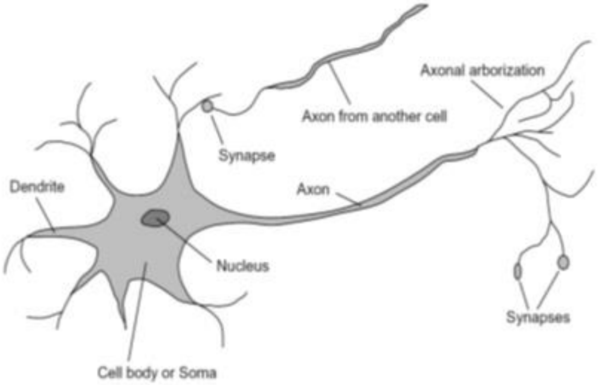
NEURAL NETWORK

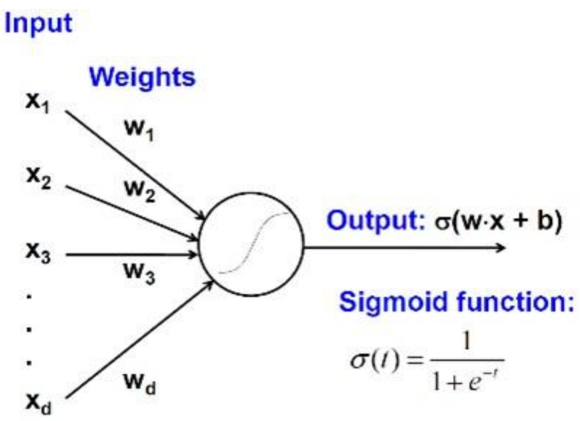
Neural Networks

- **Neural networks** are a set of algorithms, modelled loosely after the human brain, that are designed to recognize patterns.
- Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules.

Neural Networks



A biological neuron



An artificial neuron

Neural Networks

- Almost all resources on NN starts with the same example: Recognize handwritten digits
- TASK:
- 60000 digit images

as training set data.

- 10000 digit images

as test set data.

- Each image is 28x28

resolution.

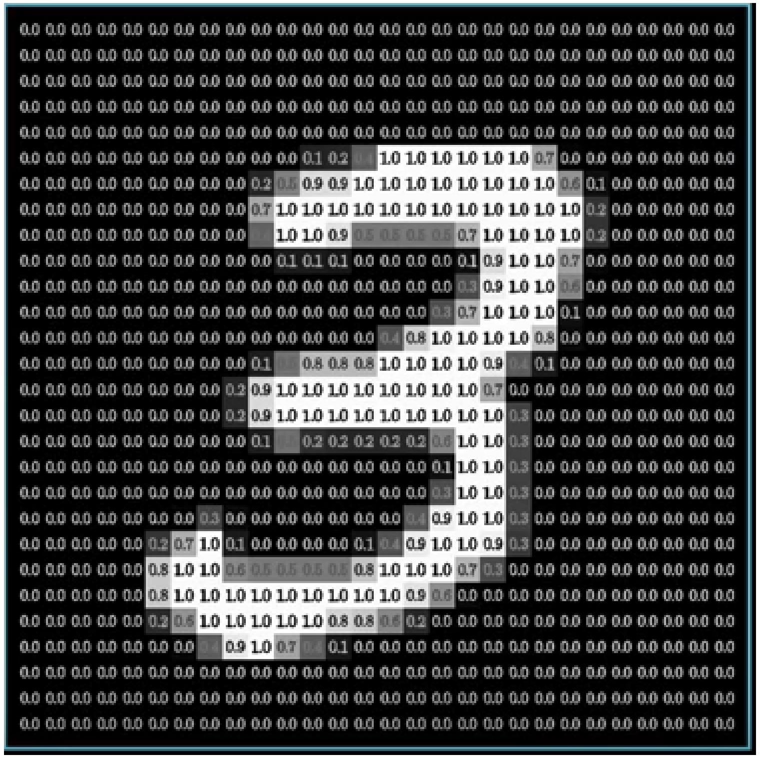
- Images are gray scale

images (8-bit, single channel)

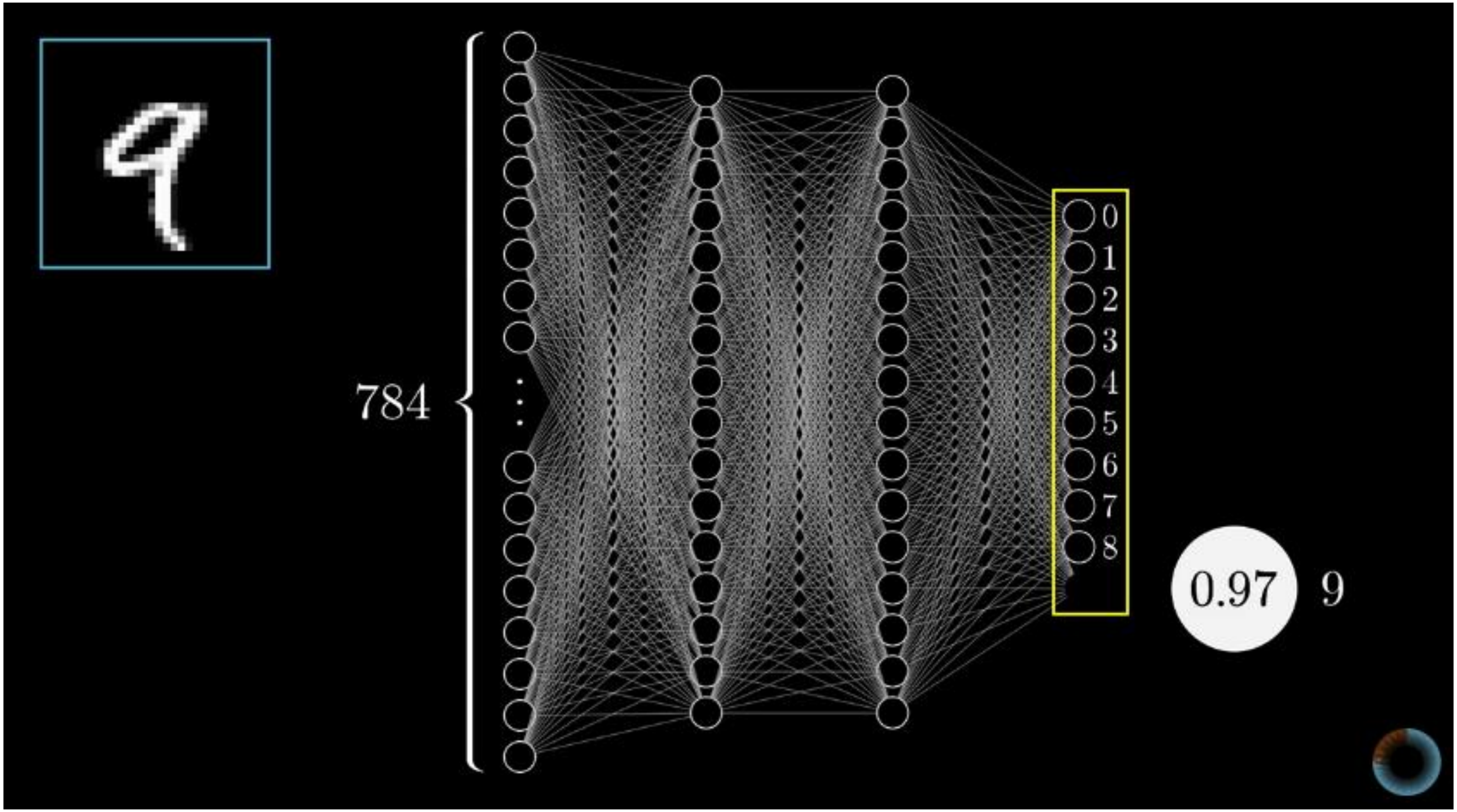
- What would be the accuracy of

recognizing the digits?

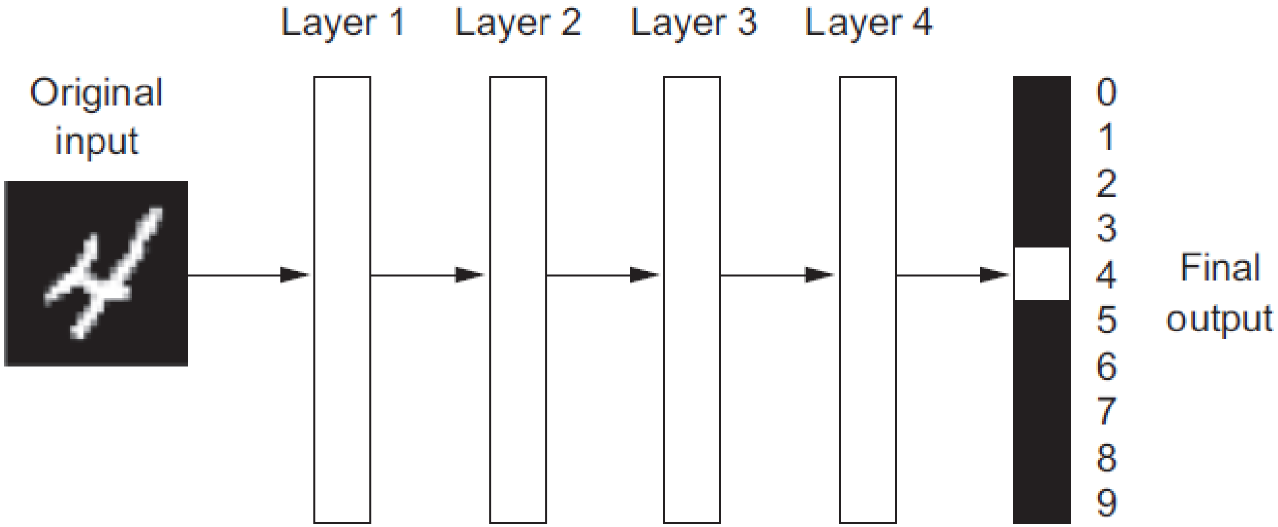
Neural Networks



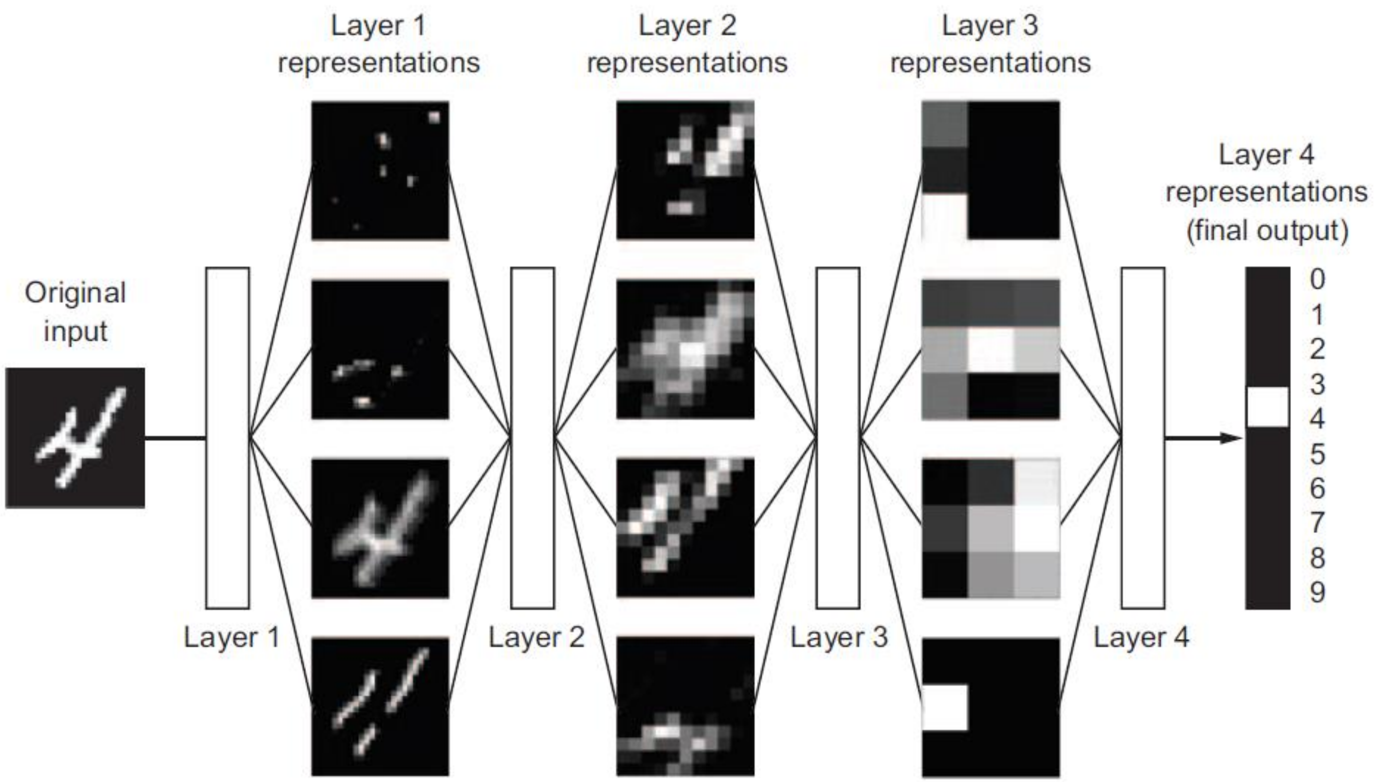
Neural Networks



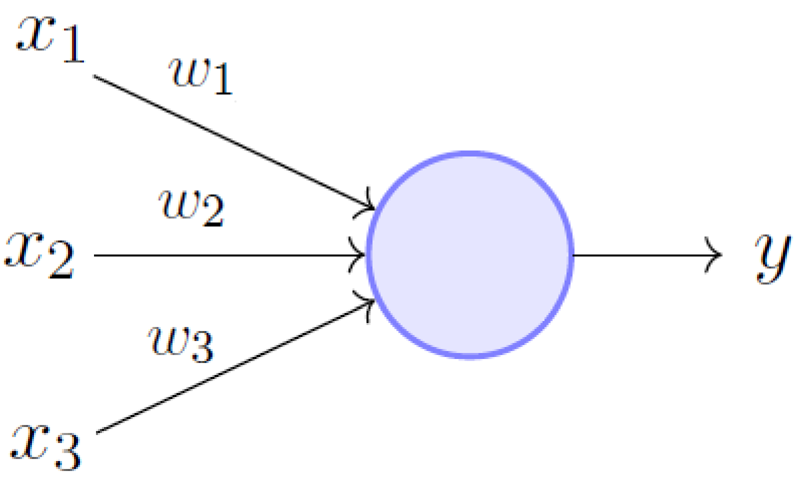
Neural Networks



Neural Networks



The building block: Perceptron



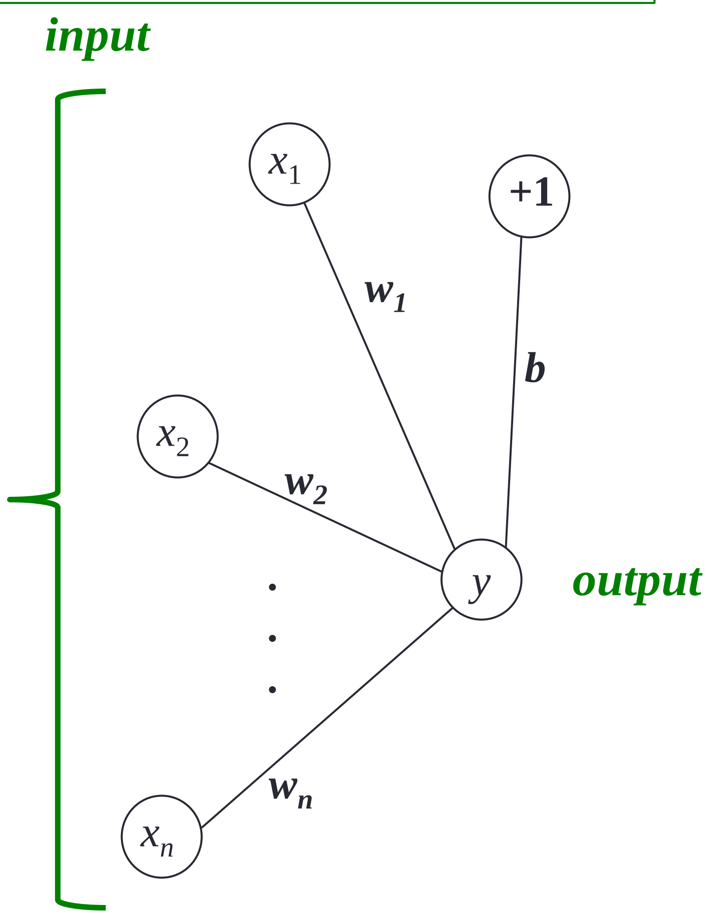
Perceptron Model (Minsky-Papert in 1969)

$$\text{output} = \begin{cases} 0 & \text{if} \quad \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if} \quad \sum_j w_j x_j > \text{threshold} \end{cases}$$

Perceptrons as simplified "neurons"

b is called the "**bias**"

$-b$ is called the "**threshold**"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > threshold$$
$$then\ y = 1,\ else\ y = 0$$

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > -b$$
$$then y = 1, else y = 0$$

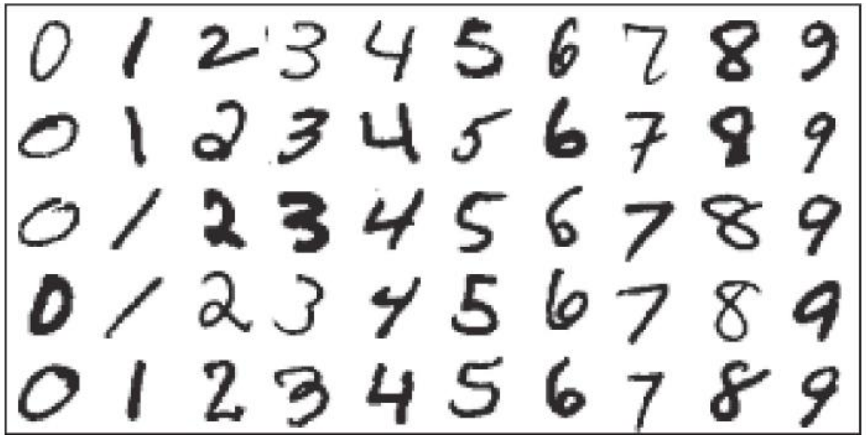
$$If\ b + w_1x_1 + w_2x_2 + \ldots + w_nx_n > 0$$
$$then y = 1, else y = 0$$

$$a(z) = \begin{cases} 1 \text{ if } z > 0 \\ 0 \text{ otherwise} \end{cases}$$

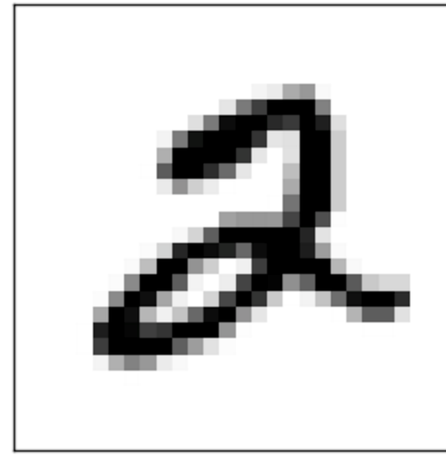
$$y = a(\mathbf{w} \cdot \mathbf{x} + b)$$

a is called an "activation function"

