

Building Regression Models using TensorFlow

LEARNING USING NEURONS

Overview

The most common use of TensorFlow is in deep learning

Neural networks are the most common type of deep learning algorithm

The basic building block of a neural network is a neuron

Linear regression can be “learnt” using a single neuron

Deep learning extends this idea to more complex, non-linear functions

Prerequisites and Course Outline

Course Outline

Learning using Neurons

Linear Regression in TensorFlow

Logistic Regression in TensorFlow

Estimators

Related Courses

Understanding the Foundations of TensorFlow

Understanding and Applying Linear Regression

Understanding and Applying Logistic Regression

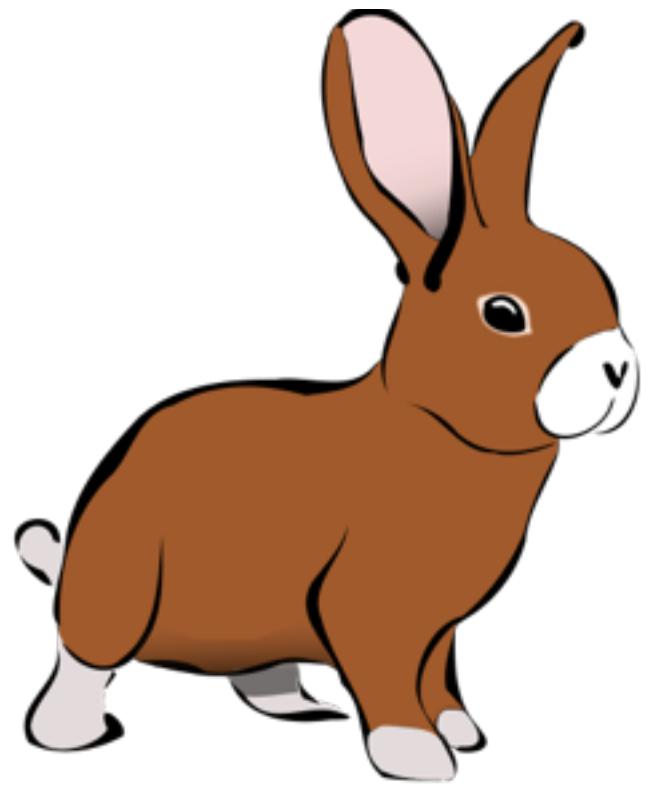
Software and Skills

Have TensorFlow installed

Know some Python programming

Understanding Deep Learning

Whales: Fish or Mammals?



Mammals

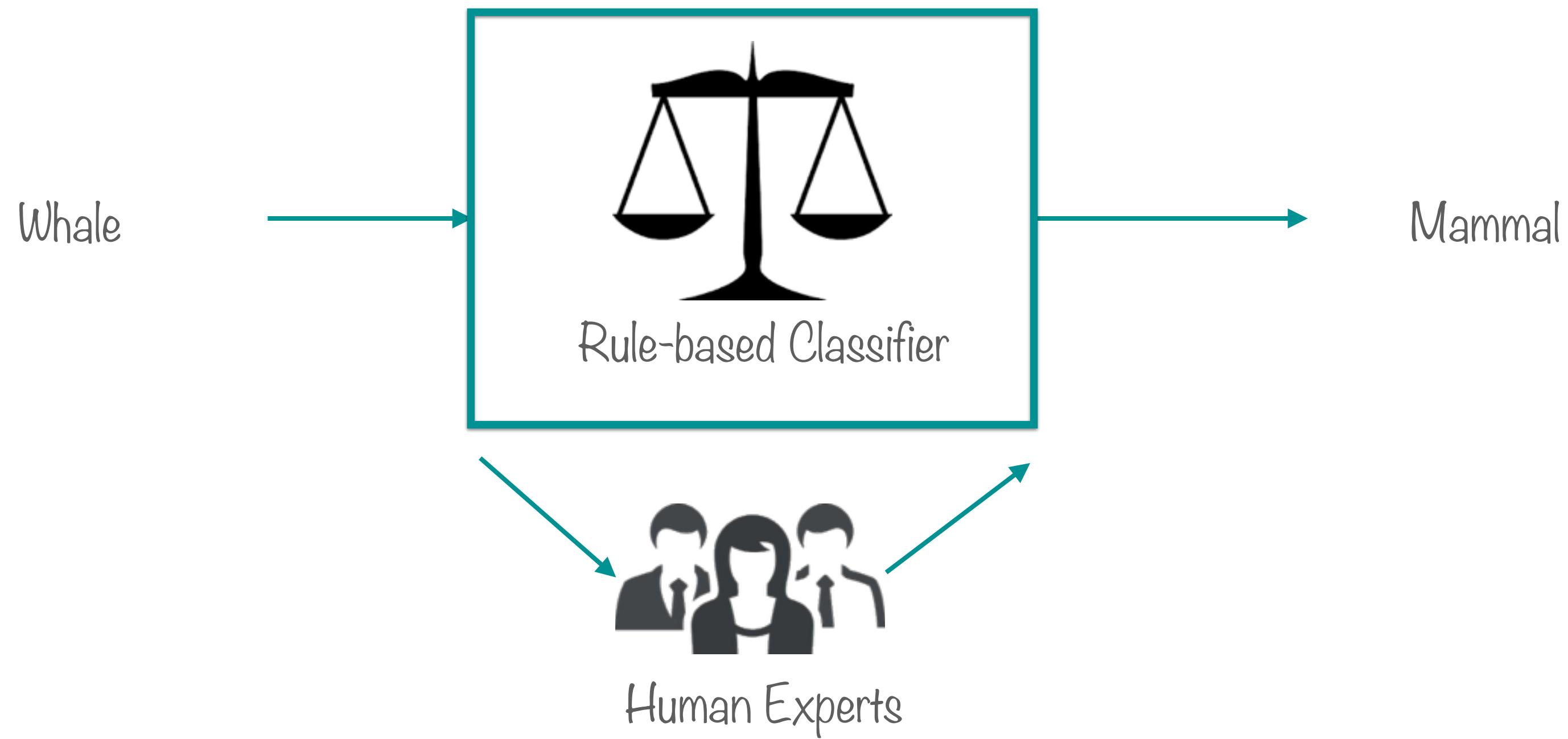
Members of the infraorder Cetacea



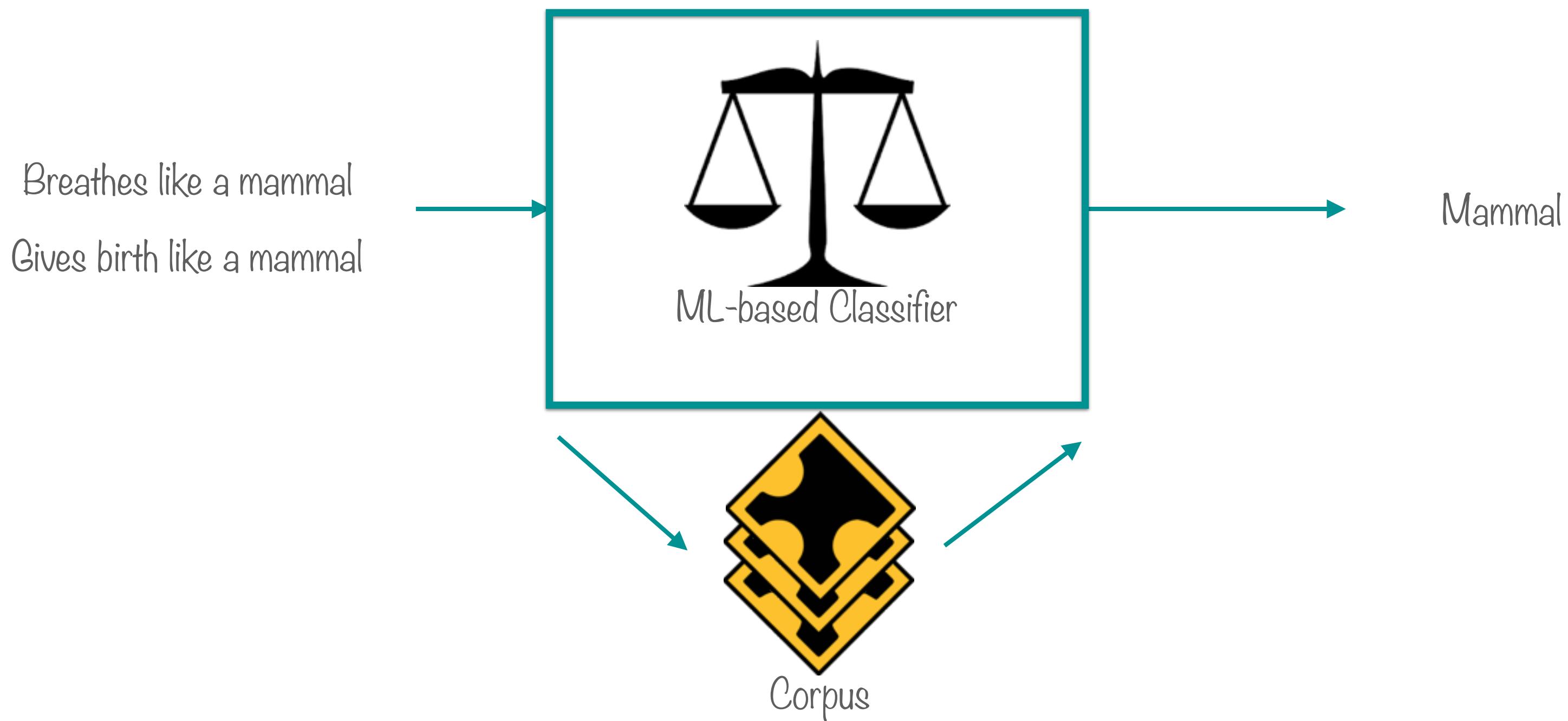
Fish

Look like fish, swim like fish, move with fish

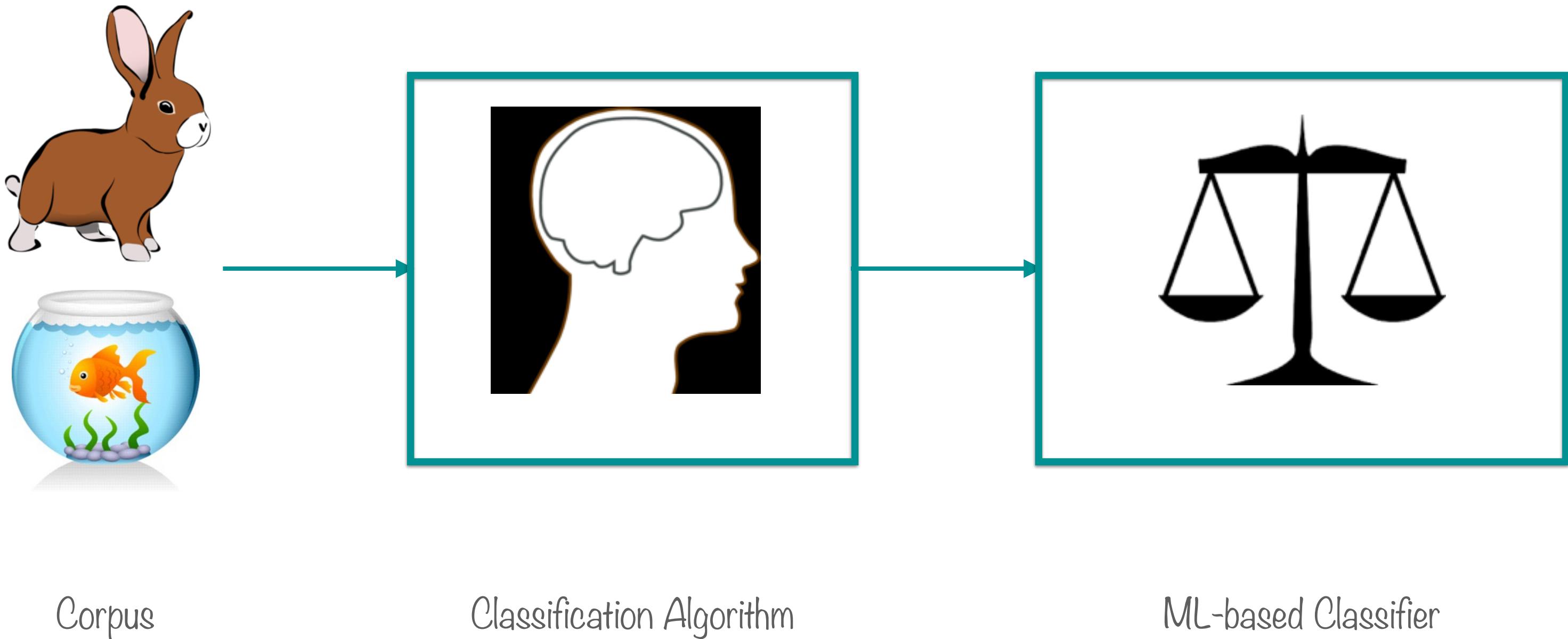
Rule-based Binary Classifier



“Traditional” ML-based Binary Classifier



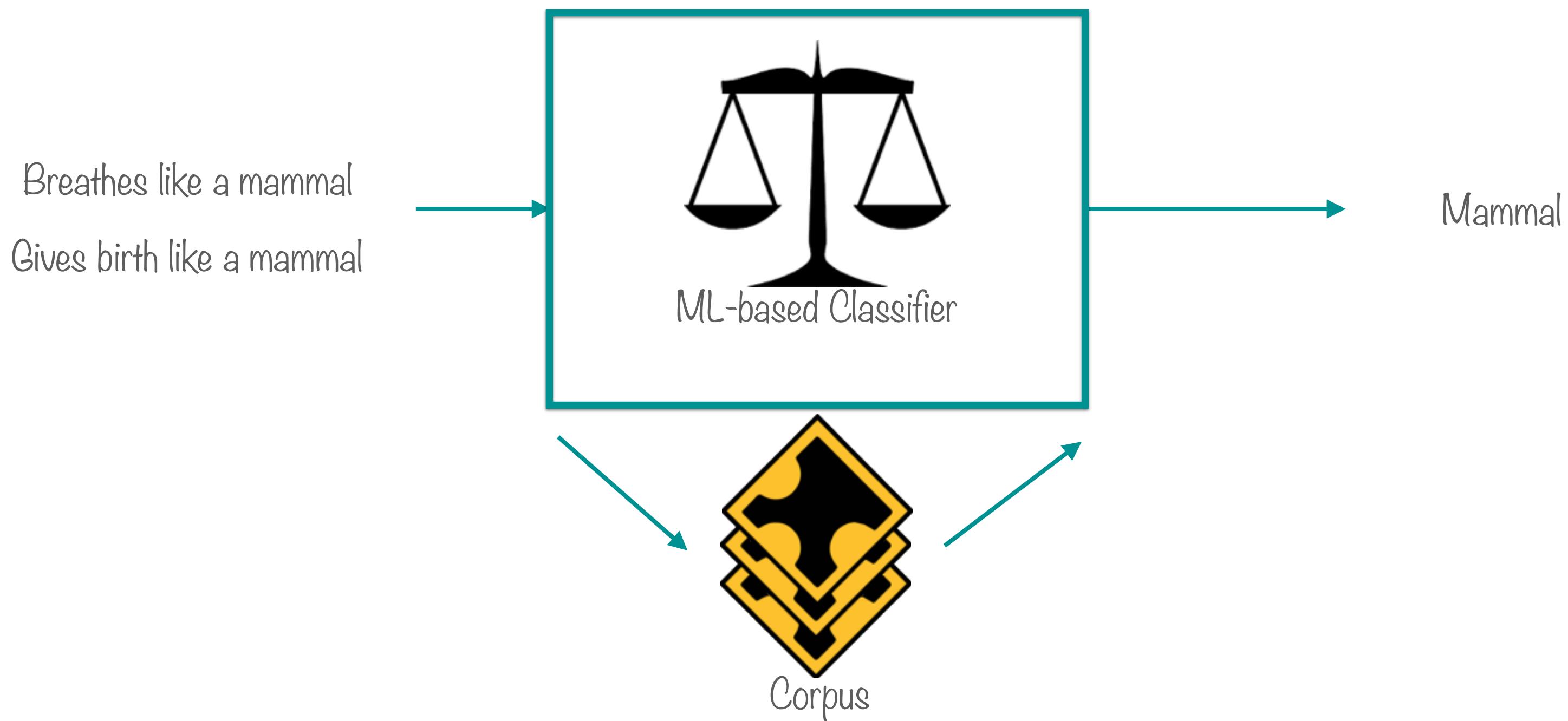
“Traditional” ML-based Binary Classifier



“Traditional” ML-based Binary Classifier

| | |
|------------------------|------------------|
| “Traditional” ML-based | Rule-based |
| Dynamic | Static |
| Experts optional | Experts required |
| Corpus required | Corpus optional |
| Training step | No training step |

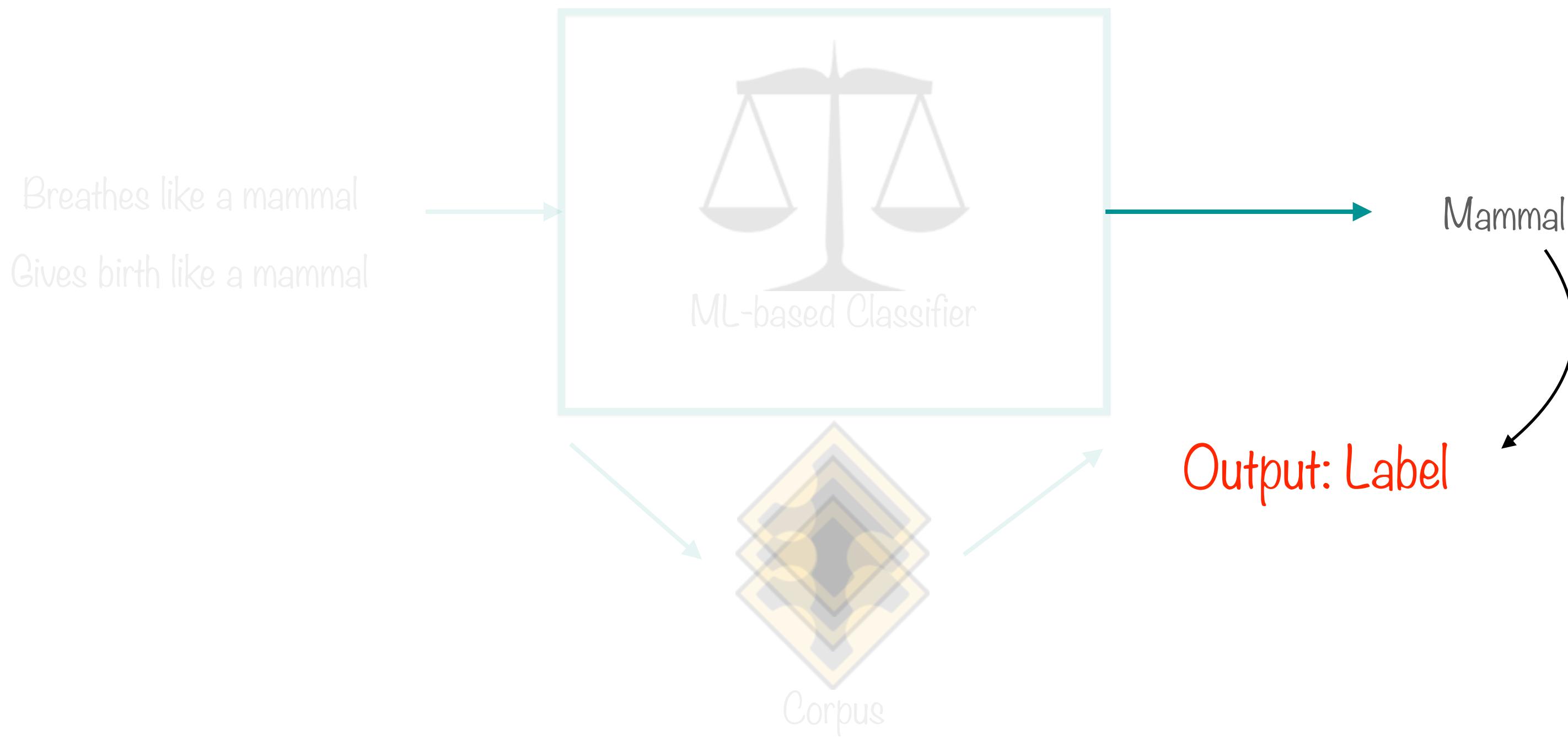
“Traditional” ML-based Binary Classifier



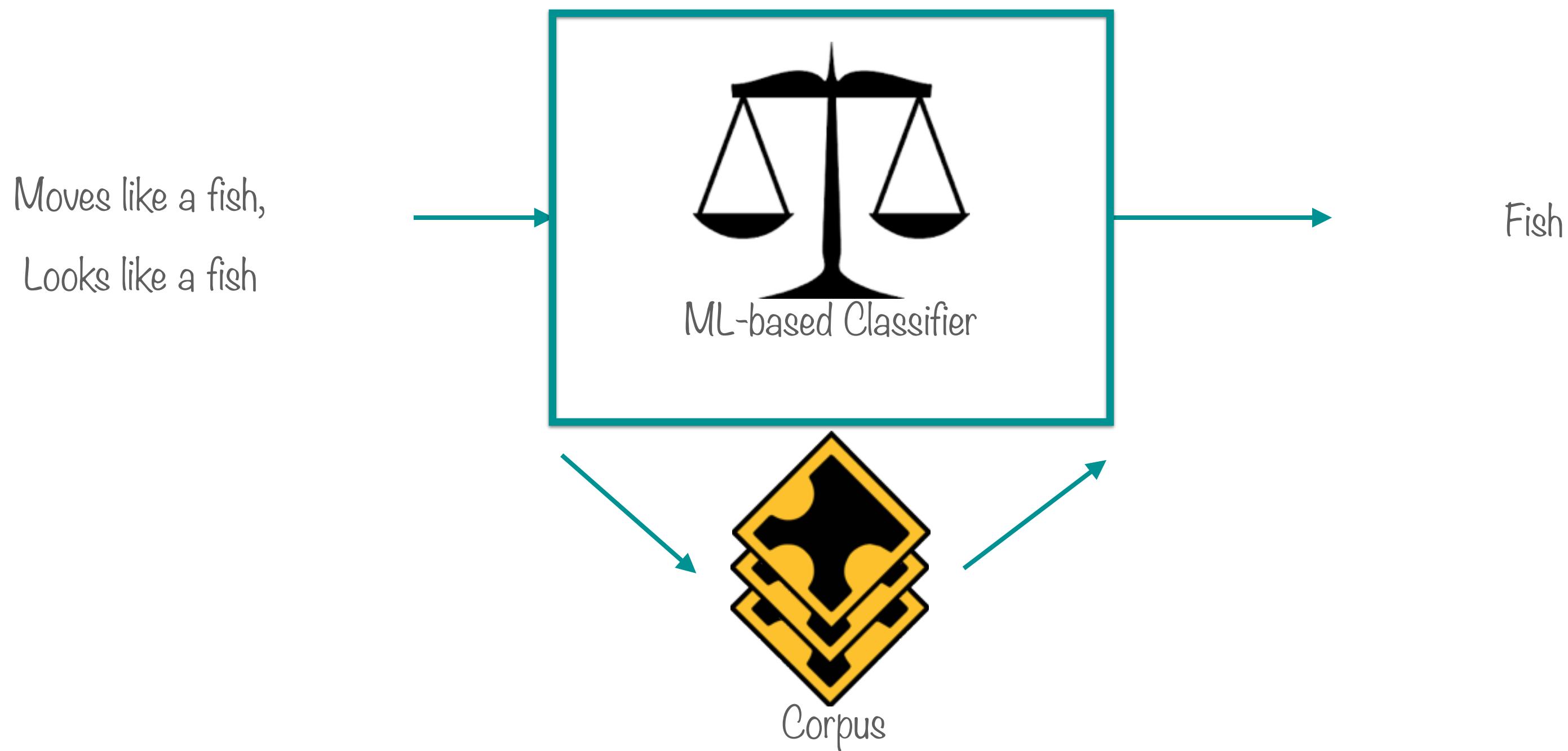
“Traditional” ML-based Binary Classifier



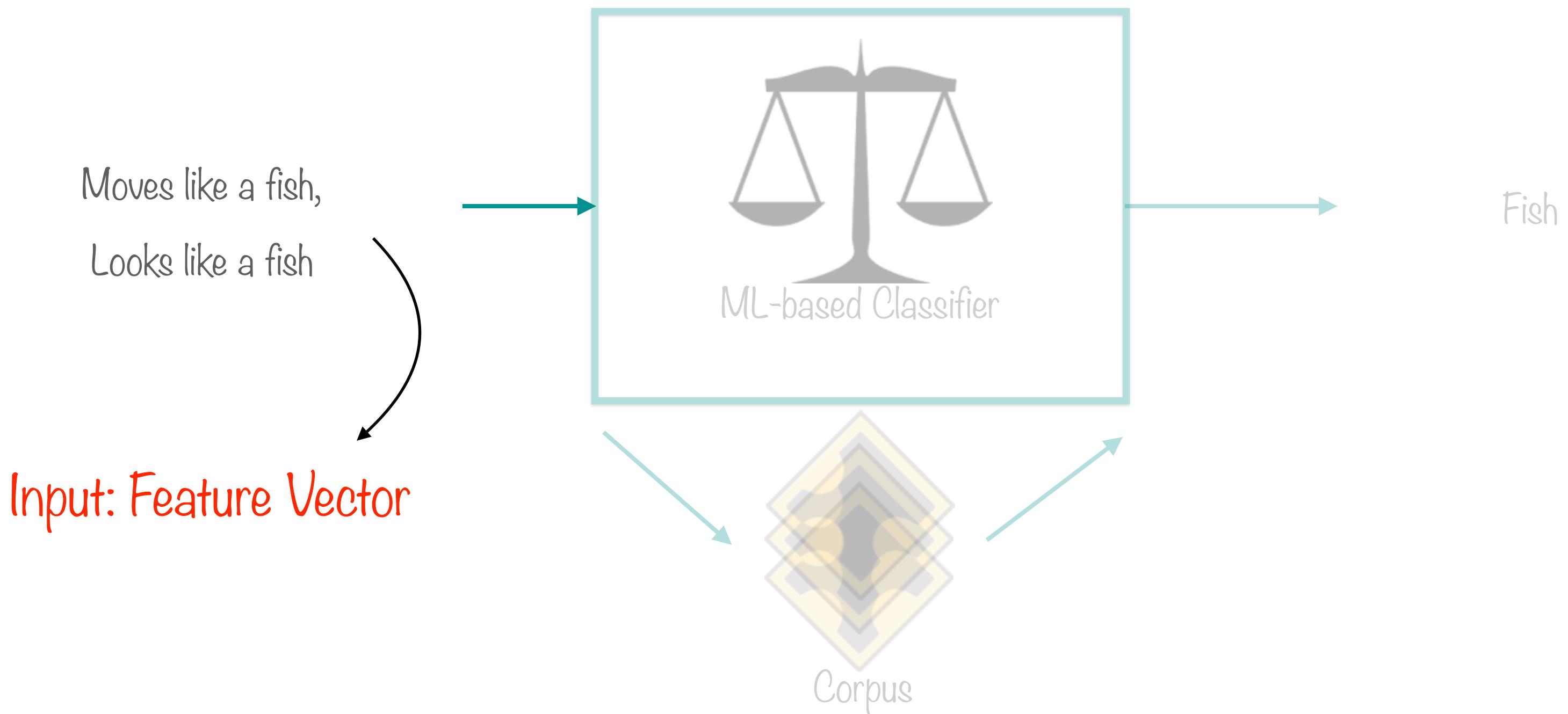
“Traditional” ML-based Binary Classifier



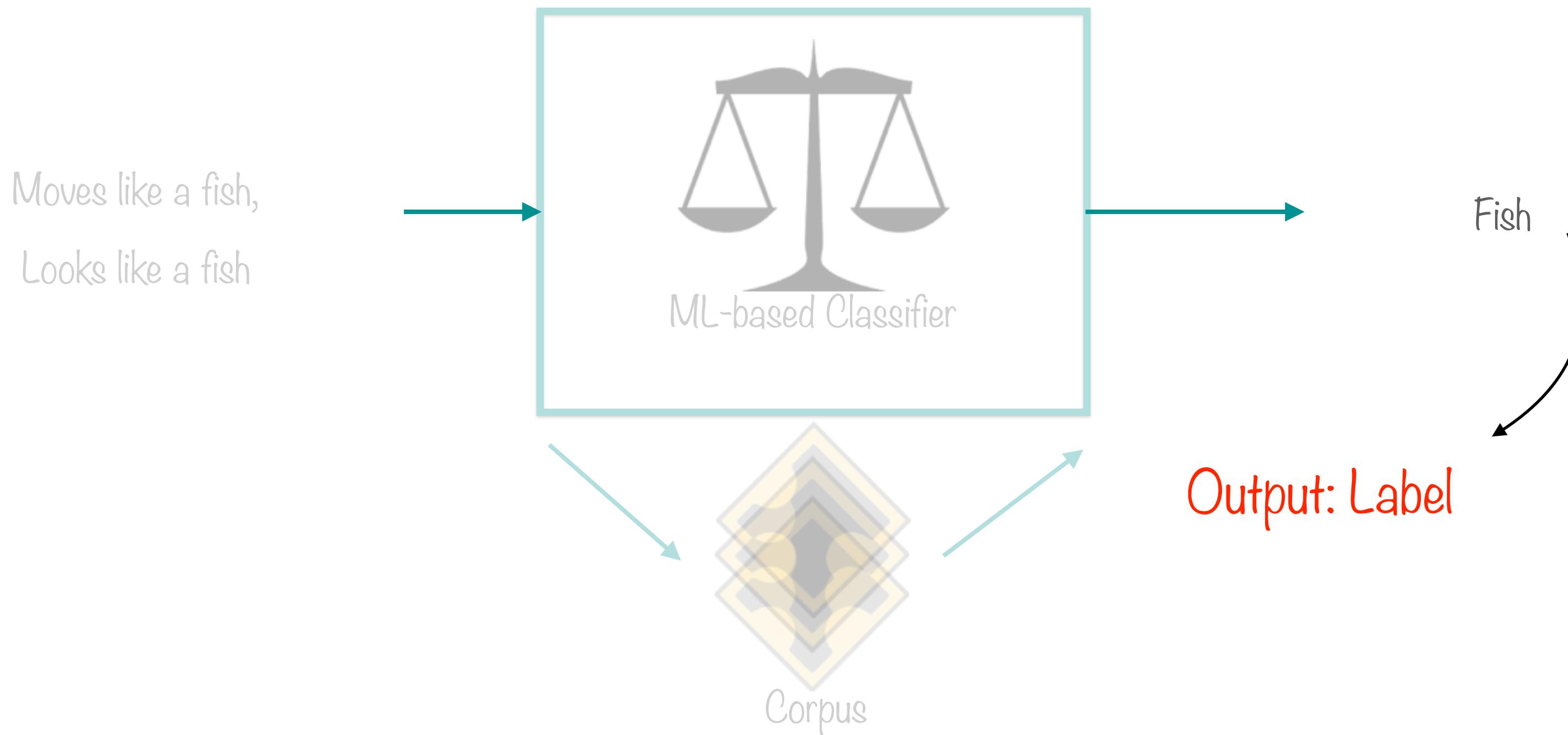
“Traditional” ML-based Binary Classifier



“Traditional” ML-based Binary Classifier



“Traditional” ML-based Binary Classifier



Feature Vectors

The attributes that the ML algorithm focuses on are called features

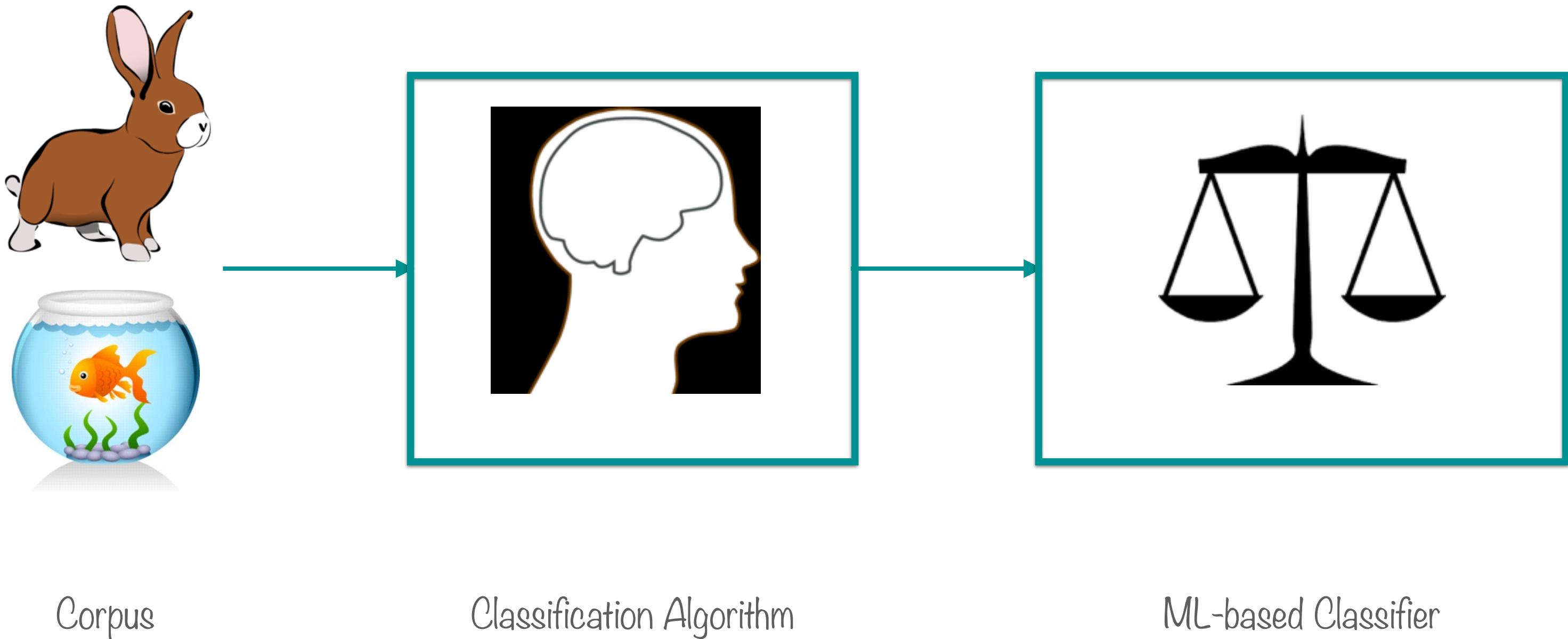
Each data point is a list - or vector - of such features

Thus, the input into an ML algorithm is a feature vector

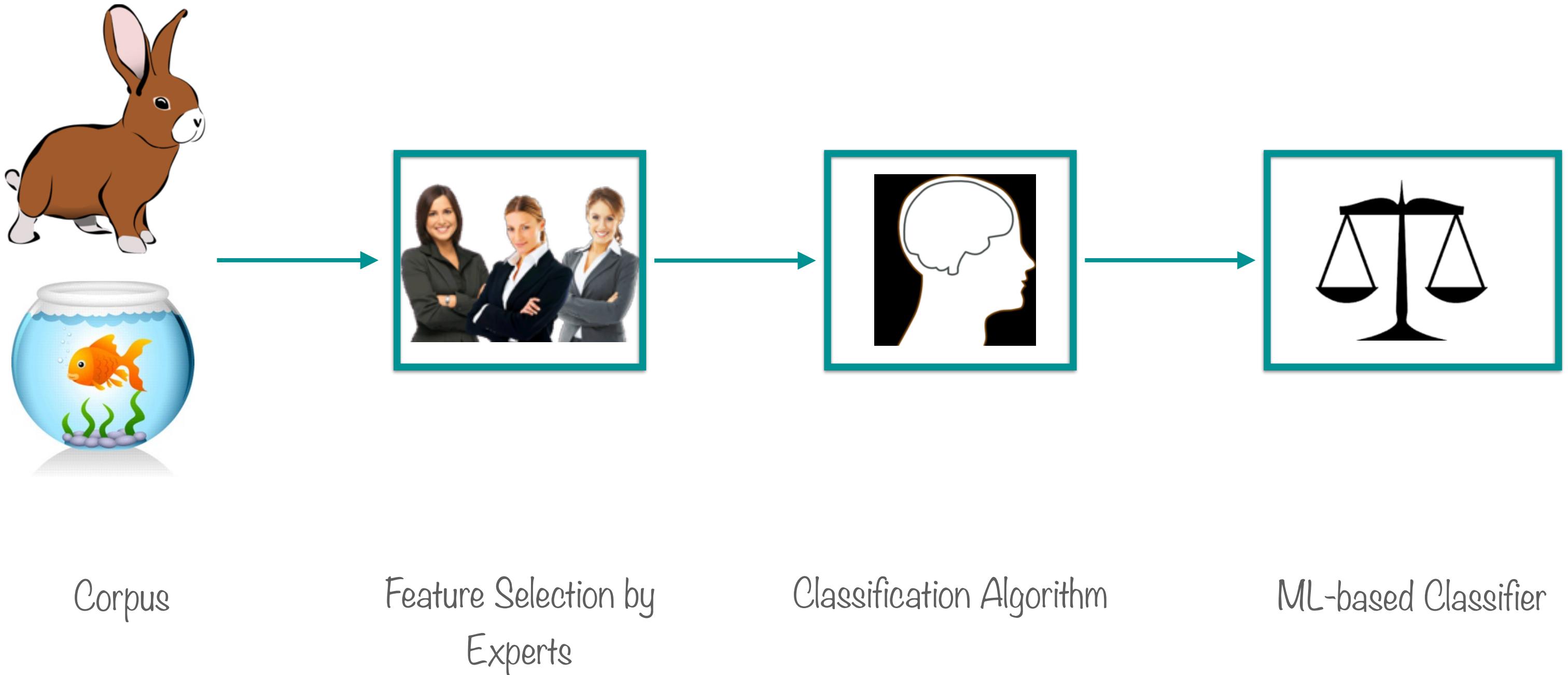
“Traditional” ML-based systems still rely on experts to decide what features to pay attention to

“Representation” ML-based systems figure out by themselves what features to pay attention to

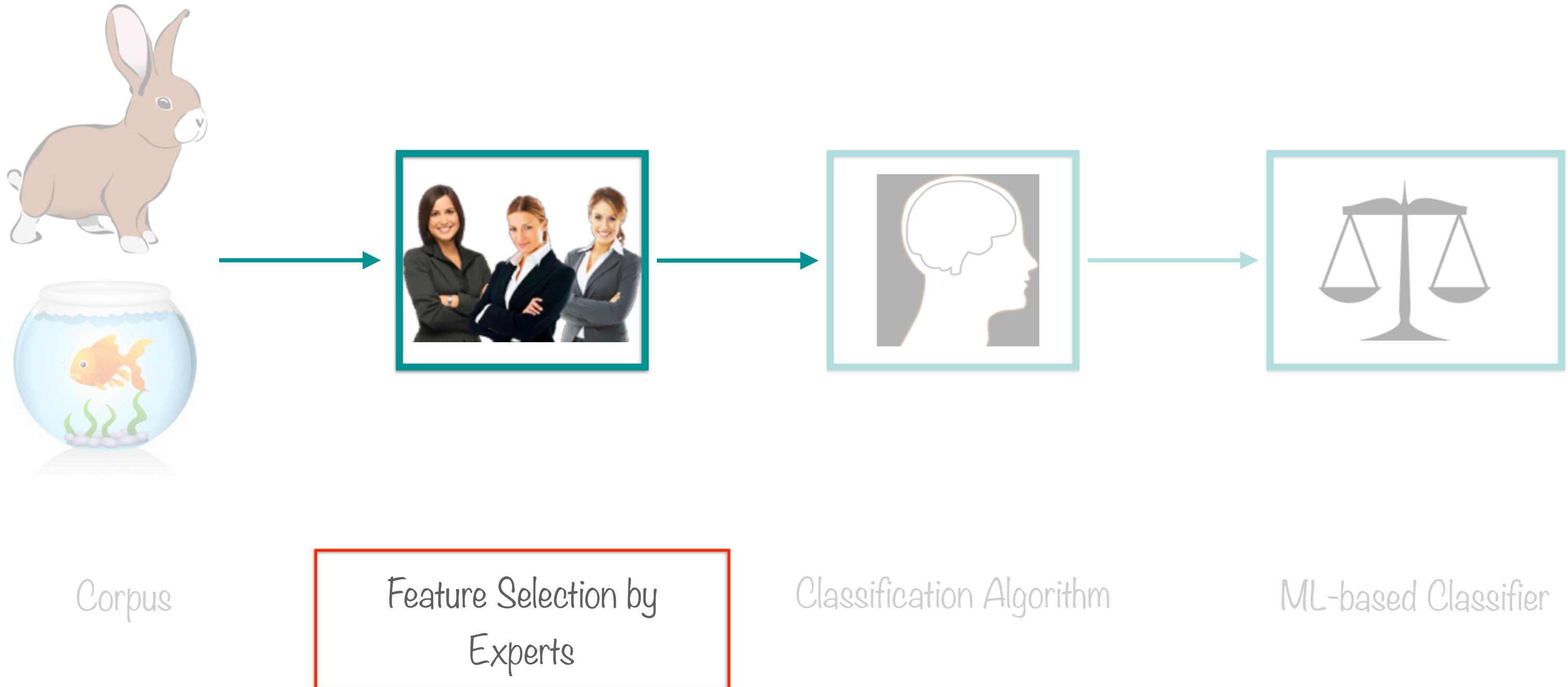
“Traditional” ML-based Binary Classifier



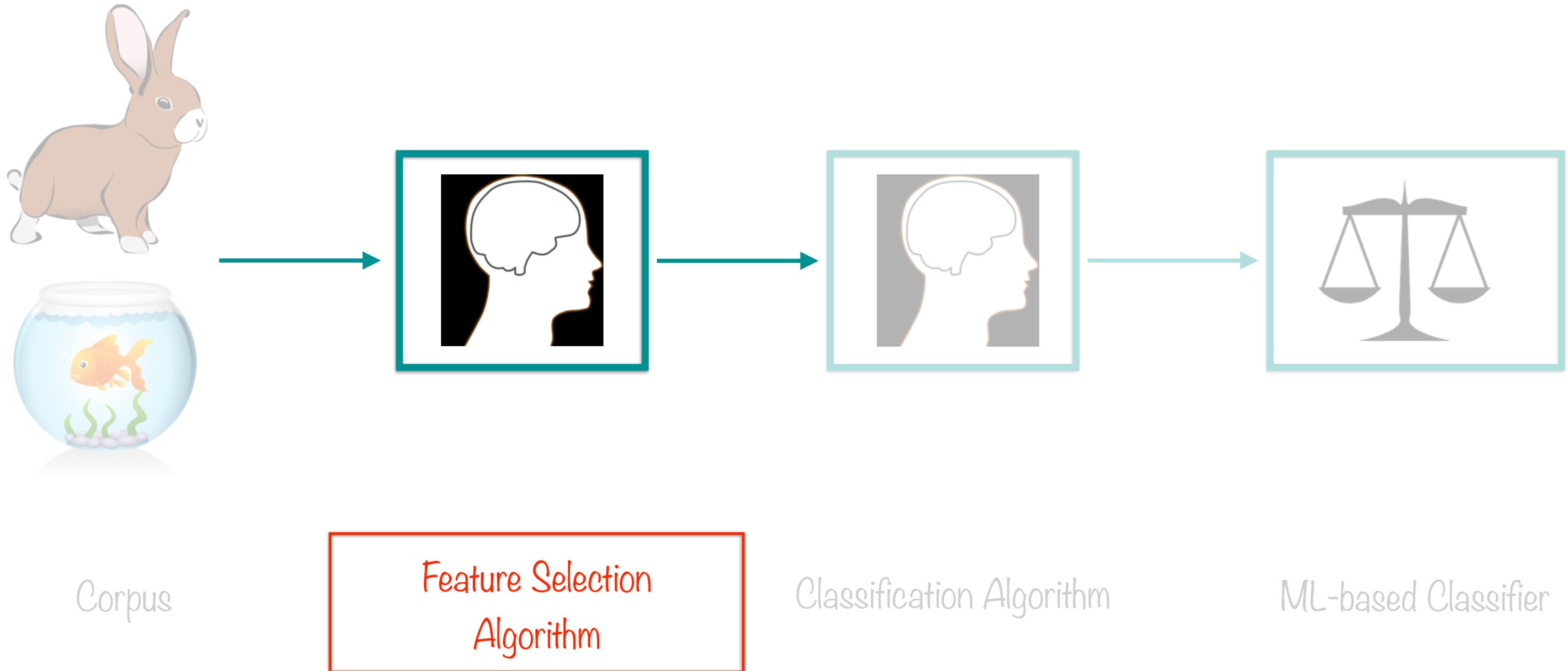
“Traditional” ML-based Binary Classifier



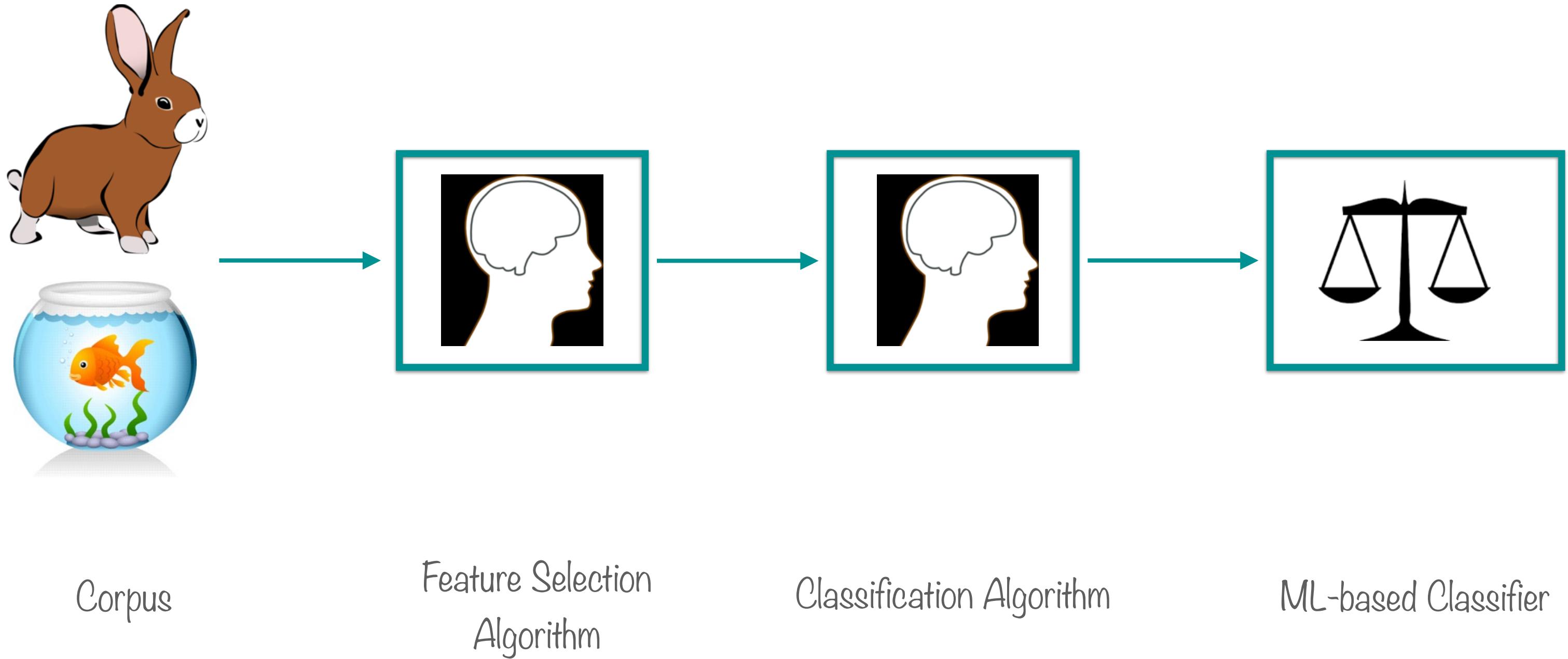
“Traditional” ML-based Binary Classifier



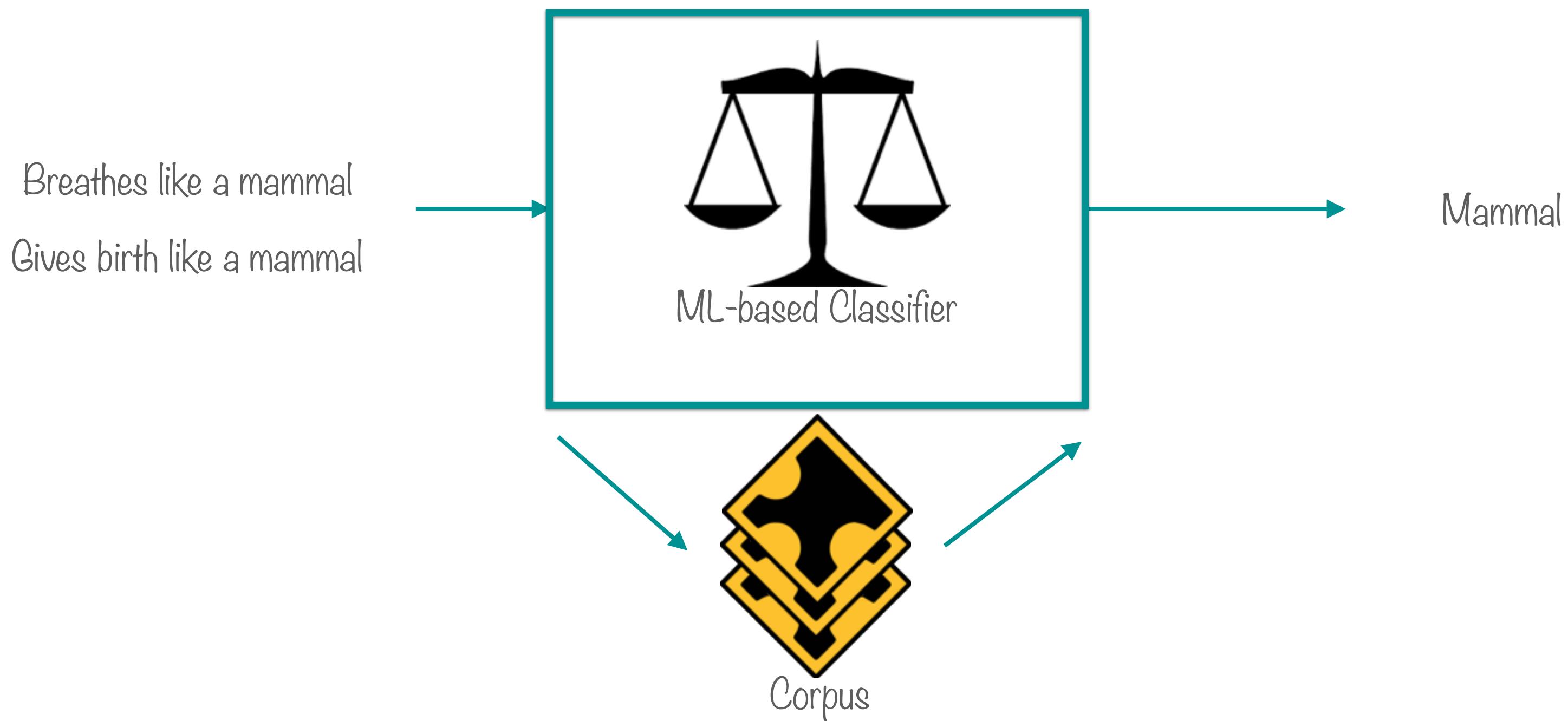
“Representation” ML-based Binary Classifier



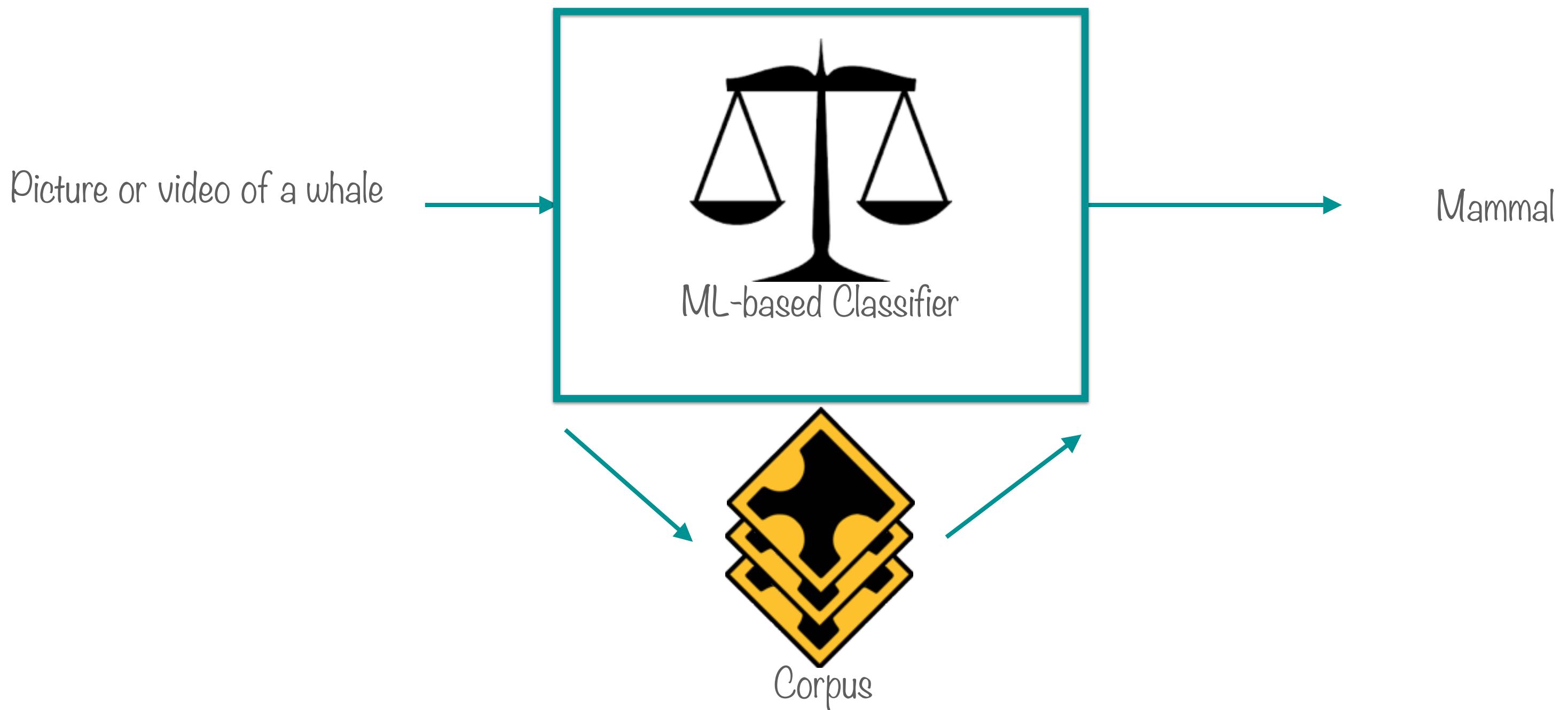
“Representation” ML-based Binary Classifier



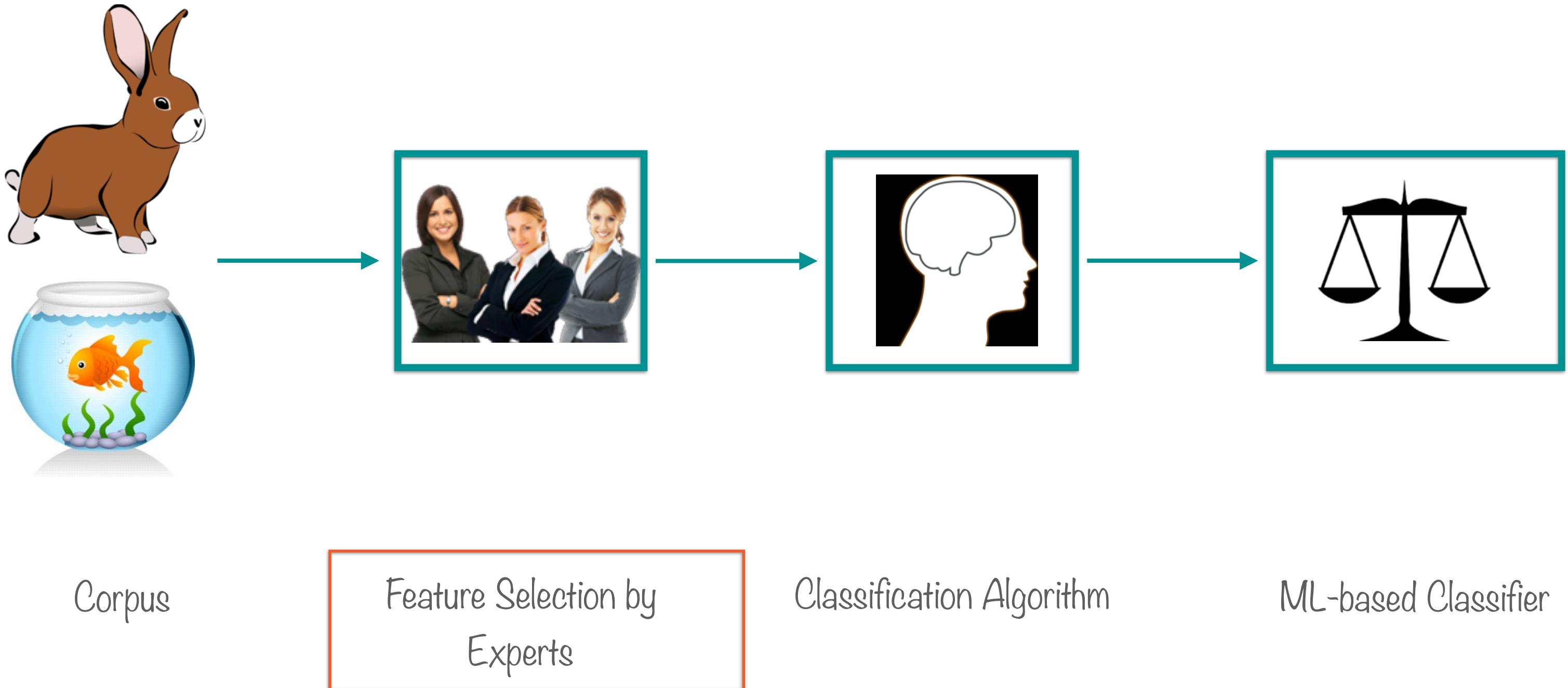
“Traditional” ML-based Binary Classifier



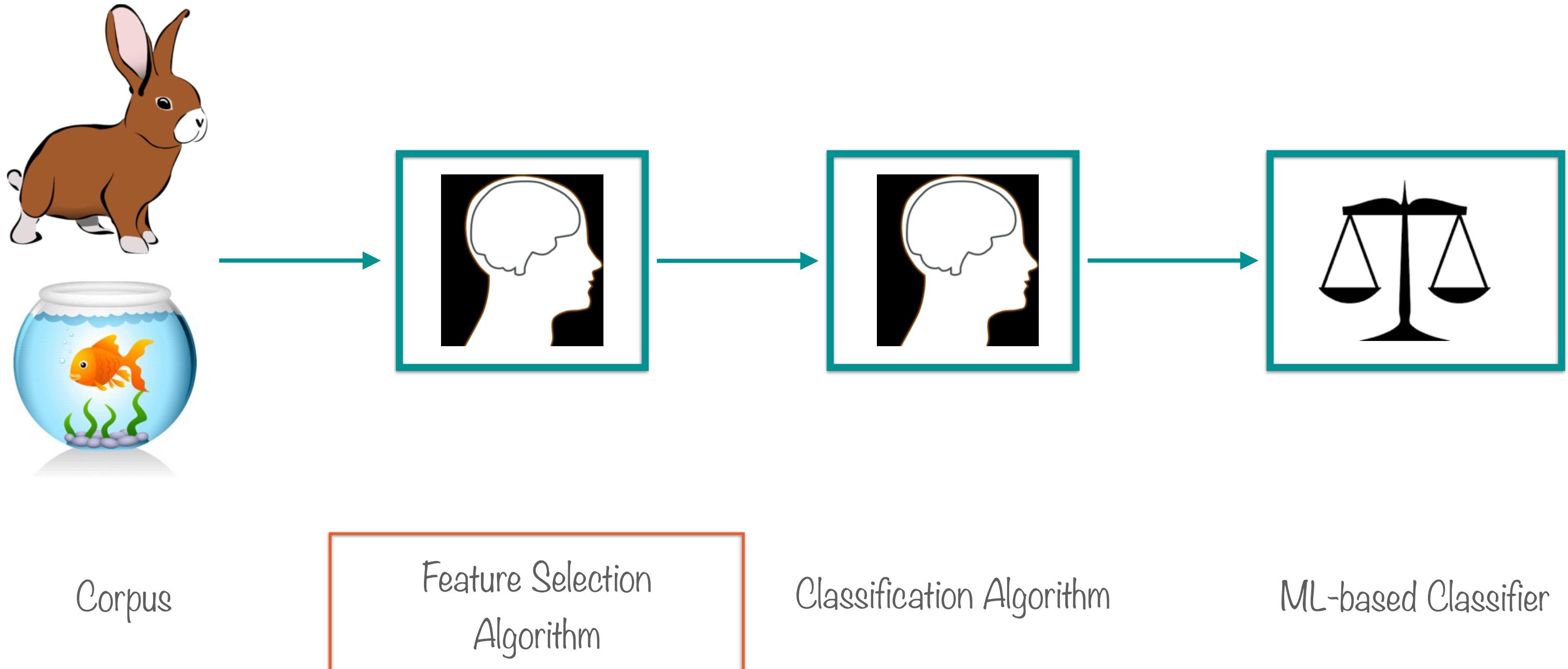
“Representation” ML-based Binary Classifier



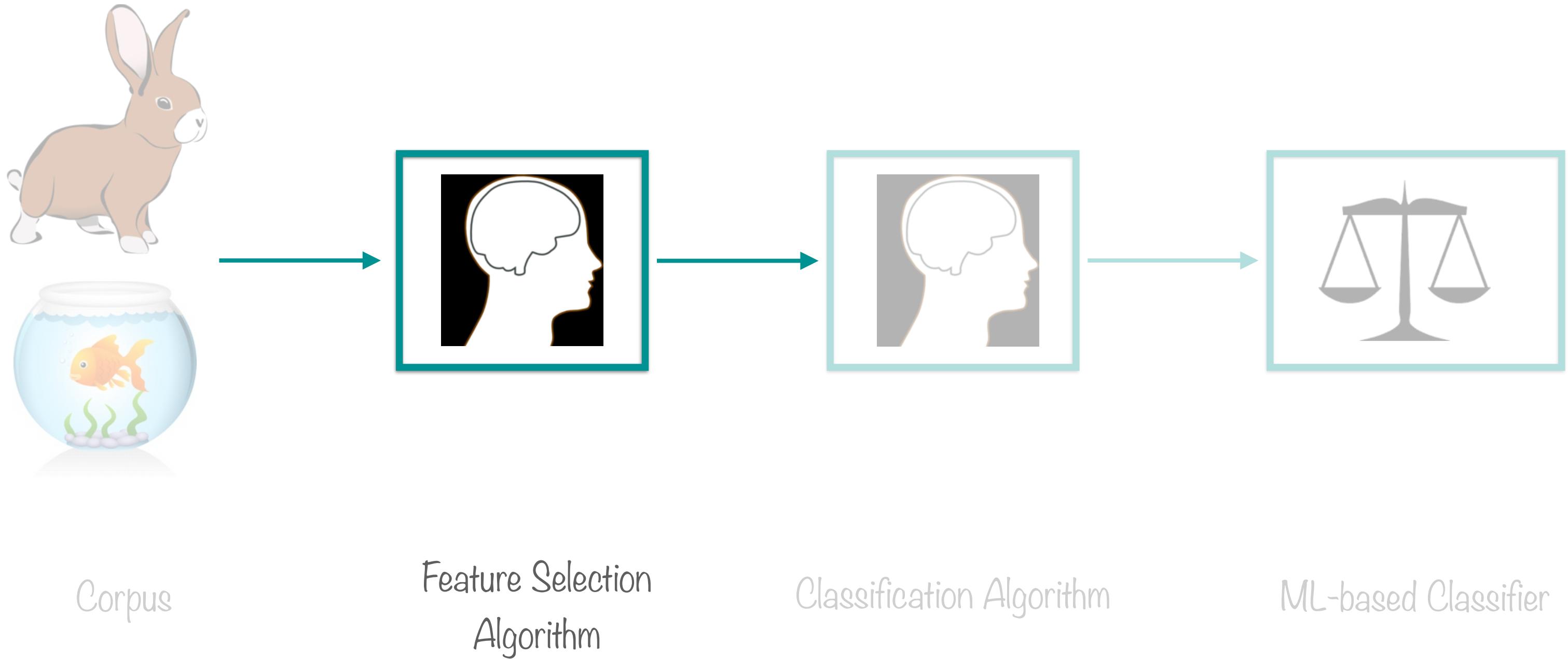
“Traditional” ML-based Binary Classifier



“Representation” ML-based Binary Classifier



“Representation” ML-based Binary Classifier



“Deep Learning” systems are one type of representation systems

Deep Learning and Neural Networks

Deep Learning

Algorithms that learn what
features matter

Neural Networks

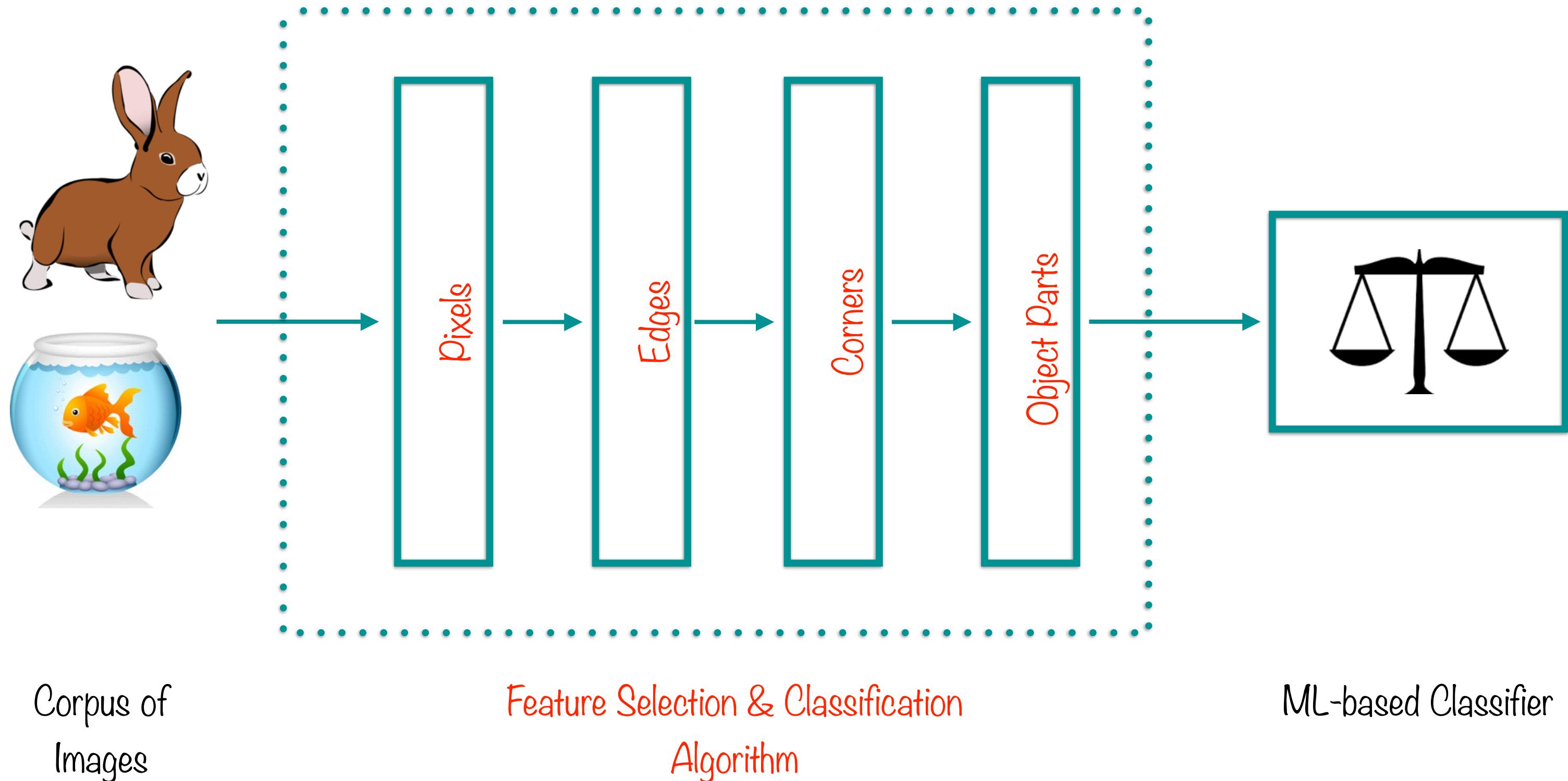
The most common class of deep
learning algorithms

Neurons

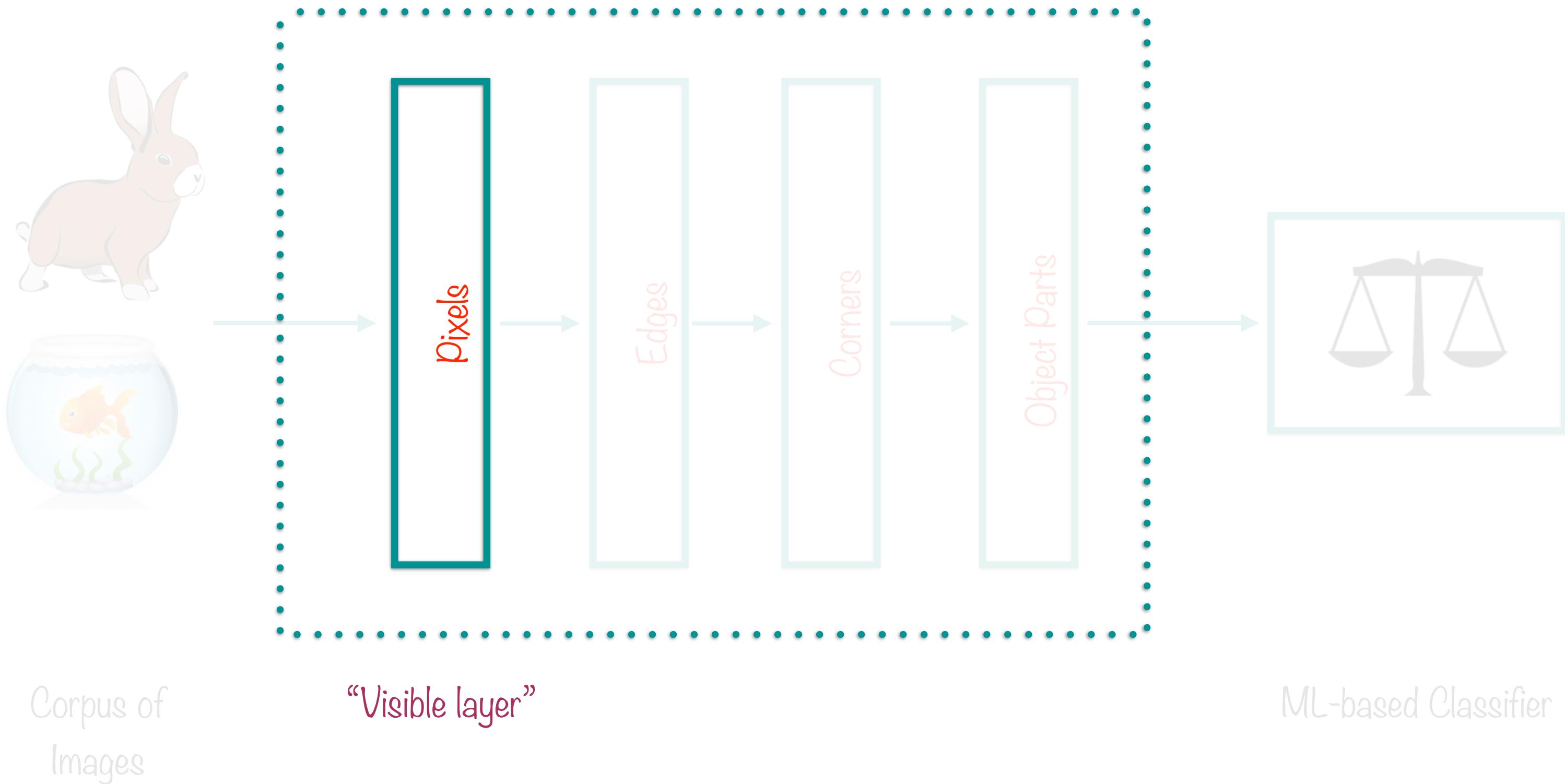
Simple building blocks that actually
“learn”

Deep Learning Book - Chapter 1 (intro), page 6

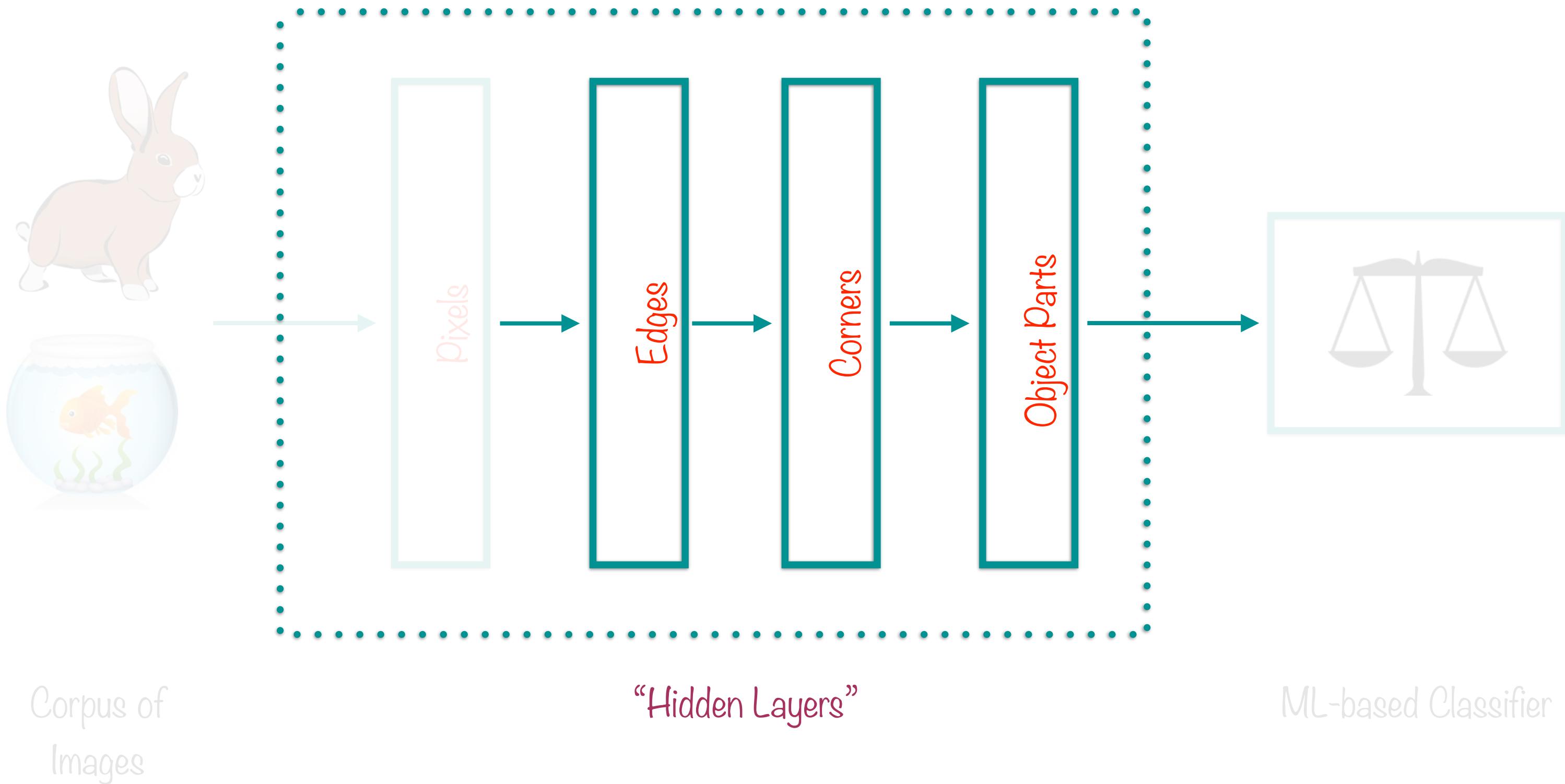
“Deep Learning”-based Binary Classifier



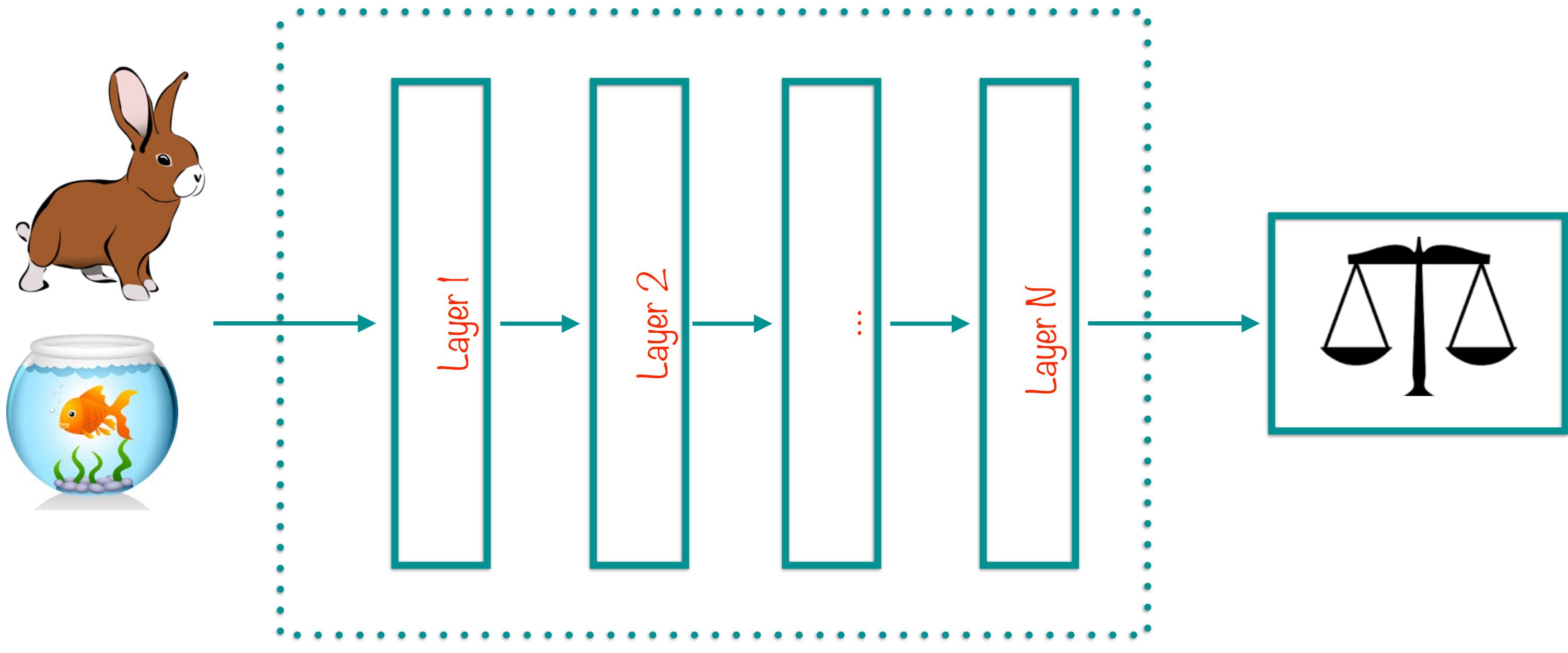
“Deep Learning”-based Binary Classifier



“Deep Learning”-based Binary Classifier



Neural Networks Introduced

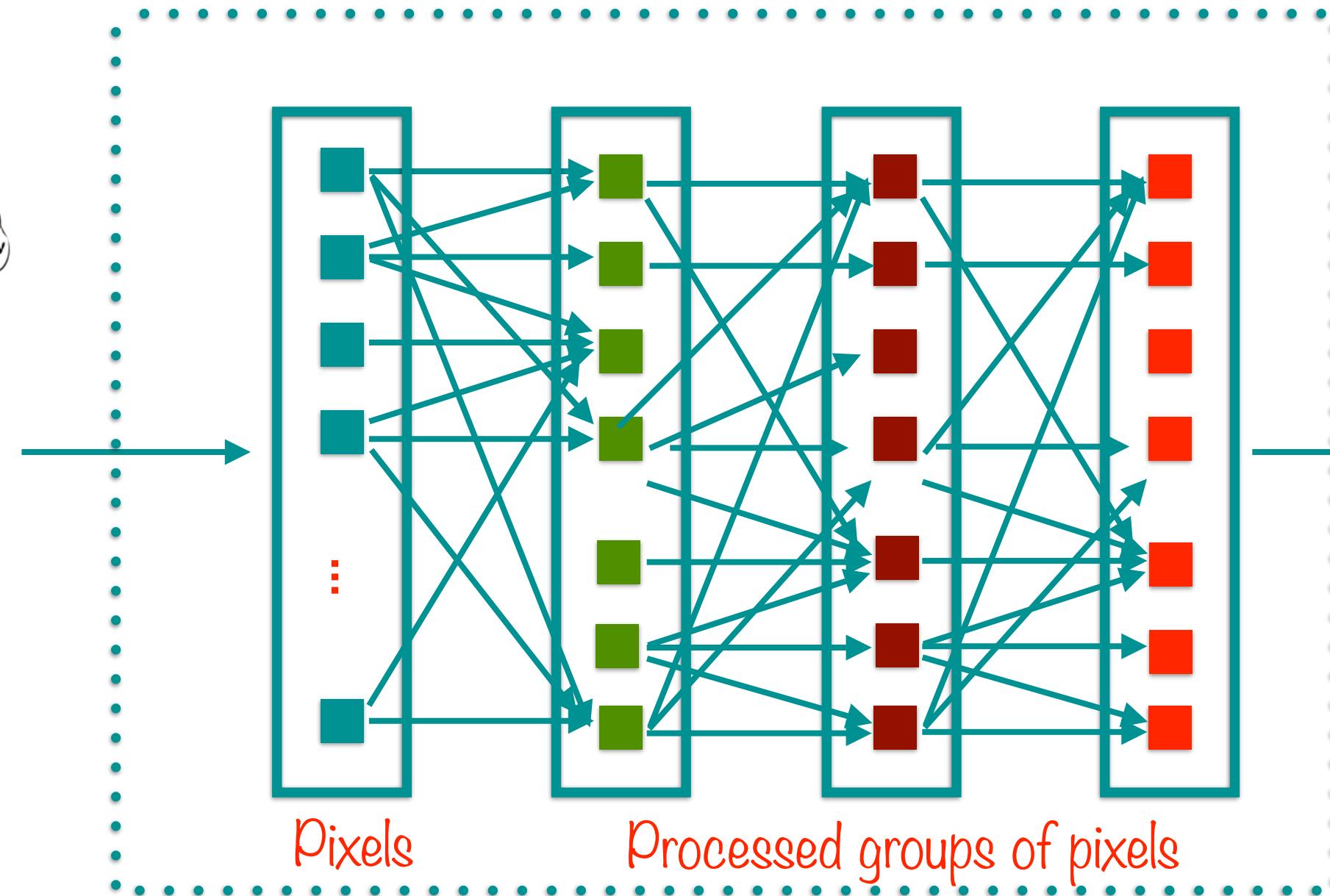


Corpus of
Images

Layers in a neural network

ML-based Classifier

Neural Networks Introduced



Corpus of
Images

Each layer consists of individual interconnected
neurons

ML-based Classifier

Neurons as Learning Units

A machine learning algorithm is an algorithm that
is able to learn from data

Learning Algorithms

A computer program is said to learn from **experience E** with respect to some class of **tasks T** and **performance measure P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**

Learning Algorithms

A computer program is said to learn from experience E with respect to some class of **tasks** T and performance measure \mathcal{P} , if its performance at tasks in T , as measured by \mathcal{P} , improves with experience E

Most common tasks in ML:

Classification, regression

Learning Algorithms

A computer program is said to learn from experience E with respect to some class of tasks T and **performance measure** β , if its performance at tasks in T , as measured by β , improves with experience E

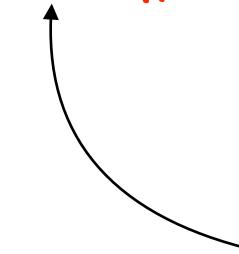


Accuracy in classification,
residual variance in regression

Learning Algorithms

A computer program is said to learn from **experience E** with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E

Training using a corpus of
labelled instances



Learning Algorithms

A computer program is said to **learn from experience** E with respect to some class of tasks T and performance measure P , if its **performance at tasks** in T , as measured by P , improves with experience E

Deep Learning and Neural Networks

Deep Learning

Algorithms that learn what
features matter

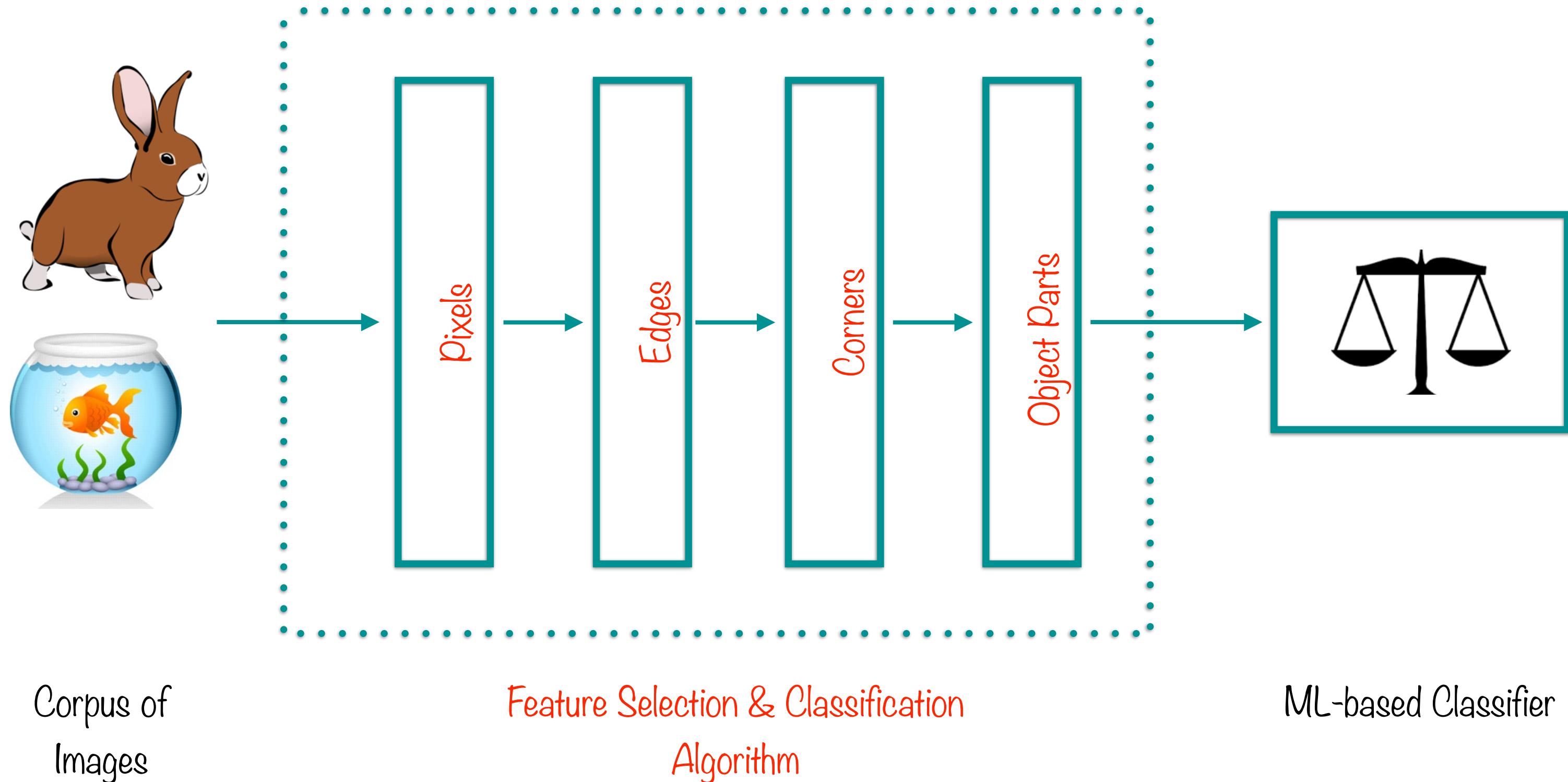
Neural Networks

The most common class of deep
learning algorithms

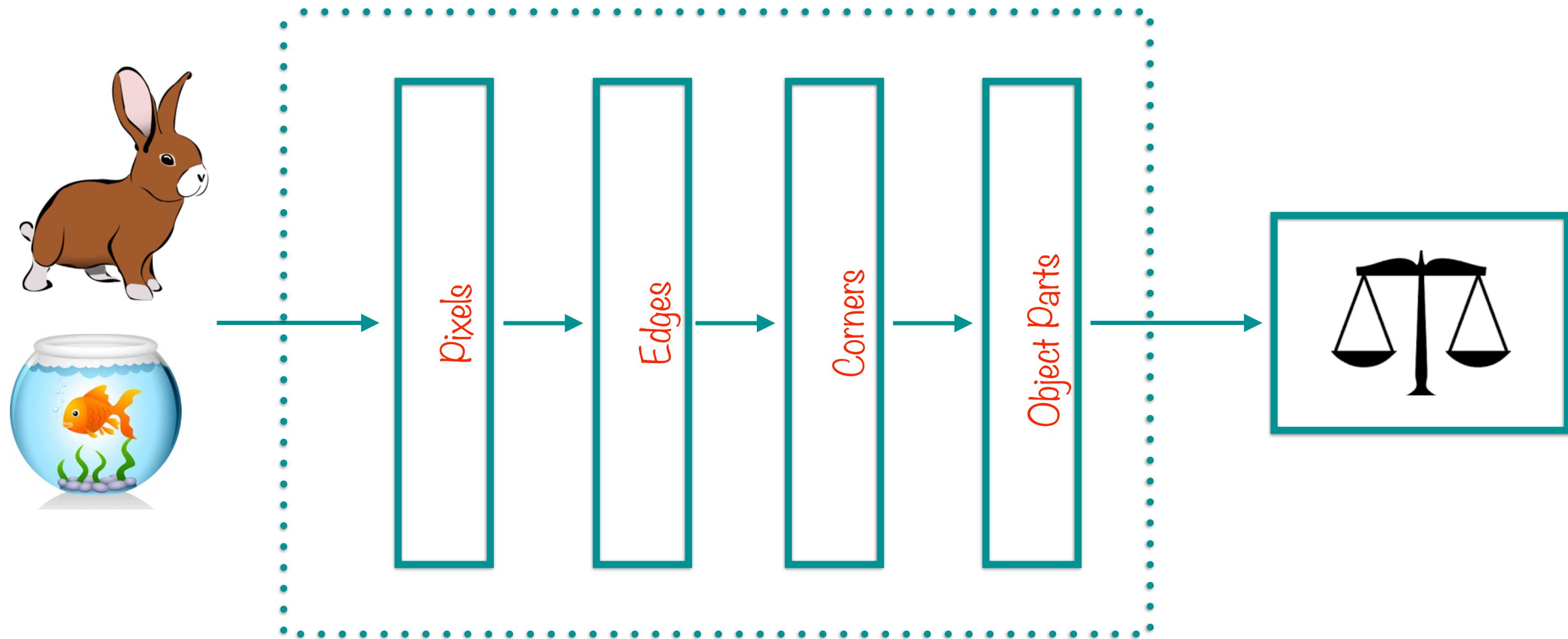
Neurons

Simple building blocks that actually
“learn”

“Deep Learning”-based Binary Classifier



Layers in the Computation Graph

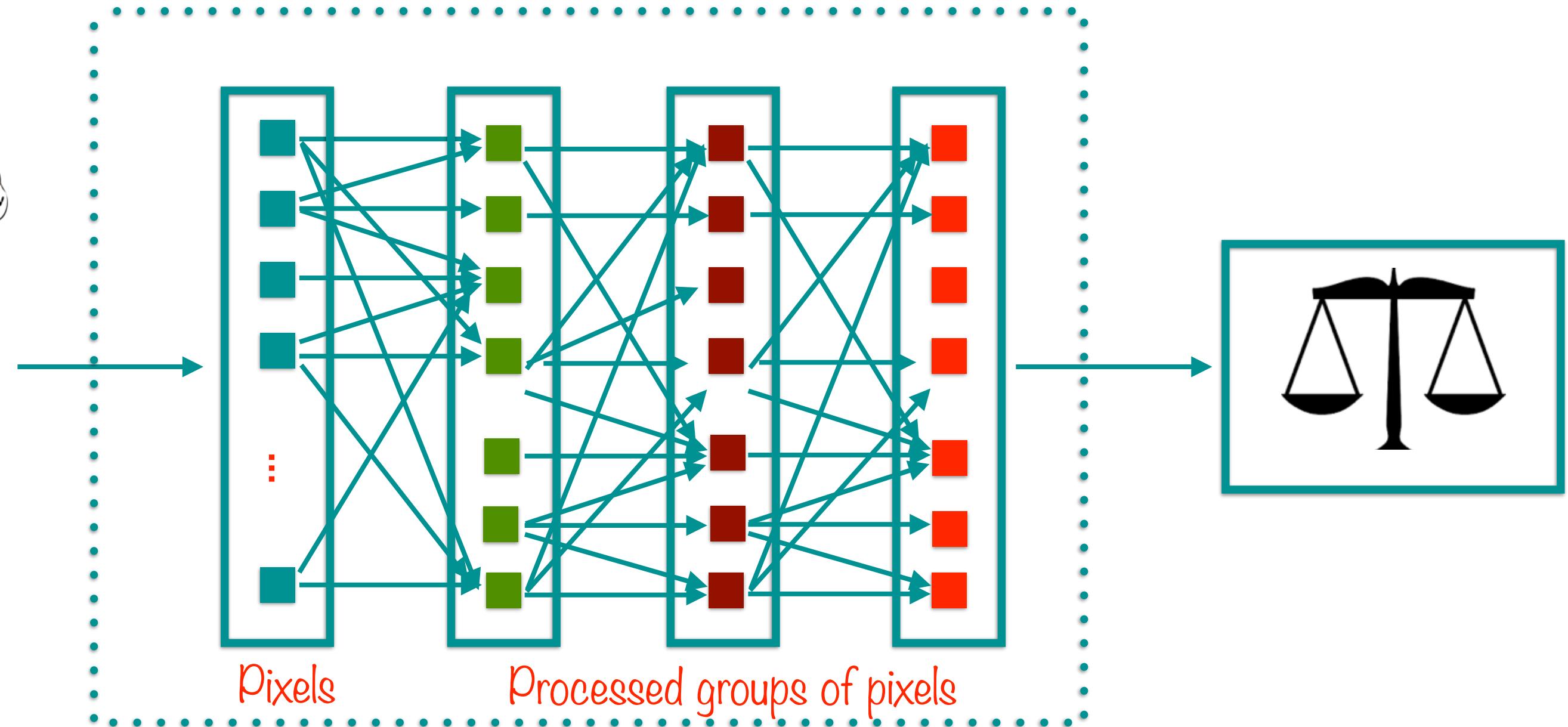


Corpus of
Images

Groups of neurons that perform similar
functions are aggregated into layers

ML-based Classifier

Neural Networks: Networks of Neurons



Corpus of
Images

Each layer consists of individual interconnected
neurons

ML-based Classifier

Deep Learning

Directed computation graphs “learn” relationships between data

The more complex the graph, the more the relationships it can “learn”

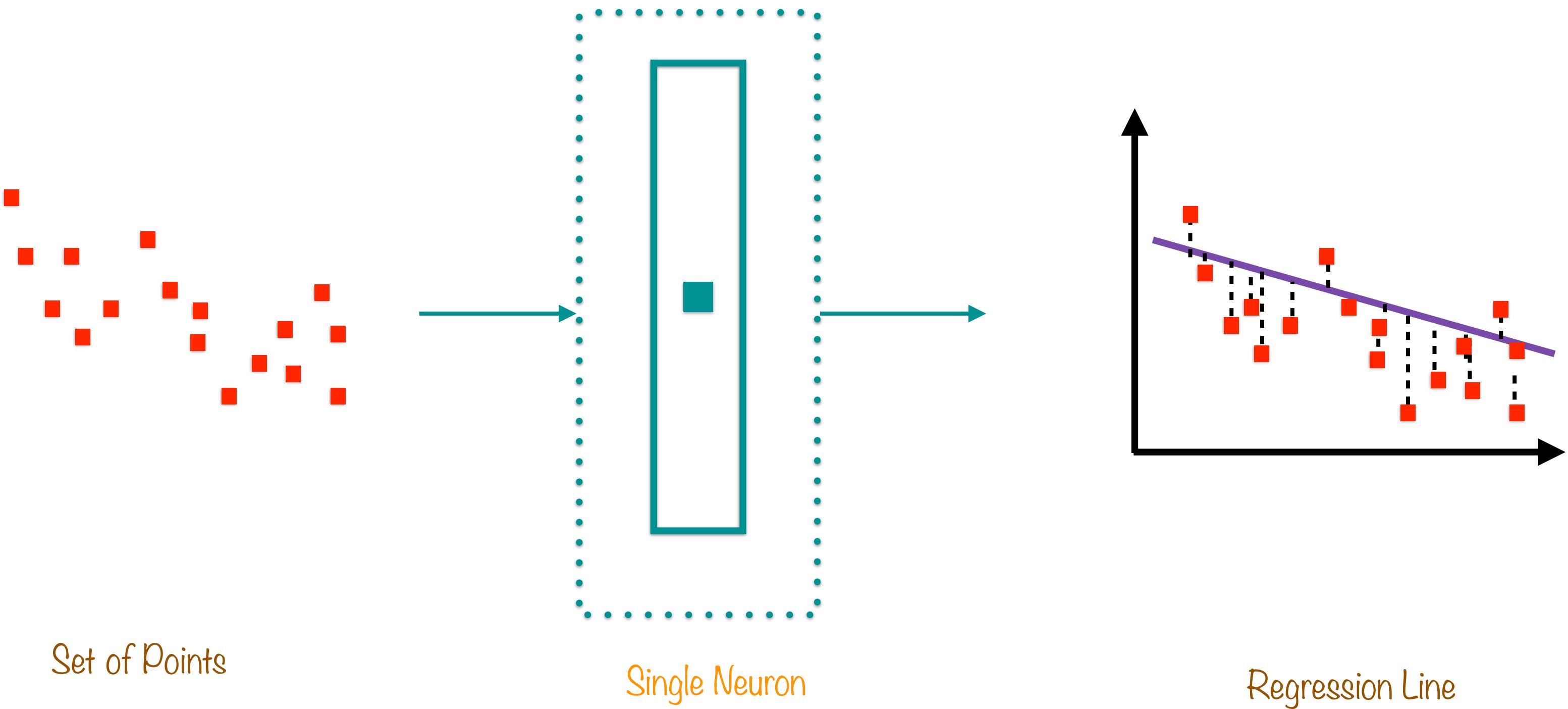
“Deep” Learning: Depth of the computation graph

$$y = Wx + b$$

“Learning” Regression

Regression can be reverse-engineered by a single neuron

Regression: The Simplest Neural Network

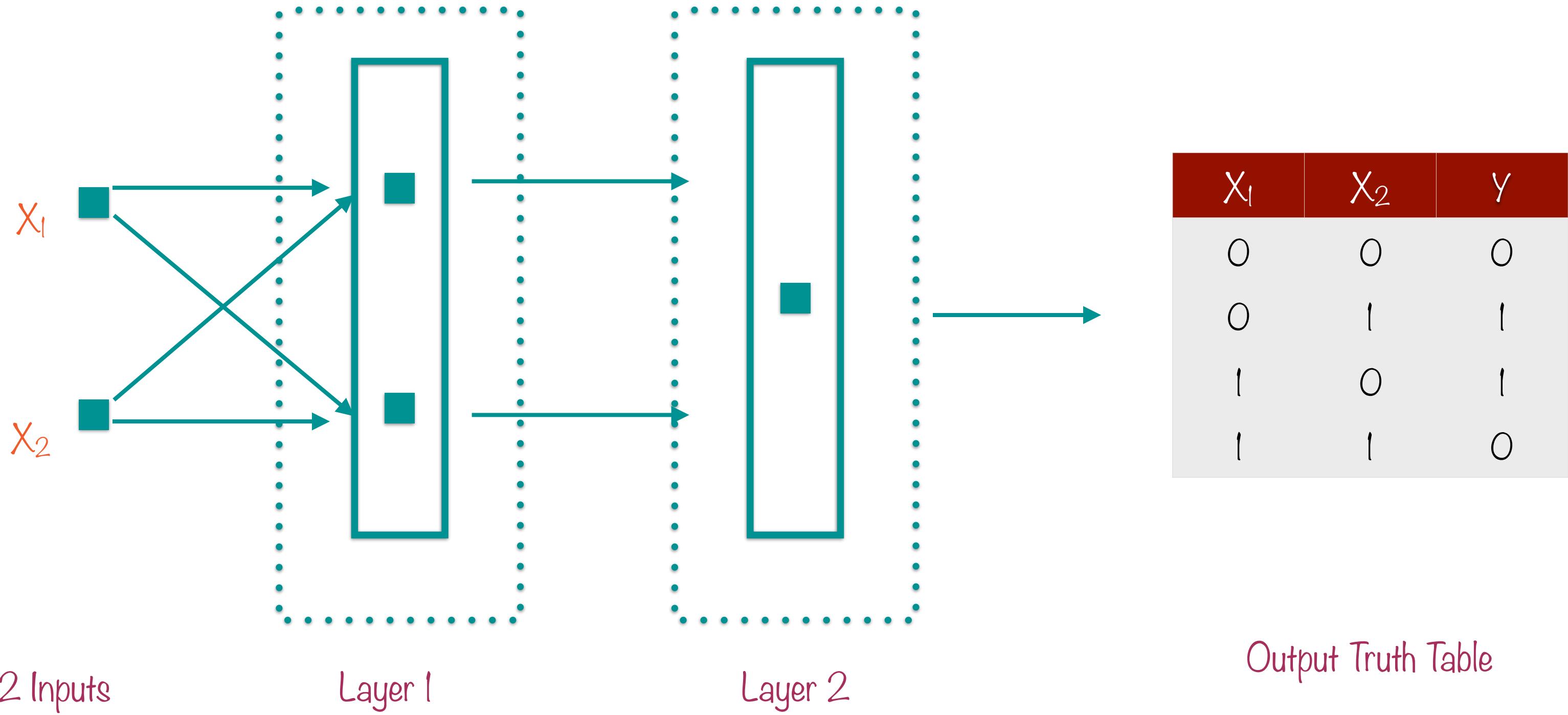


```
def XOR(x1,x2):  
    if (x1 == x2):  
        return 0  
    return 1
```

“Learning” XOR

The XOR function can be reverse-engineered using 3 neurons arranged in 2 layers

XOR: 3 Neurons, 2 Layers

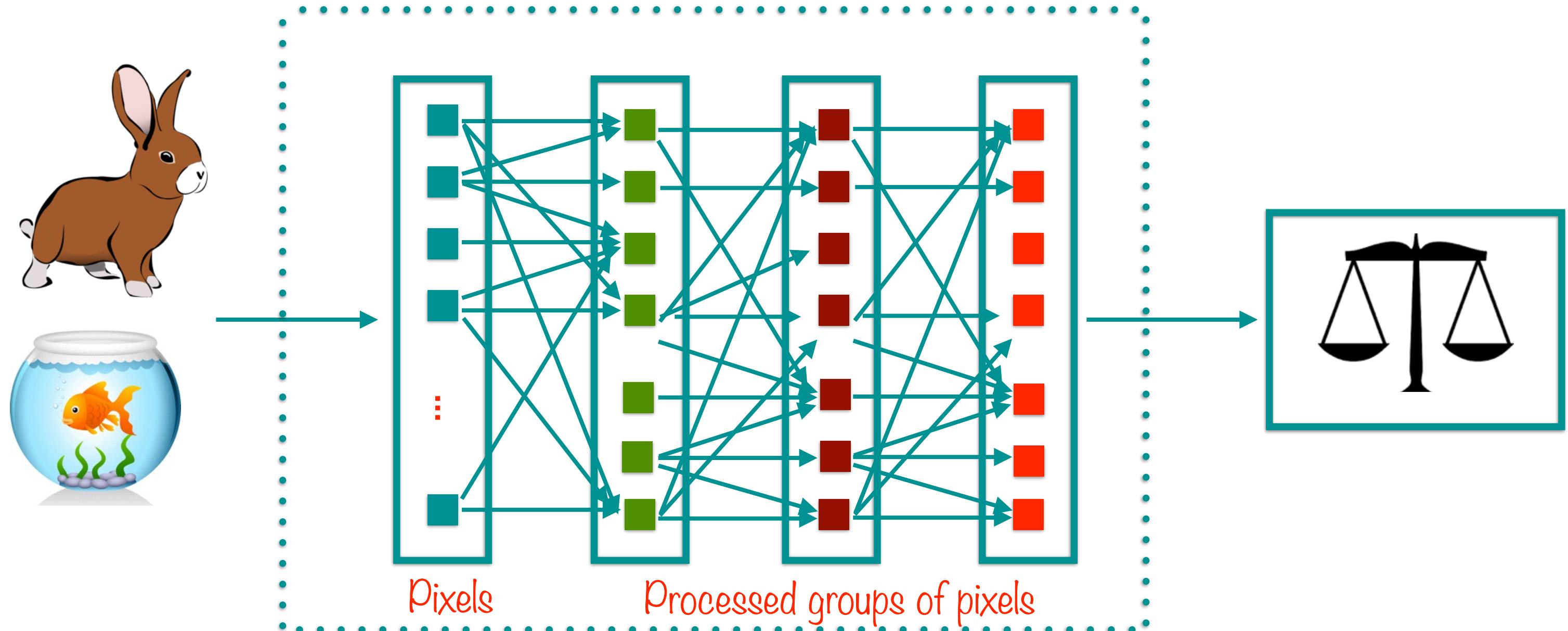


```
def doSomethingReallyComplicated(x1,x2...):  
    ...  
    ...  
    ...  
    return complicatedResult
```

“Learning” Arbitrarily Complex Functions

Adding layers to a neural network can “learn” (reverse-engineer) pretty much anything

Arbitrarily Complex Function



Corpus of Images

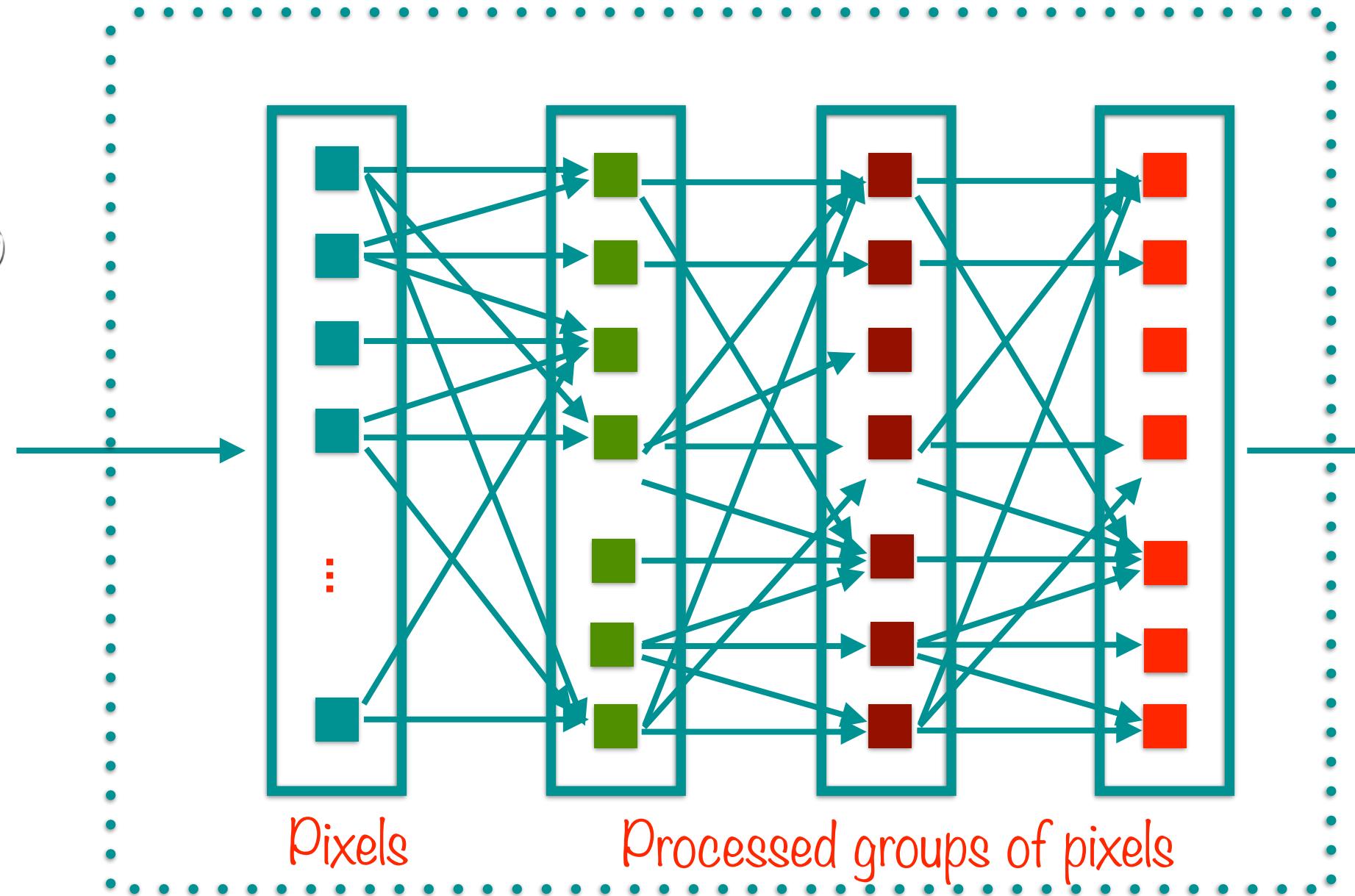
Operations (nodes) on data (edges)

ML-based Classifier

The Computational Graph



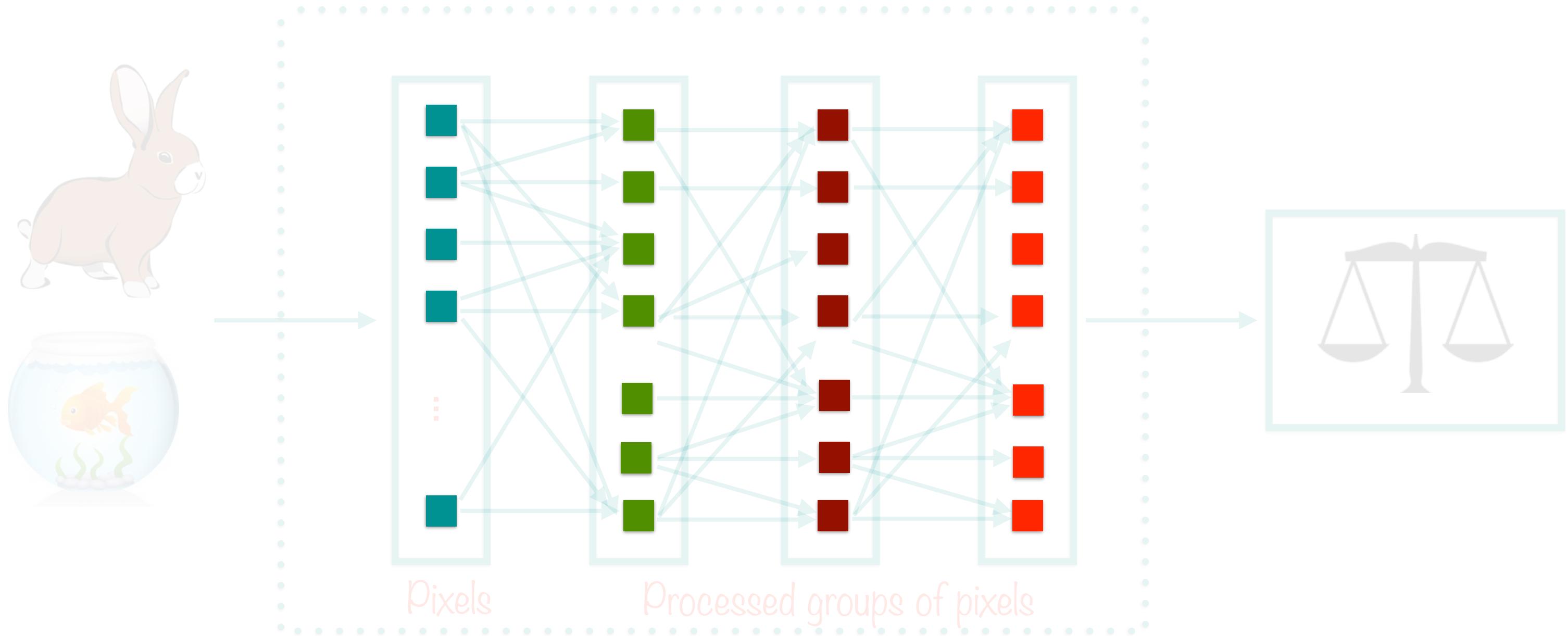
Corpus of
Images



Operations (nodes) on data (edges)

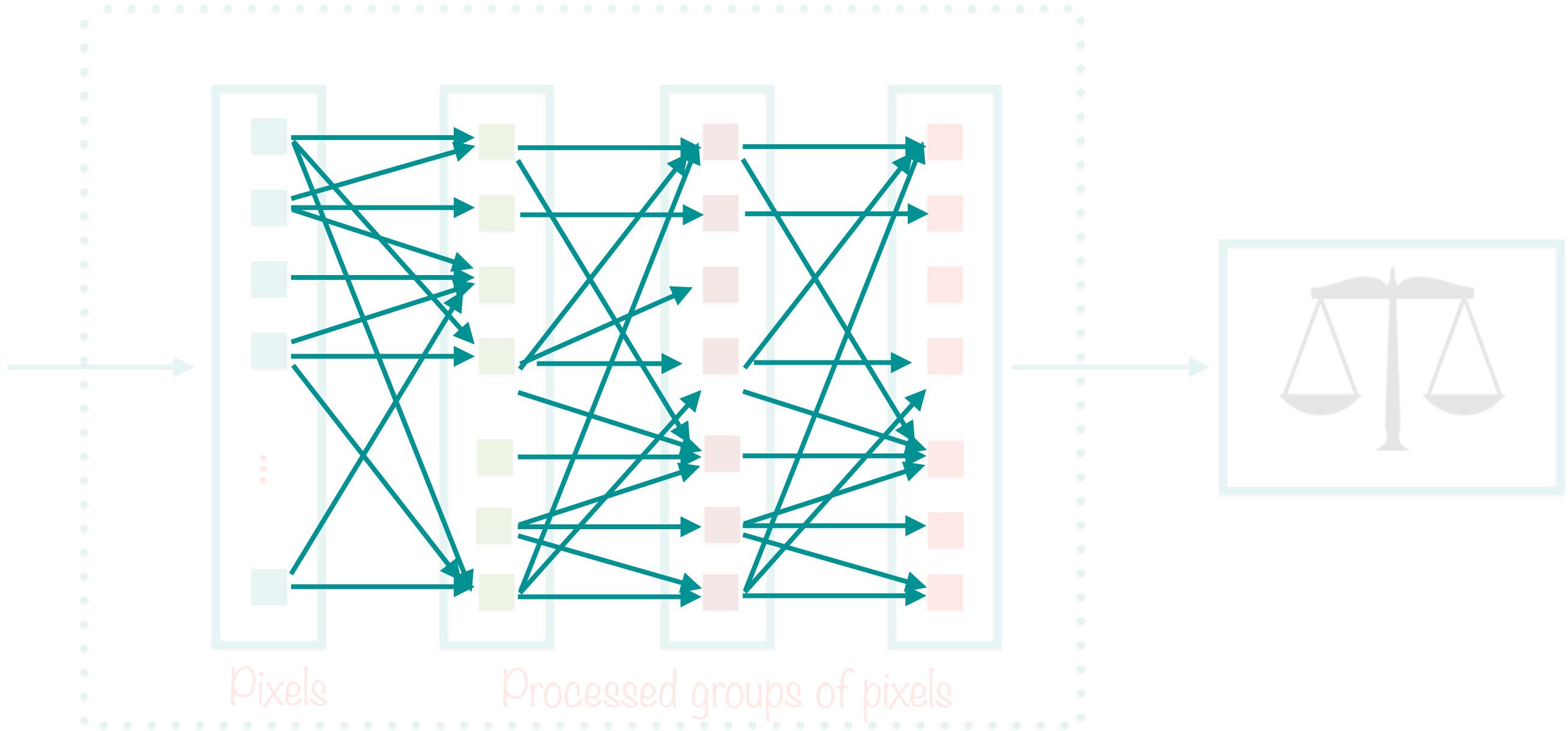
ML-based Classifier

The Computational Graph



The nodes in the computation graph are neurons
(simple building blocks)

The Computational Graph



Corpus of
Images

The edges in the computation graph are data items
called tensors

ML-based Classifier

Neurons

The nodes in the computation graph are simple entities called neurons

Each neuron performs very simple operations on data

The neurons are connected in very complex, sophisticated ways

Neural Networks

The complex interconnections between simple neurons

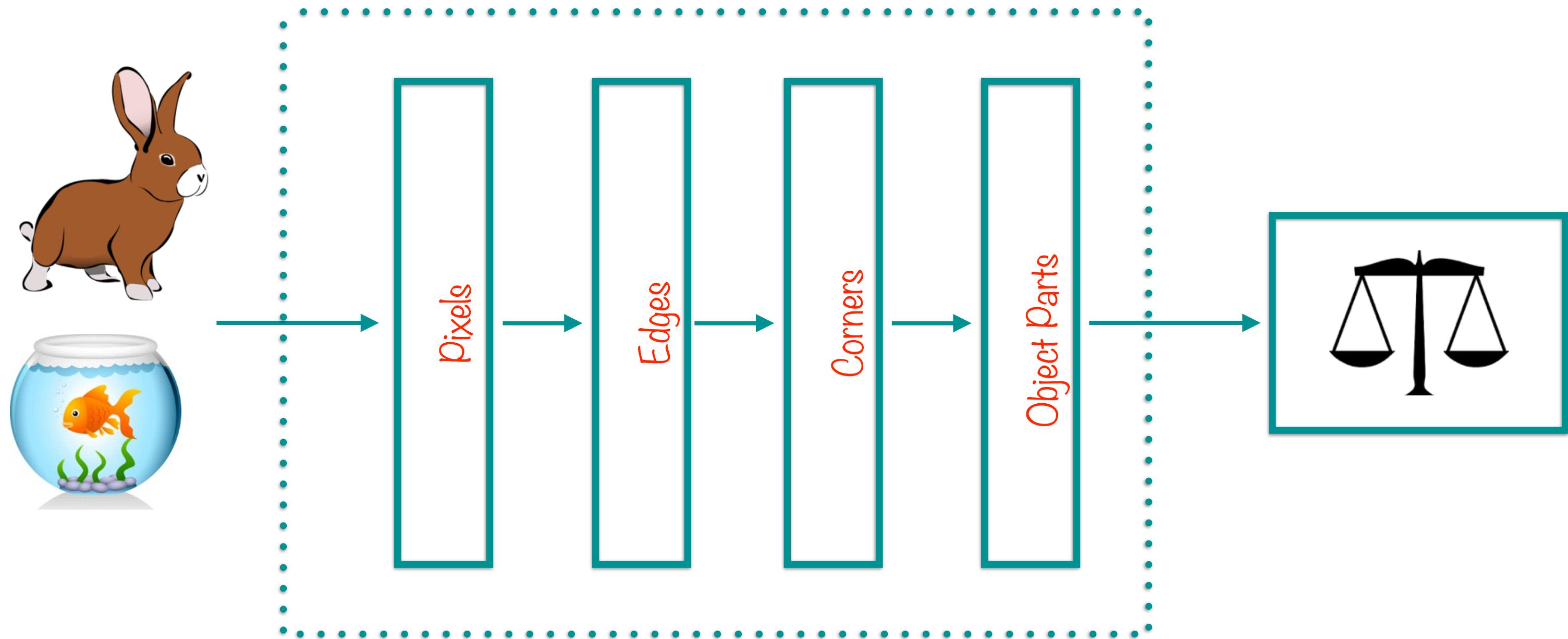
Different network configurations => different types of neural networks

- Convolutional
- Recurrent

Neural Networks

Groups of neurons that perform similar functions are aggregated into layers

Layers in the Computation Graph

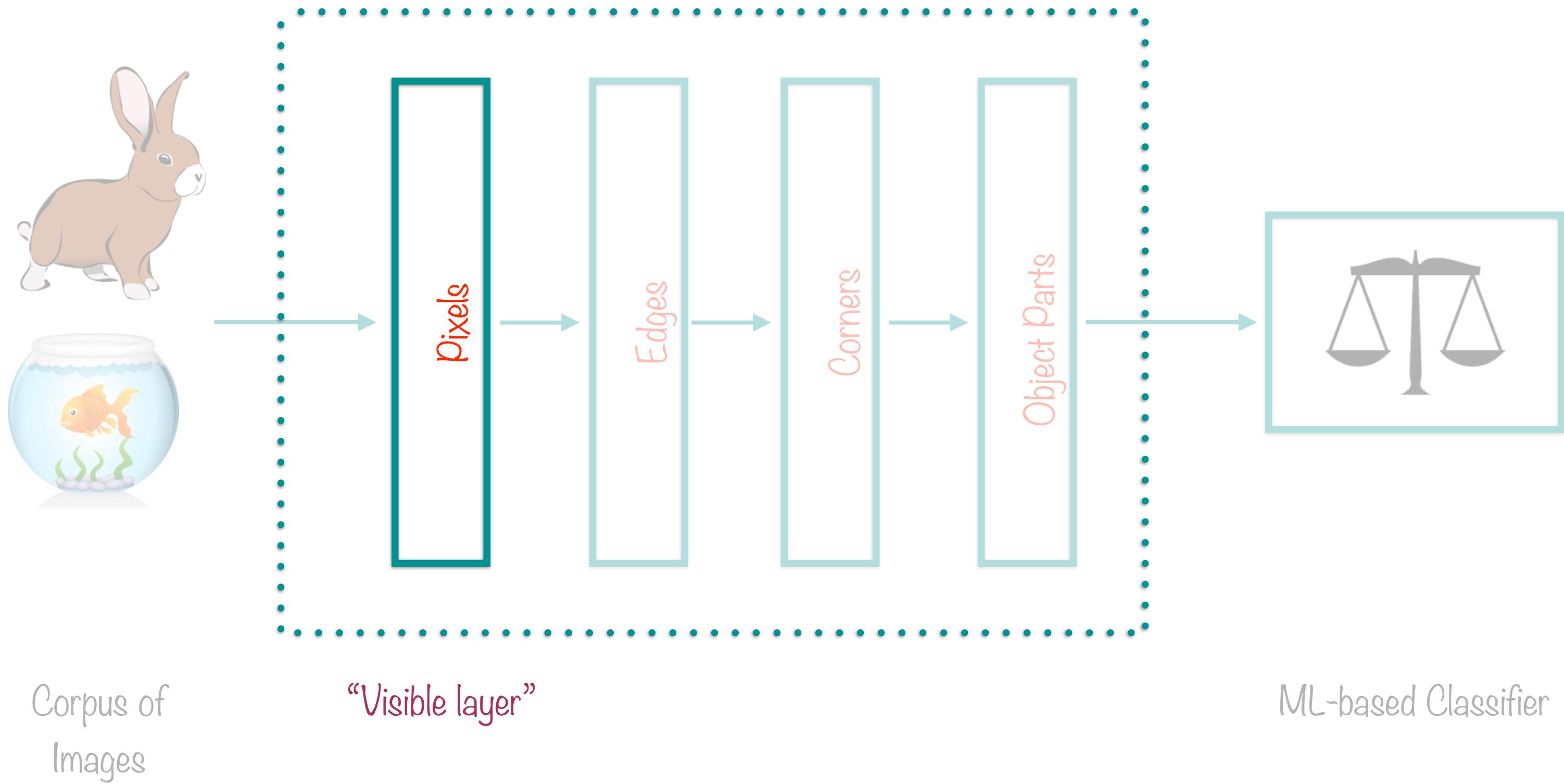


Corpus of
Images

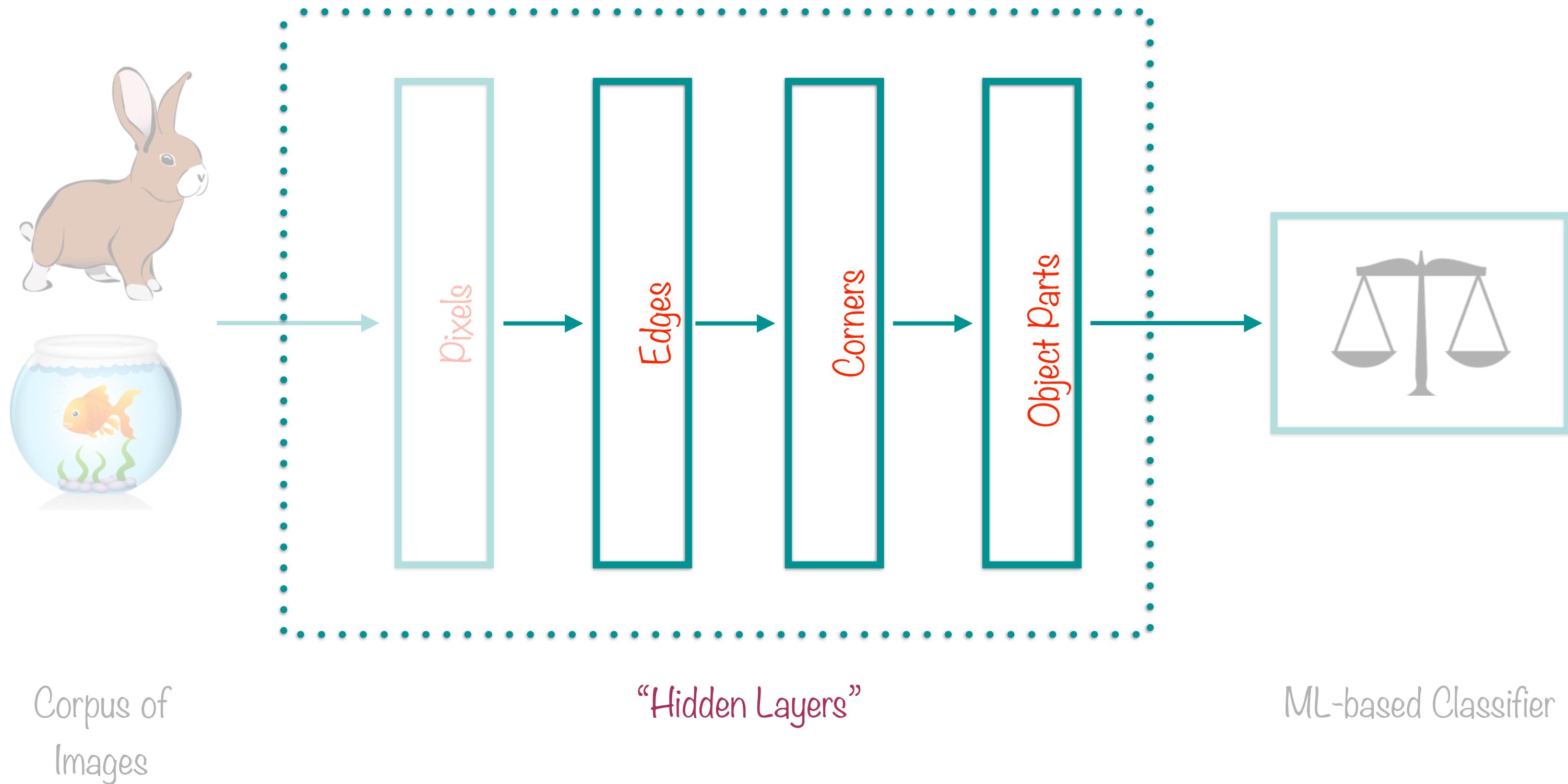
Groups of neurons that perform similar
functions are aggregated into layers

ML-based Classifier

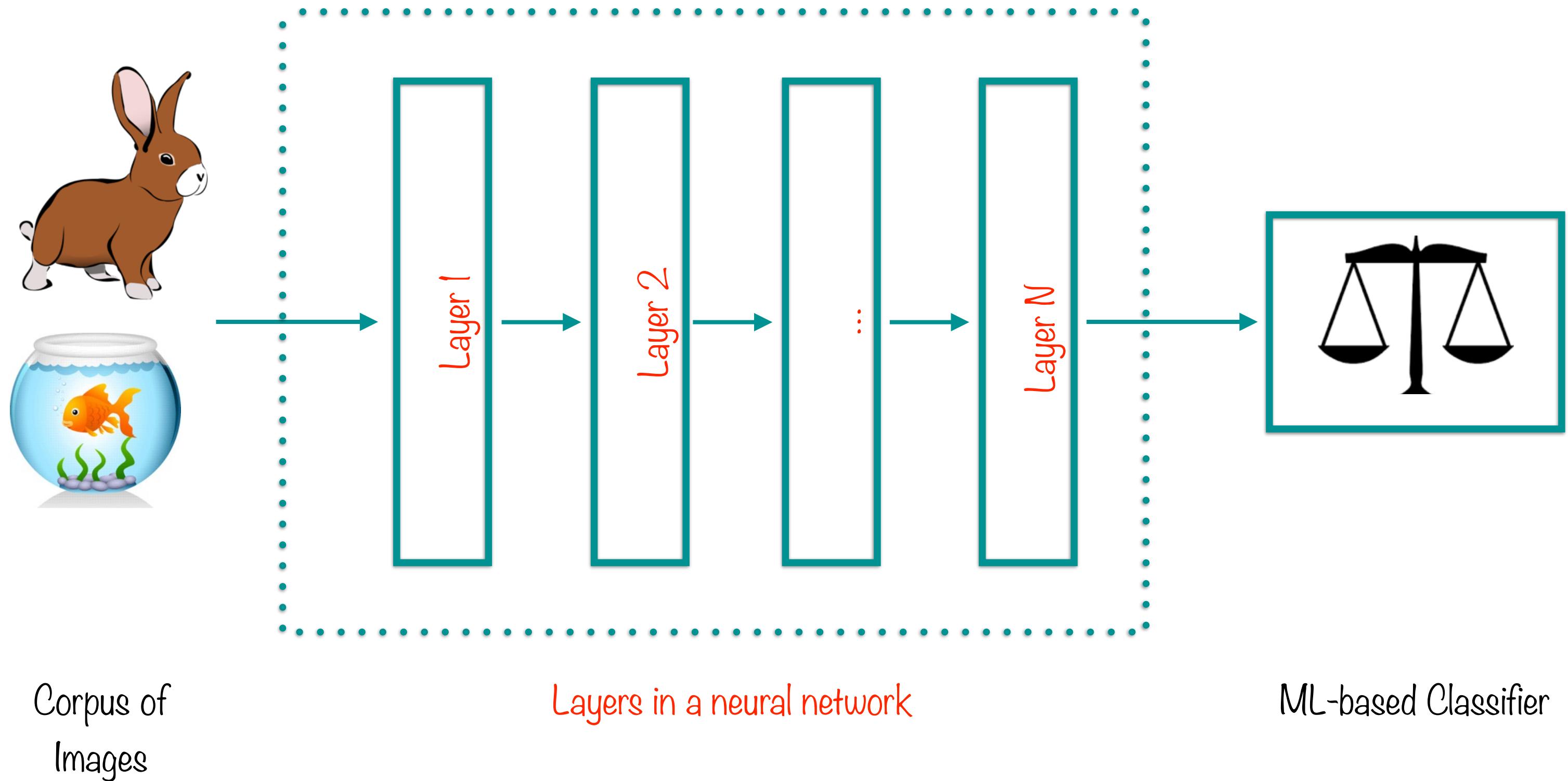
Layers in the Computation Graph



Layers in the Computation Graph



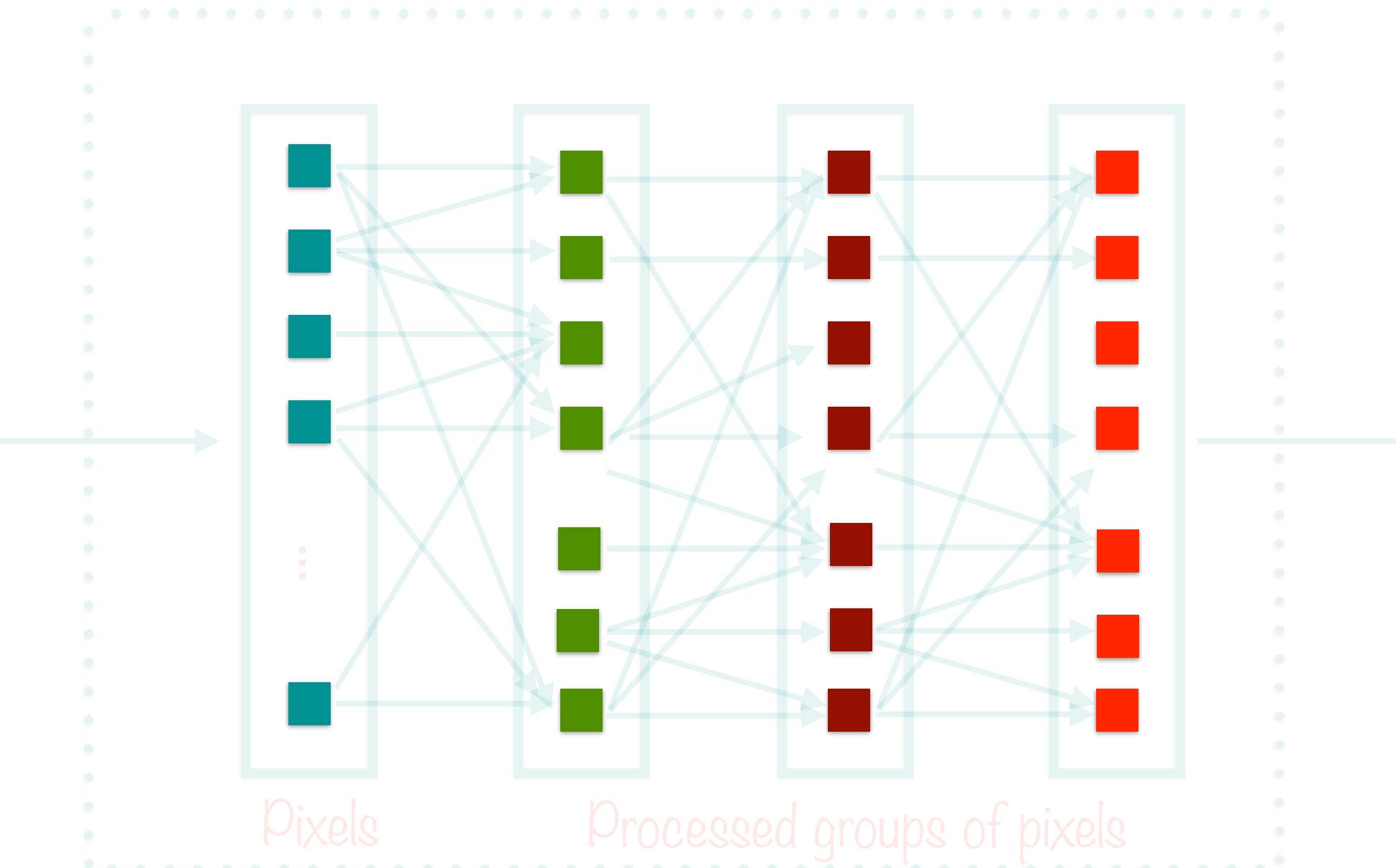
Layers in the Computation Graph



Neurons

Each layer consists of units called neurons

Neurons



Corpus of
Images

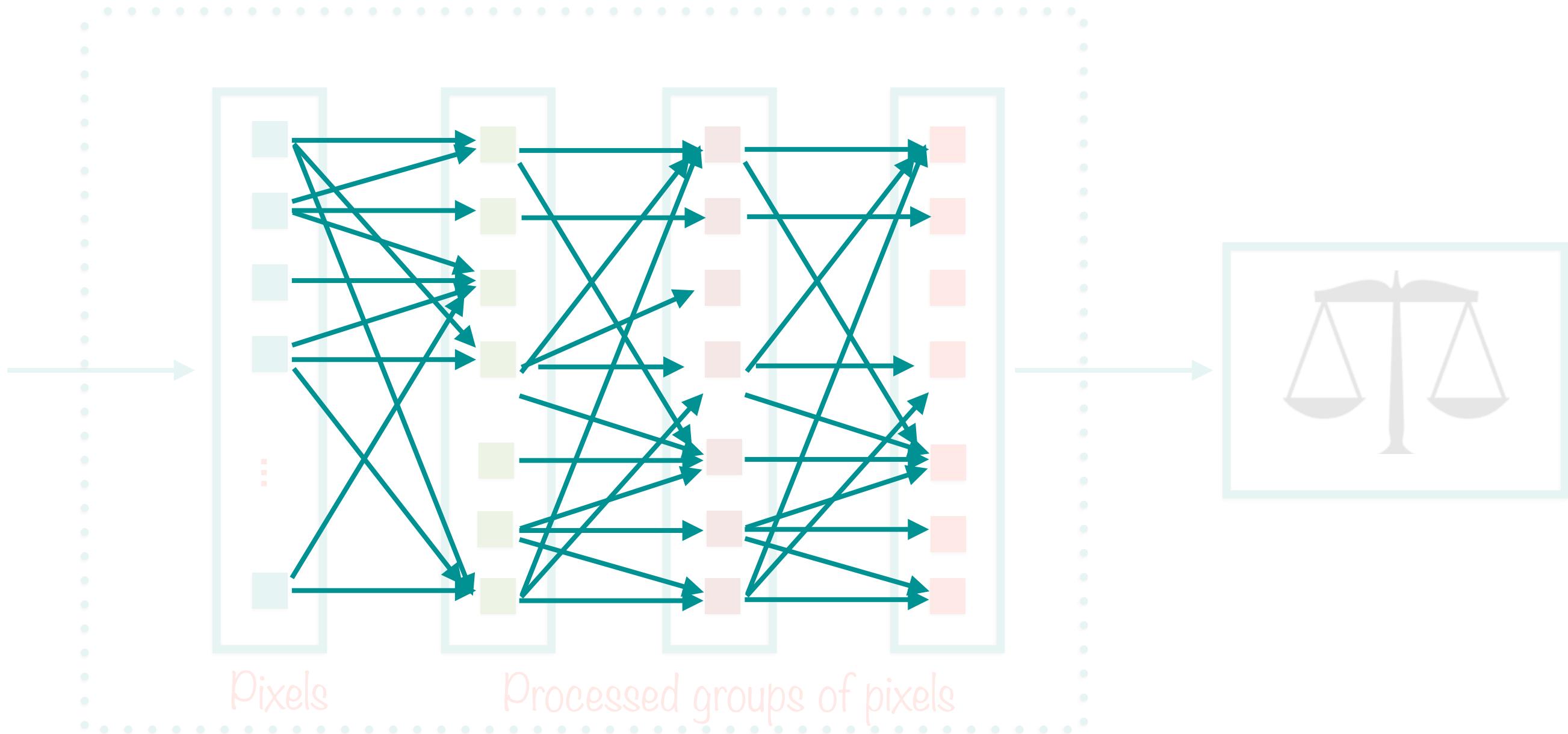
Neural Network

ML-based Classifier

Neural Networks

Neurons in a neural network can be connected in very complex ways...

