NEURAL NETWORK FOR MACHINE LEARNING

한재근 과장 | jahan@nvidia.com
유현곤 부장 | hryu@nvidia.com / 양한별 과장 | hanbyuly@nvidia.com

NVIDIA

AGENDA

Background of Neural Network

Scoring / Loss / SVM / Softmax / Regularization

Neural Network Architecture

Optimization / Training

Artificial Neural Network

Activation Function

Deep Learning

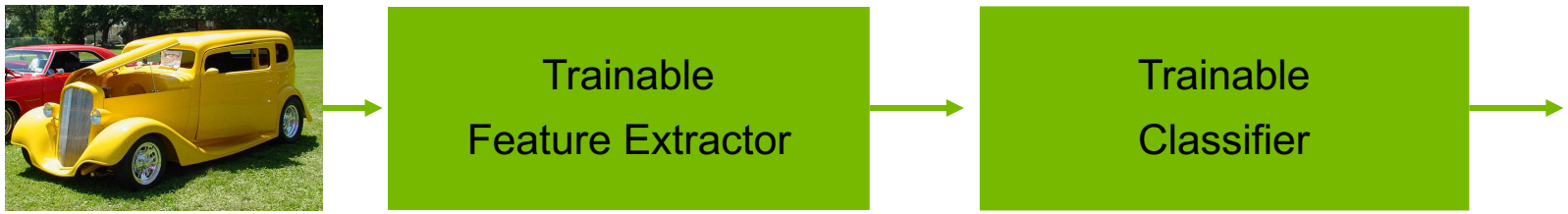
Learning Representation/Features

The traditional model of pattern recognition (since the late 50's)

Fixed/engineered features (or fixed kernel) + trainable classifier

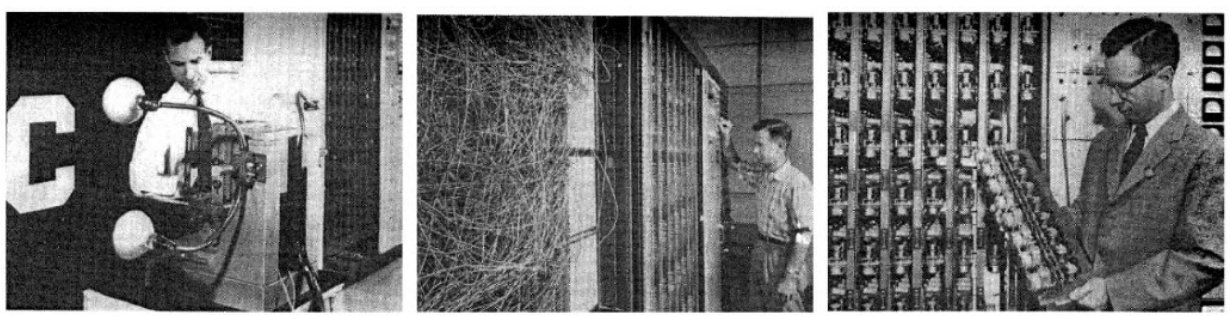
End-to-end learning / Feature learning / Deep learning

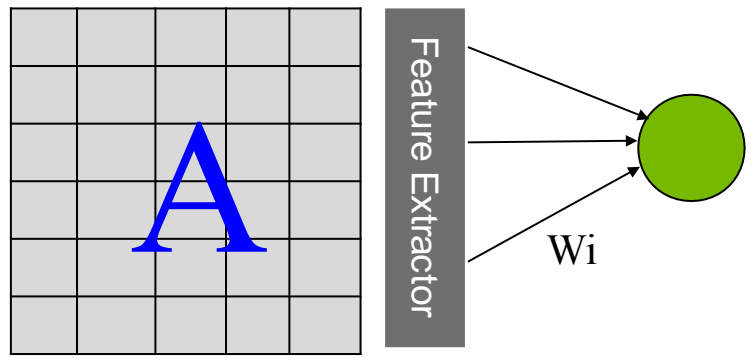
Trainable features (or kernel) + trainable classifier



This basic model has not evolved much since the 50's

- The first learning machine: the Perceptron
  - Built at Cornell in 1960
- The Perceptron was a linear classifier on top of a simple feature extractor
- The vast majority of practical applications of ML today use glorified linear classifiers or glorified template matching.
- Designing a feature extractor requires considerable efforts by experts.

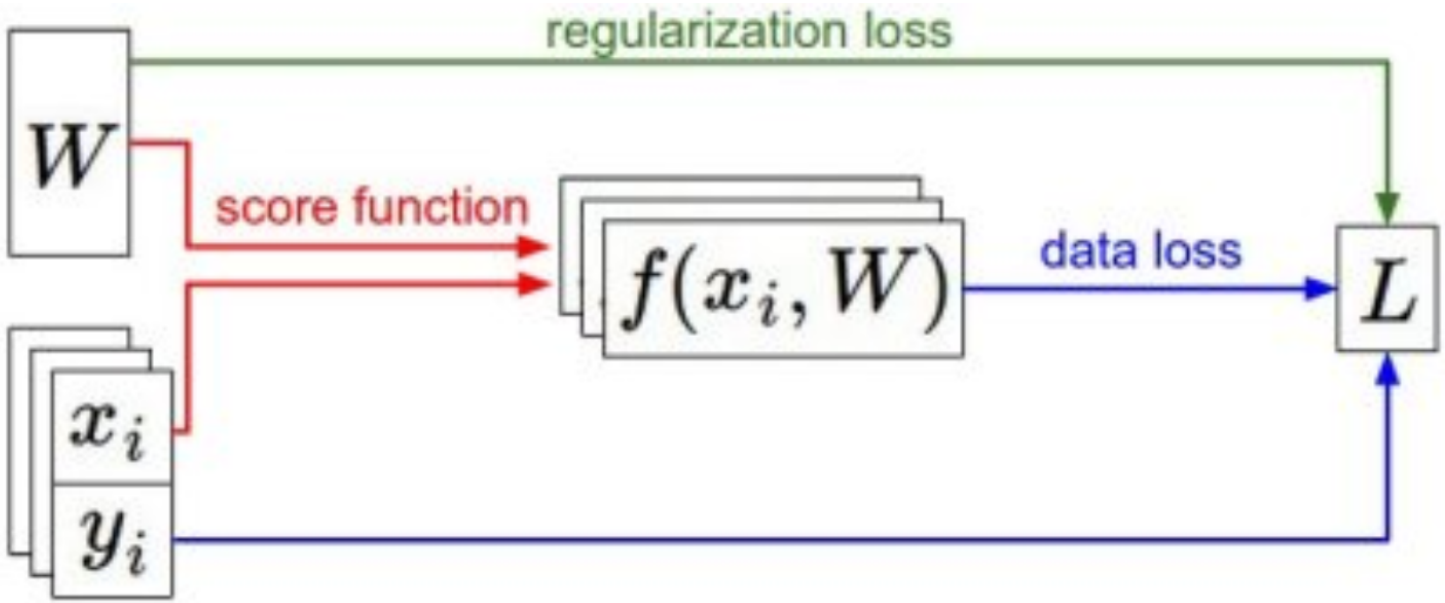


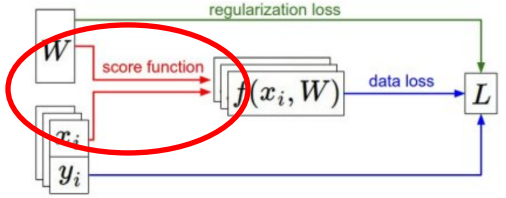


$$y = sign\left(\sum_{i=1}^{N} W_i F_i(X) + b\right)$$

Scoring

Summary





Inference

Scoring

Input Data

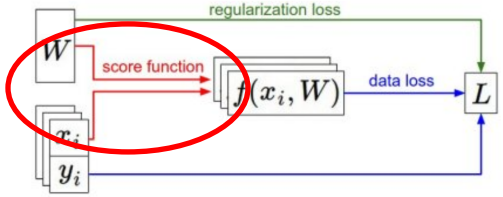
$$f(x_i, W, b) = Wx_i + b$$

Class Score

Weights

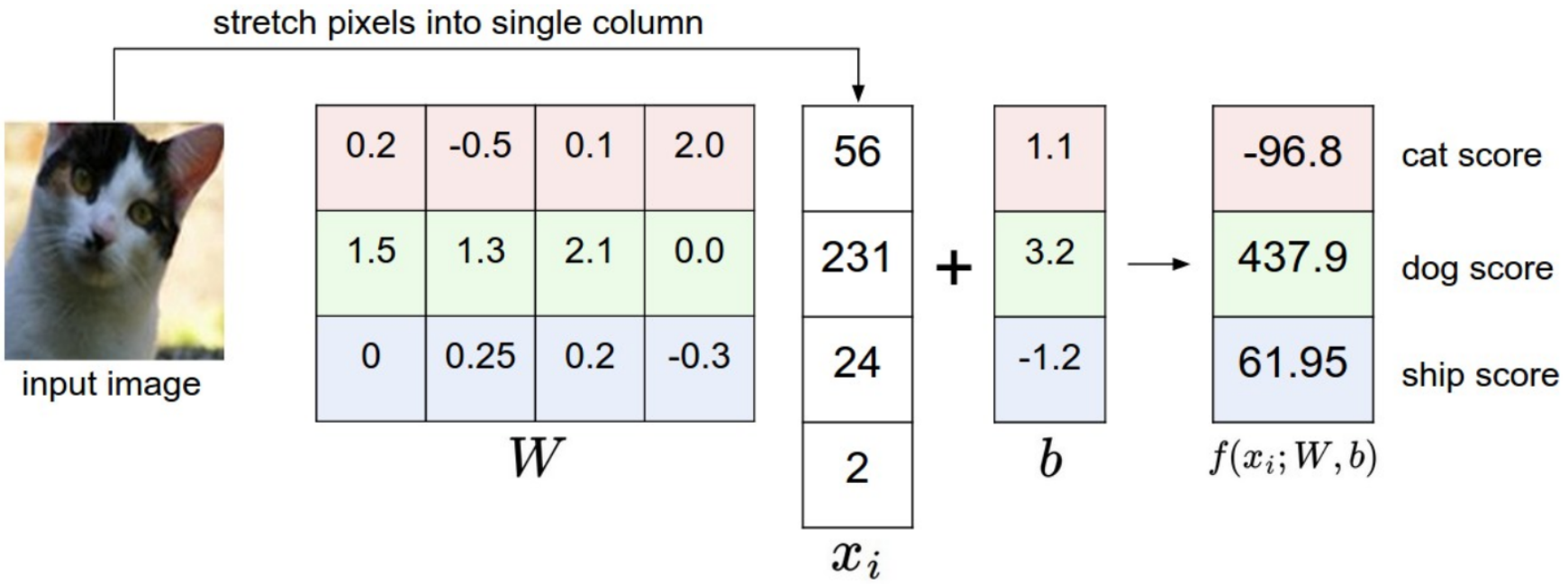
Bias

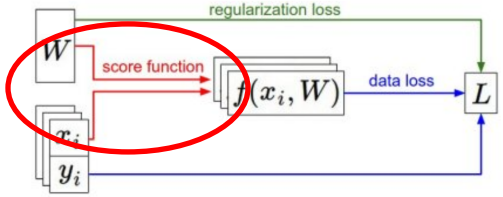
"Parameters"



