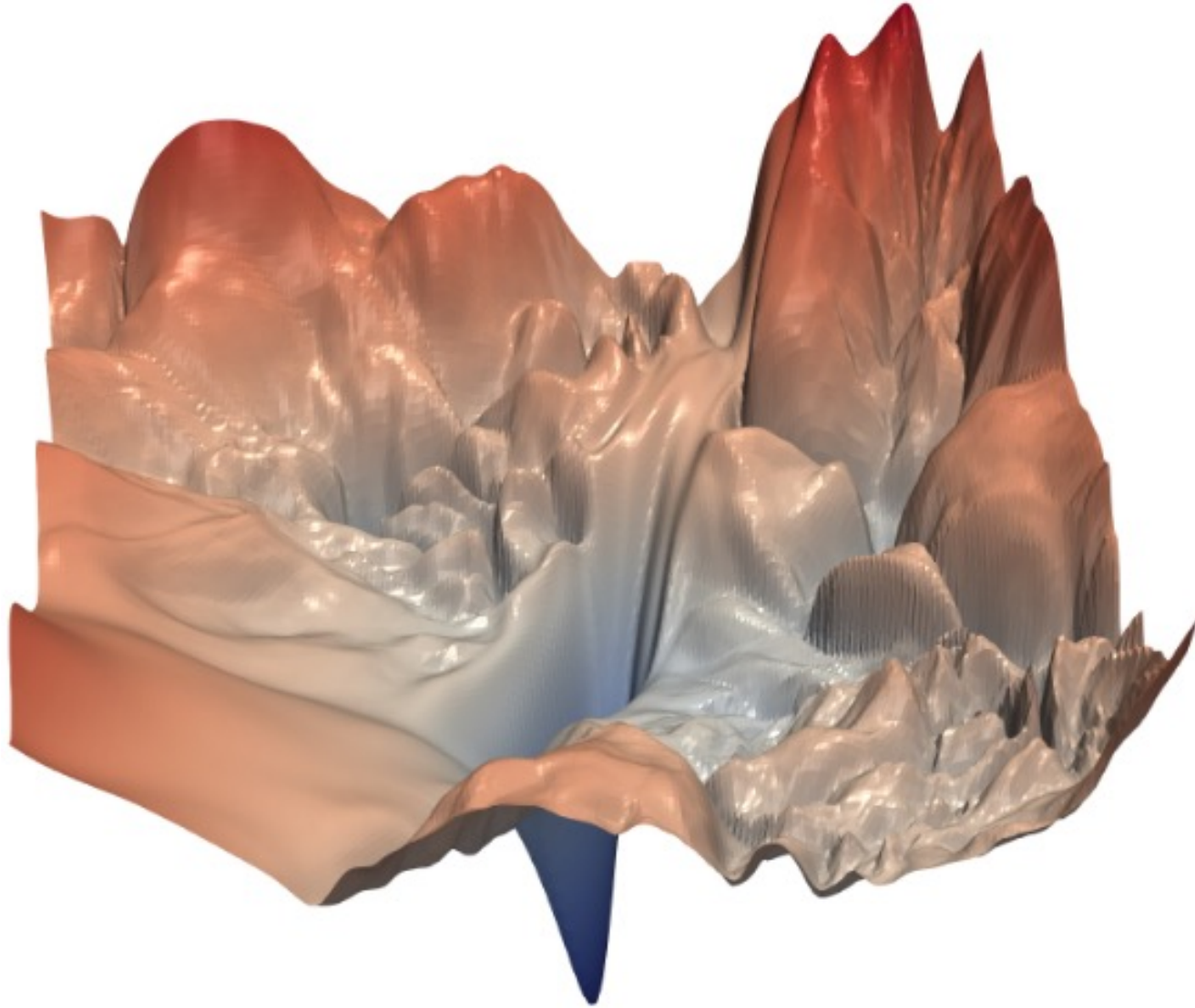
α_t : lr (learning rate)

ϵ : eps

Adam Algorithm



Typical Loss Surface in Neural Network



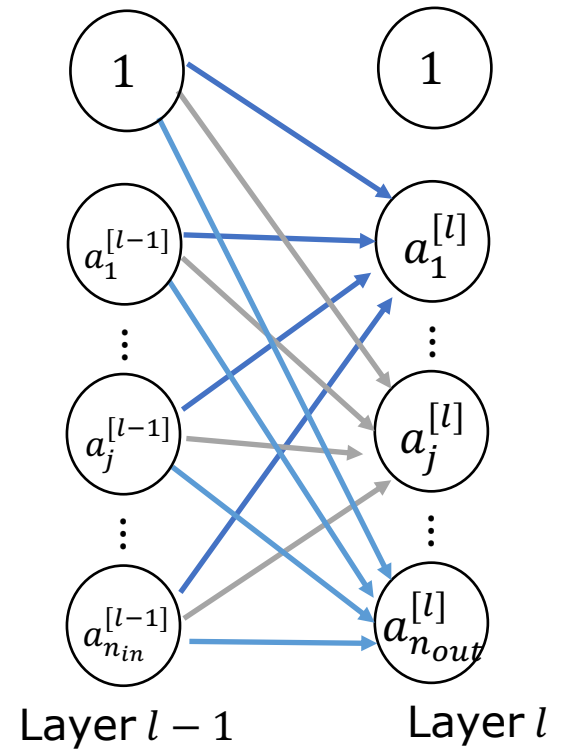
Optimizers: Summary

Algorithms	Update Rule	Characteristics
Momentum-based GD	$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{V}_{t+1}$ $\mathbf{V}_{t+1} = \beta \mathbf{V}_t - \alpha \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} \right)_{\mathbf{w}=\mathbf{w}_t}$	<ul style="list-style-type: none"> • Uses accumulation of past gradients to overcome local optimal point • Same learning rate for all weights is not good for convergence
AdaGrad	$w_{i,t+1} = w_{i,t} - \frac{\alpha}{\sqrt{A_i + \epsilon}} \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)$ $A_i \leftarrow A_i + \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)^2$	<ul style="list-style-type: none"> • Unique learning rate $\frac{\alpha}{\sqrt{A_i + \epsilon}}$ for each weight • Learning rate is inversely proportional to accumulated gradients, i.e., $A_i + \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)^2$ • Aggressive decaying of learning rate • Cannot overcome local optimal point
RMSProp	$w_{i,t+1} = w_{i,t} - \frac{\alpha}{\sqrt{A_i + \epsilon}} \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)$ $A_i \leftarrow \beta A_i + (1 - \beta) \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)^2$	<ul style="list-style-type: none"> • Unique learning rate $\frac{\alpha}{\sqrt{A_i + \epsilon}}$ for each feature • Learning rate is inversely proportional to exponentially weighted moving average of past gradients, i.e., $\beta A_i + (1 - \beta) \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)^2$ • Less aggressive decaying of learning rate which is good for convergence • Cannot overcome local optimal point
Adam (Momentum+RMSProp)	$w_{i,t+1} = w_{i,t} - \frac{\alpha_t}{\sqrt{A_i + \epsilon}} F_i$ $F_i \leftarrow \rho_f F_i + (1 - \rho_f) \frac{\partial J(\mathbf{w})}{\partial w_{i,t}}$ $A_i \leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial J(\mathbf{w})}{\partial w_{i,t}} \right)^2$ $\alpha_t = \alpha \left(\frac{\sqrt{1 - \rho^t}}{1 - \rho_f^t} \right)$	<ul style="list-style-type: none"> • Can overcome local optima due to momentum (F_i) • Unique learning rate $\frac{\alpha_t}{\sqrt{A_i + \epsilon}}$ for each feature • Less aggressive decaying of learning rate which is good for convergence

Why is training DNN hard?