Neural Networks Introduced



Corpus of
Images

Neurons in a neural network can be connected in very
complex ways...

ML-based Classifier

Neural Networks

Neurons in a neural network can be connected in very complex ways...

...But each neuron only applies two simple functions to its inputs

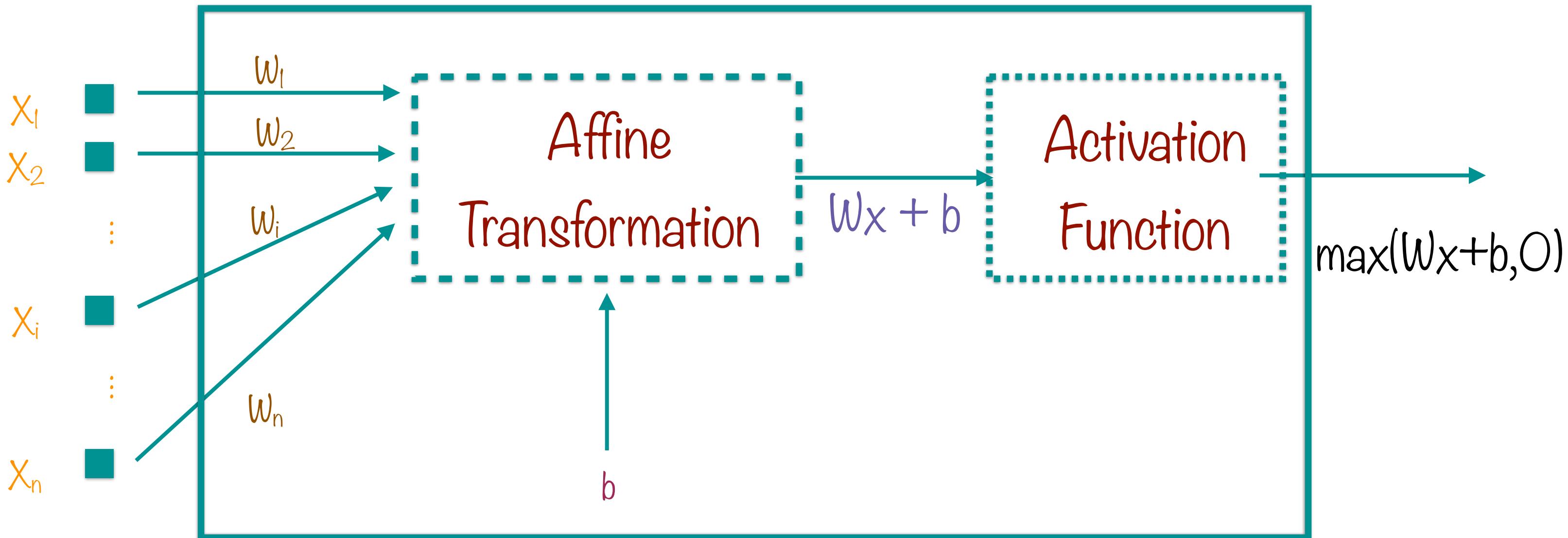
Neural Networks

Neurons in a neural network can be connected in very complex ways...

...But each neuron only applies two simple functions to its inputs

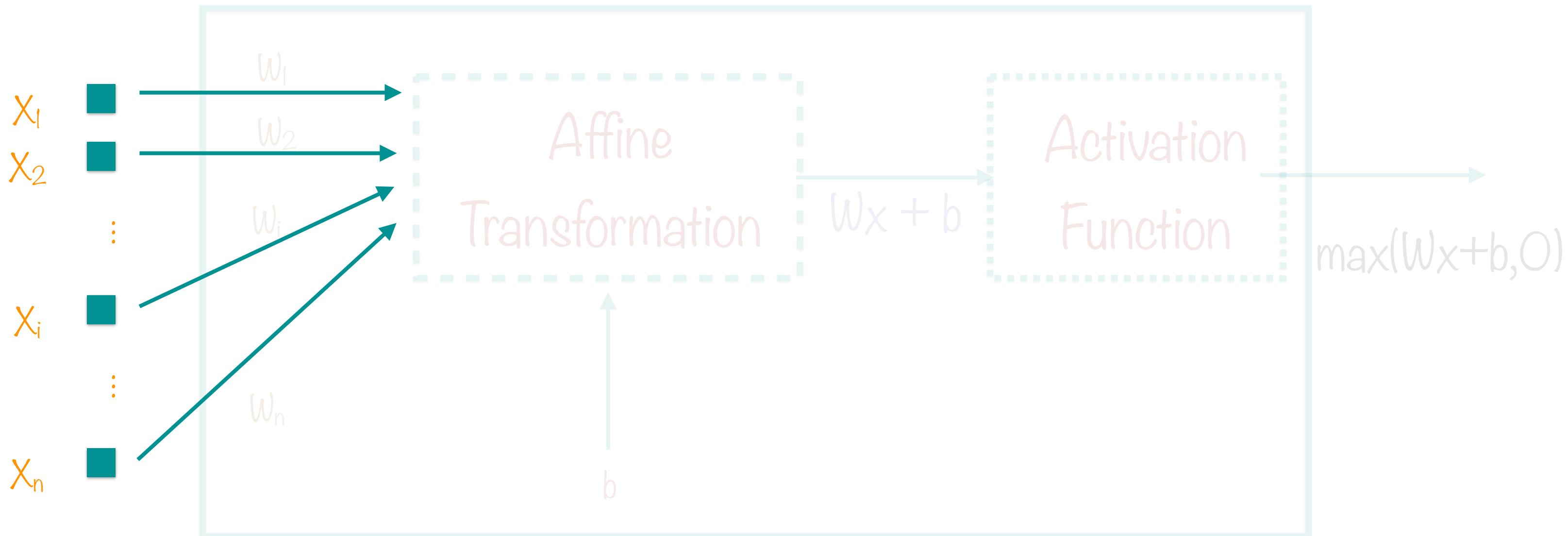
- A linear (affine) transformation
- An activation function

Operation of a Single Neuron

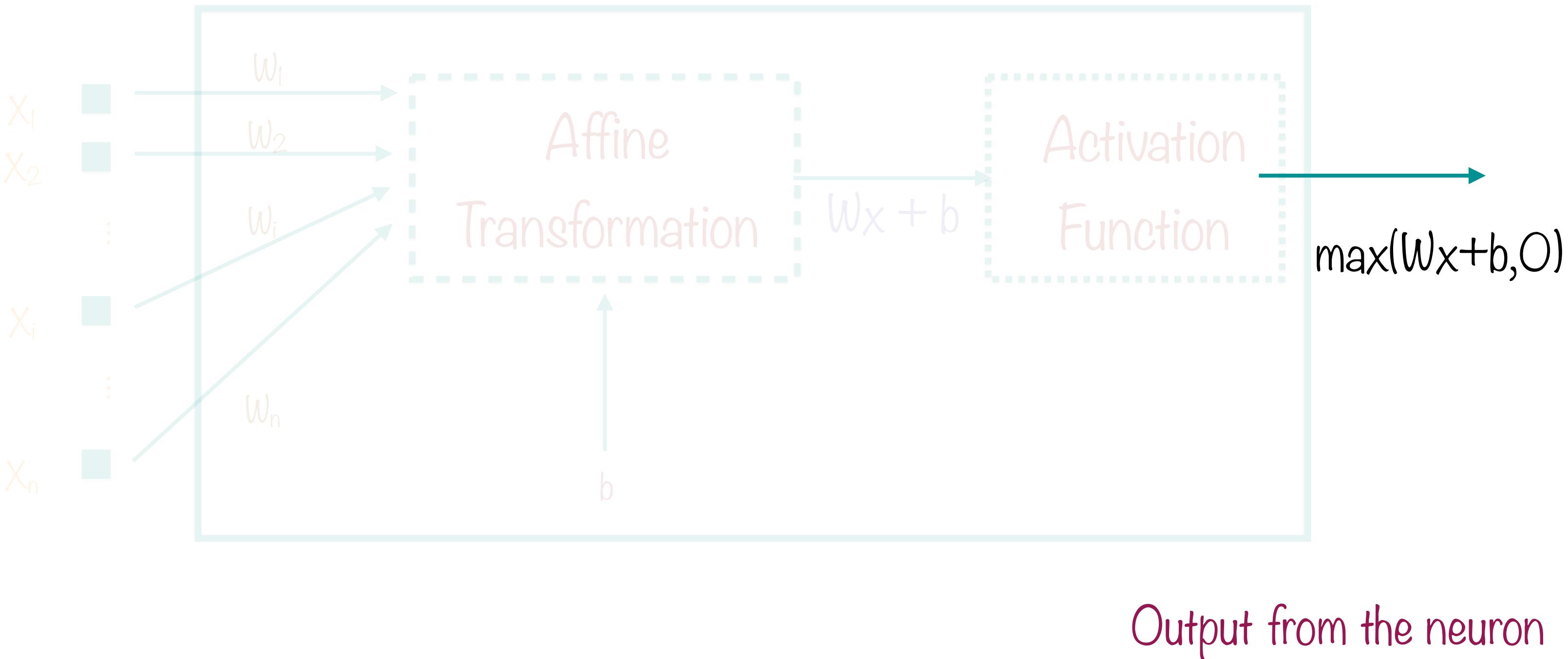


Each neuron only applies two simple functions to its inputs

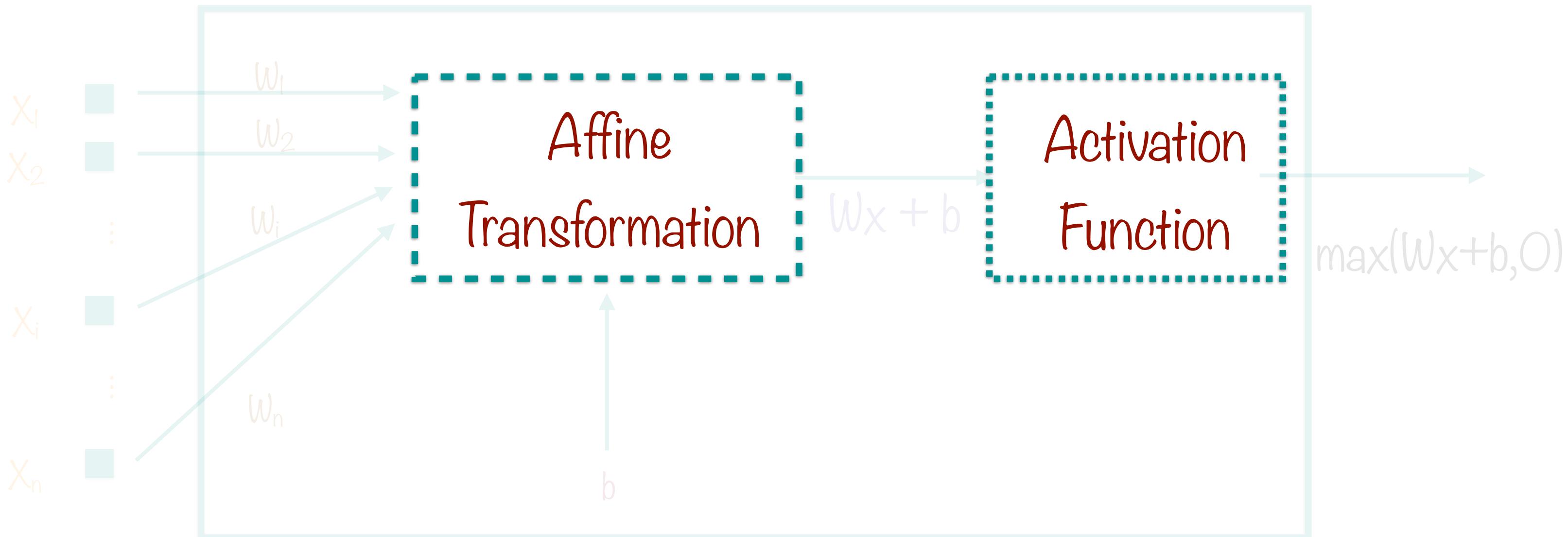
Operation of a Single Neuron



Operation of a Single Neuron

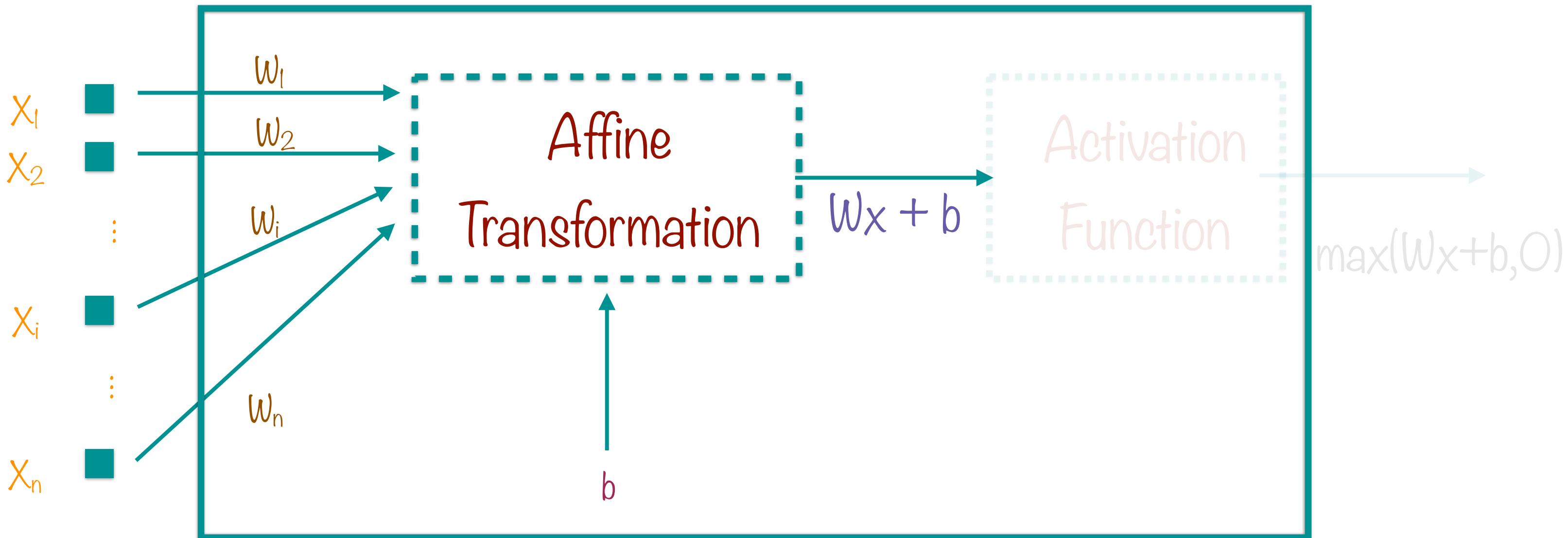


Operation of a Single Neuron



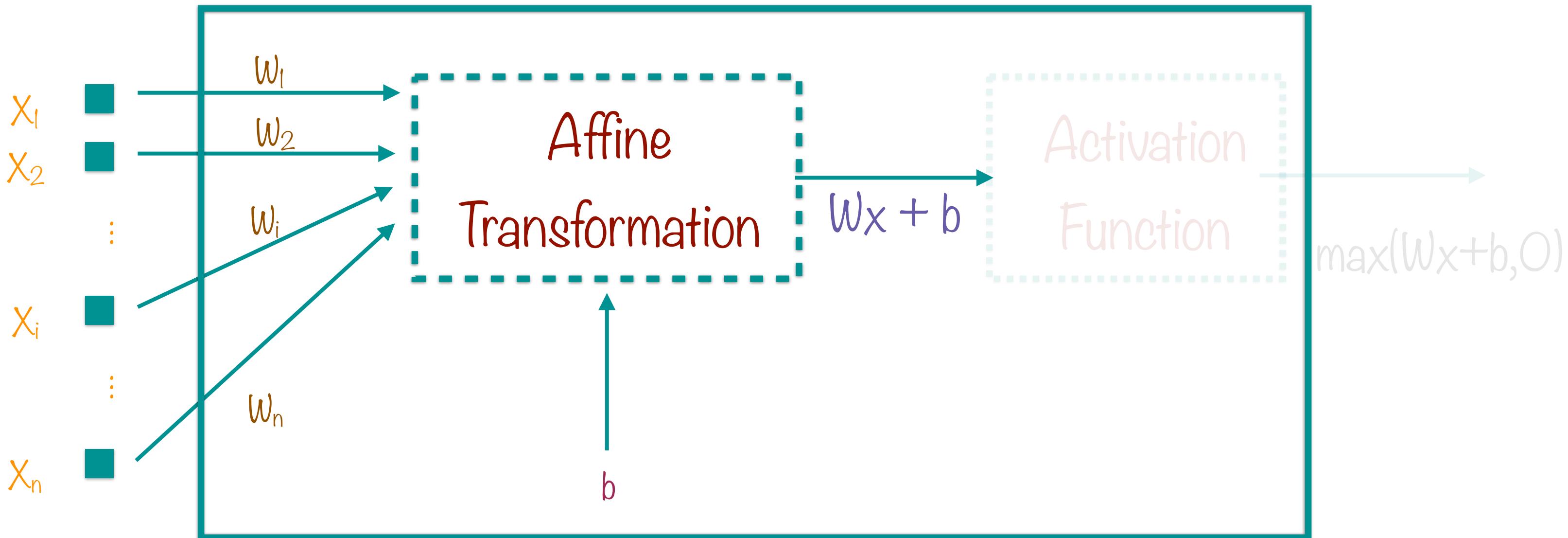
Each neuron only applies two simple functions to its inputs

Operation of a Single Neuron



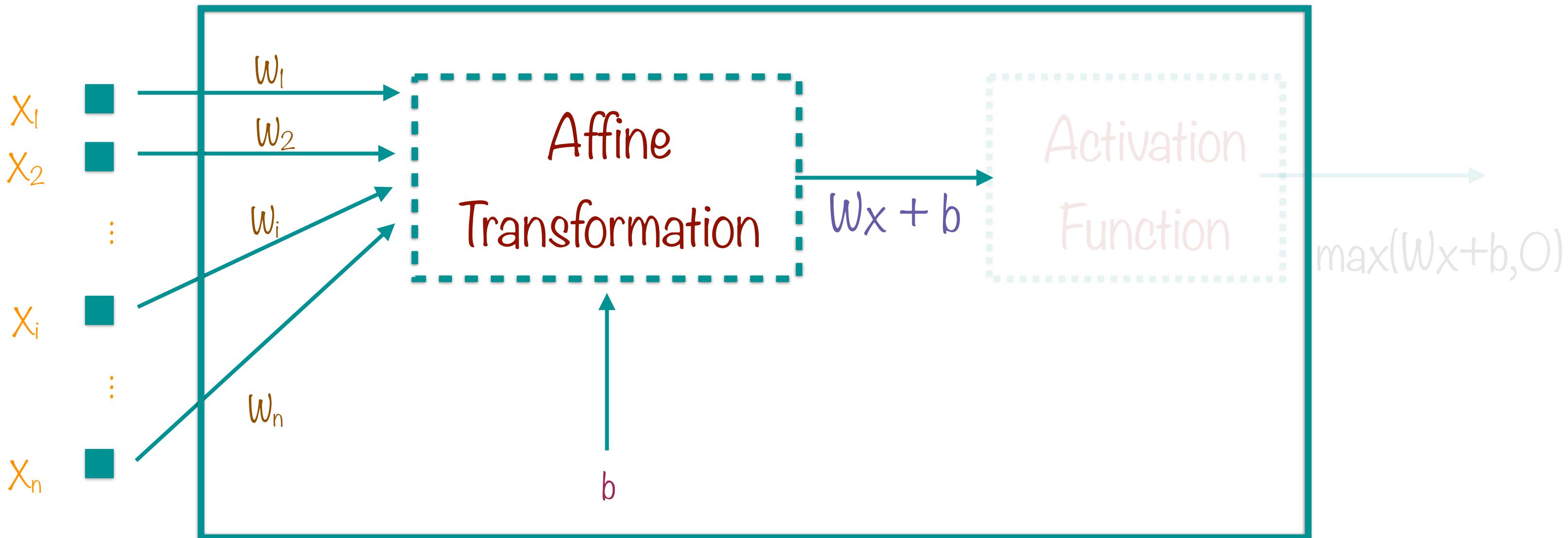
The affine transformation is just a weighted sum with a bias added

Operation of a Single Neuron



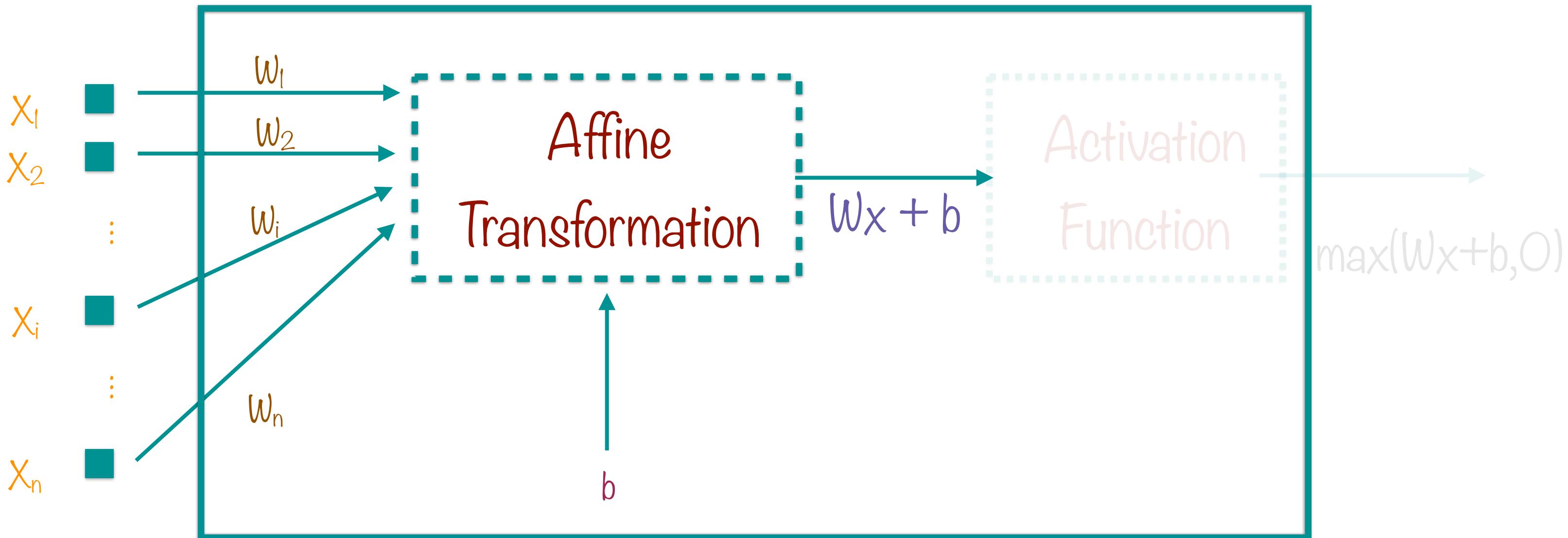
The values $w_1, w_2 \dots w_n$ are called the weights

Operation of a Single Neuron



The value b is called the bias

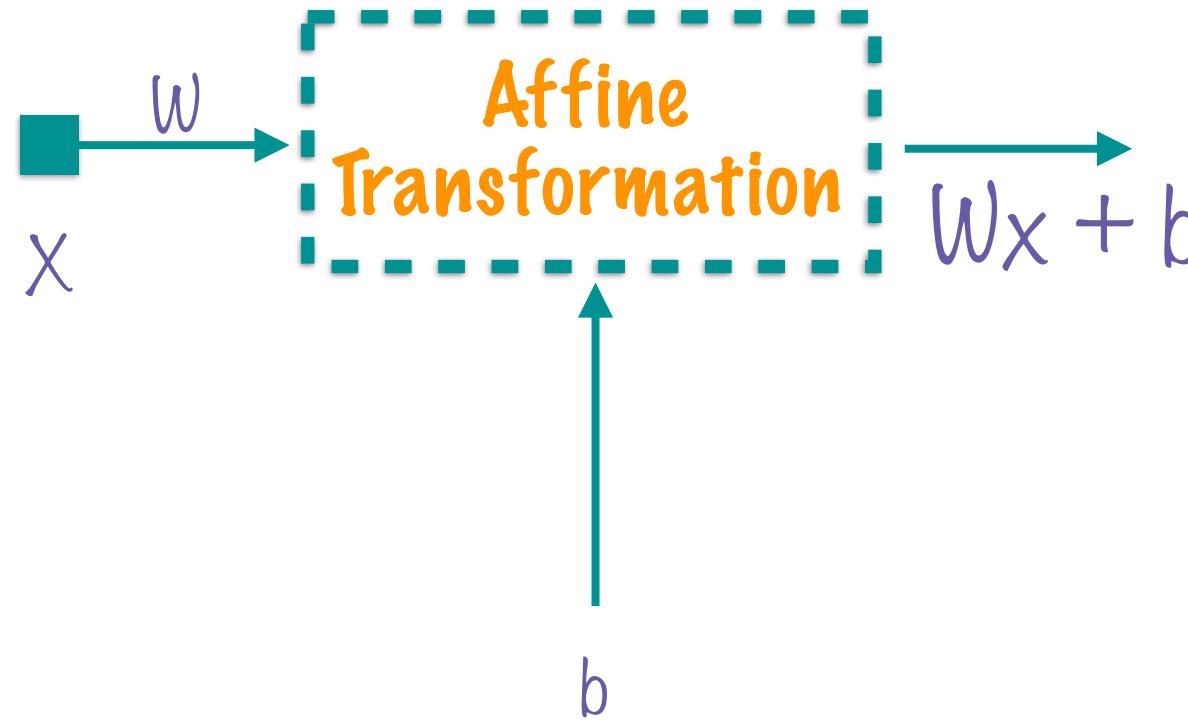
Operation of a Single Neuron



Where do the values of W and b come from?

The weights and biases of individual neurons are determined during the training process

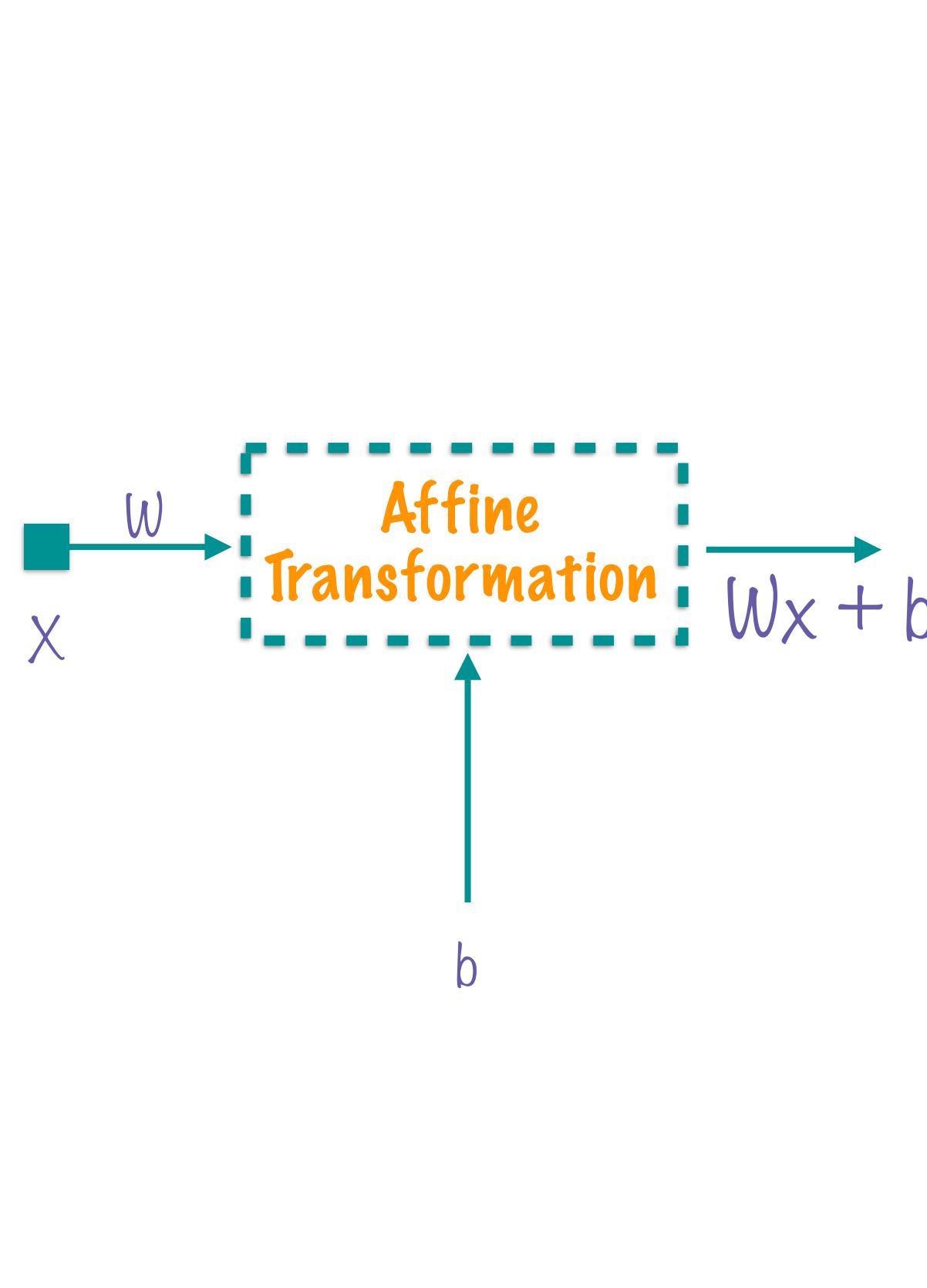
The actual training of a neural network is
managed by TensorFlow



Finding the “best” values of W and b for each neuron is crucial

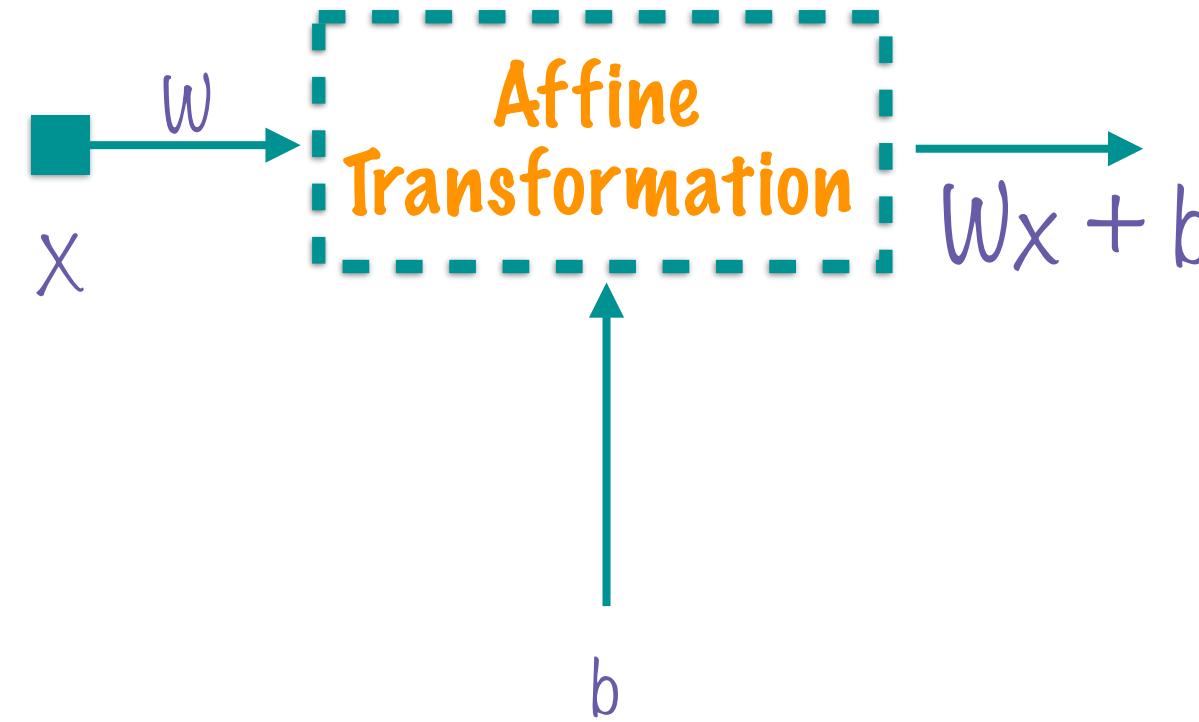
The “best” values are found using the cost function, optimiser and corpus...

...and the process of finding them is called the training process



Different types of neural networks wire up neurons in different ways

These interconnections can get very sophisticated...

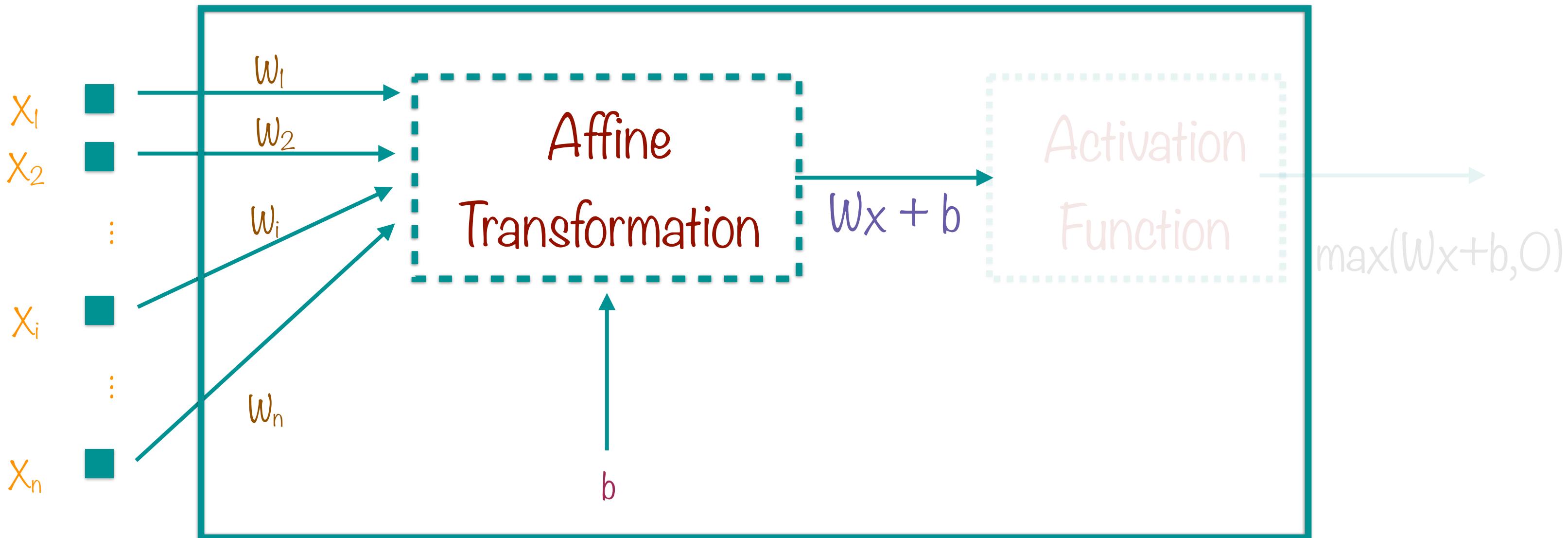


During training, the output of deeper layers may be “fed back” to find the best W, b

This is called back propagation

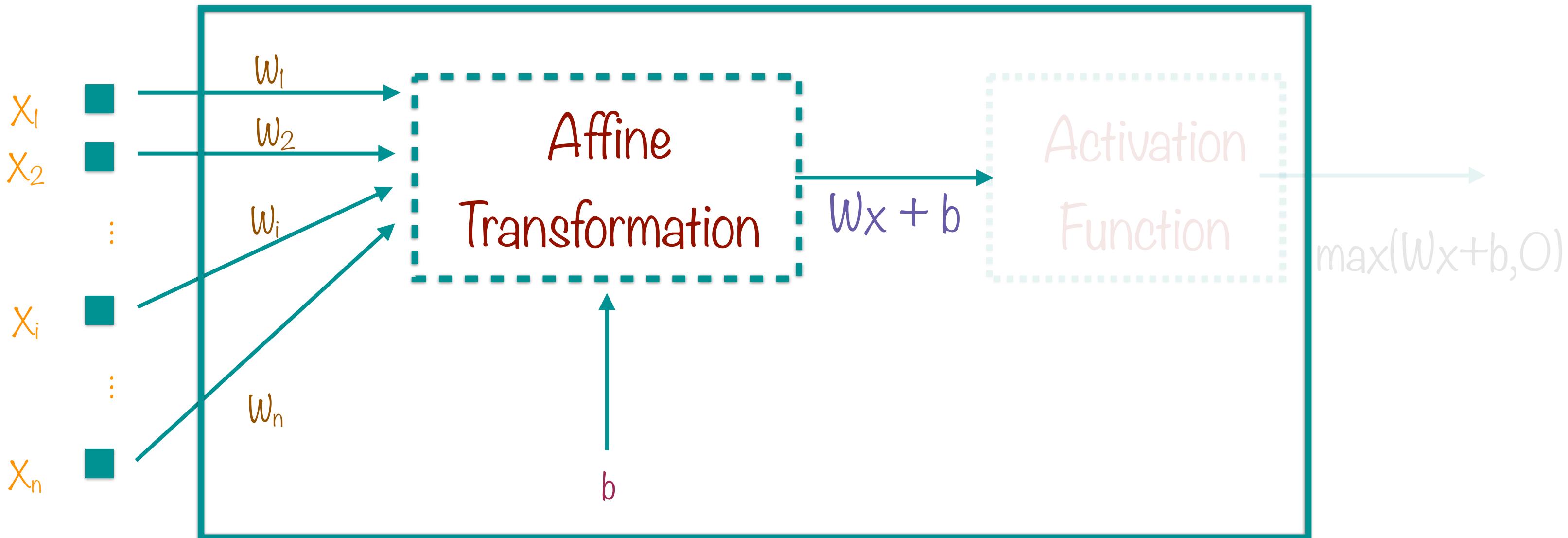
Back propagation is the standard algorithm for training neural networks

Operation of a Single Neuron



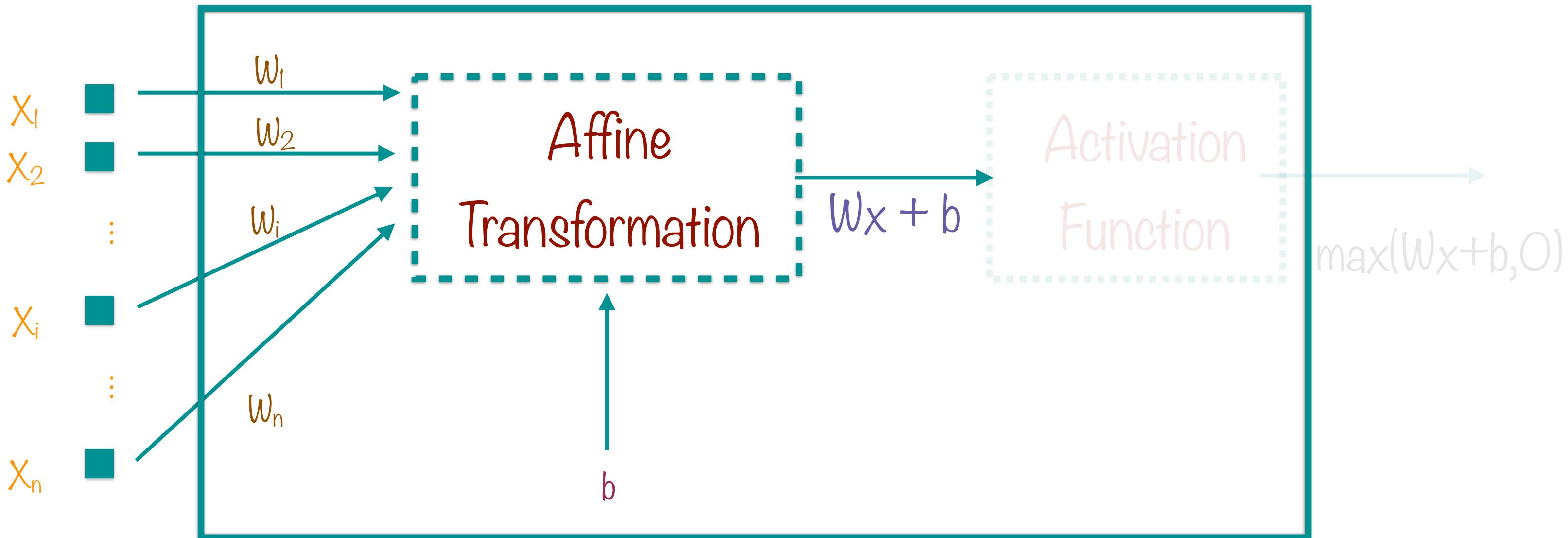
The training algorithm will use the weights to tell the neuron which inputs matter, and which do not...

Operation of a Single Neuron



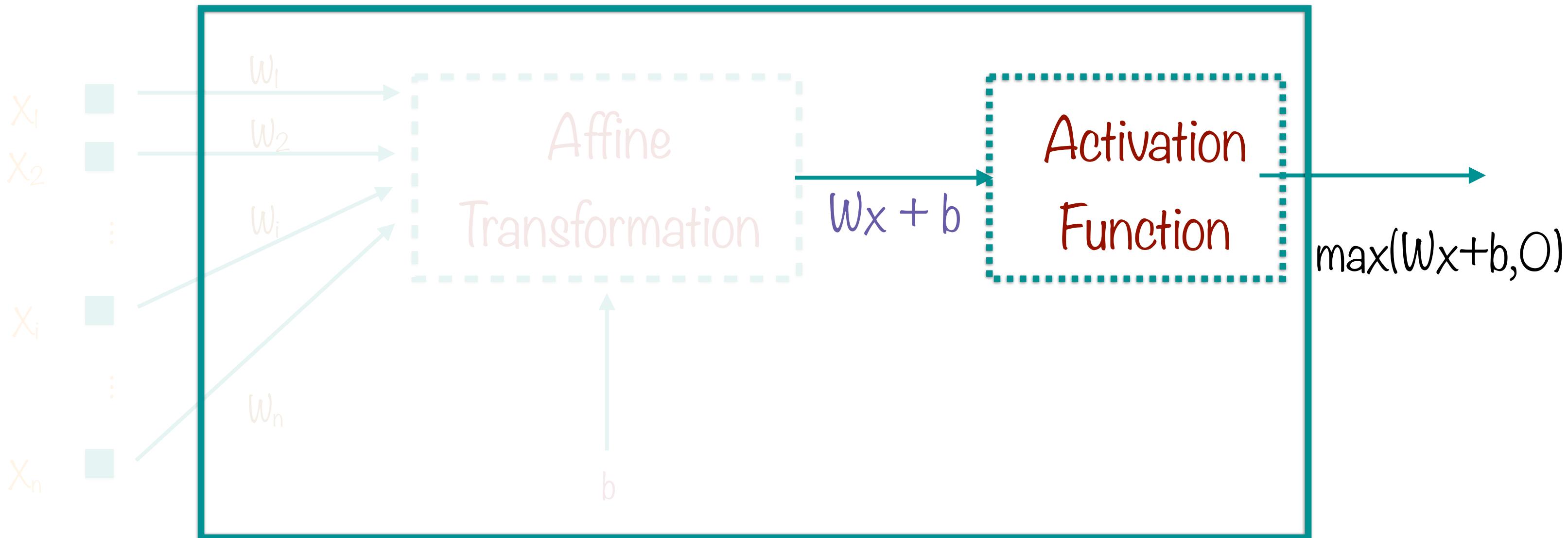
...and apply a corrective bias if needed

Operation of a Single Neuron



The linear output can only be used to learn linear functions, but we can easily generalize this...

Operation of a Single Neuron



The Activation function is a non-linear function, very often simply the $\max(0, \dots)$ function



The output of the affine transformation is chained into an activation function



The activation function is needed for the neural network to predict non-linear functions



The most common form of the activation function is the ReLU

ReLU : Rectified Linear Unit

$$\text{ReLU}(x) = \max(0, x)$$

Regression: The Simplest Neural Network

```
def doSomethingReallyComplicated(x1,x2...):
```

```
...
```

```
...
```

```
...
```

```
    return complicatedResult
```

“Learning” Arbitrarily Complex Functions

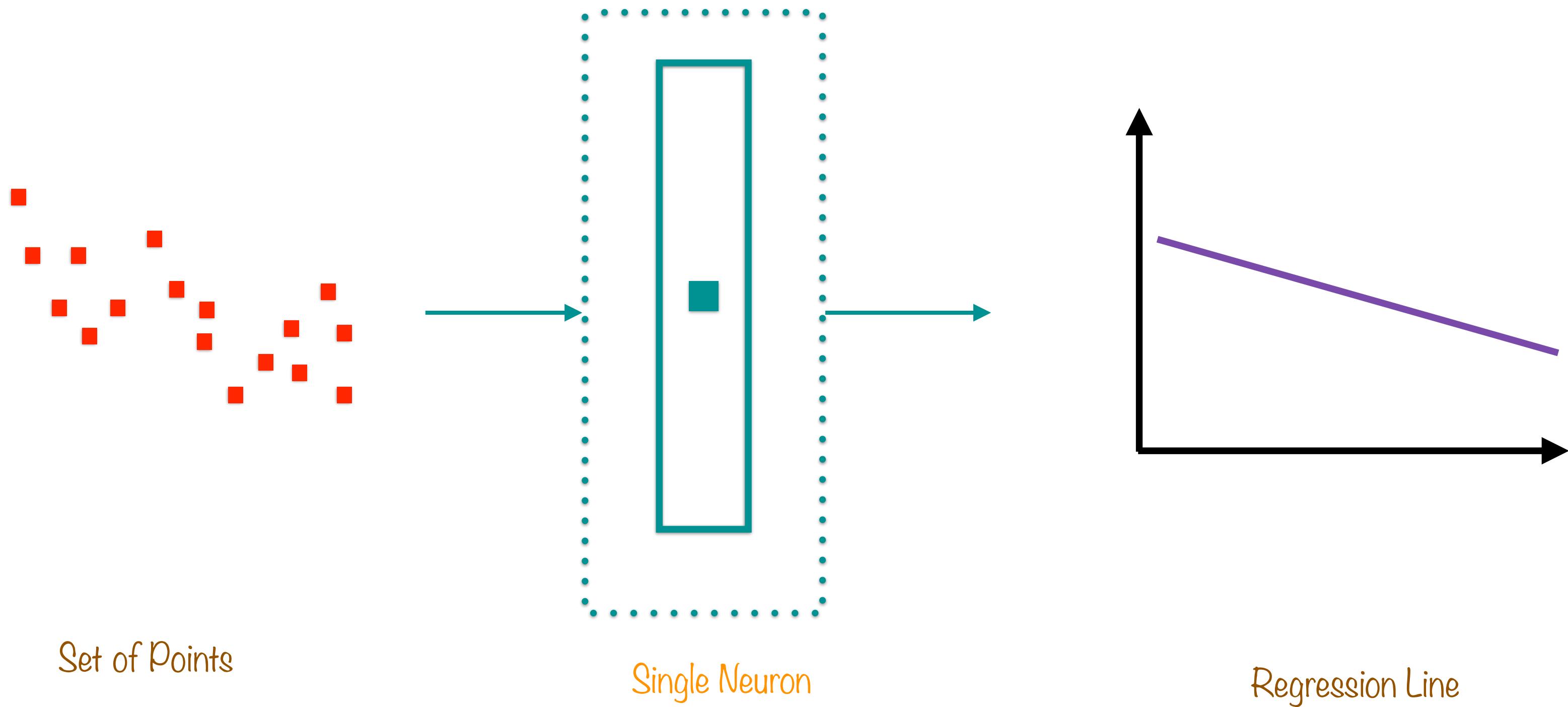
Adding layers to a neural network can “learn” (reverse-engineer) pretty much anything

$$y = Wx + b$$

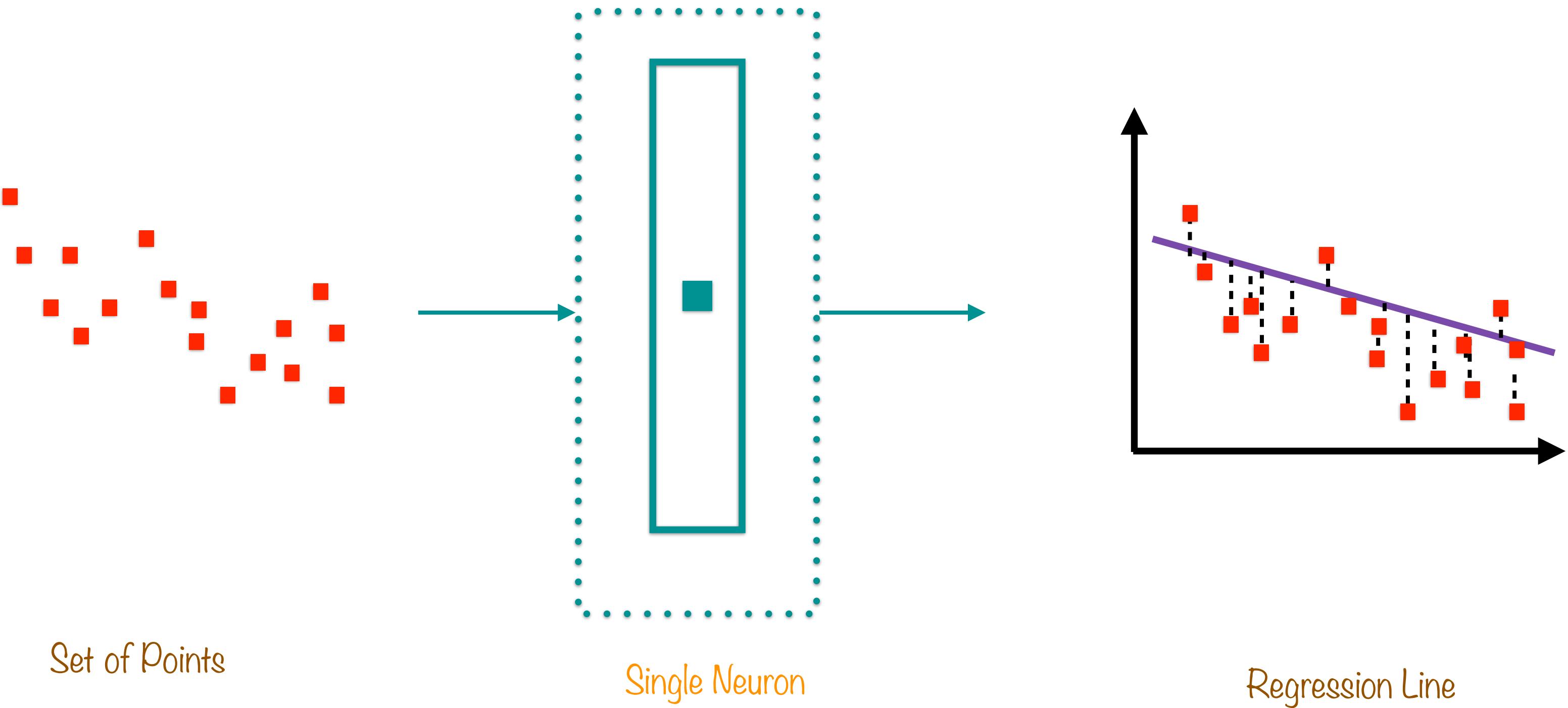
“Learning” Regression

Regression can be reverse-engineered by a single neuron

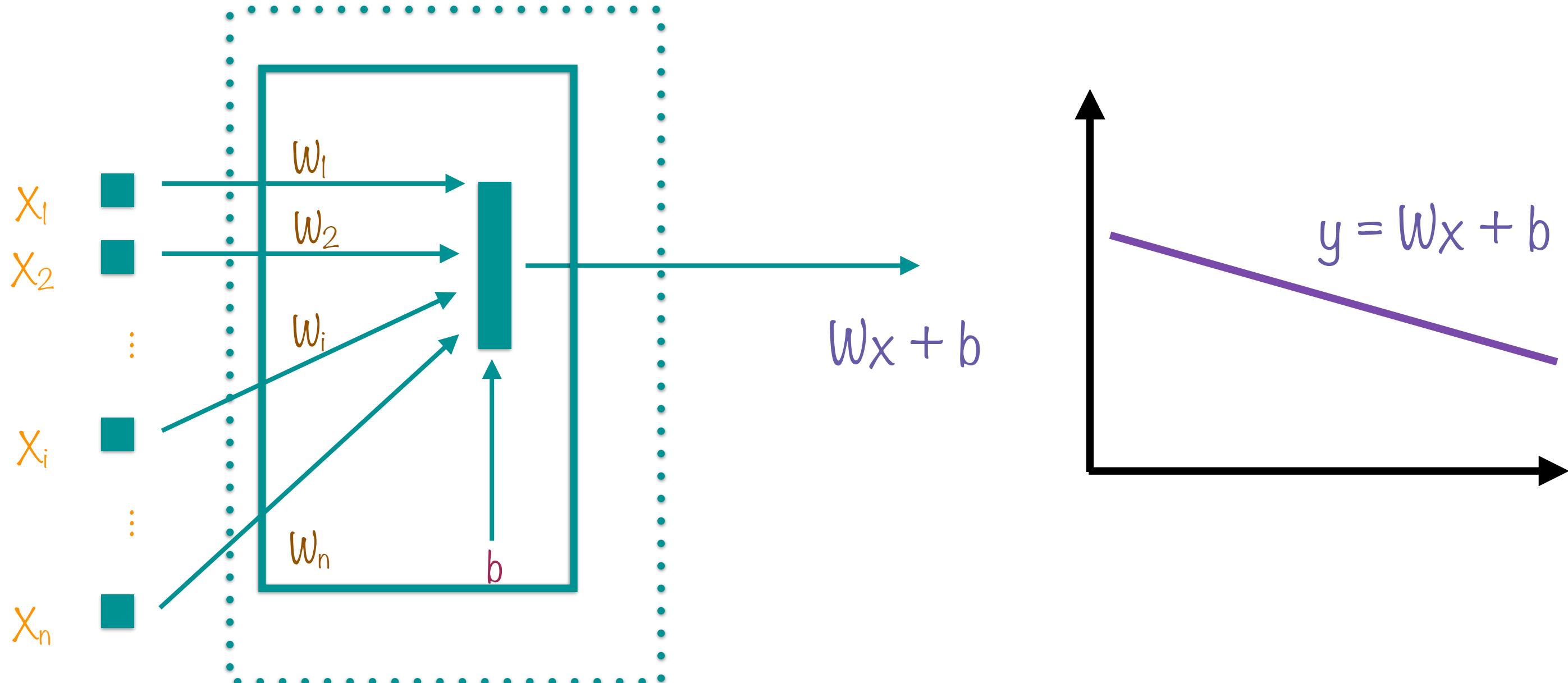
Regression: The Simplest Neural Network



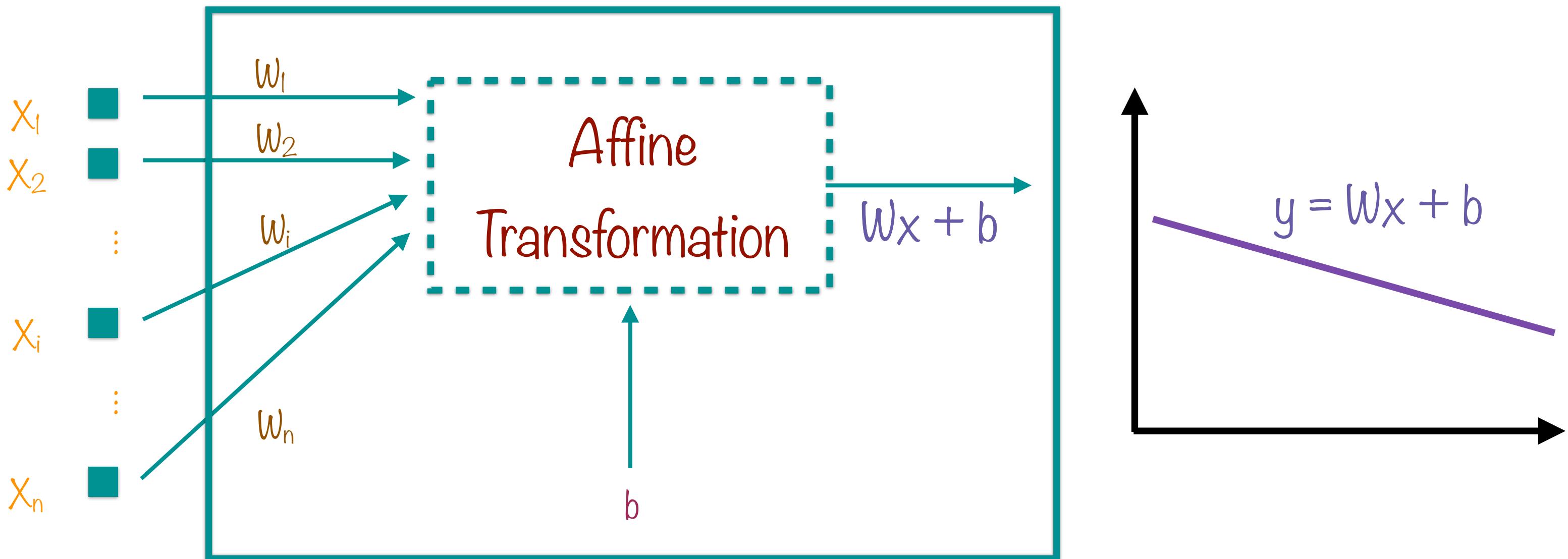
Regression: The Simplest Neural Network



Operation of a Single Neuron

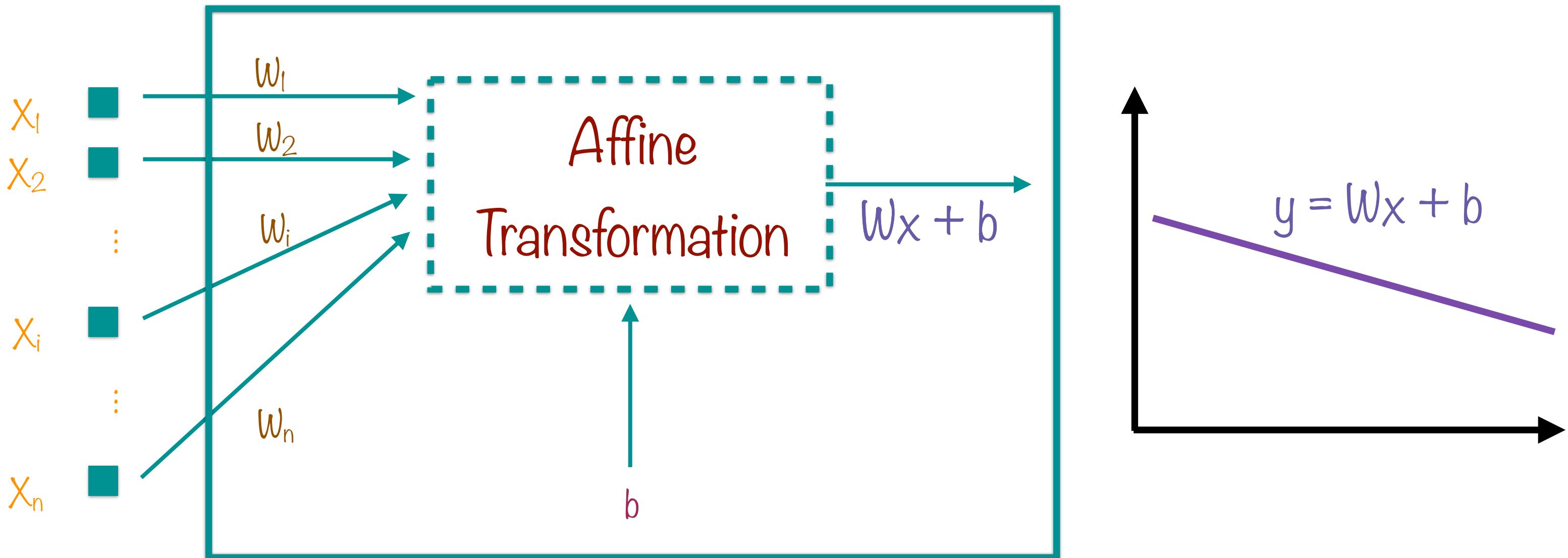


Operation of a Single Neuron



Here the neuron is an entity that finds the “best fit” line through a set of points

Operation of a Single Neuron



We are instructing the neuron to learn a linear function - so no activation
function is required at all

Operation of a Single Neuron

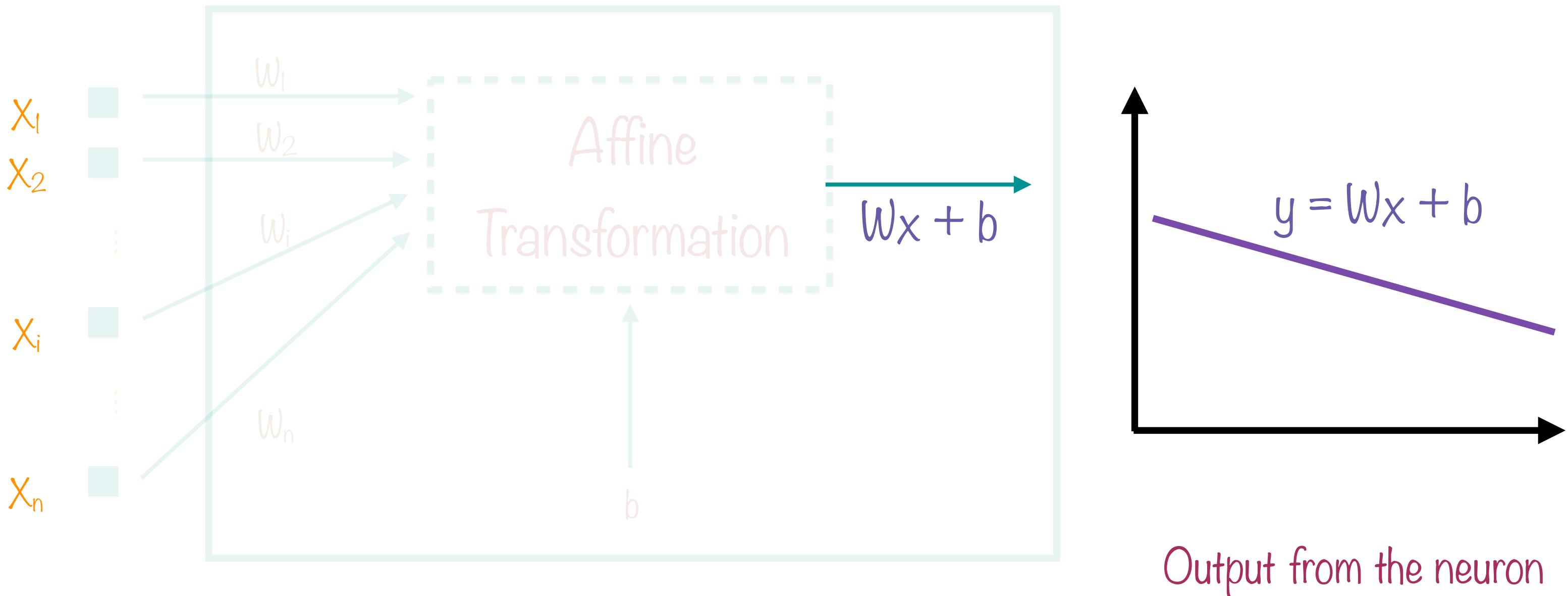


The affine transformation is just a weighted sum of the inputs with a bias added

Operation of a Single Neuron



Operation of a Single Neuron

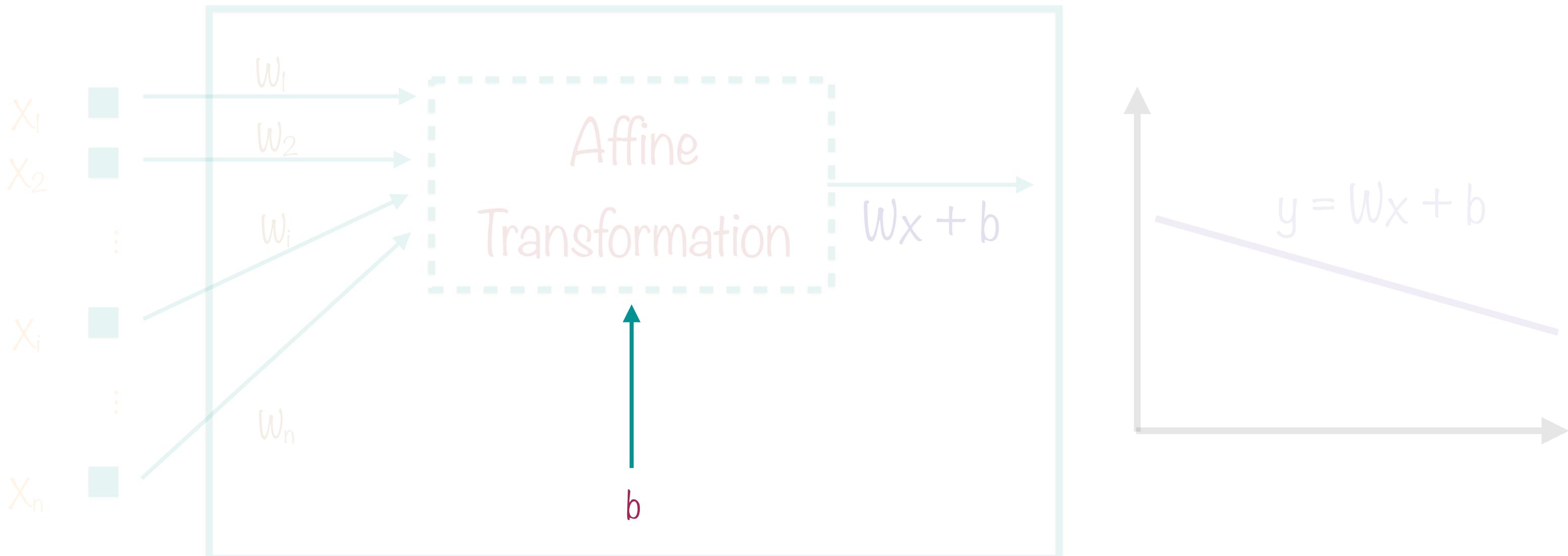


Operation of a Single Neuron



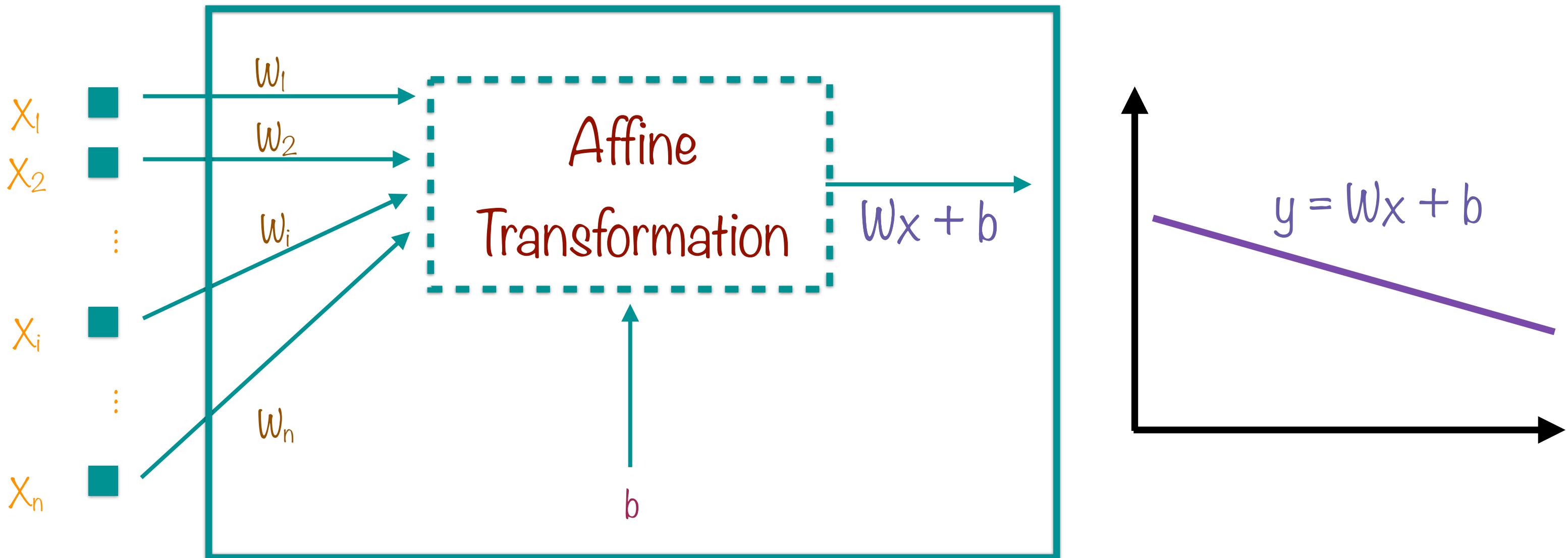
The values $w_1, w_2\dots w_n$ are called the weights

Operation of a Single Neuron



The value b is called the bias

Operation of a Single Neuron



Where do the weights W and the bias b come from?

The weights and biases of individual neurons are determined during the training process

The actual training of a neural network is
managed by TensorFlow

Simple Regression

Regression Equation:

$$y = A + Bx$$

$$y_1 = A + Bx_1$$

$$y_2 = A + Bx_2$$

$$y_3 = A + Bx_3$$

...

...

$$y_n = A + Bx_n$$

Simple Regression

Regression Equation:

$$y = A + Bx$$

$$y_1 = A + Bx_1 + e_1$$

$$y_2 = A + Bx_2 + e_2$$

$$y_3 = A + Bx_3 + e_3$$

...

...

...

$$y_n$$

$$A + Bx_n + e_n$$

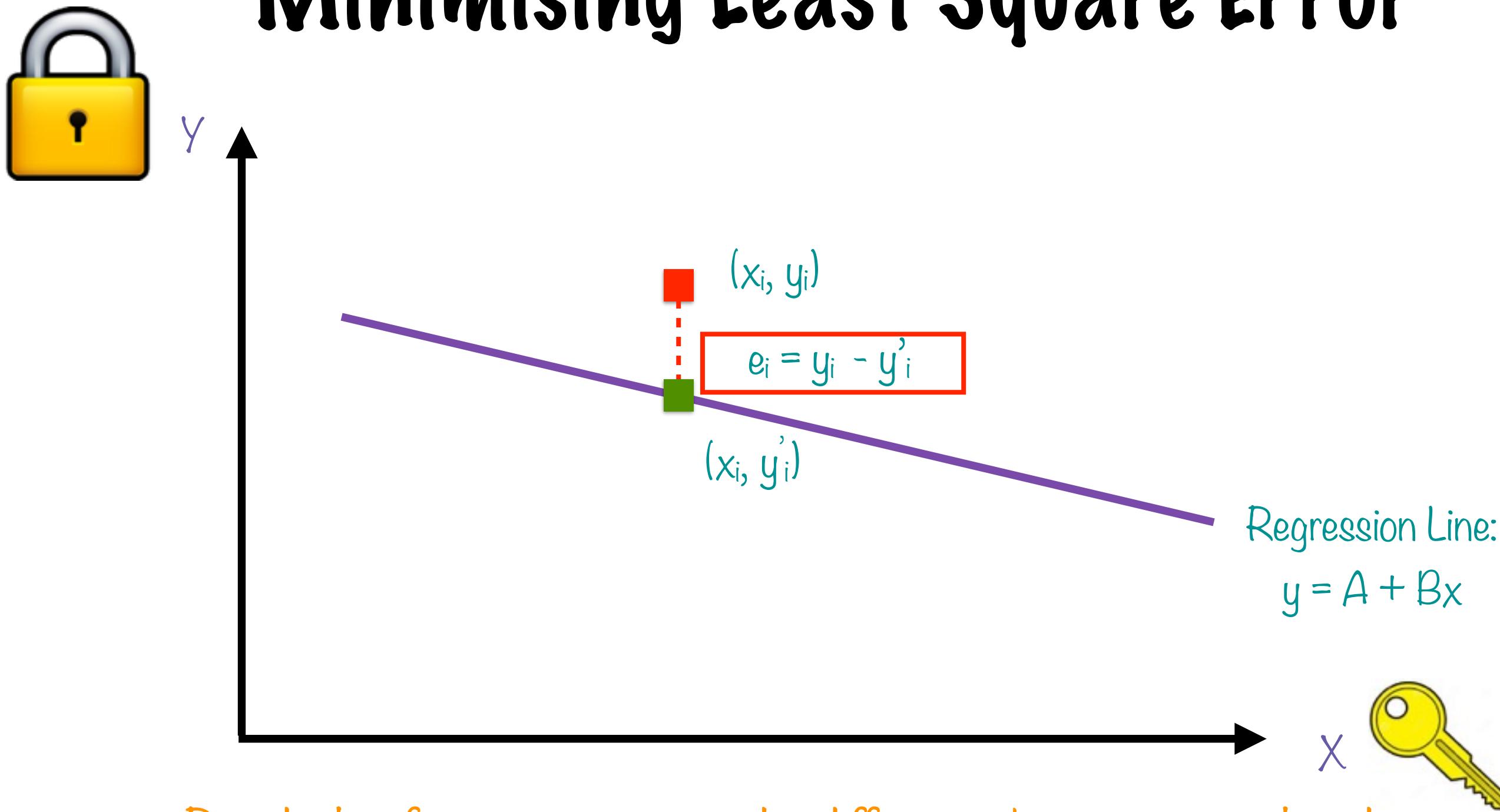
Simple Regression

Regression Equation:

$$y = A + Bx$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \\ y_n \end{bmatrix} = A \begin{bmatrix} 1 \\ \dots \\ 1 \end{bmatrix} + B \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ \dots \\ e_n \end{bmatrix}$$

Minimising Least Square Error



Residuals of a regression are the difference between actual and fitted values of the dependent variable

The “Best” Regression Line

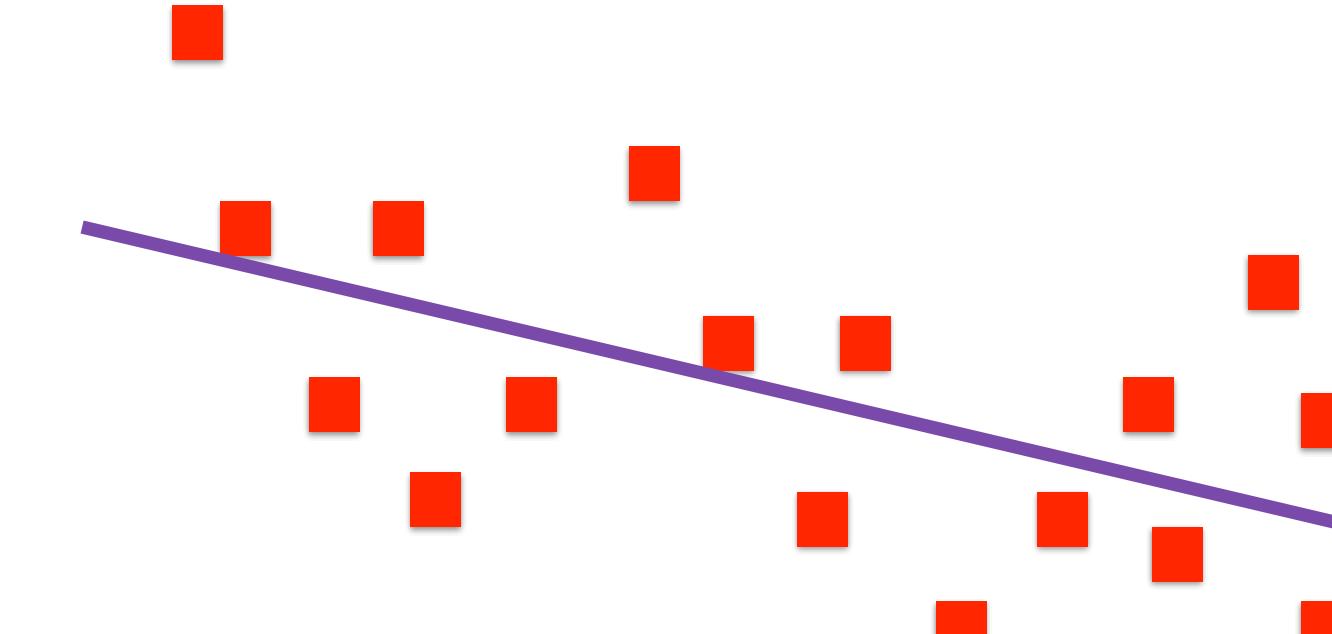


Linear Regression involves finding the “best fit” line

The “Best” Regression Line



y



Line 1: $y = A_1 + B_1x$

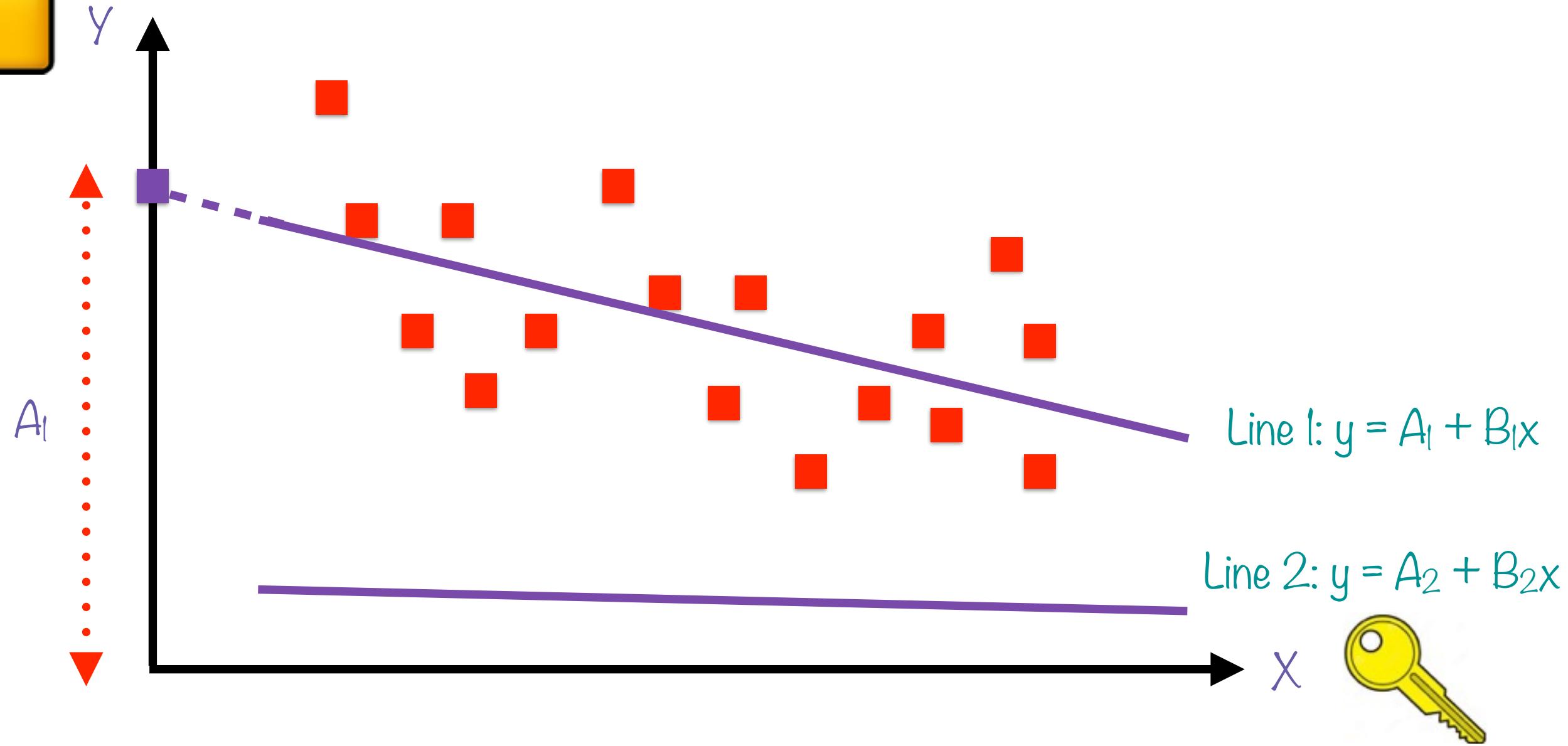
Line 2: $y = A_2 + B_2x$



x

Let's compare two lines, Line 1 and Line 2

The “Best” Regression Line



The first line has y -intercept A_1

The “Best” Regression Line



y

x increases by 1

y decreases by B_1

Line 1: $y = A_1 + B_1x$

Line 2: $y = A_2 + B_2x$

x



In the first line, if x increases by 1 unit, y decreases by B_1 units

The “Best” Regression Line



The second line has y-intercept A_2

The “Best” Regression Line

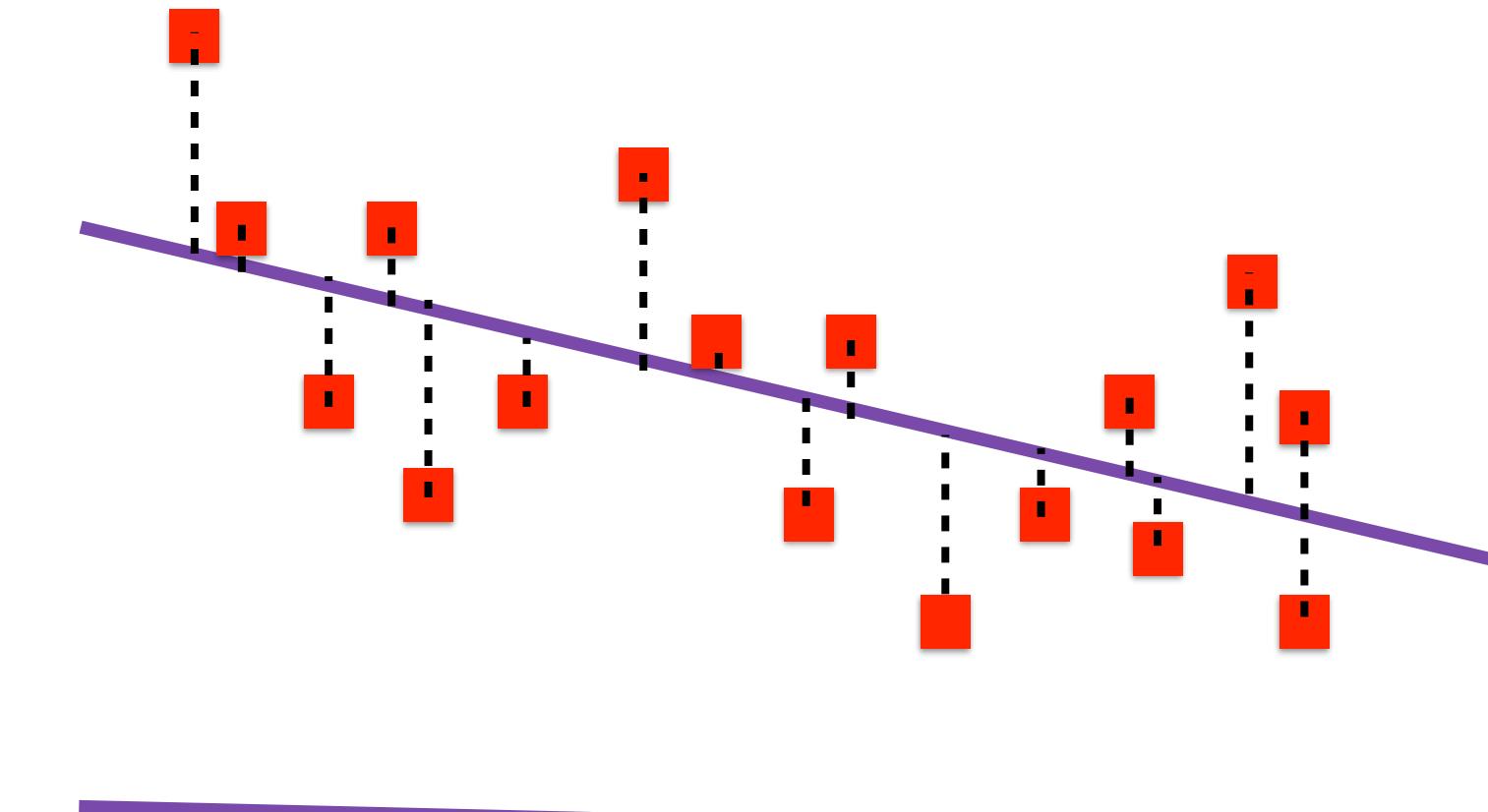


In the second line, if x increases by 1 unit, y decreases by B_2 units

Minimising Least Square Error



y



Line 1: $y = A_1 + B_1x$

Line 2: $y = A_2 + B_2x$

x

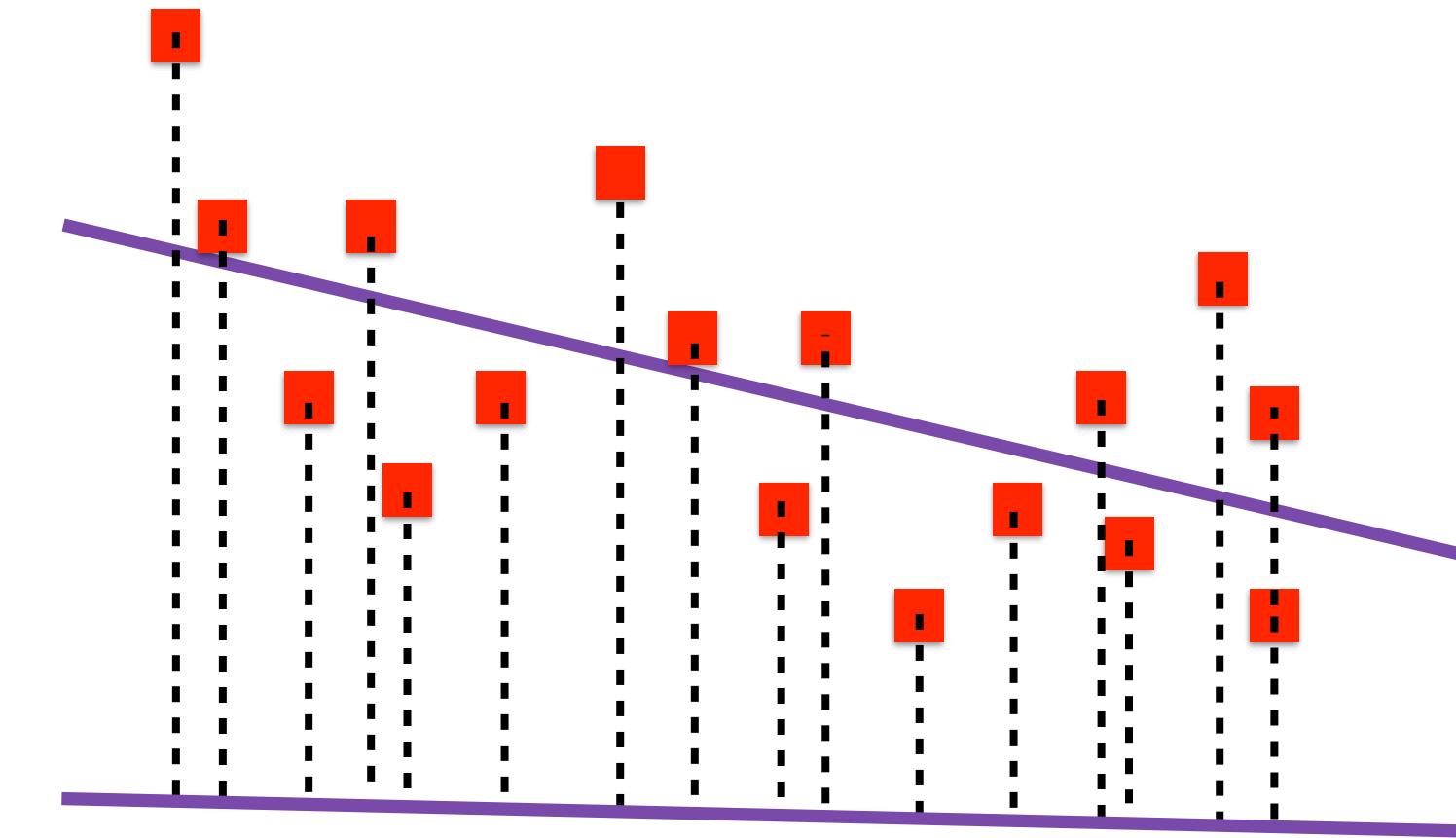


Drop vertical lines from each point to the lines A and B

Minimising Least Square Error



y



Line 1: $y = A_1 + B_1x$

Line 2: $y = A_2 + B_2x$

x



Drop vertical lines from each point to the lines A and B

Simple Regression

Regression Equation:

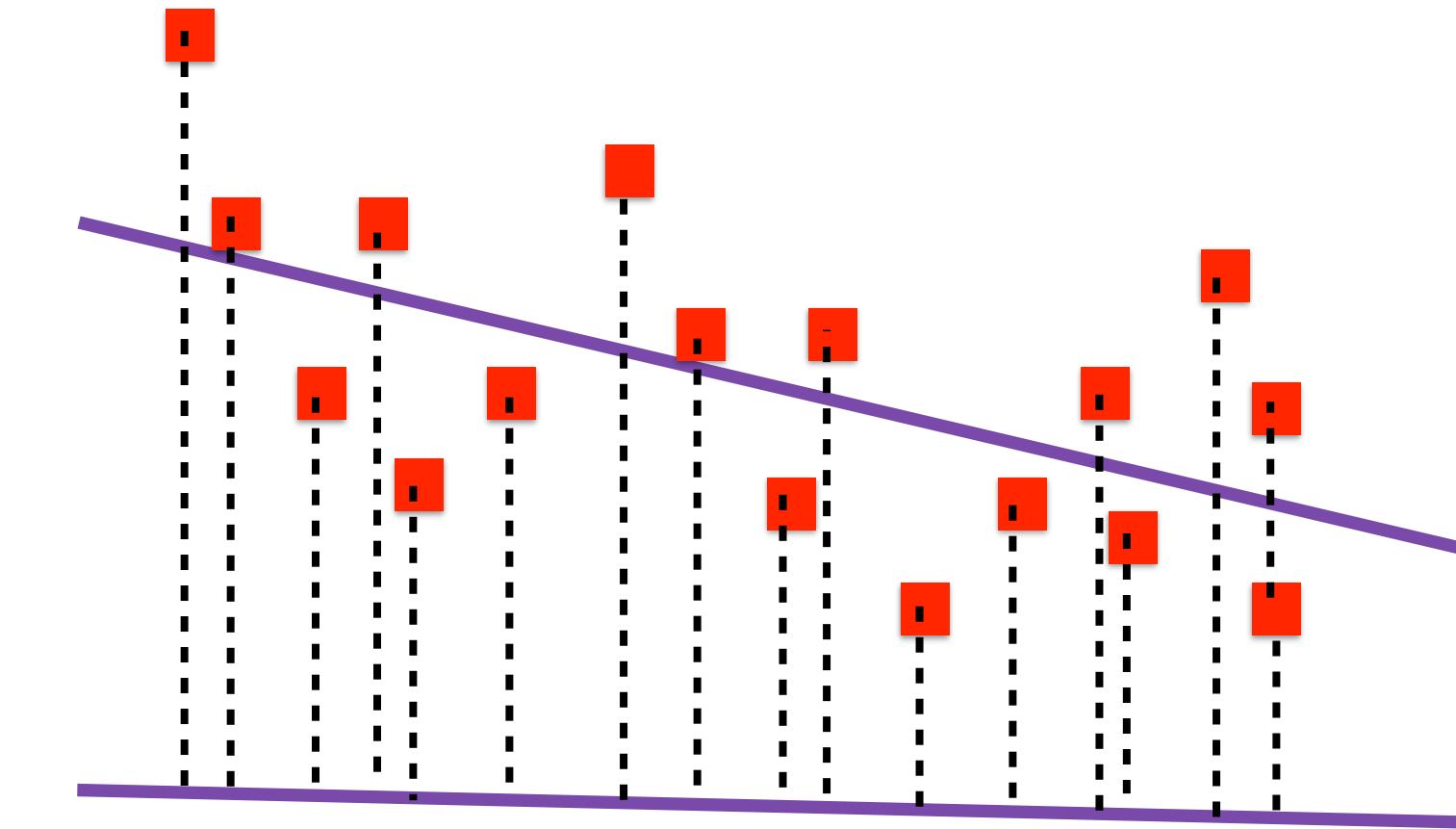
$$y = A + Bx$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \\ y_n \end{bmatrix} = A \begin{bmatrix} 1 \\ \dots \\ 1 \end{bmatrix} + B \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ \dots \\ e_n \end{bmatrix}$$

Minimising Least Square Error



y



Line 1: $y = A_1 + B_1x$

Line 2: $y = A_2 + B_2x$

x

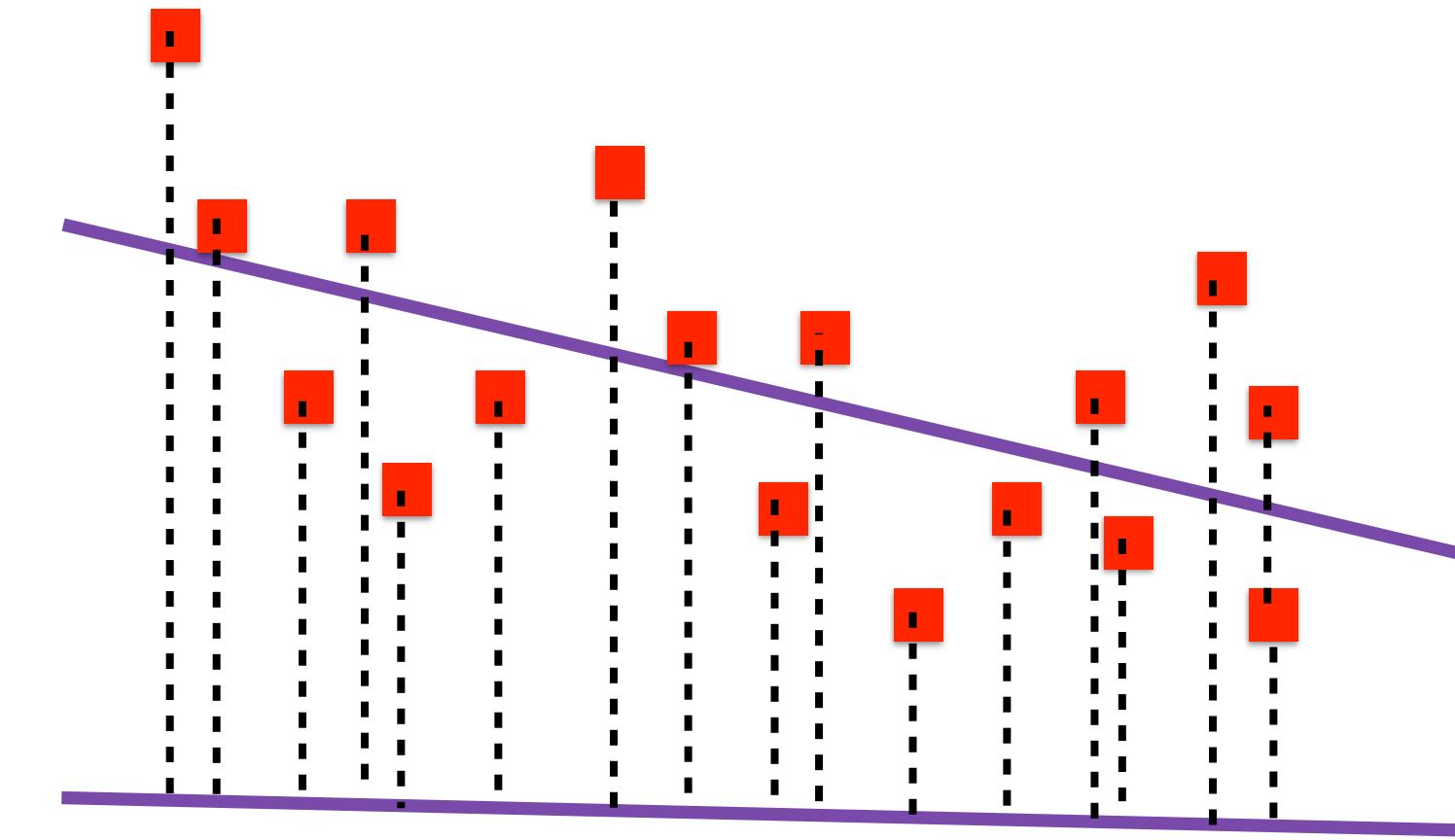


The “best fit” line is the one where the sum of the squares of the lengths of these dotted lines is minimum

Minimising Least Square Error



y



Line 1: $y = A_1 + B_1x$

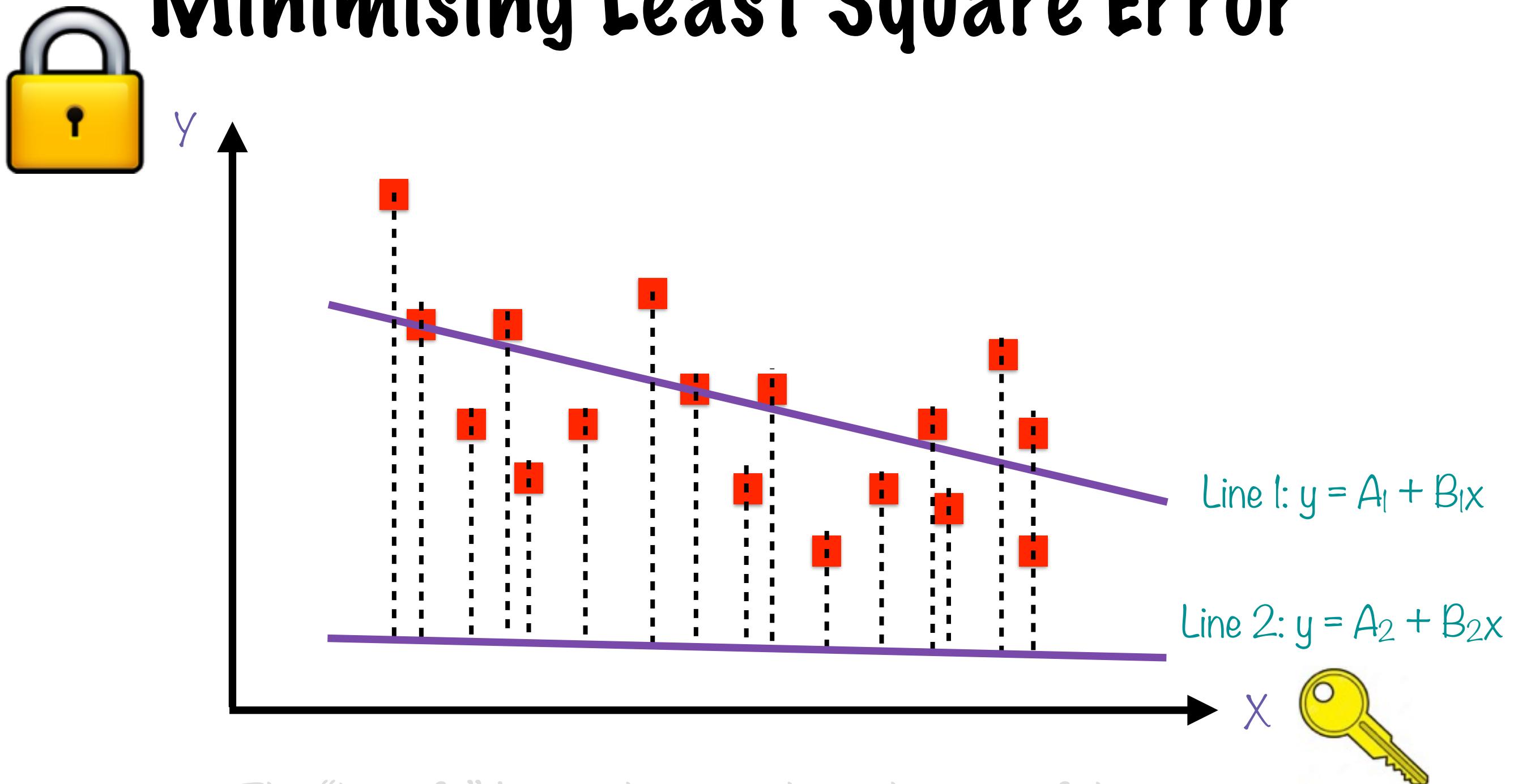
Line 2: $y = A_2 + B_2x$

x



The “best fit” line is the one where the sum of the squares of the lengths of these dotted lines is minimum

Minimising Least Square Error

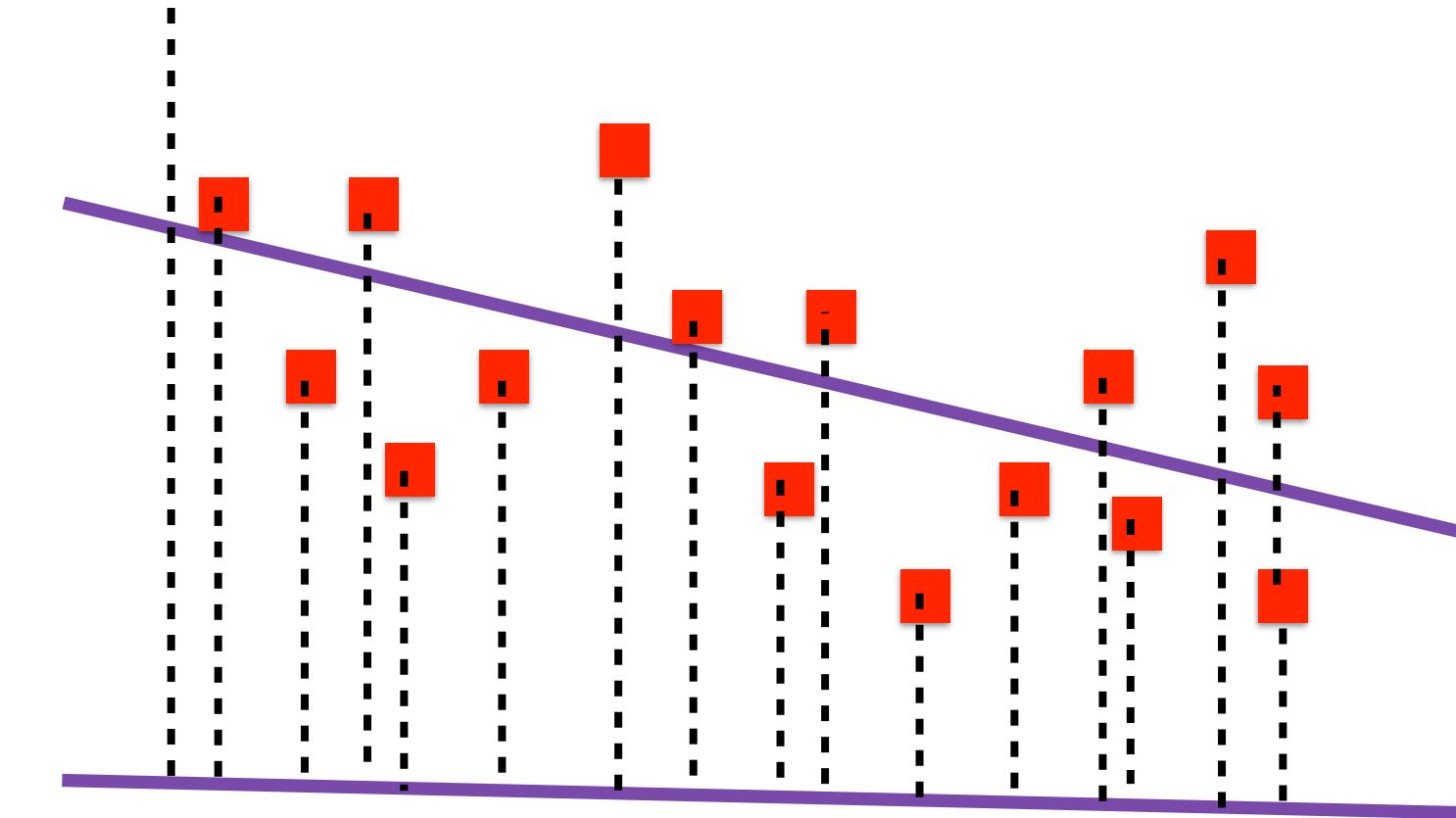


The “best fit” line is the one where the sum of the squares of the lengths of the errors is minimum

Minimising Least Square Error



y



Line 1: $y = A_1 + B_1x$

Line 2: $y = A_2 + B_2x$

x



The “best fit” line is the one where the sum of the squares of the lengths of the errors is minimum

Simple Regression

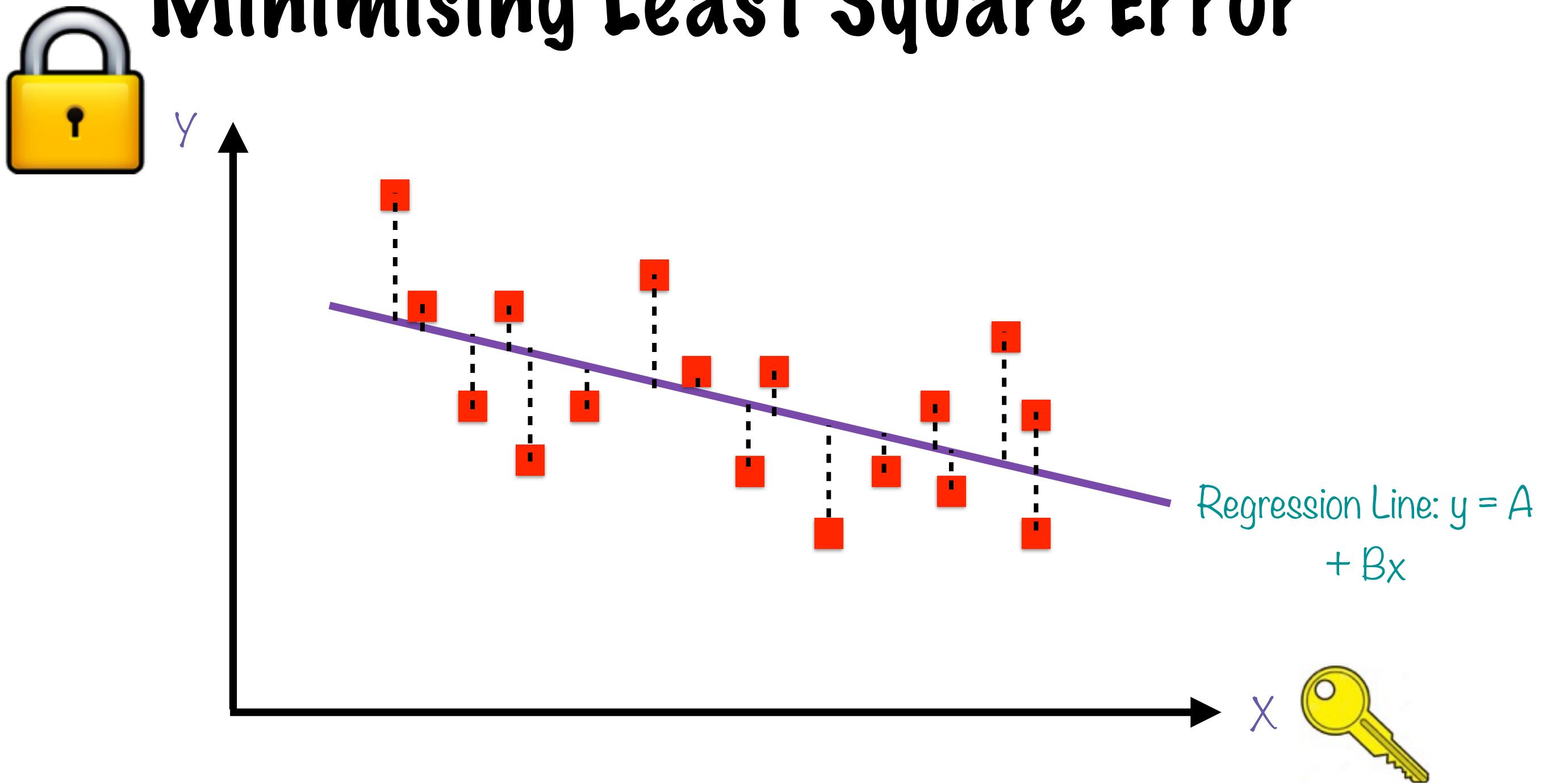
Regression Equation:

$$y = A + Bx$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \\ y_n \end{bmatrix} = A \begin{bmatrix} 1 \\ 1 \\ 1 \\ \dots \\ 1 \end{bmatrix} + B \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ \dots \\ e_n \end{bmatrix}$$

The “best fit” line is the one where the sum of the squares of the lengths of the errors is minimum

Minimising Least Square Error



The “best fit” line is called the regression line

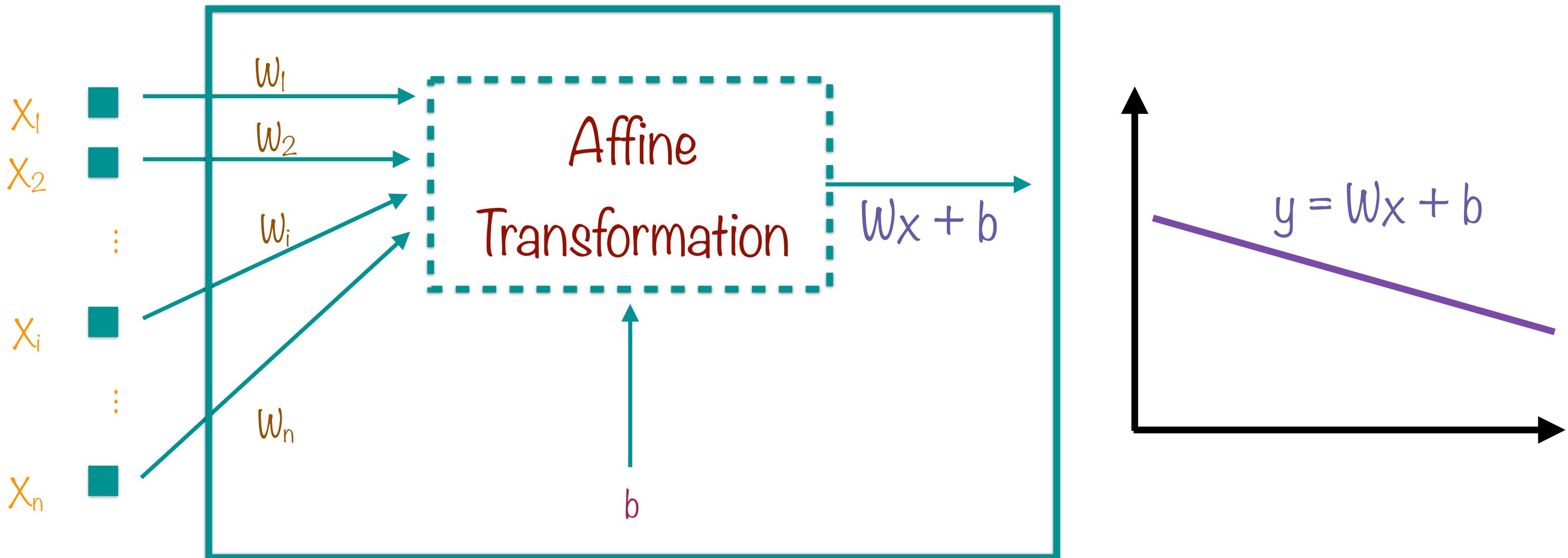
Optimizers for the “Best-fit”

Method of moments

Method of least squares

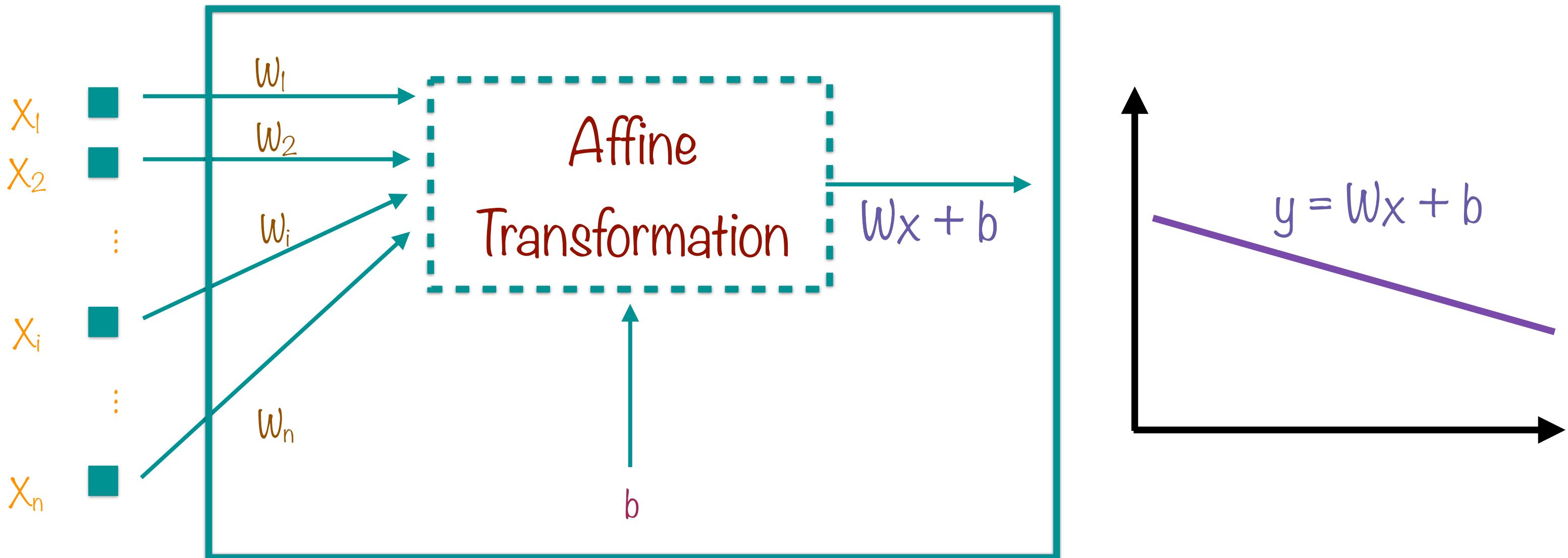
Maximum likelihood
estimation

Operation of a Single Neuron



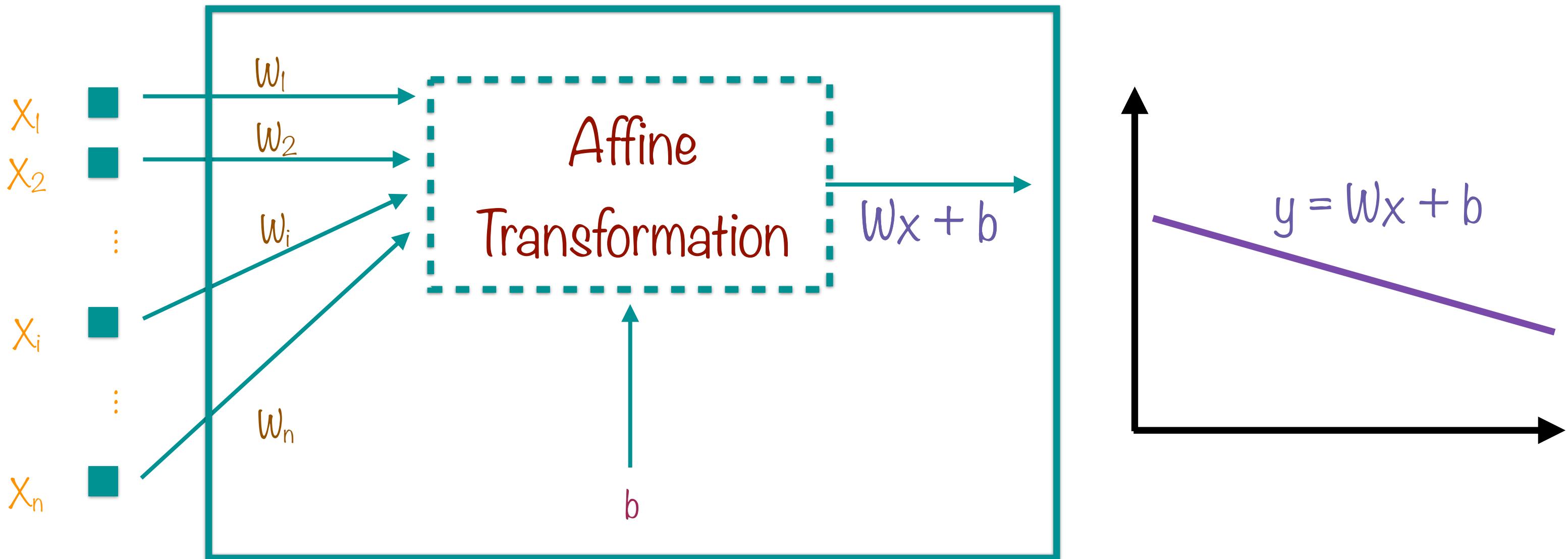
Where do the weights W and the bias b come from?

