Neural Networks

INSTRUCTOR: HONGJIE CHEN

JUNE 9TH 2022

A Diagram of Node

- Essentially a perceptron

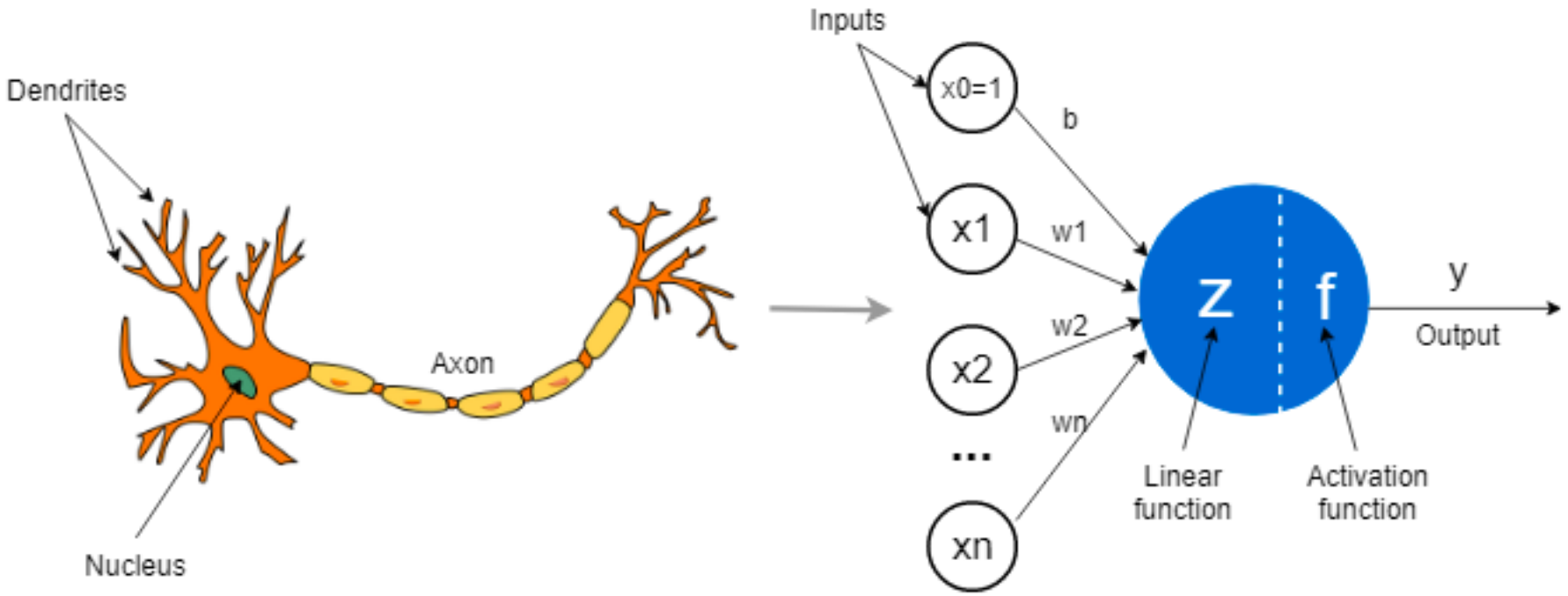


Figure credit: link

Activation Function

- Generally non-linear
  - ANN is a linear function otherwise
- Mimics the firing in neurons
  - Nodes should be “active” (output close to 1) when fed with the “right” inputs
  - Nodes should be “inactive” (output close to 0) when fed with the “wrong” inputs
- “CLICK”