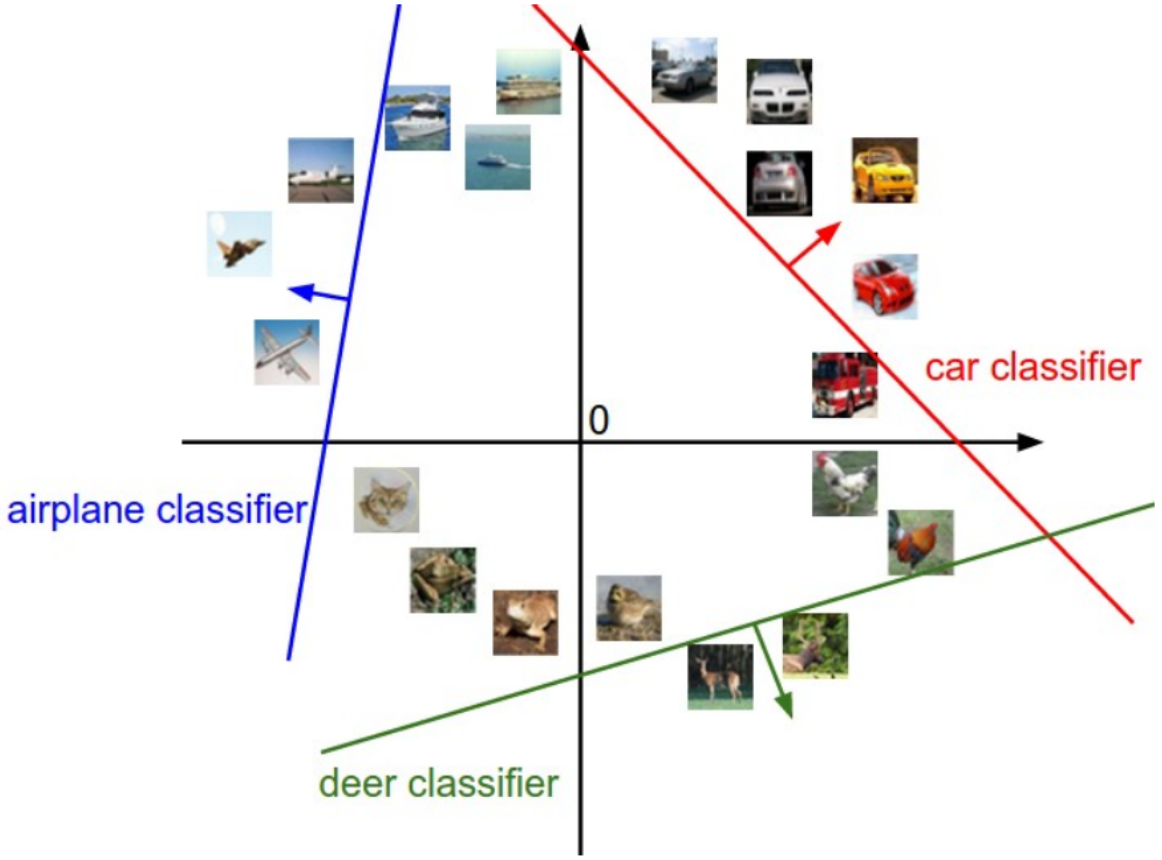
A Linear classifier for Image

$$f(x_i, W, b) = Wx_i + b$$

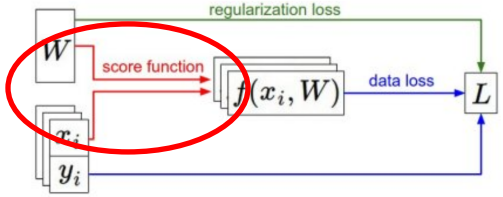




Interpretation



$$f(x_i, W, b) = Wx_i + b$$



The bias trick

$$f(x_i, W, b) = Wx_i + b \longrightarrow f(x_i, W) = Wx_i$$

0.2	-0.5	0.1	2.0
1.5	1.3	2.1	0.0
0	0.25	0.2	-0.3

W

56
231
24
2

x_i

+

1.1
3.2
-1.2

b

$\longleftrightarrow$

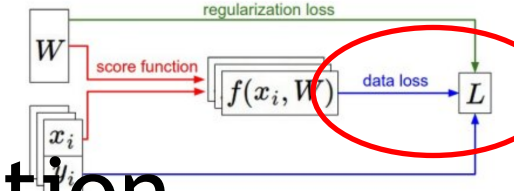
0.2	-0.5	0.1	2.0	1.1
1.5	1.3	2.1	0.0	3.2
0	0.25	0.2	-0.3	-1.2

W b

new, single W

56
231
24
2
1

x_i



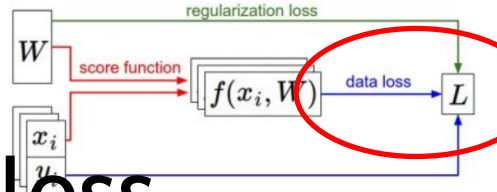
Define a **loss/cost/objective** function

We have **scores** and **labels**: $f(x_i, W)$ y_i

Example: $f(x_i, W) = [13, -7, 11]$

$y_i = 0$

We want to define a loss function L_i that describes how unhappy we are with these scores.

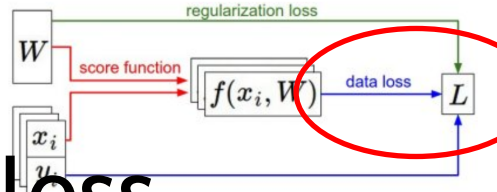


Loss example: the multiclass SVM loss

$$L_i = \sum_{j \neq y_i} \max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta)$$

(One possible generalization of Binary Support Vector Machine to multiple classes)

$$L_i = C \max(0, 1 - y_i w^T x_i) + R(W)$$



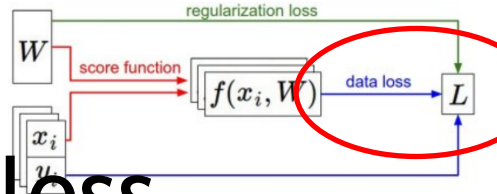
Loss example: the multiclass SVM loss

$$L_i = \sum_{j \neq y_i} \max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta)$$

loss due to example i

sum over all incorrect labels

difference between the correct class score and incorrect class score

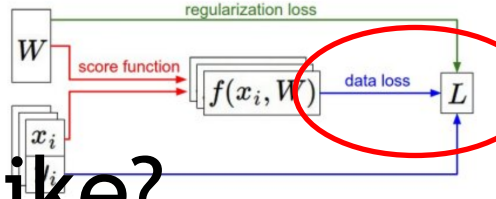


Loss example: the multiclass SVM loss

Vectorized implementation

$$L_i = \sum_{j \neq y_i} \max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta)$$

```
def L_i_vectorized(x, y, W):
  scores = W.dot(x)
  margins = np.maximum(0, scores - scores[y] + 1)
  margins[y] = 0
  loss_i = np.sum(margins)
  return loss_i
```

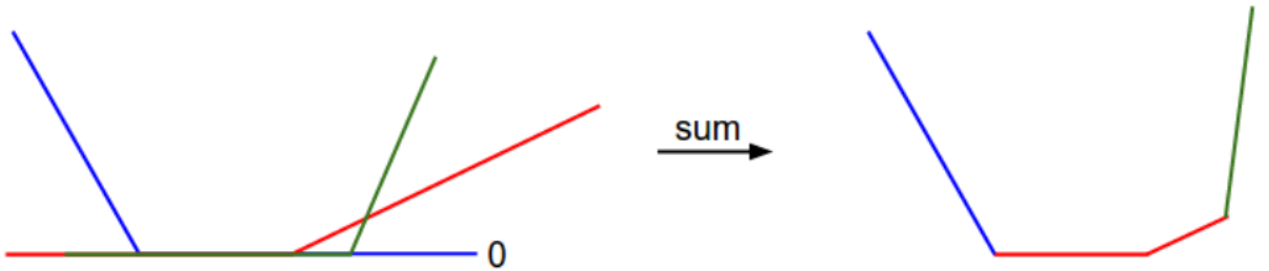


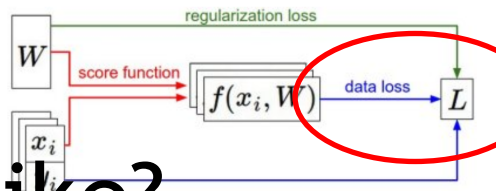
What does the Loss function look like?

$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$

Example: 3 examples with 3 classes

$$L_0 = \max(0, w_1^T x_0 - w_0^T x_0 + 1) + \max(0, w_2^T x_0 - w_0^T x_0 + 1)$$
$$L_1 = \max(0, w_0^T x_1 - w_1^T x_1 + 1) + \max(0, w_2^T x_1 - w_1^T x_1 + 1)$$
$$L_2 = \max(0, w_0^T x_2 - w_2^T x_2 + 1) + \max(0, w_1^T x_2 - w_2^T x_2 + 1)$$
$$L = (L_0 + L_1 + L_2)/3$$

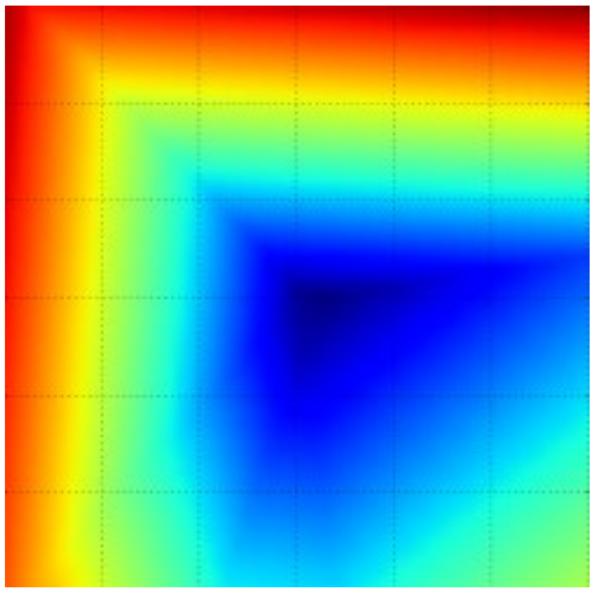




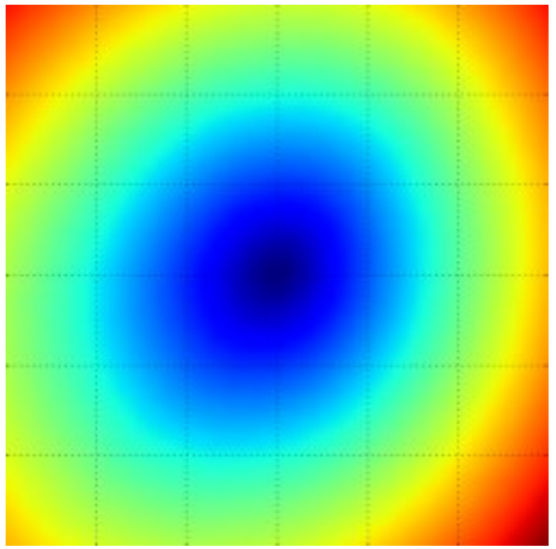
What does the Loss function look like?

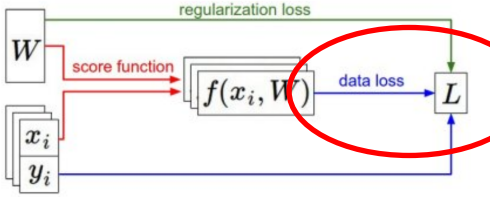
$$L_i = \sum_{j \neq y_i} \max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta)$$

$$L_i(W + aW_1 + bW_2)$$



the full data loss:





Softmax classifier

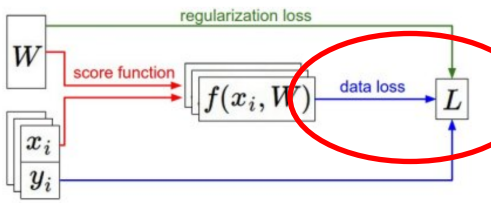
Extension of logistic regression to multiple classes

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

softmax function

$$P(y_i \mid x_i; W)$$

i.e. we're minimizing the negative log likelihood.



Example

matrix multiply + bias offset

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

W

-15
22
-44
56

x_i

+

0.0
0.2
-0.3

b

y_i 2

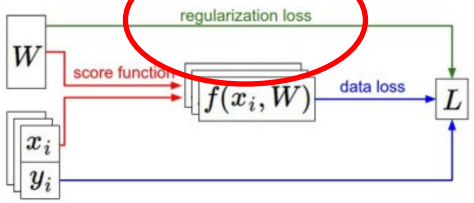
hinge loss (SVM)

-2.85
0.86
0.28

max(0, -2.85 - 0.28 + 1) +
max(0, 0.86 - 0.28 + 1)
=
1.58

cross-entropy loss (Softmax)

-2.85		0.058		0.016	
0.86	*exp* →	2.36	*normalize* → (to sum to one)	0.631	- log(0.353) = **0.452**
0.28		1.32		0.353	



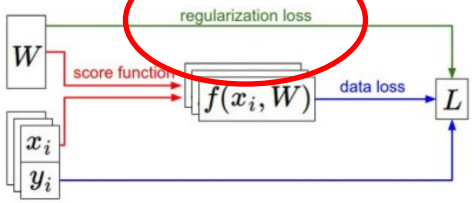
Regularization

Penalizing extreme weights

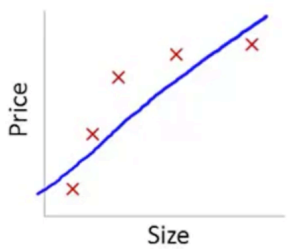
$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda R(W)$$

Regularization strength

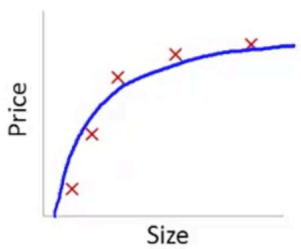


Regularization Parameter



$\theta_0 + \theta_1 x$

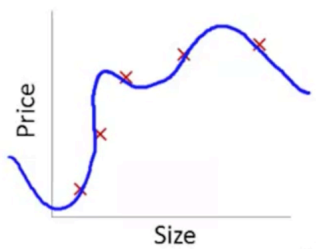
High bias
(underfit)



