Machine Learning

Amin Golzari Oskouei

a.golzari@azaruniv.ac.ir

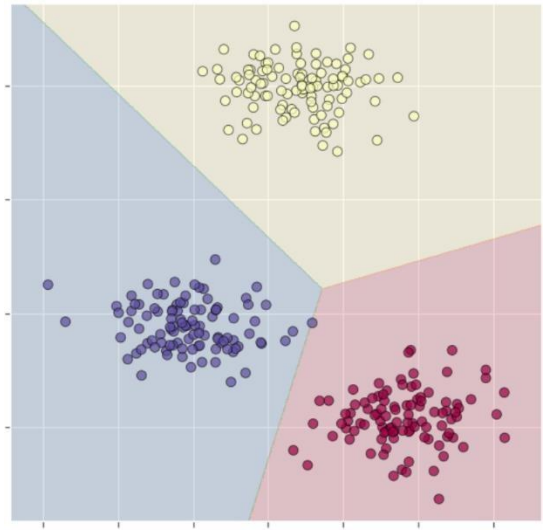
a.golzari@tabrizu.ac.ir

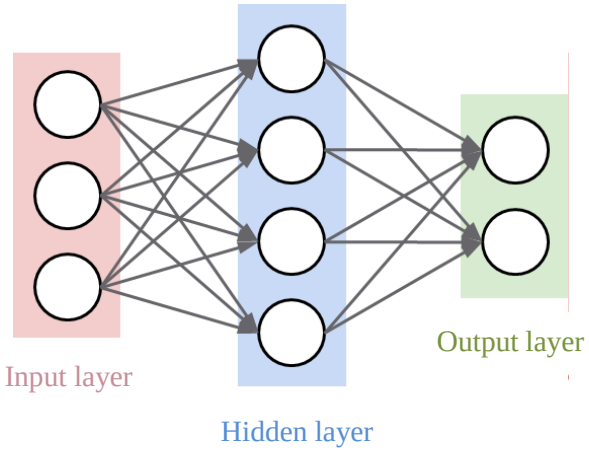
https://github.com/Amin-Golzari-Oskouei

Azarbaijan Shahid Madani University
2023

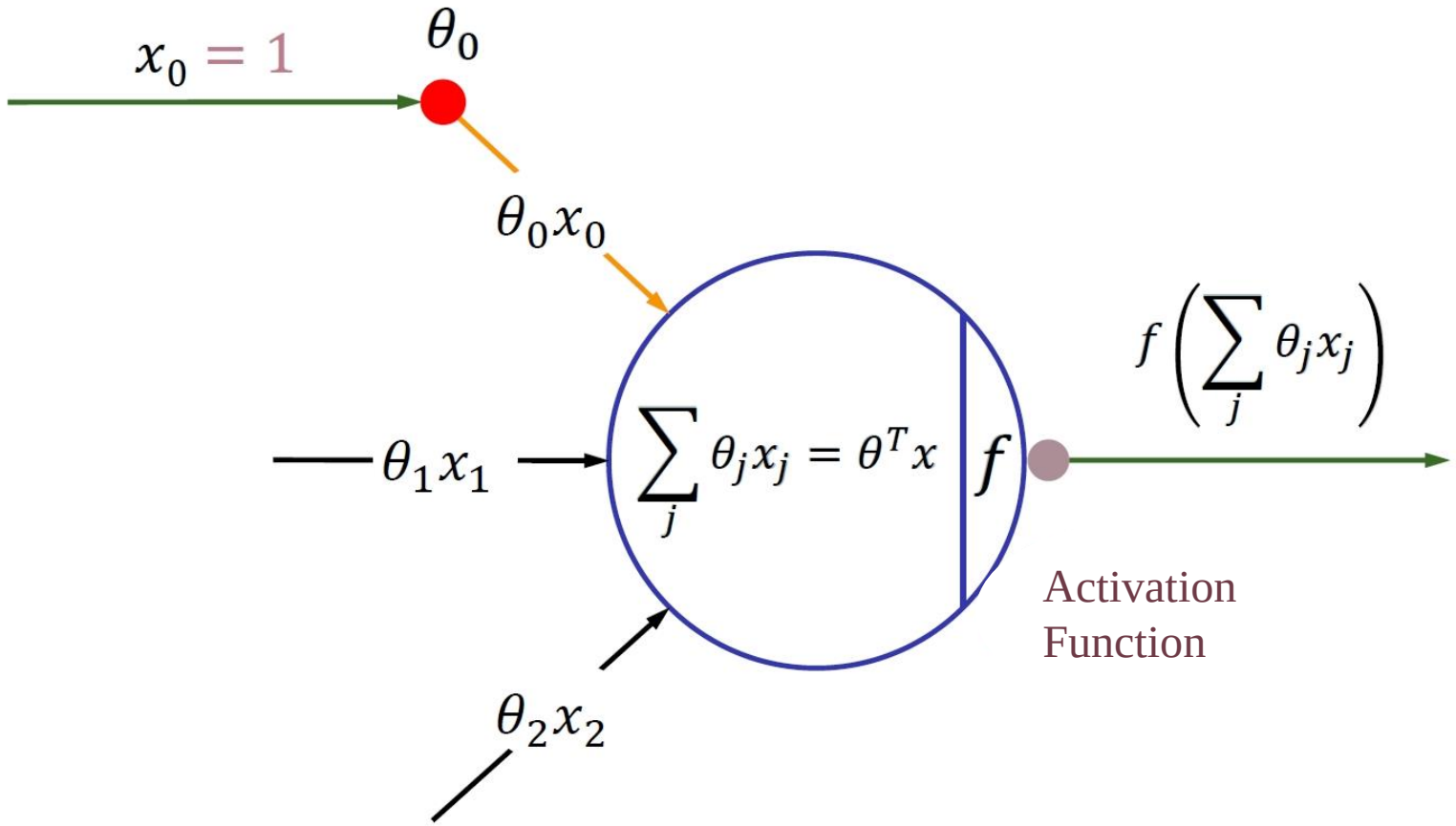
Artificial Neural Networks

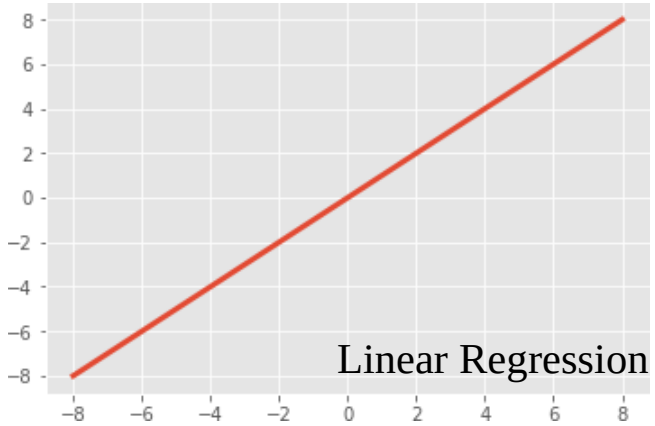
Contents

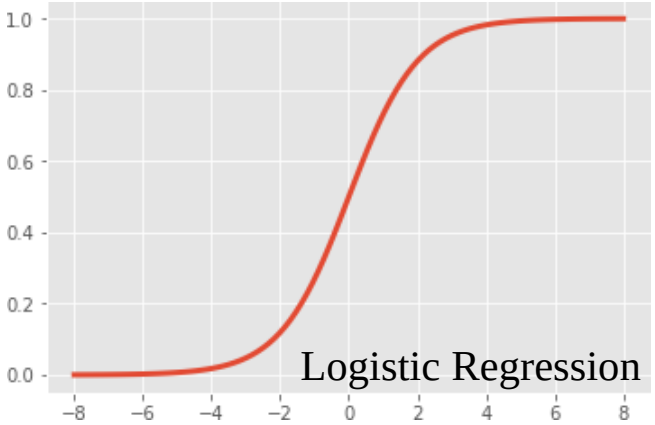




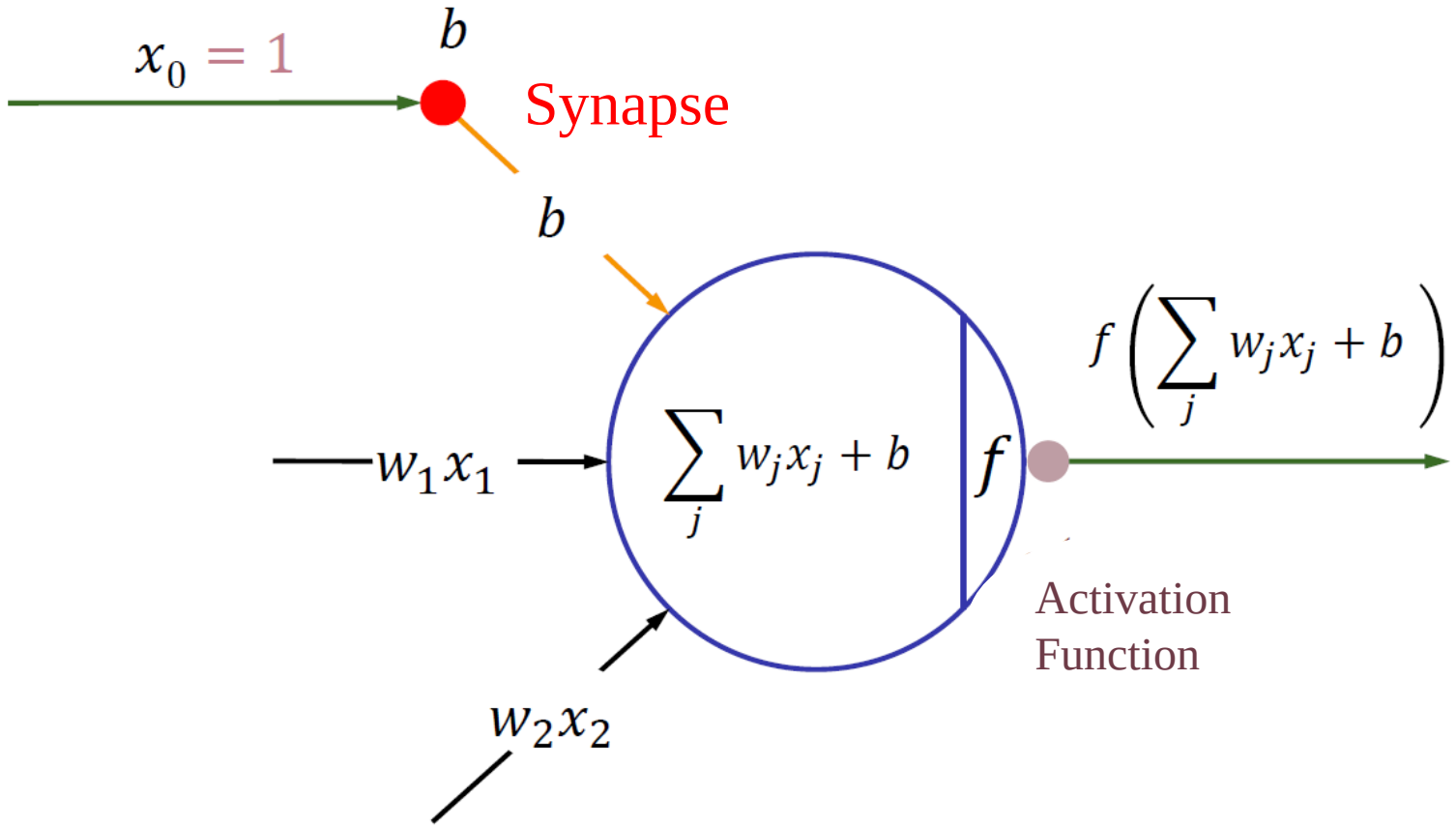
Recall: Binary Logistic Regression

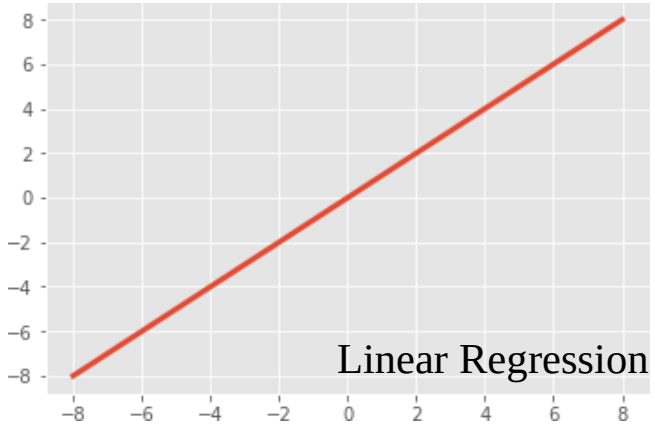


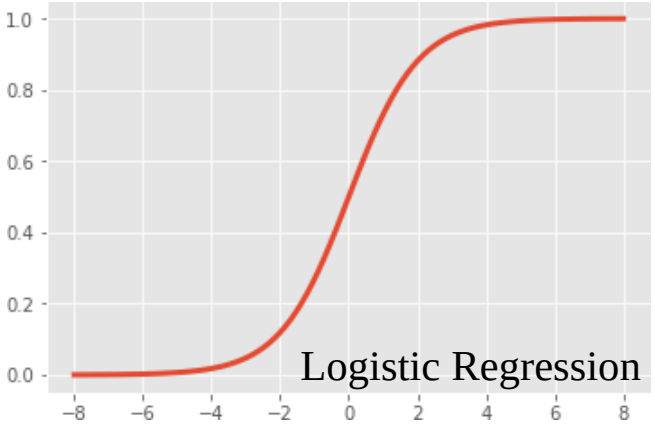




Recall: Binary Logistic Regression



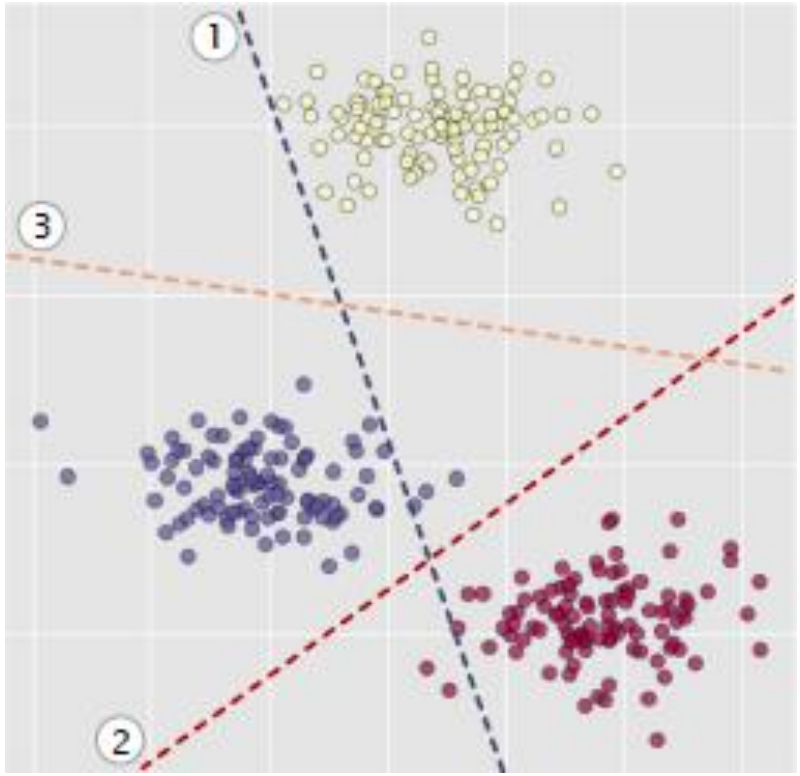




Recall: Multi-class Logistic Regression

- One against the others. Creating a classifier for each class.

$$h^{(1)}(x) = g\left(\left(\theta^{(1)}\right)^T x\right)$$
$$h^{(2)}(x) = g\left(\left(\theta^{(2)}\right)^T x\right)$$
$$h^{(3)}(x) = g\left(\left(\theta^{(3)}\right)^T x\right)$$

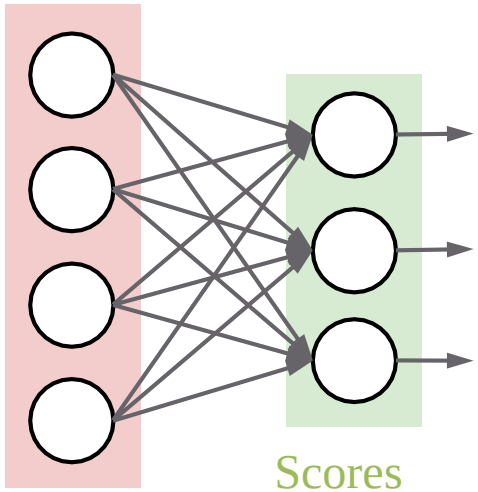


- New classification.