Recognizing Handwritten Digits



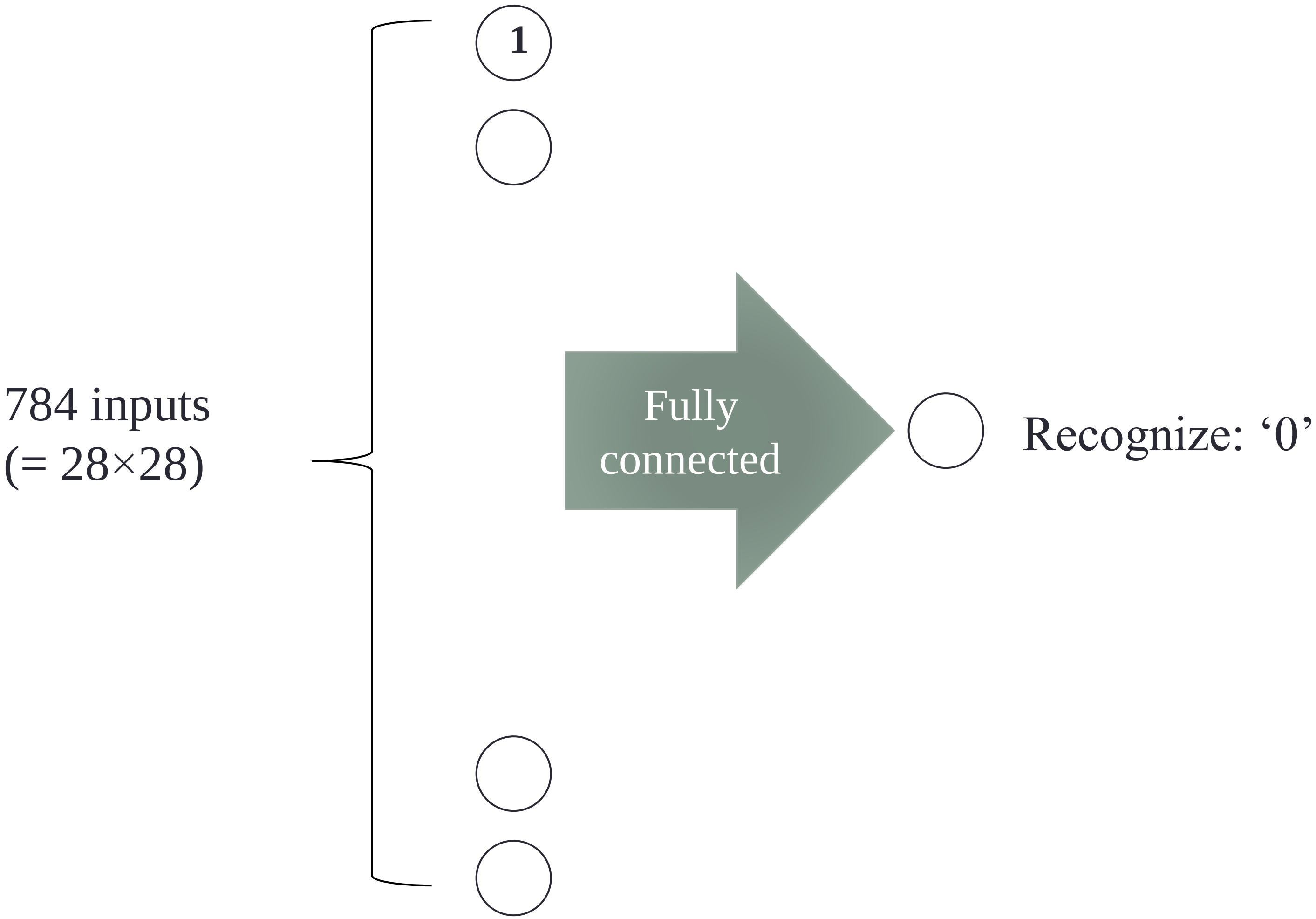
28 pixels



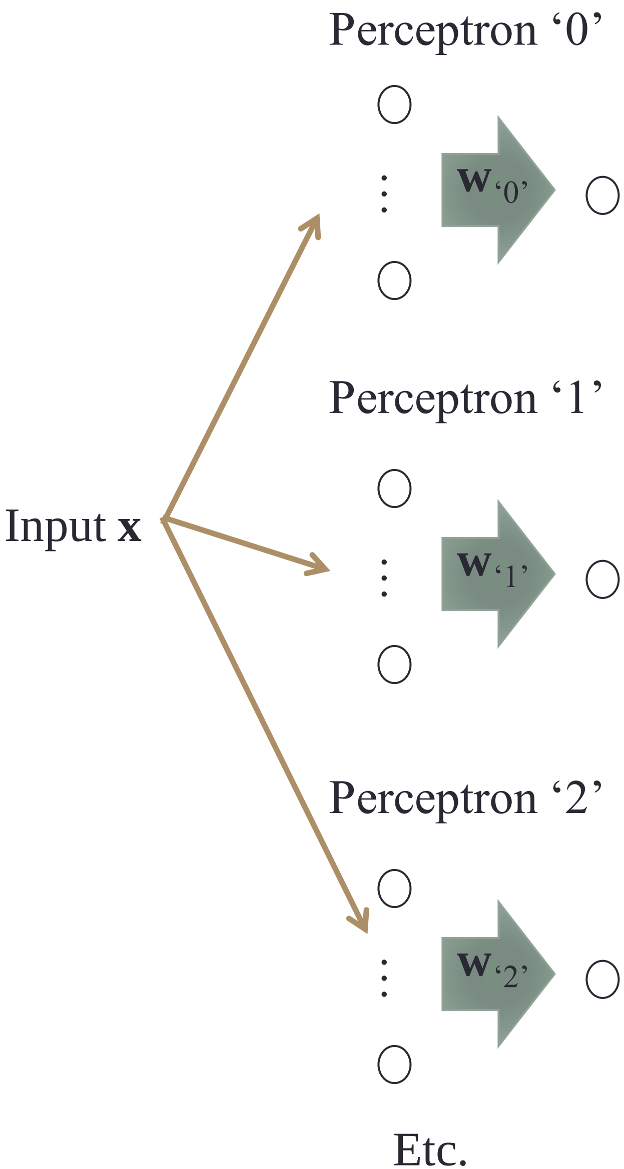
Label: “2”

28 pixels

Architecture for handwritten digits classification: 10 individual perceptrons



Processing an input



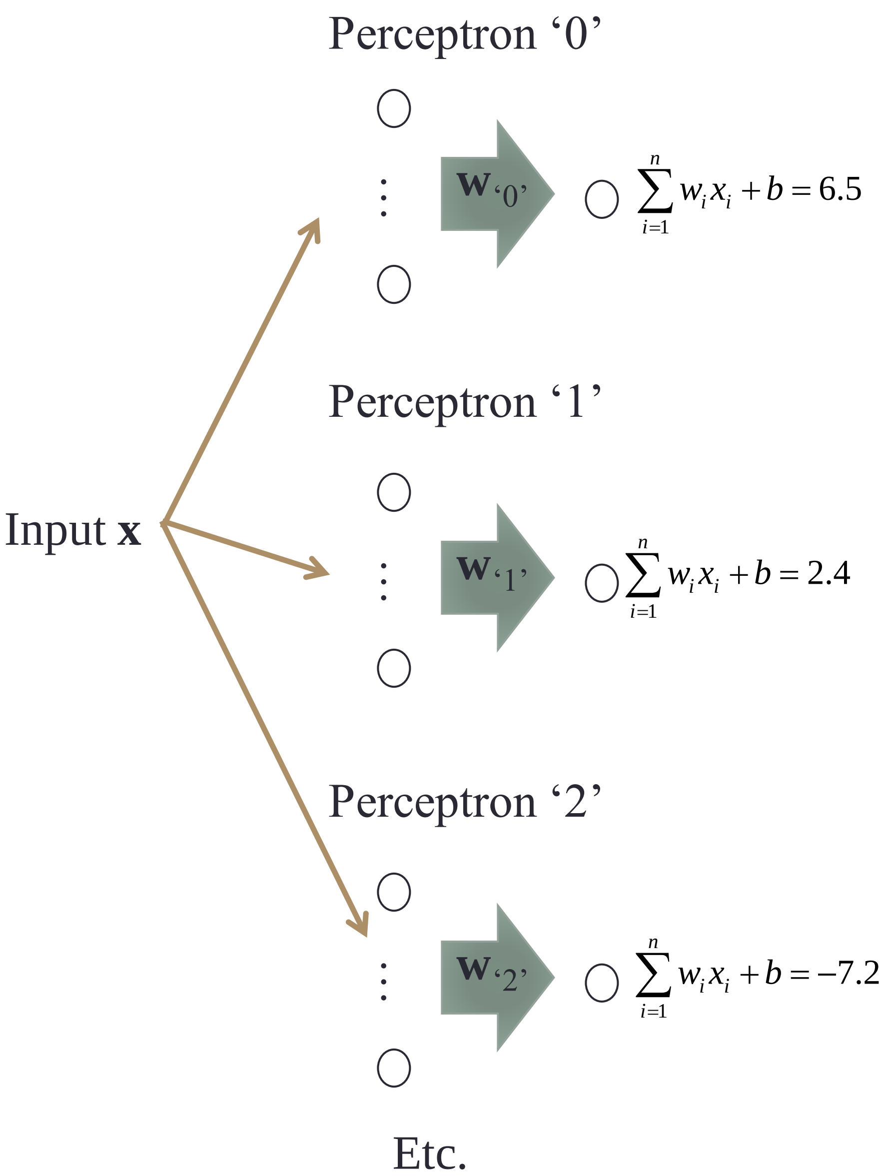
For each perceptron, compute

$$\sum_{i=0}^{n} w_i x_i$$

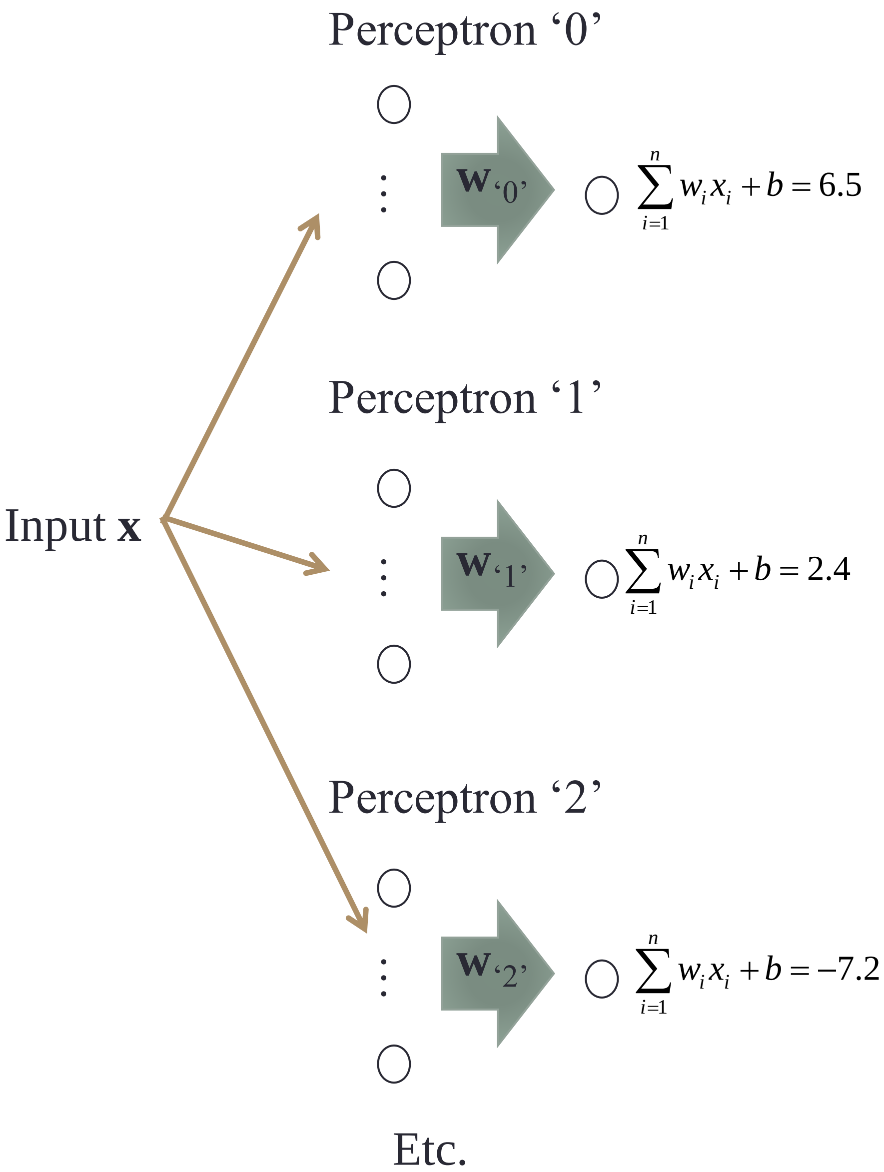
Processing an input

For each perceptron, compute

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



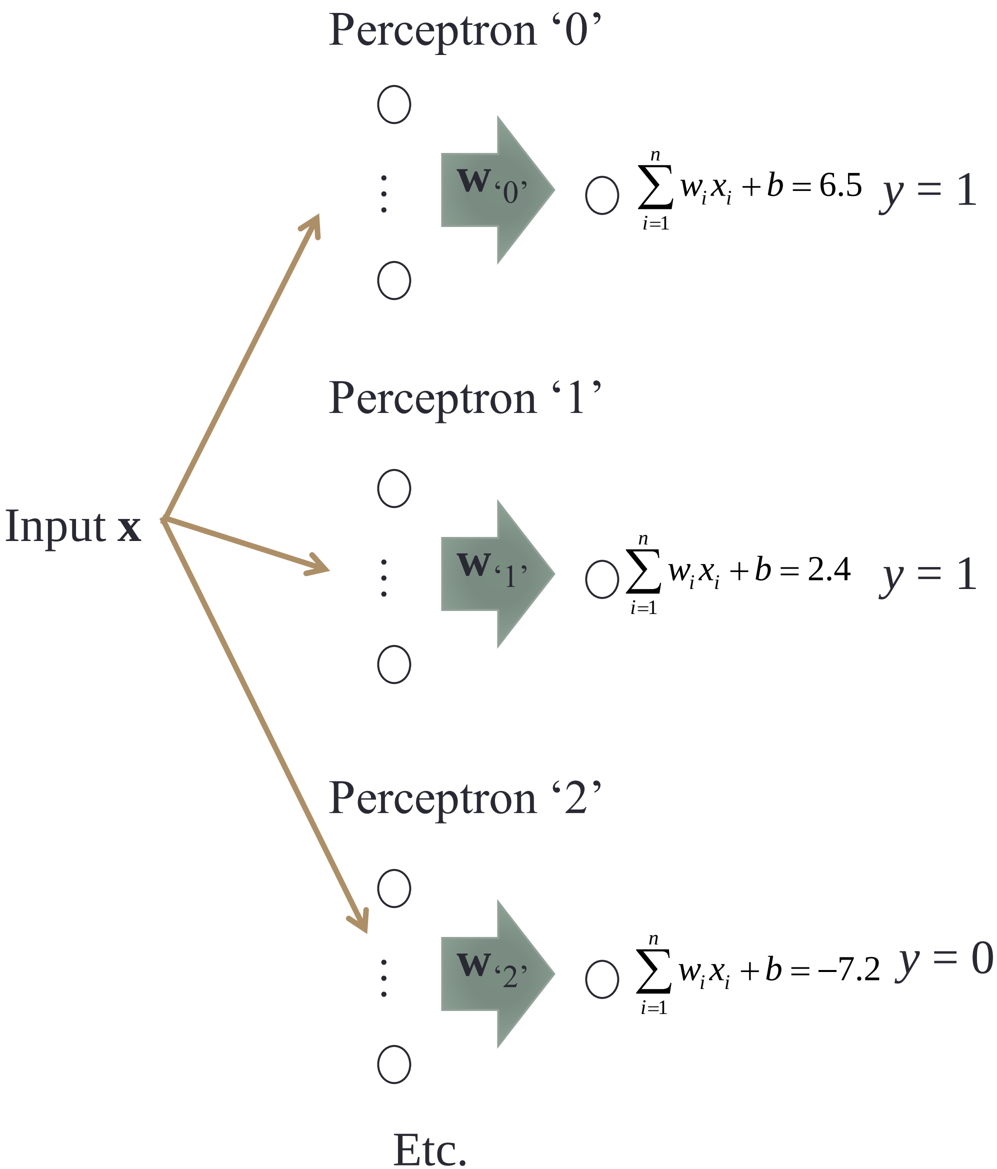
Computing outputs



For each perceptron, if

$$\sum_{i=1}^{n} w_i x_i + b > 0$$

then output $y = 1$; otherwise $y = 0$.



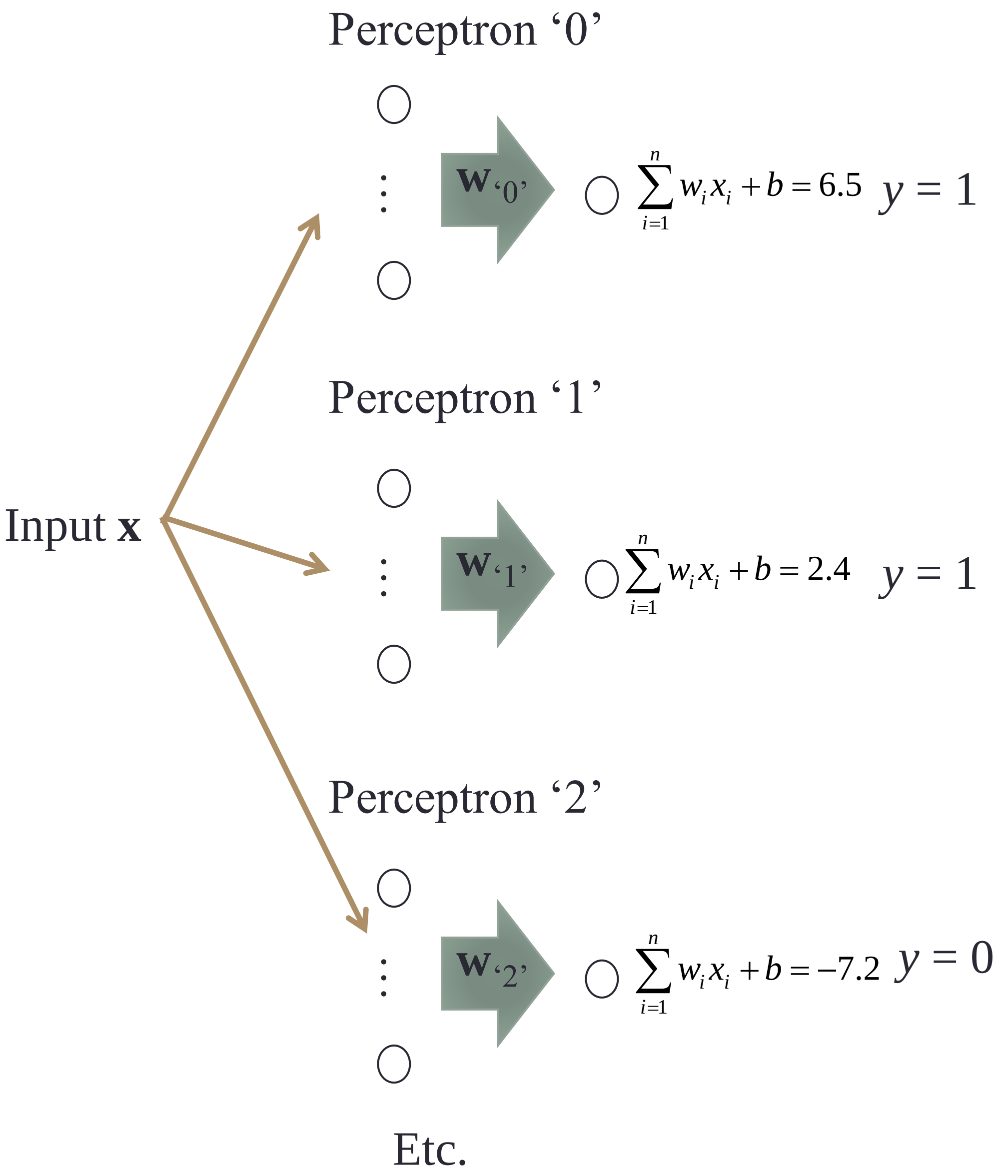
Computing outputs

For each perceptron, if

$$\sum_{i=1}^{n} w_i x_i + b > 0$$

then output $y = 1$; otherwise $y = 0$.