$\theta_0 + \theta_1 x + \theta_2 x^2$

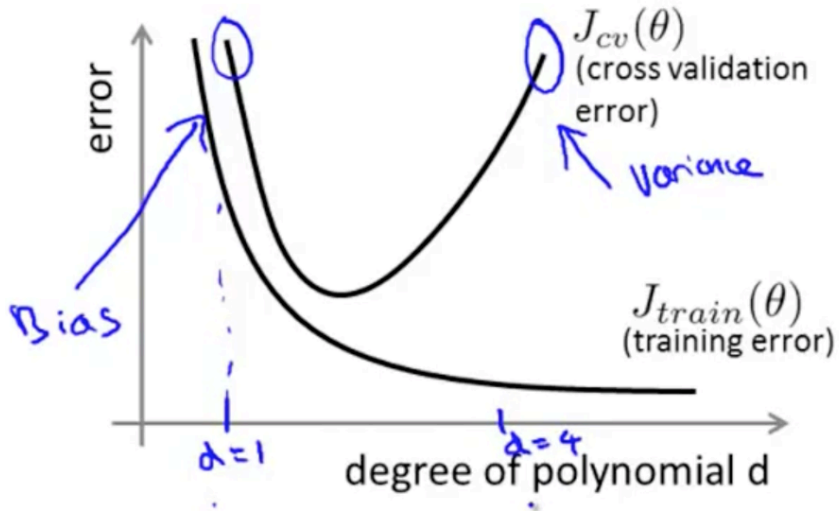
"Just right"

d =



$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

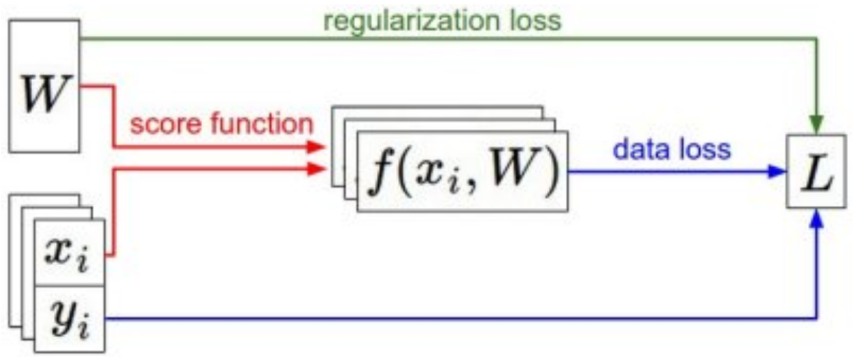
High variance
(overfit)



Summary

1. Score function

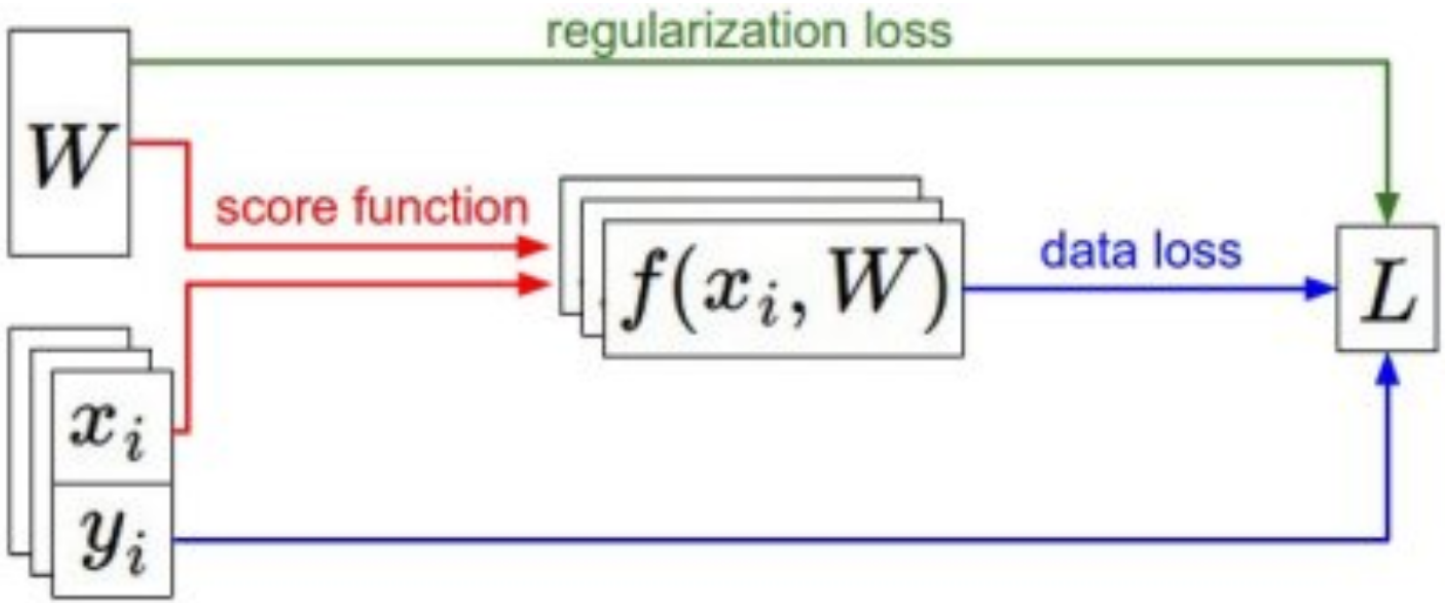
$$f(x_i, W, b) = Wx_i + b$$



2. Loss function

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda R(W)$$

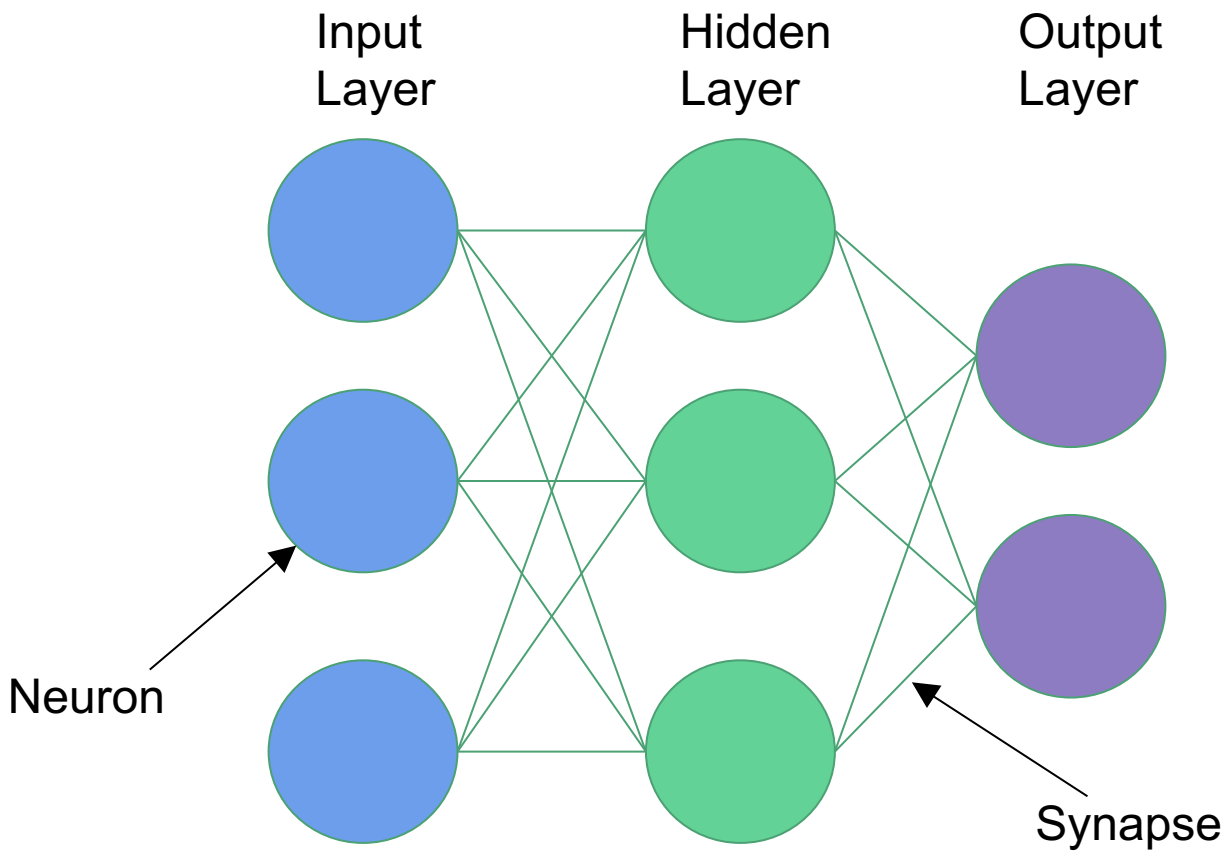
Summary



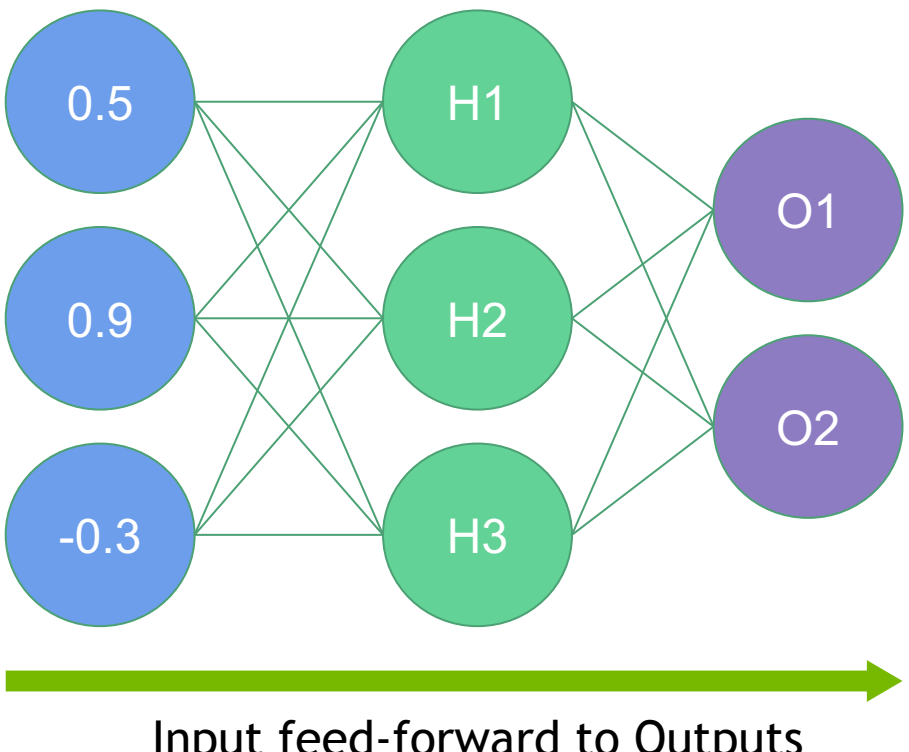
Neural Network Architecture and Optimization

Neural Network Architecture

Artificial Neural Network



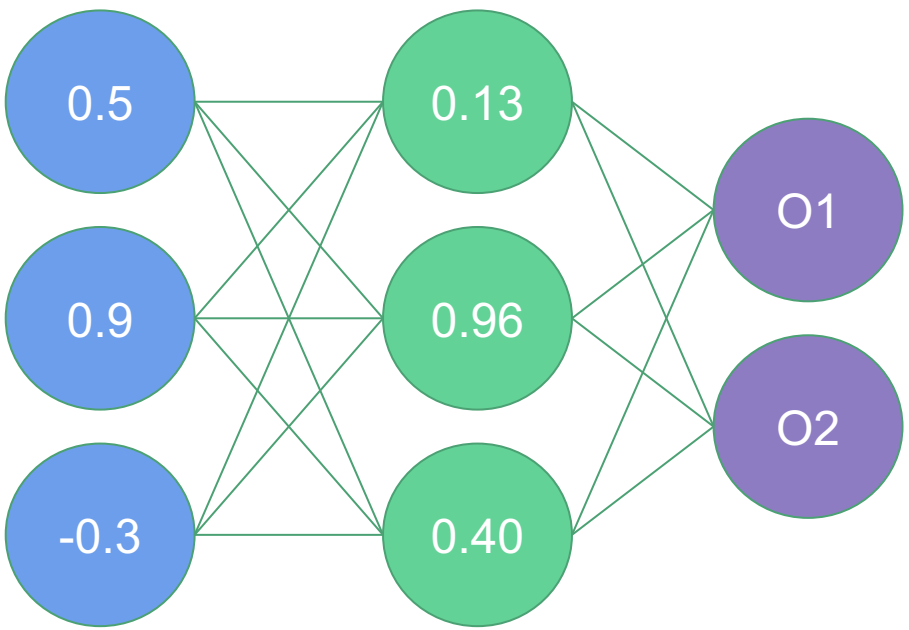
Inference



H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

Inference

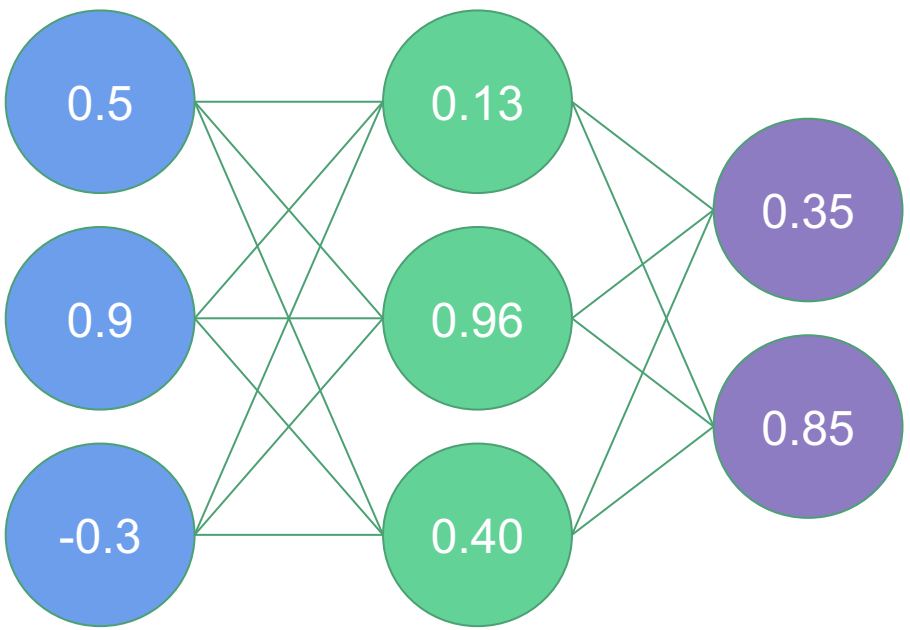


H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

H1 = S(0.5 * 1.0 + 0.9 * -2.0 + -0.3 * 2.0) = S(-1.9) = .13
H2 = S(0.5 * 2.0 + 0.9 * 1.0 + -0.3 * -4.0) = S(3.1) = .96
H3 = S(0.5 * 1.0 + 0.9 * -1.0 + -0.3 * 0.0) = S(-0.4) = .40