Operation of a Single Neuron



Where do the weights W and the bias b come from?
They are determined during the training process

Operation of a Single Neuron



This optimization is **not** carried out by the individual neuron, rather
a training algorithm will take care of this

A Slightly More Complex Neural Network

```
def doSomethingReallyComplicated(x1,x2...):
```

```
...
```

```
...
```

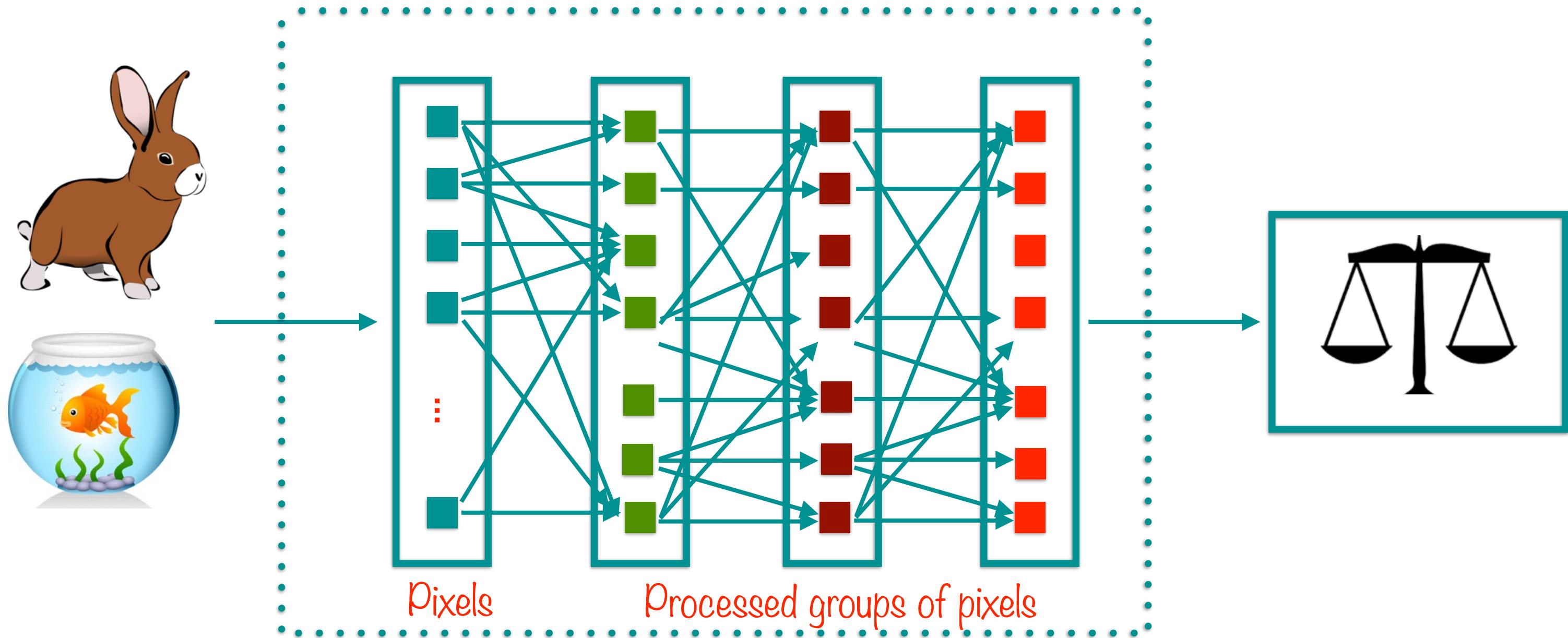
```
...
```

```
    return complicatedResult
```

“Learning” Arbitrarily Complex Functions

Adding layers to a neural network can “learn” (reverse-engineer) pretty much anything

An Arbitrarily Complex Function



Corpus of
Images

Each layer consists of individual interconnected
neurons

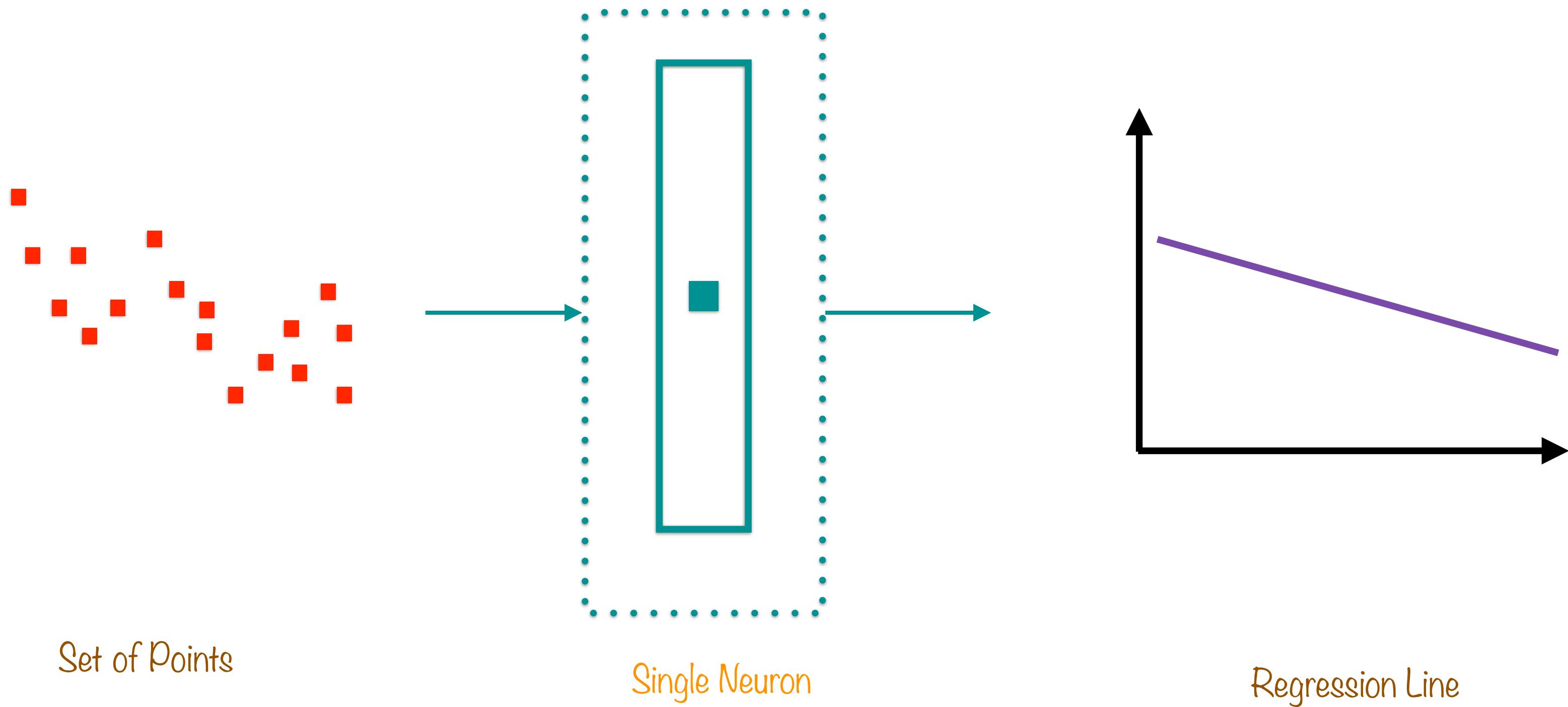
ML-based Classifier

$$y = Wx + b$$

“Learning” Regression

Regression can be learnt by a single neuron using an affine transformation alone

Regression: The Simplest Neural Network

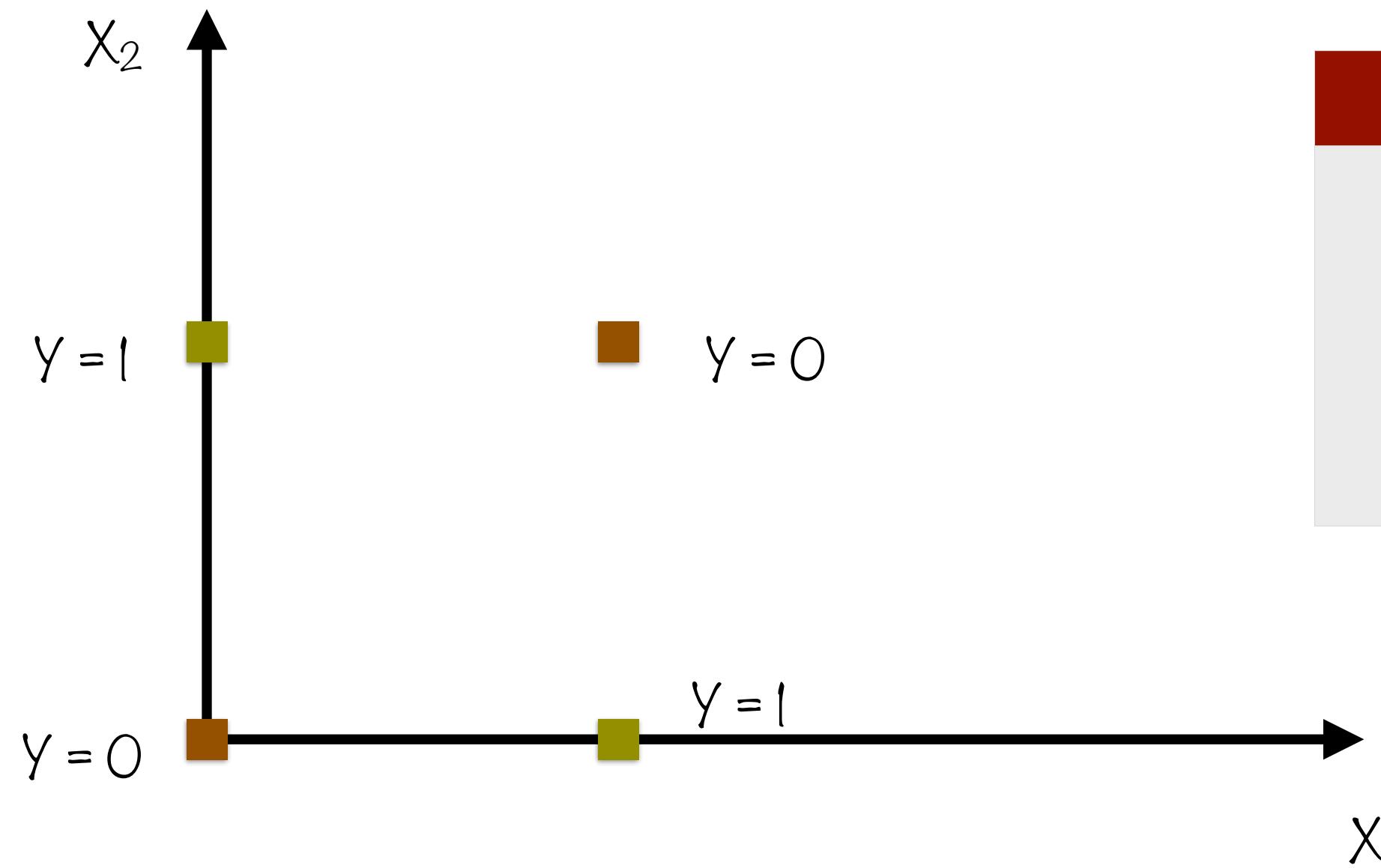


```
def XOR(x1,x2):  
    if (x1 == x2):  
        return 0  
    return 1
```

“Learning” XOR

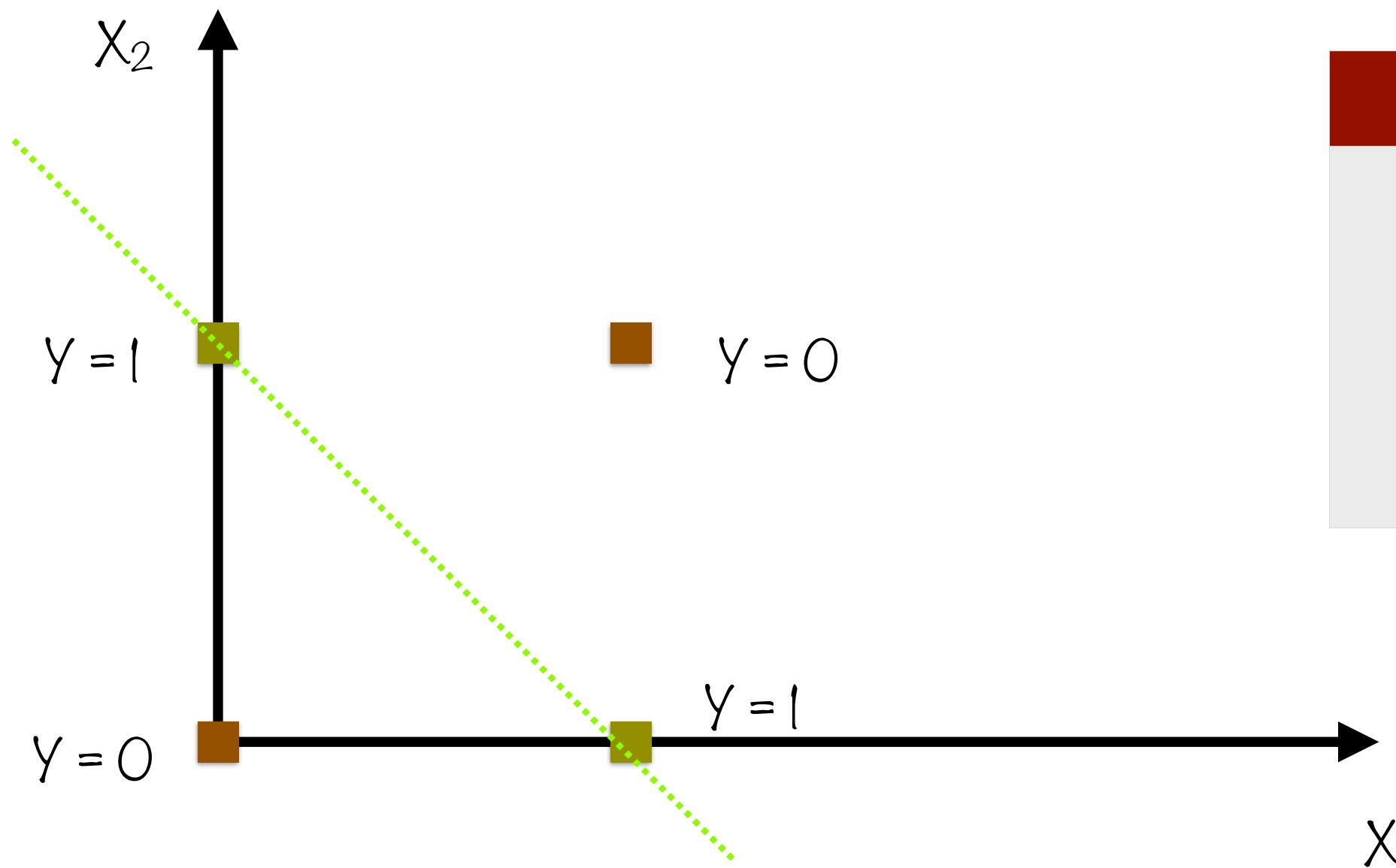
Reverse-engineering XOR requires 3 neurons (arranged in 2 layers) as well as a non-linear activation function

XOR: Not Linearly Separable



| X_1 | X_2 | Y |
|-------|-------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

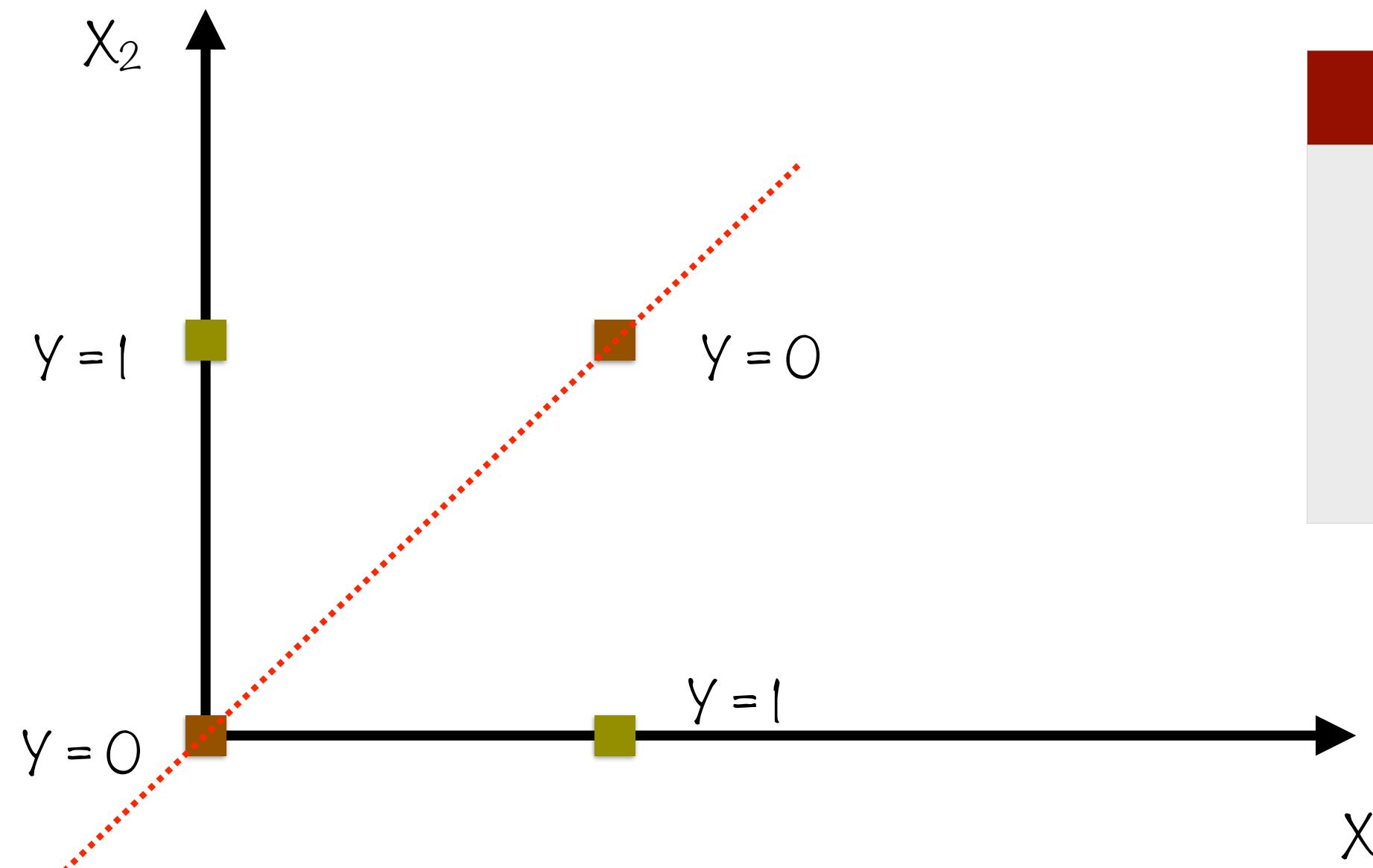
XOR: Not Linearly Separable



| X_1 | X_2 | Y |
|-------|-------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

No one straight line neatly divides the points into disjoint regions where $Y = 0$ and $Y = 1$

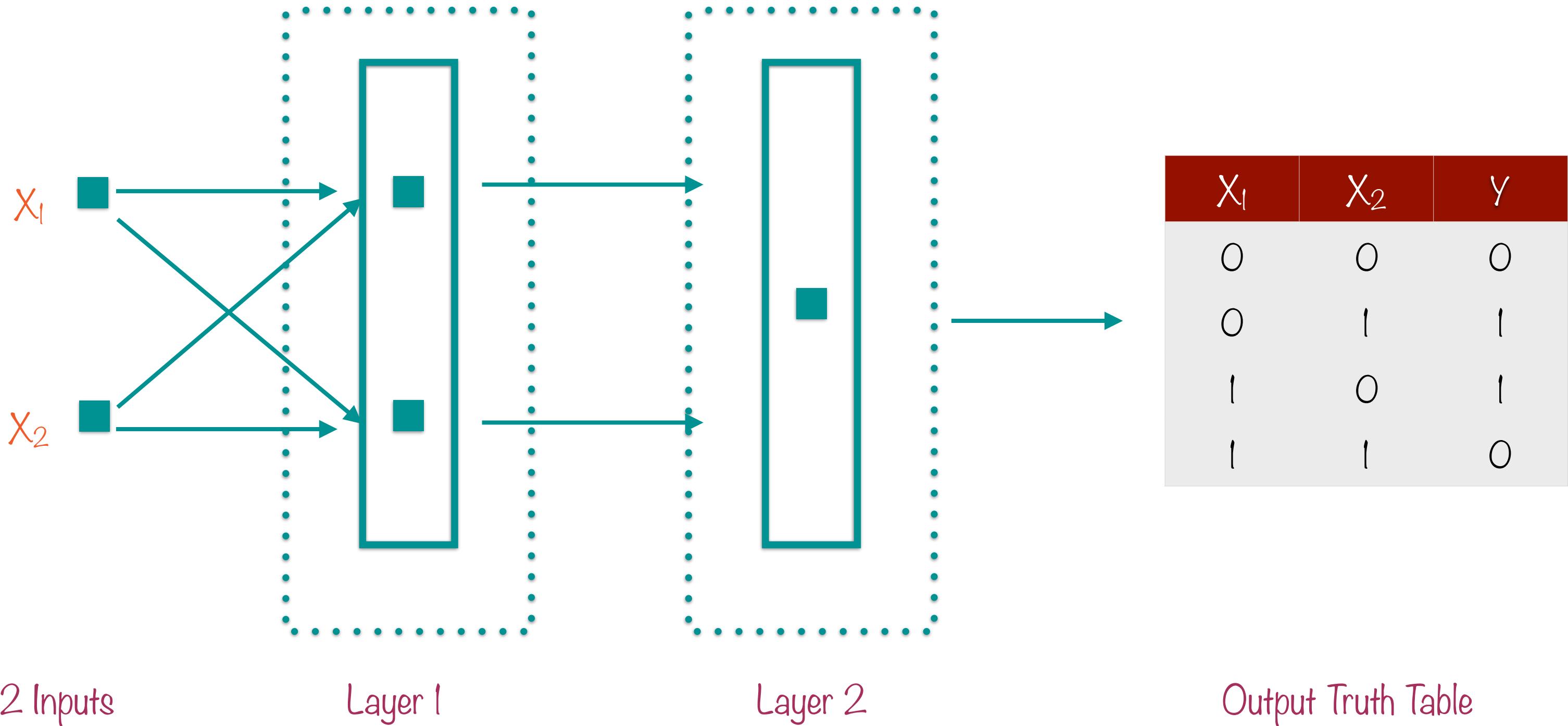
XOR: Not Linearly Separable



| X_1 | X_2 | Y |
|-------|-------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

No one straight line neatly divides the points into disjoint regions where $Y = 0$ and $Y = 1$

XOR: 3 Neurons, 2 Layers

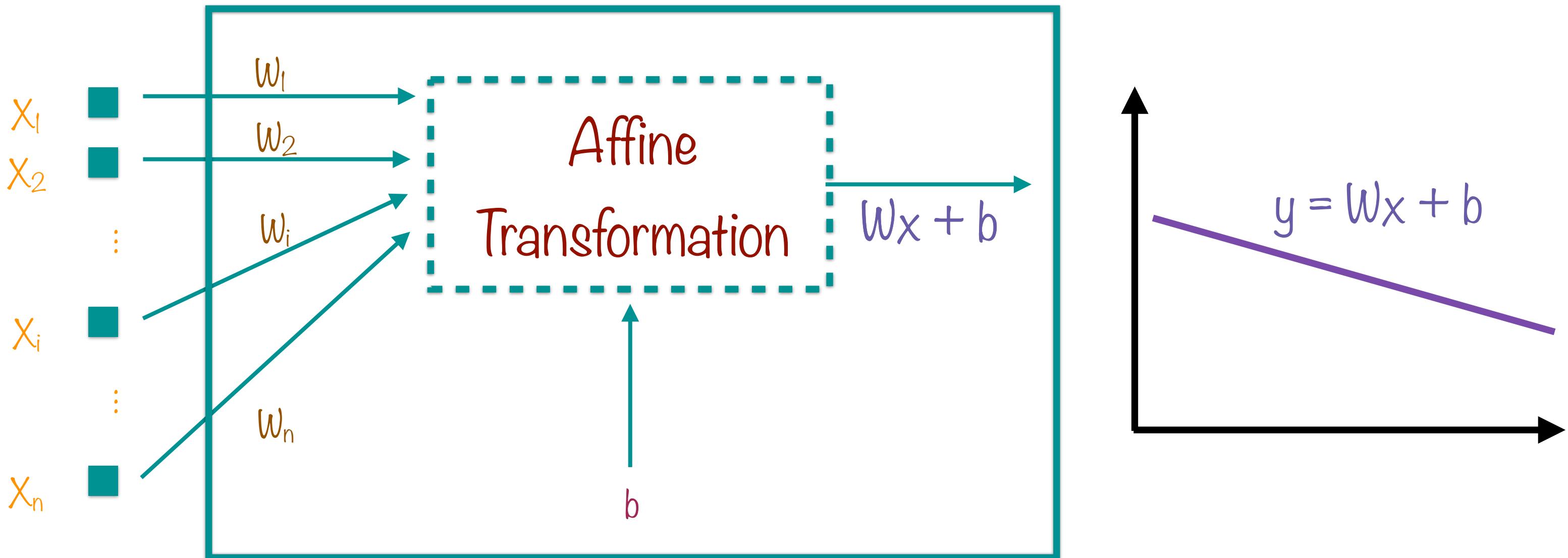


| X ₁ | X ₂ | Y |
|----------------|----------------|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

“Learning” XOR

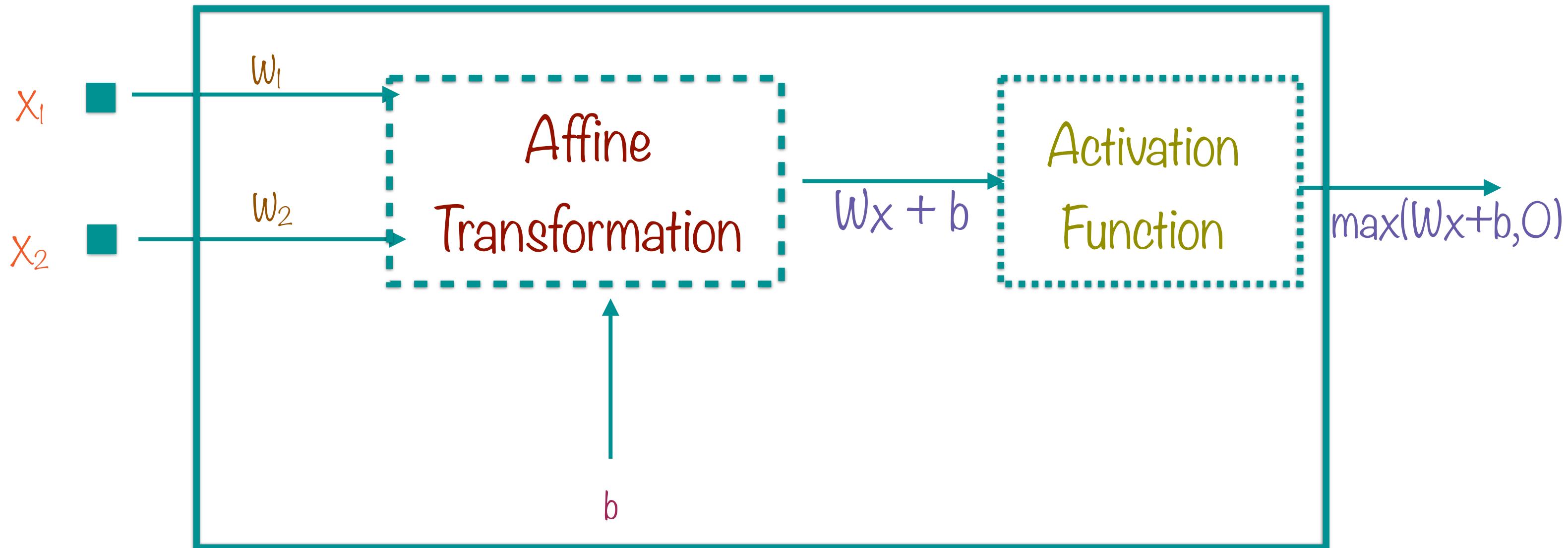
Reverse-engineering XOR requires 3 neurons (arranged in 2 layers) as well as a non-linear activation function

1-Neuron Regression



Regression could be learnt by a single neuron using a single, linear operation

Adding an Activation Function



XOR, a simple non-linear function, can be learnt by 3 neurons if we add an appropriate activation function



The activation function is needed for the neural network to predict non-linear functions

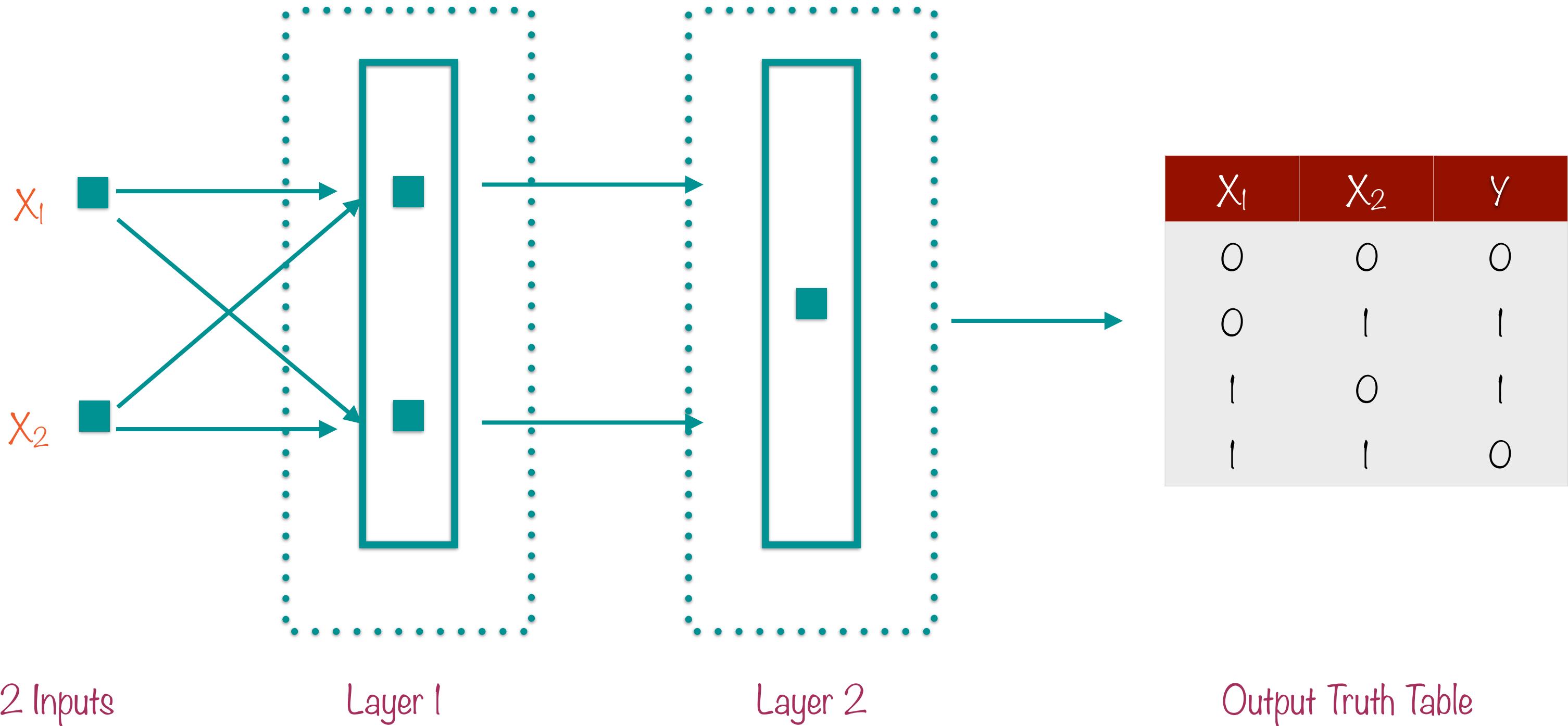


The most common form of the activation function is the
ReLU

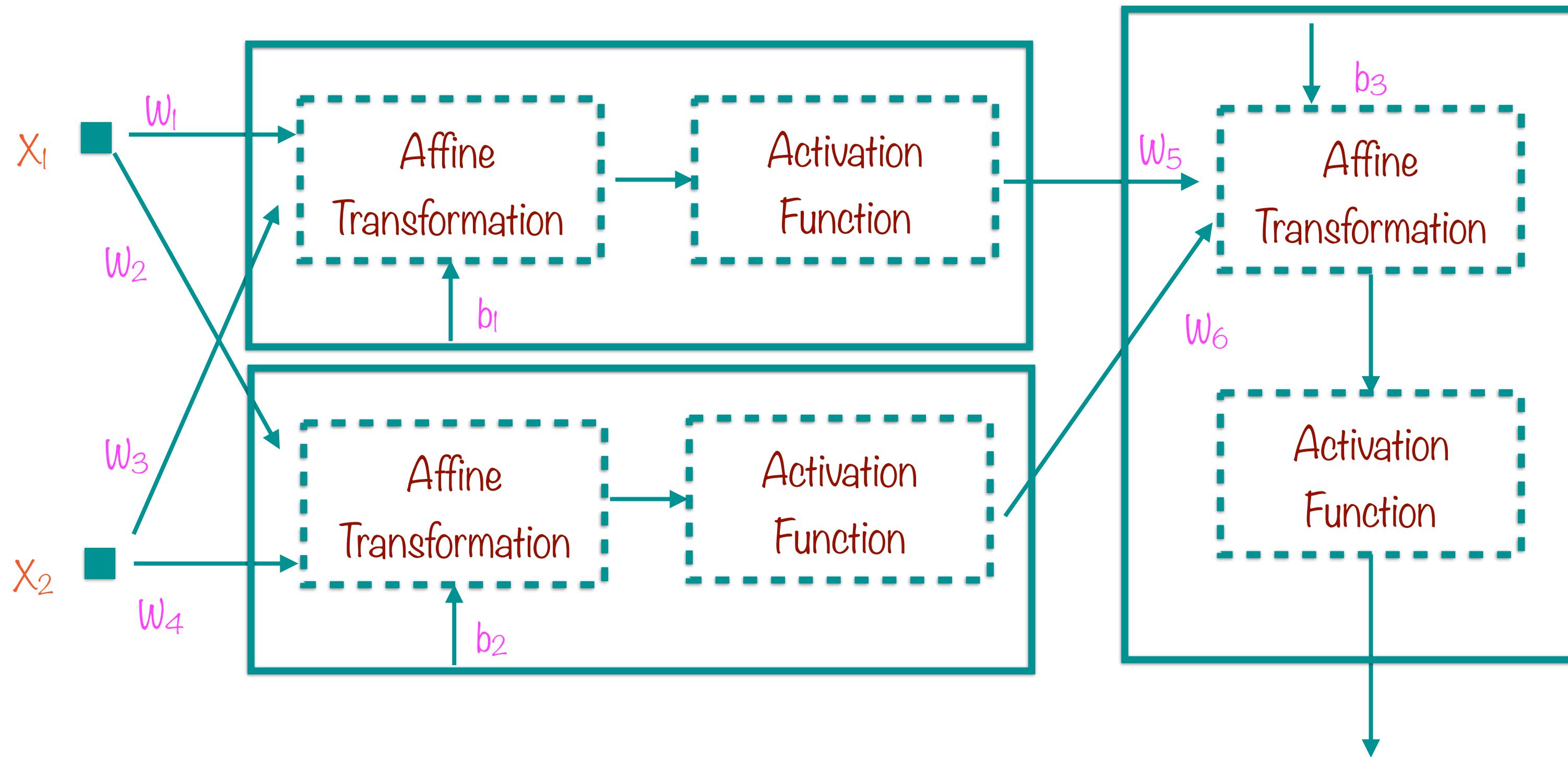
ReLU : Rectified Linear Unit

$$\text{ReLU}(x) = \max(0, x)$$

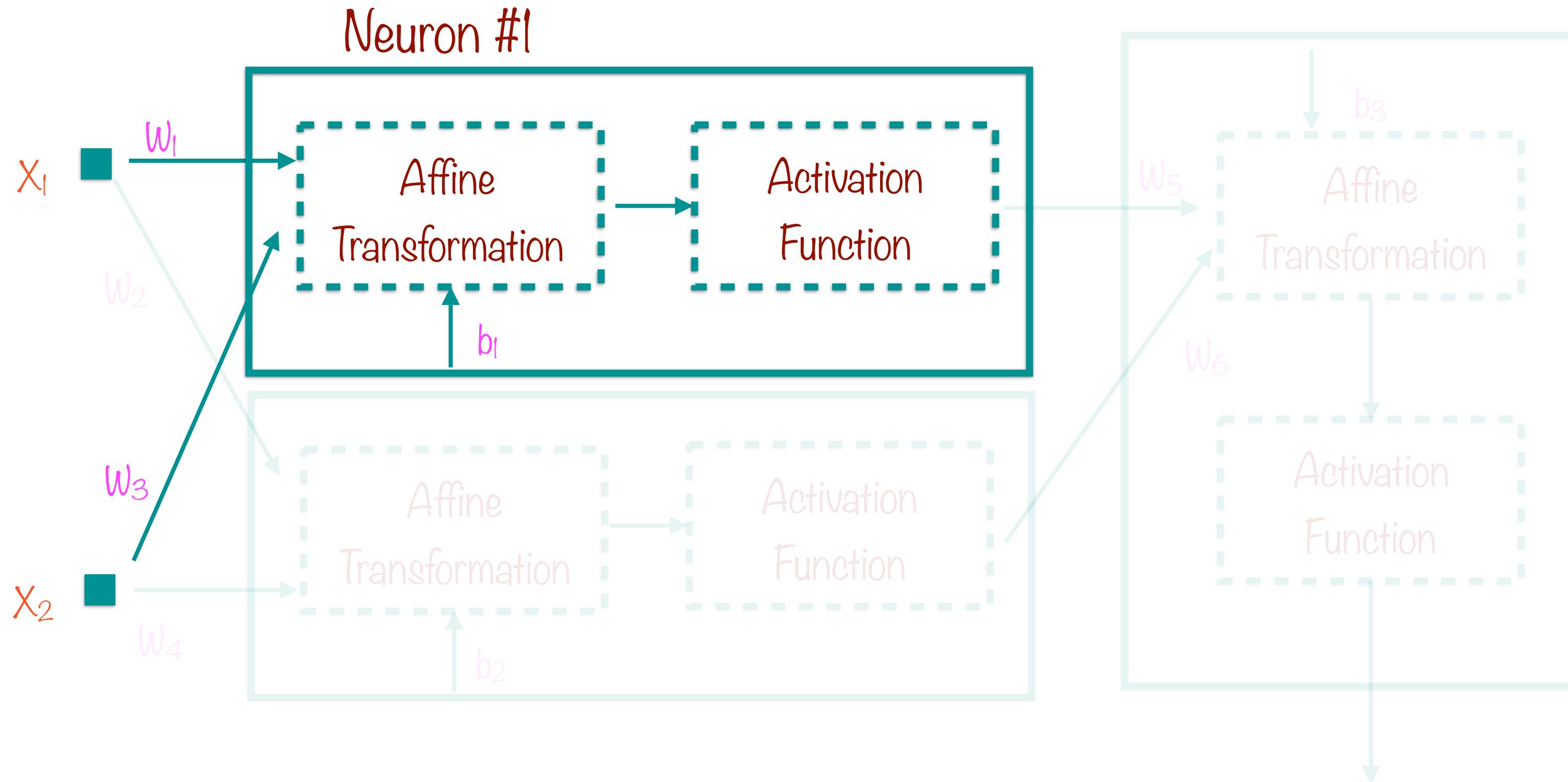
XOR: 3 Neurons, 2 Layers



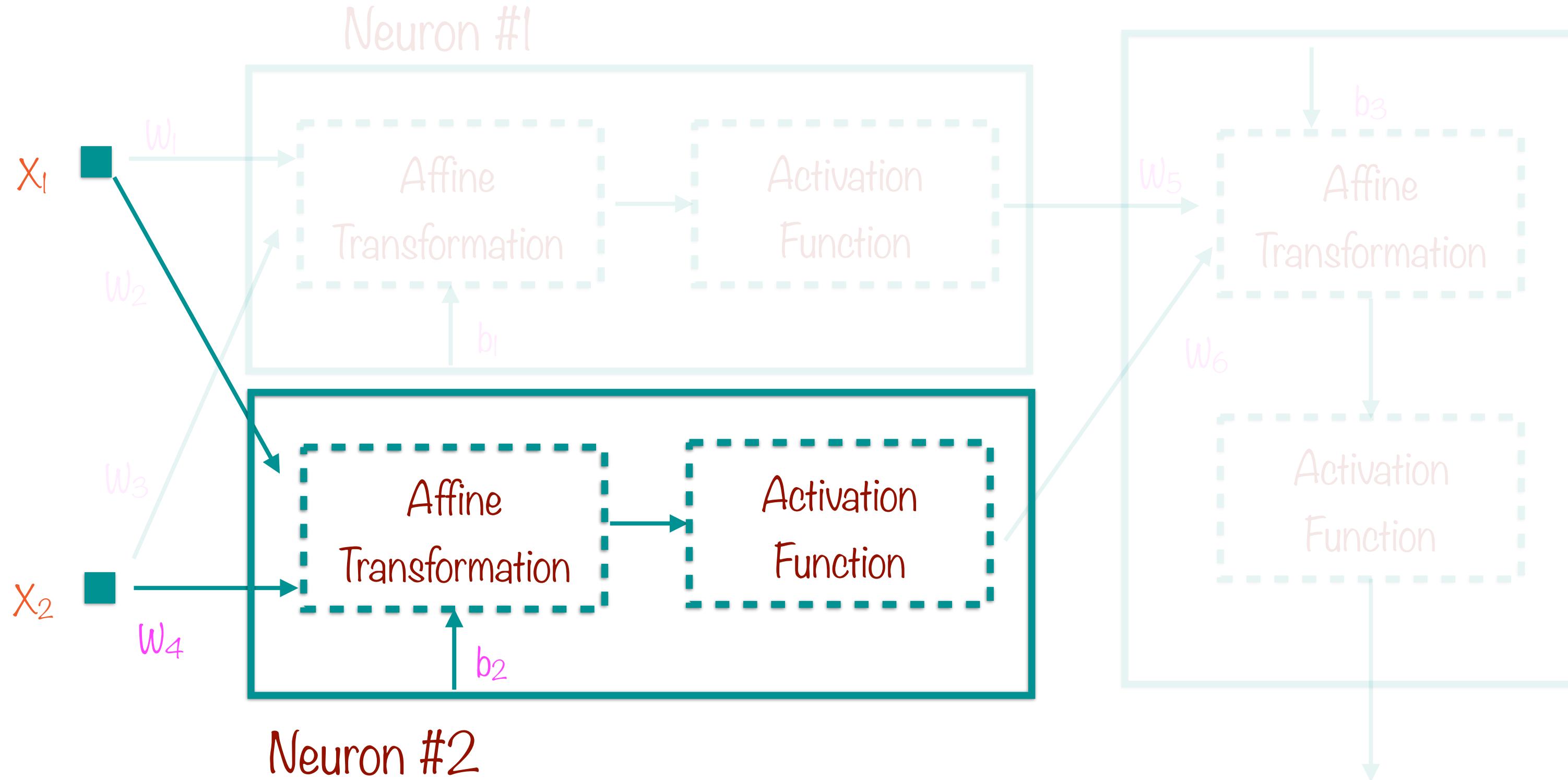
3-Neuron XOR



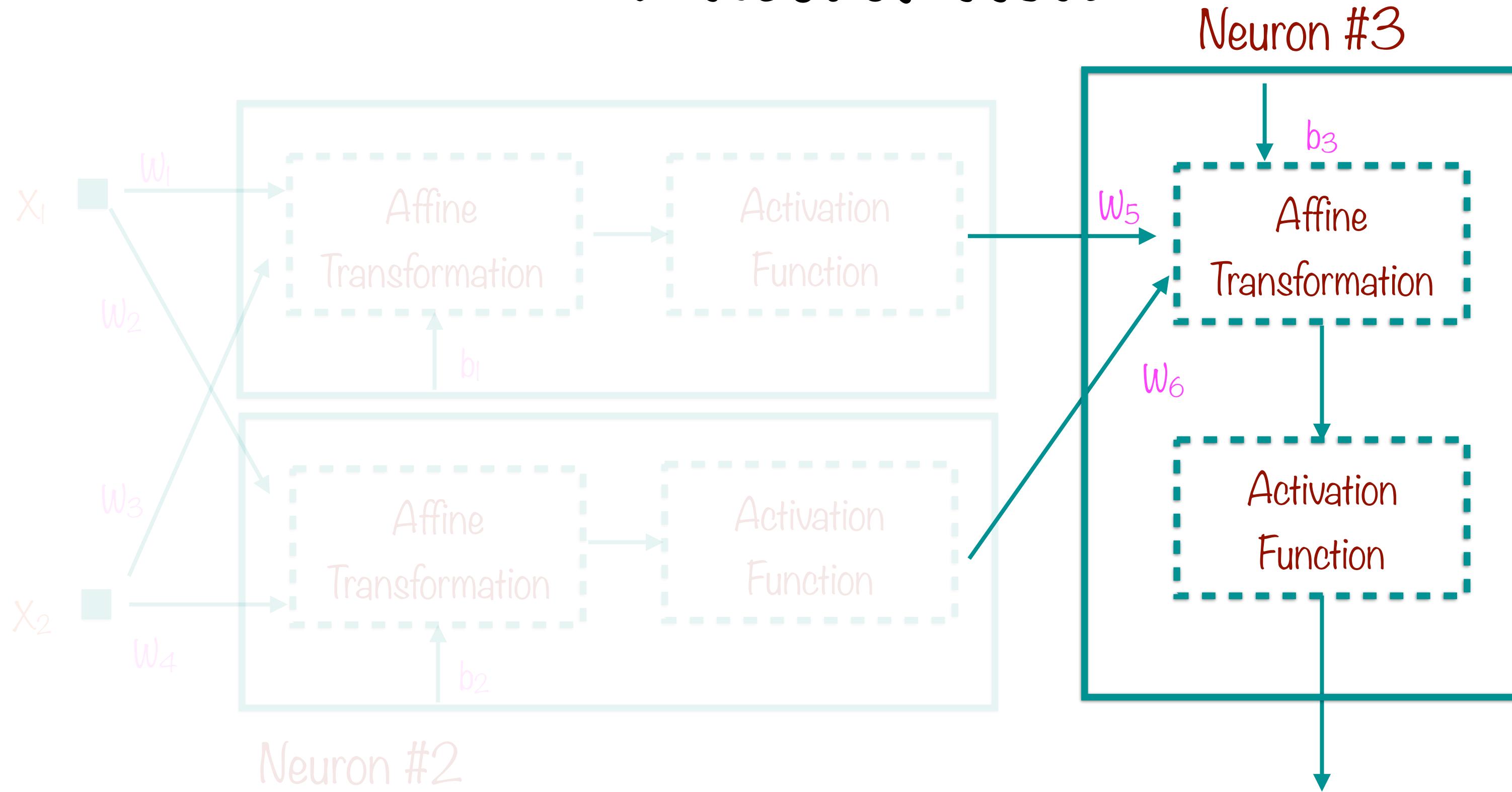
3-Neuron XOR



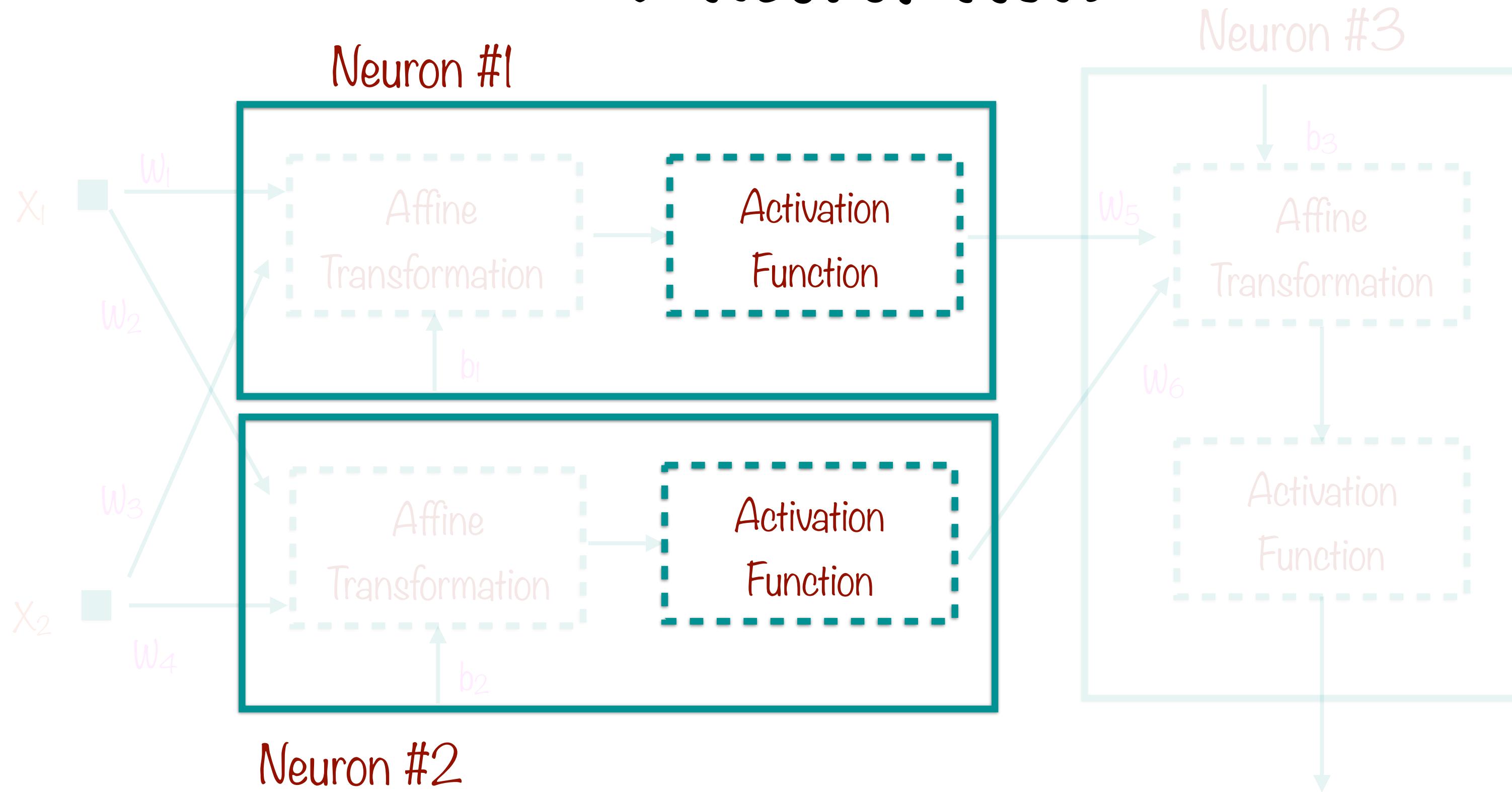
3-Neuron XOR



3-Neuron XOR



3-Neuron XOR



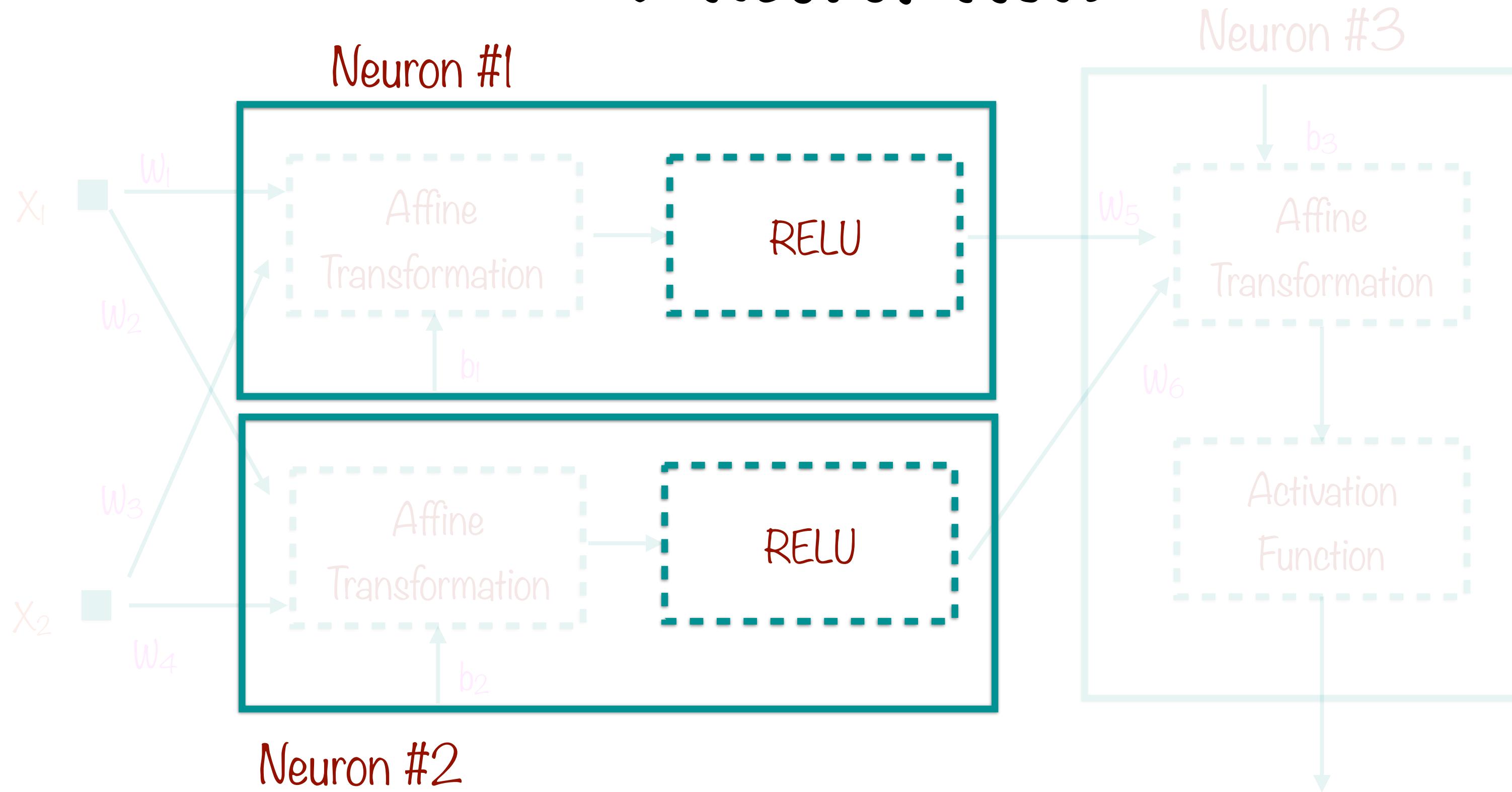


The most common form of the activation function is the ReLU

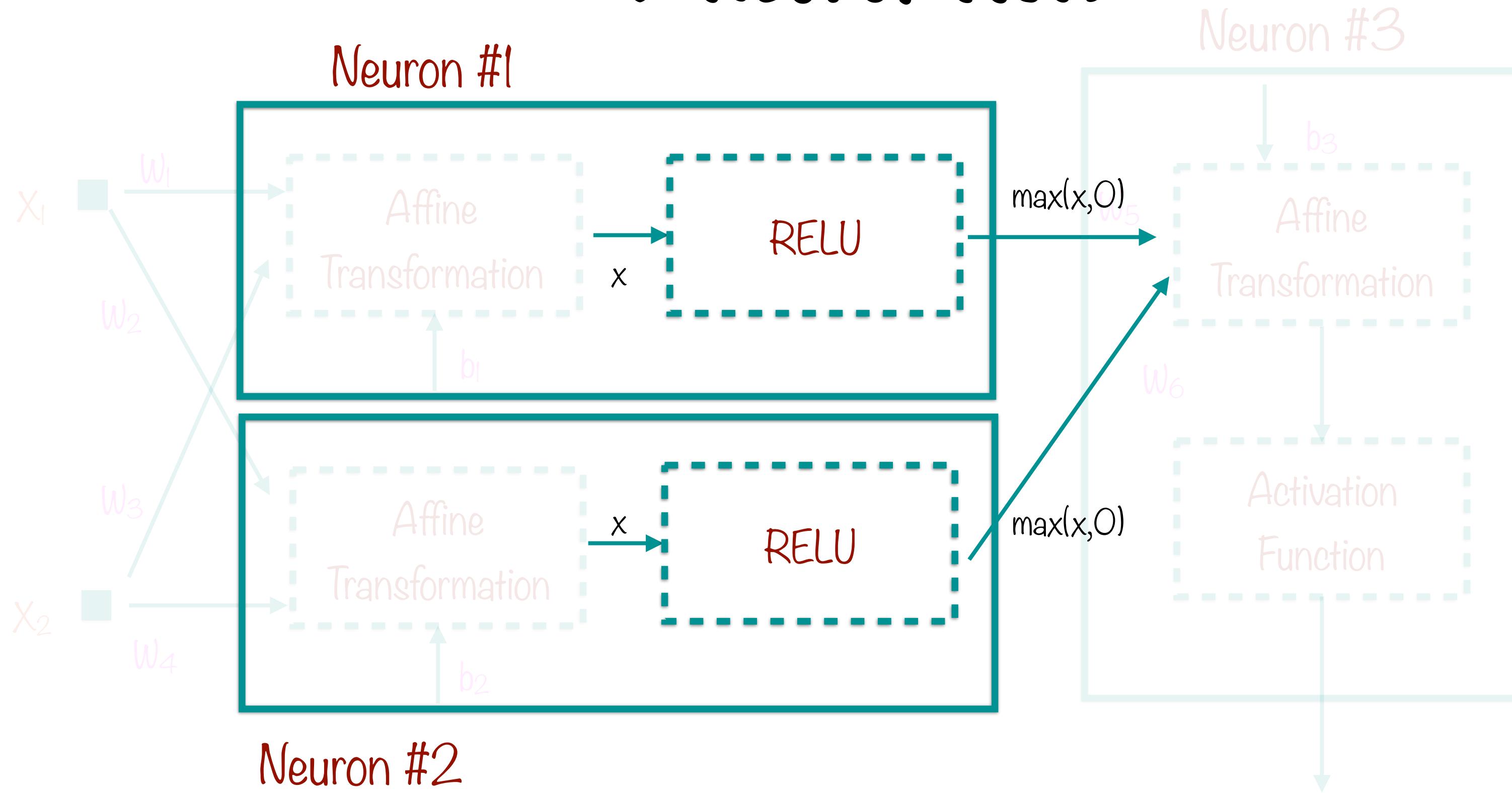
ReLU : Rectified Linear Unit

$$\text{ReLU}(x) = \max(0, x)$$

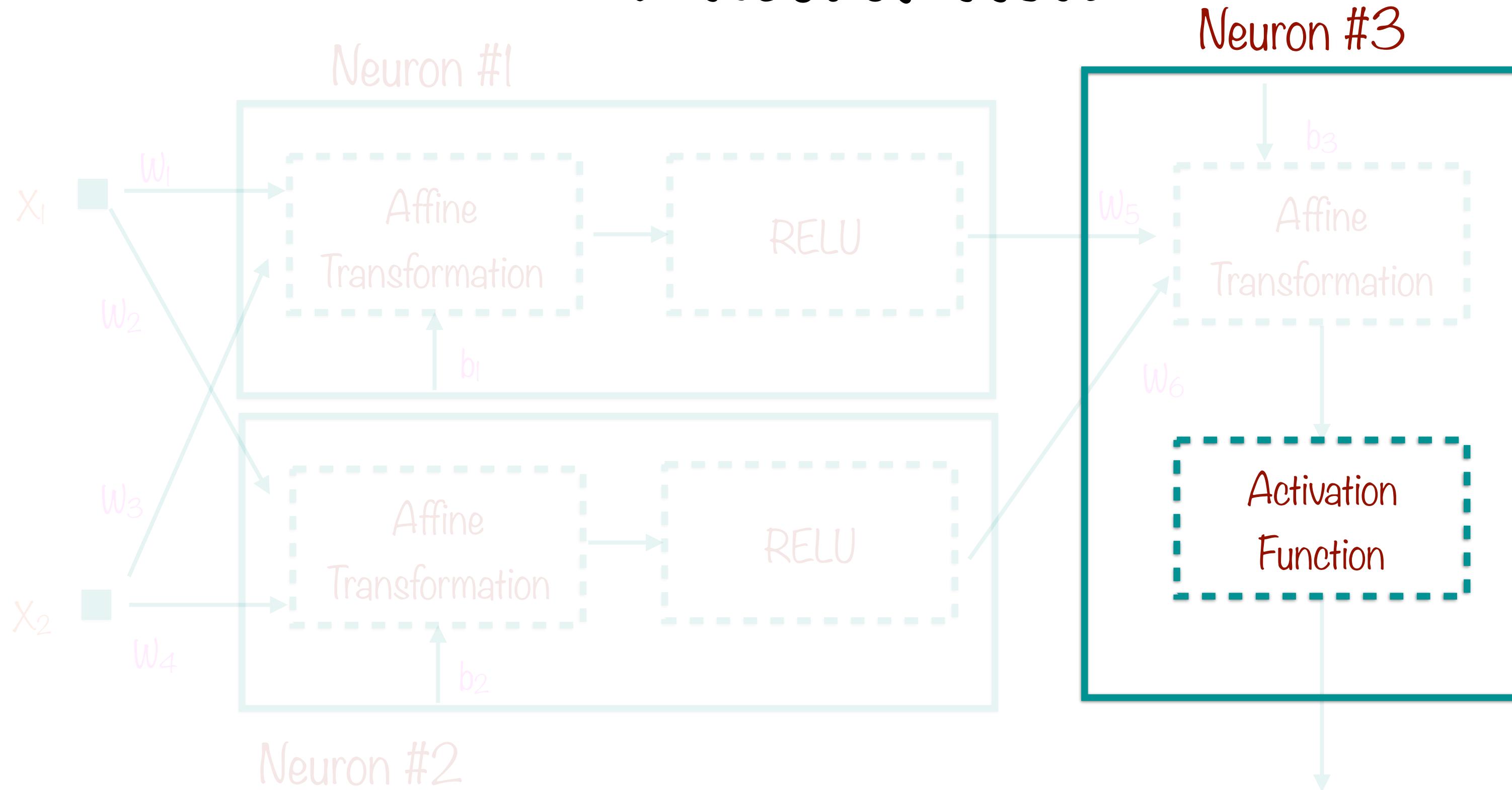
3-Neuron XOR



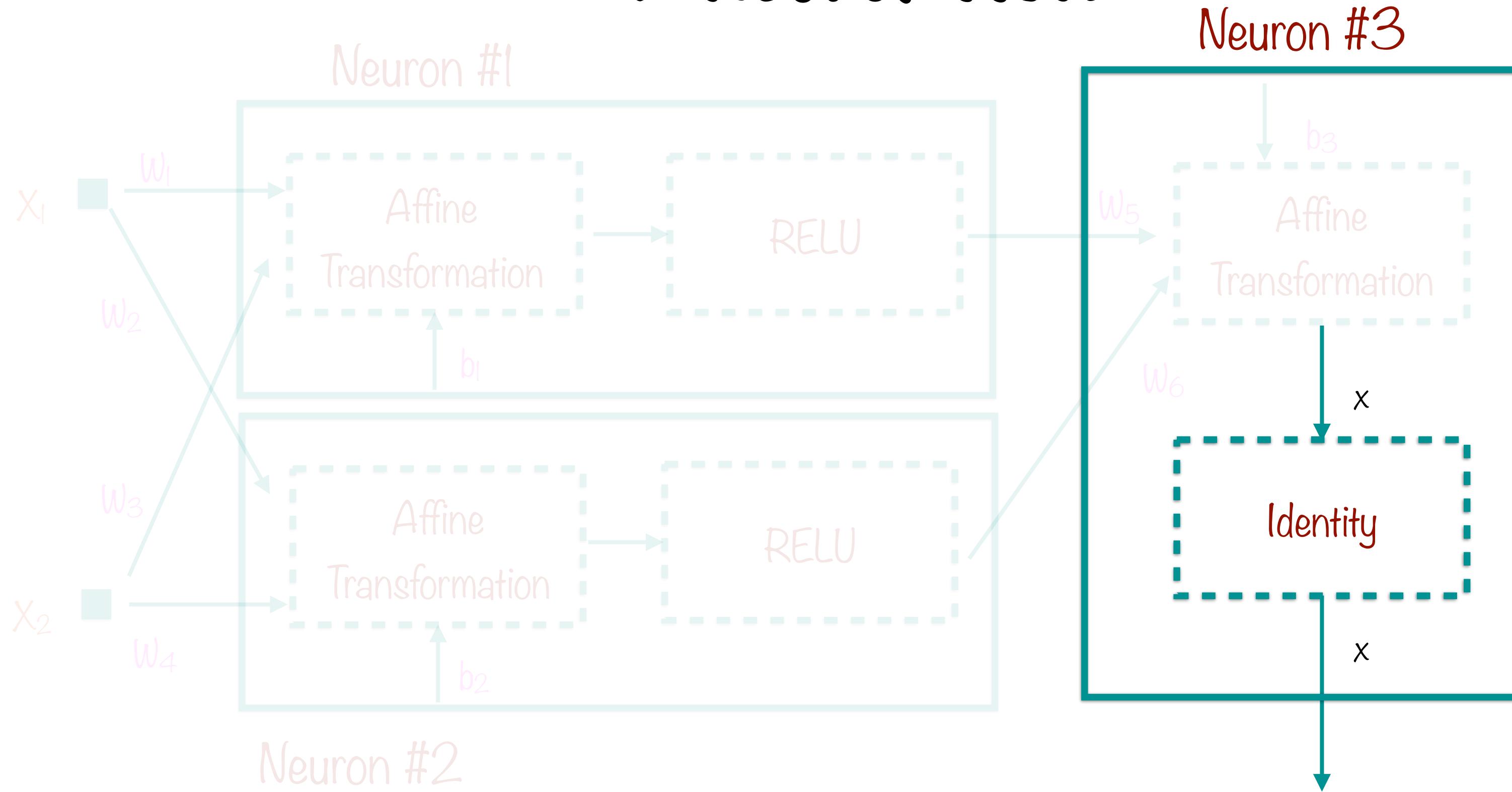
3-Neuron XOR



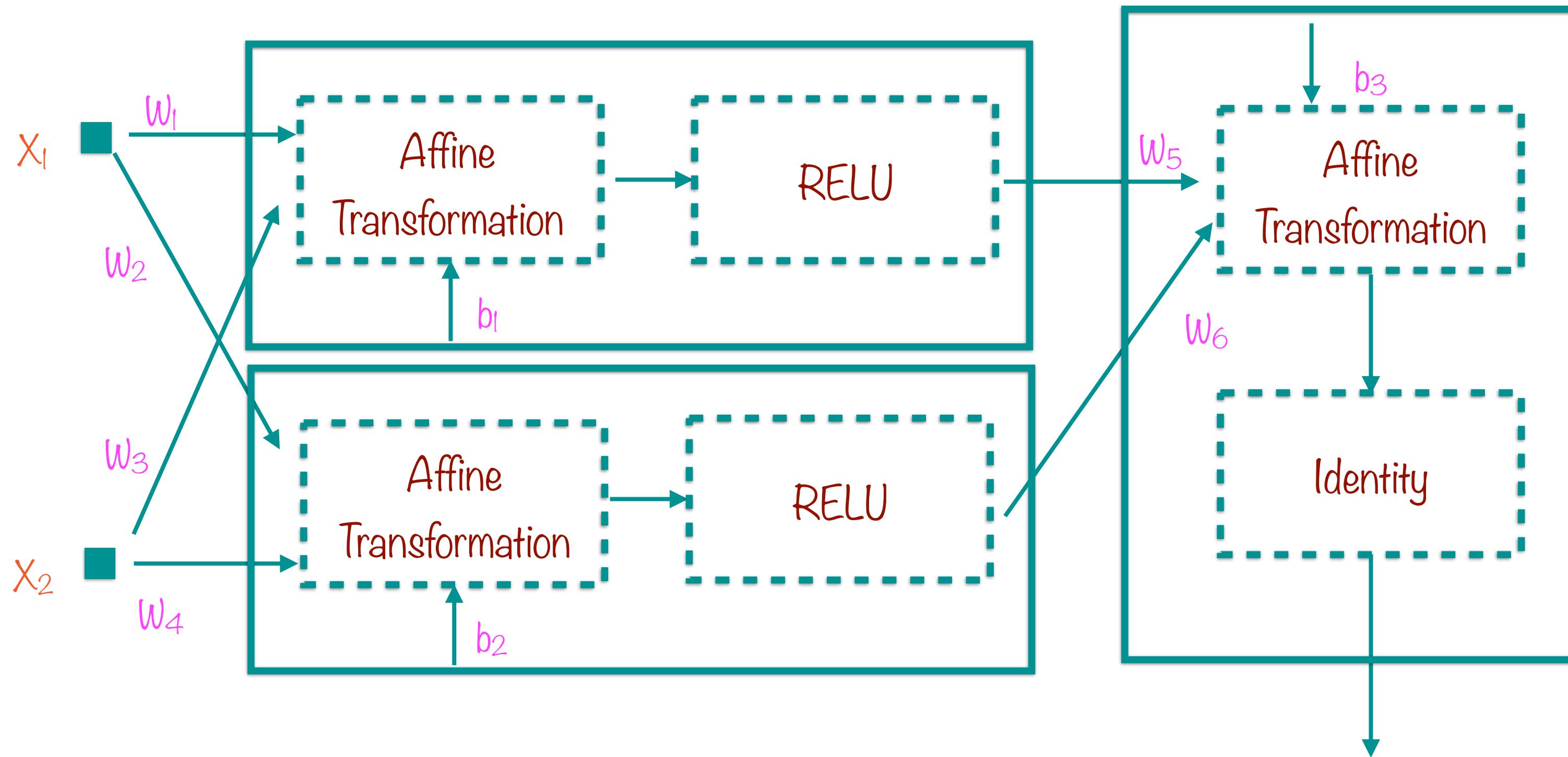
3-Neuron XOR



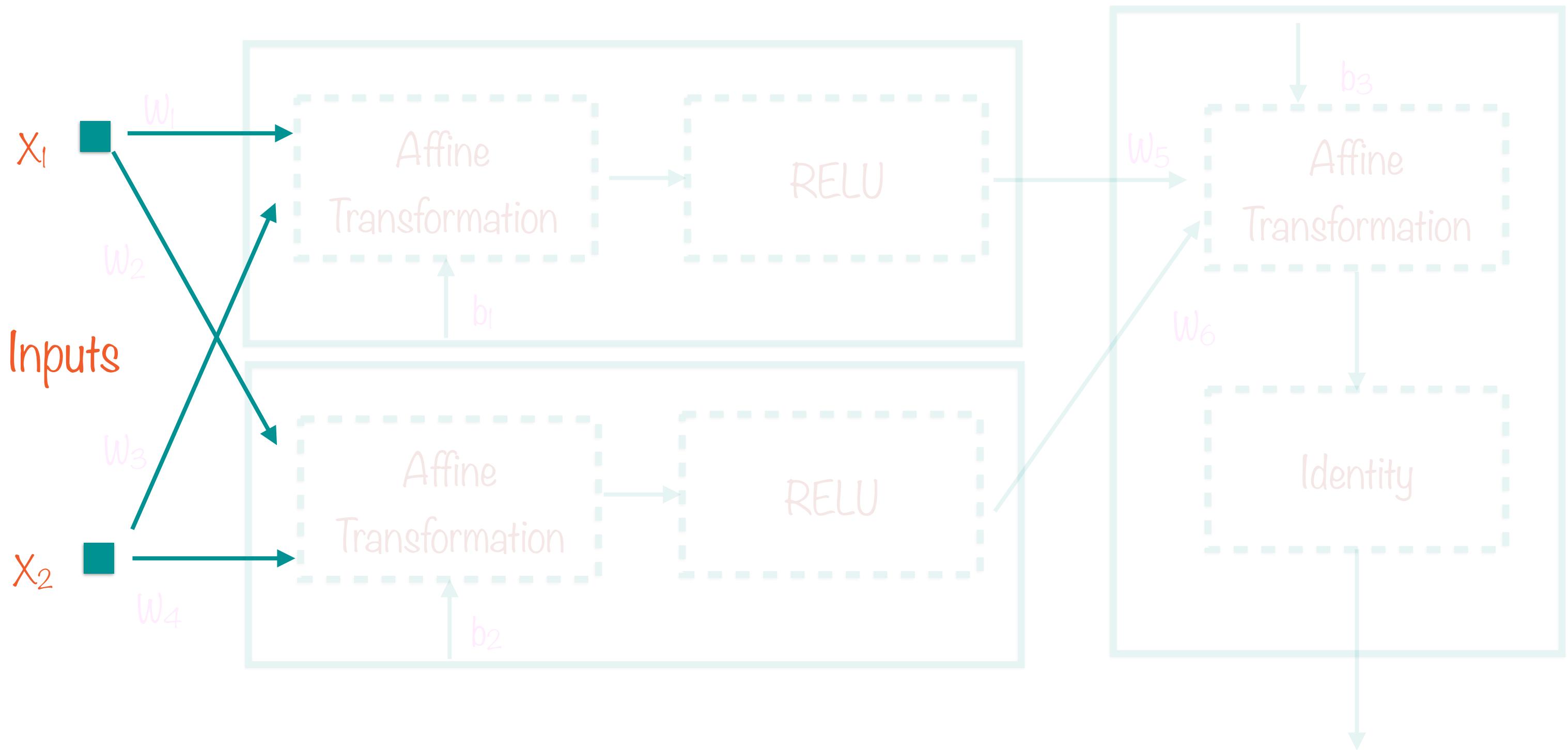
3-Neuron XOR



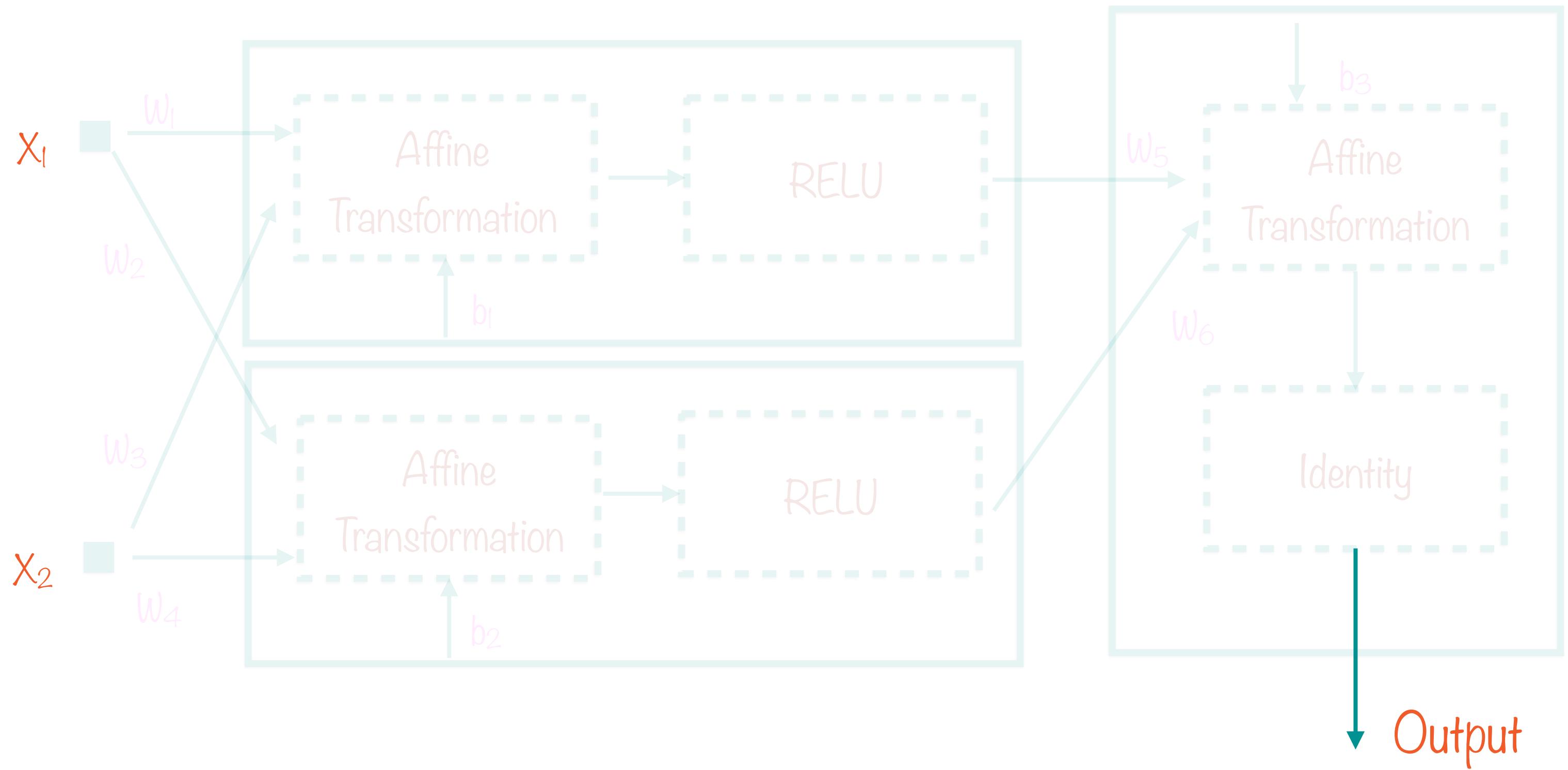
3-Neuron XOR



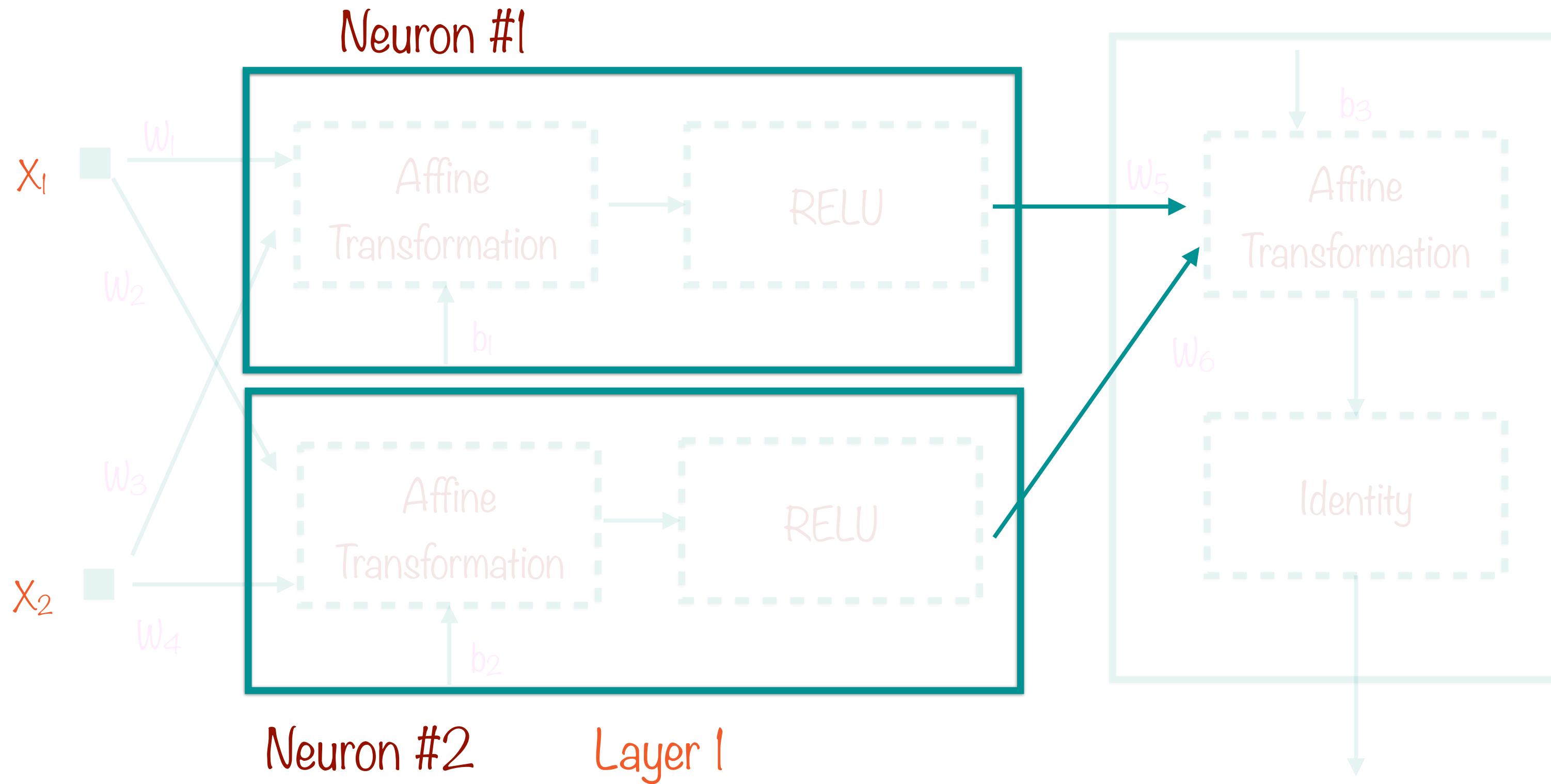
3-Neuron XOR



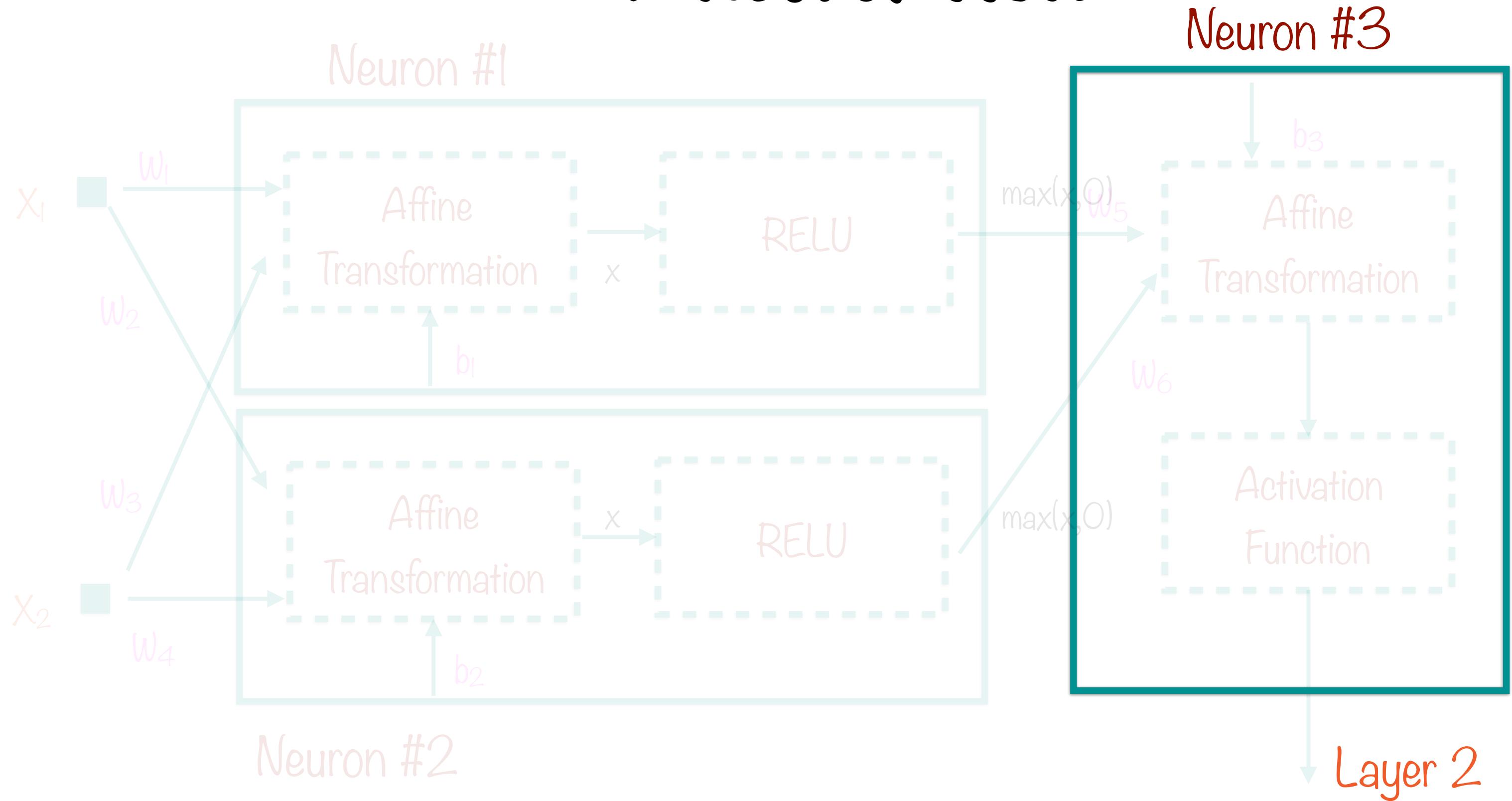
3-Neuron XOR



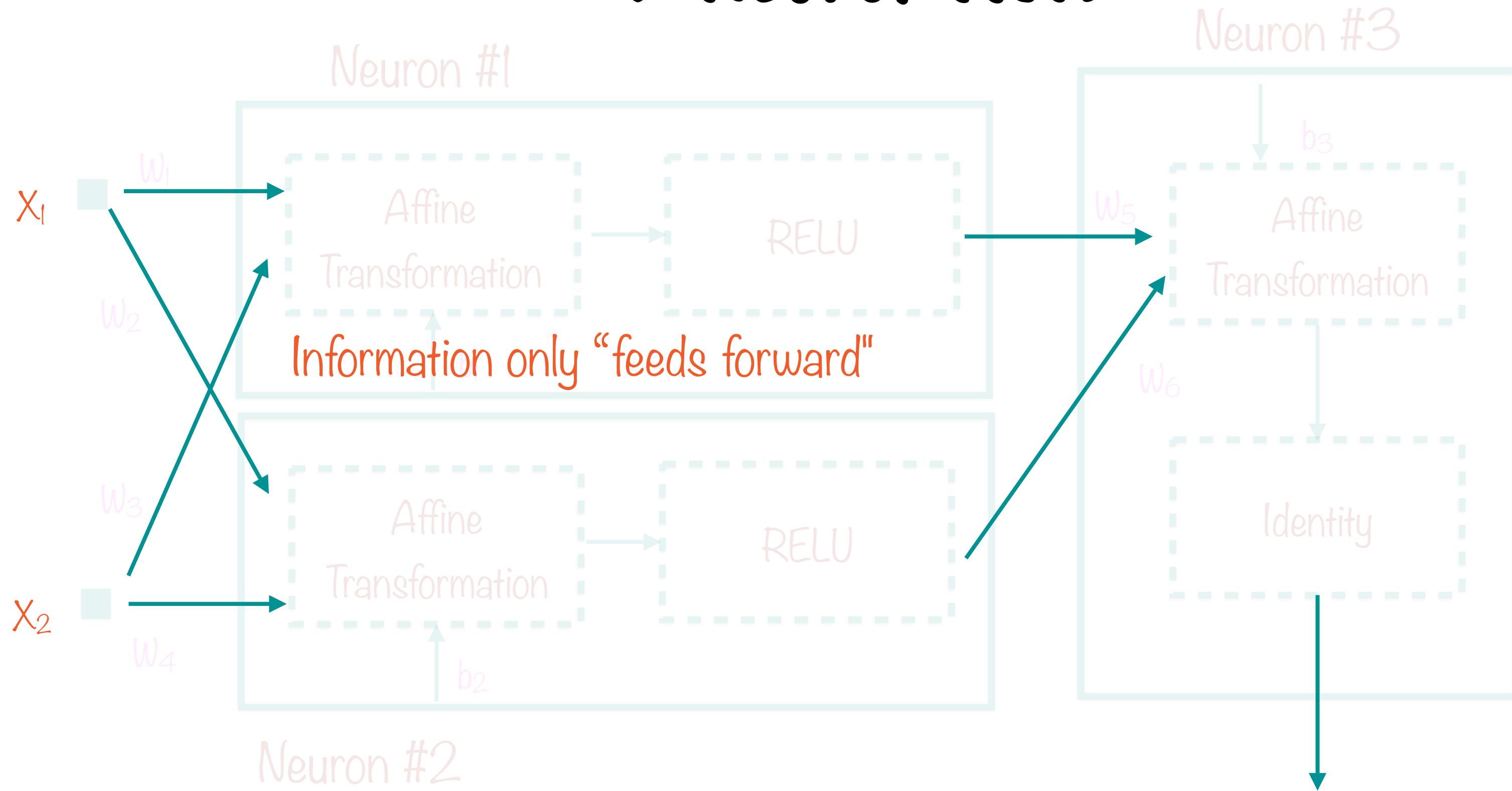
3-Neuron XOR



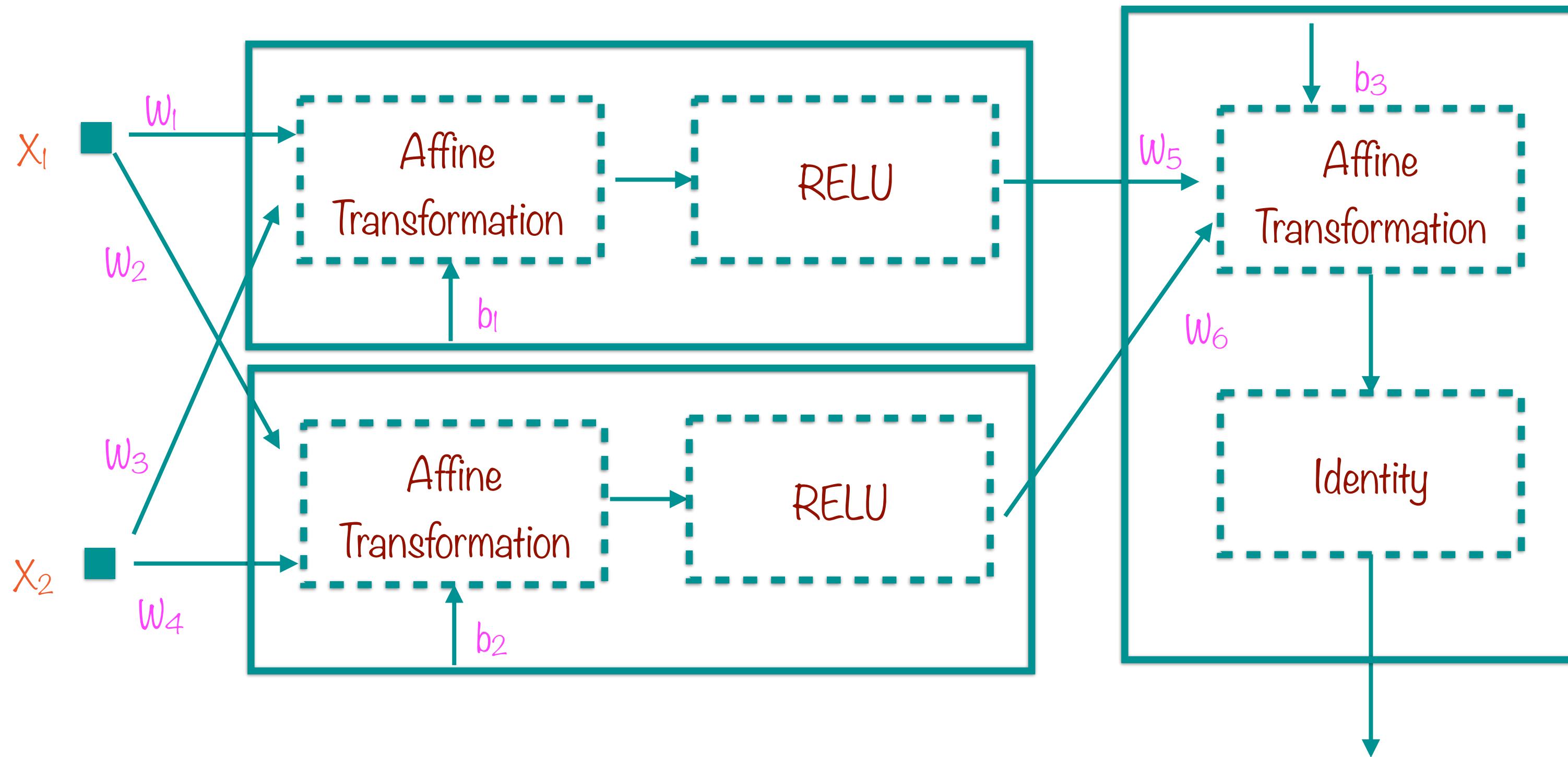
3-Neuron XOR



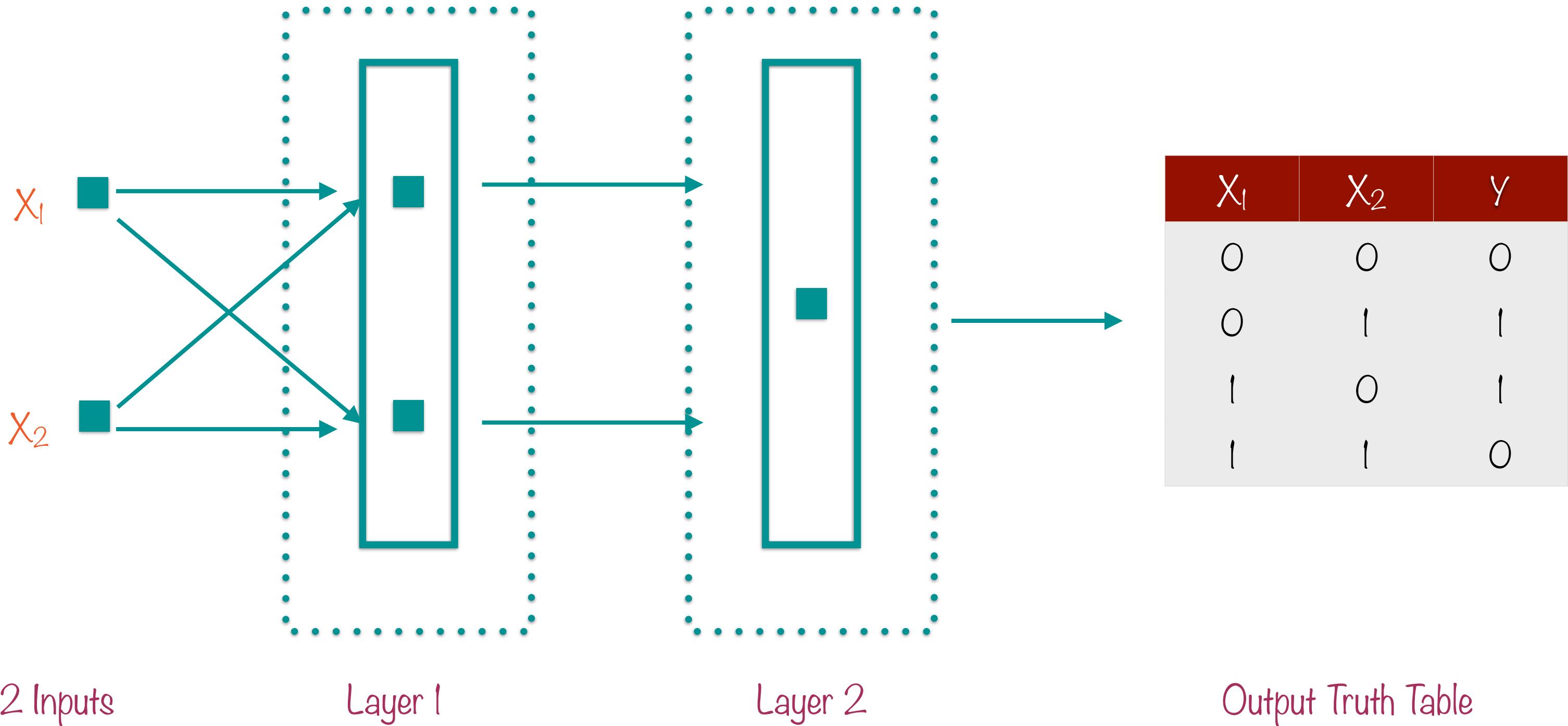
3-Neuron XOR



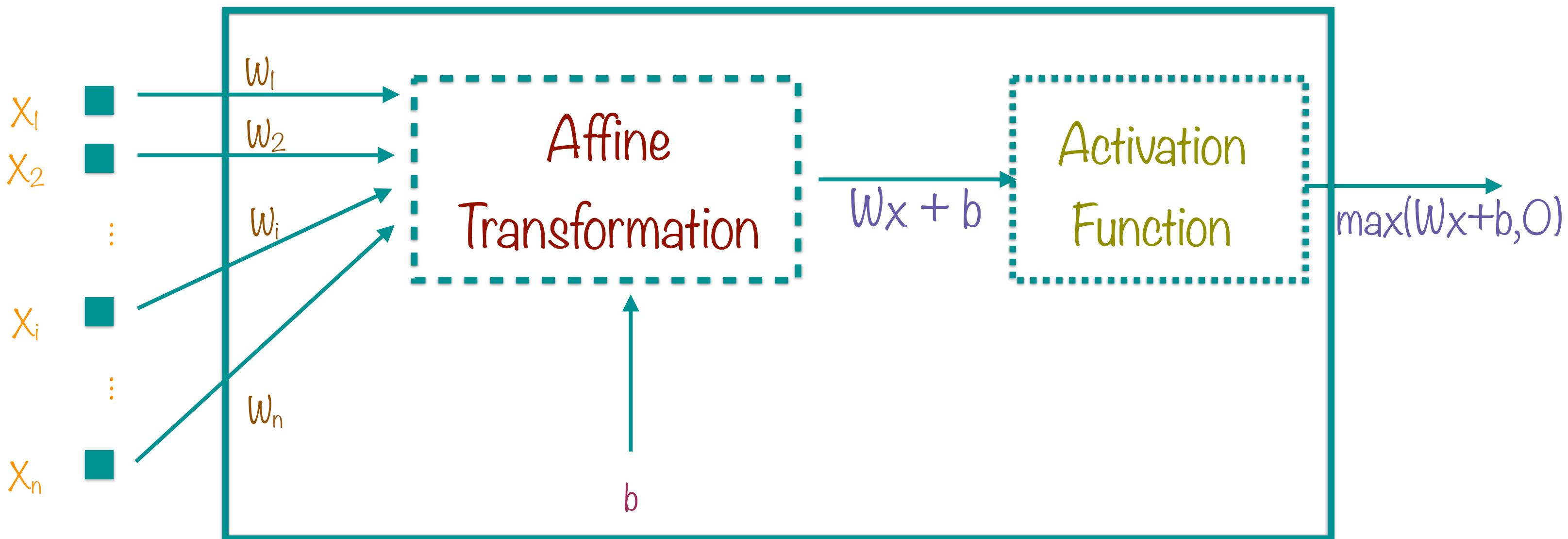
“2-Layer Feed-forward Neural Network”



XOR: 3 Neurons, 2 Layers

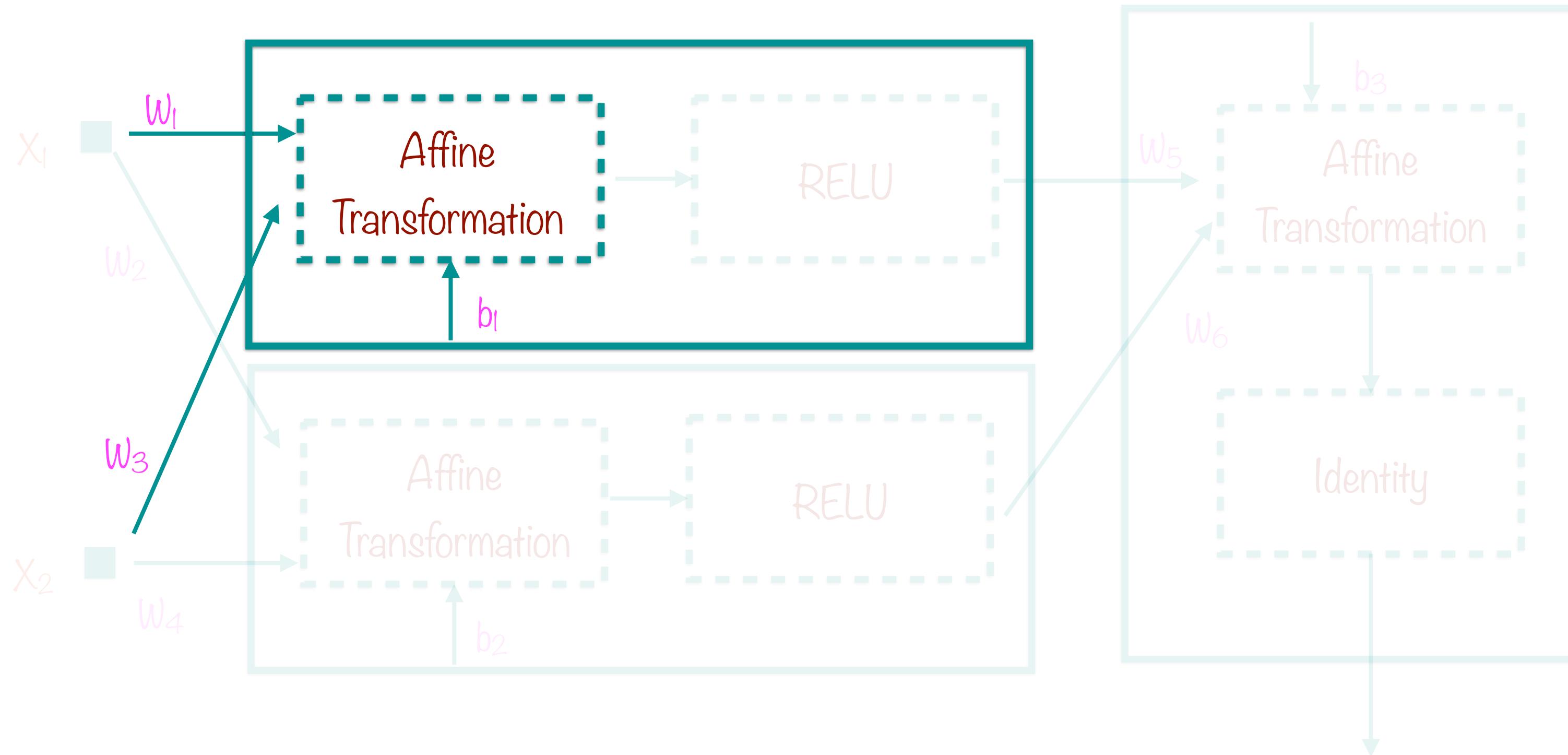


Operation of a Single Neuron

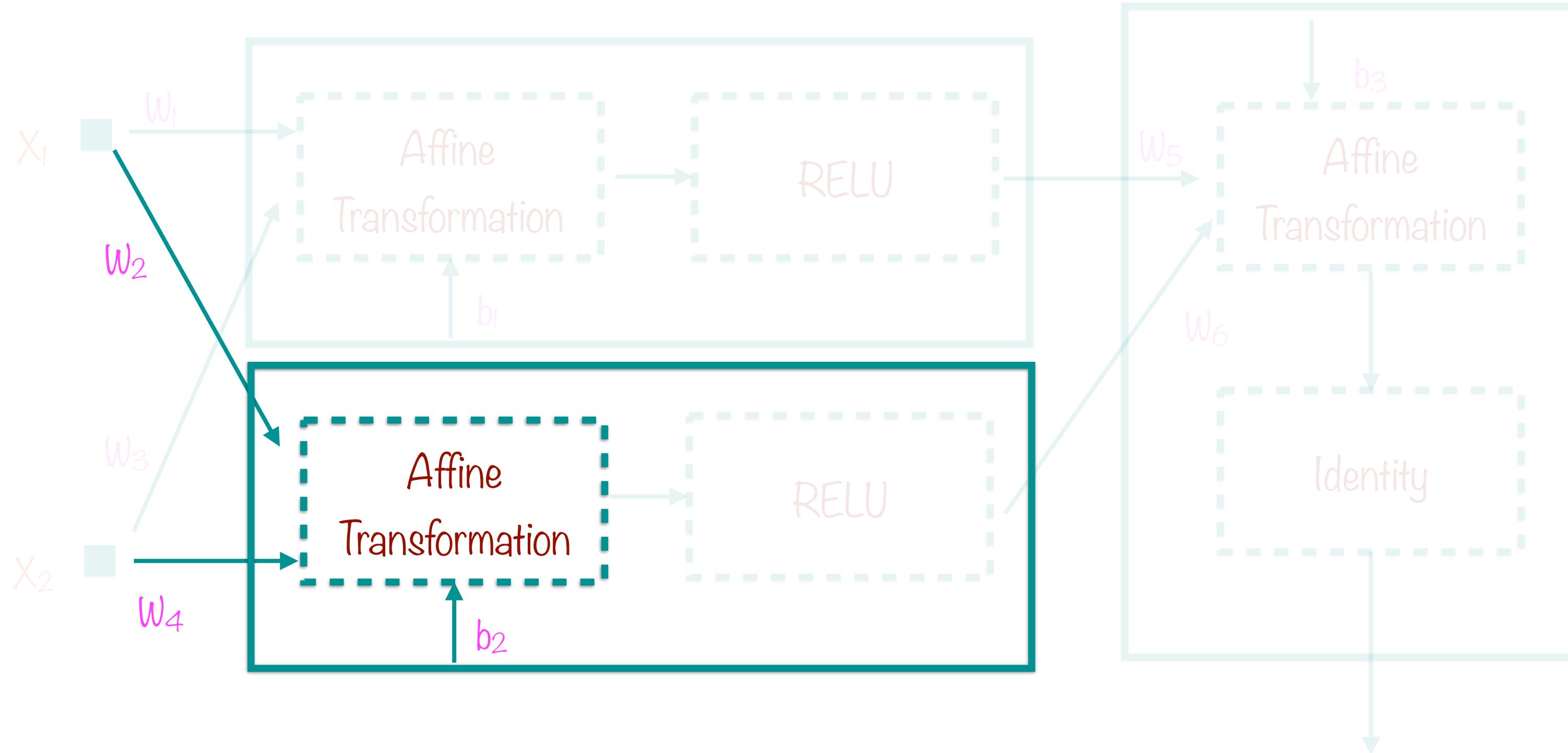


Each neuron has weights and a bias that must be calculated by the training algorithm
(done for us by TensorFlow)