$$y = \underset{c}{\operatorname{argmax}}\ h^{(c)}(x)$$

Recall: Multi-class Logistic Regression

- One against the others. Creating a classifier for each class.
- New classification.
  - example. 4 features and 3 classes



Scores

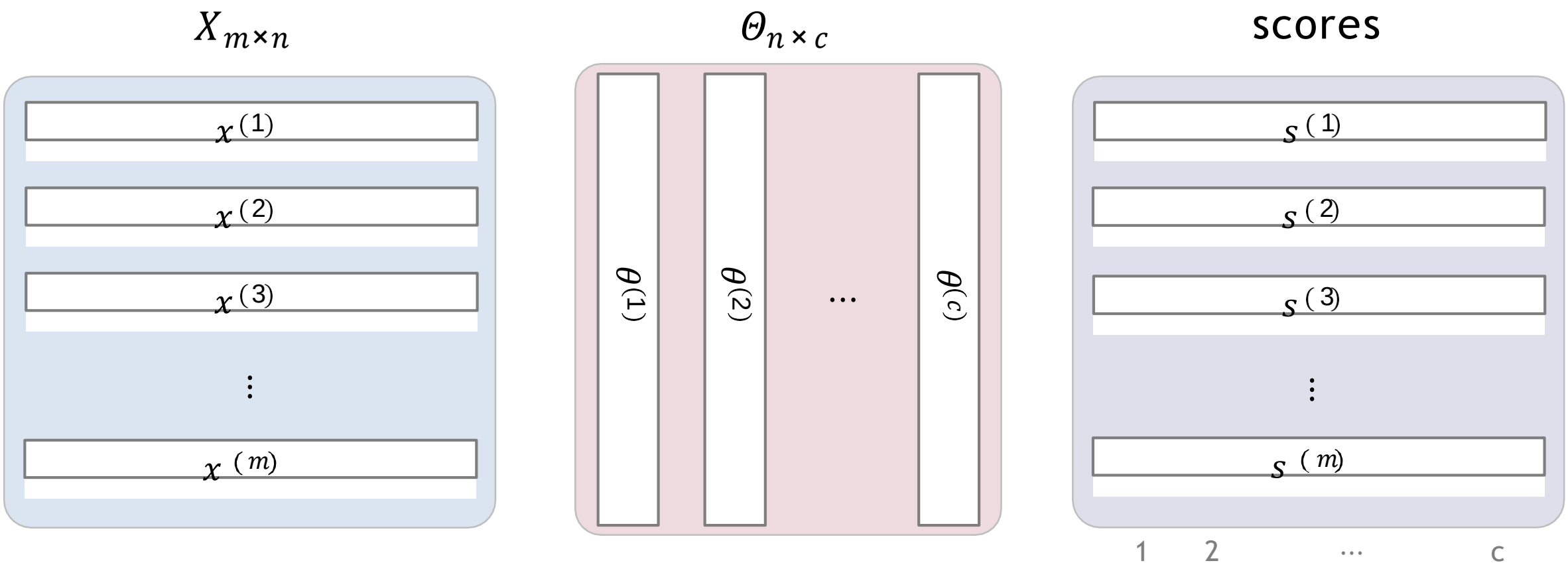
input

$$h^{(1)}(x) = g\left(\left(\theta^{(1)}\right)^T x\right)$$
$$h^{(2)}(x) = g\left(\left(\theta^{(2)}\right)^T x\right)$$
$$h^{(3)}(x) = g\left(\left(\theta^{(3)}\right)^T x\right)$$

Classification

$$y = \underset{c}{\text{argmax}}\ h^{(c)}(x)$$

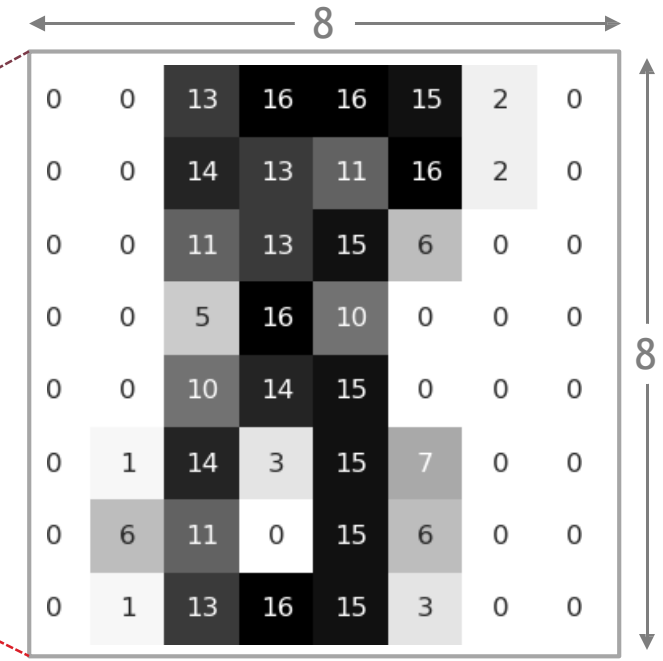
Multi-class Logistic Regression: Vectorization




```
scores = X @ Theta + theta0
```

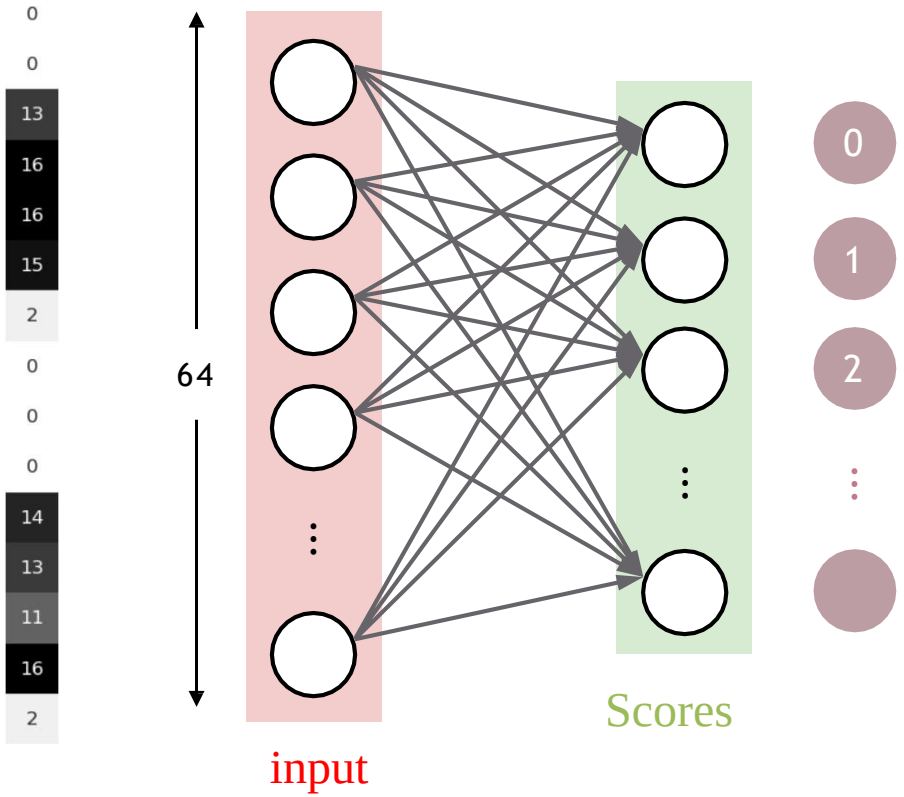
Multi-class Logistic Regression: Handwritten digits Recognition

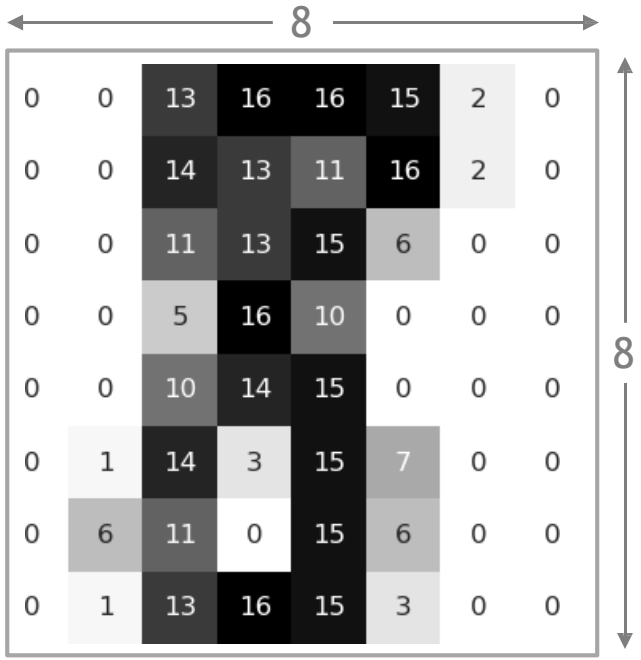
10 classes and 64 features



Multi-class Logistic Regression: Handwritten digits Recognition

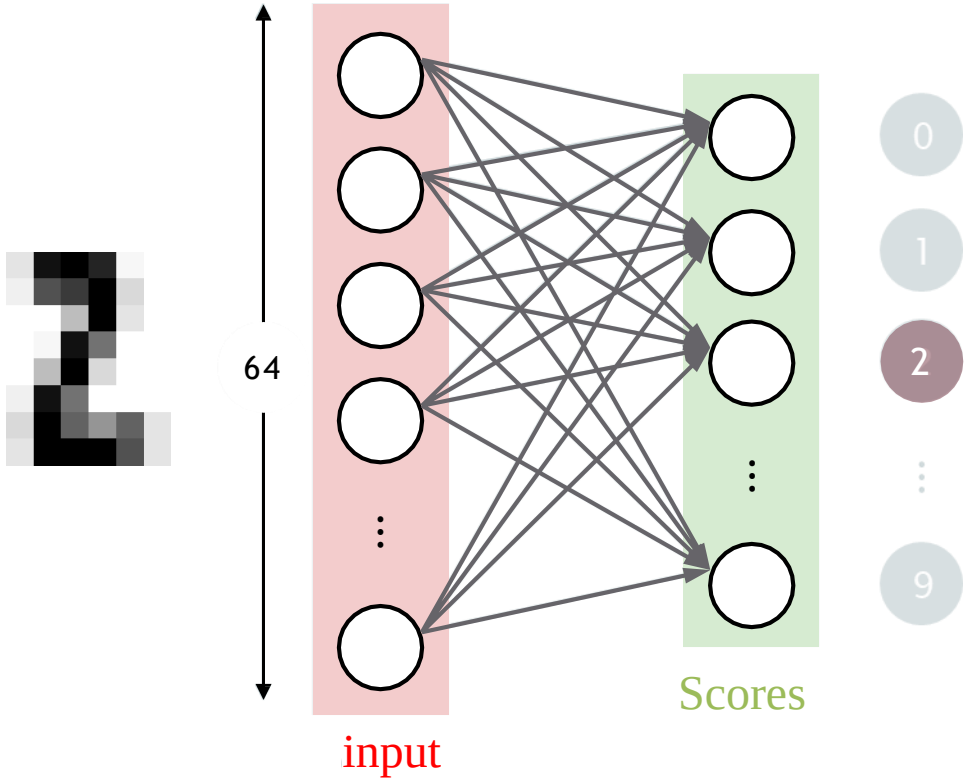
10 classes and 64 features





Multi-class Logistic Regression: Prediction

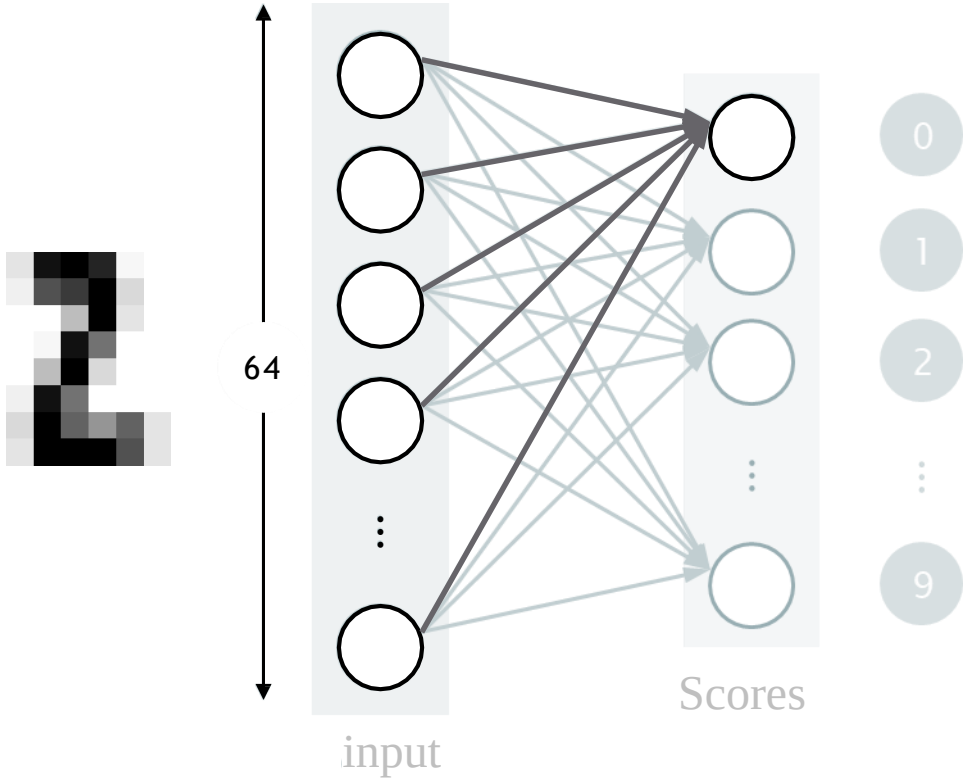
- Geometric interpretation. Calculating the similarity of the input vector with the weight vectors of each class.




```
def predict(W, b, X):  scores =
        X @ W + b
        return np.argmax(scores, axis=1)
```

Multi-class Logistic Regression: Prediction

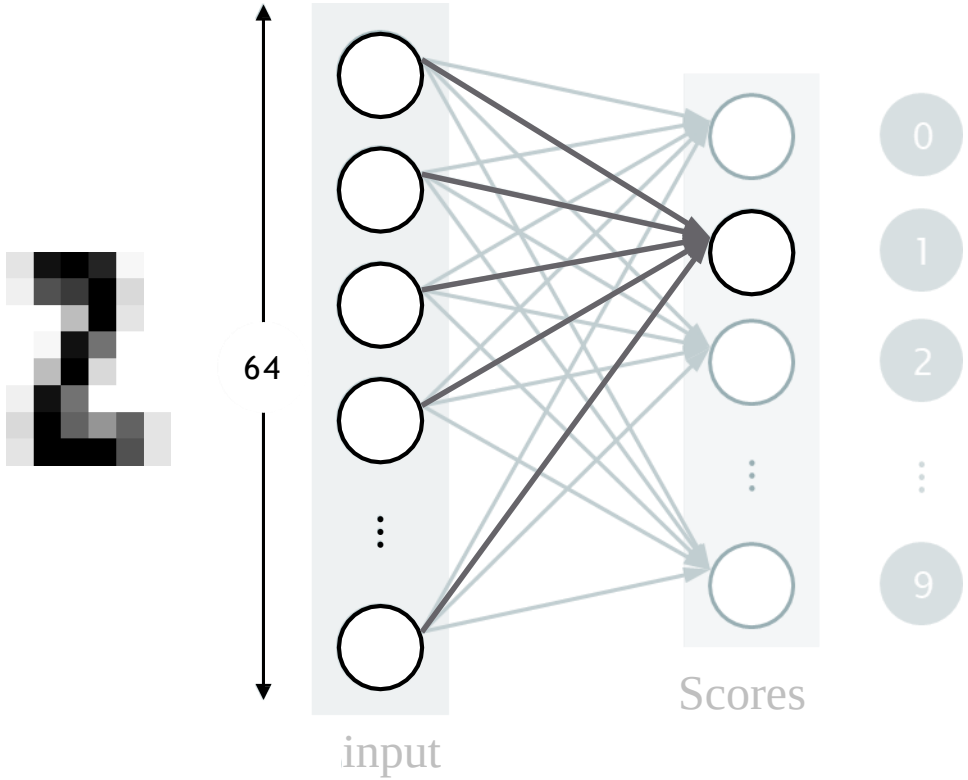
Calculation of the similarity of the input vector with parameters corresponding to the zero class




```
def predict(W, b, X):
    scores = X @ W + b
    return np.argmax(scores, axis=1)
```