Inference



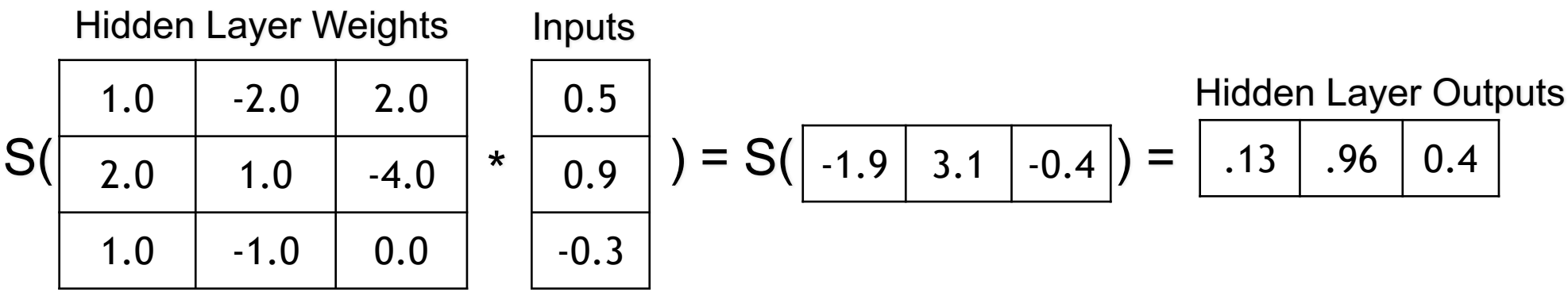
H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

O1 = S(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = S(-.63) = .35
O2 = S(.13 * 0.0 + .96 * 1.0 + .40 * 2.0) = S(1.76) = .85

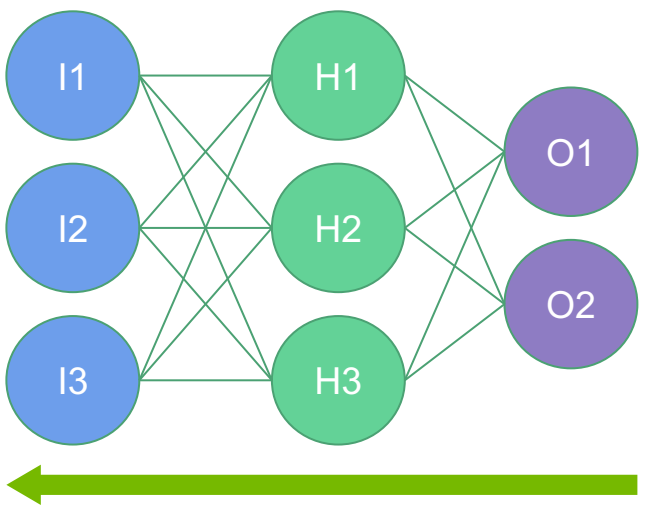
Matrix Formulation

H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)



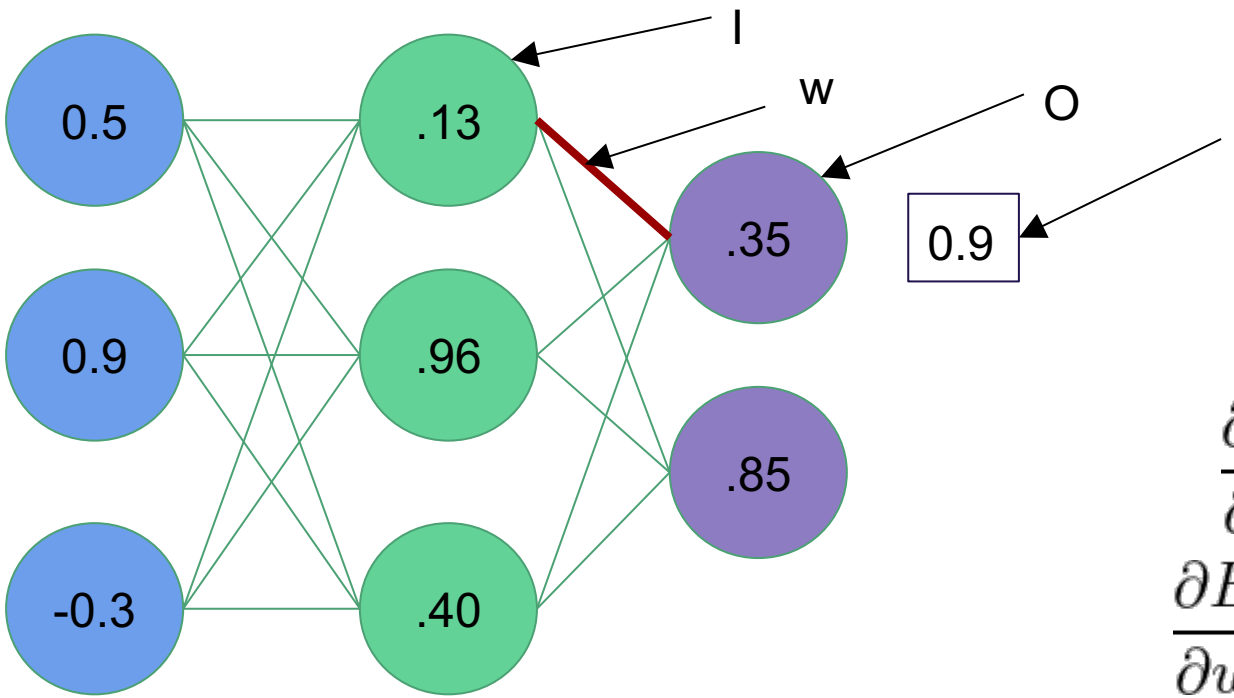
Backpropagation

- Problem: Which weights should be updated and by how much?
  - Insight: Use the derivative of the error with respect to weight to assign "blame"



backward to input to update parameters with loss

Backpropagation Example



$$\frac{\partial E}{\partial w} = I \cdot (O - T) \cdot O \cdot (1 - O)$$

$$\frac{\partial E}{\partial w} = .13 \cdot (.35 - .9) \cdot .35 \cdot (1 - .35)$$

Gradient Descent

- Gradient Descent minimizes the neural network's error
  - At each time step the error of the network is calculated on the training data
  - Then the weights are modified to reduce the error
- Gradient Descent terminates when
  - The error is sufficiently small
  - The max number of time steps has been exceeded

Gradient Descent

Getting to a minimum

The derivative of a function in 1 dimension is:

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

In n dimensions, the **gradient** is the vector of (partial) derivatives.

Gradient Descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Dynamics of **Gradient Descent**

Intuition

pull some weights up and some down

SVM

$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta)\right] + \lambda \sum_k \sum_l W_{k,l}^2$$

Softmax

$$L = \frac{1}{N}\sum_i -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) + \lambda \sum_k \sum_l W_{k,l}^2$$

always pull the weights down

Gradient intuition

$$f(x,y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \qquad \frac{\partial f}{\partial y} = x$$

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

Example: x = 4, y = -3. => f(x,y) = -12

$$\frac{\partial f}{\partial x} = -3 \qquad \frac{\partial f}{\partial y} = 4$$

partial derivatives

$$\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}]$$

gradient

If I increase x by h, how would the input of f change?

Forward pass and backpropagation intuition

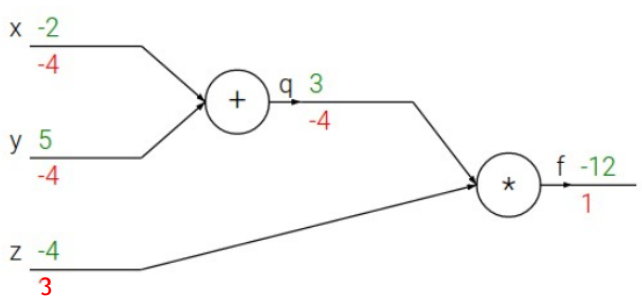
Compound expressions: $f(x, y, z) = (x + y)z$

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

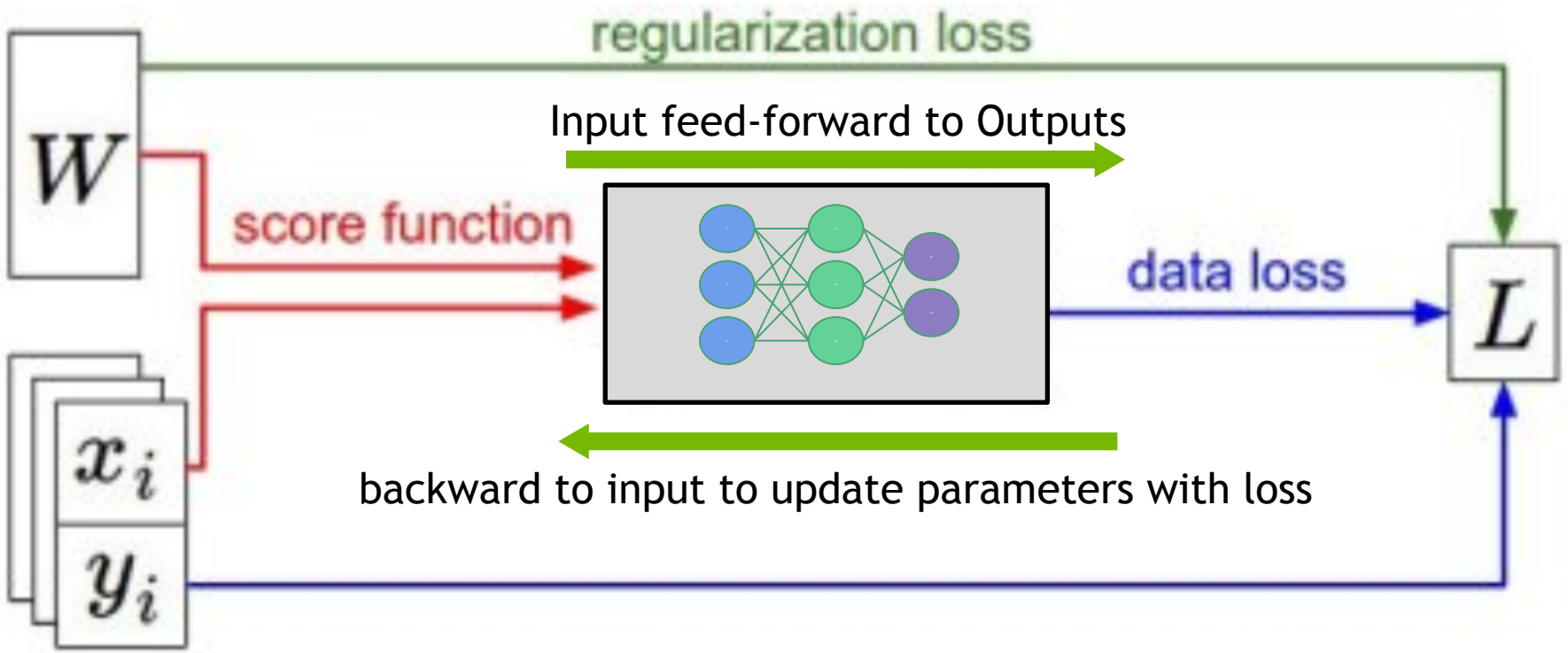
Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$




```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/fz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
# now backprop through q = x + y
dfdx = 1.0 * dfdq # dq/dx = 1. And the multiplication here is the chain rule!
dfdy = 1.0 * dfdq # dq/dy = 1
```