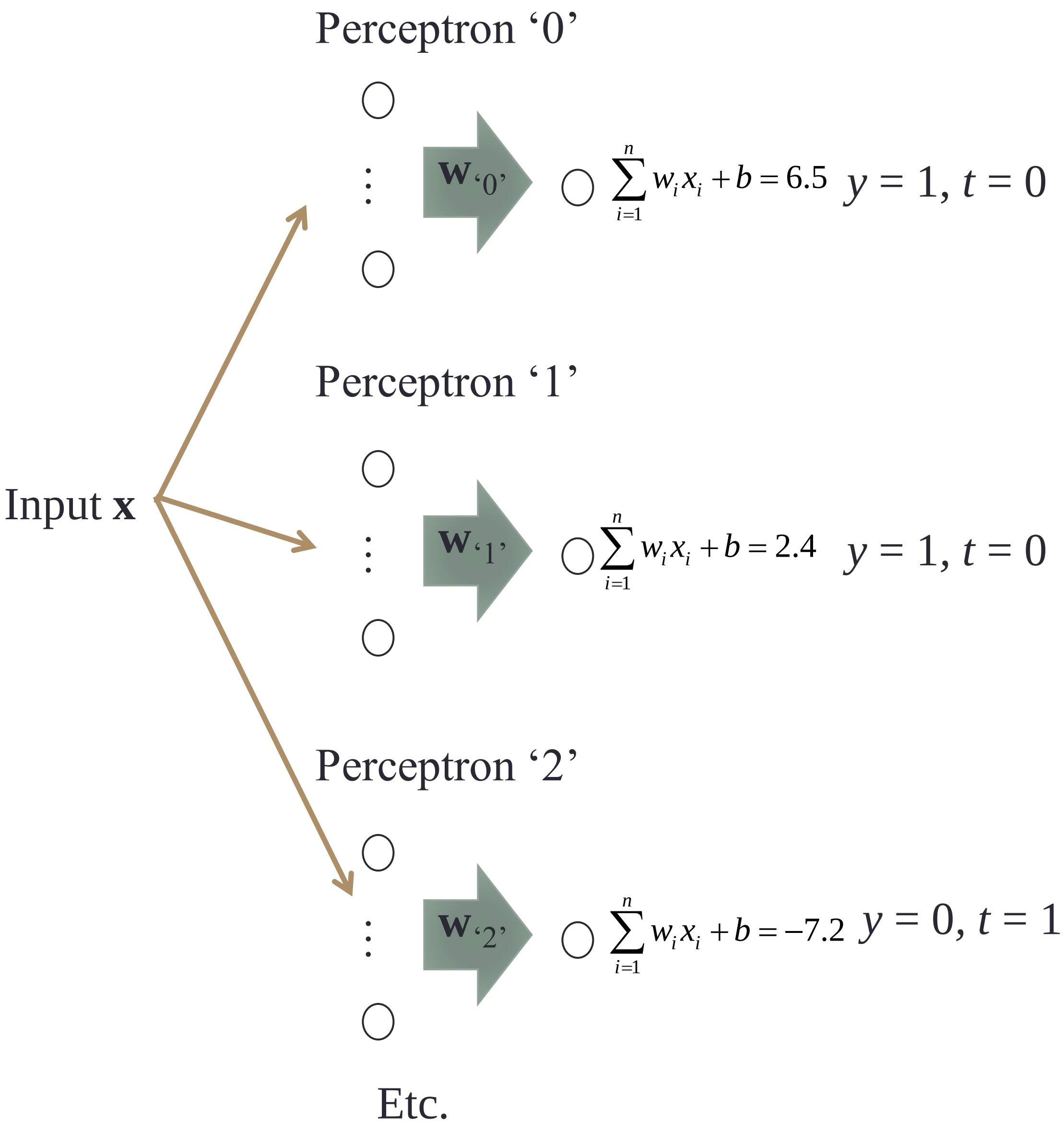
Computing targets



If input's class corresponds to perceptron's class, then target $t = 1$, otherwise target $t = 0$.

For example, suppose $\mathbf{x}$ is a '2'.

Computing targets

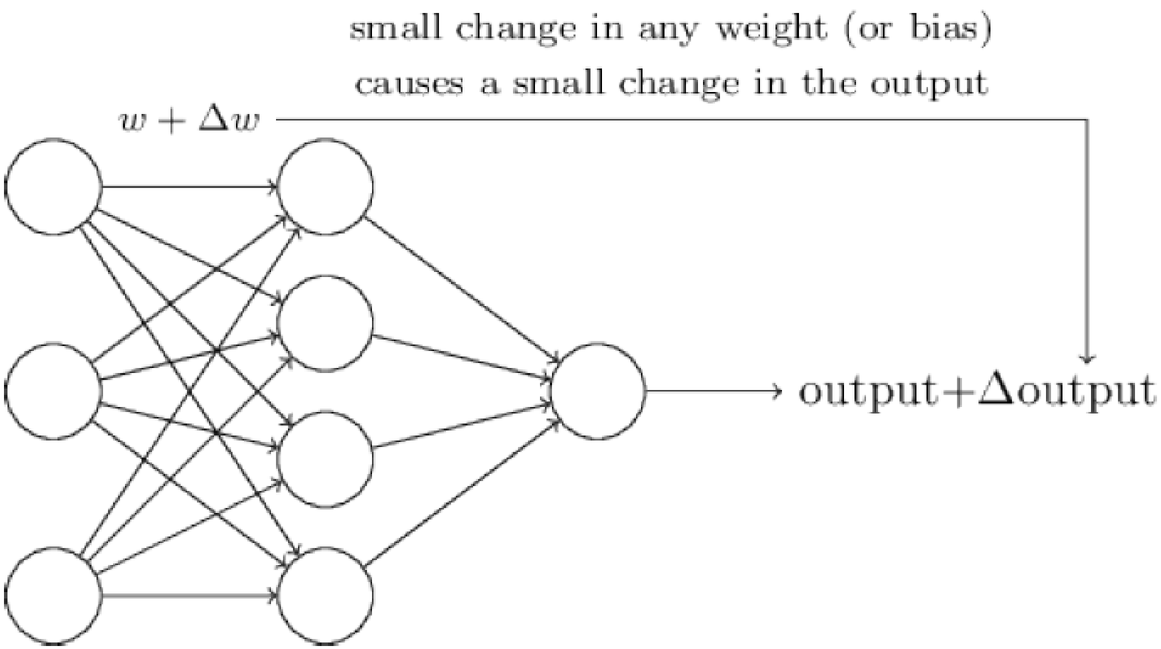


If input's class corresponds to perceptron's class, then target t = 1, otherwise target t = *0*.

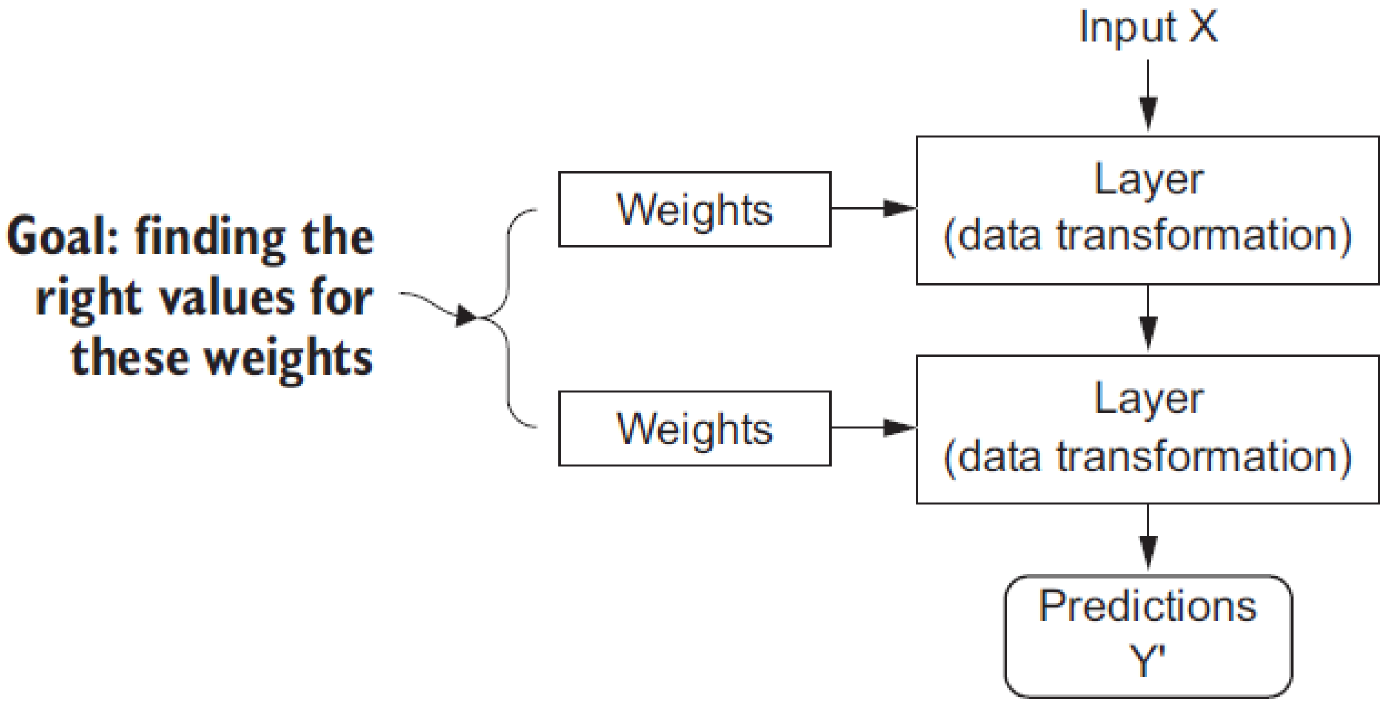
For example, suppose **x** is a '2'.

Learning in NN

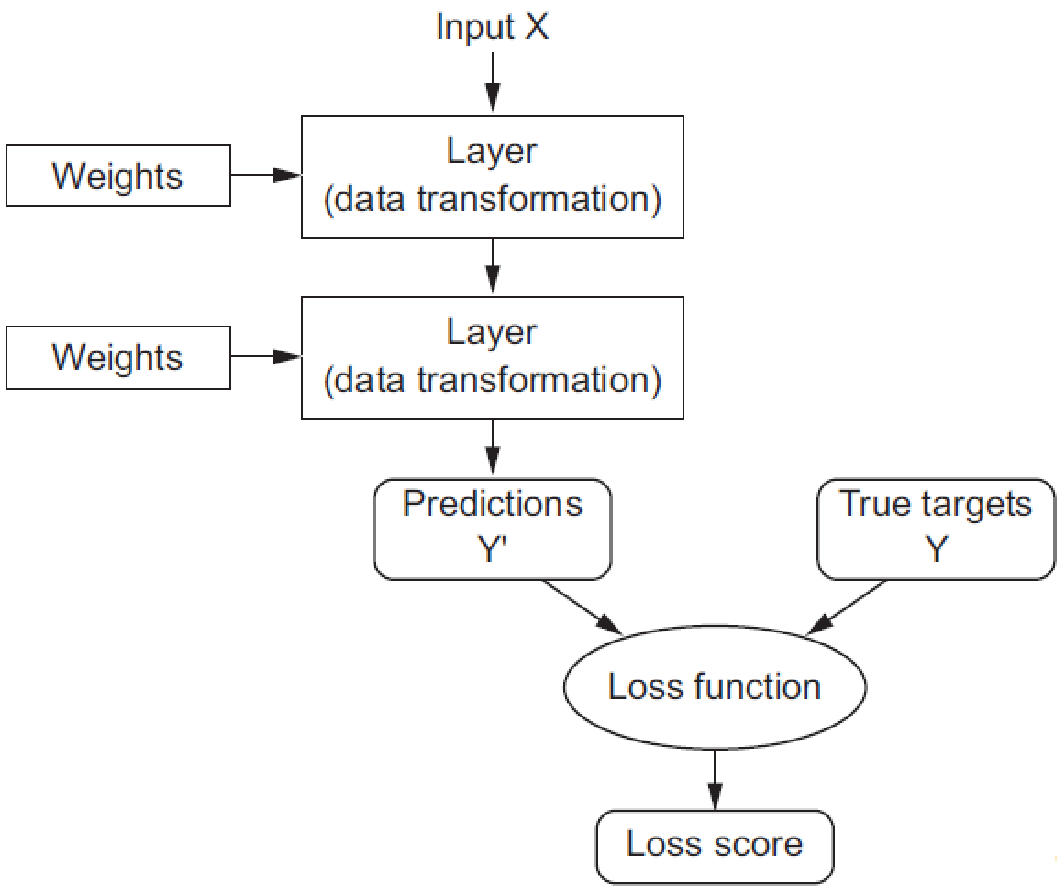
- Learning is a optimization process, calibrating the weights and biases



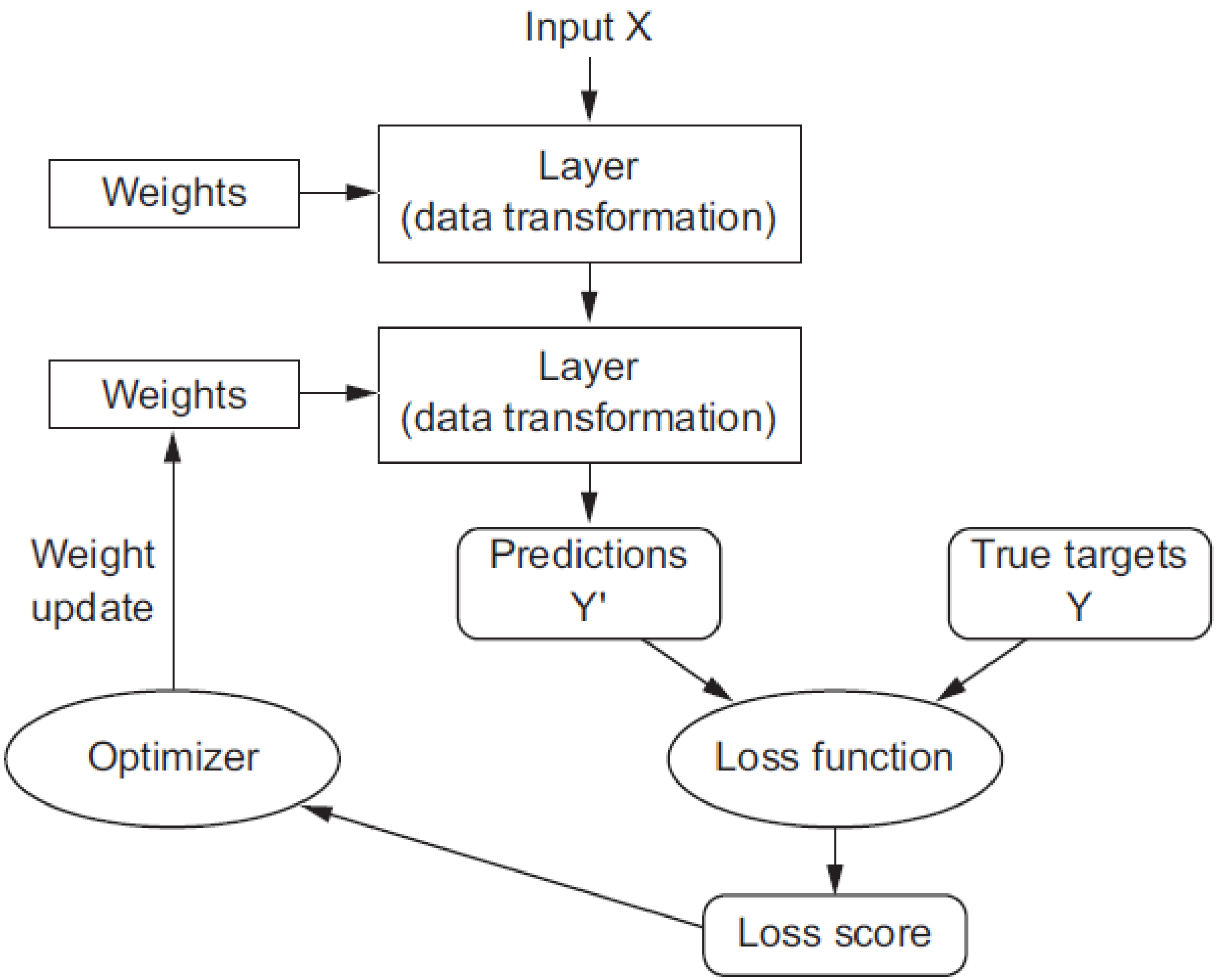
Learning in NN



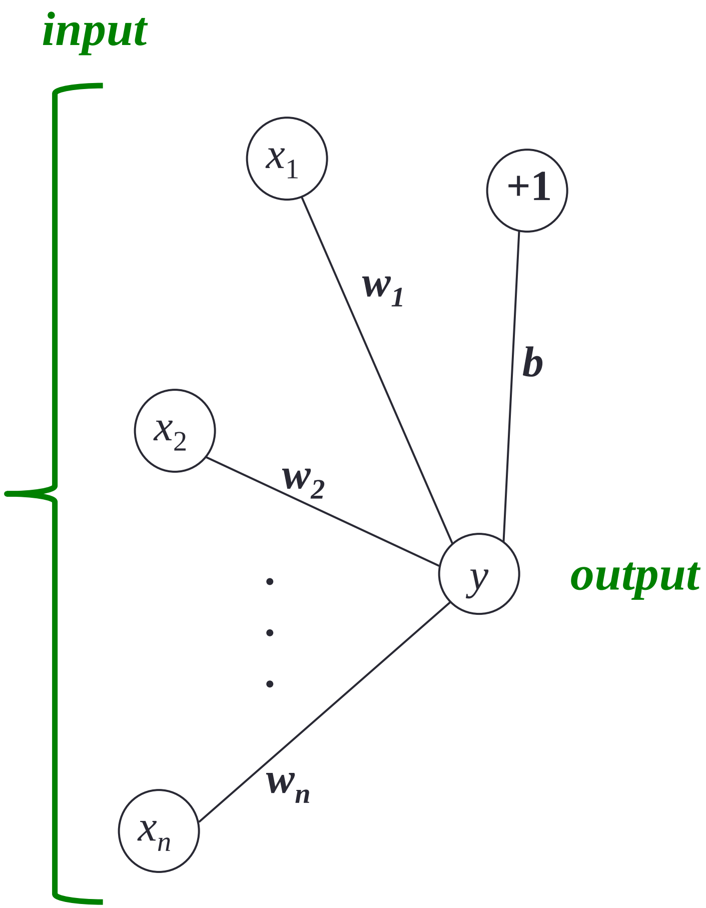
Learning in NN



Learning in NN



Perceptron Learning



Goal is to use the training data to learn a set of weights that will:

(1) correctly classify the training data

(2) generalize to unseen data

Input instance: $\mathbf{x}^k = (x_1, x_2, \ldots x_n)$,
with target class t^k Î {0,1}

Perceptron Learning

Learning is often framed as an optimization problem:

- Find **w** that minimizes average "loss":

$$J(\mathbf{w}, b) = \frac{1}{M} \sum_{k=1}^{M} L(\mathbf{w}, b, \mathbf{x}^k, t^k)$$

where M is number of training examples and L is a "loss" function.