Common Activation Functions

- Identity function

$$h(a) = a$$

- Threshold function

$$h(a) = \begin{cases} 1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

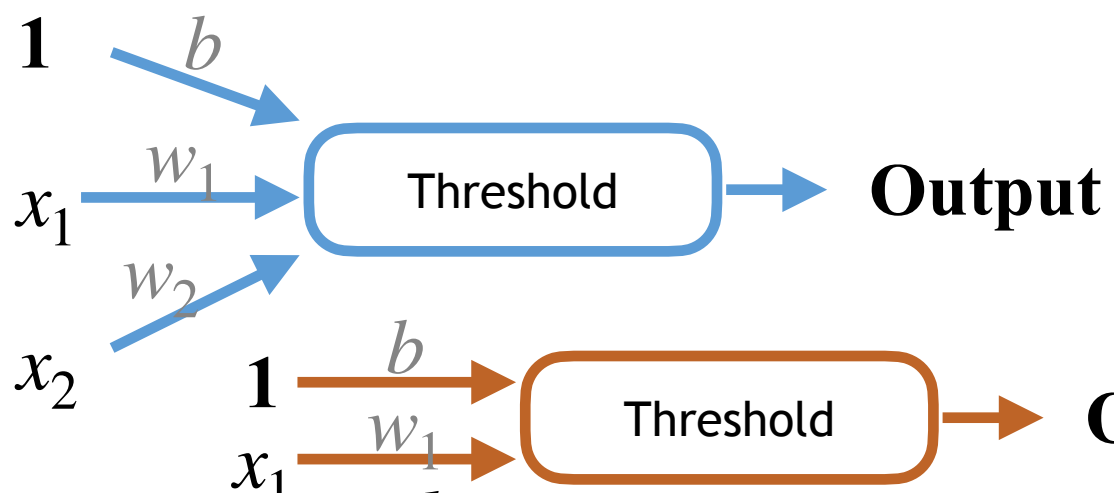
- Sigmoid

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

Very common

ANN for Boolean functions

- Can we design ANNs to imitate logical gates?
- By design, we are tuning the weights



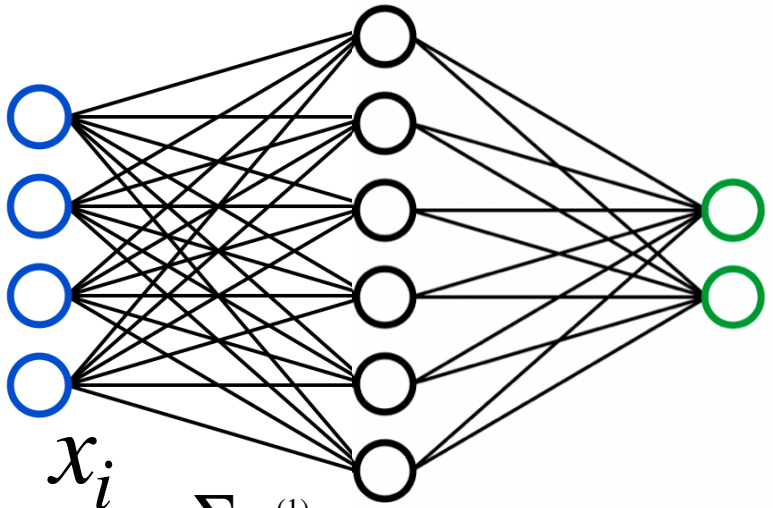
X1	X2	AND
0	0	0
0	1	0
1	0	0
1	1	1

X1	X2	OR
0	0	0
0	1	1
1	0	1
1	1	1

X1	NOT
0	1
1	0

- What should b, w_1, w_2 be to achieve **AND**, **OR**, **NOT**
- ANN can therefore approximate any logical expression

Two-layer Feed-forward Network (Exclude Input)



- Hidden nodes: $z_j = h_1(\mathbf{w_j}^{(1)T}\mathbf{x})$
- Ouput nodes: $y_k = h_2(\mathbf{w_k}^{(2)T}\mathbf{z})$

Put it together: $y_k = h_2(\sum_j w_{kj}^{(2)} h_1(\sum_i w_{ji}^{(1)} x_i))$

Regression v.s. Classification

- Regression

$$y_k = \sum_j w_{kj}^{(2)} h_1\left(\sum_i w_{ji}^{(1)} x_i\right)$$

- Classification

$$P(y_k \mid x) = h_2\left(\sum_j w_{kj}^{(2)} h_1\left(\sum_i w_{ji}^{(1)} x_i\right)\right)$$

Probability

More Activation Functions

- Gaussian: $h(a) = e^{-\frac{(a-\mu)^2}{2\sigma^2}}$
- Tanh: $h(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$
- Many more!
- https://arxiv.org/pdf/1803.08375.pdf