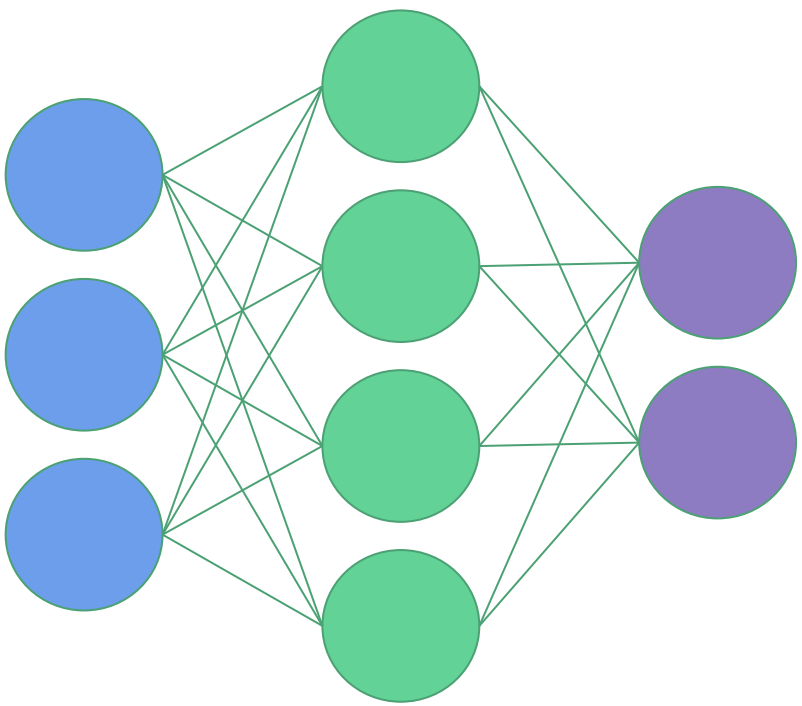
Summary


regularization loss
Input feed-forward to Outputs
W
score function
data loss
L
x_i
y_i
backward to input to update parameters with loss

Artificial Neural Network

ANN Architecture

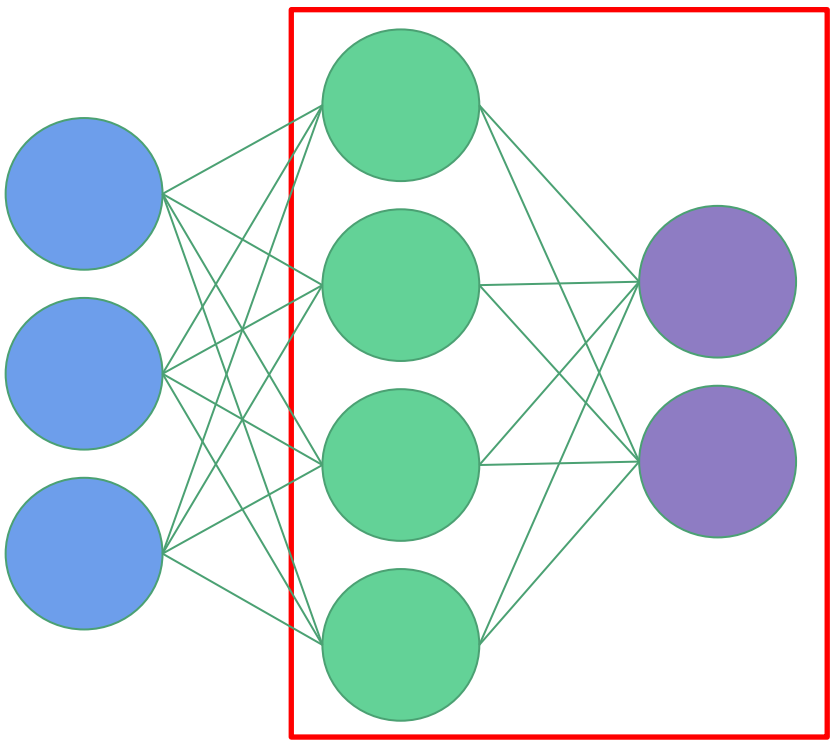
Fully Connected Layers



2-layer neural nets or
1-hidden neural nets

ANN Architecture

Trainable Parameters

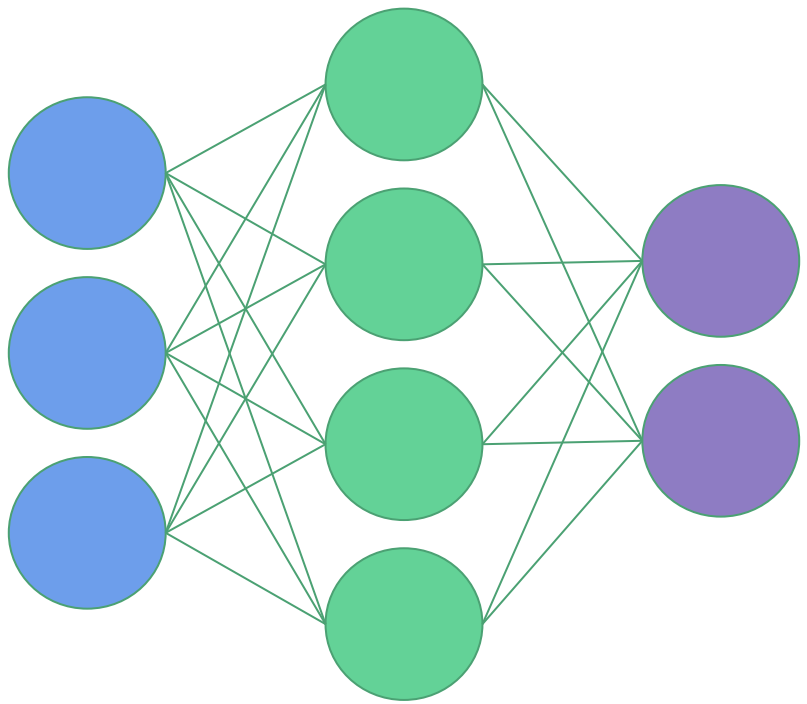


Number of Neurons: 4 + 2 = 6
Number of Weights: [4x3 + 2x4] = 20
Number of Parameters: 6 + 20 = 26

```
# foward-pass of a 3-layer neural network
f = lambda x: np.max(0,x)     # activation function (ReLU)
x = np.random.rand(3,1)       # random input vector of three numbers (3)
h1 = f(np.dot(W1, x) + b1)    # calculate hidden layer activations (4x1)
out = f(np.dot(W2, h1) + b2)  # output neuron (2x1)
```