One part of the "art" of ML is to define a good loss function.

- Here, define the loss function as follows:

$$\text{Let } y = a(\mathbf{w} \cdot \mathbf{x} + b)$$

$$L(\mathbf{w}, b, \mathbf{x}^k, t^k) = \frac{1}{2}\left(t^k - y\right)^2 \quad \text{"squared loss"}$$

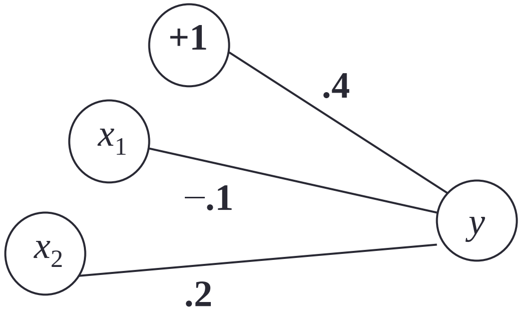
Example

Training set:

((0, 0), 1)

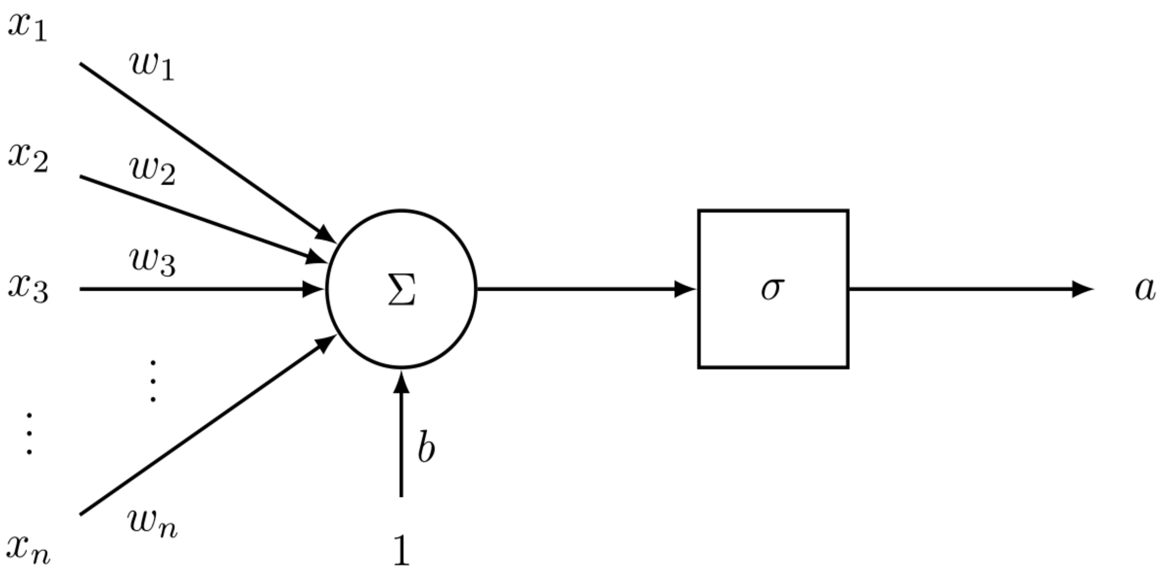
(1, 1), 0)

(1, 0), 0)



What is the average loss for this training set, using the squared loss function?

Sigmoid Neuron



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

$$z = wx + b$$

$$\sigma(wx + b) = \sigma(z)$$

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

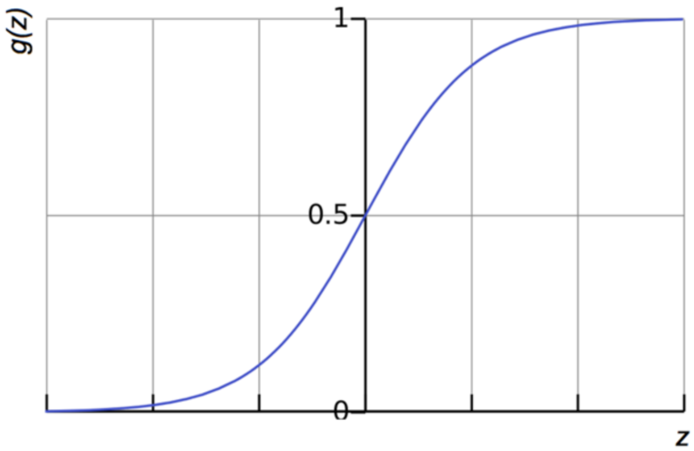
Sigmoid Function

- $\sigma(z) = \frac{1}{1+e^{-z}}$ is called the "sigmoid" or "logistic" function.
- it is an S-shaped function that "squashes" the value of $wx + b$ into the range [0, 1] so that we may interpret the output as a probability.

Bound its values between 0 and 1

- Below function is unbounded:

$$wx + b$$



- We are going to bound its output:

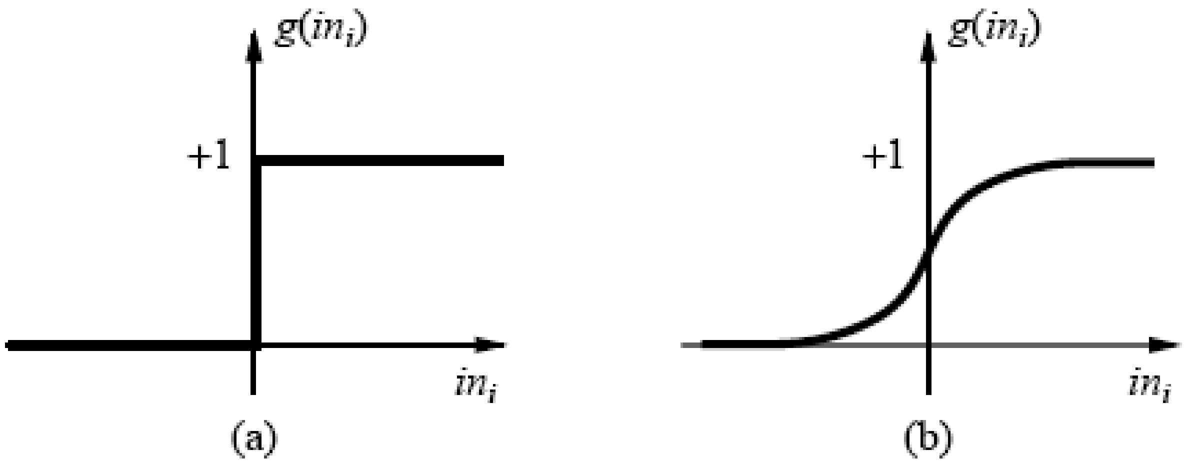
$$\sigma(\theta^T x + b),$$

where $\sigma(z)$ is sigmoid function.

$$\sigma(z) = \frac{1}{1 + exp(-z)}$$

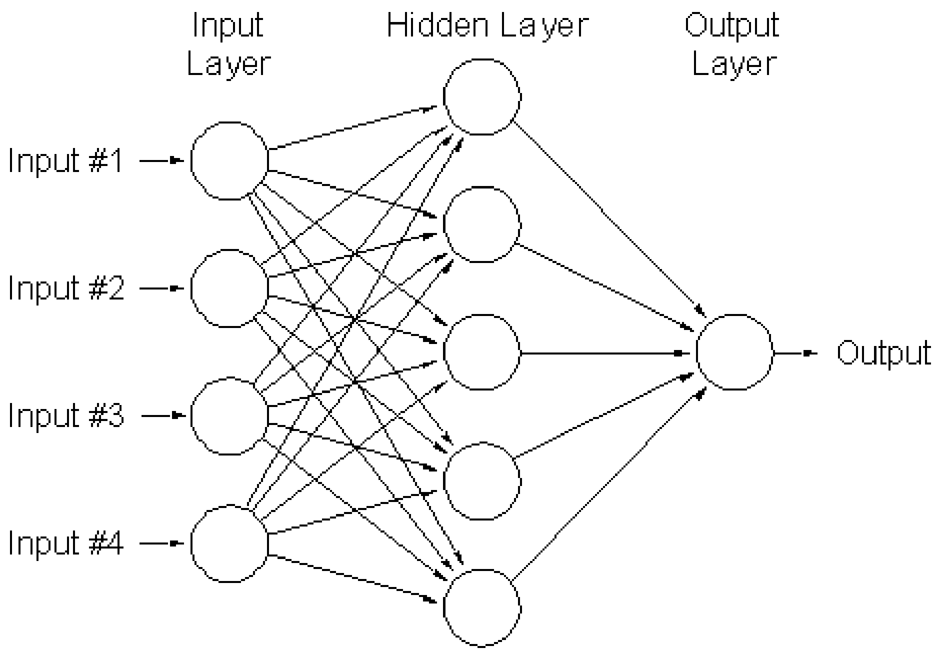
- Notice that the sigmoid function transforms our output into the range between **0 and 1**.

Step Function vs Sigmoid



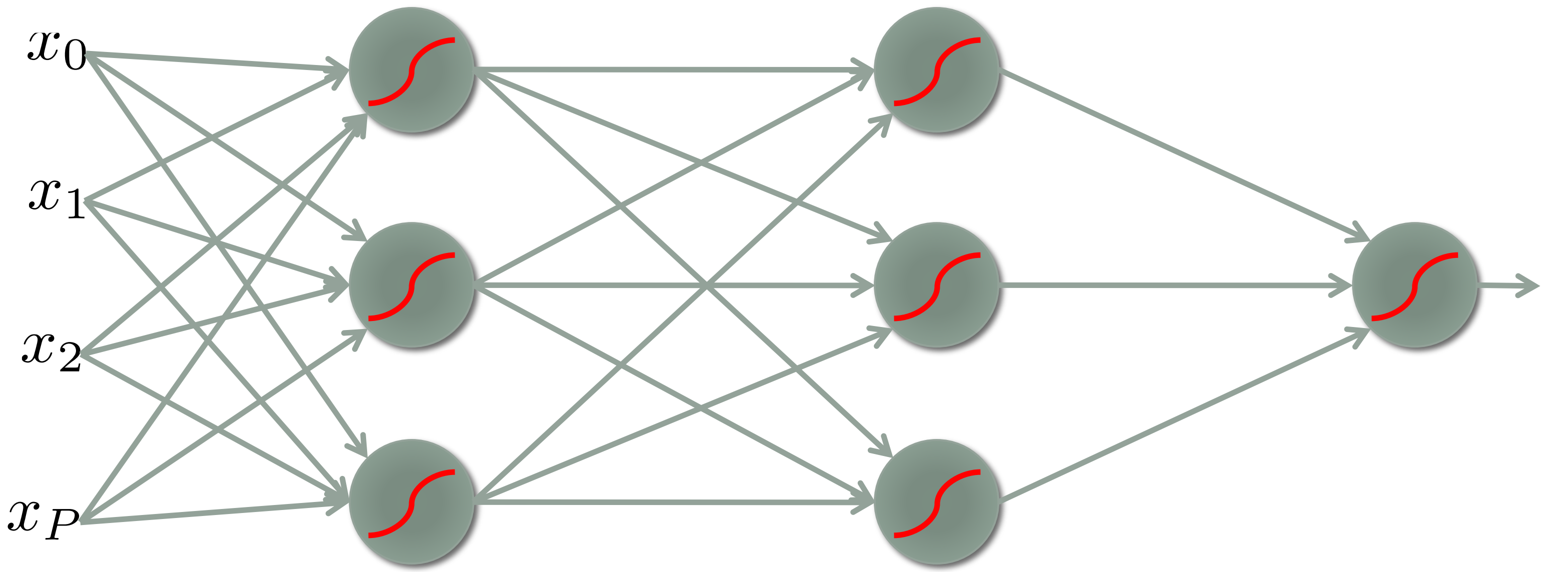
$$\Delta \text{output} \approx \sum_j \frac{\partial\, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial\, \text{output}}{\partial b} \Delta b$$

A multi-layer neural network…



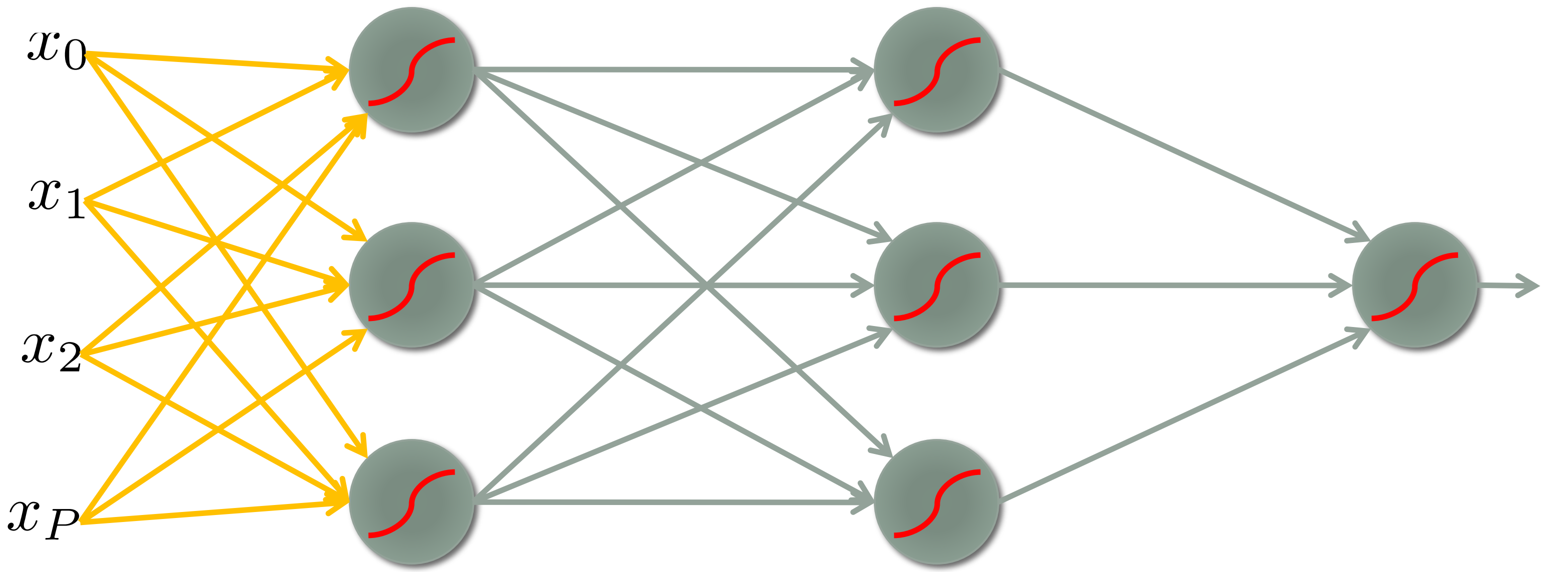
Feed-forward networks

- Cascade neurons together
- Output from one layer is the input to the next
- Each layer has its own sets of weights