Multi-class Logistic Regression: Prediction

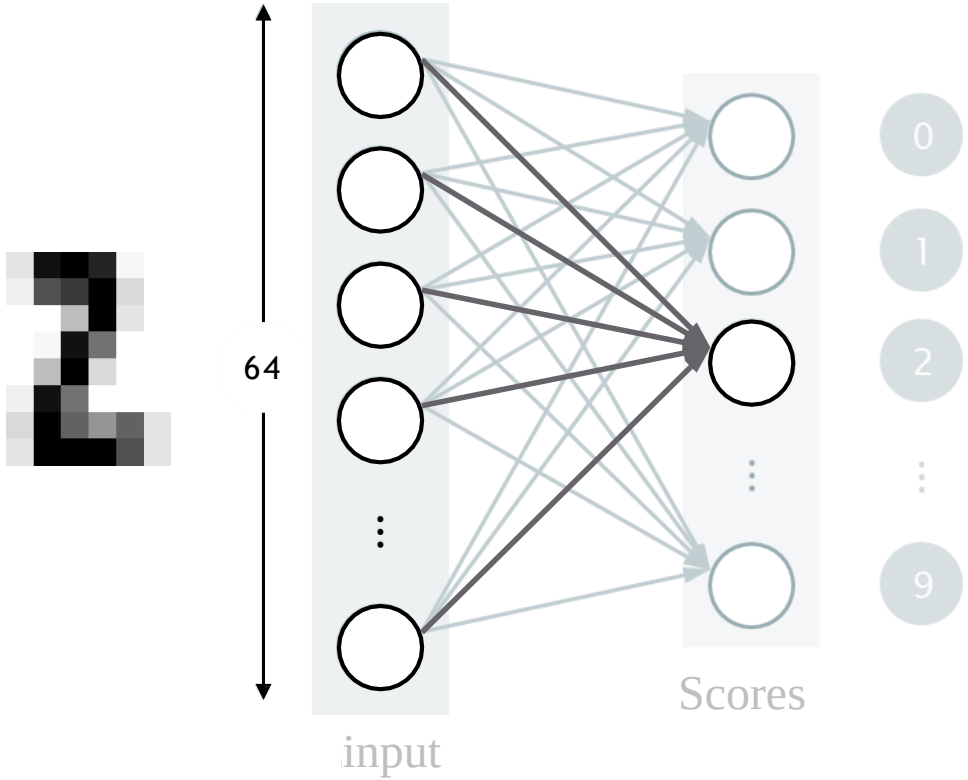
Calculation of the similarity of the input vector with parameters corresponding to the two class




```
def predict(W, b, X):
    scores = X @ W + b
    return np.argmax(scores, axis=1)
```

Multi-class Logistic Regression: Prediction

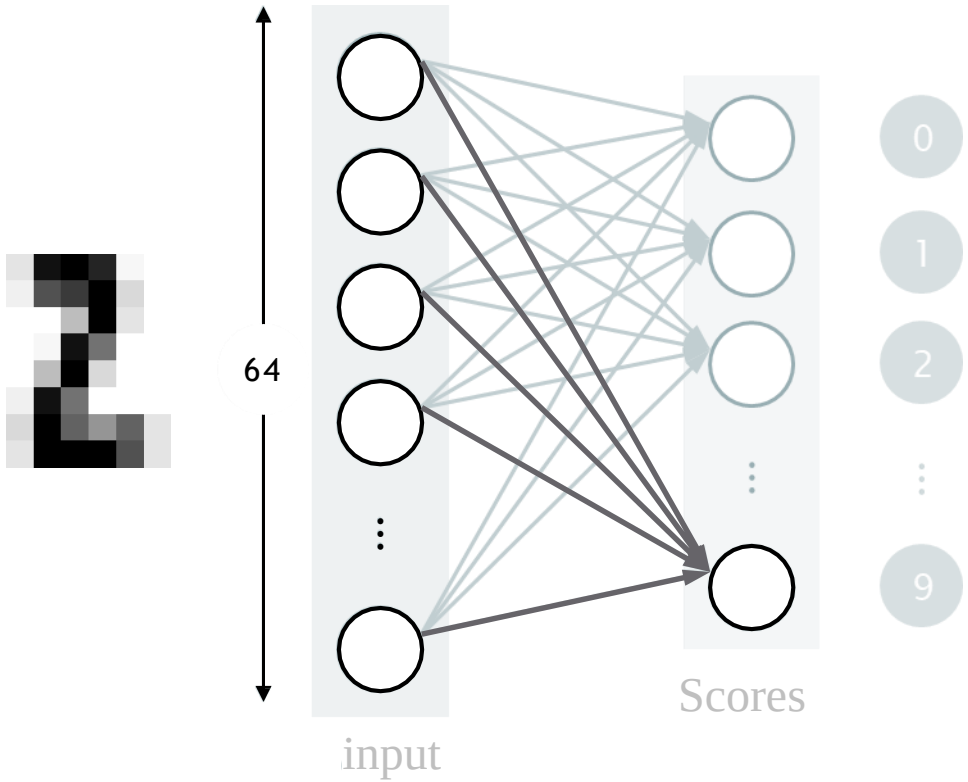
Calculation of the similarity of the input vector with parameters corresponding to the three class




```
def predict(W, b, X):
    scores = X @ W + b
    return np.argmax(scores, axis=1)
```

Multi-class Logistic Regression: Prediction

Calculation of the similarity of the input vector with parameters corresponding to nine class

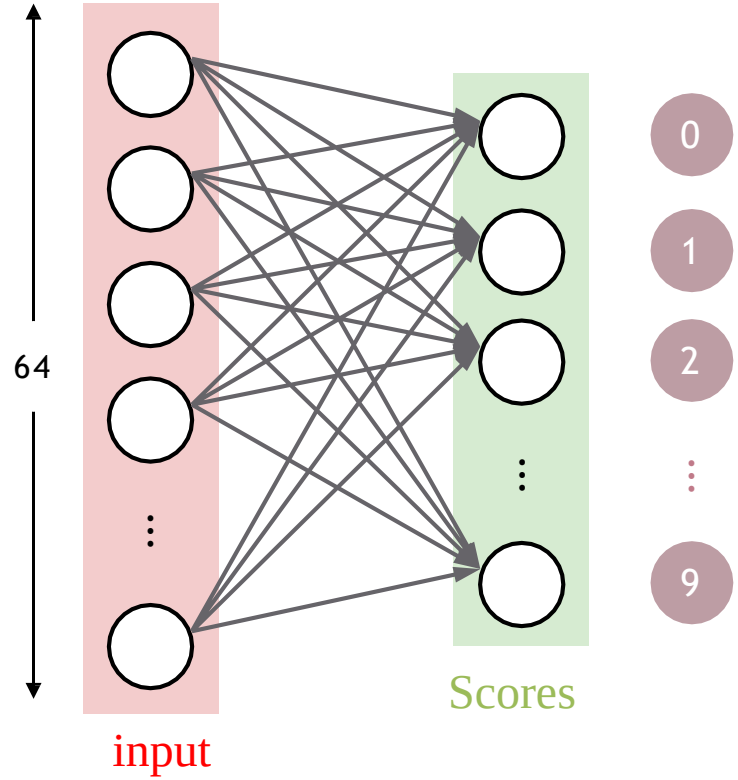



```
def predict(W, b, X):
    scores = X @ W + b
    return np.argmax(scores, axis=1)
```

SoftMax Classifier

Binary Logistic Regression: Sigmoid function

☐ Binary classification. The value of the hypothesis function indicates the probability of the input data belonging to category 1.

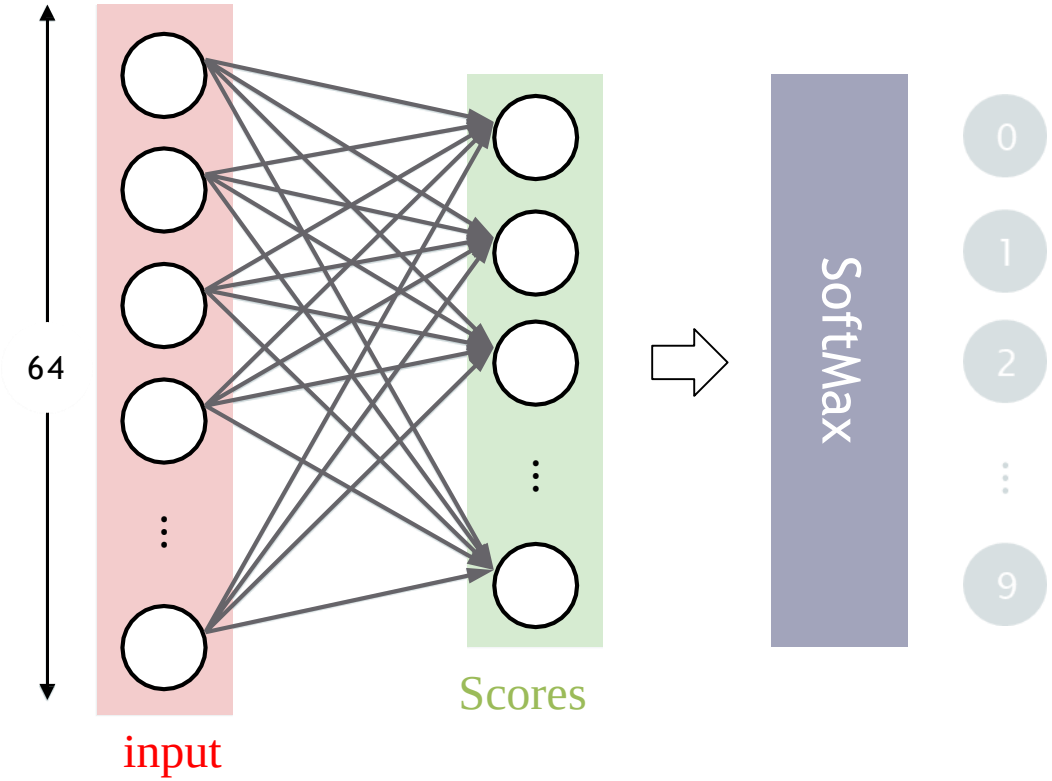


$$p = \text{sigmoid}(X @ W + b)$$

Attention. If using the sigmoid (logistic) function in a problem for multi-category classification, the sum of the hypothetical values will not necessarily be equal to one.

Binary Logistic Regression: SoftMax function

SoftMax function. In multi-category classification, in order to calculate the probability of the input vector belonging to each of the different categories, we use the SoftMax function instead of the sigmoid function.



$$P = SoftMax(X @ W + b)$$

$$\text{SoftMax}\left(s^{(i)}\right)_k = \frac{e^{sk}}{\Sigma_{j=1}^{c} e^{sj}} = p_k^{(i)}$$

Probability of input data xi belonging to class $y=k$

SoftMax Classifier (Multi-class Logistic Regression)

- Idea. Convert the score vector to a probability distribution vector!

$(x^{(i)}, y^{(i)})$

$$L_i = -\log\left(\frac{e^s y^{(i)}}{\Sigma_j e^{s_j}}\right)$$

$L_i = -\log(0.13) = 0.89$

	Scores	Power	Possibilities(not normalized)	normalization	Possibilities
Cat	**3.2**	→	**24.5**	→	**0.13**
Car	5.1	→	164.0	→	0.87
frog	-1.7	→	0.2	→	0.0

What is the minimum and maximum possible value for the cost function?

SoftMax Classifier (Multi-class Logistic Regression)

- Points. The logarithm of the probability of categories is not normalized!

$(x^{(i)}, y^{(i)})$

Cat	**3.2**
Car	5.1
frog	-1.7

$$P(Y = k|X = x^{(i)}) = \frac{e^{s_k}}{\sum_j e^{s_j}} \qquad s = f(x^{(i)}, W)$$

SoftMax function

- Goal
  - Maximization of the logarithm of the right exponent (or minimization of the negative logarithm of the right exponent)!

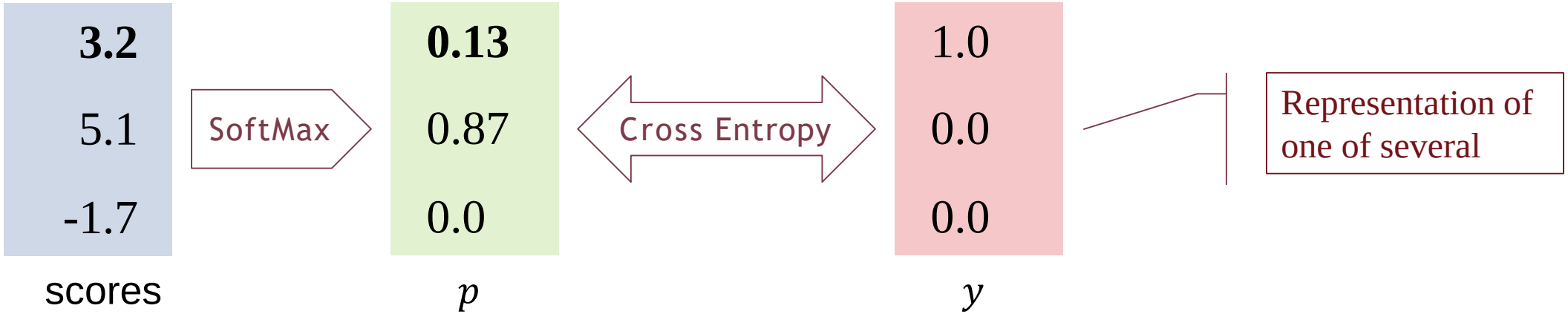
$$L_i = -\log P(Y = y^{(i)} | X = x^{(i)}) = -\log(\frac{e^{s_{y^{(i)}}}}{\sum_j e^{s_j}})$$

SoftMax Cost Function: implementation


```
def SoftMax_loss(scores, y):

        # SoftMax loss implementation.
        # y is encoded as a one-hot vector.

        p = SoftMax(scores)
        return -np.sum(y * np.log(p))
```

$(x^{(i)}, y^{(i)})$

$$L_i = \sum_{k=1}^{c} -y_k log\ p_k = -log\ p_{y^{(i)}}$$



SoftMax Classifier: Train

- Train set.

$$\{(x^{(i)}, y^{(i)})\}_{i=1}^{m} \qquad x^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & \dots & x_n^{(i)} \end{bmatrix}^T \qquad y^{(i)} \epsilon \{1,2,\dots,c\}$$

- Parameters.

$$W \epsilon \mathbb{R}^{n \times c} \qquad b \epsilon \mathbb{R}^{c}$$

- Cost Function.

$$L(W,b) = \frac{1}{m}\sum_{i=1}^{m}{}^{m} L_i + \lambda R(W) = \frac{1}{m}\sum_{i=1}^{m}{}^{m} \left(-log\, p_{y^{(i)}}\right) + \lambda \left|\left|W\right|\right|_2^2$$

SoftMax Classifier: Train

- cost Function.

$$L(W,b) = \frac{1}{m}\sum_{i=1}^{m}\left(-\log p_{y^{(i)}}\right) + \lambda||W||_2^2$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(-\log\left(\frac{e^{s_{y^{(i)}}}}{\sum_{j=1}^{c} e^{s_j}}\right)\right) + \lambda||W||_2^2$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(-\log\left(\frac{e^{\left((w^{y^{(i)}})^T x^{(i)} + b^{y^{(i)}}\right)}}{\sum_{i=1}^{c} e^{\left((W^{(j)})^T x^{(i)} + b^{(j)}\right)}}\right)\right) + \lambda||W||_2^2$$

SoftMax Classifier: Train

$$L_i = -\log p_{y^{(i)}} \quad p_k = \frac{e^{s_k}}{\sum_{j=1}^{c} e^{s_j}} \quad s_k = \left(W^k\right)^T x^{(i)} + b^{(k)}$$

$$\frac{\partial L_i}{\partial w^{(k)}} = ? \quad k \epsilon \{1,2,\dots,c\}$$

$$\frac{\partial L_i}{\partial w^{(k)}} = \frac{\partial L_i}{\partial p_{y^{(i)}}} \cdot \frac{\partial p_{y^{(i)}}}{\partial s_k} \cdot \frac{\partial s_k}{\partial w^{(k)}}$$

$$= \left(-\frac{1}{p_{y^{(i)}}}\right) p_k(1 - p_k)\, x^{(i)}$$

$$= (p_k - 1)x^{(i)}$$

K = y(i)

$$\frac{\partial L_i}{\partial w^{(k)}} = \frac{\partial L_i}{\partial p_{y^{(i)}}} \cdot \frac{\partial p_{y^{(i)}}}{\partial s_k} \cdot \frac{\partial s_k}{\partial w^{(k)}}$$

$$= \left(-\frac{1}{p_{y^{(i)}}}\right) p_{y^{(i)}}(-p_k)\, x^{(i)}$$

$$= (p_k)x^{(i)}$$

K ≠ y(i)

