Lecture 4
Theory of Deep Learning

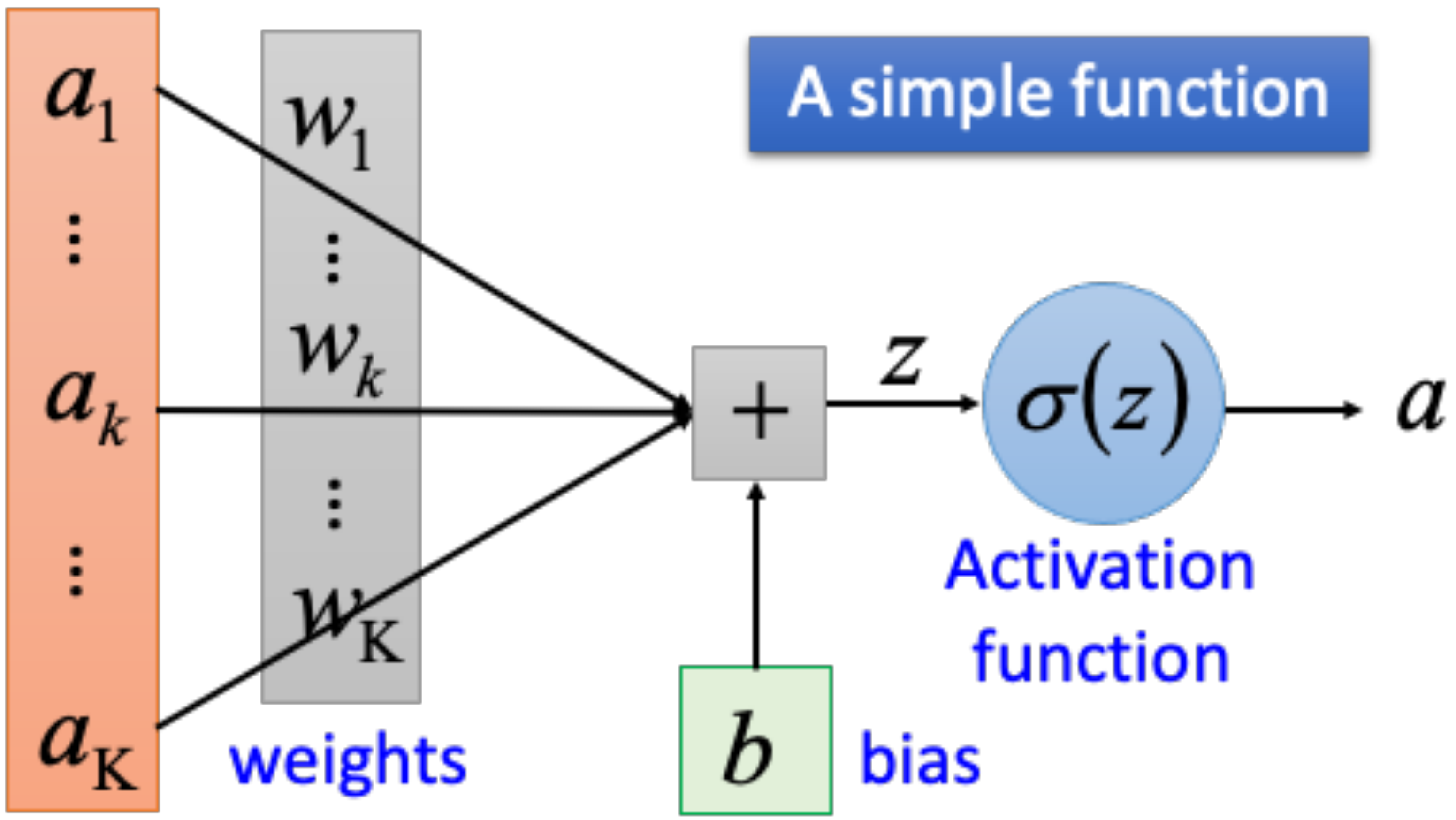
Xu Yuan
University of Louisiana at Lafayette

Key Elements in Neural Network

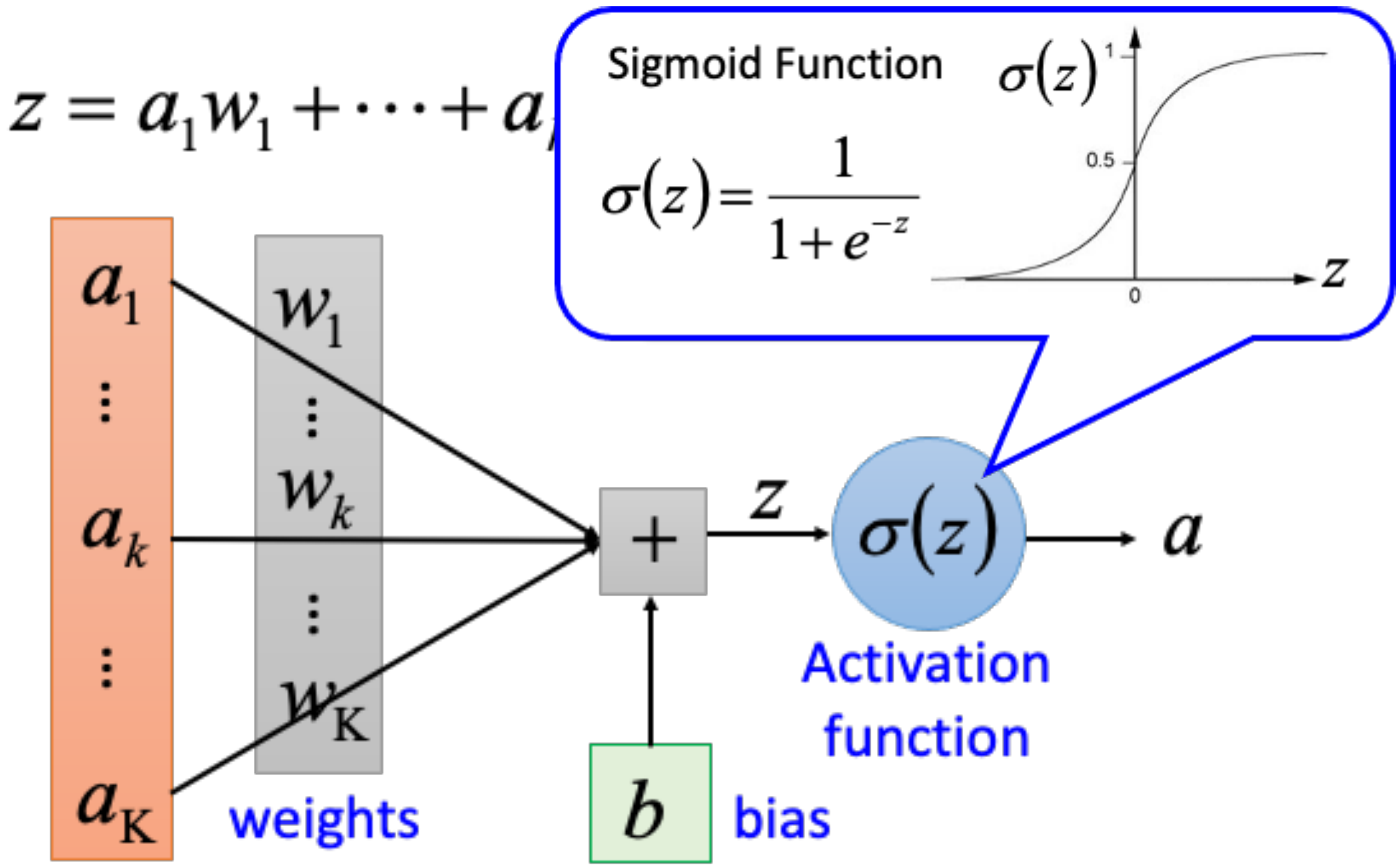
- Activation Function
- Softmax Function
- Mathematical Expression for Network Function
- Learning Rate
- Gradient Descent
- Momentum
- Maxout
- Dropout

Single Neuron

$$z = a_1 w_1 + \cdots + a_k w_k + \cdots + a_K w_K + b$$

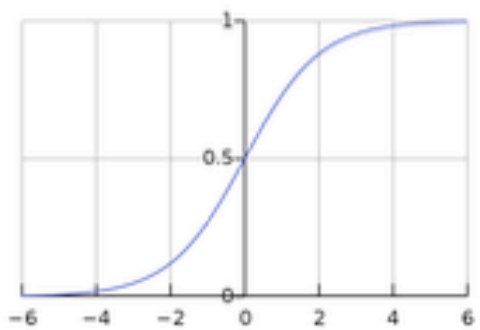


Activation Function

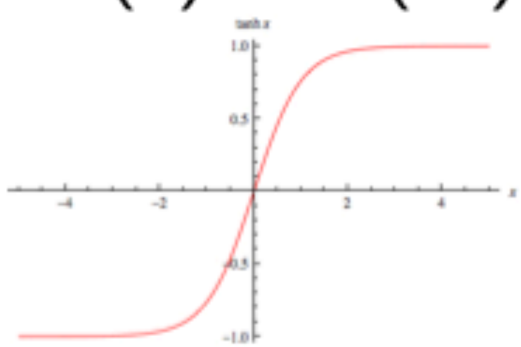


Various Activation Function

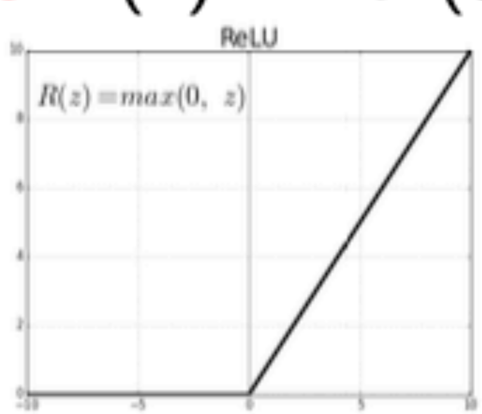
Sigmoid: $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$



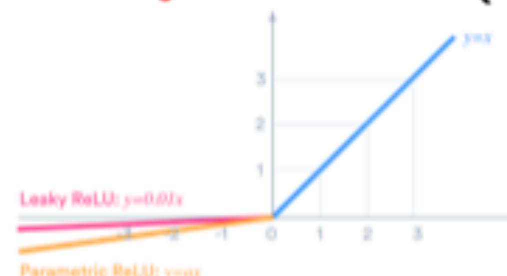
tanh: $f(x) = 2\sigma(2x) - 1$



ReLU: $f(x) = \max(0, x)$



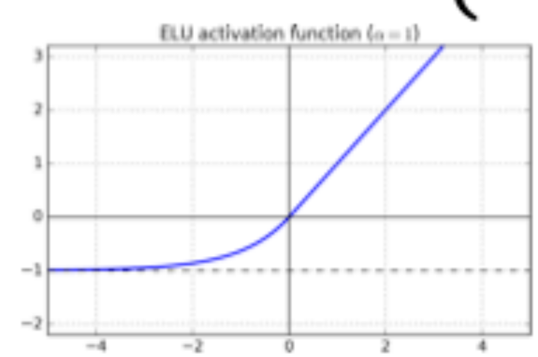
Leaky ReLU: $f(x) = \max(\alpha x, x)$



Maxout

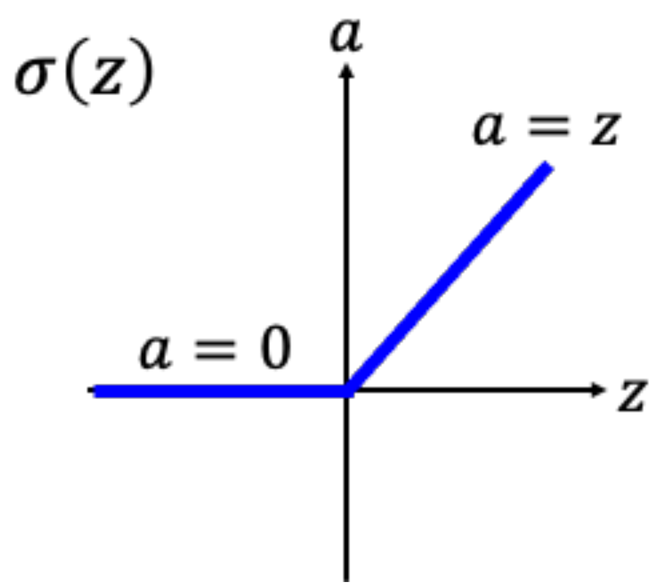
$$\max(\mathrm{w}_1^T\mathrm{x} + b_1, \mathrm{w}_2^T\mathrm{x} + b_2)$$