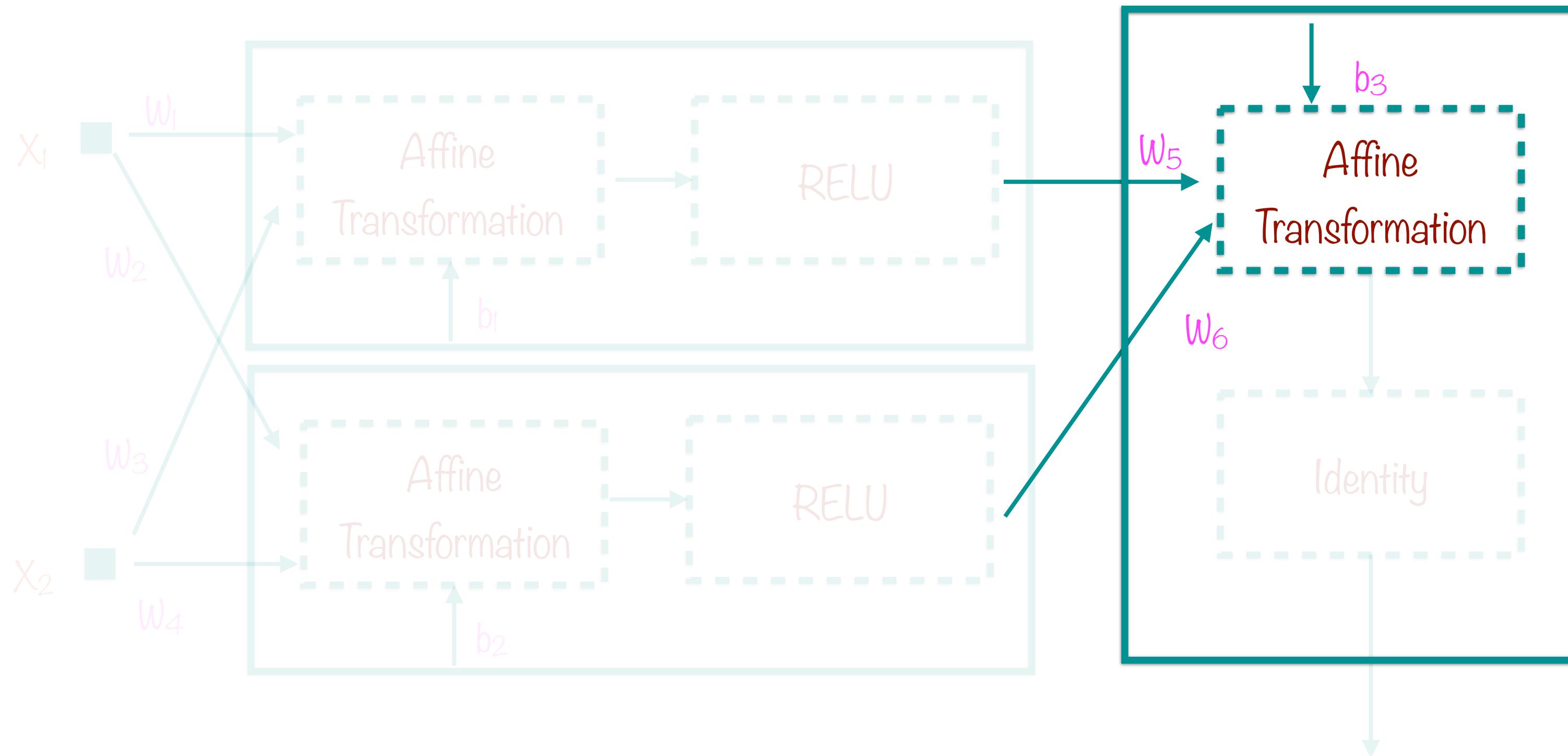
Weights and Bias of Neuron #1



Weights and Bias of Neuron #2

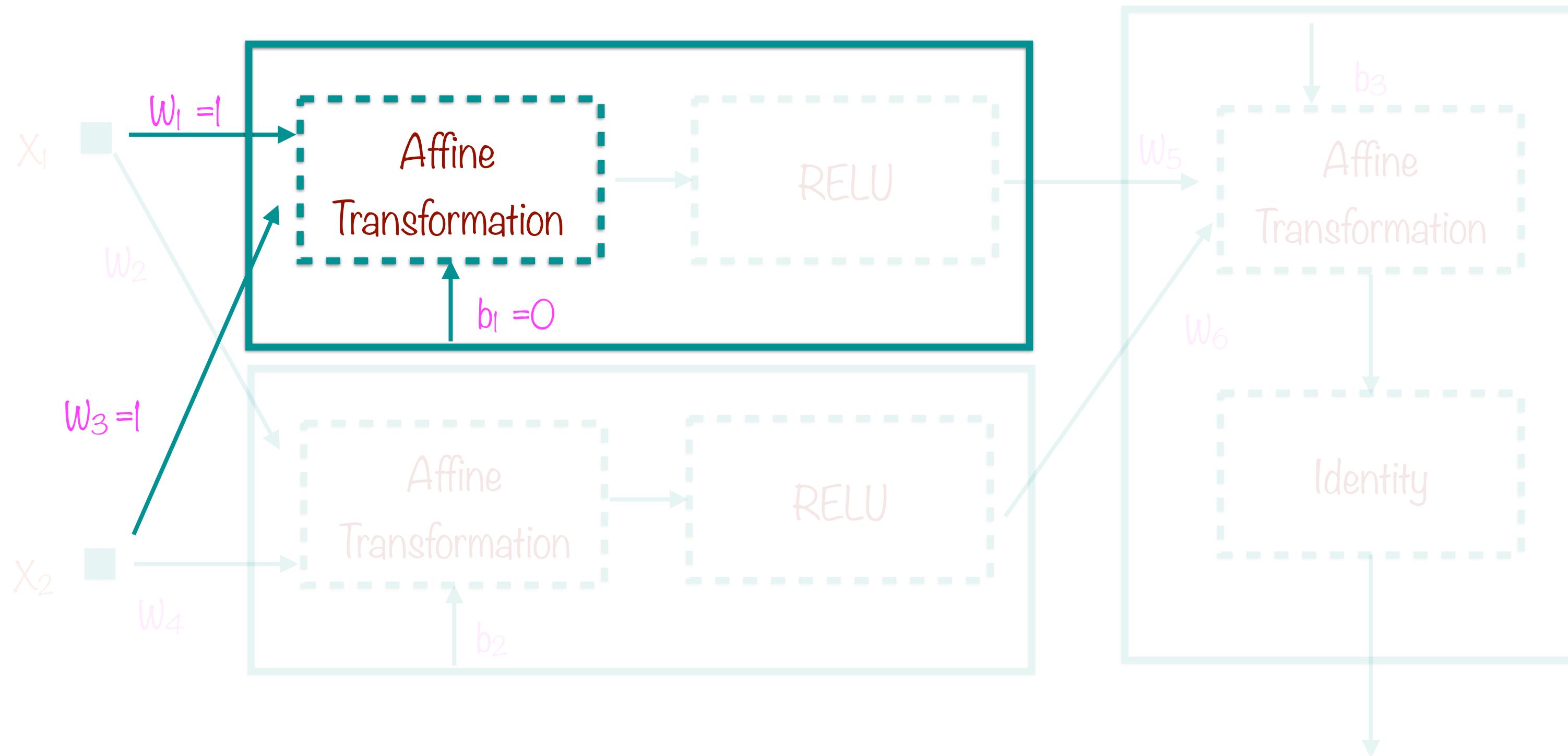


Weights and Bias of Neuron #3

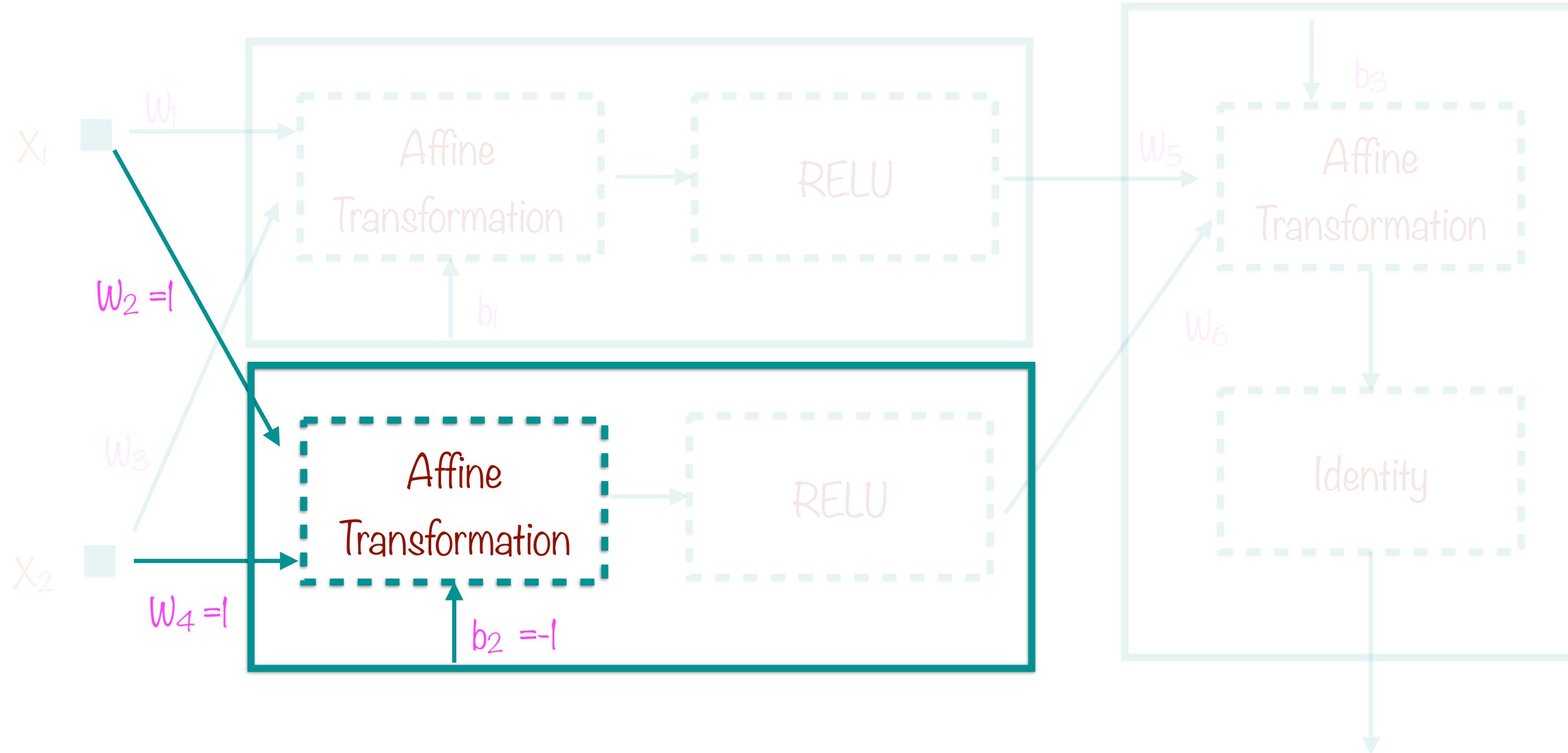


The weights and biases of individual neurons are determined during the training process

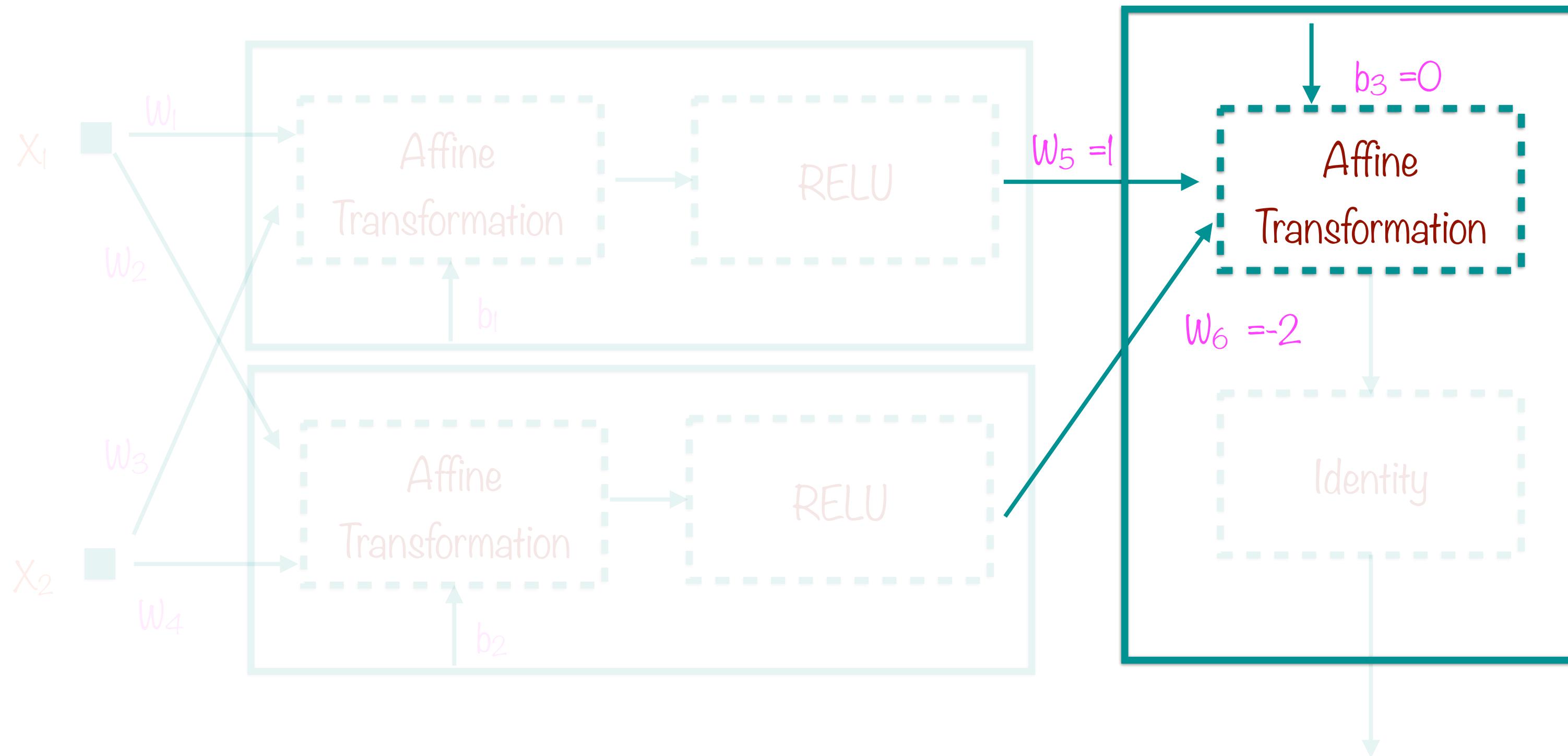
Weights and Bias of Neuron #1



Weights and Bias of Neuron #2



Weights and Bias of Neuron #3

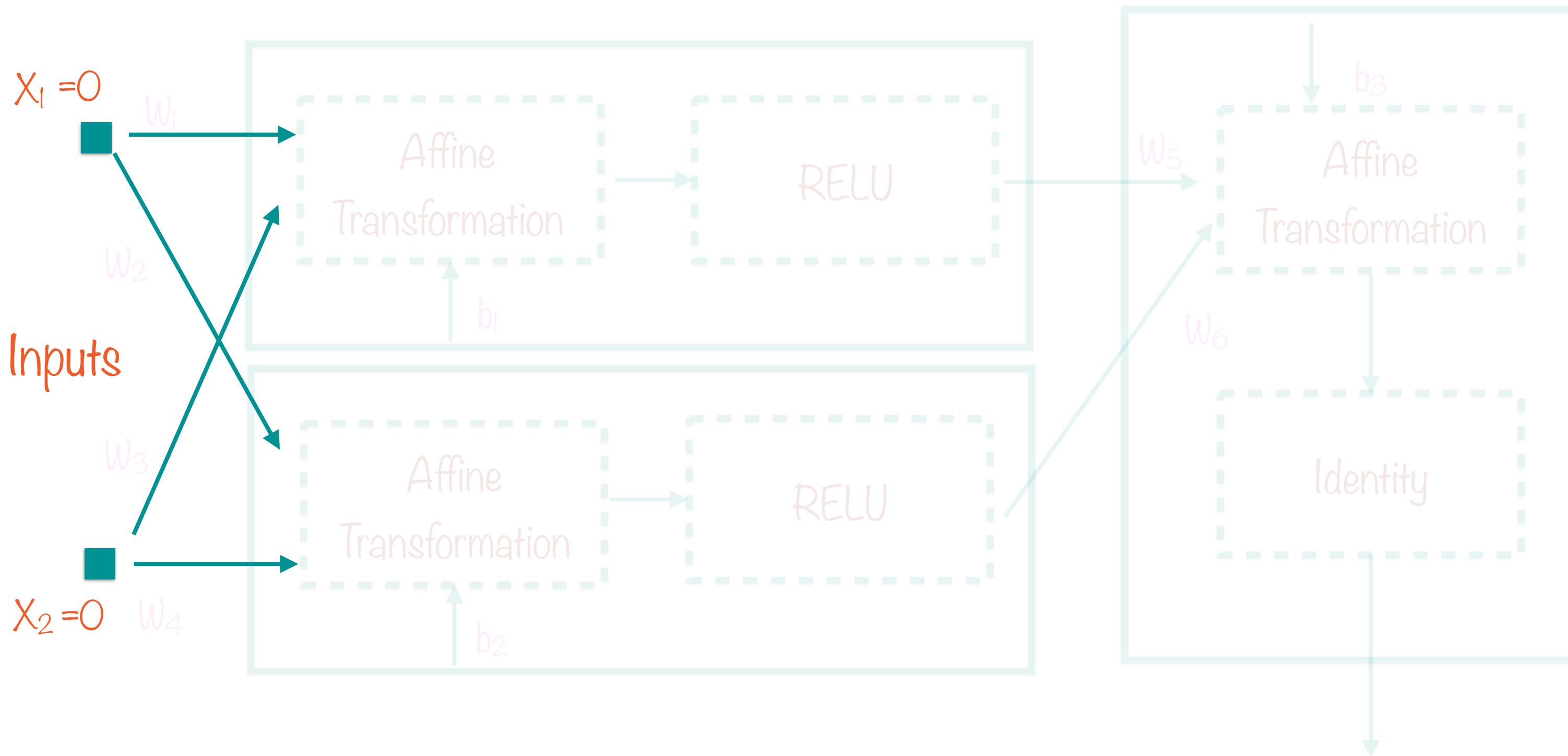


| X ₁ | X ₂ | Y |
|----------------|----------------|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

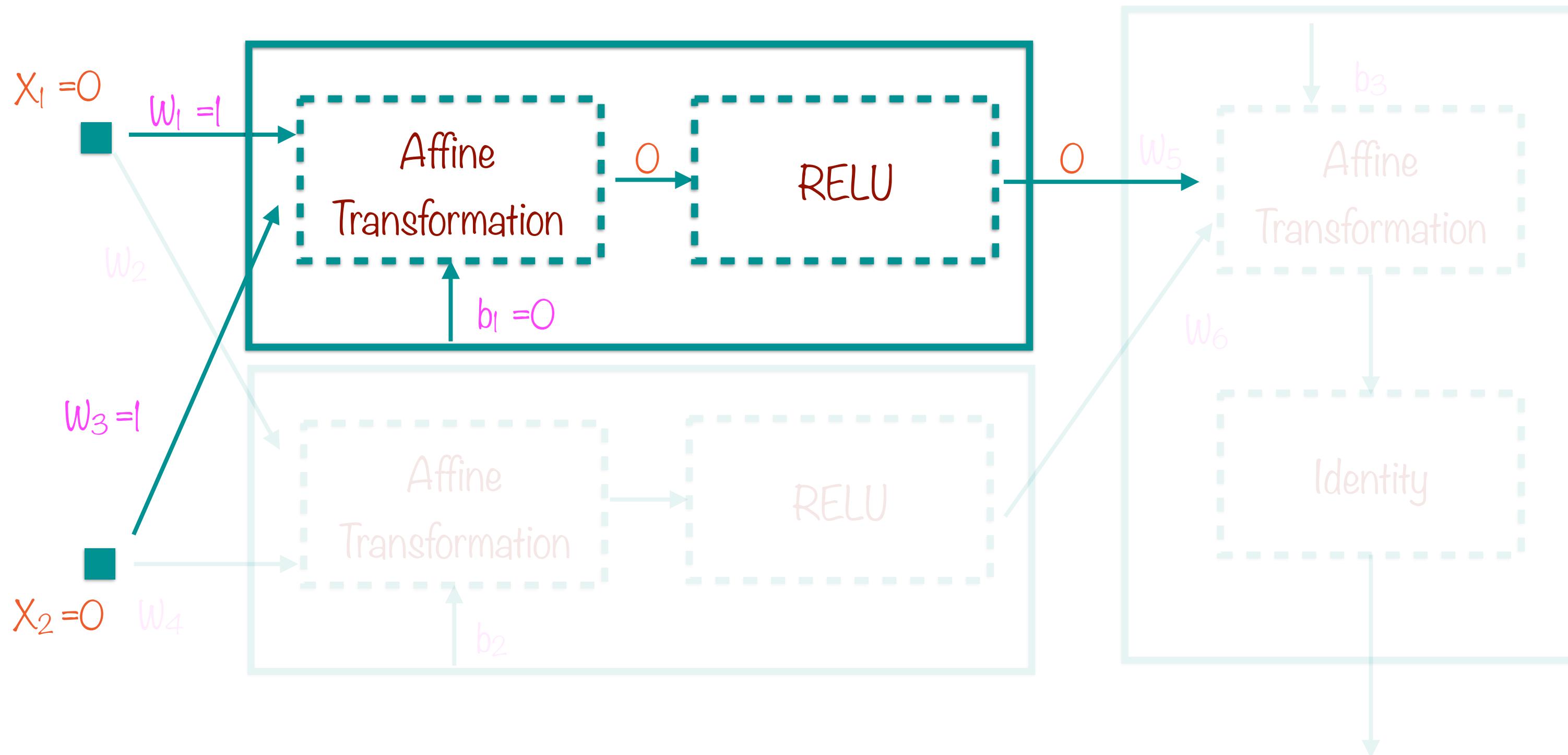
“Learning” XOR

Reverse-engineering XOR requires 3 neurons (arranged in 2 layers) as well as a non-linear activation function

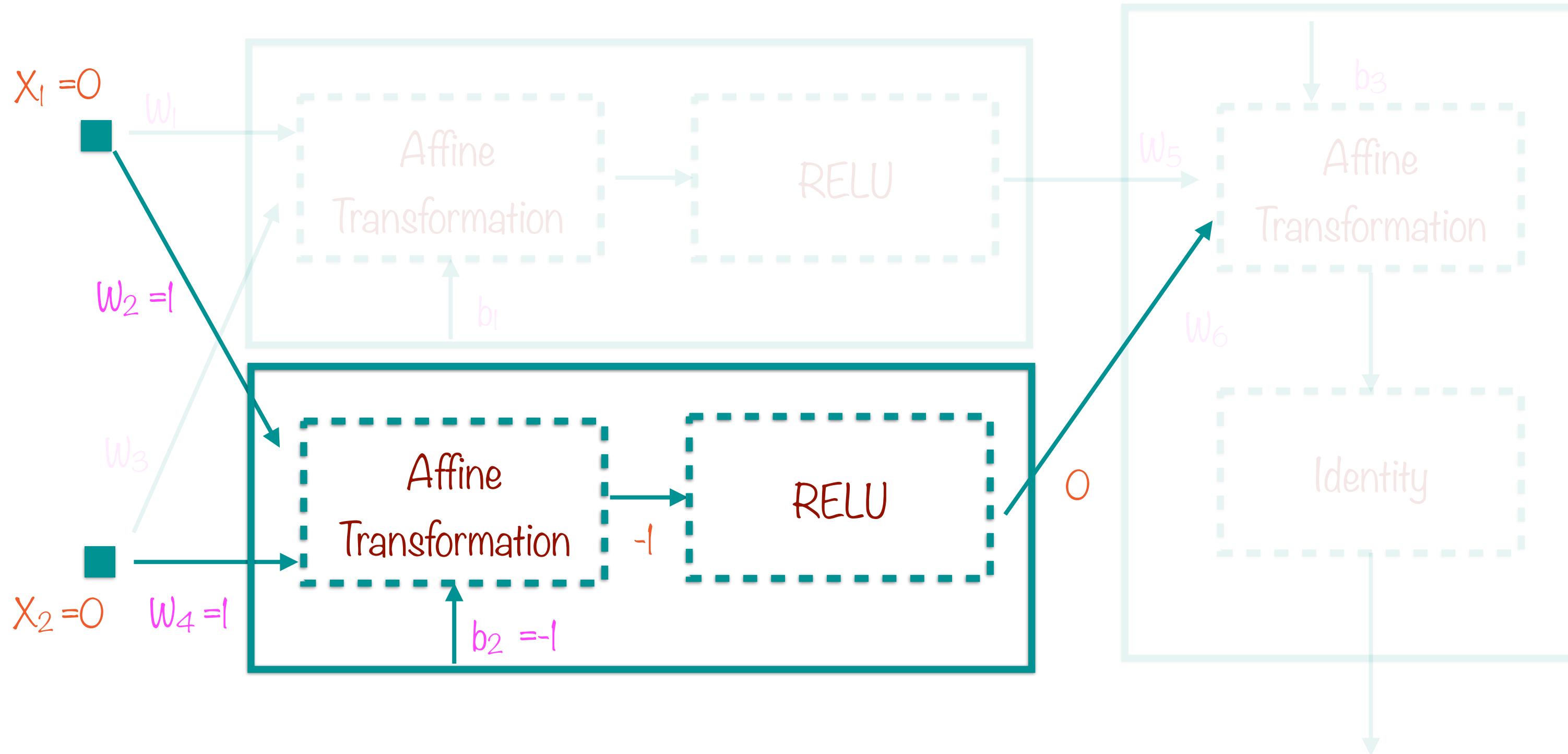
3-Neuron XOR



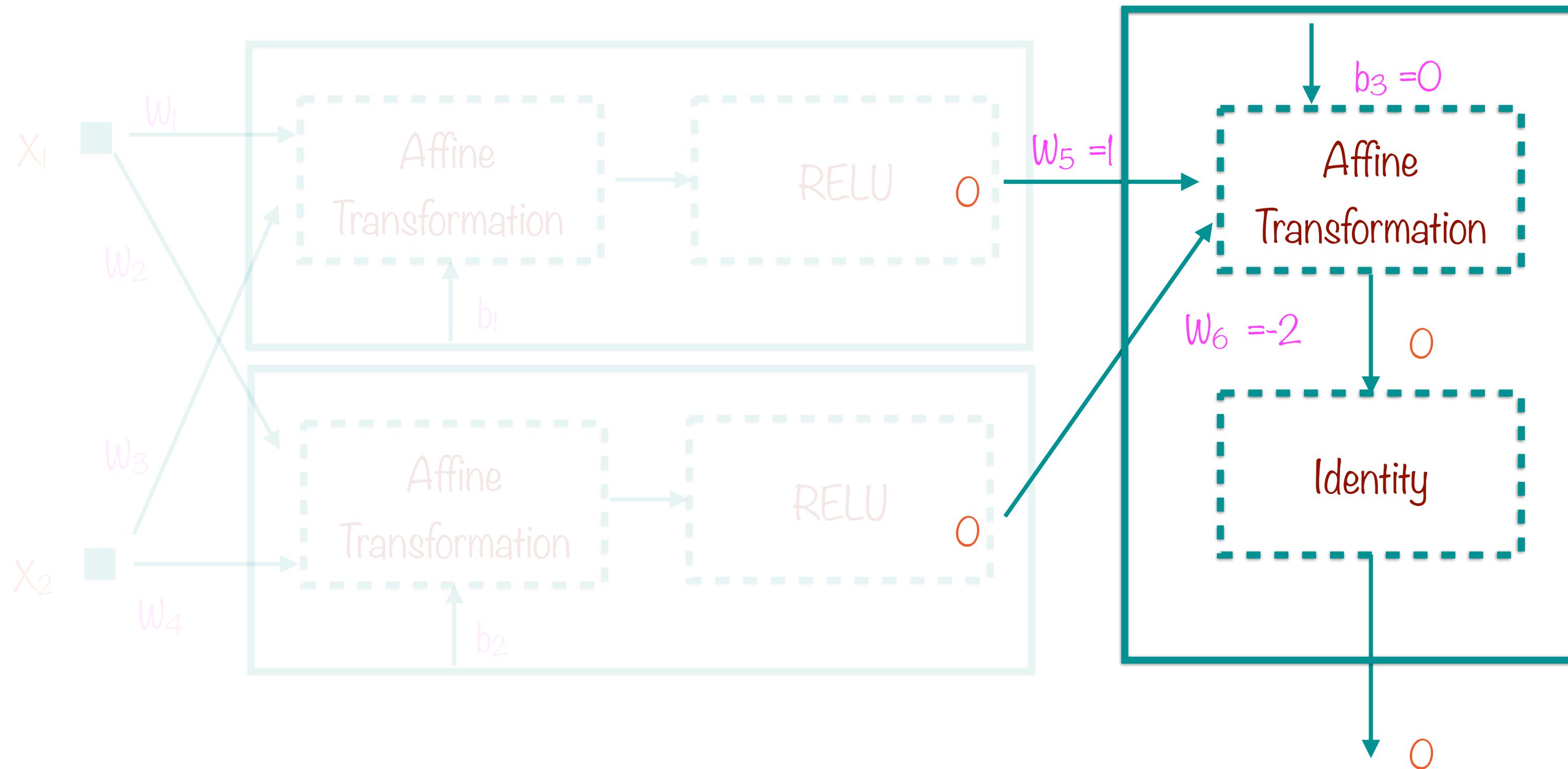
Weights and Bias of Neuron #1



Weights and Bias of Neuron #2



Weights and Bias of Neuron #3

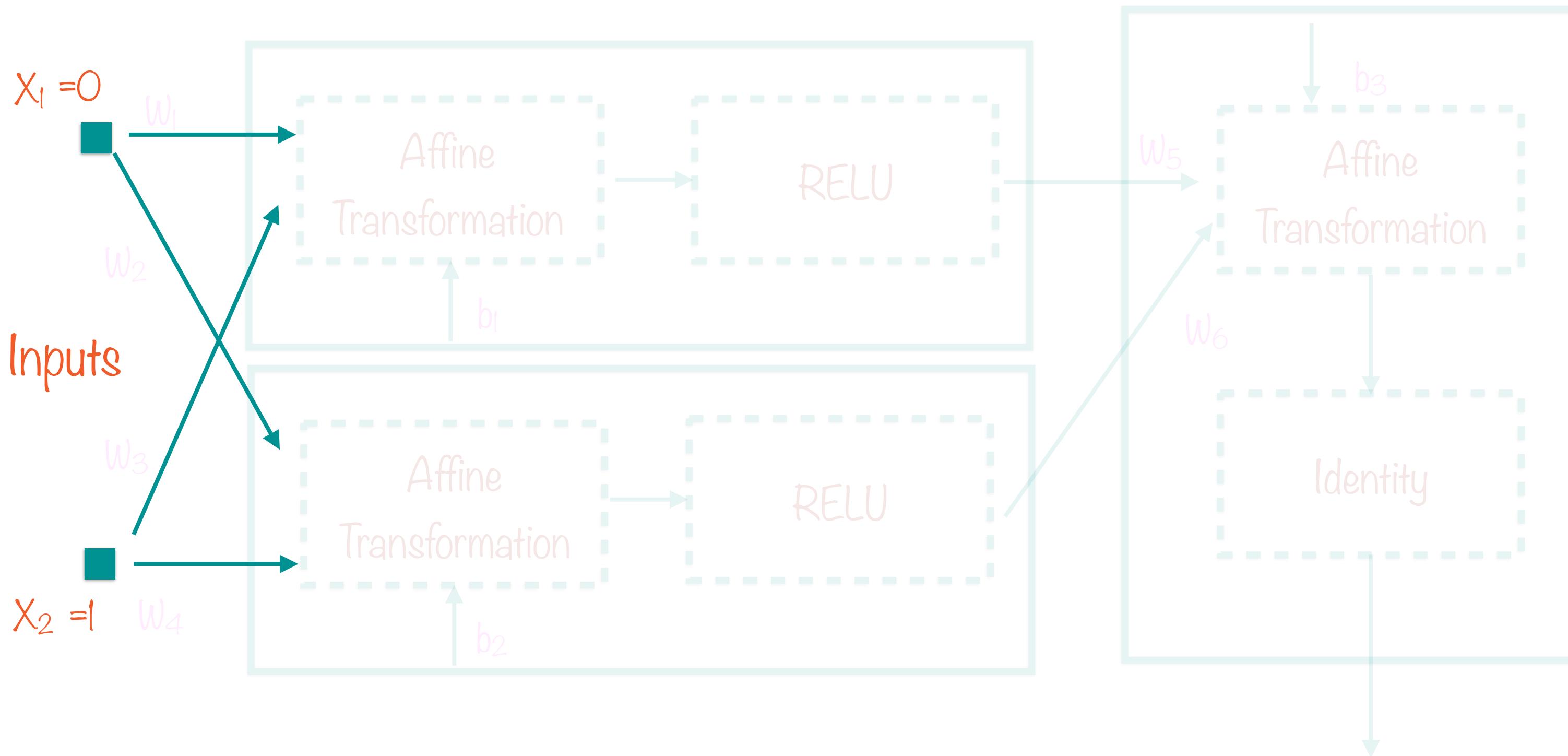


| X_1 | X_2 | Y |
|-------|-------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

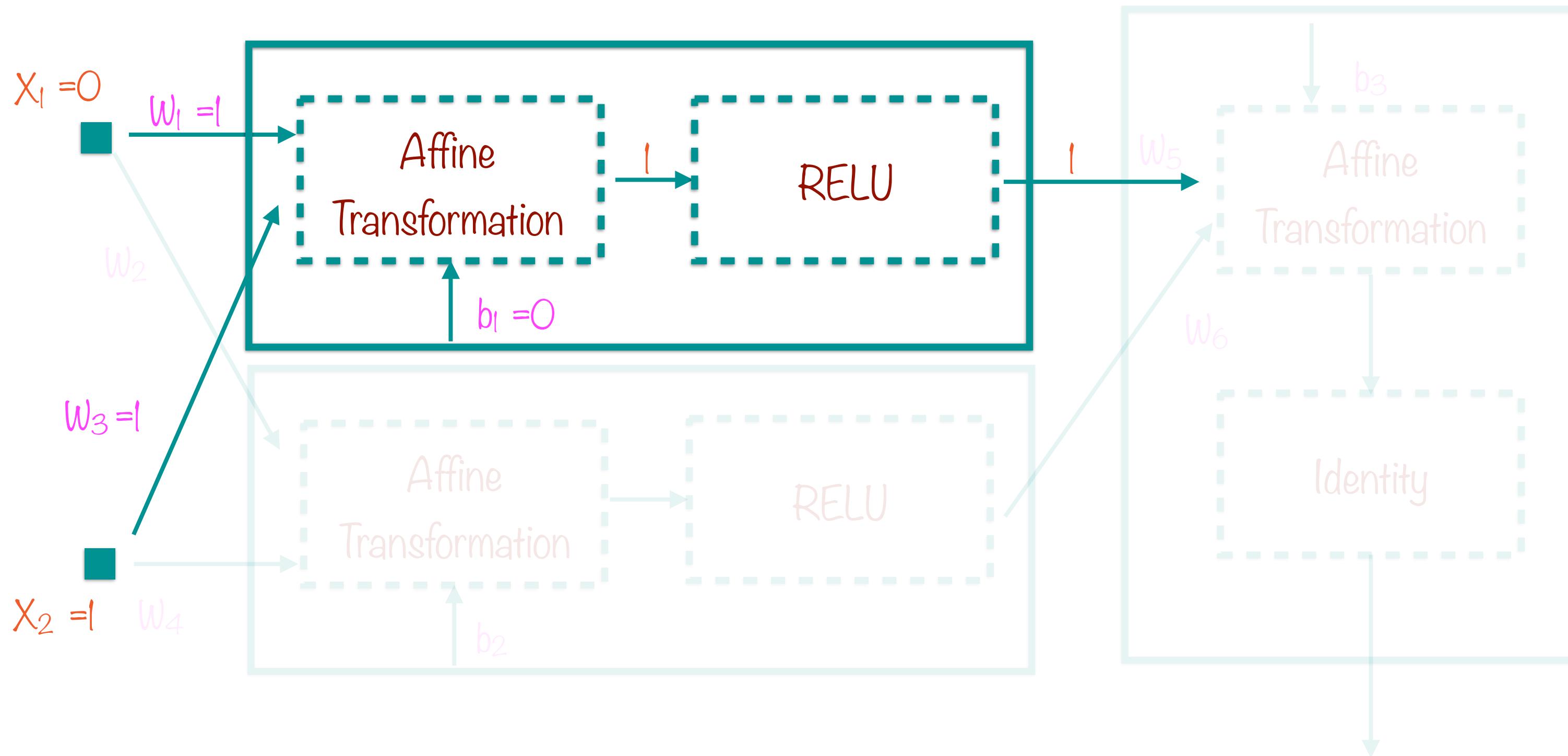
“Learning” XOR

Reverse-engineering XOR requires 3 neurons (arranged in 2 layers) as well as a non-linear activation function

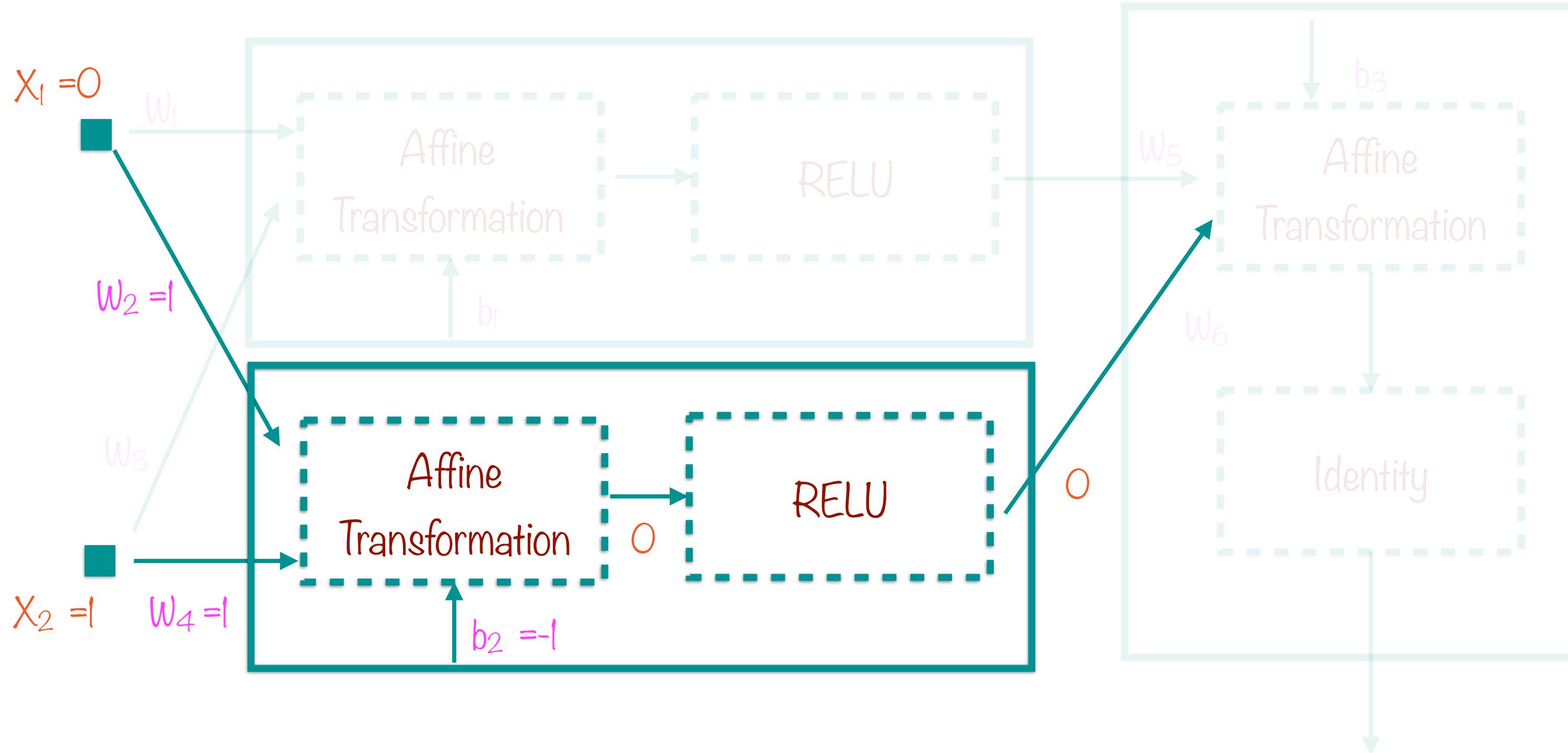
3-Neuron XOR



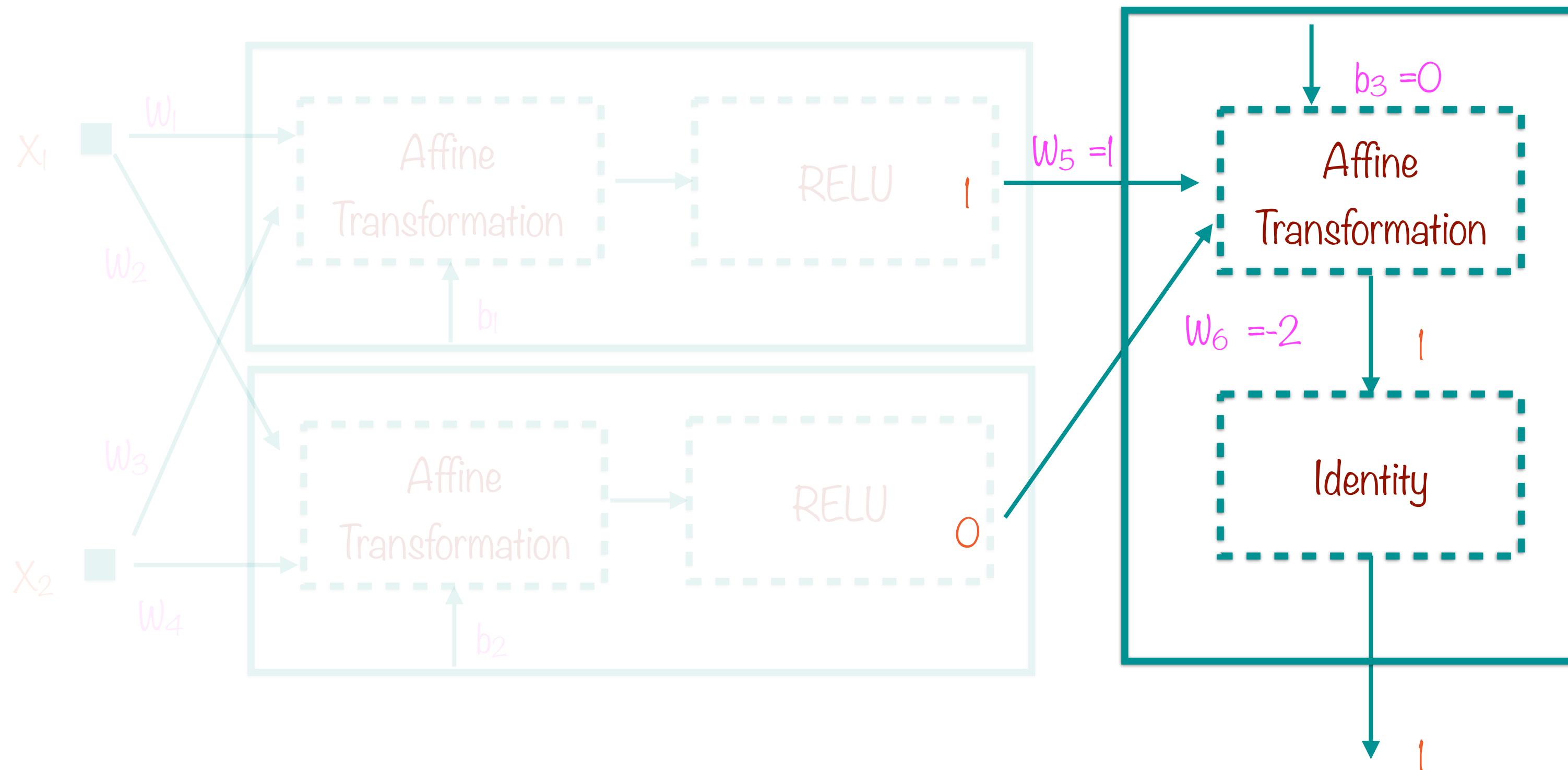
Weights and Bias of Neuron #1



Weights and Bias of Neuron #2



Weights and Bias of Neuron #3

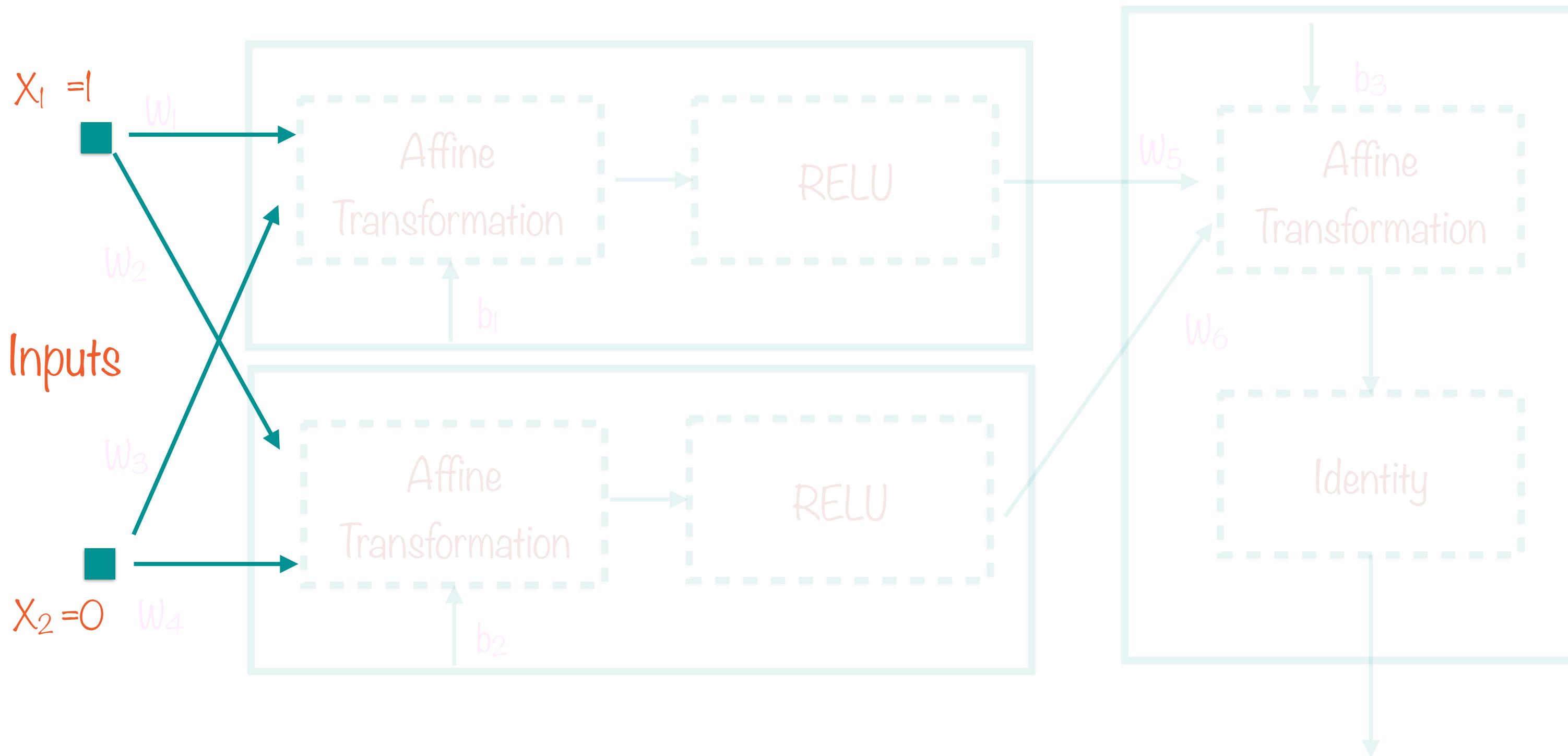


| X_1 | X_2 | Y |
|-------|-------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

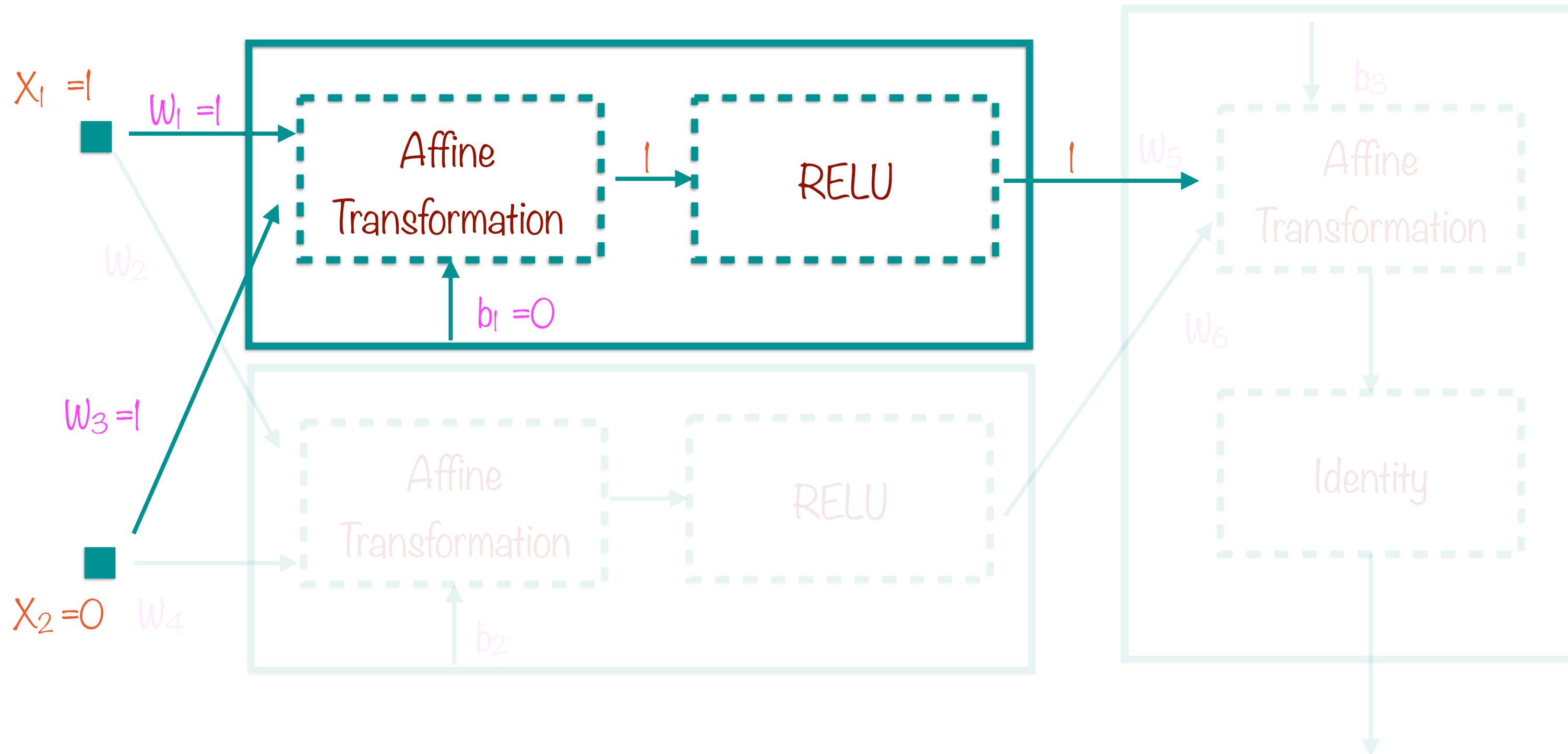
“Learning” XOR

Reverse-engineering XOR requires 3 neurons (arranged in 2 layers) as well as a non-linear activation function

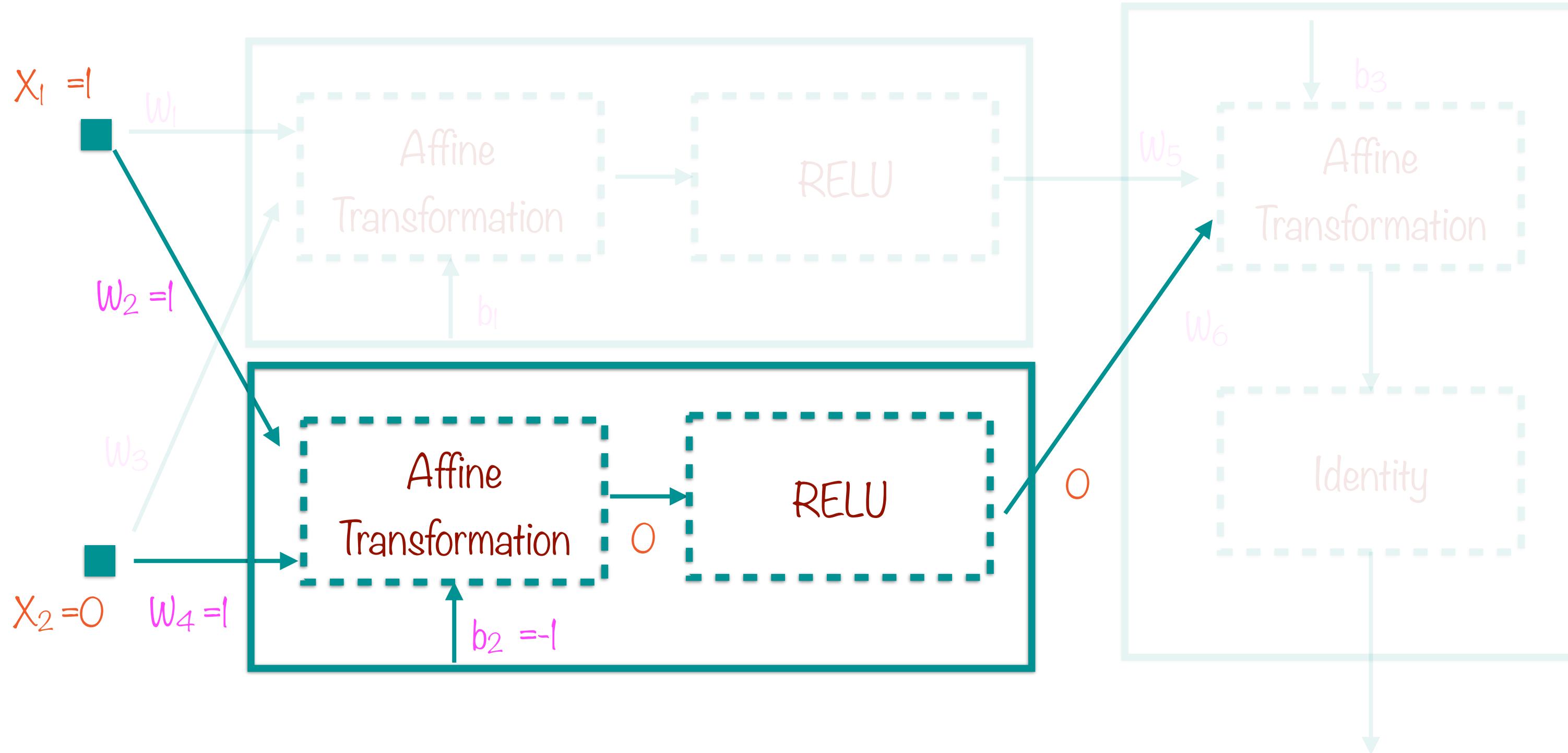
3-Neuron XOR



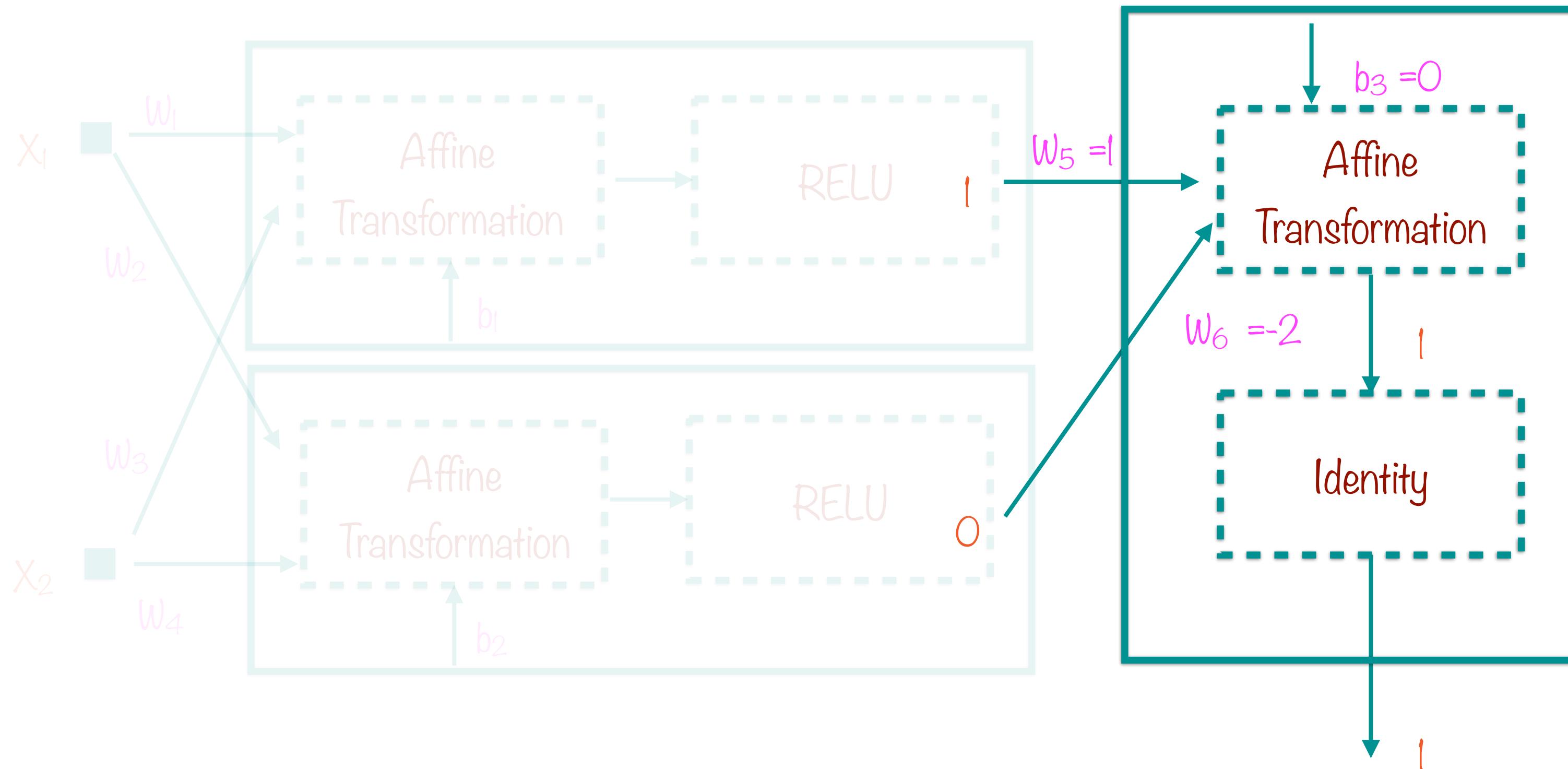
Weights and Bias of Neuron #1



Weights and Bias of Neuron #2



Weights and Bias of Neuron #3

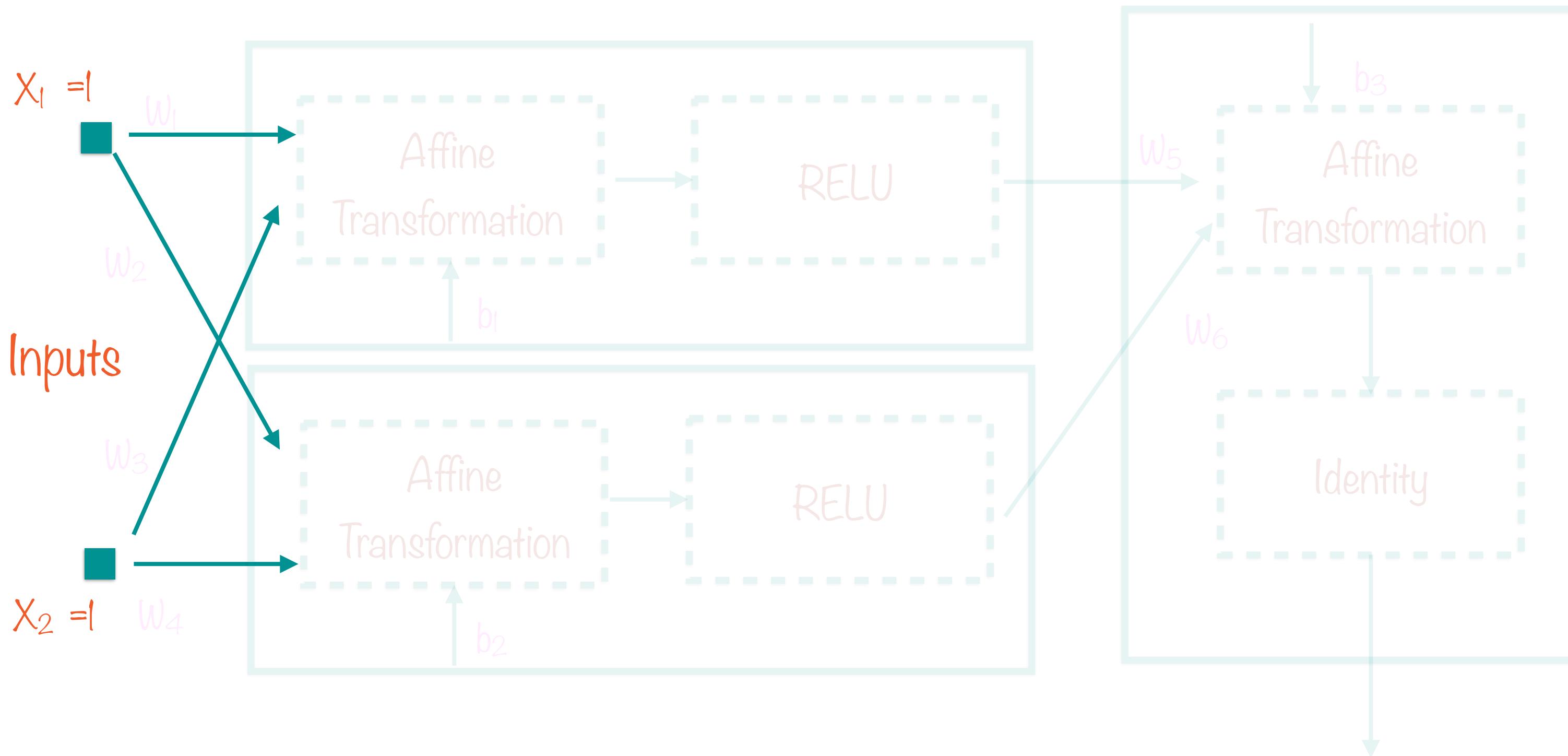


| X ₁ | X ₂ | Y |
|----------------|----------------|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

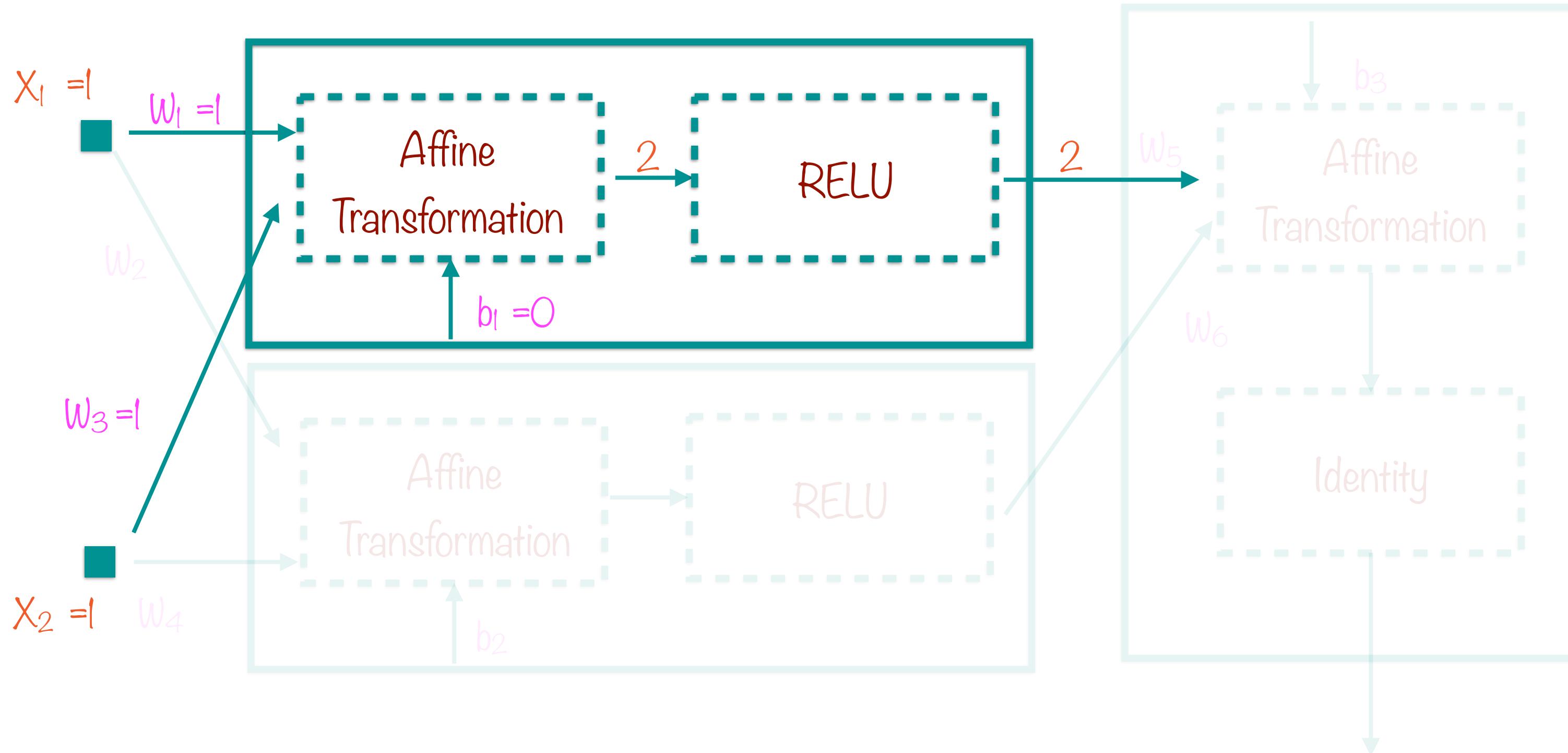
“Learning” XOR

Reverse-engineering XOR requires 3 neurons (arranged in 2 layers) as well as a non-linear activation function

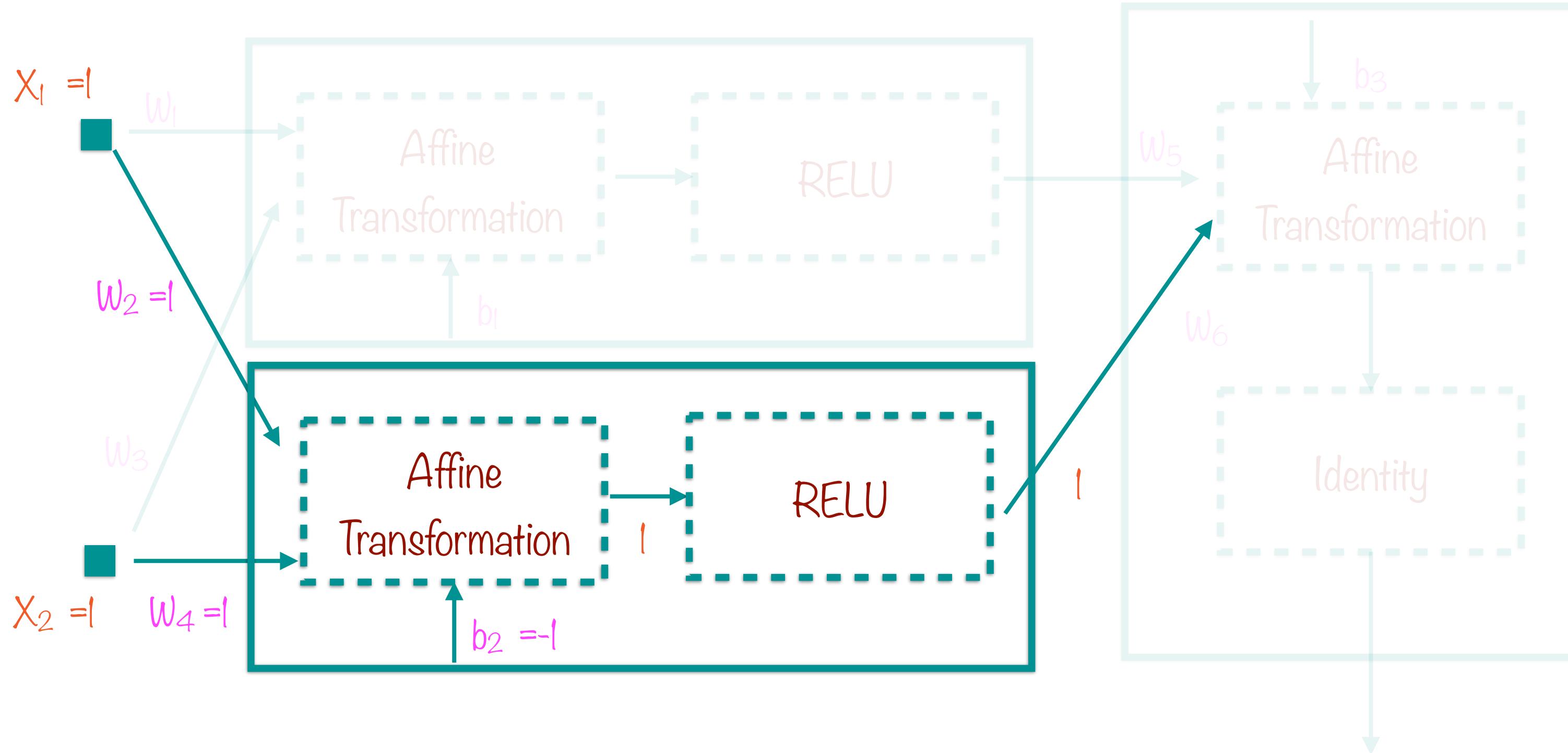
3-Neuron XOR



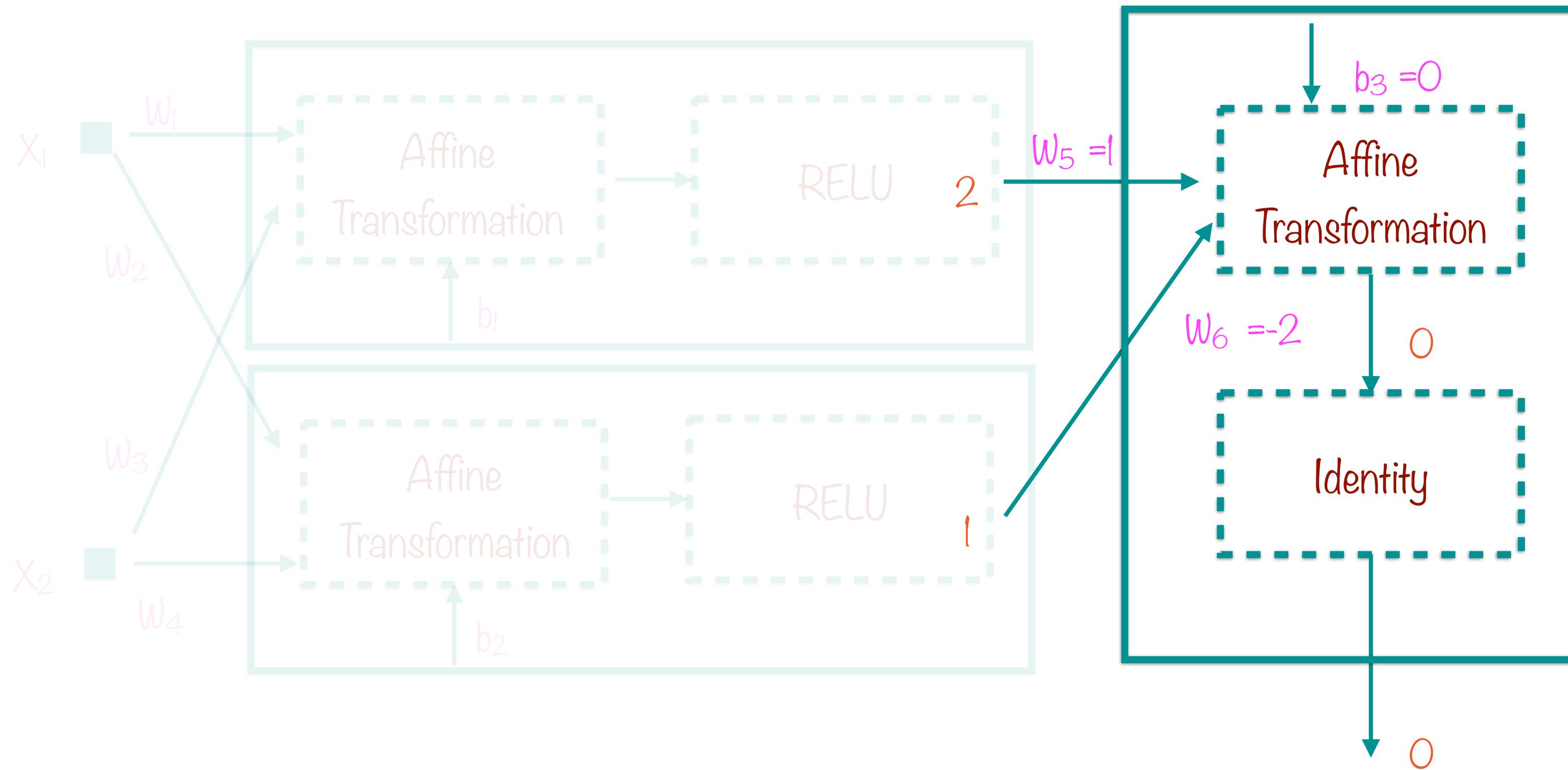
Weights and Bias of Neuron #1



Weights and Bias of Neuron #2



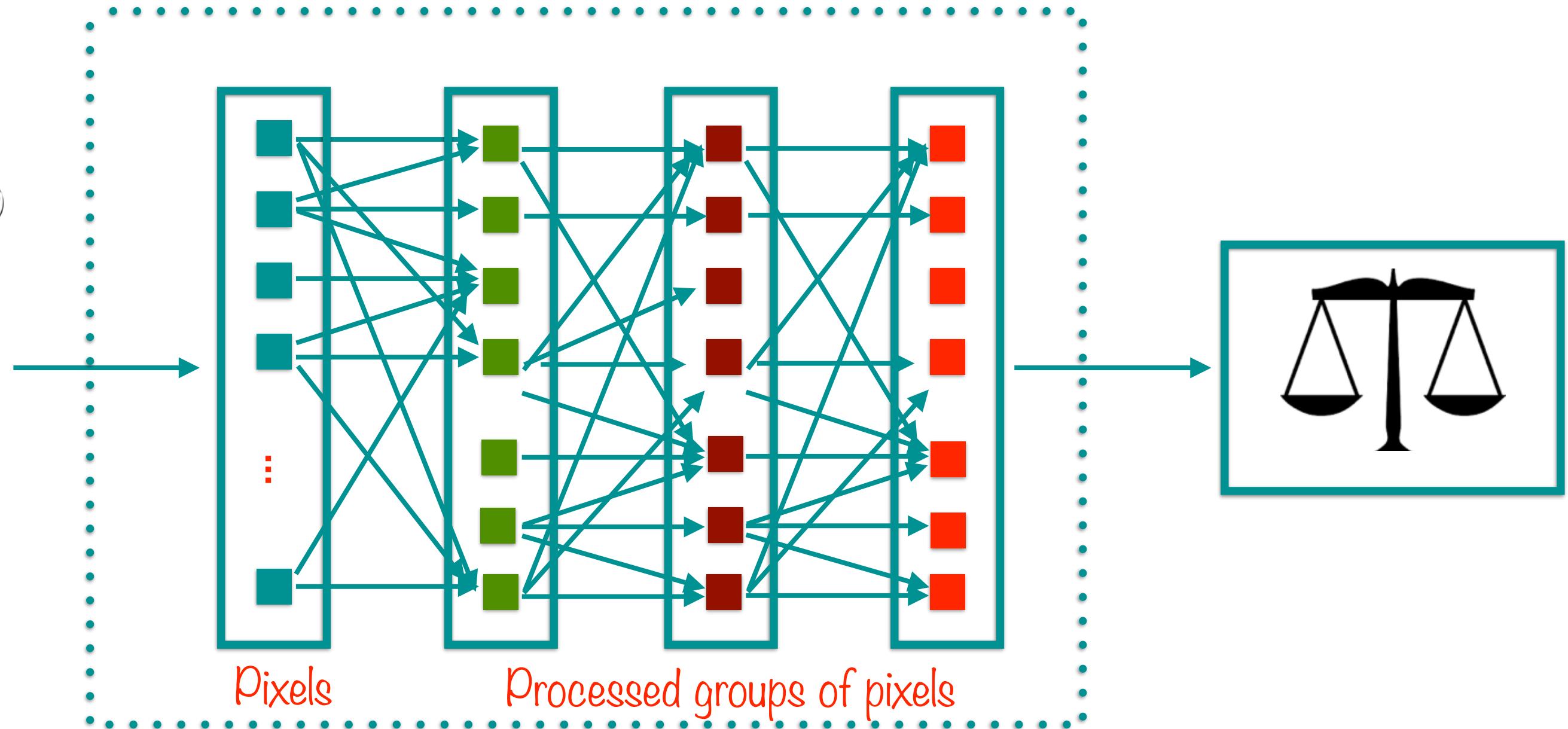
Weights and Bias of Neuron #3



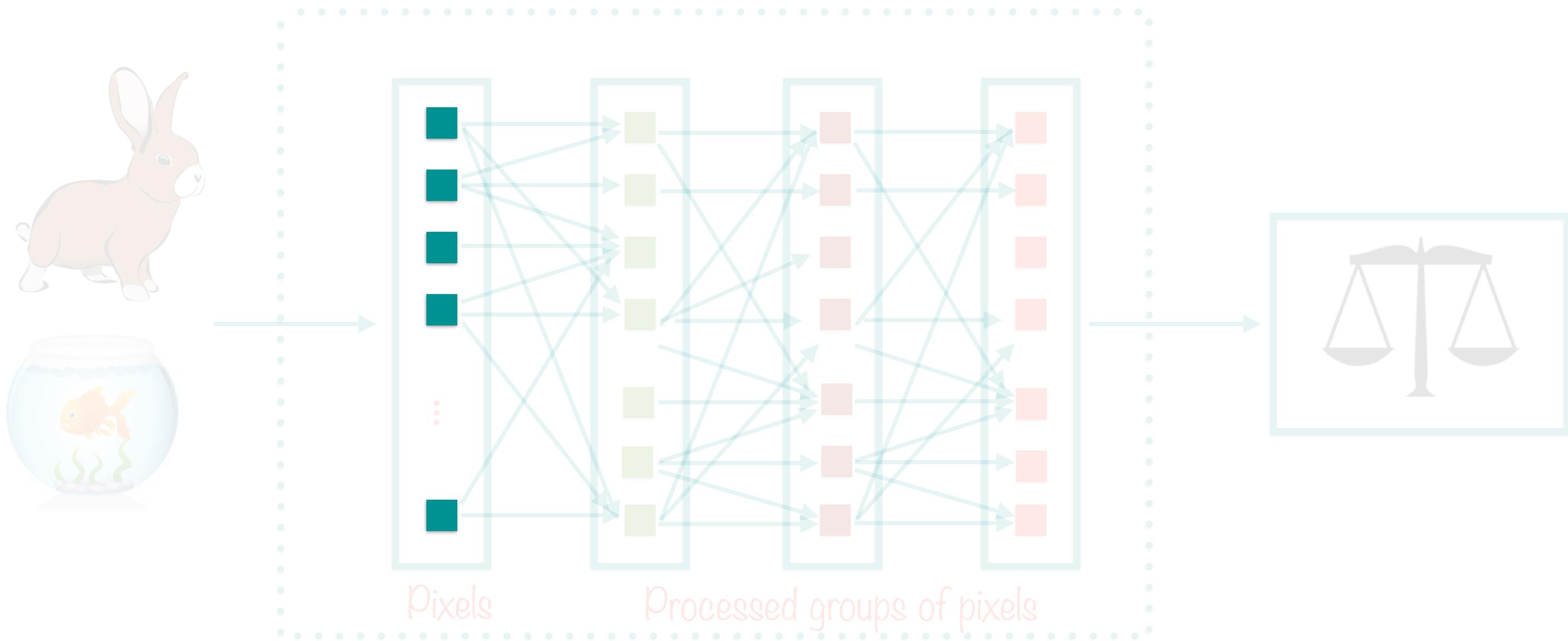
Choice of Activation Function



Corpus of
Images



Choice of Activation Function

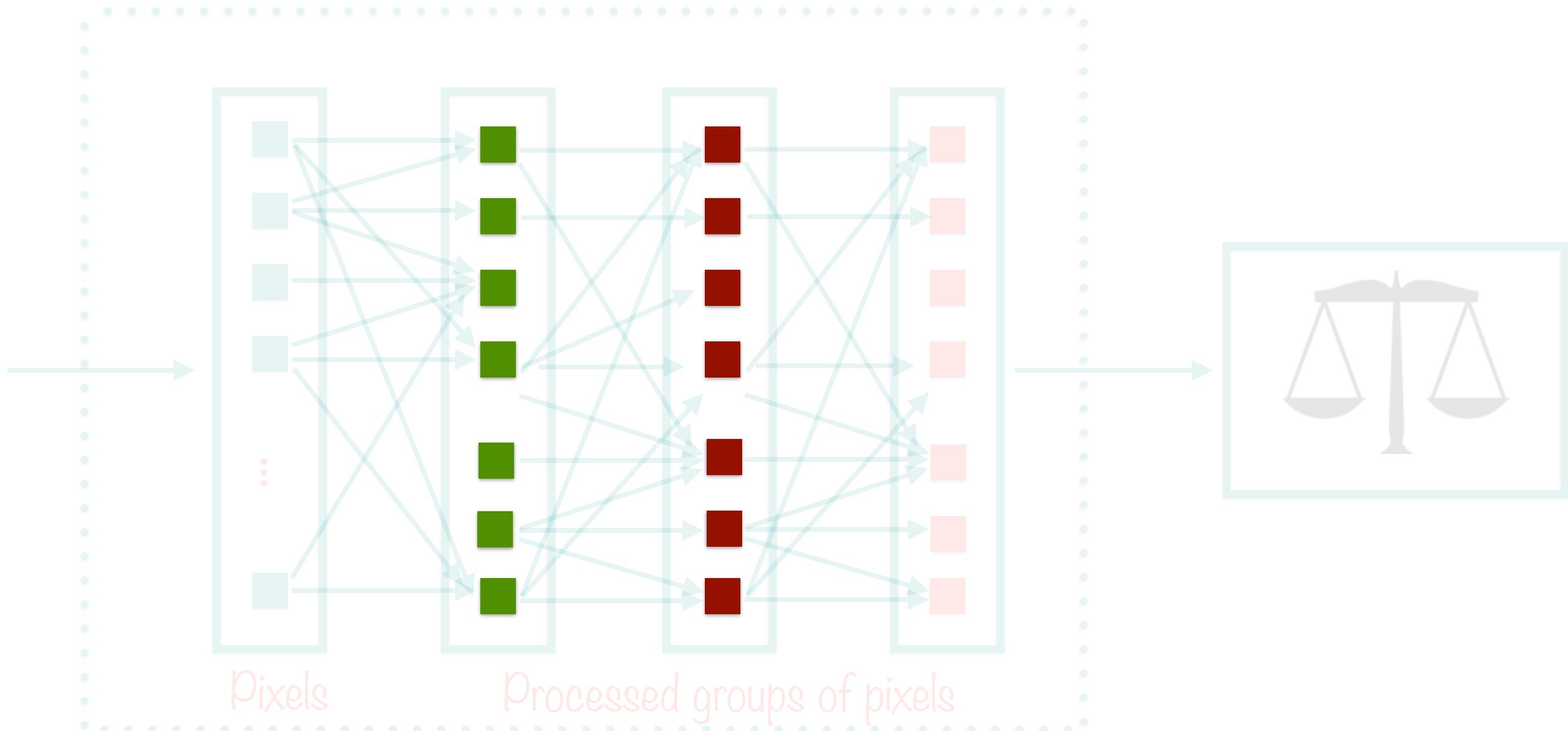


Corpus of
Images

Input layers use identity function as
activation: $f(x) = x$

ML-based Classifier

Choice of Activation Function

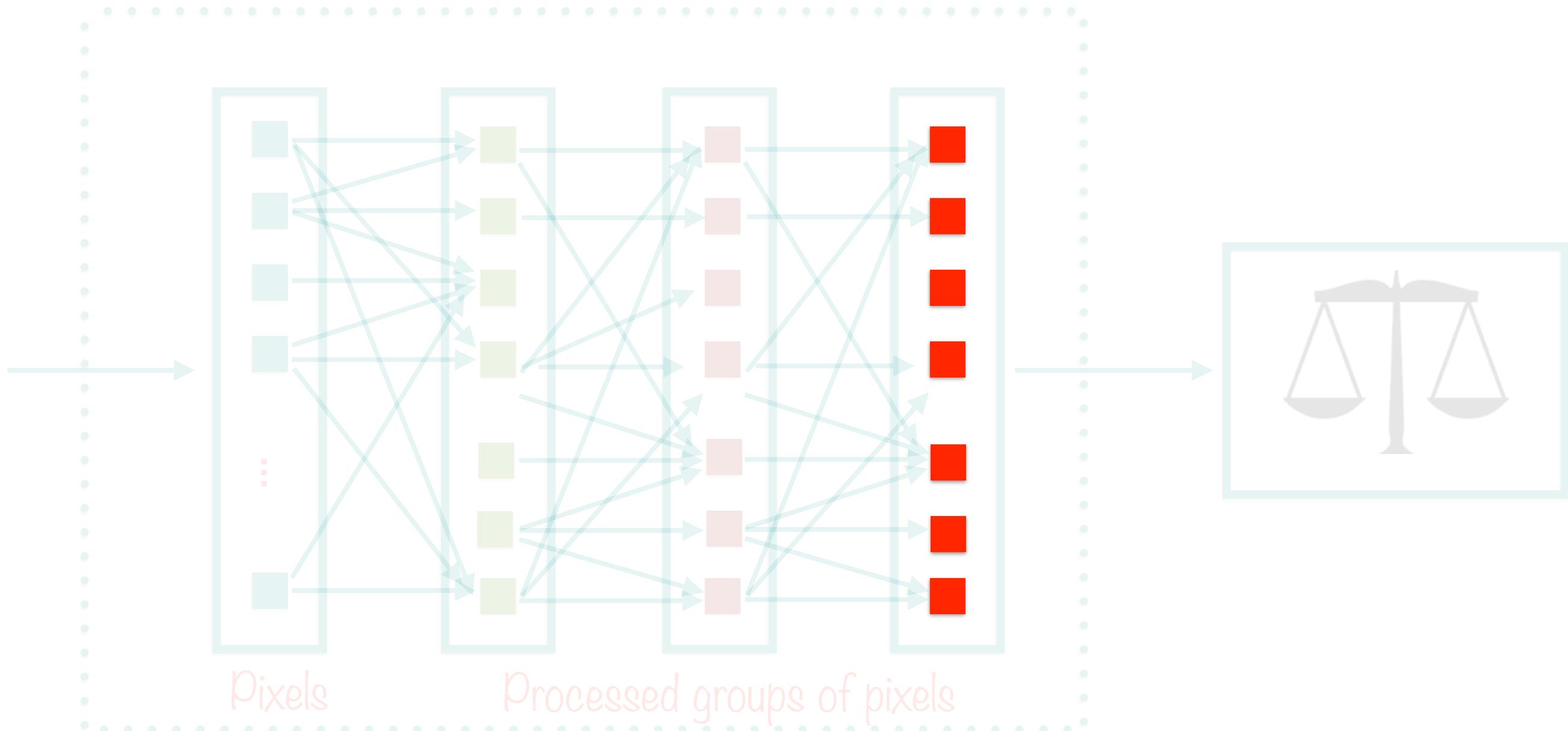


Corpus of
Images

Inner hidden layers typically use ReLU as
activation function

ML-based Classifier

Choice of Activation Function

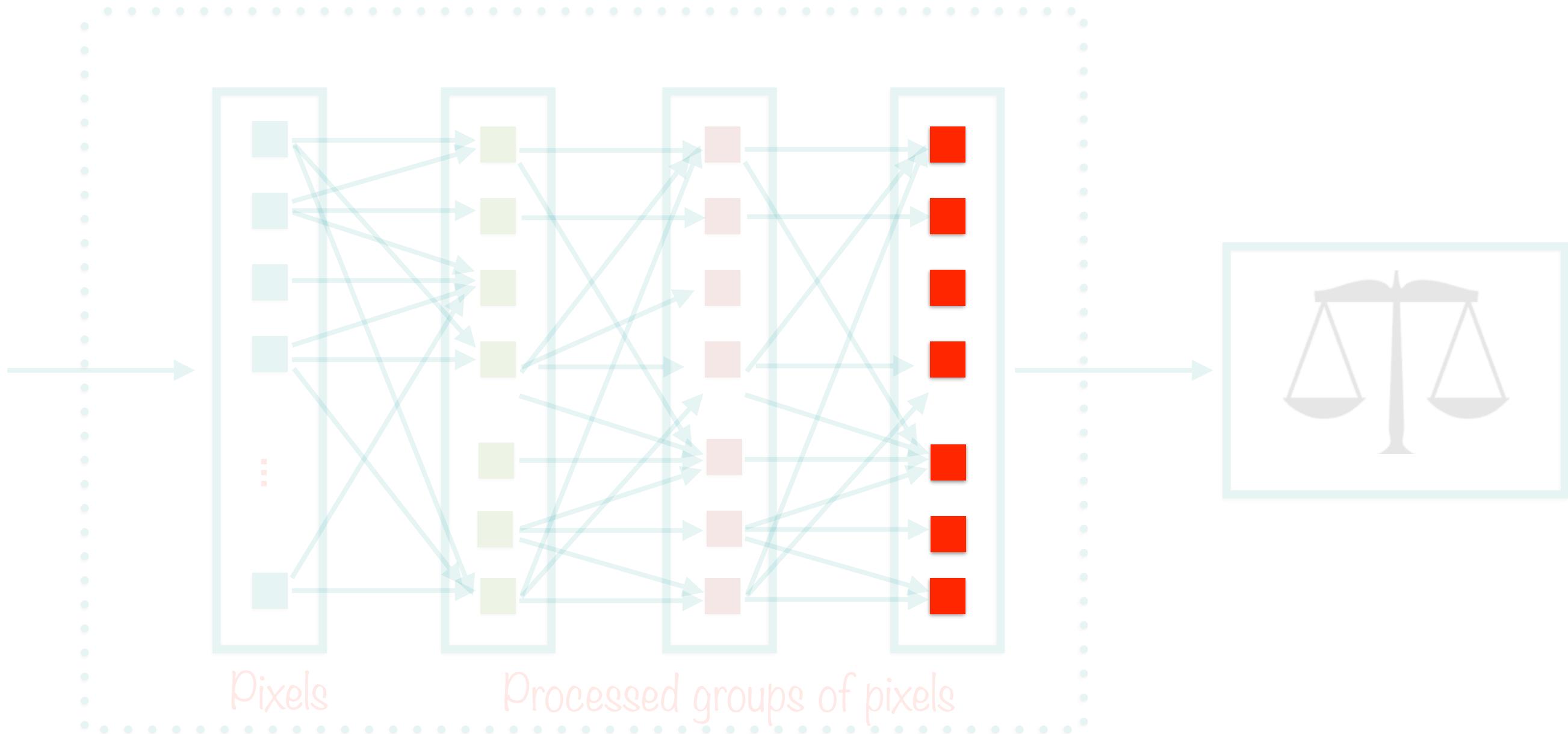


Corpus of
Images

Output layer in our XOR example used
the identity function

ML-based Classifier

Choice of Activation Function



Corpus of
Images

Output layer in classification will often use
SoftMax

ML-based Classifier



Another very common form of the activation function is the SoftMax

SoftMax(x) outputs a number between 0 and 1

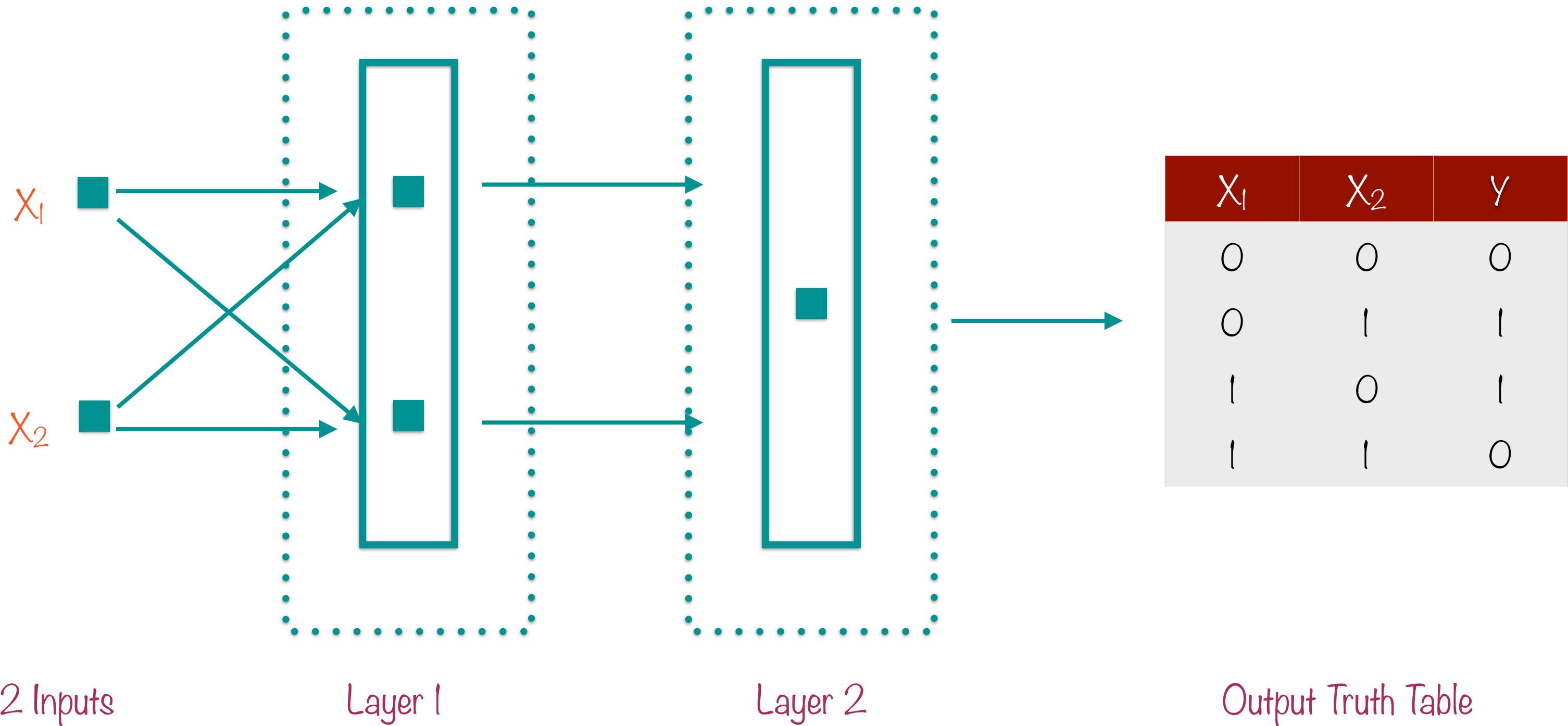
This output can be interpreted as a probability

```
def XOR(x1,x2):  
    if (x1 == x2):  
        return 0  
    return 1
```

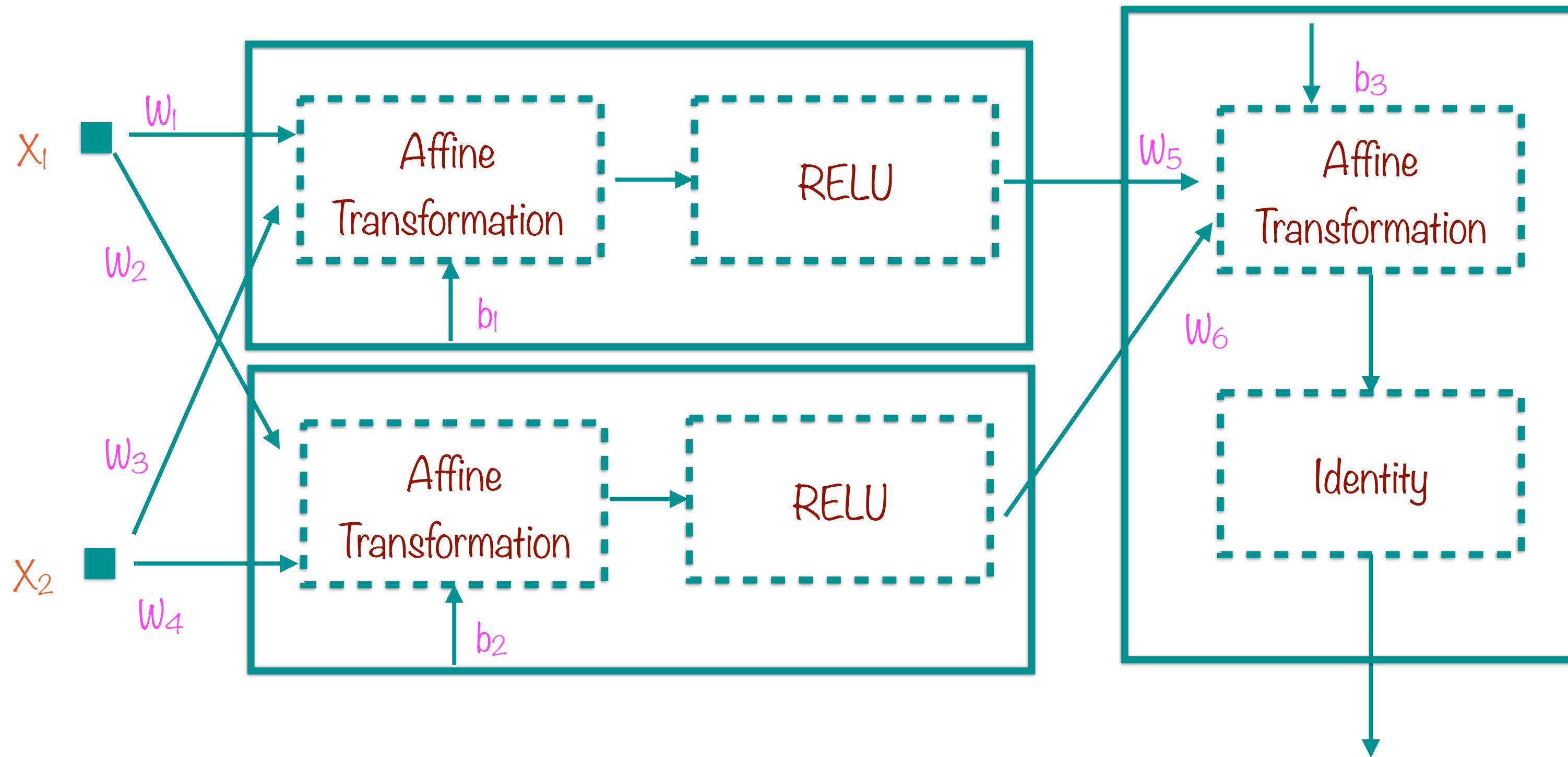
“Learning” XOR

Reverse-engineering XOR requires 3 neurons (arranged in 2 layers) as well as a non-linear activation function

XOR: 3 Neurons, 2 Layers



3-Neuron XOR



```
def doSomethingReallyComplicated(x1,x2...):
```

```
...
```

```
...
```

```
...
```

```
    return complicatedResult
```

“Learning” Arbitrarily Complex Functions

Adding layers to a neural network can “learn” (reverse-engineer) pretty much anything

Summary

A neuron is the smallest entity in a neural network

Linear regression can be learnt by a single neuron

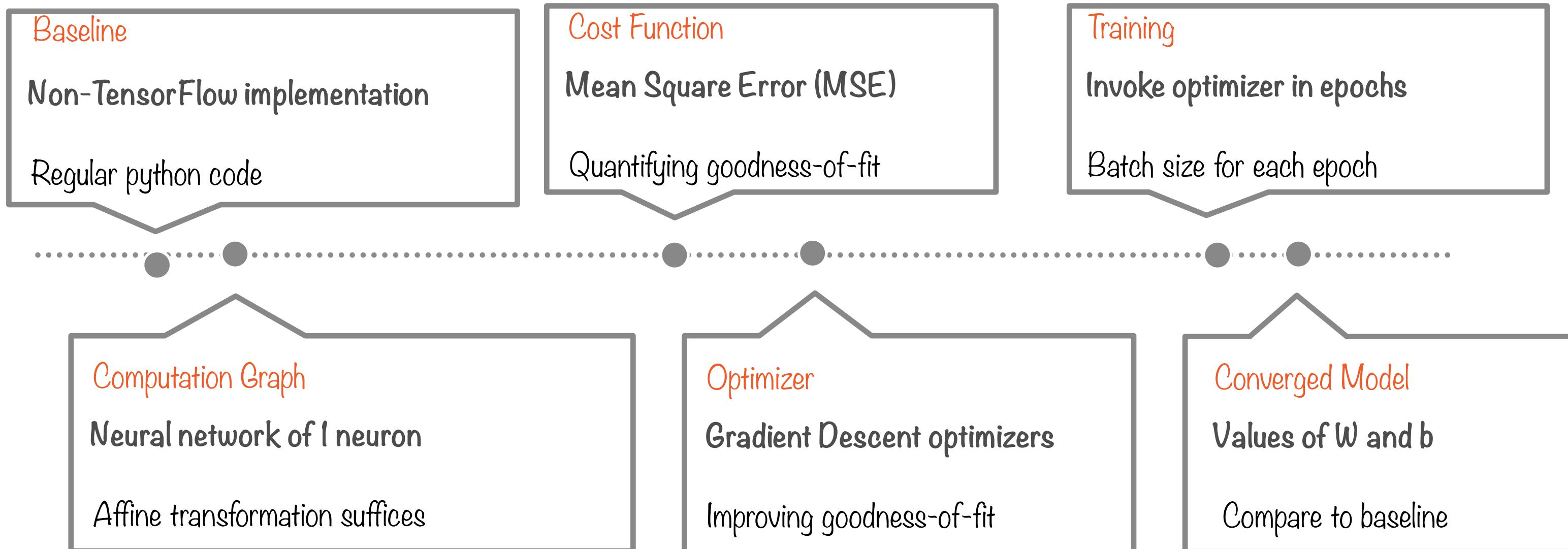
A more complex function such as XOR requires more neurons

Combinations of interconnected neurons can “learn” virtually anything

Training such networks to use the “best” parameter values is vital

Building Linear Regression Models Using TensorFlow

Implementing Regression in TensorFlow



Simple Regression



Cause

Independent variable



Effect

Dependent variable

Implementing Regression in TensorFlow

Baseline

Non-TensorFlow implementation

Regular python code



Regression in Python

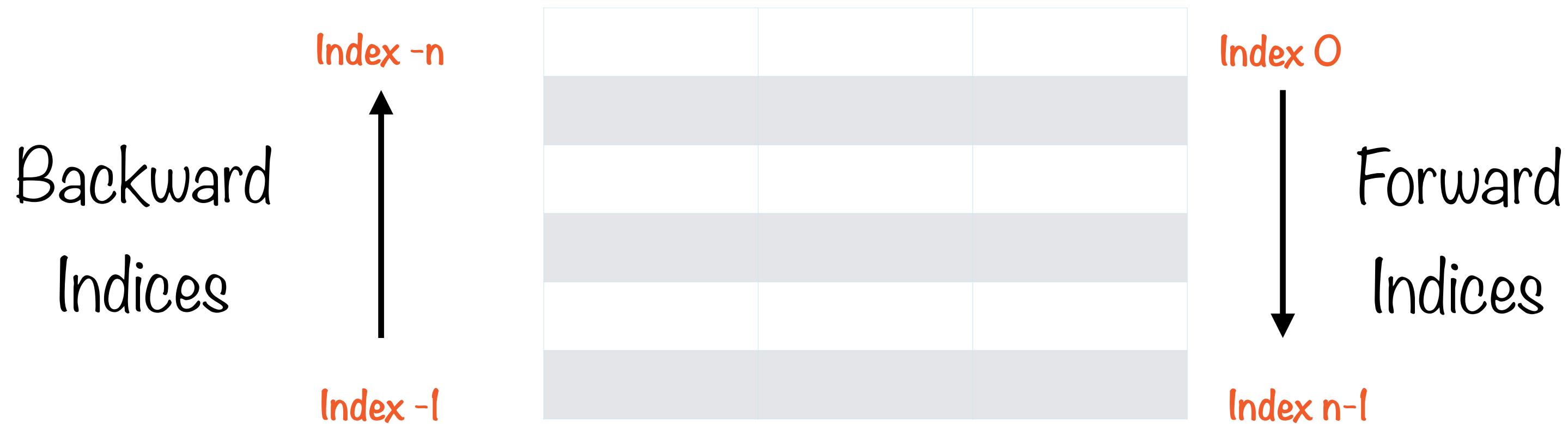
Pandas for dataframes

NumPy for arrays

Statsmodels for regression

Matplotlib for plots

Negative Indices In Python

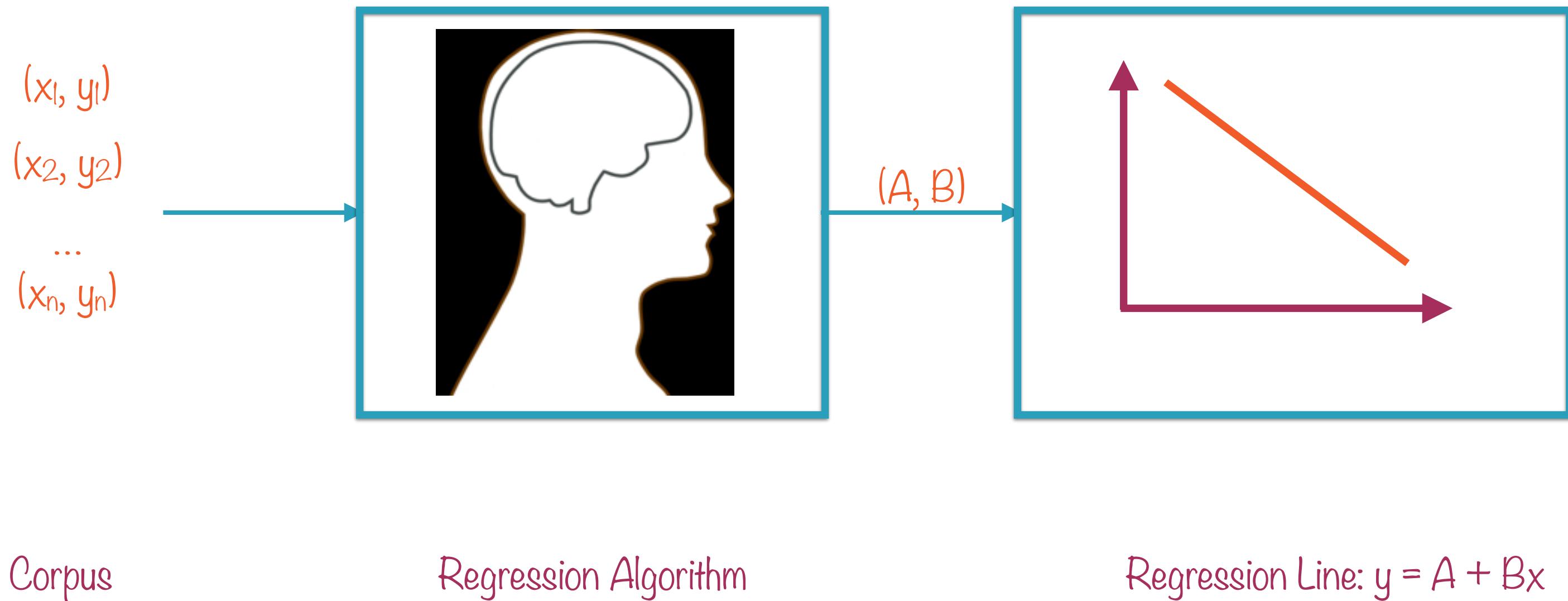


Prices to Returns

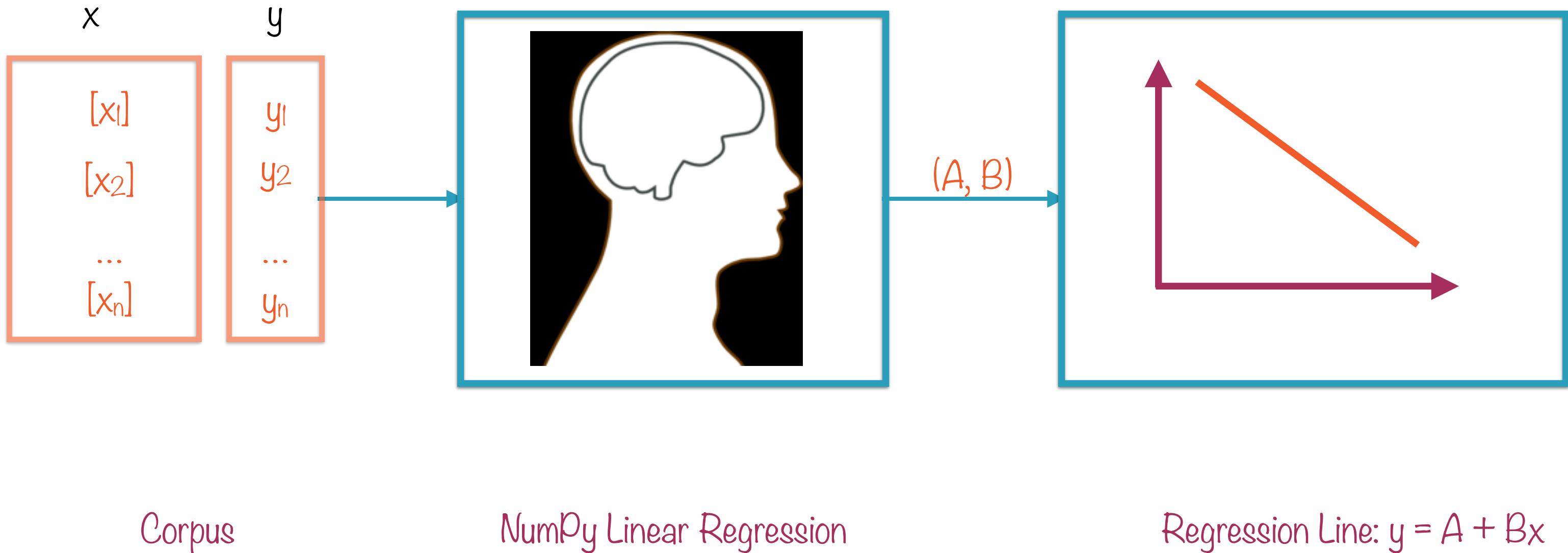
$$\text{Returns} = \frac{\text{Prices}[:-1]}{\text{Prices}[1:]} - 1$$

The diagram illustrates the calculation of returns from prices. It shows two matrices: one for 'Prices' with dimensions $n \times n$ and another for 'Returns' with dimensions $n \times n$. The 'Prices' matrix has its first row removed (indicated by the slice $[:-1]$) and its last column removed (indicated by the slice $[1:]$). The 'Returns' matrix has its first row removed (indicated by the slice $[:-1]$) and its last column removed (indicated by the slice $[1:]$). The division operation is shown as a fraction with the 'Prices' matrix as the numerator and the 'Returns' matrix as the denominator.