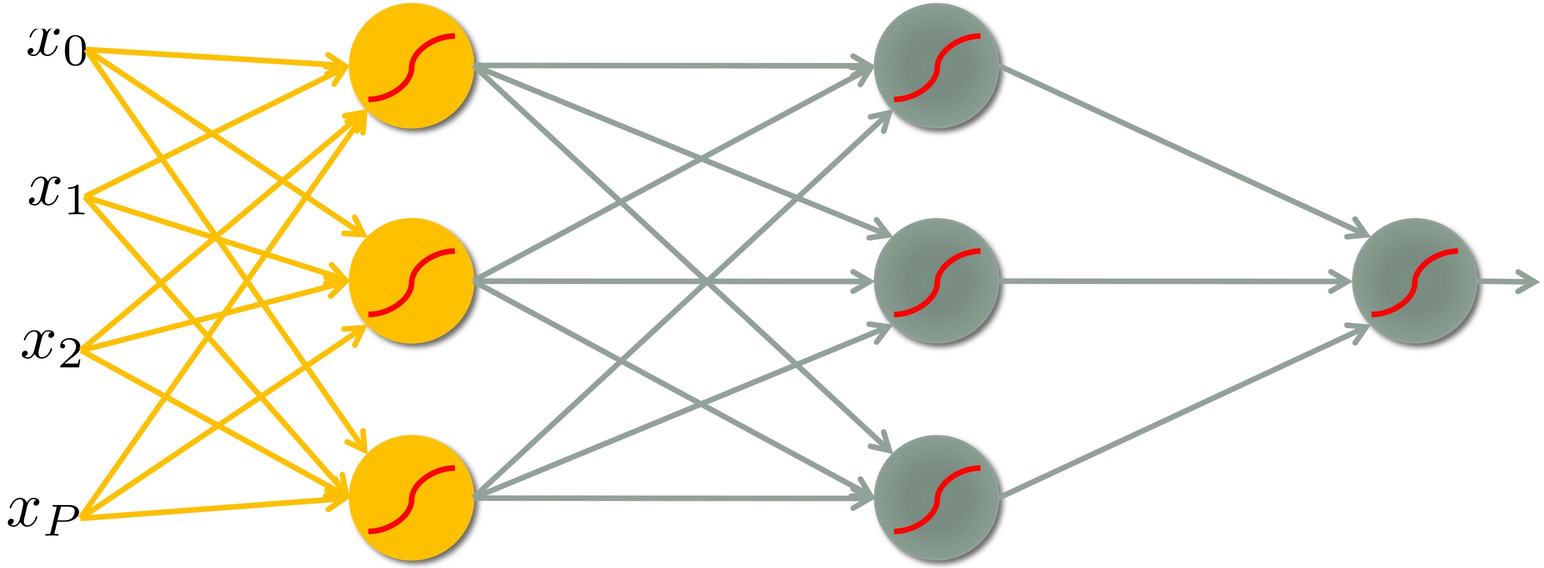
Feed-forward networks

- Predictions are fed forward through the network to classify



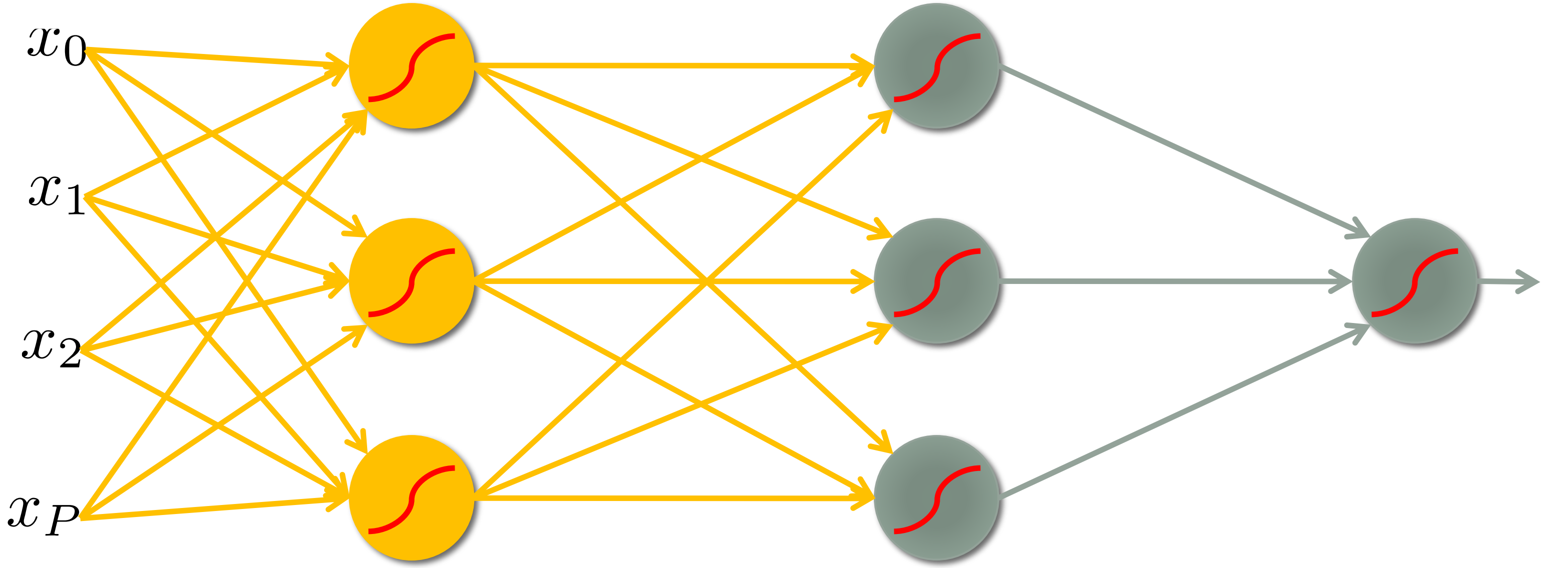
Feed-forward networks

- Predictions are fed forward through the network to classify



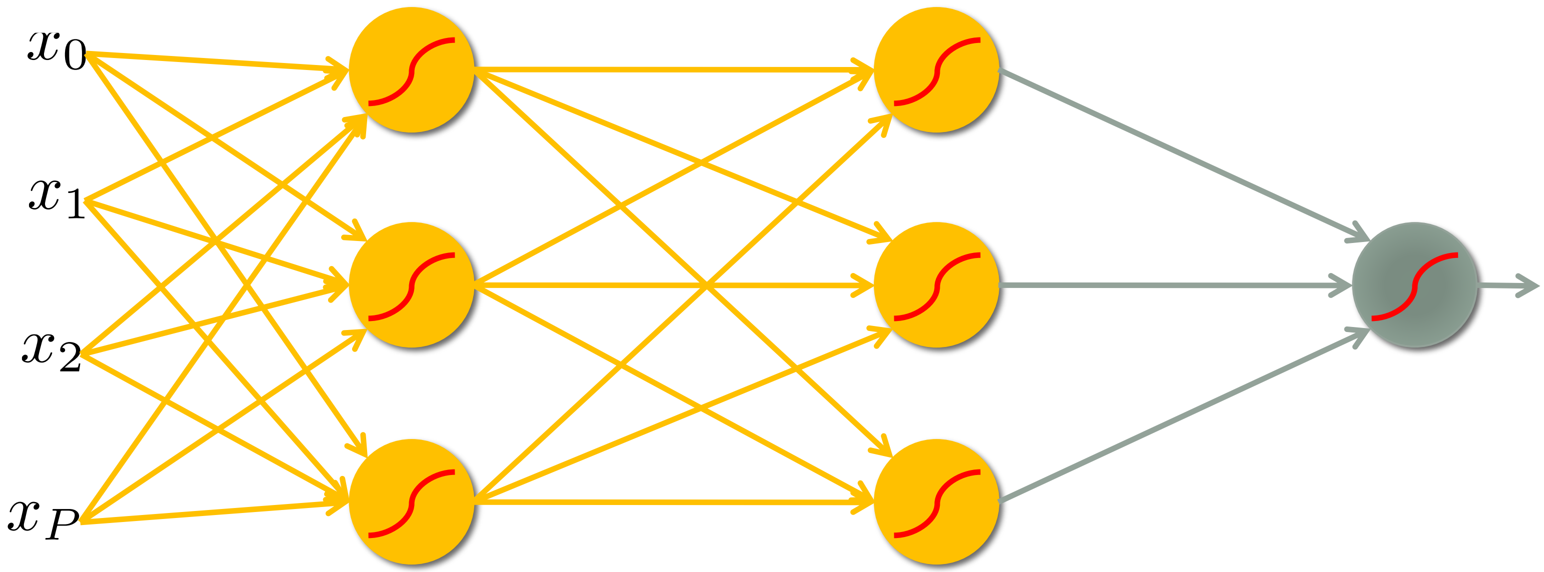
Feed-forward networks

- Predictions are fed forward through the network to classify



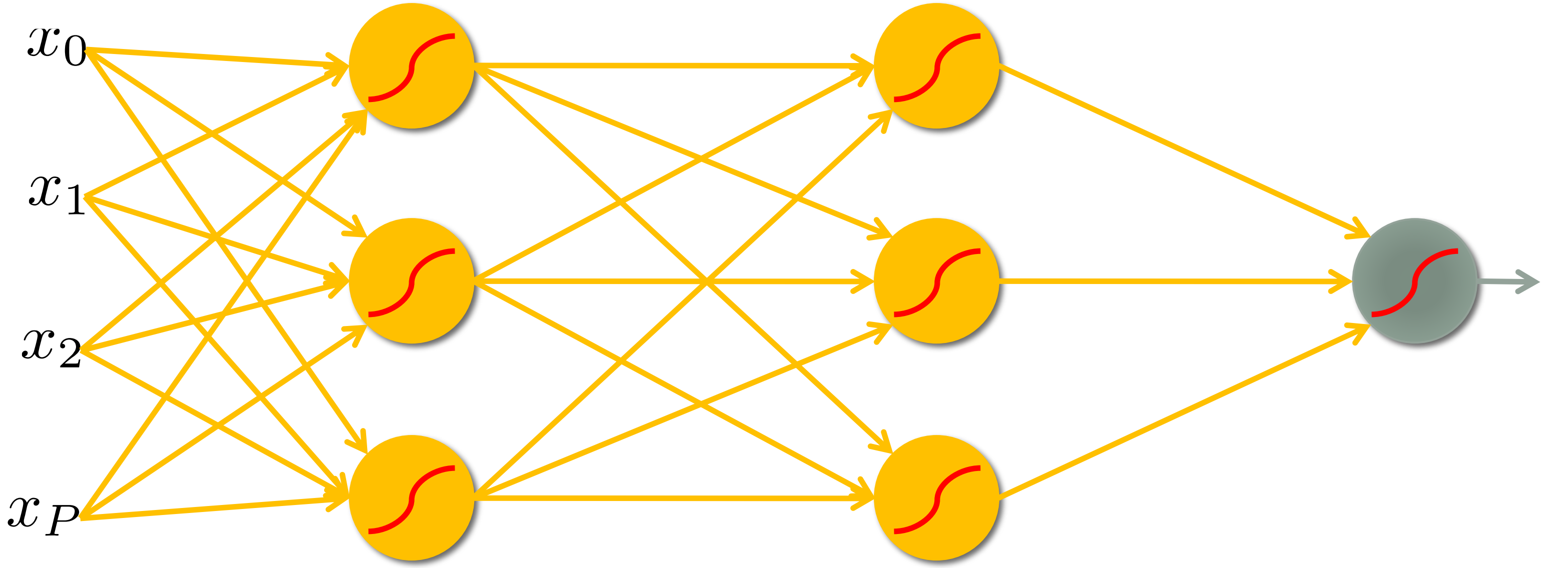
Feed-forward networks

- Predictions are fed forward through the network to classify



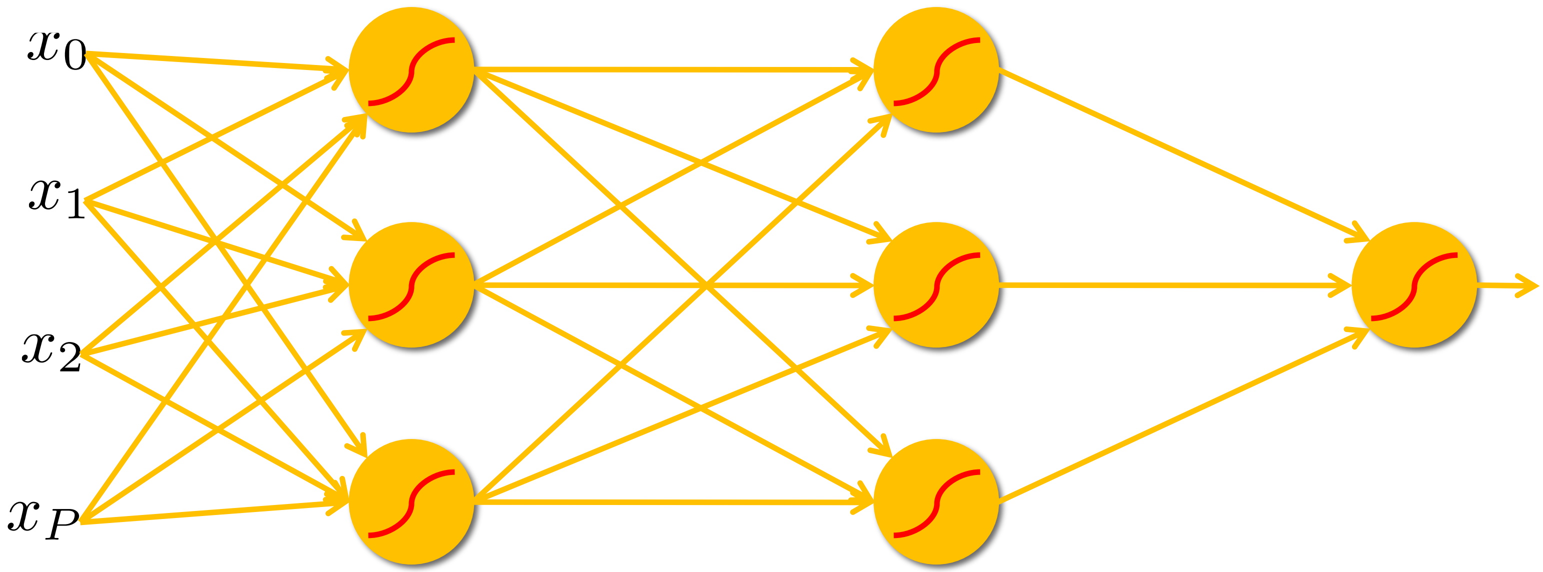
Feed-forward networks

- Predictions are fed forward through the network to classify

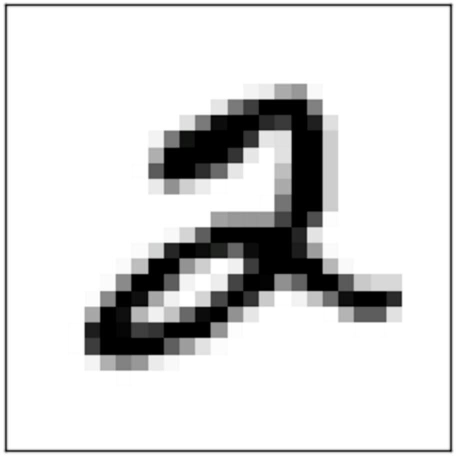


Feed-forward networks

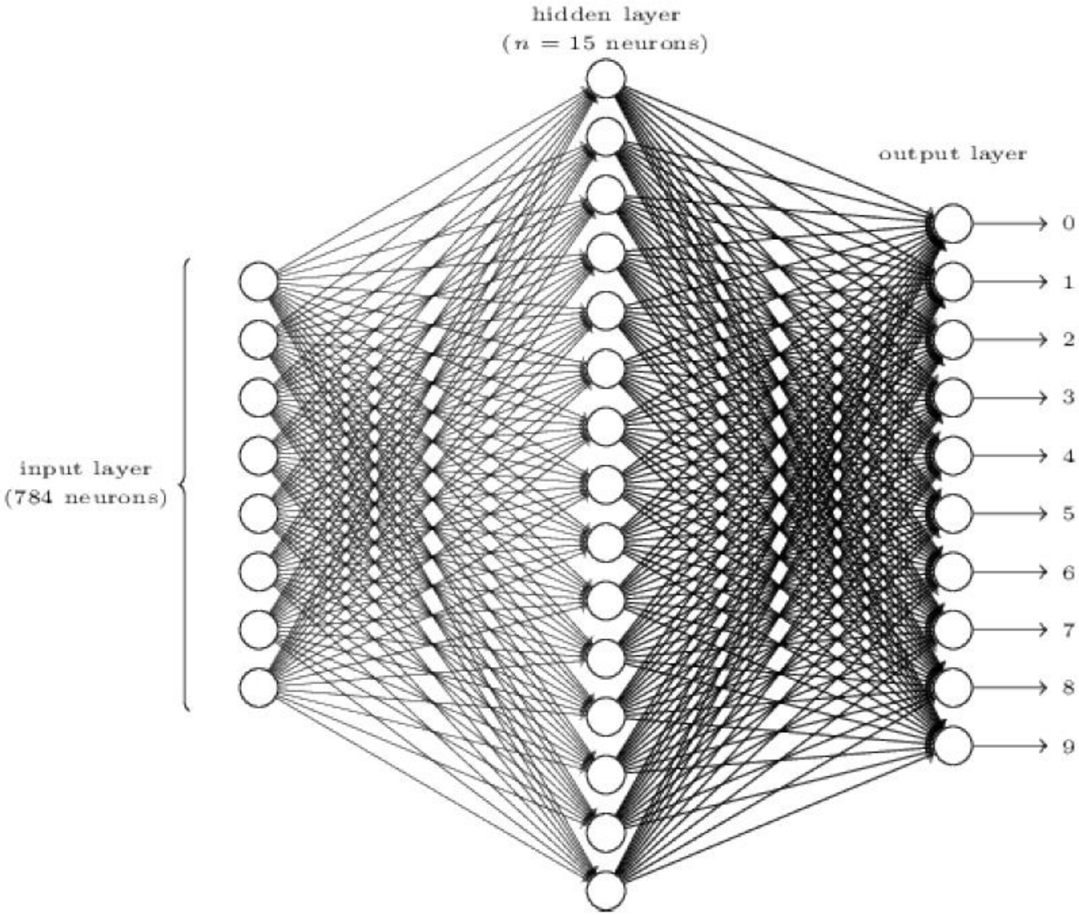
- Predictions are fed forward through the network to classify