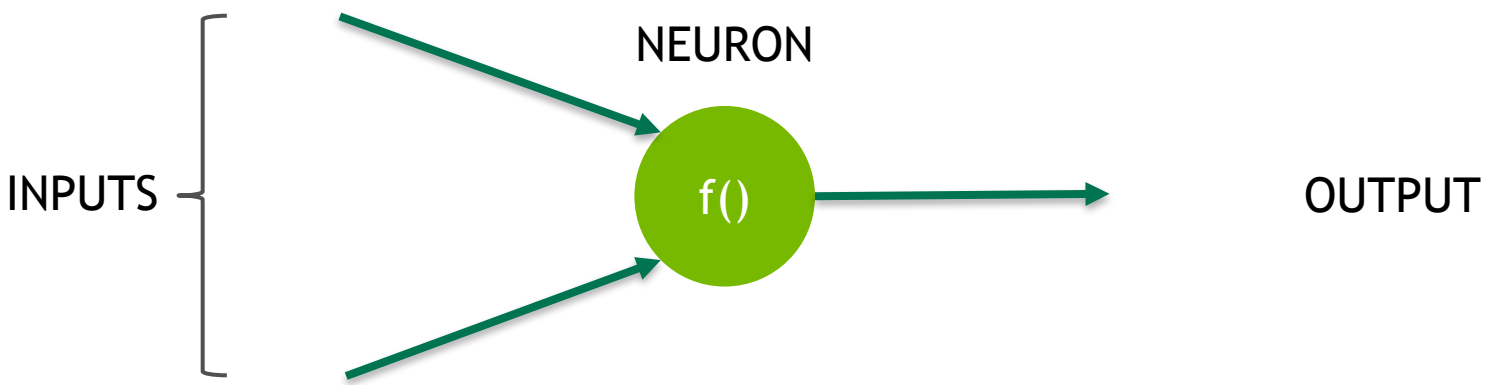
ANN Architecture

Trainable Parameters

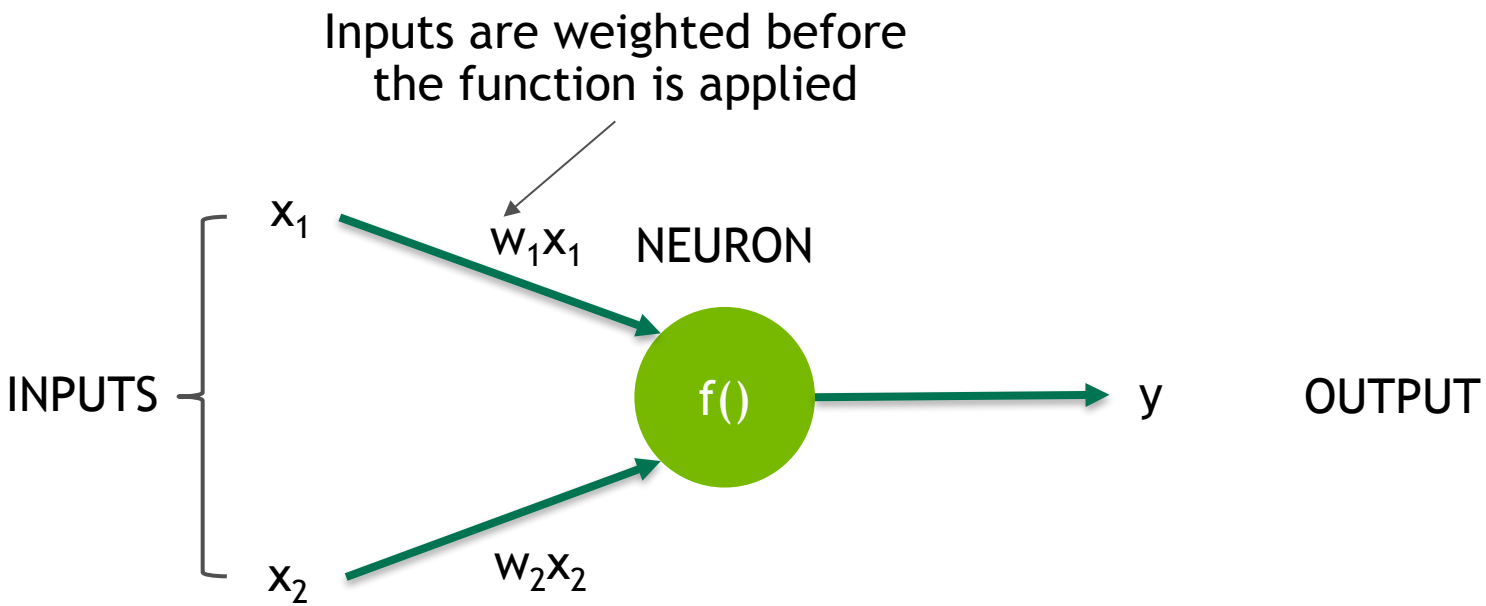


Modern CNNs: ~10 million neurons
Human visual cortex: ~5 billion neurons

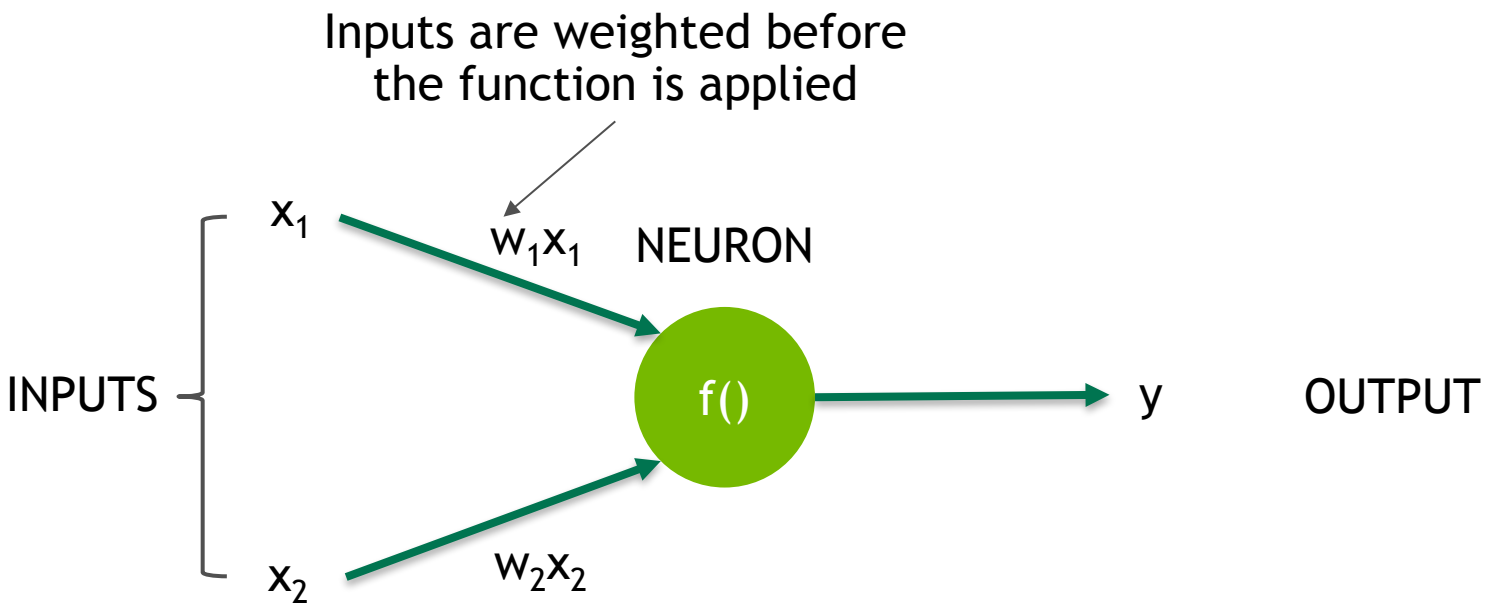
The artificial neuron



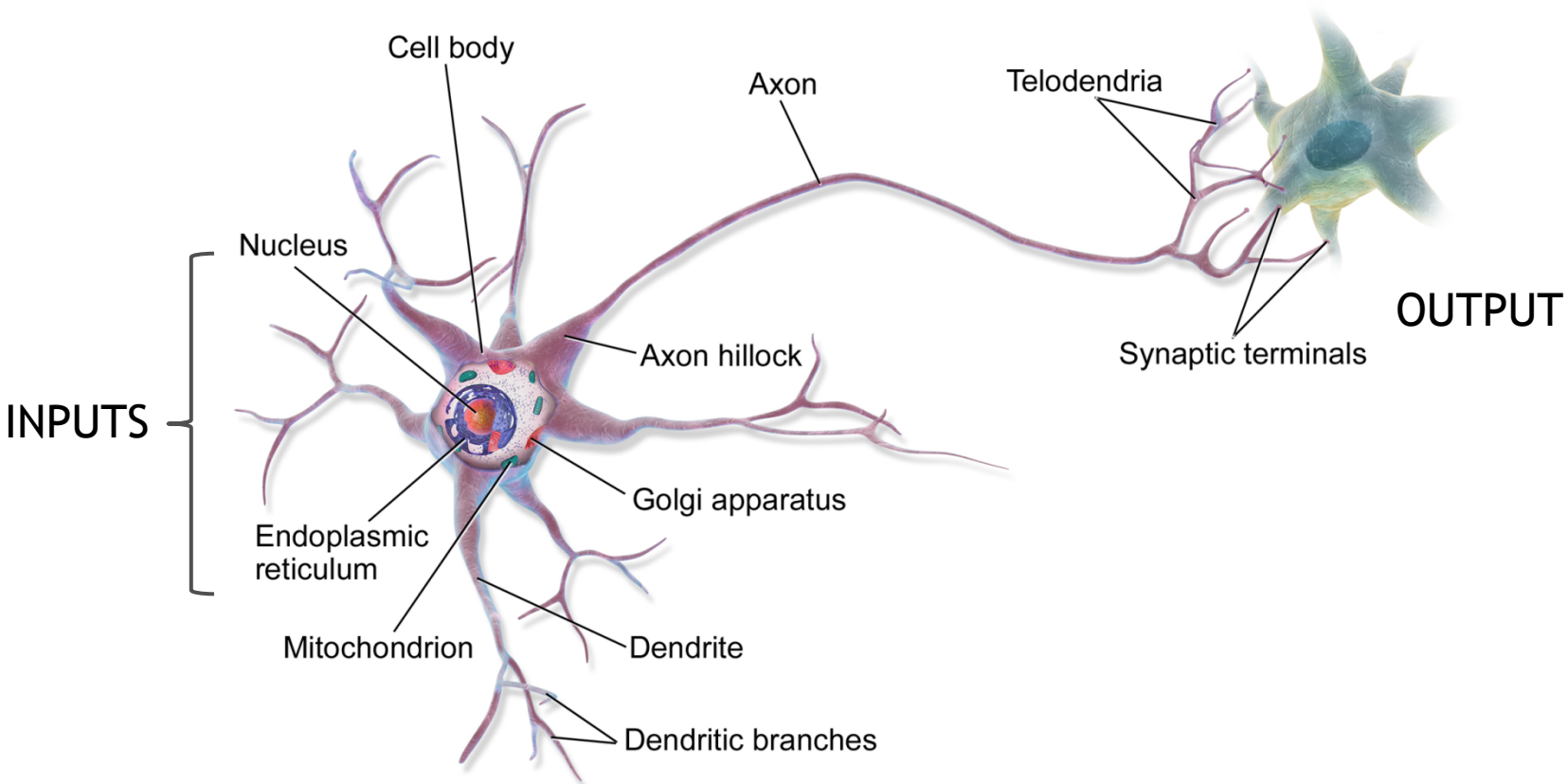
The artificial neuron



The artificial neuron

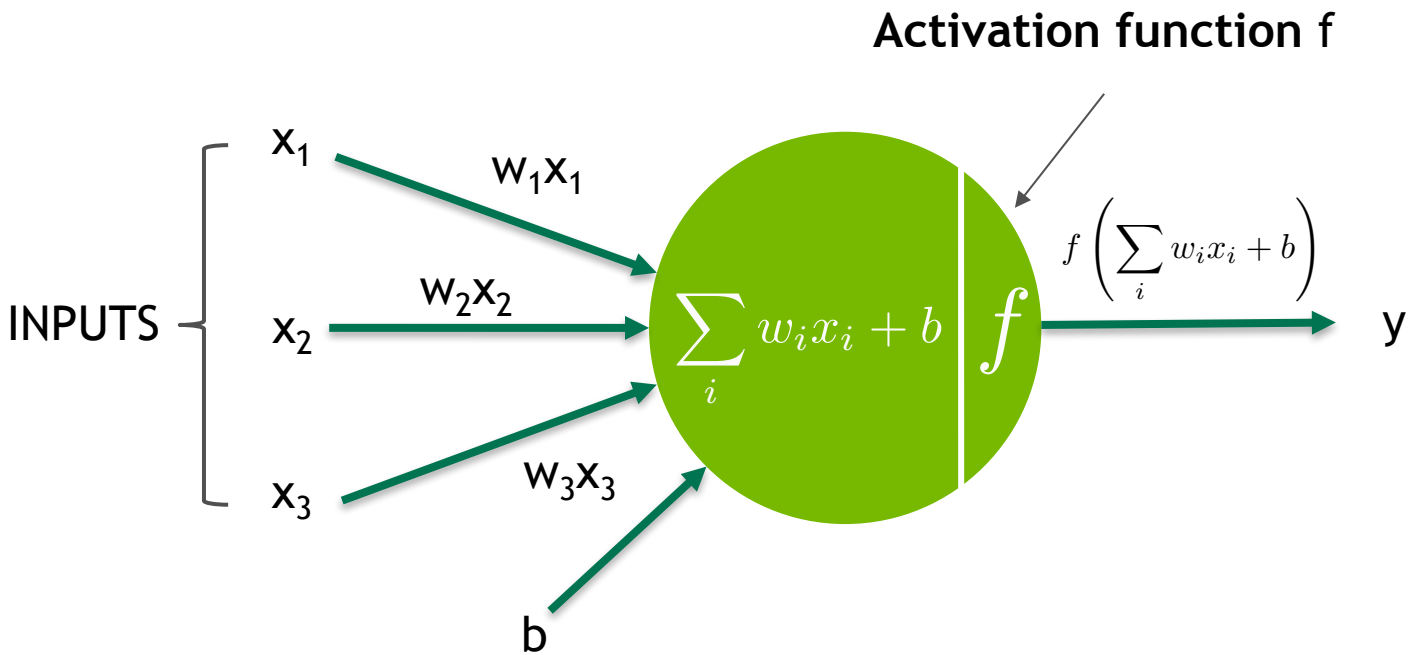


Biological Inspiration



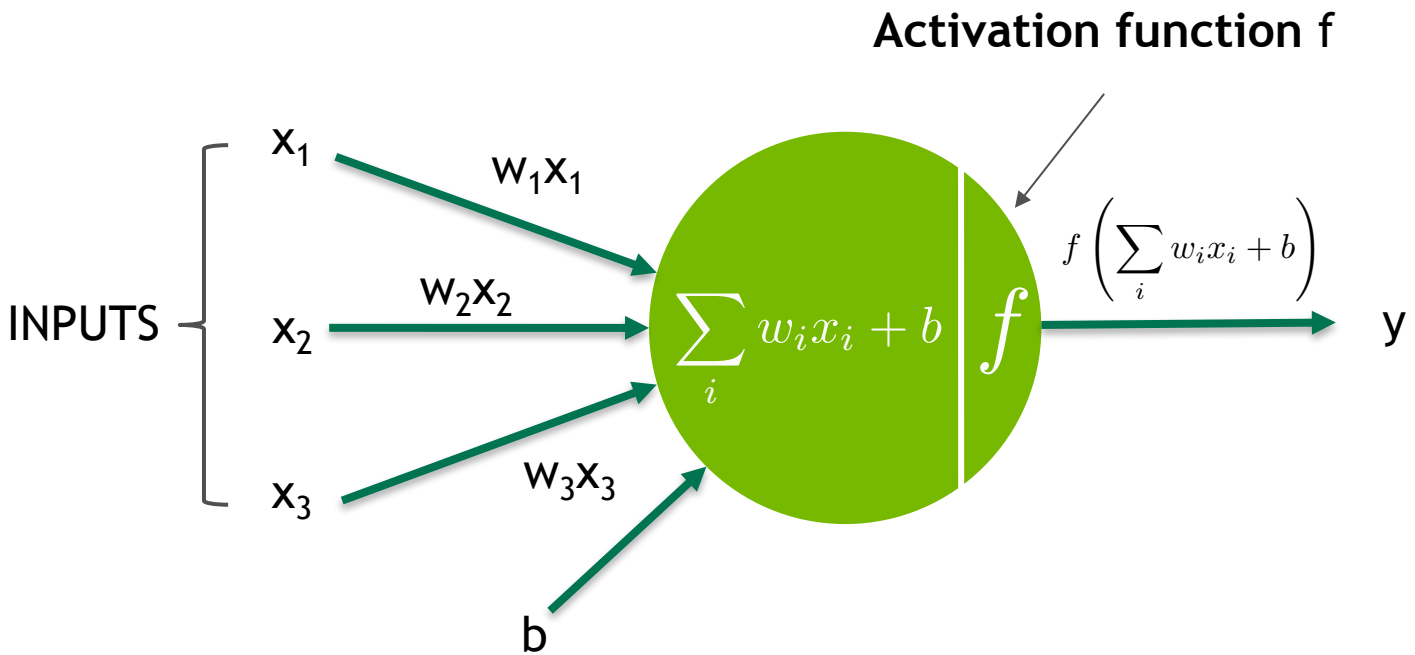
The artificial neuron

More detailed view



The artificial neuron

More detailed view

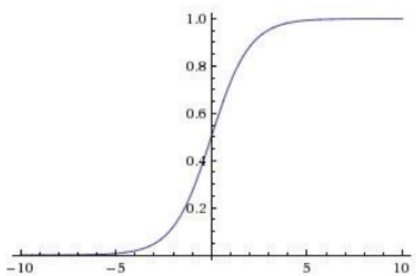


Example Python implementation:

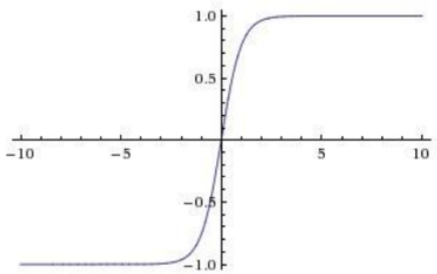
```python
class Neuron:

    def neuron_fire(self, inputs):
        """ assume inputs and weights are 1-D arrays and bias is a number """
        cell_body_sum = np.sum(inputs * self.weights) + self.bias
        activation = np.max(0,cell_body_sum) # ReLU activation function
        return activation
```

Activation functions

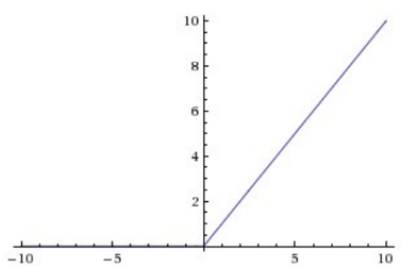
- Activation Functions are applied to the inputs at each neuron
- A range of non-linear functions have been used
- Non-linearity is important because a multi-layer ANN with linear activations would still only be able to learn composite linear functions