- Vanishing/exploding gradients as depth increases
- Internal covariance shift
 - Changing activation distribution across layers
- Optimization pathologies
 - Saddle points, ill-conditioned curvature
- Overfitting

Slow convergence
Unstable loss
Sensitivity to learning rate



$$f^{[l]} \equiv (\mathbf{W}^{[l]})^T \mathbf{a}^{[l-1]}$$

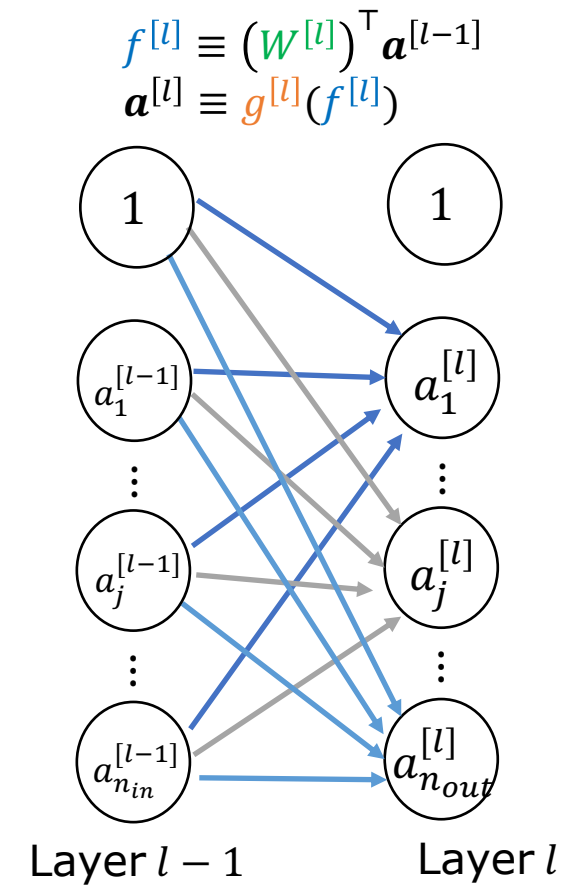
$$\mathbf{a}^{[l]} \equiv \mathbf{g}^{[l]}(f^{[l]})$$

$$\hat{y}'(\mathbf{W}^{[l-1]}) = \mathbf{a}^{[l-2]} (\boldsymbol{\delta}^{[l-1]})^T$$

$$\boldsymbol{\delta}^{[l-1]} = \mathbf{g}'^{[l-1]}(f^{[l-1]}) \mathbf{W}^{[l]} \boldsymbol{\delta}^{[l]}$$

Potential Solutions

- Weight (parameter) Initialization
- Normalization
- Skip Connections
- Dropout for regularization



$$\hat{y}'(W^{[l-1]}) = a^{[l-2]} (\delta^{[l-1]})^T$$

$$\delta^{[l-1]} = g'^{[l-1]}(f^{[l-1]}) W^{[l]} \delta^{[l]}$$

Weight Initialization: Symmetry Problem

- Suppose all weights are initialized to same value (for example to '1')
 - All perceptrons in hidden layers output same value
 - Derivatives of all weights would be same
 - All weight updates would be same
 - In other words, only one perceptron is enough and all other perceptrons would be redundant
- Initialize weights with random values to avoid symmetry problem
- How to choose random values?
 - Which distribution and parameters of distribution (mean and variance)?

Weight Initialization: Gradient Issues

- **Activations:**

- Too large variance leads to exploding gradient and too small variance leads to vanishing gradient

- **Objective:**

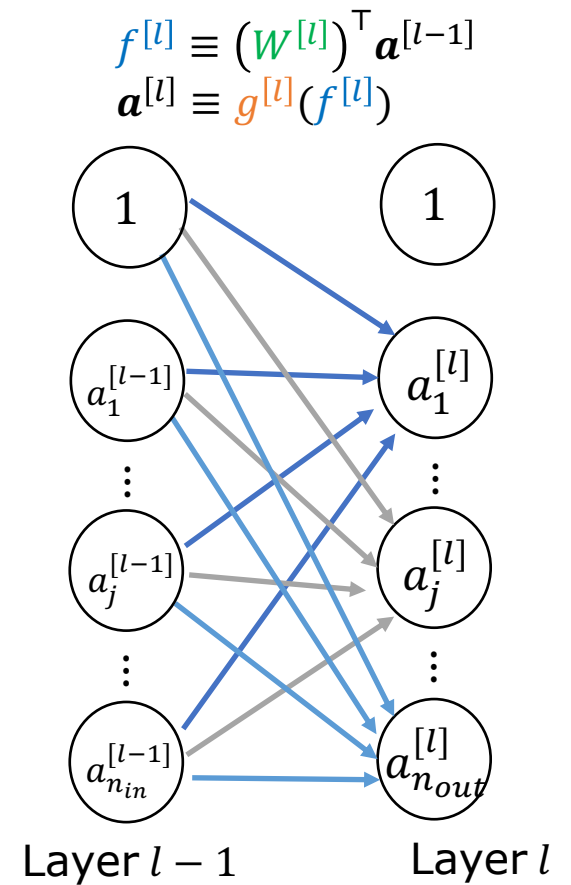
- Forward Pass: $Var(\mathbf{a}^{[l]}) = Var(\mathbf{a}^{[l-1]})$
- Backward Pass: $Var(\hat{y}'(\mathbf{a}^{[l]})) = Var(\hat{y}'(\mathbf{a}^{[l-1]}))$

- **What we have?**

- Forward Pass: $Var(\mathbf{a}^{[l]}) \approx c_g n_{in} \sigma_W^2 Var(\mathbf{a}^{[l-1]})$, where c_g is the ratio of variance of output and input of nonlinear function
- Backward Pass: $Var(\hat{y}'(\mathbf{a}^{[l-1]})) \approx m_g n_{out} \sigma_W^2 Var(\hat{y}'(\mathbf{a}^{[l]}))$,
where $m_g = E\left(\left(\mathbf{g}'^{[l]}(\mathbf{f}^{[l]})\right)^2\right)$