SoftMax Loss function: Train

$$L_i = -\log p_{y^{(i)}} = -log\left(\frac{e^{s_y{}^{(i)}}}{\sum_{j=1}^{c} e^{s_j}}\right)$$

- Random gradient descent algorithm.

$$\frac{\partial L_i}{\partial w^{(k)}} = \left(p_{y_k} - 1\right)x^{(i)} \longrightarrow w^{(k)} = w^{(k)} - \alpha\left(p_{y_k} - 1\right)x^{(i)} \qquad (k = y^{(i)})$$

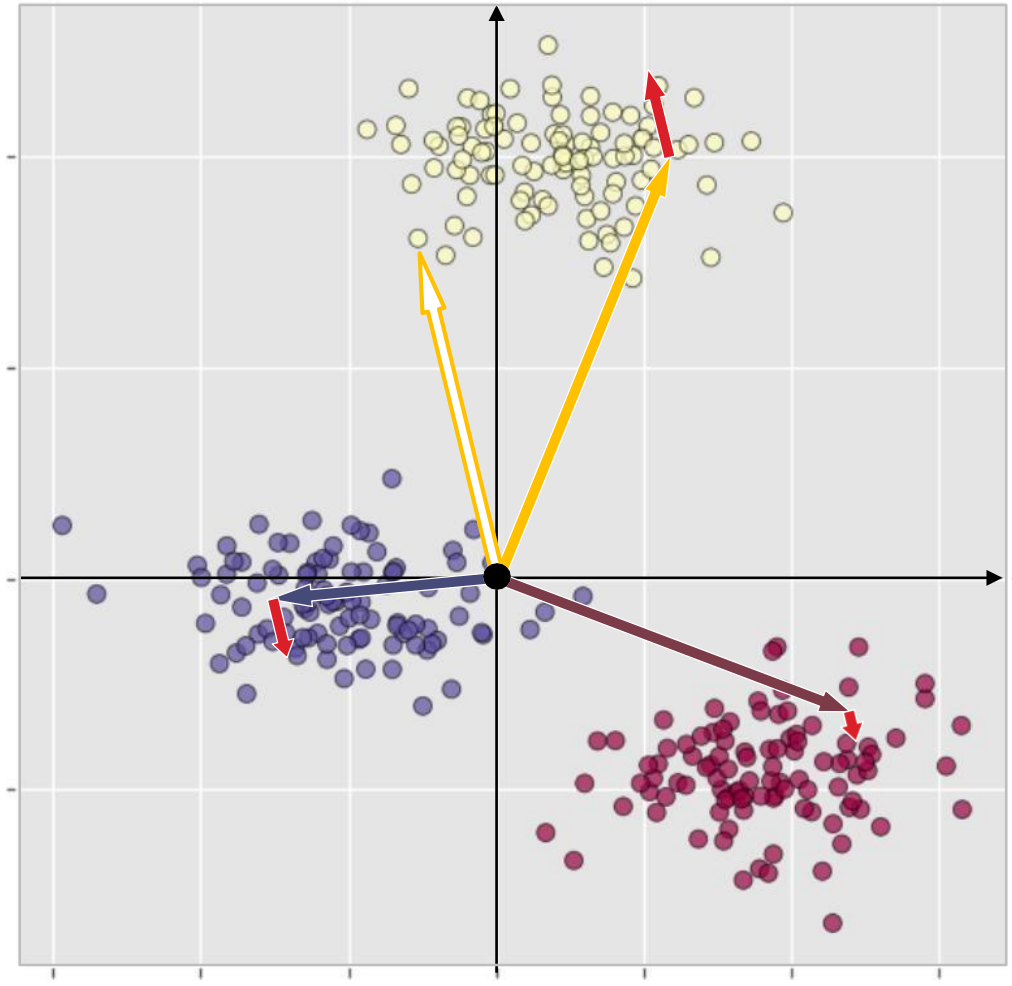
$$\frac{\partial L_i}{\partial w^{(k)}} = \left(p_{y_k}\right)x^{(i)} \longrightarrow w^{(k)} = w^{(k)} - \alpha\left(p_{y_k}\right)x^{(i)} \qquad (k \neq y^{(i)})$$

SoftMax Loss function: Train

- Geometric interpretation.

$$w^{(k)} = w^{(k)} - \alpha\left(p_{y_k} - 1\right)x^{(i)} \qquad (k = y^{(i)})$$

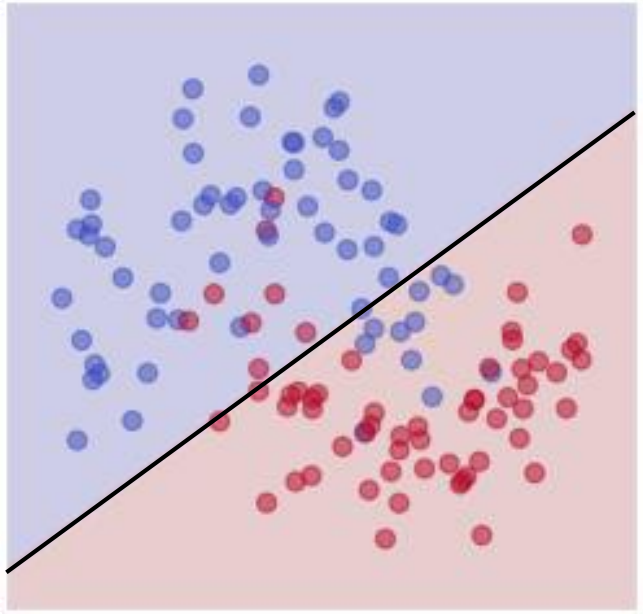
$$w^{(k)} = w^{(k)} - \alpha\left(p_{y_k}\right)x^{(i)} \qquad (k \neq y^{(i)})$$



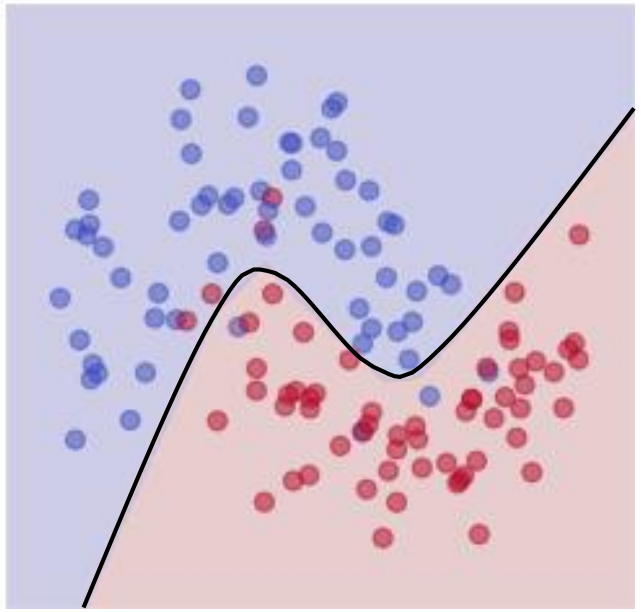
Neural Networks

Neural Networks: Motivation

- Logistic regression is a linear classification method.
  - If the data is not linearly separable, we need to add higher order features.



Linear decision boundary

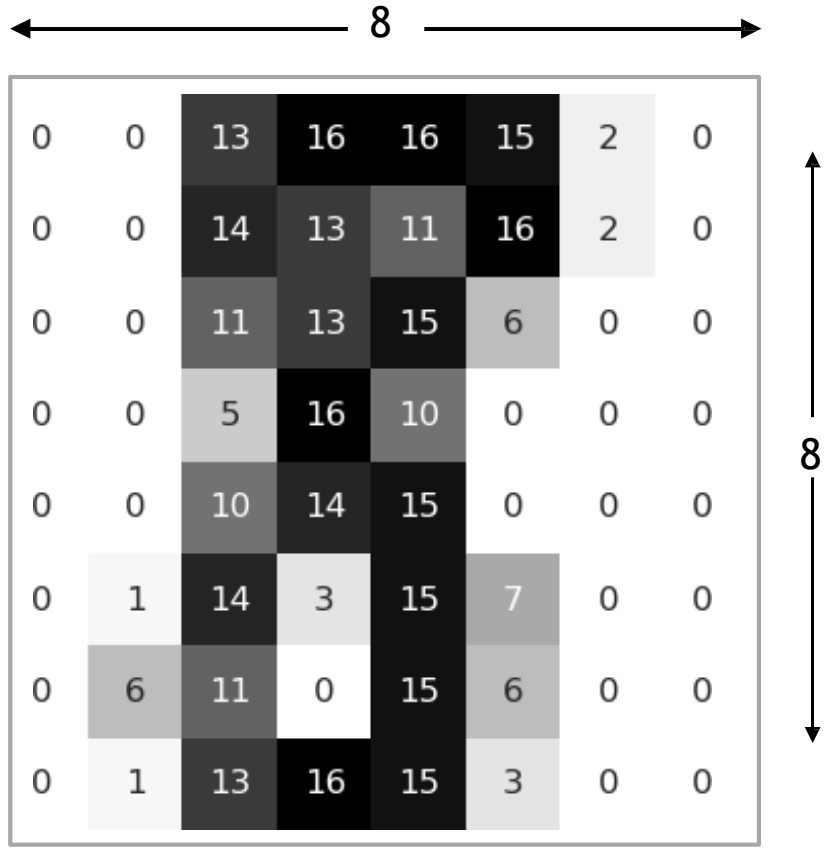


Nonlinear decision boundary

Neural Networks: Motivation

- Logistic regression is a linear classification method.
  - If the data is not linearly separable, we need to add higher order features.

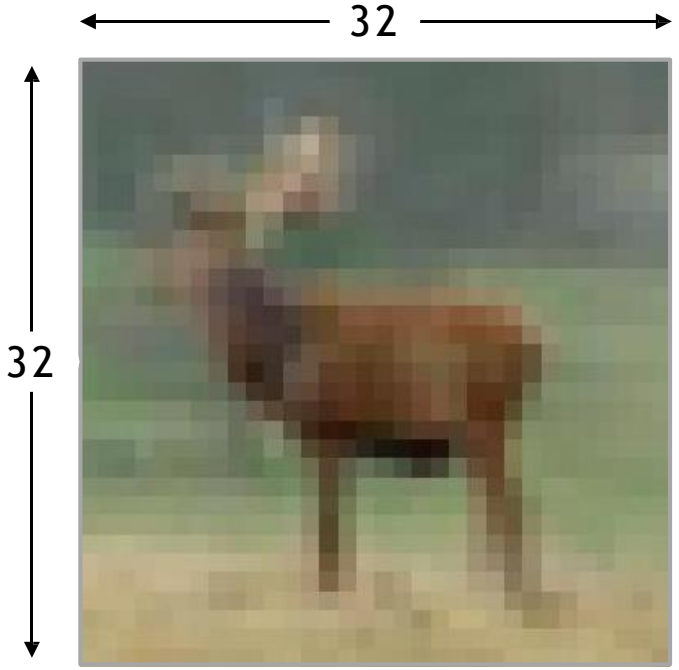
- Example. Recognizing handwritten digits [images 8 by 8]
  - Number of second-order features: more than 2,000
  - Number of third-order features: more than 40,000



Neural Networks: Motivation

- Logistic regression is a linear classification method.
  - If the data is not linearly separable, we need to add higher order features.

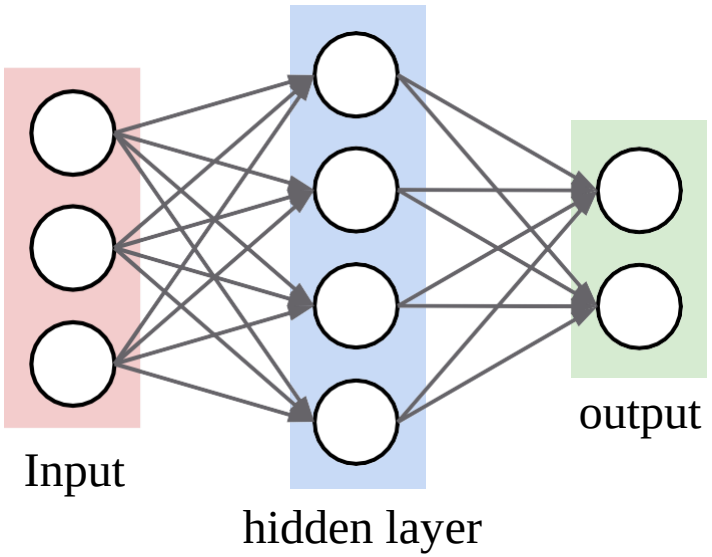
- Example. Color images [32 x 32 x 3]
  - Number of second-order features: more than 470,000
  - Number of third-order features: more than 5,000,000
  - ...



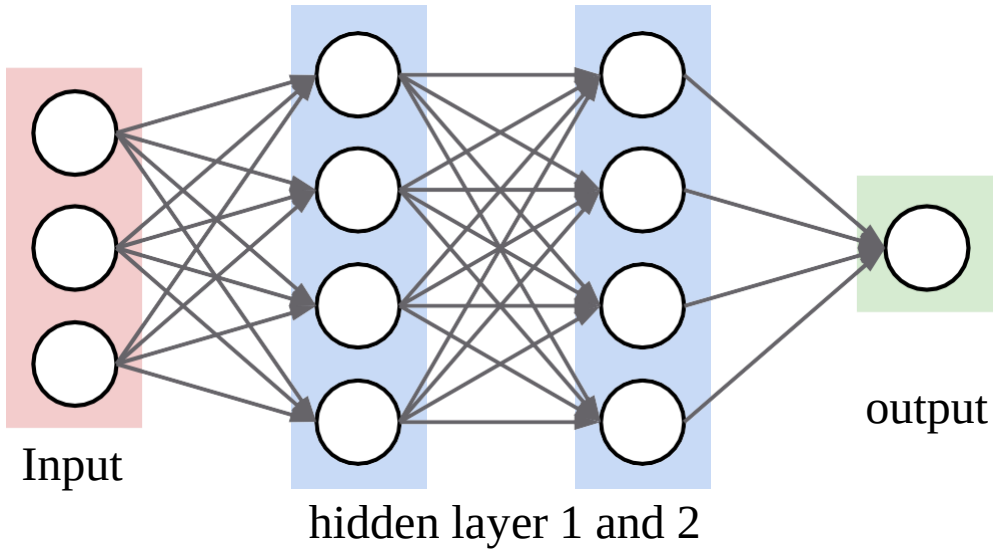
We can remove many features using setting!!!

Neural Networks: Learning New Features

- Neural networks can learn the high-level features they need by combining low-level features.



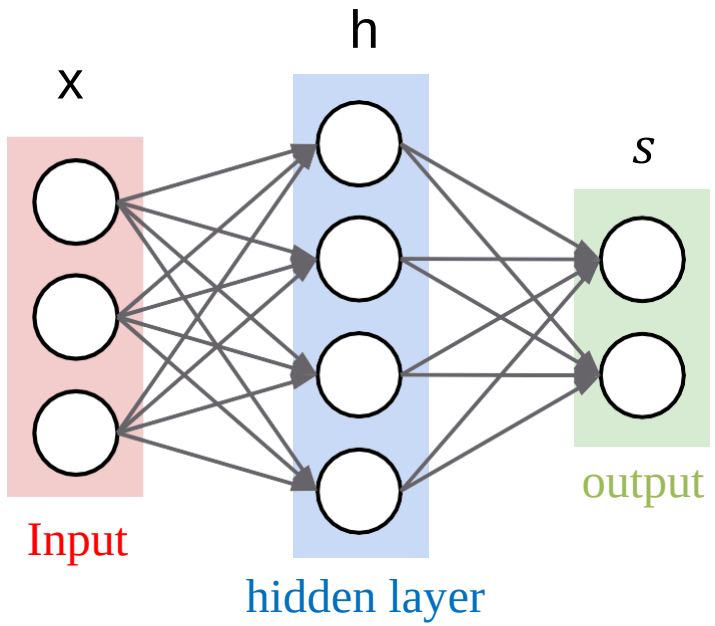
2 layer neural network



3 layer neural network

Neural Networks: Learning New Features

- Neural networks can learn the high-level features they need by combining low-level features.