Weight Optimization

- $y_k = h_2(\sum_j w_{kj}^{(2)} h_1(\sum_i w_{ji}^{(1)} x_i))$

- Parameters: $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots$
- Objective: Error minimization

- Approach:
  - **Backpropagation**

Possible Error Function

- Recall the square error term in the regression model

- Here, we assume

  n **here is the number of samples**

  $$E(W) = \frac{1}{2}\sum_n E_n(W)^2 = \frac{1}{2}\sum_n \|f(\mathbf{x}_n, \mathbf{W}) - y_n\|_2^2$$

  $$f(\mathbf{x}, \mathbf{W}) = \sum_j w_{kj}^{(2)} \sigma\left(\sum_i w_{ji}^{(1)} x_i\right)$$

  Linear **Non-linear**

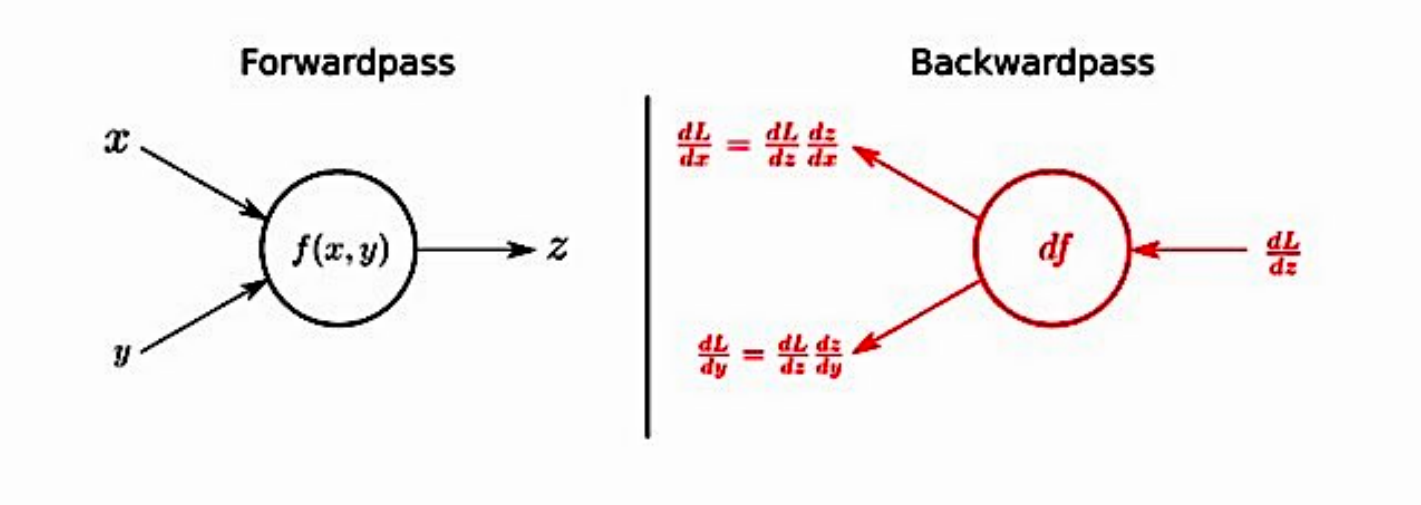
Gradient Descent with ANN

- For each sample $(\mathbf{x}_n, \mathbf{y}_n)$, update weights
  - $w_{ji} \leftarrow w_{ji} - \eta \dfrac{\partial E_n}{\partial w_{ji}}$
  - $w_{kj} \leftarrow w_{kj} - \eta \dfrac{\partial E_n}{\partial w_{kj}}$

- How to efficiently compute the gradient descent?