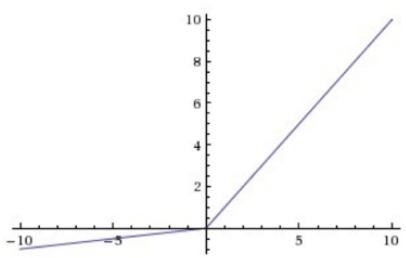
Sigmoid



tanh(x)



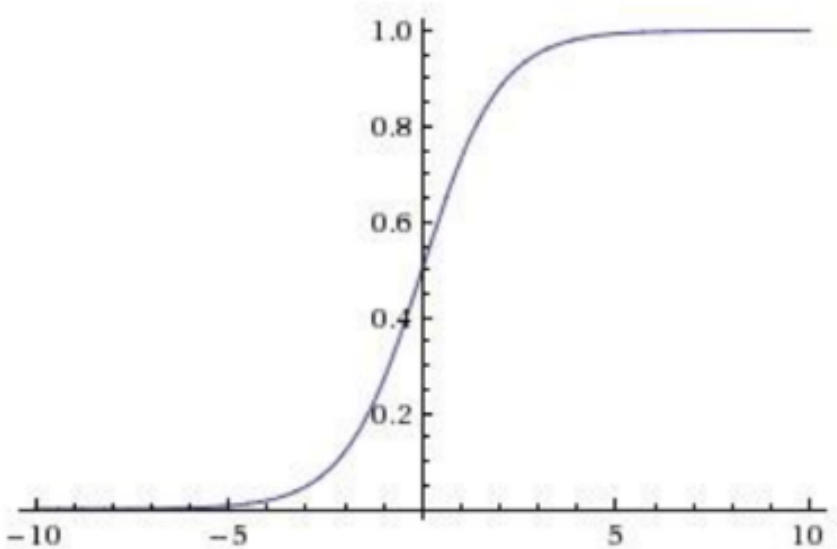
ReLU



Leaky ReLU

Activation functions

Sigmoid



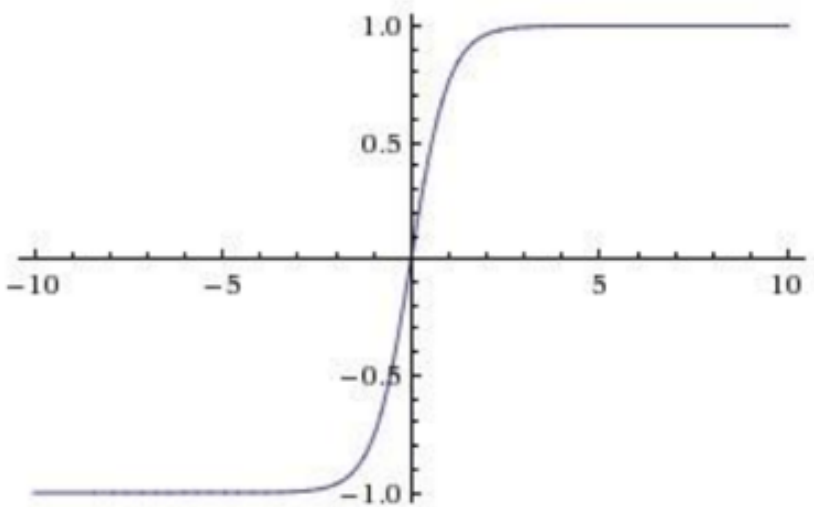
Sigmoid

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

- Squashes numbers to the range [0,1]
- Historically popular, but they have big practical problems that prevent network training

Activation functions

Hyperbolic Tangent



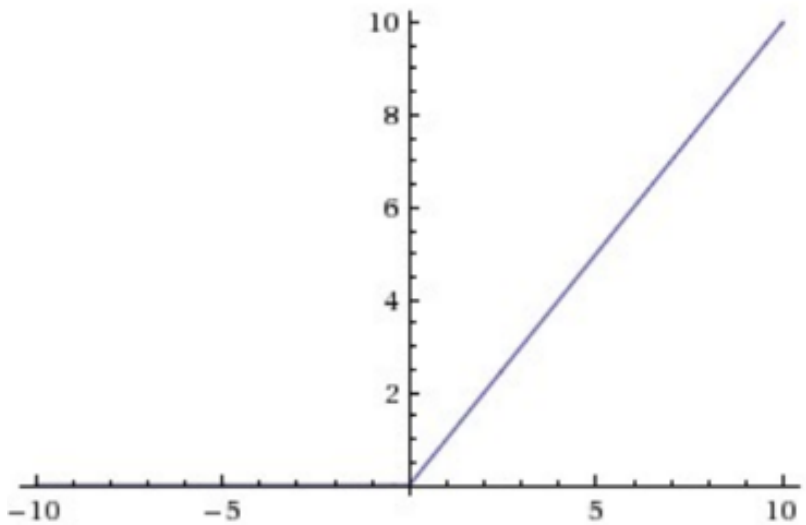
tanh(x)

- Squashes numbers to range [-1,1]
  - Zero-centered is good
- Still have practical issues

Activation functions

Rectified Linear Units

$$f(x) = max(0, x)$$

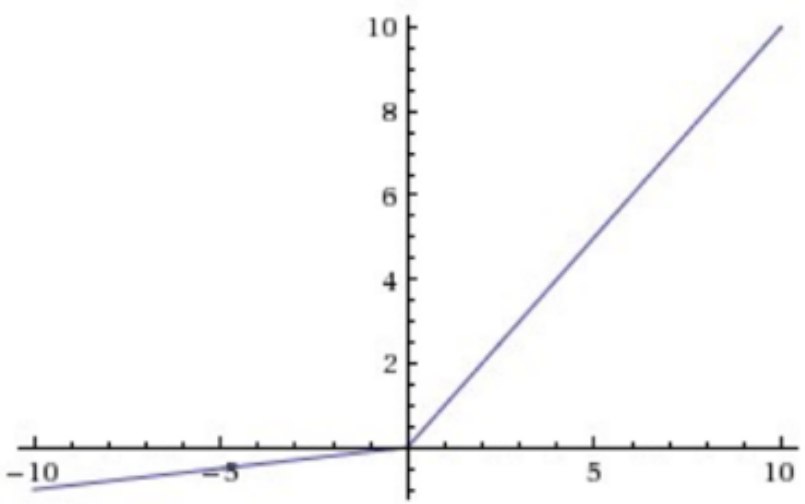


ReLU

- Most popular in current DL work
- Computationally efficient, ReLU neurons can be trained much faster
- Addresses the practical issues of sigmoid and tanh
- Neurons can “die” during training

Activation functions

Leaky ReLU



Leaky ReLU

- Computationally efficient
- Will not “die” like ReLU

Activation functions

In practice

Use ReLU

Try out Leaky ReLU

You can try out tanh but it probably won't work as well as ReLU

Never use sigmoid

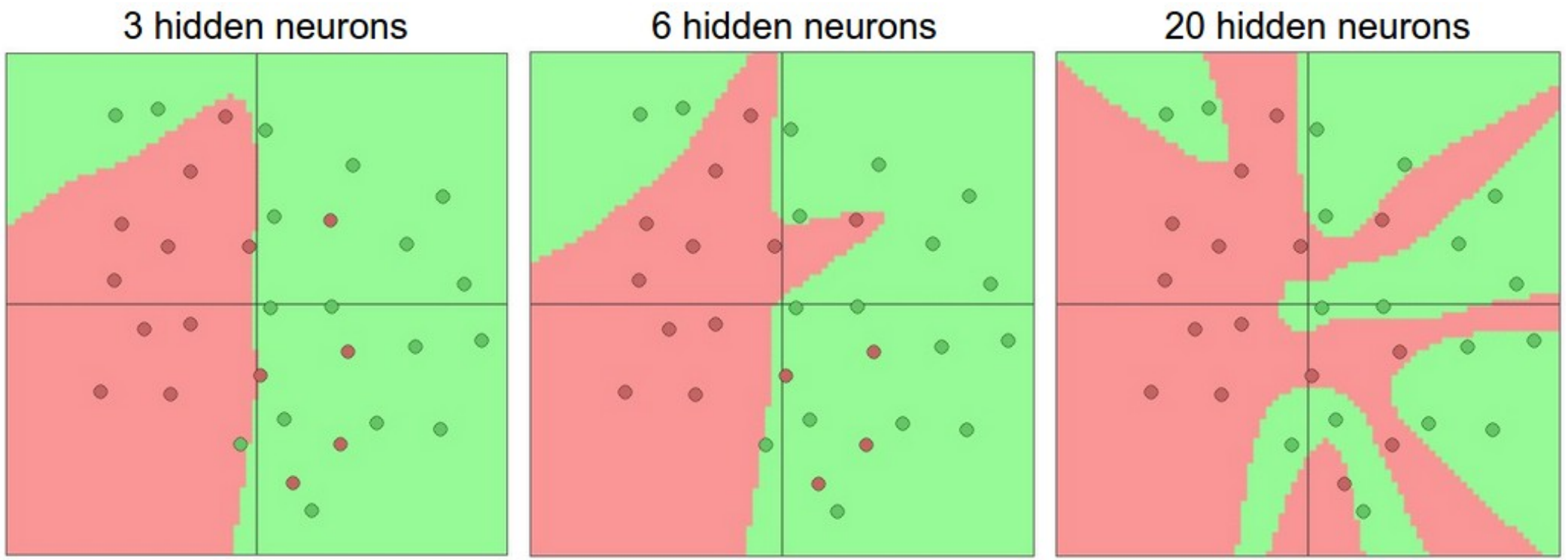
Training Neural Networks

- Procedure for training Neural Networks
  - Perform inference on the training set
  - Calculate the error between the predictions and actual labels of the training set
  - Determine the contribution of each Neuron to the error
  - Modify the weights of the Neural Network to minimize the error

- Error contributions are calculated using Backpropagation
- Error minimization is achieved with Gradient Descent

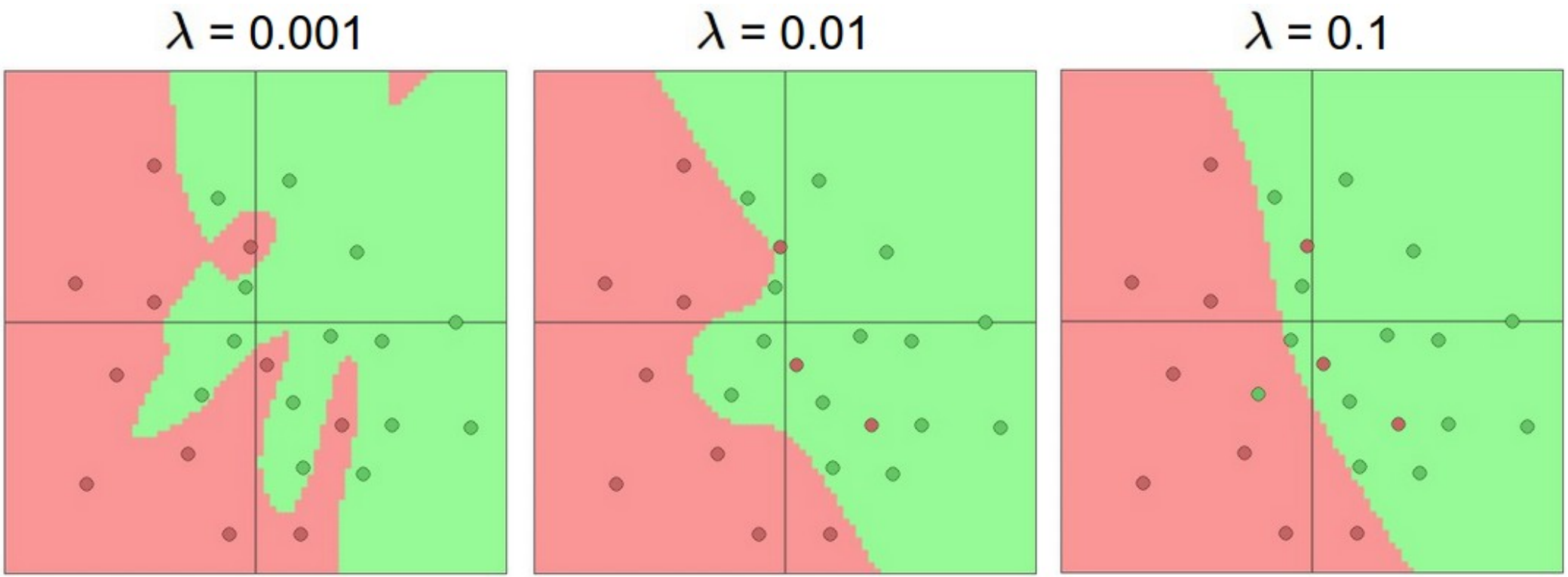
Setting the number or layers and their size