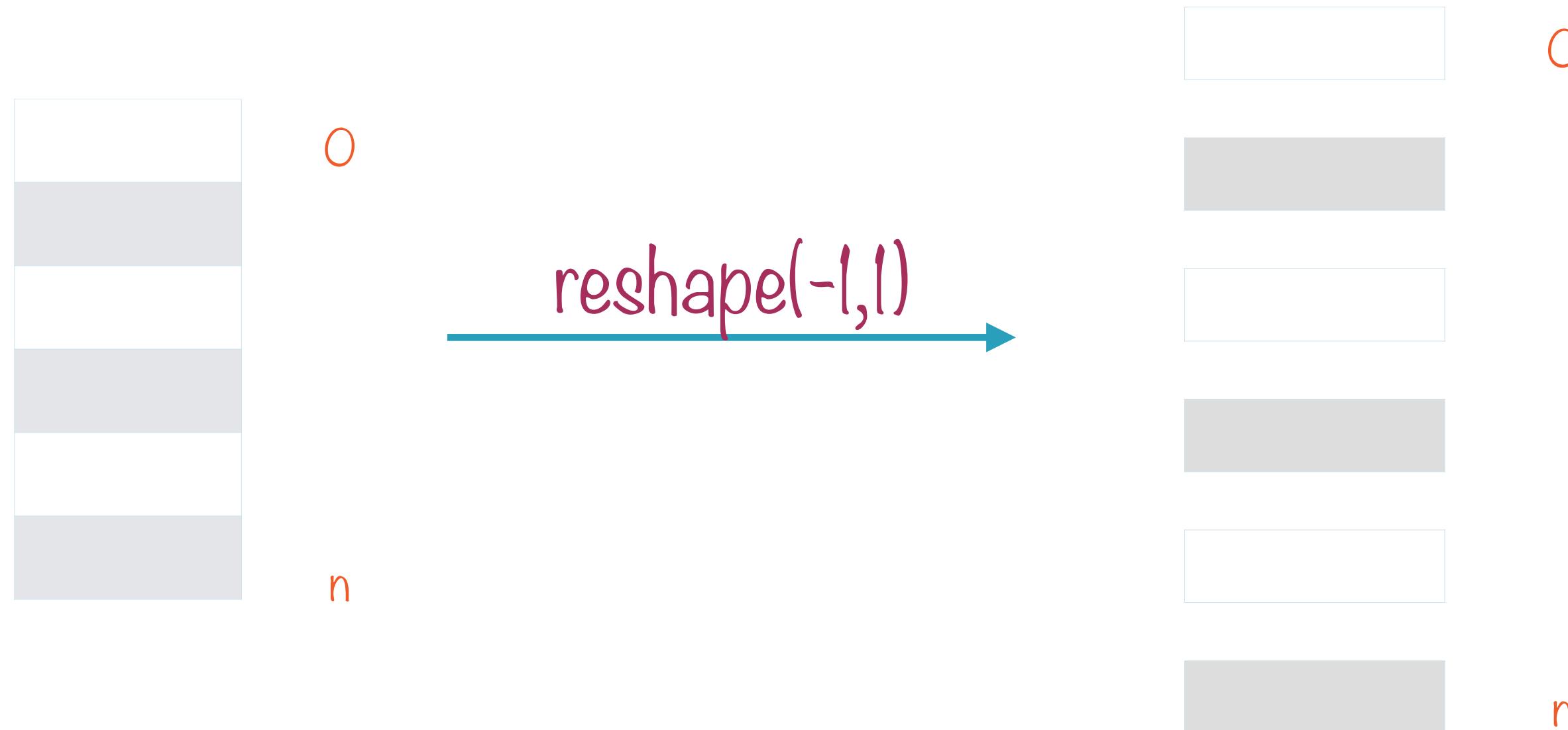
ML-based Regression Model



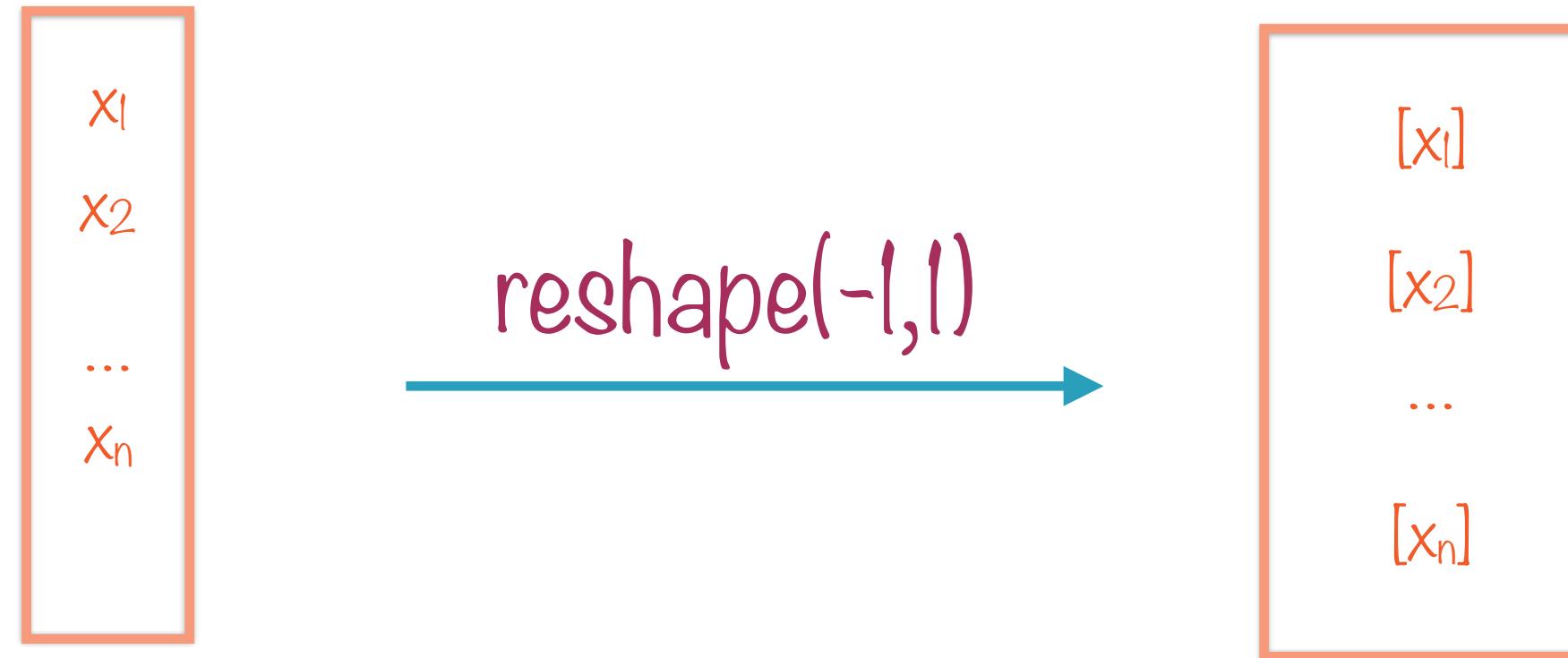
ML-based Regression Model



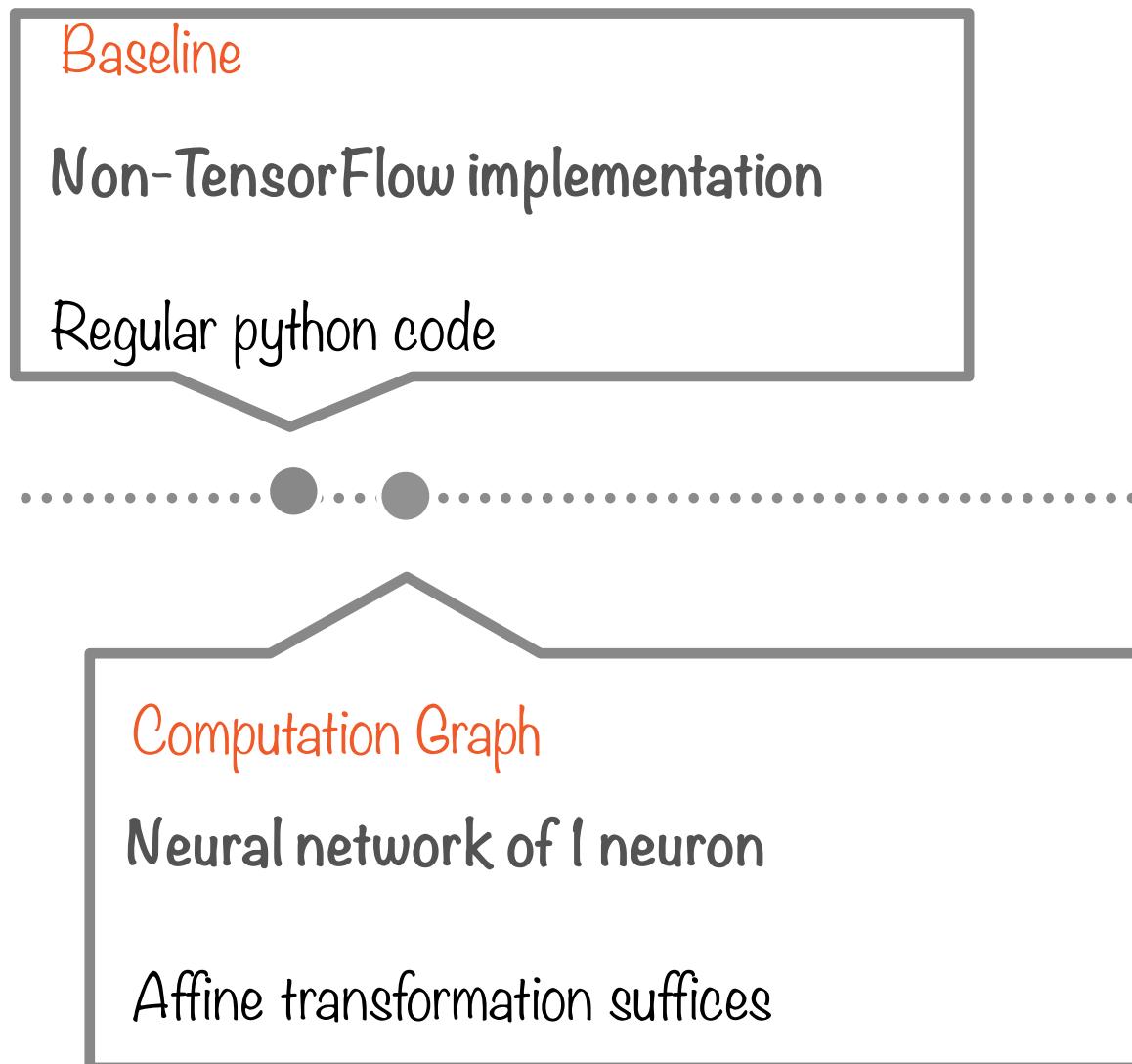
Reshaping in NumPy



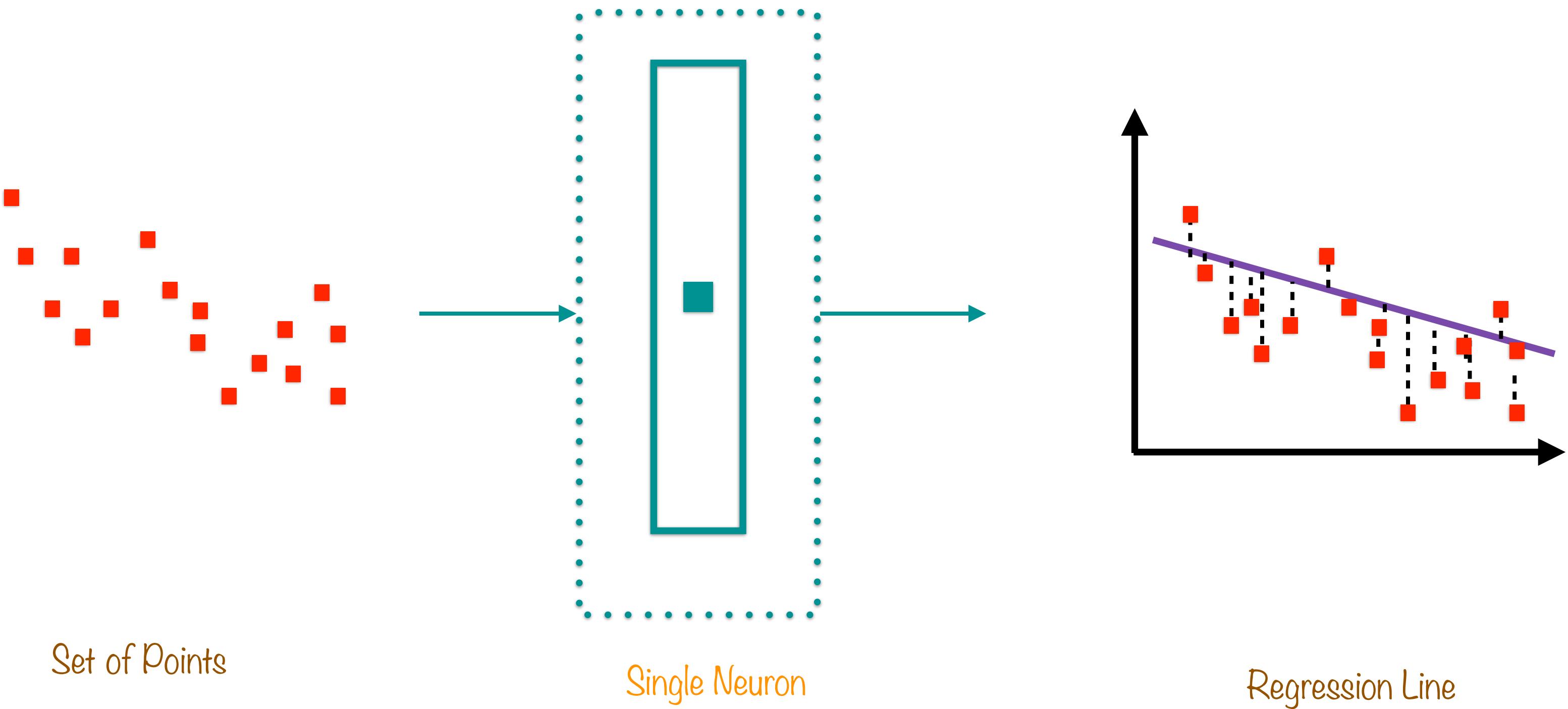
Reshaping in NumPy



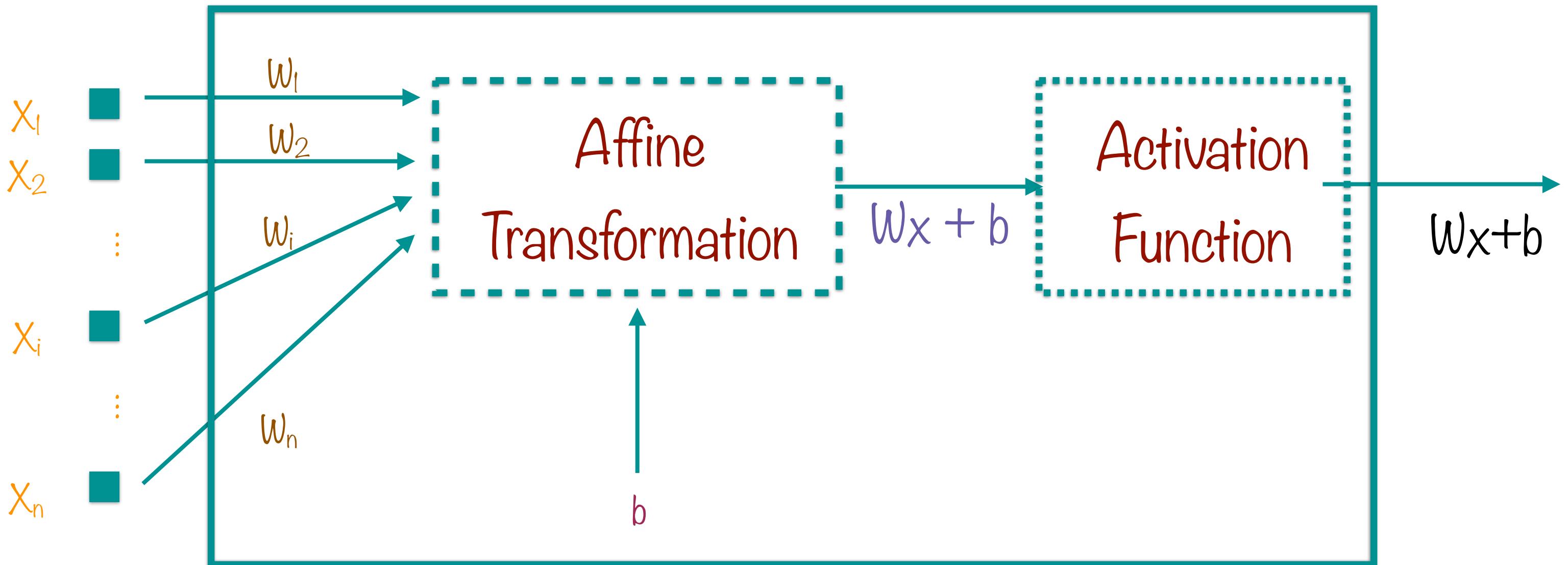
Implementing Regression in TensorFlow



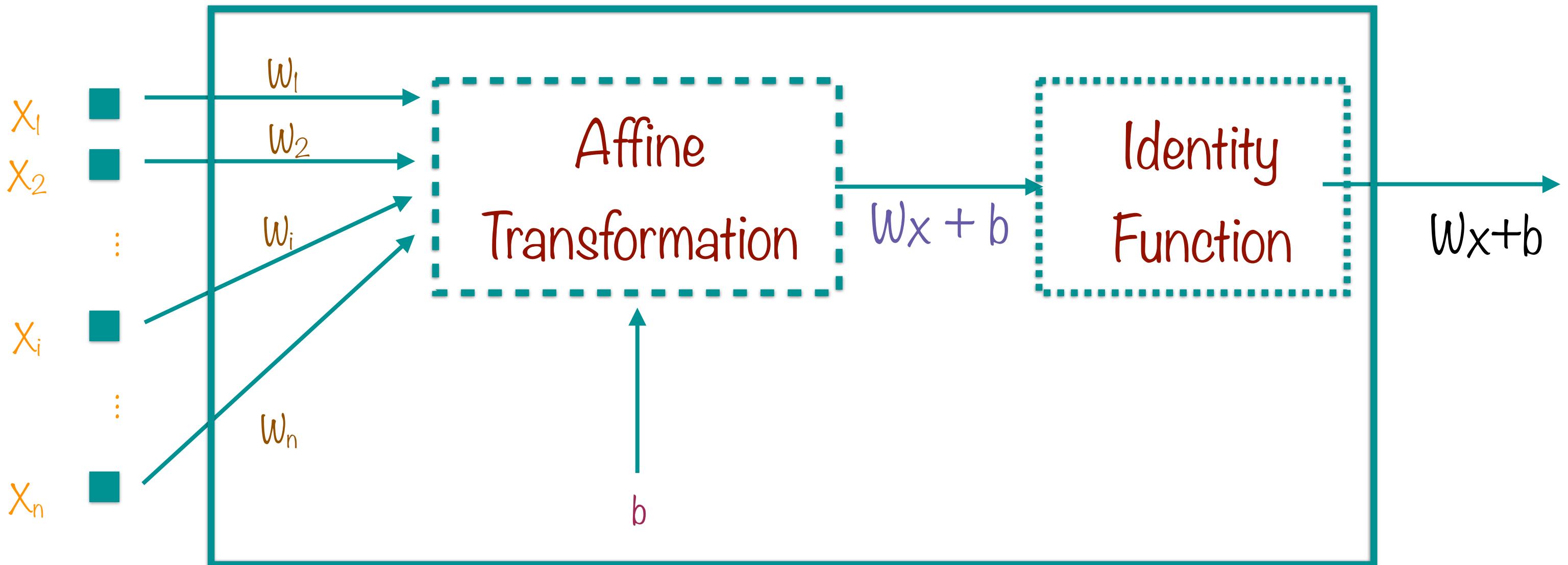
Regression: The Simplest Neural Network



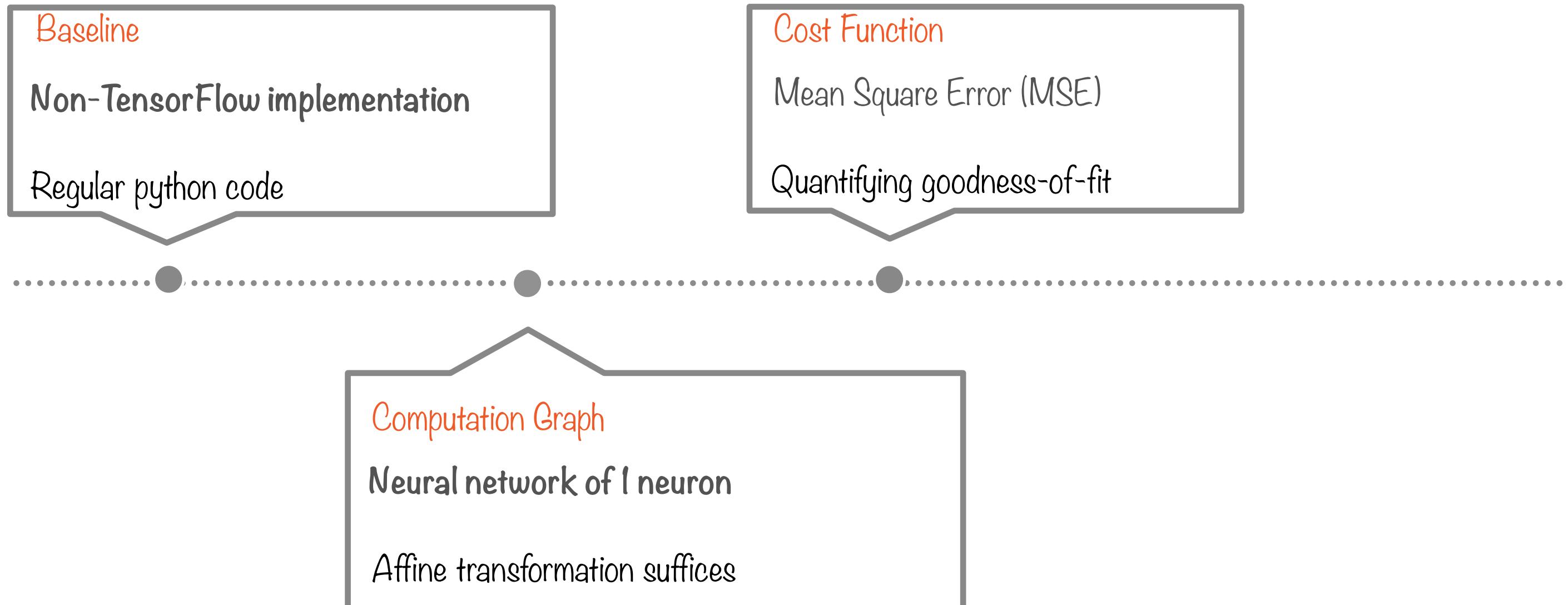
Regression: The Simplest Neural Network



Regression: The Simplest Neural Network



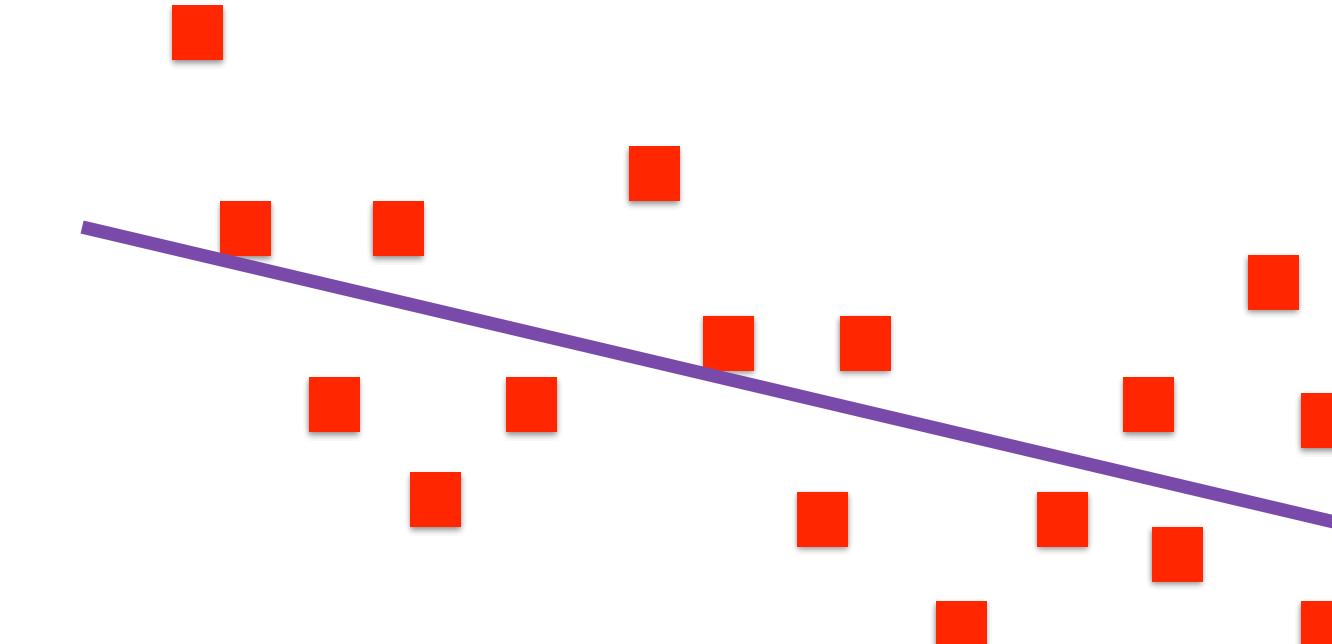
Implementing Regression in TensorFlow



The “Best” Regression Line



y



Line 1: $y = A_1 + B_1x$



Line 2: $y = A_2 + B_2x$



x

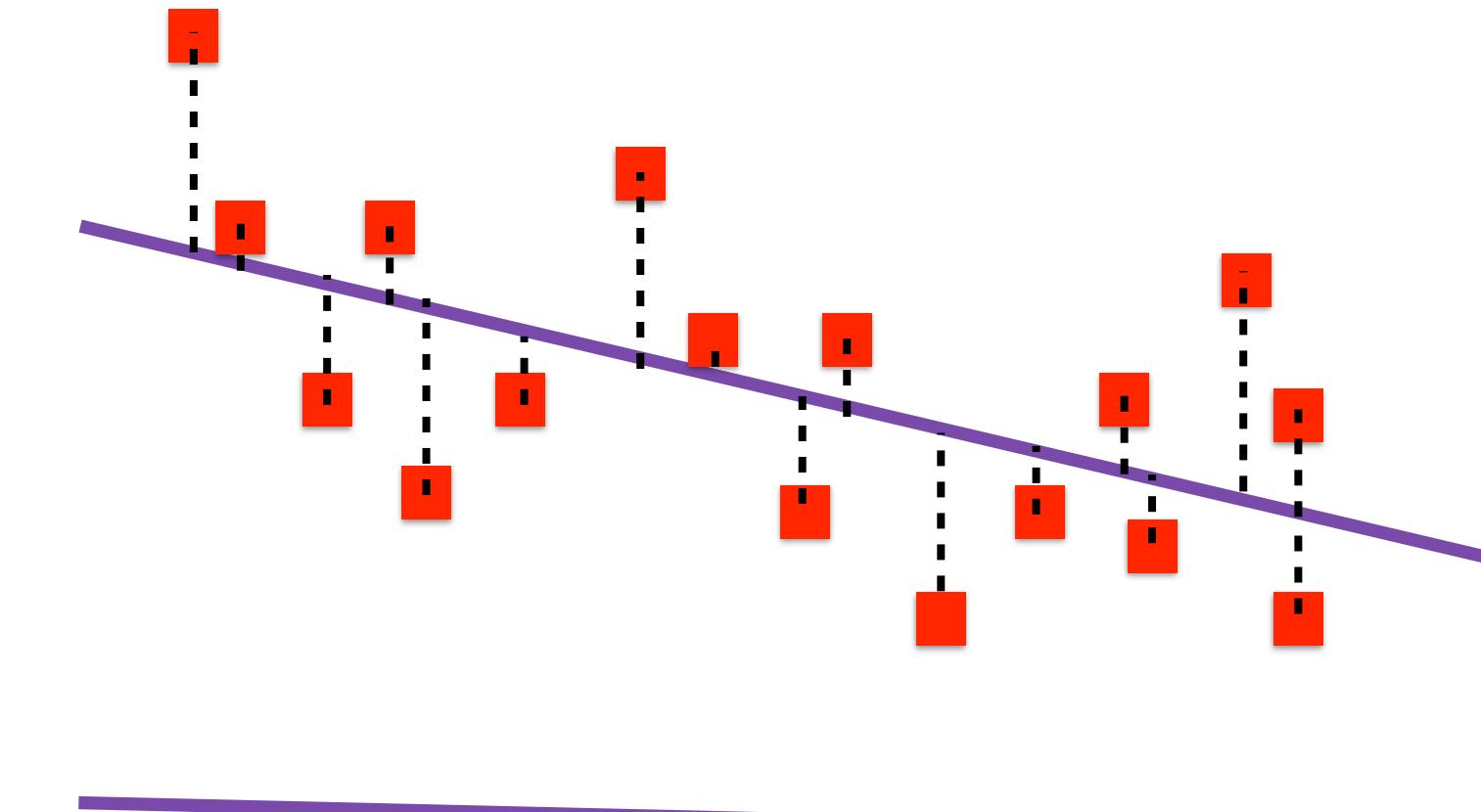


Let's compare two lines, Line 1 and Line 2

Minimising Least Square Error



y



Line 1: $y = A_1 + B_1x$

Line 2: $y = A_2 + B_2x$

x

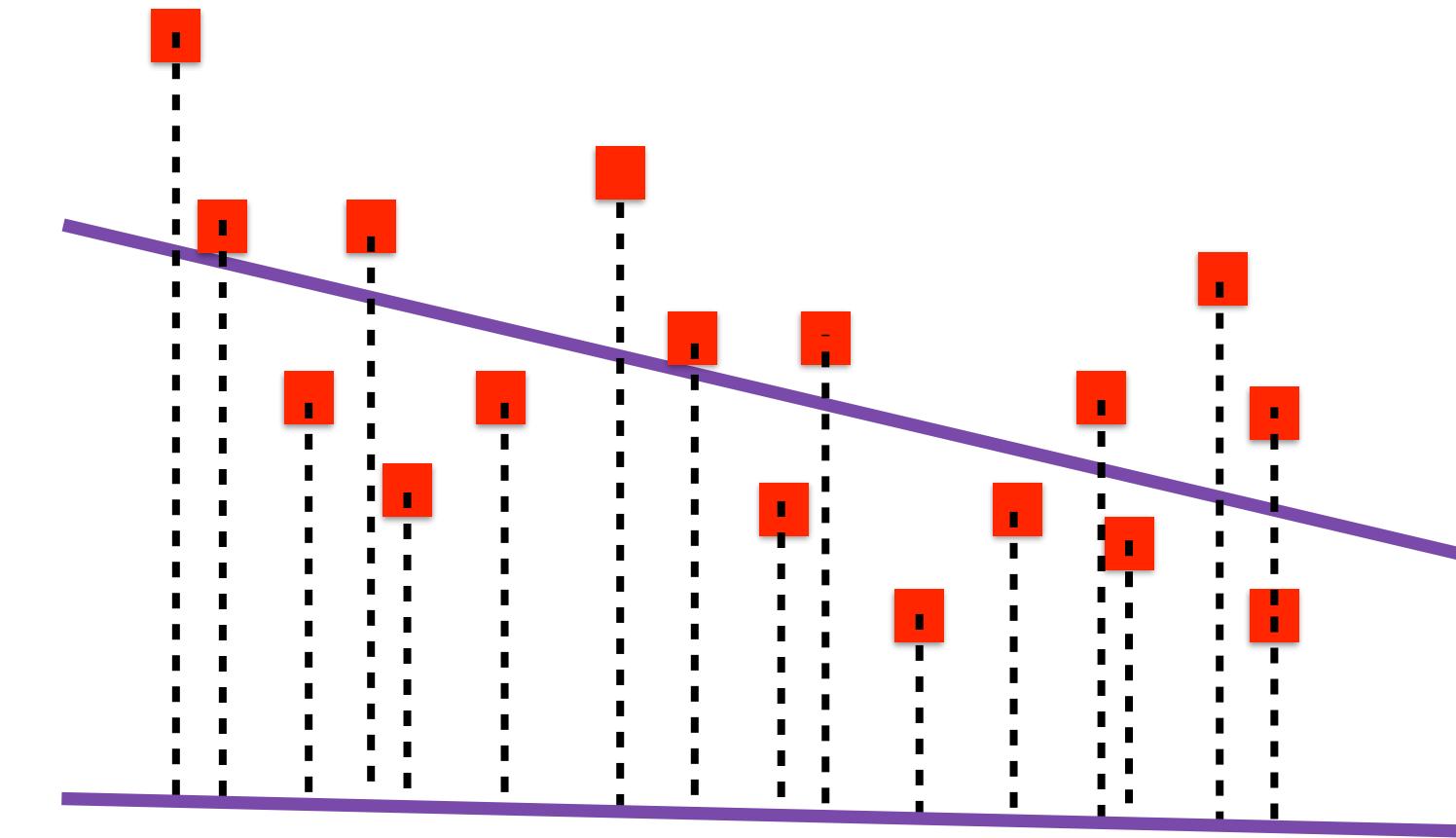


Drop vertical lines from each point to the lines A and B

Minimising Least Square Error



y



Line 1: $y = A_1 + B_1x$

Line 2: $y = A_2 + B_2x$

x



Drop vertical lines from each point to the lines A and B

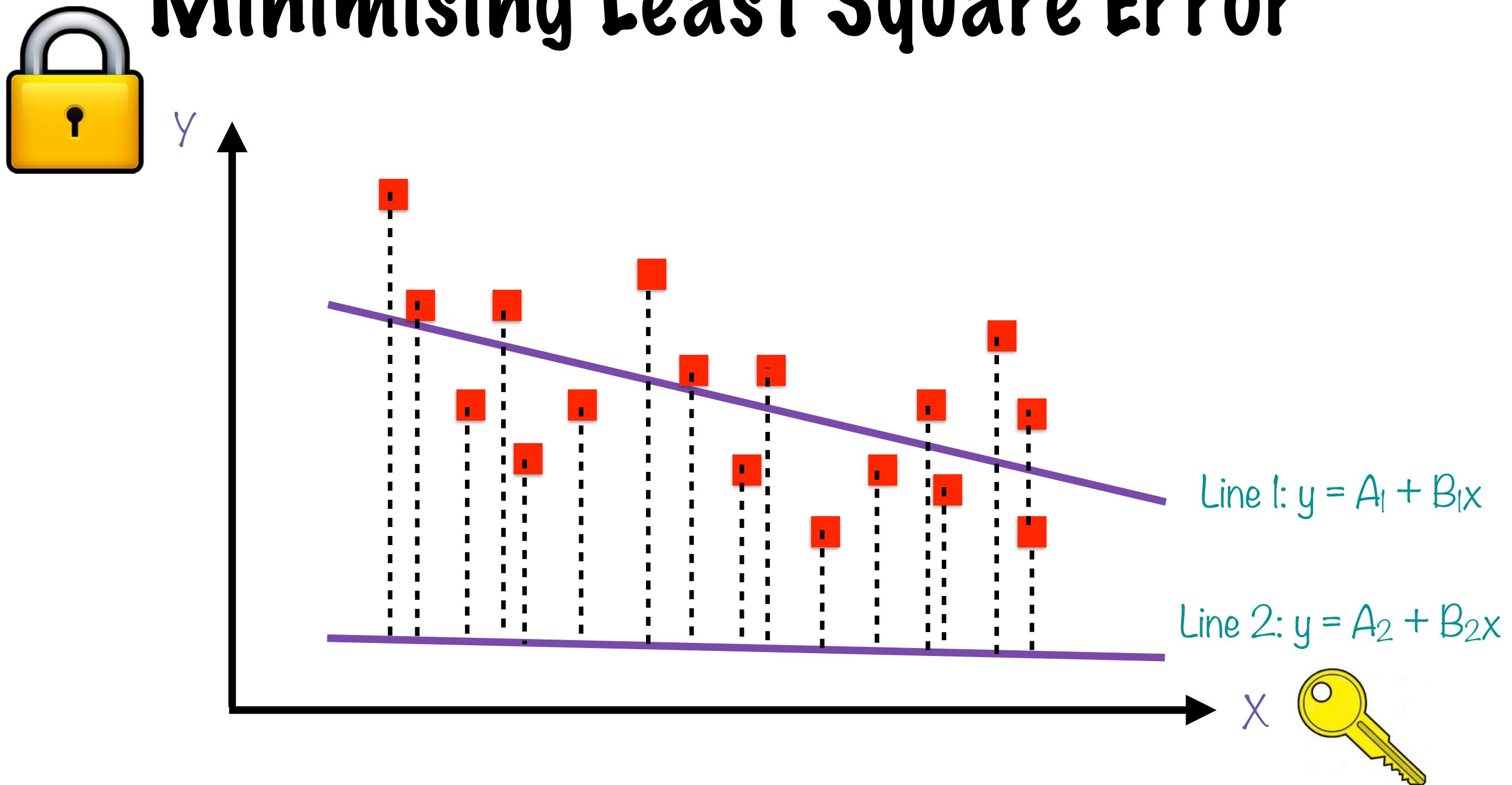
Simple Regression

Regression Equation:

$$y = A + Bx$$

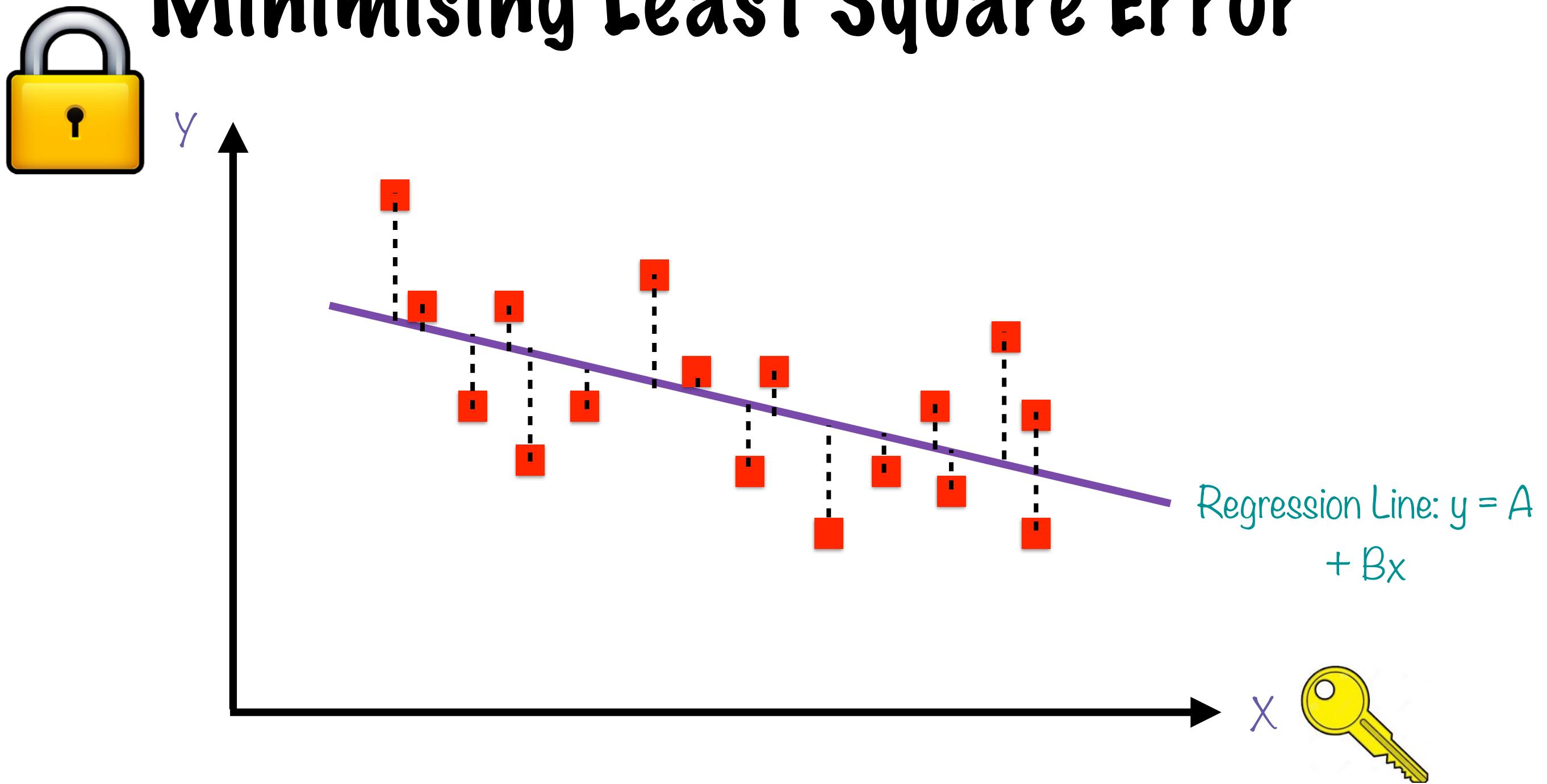
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \\ y_n \end{bmatrix} = A \begin{bmatrix} 1 \\ \dots \\ 1 \end{bmatrix} + B \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ \dots \\ e_n \end{bmatrix}$$

Minimising Least Square Error



The “best fit” line is the one where the sum of the squares of the lengths of these dotted lines is minimum

Minimising Least Square Error



The “best fit” line is called the regression line

Implementing Regression in TensorFlow

Baseline
Non-TensorFlow implementation
Regular python code

Cost Function
Mean Square Error (MSE)
Quantifying goodness-of-fit

Computation Graph
Neural network of 1 neuron
Affine transformation suffices

Optimizer
Gradient Descent optimizers
Improving goodness-of-fit

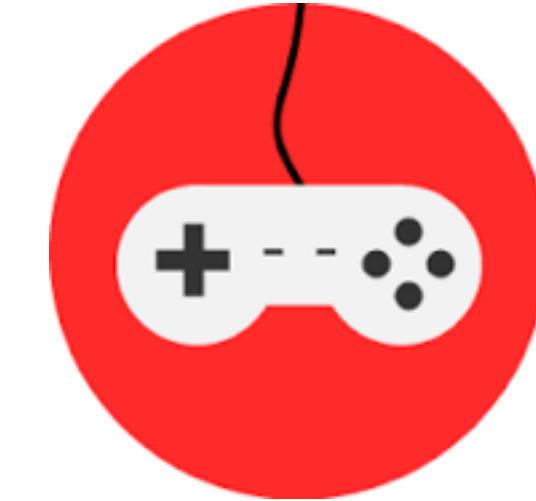
Why Choosing Is Complicated



What do we really want to achieve?



What is slowing us down?



What do we really control?

Choosing involves answering complicated questions

Why Optimization Helps



What do we really want to achieve?



What is slowing us down?



What do we really control?

Optimization forces us to mathematically pin down answers to these questions

Framing the Optimization Problem



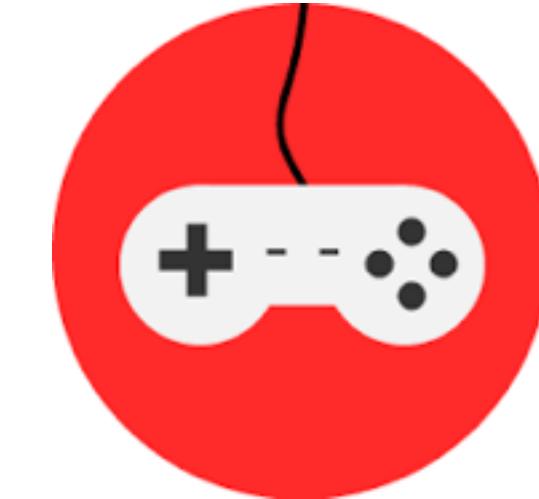
Objective Function

What we would like to achieve



Constraints

What slows us down



Decision Variables

What we really control

Collectively, these answers constitute the optimization problem

Linear Regression as an Optimization Problem



Objective Function

Minimize variance of the residuals (MSE)



Constraints

Express relationship as a straight line

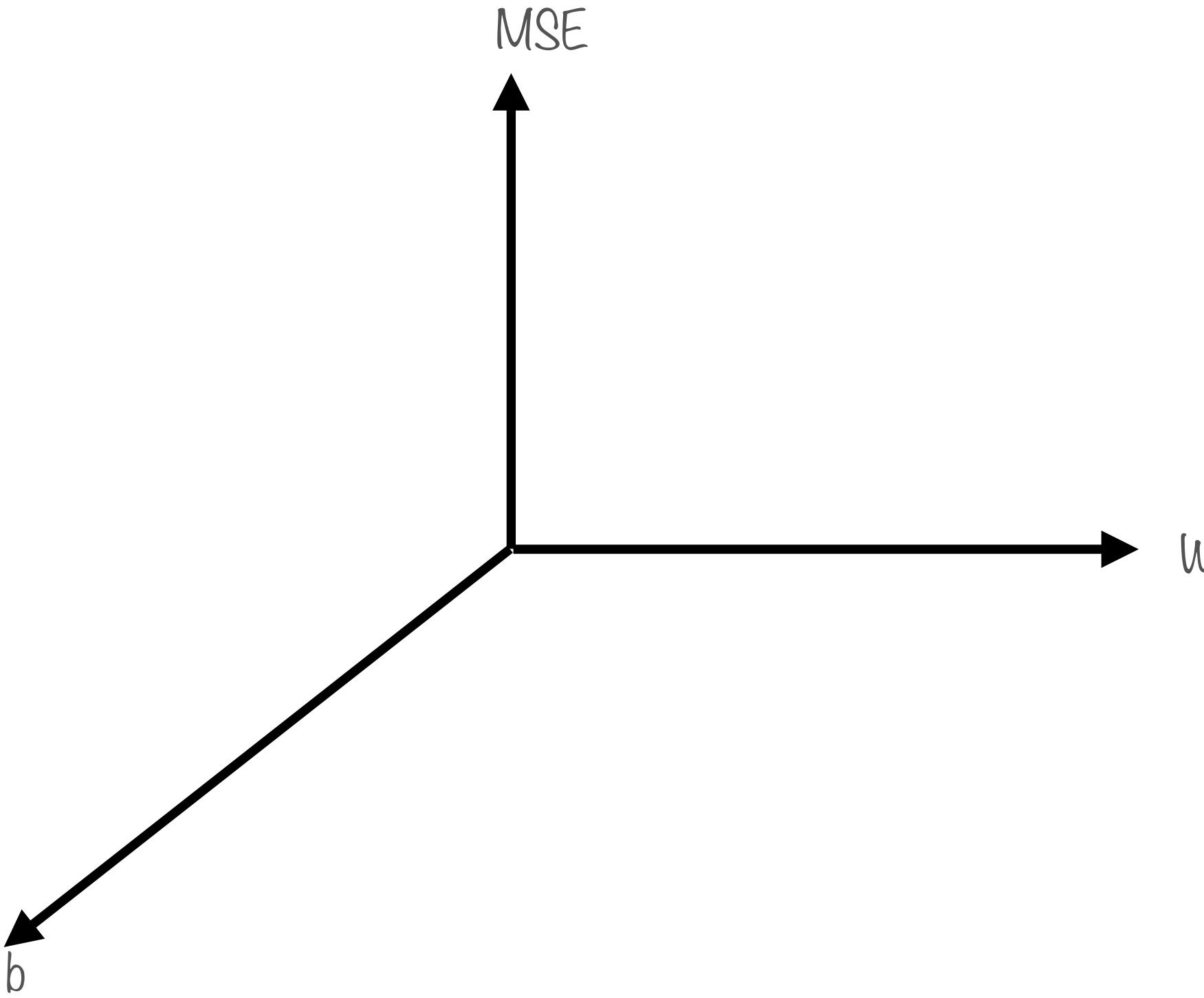
$$y = Wx + b$$



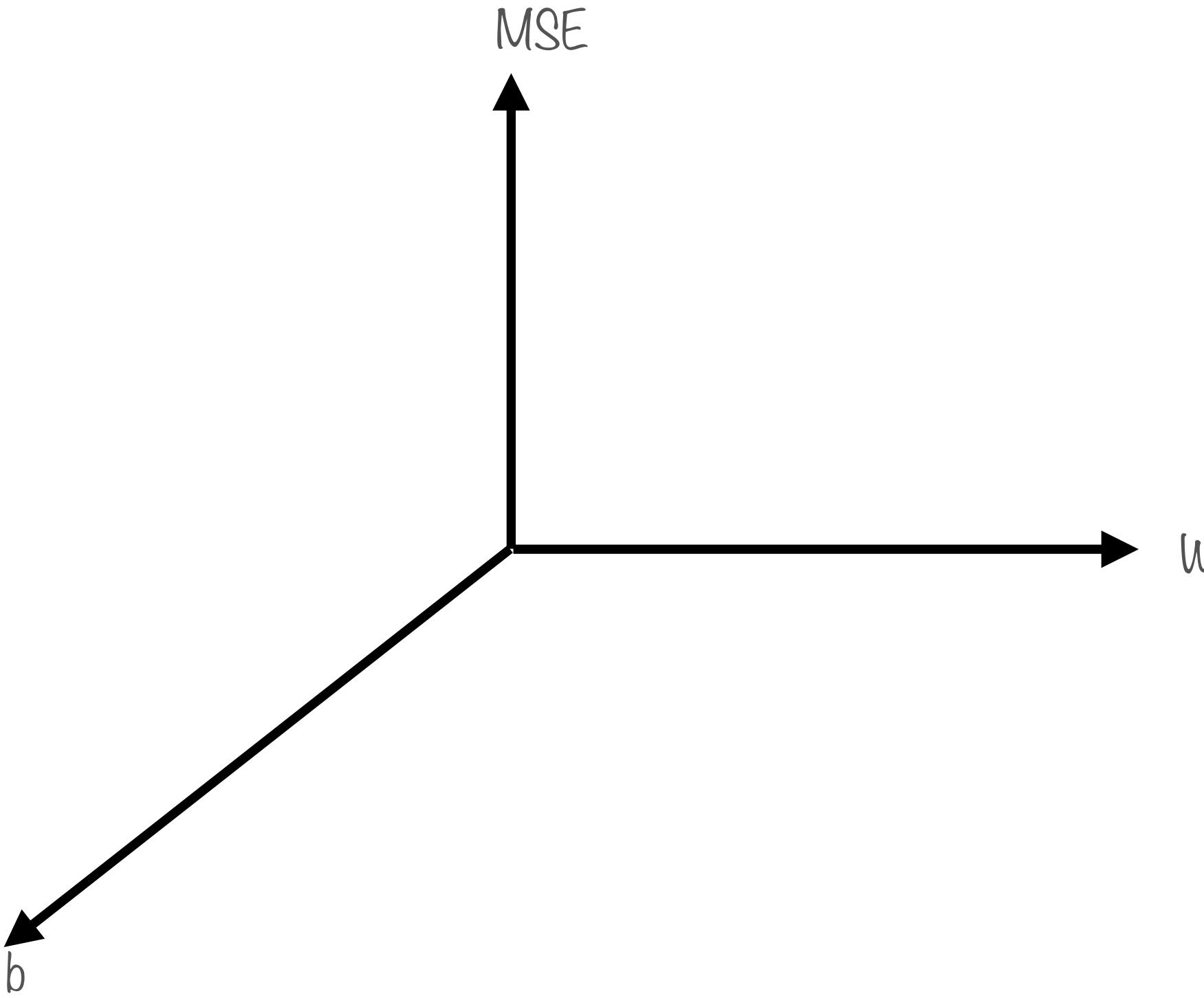
Decision Variables

Values of W and b

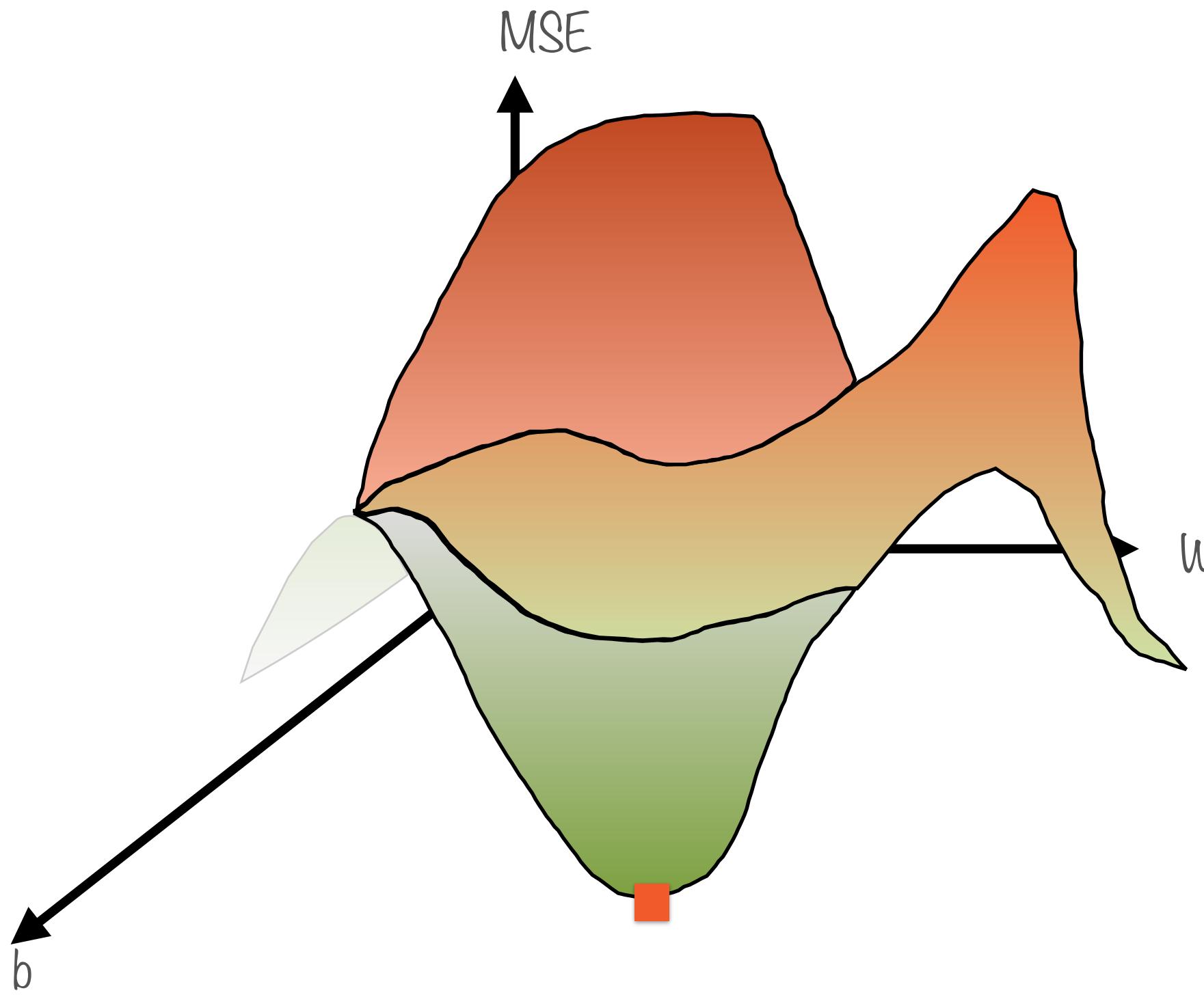
Minimizing MSE



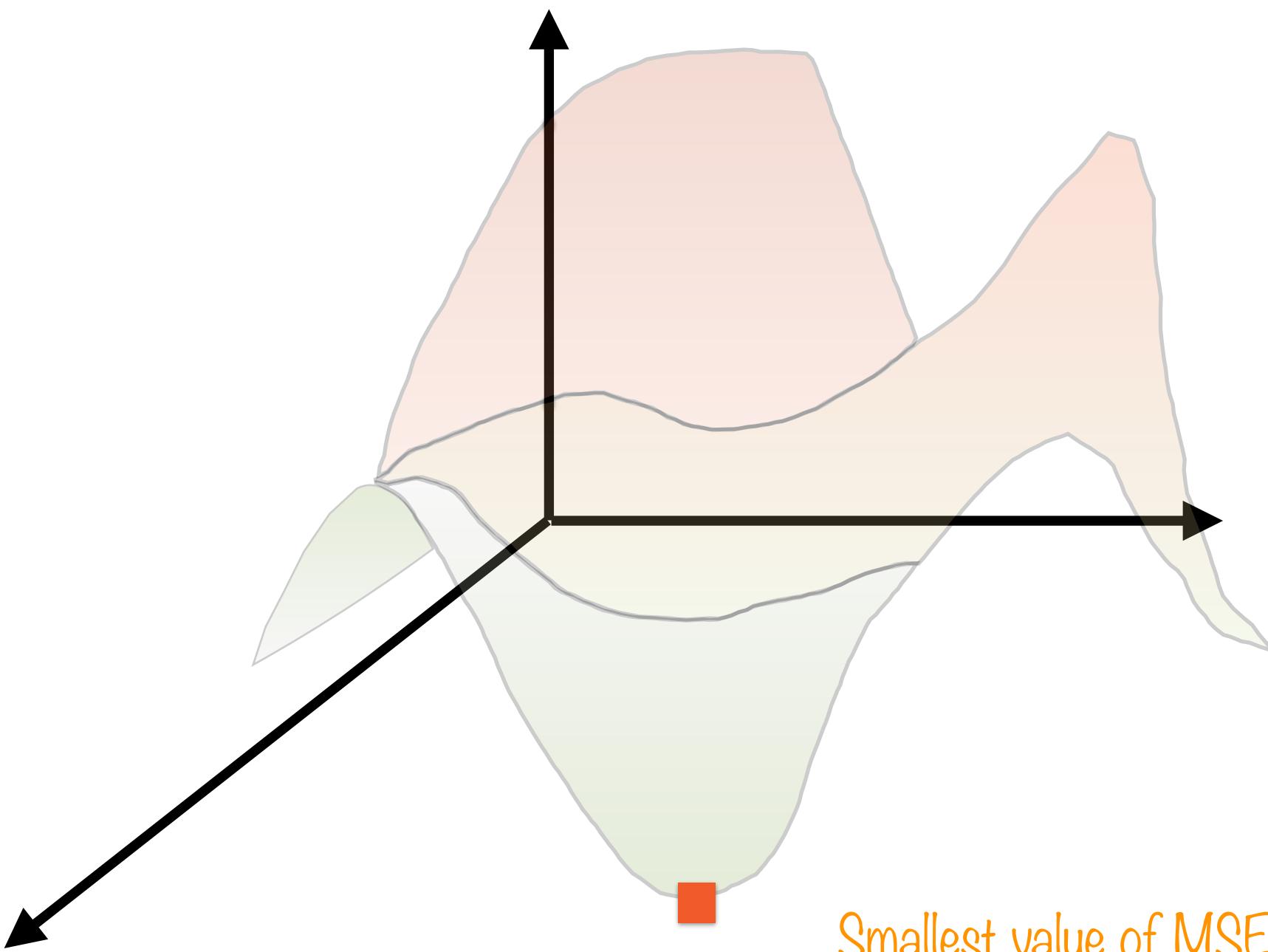
Minimizing MSE



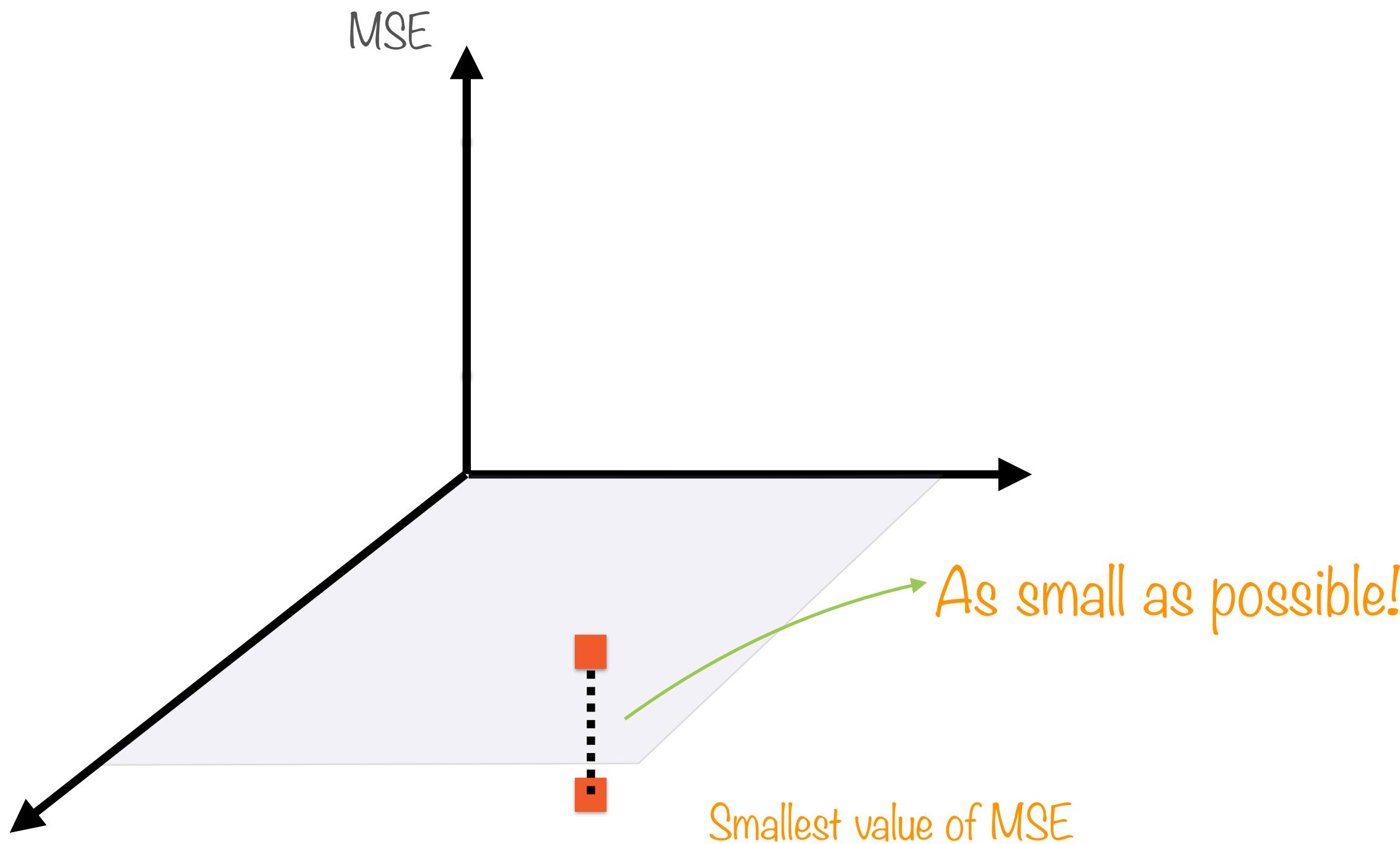
Minimizing MSE



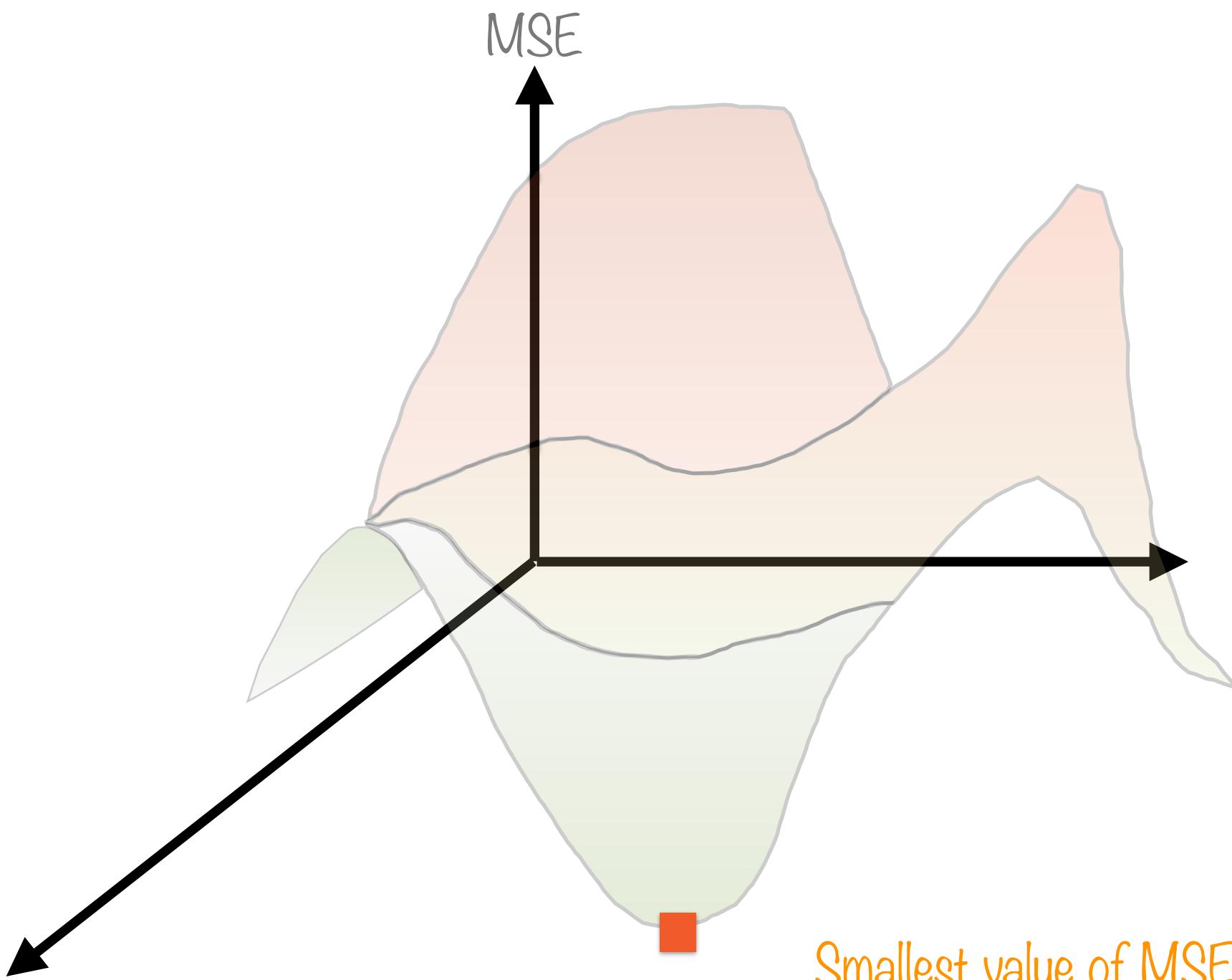
Minimizing MSE



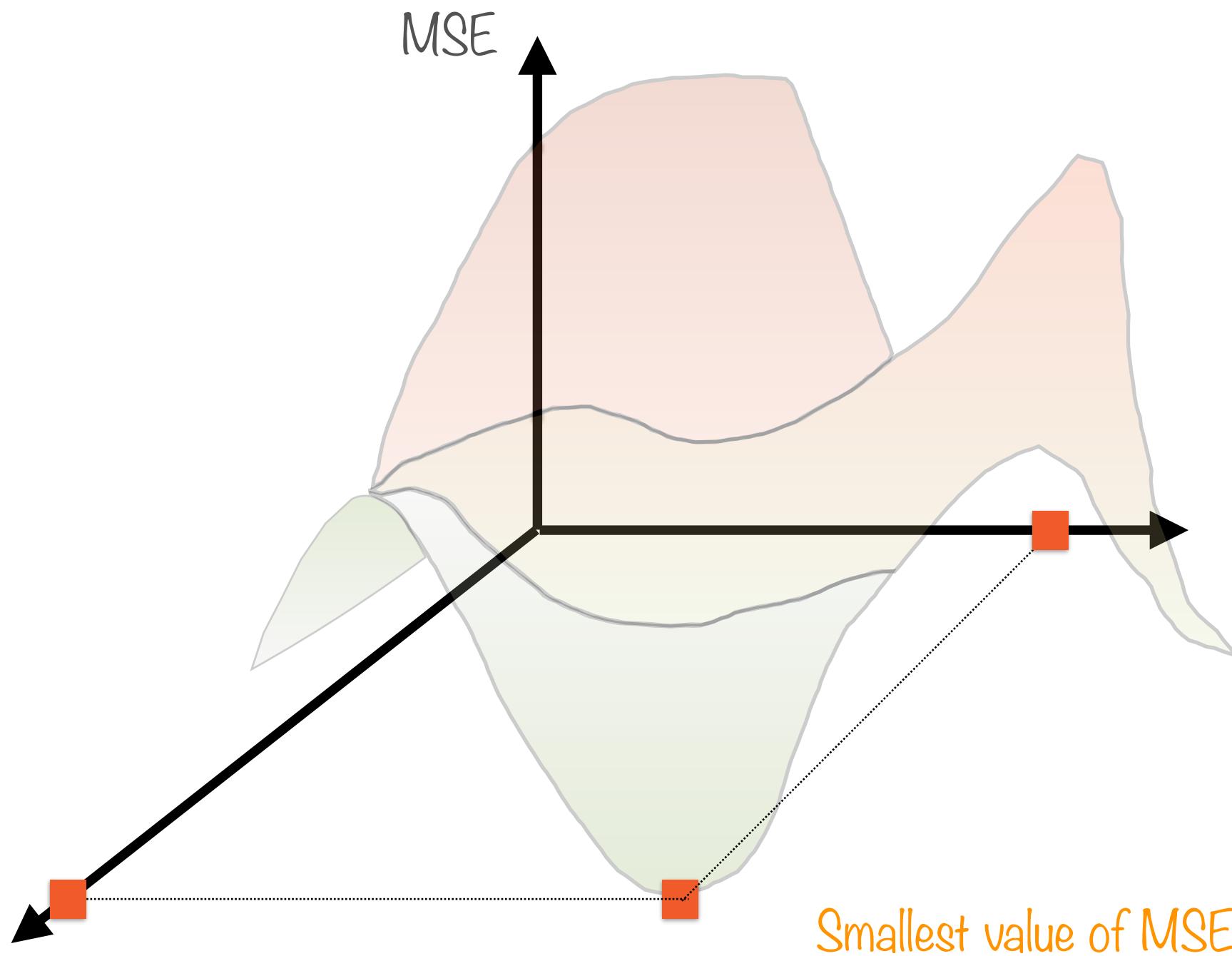
Minimizing MSE



Minimizing MSE

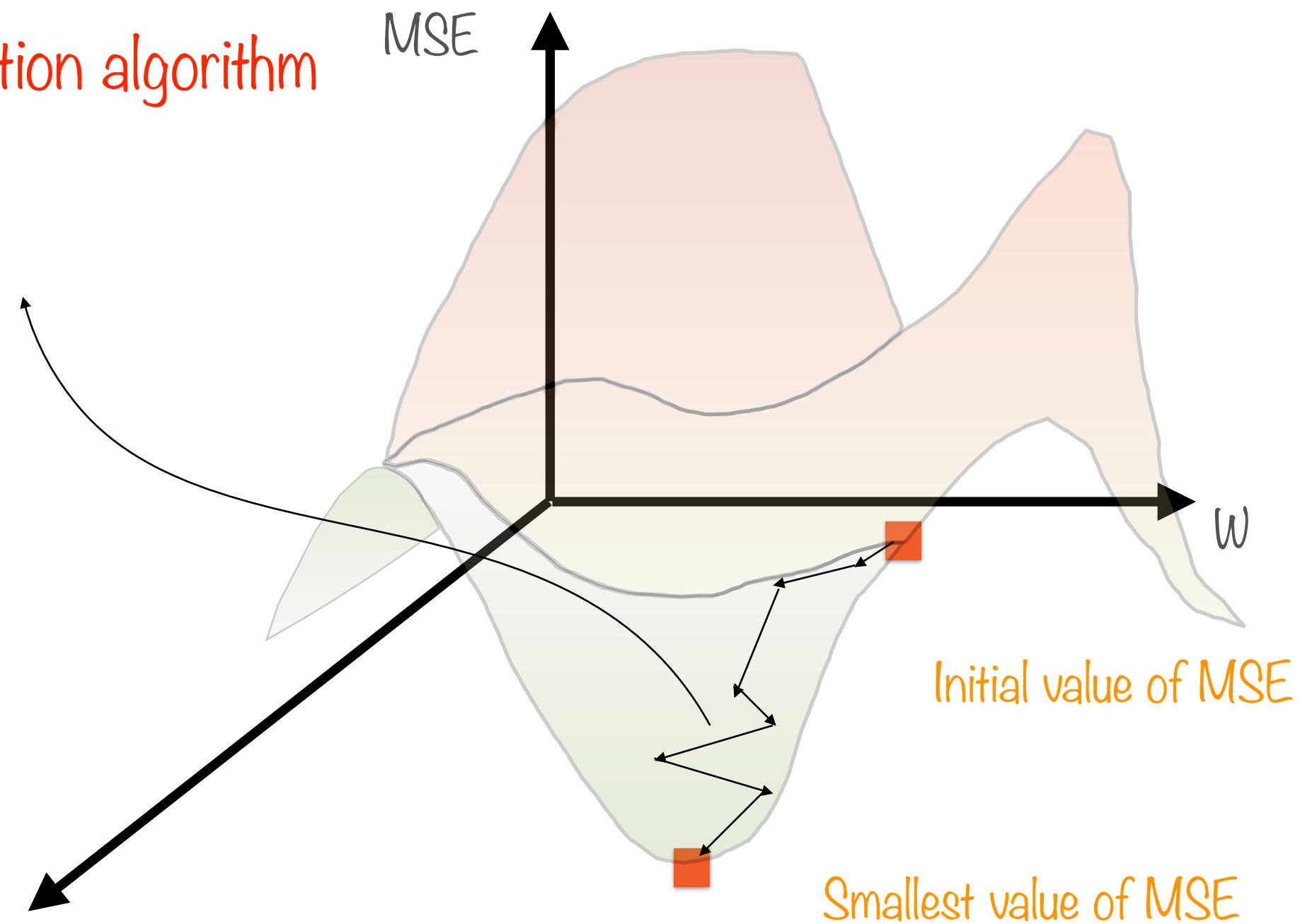


Minimizing MSE

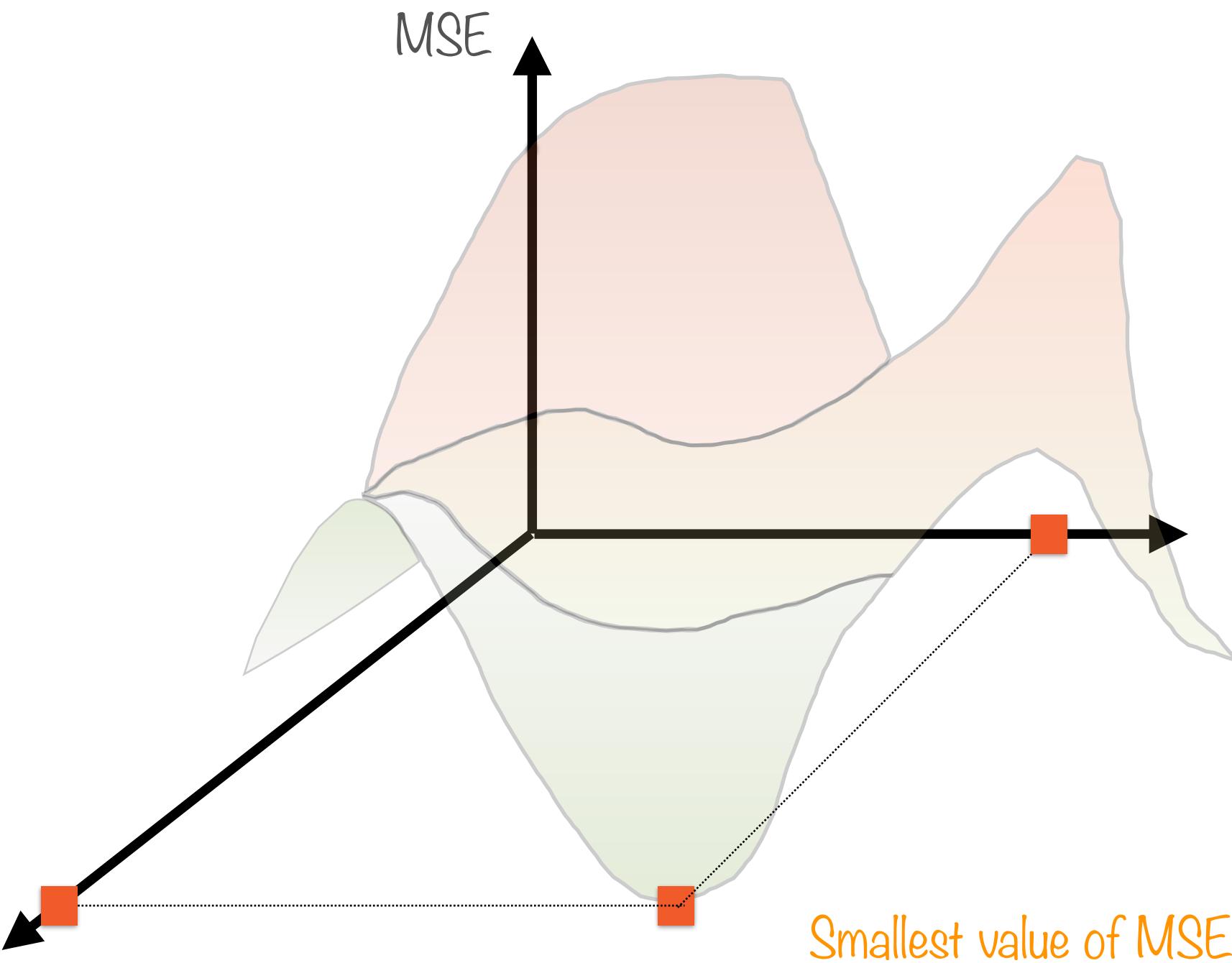


“Gradient Descent”

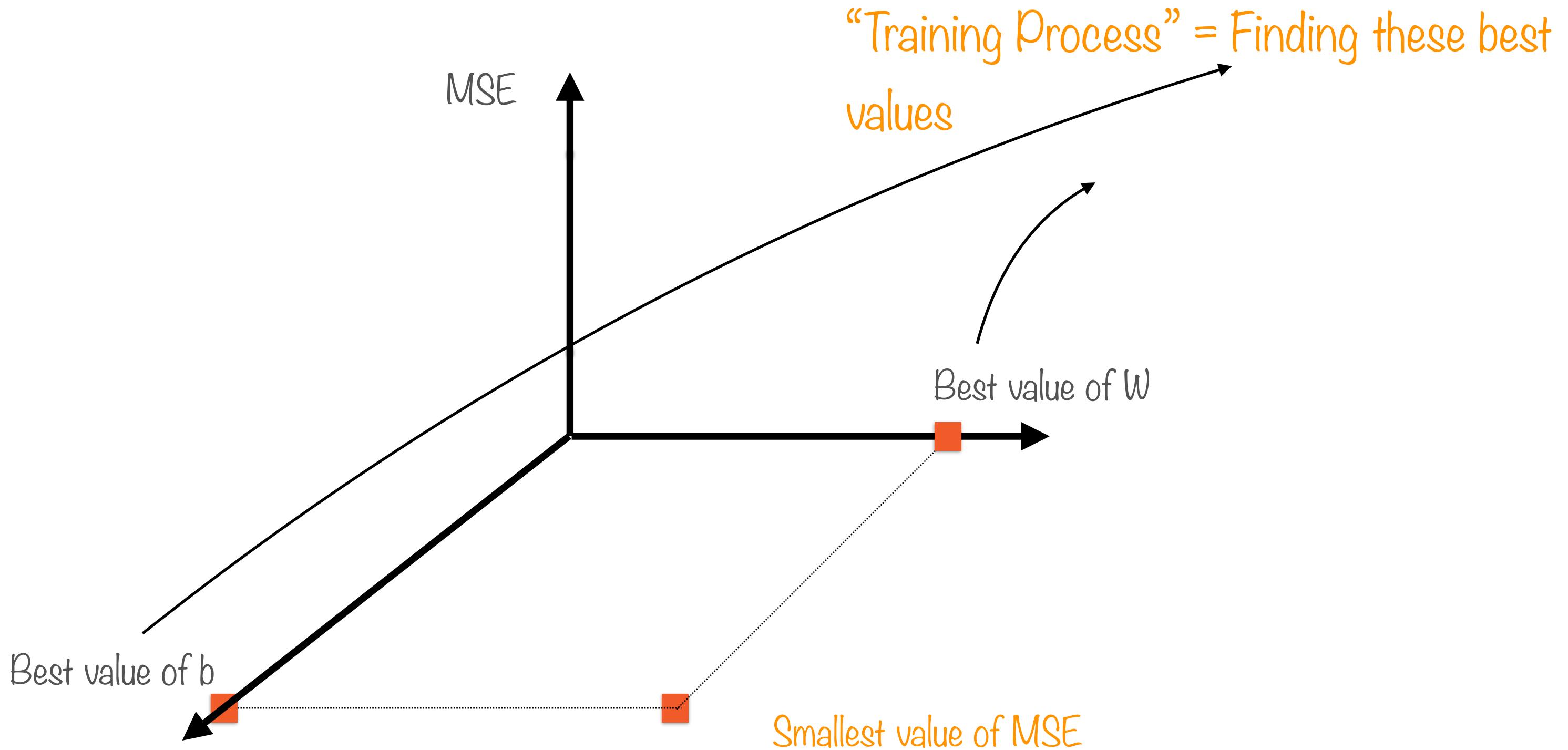
Converging on the “best” value
using an optimization algorithm



Minimizing MSE

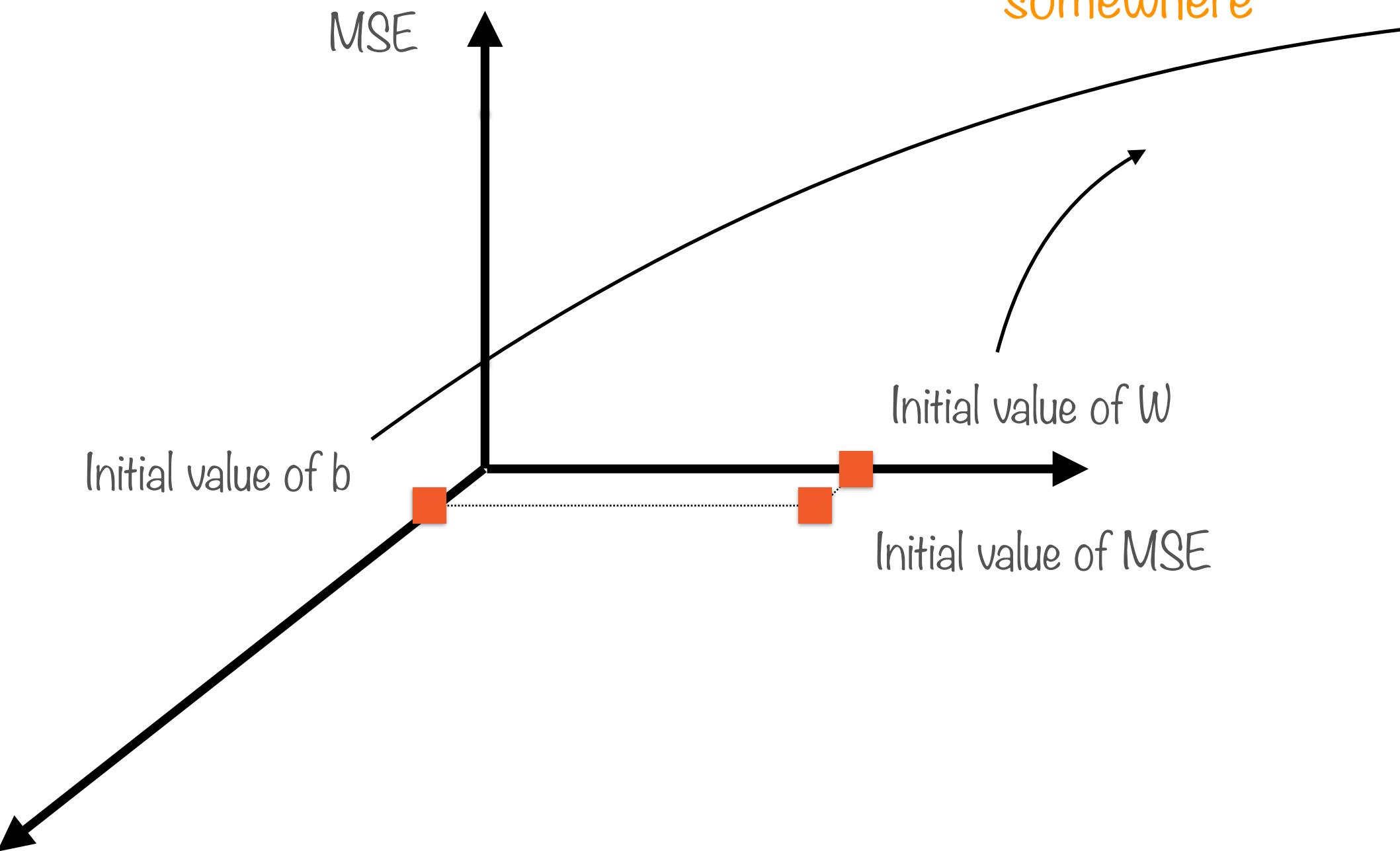


“Training” the Algorithm



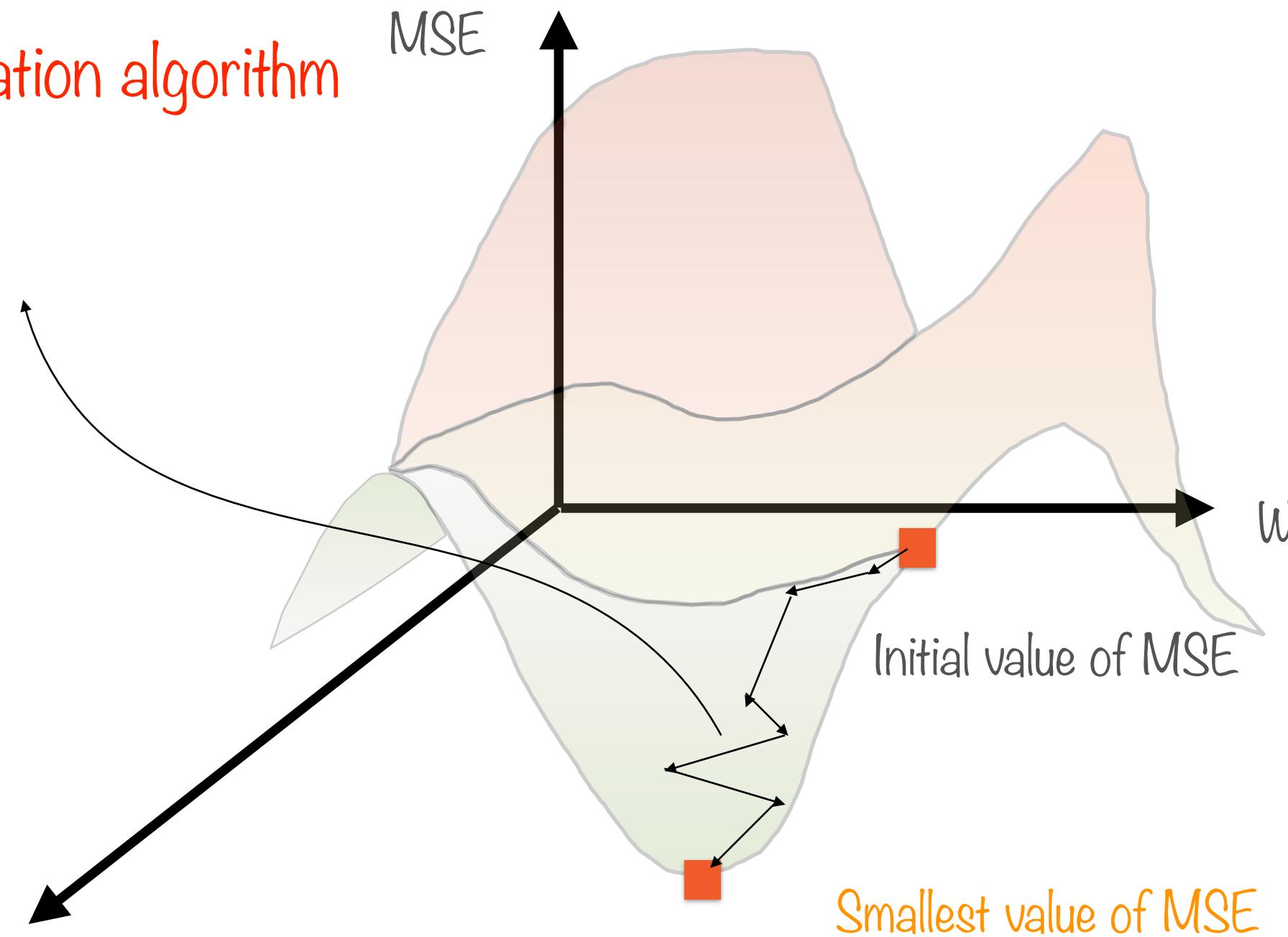
Start Somewhere

Initial values - have to start somewhere



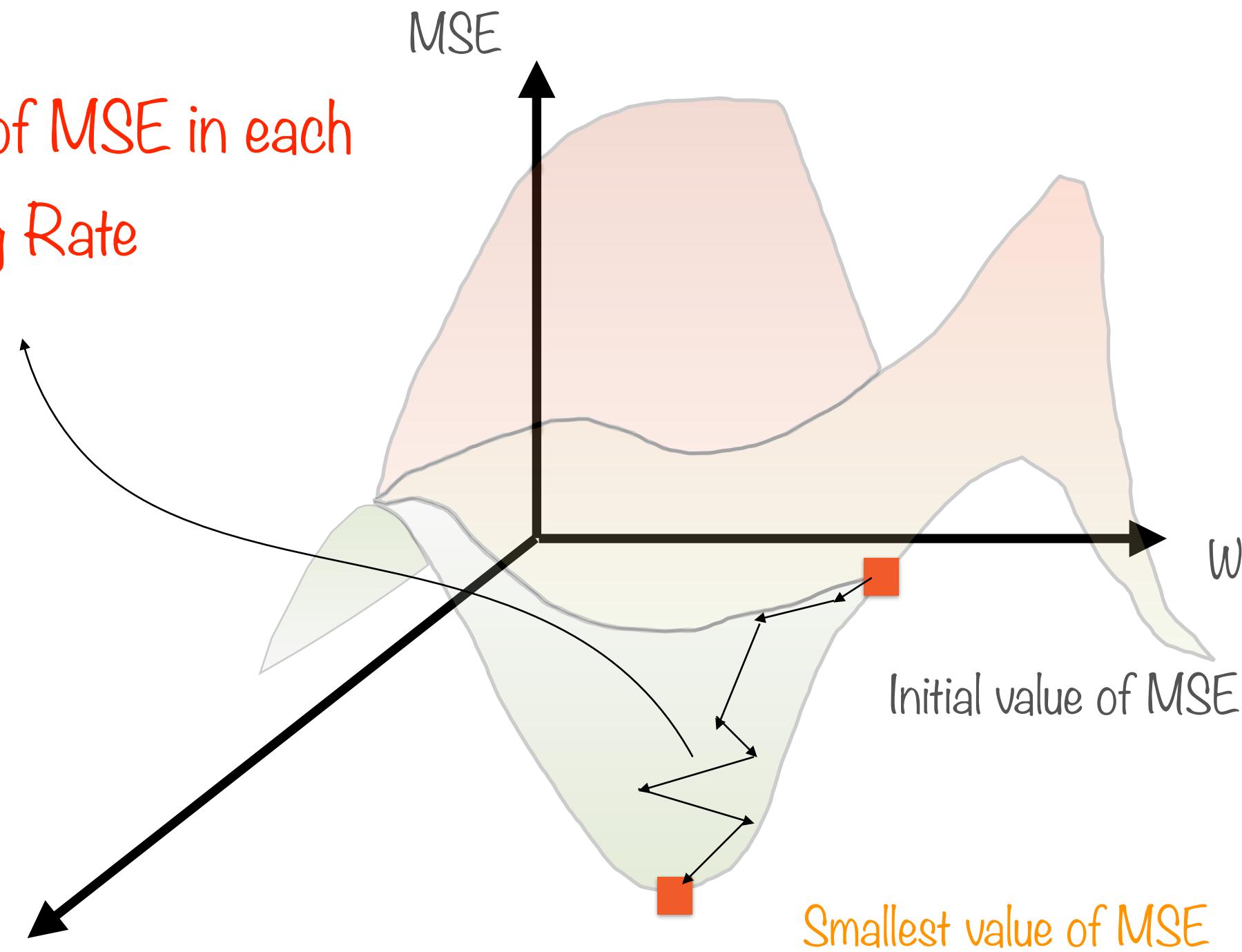
“Gradient Descent”

Converging on the “best” value
using an optimization algorithm



“Learning Rate”

Change in value of MSE in each epoch = Learning Rate



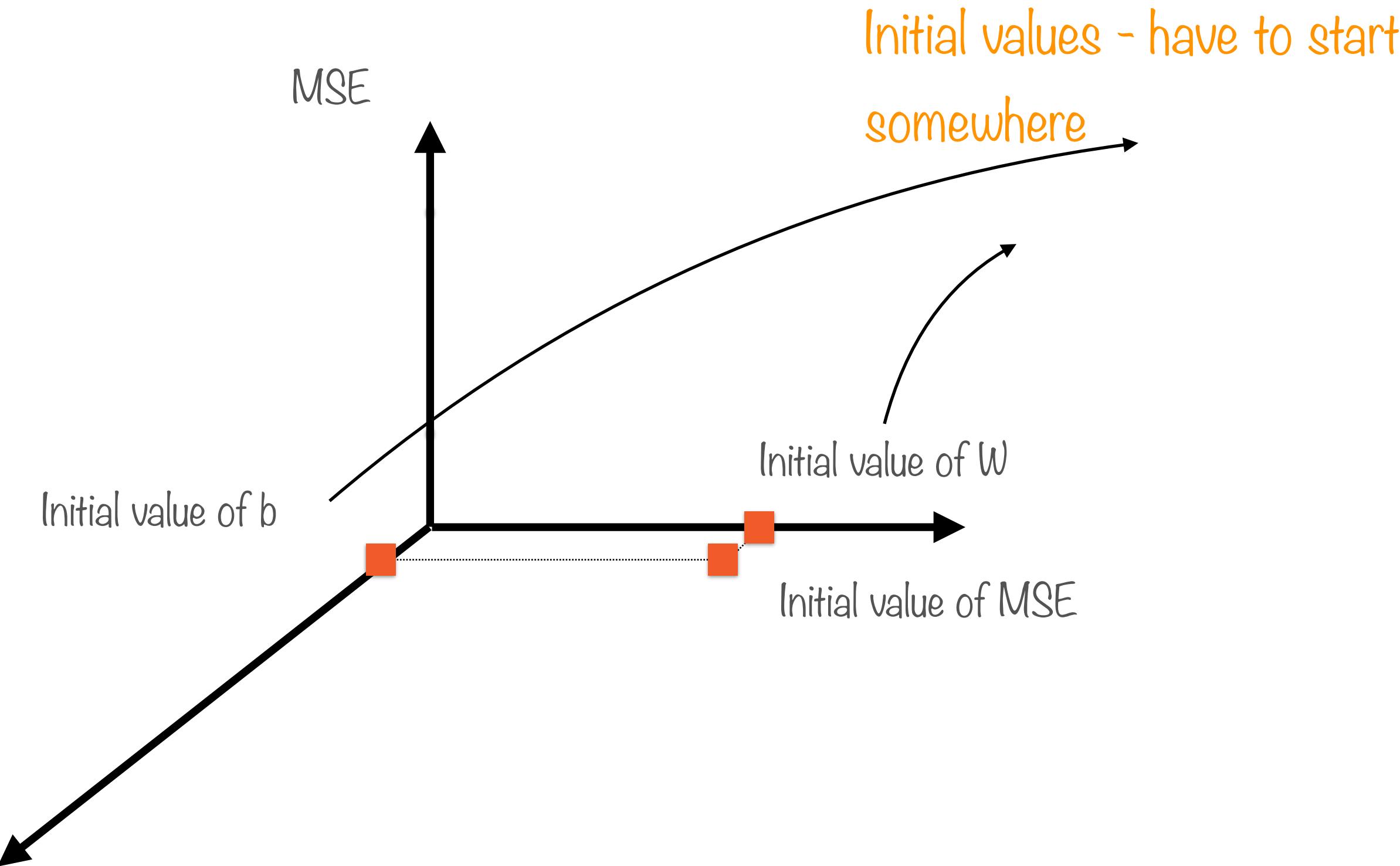
Gradient Descent Optimizers

`tf.train.GradientDescentOptimizer`

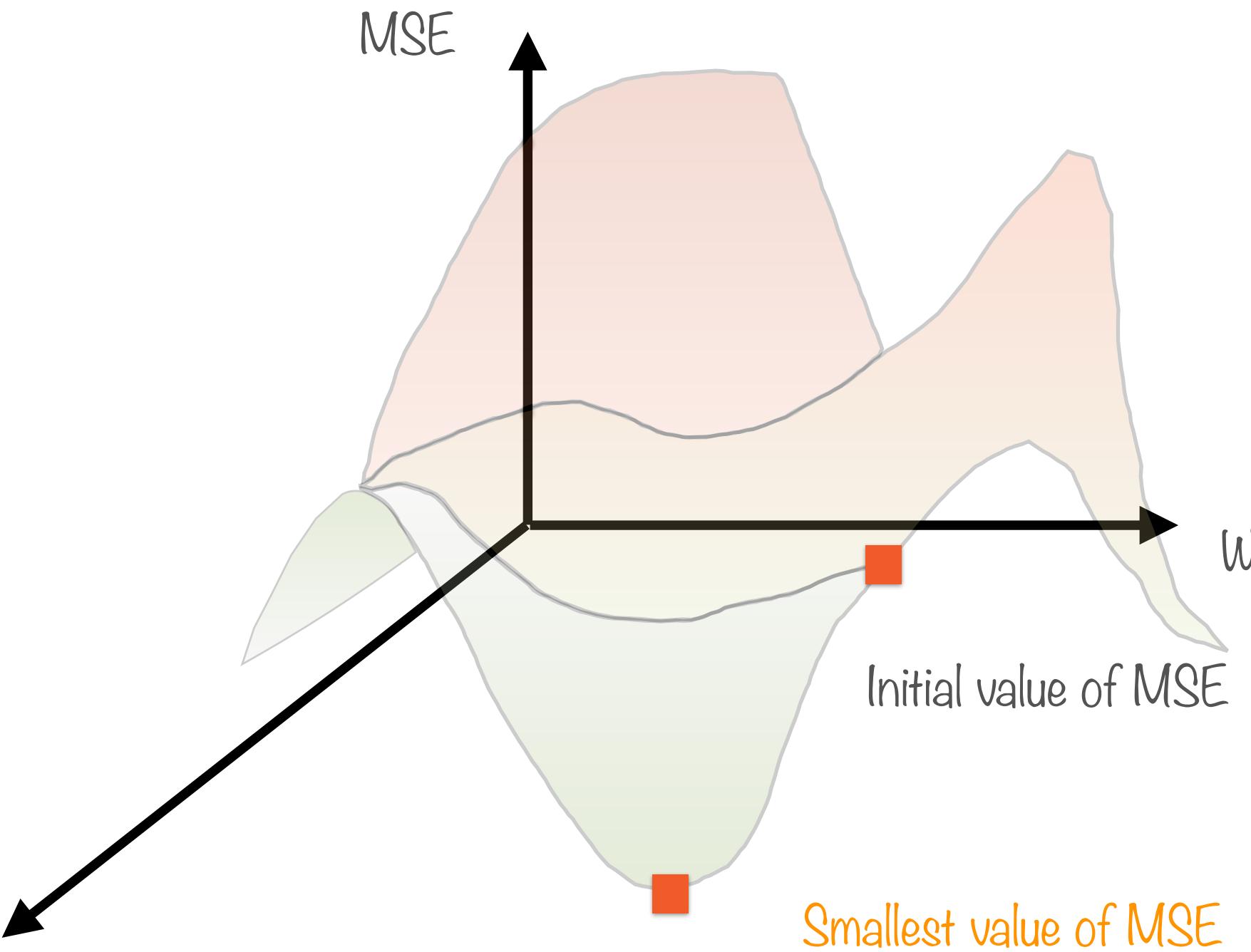
`tf.train.AdamOptimizer`

`tf.train.FtrlOptimizer`

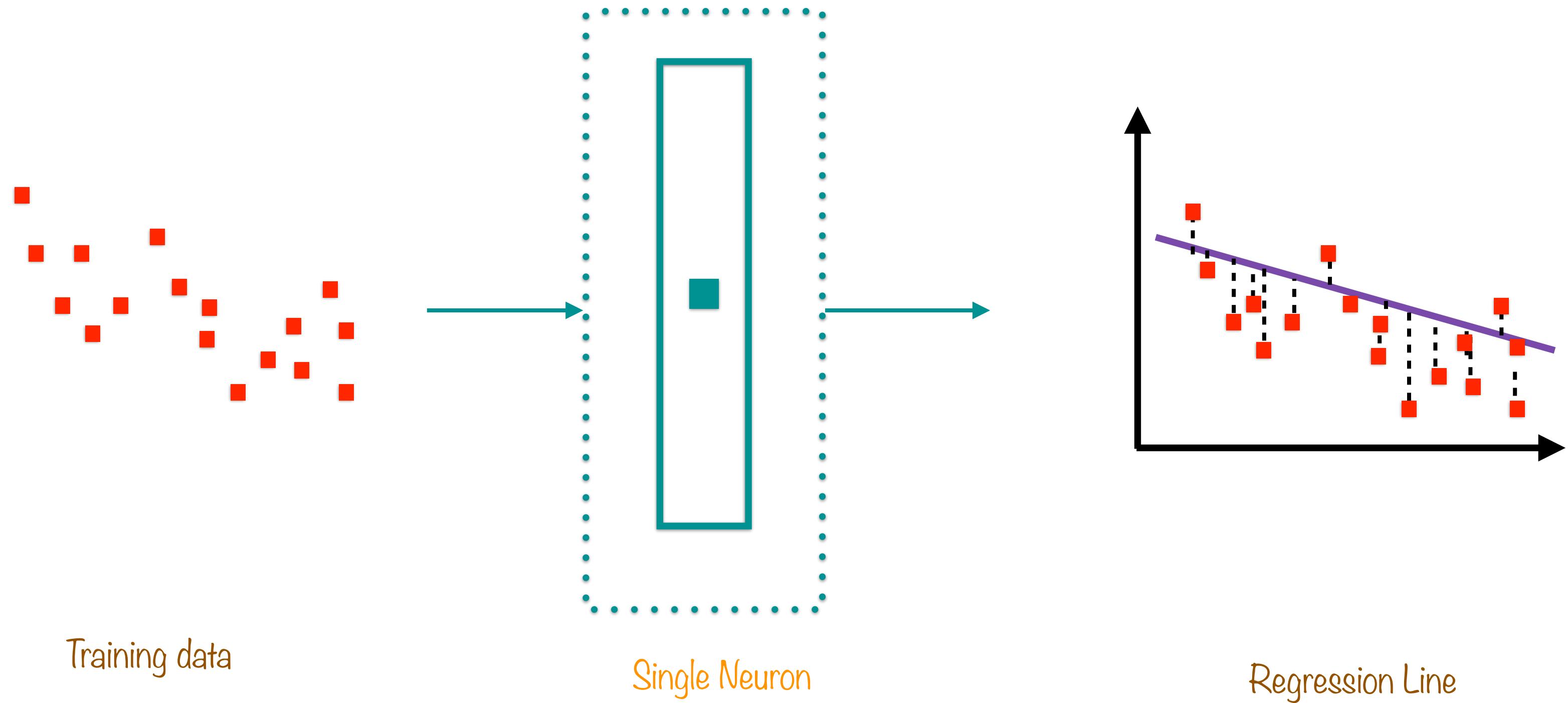
Start Somewhere



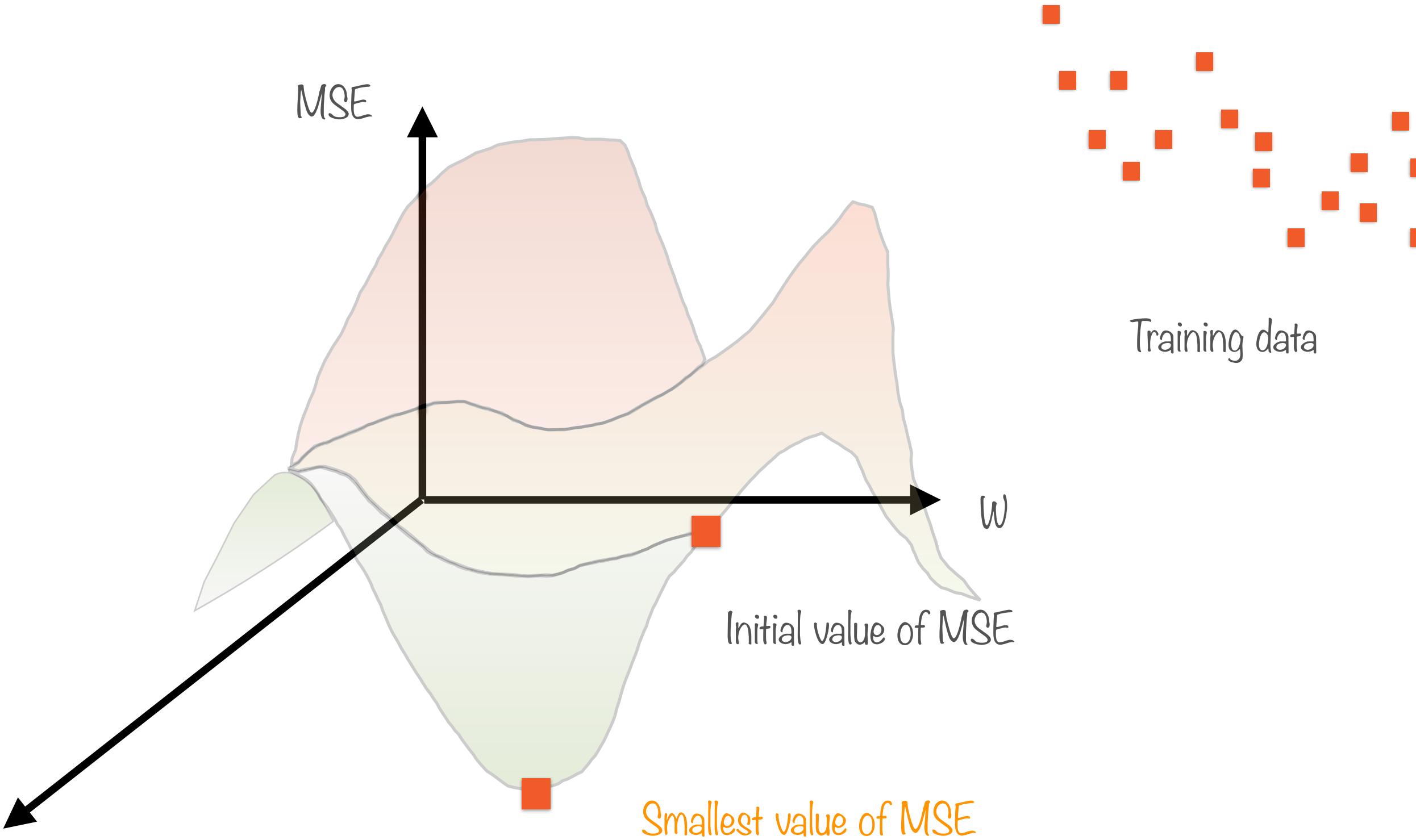
Minimizing MSE



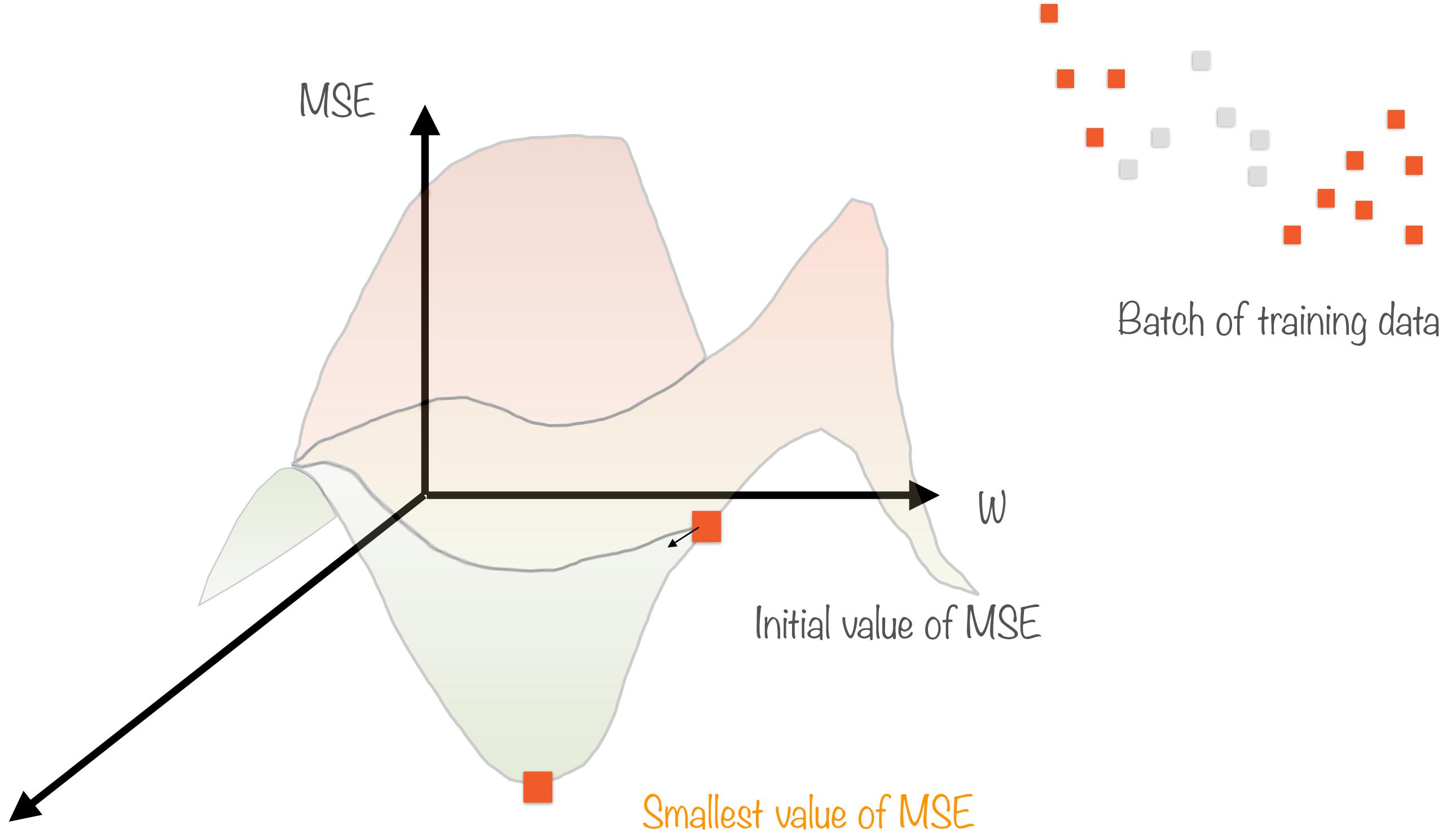
Regression: The Simplest Neural Network



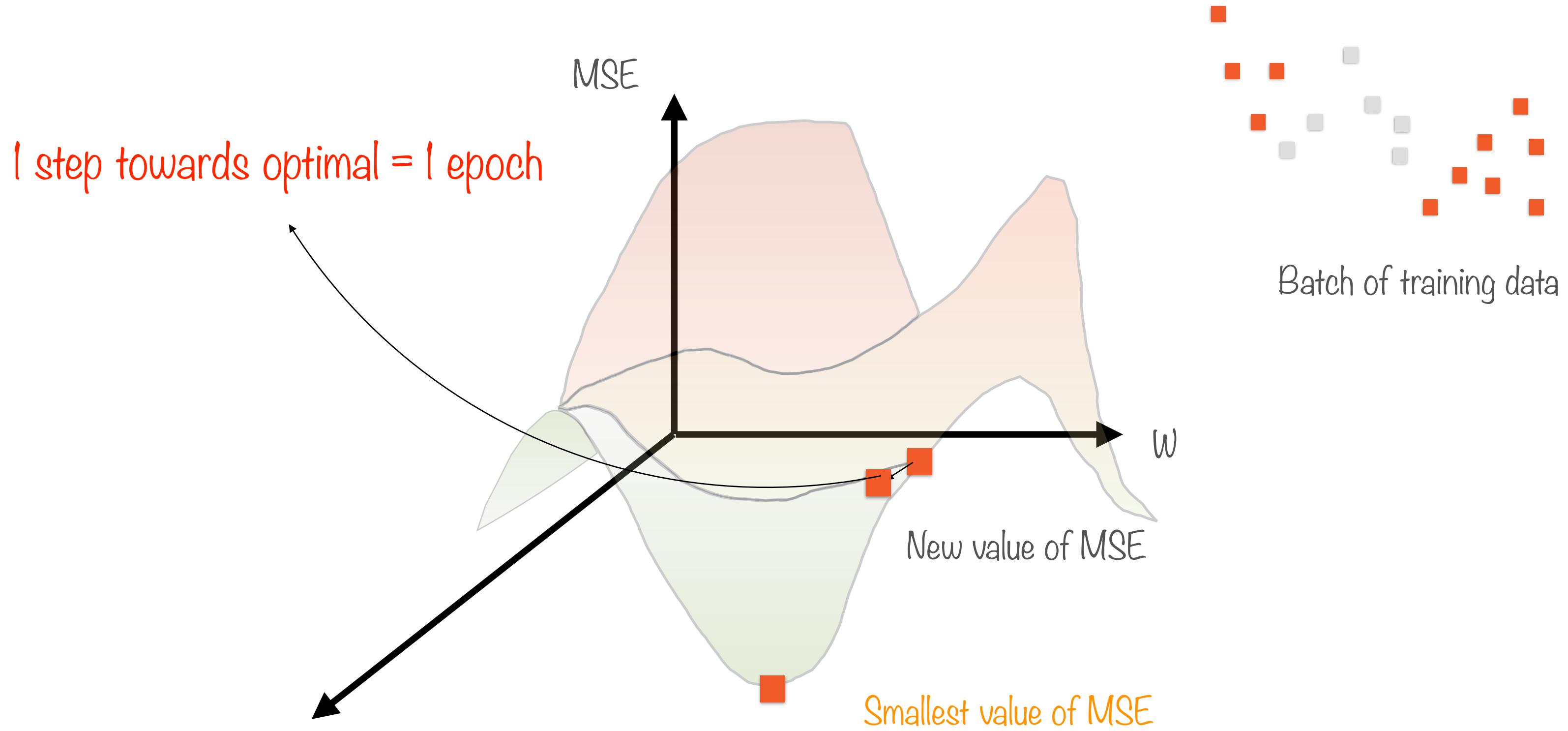
Minimizing MSE



Minimizing MSE

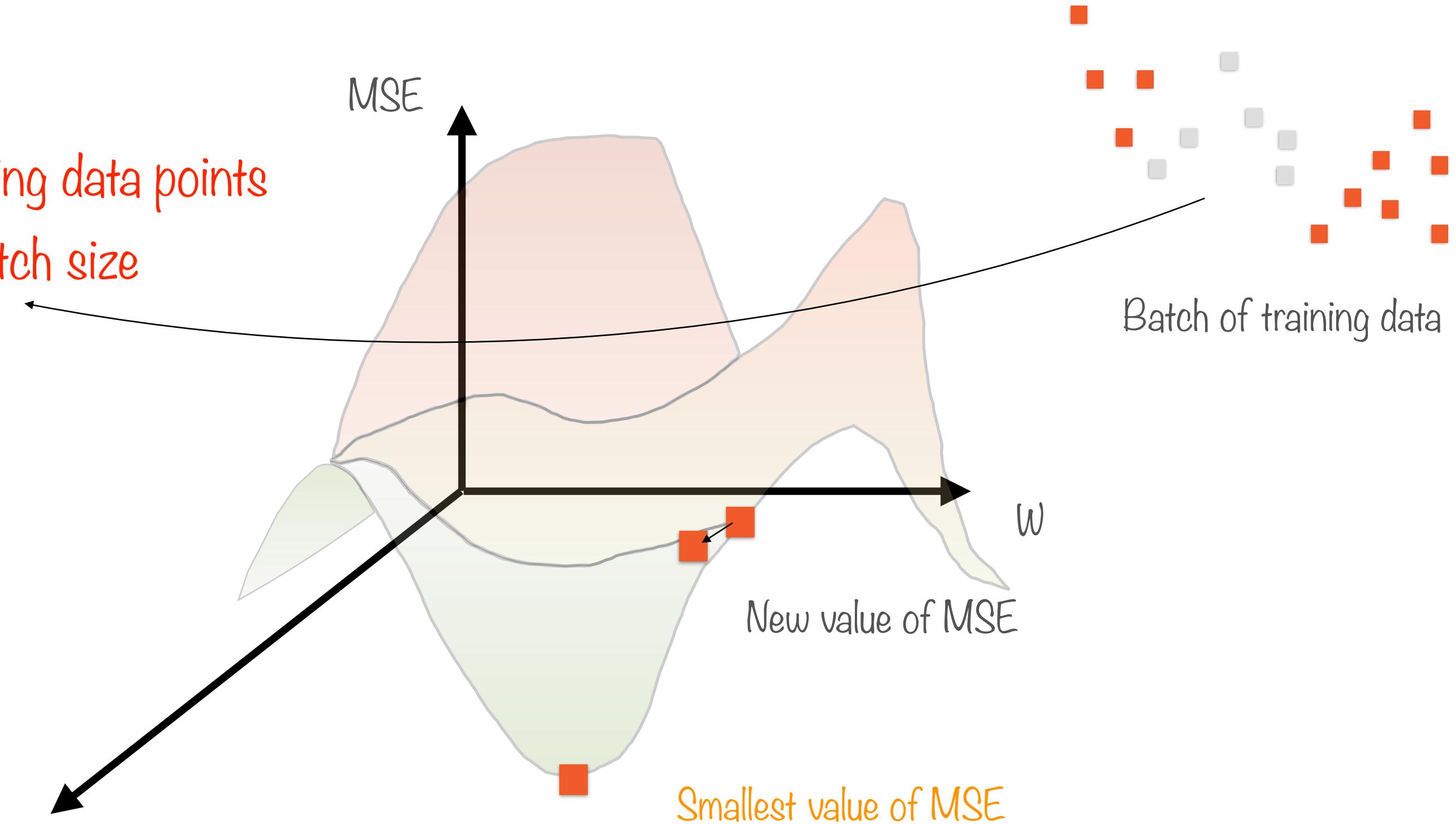


“Epoch”

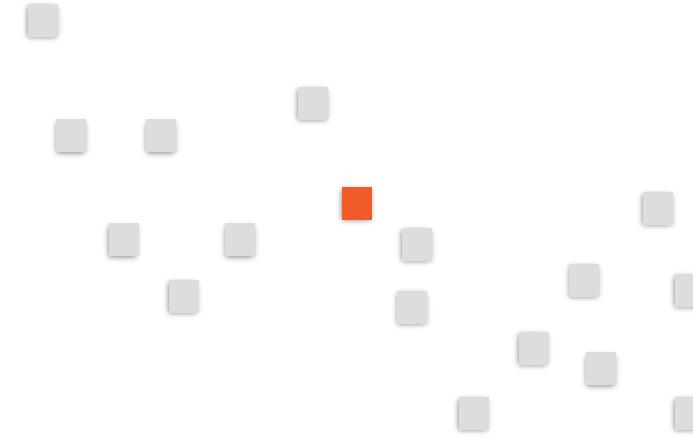


“Batch Size”

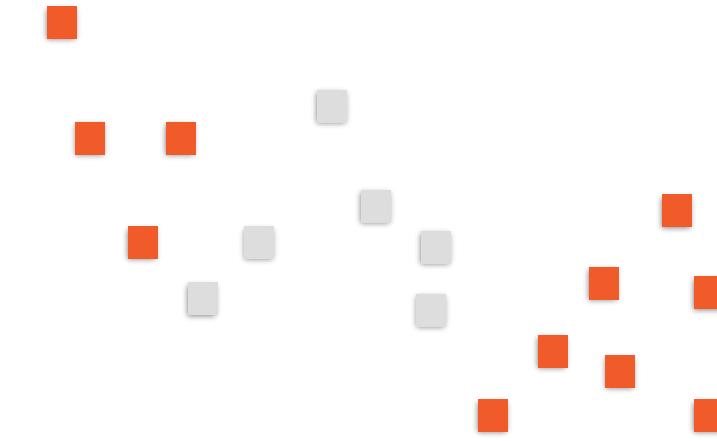
Number of training data points
considered = batch size



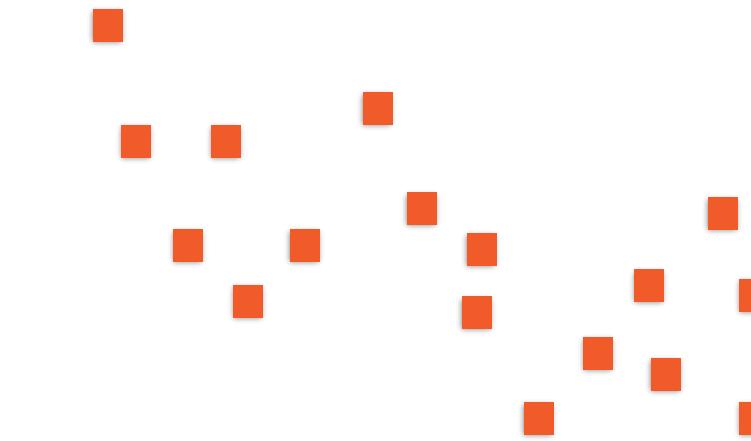
“Batch Size”



Stochastic Gradient
Descent
1 point at a time



Mini-batch Gradient
Descent
Some subset in each batch



Batch Gradient Descent
All training data in each batch

Implementing Regression in TensorFlow

Baseline
Non-TensorFlow implementation
Regular python code

Cost Function
Mean Square Error (MSE)
Quantifying goodness-of-fit

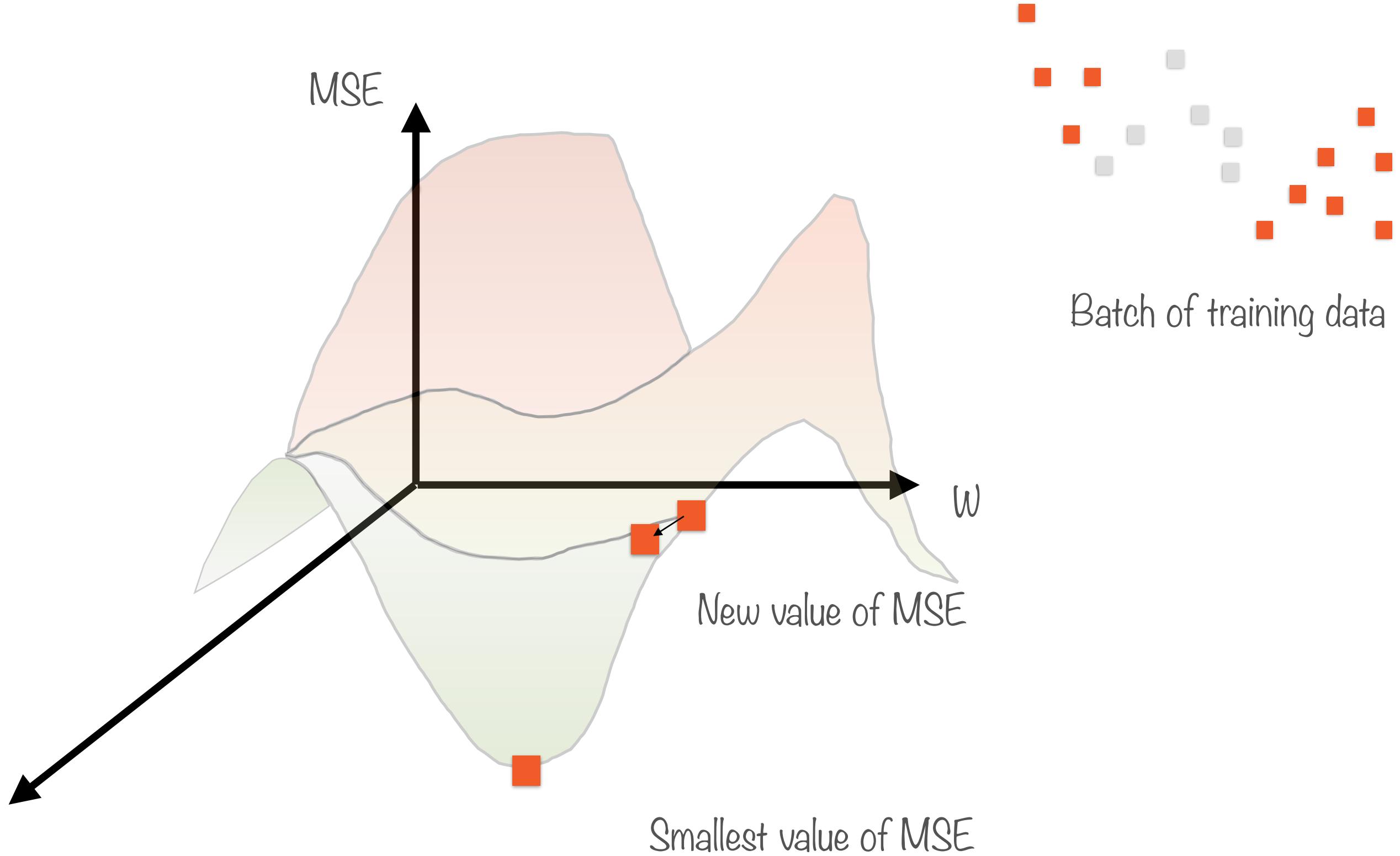
Training
Invoke optimizer in epochs
Batch size for each epoch



Computation Graph
Neural network of 1 neuron
Affine transformation suffices

Optimizer
Gradient Descent optimizers
Improving goodness-of-fit

Minimizing MSE



Decisions in Training

Initial values

Type of optimizer

Number of epochs

Batch size

Simple Regression



Cause

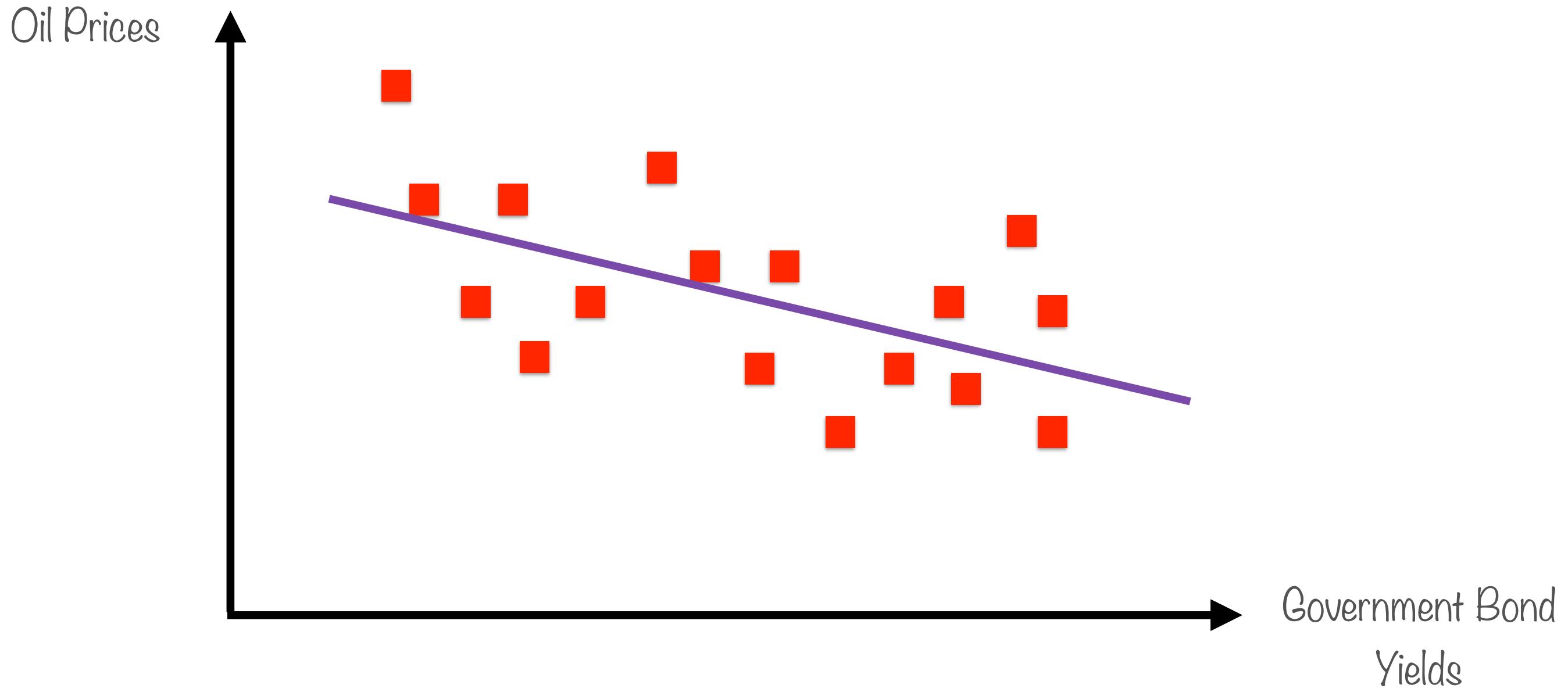
Independent variable



Effect

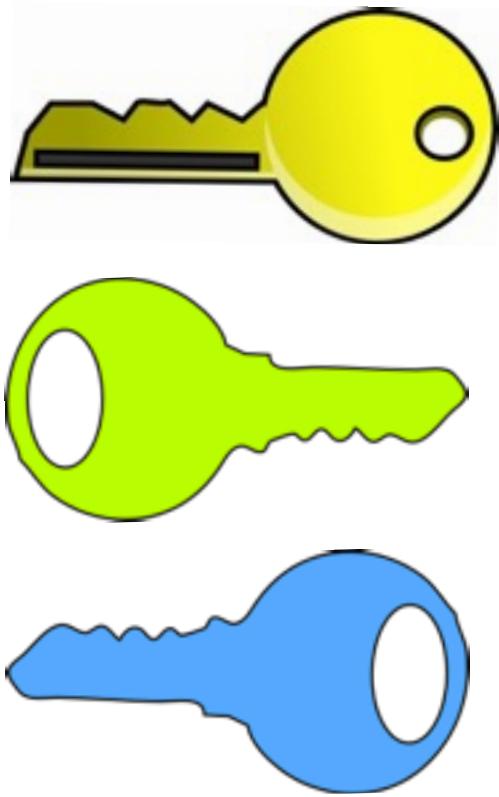
Dependent variable

Simple Regression



One cause, one effect

Multiple Regression



Causes

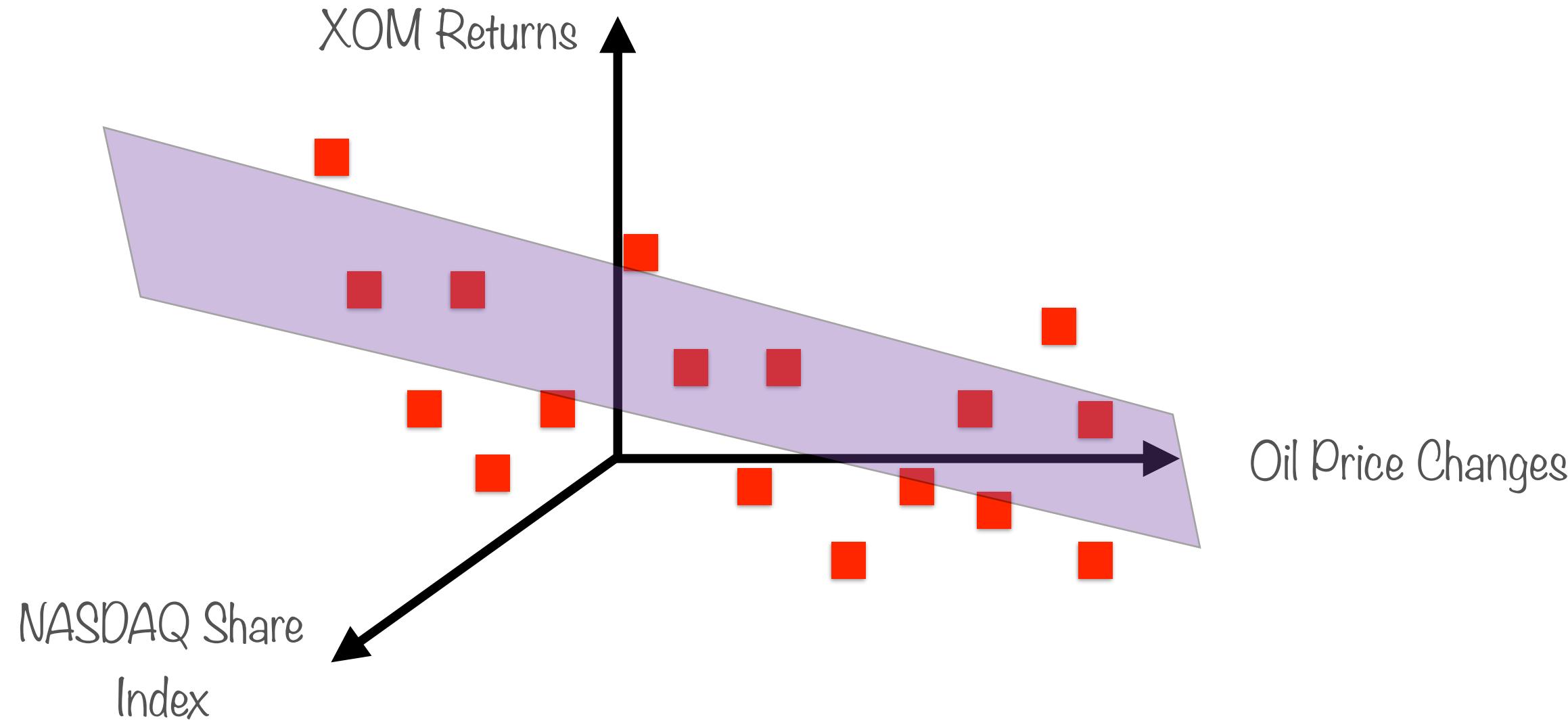
Independent variables



Effect

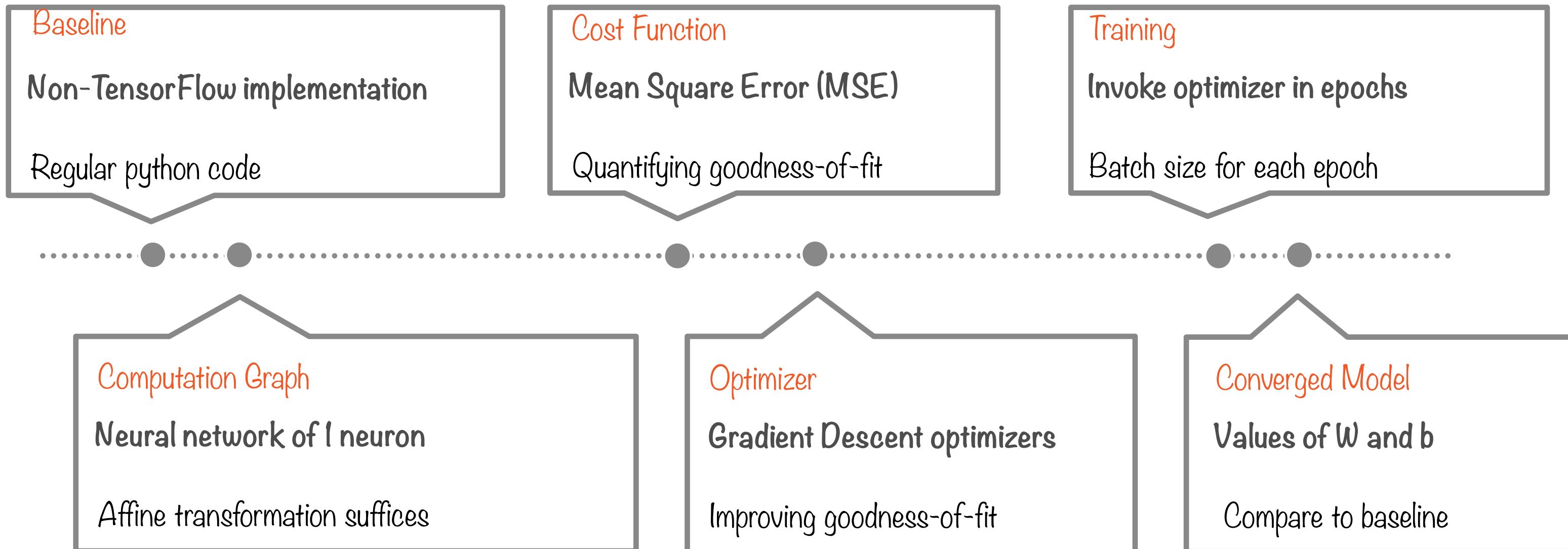
Dependent variable

Multiple Regression



Many causes, one effect

Implementing Regression in TensorFlow



Summary

- Implement linear regression without using TensorFlow
- Define computation graph of just one neuron
- Set up the cost function
- Use various gradient descent optimizers
- Understand gradient descent process
- Train the model to get a converged regression model

