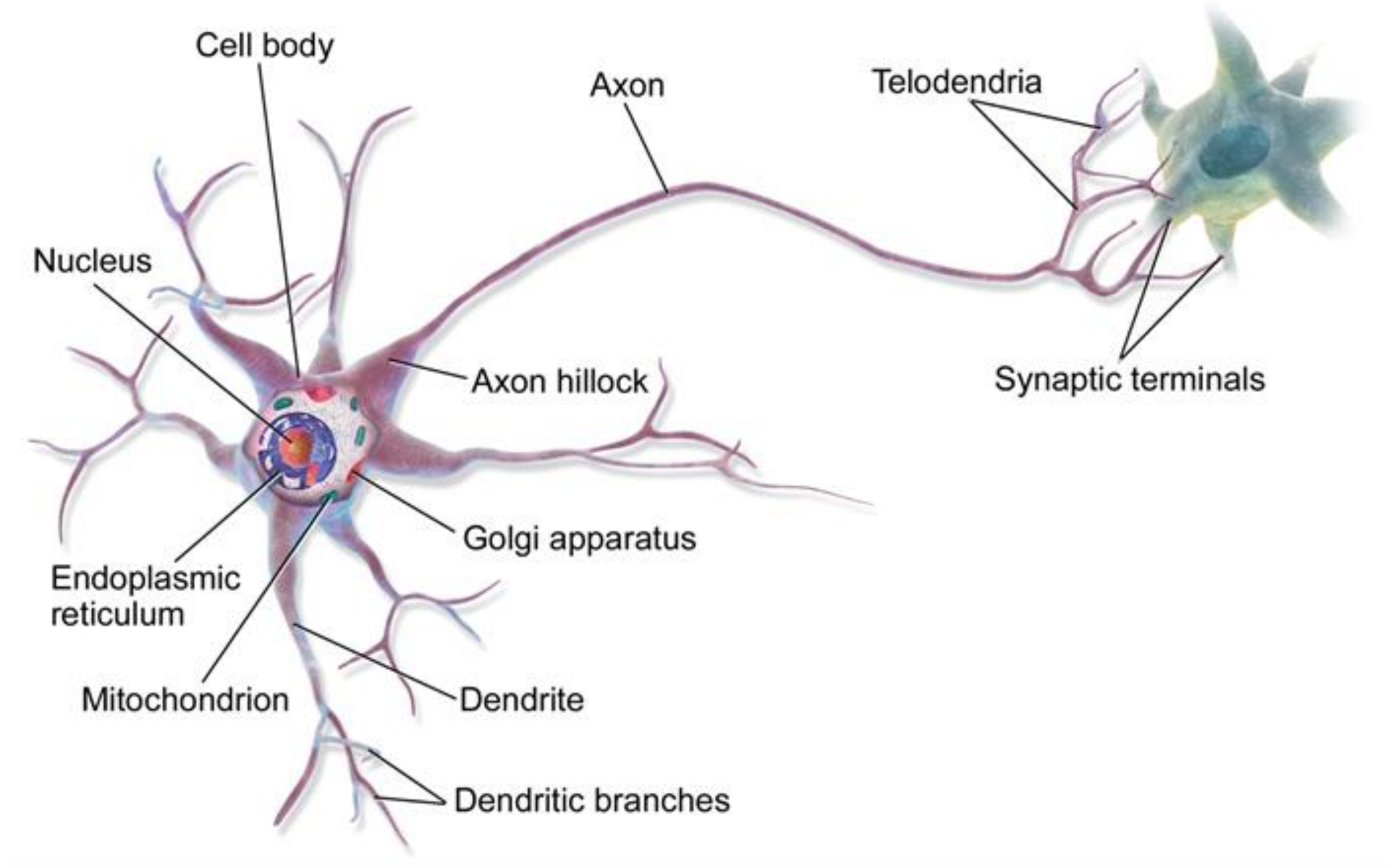
Neural Networks

NLP II 2025
Assoc. Prof. Attapol Thamrongrattanarit

Neural Networks

- A neural network is a machine learning (AI) model inspired by and based loosely on the human brain that processes information through interconnected nodes or neurons.


Cell body
Axon
Telodendria
Nucleus
Axon hillock
Synaptic terminals
Golgi apparatus
Endoplasmic
reticulum
Mitochondrion
Dendrite
Dendritic branches

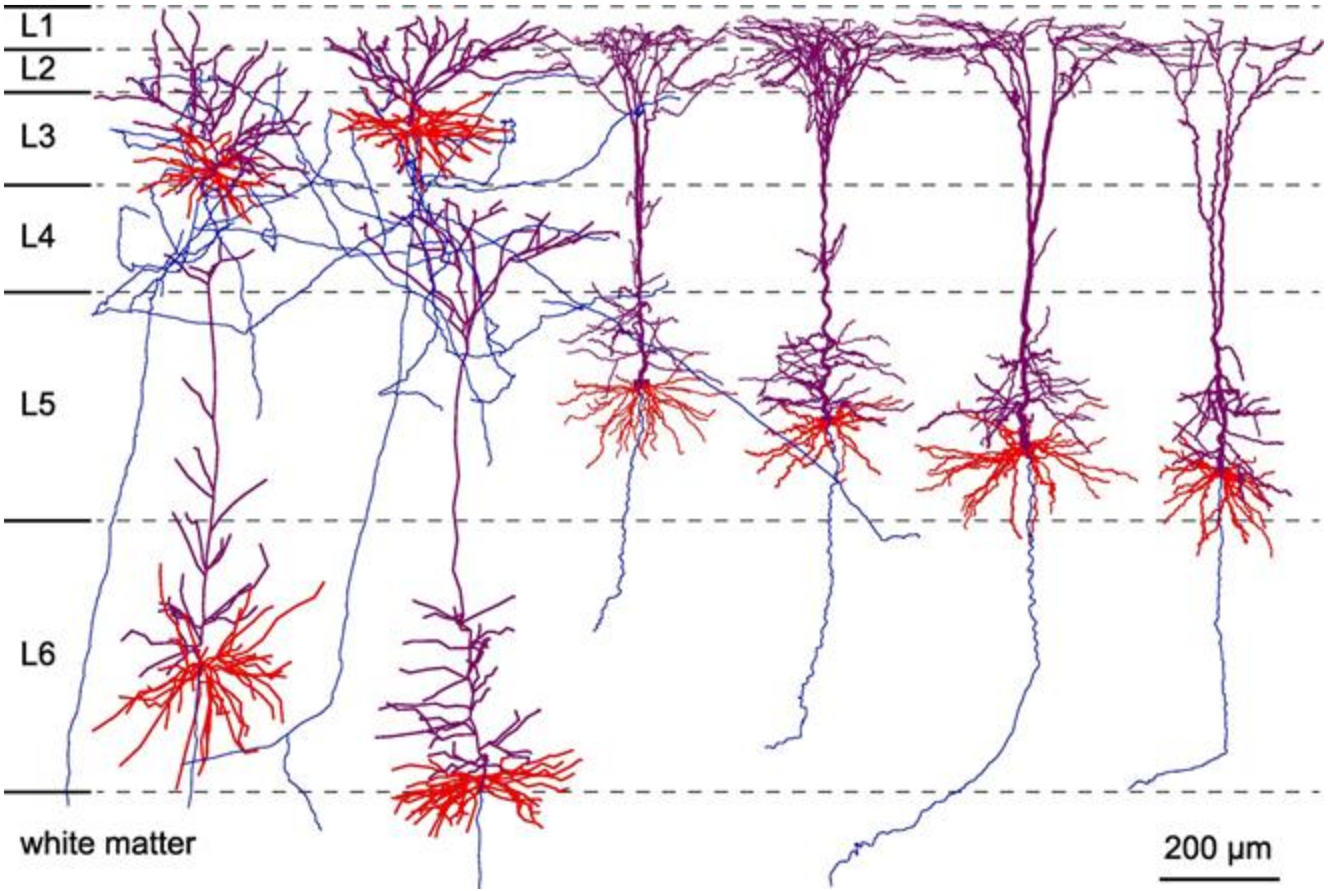
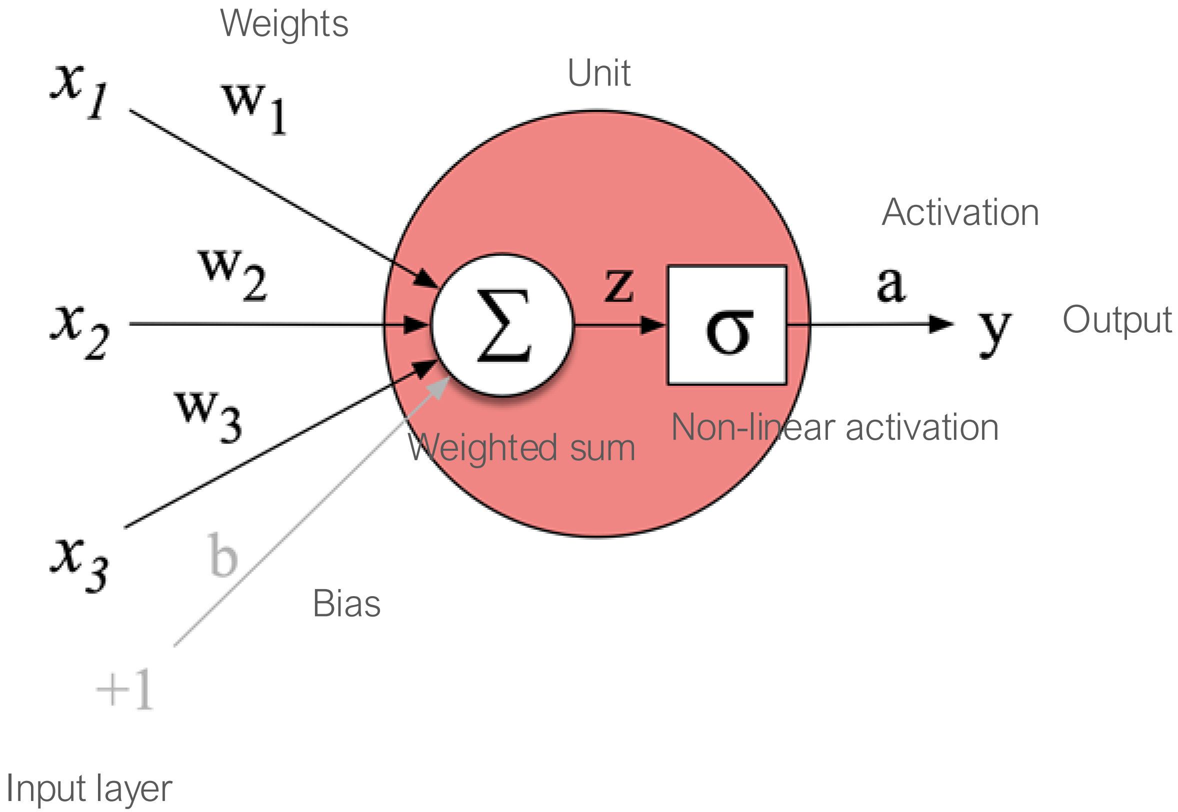


