Building Simple Neural Networks

Janani Ravi
CO-FOUNDER, LOONYCORN
www.loonycorn.com

Overview

PyTorch uses the Autograd library for backpropagation during training

Autograd relies on reverse-mode automatic differentiation

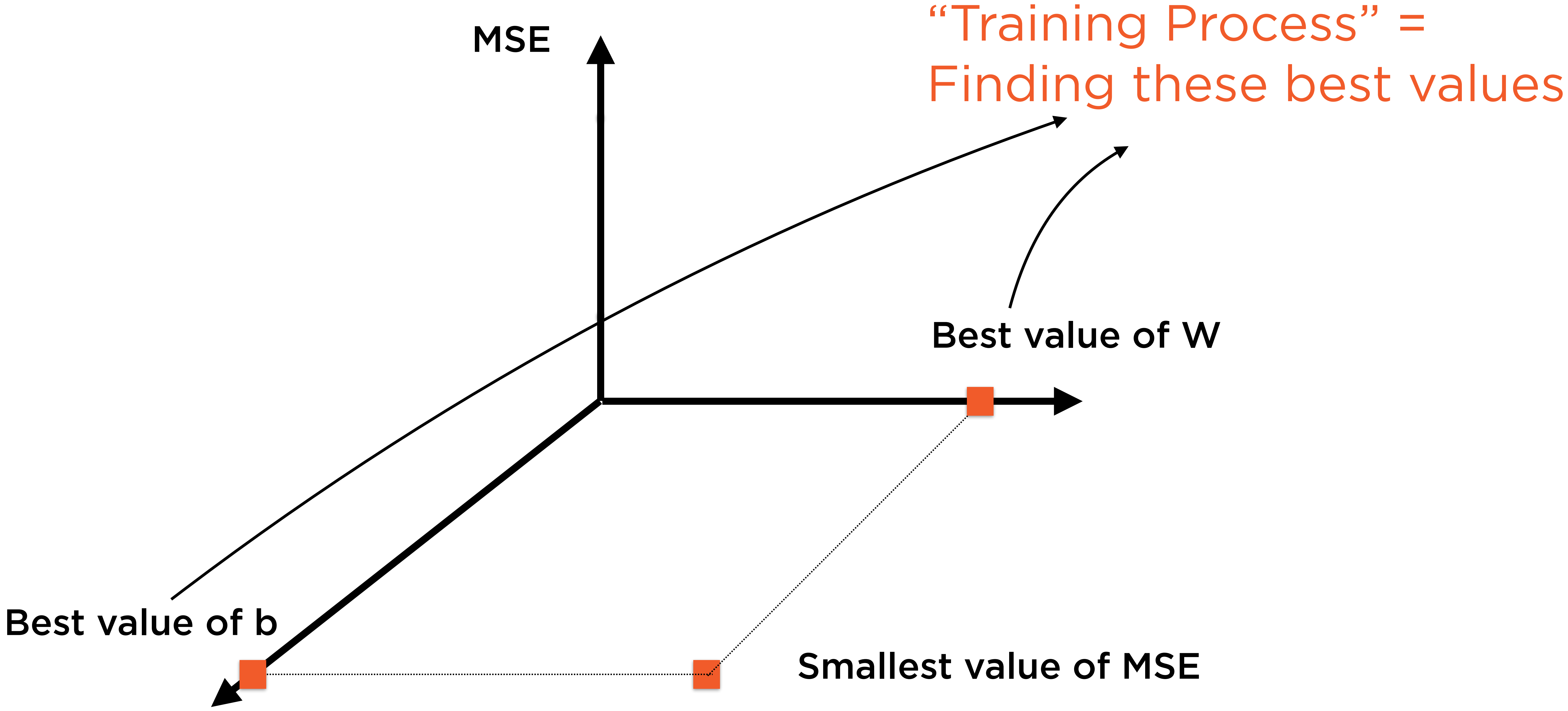
Conceptually similar to autodiff in TensorFlow