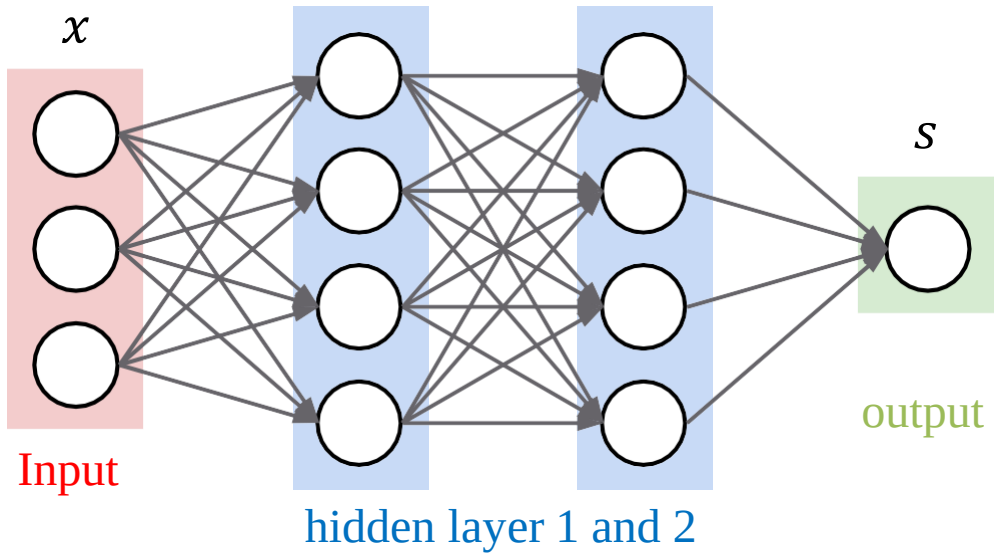
$s = Wx + b$ — Linear classification

$h = f(W_1 x + b_1)$ — 2 layer neural network

$s = W_2 h + b_2$

Neural Networks: Learning New Features

- Neural networks can learn the high-level features they need by combining low-level features.



$h = f(W_1 x + b_1)$

$s = W_2 h + b_2$

2 layer neural network

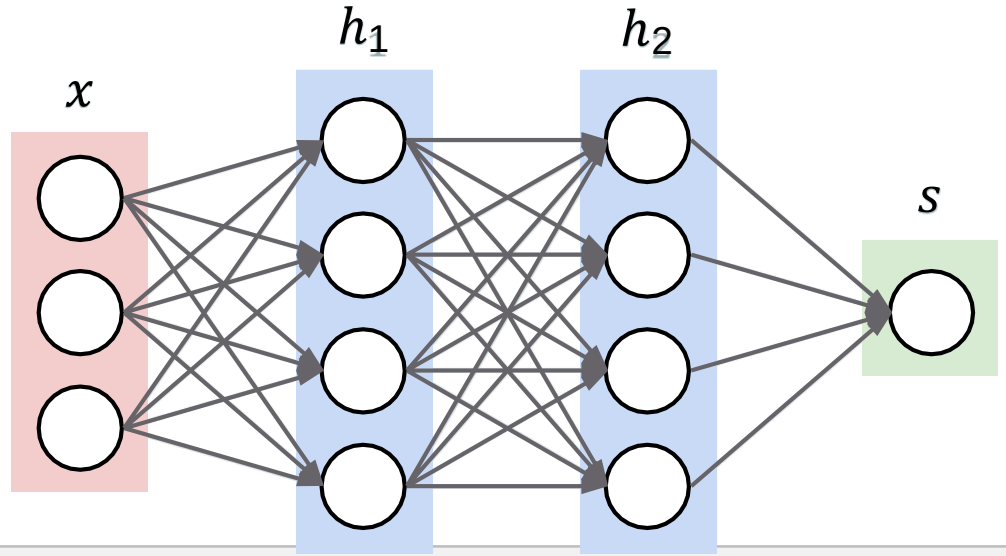
$h_1 = f(W_1 x + b_1)$

$h_2 = f(W_2 h_1 + b_2)$

$s = W_3 h_2 + b_3$

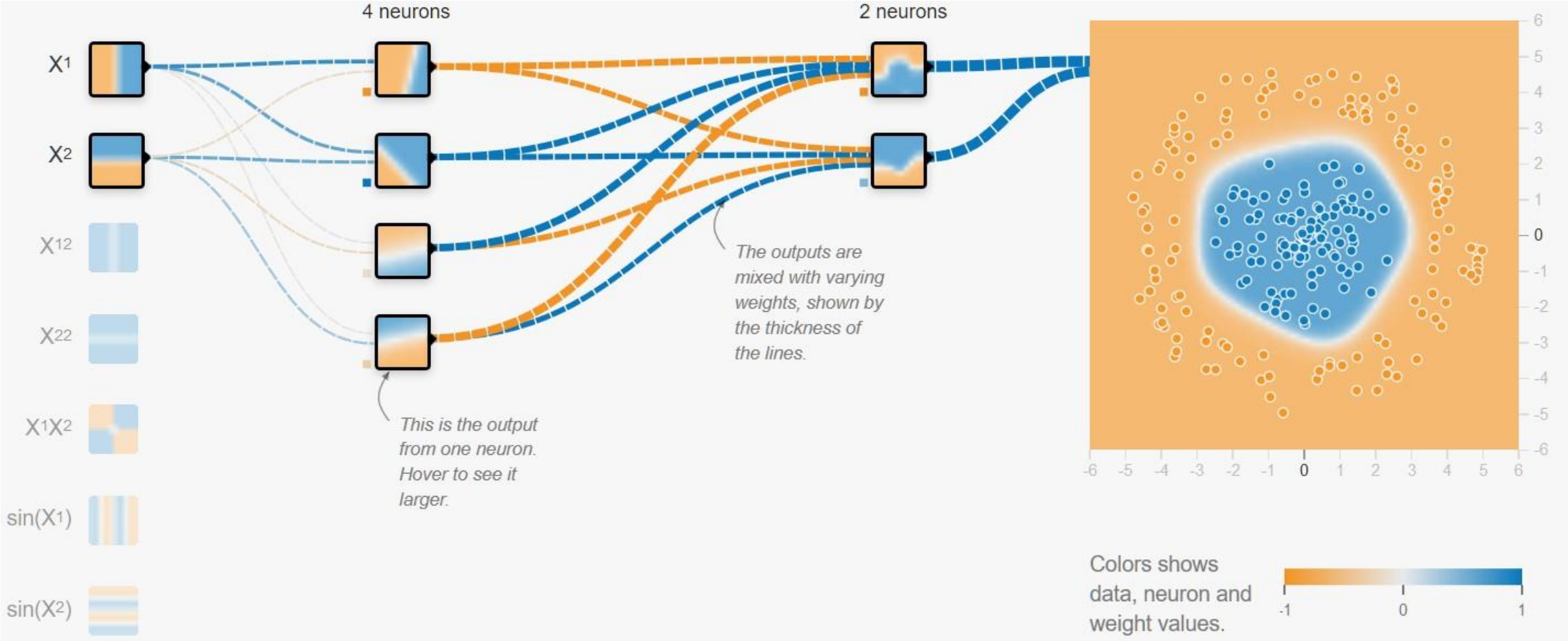
3 layer neural network

Neural Networks: Implementation of Forward Propagation




```
f = lambda x: 1.0 / (1.0 + np.exp(-x))                    # activation function (sigmoid)
x = np.random.randn(3, 1)                                 # random input vector (3x1)

h1 = f(W1 @ x  + b1)                                      # first hidden layer activations (4x1)
h2 = f(W2 @ h1 + b2)                                      # second hidden layer activations (4x1)
s  =   W3 @ h2 + b3                                       # scores (1x1)
```

Interactive Visualization



Activation Functions

- 3 layer neural network

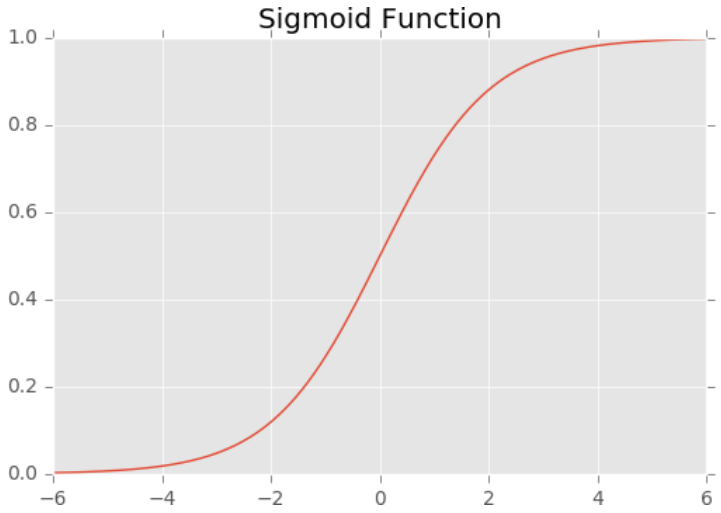
$$s = W_3 f(W_2 f(W_1 x))$$

- Importance of nonlinear activity functions in hidden layers.
  - Not using non-linear activity functions in hidden layers makes the neural network become a simple linear bundle!

$$s = W_3(W_2(W_1 x))$$
$$= (W_3 W_2 W_1)x$$
$$= Wx$$

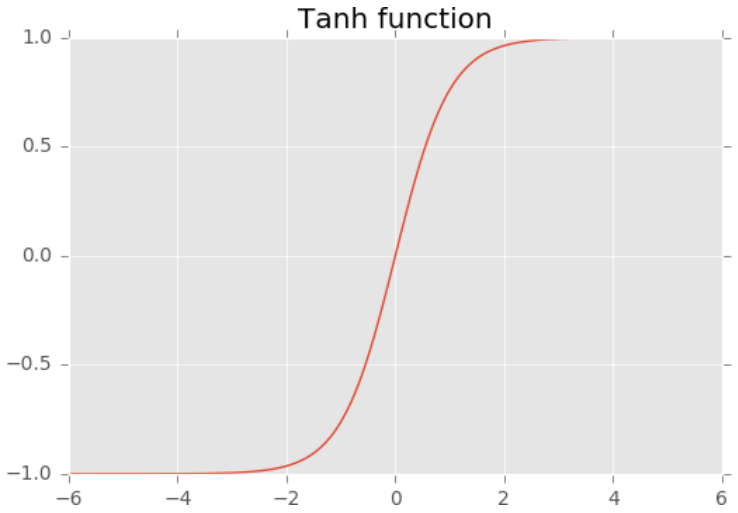
Activation Functions

Sigmoid



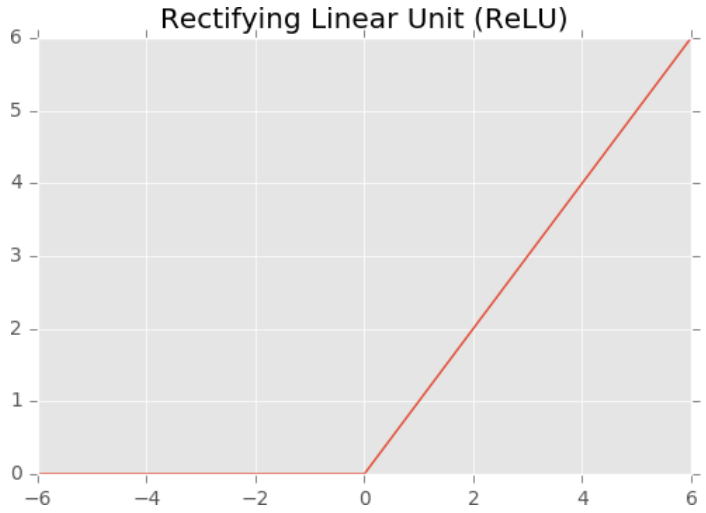
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Hyperbolic tangent



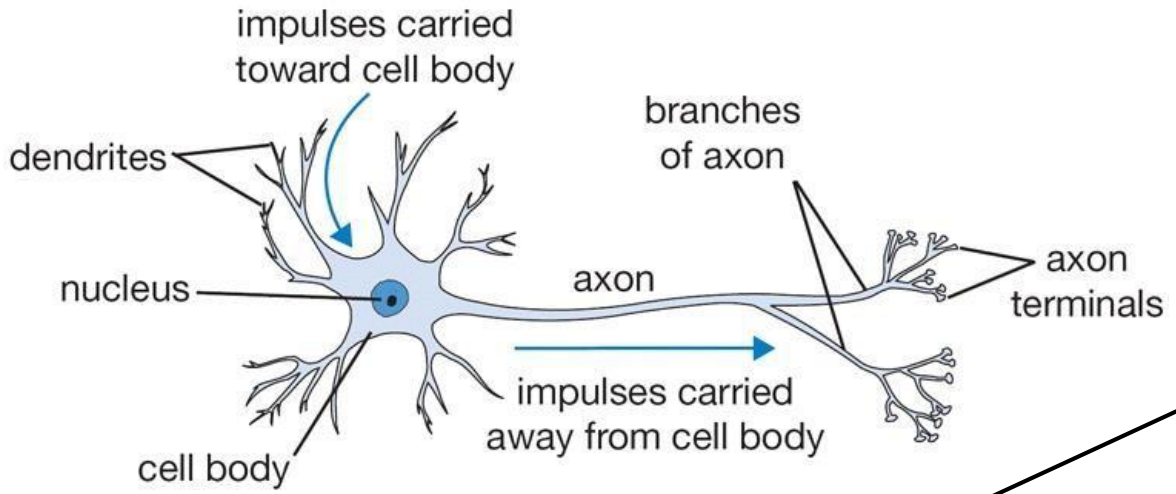
$$\tanh(x)$$

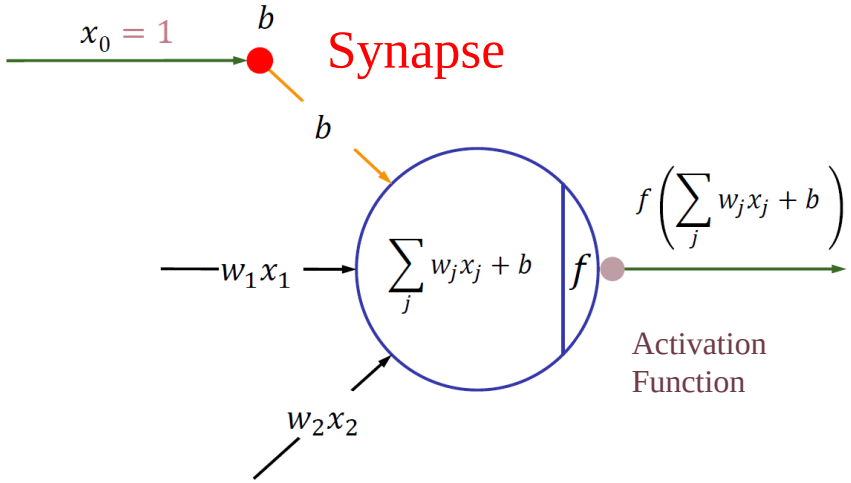
RELU



$$\max(0, x)$$

Neurons and Neural Networks





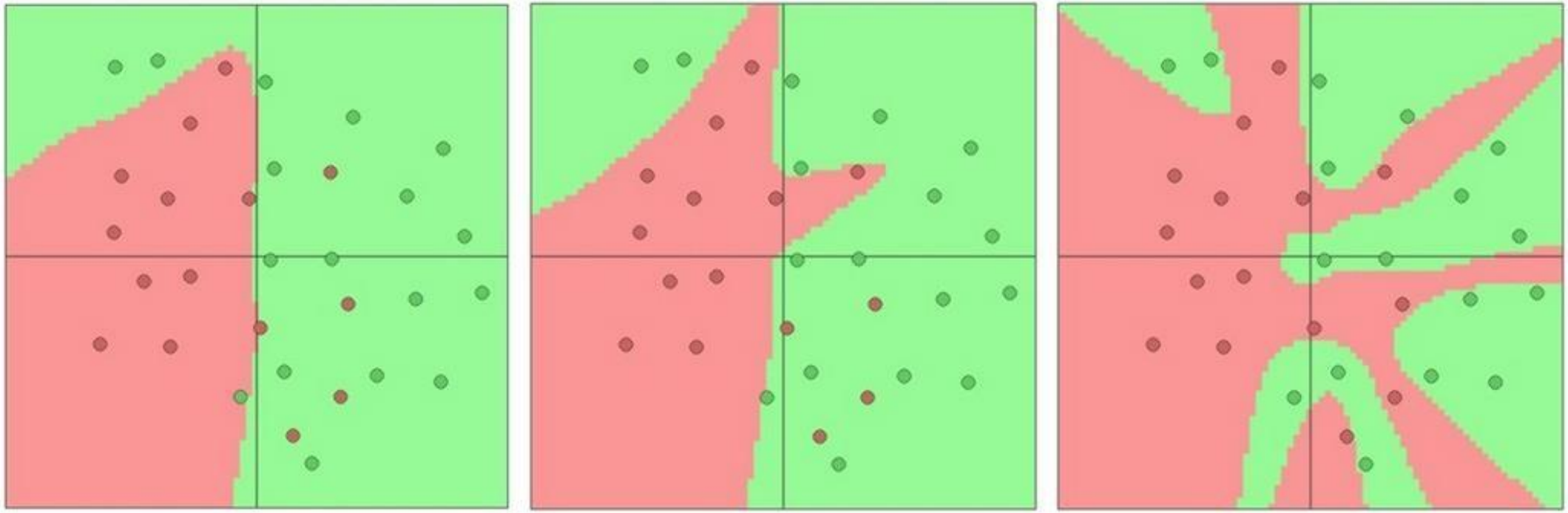

```
# assume w and x are 1d numpy arrays

net_input = np.sum(w * x) + b                # weighted sum of inputs
output = 1.0 / (1.0 + np.exp(-net_input))    # sigmoid function
```