Backpropagation in Two Phases

- Forward phase: compute output z_j for each node j



- Backward phase: compute gradient δ_j for each node j

Figure credit: link

Forward Phase

- Propagate inputs forward through the network to compute the output of each node (hidden or not hidden)

- Output z_j at node j

$$z_j = h(a_j) \text{ where } a_j = \sum_i w_{ji} z_i$$

Backward Phase

- Use chain rule to recursively compute gradient
  - For each weight w_{ji}
  - $\delta_{ji} = \frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$
  - Then $\delta_j = \frac{\partial E_n}{\partial a_j} = \begin{cases} h'(a_j)(z_j - y_j), j \in \text{output nodes} \\ h'(a_j)\sum_k w_{kj}\delta_k, j \in \text{hidden nodes} \end{cases}$
  - With $a_j = \sum_i w_{ji} z_i \Rightarrow \frac{\partial a_j}{\partial w_{ji}} = z_i$
  - Thus, $\delta_{ji} = \frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i$

$$E(W) = \frac{1}{2}\sum_n E_n(W)^2 = \frac{1}{2}\sum_n \|f(\mathbf{x}_n, \mathbf{W}) - y_n\|_2^2$$

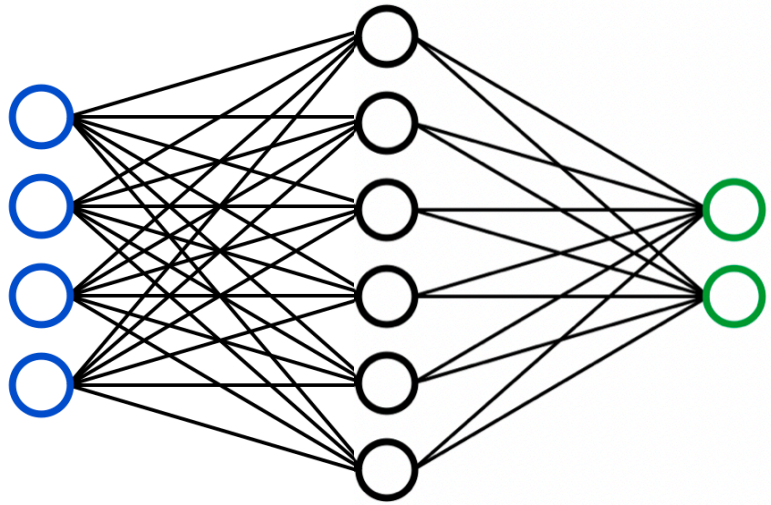
$$f(\mathbf{x}, \mathbf{W}) = \sum_j w_{kj}^{(2)}\sigma(\sum_i w_{ji}^{(1)} x_i)$$

Derivative $h'(a)$

- Smartly select activation function

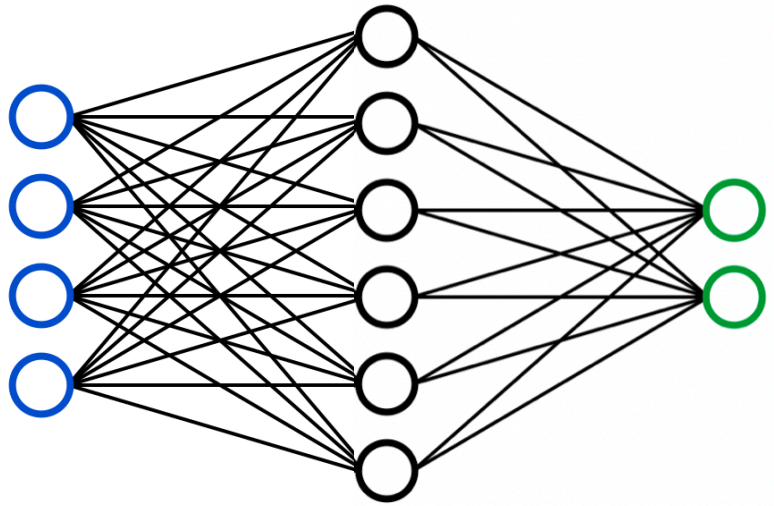
- For example, $h(a) = \tanh(a) = \dfrac{e^a - e^{-a}}{e^a + e^{-a}}$
- $h'(a) = 1 - \tanh^2(a)$