- **So, what to do?**

- $c_g n_{in} \sigma_W^2 = m_g n_{out} \sigma_W^2 = 1$



$$\hat{y}'(\mathbf{a}^{[l]}) = \frac{\partial \hat{y}}{\partial \mathbf{a}^{[l]}} \quad \hat{y}'(\mathbf{a}^{[l-1]}) = \frac{\partial \hat{y}}{\partial \mathbf{a}^{[l-1]}}$$

$$\begin{aligned} \hat{y}'(\mathbf{a}^{[l-1]}) &= \frac{\partial \mathbf{f}^{[l]}}{\partial \mathbf{a}^{[l-1]}} \frac{\partial \mathbf{g}^{[l]}}{\partial \mathbf{f}^{[l]}} \hat{y}'(\mathbf{a}^{[l]}) \\ &= \mathbf{W}^{[l]} \mathbf{g}'^{[l]}(\mathbf{f}^{[l]}) \hat{y}'(\mathbf{a}^{[l]}) \end{aligned}$$

Weight Initialization: Strategies

Strategy	Recommended activations	Weight distribution / scale
Xavier (Glorot) Uniform	tanh, sigmoid, linear	$W \sim U\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$
Xavier (Glorot) Normal	tanh, sigmoid, linear	$W \sim N\left(0, \frac{2}{n_{in} + n_{out}}\right)$
He (Kaiming) Uniform	ReLU, GELU, ELU (ReLU-family)	$W \sim U\left(-\frac{2}{n_{in}}, \frac{2}{n_{in}}\right)$
He (Kaiming) Normal	ReLU, GELU, ELU (ReLU-family)	$W \sim N\left(0, \frac{2}{n_{in}}\right)$

n_{in} : # of input neurons (aka fan_{in}), n_{out} : # of output neurons (aka fan_{out})

Normalization: Why we need it?

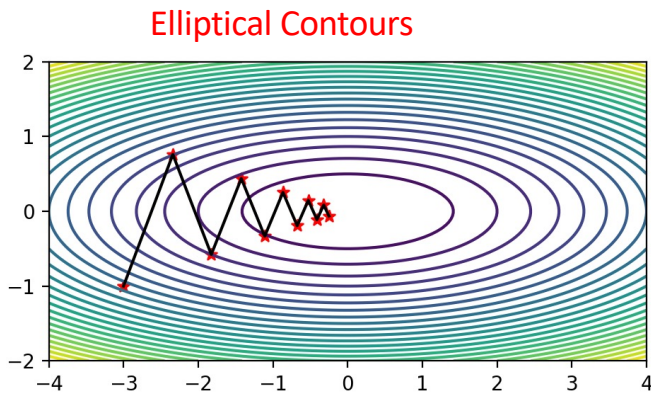
- Stabilize and accelerate training
 - Improves conditioning
 - Enables larger learning rates with faster convergence.
- Improve gradient flow
 - Reduces vanishing/exploding gradients, allowing deeper networks
 - Make training less sensitive to weight initialization and learning rate choices.
- Decouple scale from representation
 - Makes networks robust to input scale and distribution shifts across layers

Normalization Techniques

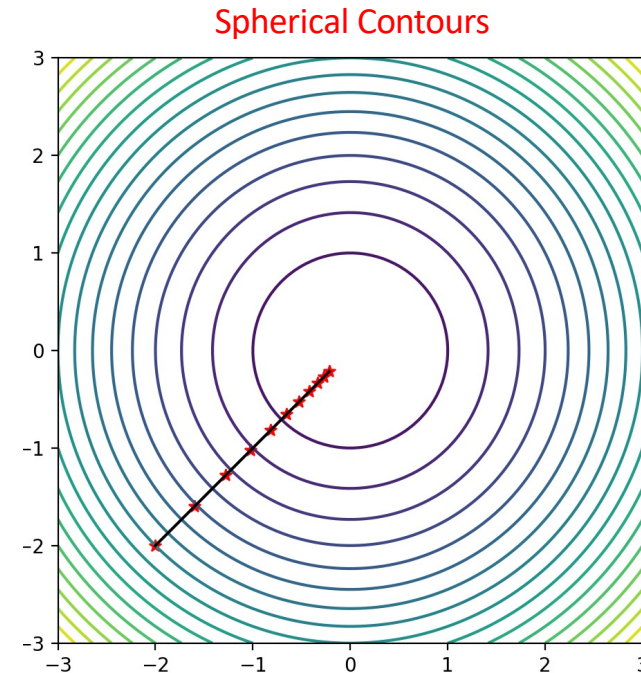
- Input Normalization
- Batch Normalization (BatchNorm)
- Layer Normalization (LayerNorm)
- RMSNorm
- Instance Normalization (InstanceNorm)
- Group Normalization (GroupNorm)
- Weights Normalization

Data Normalization

- Normalize the inputs to similar scale: $[-1, 1]$ *or* $[0, 1]$



Unnormalized Data



Normalized Data

Data Normalization

Minibatch of size $n = 5$

	x_1	x_2	x_3	x_4
$x_i^{(1)}$	2	80	400	0.5
$x_i^{(2)}$	4	90	300	0.7
$x_i^{(3)}$	6	70	500	0.4
$x_i^{(4)}$	8	85	600	0.6
$x_i^{(5)}$	10	95	200	0.8



Normalized Minibatch of size $n = 5$

\bar{x}_1	\bar{x}_2	\bar{x}_3	\bar{x}_4
-1.414	-0.465	0	-0.707
-0.707	0.698	-0.707	0.707
0	-1.628	0.707	-1.414
0.707	0.116	1.414	0
1.414	1.279	-1.414	1.414

x_1	x_2	x_3	x_4
$m_1 = 6$	$m_2 = 84$	$m_3 = 400$	$m_4 = 0.6$
$\sigma_1 = 2.828$	$\sigma_2 = 8.602$	$\sigma_3 = 141.421$	$\sigma_4 = 0.141$

Batch Normalization

Minibatch: $n = 5$ samples and $m = 4$ features

- For each mini-batch of size n :

- Calculate **per-feature** mean and variance :

- $\mu_i = \frac{1}{n} \sum_{j=1}^n x_i^{(j)}, i = \{1, 2, \dots, m\}$

- $\sigma_i^2 = \frac{1}{n} \sum_{j=1}^n \left(x_i^{(j)} - \mu_i \right)^2, i = \{1, 2, \dots, m\}$

- Normalize the data

- $\bar{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$, where ϵ is a small constant for numerical stability

- Scale and shift with learnable parameters $\gamma \in \mathbb{R}^{n \times 1}$ and $\beta \in \mathbb{R}^{n \times 1}$

- $y_i = \gamma \circ \bar{x}_i + \beta$

- Learnable parameters allows undoing of normalization if needed

- Uses **running-average** and **running-variance** from training data during **eval** or inference mode (refer code for details)