Overfitting, generalization and outliers

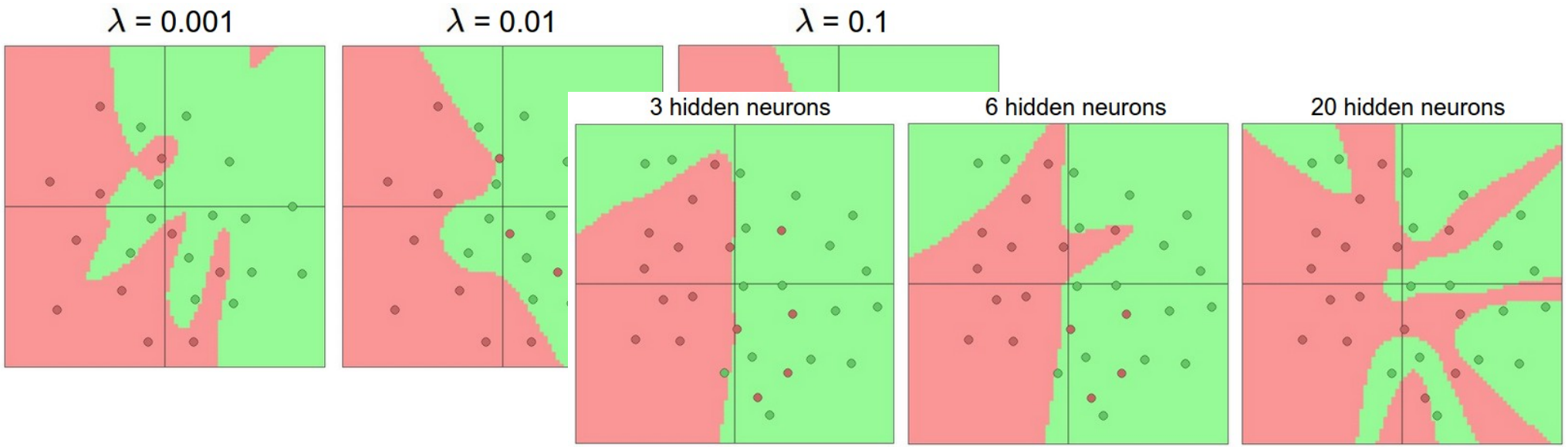


Effect of regularization

Overfitting, generalization and outliers



Network size and overfitting



Takeaway:

- Don’t be afraid of overfitting: use a network as big as you can train
- Use regularization to control overfitting

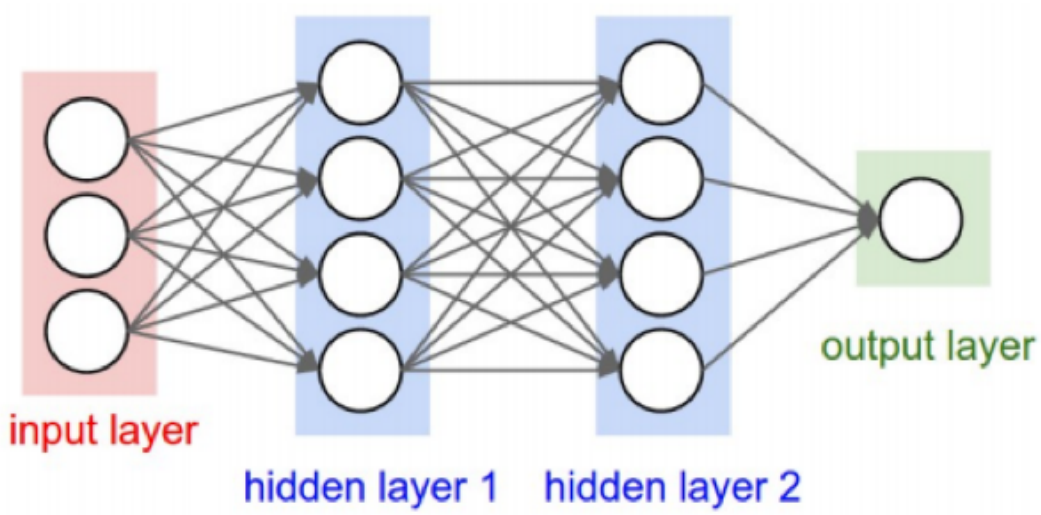
Vectorized feed-forward computation

```
1  f = lambda x: np.max(0,x)
2
3  class Neuron:
4
5      def neuron_fire(self, inputs):
6          """ assume inputs and weights are 1-D arrays and bias is a number """
7          cell_body_sum = np.dot(self.weights, inputs) + self.bias
8          activation = f(cell_body_sum) # ReLU activation function
9          return activation
```

Feed forward between two fully connected layers as a matrix-vector multiplication (followed by activation)

This is very good for GPU acceleration

Example feed-forward computation of network




```
# forward-pass of a 3-layer neural network
f = lambda x: np.max(0,x)          # activation function (ReLU)
x = np.random.rand(3,1)            # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1)         # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2)        # calculate second hidden layer activations (4x1)
out = f(np.dot(W3, h2) + b3)       # output neuron (1x1)
```

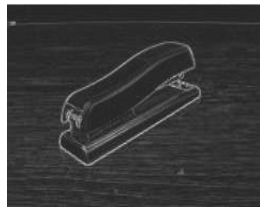
Conclusion

ANNs in Deep Learning

Traditional Machine Perception – hand tuned features

Raw data

Feature extraction



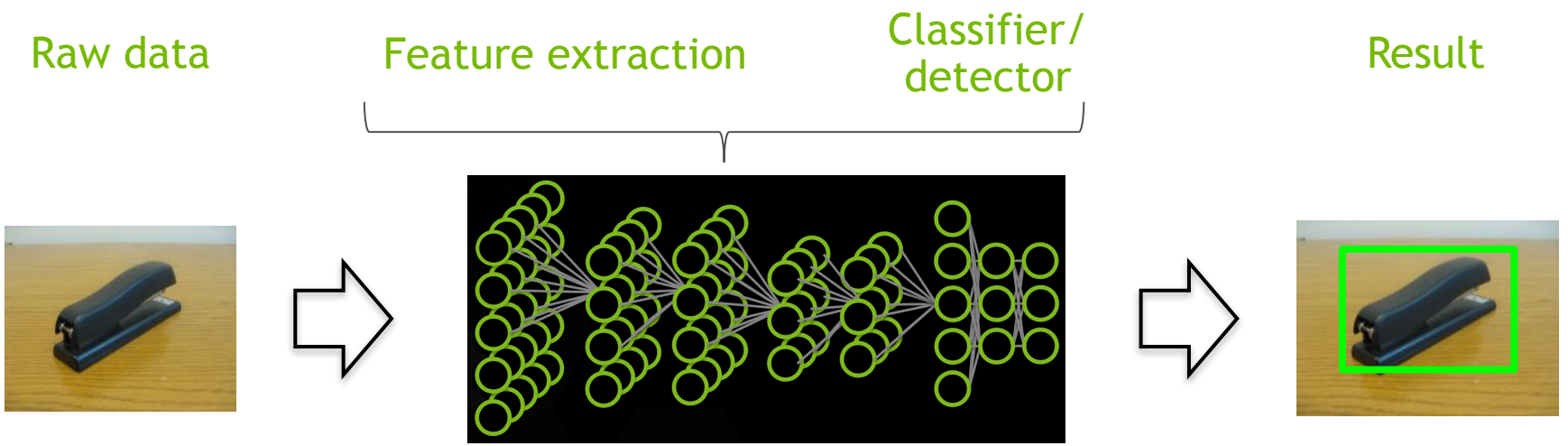
Classifier/ detector

e.g. SVM

Result

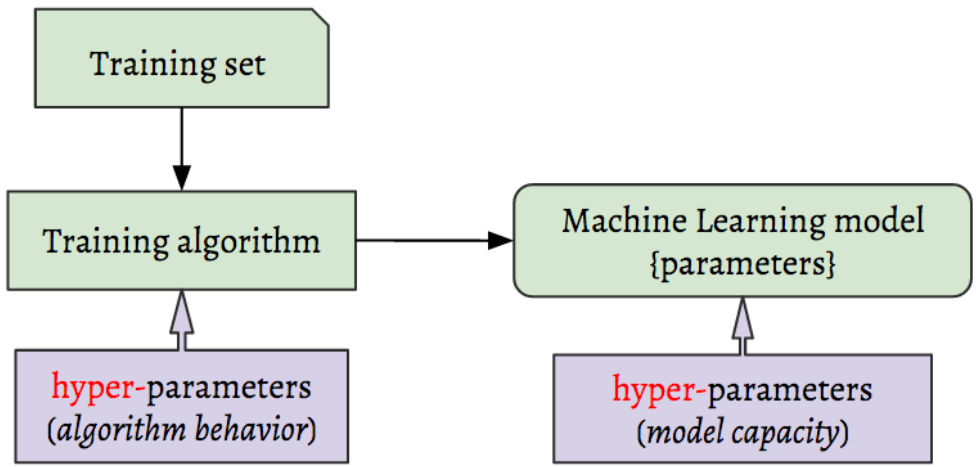
ANNs in Deep Learning

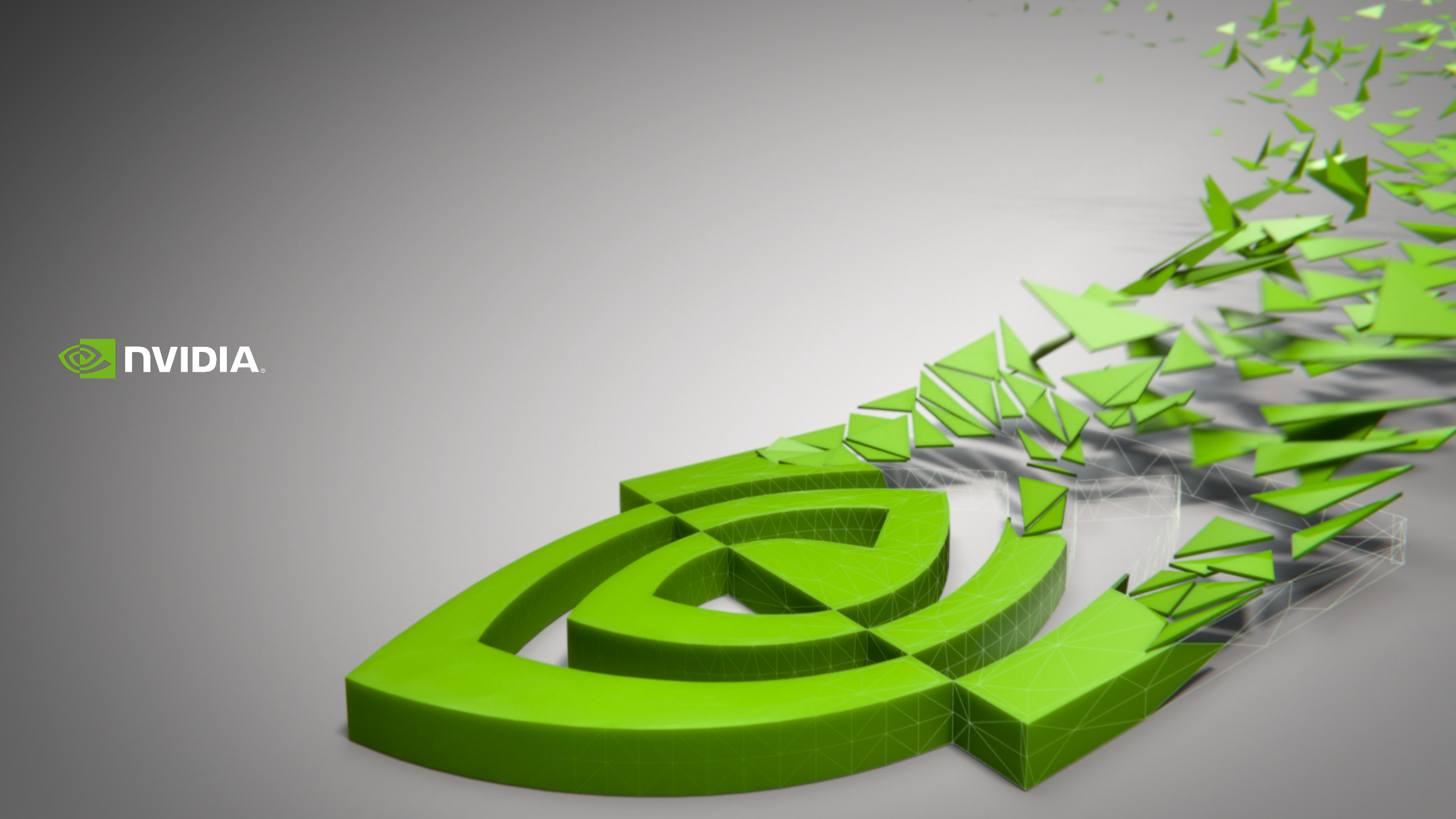
Deep Learning Machine Perception



Hyperparameter

You will see every point saying **“it depends”**




NVIDIA