Derivatives Altogether



- Forward Phase
  - Hidden nodes: $a_j =$ $z_j =$
  - Output nodes: $a_k =$ $z_k =$
- Backward Phase
  - Output nodes: $\delta_k =$
  - Hidden nodes: $\delta_j =$
- Gradients
  - Hidden layer: $\frac{\partial E_n}{w_{ji}} =$
  - Output layer: $\frac{\partial E_n}{w_{kj}} =$

Derivatives Solution



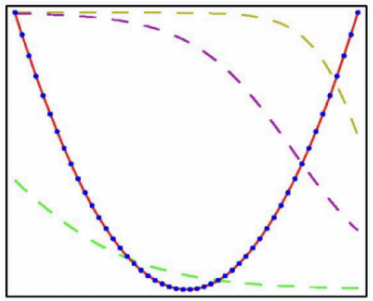
- Forward Phase
  - Hidden nodes: $a_j = \sum_i w_{ji} x_i$ $\qquad z_j = \tanh(a_j)$
  - Output nodes: $a_k = \sum_j w_{kj} z_j$ $\qquad z_k = a_k$
- Backward Phase
  - Output nodes: $\delta_k = z_k - y_k$ $\qquad \delta_k = \dfrac{\partial E_n}{\partial a_k}$
  - Hidden nodes: $\delta_j = [1 - z_j^2] \sum_k w_{kj} \delta_k$ $\qquad \delta_j = \dfrac{\partial E_n}{\partial a_j}$
- Gradients
  - Hidden layer: $\dfrac{\partial E_n}{w_{ji}} = \delta_j x_i = [1 - z_j^2] \sum_k w_{kj} \delta_k x_i$
  - Output layer: $\dfrac{\partial E_n}{w_{kj}} = \delta_k z_j = (z_k - y_k) z_j$

$$\frac{\partial E_n}{w_{ji}} = \sum_k \frac{\partial E_n}{\partial \delta_k} \frac{\partial \delta_k}{\partial z_k} \frac{\partial z_k}{\partial a_k} \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

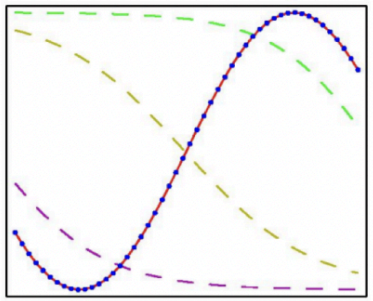
None Linear Regression Examples

- Two layer networks with 3 tanh hidden units and 1 output unit

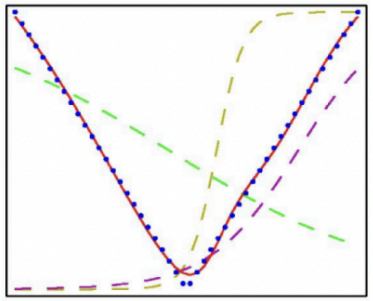
$y = x^2$



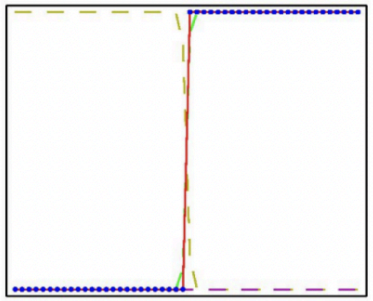
$y = \sin x$



$y = |x|$



$y = \int_{-\infty}^{x} \delta(t)dt$



Sum three dash curves to the dot curve

ANN Analysis

- Efficient: linear to number of weights
- Local minima
  - Randomly start from different places
- Convergence:
  - Slow convergence (step size)
  - Can get trapped in local optima, and never converge
- Overfitting
  - Solution: Early stopping, regularization, dropout