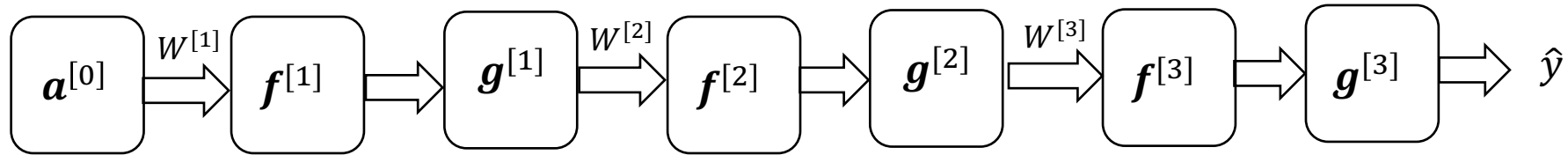
	x_1	x_2	x_3	x_4
$x_i^{(1)}$	2	80	400	0.5
$x_i^{(2)}$	4	90	300	0.7
$x_i^{(3)}$	6	70	500	0.4
$x_i^{(4)}$	8	85	600	0.6
$x_i^{(5)}$	10	95	200	0.8
	μ_1 σ_1^2	μ_2 σ_2^2	μ_3 σ_3^2	μ_4 σ_4^2

$$\mathbf{x}_i = \begin{bmatrix} x_i^{(1)} \\ \vdots \\ x_i^{(n)} \end{bmatrix} \in \mathbb{R}^{n \times 1}$$

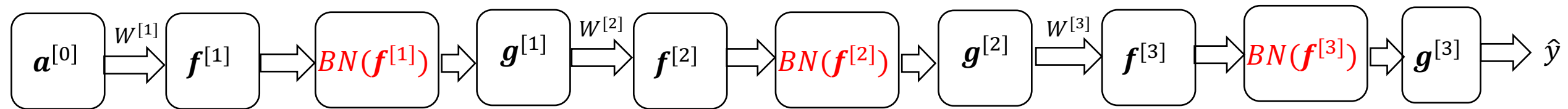
$$\bar{\mathbf{x}}_i \in \mathbb{R}^{n \times 1}$$

Batch Normalization in NN

Neural Network without Batch Normalization



Neural Network with Batch Normalization



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., sioffe@google.com

Christian Szegedy
Google Inc., szegedy@google.com

<https://arxiv.org/pdf/1502.03167>

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

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

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

Why batch normalization?

- Stabilizes Learning by mitigating *internal covariate shift*
- Acts as a regularizer
 - Reduces the need for dropout
- Enables Higher Learning Rates
 - Stable distributions allow large learning rates
 - Speed up convergence
- Improves Gradient Flow
 - Prevents activations from becoming too large or small
 - Avoids vanishing or exploding gradients

Layer Normalization

Minibatch: $n = 5$ samples and $m = 4$ features

- For each sample with m features:
 - Calculate **per-sample** mean and variance:

- $\mu_i = \frac{1}{m} \sum_{j=1}^m x_i^{(j)}, i = \{1, 2, \dots, n\}$

- $\sigma_i^2 = \frac{1}{m} \sum_{j=1}^m \left(x_i^{(j)} - \mu_i \right)^2, i = \{1, 2, \dots, n\}$

- Normalize the data

- $\bar{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}},$ where ϵ is a small constant for numerical stability

- Scale and shift with learnable parameters $\gamma \in \mathbb{R}^{m \times 1}$ and $\beta \in \mathbb{R}^{m \times 1}$

- $y_i = \gamma \circ \bar{x}_i + \beta$

- Learnable parameters allows undoing of normalization if needed

	x_1	x_2	x_3	x_4		
$x_i^{(1)}$	2	80	400	0.5	μ_1	σ_1^2
$x_i^{(2)}$	4	90	300	0.7	μ_2	σ_2^2
$x_i^{(3)}$	6	70	500	0.4	μ_3	σ_3^2
$x_i^{(4)}$	8	85	600	0.6	μ_4	σ_4^2
$x_i^{(5)}$	10	95	200	0.8	μ_5	σ_5^2

$$x_i = \begin{bmatrix} x_i^{(1)} \\ \vdots \\ x_i^{(n)} \end{bmatrix} \in \mathbb{R}^{n \times 1}$$

$$\bar{x}_i \in \mathbb{R}^{n \times 1}$$

Why Layer Normalization?

- Batch-size agnostic
 - Works consistently for batch size 1 or variable batch sizes
- Ideal for sequence models
 - Sequences are usually of variable length
 - BatchNorm is NOT preferred due to dependency of batch statistics on length of data
- Widely used in Transformers and RNNs,

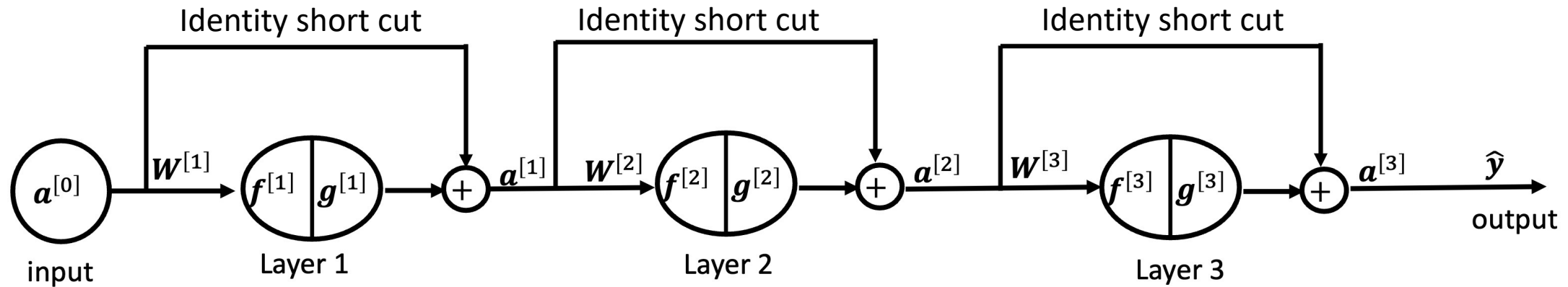
How to Choose Normalization Techniques?

Use Case	Normalization Technique
Always	Data Normalization
CNNs with large batches	Batch Normalization
Sequence Models and Transformers	Layer Norm, RMS Norm, PreNorm