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



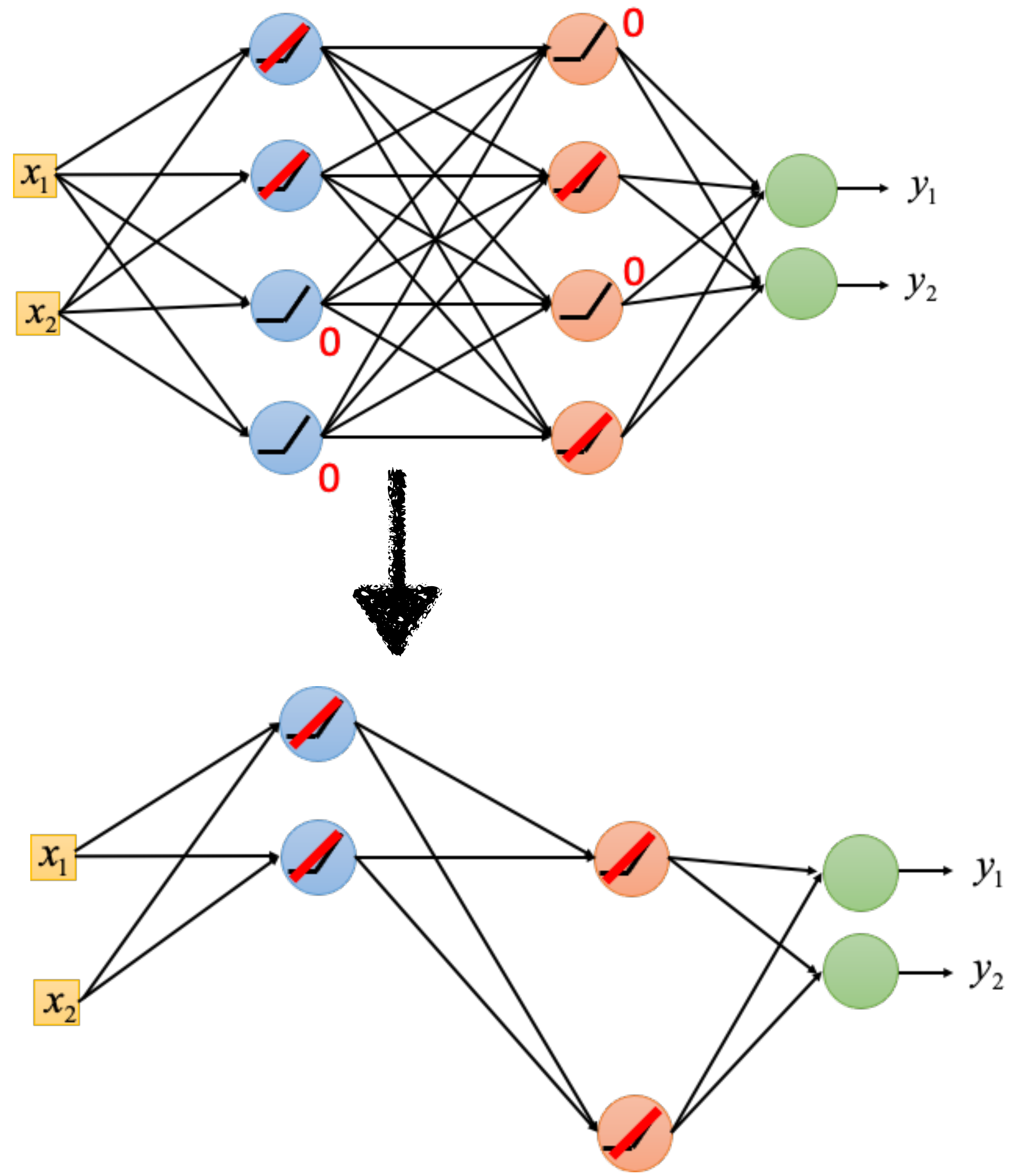
ReLU

- **Rectifier Linear Unit**



a=max(0, z)

ReLU



Output Layer

- **Softmax Layer**

$$y_1 = e^{z_1} \Big/ \sum_{j=1}^{3} e^{z_j}$$

$$y_2 = e^{z_2} \Big/ \sum_{j=1}^{3} e^{z_j}$$

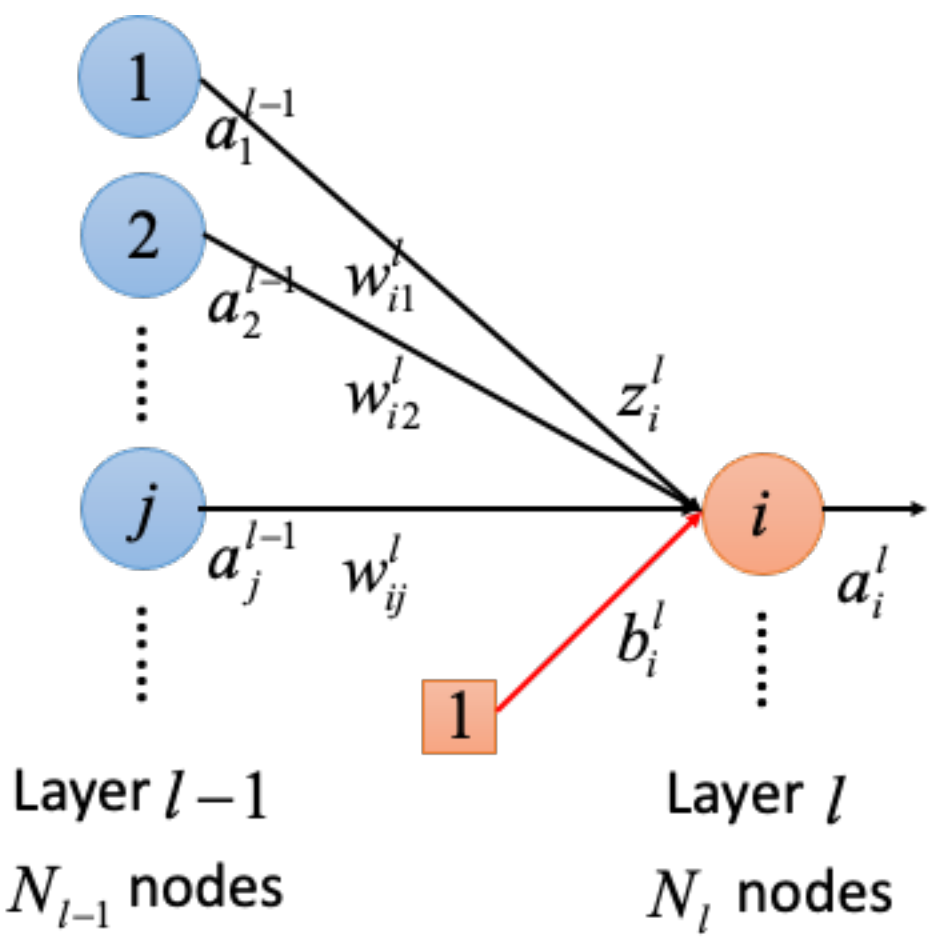
$$y_3 = e^{z_3} \Big/ \sum_{j=1}^{3} e^{z_j}$$

```
model.add( Dense(output_dim=10 ) )
model.add( Activation('softmax') )
```

```
model.add( Dense( input_dim=28*28,
                  output_dim=500 ))
model.add( Activation('sigmoid') )
```

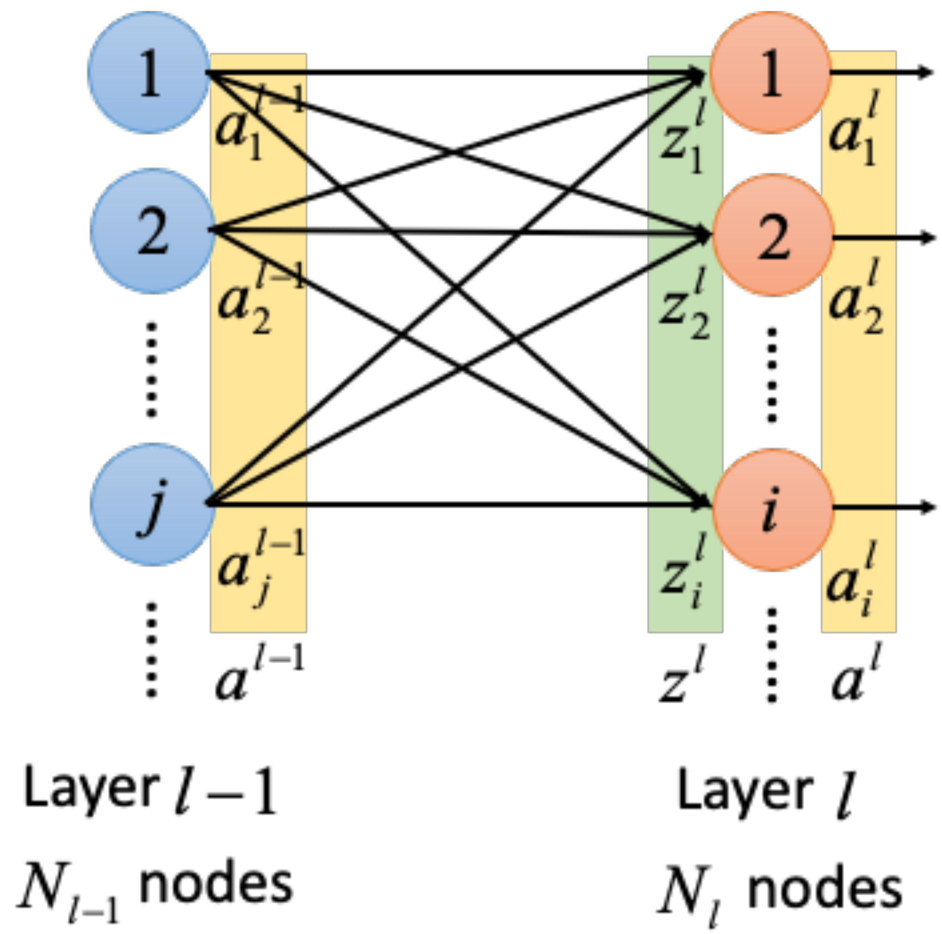
```
model2.add(Dense(output_dim=100))
model2.add(Activation('relu'))
model2.add(Dense(output_dim=10))
model2.add(Activation('softmax'))
```