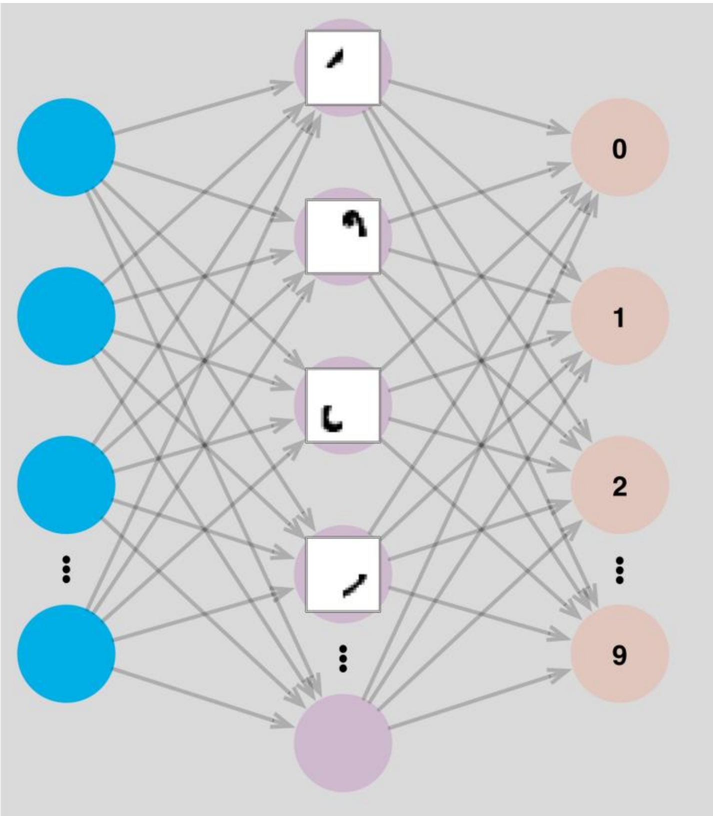
Recognizing Handwritten Digits



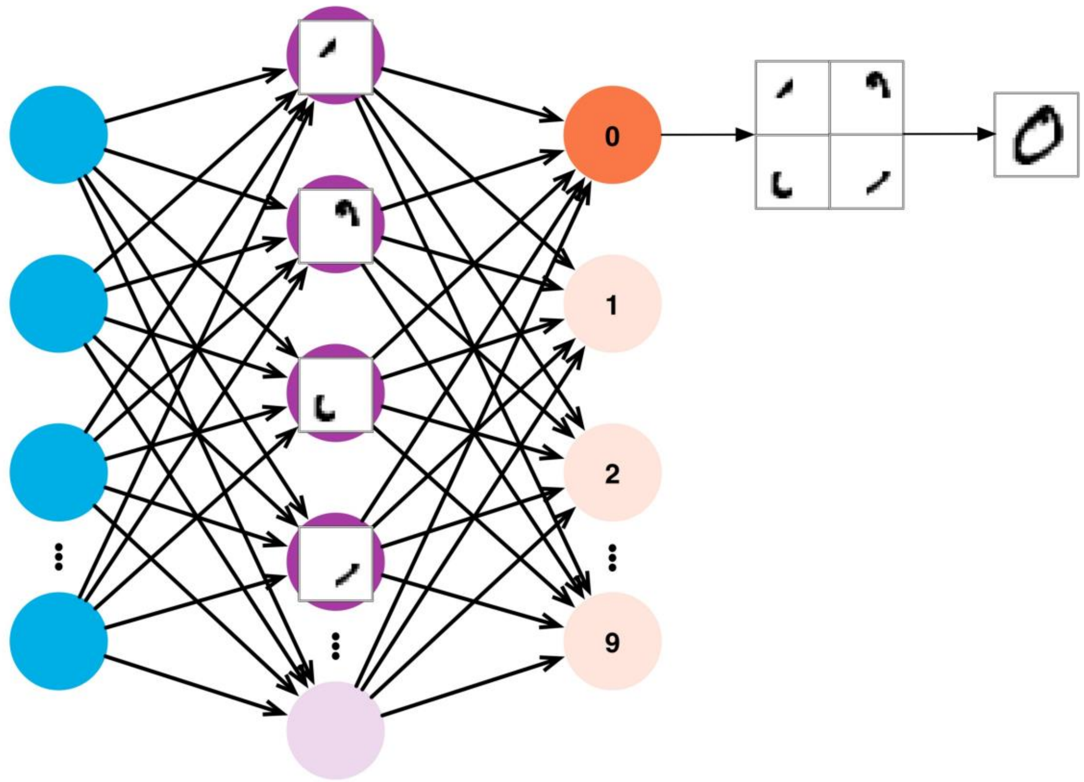
28x28 pixels



Hidden Layer

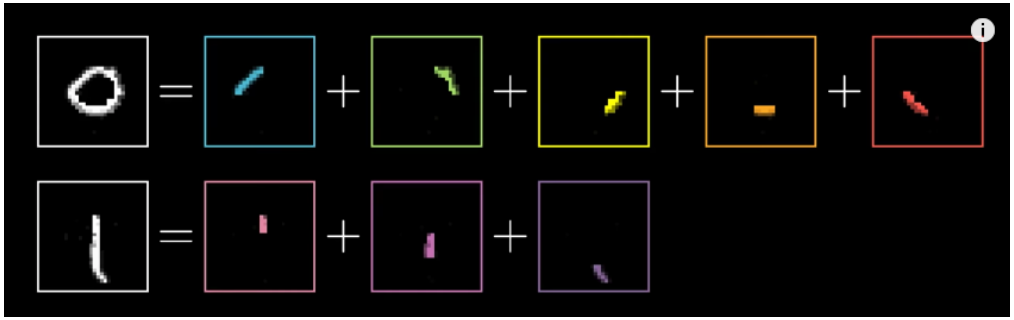


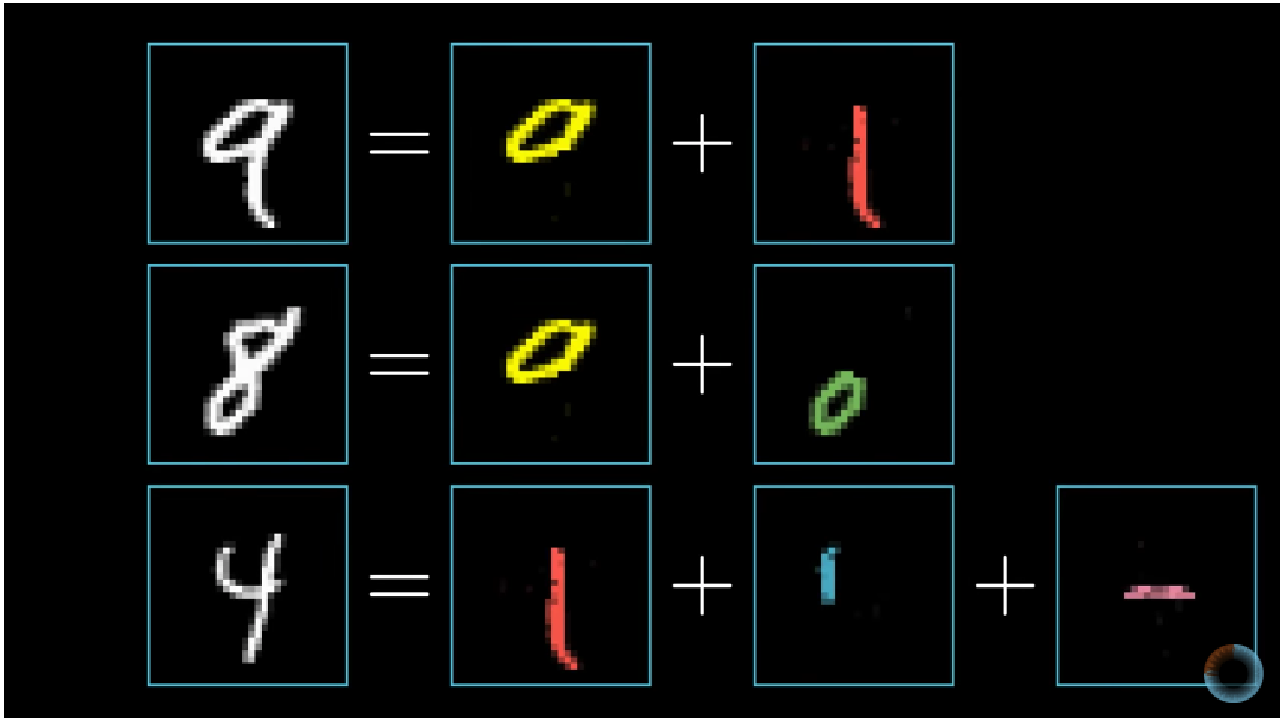
A Trained Neural Network in which each hidden neuron classifies certain pattern



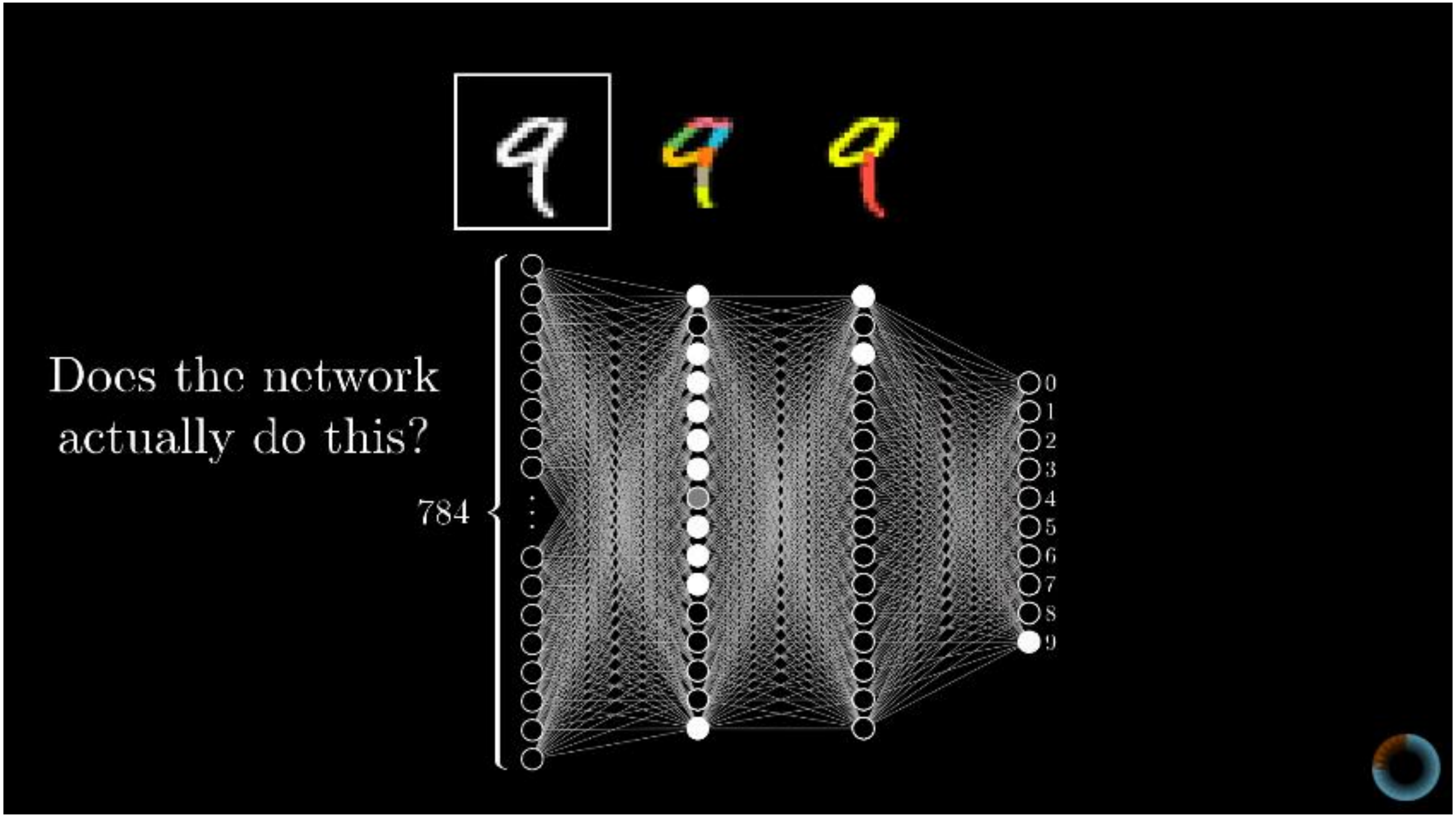
Feeding a handwritten digit of 0 should trigger the 4 hidden layer neurons, and then the first output neuron

Recognizing Handwritten Digits

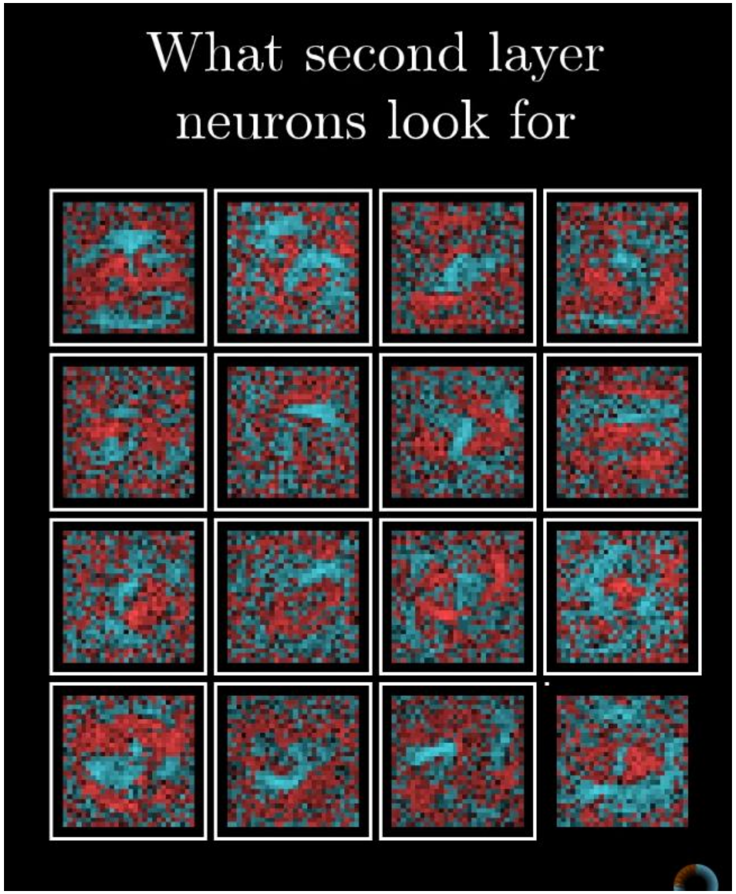




Recognizing Handwritten Digits

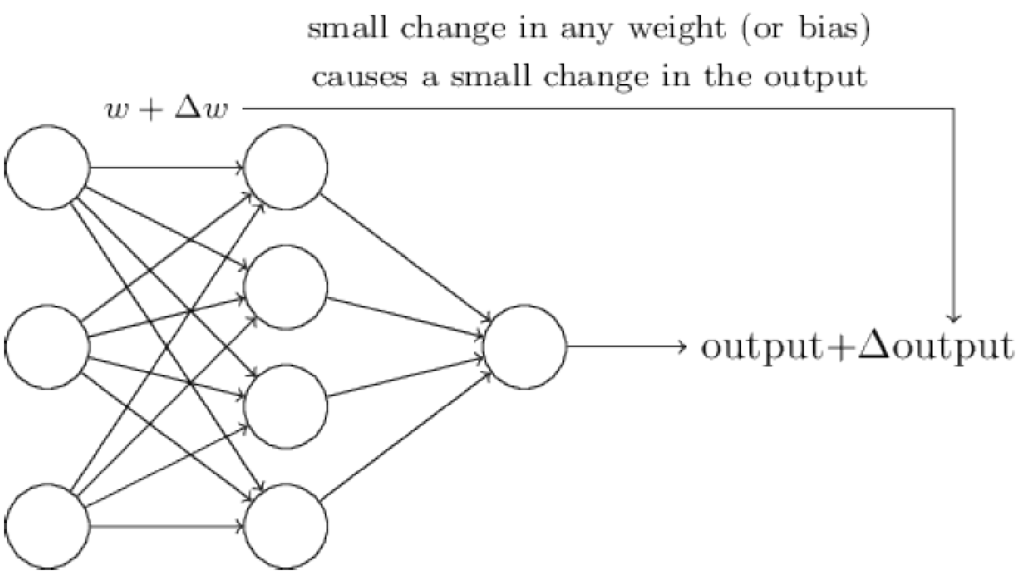


Recognizing Handwritten Digits



Learning in NN

- Learning with sigmoid function



How do we train neural networks?

- No closed-form solution for the weights (i.e. we cannot set up a system A*w = b)
- We will iteratively find such a set of weights that allow the outputs to match the desired outputs
- We want to minimize a loss function (a function of the weights in the network)

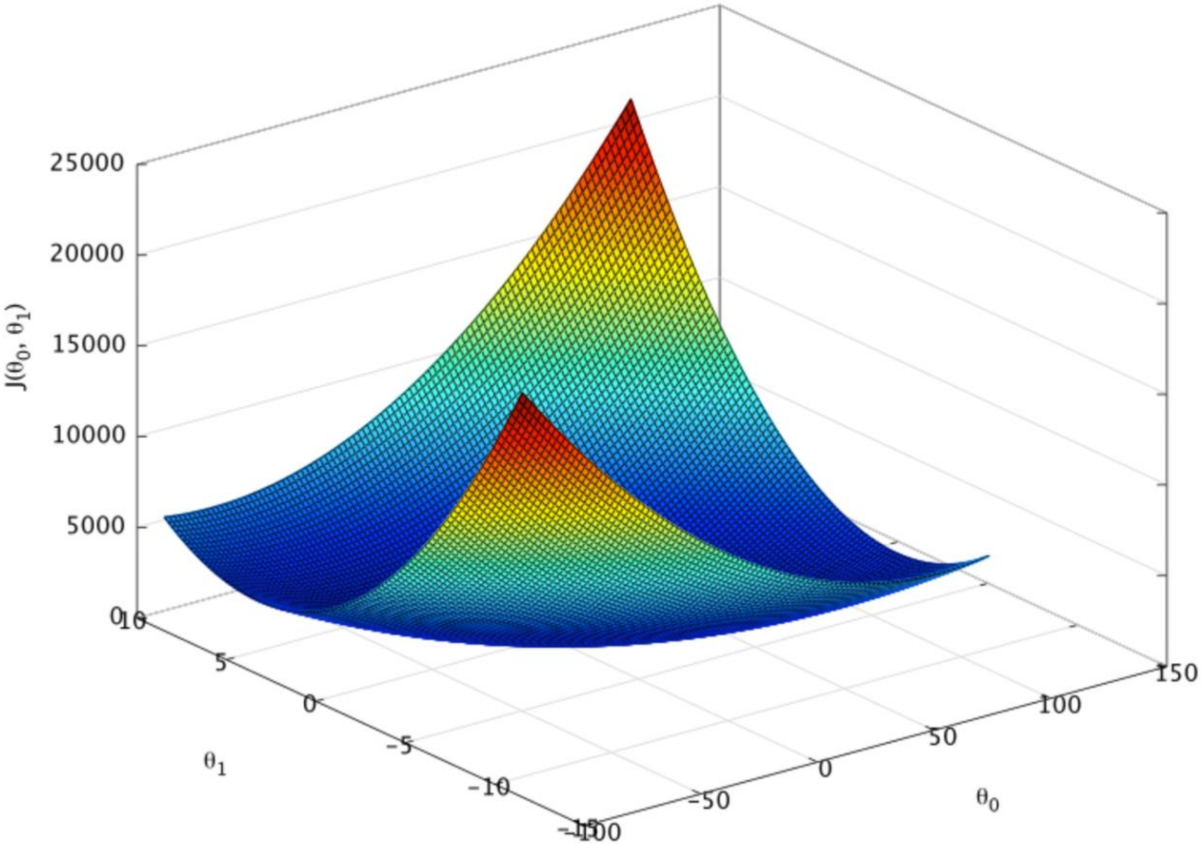
Evaluating the NN

- We use a cost function to quantify how well our NN operates

$$C(w,b) = \frac{1}{2n}\sum_i (y - a(w,b))^2$$

- the penalty for a bad guess goes up quadratically with the difference between the guess and the correct answer
- it acts as a very “strict” measurement of wrongness.

Plot of Cost Function



- We see that our goal is to find parameters such that our cost function is as small as possible.
- Take the gradients in different parameter directions.

How to minimize the loss function?

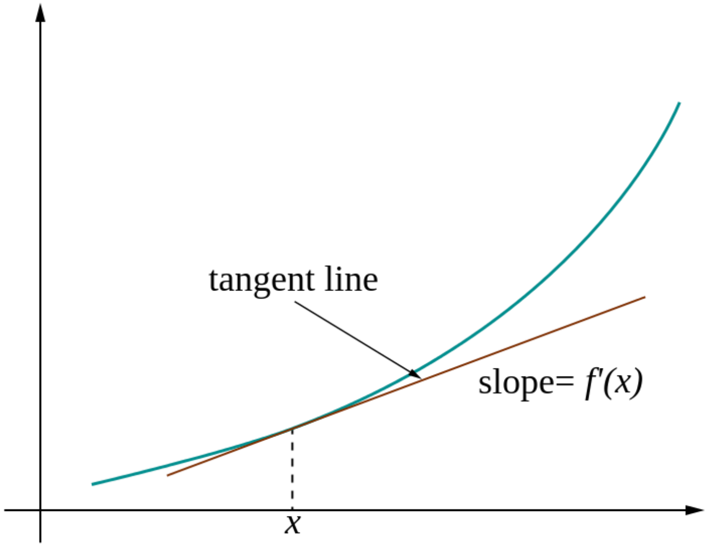
In 1-dimension, the derivative of a function:

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

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

Loss gradients

- How does the loss change as a function of the weights
- We want to change the weights in such a way that makes the loss decrease as fast as possible



current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:	**W + h** (first dim)**:**	**gradient dW:**
[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,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25322**	

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (first dim)**:**

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25322

gradient dW:

[**-2.5**,
?,
?,

(1.25322 - 1.25347)/0.0001
= -2.5

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

?,
?,…]

current W:	**W + h** (second dim)**:**	**gradient dW:**
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11 + **0.0001**,	?,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25353**	

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (second dim)**:**

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25353

gradient dW:

[-2.5,
0.6,
?,
?,

(1.25353 - 1.25347)/0.0001
= 0.6

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

?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

gradient dW:

[-2.5,
0.6,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

dW = ...
(some function
data and W)

gradient dW:

[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,…]