Skip Connections

Improve gradient flows

Skip Connection: Scalar Neural Network



$$a^{[0]} = x$$

$$f^{[1]} = W^{[1]}a^{[0]}$$

$$a^{[1]} = g^{[1]}(f^{[1]}) + a^{[0]}$$

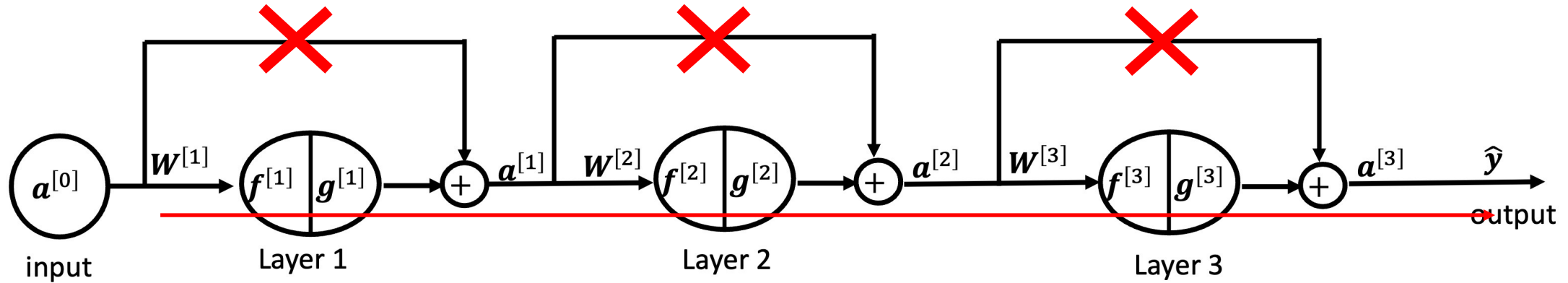
$$f^{[2]} = W^{[2]}a^{[1]}$$

$$a^{[2]} = g^{[2]}(f^{[2]}) + a^{[1]}$$

$$f^{[3]} = W^{[3]}a^{[2]}$$

$$a^{[3]} = g^{[3]}(f^{[3]}) + a^{[2]}$$

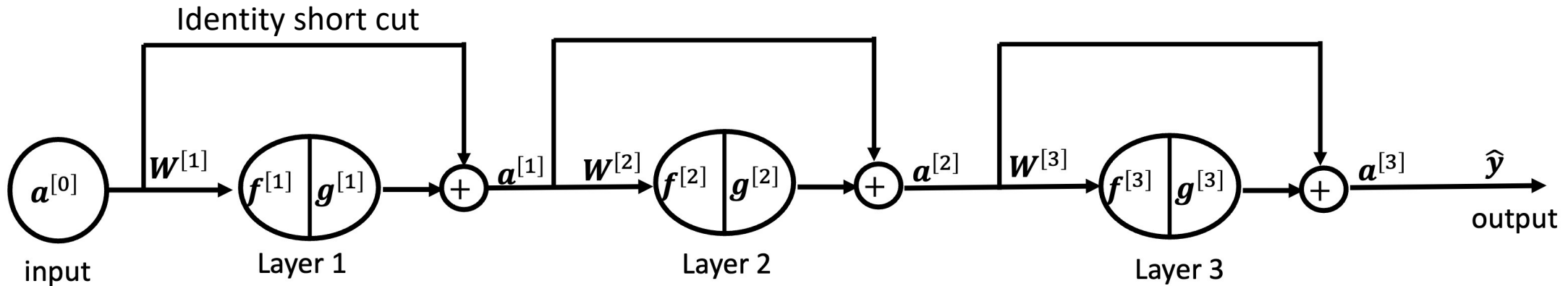
Without Skip Connections



$$\frac{\partial \hat{y}}{\partial W^{[1]}} = \frac{\partial g^{[3]}}{\partial W^{[1]}} = \frac{\partial g^{[3]}}{\partial f^{[3]}} \frac{\partial f^{[3]}}{\partial g^{[2]}} \frac{\partial g^{[2]}}{\partial f^{[2]}} \frac{\partial f^{[2]}}{\partial g^{[1]}} \frac{\partial g^{[1]}}{\partial f^{[1]}} \frac{\partial f^{[1]}}{\partial W^{[1]}}$$

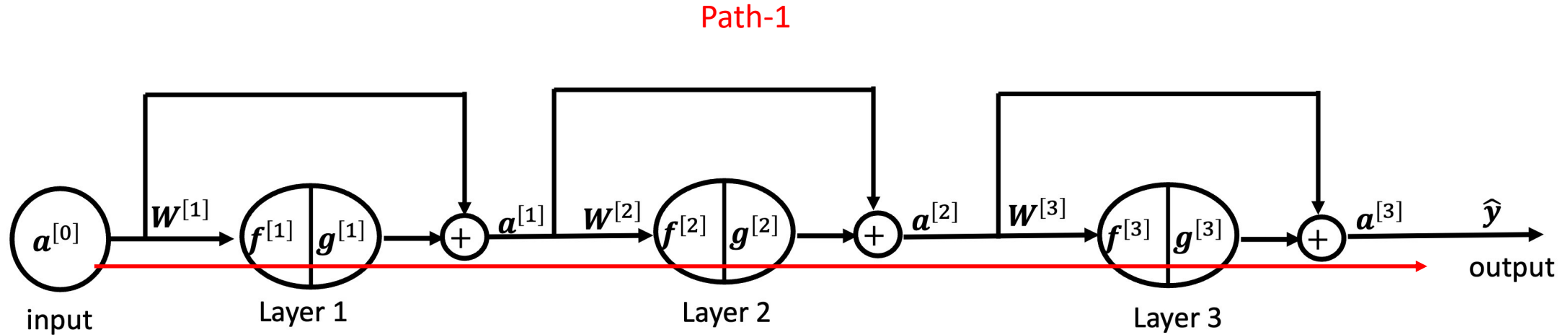
Skip Connection: Gradient Flows

Identify the number of paths through which the effect of change in $W^{[1]}$ propagates to output.



$\frac{\partial L}{\partial W^{[1]}}$ indicates the variation of loss for a change in weight $W^{[1]}$

With Skip Connection



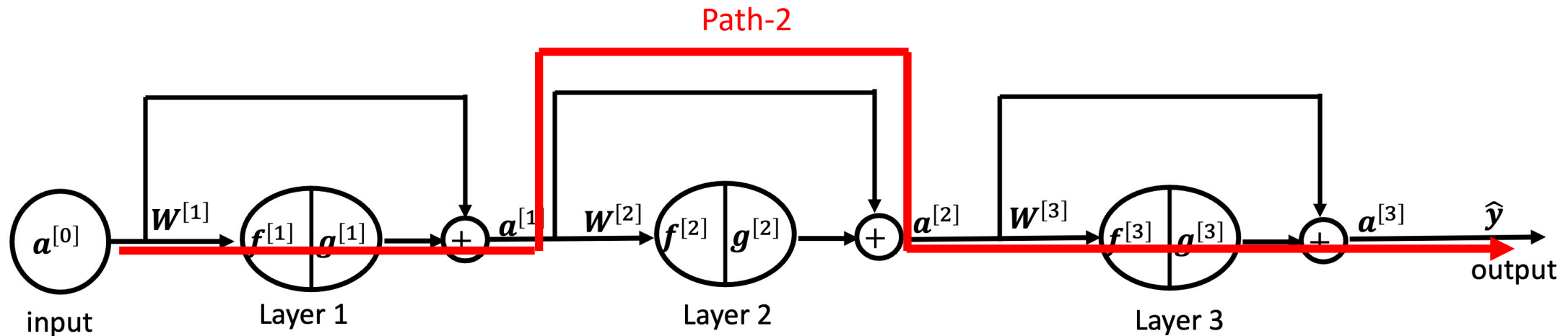
$$\frac{\partial L}{\partial W^{[1]}} = (a^{[3]} - y) \cdot \left(\delta^{[1]} + \delta^{[3]} \cdot W^{[3]} \cdot g'^{[1]}(f^{[1]}) + g'^{[2]}(f^{[2]}) \cdot W^{[2]} \cdot g'^{[1]}(f^{[1]}) + g'^{[1]}(f^{[1]}) \right) \cdot a^{[0]}$$