Building Logistic Regression Models Using TensorFlow

Overview

Given causes, predict probability of effects - that's logistic regression

Linear regression and logistic regression are similar, yet quite different

Logistic regression can be used for categorical y-variables

Logistic regression in TensorFlow differs from linear regression in two ways

- Softmax as the activation function
- cross-entropy as the cost function

Two Approaches to Deadlines



Start 5 minutes before deadline

Good luck with that



Start 1 year before deadline

Maybe overkill

Neither approach is optimal

Starting a Year in Advance

Probability of meeting the deadline



100%

Probability of getting other important work done

0%

Starting Five Minutes in Advance

Probability of meeting the deadline

0%

Probability of getting other important work done



100%

The Goldilocks Solution

Work fast

Start very late and hope for the best

Work smart

Start as late as possible to be sure to make it

Work hard

Start very early and do little else

As usual, the middle path is best

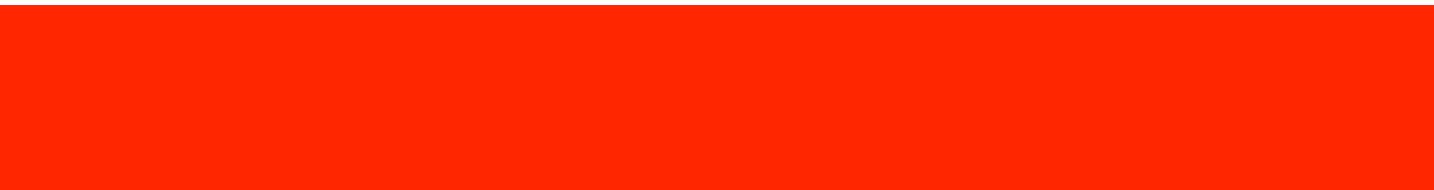
Working Smart

Probability of meeting the deadline



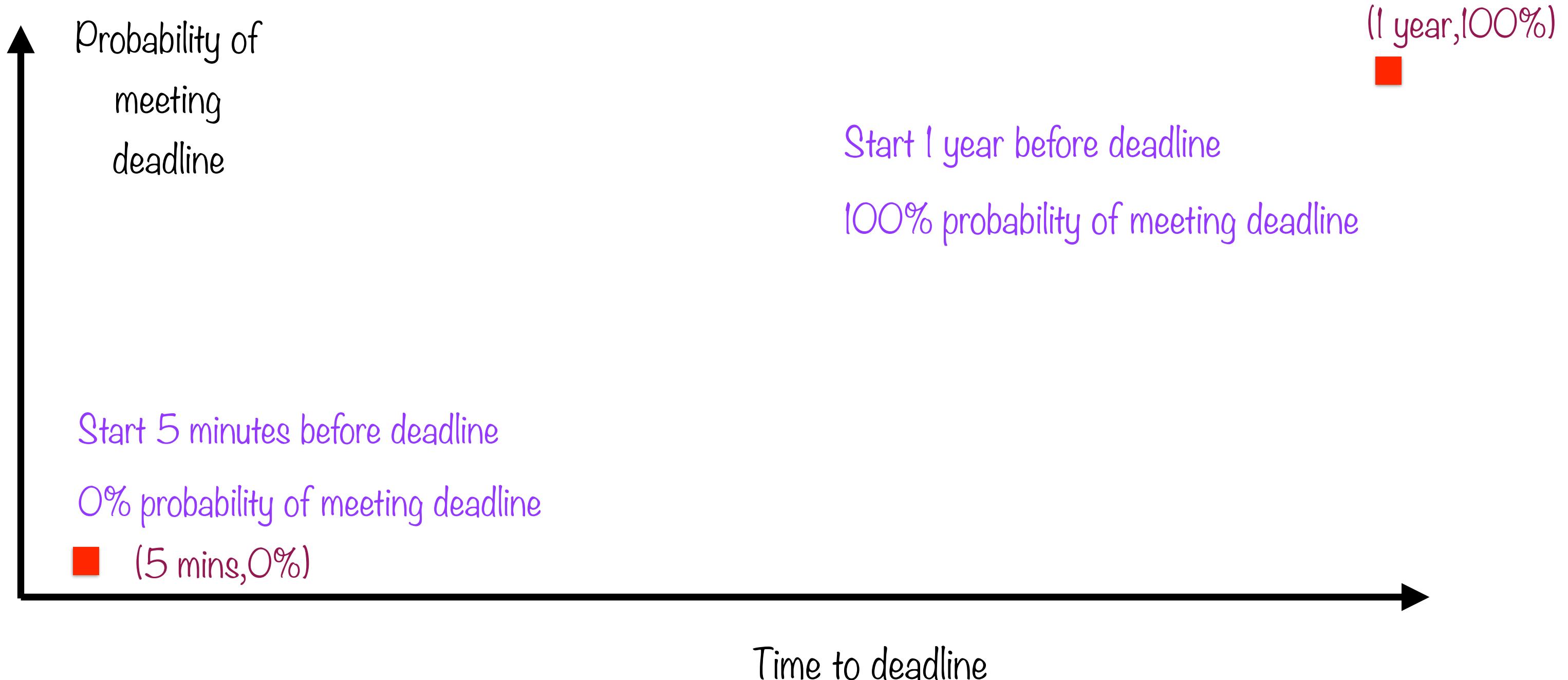
95%

Probability of getting other important work done



95%

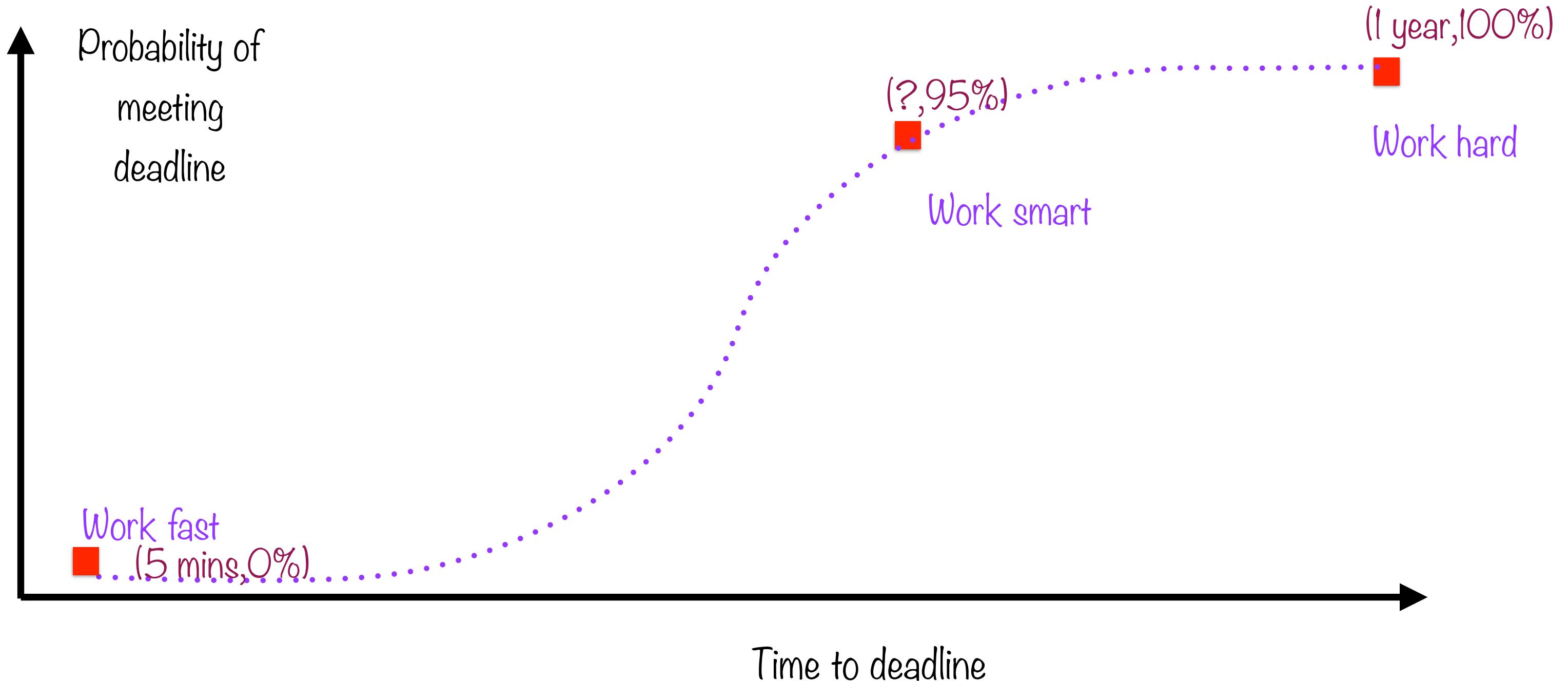
Working Hard, Fast, Smart



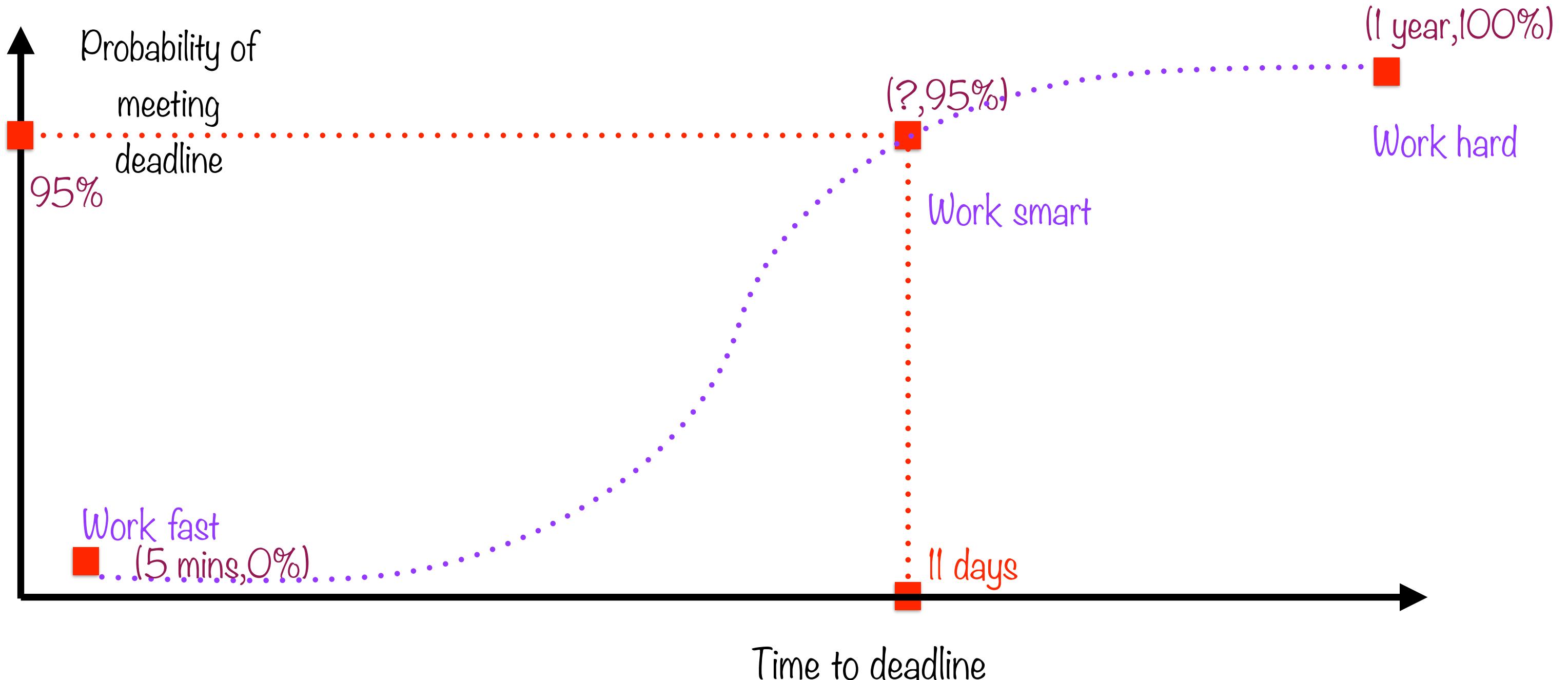
Working Hard, Fast, Smart



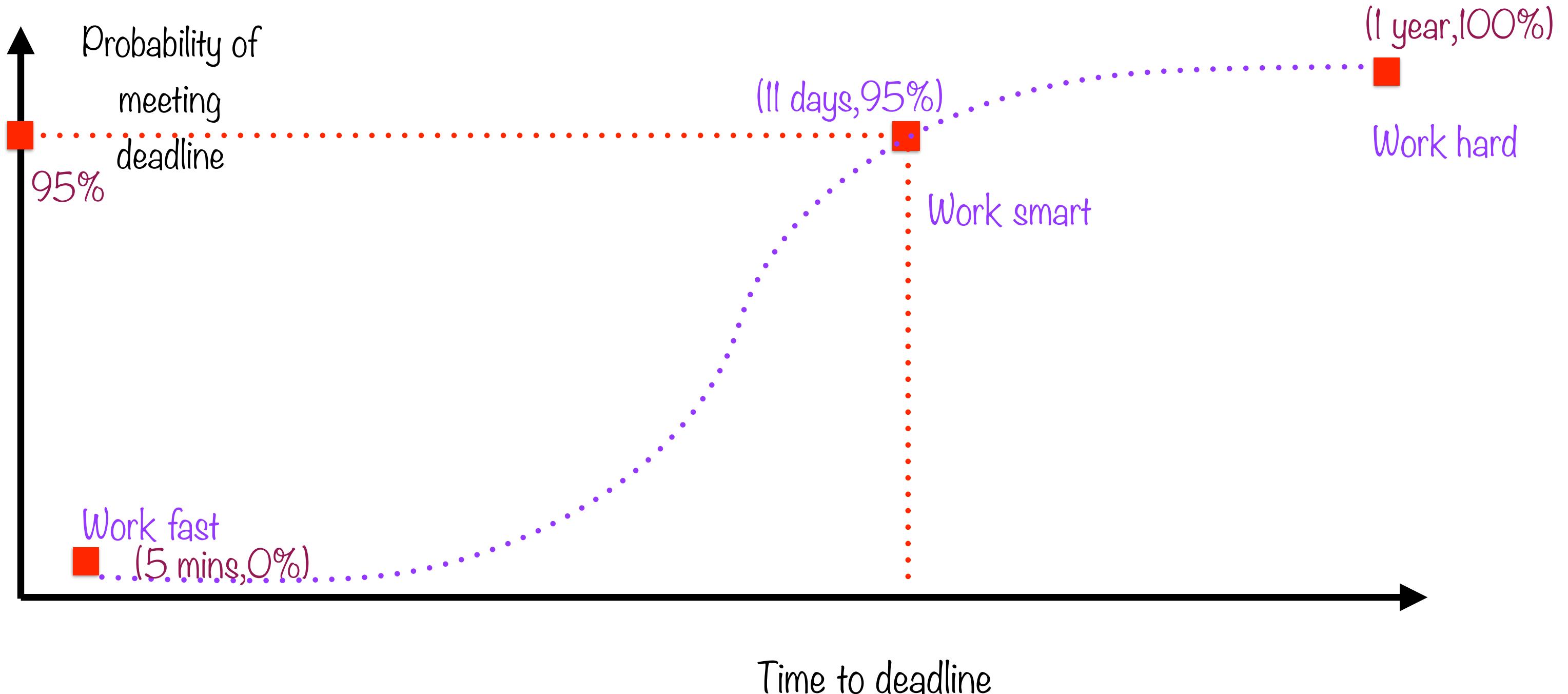
Working Hard, Fast, Smart



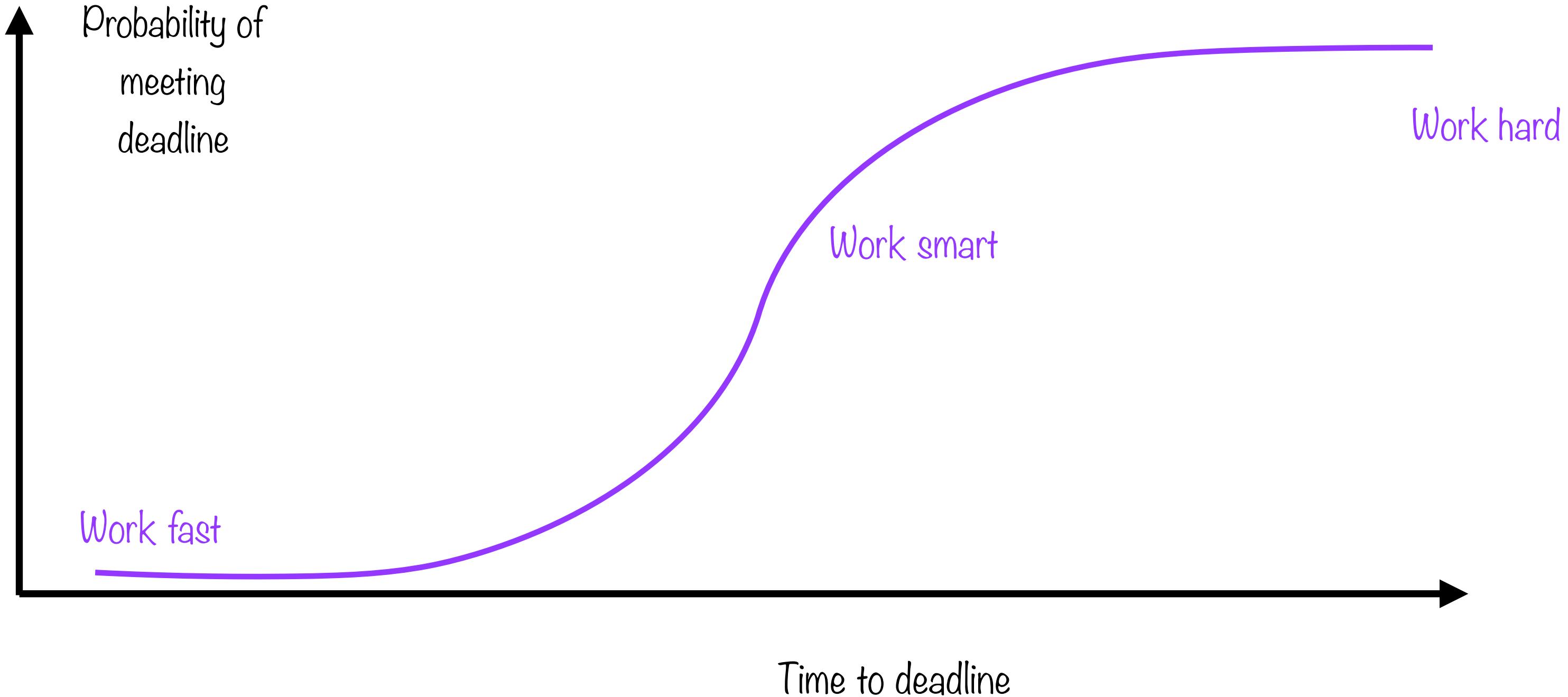
Working Hard, Fast, Smart



Working Hard, Fast, Smart

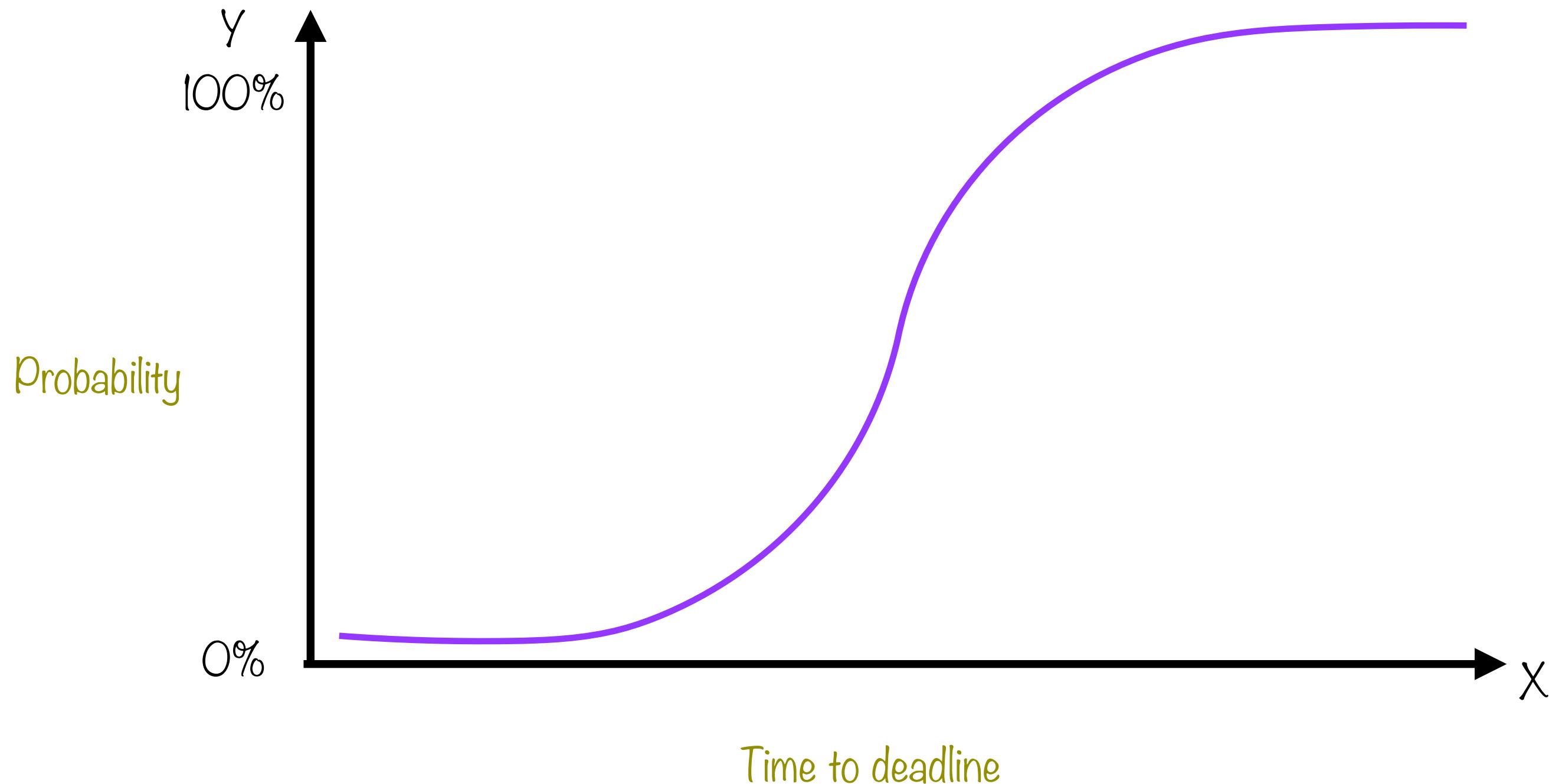


Working Hard, Fast, Smart

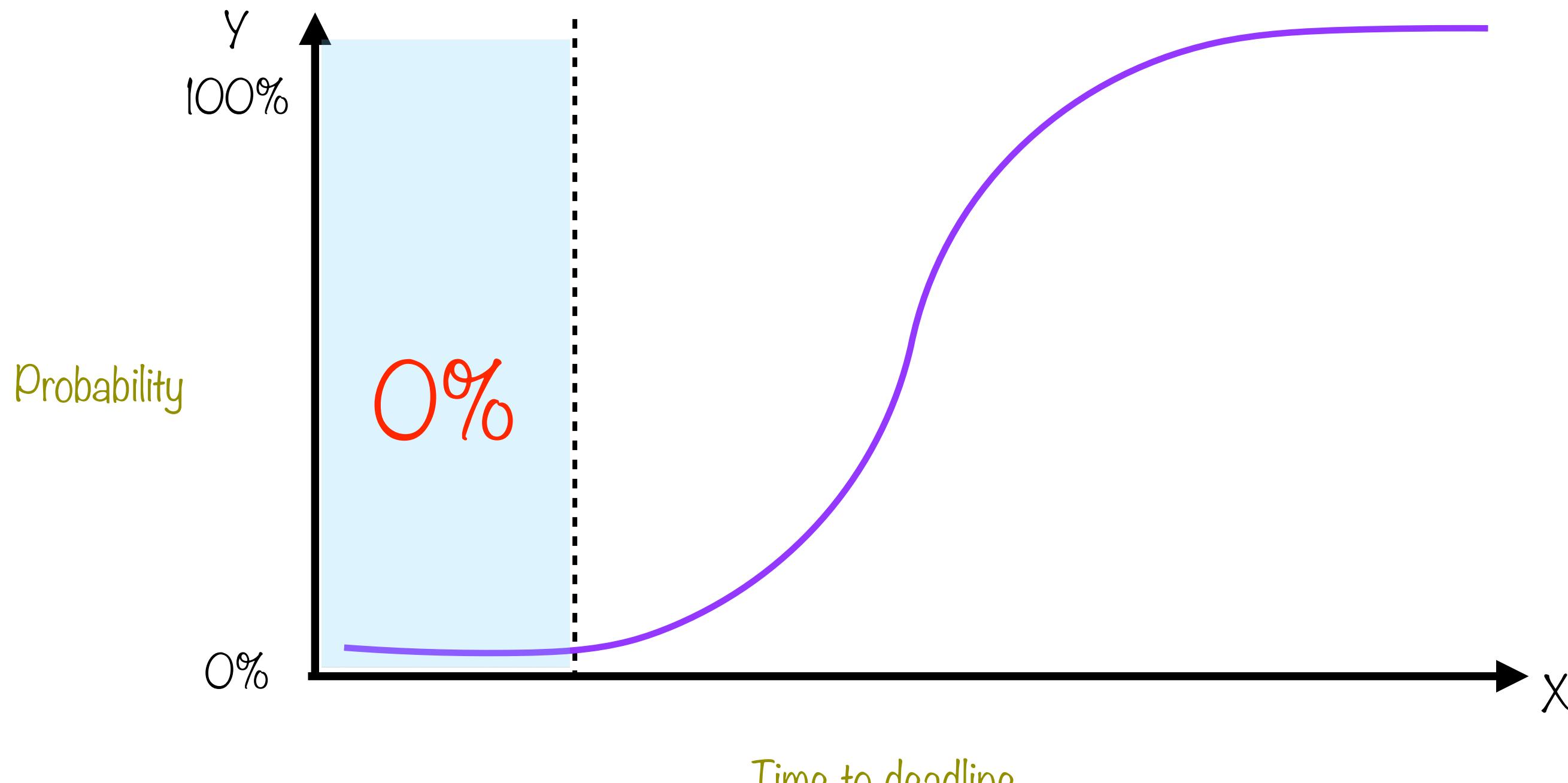


Logistic Regression helps find how probabilities
are changed by actions

Working Smart with Logistic Regression

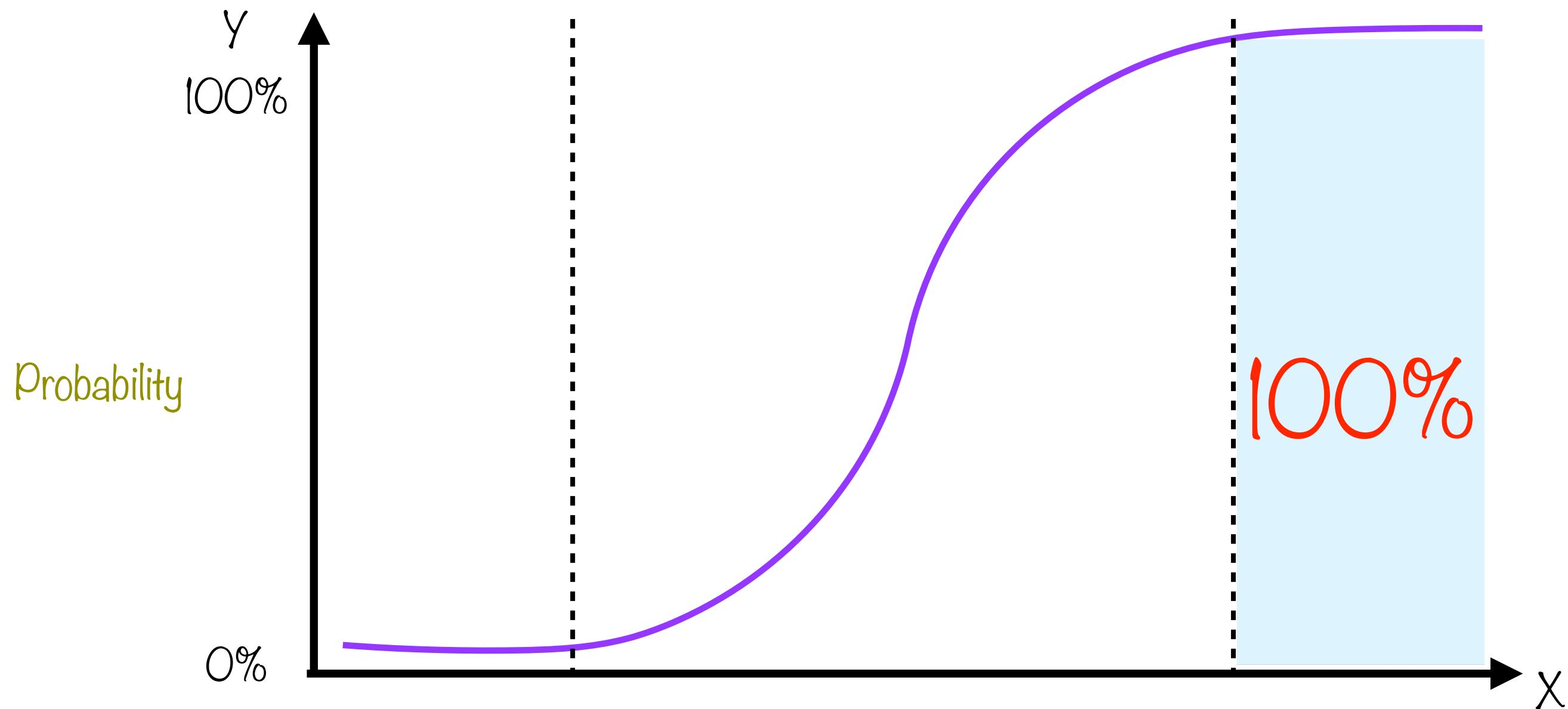


Working Smart with Logistic Regression



Start too late, and you'll definitely miss

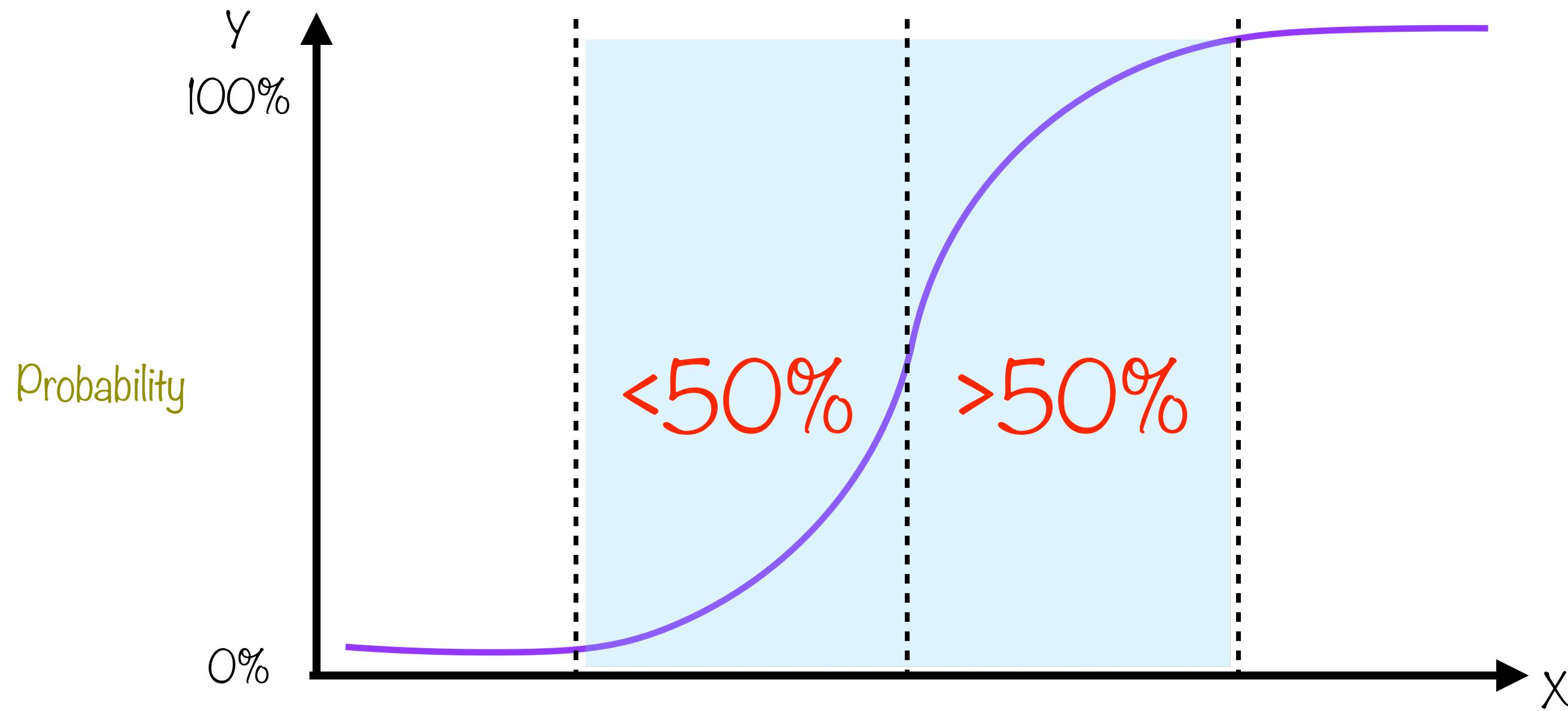
Working Smart with Logistic Regression



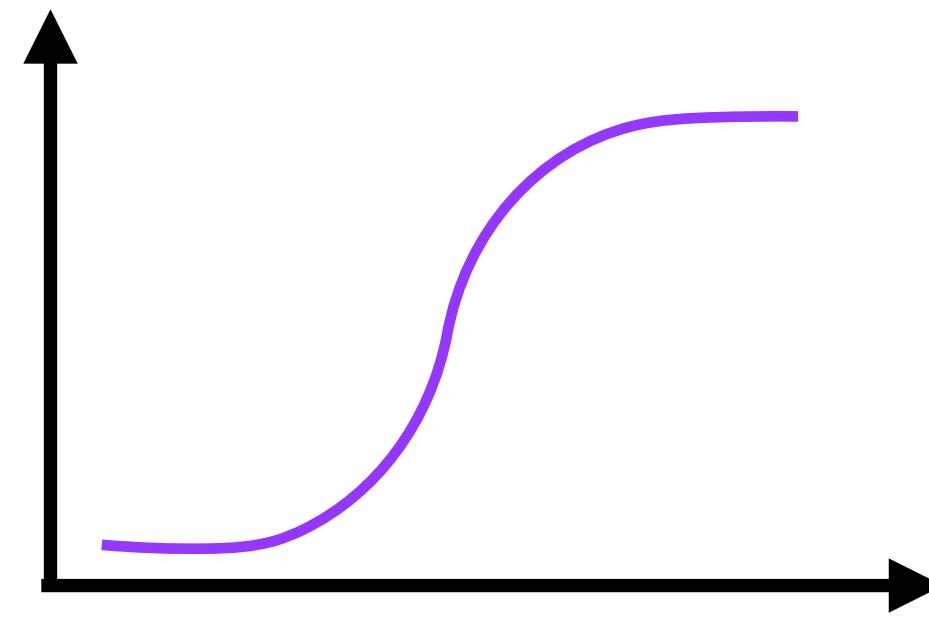
Time to deadline

Start too early, and you'll definitely make it

Working Smart with Logistic Regression



Time to deadline
Working smart is knowing when to start



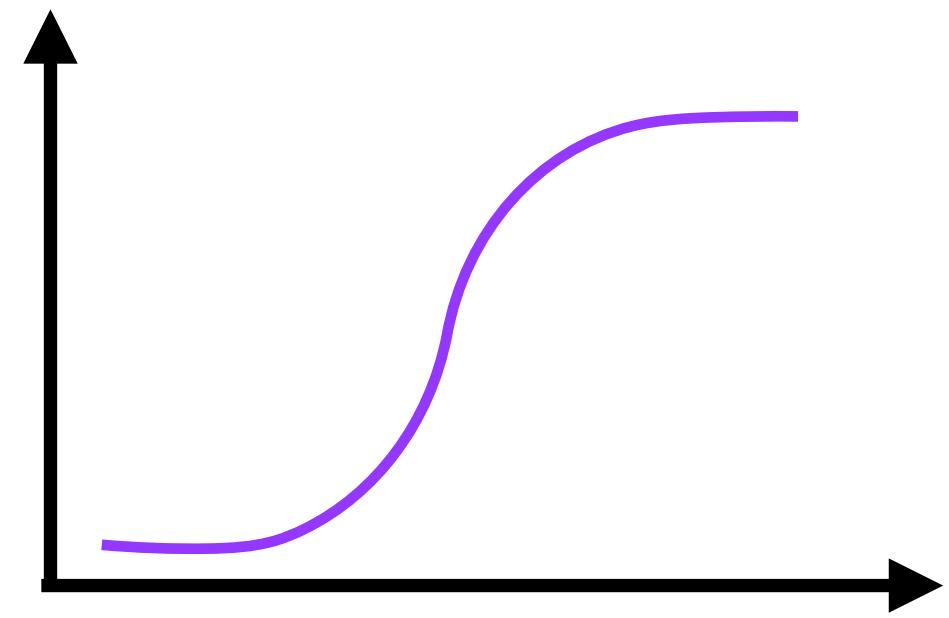
Y-axis: probability of meeting deadline

X-axis: time to deadline

Meeting or missing deadline is binary

Probability curve flattens at ends

- floor of 0
- ceiling of 1



y: hit or miss? (0 or 1?)

x: start time before deadline

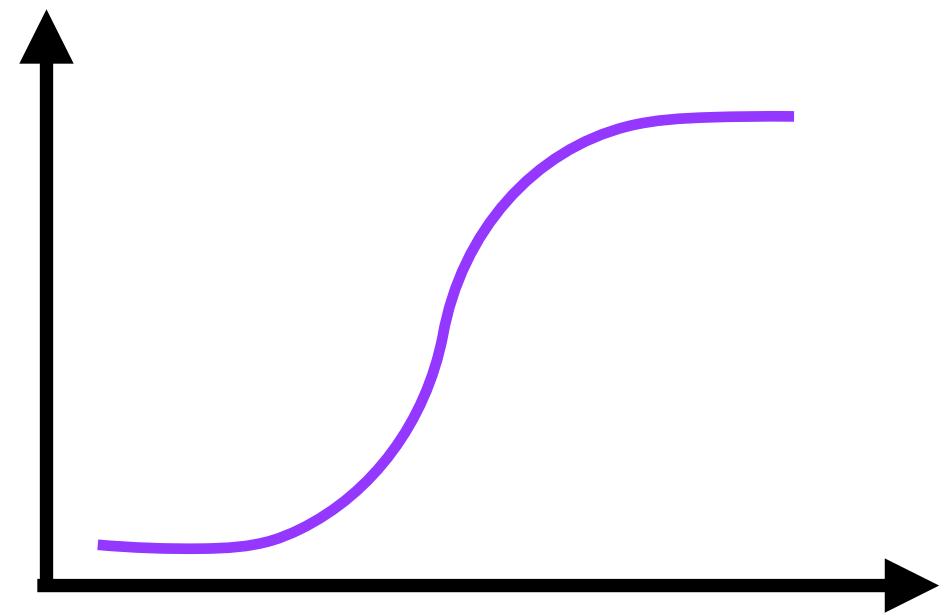
$p(y)$: probability of $y = 1$

$$p(y_i) = \frac{1}{1 + e^{-(A+Bx_i)}}$$

Logistic regression involves finding the “best fit” such curve

- A is the intercept
- B is the regression coefficient

(e is the constant 2.71828)



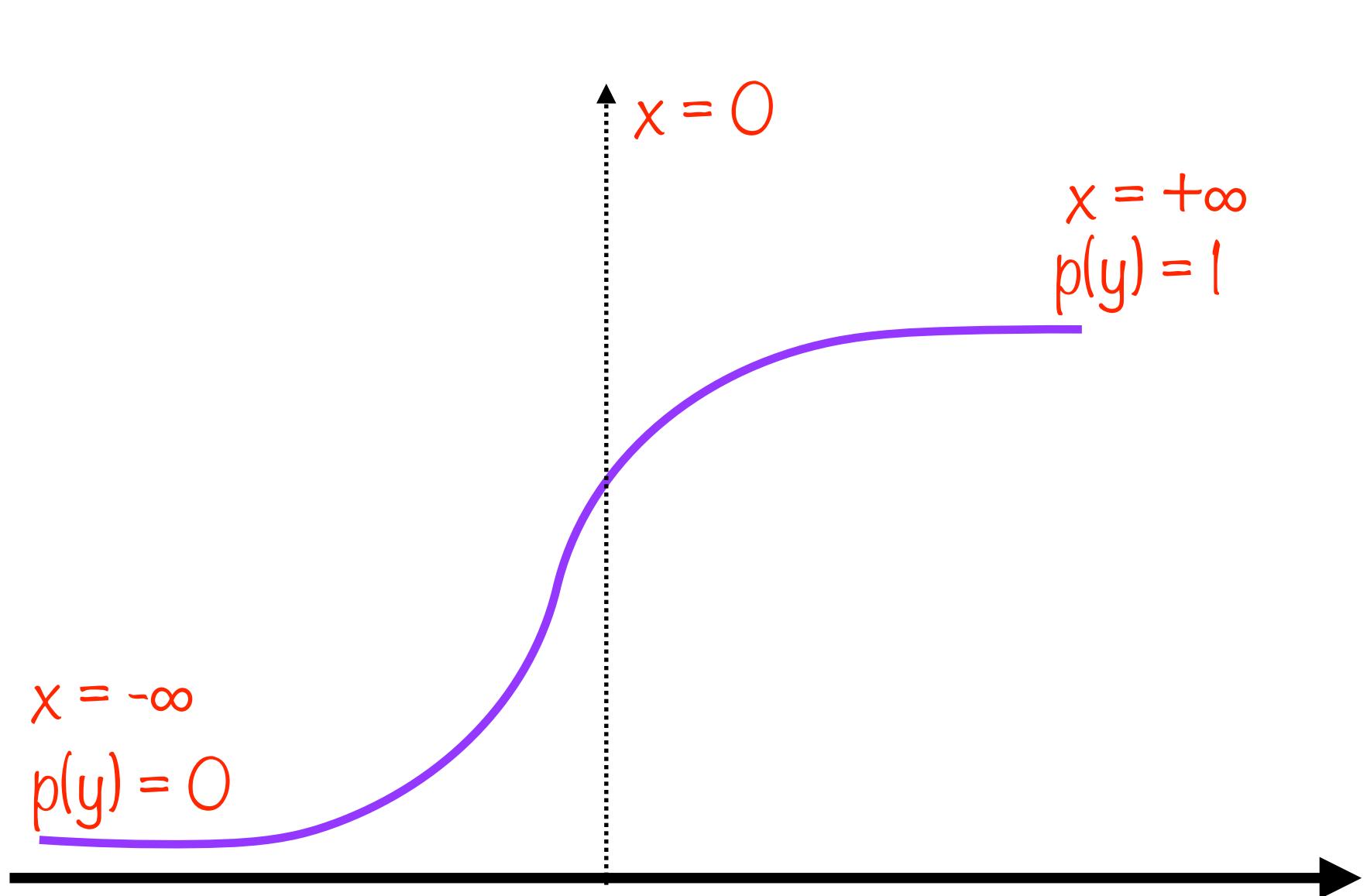
S-curves are widely studied, well understood

$$y = \frac{1}{1 + e^{-(A+Bx)}}$$

Logistic regression uses S-curve to estimate probabilities

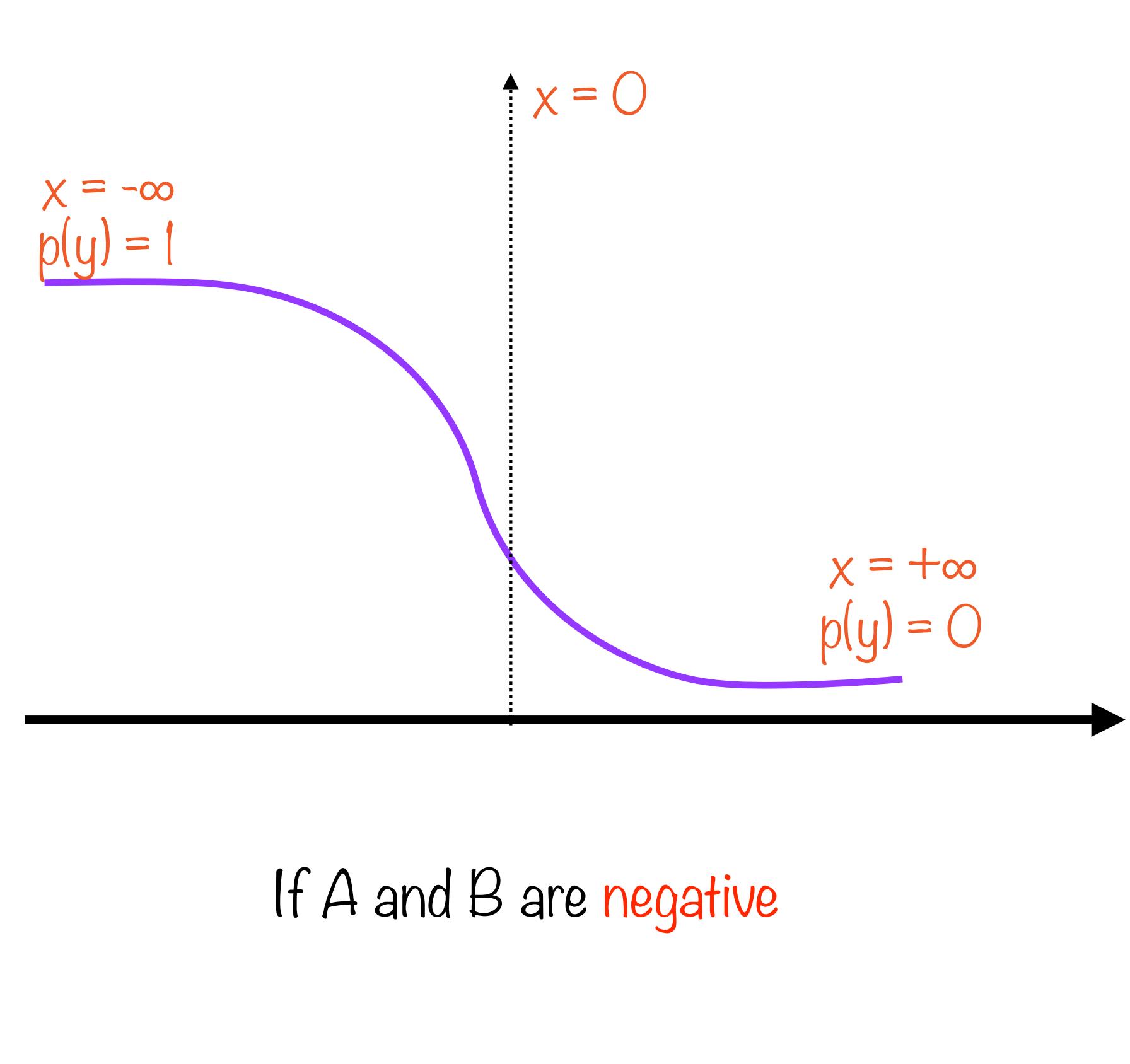
$$p(y) = \frac{1}{1 + e^{-(A+Bx)}}$$

$$p(y_i) = \frac{1}{1 + e^{-(A+Bx_i)}}$$



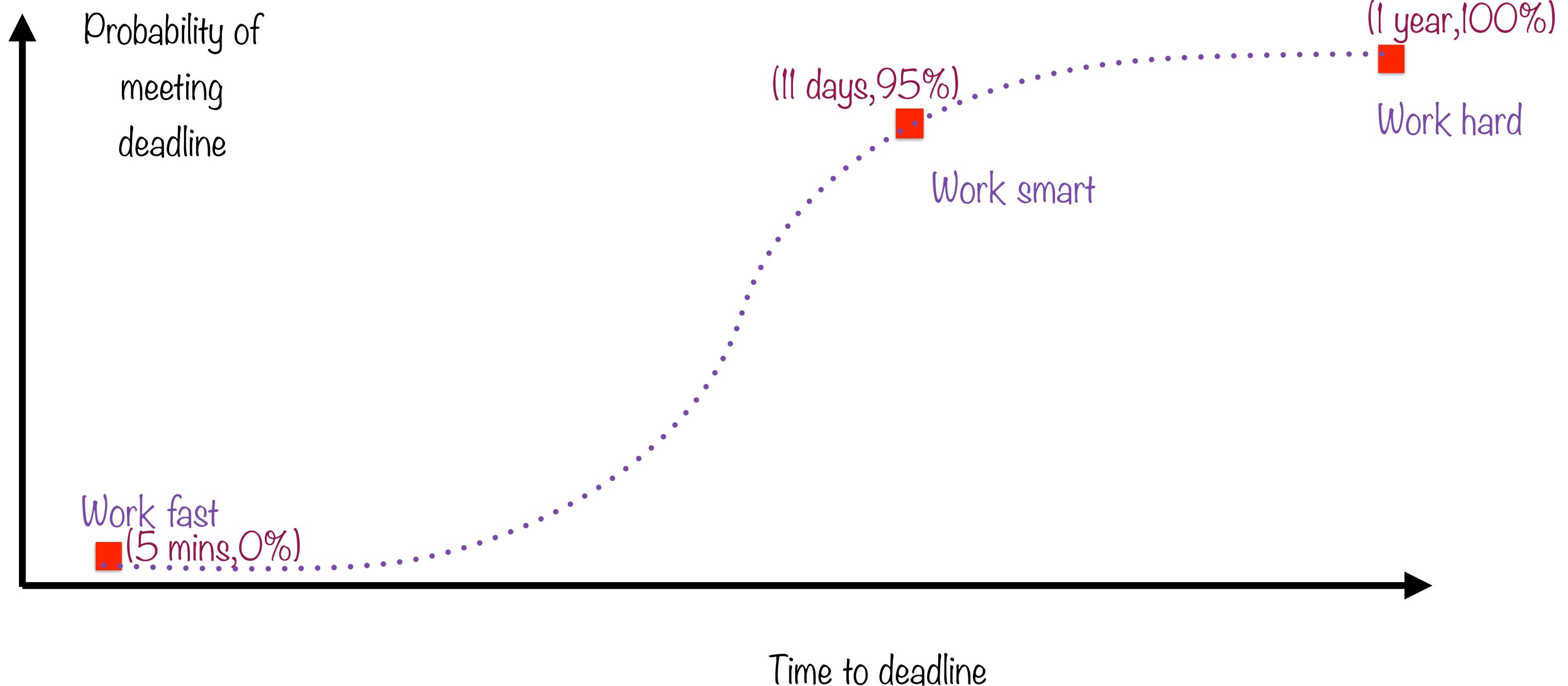
If A and B are positive

$$p(y_i) = \frac{1}{1 + e^{-(A+Bx_i)}}$$

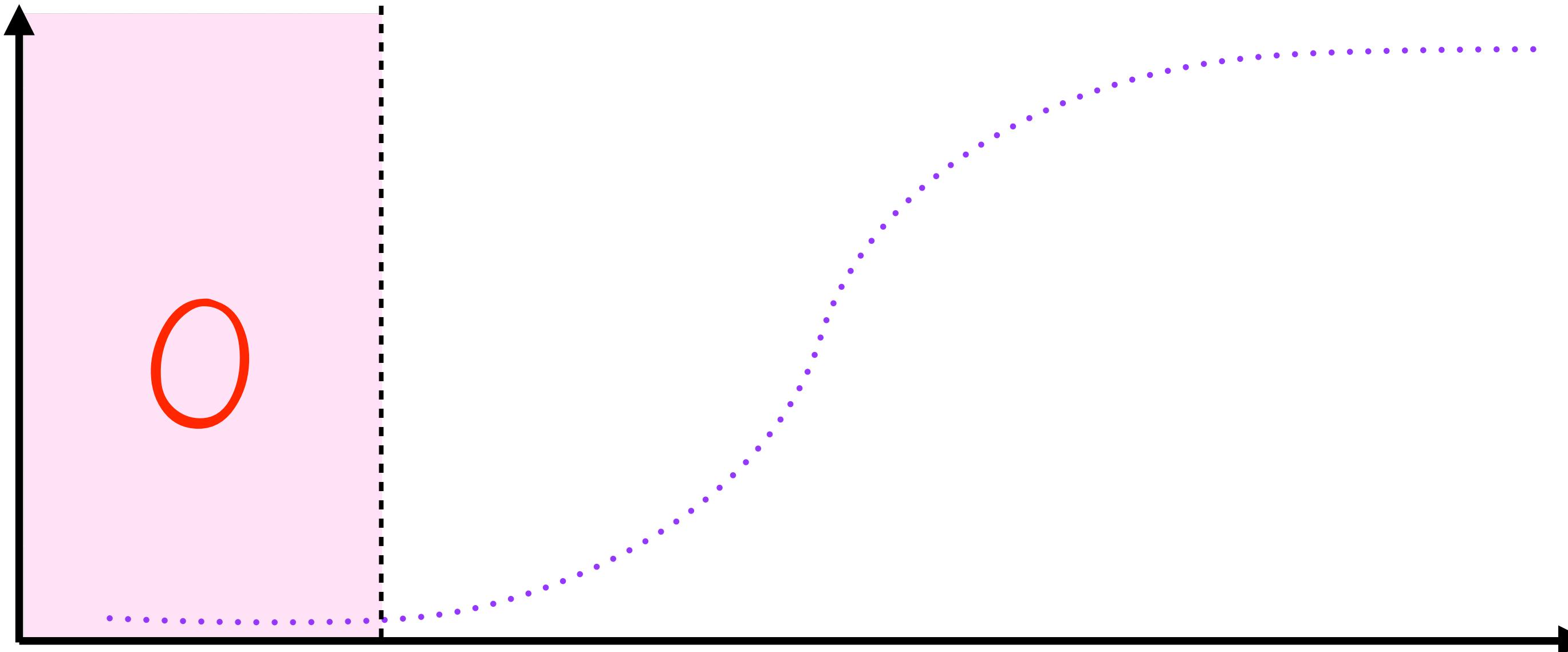


If A and B are negative

Working Hard, Fast, Smart

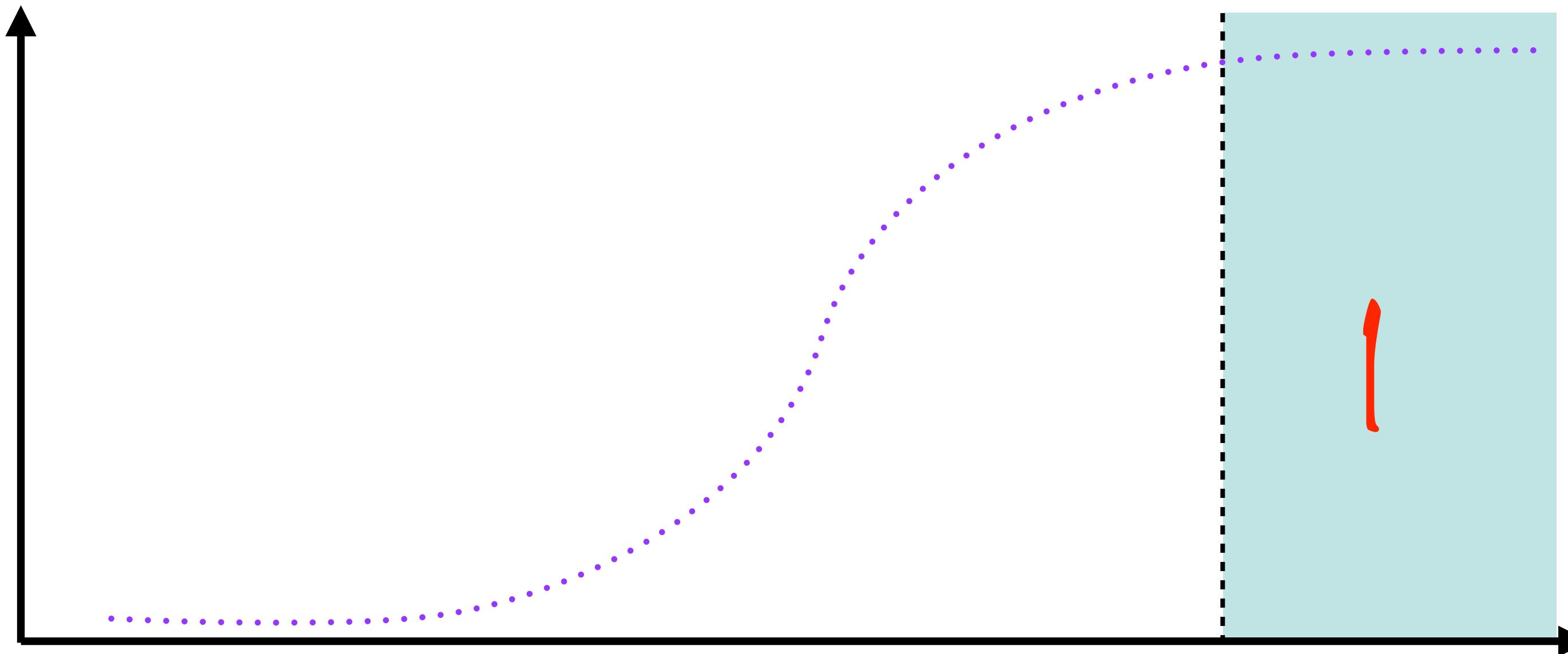


Working Hard, Fast, Smart



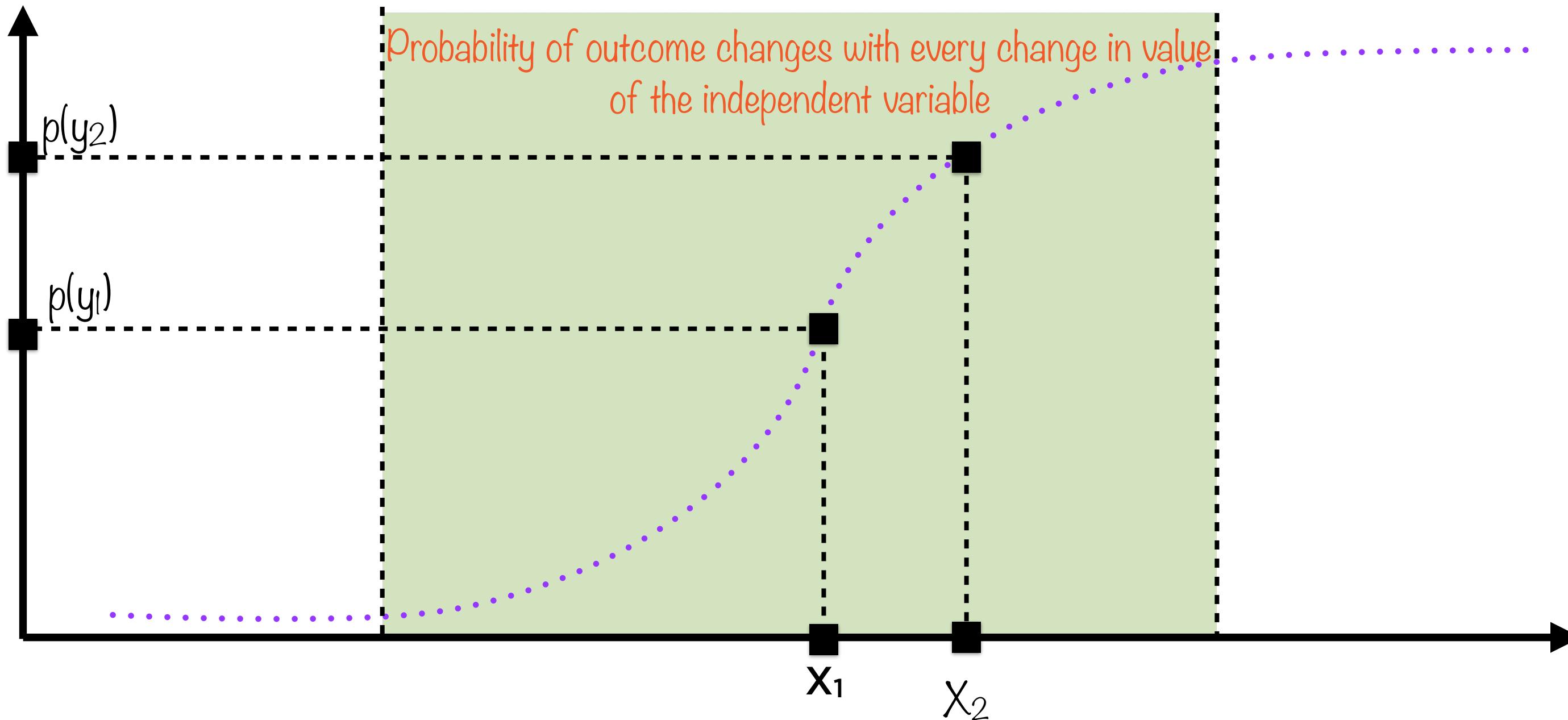
Minimum value of $p(y_i)$

Working Hard, Fast, Smart



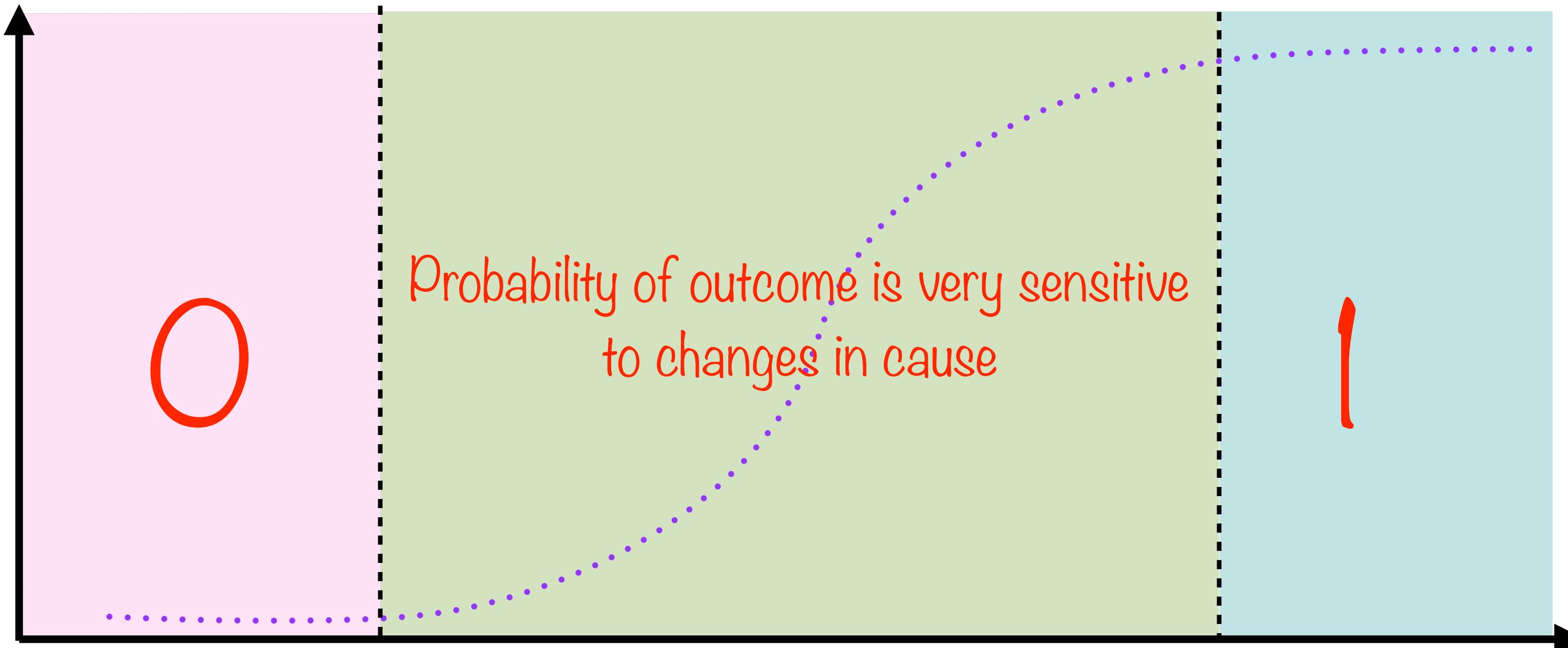
Maximum value of $p(y_i)$

Working Hard, Fast, Smart



Between maximum and minimum values of $p(y_i)$

Logistic Regression



$$p(y_i) = \frac{1}{1 + e^{-(A+Bx_i)}}$$

Categorical and Continuous Variables

Continuous

Can take an infinite set of values
(height, weight, income...)

Categorical

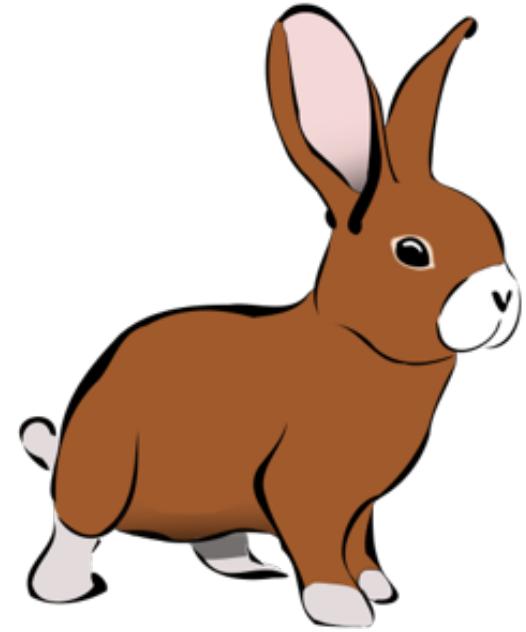
Can take a finite set of values
(male/female, day of week...)

Categorical variables that can take just two values are called
binary variables

Logistic Regression helps estimate how
probabilities of categorical variables are
influenced by causes

Logistic Regression in Classification

Whales: Fish or Mammals



Mammal

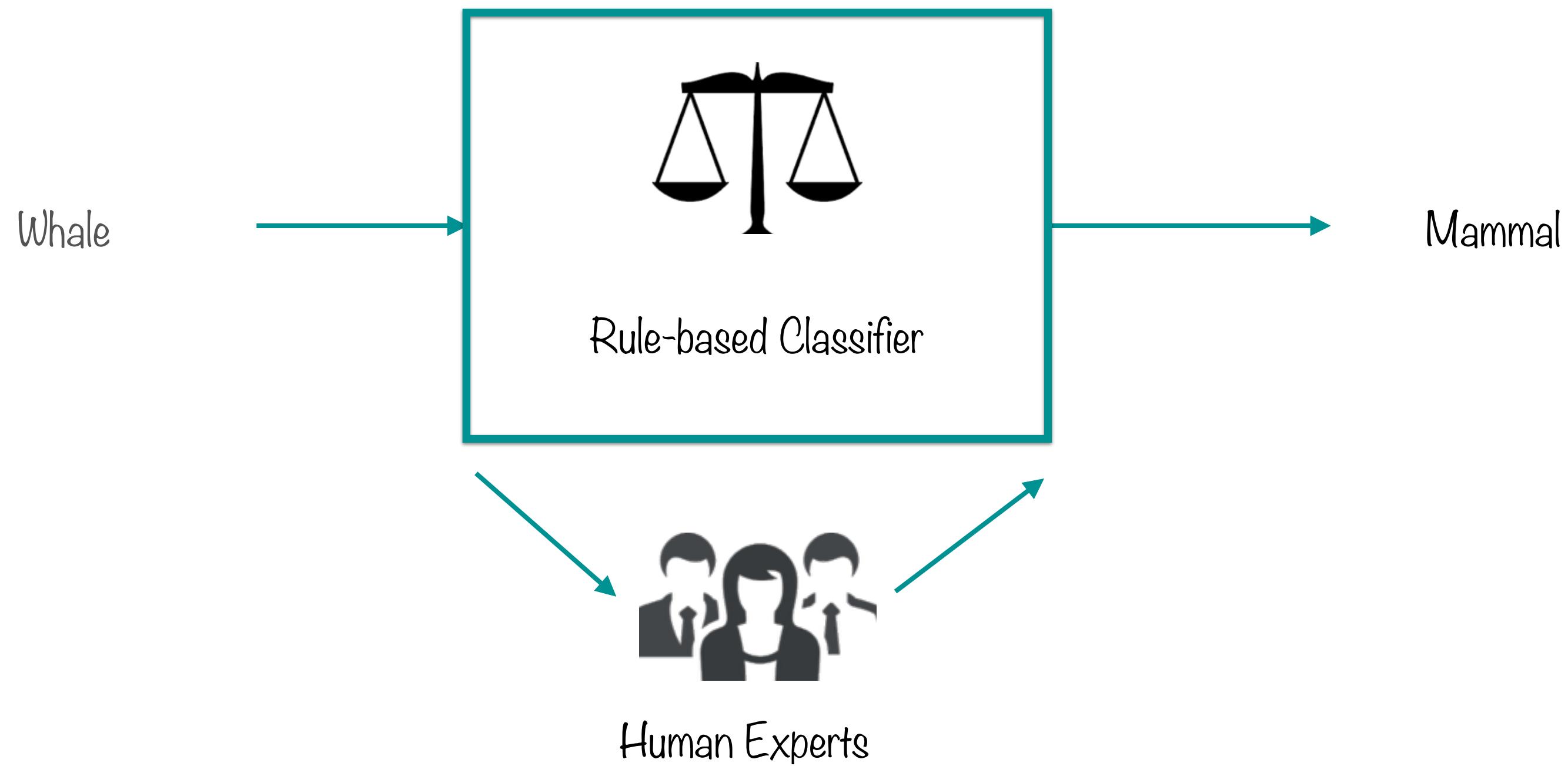
Member of the infraorder Cetacea



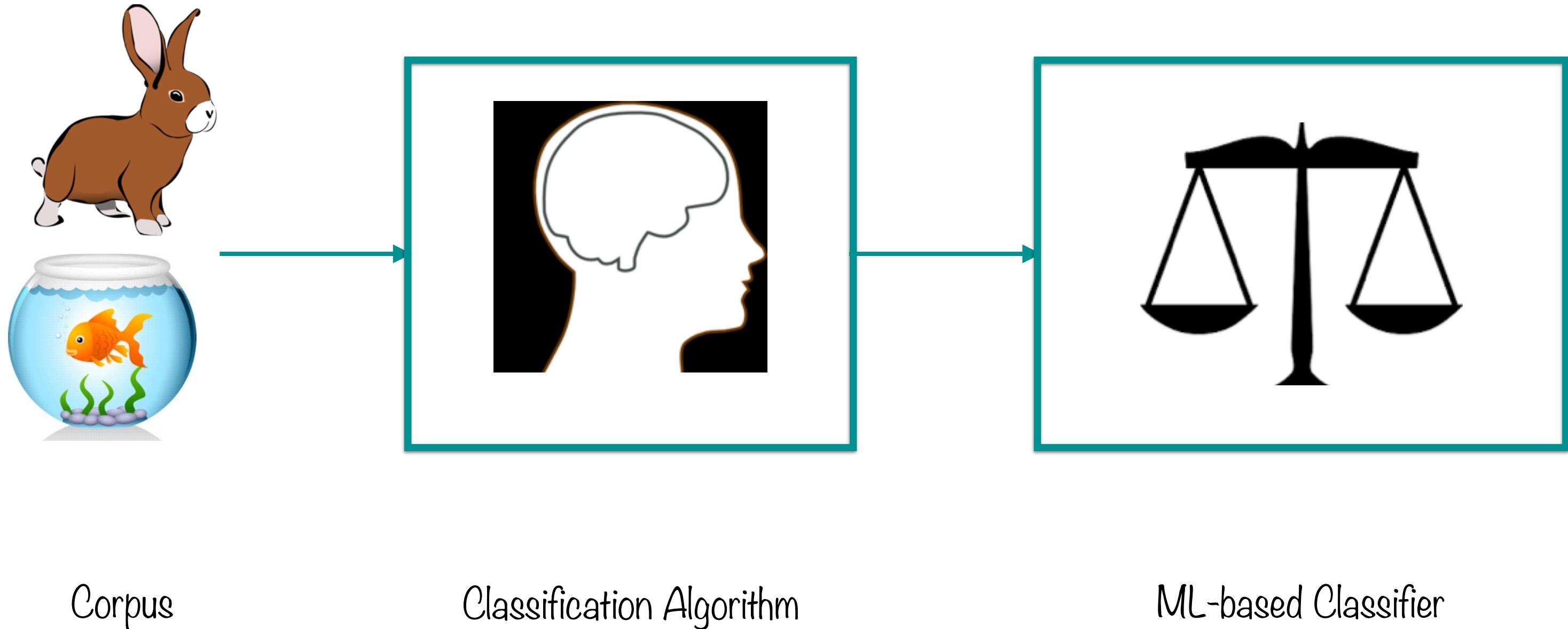
Fish

Looks like a fish, swims like a fish, moves like a fish

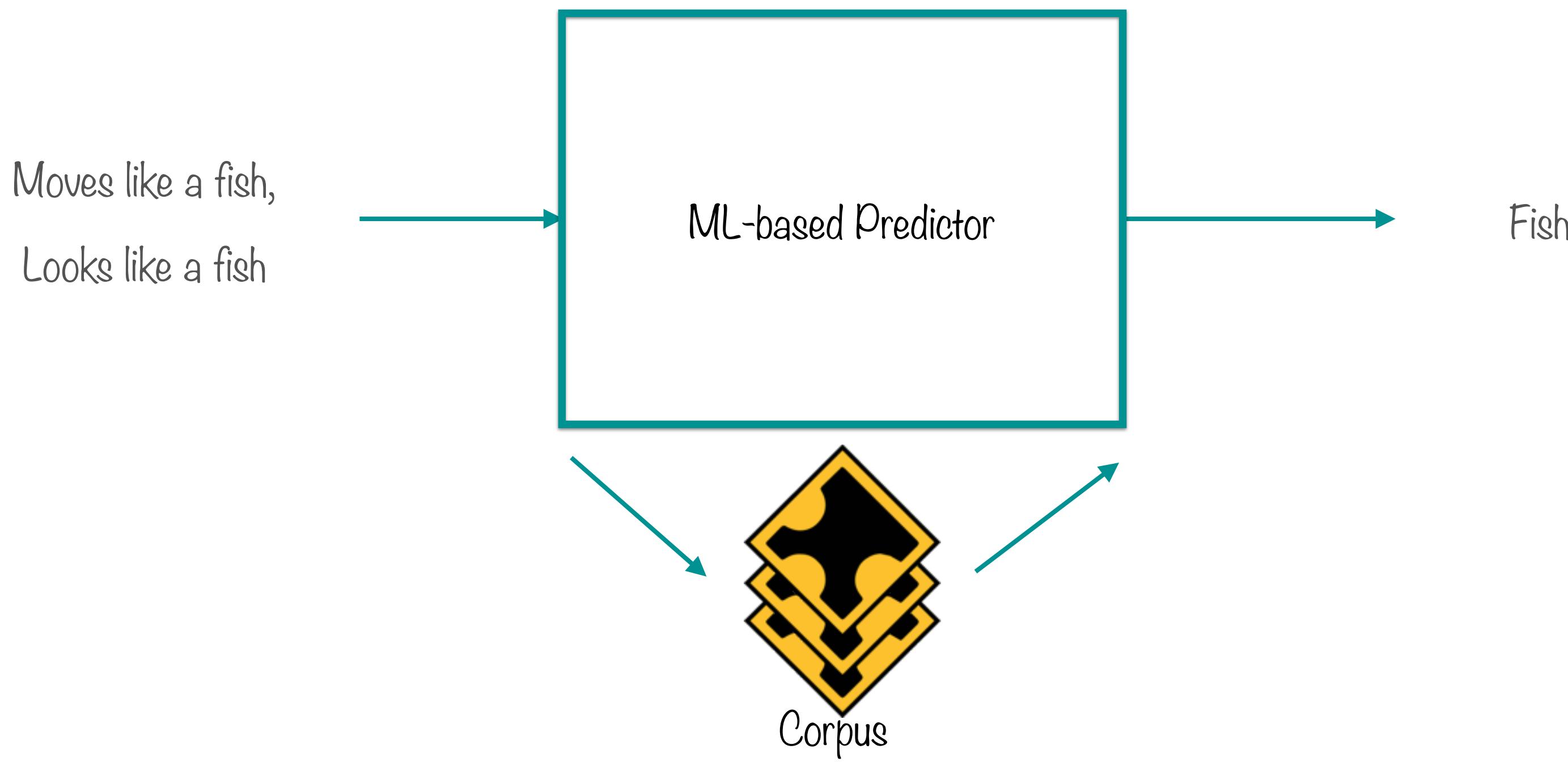
Rule-based Binary Classifier



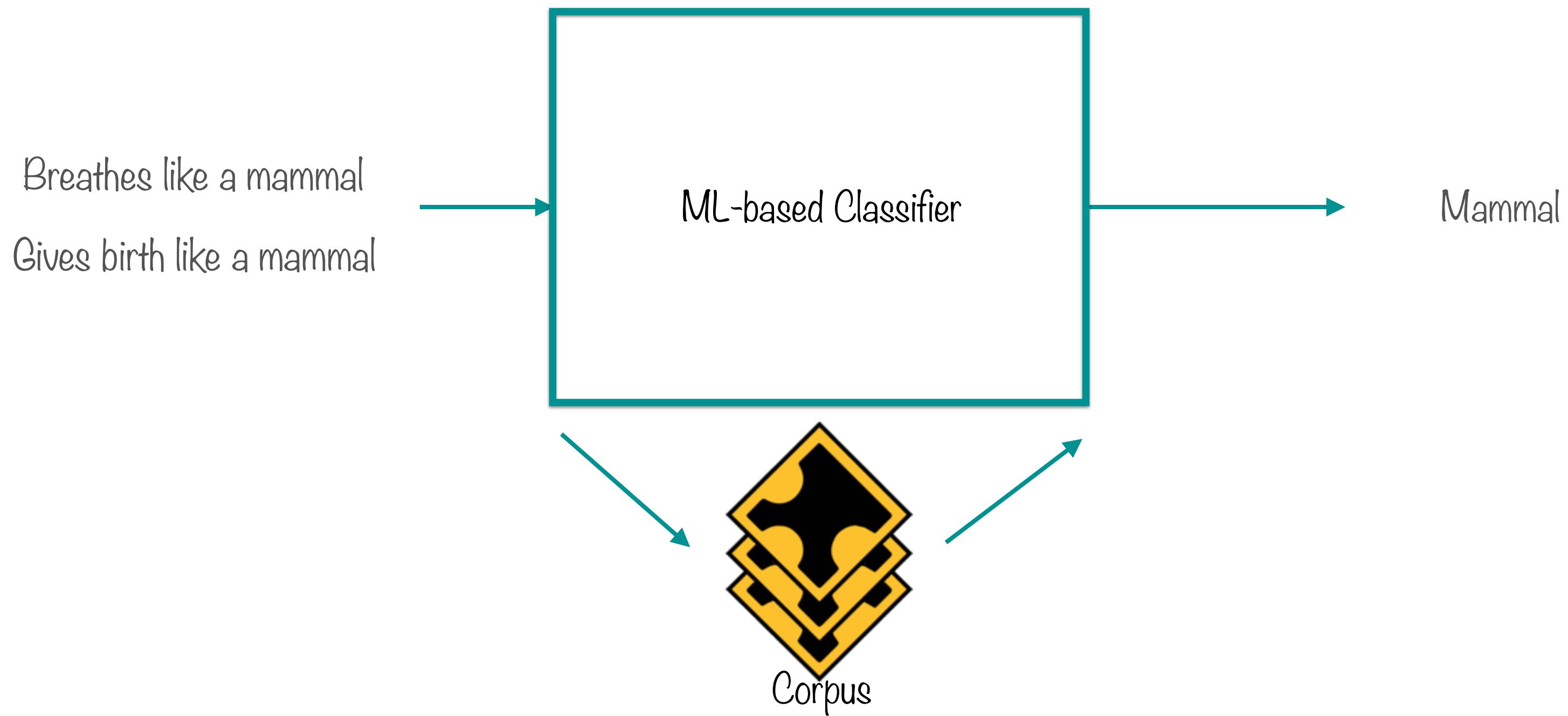
ML-based Binary Classifier



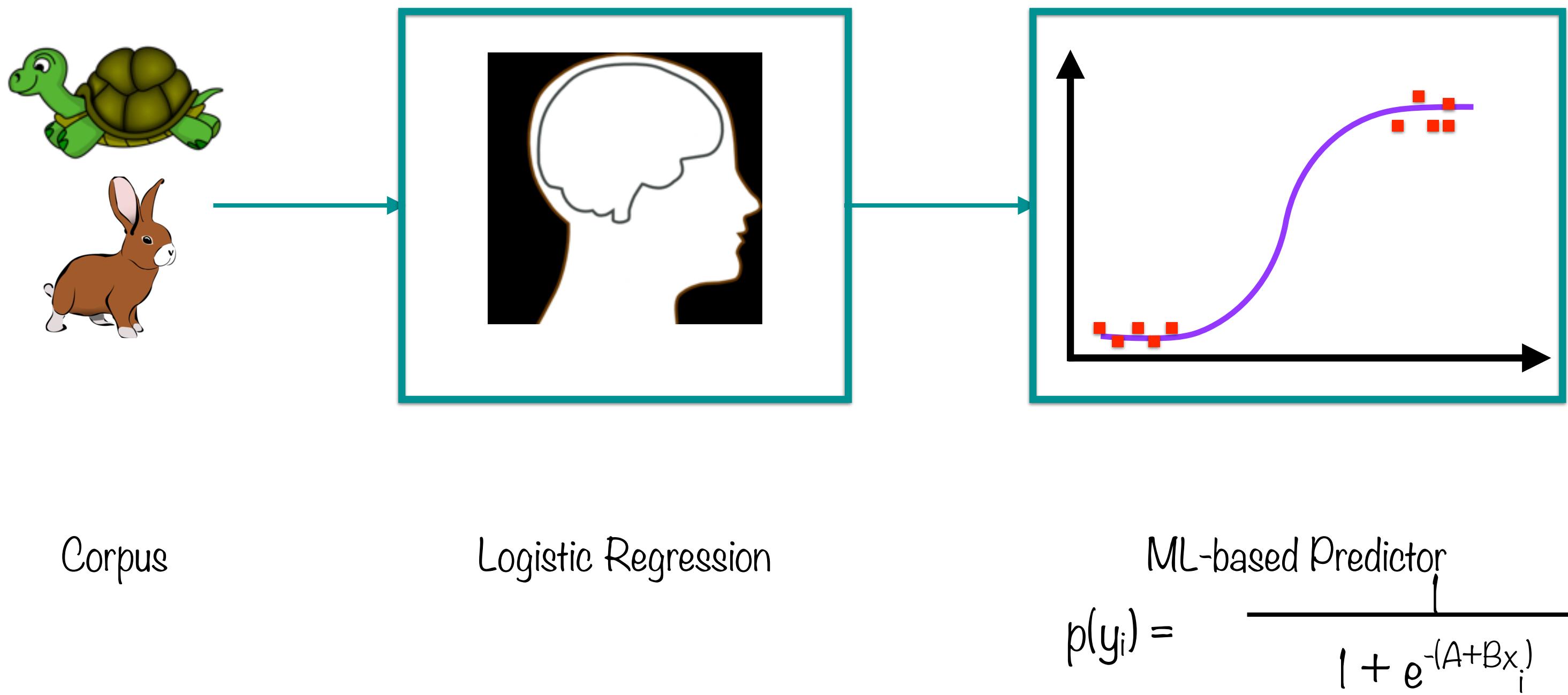
ML-based Binary Classifier



ML-based Binary Classifier

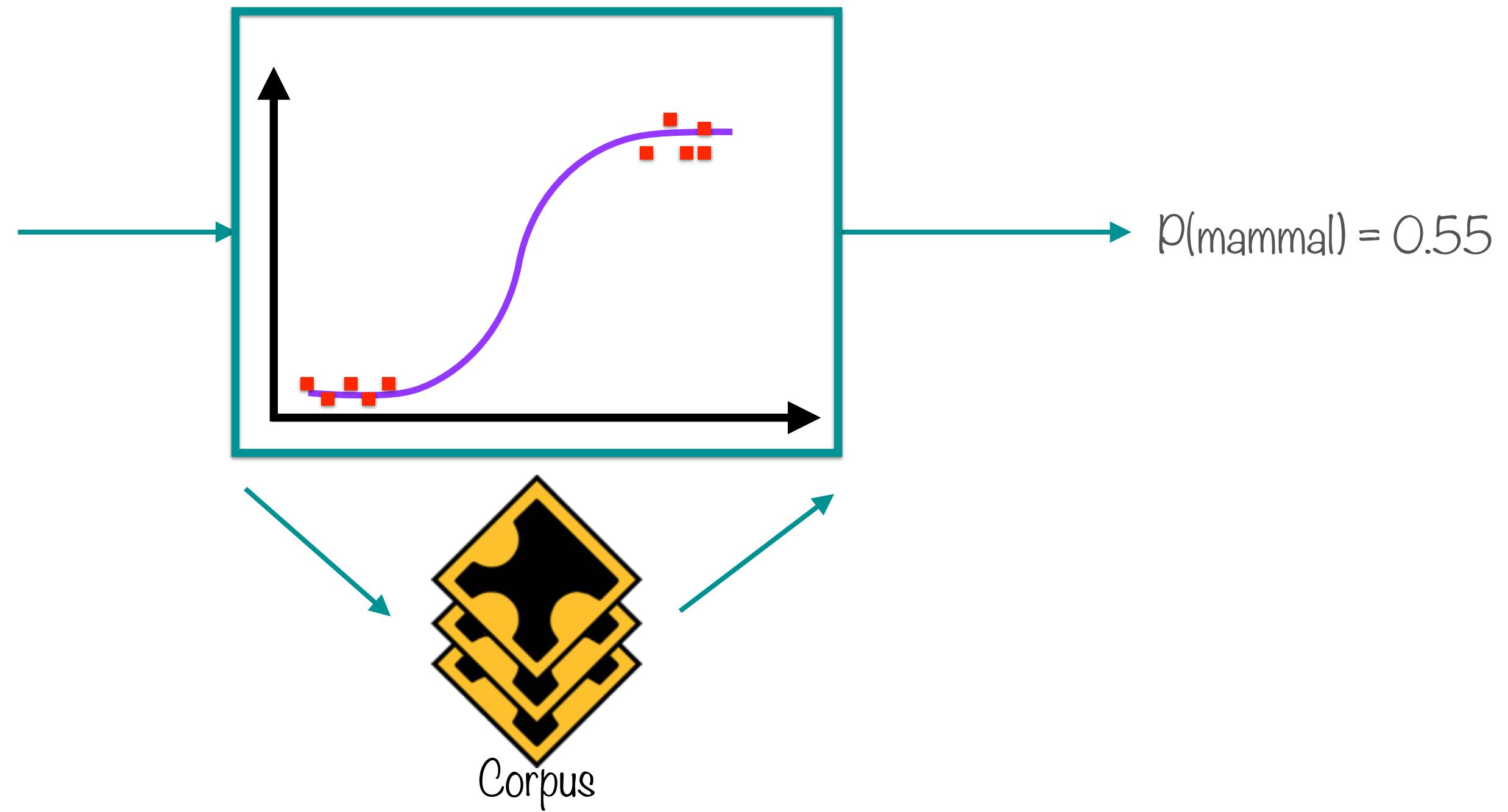


ML-based Predictor

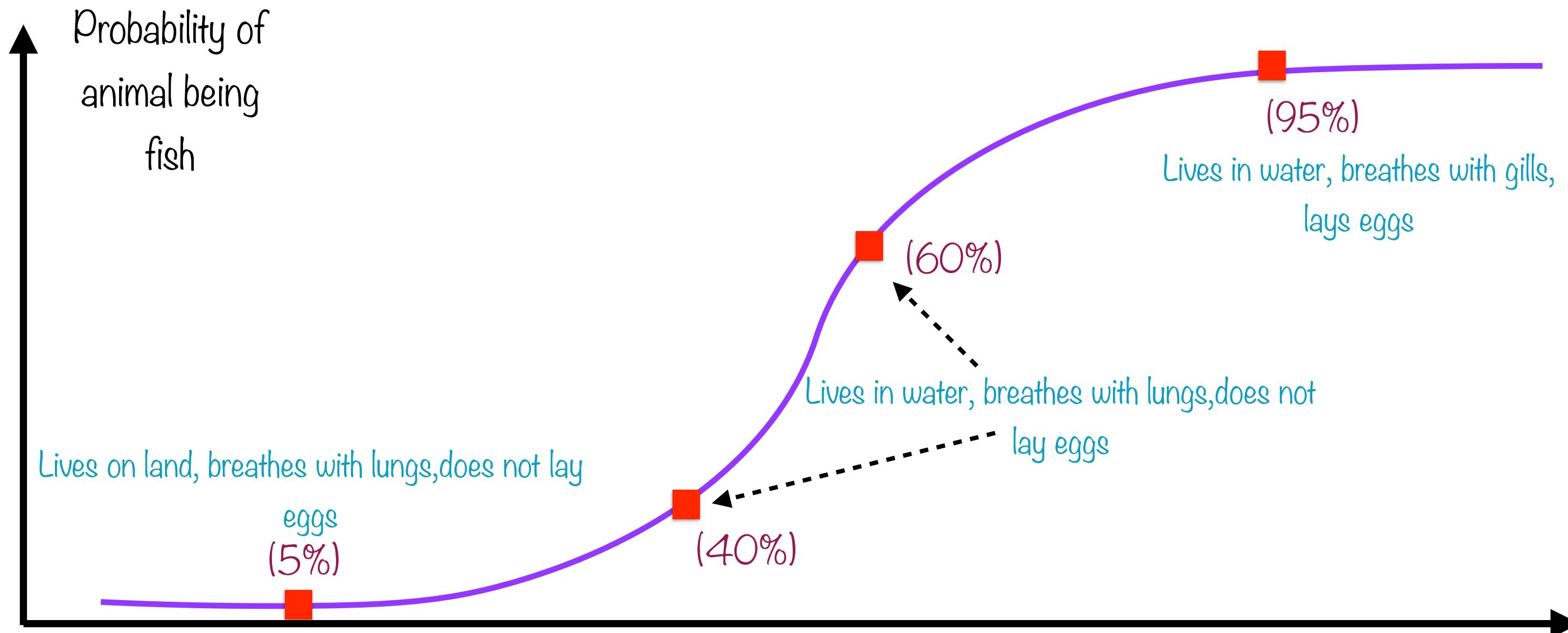


ML-based Predictor

Lives in water, breathes
with lungs, does not lay
eggs

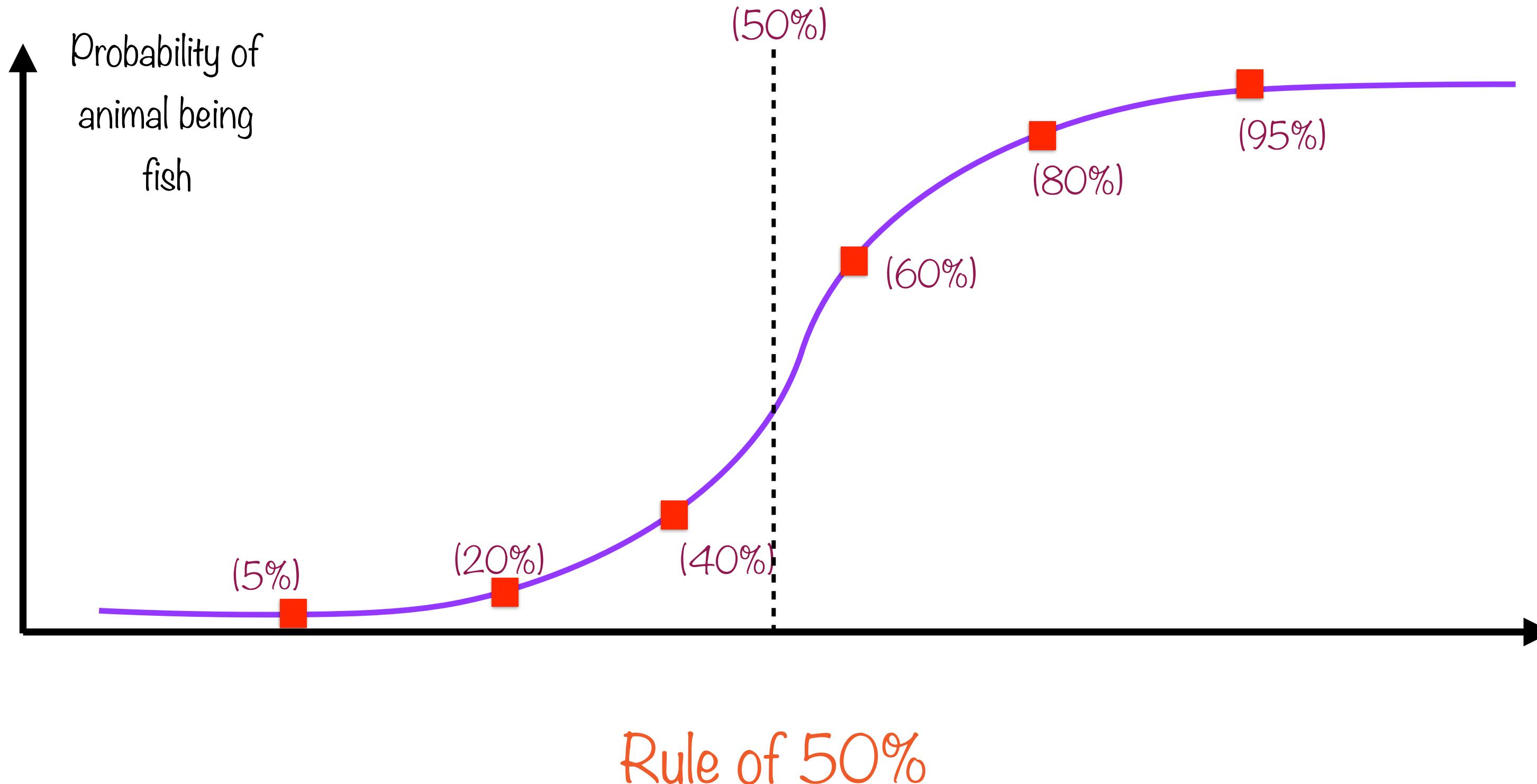


Applying Logistic Regression

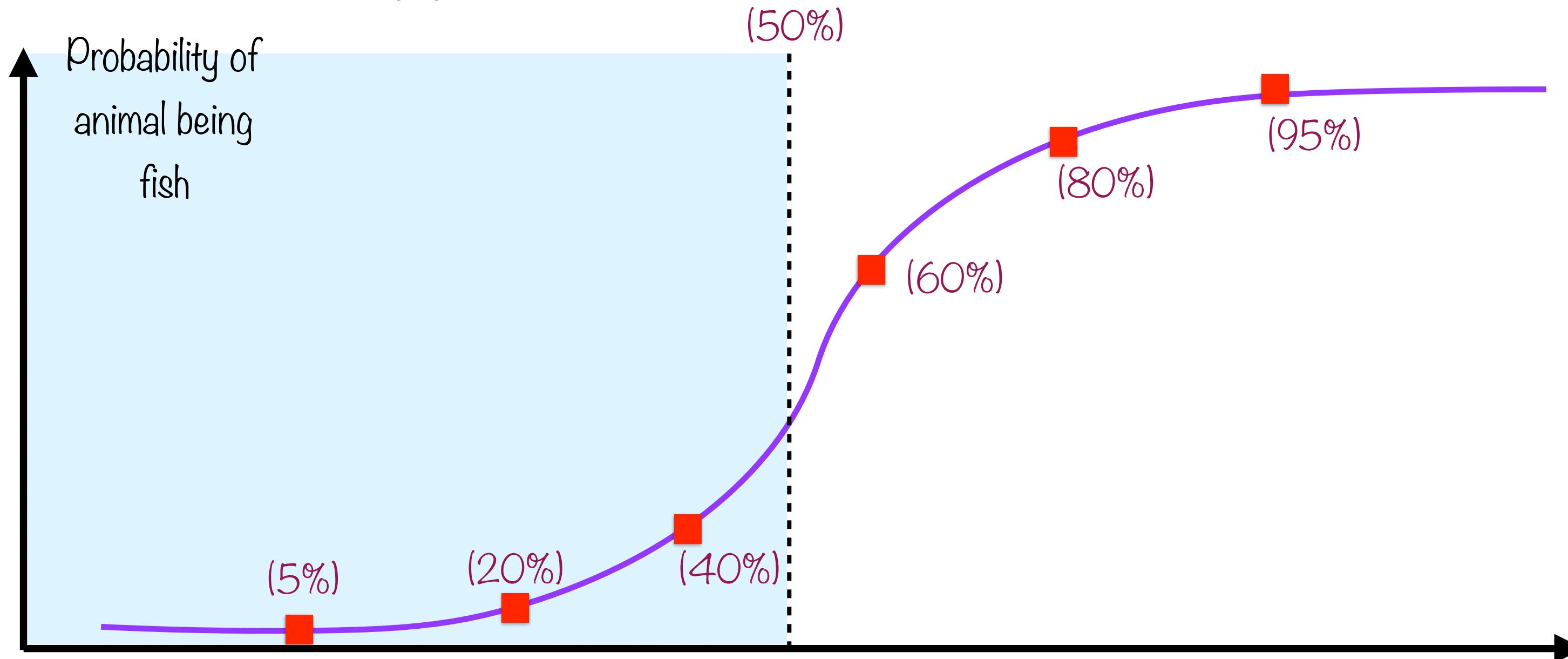


Whales: Fish or Mammals?

Applying Logistic Regression

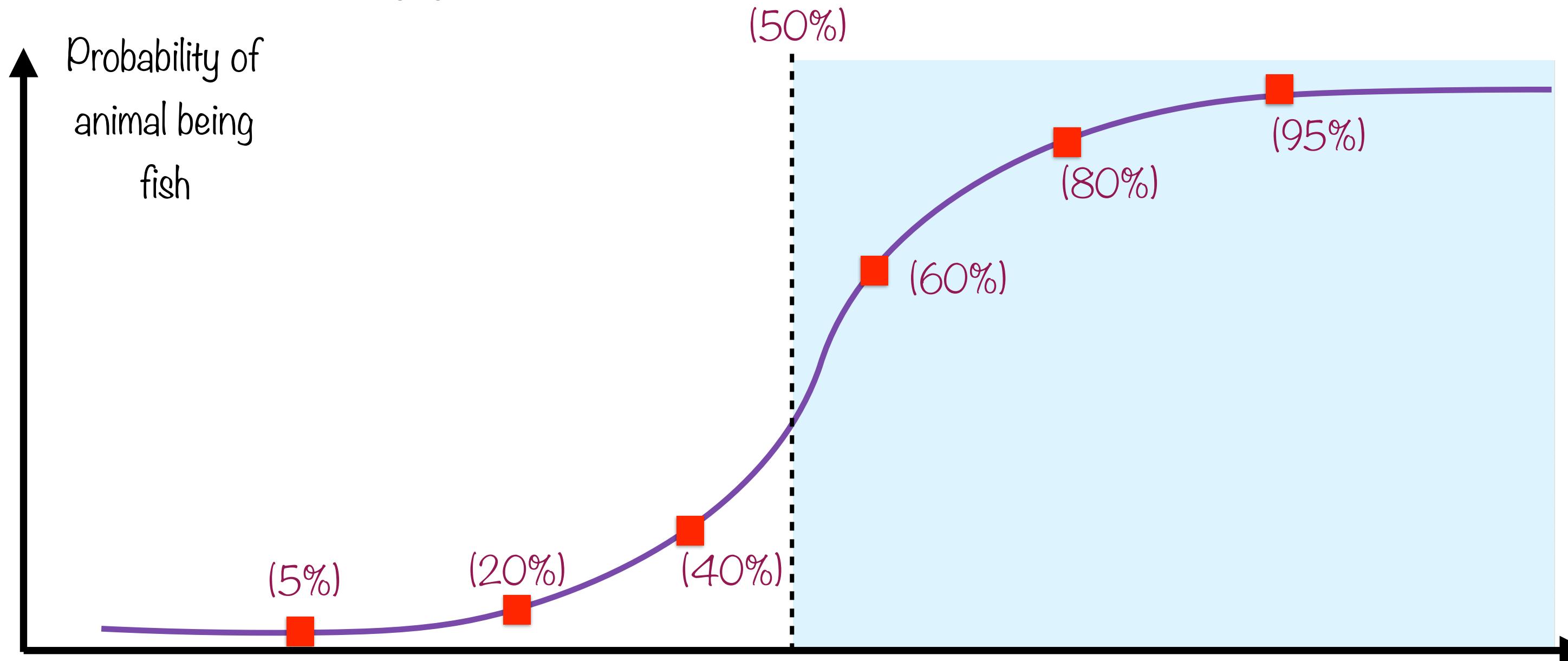


Applying Logistic Regression



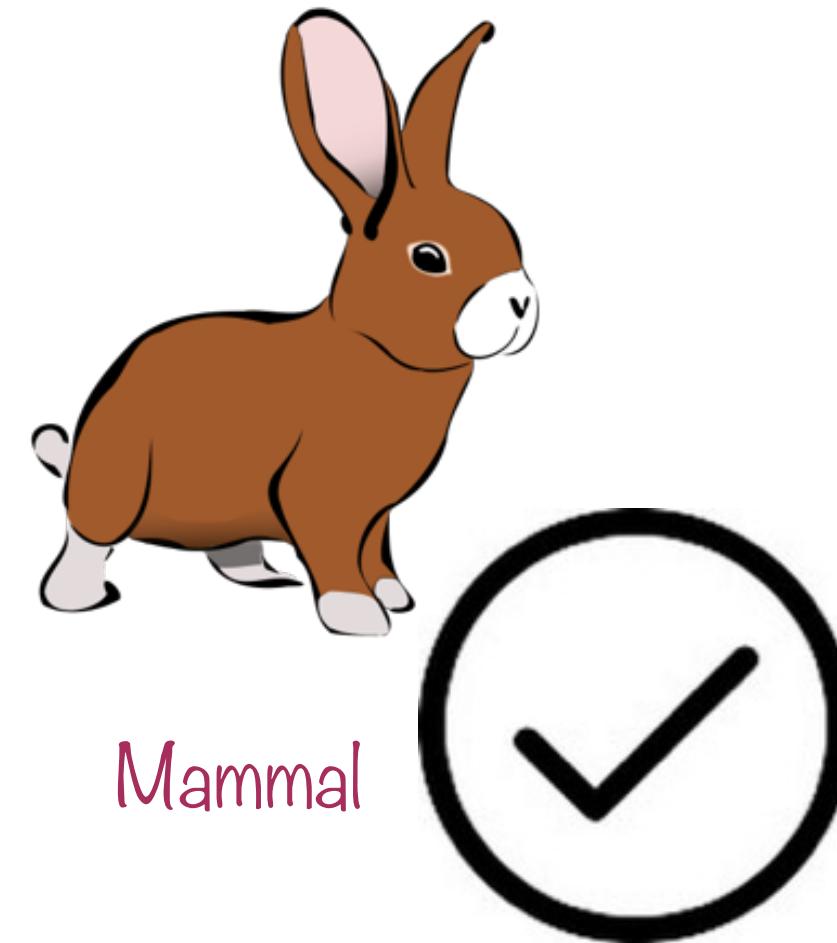
If probability < 50%, it's a mammal

Applying Logistic Regression



If probability < 50%, it's a mammal

Applying Logistic Regression



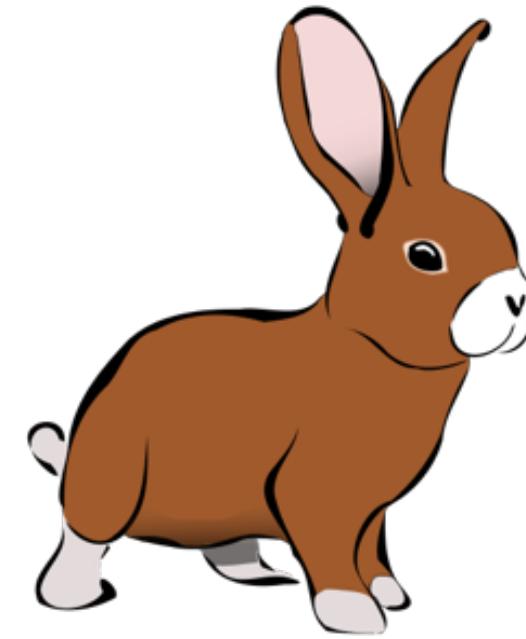
Mammal



Fish

Probability of whales being fish < 50%

Applying Logistic Regression



Mammal



Fish



Probability of whales being fish > 50%

Logistic Regression and Linear Regression

X Causes Y



Cause

Independent variable



Effect

Dependent variable

X Causes Y



Cause

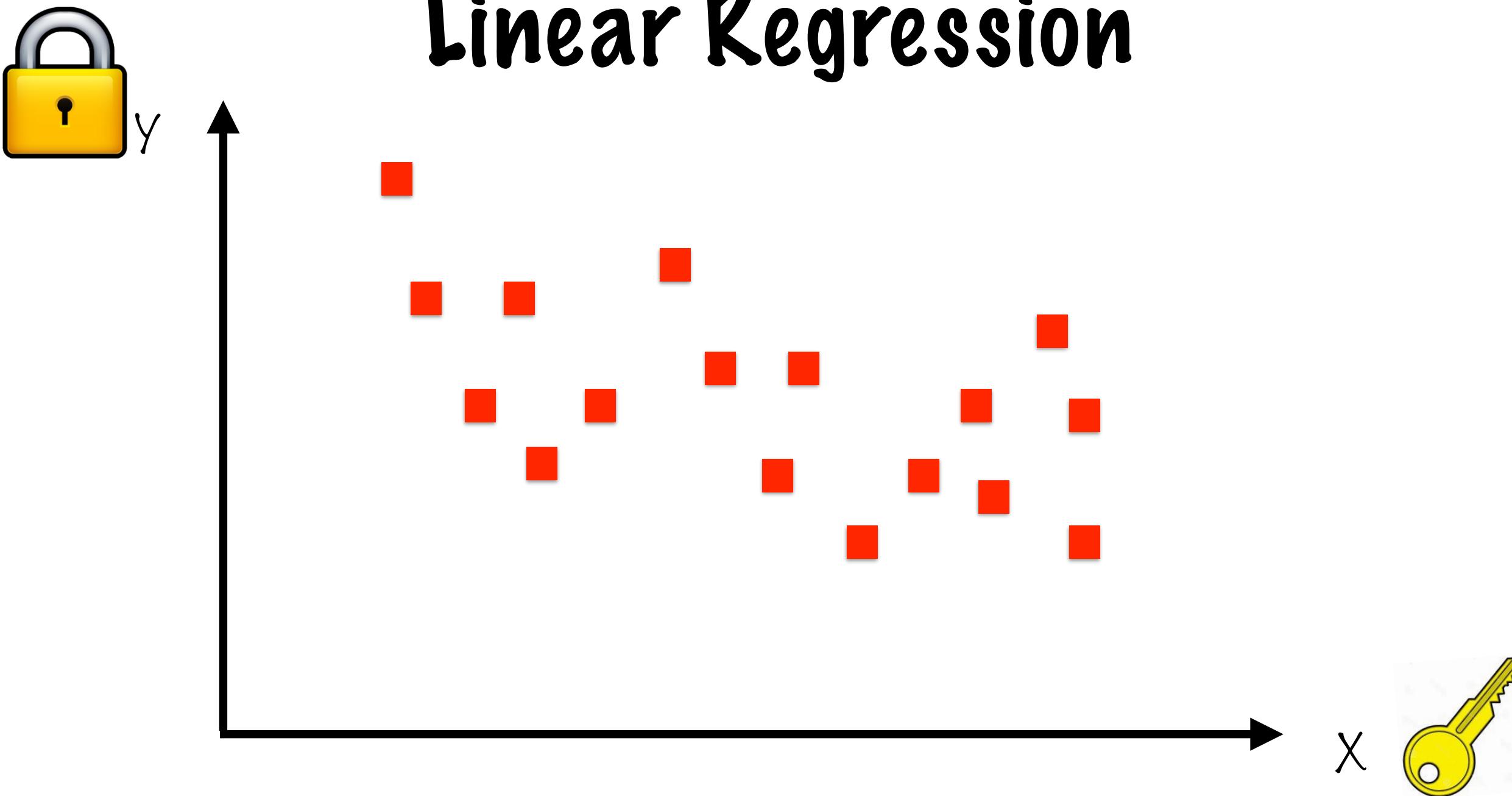
Explanatory variable



Effect

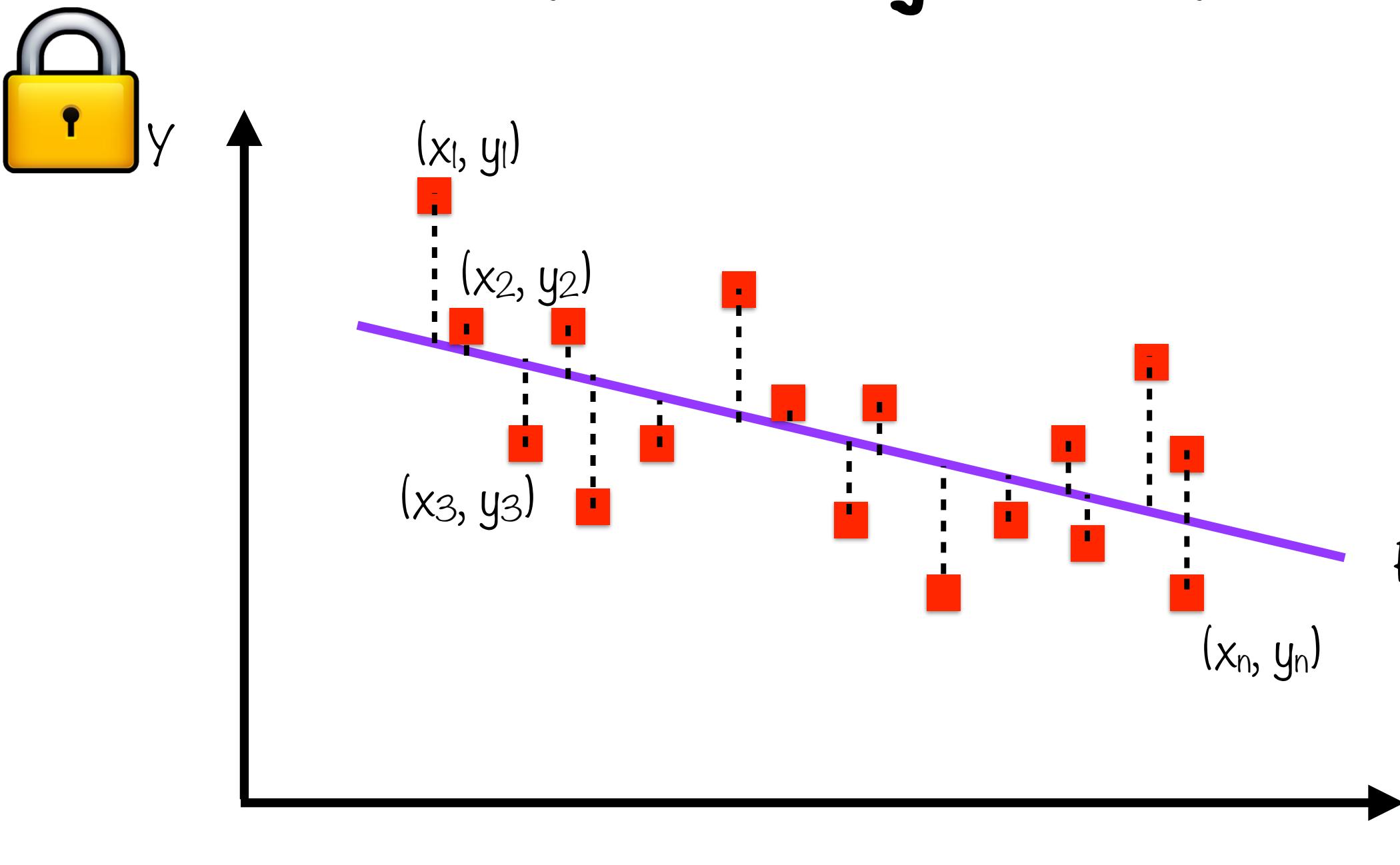
Dependent variable

Linear Regression



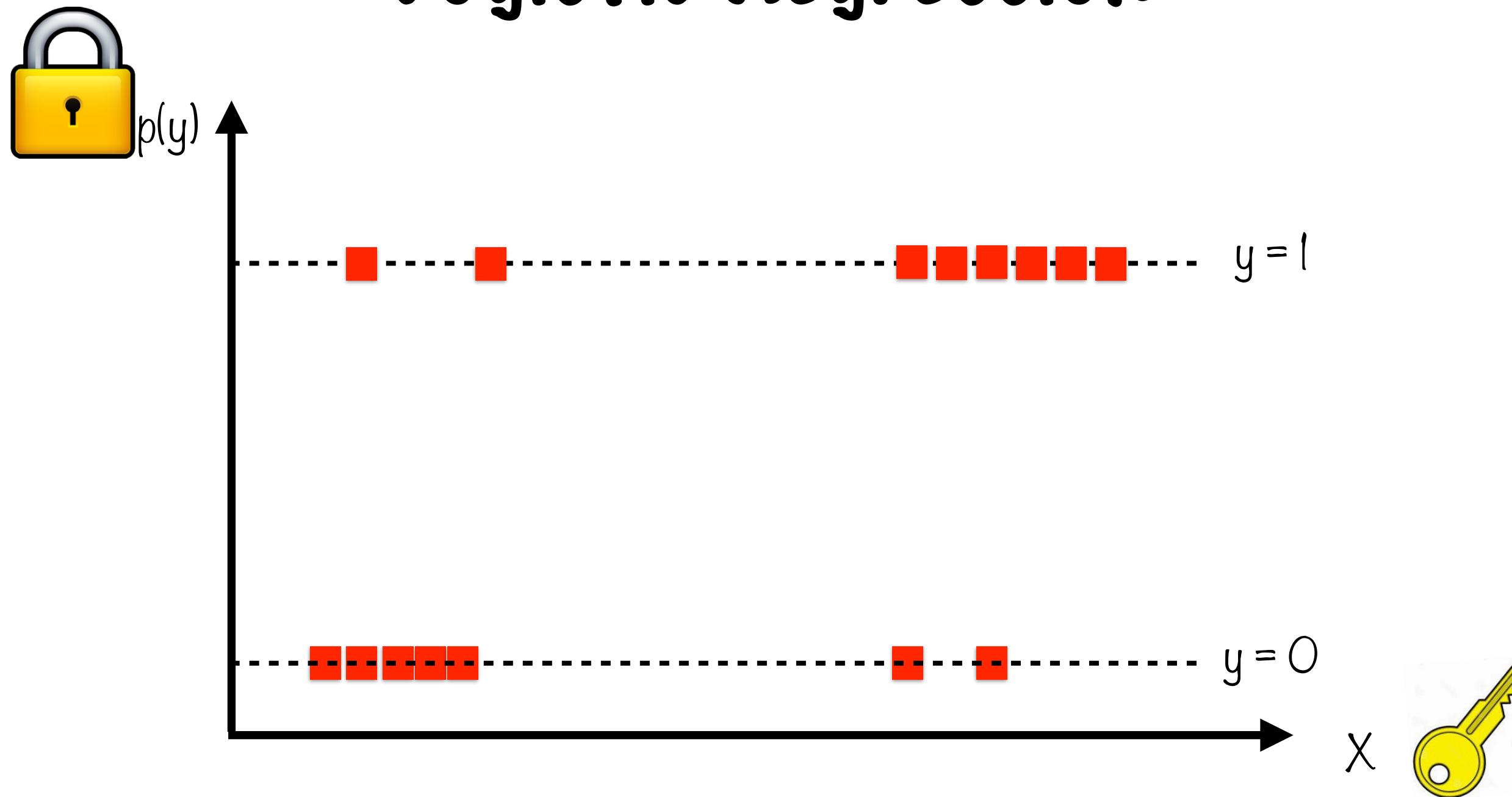
Represent all n points as (x_i, y_i) , where $i = 1$ to n

Linear Regression



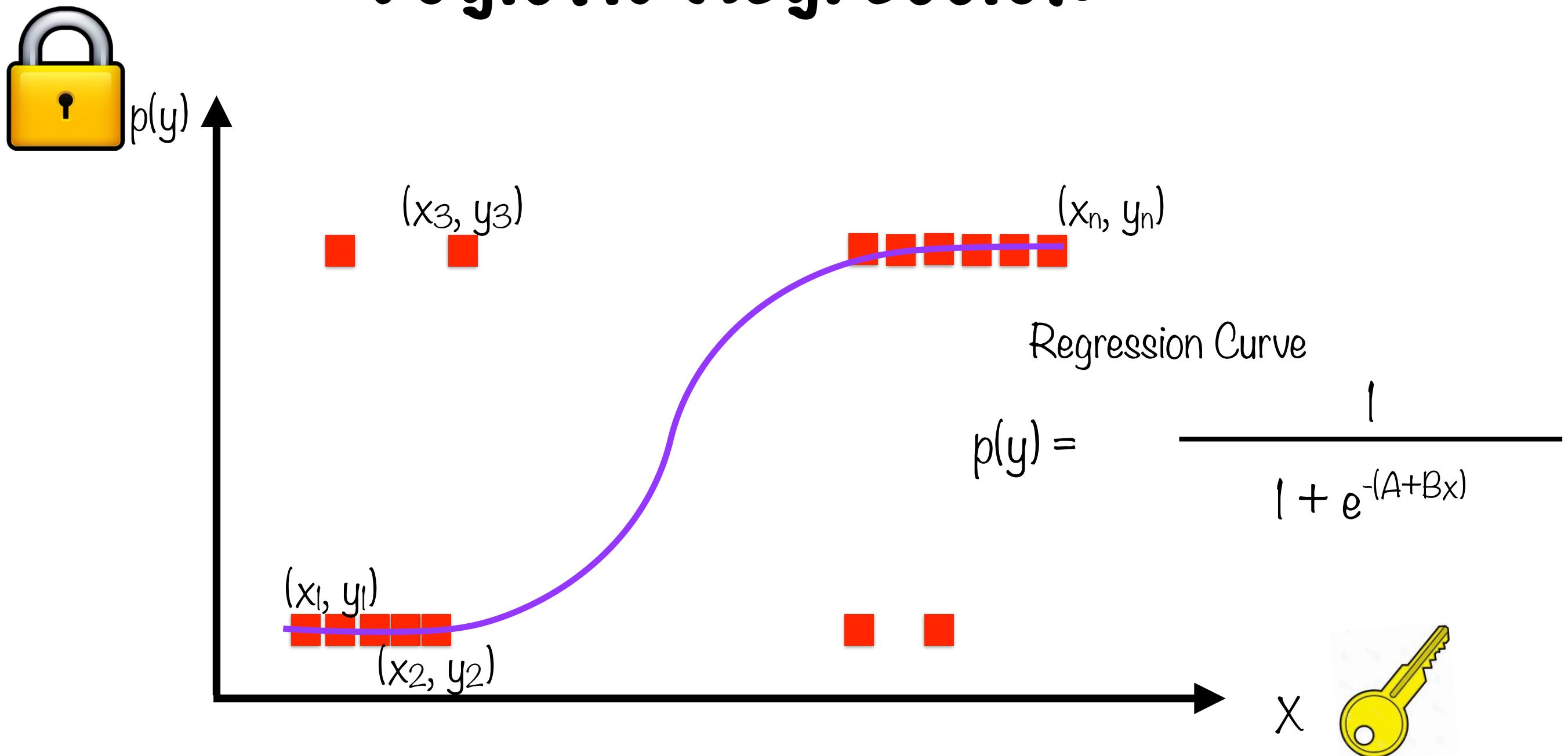
Represent all n points as (x_i, y_i) , where $i = 1$ to n

Logistic Regression



Represent all n points as (x_i, y_i) , where $i = 1$ to n

Logistic Regression



Represent all n points as (x_i, y_i) , where $i = 1$ to n

Similar, yet Different

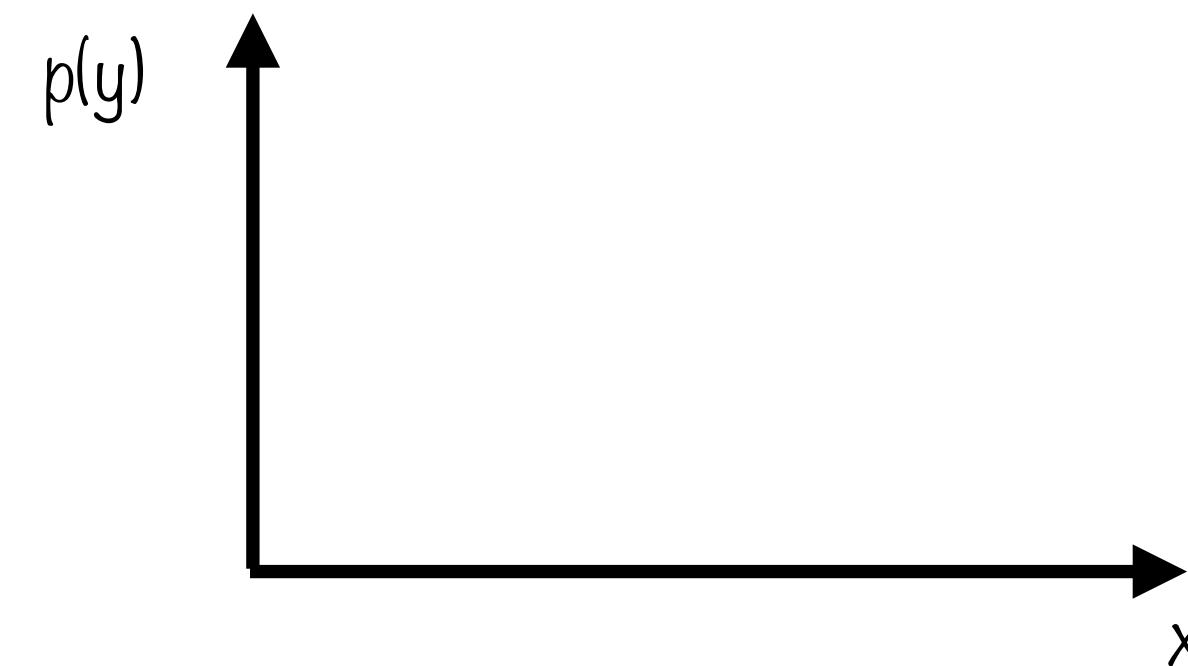
Linear Regression

Given causes, predict effect



Logistic Regression

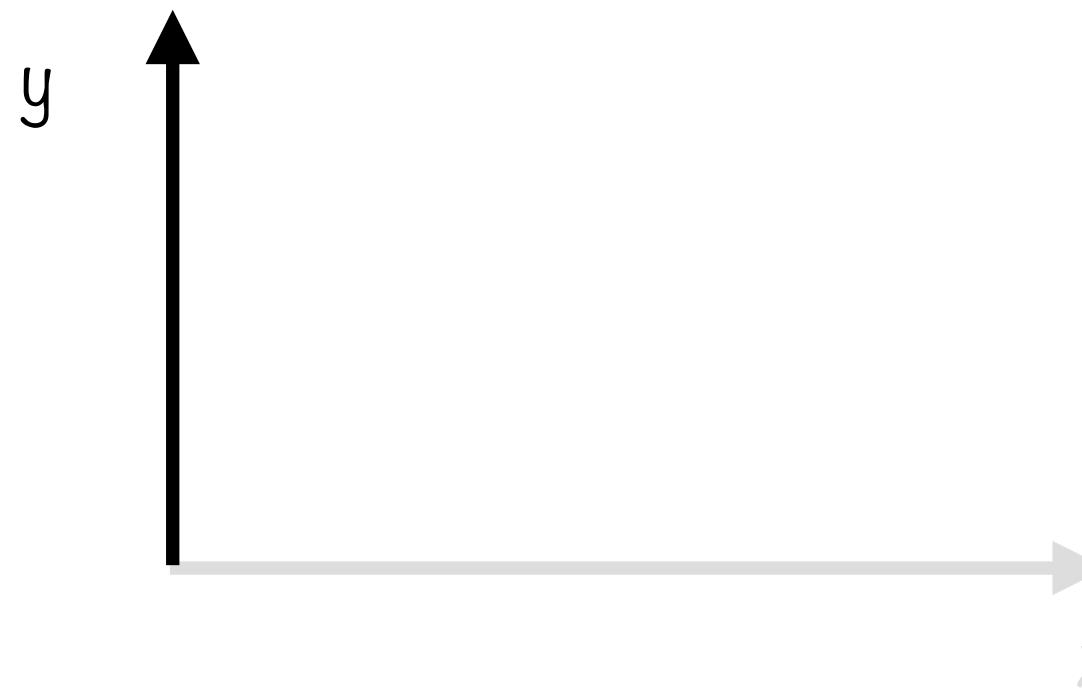
Given causes, predict probability of effect