Activation Functions

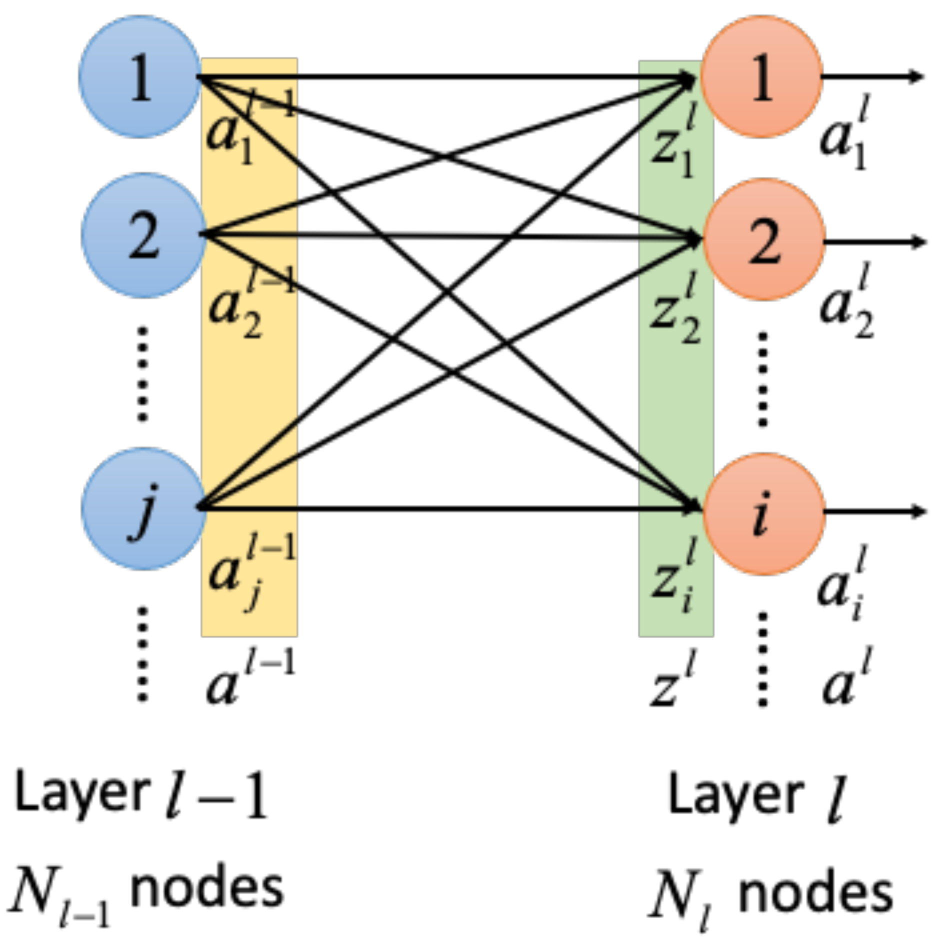


$$z_i^l = w_{i1}^l a_1^{l-1} + w_{i2}^l a_2^{l-1} \ldots + b_i^l$$

Relations between Layer Outputs



Relations between Layer Outputs



$$z_1^l = w_{11}^l a_1^{l-1} + w_{12}^l a_2^{l-1} + \cdots + b_1^l$$

$$z_2^l = w_{21}^l a_1^{l-1} + w_{22}^l a_2^{l-1} + \cdots + b_2^l$$

$$\vdots$$

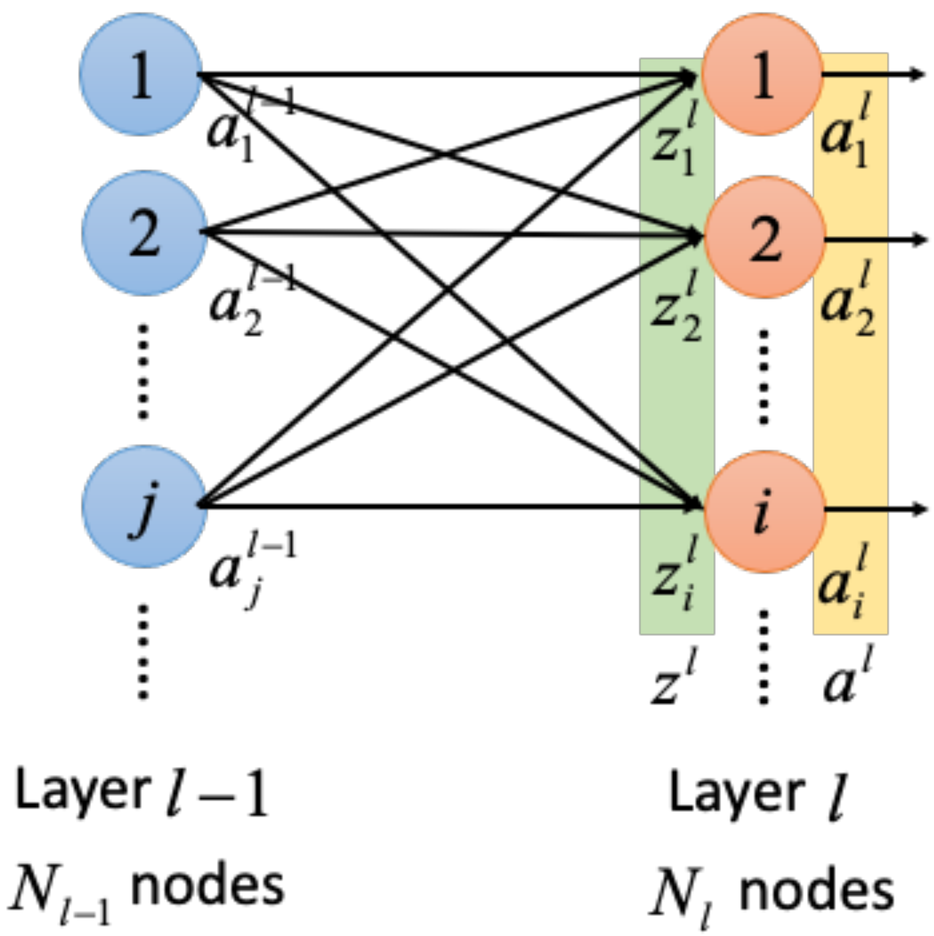
$$z_i^l = w_{i1}^l a_1^{l-1} + w_{i2}^l a_2^{l-1} + \cdots + b_i^l$$

$$\vdots$$

$$\begin{bmatrix} z_1^l \\ z_2^l \\ \vdots \\ z_i^l \\ \vdots \end{bmatrix} = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots \\ w_{21}^l & w_{22}^l & \\ \vdots & & \ddots \\ & & \end{bmatrix} \begin{bmatrix} a_1^{l-1} \\ a_2^{l-1} \\ \vdots \\ a_i^{l-1} \\ \vdots \end{bmatrix} + \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_i^l \\ \vdots \end{bmatrix}$$

$$z^l = W^l a^{l-1} + b^l$$

Relations between Layer Outputs

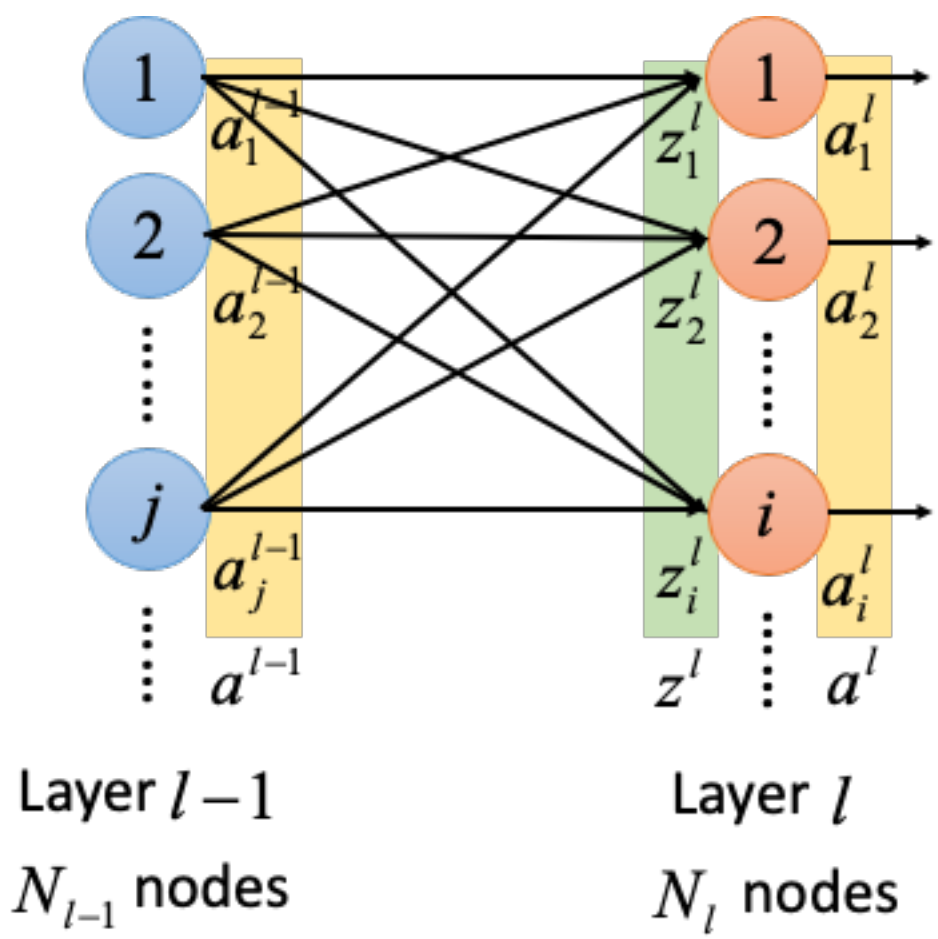


$$a_i^l = \sigma\left(z_i^l\right)$$

$$\begin{bmatrix} a_1^l \\ a_2^l \\ \vdots \\ a_i^l \\ \vdots \end{bmatrix} = \begin{bmatrix} \sigma\left(z_1^l\right) \\ \sigma\left(z_2^l\right) \\ \vdots \\ \sigma\left(z_i^l\right) \\ \vdots \end{bmatrix}$$