Determine The Number And Size of Layers

3 neurons in the hidden layer

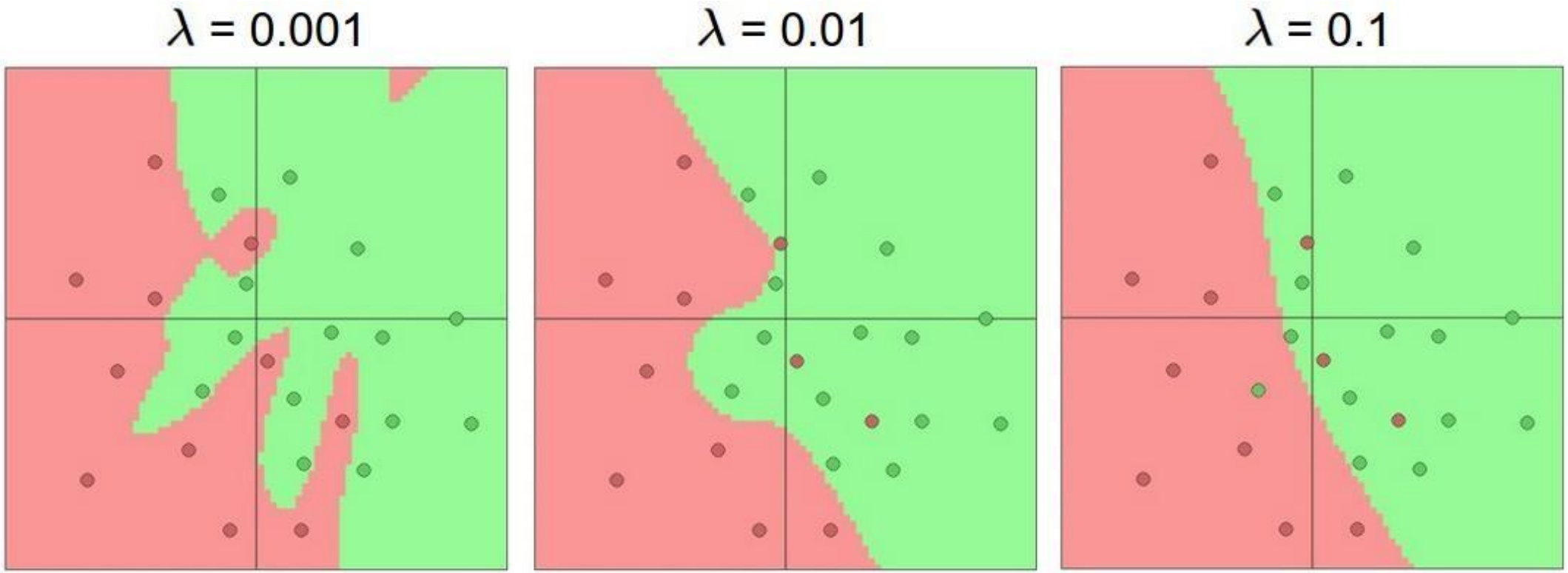
6 neurons in the hidden layer

20 neurons in the hidden layer



More neurons = more capacity

Determine The Number And Size of Layers

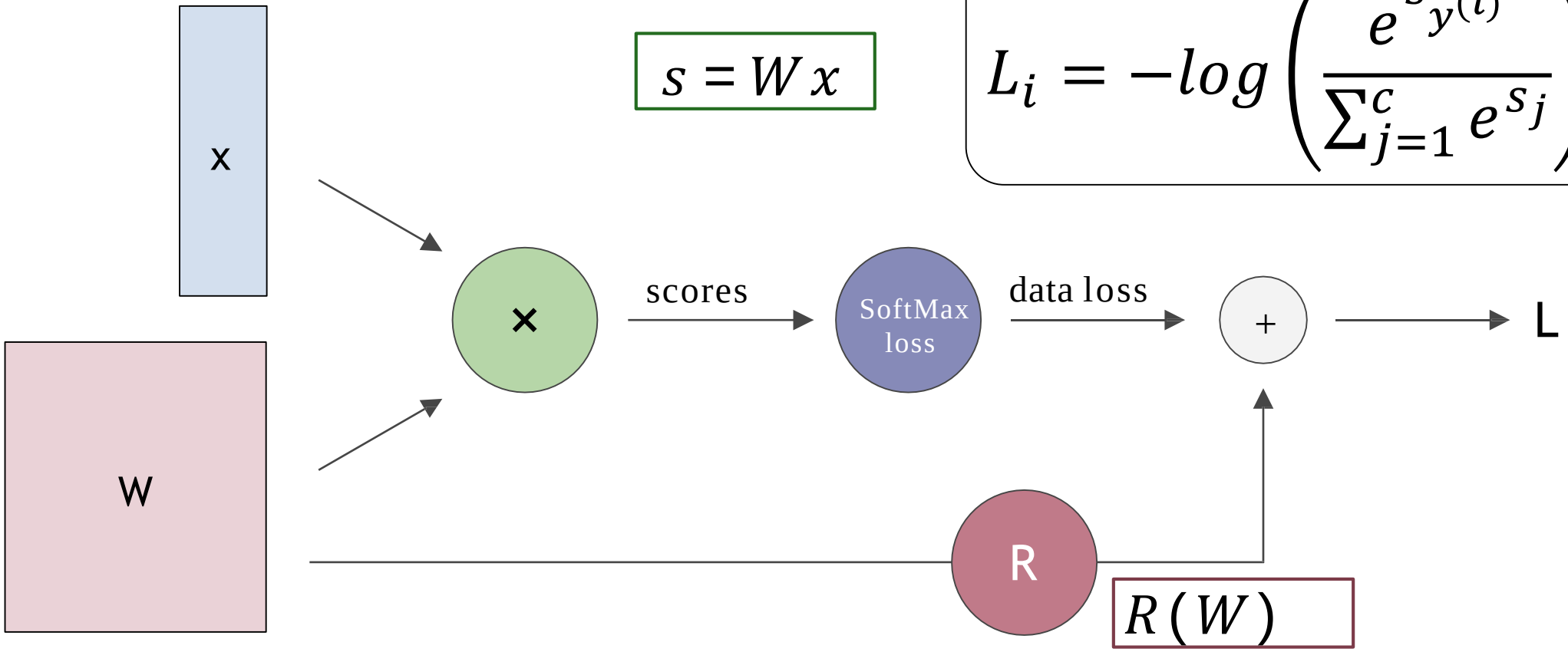


Do not use the size of the neural network as a regulator and use a **more robust** method instead.

L2

Back Propagation

SoftMax Classifier: Computational graph



Recall: Calculation of gradients in SoftMax category

- Cost Function.

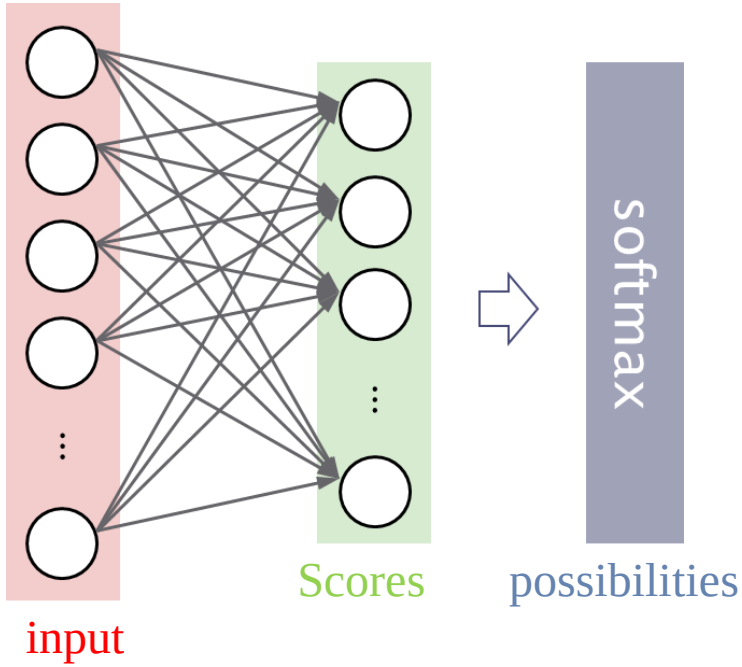
$$L = \frac{1}{m}\left(\sum_{i=1}^{m} -log\left(\frac{e^{s_{y^{(i)}}}}{\sum_{j=1}^{c} e^{s_j}}\right)\right) + \lambda||W||_2^2$$

- Calculation of gradients.

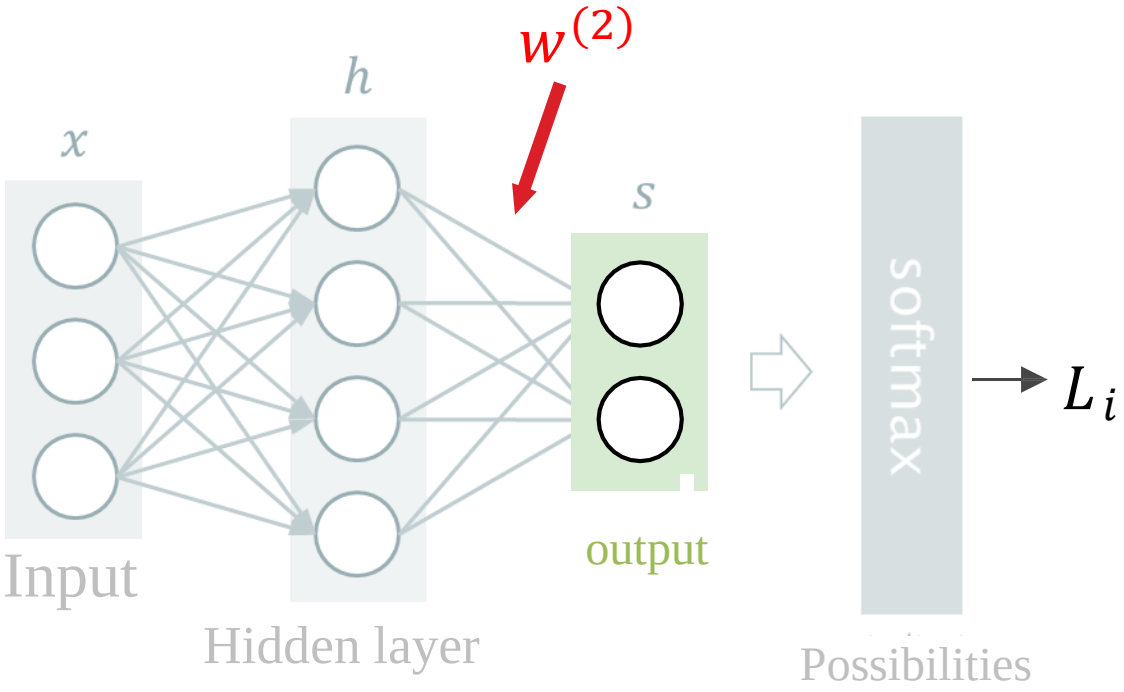
$$\frac{\partial L_i}{\partial w^{(k)}} = \frac{\partial L_i}{\partial s_k} \cdot \frac{\partial s_k}{\partial w^{(k)}}$$

$$= \left(p_k - \{y^{(i)} == k\}\right)x^{(i)}$$

$$\{y^{(i)} == k\} = \begin{cases} 1, & y^{(i)} = k \\ 0, & y^{(i)} \neq k \end{cases}$$



Back Propagation: Calculation of gradients

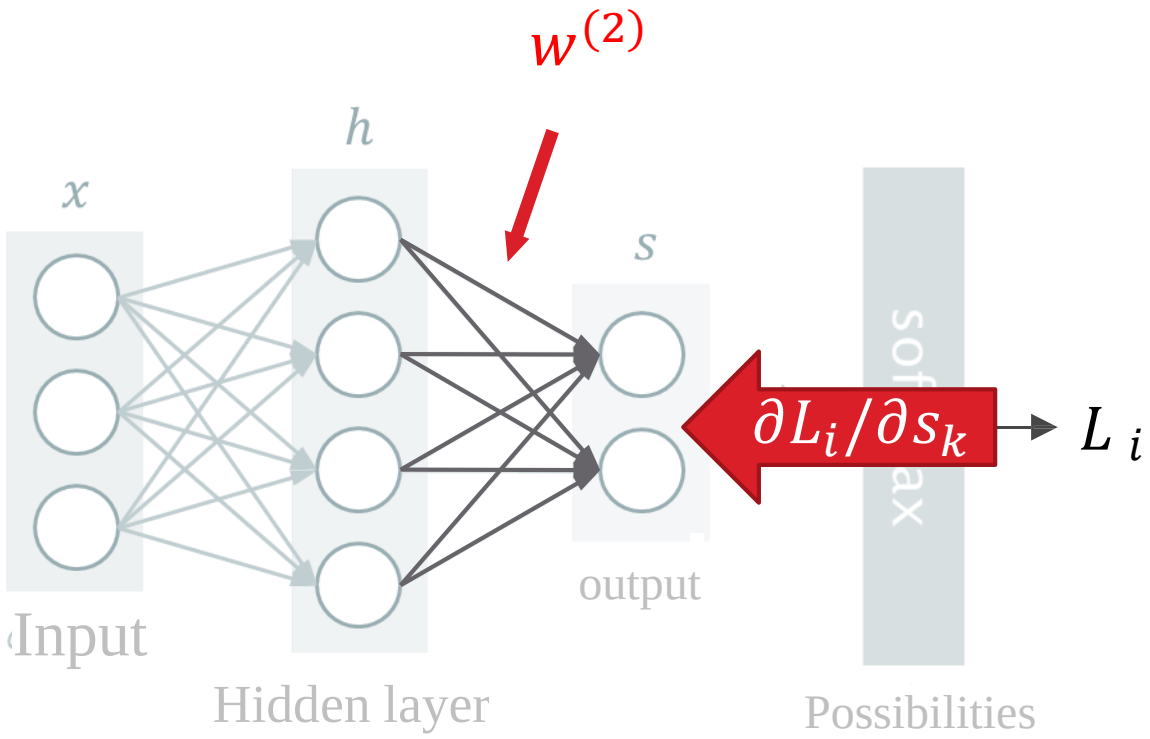


$$\frac{\partial L_i}{\partial w^{(2)}} = \frac{\partial L_i}{\partial s_k} \cdot \frac{\partial s_k}{\partial w^{(2)}}$$

$$= \left(p_k - \{y^{(i)} == k\}\right)$$

$$L = \frac{1}{m}\left(\sum_{i=1}^{m} -log\left(\frac{e^{s_{y^{(i)}}}}{\sum_{j=1}^{c} e^{s_j}}\right)\right) + \lambda||W||_2^2$$