Image from Georgiev et al., 2020

Neural Network Unit



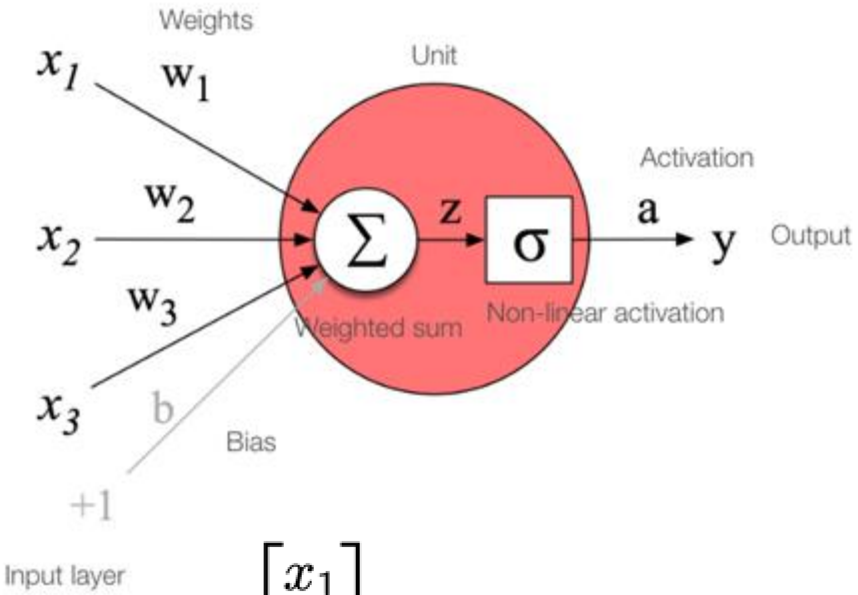
Neural Unit

$$z = \sum_{i=1}^{n} w_i x_i + b$$

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

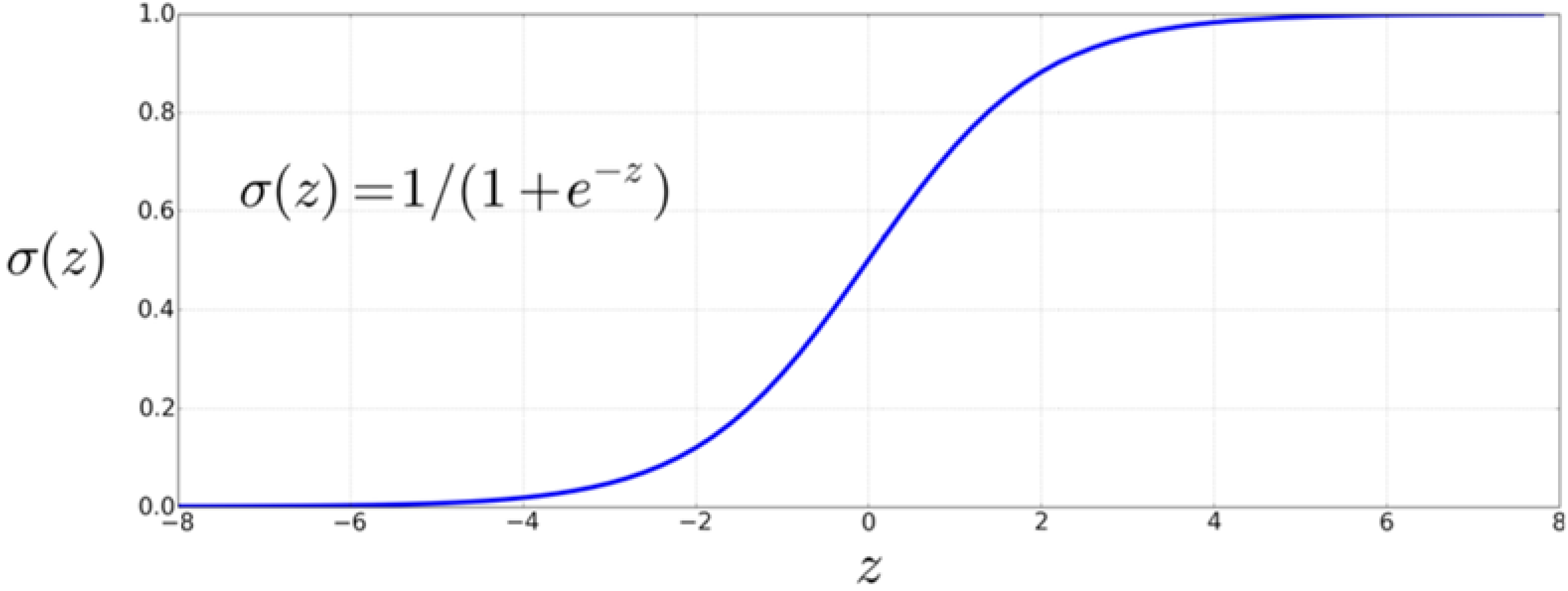
$$y = a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$



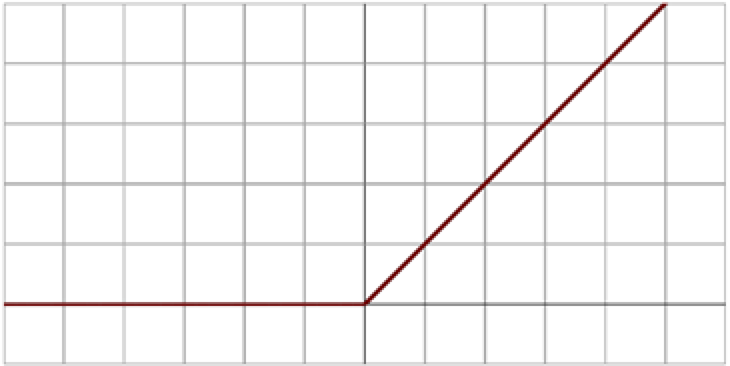
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Non-linear activation function: sigmoid



$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x} + b))}$$

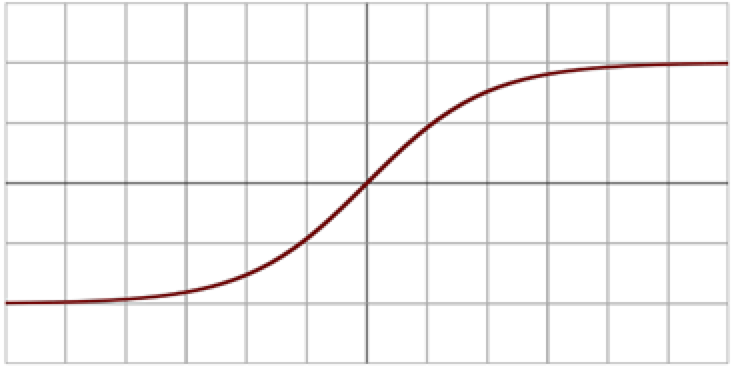
Non-linear activation function: ReLU and tanh