$$a^l = \sigma\left(z^l\right)$$

Relations between Layer Outputs

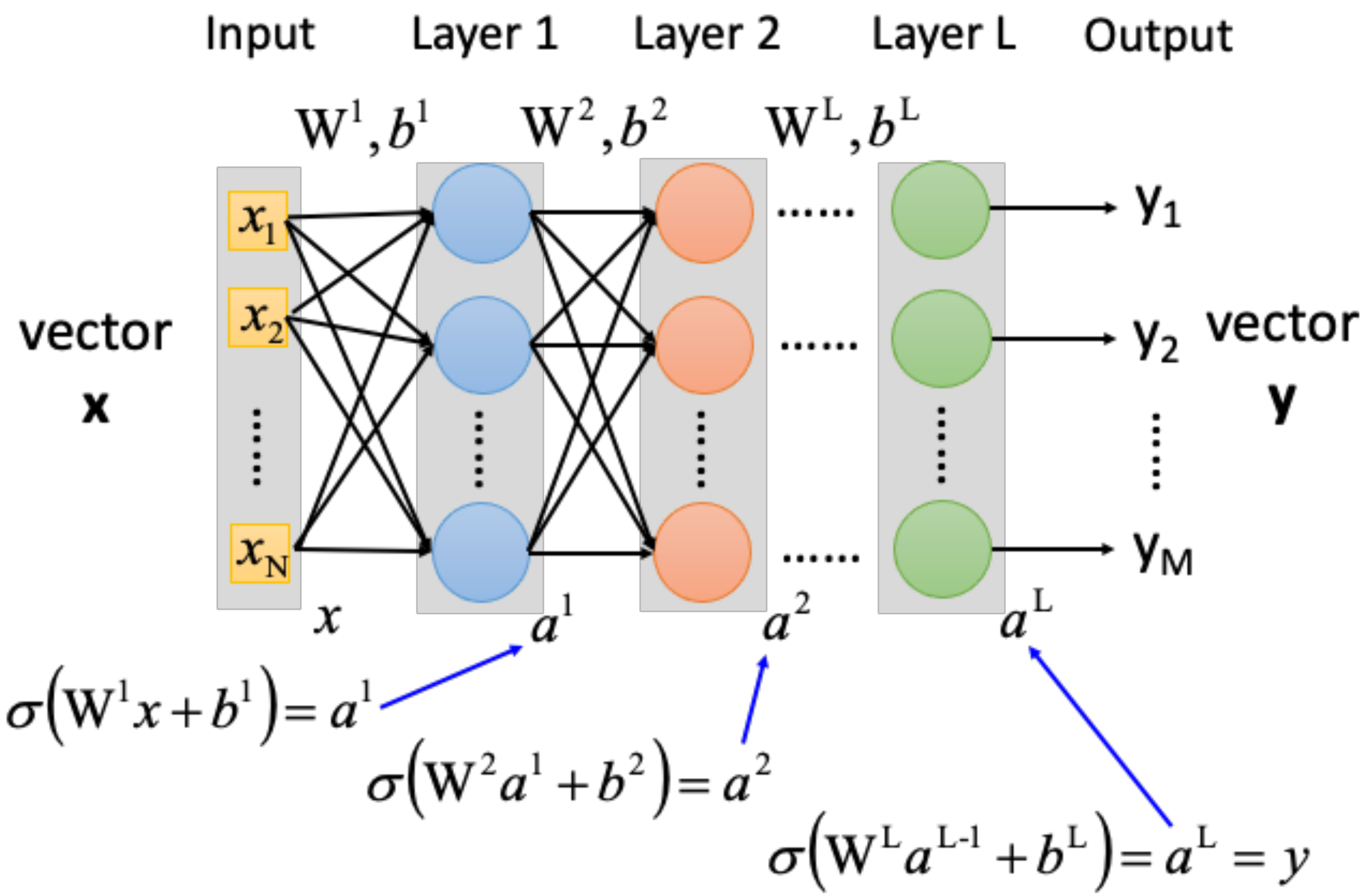


$$z^l = W^l a^{l-1} + b^l$$

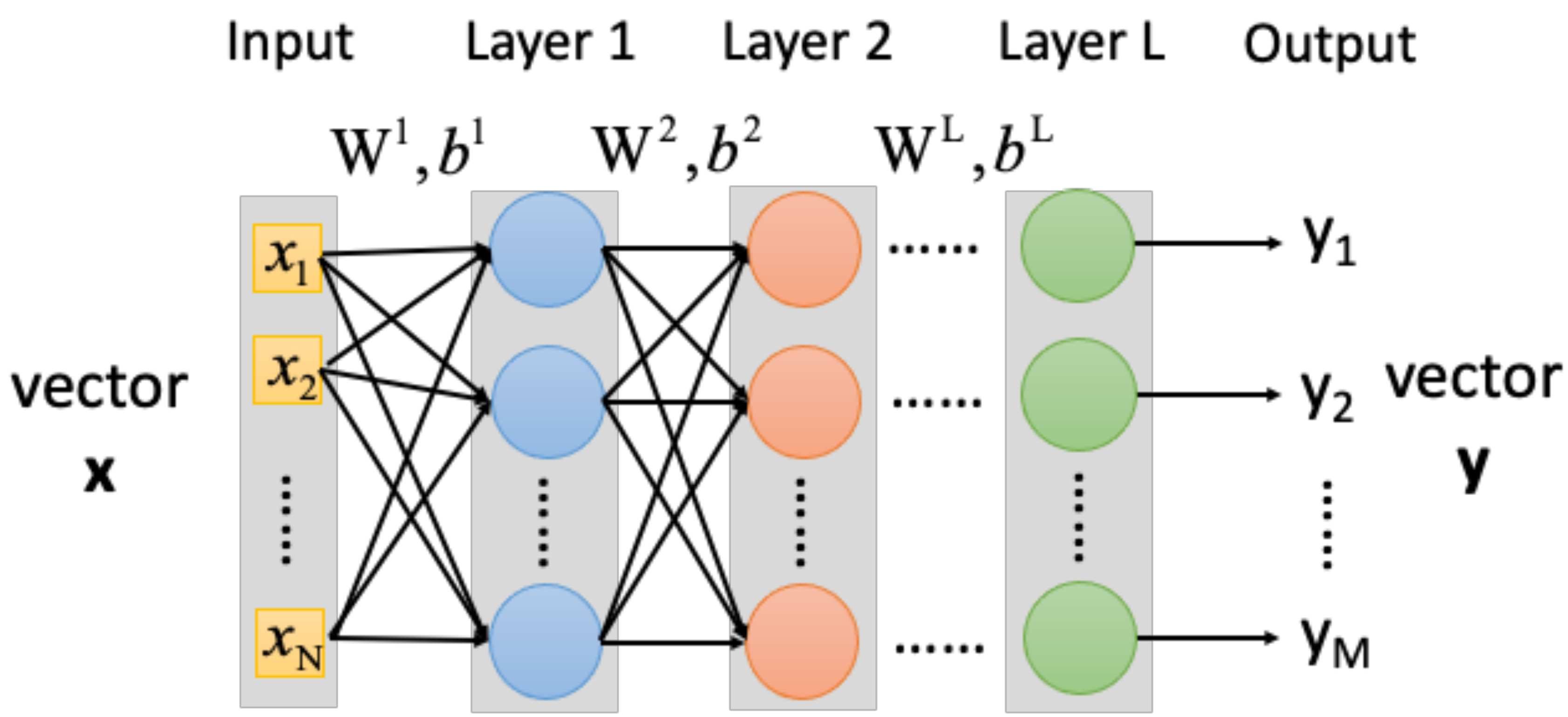
$$a^l = \sigma(z^l)$$

$$a^l = \sigma(W^l a^{l-1} + b^l)$$

Functions of Neural Network

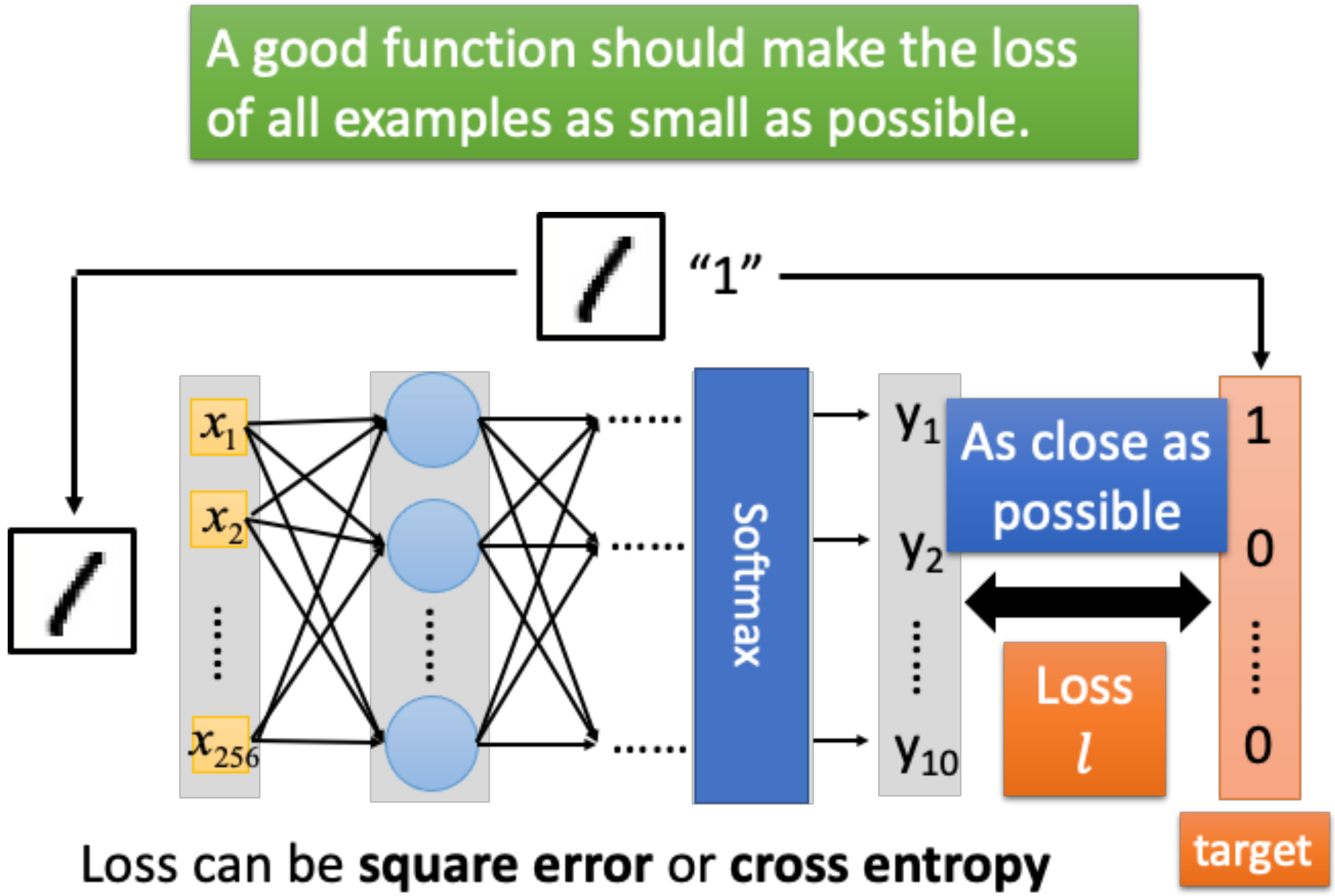


Uniform Expression



$$
\begin{aligned}
y &= f(x) \\
&= \sigma\left(W^L \ldots \sigma\left(W^2 \sigma\left(W^1 x + b^1\right) + b^1\right) \ldots + b^L\right)
\end{aligned}
$$

Good Function = Loss as Small as Possible



Loss can be **square error** or **cross entropy** between the network output and target

Loss Functions

- **Square Error:** $\sum_{i=1}^{10}(y_i - \hat{y}_i)^2$

```
model.compile(loss='mse',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

- **Cross-entropy**$-\sum_{i=1}^{10}\hat{y}_i ln y_i$

```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

Best Functions = best Parameters

$$y = f(x) = \sigma\left(W^L \ldots \sigma\left(W^2 \sigma\left(W^1 x + b^1\right) + b^1\right) \ldots + b^L\right)$$

function set

because different parameters W and b lead to different function

Formal way to define a function set:

$$f(x; \underline{\theta}) \rightarrow \text{parameter set}$$

$$\theta = \left\{W^1, b^1, W^2, b^2 \cdots W^L, b^L\right\}$$

Pick the “best” function f* → Pick the “best” parameter set θ*

How to Determine Parameters

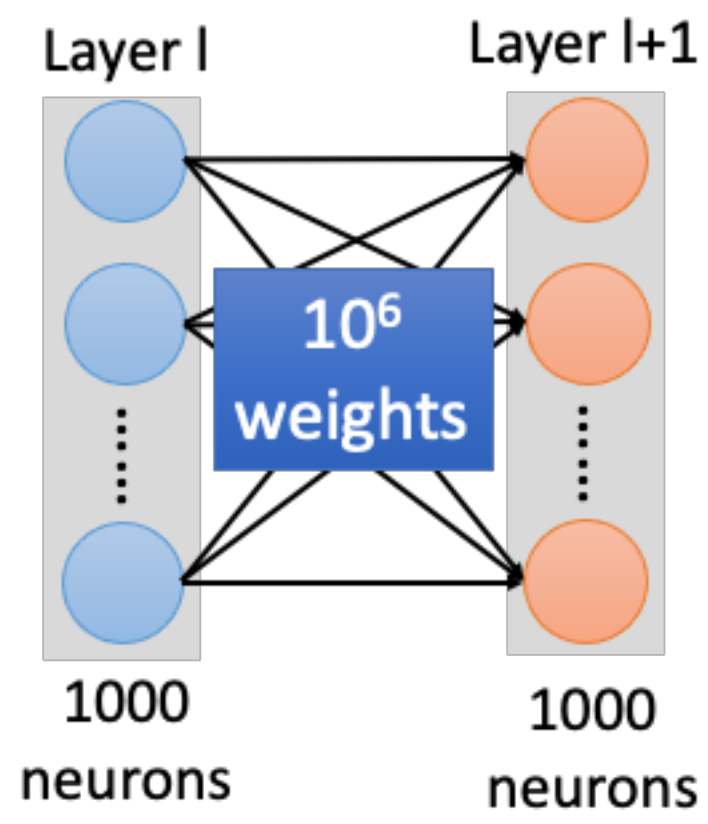
Find ***network parameters θ^**** that minimize total loss L

Enumerate all possible values

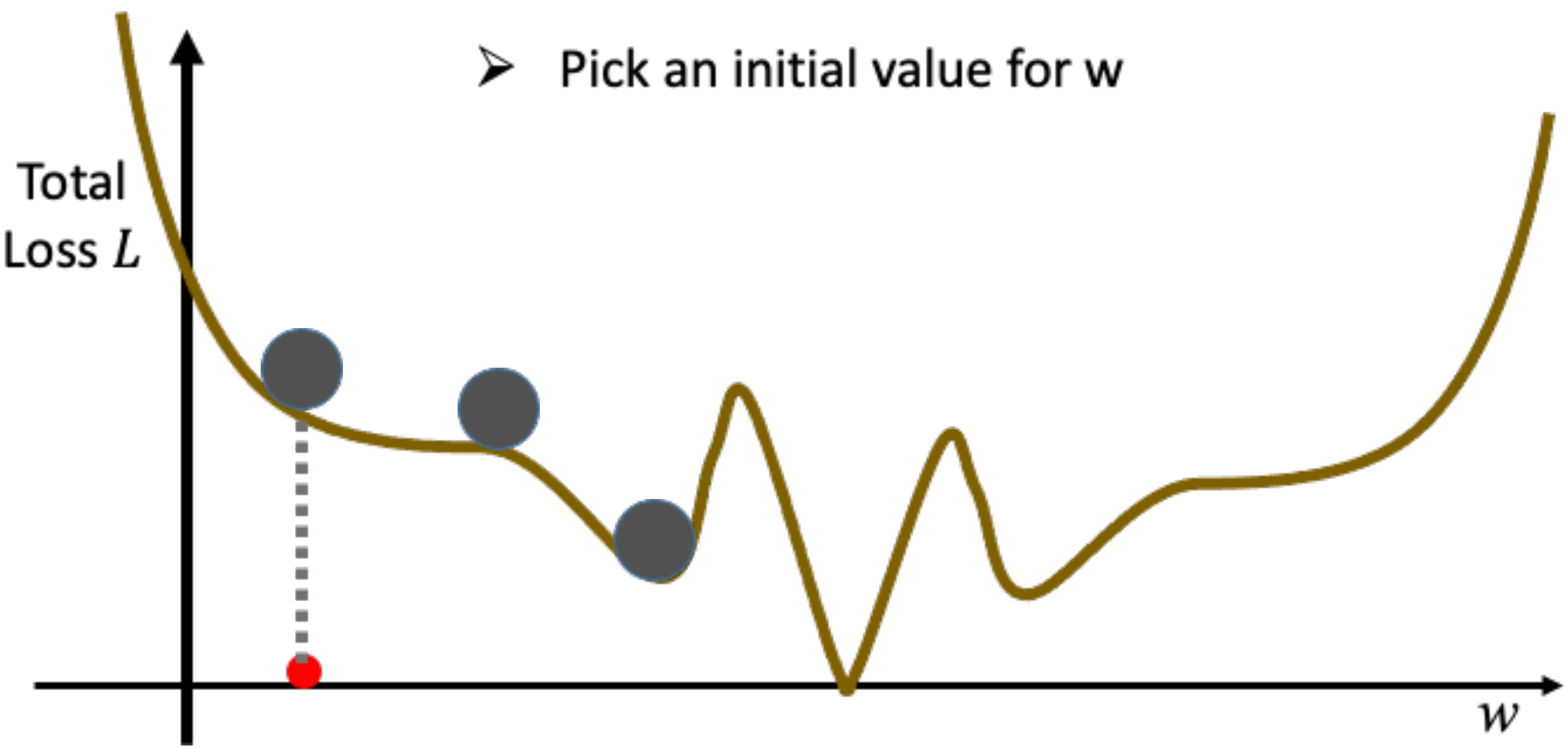
Network parameters $\theta =$
$$\underbrace{\{w_1, w_2, w_3, \cdots, b_1, b_2, b_3, \cdots\}}$$