$$\delta^{[1]} = \delta^{[2]} W^{[2]} g'^{[1]}(f^{[1]})$$

$$\delta^{[2]} = \delta^{[3]} W^{[3]} g'^{[2]}(f^{[2]})$$

$$\delta^{[3]} = g'^{[3]}(f^{[3]})$$

With Skip Connection



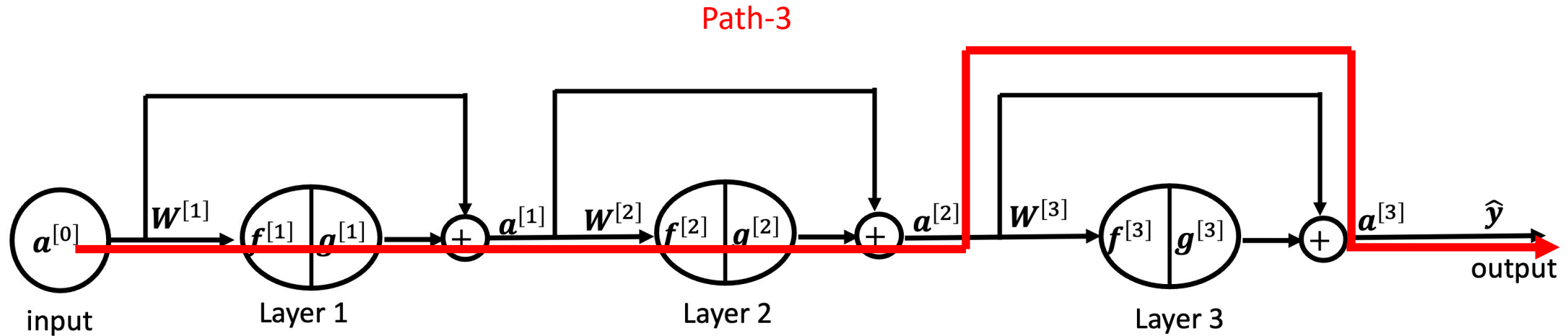
$$\frac{\partial L}{\partial W^{[1]}} = (a^{[3]} - y) \cdot \left(\delta^{[1]} + \delta^{[3]} \cdot W^{[3]} \cdot g'^{[1]}(f^{[1]}) + g'^{[2]}(f^{[2]}) \cdot W^{[2]} \cdot g'^{[1]}(f^{[1]}) + g'^{[1]}(f^{[1]}) \right) \cdot a^{[0]}$$

$$\delta^{[1]} = \delta^{[2]} W^{[2]} g'^{[1]}(f^{[1]})$$

$$\delta^{[2]} = \delta^{[3]} W^{[3]} g'^{[2]}(f^{[2]})$$

$$\delta^{[3]} = g'^{[3]}(f^{[3]})$$

With Skip Connection



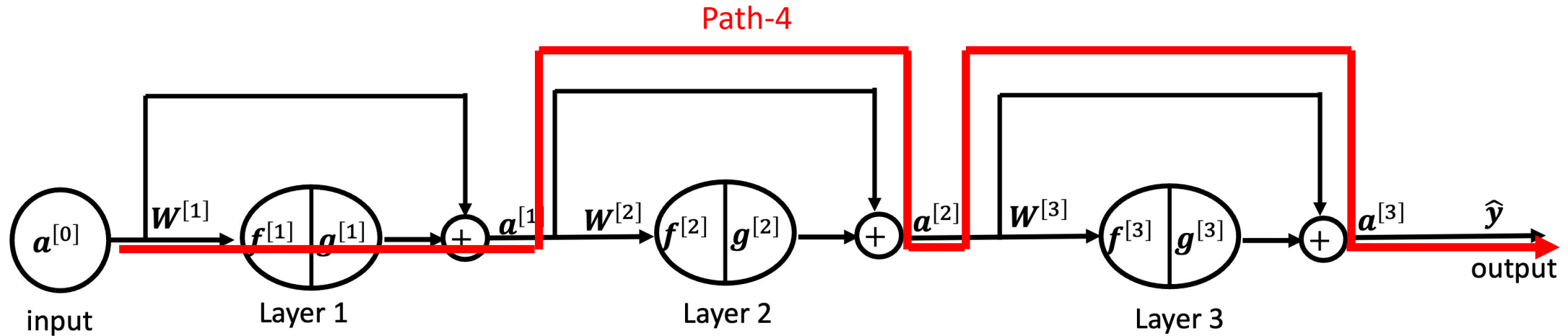
$$\frac{\partial L}{\partial W^{[1]}} = (a^{[3]} - y) \cdot \left(\delta^{[1]} + \delta^{[3]} \cdot W^{[3]} \cdot g'^{[1]}(f^{[1]}) + g'^{[2]}(f^{[2]}) \cdot W^{[2]} \cdot g'^{[1]}(f^{[1]}) + g'^{[1]}(f^{[1]}) \right) \cdot a^{[0]}$$

$$\delta^{[1]} = \delta^{[2]} W^{[2]} g'^{[1]}(f^{[1]})$$

$$\delta^{[2]} = \delta^{[3]} W^{[3]} g'^{[2]}(f^{[2]})$$

$$\delta^{[3]} = g'^{[3]}(f^{[3]})$$

With Skip Connection



$$\frac{\partial L}{\partial W^{[1]}} = (a^{[3]} - y) \cdot \left(\delta^{[1]} + \delta^{[3]} \cdot W^{[3]} \cdot g'^{[1]}(f^{[1]}) + g'^{[2]}(f^{[2]}) \cdot W^{[2]} \cdot g'^{[1]}(f^{[1]}) + g'^{[1]}(f^{[1]}) \right) \cdot a^{[0]}$$

$$\delta^{[1]} = \delta^{[2]} W^{[2]} g'^{[1]}(f^{[1]})$$

$$\delta^{[2]} = \delta^{[3]} W^{[3]} g'^{[2]}(f^{[2]})$$

$$\delta^{[3]} = g'^{[3]}(f^{[3]})$$

Dropout: Prevents overfitting

Journal of Machine Learning Research 15 (2014) 1929-1958

Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava

Geoffrey Hinton

Alex Krizhevsky

Ilya Sutskever

Ruslan Salakhutdinov

Department of Computer Science

University of Toronto

10 Kings College Road, Rm 3302

Toronto, Ontario, M5S 3G4, Canada.

NITISH@CS.TORONTO.EDU

HINTON@CS.TORONTO.EDU

KRIZ@CS.TORONTO.EDU

ILYA@CS.TORONTO.EDU

RSALAKHU@CS.TORONTO.EDU

Dropout: How it works?