$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

$$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Rectified Linear Unit (ReLU)
activation function



$$f(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

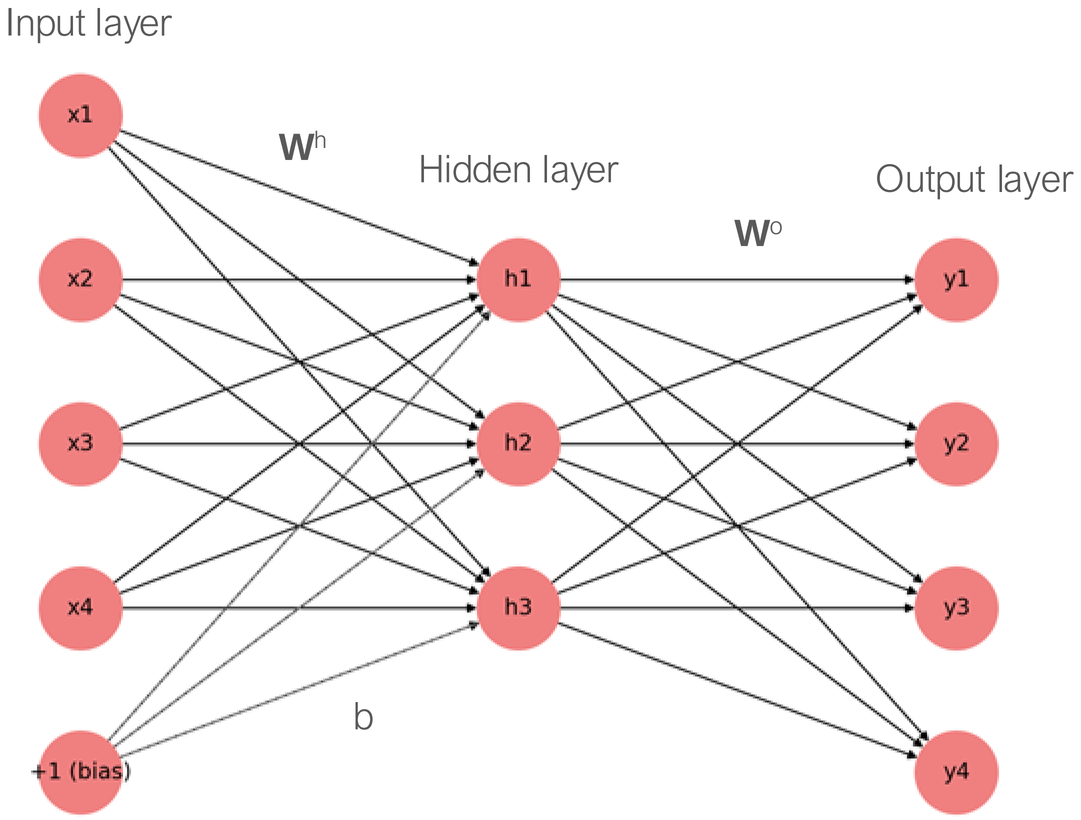
$$f'(x) = 1 - f(x)^2$$

Hyperbolic tangent
activation function

Feedforward Neural Networks

Hidden layer and hidden unit

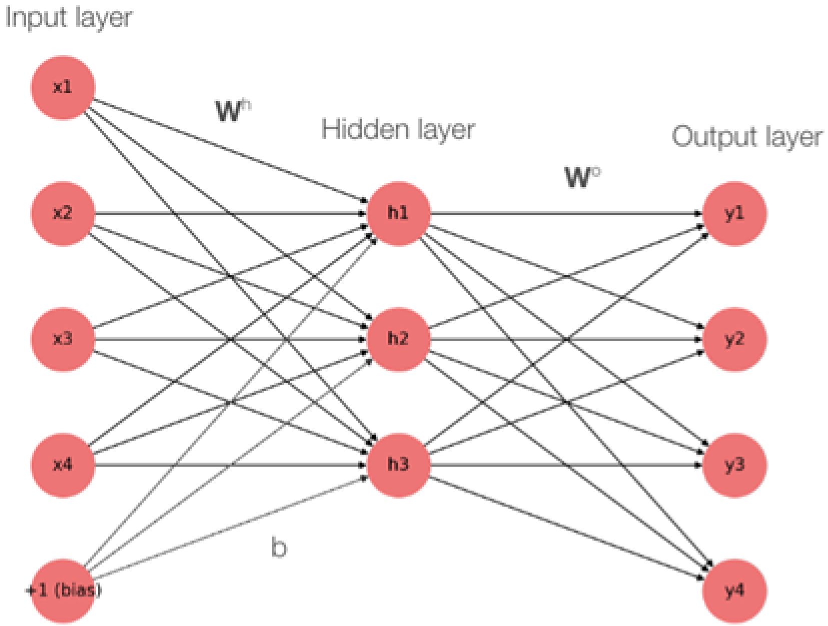
- A hidden unit is a neural unit (taking a weighted sum of its inputs and then applying non-linear activation function)
- In a standard setting, the input and the hidden layers are fully-connected, which means each hidden unit in the hidden layer sums over all the input units.



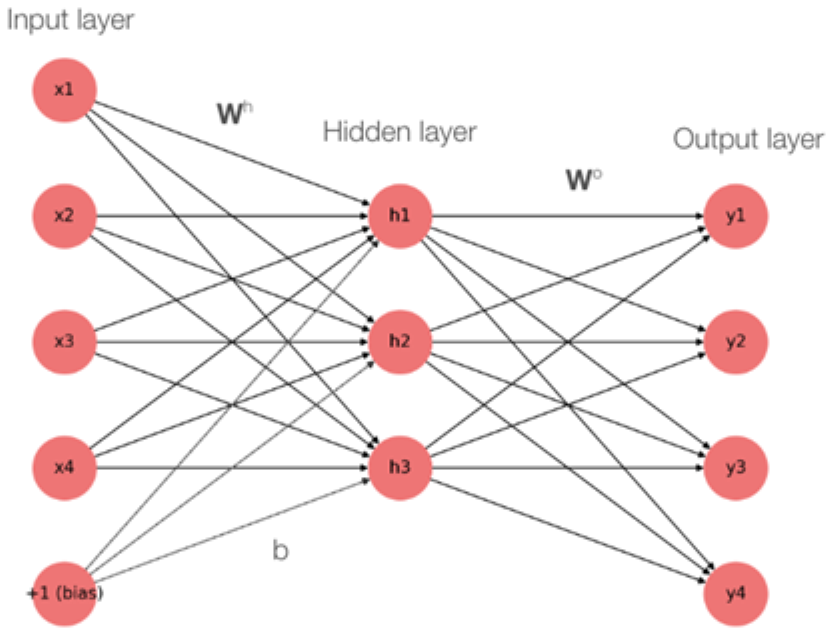
A two-layer feedforward neural network (one hidden layer and one output layer)

$$\mathbf{z} = W^h\mathbf{x} + b = \begin{bmatrix} 0.3 & -0.7 & 0.5 & 0.1 \\ -0.2 & 0.8 & -0.4 & 0.3 \\ 0.6 & -0.5 & 0.2 & -0.3 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.2 \\ 0.1 \\ 0.9 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.3 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.25 \\ -0.01 \\ 0.15 \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{z}) = \begin{bmatrix} \sigma(0.25) \\ \sigma(-0.01) \\ \sigma(0.15) \end{bmatrix} = \begin{bmatrix} 0.5622 \\ 0.4975 \\ 0.5374 \end{bmatrix}$$



$$\mathbf{h} = \sigma(\mathbf{z}) = \begin{bmatrix} \sigma(0.25) \\ \sigma(-0.01) \\ \sigma(0.15) \end{bmatrix} = \begin{bmatrix} 0.5622 \\ 0.4975 \\ 0.5374 \end{bmatrix}$$