Millions of parameters

E.g. speech recognition: 8 layers and 1000 neurons each layer

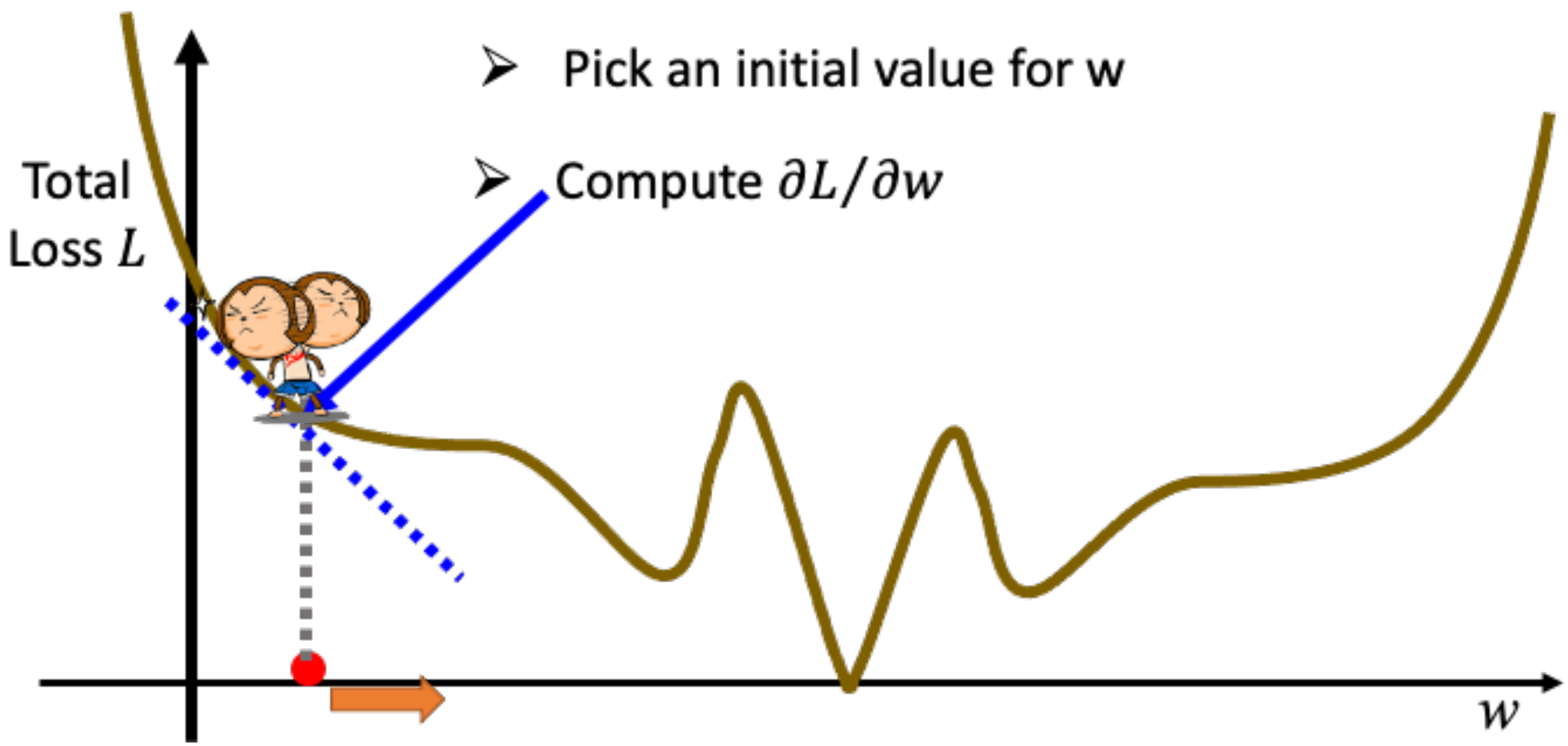


Gradient Descent



Network parameters $\theta =$
$\{w_1, w_2, \cdots, b_1, b_2, \cdots\}$

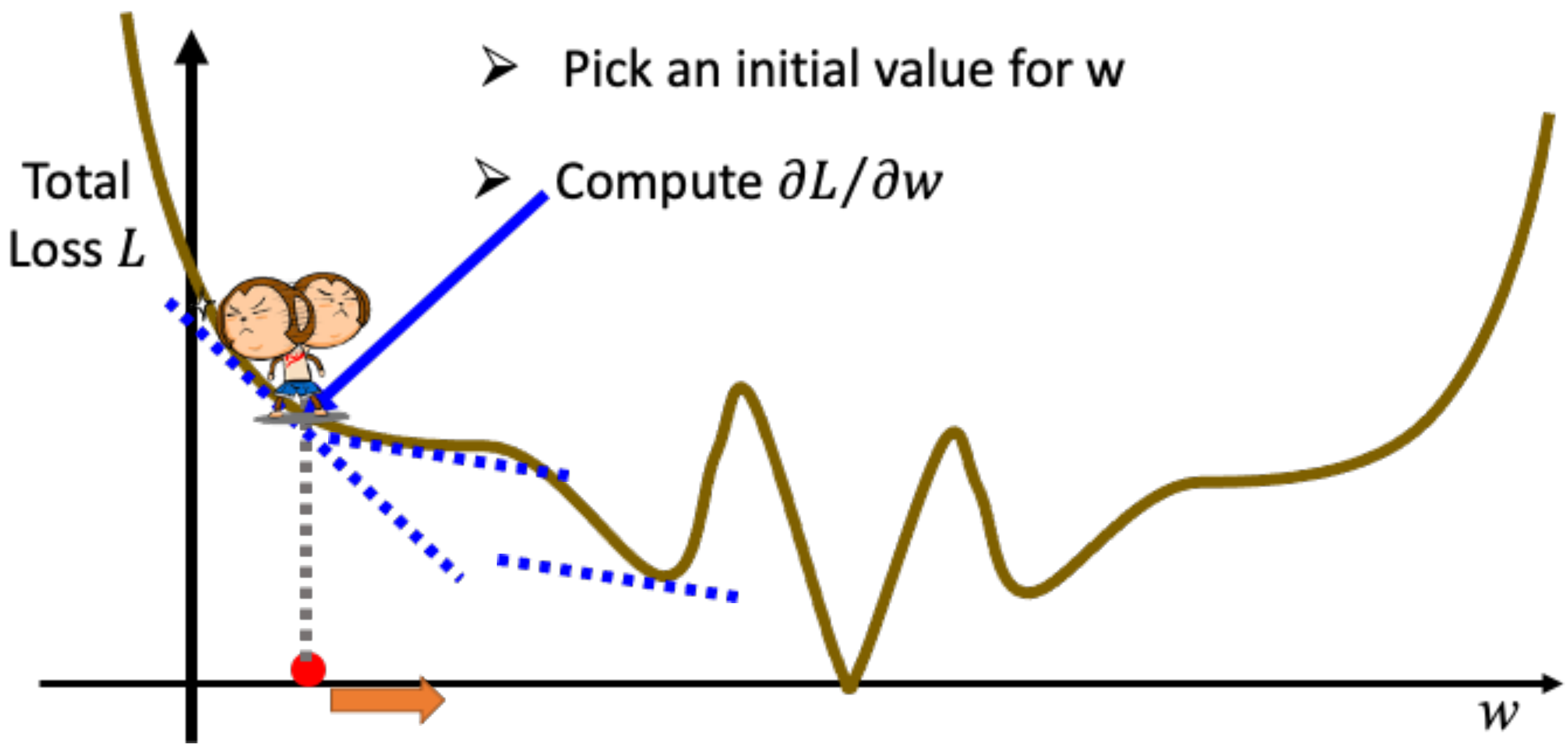
Gradient Descent



Negative → Increase w

Positive → Decrease w

Gradient Descent

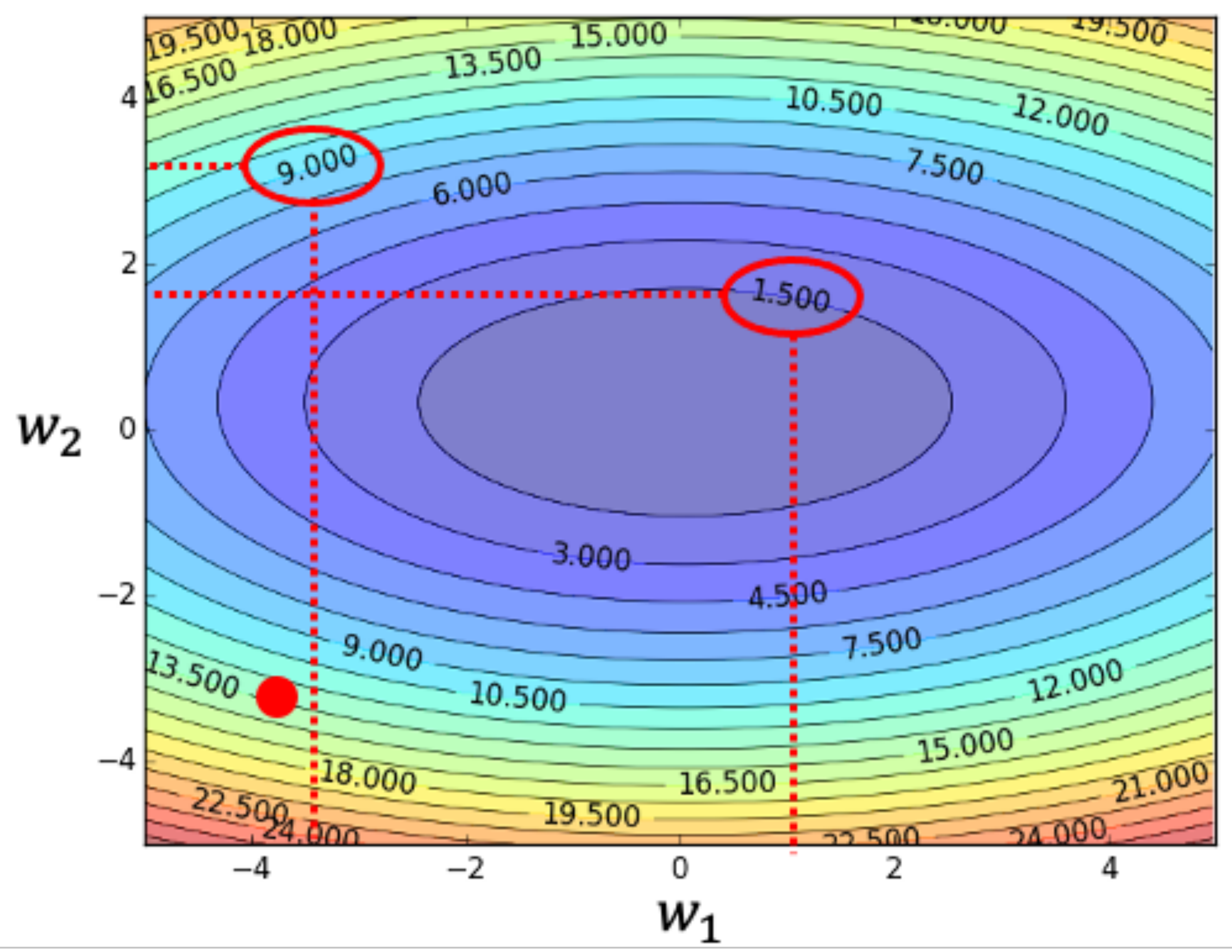


$$w \leftarrow w - \eta \partial L/\partial w$$

η is called *"learning rate"*

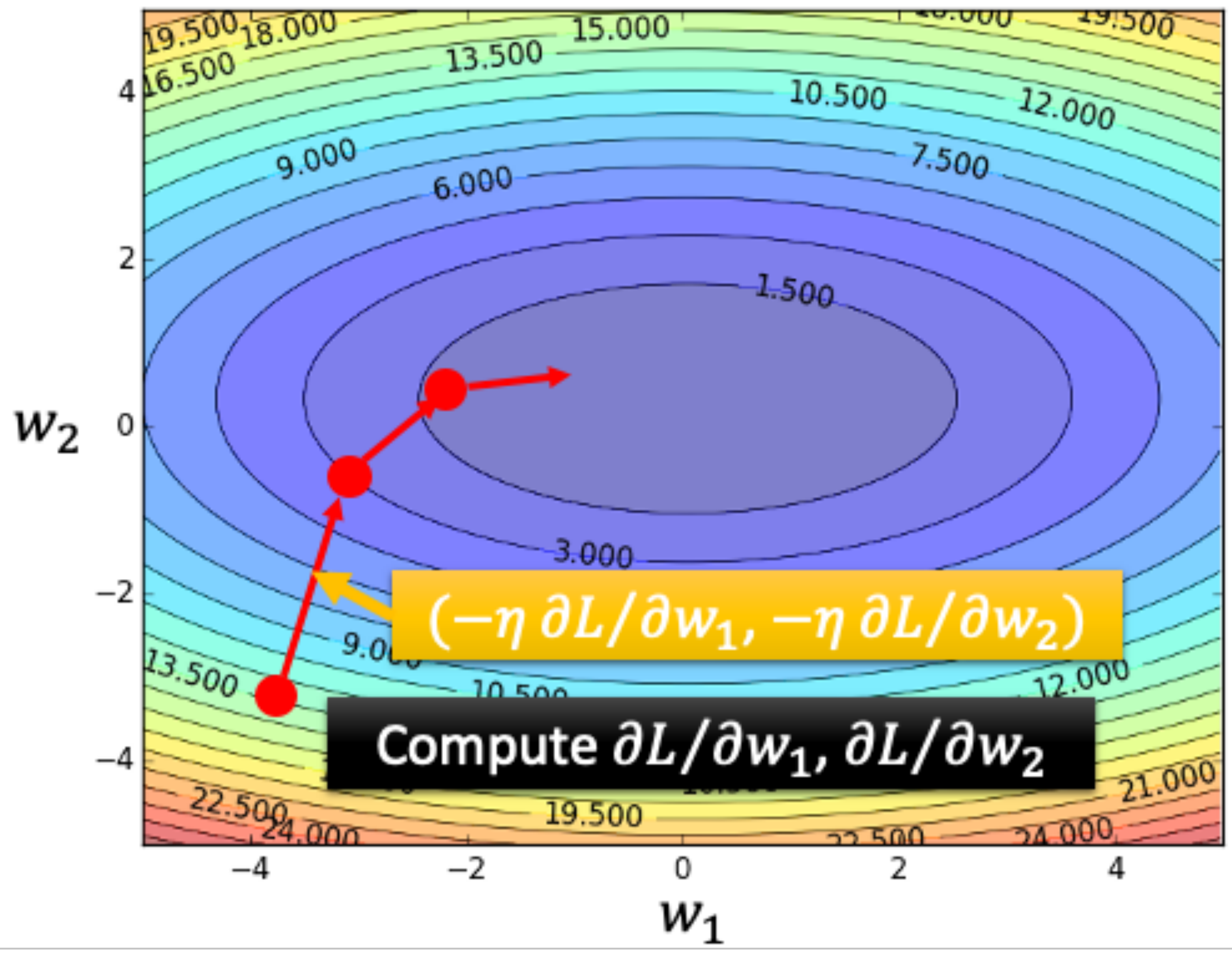
Until $\partial L/\partial w$ is approximately small