Similar, yet Different

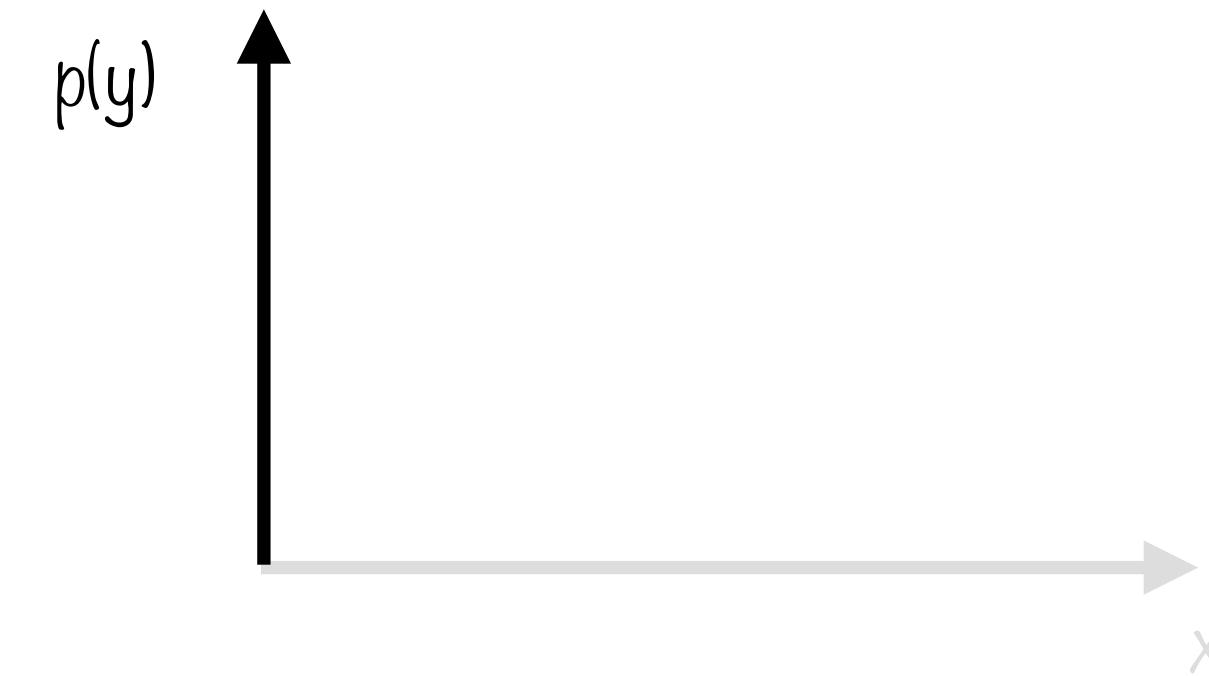
Linear Regression

Effect variable (y) must be continuous



Logistic Regression

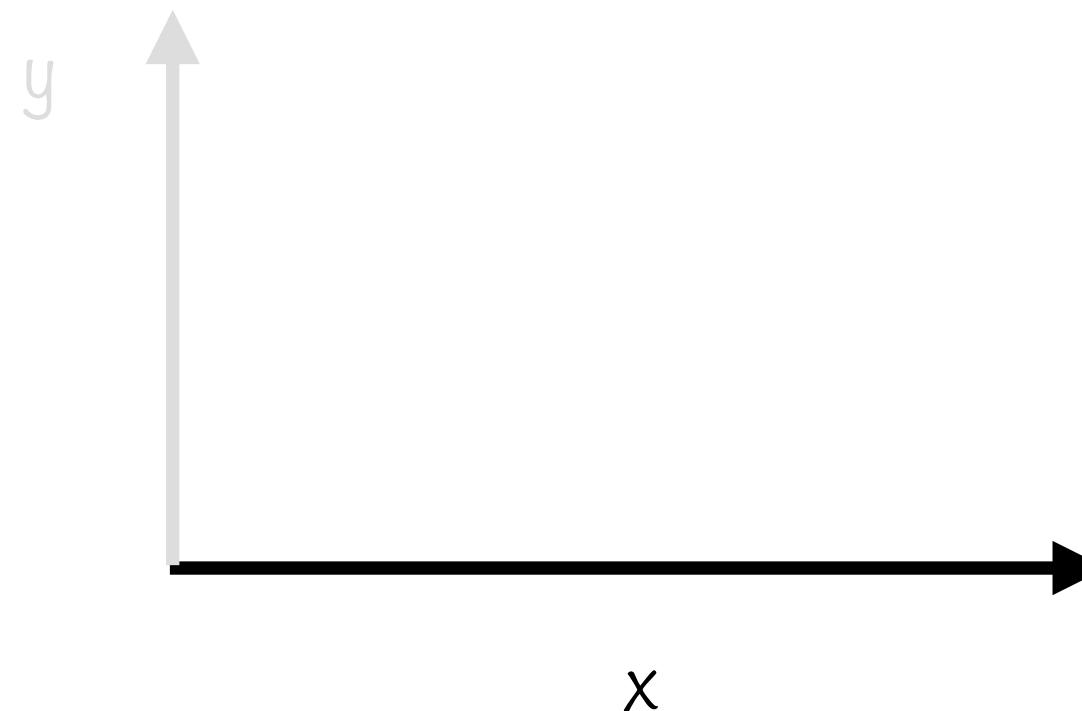
Effect variable (y) must be categorical



Similar, yet Different

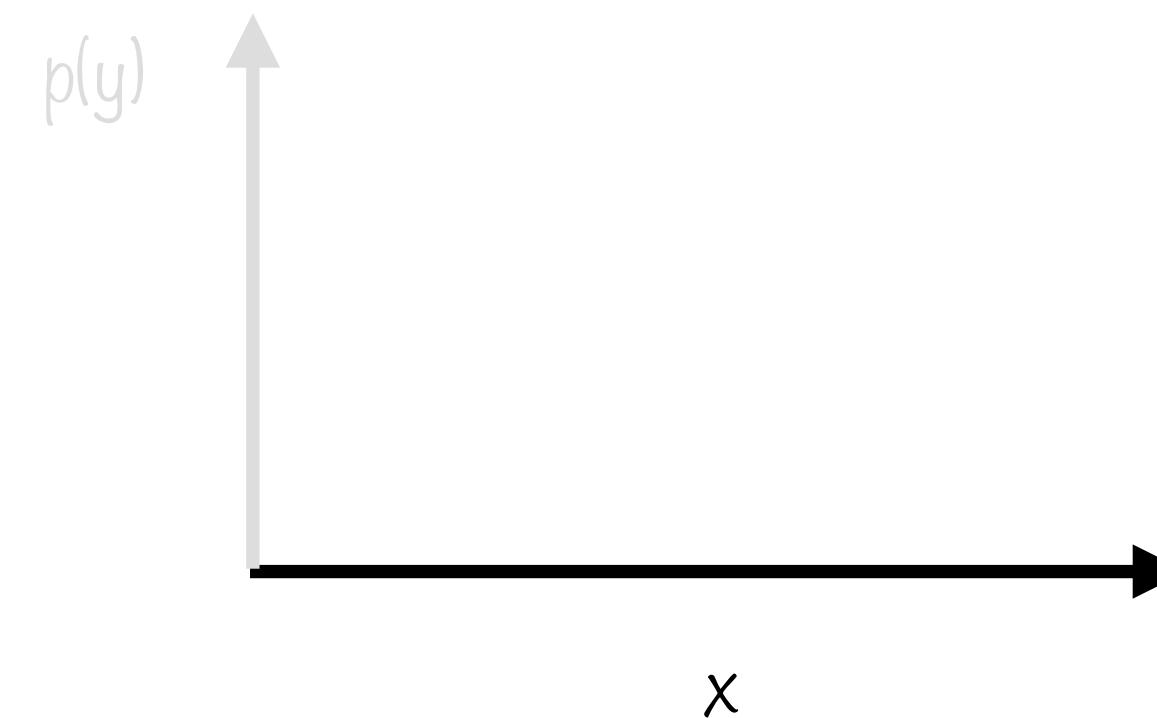
Linear Regression

Cause variables (x) can be continuous or categorical



Logistic Regression

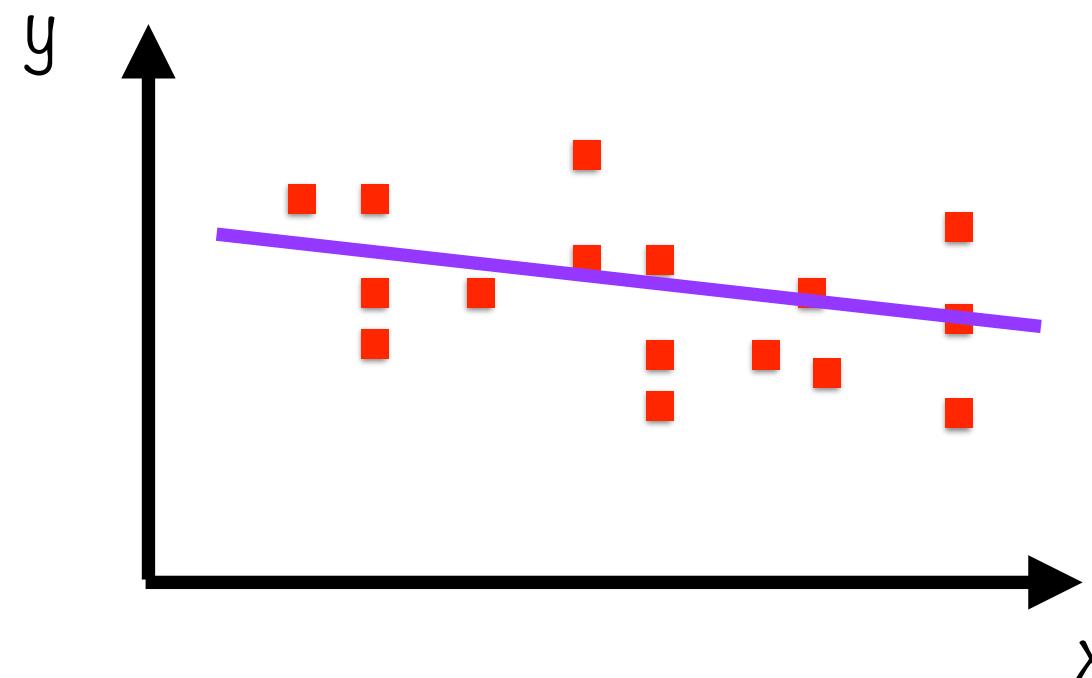
Cause variables (x) can be continuous or categorical



Similar, yet Different

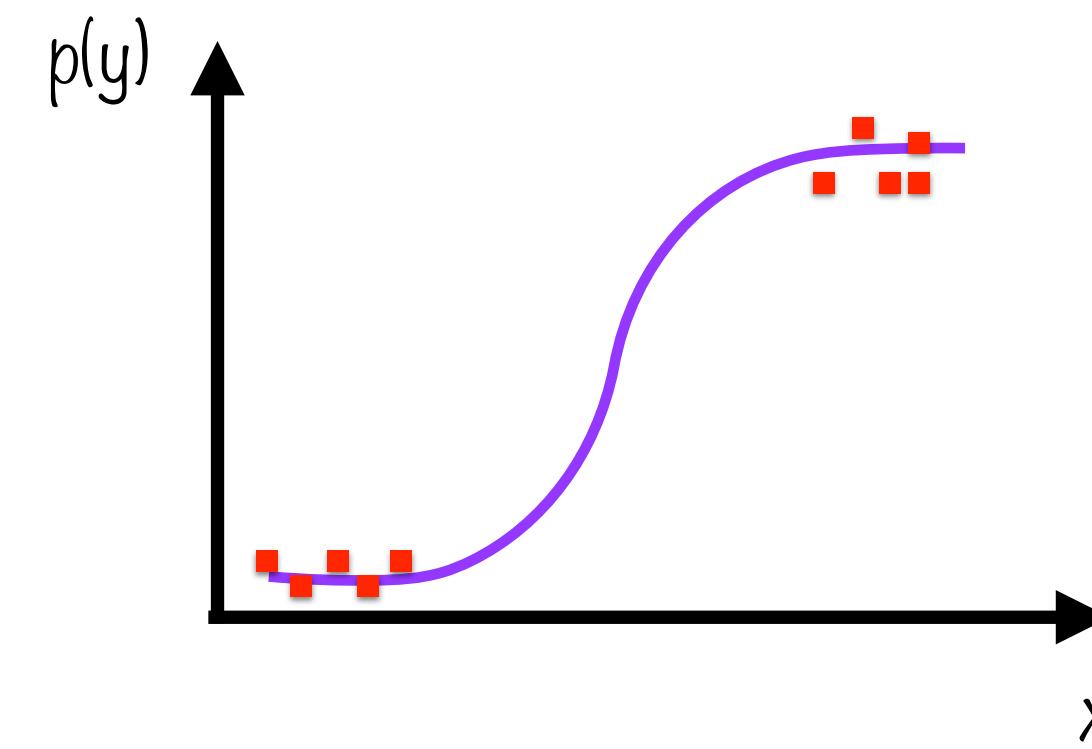
Linear Regression

Connect the dots with a straight line



Logistic Regression

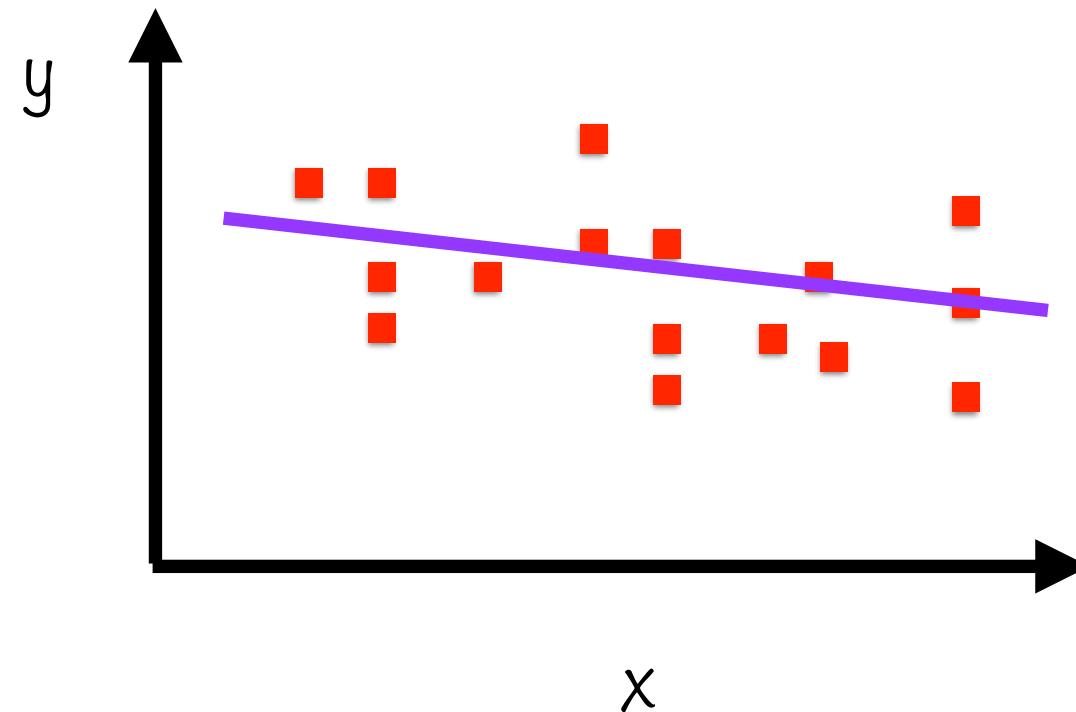
Connect the dots with an S-curve



Similar, yet Different

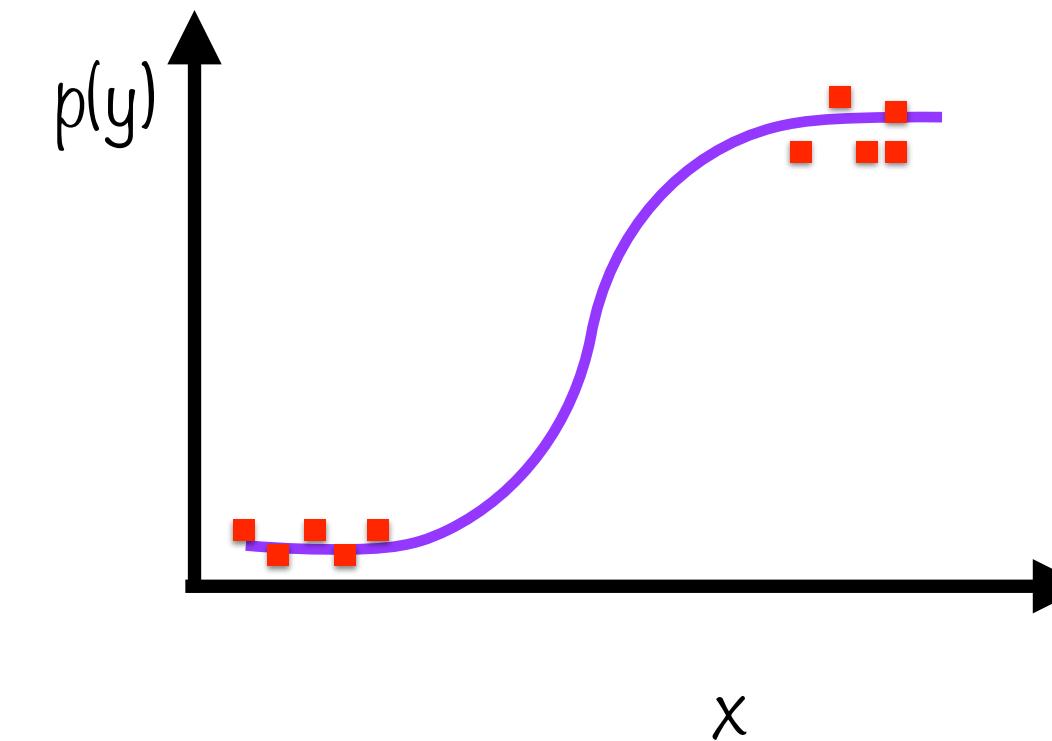
Linear Regression

$$y_i = A + Bx_i$$



Logistic Regression

$$p(y_i) = \frac{1}{1 + e^{-(A+Bx_i)}}$$



Similar, yet Different

Linear Regression

$$y_i = A + Bx_i$$

Objective of regression is to find A, B that “best fit” the data

Logistic Regression

$$p(y_i) = \frac{1}{1 + e^{-(A+Bx_i)}}$$

Objective of regression is to find A, B that “best fit” the data

Similar, yet Different

Linear Regression

$$y_i = A + Bx_i$$

Relationship is already linear (by assumption)

Logistic Regression

$$\ln\left(\frac{p(y_i)}{1 - p(y_i)}\right) = A + Bx_i$$

Relationship can be made linear (by log transformation)

Similar, yet Different

Linear Regression

$$y_i = A + Bx_i$$

Solve regression problem using cookie-cutter
solvers

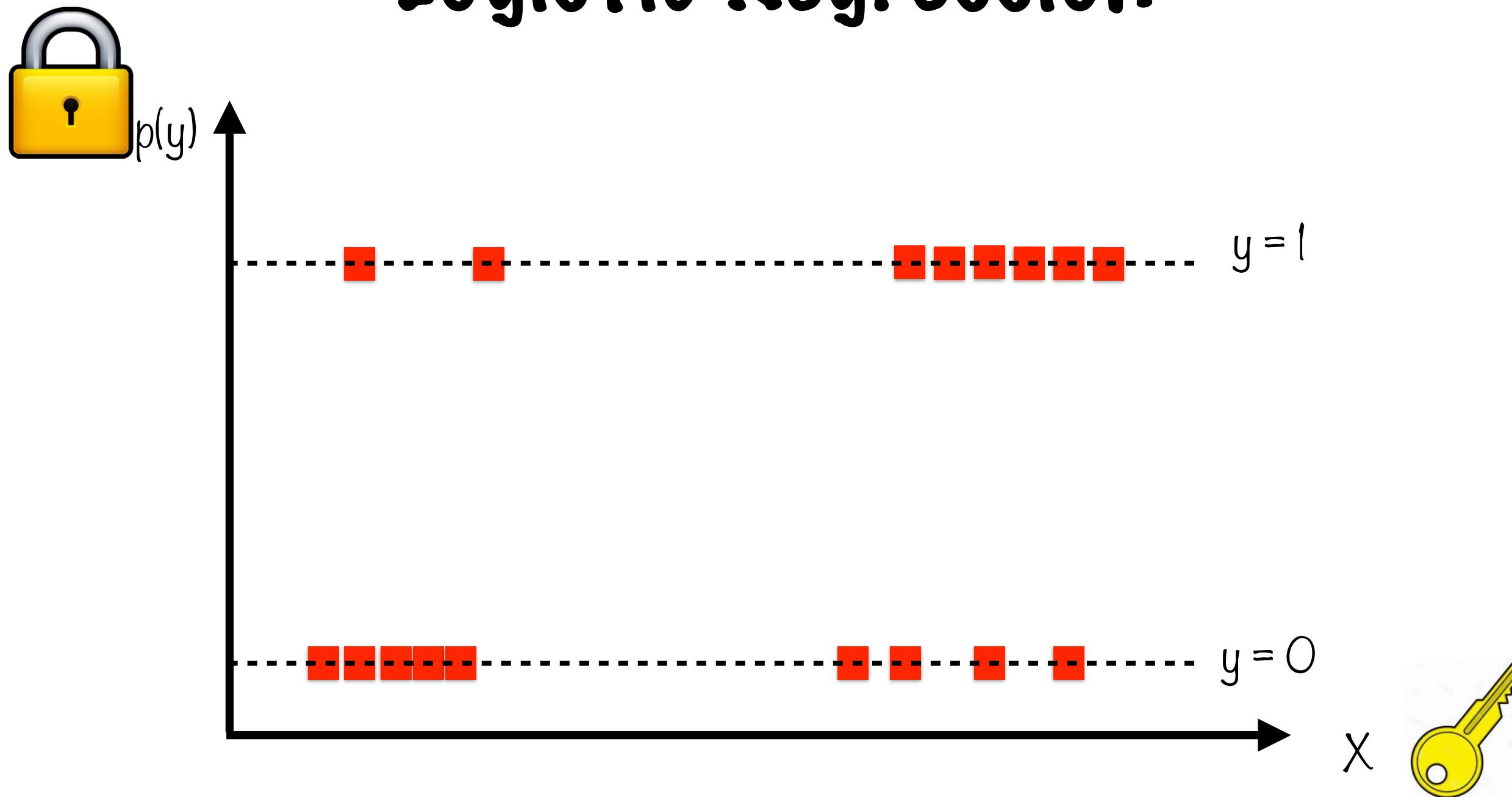
Logistic Regression

$$\text{logit}(p) = A + Bx_i$$

$$\text{logit}(p) = \ln\left(\frac{p}{1-p}\right)$$

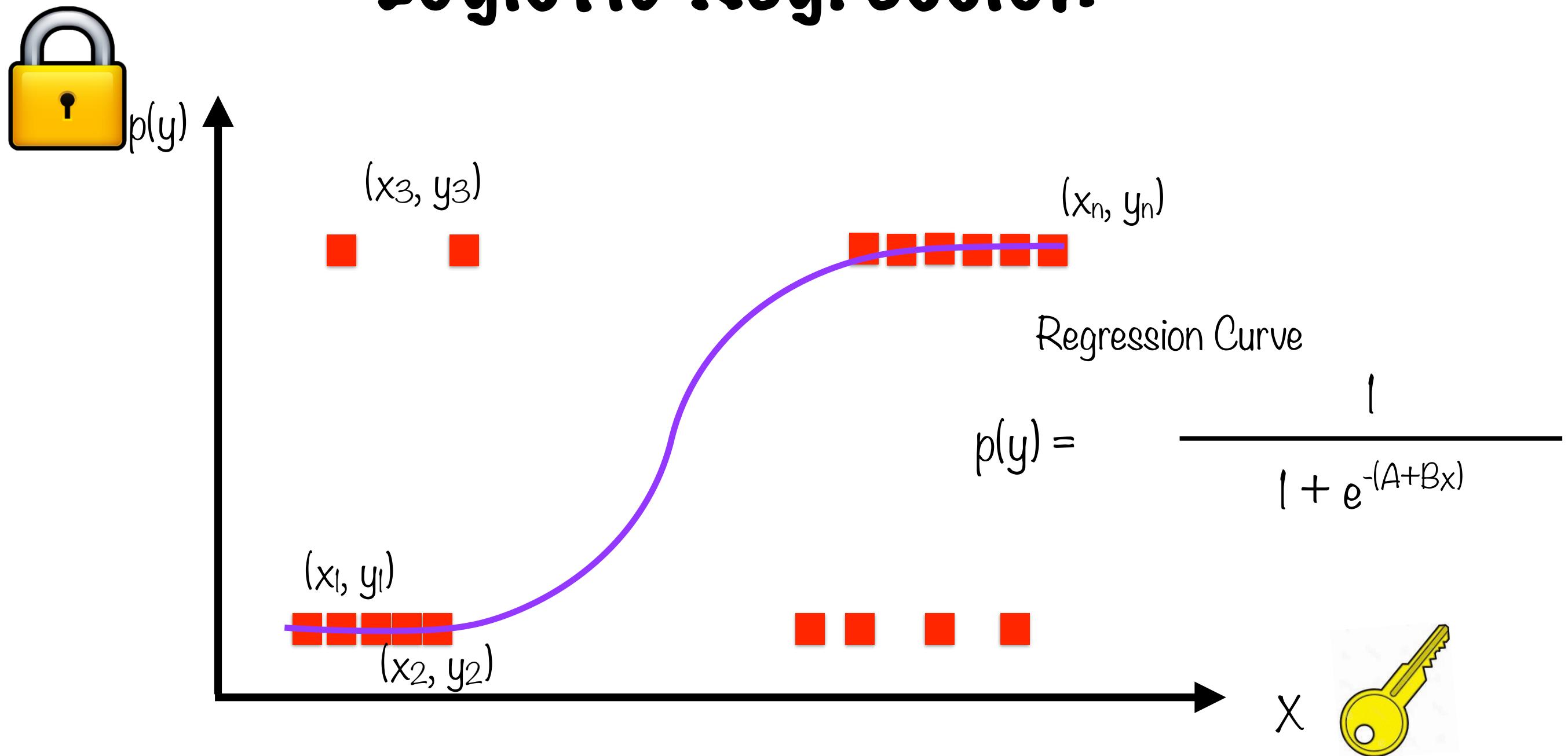
Solve regression problem using cookie-cutter
solvers

Logistic Regression



Represent all n points as (x_i, y_i) , where $i = 1$ to n

Logistic Regression



Represent all n points as (x_i, y_i) , where $i = 1$ to n

Linear Regression

$$y = A + Bx$$

$$y_1 = A + Bx_1$$

$$y_2 = A + Bx_2$$

$$y_3 = A + Bx_3$$

...

...

$$y_n = A + Bx_n$$

Logistic Regression

$$p(y) = \frac{1}{1 + e^{-(A+Bx)}}$$

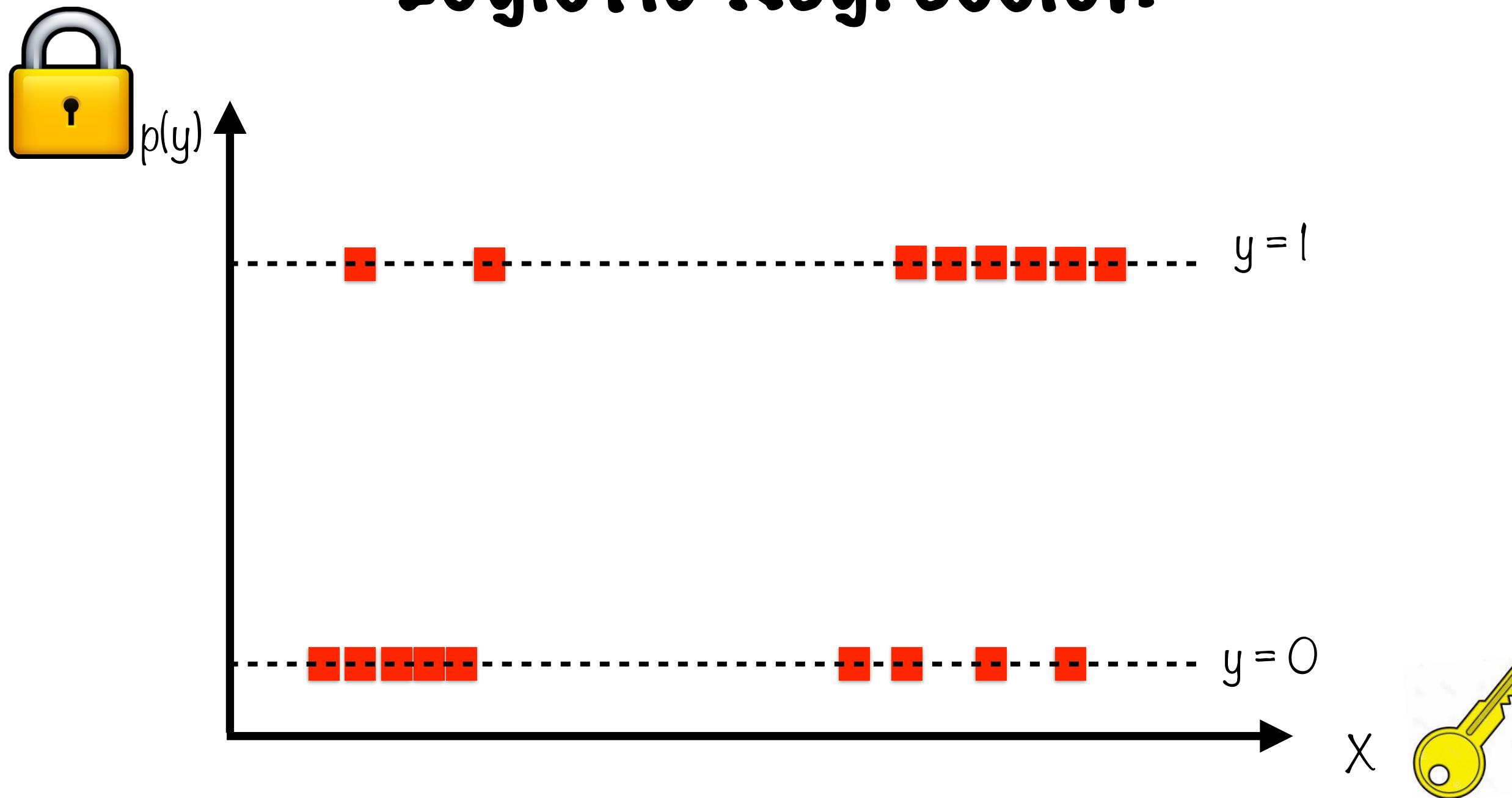
$$p(y_i) = \frac{1}{1 + e^{-(A+Bx_i)}}$$

$$p(y_1) = \frac{1}{1 + e^{-(A+Bx_1)}}$$

...

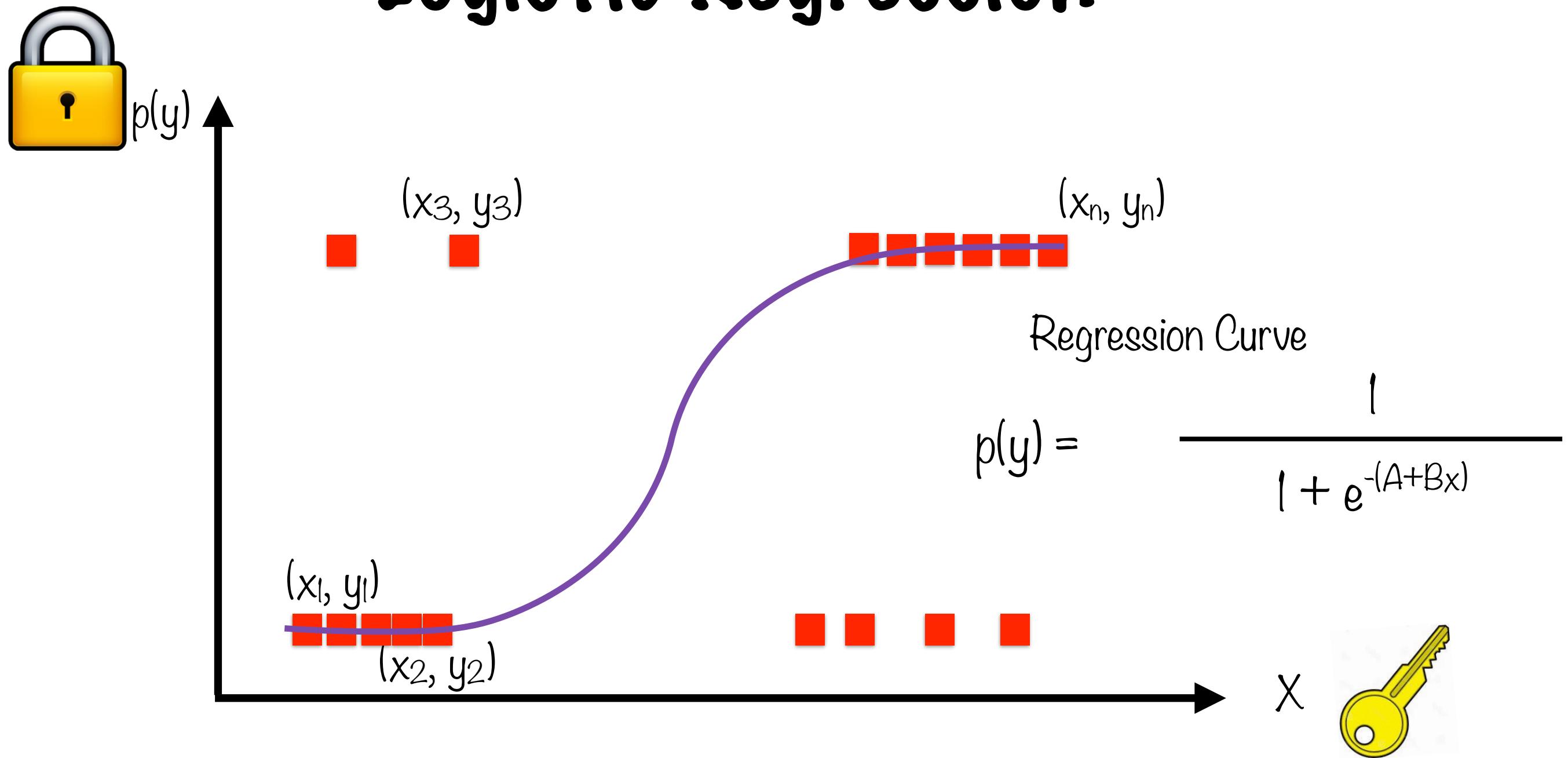
$$p(y_n) = \frac{1}{1 + e^{-(A+Bx_n)}}$$

Logistic Regression



Represent all n points as (x_i, y_i) , where $i = 1$ to n

Logistic Regression



Represent all n points as (x_i, y_i) , where $i = 1$ to n

Logistic Regression

Regression Equation:

$$p(y_i) = \frac{1}{1 + e^{-(A+Bx_i)}}$$

Solve for A and B that “best fit” the data

Odds from Probabilities

$$\text{Odds}(p) = \frac{p}{1-p}$$

Odds of an Event

$$p = \frac{1}{1 + e^{-(A+Bx)}}$$

$$p = \frac{e^{A+Bx}}{1 + e^{A+Bx}}$$

$$1 - p = 1 - \frac{e^{-A-Bx}}{1 + e^{-A-Bx}}$$

Odds of an Event

$$1-p = 1 - \frac{e^{A+Bx}}{1+e^{A+Bx}}$$

$$1-p = \frac{1+e^{A+Bx} - e^{A+Bx}}{1+e^{A+Bx}}$$

$$1-p = \frac{1}{1+e^{A+Bx}}$$

Odds of an Event

$$p = \frac{e^{A+Bx}}{1+e^{A+Bx}}$$

$$1-p = \frac{1}{1+e^{A+Bx}}$$

$$\text{Odds}(p) = \frac{p}{1-p} = e^{A+Bx}$$

Logit Is Linear

$$\text{Odds}(p) = \frac{p}{1-p} = e^{A+Bx}$$

$$\text{logit}(p) = A + Bx$$

$\ln(\text{Odds}(p))$ is called the logit function

Logit Is Linear

$$\ln \text{Odds}(p) = \ln(p) - \ln(1-p)$$

$$p = \frac{1}{1 + e^{-(A+Bx)}}$$

$$\text{logit}(p) = \ln \text{Odds}(p) = A + Bx$$

This is a linear function!

Logistic Regression can be solved via linear regression on the logit function (log of the odds function)

Logistic Regression in TensorFlow

Logistic Regression



Cause

Changes in S&P 500



Effect

Changes in price of Google Stock

Logistic Regression

y = Returns on
Google stock
(GOOG)

x = Returns on
S&P 500
(S&P500)

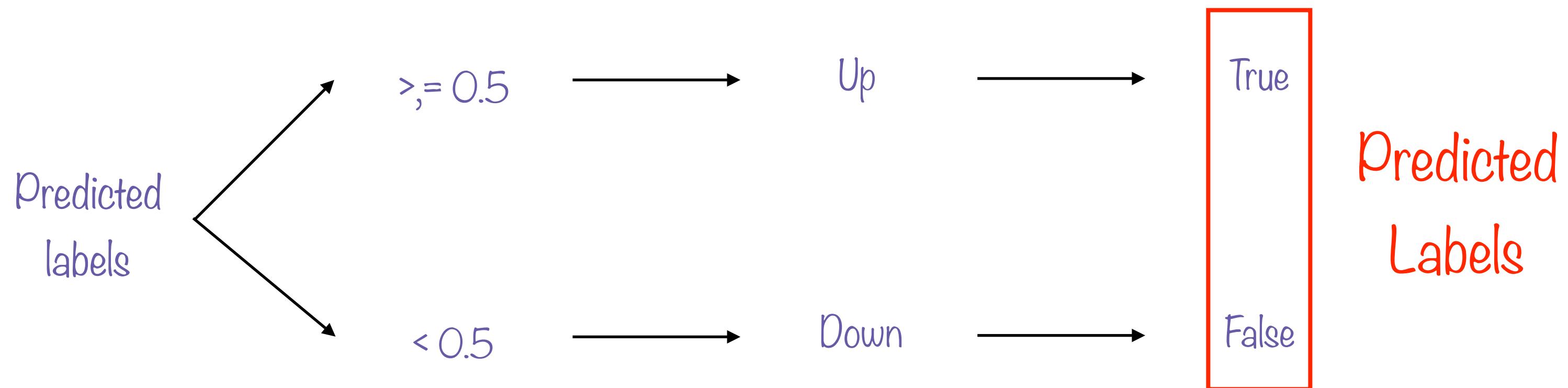
Logistic Regression

$$p(y_i) = \frac{1}{1 + e^{-(A+Bx_i)}}$$

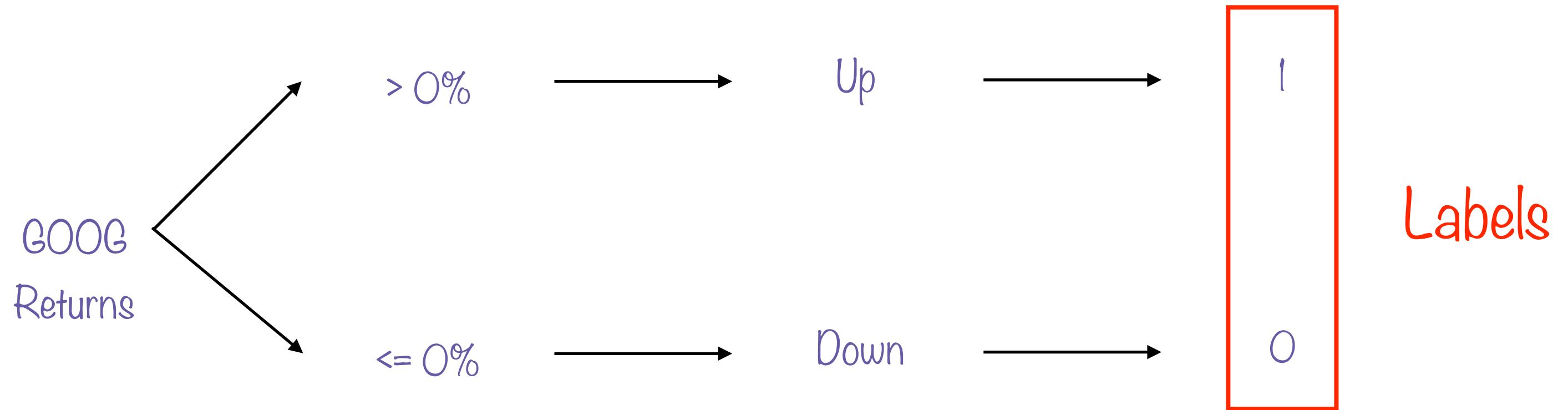
$p(y)$ = Probability of Google
going up in the current month i

x = Returns on S&P 500
for current month

Logistic Regression



Set up the Problem



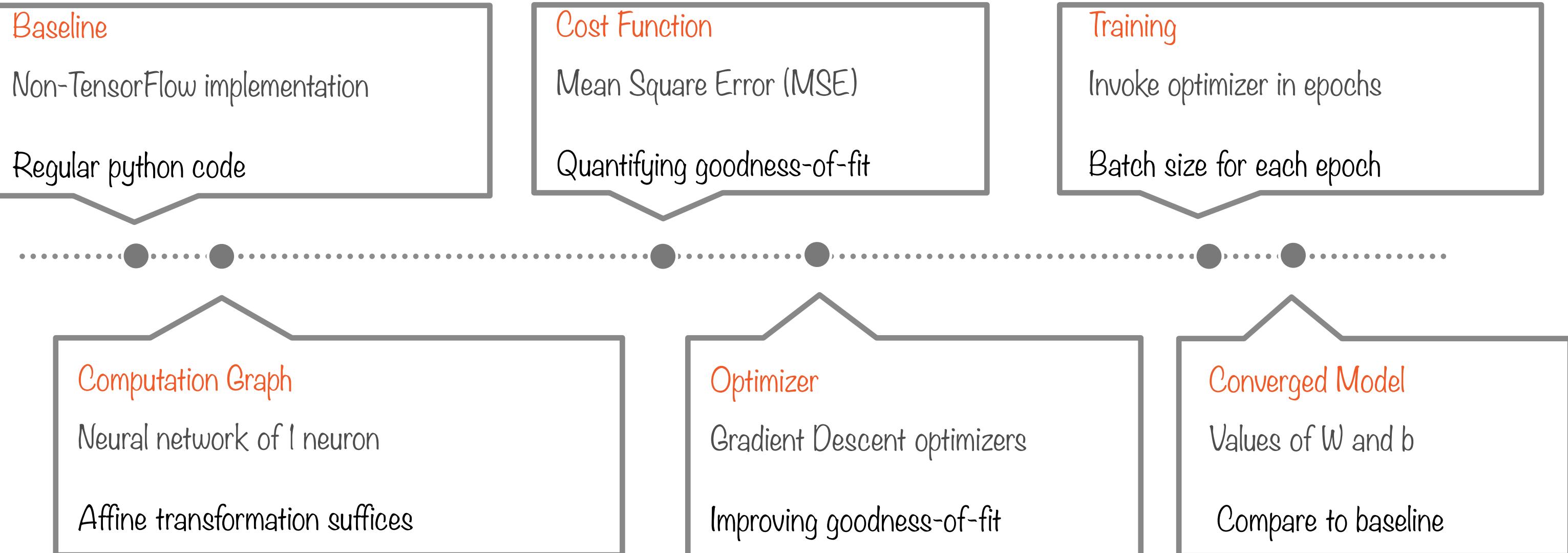
Label GOOG returns as binary (1,0)

Prediction Accuracy

| DATE | ACTUAL | PREDICTED |
|------------|--------|-----------|
| 2005-01-01 | NA | NA |
| 2005-02-01 | 0 | 1 |
| 2005-03-01 | 0 | 0 |
| | | |
| 2017-01-01 | 1 | 1 |
| 2017-02-01 | 1 | 1 |

Compare GOOG's actual labels vs. predicted labels

Linear Regression in TensorFlow



Linear Regression in TensorFlow

Baseline
Non-TensorFlow implementation
Regular python code

Cost Function
Mean Square Error (MSE)
Quantifying goodness-of-fit

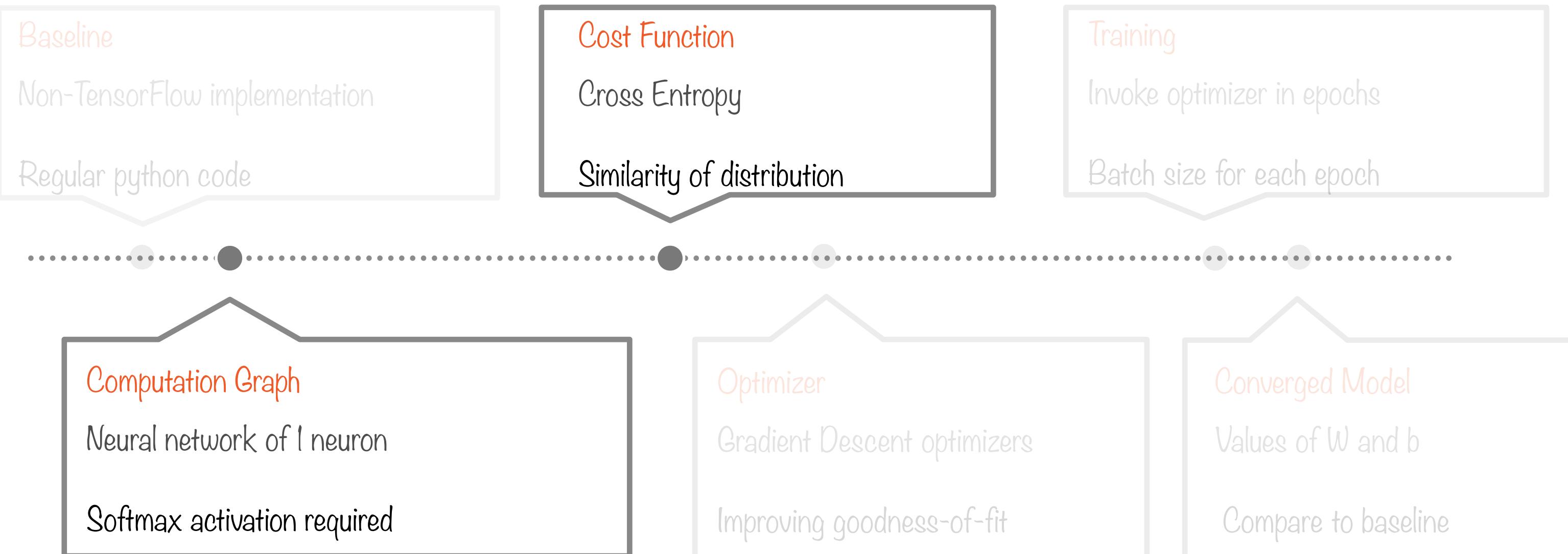
Training
Invoke optimizer in epochs
Batch size for each epoch

Computation Graph
Neural network of 1 neuron
Affine transformation suffices

Optimizer
Gradient Descent optimizers
Improving goodness-of-fit

Converged Model
Values of W and b
Compare to baseline

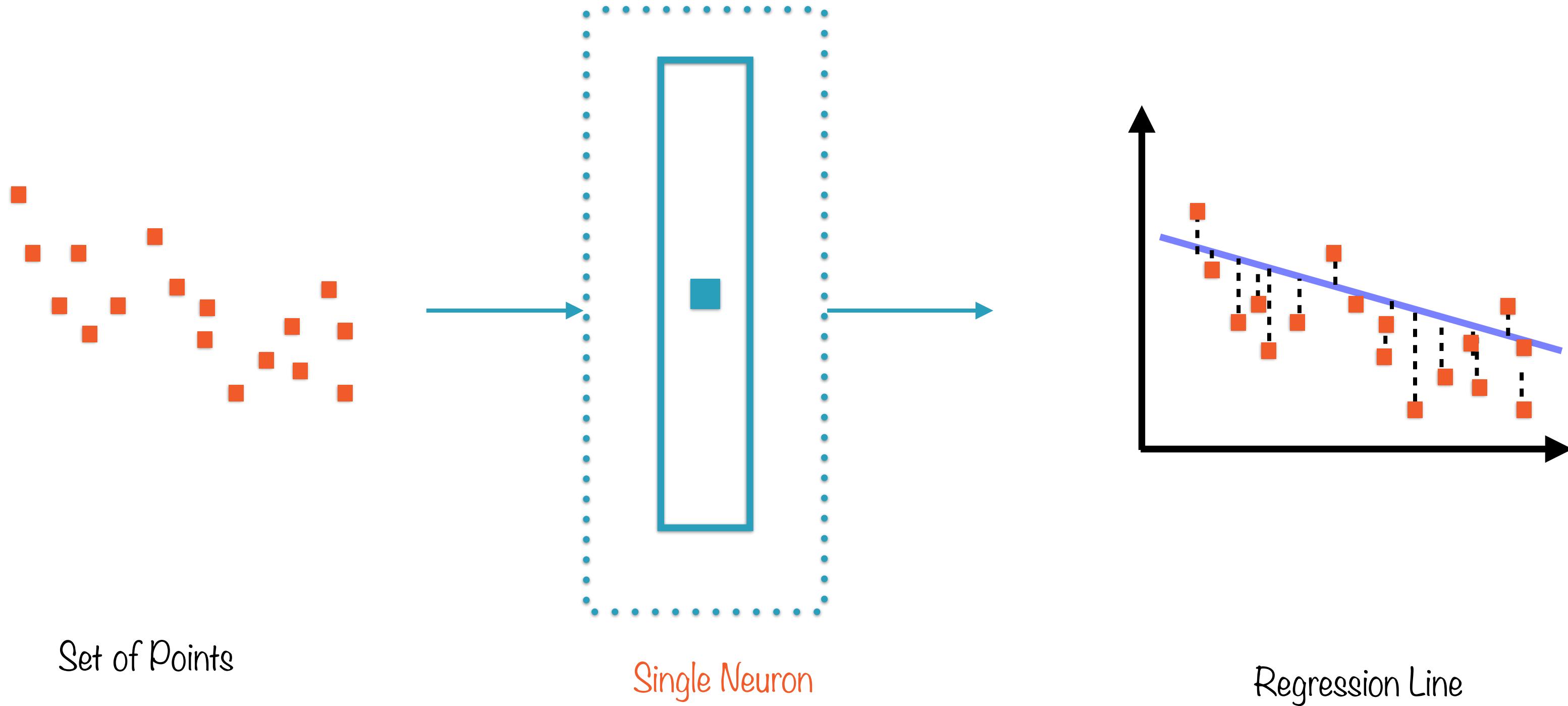
Logistic Regression in TensorFlow



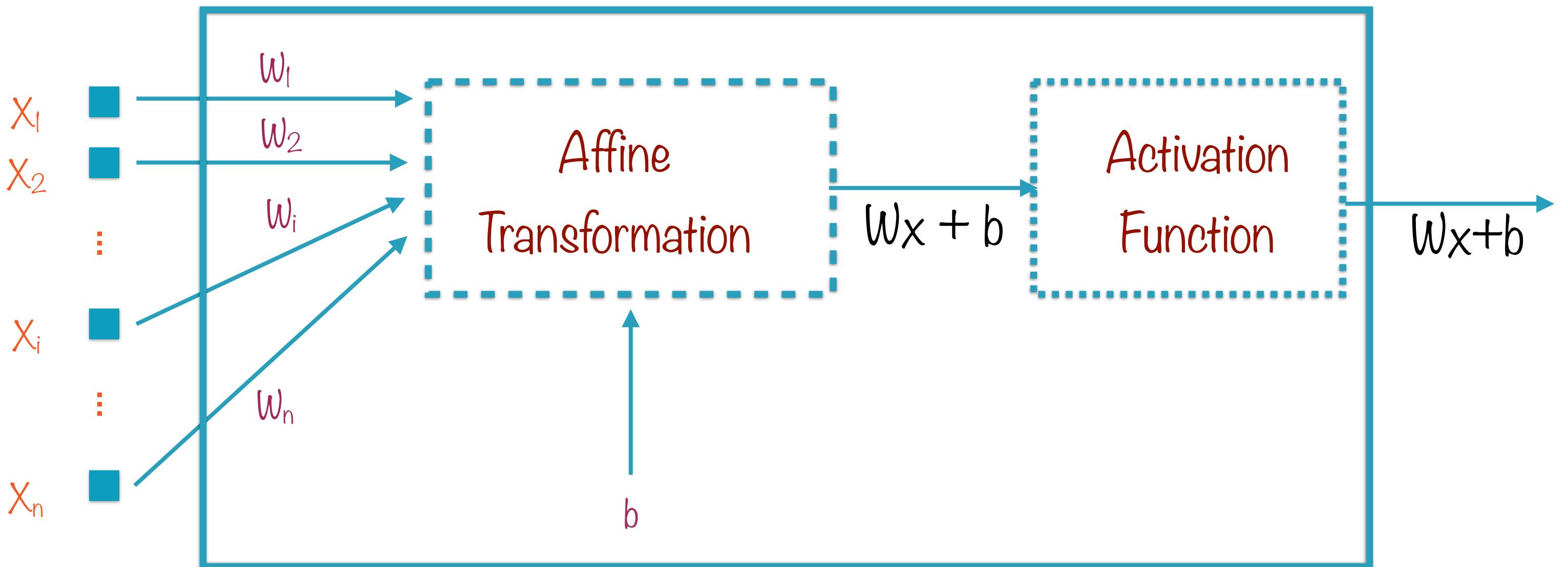
Logistic Regression in TensorFlow



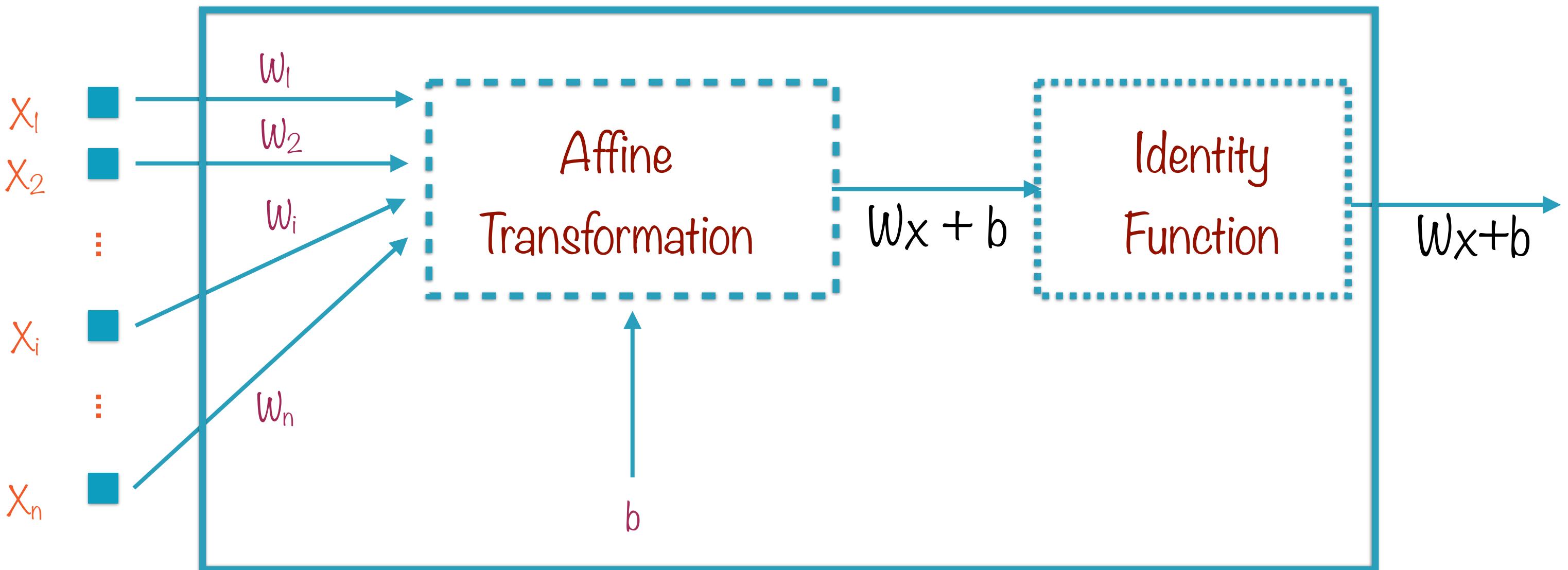
Linear Regression with One Neuron



Linear Regression with One Neuron



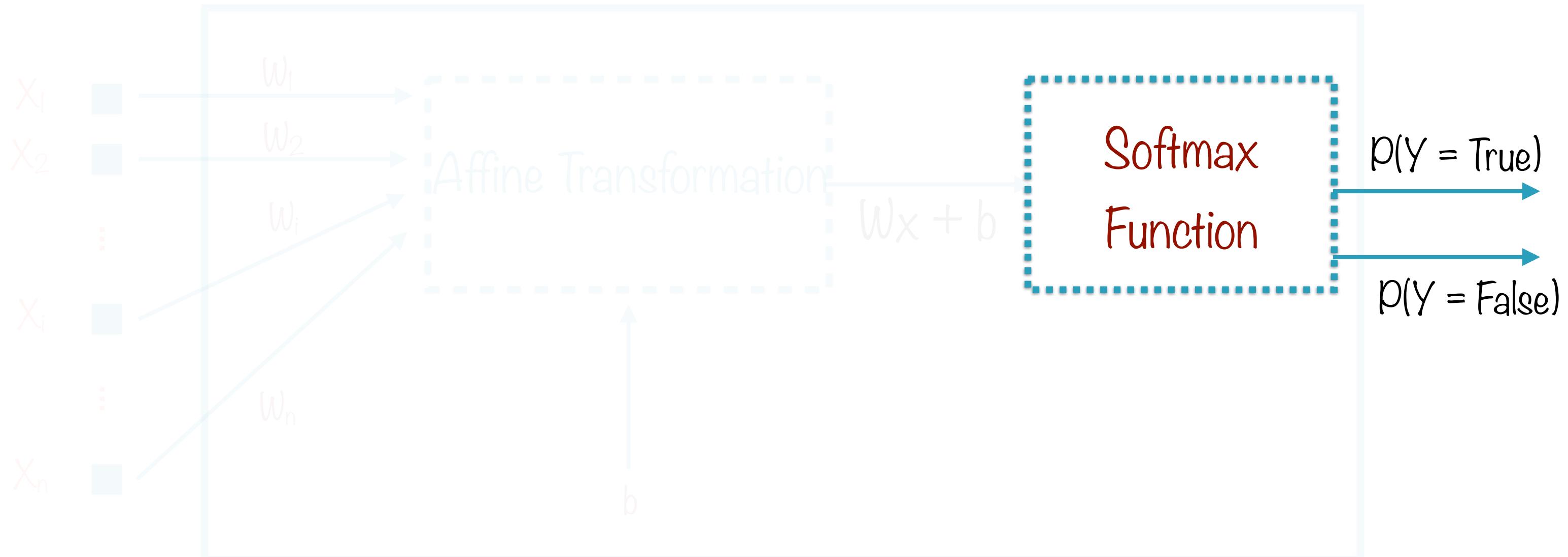
Linear Regression with One Neuron



Linear Regression with One Neuron



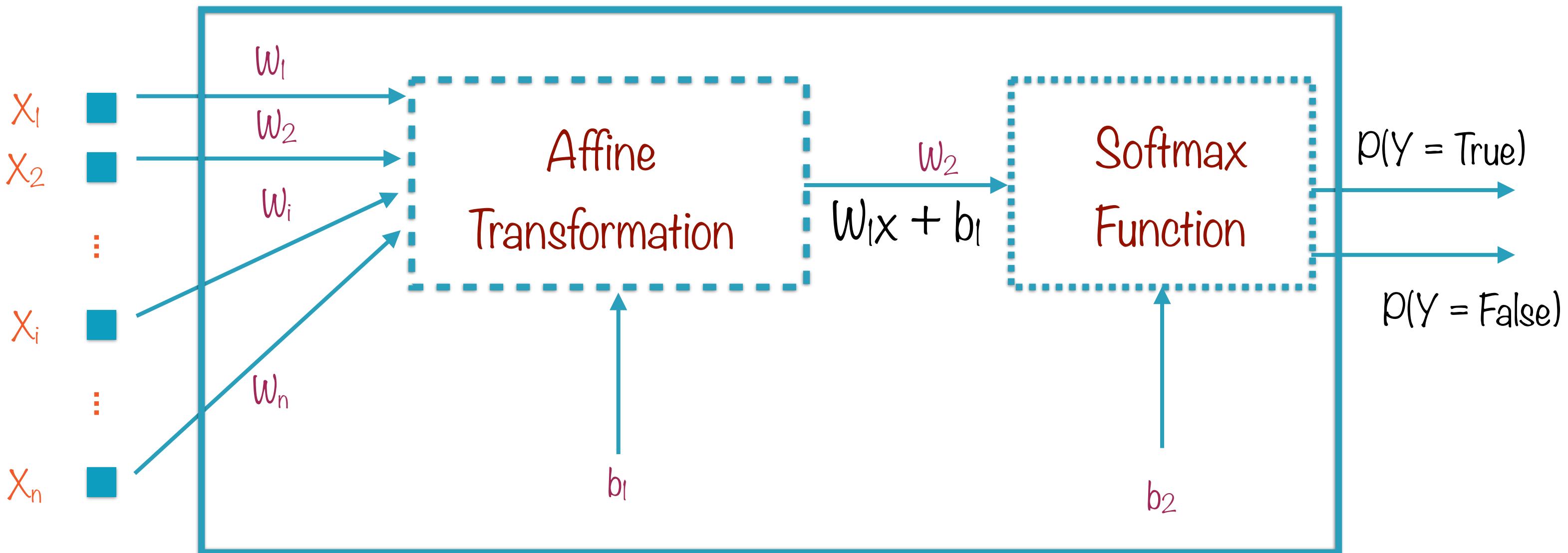
Logistic Regression with One Neuron



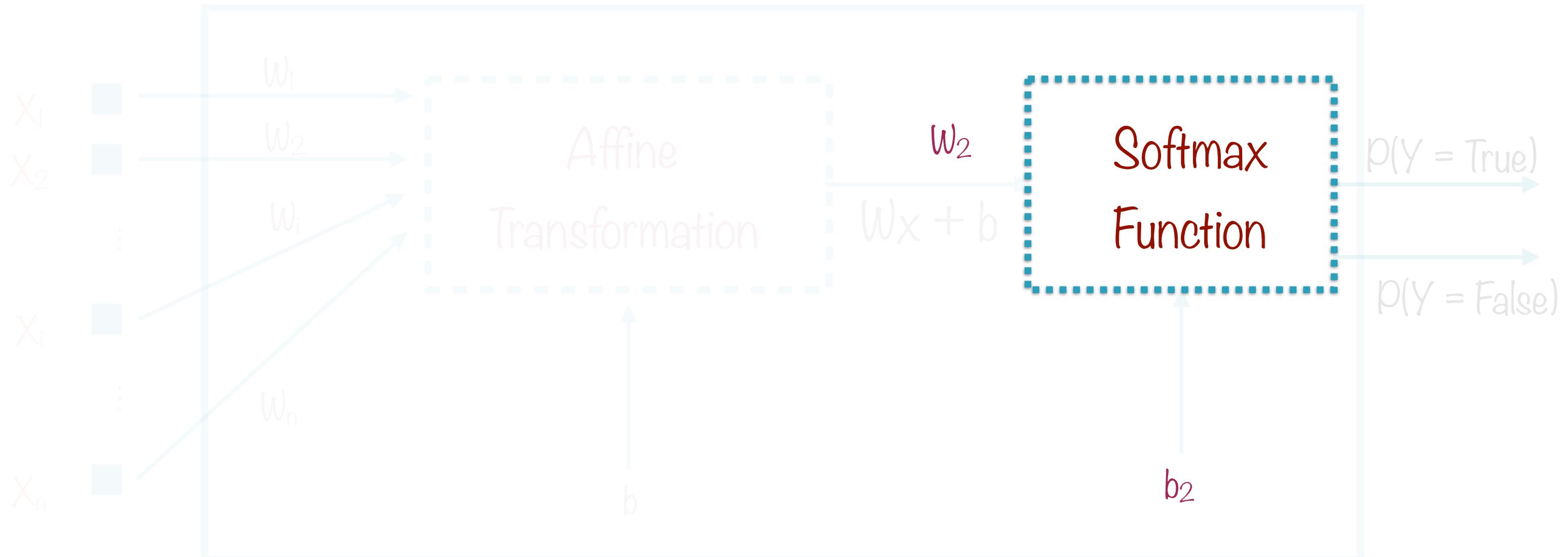
Logistic Regression with One Neuron



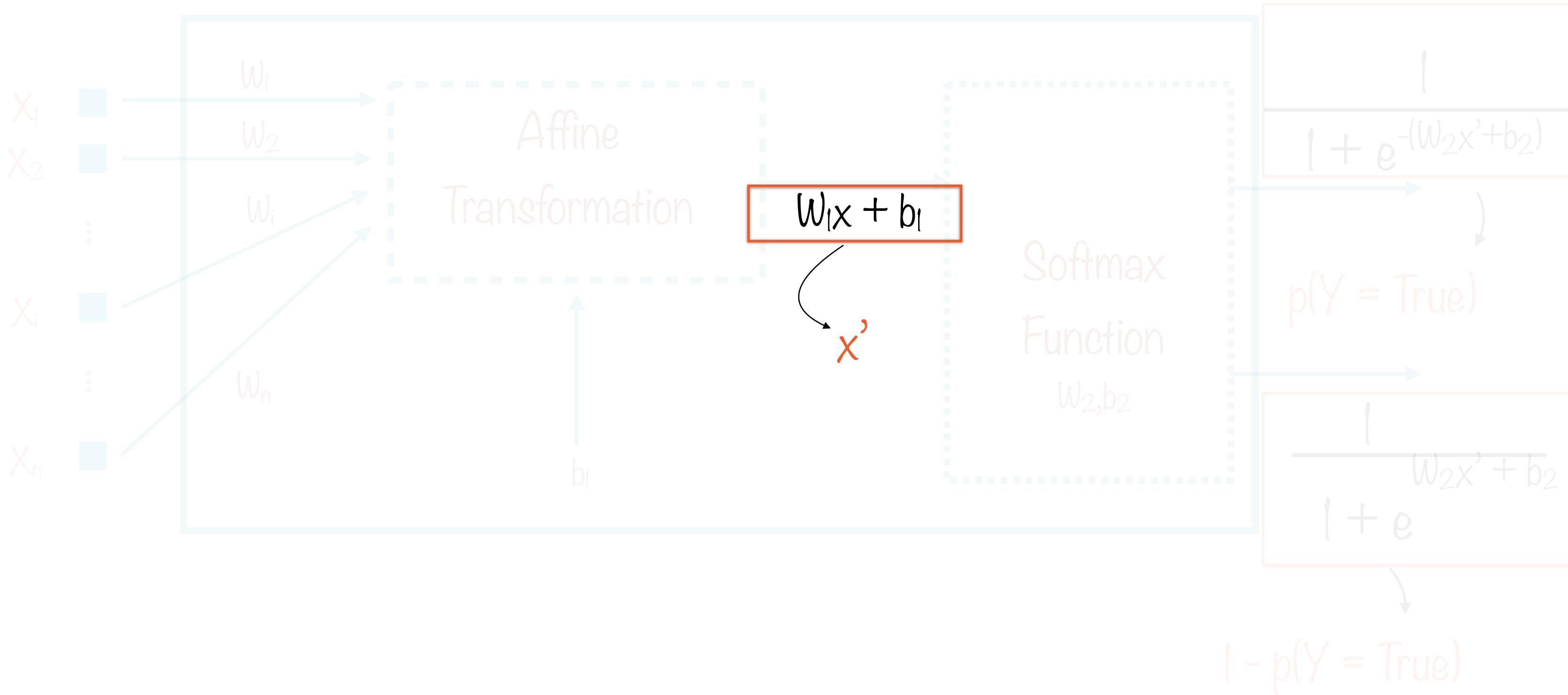
Logistic Regression with One Neuron



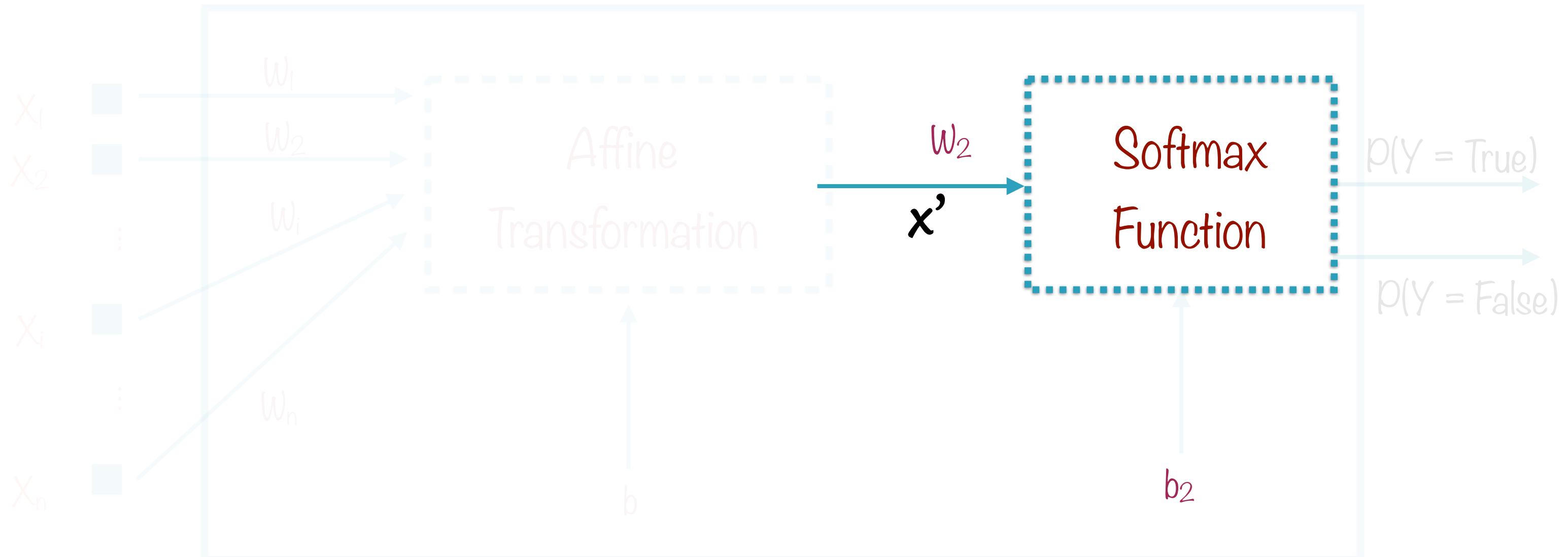
Logistic Regression with One Neuron



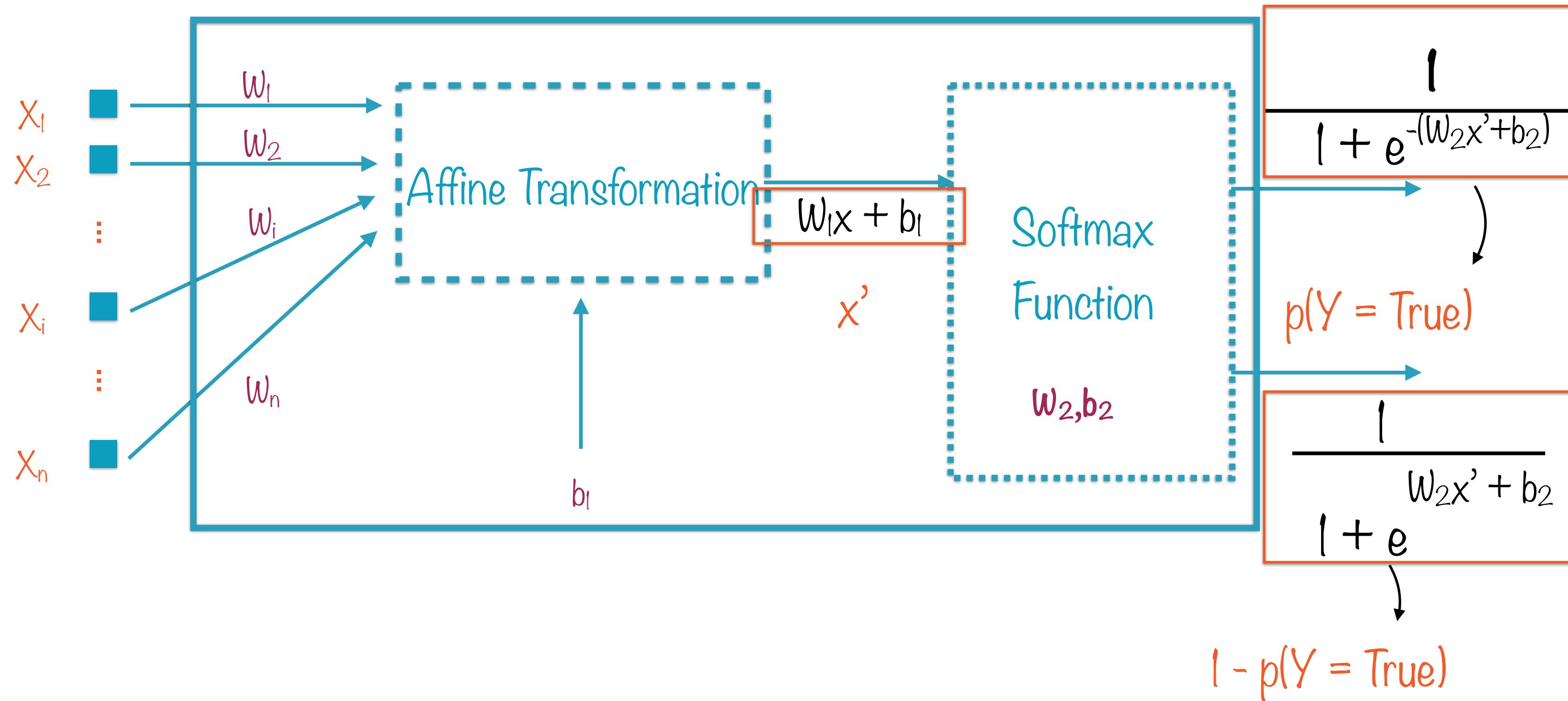
Logistic Regression with One Neuron



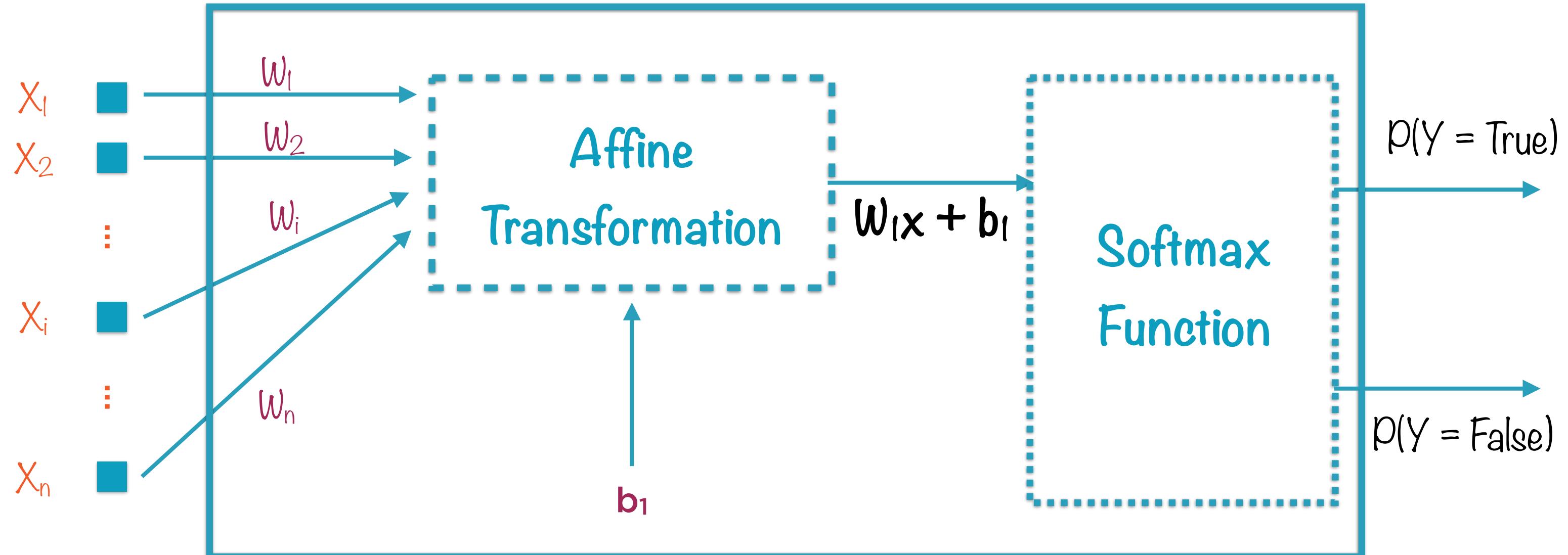
Logistic Regression with One Neuron



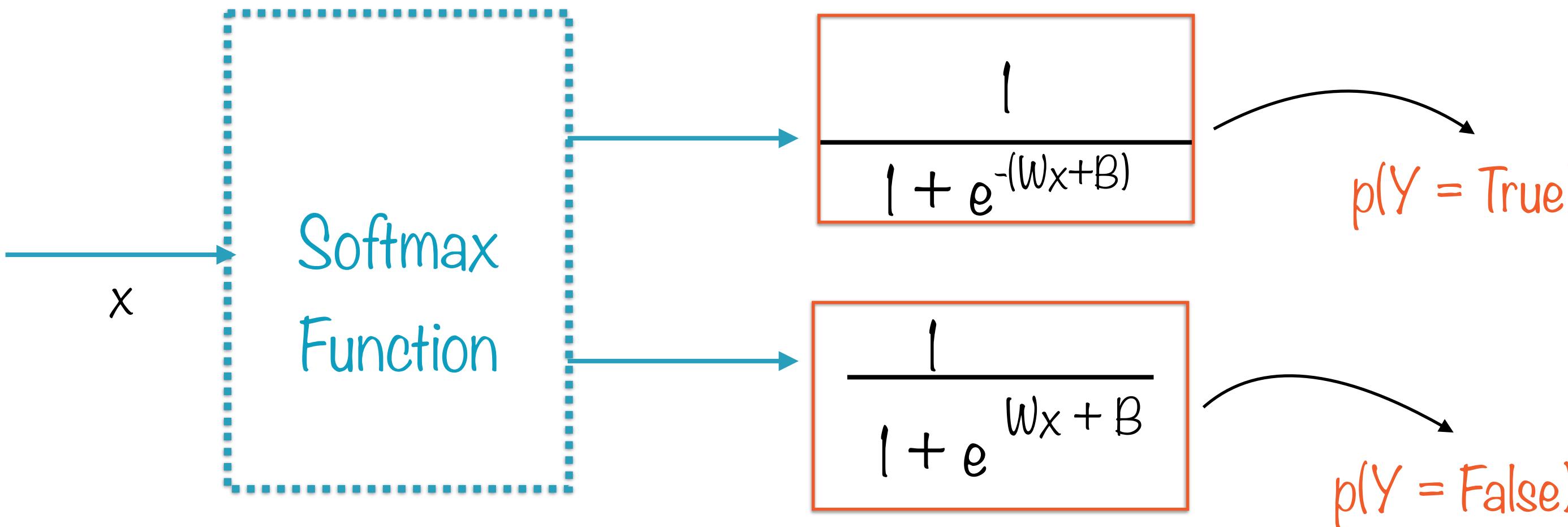
Logistic Regression with One Neuron



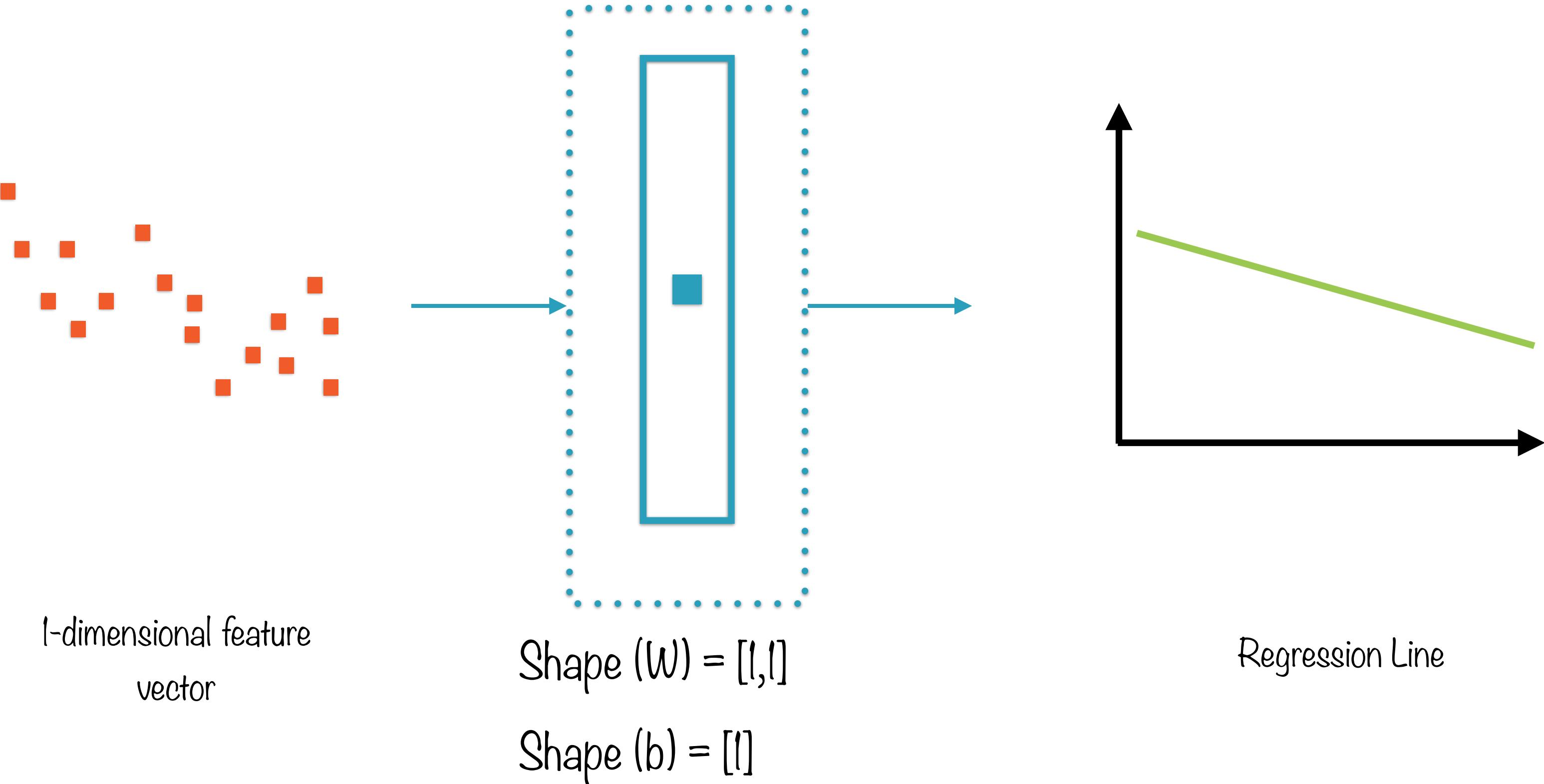
Logistic Regression with One Neuron



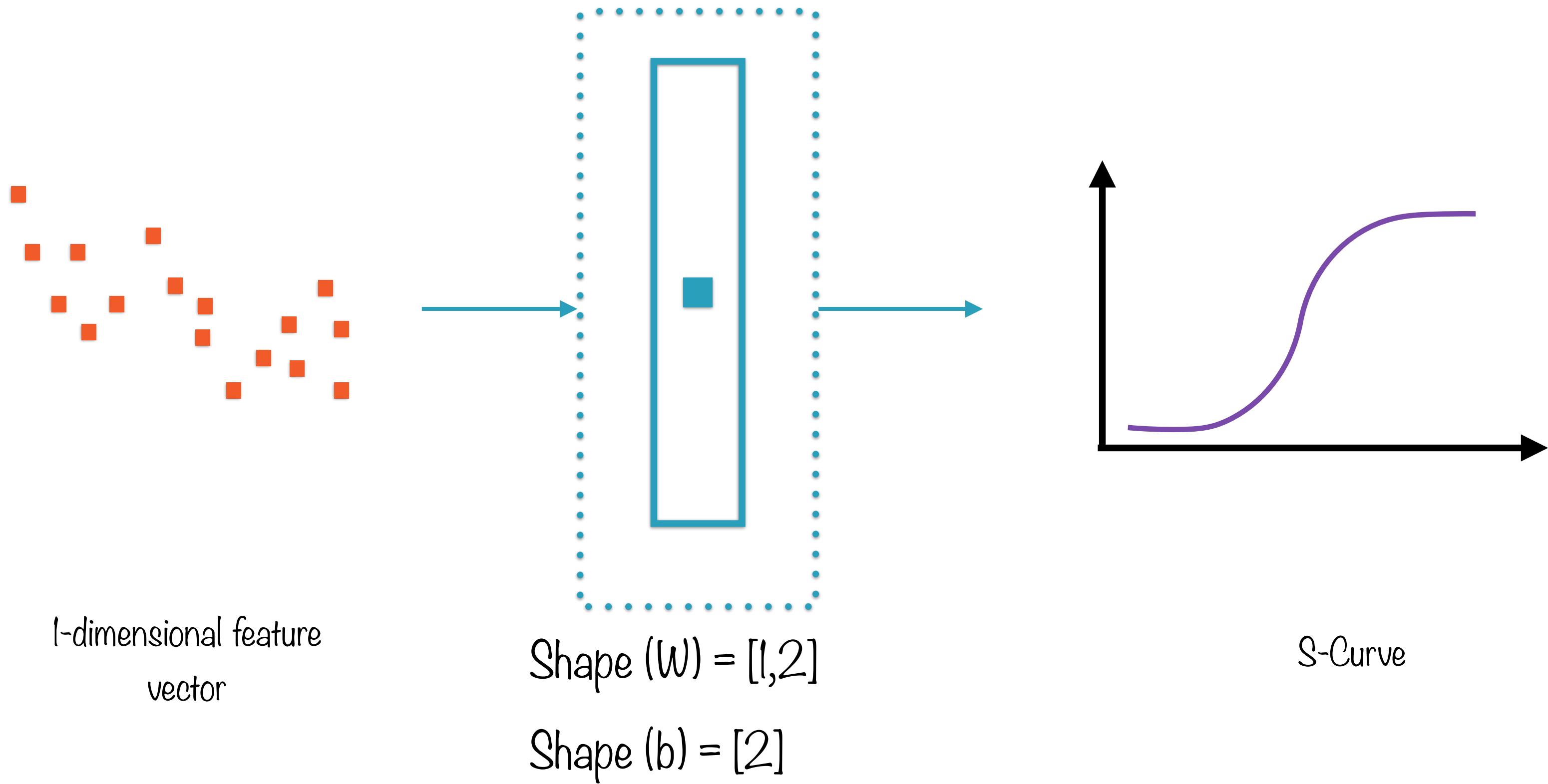
SoftMax for True/False Classification



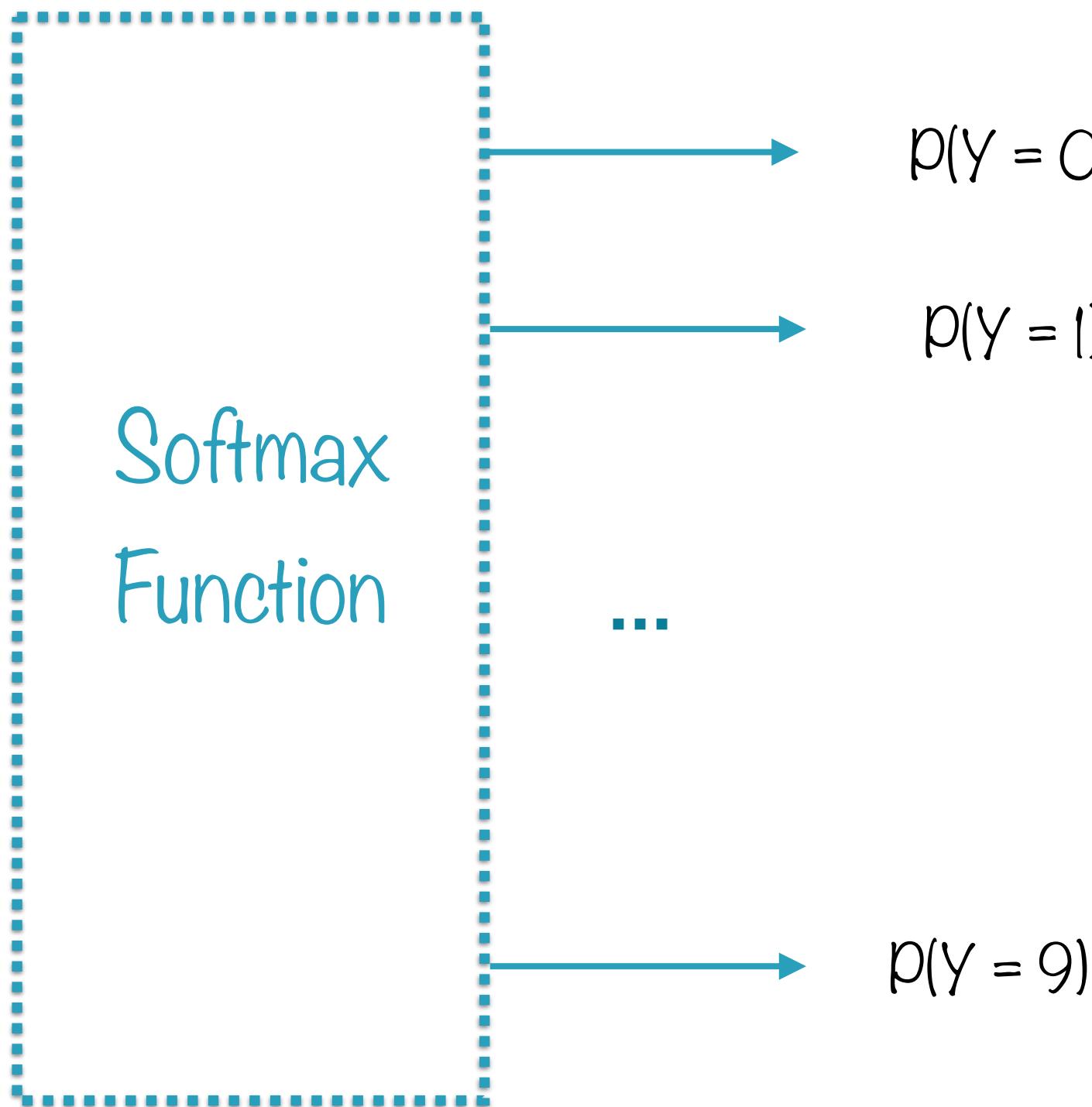
Linear Regression with One Neuron



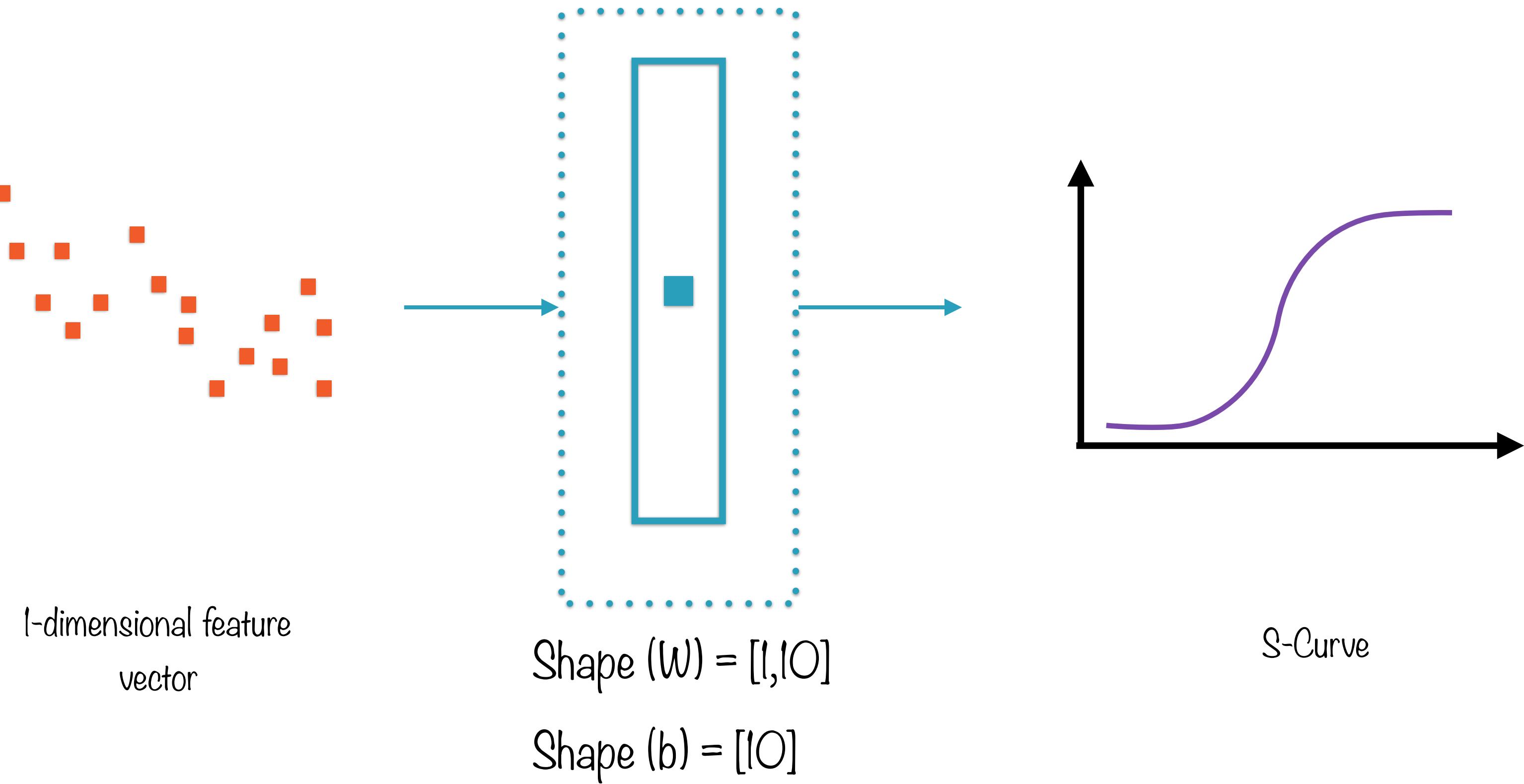
Logistic Regression with One Neuron



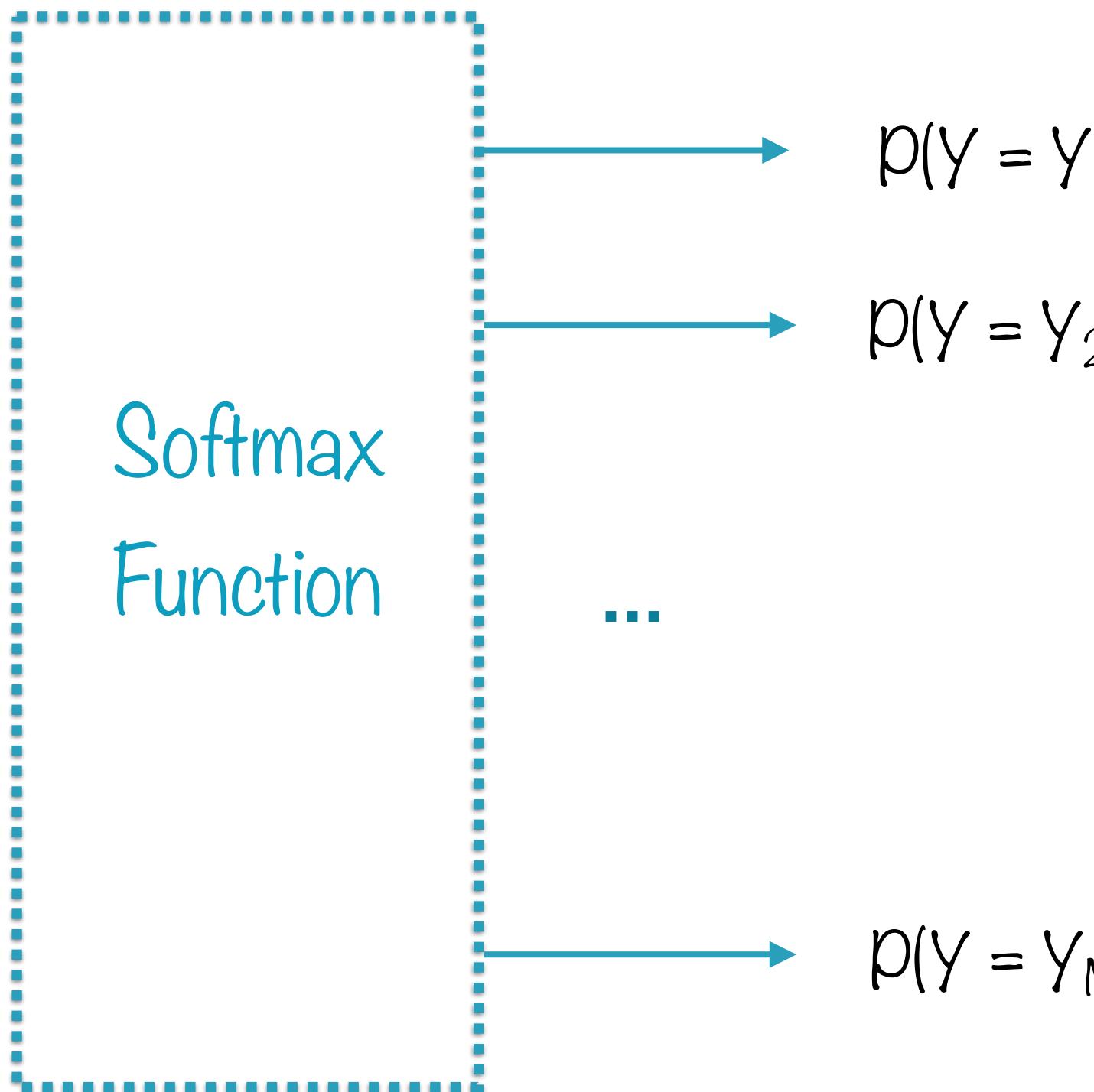
SoftMax for Digit Classification



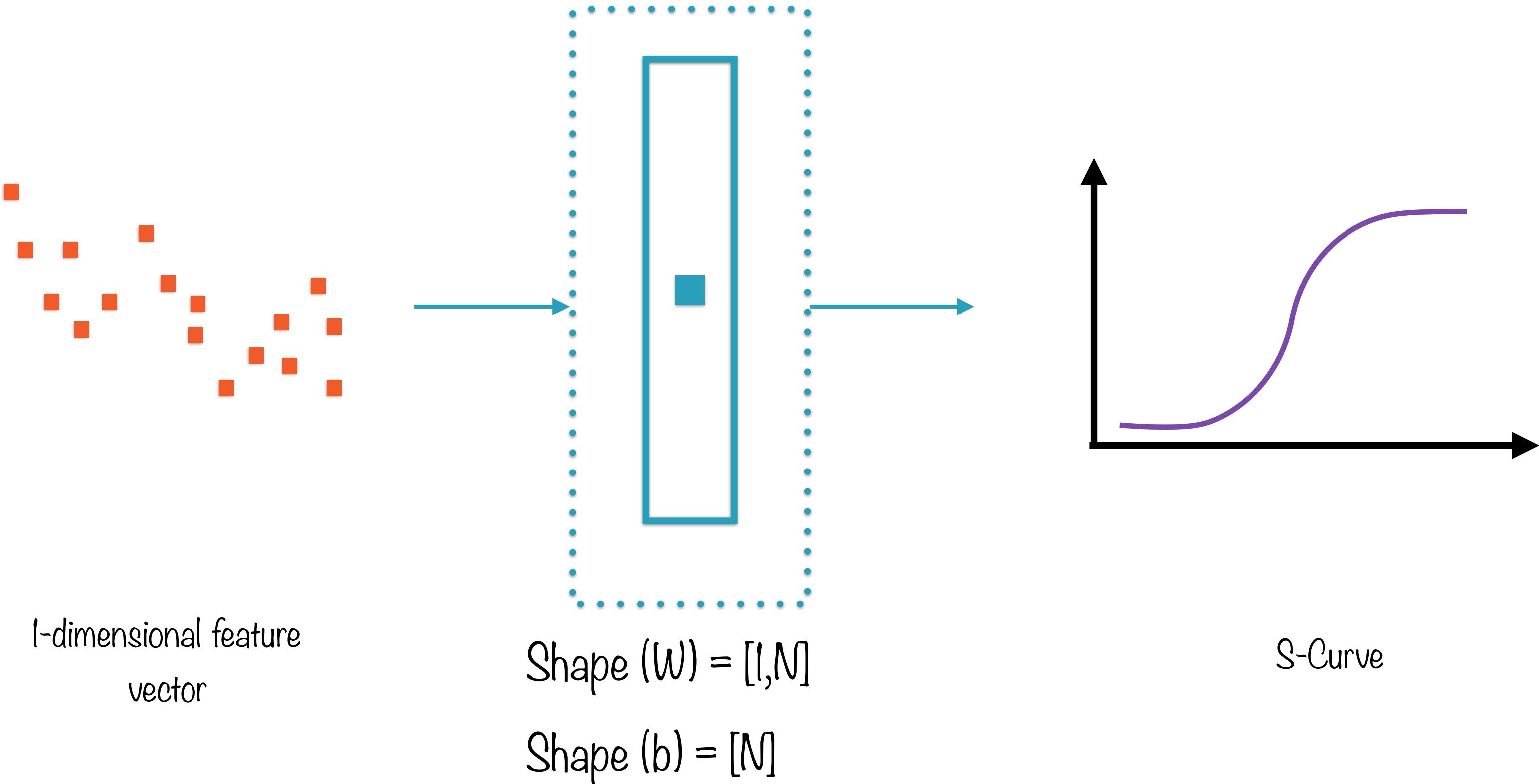
SoftMax for Digit Classification



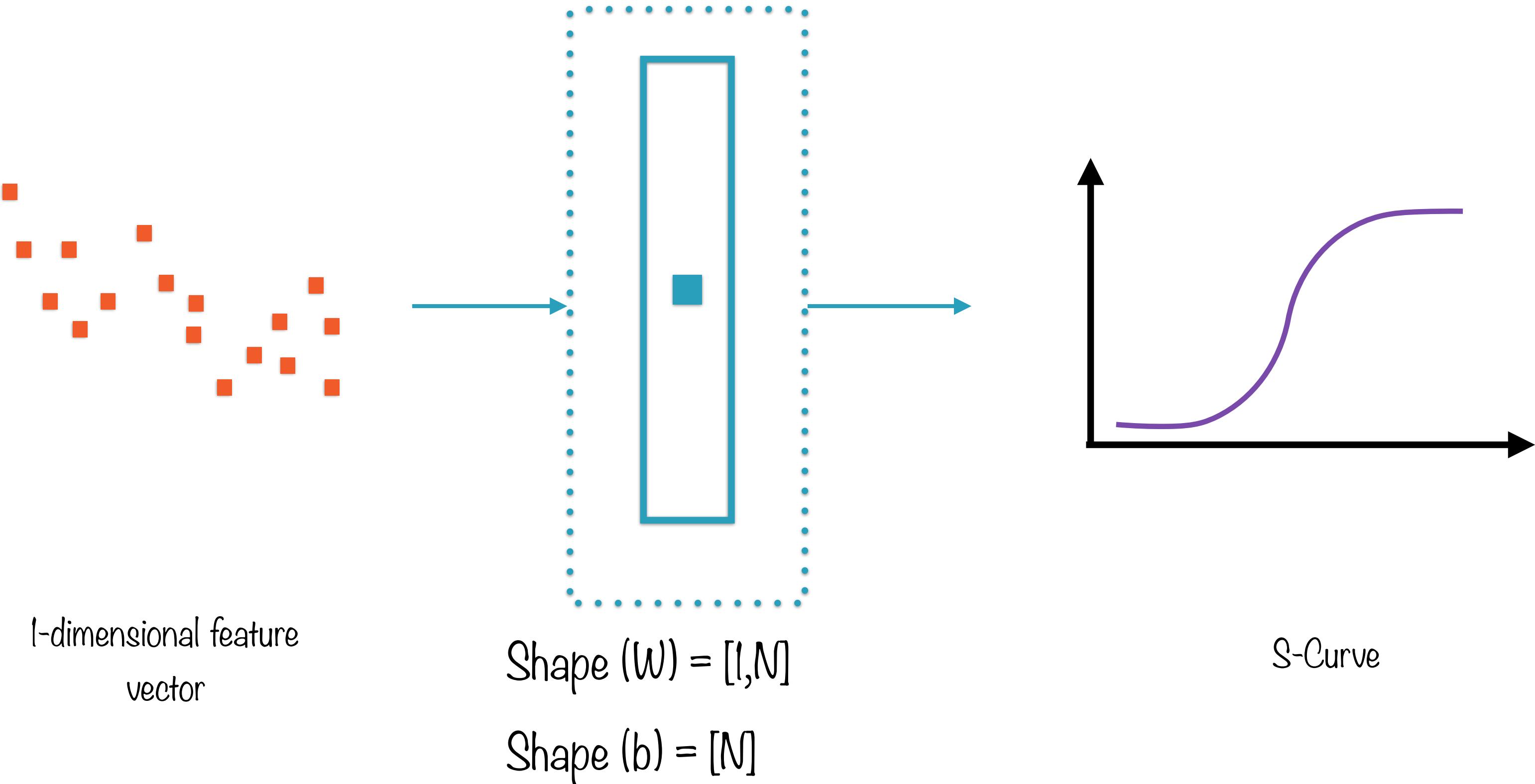
SoftMax N-category Classification



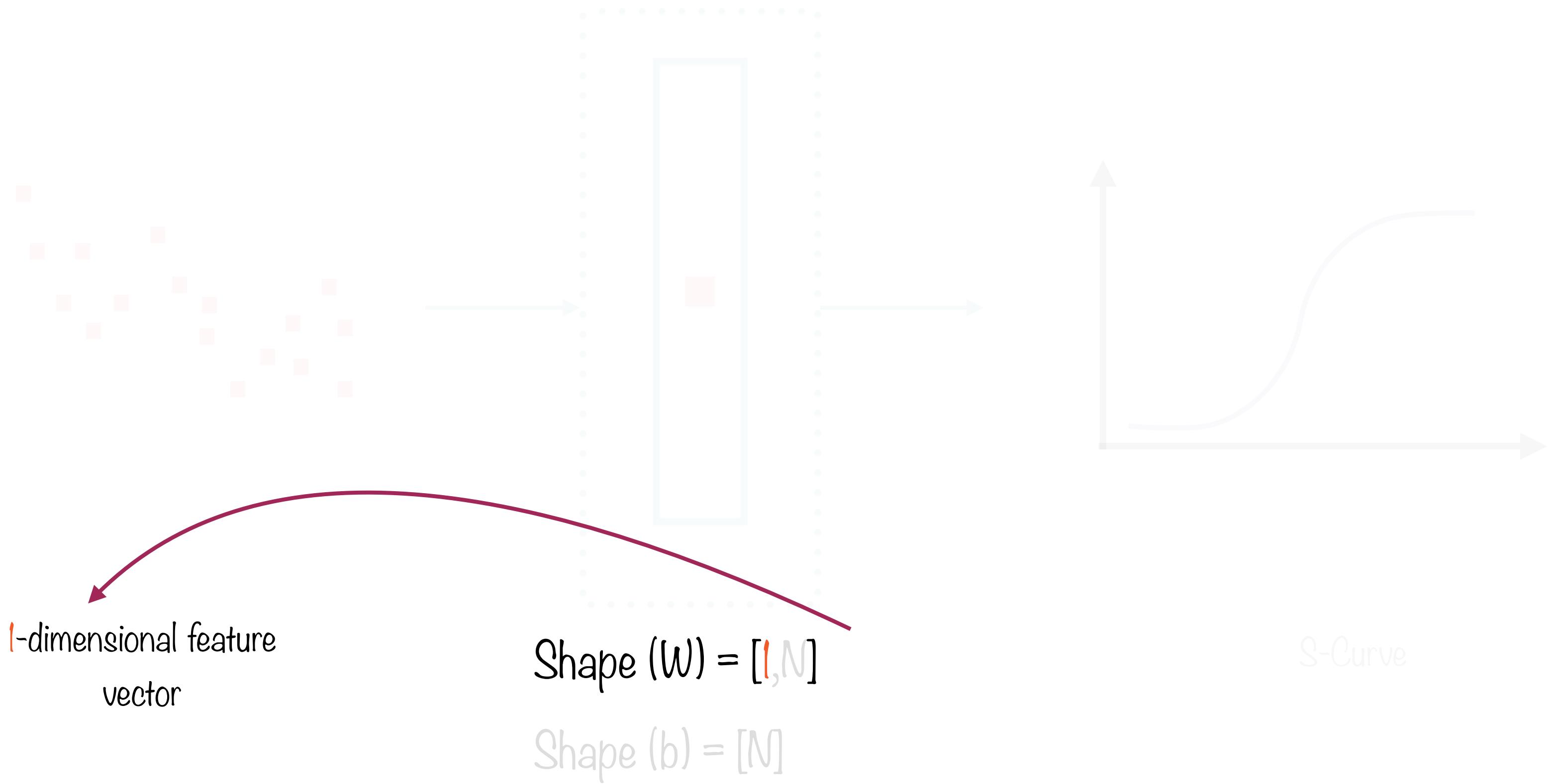
SoftMax N-category Classification



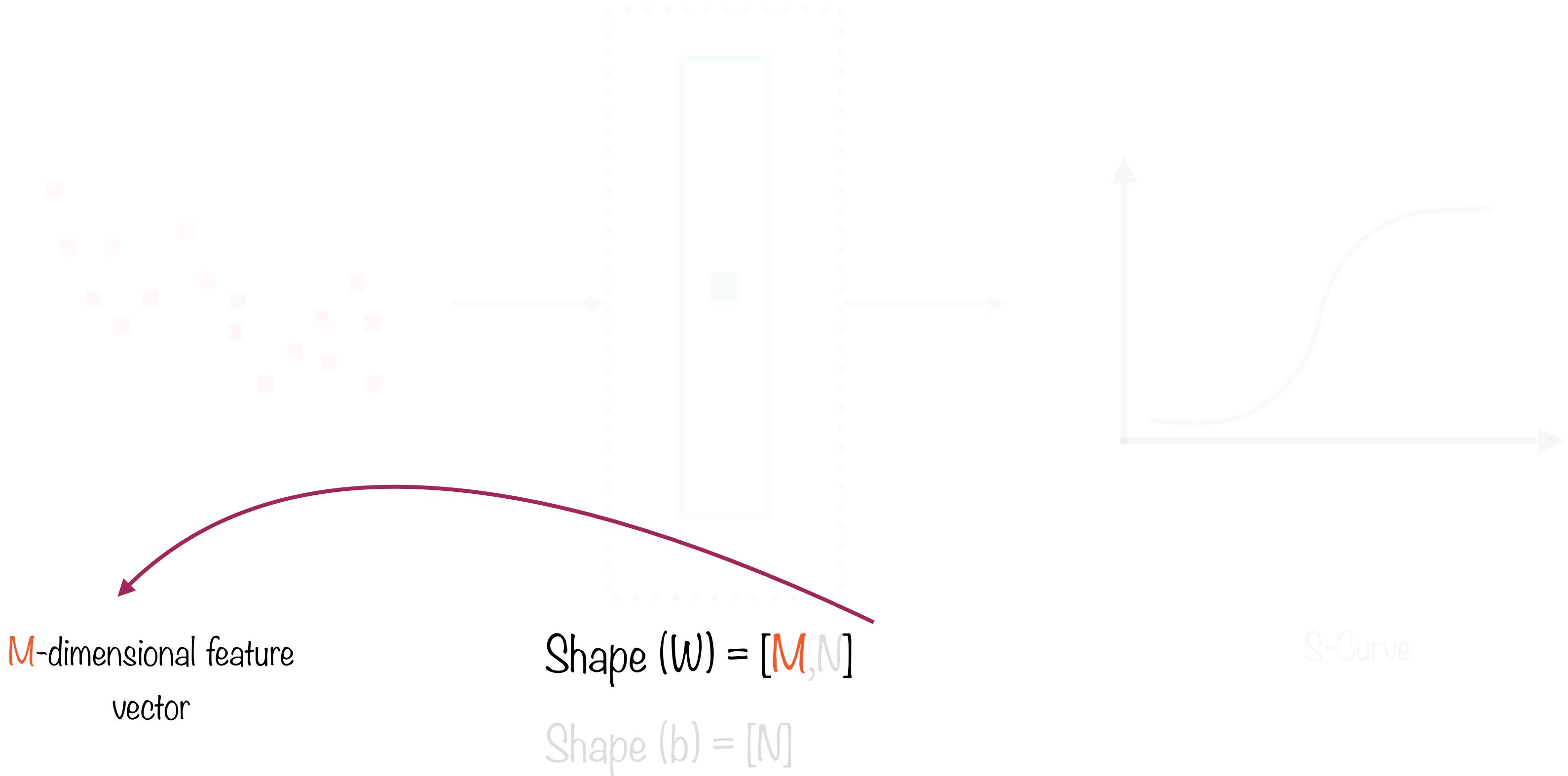
SoftMax N-category Classification



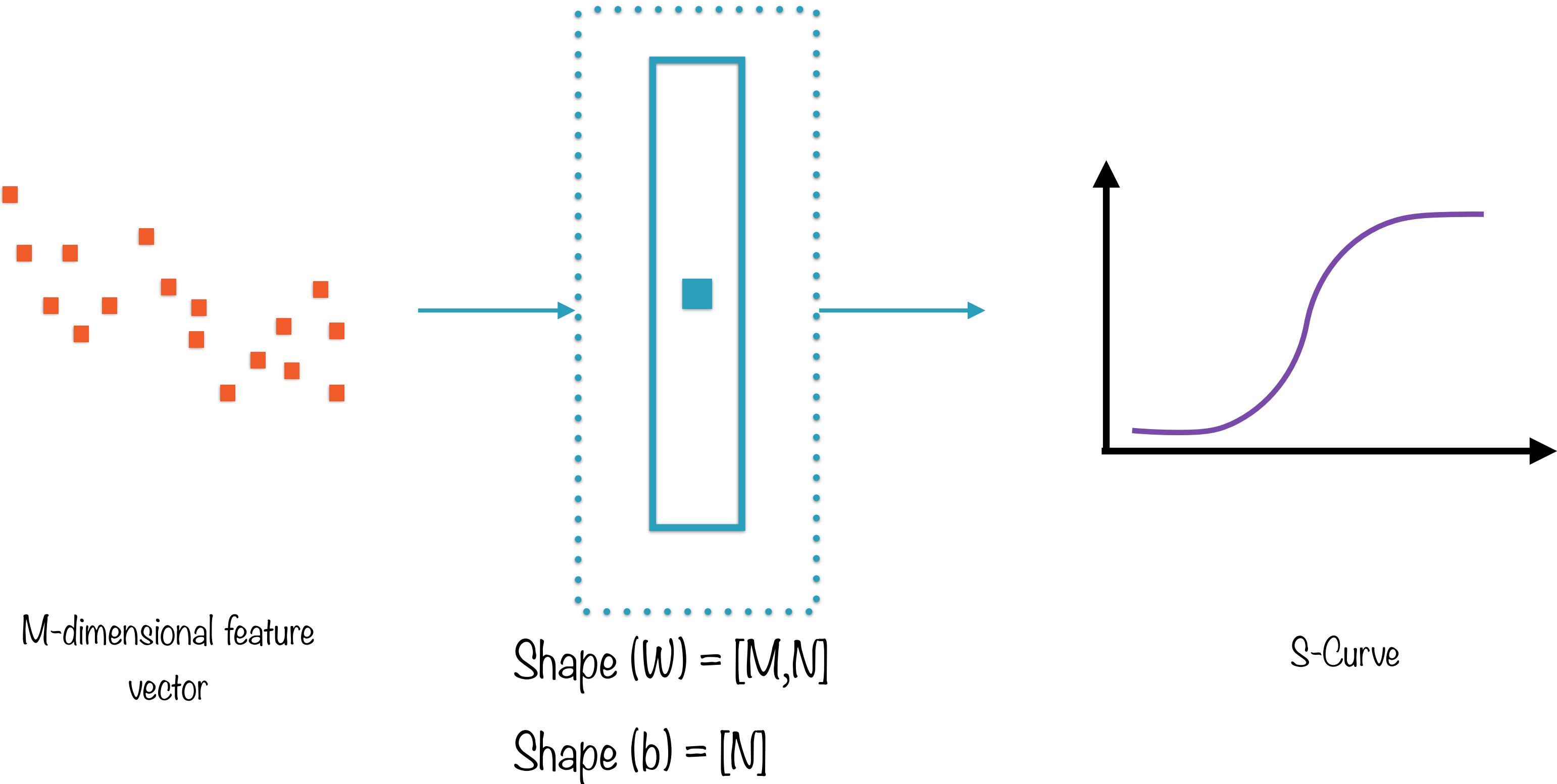
SoftMax N-category Classification



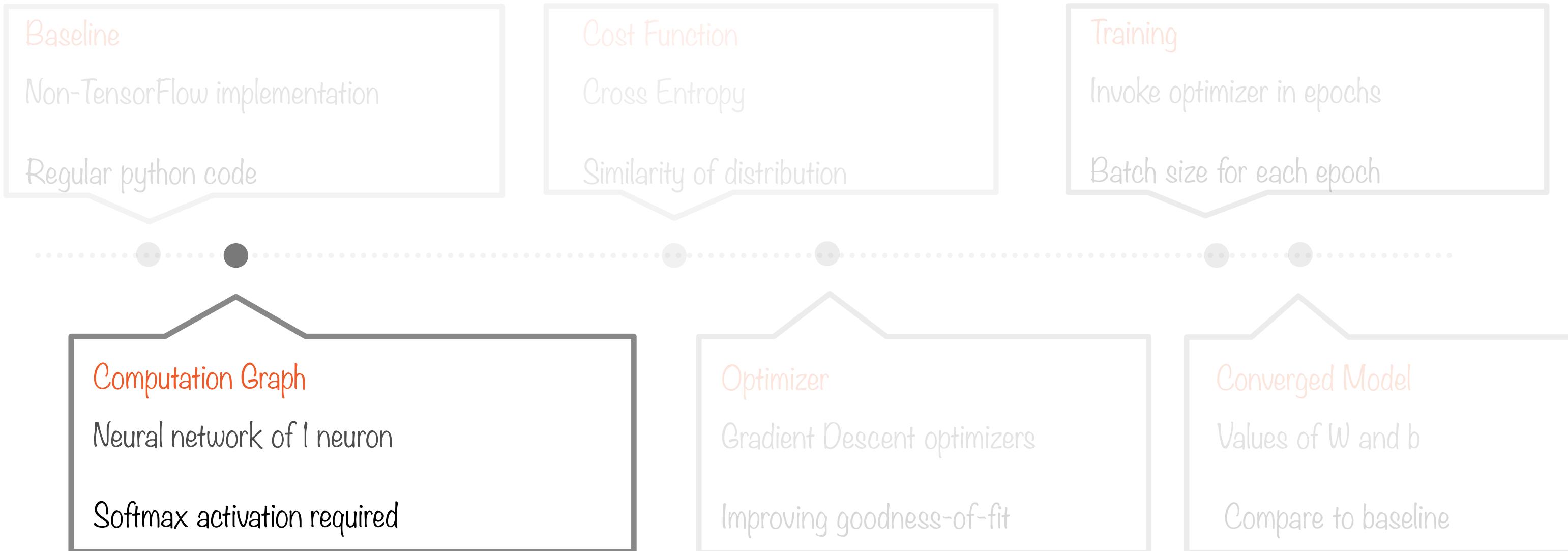
SoftMax N-category Classification



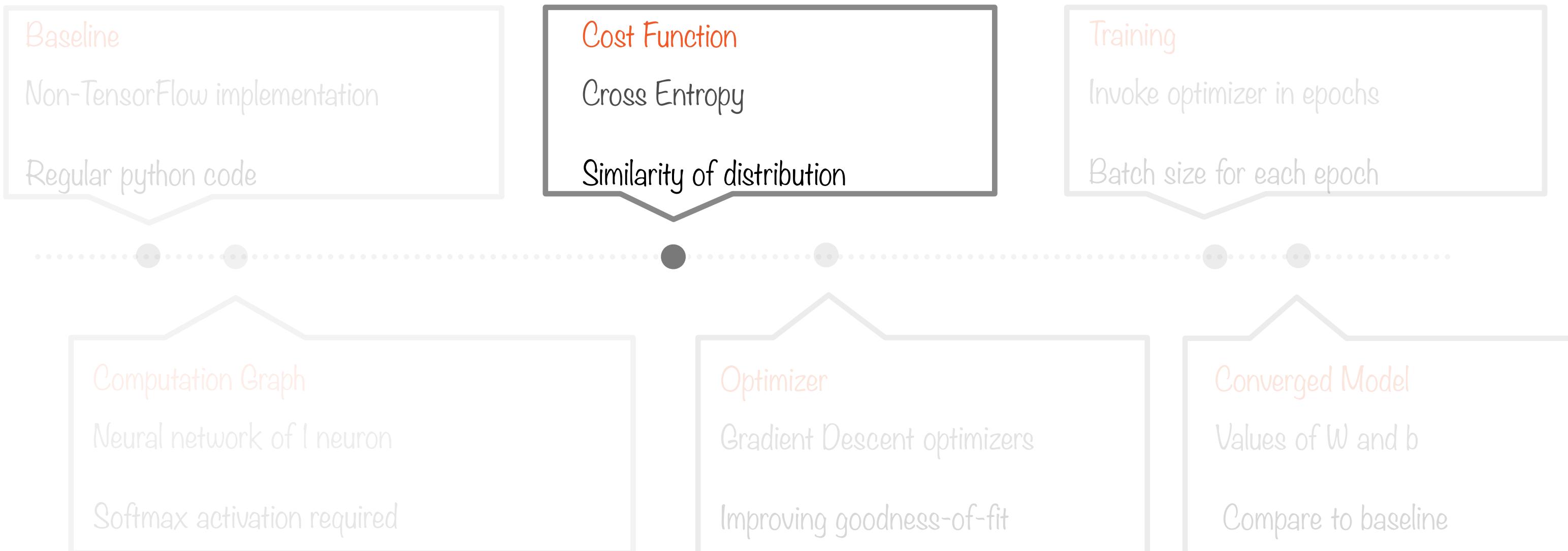
SoftMax N-category Classification



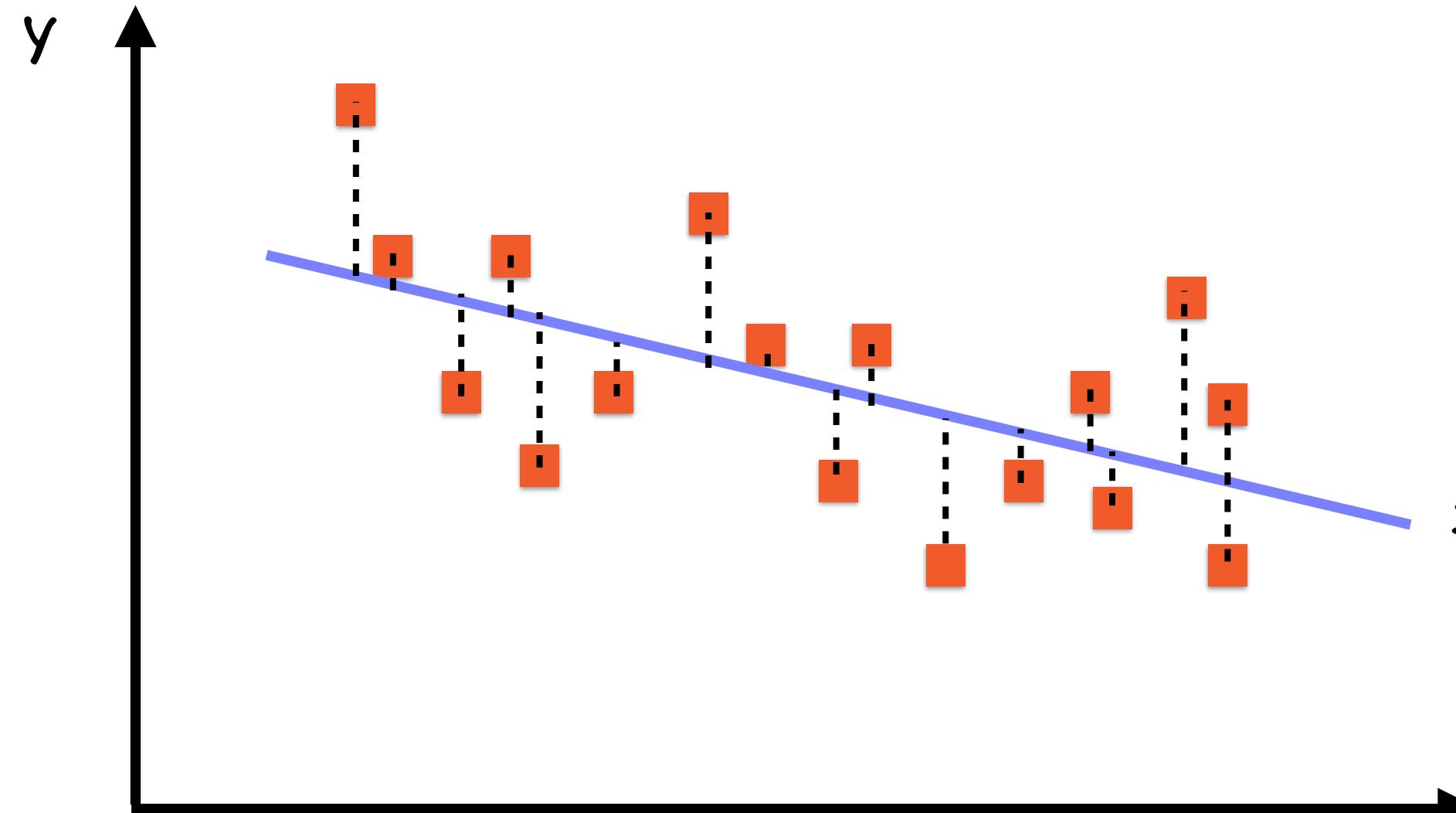
Logistic Regression in TensorFlow



Logistic Regression in TensorFlow



Linear Regression and MSE



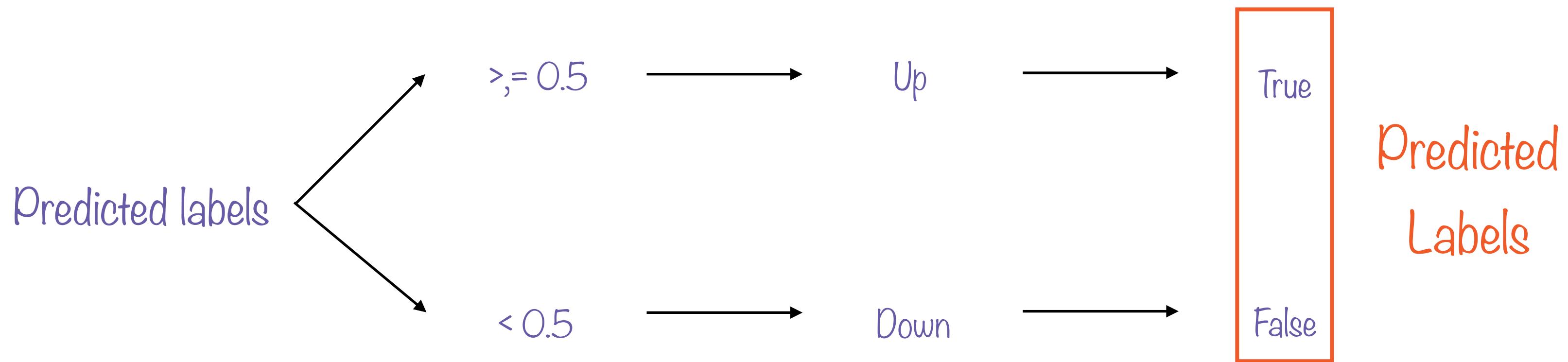
$$\text{Regression Line: } y = A + Bx$$

x

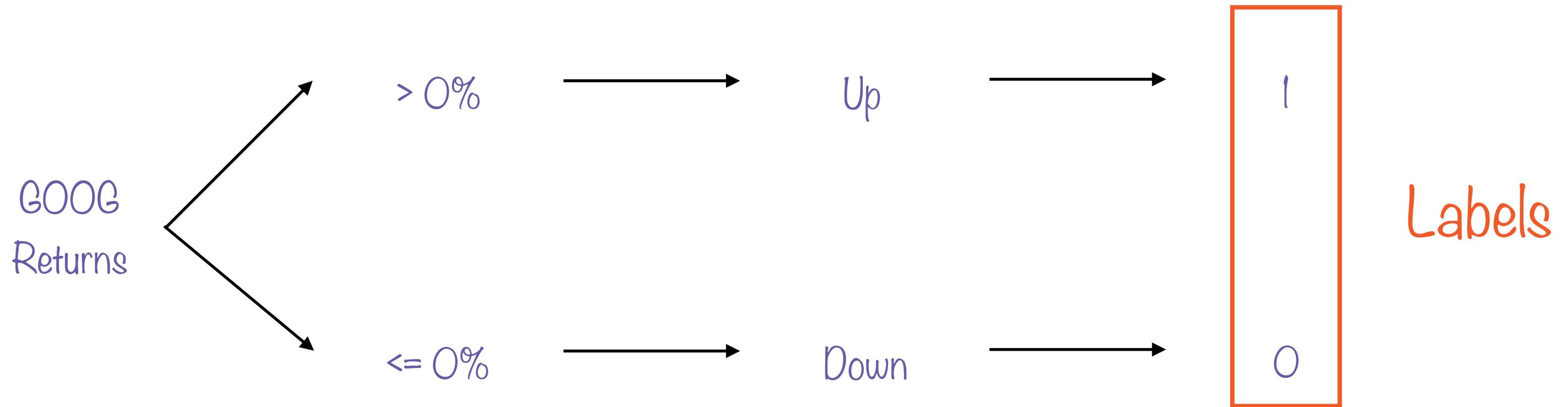


The “best fit” line is called the regression line

Logistic Regression



Set up the Problem



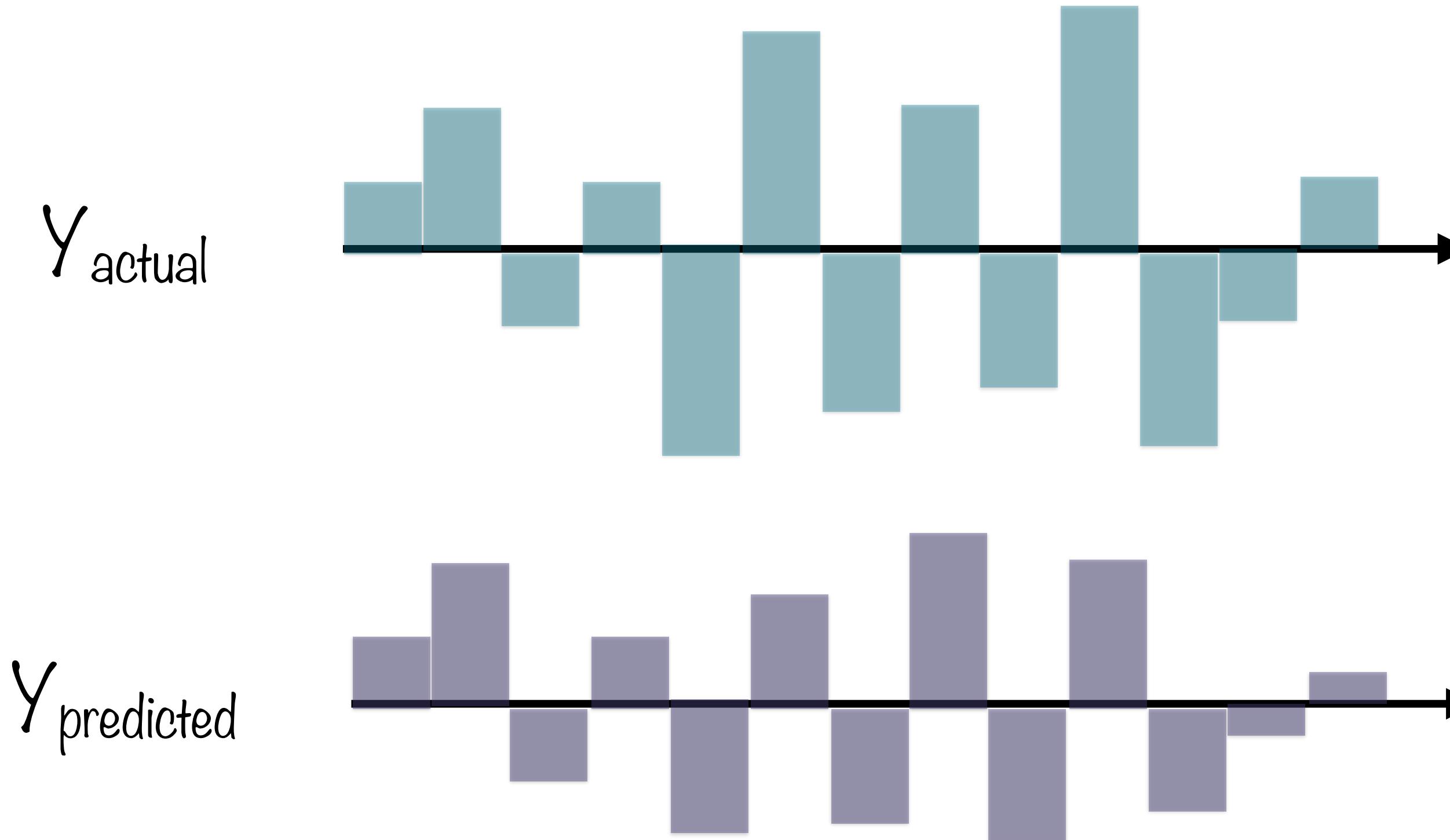
Label GOOG returns as binary (1,0)