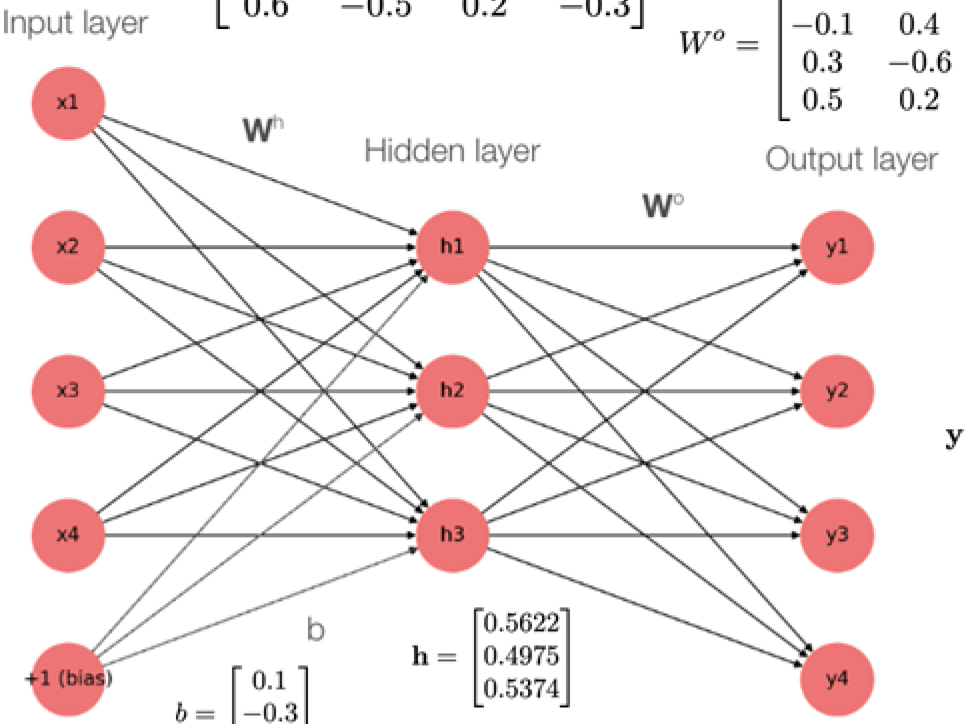
$$W^o h = \begin{bmatrix} 0.2 & -0.3 & 0.5 \\ -0.1 & 0.4 & -0.2 \\ 0.3 & -0.6 & 0.1 \\ 0.5 & 0.2 & -0.4 \end{bmatrix} \begin{bmatrix} 0.5622 \\ 0.4975 \\ 0.5374 \end{bmatrix} = \begin{bmatrix} 0.2319 \\ 0.0353 \\ -0.0761 \\ 0.1656 \end{bmatrix}$$

$$\mathbf{y} = \mathrm{softmax}(W^o h) = \frac{e^{W^o h}}{\sum e^{W^o h}} = \begin{bmatrix} \frac{e^{0.2319}}{\sum e^{W^o h}} \\ \frac{e^{0.0353}}{\sum e^{W^o h}} \\ \frac{e^{-0.0761}}{\sum e^{W^o h}} \\ \frac{e^{0.1656}}{\sum e^{W^o h}} \end{bmatrix} = \begin{bmatrix} 0.2863 \\ 0.2352 \\ 0.2104 \\ 0.2680 \end{bmatrix}$$

$$W^h = \begin{bmatrix} 0.3 & -0.7 & 0.5 & 0.1 \\ -0.2 & 0.8 & -0.4 & 0.3 \\ 0.6 & -0.5 & 0.2 & -0.3 \end{bmatrix}$$

$$W^o = \begin{bmatrix} 0.2 & -0.3 & 0.5 \\ -0.1 & 0.4 & -0.2 \\ 0.3 & -0.6 & 0.1 \\ 0.5 & 0.2 & -0.4 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} 0.5 \\ 0.2 \\ 0.1 \\ 0.9 \end{bmatrix}$$



$$\mathbf{y} = \begin{bmatrix} 0.2863 \\ 0.2352 \\ 0.2104 \\ 0.2680 \end{bmatrix}$$

$$\mathbf{h} = \begin{bmatrix} 0.5622 \\ 0.4975 \\ 0.5374 \end{bmatrix}$$

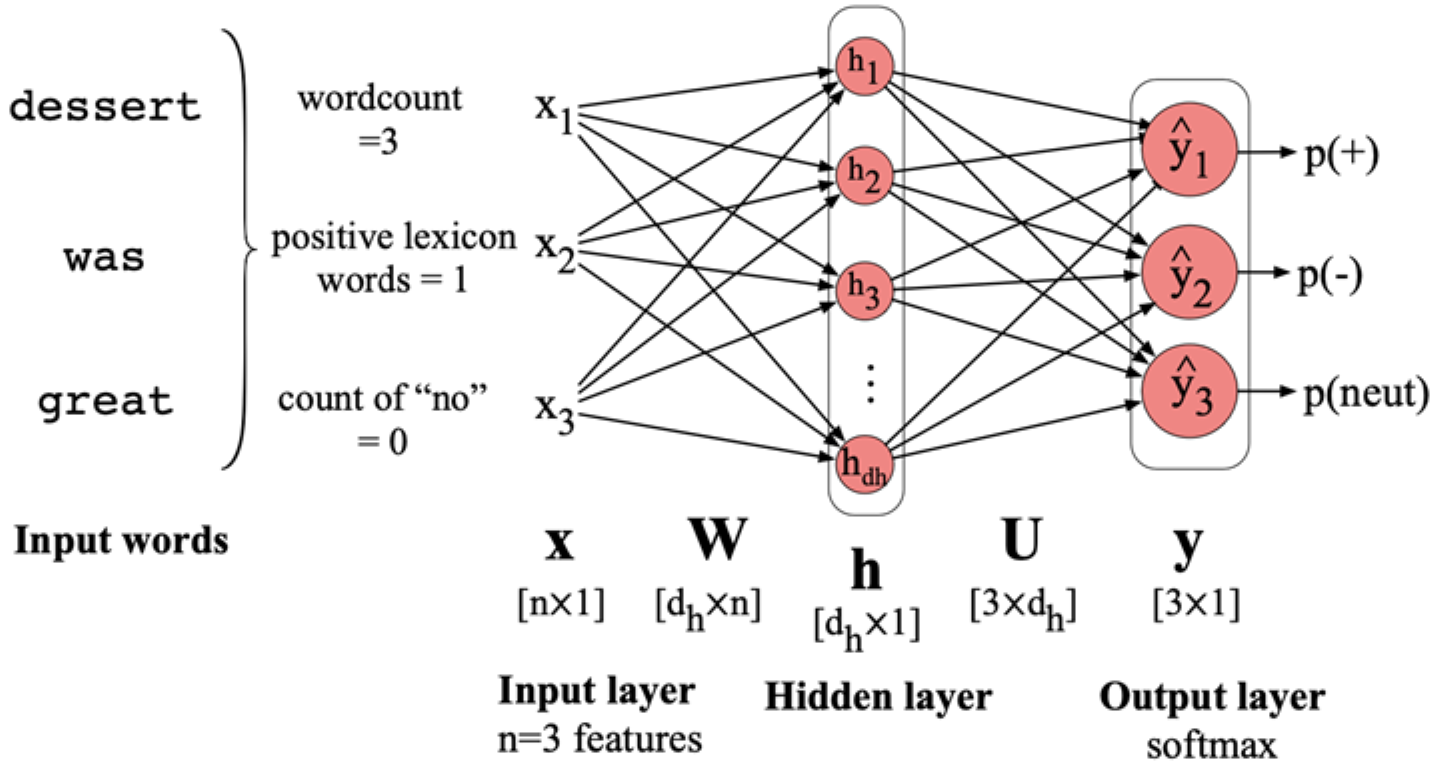
$$b = \begin{bmatrix} 0.1 \\ -0.3 \\ 0.2 \end{bmatrix}$$

Feedforward neural network

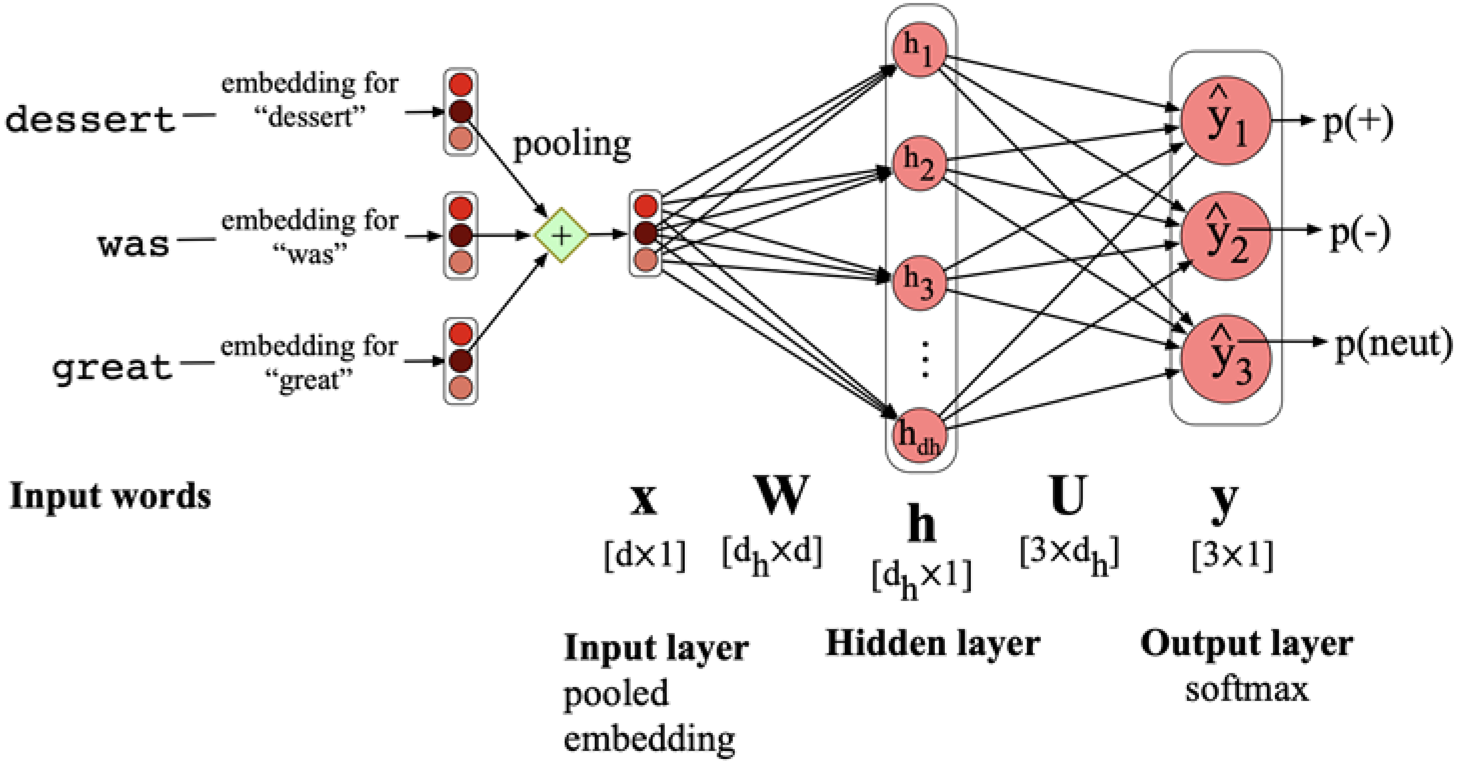
$$\mathbf{h} = \sigma(W^h \cdot \mathbf{x} + \mathbf{b})$$