Gradient Descent

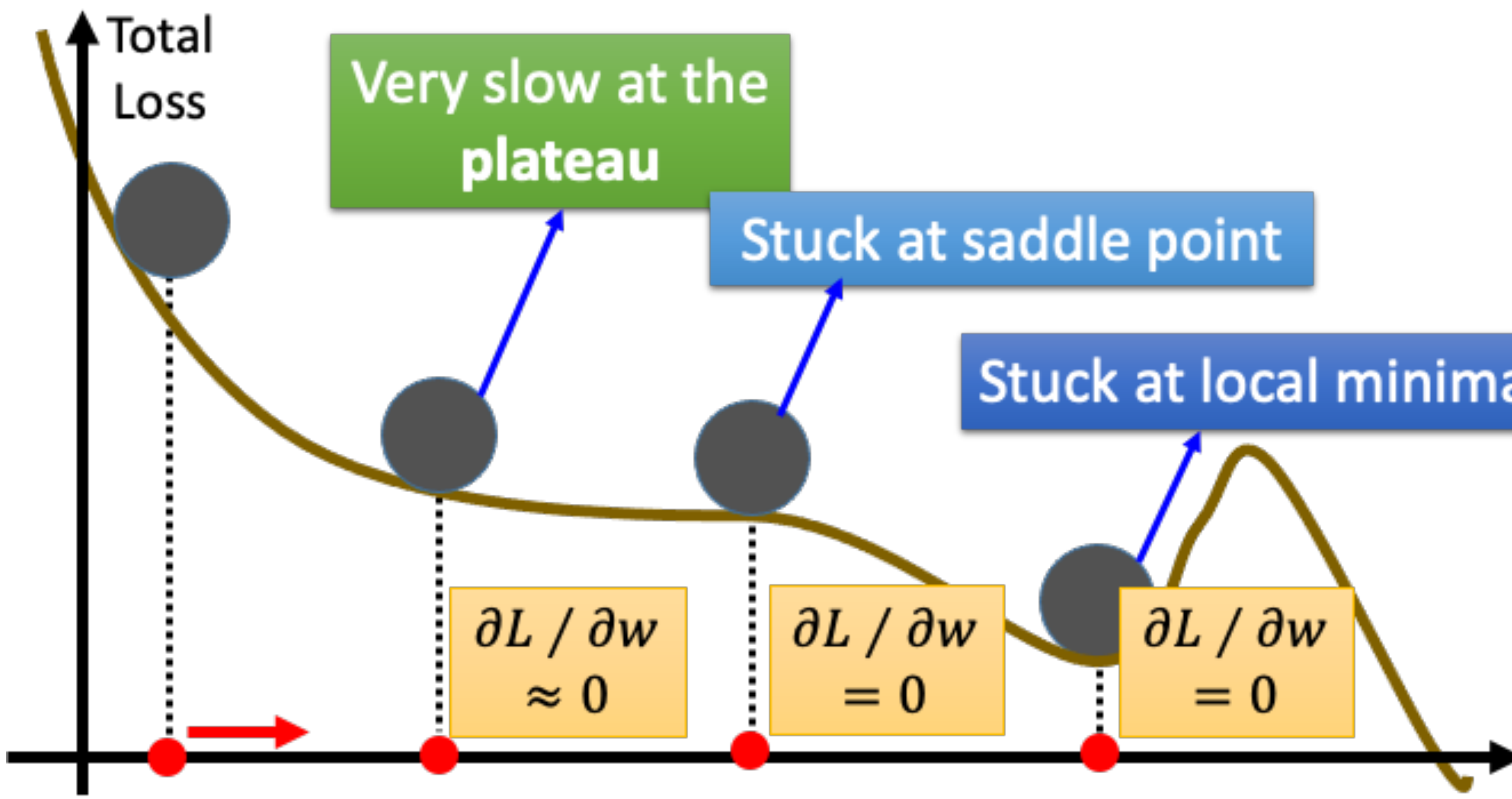


Randomly pick up a start point

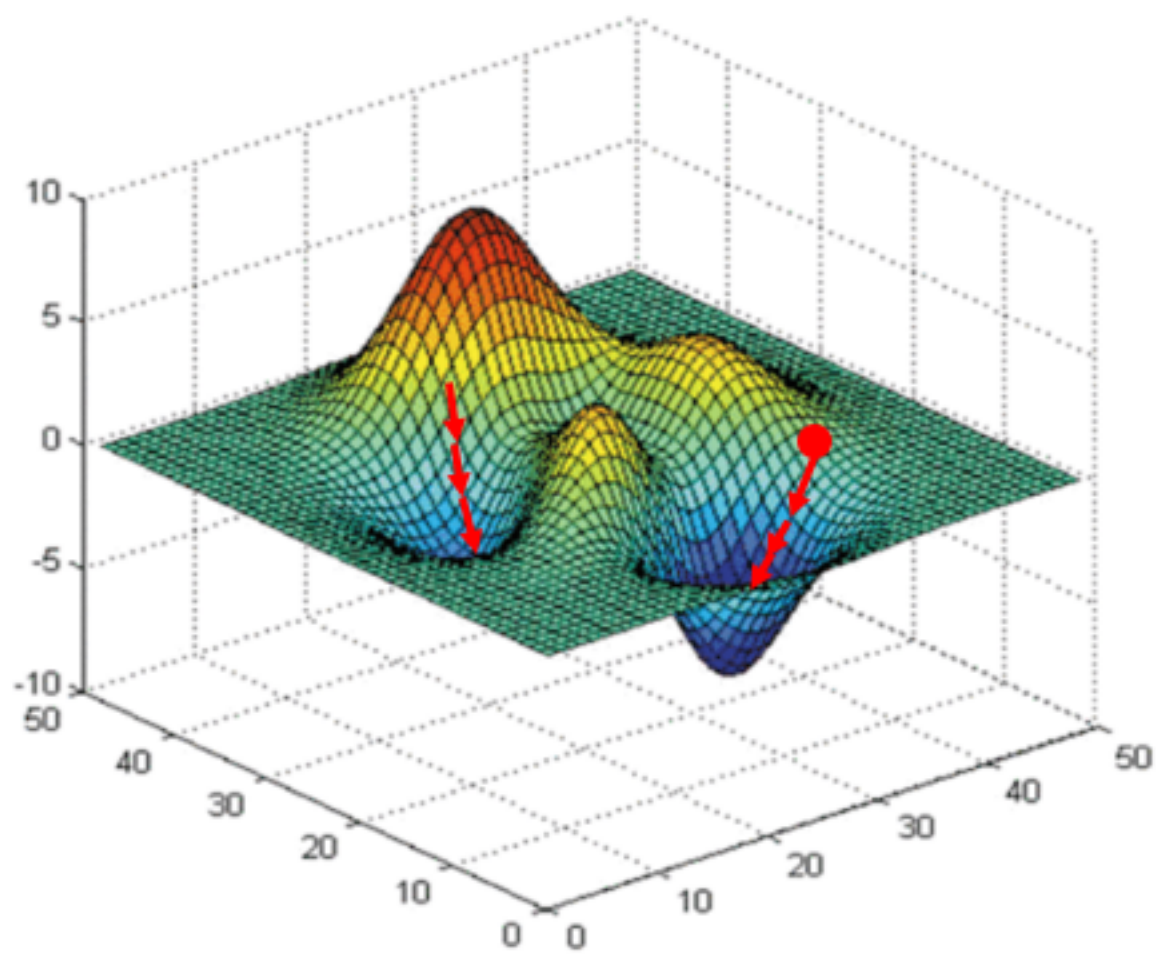
Gradient Descent



Local Minima

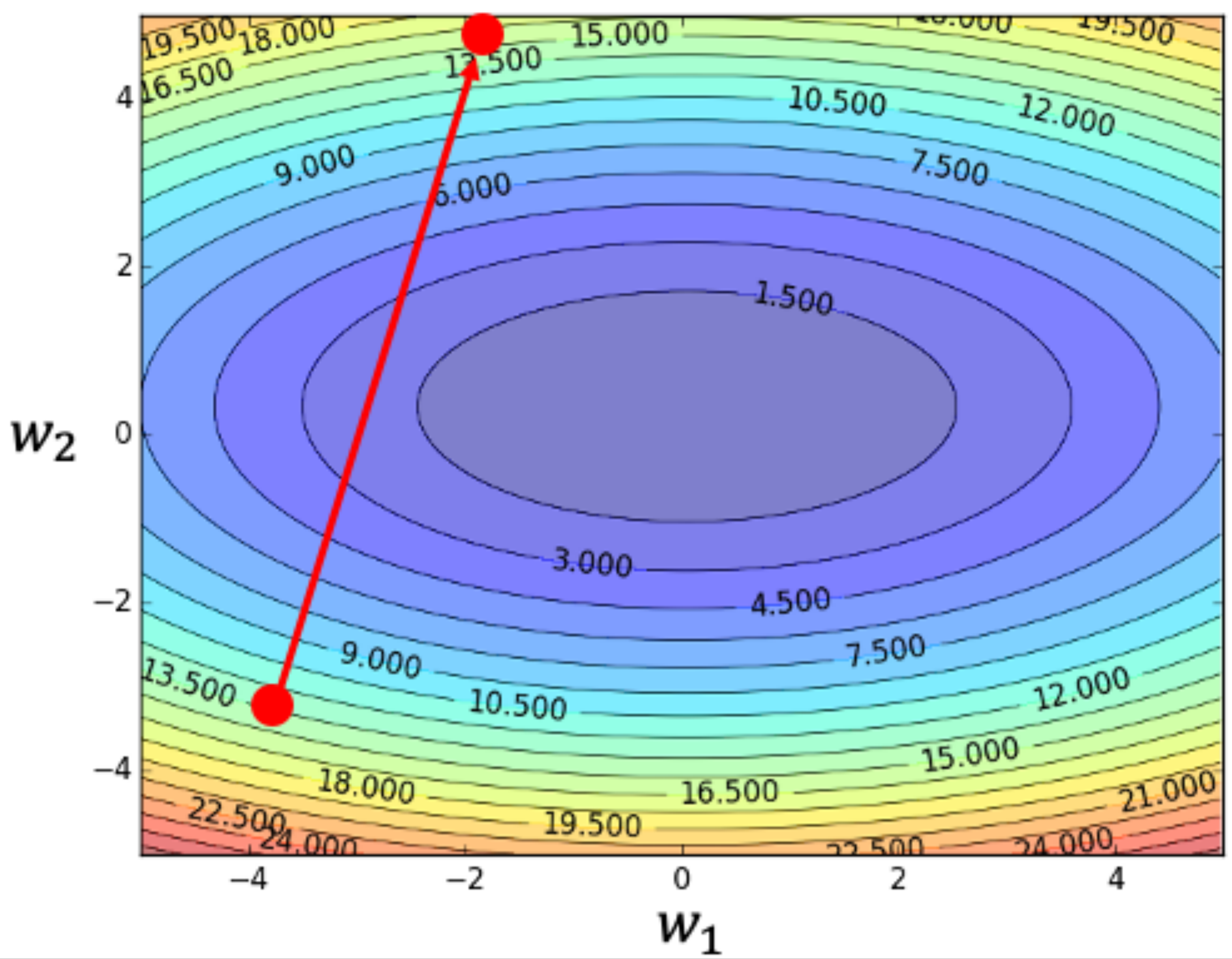


Local Minima



Different initial points reach different local minima!

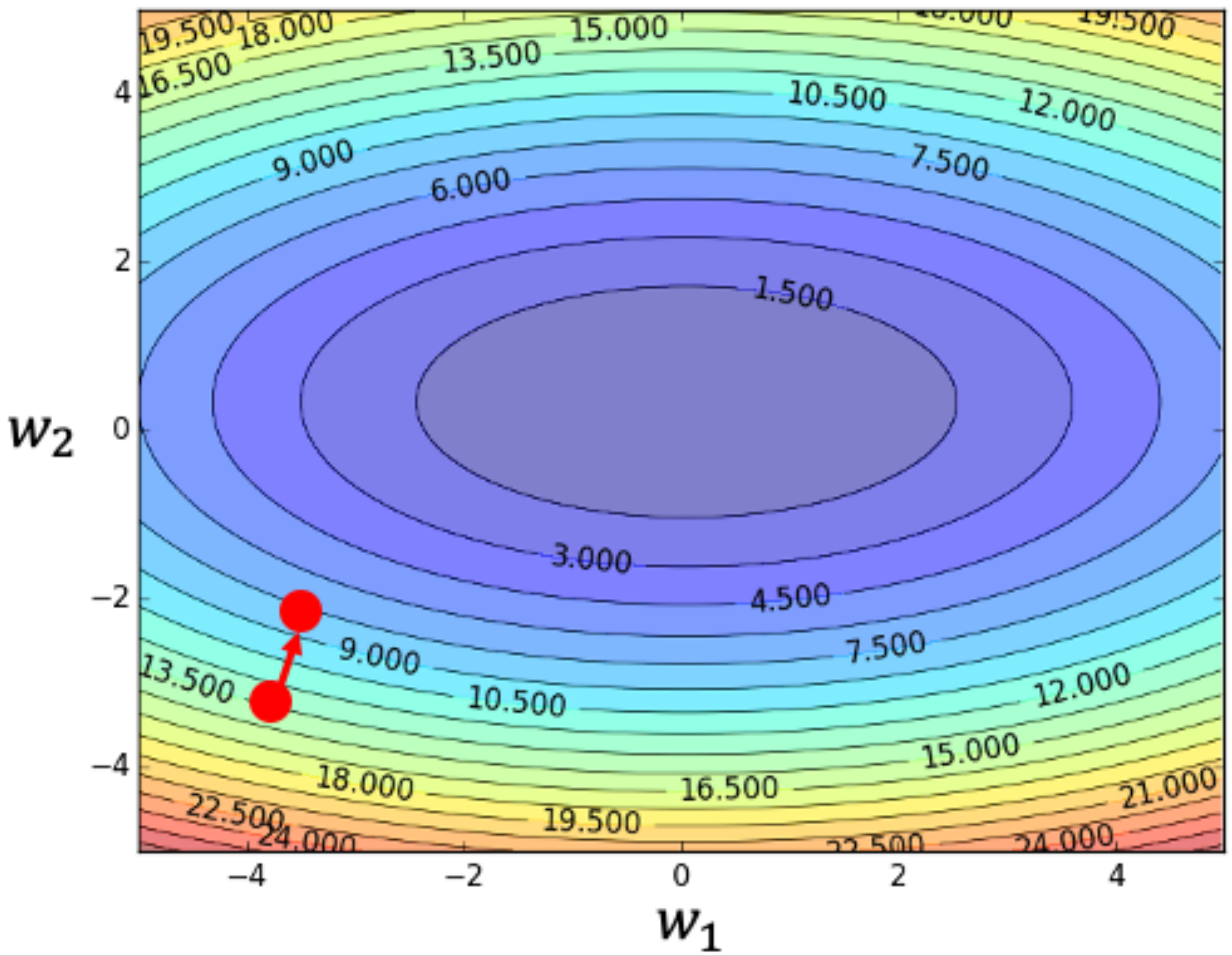