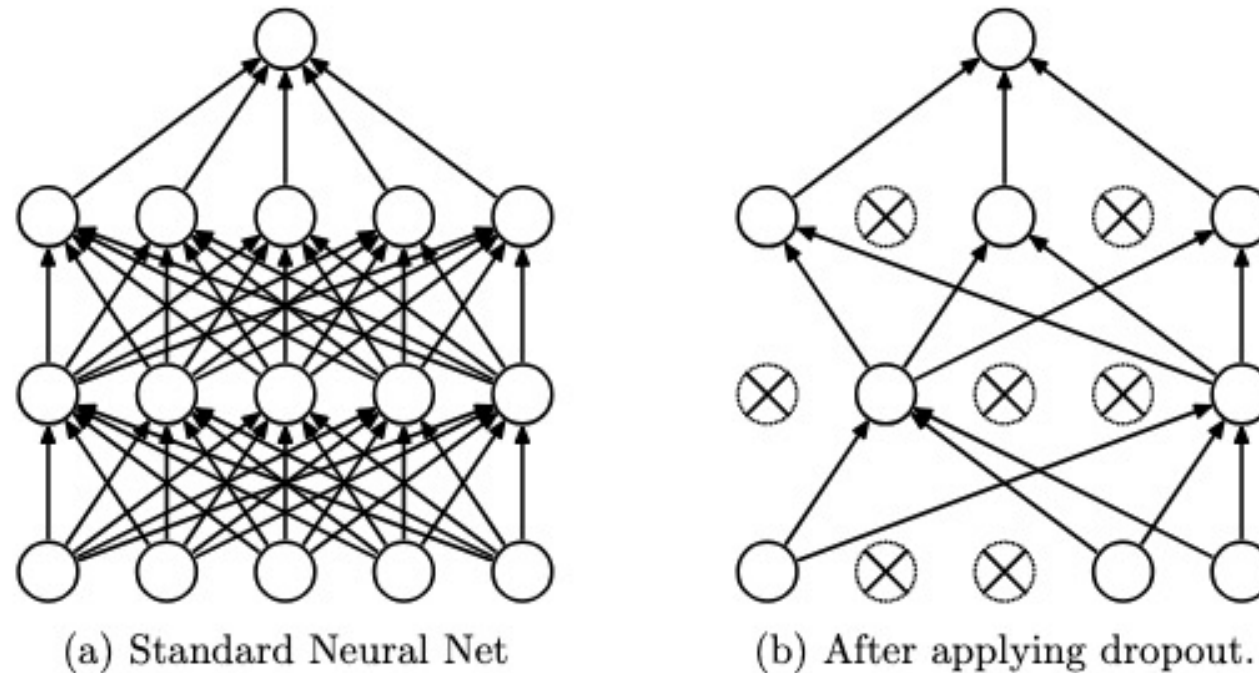


Figure 1: Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Dropout: How it works?

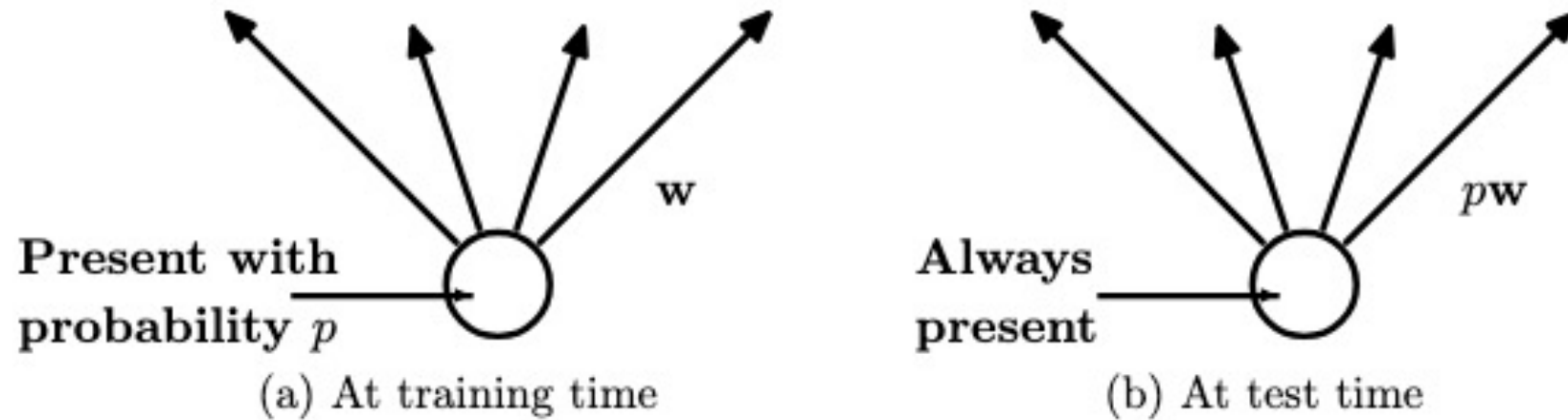
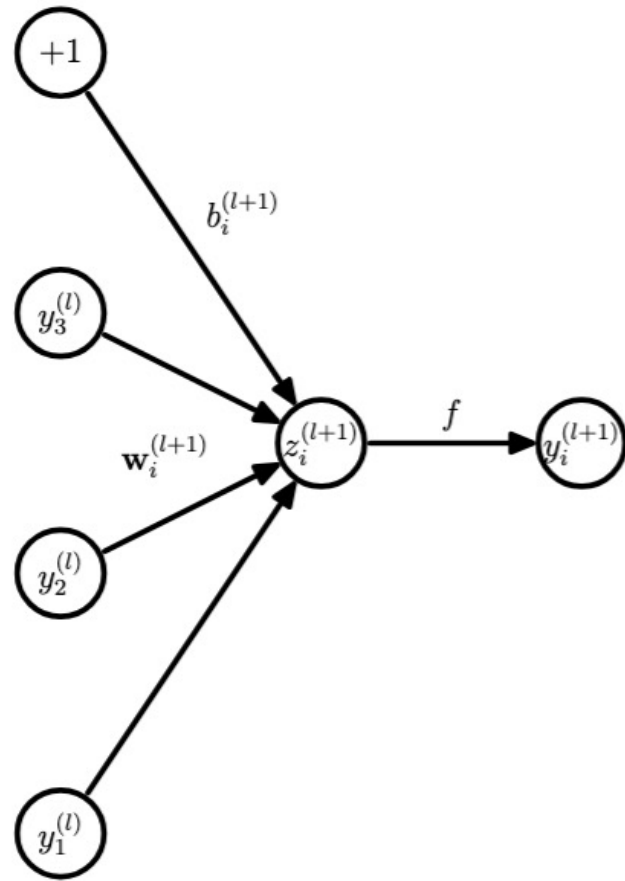
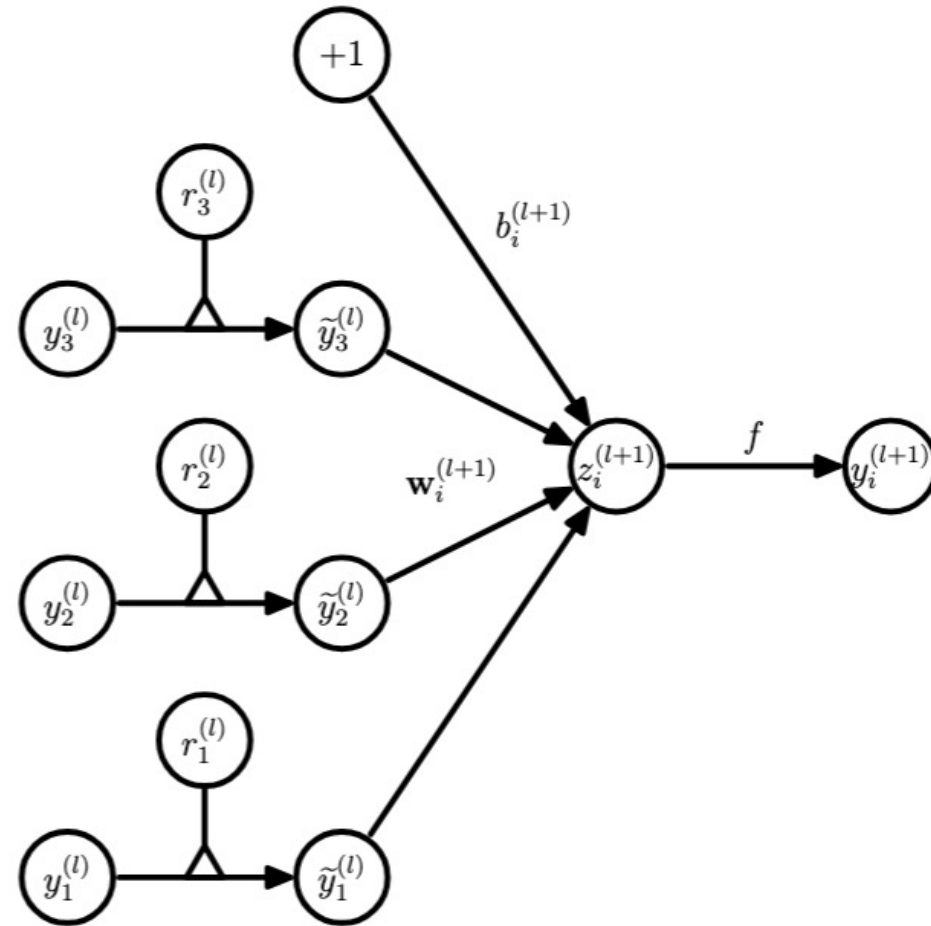