Gradient descent

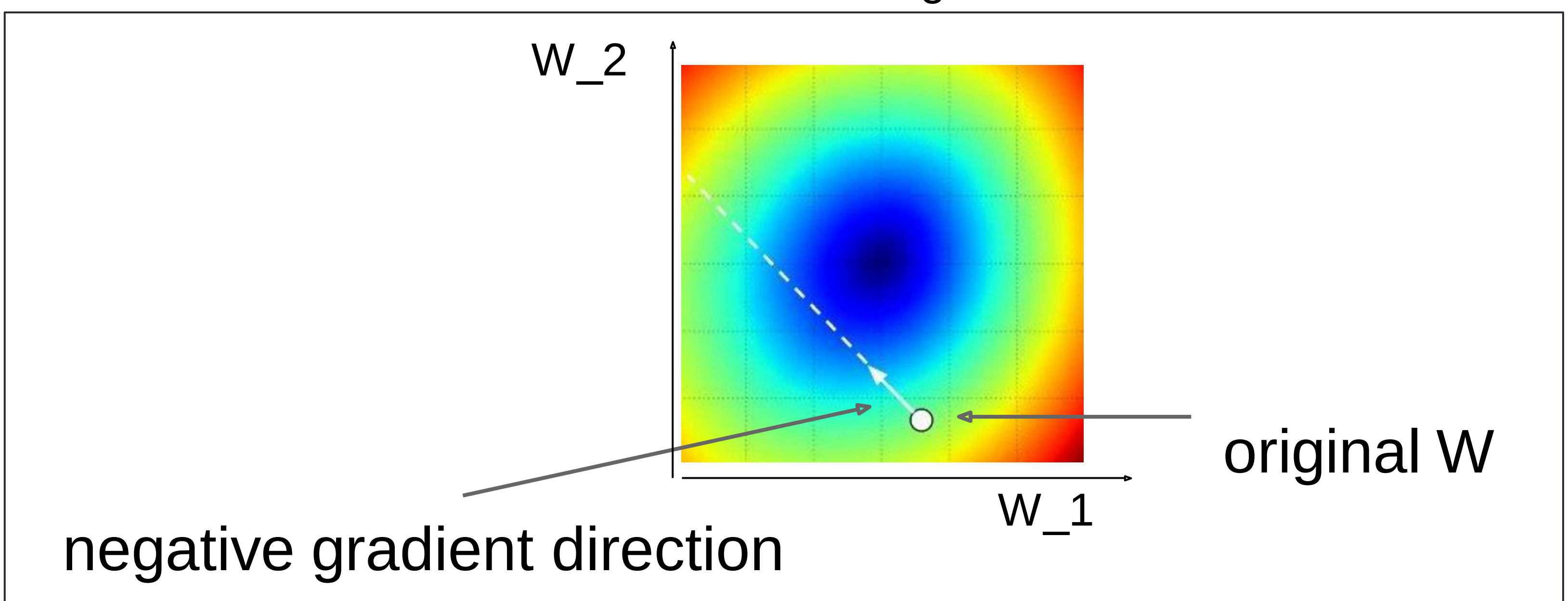
Gradient descent

- We'll update weights
- Move in direction opposite to gradient:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

Time

Learning rate



Gradient descent

- Suppose we change just two weight values, w_1 and w_2

$$\Delta C \approx \frac{\partial C}{\partial w_1}\Delta w_1 + \frac{\partial C}{\partial w_2}\Delta w_2$$

- Gradient

$$\nabla C = \left(\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}\right)^T$$

$$\Delta C \approx \nabla C \cdot \Delta w$$

Gradient descent

- If we choose

$$\Delta w = -\eta \nabla C$$

- Then

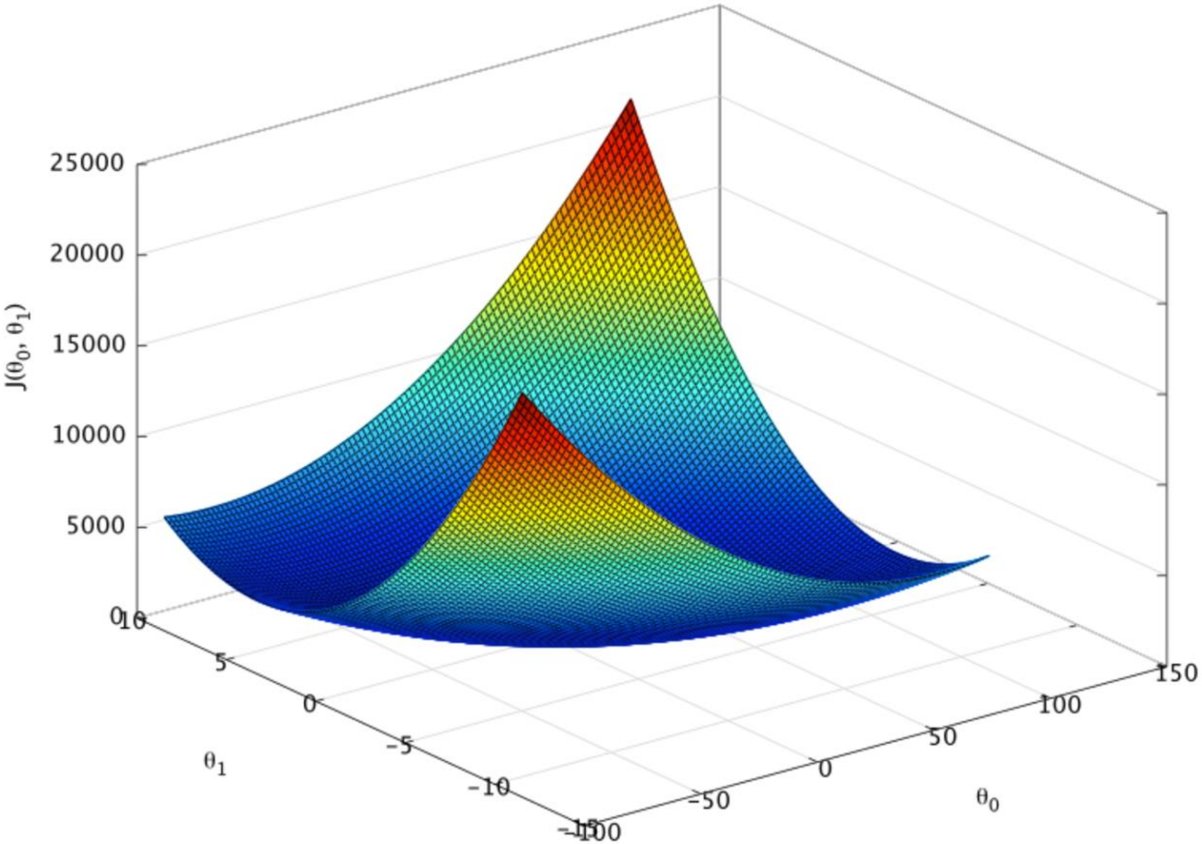
$$\Delta C \approx \nabla C \cdot \Delta w \approx -\eta \nabla C \cdot \nabla C$$

$$w \to w' = w - \eta \nabla C$$

Gradient descent

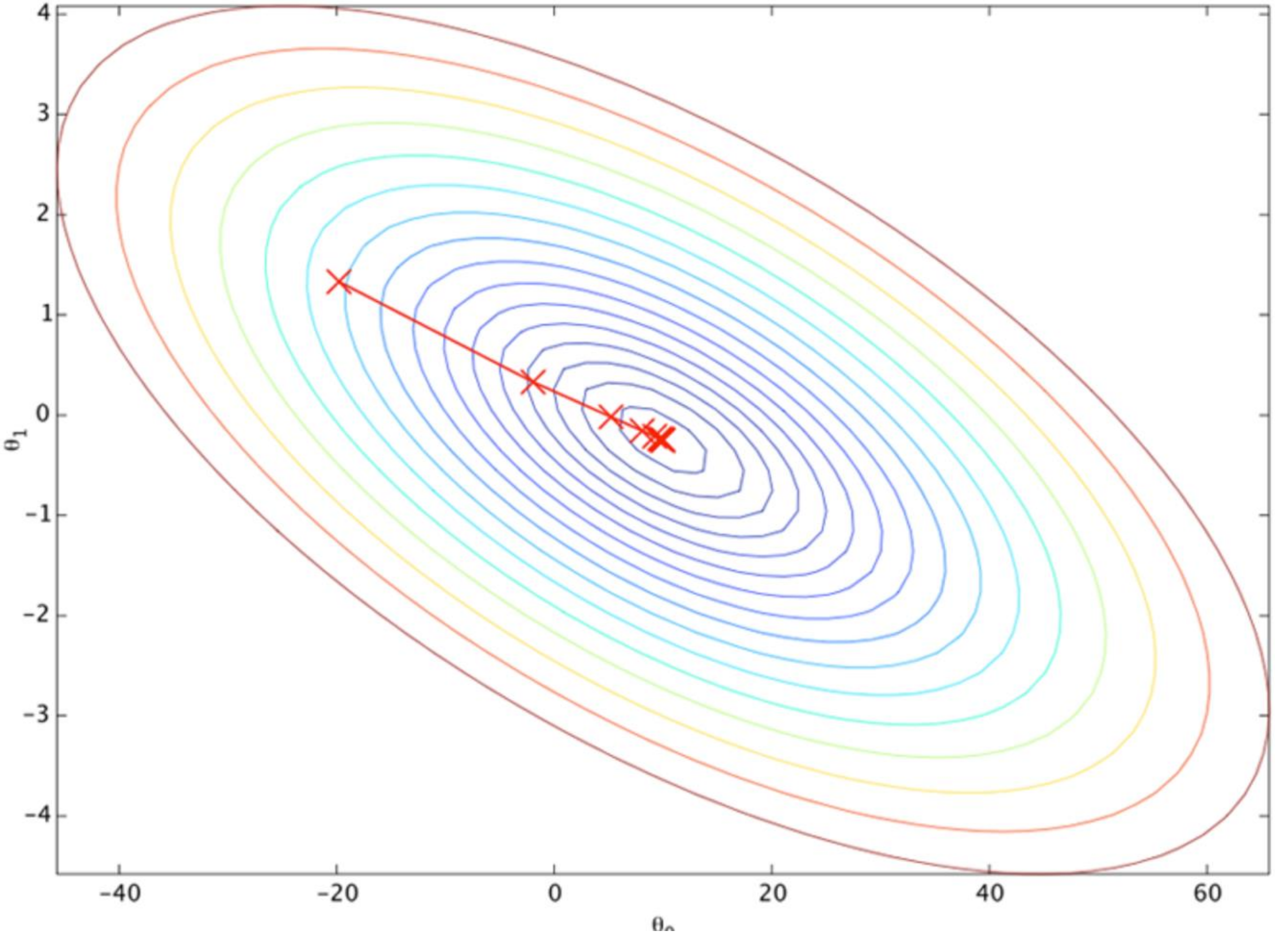
- Iteratively *subtract* the gradient with respect to the model parameters (w)
- I.e. we're moving in a direction opposite to the gradient of the loss
- I.e. we're moving towards *smaller* loss

Plot of Cost Function

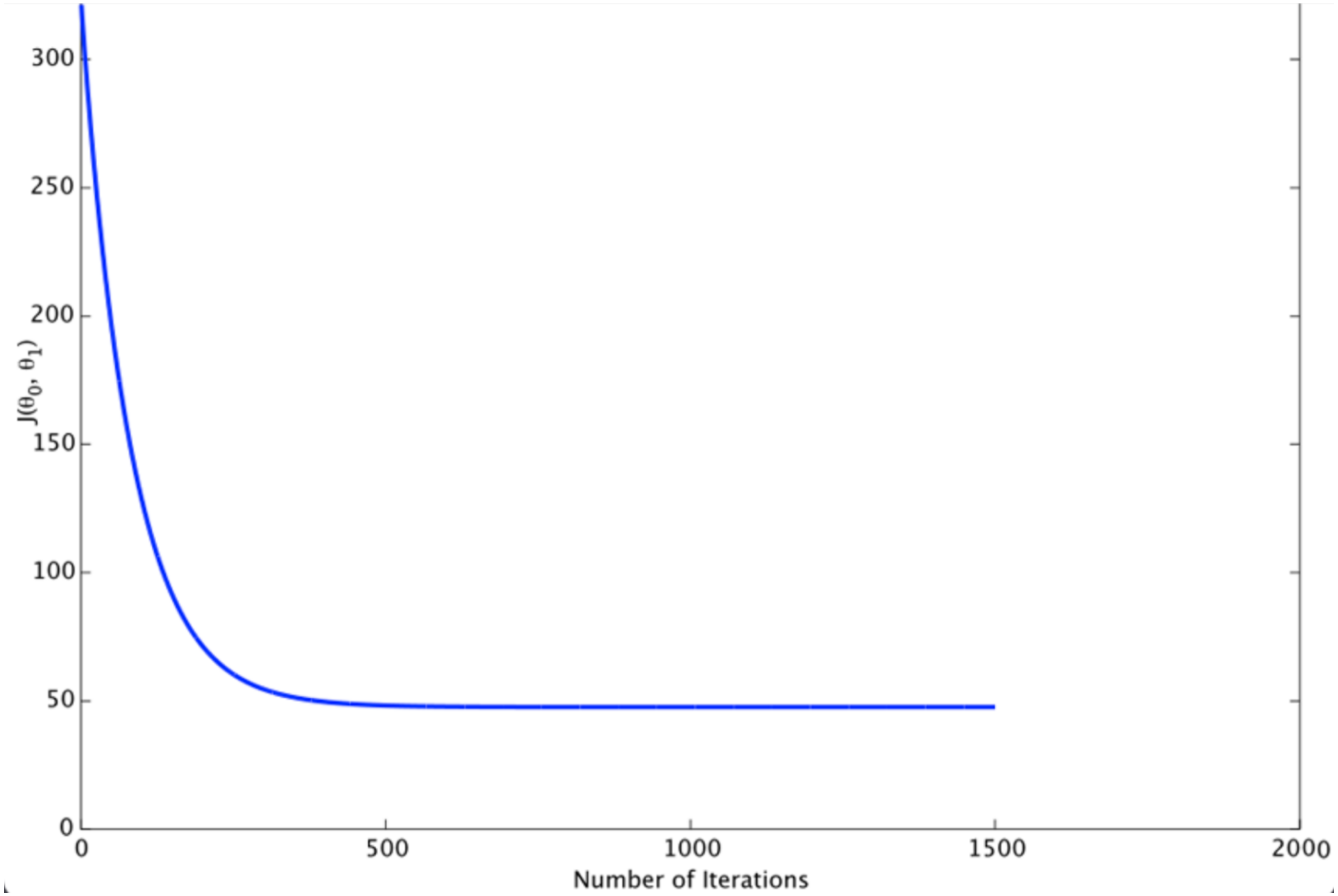


- We see that our goal is to find parameters such that our cost function is as small as possible.
- Take the gradients in different parameter directions.

Gradient descent iterations



Cost function vs iterations



Gradient descent

- In general case $w_1, w_2 \ldots w_k$

$$\nabla C = \left(\frac{\partial C}{\partial w_1}, \cdots, \frac{\partial C}{\partial w_k}\right)^T$$

$$\Delta C \approx \nabla C \cdot \Delta w$$

$$w \rightarrow w' = w - \eta \frac{\partial C}{\partial w}$$

- What about the bias?

$$b \rightarrow b' = b - \eta \frac{\partial C}{\partial b}$$

Gradient descent ~ Challenges

- Iterations

$$w \rightarrow w' = w - \eta \frac{\partial C}{\partial w}, \quad b \rightarrow b' = b - \eta \frac{\partial C}{\partial b}$$

- Huge number of instances!
- Remember

$$C(w,b) = \frac{1}{2n} \sum_i (y - a(w,b))^2$$

$$C = \frac{1}{n} \sum_i C^i \qquad \nabla C = \frac{1}{n} \sum_i \nabla C^i$$

(Batch) Gradient Descent

1. Initialize $\mathbf{w} := 0^{m-1}, b := 0$
2. for epoch $e \in [1, \ldots, E]$:
   - 2.1. shuffle $\mathcal{D}$ to prevent cycles
   - 2.2. for every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:
     - 2.2.1. compute prediction $\hat{y}^{[i]} := h(\mathbf{x}^{[i]})$
   - 2.3. compute loss $\mathcal{L} := \frac{1}{n} \sum_{i=1}^{n} L(\hat{y}^{[i]}, y^{[i]})$
   - 2.4. compute gradients $\Delta \mathbf{w} := -\nabla_{\mathcal{L}^{[i]}} \mathbf{w},\ \Delta b := -\frac{\partial \mathcal{L}}{\partial b}$
   - 2.5. update parameters $\mathbf{w} := \mathbf{w} + \Delta \mathbf{w},\ b := +\Delta b$

Stochastic Gradient Descent

- Randomly choose an instances with a sequence iterate over all instances

$$\nabla C = \frac{1}{n} \sum_i \nabla C^i \approx \nabla C^i$$

Stochastic Gradient Descent

1. Initialize $\mathbf{w} := 0^{m-1}, b := 0$
2. for epoch $e \in [1, \ldots, E]$:
   - 2.1. shuffle $\mathcal{D}$ to prevent cycles
   - 2.2. for every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:
     - 2.2.3. compute prediction $\hat{y}^{[i]} := h(\mathbf{x}^{[i]})$
     - 2.2.4. compute loss $\mathcal{L}^{[i]} := L(\hat{y}^{[i]}, y^{[i]})$
     - 2.2.5. compute gradients $\Delta \mathbf{w} := -\nabla_{\mathcal{L}^{[i]}} \mathbf{w},\ \Delta b := -\frac{\partial \mathcal{L}^{[i]}}{\partial b?}$
     - 2.2.6. update parameters $\mathbf{w} := \mathbf{w} + \Delta \mathbf{w},\ b := +\Delta b$

Mini-Batch Stochastic Gradient Descent

- Randomly choose a subset of instances, say m

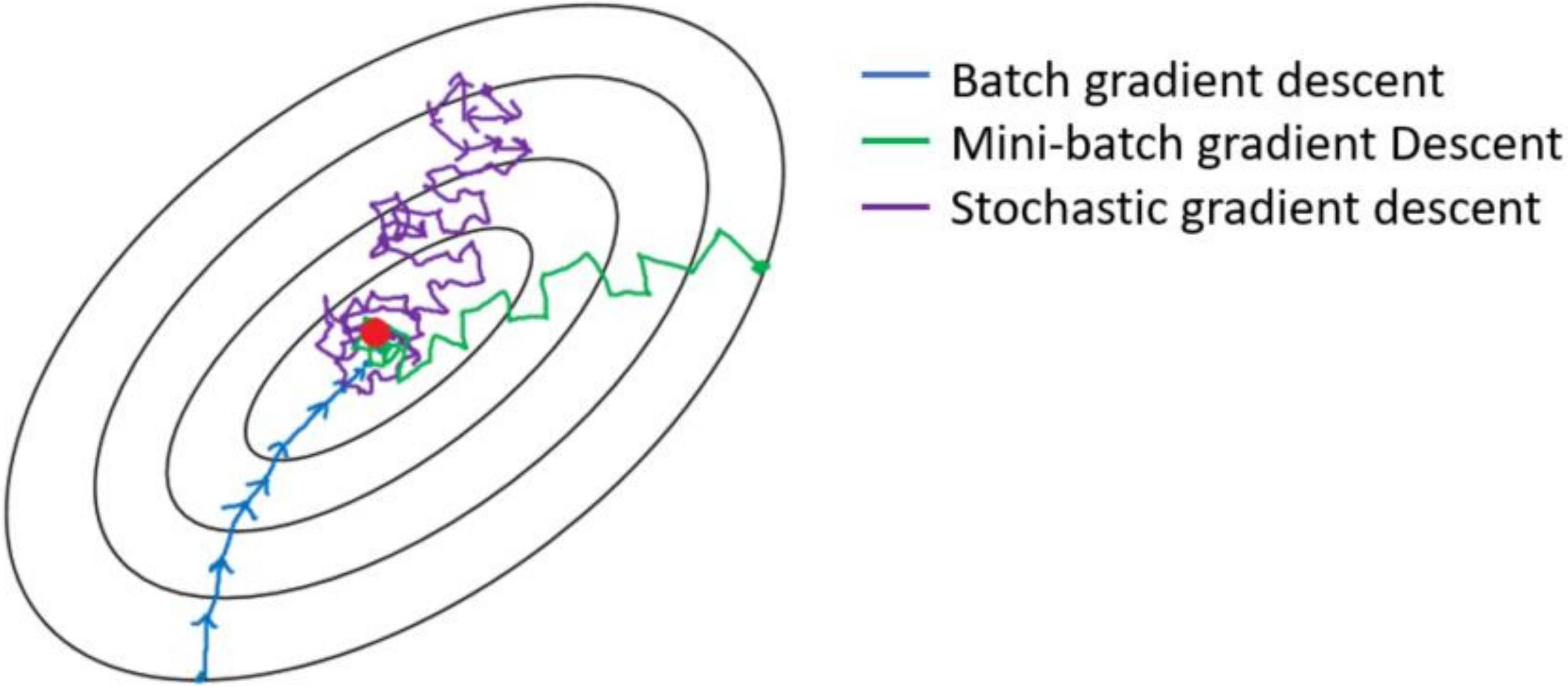
$$\nabla C = \frac{1}{n}\sum_{i} \nabla C^{i} \approx \frac{1}{m}\sum_{i} \nabla C^{i}$$