Local Minima



$w \leftarrow w - \eta \partial L / \partial w$

If learning rate is too large, total loss may not decrease

Learning Rate



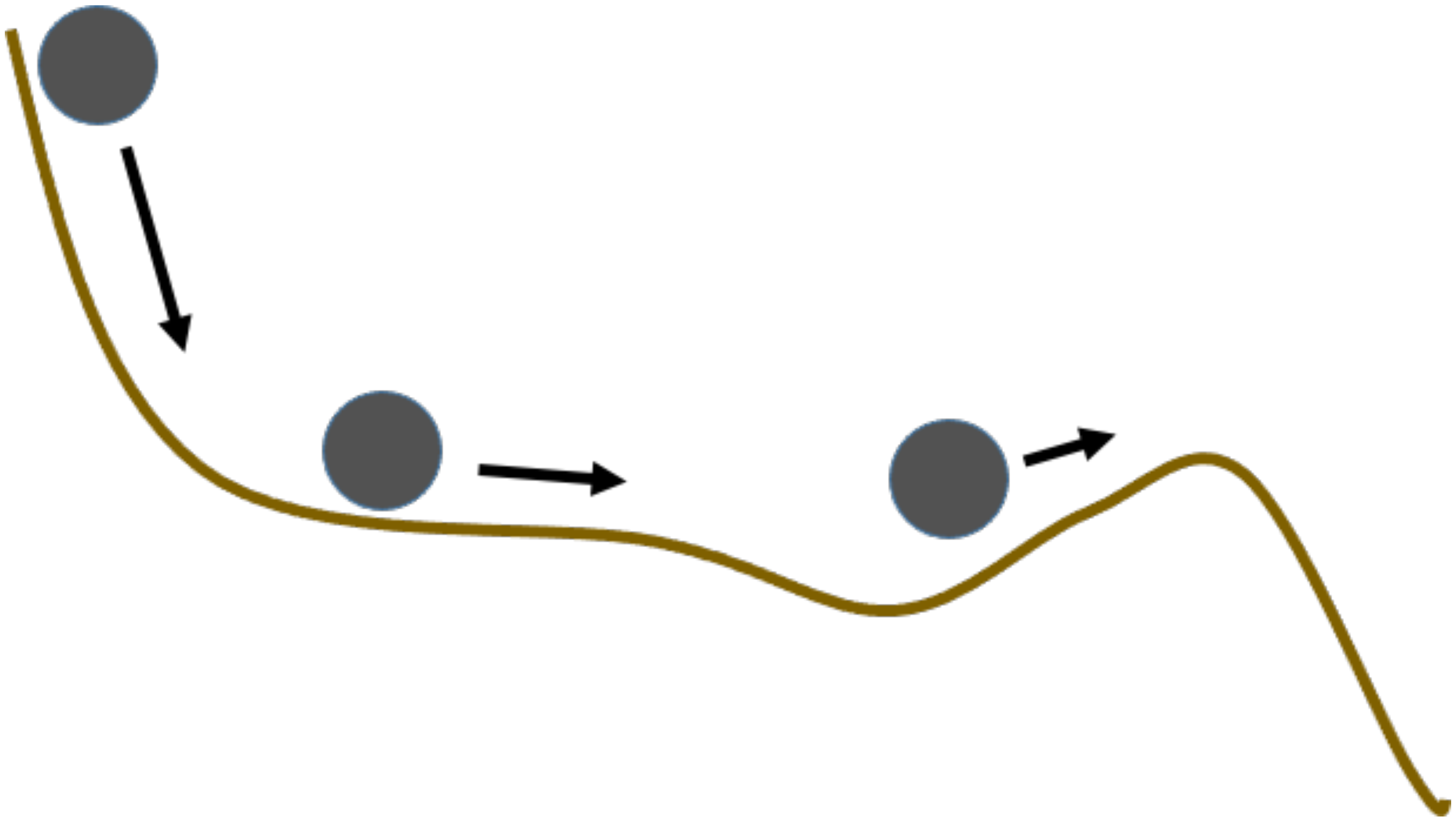
$$w \leftarrow w - \eta \partial L / \partial w$$

If learning rate is too small, training would be too slow!