Figure 2: **Left:** A unit at training time that is present with probability p and is connected to units in the next layer with weights w . **Right:** At test time, the unit is always present and the weights are multiplied by p . The output at test time is same as the expected output at training time.

Dropout: How it works?



(a) Standard network



(b) Dropout network

Figure 3: Comparison of the basic operations of a standard and dropout network.

Dropout: Lowers Training Error

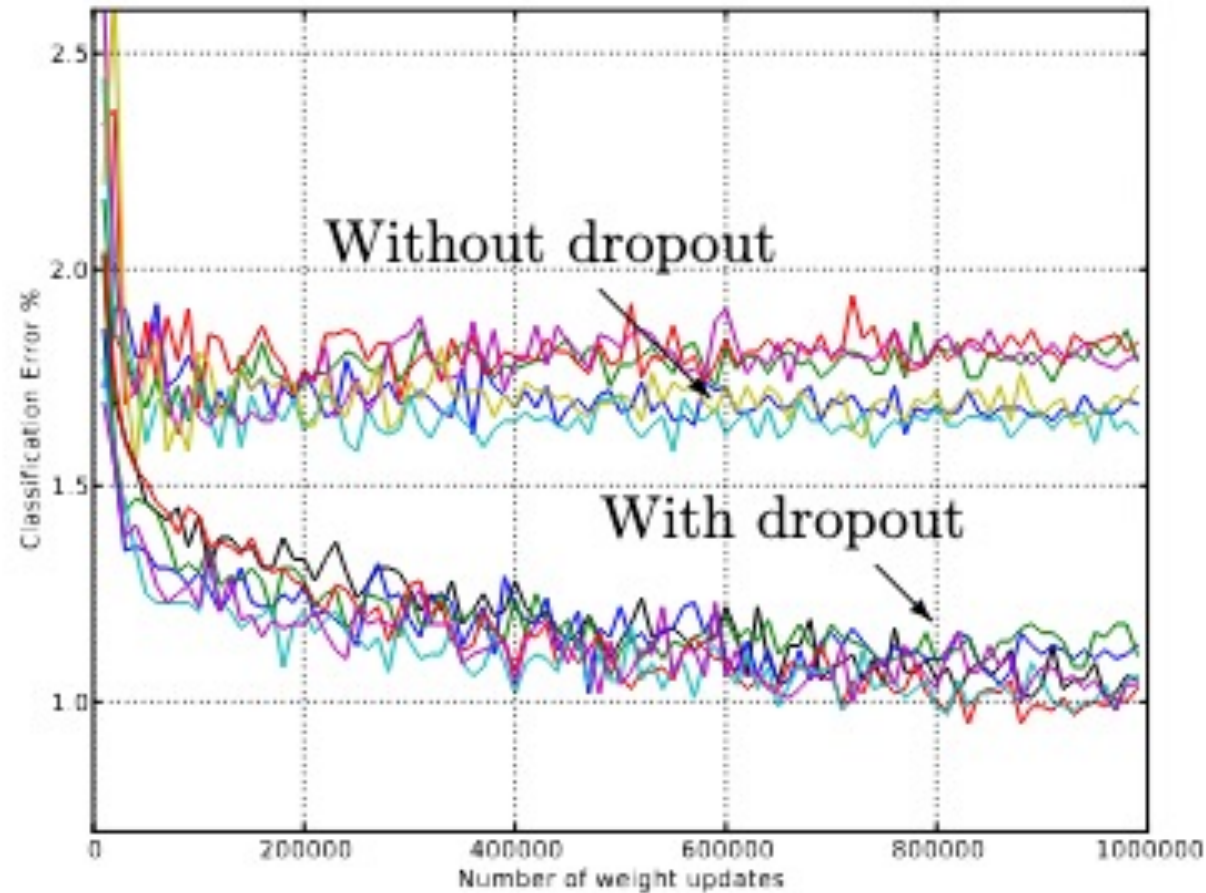


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Inverted Dropout: Used in Practice

$$f^{[l]} \equiv (W^{[l]})^T a^{[l-1]}$$

$$a^{[l]} \equiv g^{[l]}(f^{[l]})$$

- Scales activation with $\left(\frac{1}{p}\right)$ during training
- During Training: $\tilde{a}^{[l]} = \frac{\mathbf{m}^{[l]} \circ a^{[l]}}{p_l}$
 - $\mathbf{m}^{[l]}$ is the mask vector at l -th layer
 - Elements of mask vector are drawn from $Bernouli(p_l)$, where p_l is the dropout probability at l -th layer
- During Inference: $\tilde{a}^{[l]} = a^{[l]}$
- Scaling activation with inverted p_l preserves mean during training
- PyTorch: `nn.Dropout(p = drop_rate)` uses inverted dropout