Slow Convergence

- Too slow or overshoot

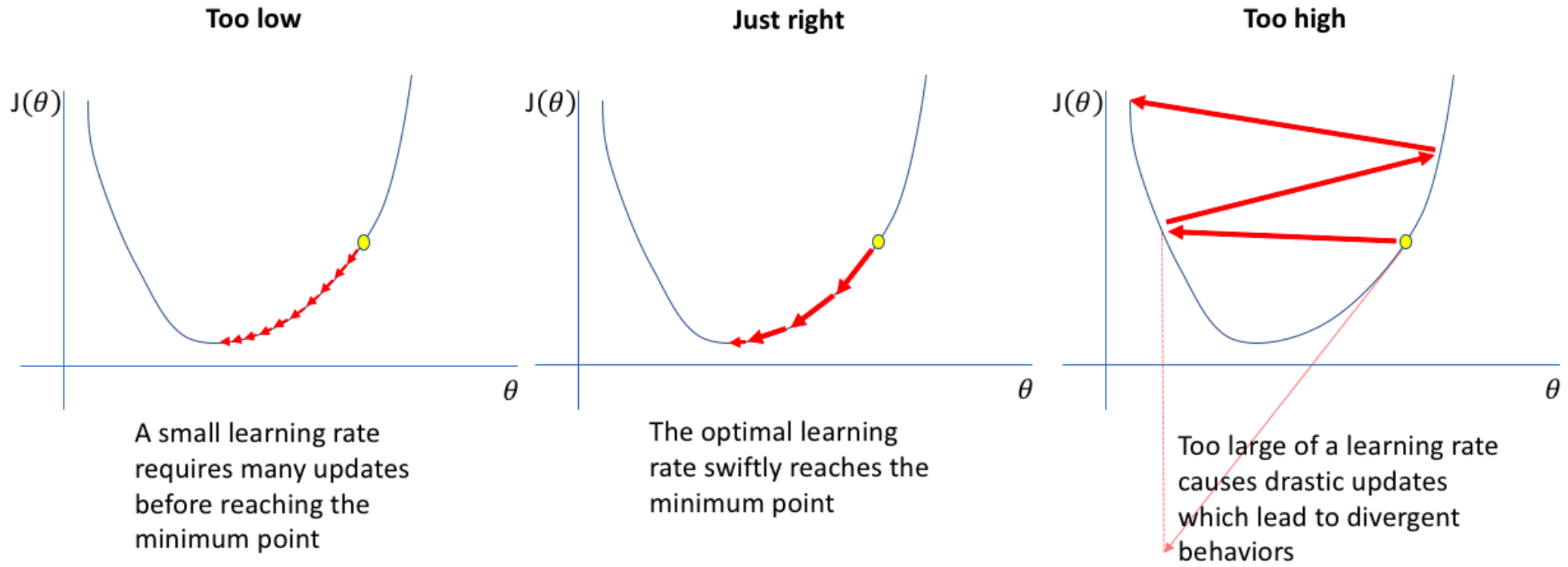


Figure credit: link

Optimizer: Adaptive Gradients

- **Idea**: adjust the learning rate for each dimension separately
- ***AdaGrad*** $(r_0 = 1)$:
  - $r_t \leftarrow r_{t-1} + (\frac{\partial E_n}{\partial w_{ji}})^2$
  - $w_{ji} \leftarrow w_{ji} - \frac{\eta}{\sqrt{r_t}} \frac{\partial E_n}{\partial w_{ji}}$
- **Problem**: learning rate $\frac{\eta}{\sqrt{r_t}}$ decays too quickly

If last step is great, walk more slowly

Optimizer: RMSprop

- **Idea**: weighted between old r_{t-1} and new error
- **RMSprop**:
  - $r_t \leftarrow \alpha r_{t-1} + (1-\alpha)(\frac{\partial E_n}{\partial w_{ji}})^2 \ (0 \leq \alpha \leq 1)$
  - $w_{ji} \leftarrow w_{ji} - \frac{\eta}{\sqrt{r_t}} \frac{\partial E_n}{\partial w_{ji}}$
- **Problem**: lacks momentum

Optimizer: Adaptive Moment Estimation

- **Idea**: Use gradient with moving average to induce momentum

- **Adam**:
  - $r_t \leftarrow \alpha r_{t-1} + (1-\alpha)(\frac{\partial E_n}{\partial w_{ji}})^2 \ (0 \leq \alpha \leq 1)$
  - $s_t \leftarrow \beta s_{t-1} + (1-\beta)(\frac{\partial E_n}{\partial w_{ji}}) \ (0 \leq \beta \leq 1)$
  - $w_{ji} \leftarrow w_{ji} - \frac{\eta}{\sqrt{r_t}} s_t$