Learning Rate

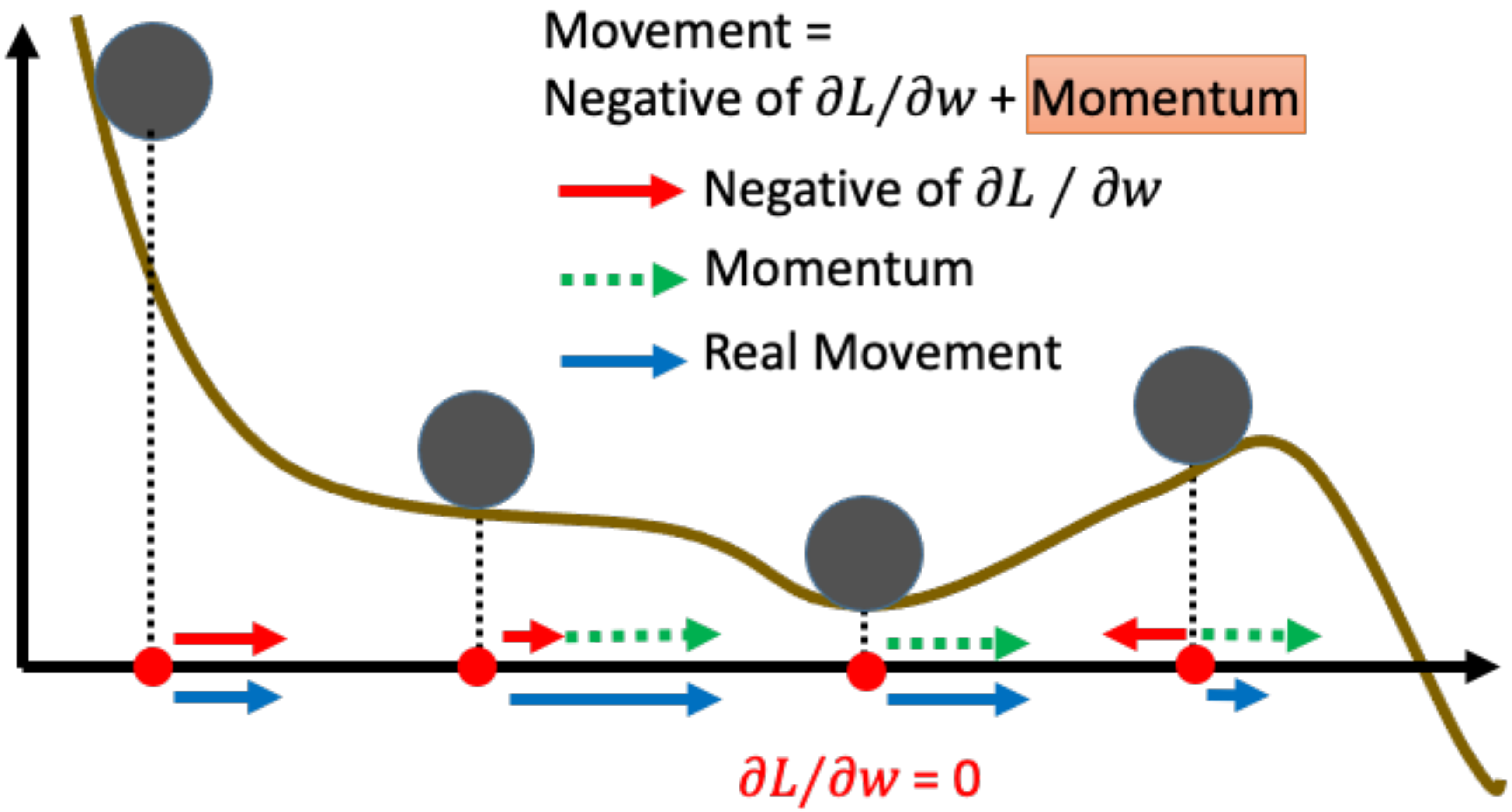
- **At the beginning, we can set a large learning rate**
- **After several epochs, we reduce the learning rate**
- **Giving different parameters different learning rate**

Momentum



How about put this phenomenon in gradient descent

Momentum

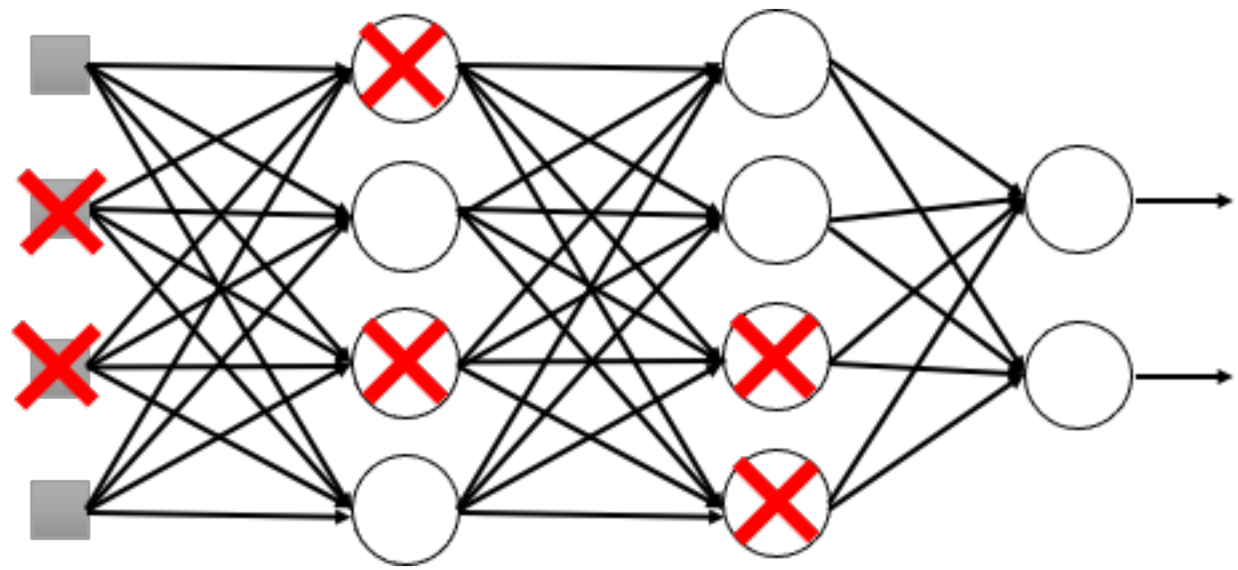



```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])
```

Dropout

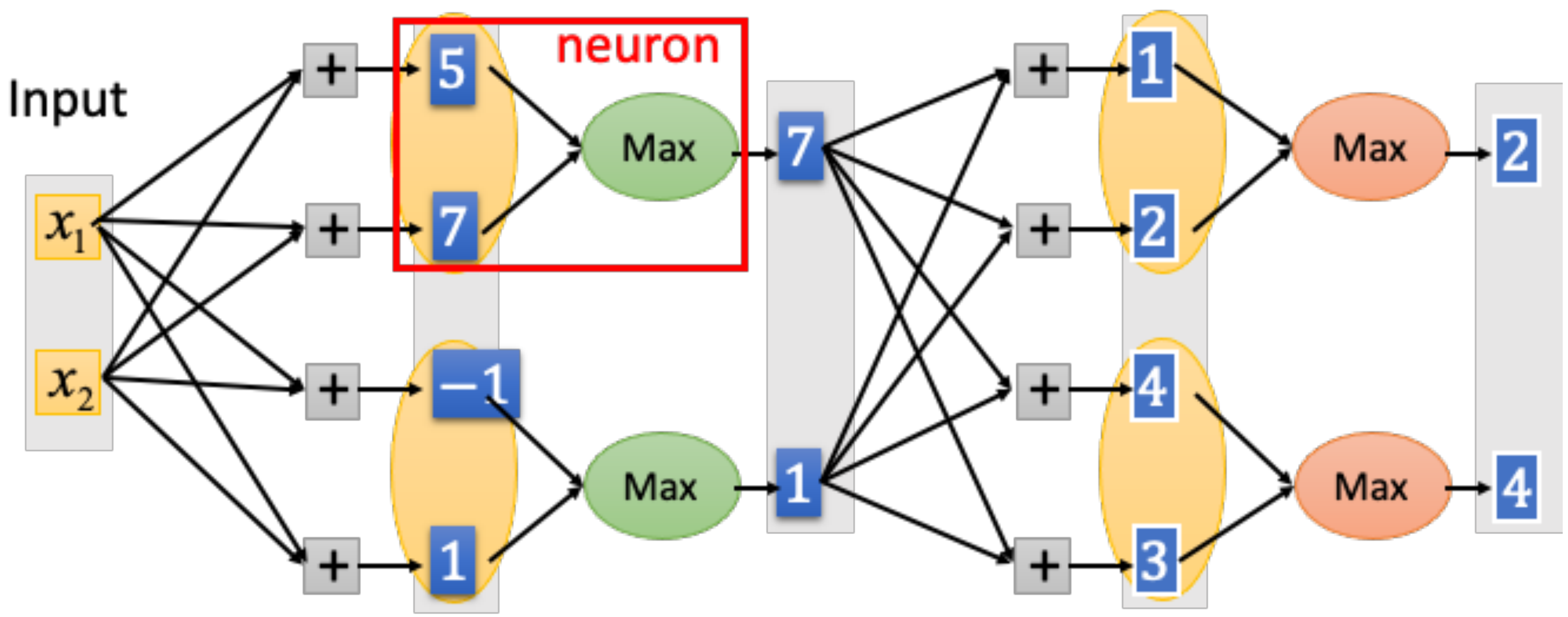
Training:



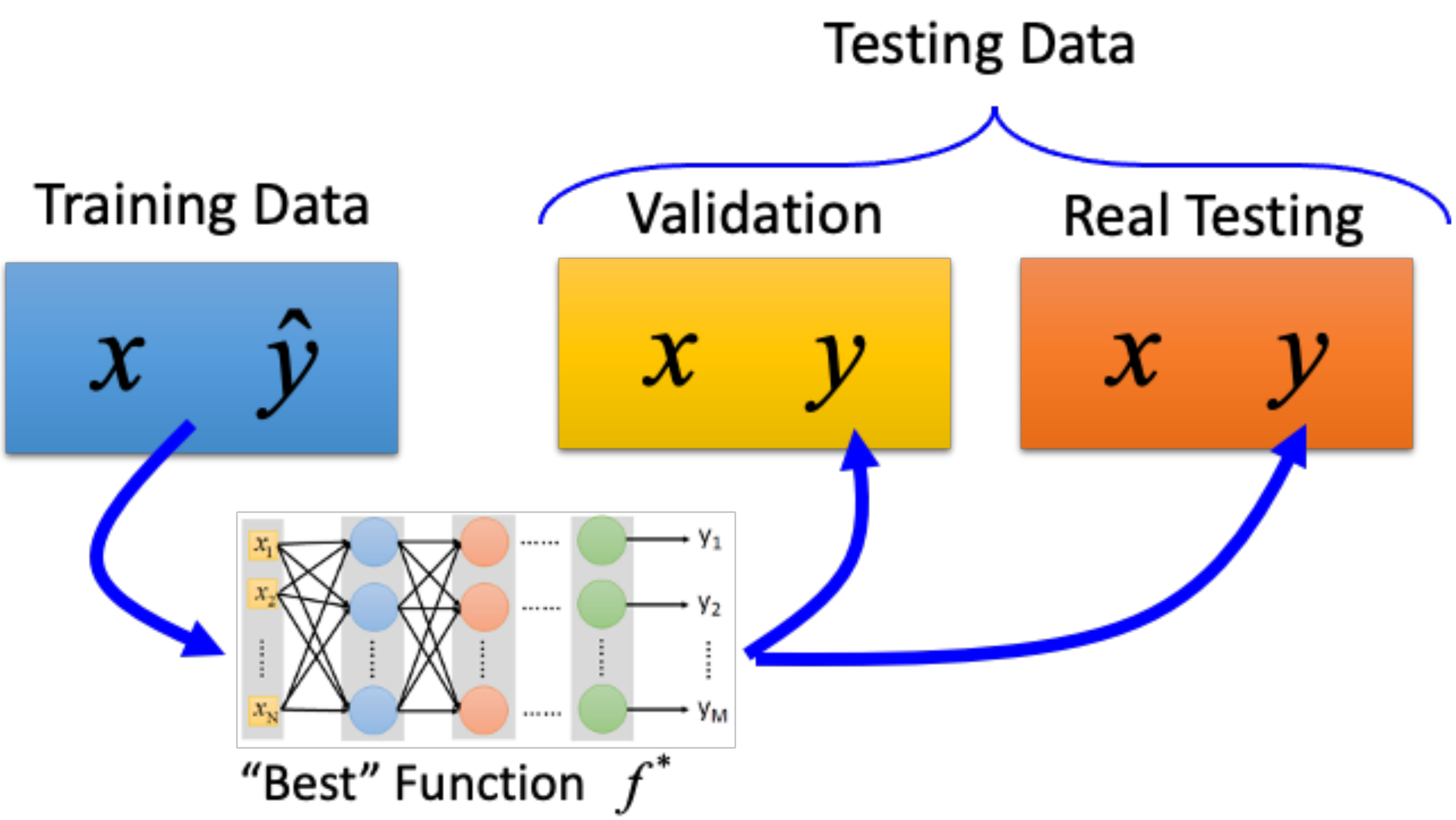
Each neuron has p% to dropout in each epoch!

```
model.add( dropout(0.8) )
```

Maxout



In Practice



Testing Data
Training Data
Validation
Real Testing
x $\hat{y}$
x y
x y
x_1
x_2
x_N
y_1
y_2
y_M
"Best" Function f^*