$$\mathbf{y} = \text{softmax}(W^o \cdot \mathbf{h})$$

Feedforward Neural Networks for NLP: Classification



Feedforward network sentiment analysis with hand-built features



Feedforward network sentiment analysis with pooled word embeddings

Notation

- Suppose
  - X is n x d matrix (n = number of instances, d = embedding size)
  - W is d x k matrix (k = number of hidden units)
    - w_{ij} = the weight between x_i and h_j
- Then H = sigma(XW + **b**)
  - then h_{nj} = the hidden unit j for the instance n
- This is a more common notation for neural network

Training Neural Network

Training a neural network: loss function

- Training a neural network = finding the weights (parameters) that optimize (minimize) loss function.
- Define a loss function that measures the distance between the system output and the gold-standard output. We usually use cross-entropy loss:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)} \quad \text{(where } c \text{ is the correct class)}$$

Computing the gradient of a loss function

$$\begin{aligned}\frac{\partial L_{\mathrm{CE}}(\hat{\mathbf{y}},\mathbf{y})}{\partial \mathbf{w}_{k,i}} &= -(\mathbf{y}_k - \hat{\mathbf{y}}_k)\mathbf{x}_i \\ &= -(\mathbf{y}_k - p(\mathbf{y}_k = 1|\mathbf{x}))\mathbf{x}_i \\ &= -\left(\mathbf{y}_k - \frac{\exp(\mathbf{w_k}\cdot\mathbf{x}+b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w_j}\cdot\mathbf{x}+b_j)}\right)\mathbf{x}_i\end{aligned}$$

Reminder: Gradient Descent

- Update equation:

$$\theta_j^{new} = \theta_j^{old} - \underline{\alpha} \frac{\partial}{\partial \theta_j^{old}} J(\theta)$$

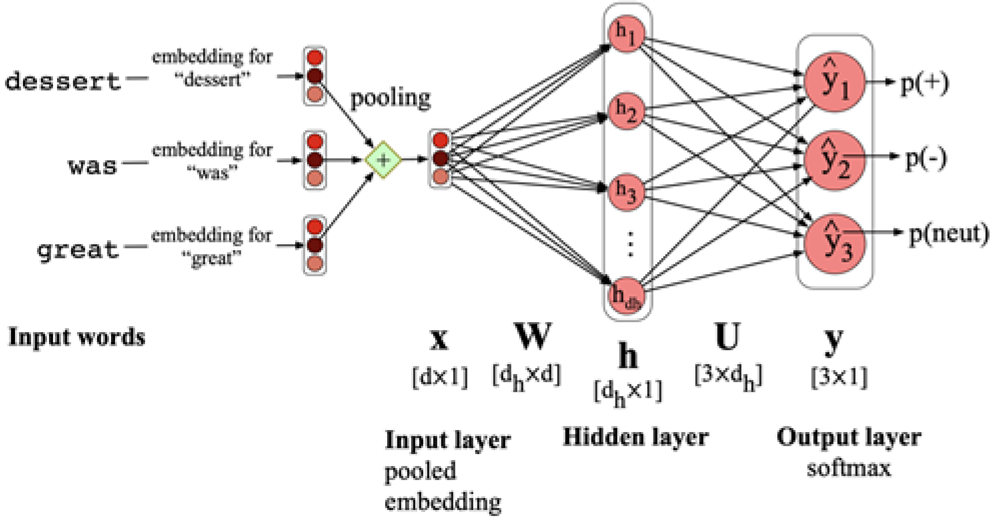
$\boxed{\alpha = \text{step size or learning rate}}$

- Algorithm:

```
while True:
    theta_grad = evaluate_gradient(J,corpus,theta)
    theta = theta - alpha * theta_grad
```

Computing gradient

- When training neural networks, we must take the derivative with respect to all of the parameters in order to optimize loss.
- How do we compute derivative for all weights?



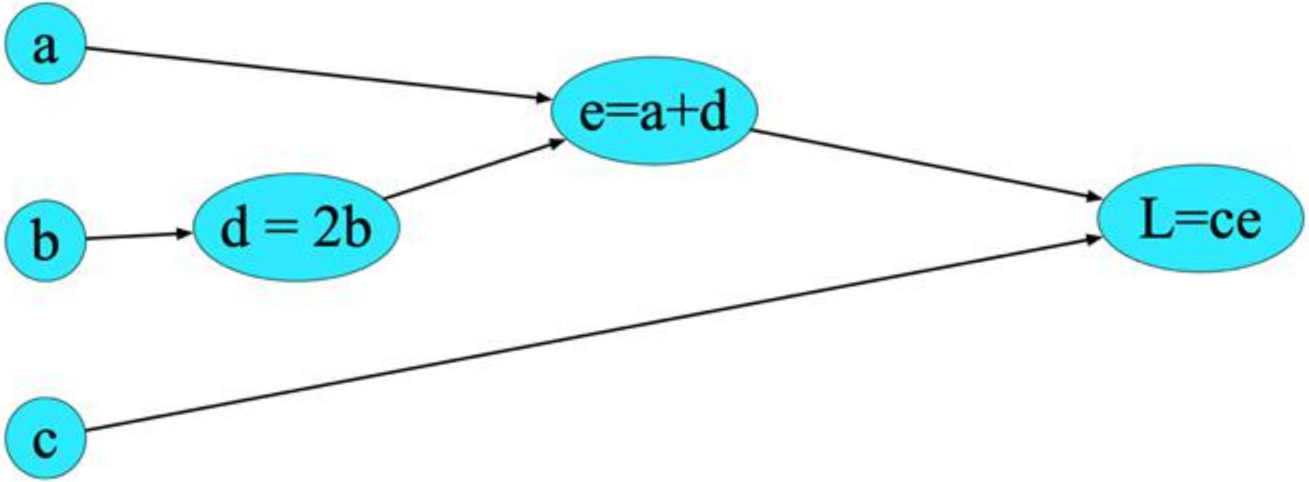
Automatic differentiation

- In practice, we use python library to compute derivative. But here's what you need to know
- When we set up a neural network, we define a computation graph, which represents what needs to be computed in the model.
- Computing derivatives can be done efficiently through this computation graph.