Back Propagation: Calculation of gradients

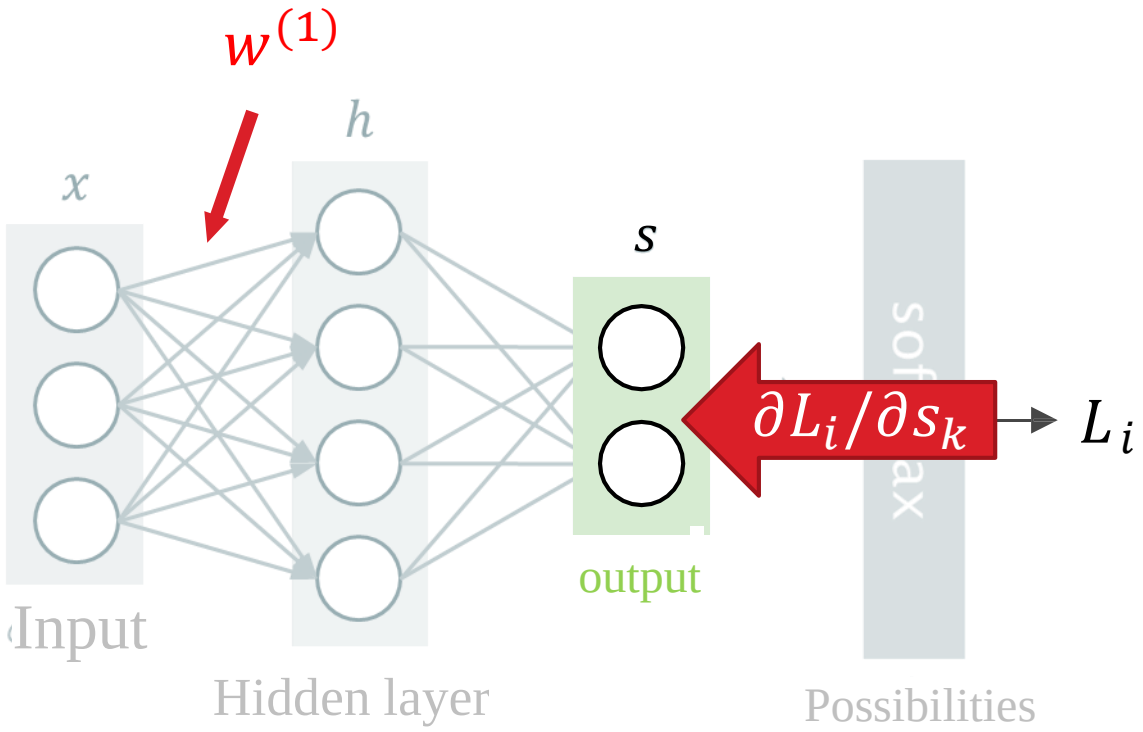


$$\frac{\partial L_i}{\partial w^{(2)}} = \frac{\partial L_i}{\partial s_k} \cdot \frac{\partial s_k}{\partial w^{(2)}}$$

$$= \left(p_k - \{y^{(i)} == k\}\right) h$$

$$L = \frac{1}{m}\left(\sum_{i=1}^{m} -log\left(\frac{e^{s_{y^{(i)}}}}{\sum_{j=1}^{c} e^{s_j}}\right)\right) + \lambda ||W||_2^2$$

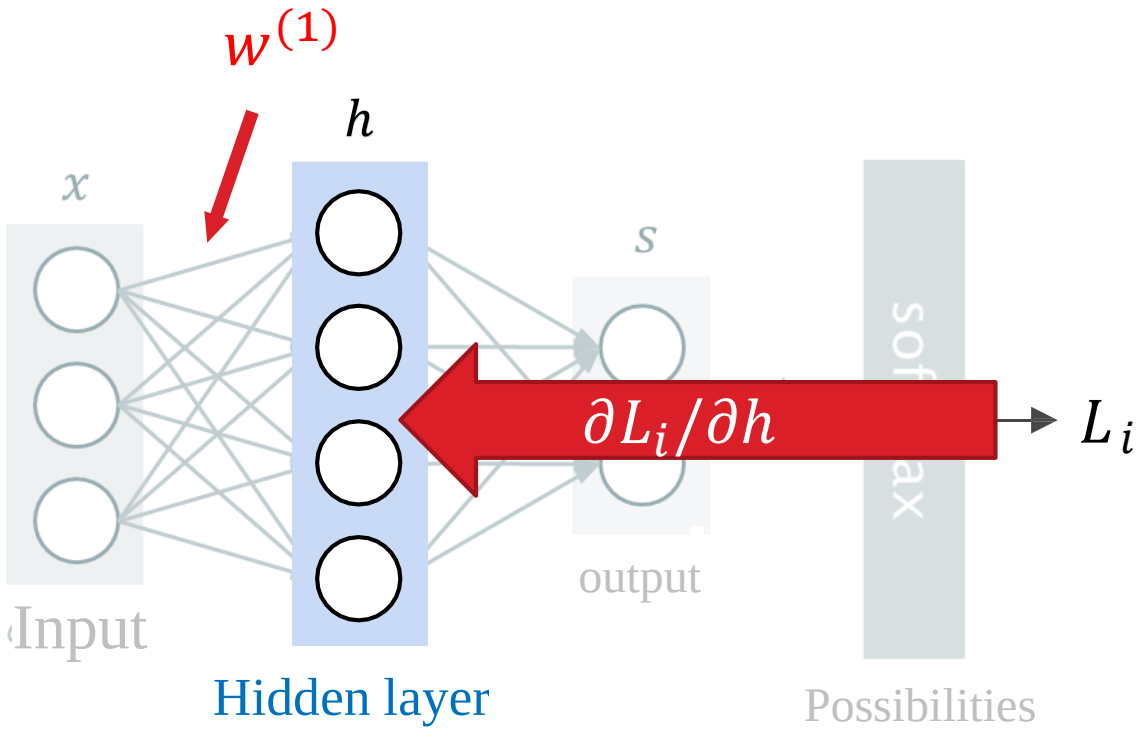
Back Propagation: Calculation of gradients



$$\frac{\partial L_i}{\partial w^{(1)}} = \frac{\partial L_i}{\partial s_k} \cdot \frac{\partial s_k}{\partial h} \cdot \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial W^{(1)}}$$

$$L = \frac{1}{m}\left(\sum_{i=1}^{m} -log\left(\frac{e^{s_{y^{(i)}}}}{\sum_{j=1}^{c} e^{s_j}}\right)\right) + \lambda||W||_2^2$$

Back Propagation: Calculation of gradients

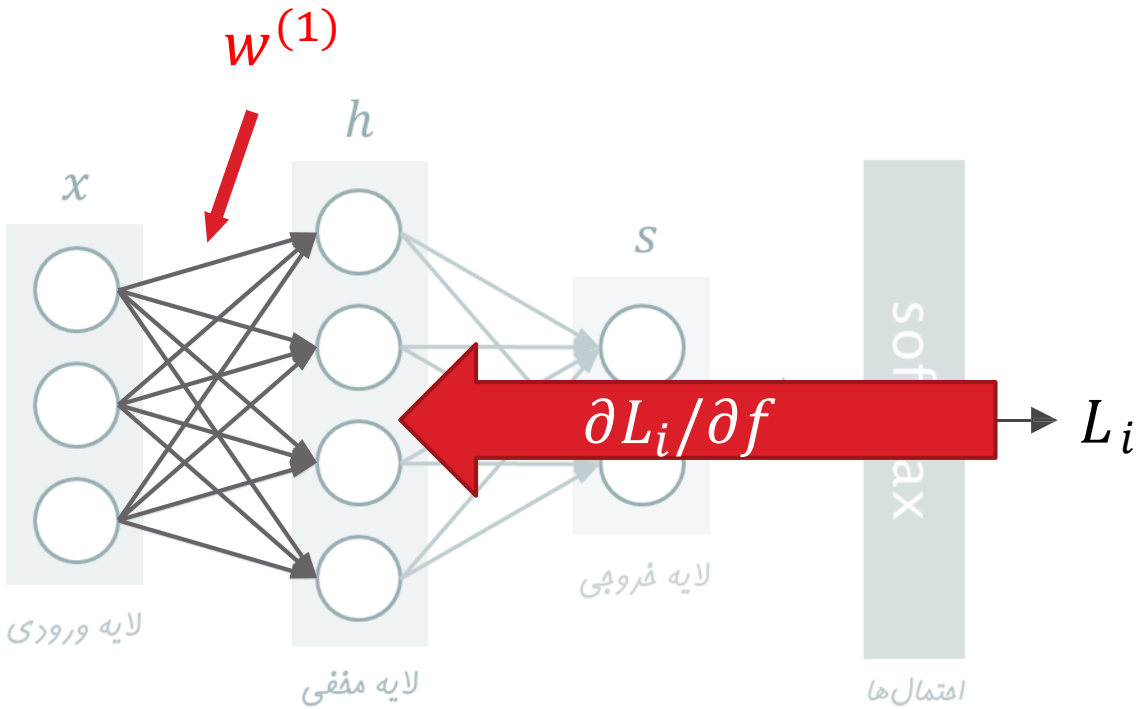


$$\frac{\partial L_i}{\partial w^{(1)}} = \frac{\partial L_i}{\partial s_k} \cdot \frac{\partial s_k}{\partial h} \cdot \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial W^{(1)}}$$

$$= \frac{\partial L_i}{\partial h} \cdot \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial W^{(1)}}$$

$$L = \frac{1}{m}\left(\sum_{i=1}^{m} -log\left(\frac{e^{s_{y^{(i)}}}}{\sum_{j=1}^{c} e^{s_j}}\right)\right) + \lambda||W||_2^2$$

Back Propagation: Calculation of gradients



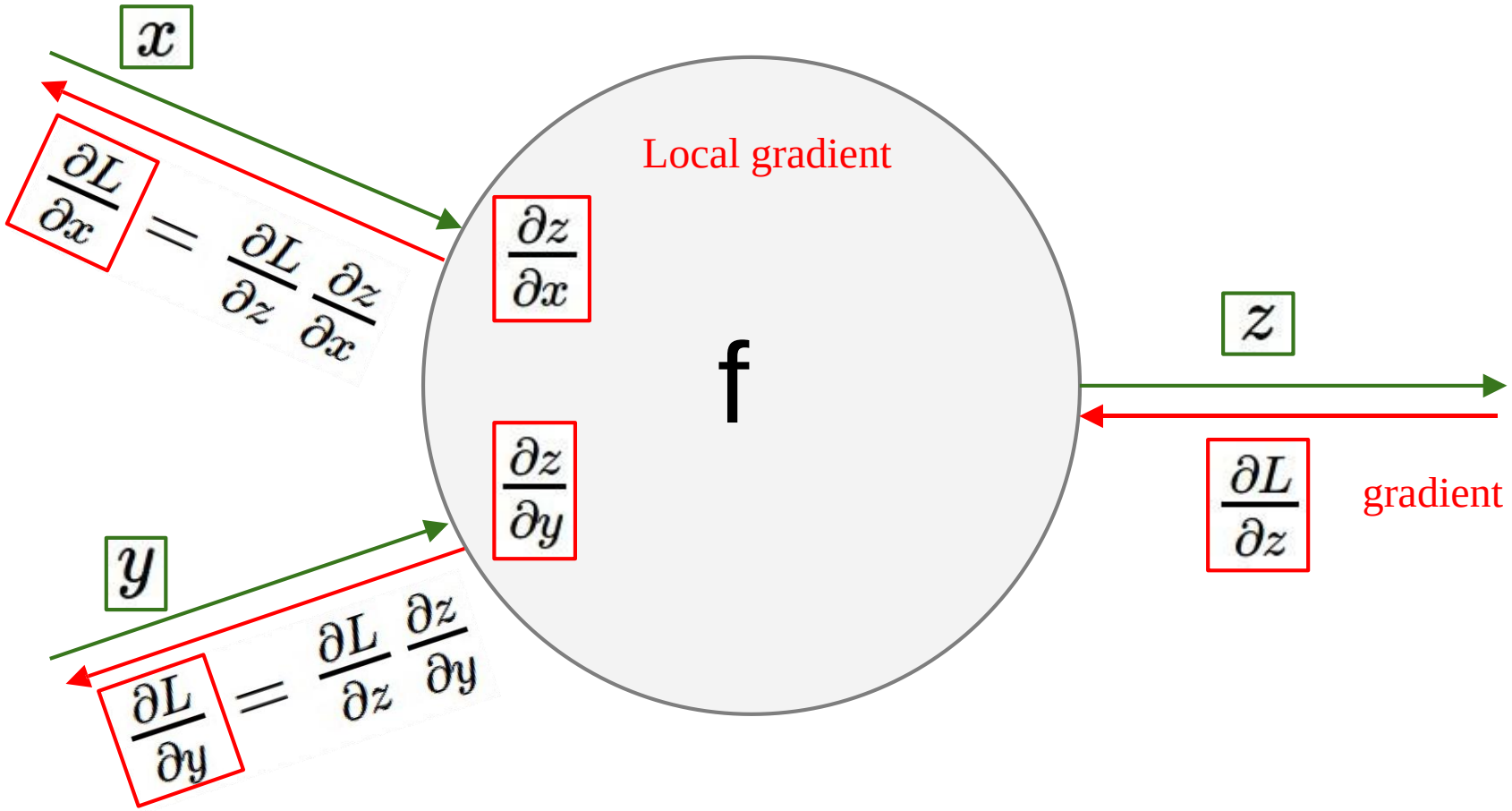
$$\frac{\partial L_i}{\partial w^{(1)}} = \frac{\partial L_i}{\partial s_k} \cdot \frac{\partial s_k}{\partial h} \cdot \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial W^{(1)}}$$

$$= \frac{\partial L_i}{\partial h} \cdot \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial W^{(1)}}$$

$$= \frac{\partial L_i}{\partial f} \cdot \frac{\partial f}{\partial W^{(1)}}$$

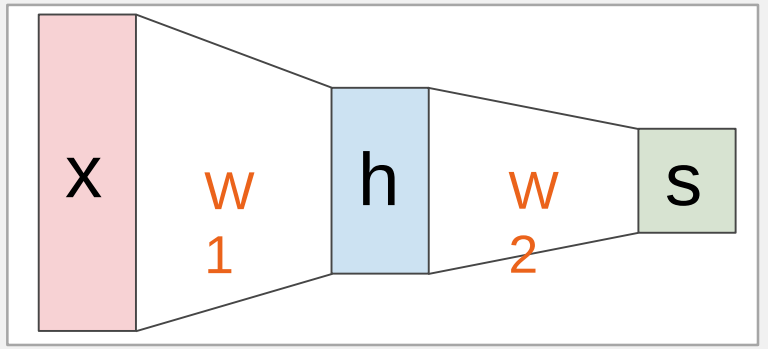
$$L = \frac{1}{m}\left(\sum_{i=1}^{m} -log\left(\frac{e^{s_{y^{(i)}}}}{\sum_{j=1}^{c} e^{s_j}}\right)\right) + \lambda||W||_2^2$$

Back Propagation: Calculation of gradients



Implementation of a two-layer neural network


```
# receive W1, W2, b1, b2 (weights/biases), X (data)


# forward pass:
h =      #... function of X, W1, b1
scores =  #... function of h, W2, b2
loss =    #... (several lines of code to evaluate SoftMax loss)
# backward pass:
dscores =      #... dL/dscores
dh, dW2, db2 =  #... dL/dh, dL/dW2, dl/db2
dW1, db1 =      #... dL/dW1, dL/db1
```



Summing up

- The number of parameters in a neural network can be very large:
  - It is impossible to write the relationship related to the gradient of all parameters manually!
- Publication. Applying the chain rule recursively throughout a computational graph in order toCalculation of the gradient of the cost function with respect to parameters, inputs and intermediate values.
- Computational graph. A graph structure where each node performs forward calculations and backward calculationsThe implementation is the same.
  - Forward calculations. Calculate the result of an operation and store the intermediate values needed to calculate the gradient.
  - Backward calculations. Using the chain rule to calculate the gradient of the cost function with respect to the inputs.