- This subset is referred as mini-batch

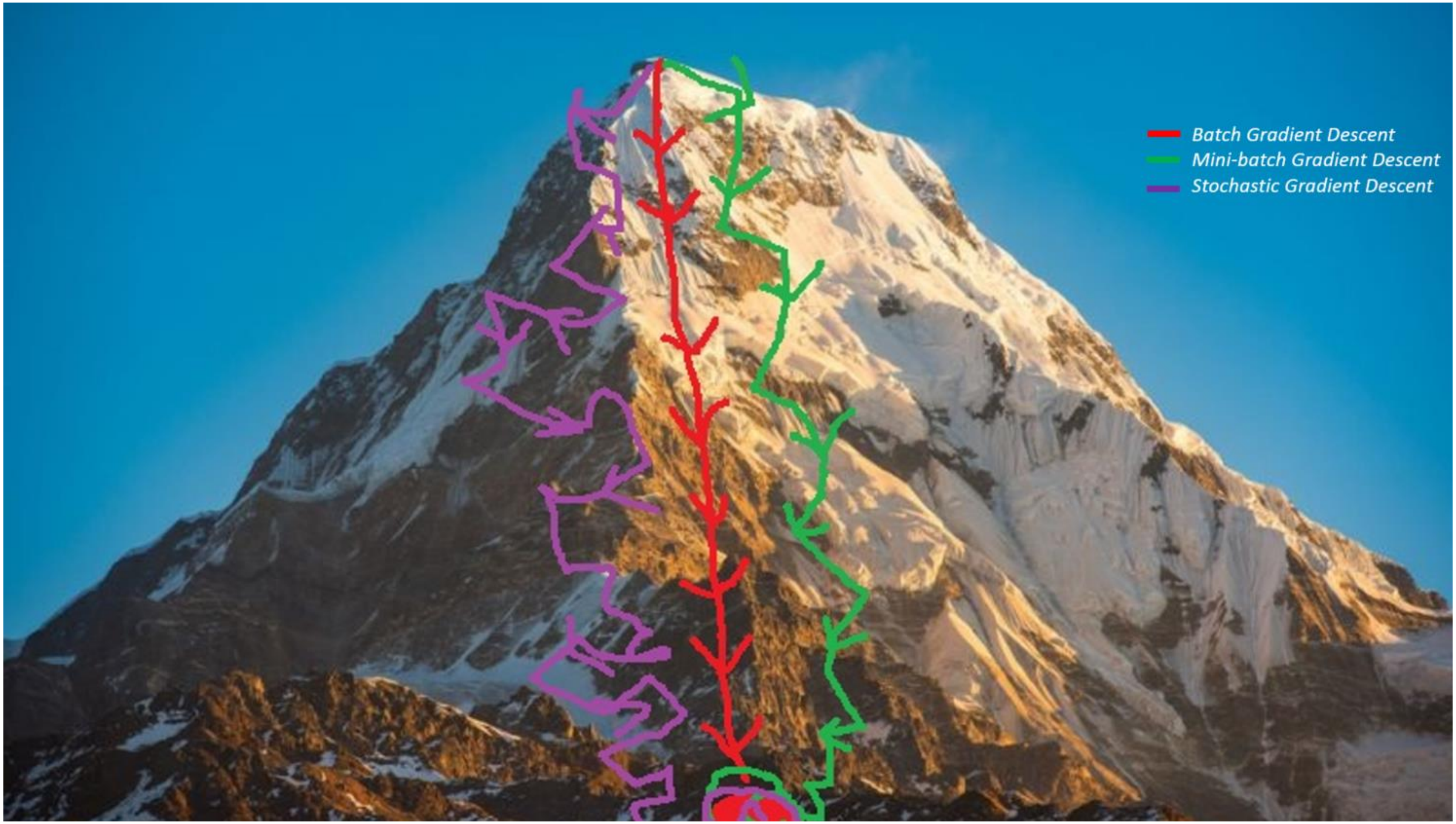
Mini-Batch Stochastic Gradient Descent

1. Initialize $\mathbf{w} := 0^{m-1}, b := 0$
2. for epoch $e \in [1, \ldots, E]$:
    - 2.1. shuffle $\mathcal{D}$ to prevent cycles
    - 2.2. for $i \in [1, \ldots, m]$ (where m is the minibatch size):
        - 2.2.1. draw random example **without** replacement: $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
    - 2.3. compute loss $\mathcal{L} := \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{[i]}, y^{[i]})$
    - 2.4. compute gradients $\Delta \mathbf{w} := -\nabla_{\mathcal{L}} \mathbf{w}, \ \Delta b := -\frac{\partial \mathcal{L}}{\partial b}$
    - 2.5. update parameters $\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \ b := +\Delta b$

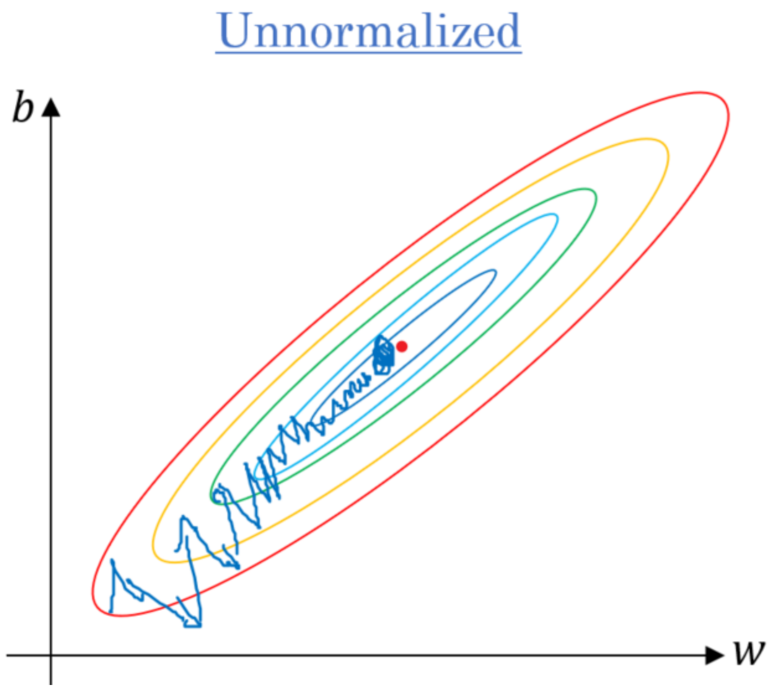
Mini-Batch Stochastic Gradient Descent

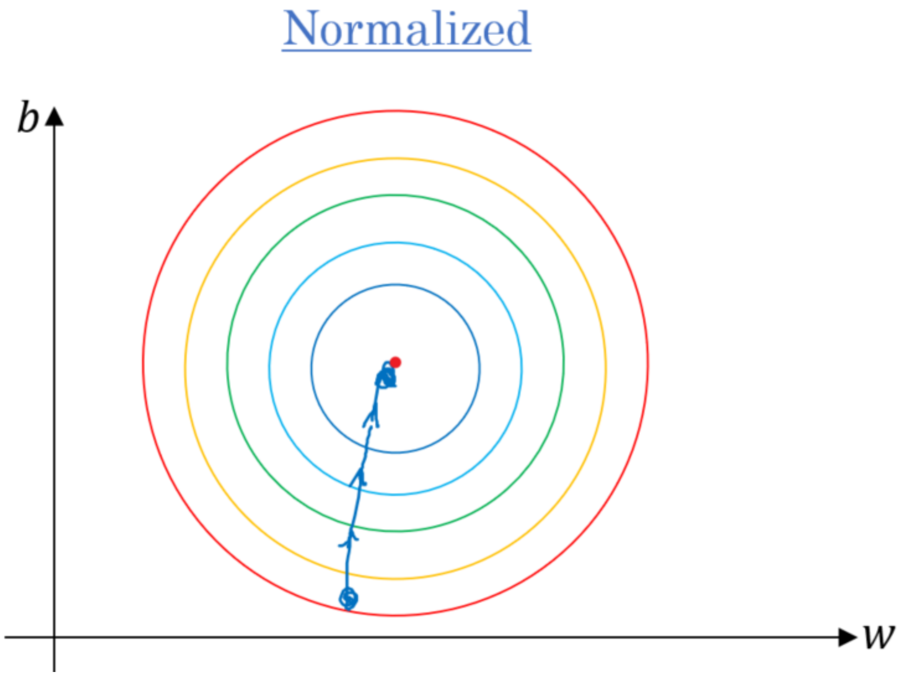


Mini-Batch Stochastic Gradient Descent



Scaling the Data





Gradient descent in multi-layer nets

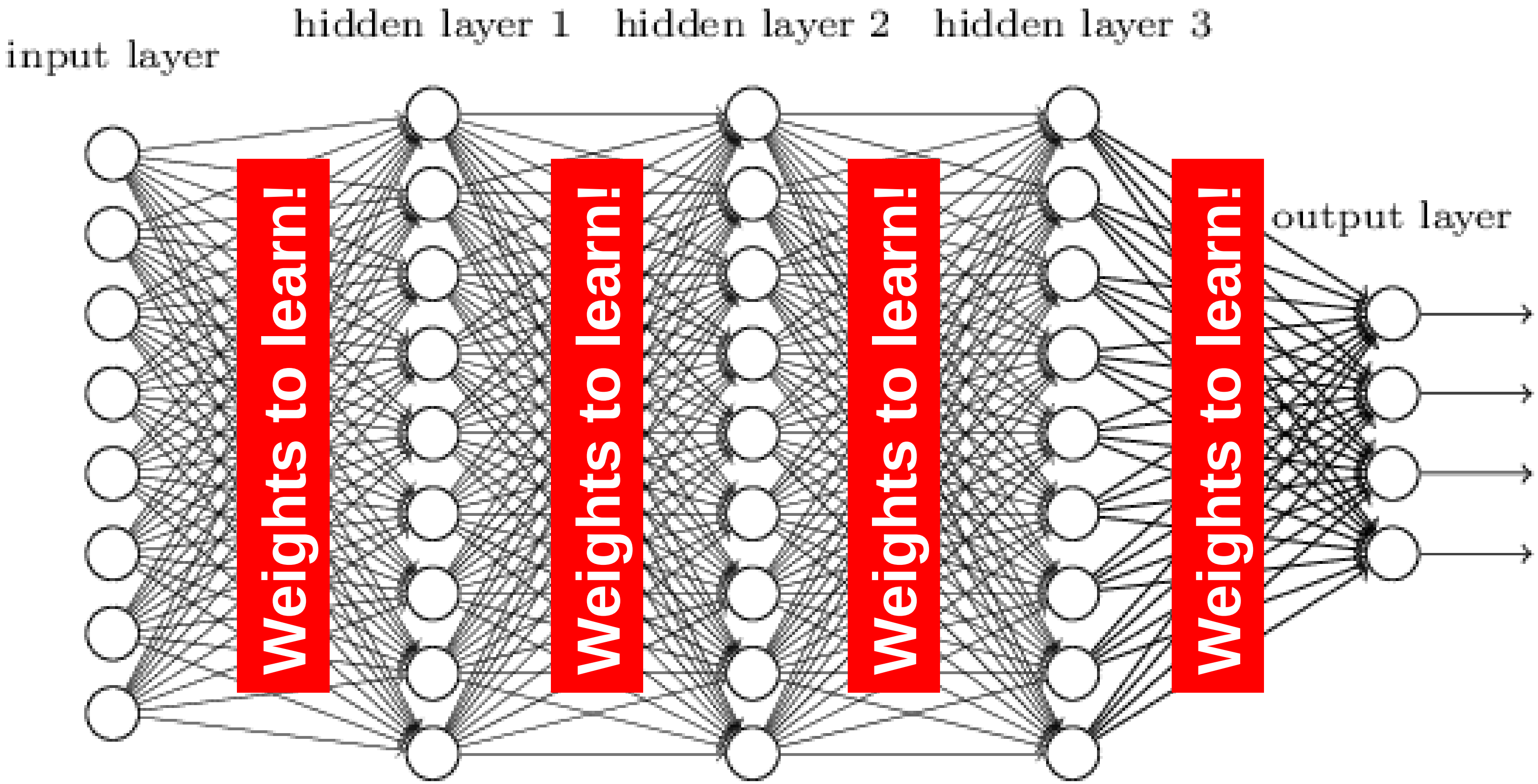
- We'll update weights
- Move in direction opposite to gradient:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

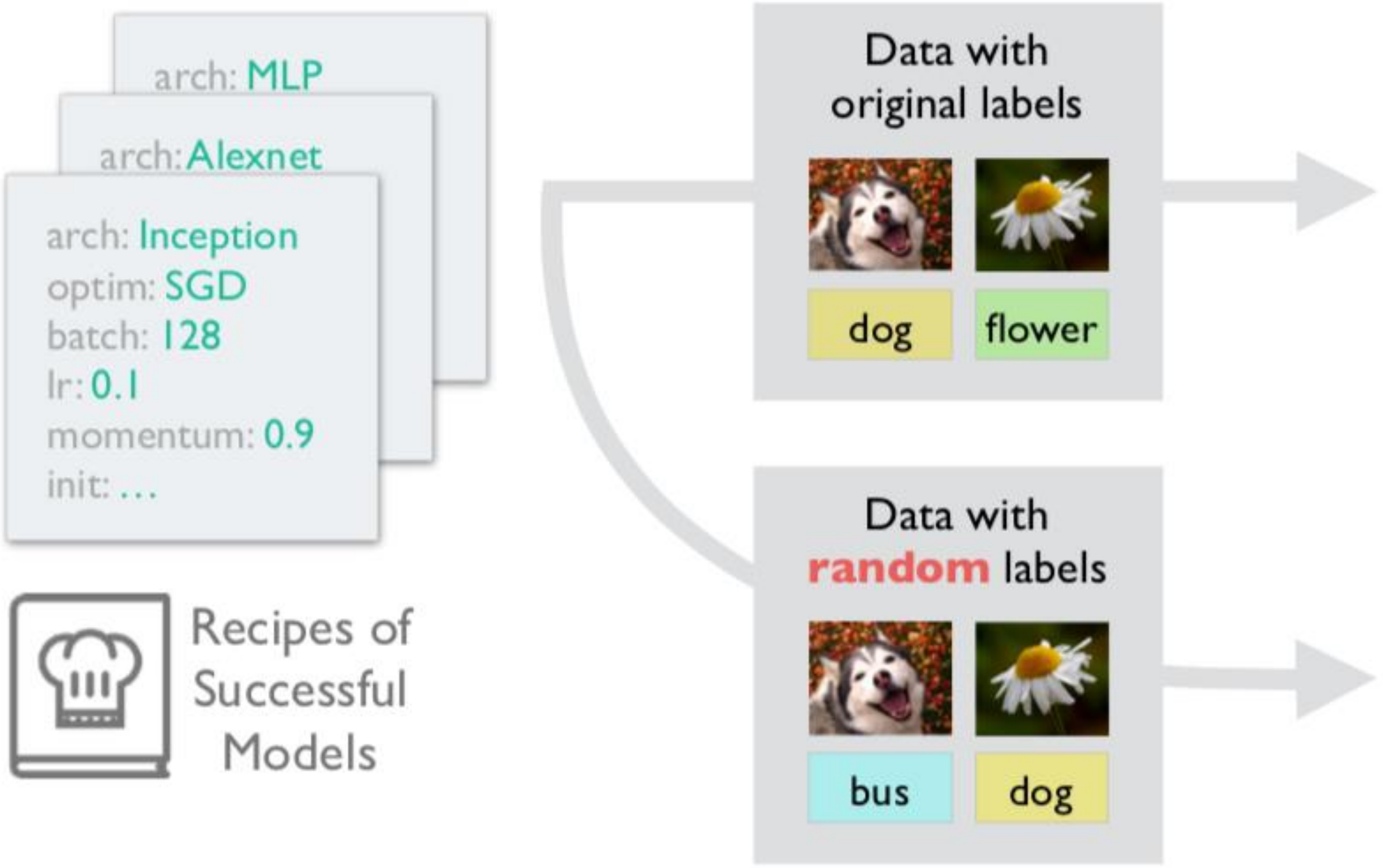
- How to update the weights at all layers?
- Answer: backpropagation of error from higher layers to lower layers

Deep neural networks

- Lots of hidden layers
- Depth = power (usually)



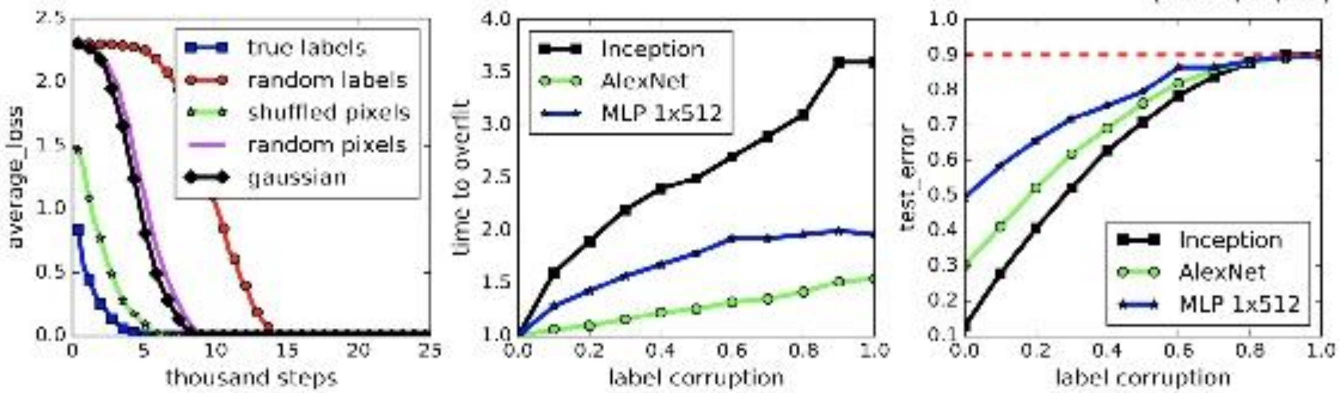
Deep neural networks – Too Good ?



Deep neural networks – Too Good ?

Random Labeling of True Data

(from paper)



(a) learning curves (b) convergence slowdown (c) generalization error growth

But it still fits perfectly!
It memorizes all random labels!