Example: $L(a,b,c) = c(a+2b)$

Computations:

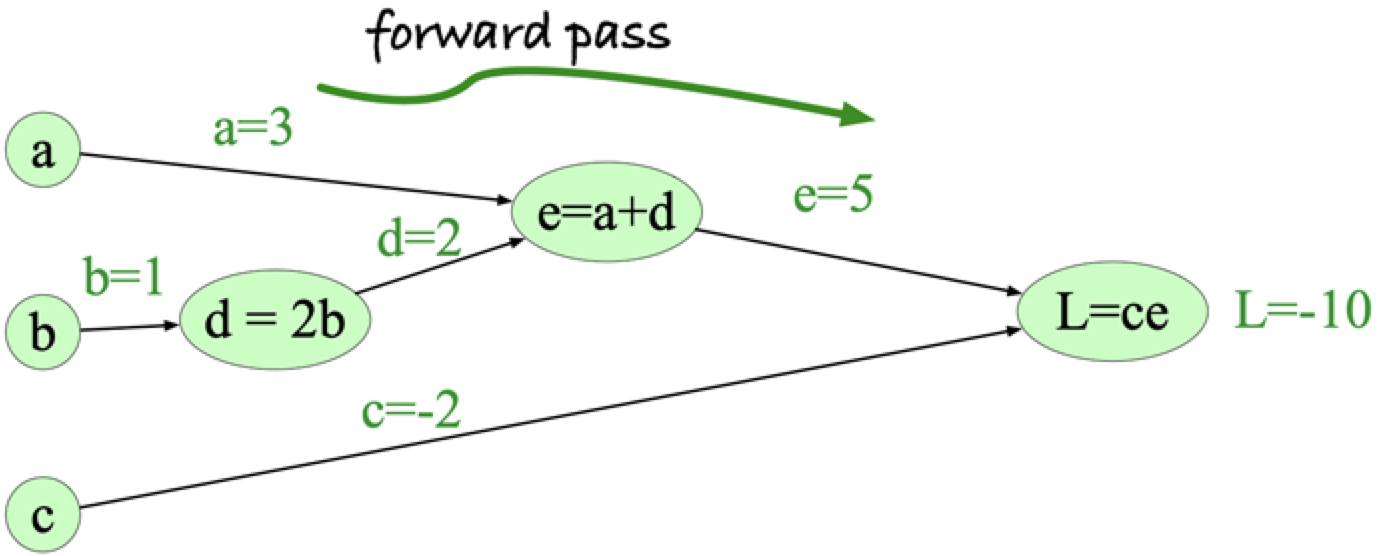
$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$



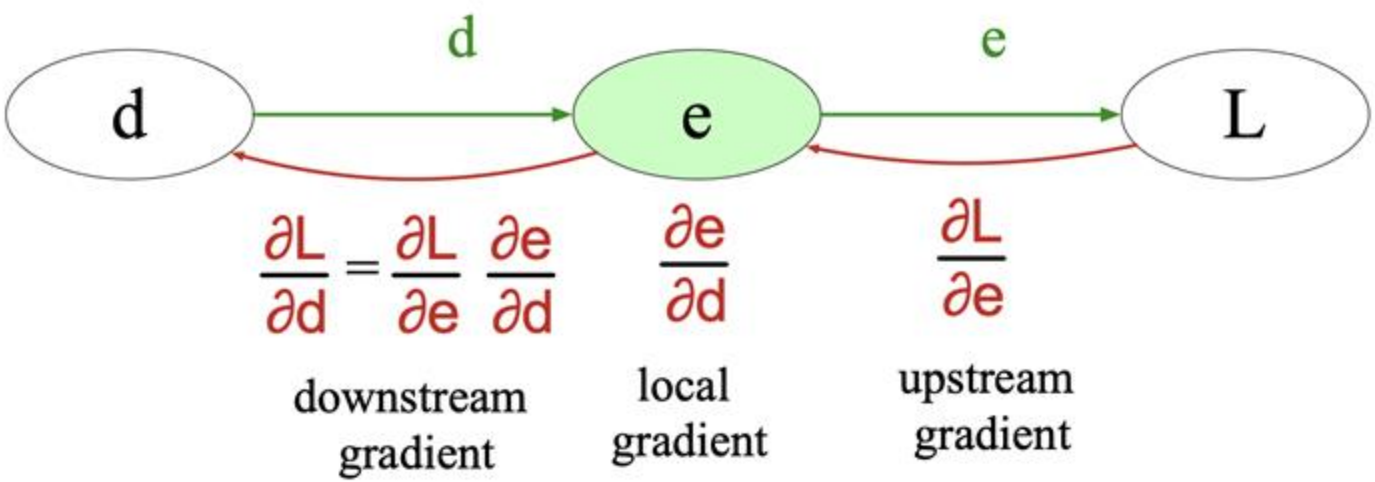
Example: $L(a,b,c) = c(a+2b)$

Computations:

$$
\begin{aligned}
d &= 2 * b \\
e &= a + d \\
L &= c * e
\end{aligned}
$$



Passing the gradient back from the final node



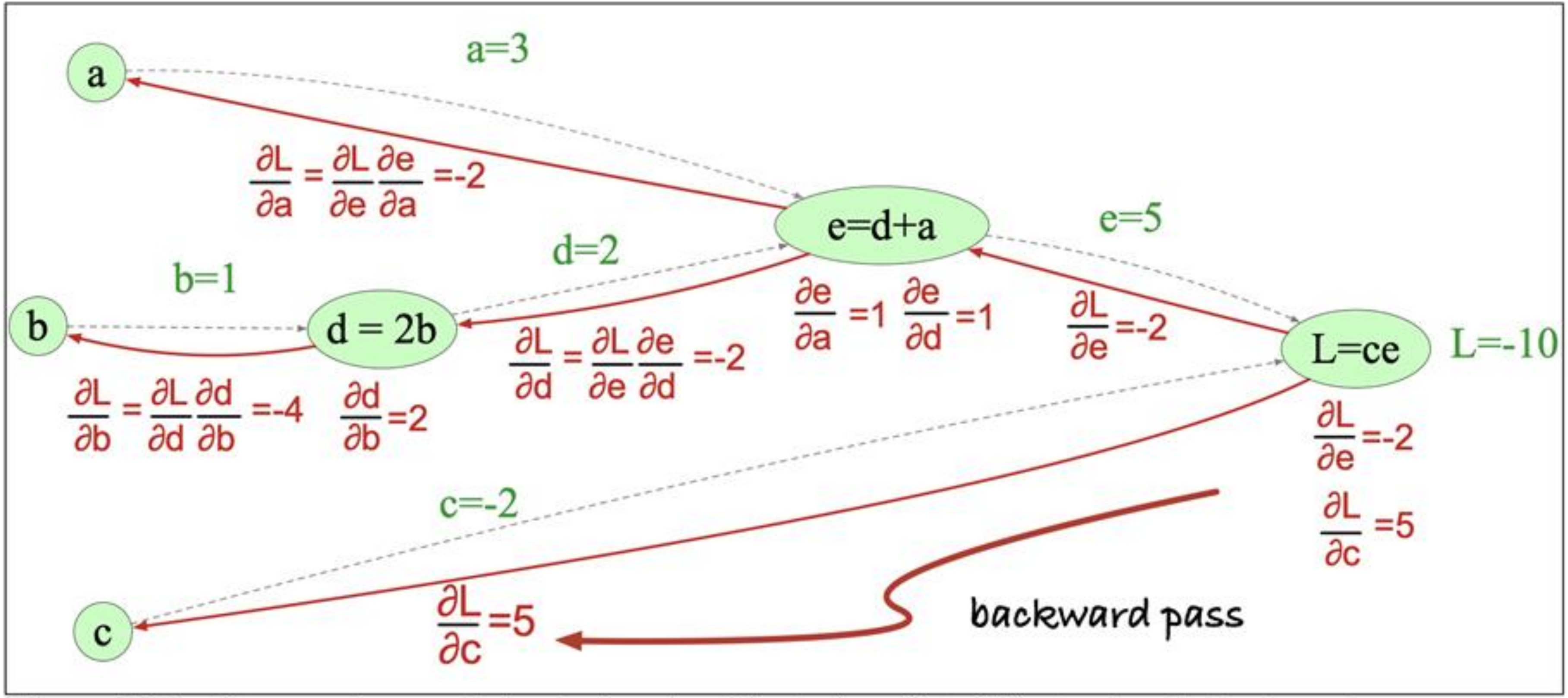
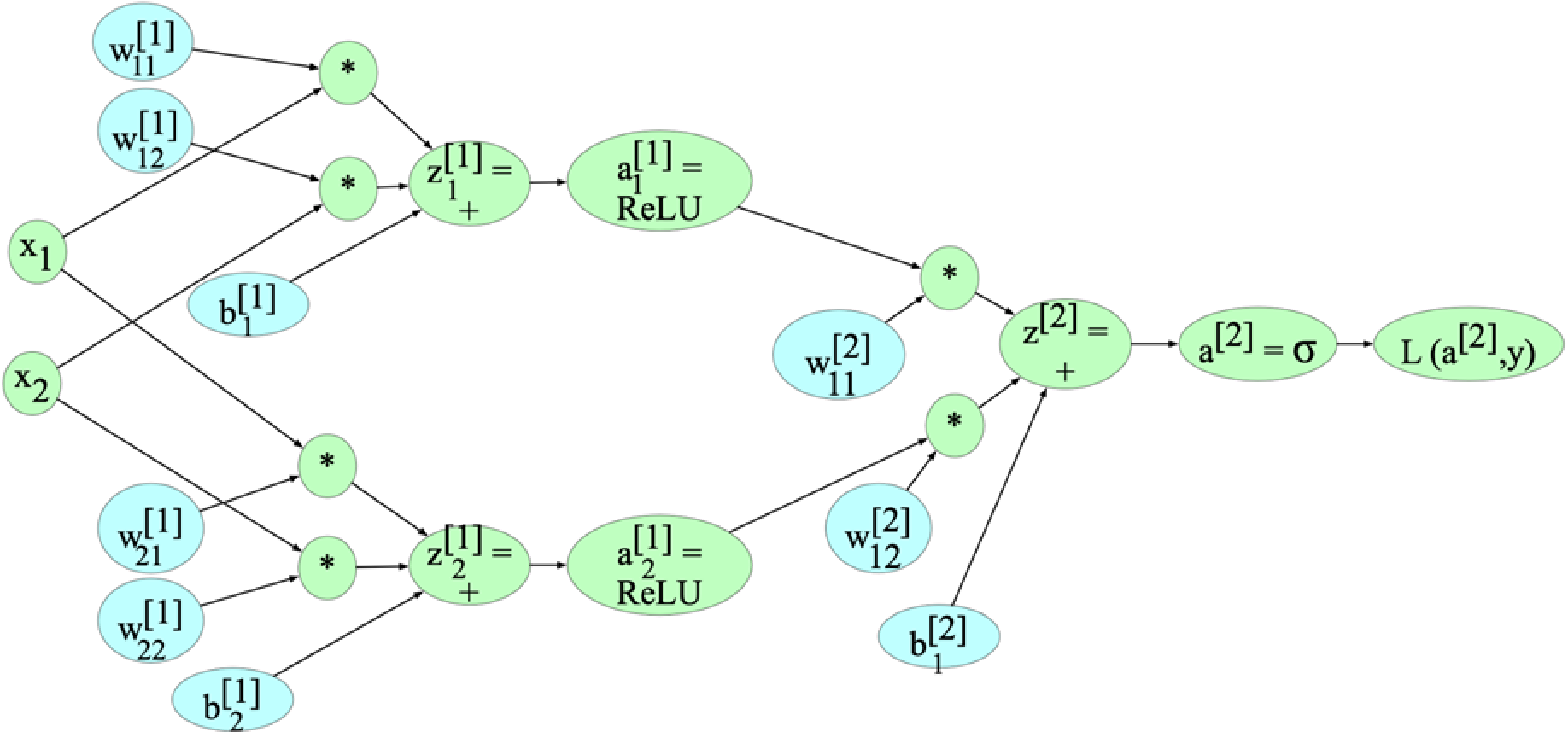