Prediction Accuracy

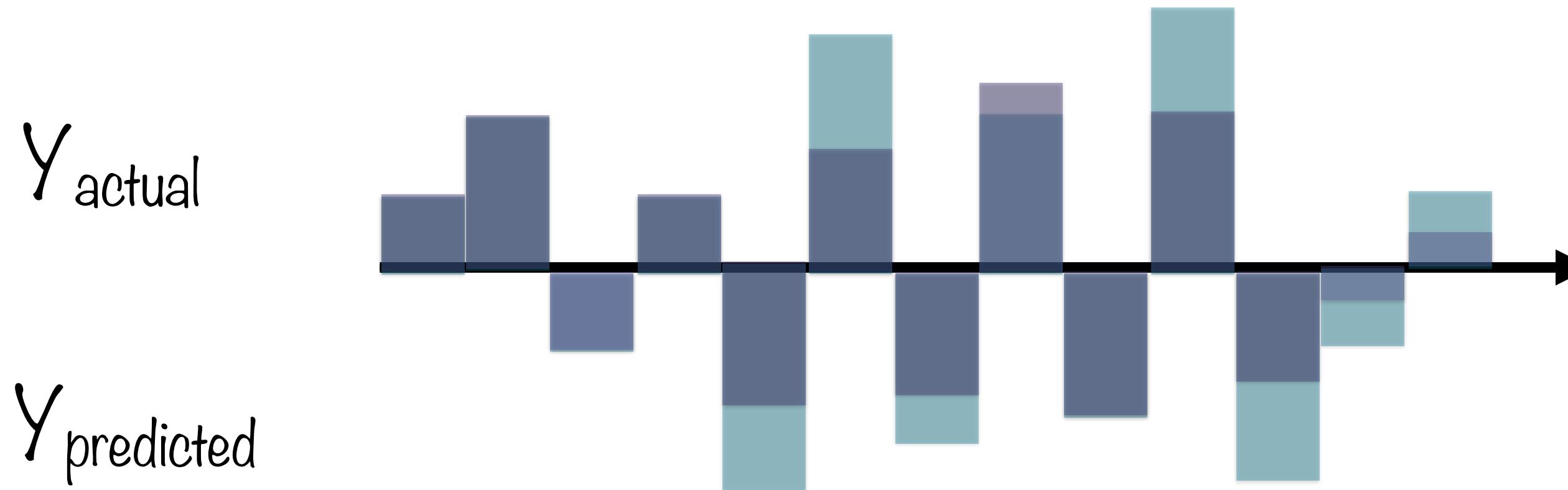
| DATE | ACTUAL | PREDICTED |
|------------|--------|-----------|
| 2005-01-01 | NA | NA |
| 2005-02-01 | 0 | 1 |
| 2005-03-01 | 0 | 0 |
| | | |

Compare GOOG's actual labels vs. predicted labels

Intuition: Low Cross Entropy

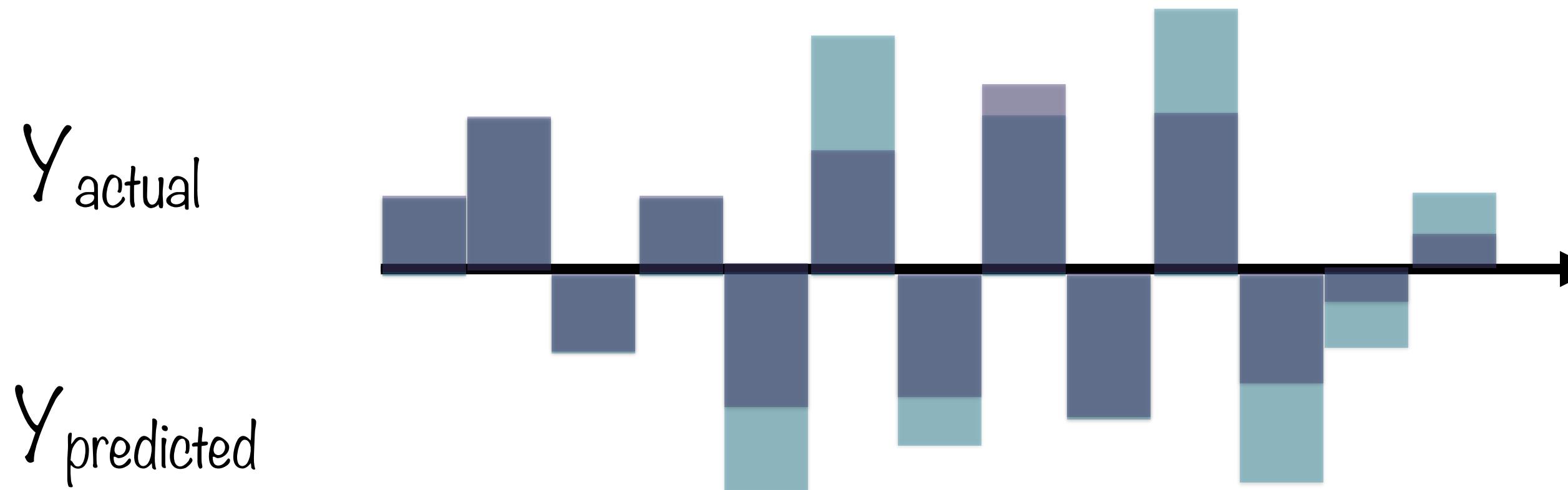


Intuition: Low Cross Entropy



The labels of the two series are in-synch

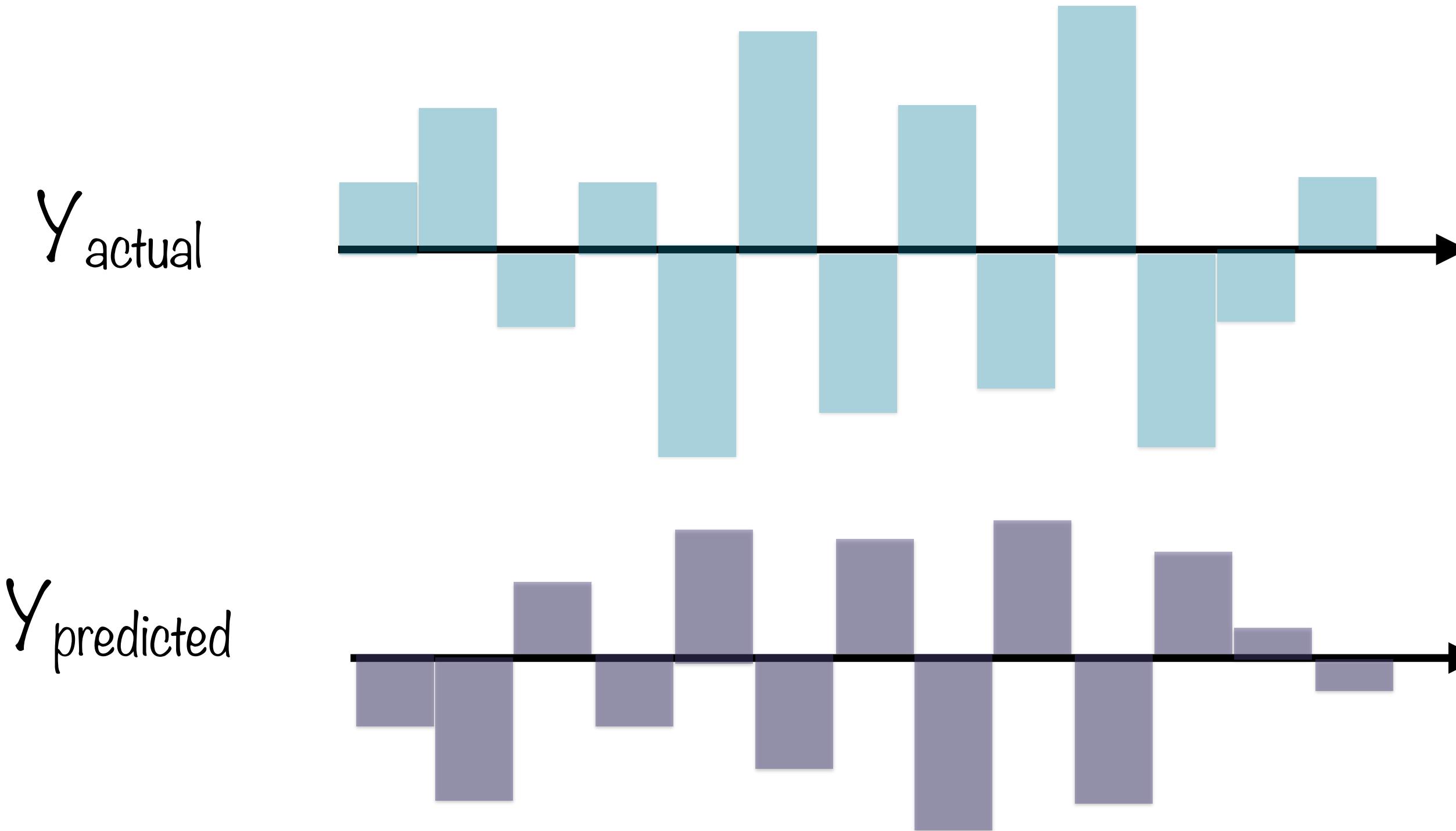
Intuition: Low Cross Entropy



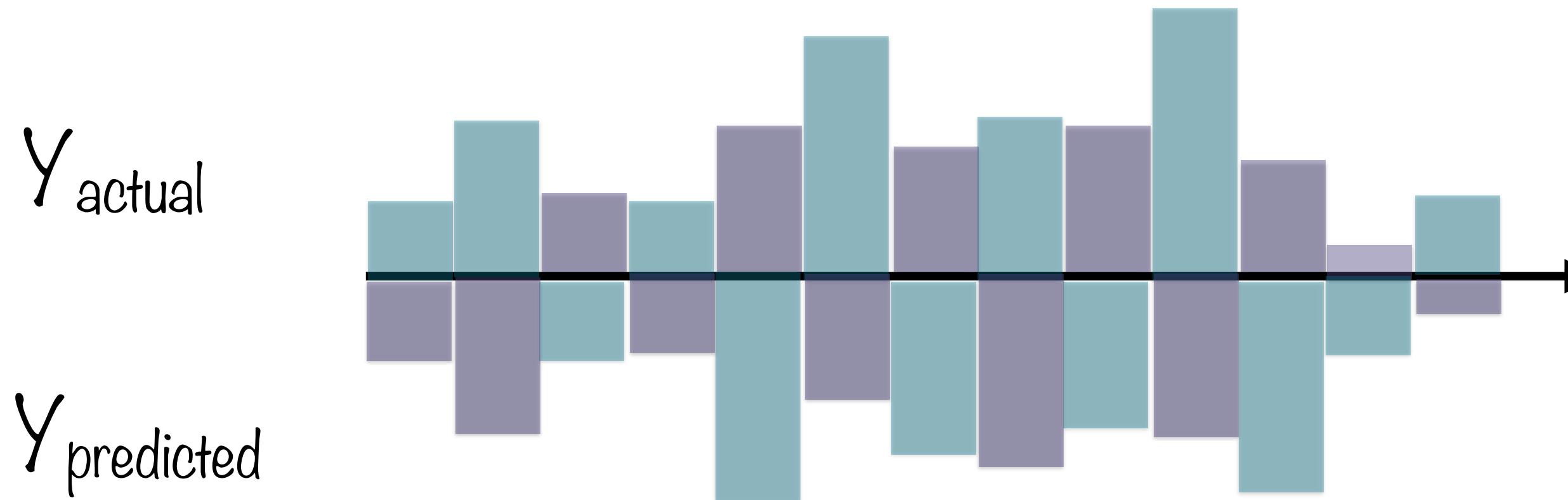
- $\text{Sum}(Y_{actual} * \log [Y_{predicted}])$ will be small

Cross Entropy

Intuition: Low Cross Entropy

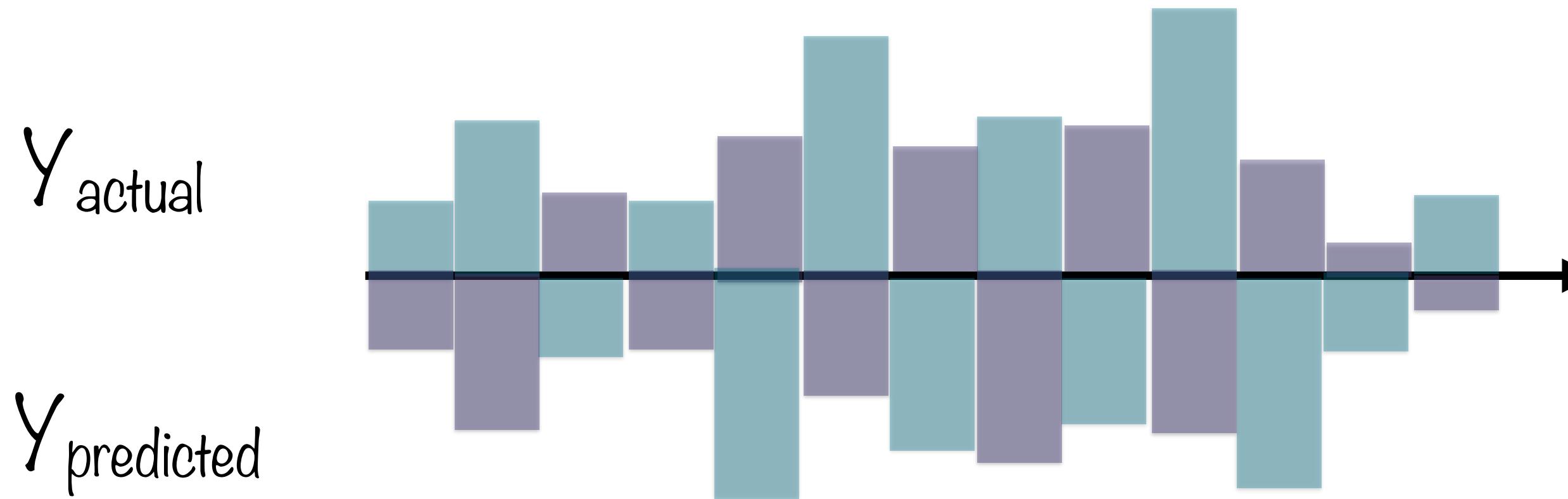


Intuition: High Cross Entropy



The labels of the two series are out-of-synch

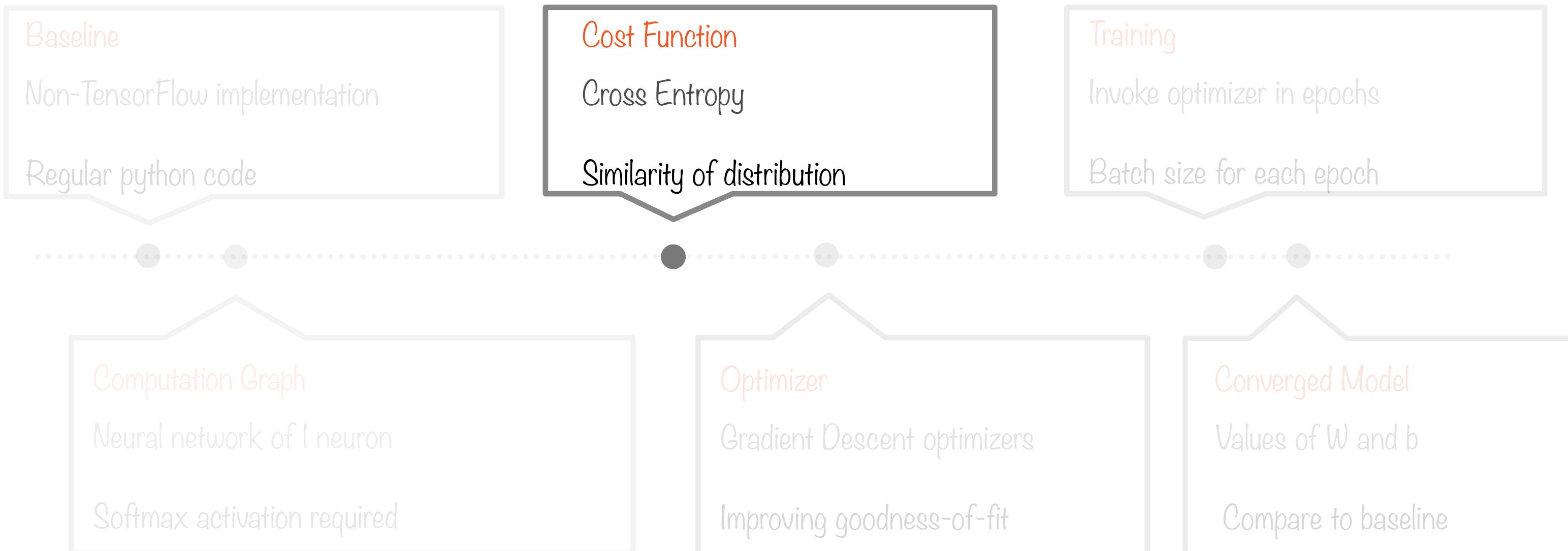
Intuition: High Cross Entropy



$-\text{Sum}(Y_{\text{actual}} * \log [Y_{\text{predicted}}])$ will be large

Cross Entropy

Logistic Regression in TensorFlow



Logistic Regression in TensorFlow

Baseline
Non-TensorFlow implementation
Regular python code

Cost Function
Cross Entropy
Similarity of distribution

Training
Invoke optimizer in epochs
Batch size for each epoch



Computation Graph
Neural network of 1 neuron
Softmax activation required

Optimizer
Gradient Descent optimizers
Improving goodness-of-fit

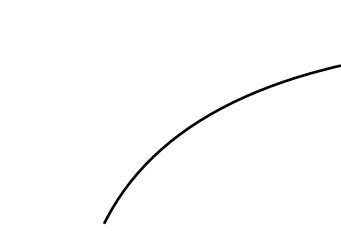
Converged Model
Values of W and b
Compare to baseline

`tensorflow.argmax(y,l)`

Finding the index of the largest element

Return the index of the largest element of tensor y along dimension k

`tensorflow.argmax(y,l)`



Tensor

Finding the index of the largest element

Return the index of the largest element of tensor y along dimension k

`tensorflow.argmax(y,1)`

Dimension

Finding the index of the largest element

Return the index of the largest element of tensor y along dimension k

tf.argmax

Tensor y

Index = 0

1

2

3

4

5

Dimension 0

Dimension 1

| | | |
|--|--|-----|
| | | 5 |
| | | 15 |
| | | 12 |
| | | 100 |
| | | 74 |
| | | 33 |

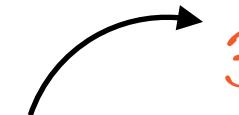
`tf.argmax(y,1)`

tf.argmax

Tensor y

Index = 0
1
2
3
4
5

Return value



Dimension 0

Dimension 1

| | | |
|--|--|-----|
| | | 5 |
| | | 15 |
| | | 12 |
| | | 100 |
| | | 74 |
| | | 33 |

tf.argmax(y, 1)

tf.argmax

Tensor y

Index = 0
1
2
3
4
5

Return value



Dimension 0

Dimension 1

| | | |
|--|--|-----|
| | | 5 |
| | | 15 |
| | | 12 |
| | | 100 |
| | | 74 |
| | | 33 |

tf.argmax(y,1)

tf.argmax

Tensor y

Index = 0

1

2

3

4

5

Dimension 0

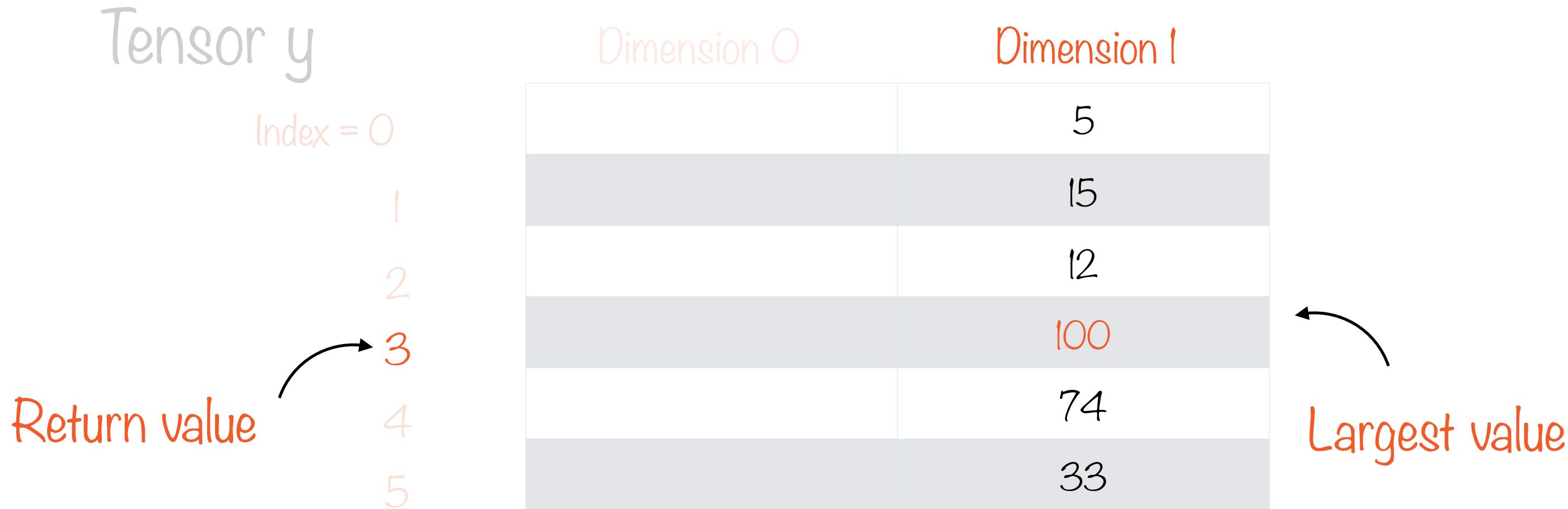
Dimension 1

| | |
|--|-----|
| | 5 |
| | 15 |
| | 12 |
| | 100 |
| | 74 |
| | 33 |

`tf.argmax(y, 1)`



tf.argmax



tf.argmax

Tensor y

Index = 0

Index = M

| | Dimension 1 | ... | Dimension N |
|-----------|-------------|-----|-------------|
| Index = 0 | | | |
| | | | |
| | | | |
| Index = M | | | |

`tf.argmax(y, 1)`

```
tf.equal(tf.argmax(y_,l), tf.argmax(y,l))
```

Two invocations of `tf.argmax`

Once on actual labels $y_$, once on predicted values y

Actual labels

`tf.equal(tf.argmax(y_,1), tf.argmax(y,1))`

Two invocations of `tf.argmax`

Once on actual labels $y_$, once on predicted values y

Predicted labels

`tf.equal(tf.argmax(y_,1), tf.argmax(y,1))`

Two invocations of `tf.argmax`

Once on actual labels $y_$, once on predicted values y

`tf.equal(tf.argmax(y_, 1), tf.argmax(y, 1))`

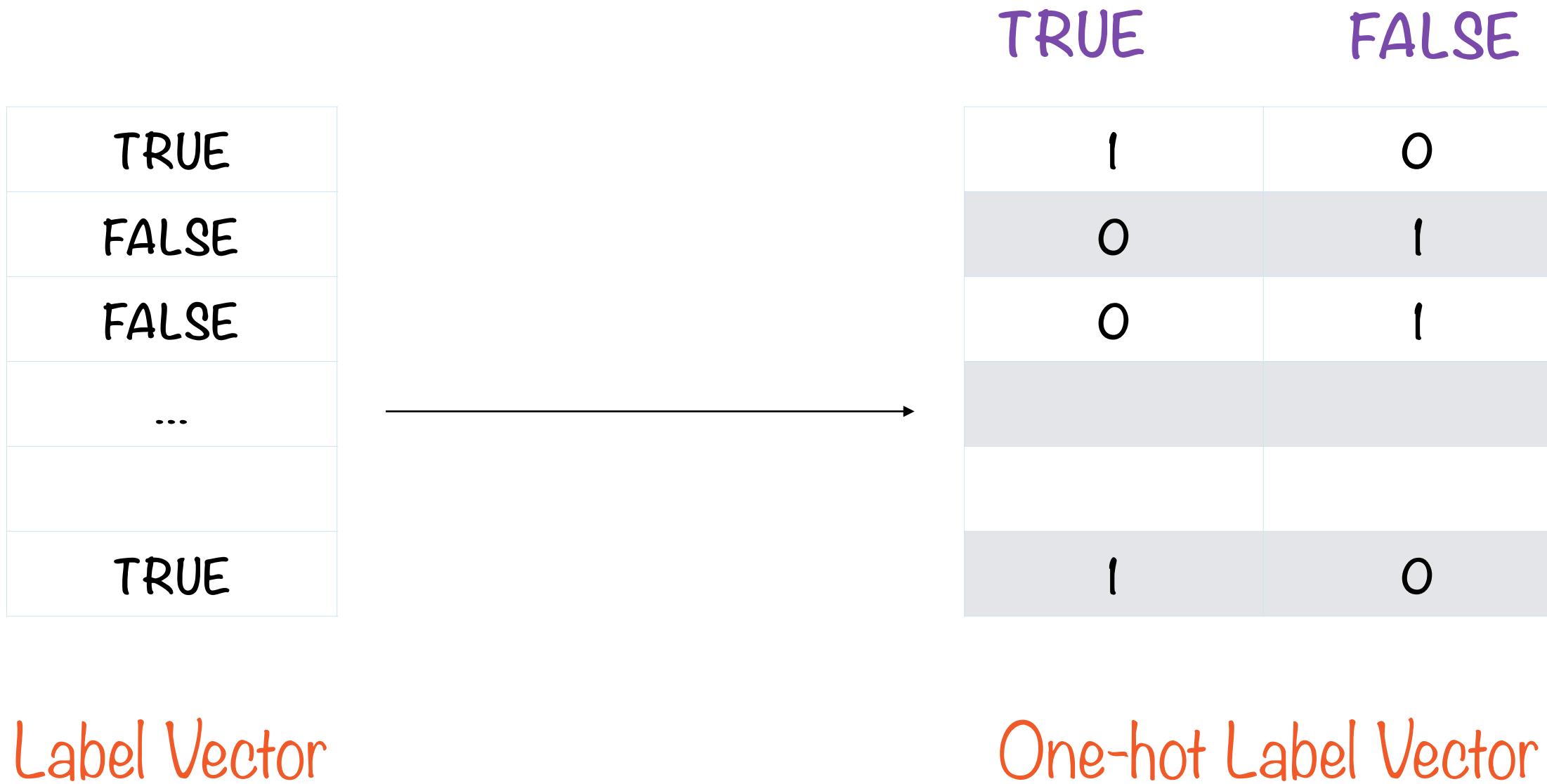


One-hot

Two invocations of `tf.argmax`

Once on actual labels $y_$, once on predicted values y

One-hot Representation



One-hot y_-



Label Vector

One-hot Label Vector

$\text{argmax}(\mathbf{y}_{\perp 1})$

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |

One-hot Label Vector

| | | | | |
|---|---|---|-----|---|
| 0 | 1 | 1 | ... | 0 |
| 1 | 0 | 0 | ... | 0 |
| 0 | 1 | 0 | ... | 0 |
| 0 | 0 | 1 | ... | 0 |
| 0 | 0 | 0 | ... | 0 |

Index of one-hot element

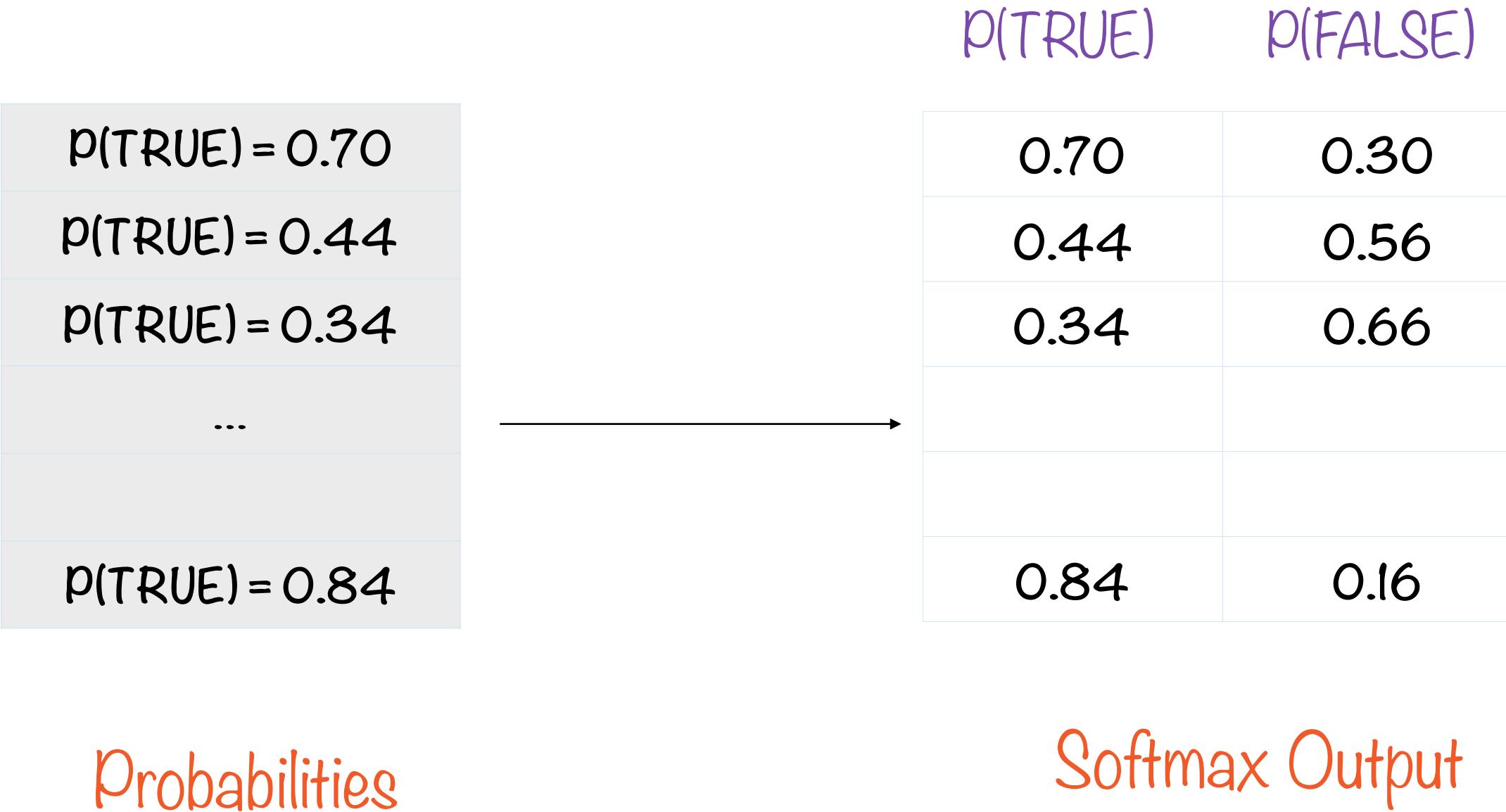
Predicted labels

`tf.equal(tf.argmax(y_,1), tf.argmax(y,1))`

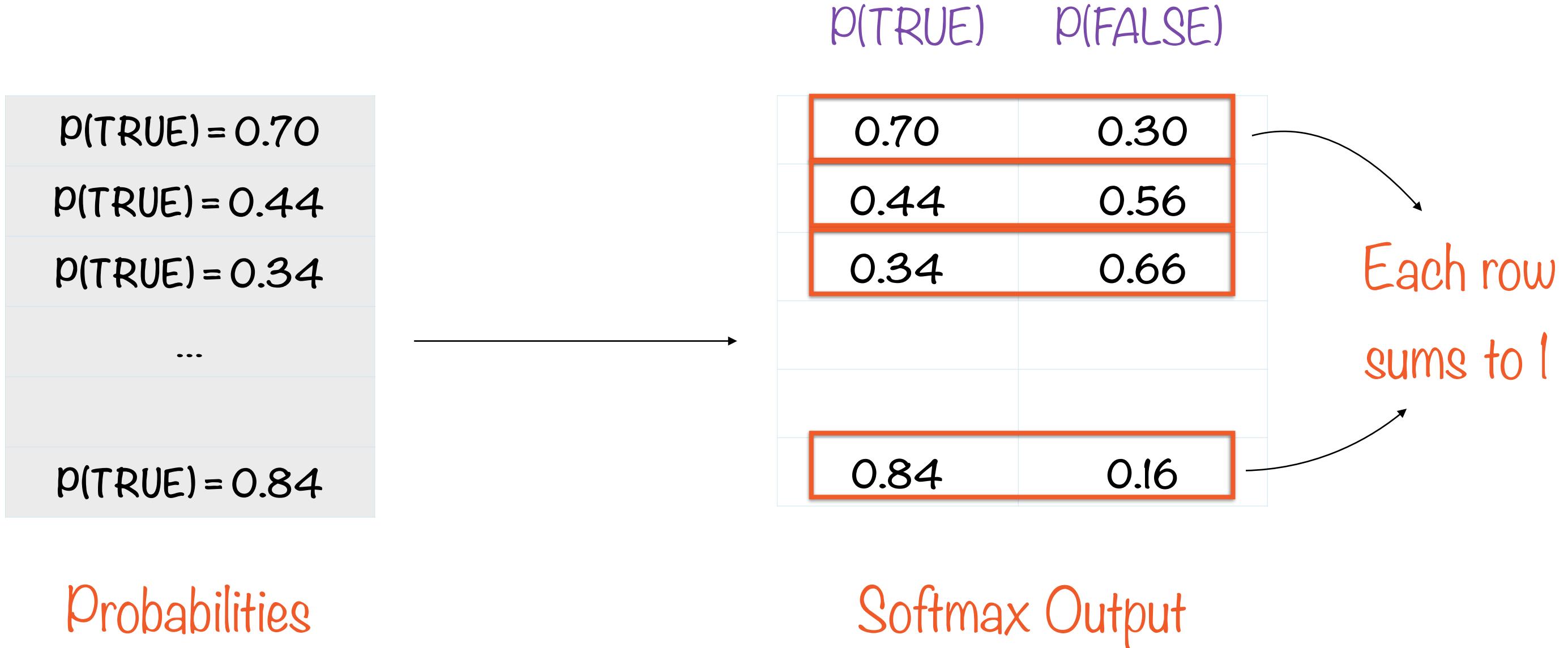
Two invocations of `tf.argmax`

Once on actual labels $y_$, once on predicted values y

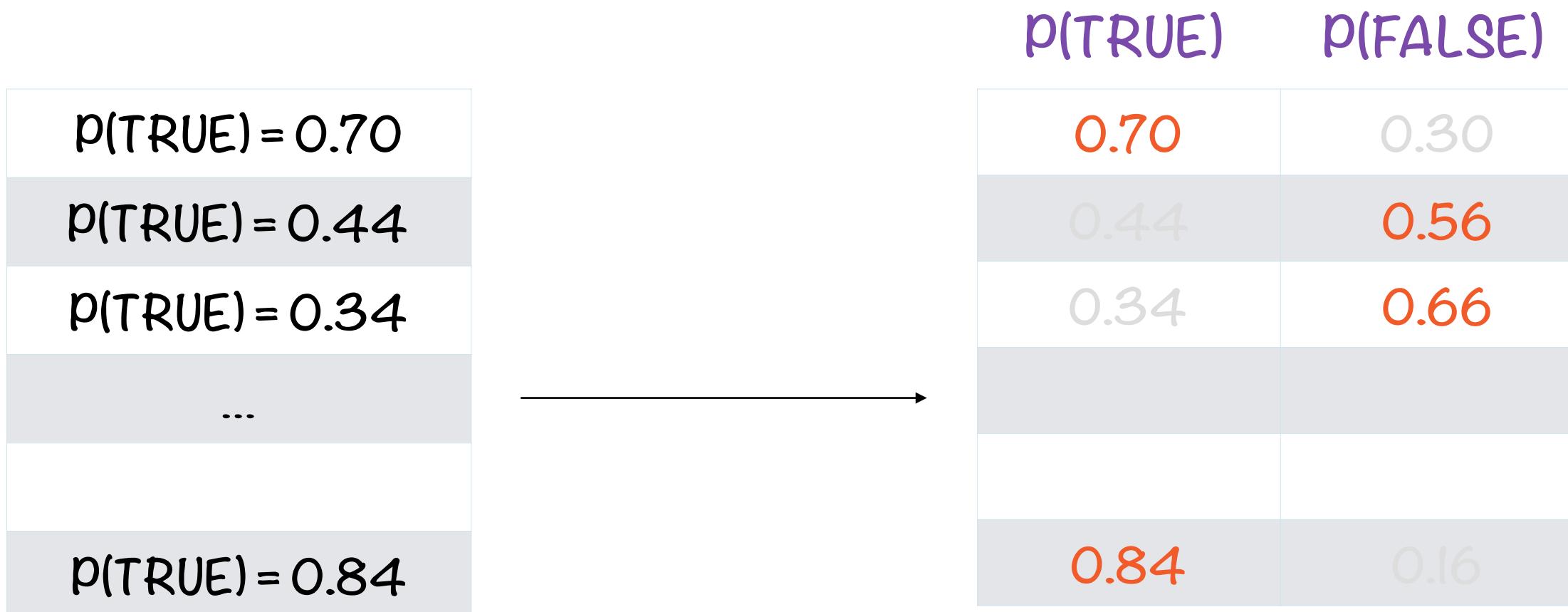
Predicted Probabilities y



Predicted Probabilities y



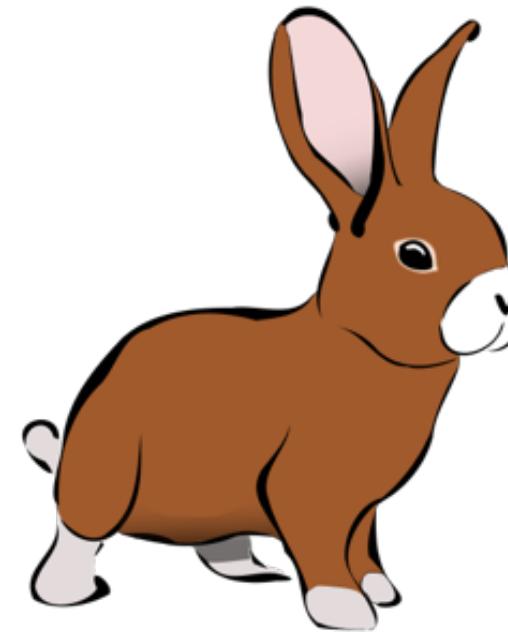
Predicted Probabilities y



Probabilities

Softmax Output

Rule of 50% in Binary Classification



Mammal



Fish

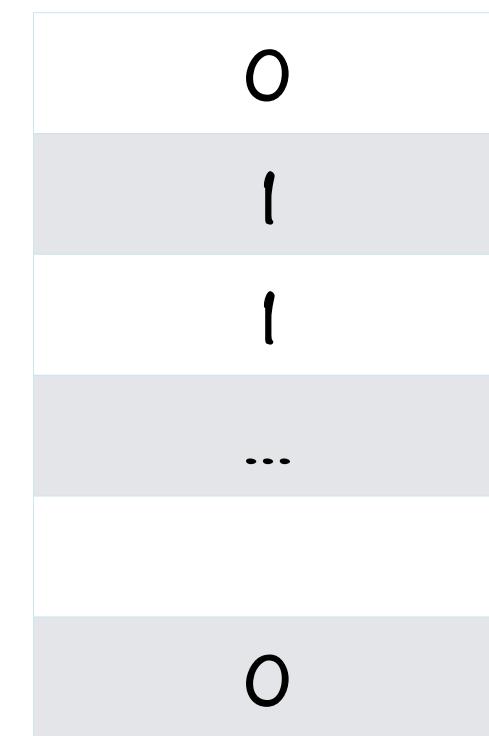


Probability of whales being Fish < 50%

$\text{argmax}(y, l)$

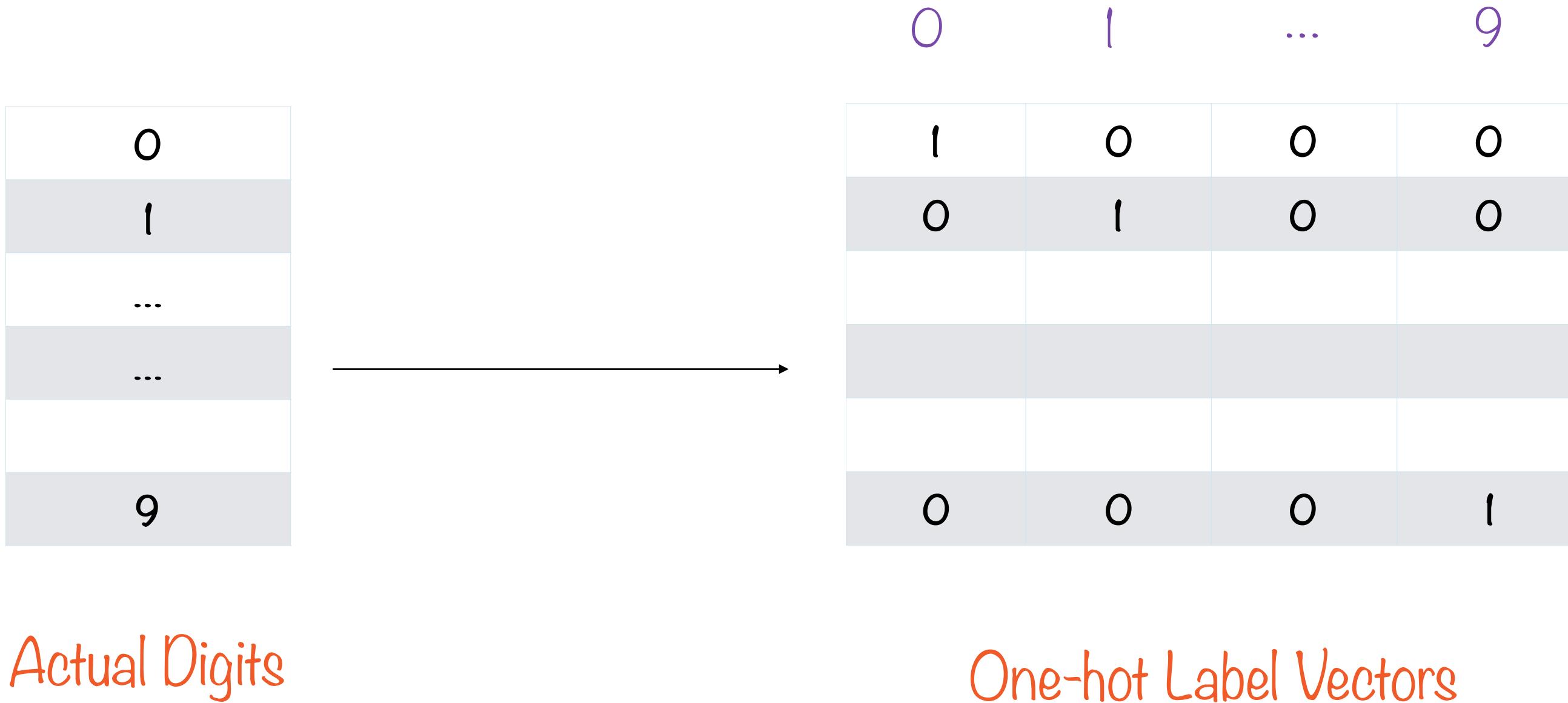
| $p(\text{TRUE})$ | $p(\text{FALSE})$ |
|------------------|-------------------|
| 0.70 | 0.30 |
| 0.44 | 0.56 |
| 0.34 | 0.66 |
| | |
| 0.84 | 0.16 |

Softmax Output

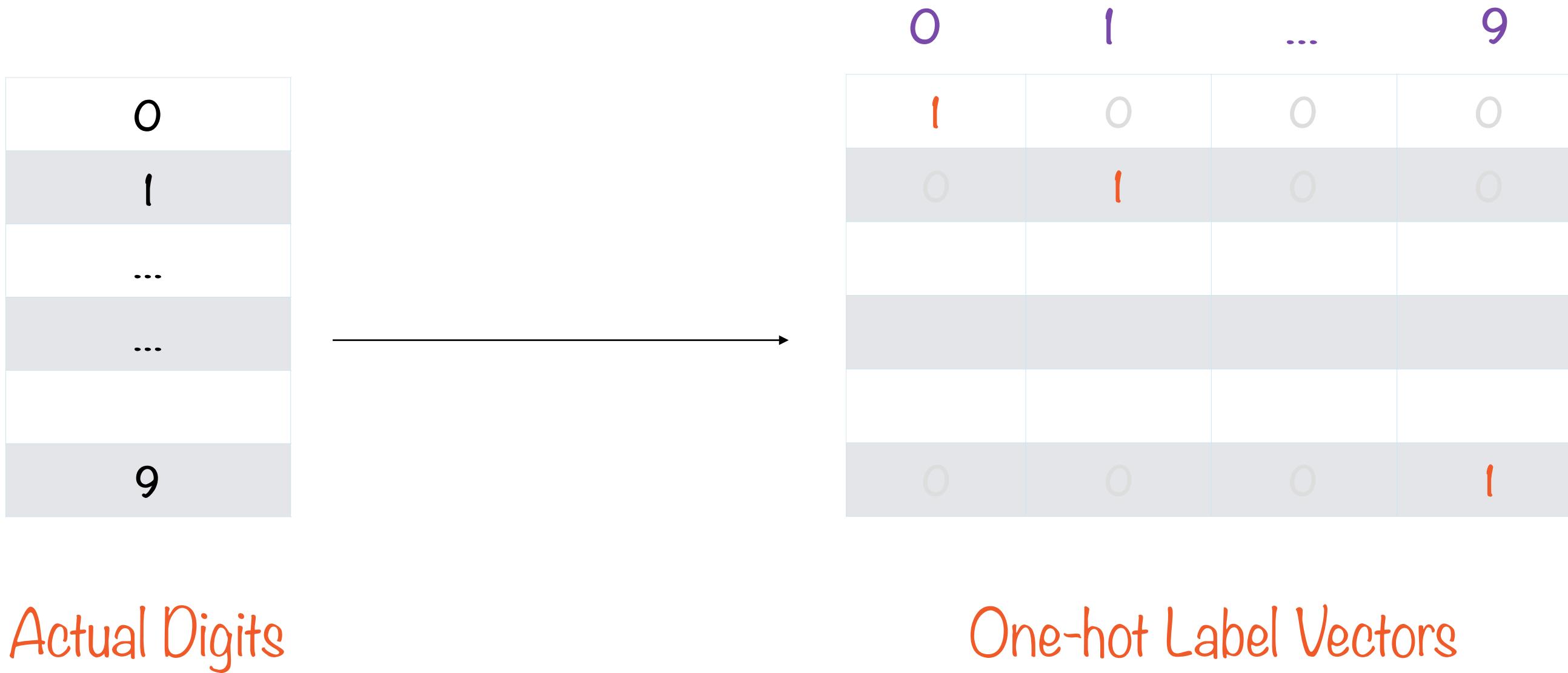


$\text{argmax}(y, l)$

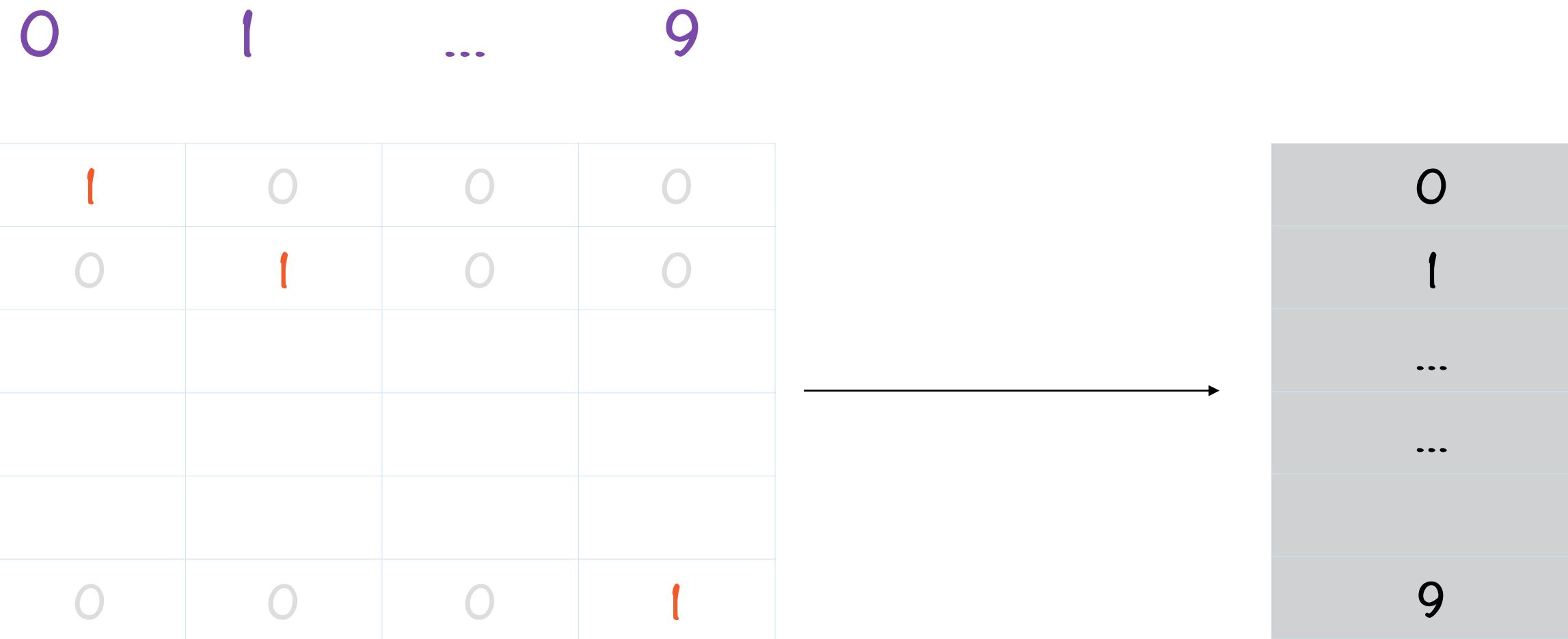
One-hot Vectors with Digit Classes



$y_$: One-hot Vectors with Digit Classes



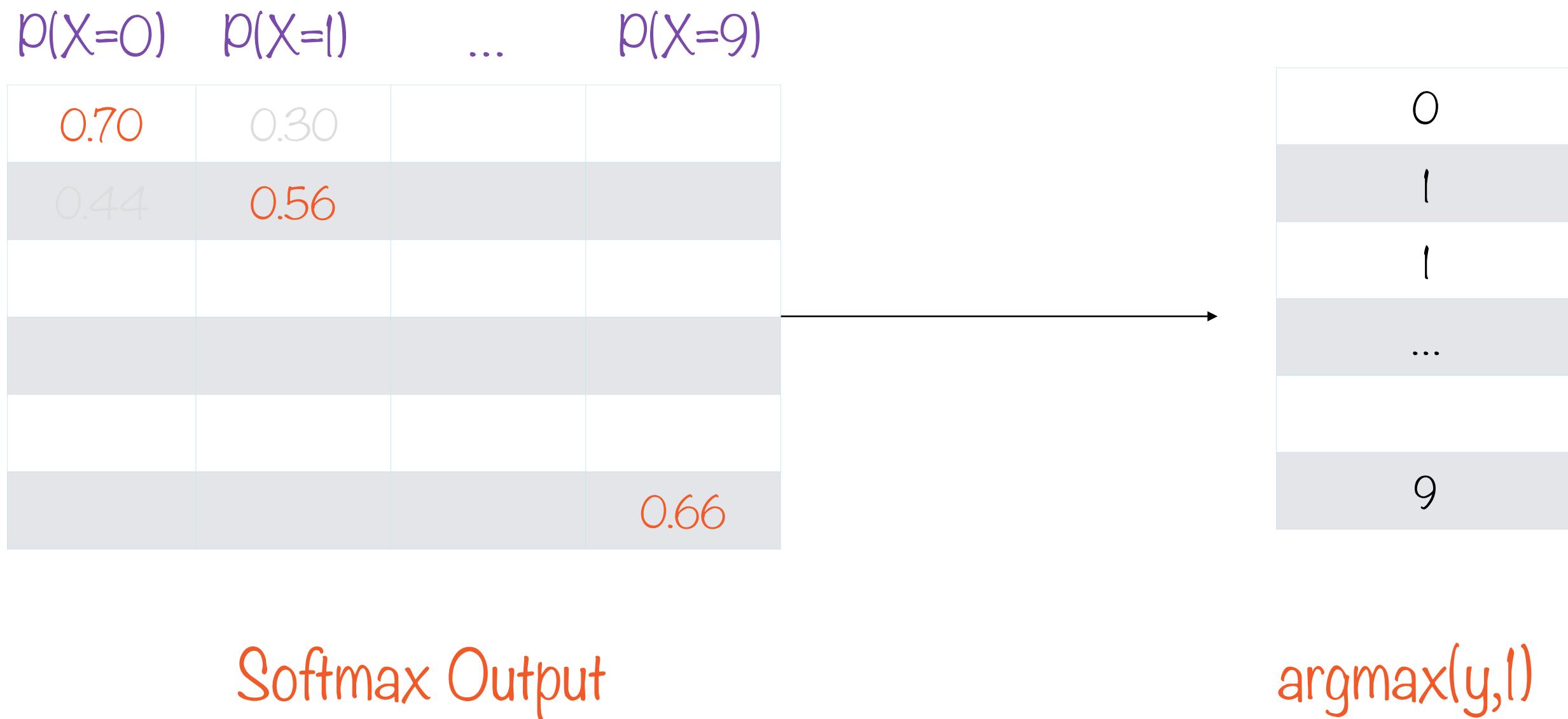
$\text{argmax}(\mathbf{y}_{:,1})$



One-hot Label Vectors

$\text{argmax}(\mathbf{y}_{:,1})$

Digit Classification



y : Predicted Probabilities

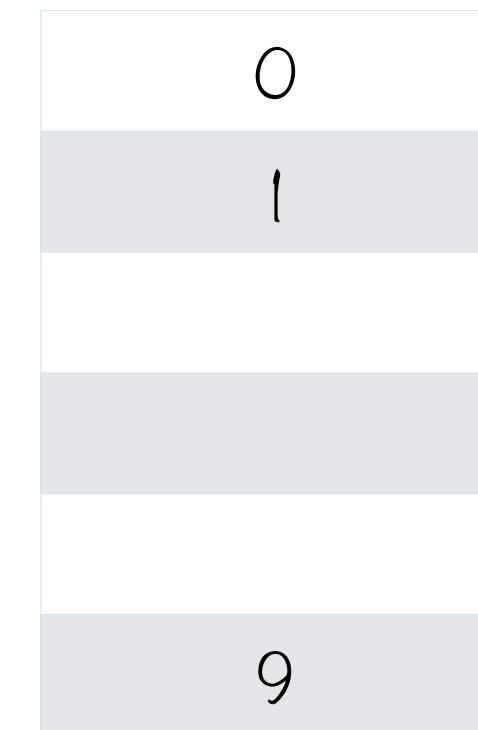
$p(X=0)$ $p(X=1)$

...

$p(X=9)$

| | | | |
|------|------|------|--|
| 0.70 | 0.30 | | |
| 0.44 | 0.56 | | |
| | | | |
| | | | |
| | | 0.66 | |

Softmax Output



$\text{argmax}(y, l)$

```
tf.equal(tf.argmax(y_,l), tf.argmax(y,l))
```

Two invocations of `tf.argmax`

Once on actual labels $y_$, once on predicted values y

Actual labels

`tf.equal(tf.argmax(y_,1), tf.argmax(y,1))`

Two invocations of `tf.argmax`

Once on actual labels $y_$, once on predicted values y

Predicted labels

`tf.equal(tf.argmax(y_,1), tf.argmax(y,1))`

Two invocations of `tf.argmax`

Once on actual labels $y_$, once on predicted values y

Two invocations of `tf.argmax`

Tensor of actual labels

Tensor of predicted labels

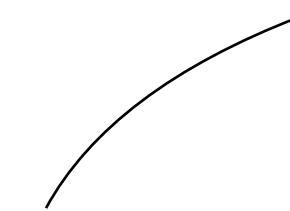
`tf.equal(tf.argmax(y_,1), tf.argmax(y,1))`

Once on actual labels $y_$, once on predicted values y

Two invocations of `tf.argmax`

`tf.equal(tf.argmax(y_,1), tf.argmax(y,1))`

List of True, False values



Once on actual labels $y_$, once on predicted values y

Two invocations of `tf.argmax`

True: Correct prediction

False: Incorrect prediction

`tf.equal(tf.argmax(y_,1), tf.argmax(y,1))`

Once on actual labels $y_$, once on predicted values y

Building Generalized Linear Models Using Estimators

Overview

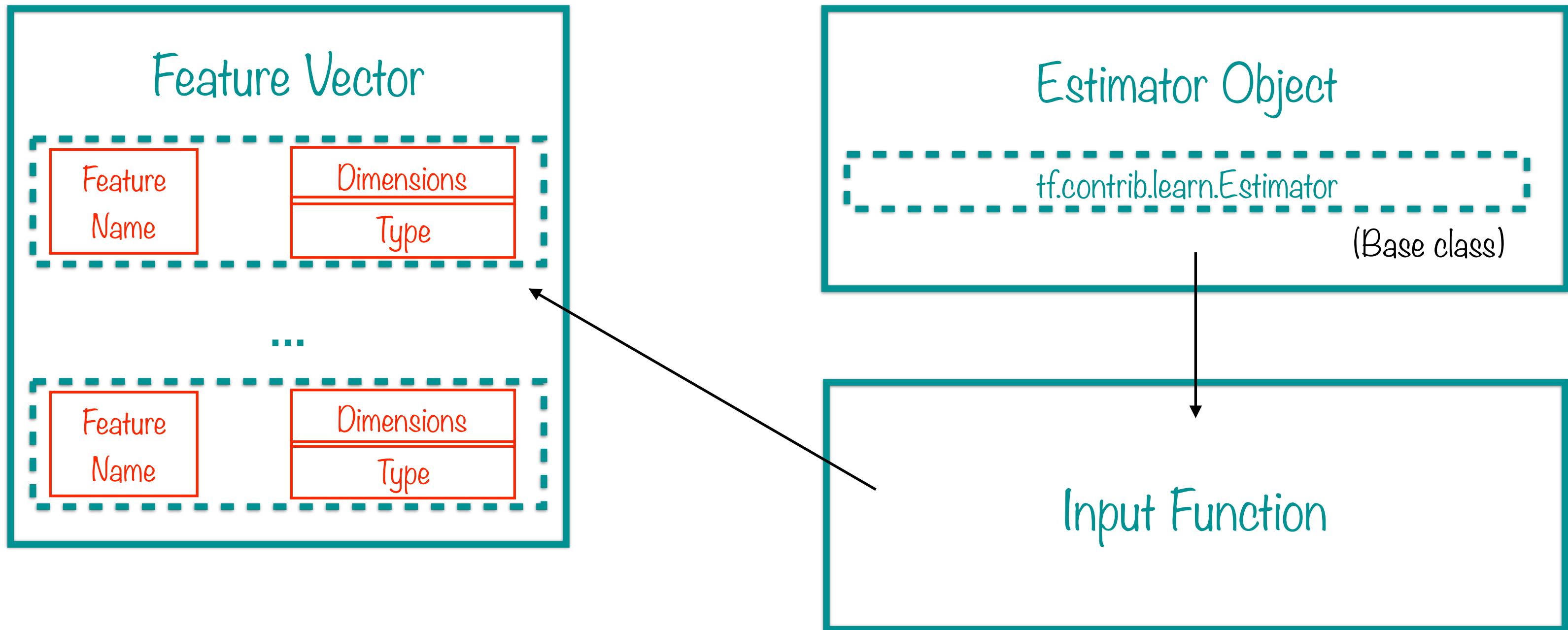
Estimators are cookie-cutter TensorFlow APIs for many standard problems

These high-level APIs reside in `tf.learn` and `tf.contrib.learn`

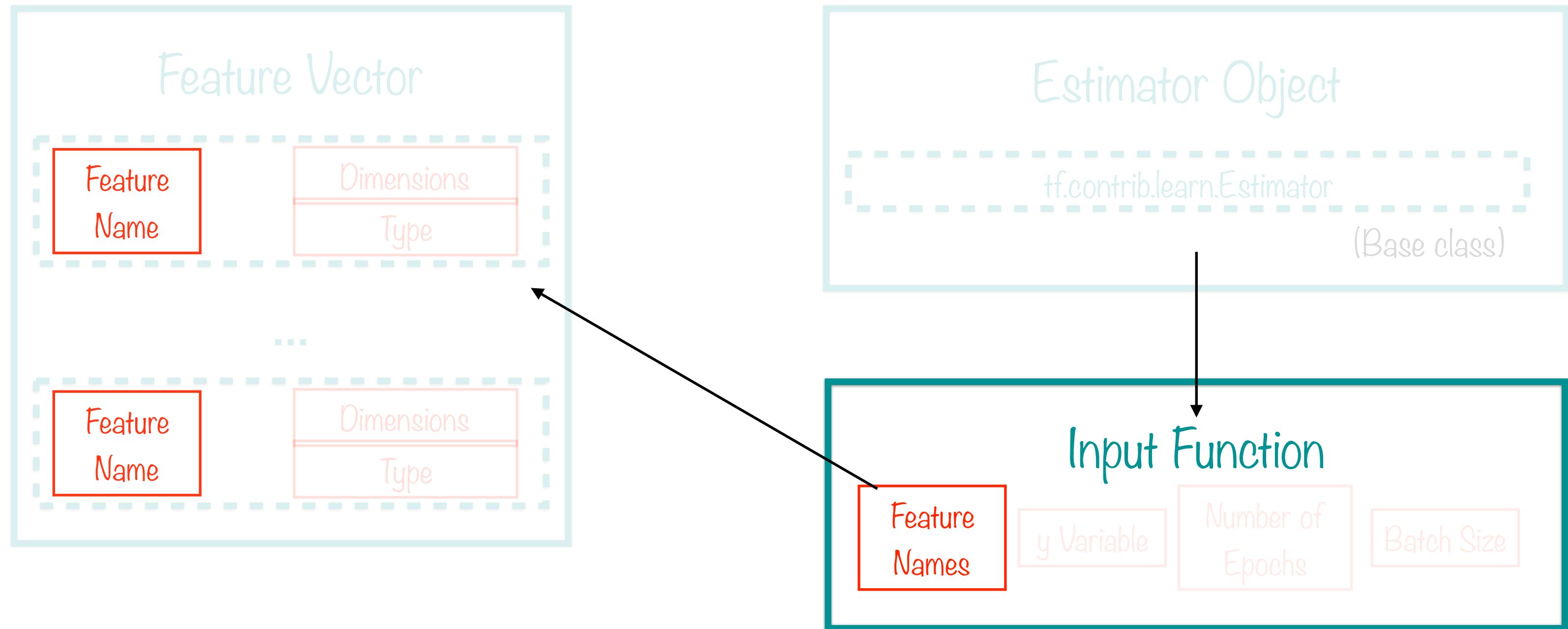
Estimators can be extended by plugging custom models into a base class

That extension relies on composition rather than inheritance

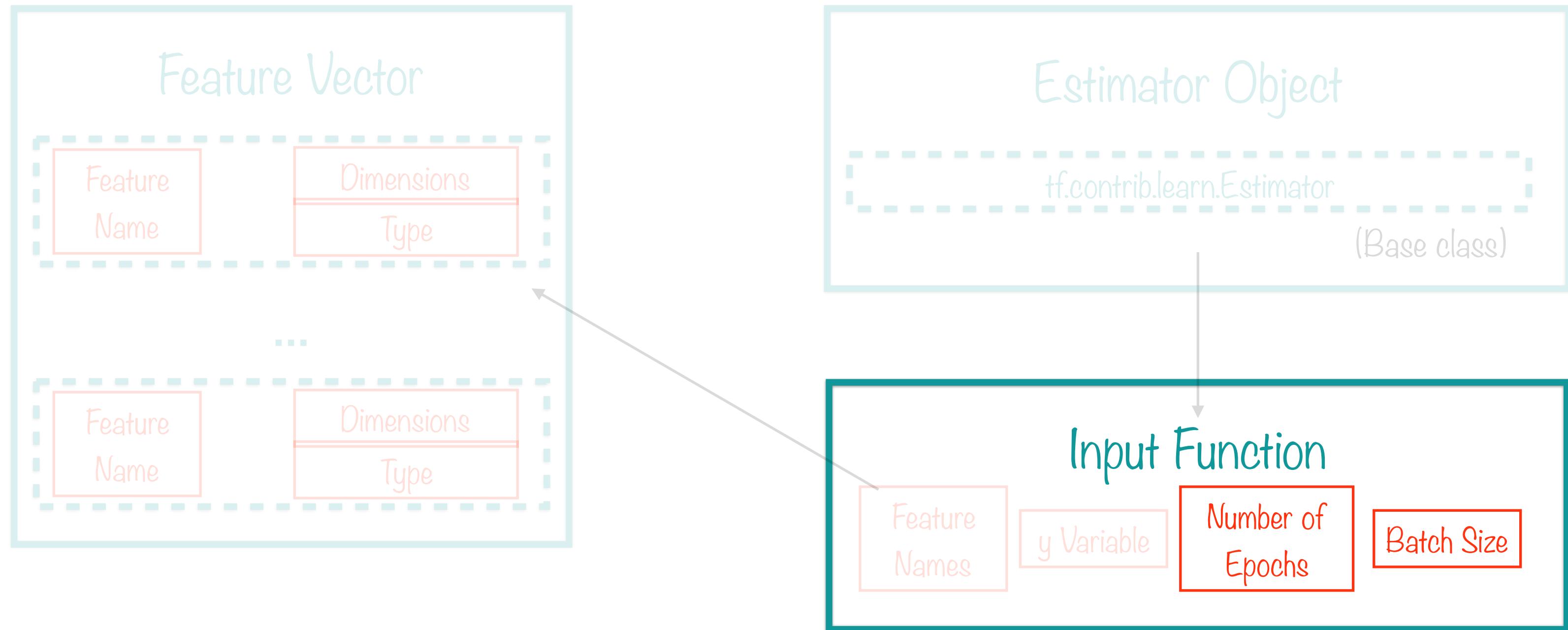
How Estimators Work



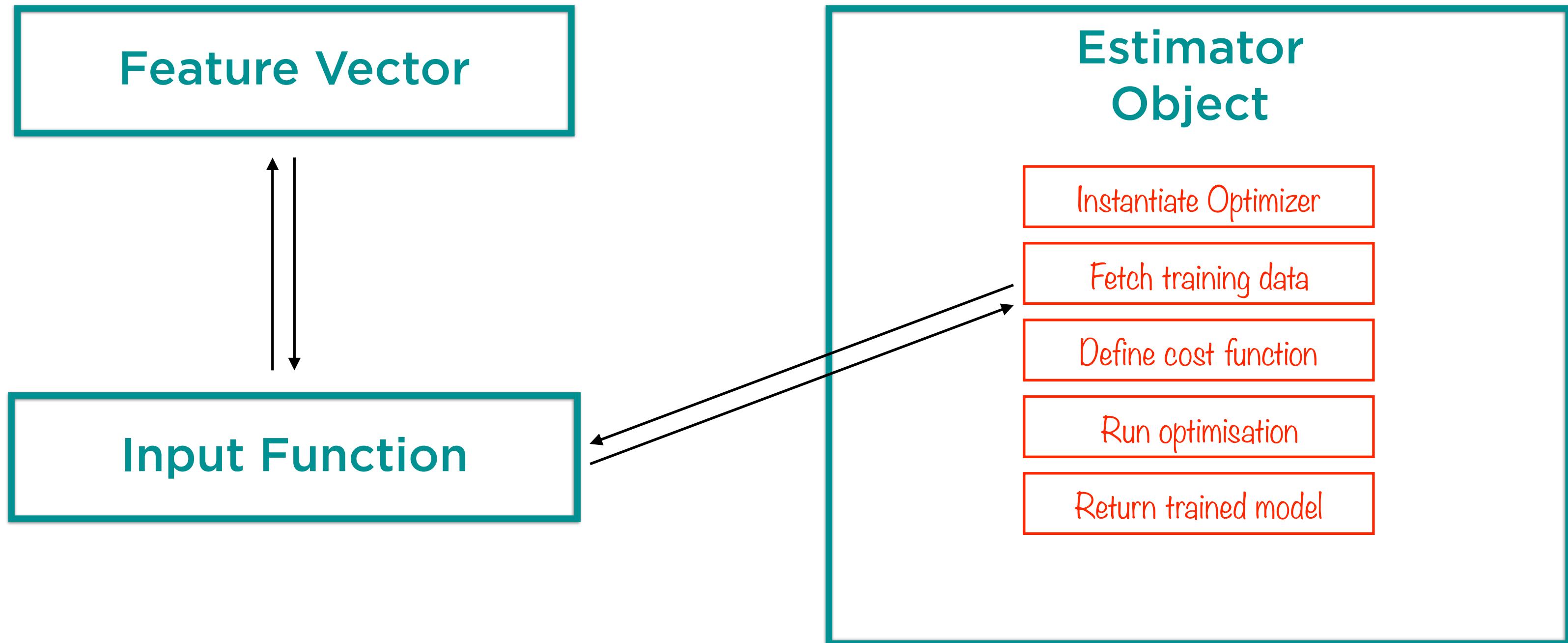
How Estimators Work



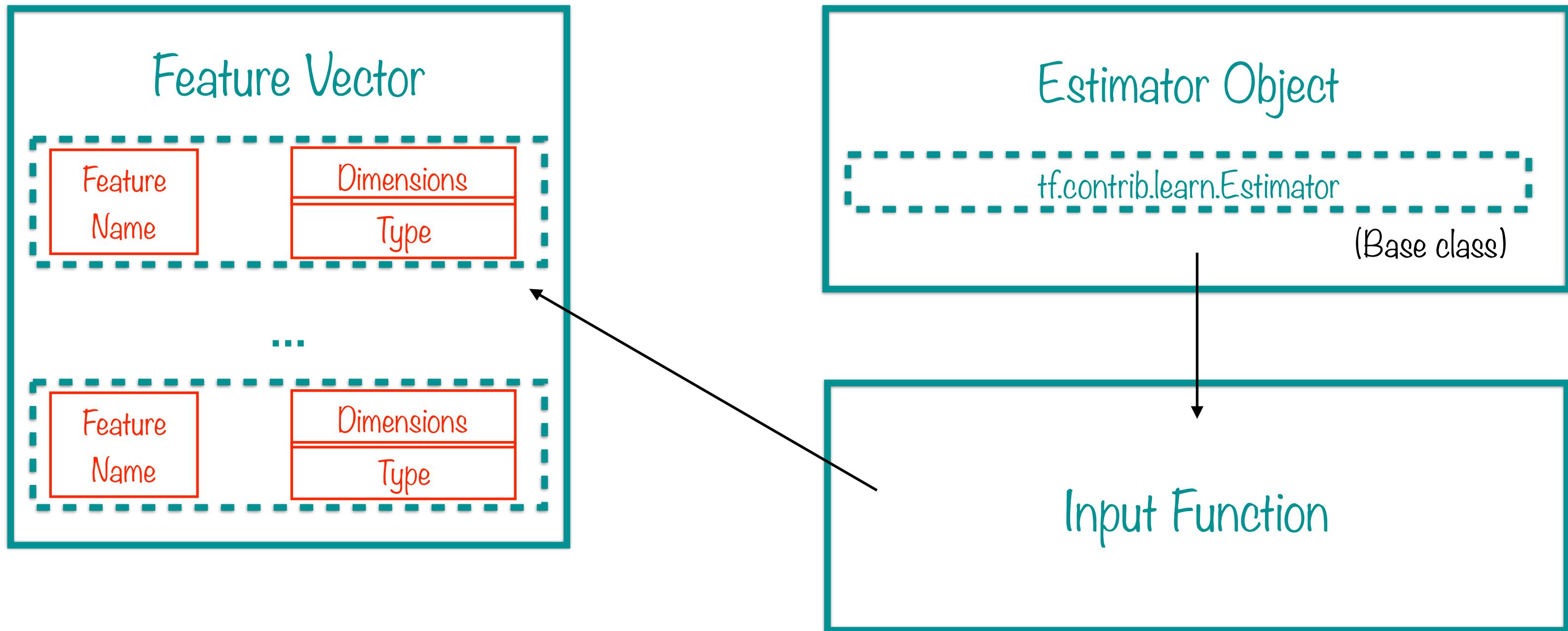
How Estimators Work



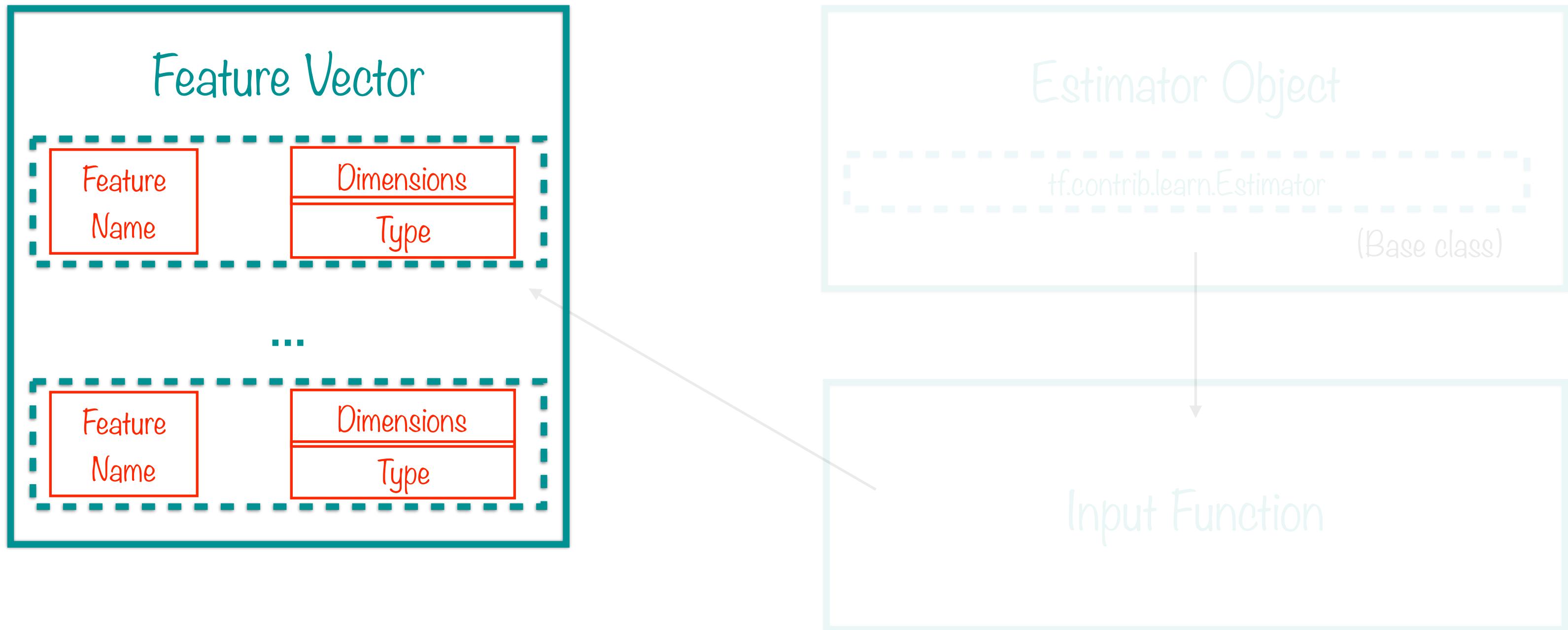
How Estimators Work



How Estimators Work



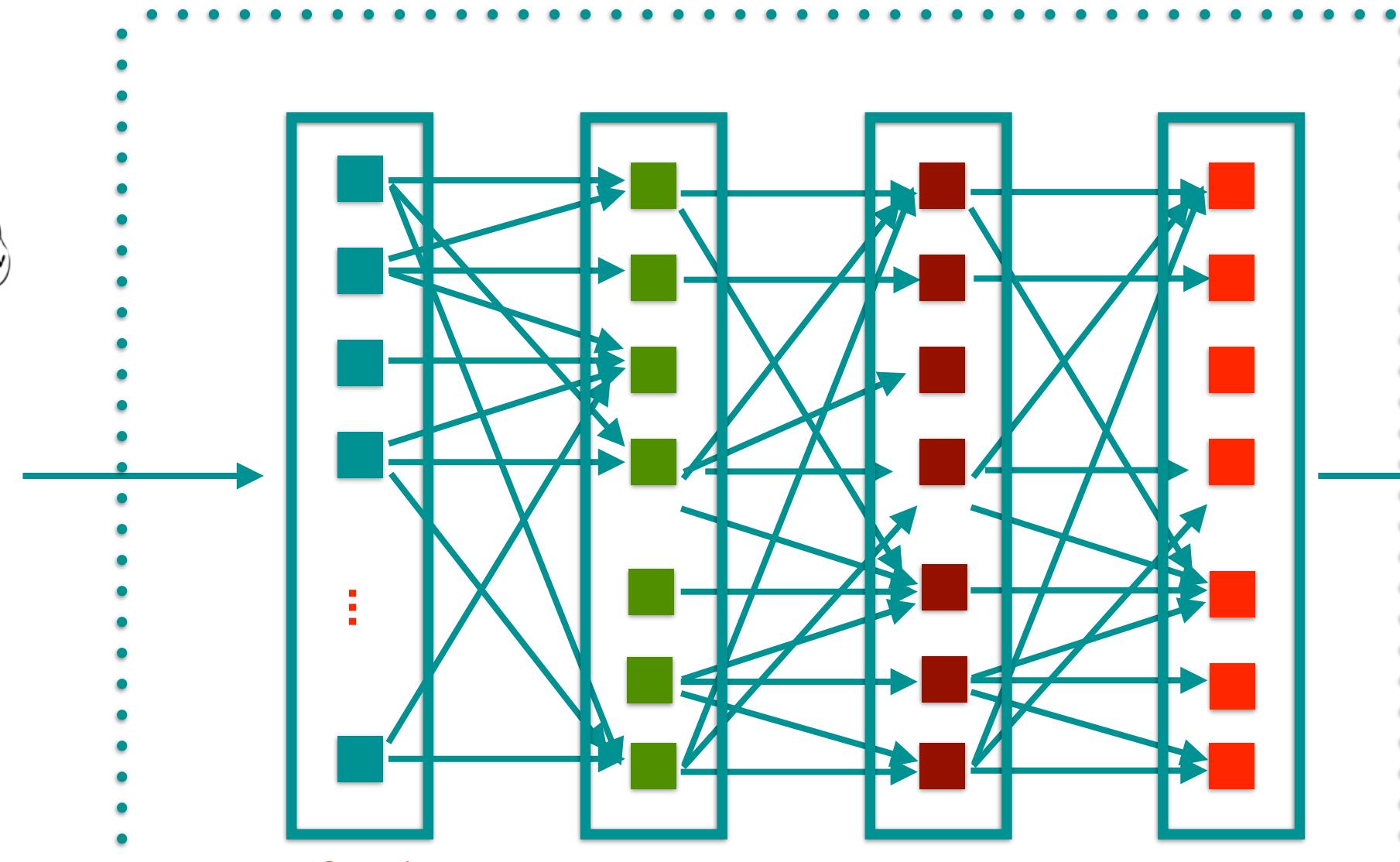
How Estimators Work



Complex Neural Networks



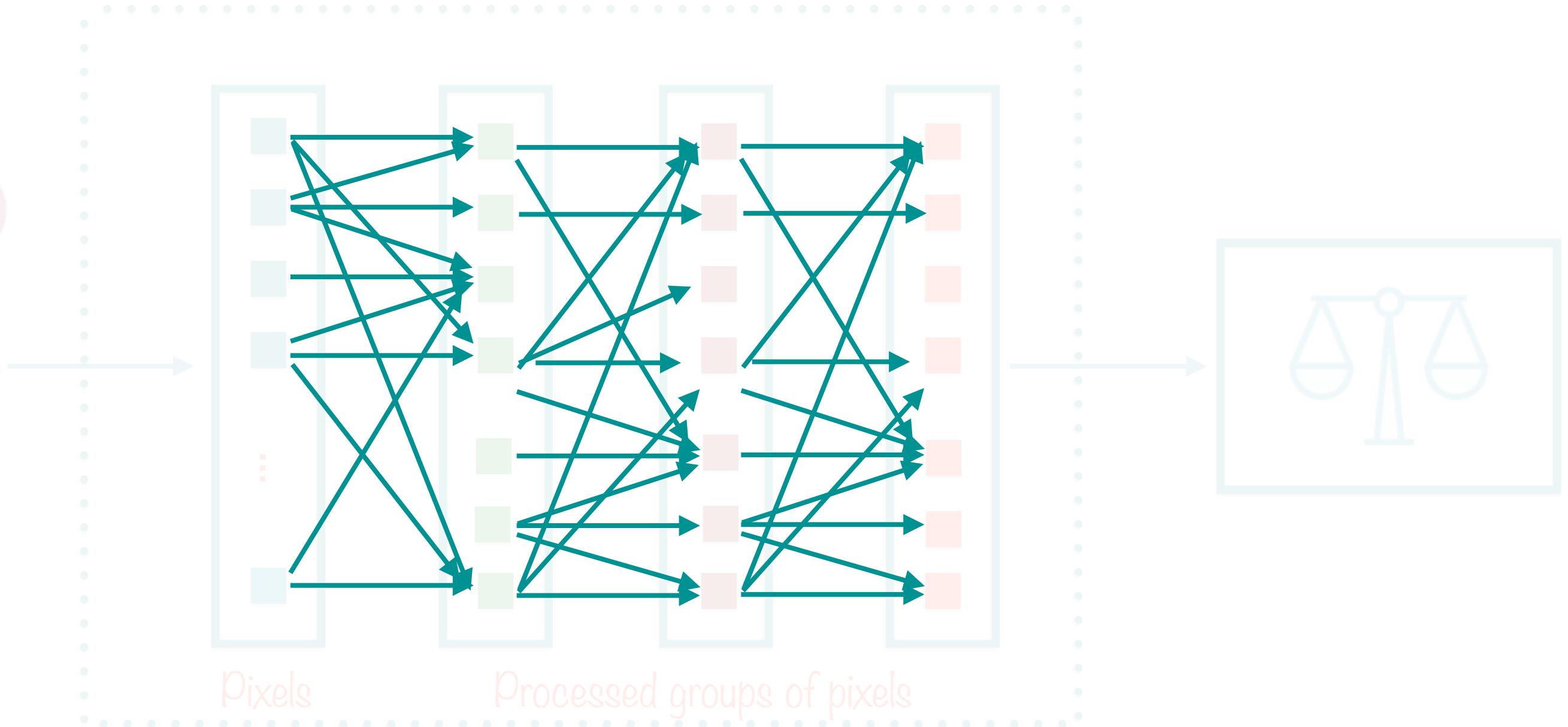
Corpus of
Images



Operations (nodes) on data
(edges)

ML-based Classifier

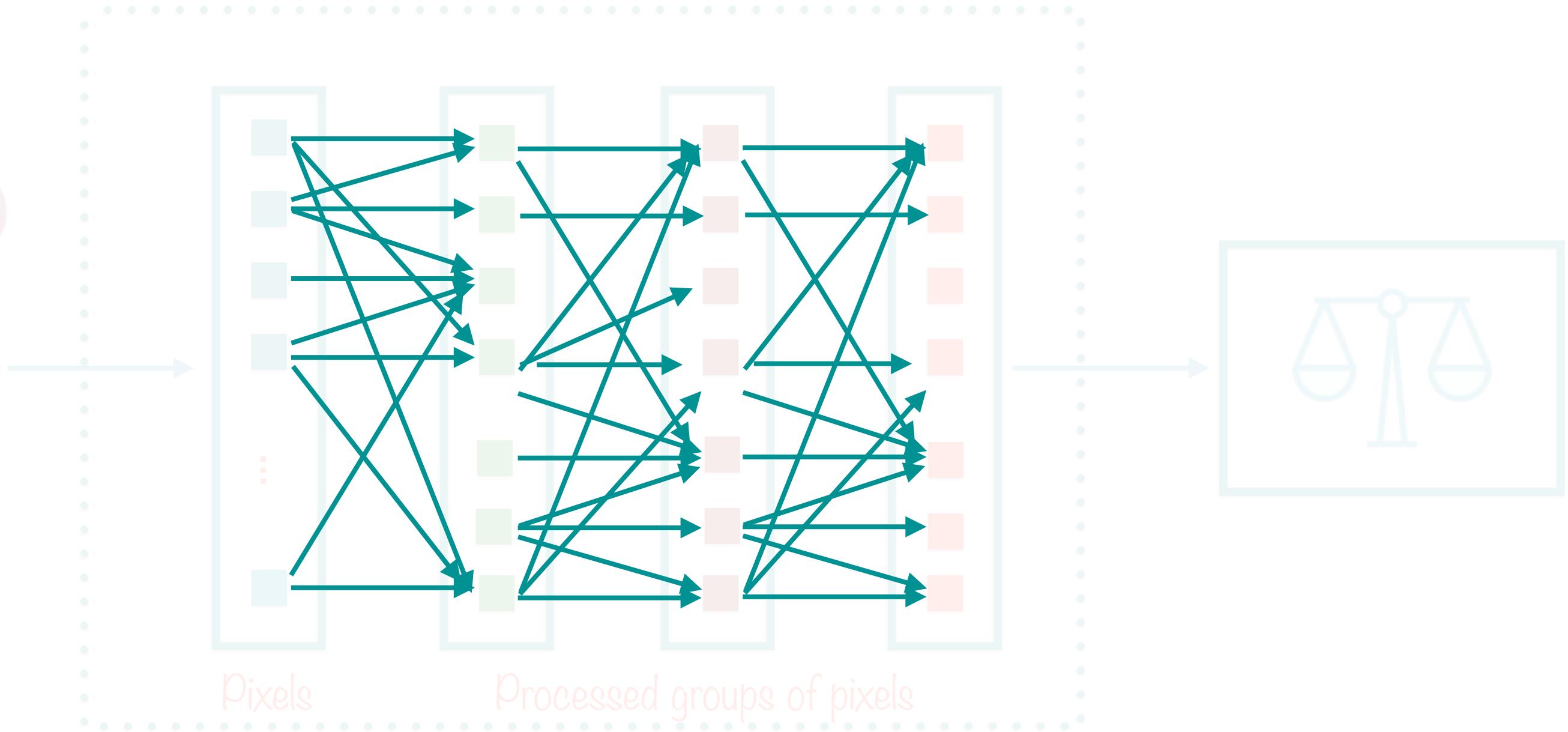
Complex Neural Networks



Neurons in a neural network can be connected in
very complex ways...

ML-based Classifier

Complex Neural Networks

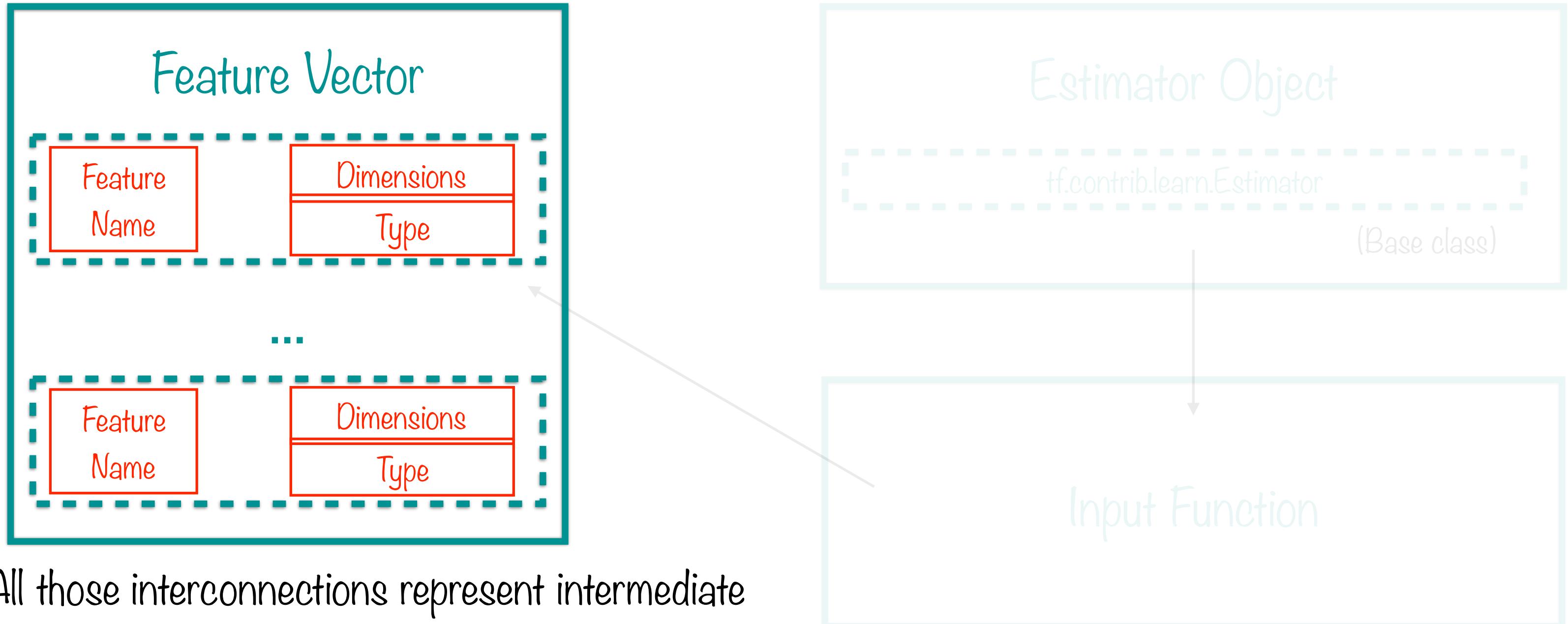


Corpus of
Images

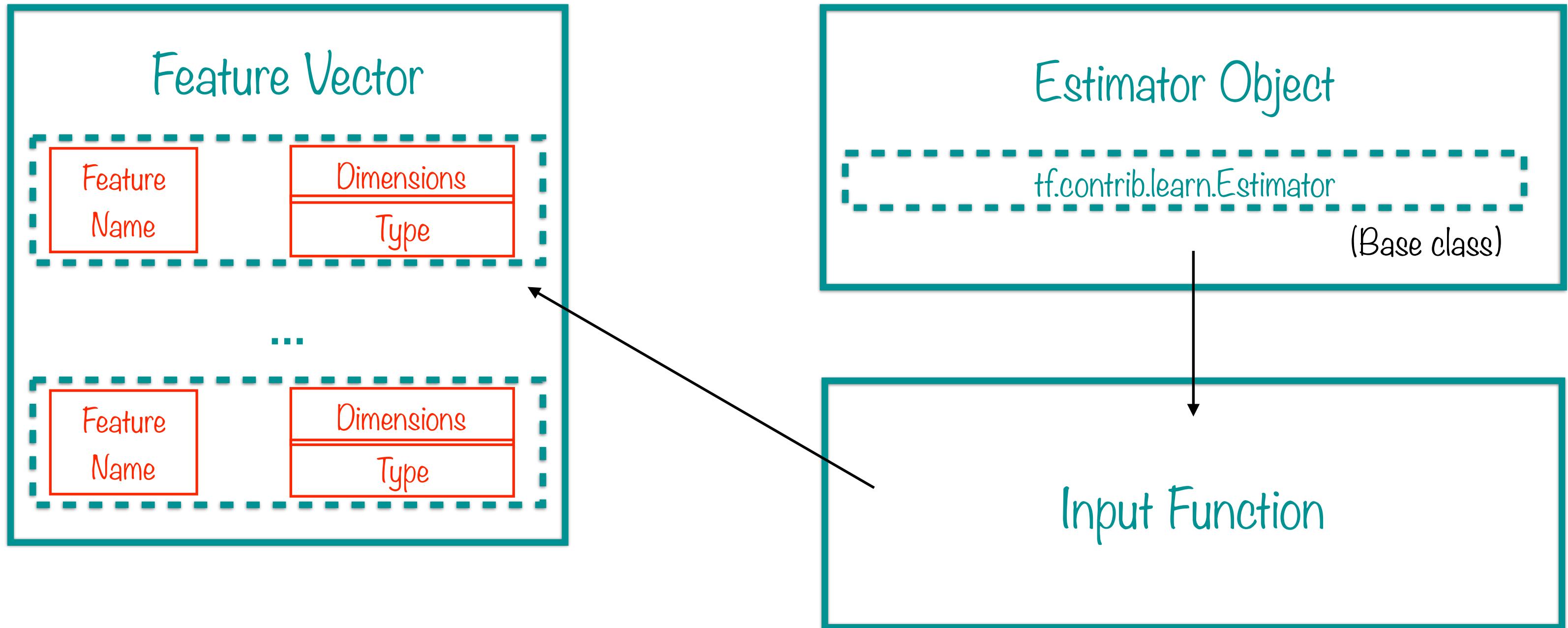
All those interconnections represent intermediate
feature vector data!

ML-based Classifier

How Estimators Work



How Estimators Work



Linear Regression in TensorFlow

Baseline
Non-TensorFlow implementation
Regular python code

Cost Function
Mean Square Error (MSE)
Quantifying goodness-of-fit

Training
Invoke optimizer in epochs
Batch size for each epoch



Computation Graph
Neural network of 1 neuron
Affine transformation suffices

Optimizer
Gradient Descent optimizers
Improving goodness-of-fit

Converged Model
Values of W and b
Compare to baseline

Logistic Regression in TensorFlow

Baseline
Non-TensorFlow implementation
Regular python code

Cost Function
Cross Entropy
Similarity of distribution

Training
Invoke optimizer in epochs
Batch size for each epoch

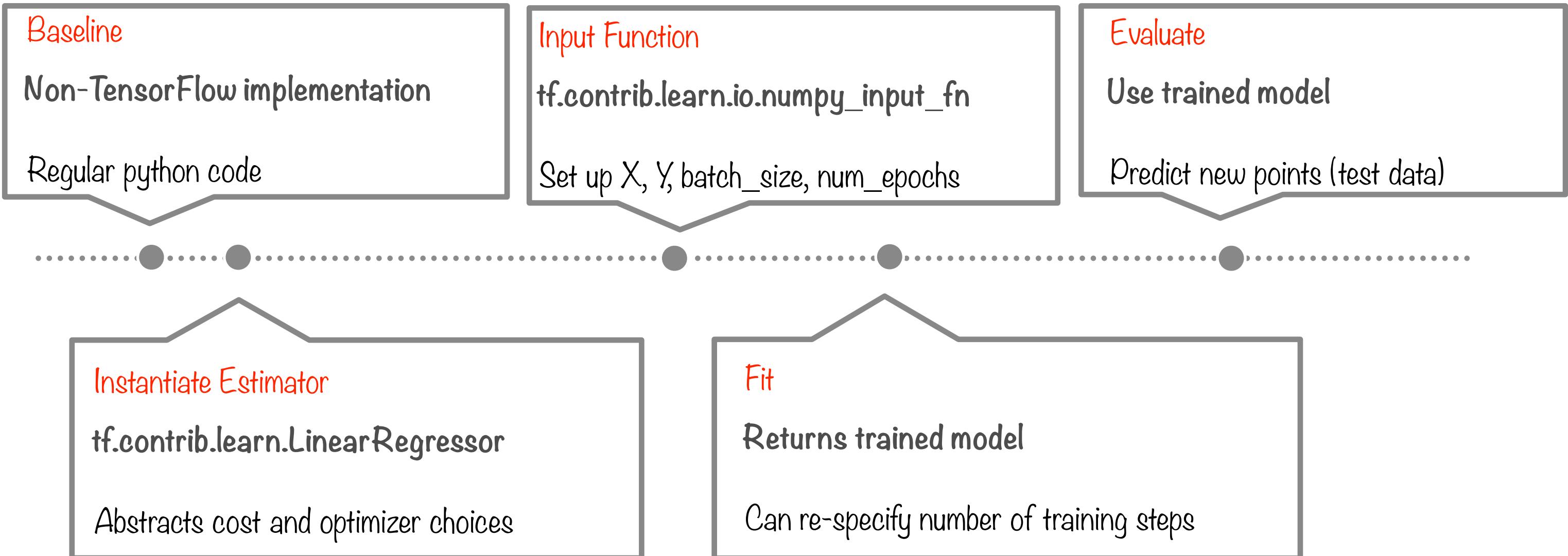


Computation Graph
Neural network of 1 neuron
Softmax activation required

Optimizer
Gradient Descent optimizers
Improving goodness-of-fit

Converged Model
Values of W and b
Compare to baseline

Linear Regression with an Estimator



Course Outline

Learning using Neurons

Linear Regression in TensorFlow

Logistic Regression in TensorFlow

Estimators

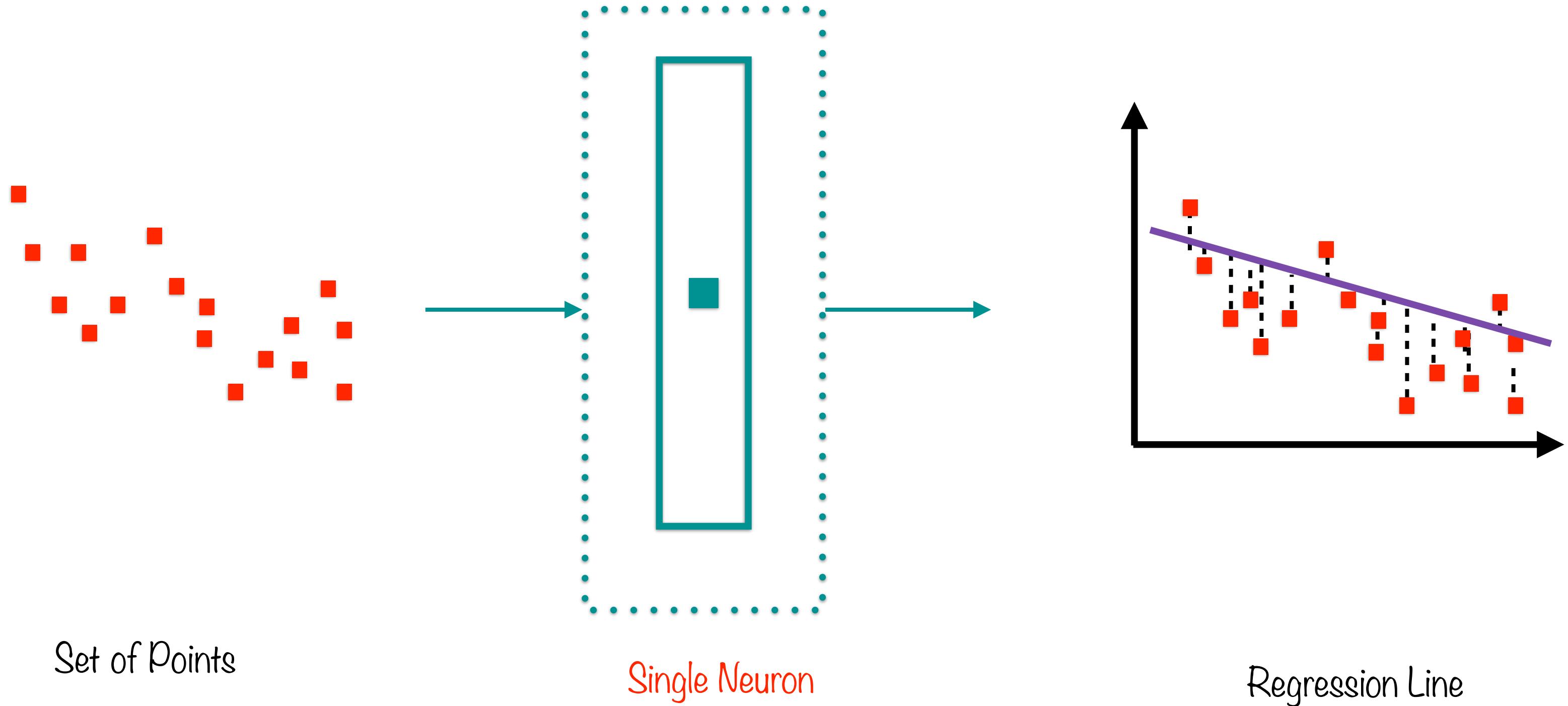
“Representation” ML-based systems figure out by themselves what features to pay attention to

$$y = Wx + b$$

“Learning” Regression

Regression can be reverse-engineered by a single neuron

Regression: The Simplest Neural Network

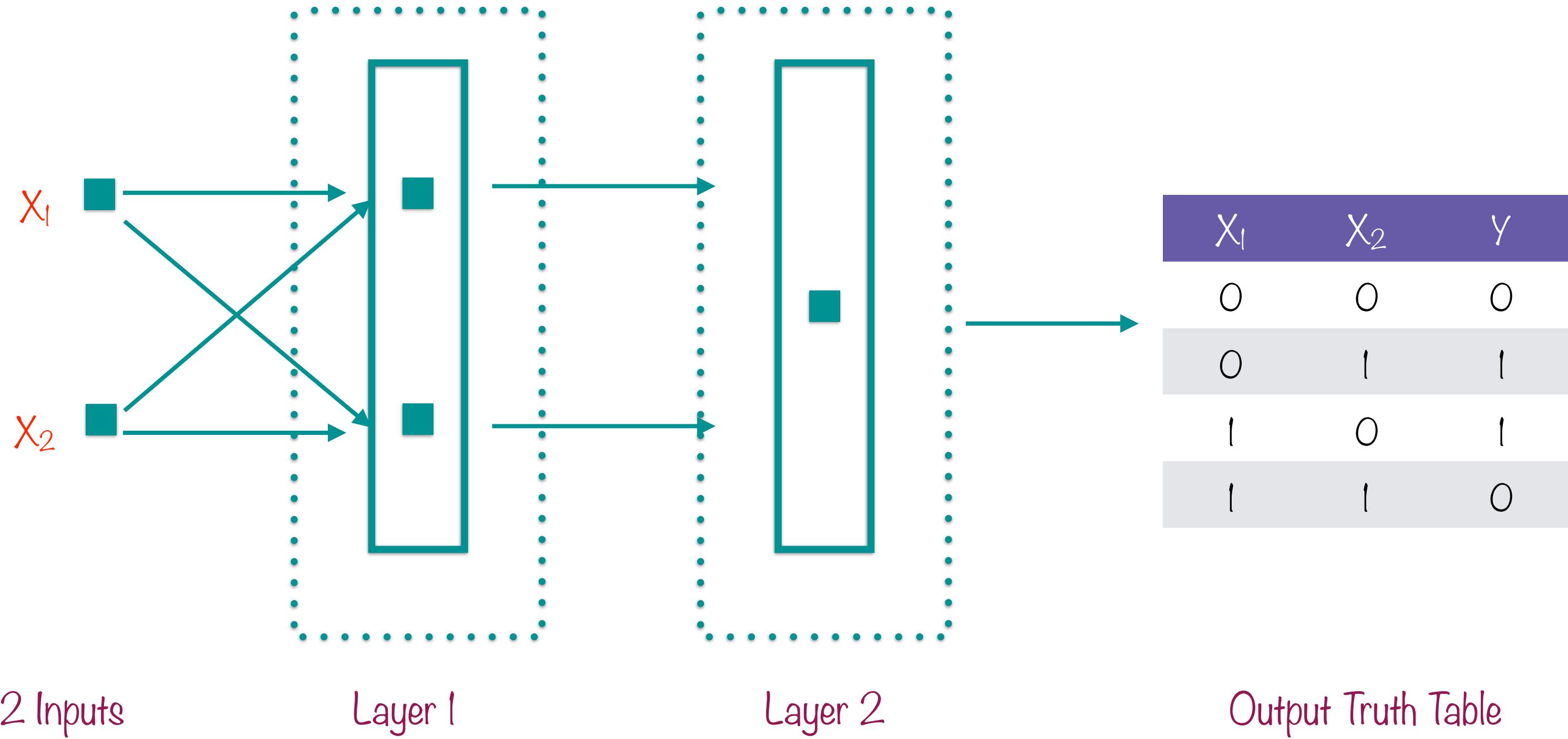


```
def XOR(x1,x2):  
    if (x1 == x2):  
        return 0  
    return 1
```

“Learning” XOR

The XOR function can be reverse-engineered using 3 neurons arranged in 2 layers

XOR: 3 Neurons, 2 Layers



```
def doSomethingReallyComplicated(x1,x2...):
```

```
...
```

```
...
```

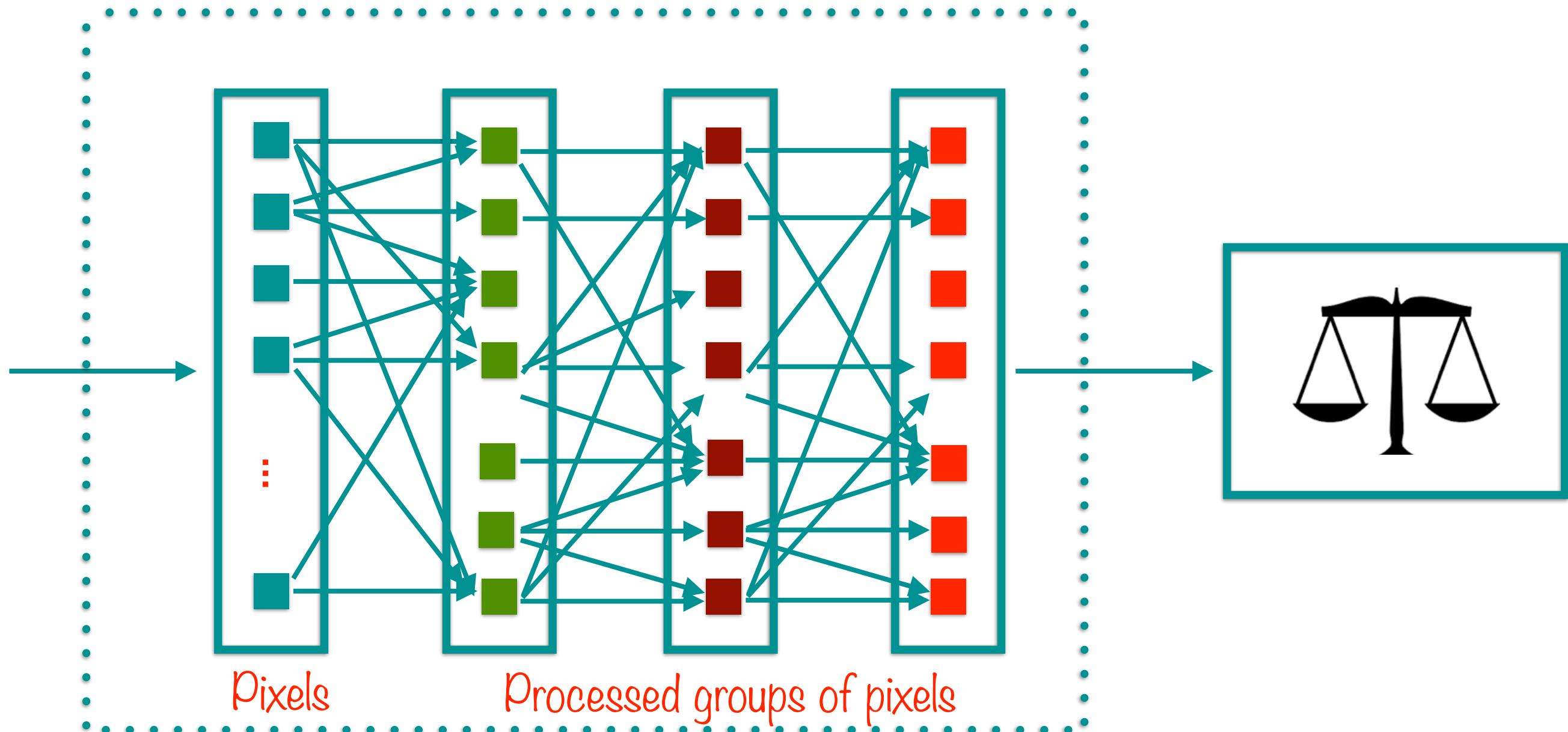
```
...
```

```
    return complicatedResult
```

“Learning” Arbitrarily Complex Functions

Adding layers to a neural network can “learn” (reverse-engineer) pretty much anything

Arbitrarily Complex Function



Corpus of
Images

Operations (nodes) on data
(edges)

ML-based Classifier

Linear Regression in TensorFlow

Baseline
Non-TensorFlow implementation
Regular python code

Cost Function
Mean Square Error (MSE)
Quantifying goodness-of-fit

Training
Invoke optimizer in epochs
Batch size for each epoch



Computation Graph
Neural network of 1 neuron
Affine transformation suffices

Optimizer
Gradient Descent optimizers
Improving goodness-of-fit

Converged Model
Values of W and b
Compare to baseline

Logistic Regression in TensorFlow

Baseline
Non-TensorFlow implementation
Regular python code

Cost Function
Cross Entropy
Similarity of distribution

Training
Invoke optimizer in epochs
Batch size for each epoch



Computation Graph
Neural network of 1 neuron
Softmax activation required

Optimizer
Gradient Descent optimizers
Improving goodness-of-fit

Converged Model
Values of W and b
Compare to baseline

Logistic Regression Using Estimators