Advanced Optimization Methods

Advanced Optimization


```
from scipy.optimize import minimize

minimize(J, x0, args=(X_train, y_train), method='CG', jac=True)
```

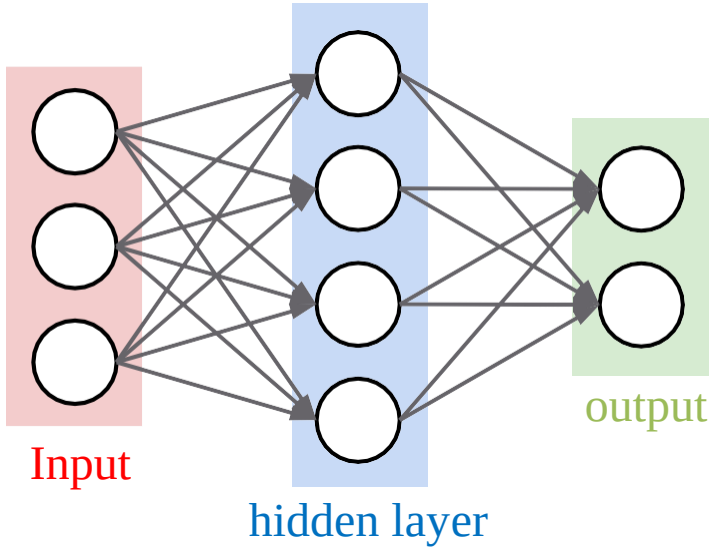
Implementation of the cost function

Parameters and gradients must be in the form of a vector

$$L = \frac{1}{m}\left(\sum_{i=1}^{m} -log\left(\frac{e^{s_{y^{(i)}}}}{\sum_{j=1}^{c} e^{s_j}}\right)\right) + \lambda||W||_2^2$$

Advanced Optimization

☐ Send parameters. Before sending the parameters to the optimization function, we must convert them all into a vector.



$W^{(1)} \in \mathbb{R}^{F \times H}$ $b^{(1)} \in \mathbb{R}^{H}$

$W^{(2)} \in \mathbb{R}^{H \times C}$ $b^{(2)} \in \mathbb{R}^{C}$

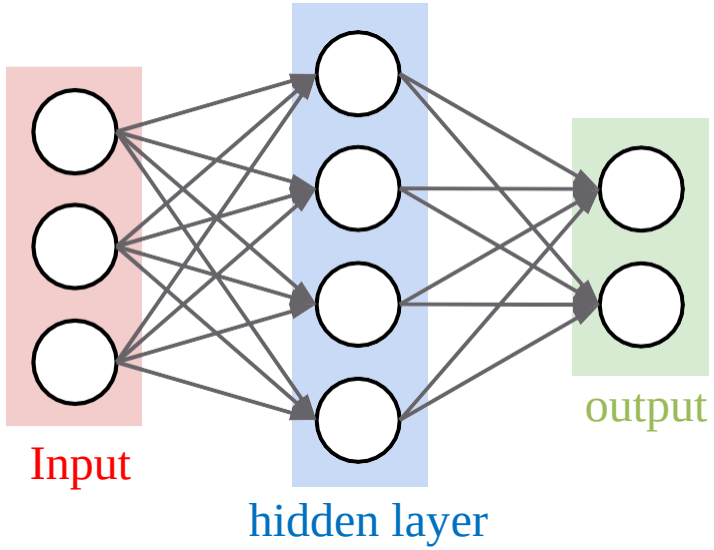
→ W

$F = 784$ $H = 20$ $C = 10$

```
W = np.concatenate((W1.ravel(), b1, W2.ravel(), b2), axis=0)
```

Advanced Optimization

- Get gradients. Before we get the gradients, we need to convert them all into a vector.



$F = 784 \qquad H = 20 \qquad C = 10$

$$dW^{(1)} \in \mathbb{R}^{F \times H} \qquad db^{(1)} \in \mathbb{R}^{H}$$

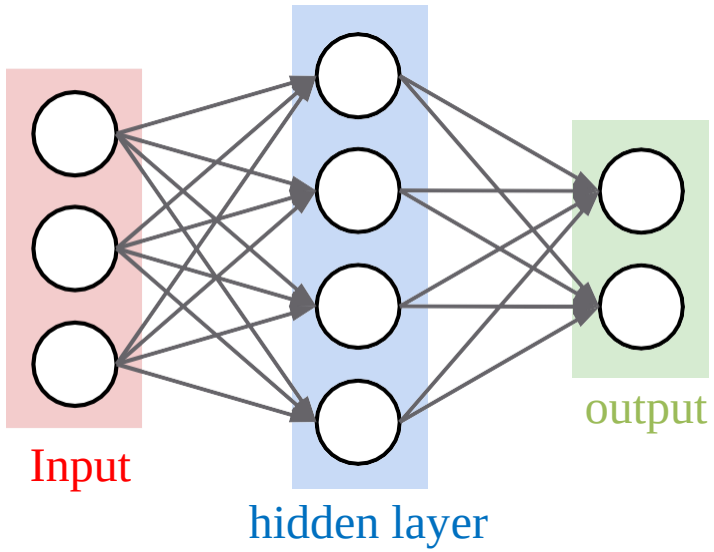
$$dW^{(2)} \in \mathbb{R}^{H \times C} \qquad db^{(2)} \in \mathbb{R}^{C}$$

$\Rightarrow dW$

```
dW = np.concatenate((dW1 .ravel () , db1 , dW2 .ravel (), db2 ), axis 0
```

Advanced Optimization

- Getting parameters. After optimization, we need to separate the different parameters.



$F = 784$ $H = 20$ $C = 10$

$$W^{(1)} \in \mathbb{R}^{F \times H} \qquad b^{(1)} \in \mathbb{R}^{H}$$

$$W^{(2)} \in \mathbb{R}^{H \times C} \qquad b^{(2)} \in \mathbb{R}^{C}$$

$\Leftarrow W$

```
W1 = np.reshape(W[: F * H], (F, H))  b1 = W[F * H: (F + 1) * H]
W2 =     # ... get W2 and reshape it  b2 =       # ... get b2
```

Advanced Optimization: Steps

- Create and randomize the matrices and vectors $W^{(i)}$ and $b^{(i)}$.

```
# create and init parameters W1, W2
W1 =     np.random.randn(F,     H) *    0.001
W2 =     np.random.randn(H,     C) *    0.001

b1 =  np.zeros((H,))
b2 =  np.zeros((C,))
```

- Convert matrices and vectors Wi and bi to vector W.

```
W = np.concatenate((W1.ravel(), b1, W2.ravel(), b2), axis=0)
```

Advanced Optimization: Steps

- Call the optimization function as follows:

```
result = minimize(J, x0=W, args=(X_train, y_train), method='CG', jac=True)
```

- After optimization, save the parameter value as follows:

```
W = result.x
```

- Extract the matrices and vectors $W^{(i)}$ and $b^{(i)}$ from the vector W.

```
W1 = np.reshape(W[: F * H], (F, H))
b1 = W[F * H: (F + 1) * H]

W2 =     # ... get W2 and reshape it  b2 =
         # ... get b2
```

Gradient Check

Numerical Estimation of Gradients

- The derivative of the function. In a 1-dimensional space

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- In a multidimensional space, the gradient of a function is a vector of partial derivatives.

Numerical Estimation of Gradients

W	$W + h$		$d\,W$
0.34	0.34	+ 0.0001	?
-1.11	-1.11		?
0.78	0.78		?
0.12	0.12		?
0.55	0.55		?
2.81	2.81		?
-3.10	-3.10		?
-1.50	-1.50		?
0.33	0.33		?
...	...		...
Loss 1.25347	**Loss 1.25322**		

Numerical Estimation of Gradients

W	$W + h$		dW
0.34	0.34	+ 0.0001	-2.50
-1.11	-1.11		?
0.78	0.78		?
0.12	0.12		?
0.55	0.55		
2.81	2.81		
-3.10	-3.10		
-1.50	-1.50		
0.33	0.33		?
...	...		...
Loss 1.25347	**Loss 1.25322**		

(1.25322 - 1.25347) / 0.0001

= **-2.5**

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Numerical Estimation of Gradients

W	$W + h$		$d\,W$
0.34	0.34		-2.50
-1.11	-1.11	+ 0.0001	?
0.78	0.78		?
0.12	0.12		?
0.55	0.55		?
2.81	2.81		?
-3.10	-3.10		?
-1.50	-1.50		?
0.33	0.33		?
...	...		...
Loss 1.25347	**Loss 1.25353**		

Numerical Estimation of Gradients

W	$W + h$		dW
0.34	0.34		-2.50
-1.11	-1.11	+ 0.0001	**0.60**
0.78	0.78		?
0.12	0.12		?
0.55	0.55		
2.81	2.81		
-3.10	-3.10		
-1.50	-1.50		
0.33	0.33		?
...	...		...
Loss 1.25347	**Loss 1.25353**		

(1.25353 - 1.25347) / 0.0001

= **0.6**

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Numerical Estimation of Gradients

W	$W + h$		$d\,W$
0.34	0.34		-2.50
-1.11	-1.11		0.60
0.78	0.78	+ 0.0001	?
0.12	0.12		?
0.55	0.55		?
2.81	2.81		?
-3.10	-3.10		?
-1.50	-1.50		?
0.33	0.33		?
...	...		...
Loss 1.25347	**Loss 1.25347**		

Numerical Estimation of Gradients

W	$W + h$		dW
0.34	0.34		-2.50
-1.11	-1.11		0.60
0.78	0.78	+ 0.0001	**0.00**
0.12	0.12		?
0.55	0.55		
2.81	2.81		
-3.10	-3.10		
-1.50	-1.50		
0.33	0.33		?
...	...		...
Loss 1.25347	**Loss 1.25347**		

(1.25347 - 1.25347) / 0.0001

= **0.0**

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Numerical Estimation of Gradients


```
def eval_numerical_gradient(f, x):

    fx = f(x)
    grad = np.zeros(x.shape)
    h = 0.00001

    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])

    while not it.finished:

        ix = it.multi_index
        old_value = x[ix]
        x[ix] += h
        fxh = f(x) # evaluate f(x + h)
        x[ix] = old_value

        grad[ix] = (fxh – fx) / h # compute the partial derivative
        it.iternext() # step to next dimension
return grad
```

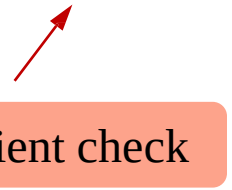
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Disadvantages
  - Approximate
  - very time consuming

Gradient Check

- In summary.
  - Numerical gradient: approximate, time consuming, easy to implement!
  - Analytical gradient: accurate, fast, possibility of error in implementation!

- In practice.
  - We always use analytical gradients.
  - But in order to ensure the correctness of the implementation, we compare the analytical gradient with the numerical gradient.

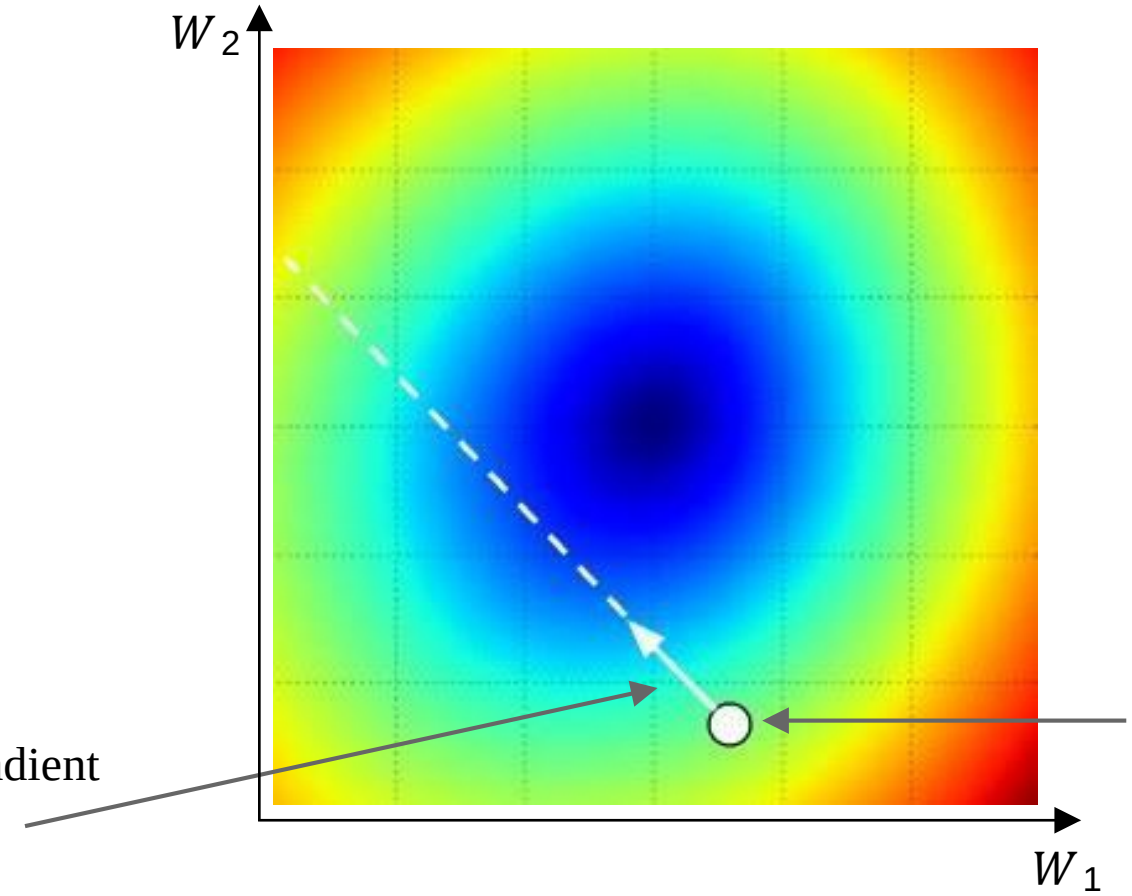


Descending Gradient With Small Batches

Gradient Descent Algorithm


```
# Vanilla Gradient Descent

while True:
    gradient = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * gradient # weight update
```

Gradient Descent Algorithm



A More Efficient Version of Gradient Descent Algorithm

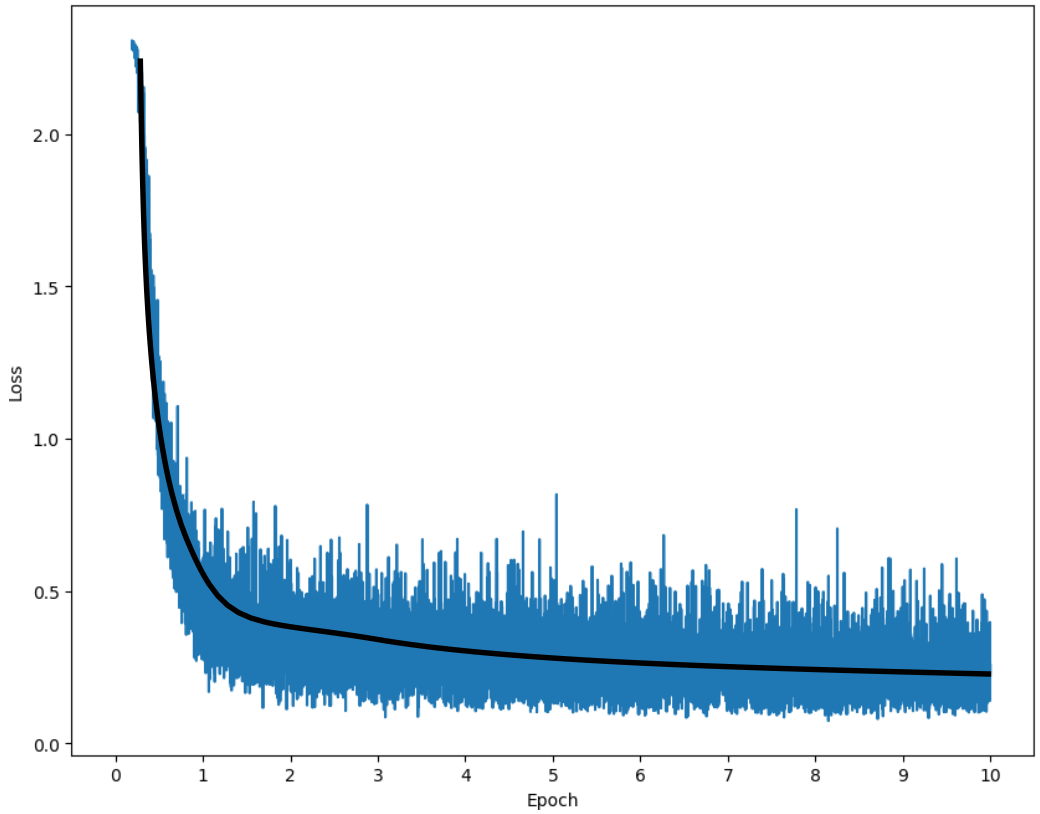
- A descending gradient with small batches.
  - Use only a small part of the training data to calculate the gradient of the cost function in each iteration.

```
while True:

data_batch = sample_training_data(data, 256) # sample 256 examples

gradient = evaluate_gradient(loss_fun, data_batch, weights)

weights += -step_size * gradient # weight update
```

  - Common values for batch size: 32, 64, 128 and 256.

A Descending Gradient With Small Batches

- Example of implementation of gradient descent algorithm with small batches in a neural network.



The cost will decrease over time.