Figure 7.14 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.



$w_{11}^{[1]}$
$w_{12}^{[1]}$
x_1
x_2
$b_1^{[1]}$
$w_{21}^{[1]}$
$w_{22}^{[1]}$
$b_2^{[1]}$
*
*
*
*
$z_1^{[1]} = +$
$z_2^{[1]} = +$
$a_1^{[1]} = \text{ReLU}$
$a_2^{[1]} = \text{ReLU}$
*
*
$w_{11}^{[2]}$
$w_{12}^{[2]}$
$b_1^{[2]}$
$z^{[2]} = +$
$a^{[2]} = \sigma$
$L\,(a^{[2]}, y)$

Automatic differentiation on computation graphs

- Set up the model in a computation graph in terms of the loss function
- Forward pass: compute the loss (we need it for computing gradient)
- Backward pass:
  - Compute local gradient from the last node in the graph
  - Pass the gradient backward in the graph

Tricks for training neural networks

Stochastic gradient descent

```
theta = model parameters
for i in range(number of epochs):
    for x,y in training set:
        loss = compute_loss(theta, x, y)
        grad = compute_gradient(theta, x, y, loss)
        theta = theta - alpha * grad
```

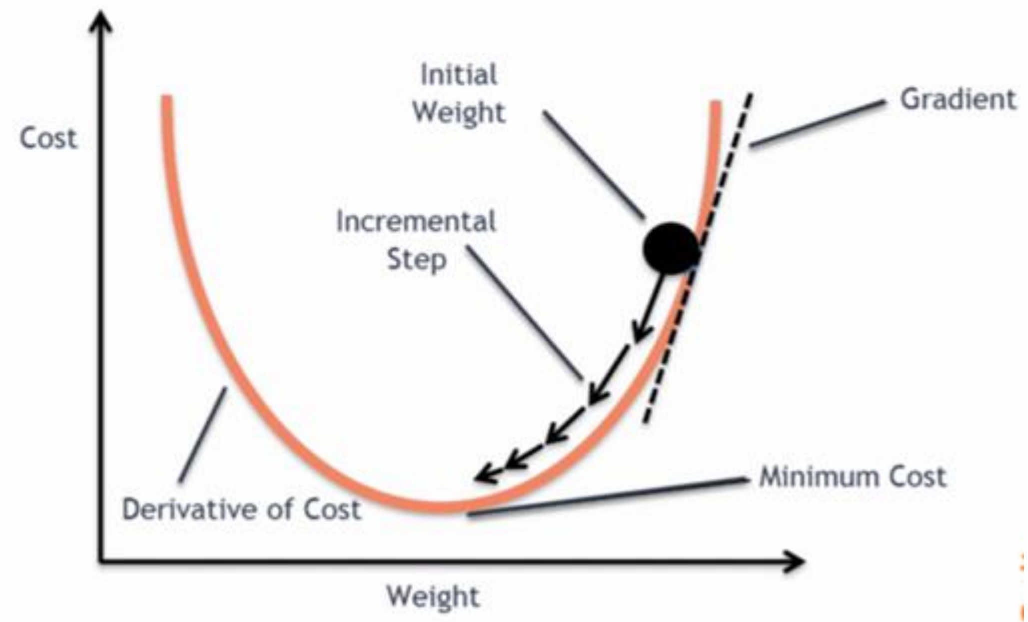
Training vocab

iteration = การอัพเดทพารามิเตอร์ทั้งหมด หนึ่งรอบ

epoch = การ loop บน training set หนึ่งครั้ง = หลาย iteration

Gradient Descent

การใช้ Error (ซึ่งมาจากการ diff objective/loss function) นำมาปรับ parameter เพื่อให้ loss function ต่ำลง



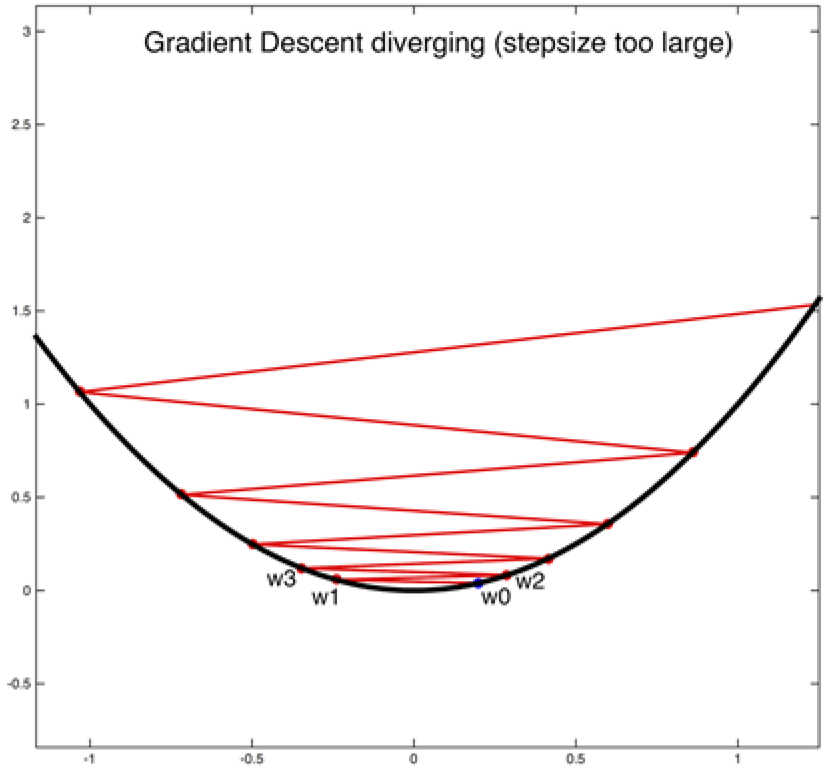
https://blog.clairvoyantsoft.com/the-ascent-of-gradient-descent-23356390836f

Problems

- Learning rate = step size = ควรจะก้าวสั้น ก้าวยาว
    - ก้าวช้า Parameter ขยับช้าเกินไป
    - ก้าวเร็ว Parameter ขยับเร็วเกินไป มัวแต่เลยไปเลยมา

ก้าวใหญ่ไป

กระโดดไป กระโดดมา



Adaptive Learning Rate Optimizer

- Momentum
- RMSProp
- AdaGrad

Momentum

momentum

learning rate

velocity

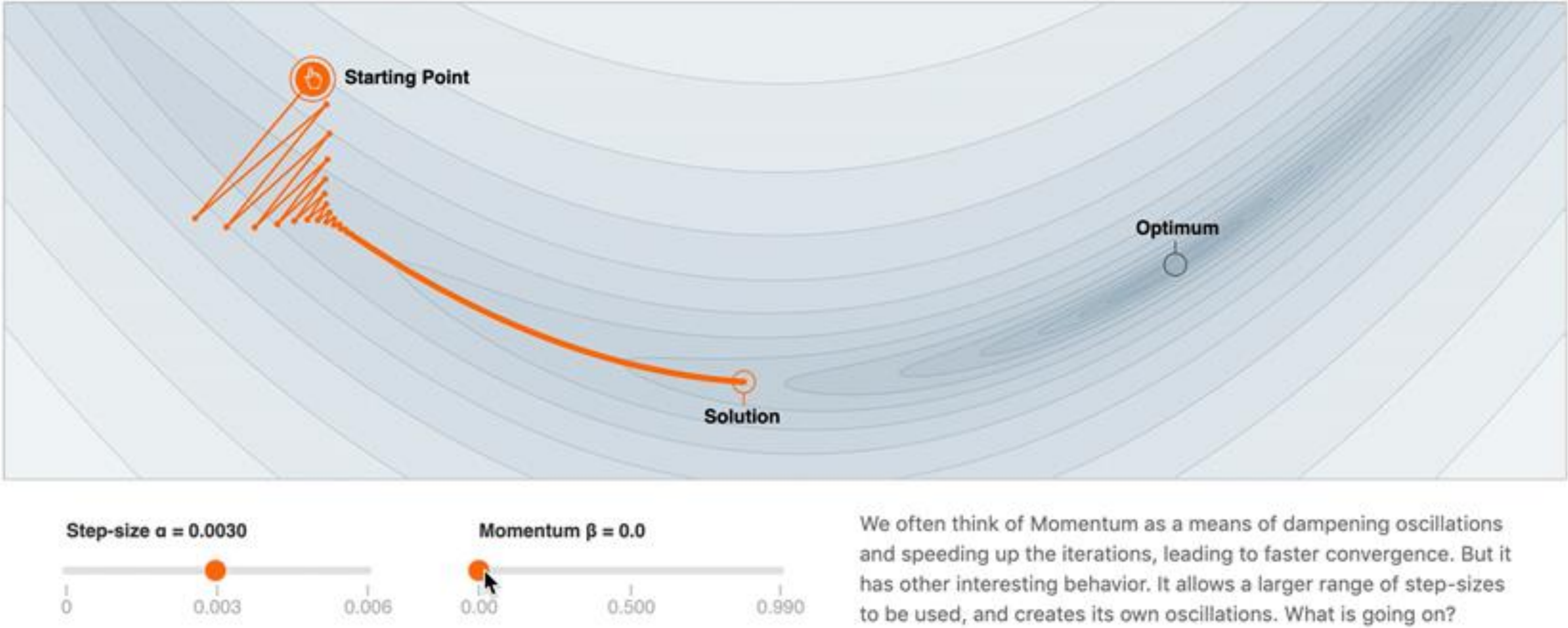
$$v_t = \gamma v_{t-1} + \eta w'_t$$

$$w_t = w_{t-1} - v_t$$

Momentum

$$v_t = \gamma v_{t-1} + \eta w'_t$$

$$w_t = w_{t-1} - v_t$$



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

RMSProp

learning rate

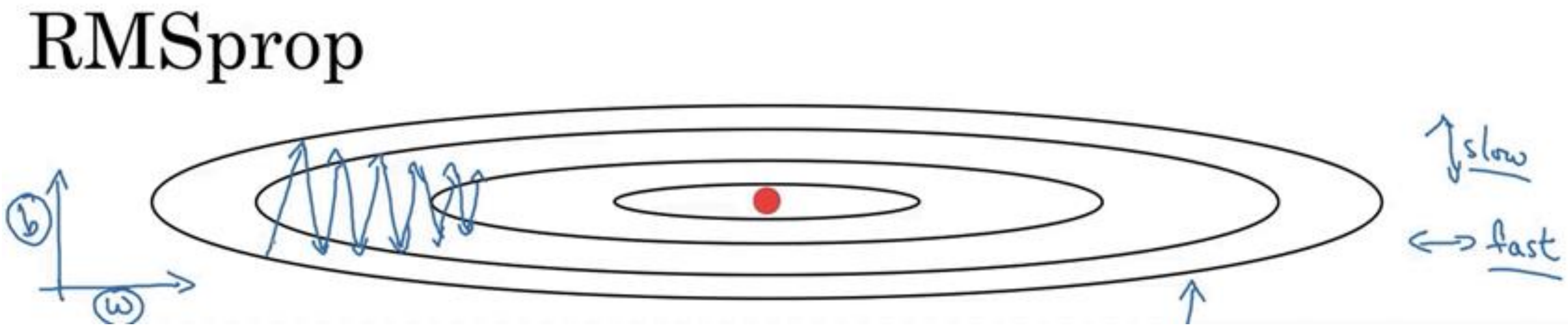
smoothing = small

$$w_{t,i} = w_{t-1,i} - \frac{\eta}{\sqrt{\epsilon + E[w'_{t,i}]_t}} w'_{t,i}$$

$$E[w'_{t,i}]_t = (1-\gamma) w'^2_{j,i} + \gamma E[w'_{t-1}]_{t-1}$$

decaying running average =
ค่าเฉลี่ยที่ให้ค่าน้ำหนักของของใหม่มากกว่า

decay rate



Momentum (blue) and RMSprop (green)

AdaGrad

learning rate

$$w_{t,i} = w_{t-1,i} - \frac{\eta}{\sqrt{\epsilon + \sum_{j=1}^{t-1} w'^2_{j,i}}} w'_{t,i}$$

ค่านี้จะใหญ่ขึ้นเรื่อย ๆ ทำให้ก้าวช้าลงเรื่อย ๆ

Adaptive Learning Rate

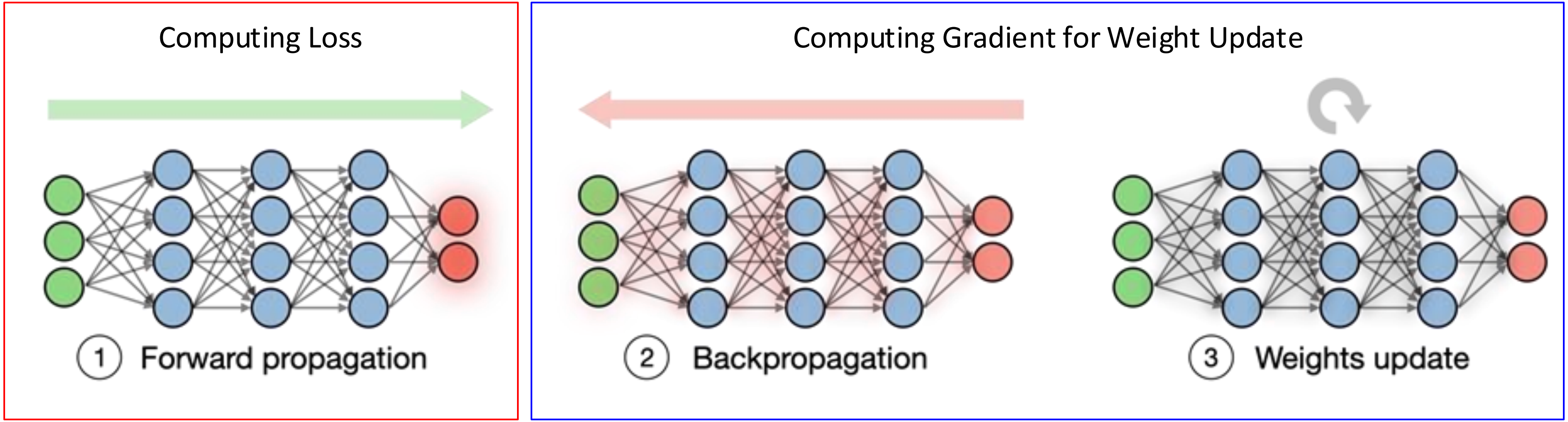
```
theta = model parameters
optimizer = AdaptiveOptimizer()  # e.g., Adam, RMSProp, etc.

for i in range(number of epochs):
    for x, y in training set:
        loss = compute_loss(theta, x, y)
        grad = compute_gradient(theta, x, y, loss)
        optimizer.update_statistics(grad)
        alpha = optimizer.get_learning_rate()
        theta = theta - alpha * optimizer.modify_gradient(grad)
```

Optimizers

- ไม่มีข้อตกลงแน่นอนว่าอันไหนดีกว่าอันไหน ในสถานการณ์ไหน
- Optimizer ที่เป็นที่นิยมแต่ไม่ได้พูดถึง
    - Adam
    - AdaDelta

Training Process



https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks

Text	Label

Stochastic Gradient Descent

Text	Label

Batch Gradient Descent

Mini-batching

เพราะคำนวณ gradient ใช้เวลานานเกินไป ถ้าคำนวณจากทุกแถว

mini-batch size = 3
number of rows = 15
number of mini-batches = 5

1 epoch = 5 iterations

Text	Label

Adaptive Learning Rate + Minibatching

```
theta = model parameters
optimizer = AdaptiveOptimizer()  # e.g., Adam, RMSProp, etc.
data_loader = DataLoader(training set)

for i in range(number of epochs):
    for x_batch, y_batch in data_loader:
        loss = compute_loss(theta, x, y)
        grad = compute_gradient(theta, x, y, loss)
        optimizer.update_statistics(grad)
        alpha = optimizer.get_learning_rate()
        theta = theta - alpha * optimizer.modify_gradient(grad)
```