Build a simple NN model for regression and classification

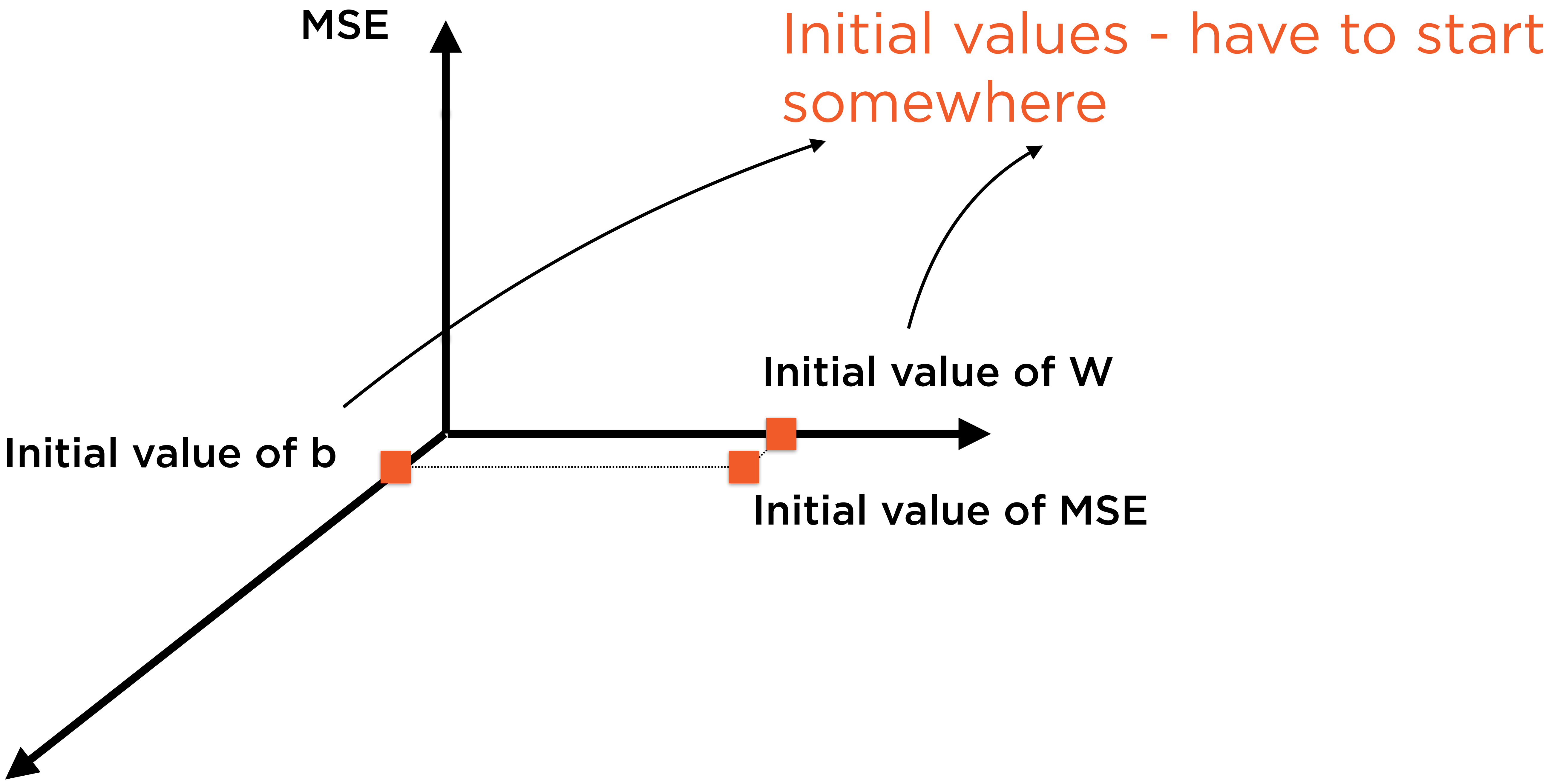
NLL as loss function and LogSoftMax as output layer

Training a neural network uses Gradient Descent to find the weights of the model parameters

“Training” the Algorithm

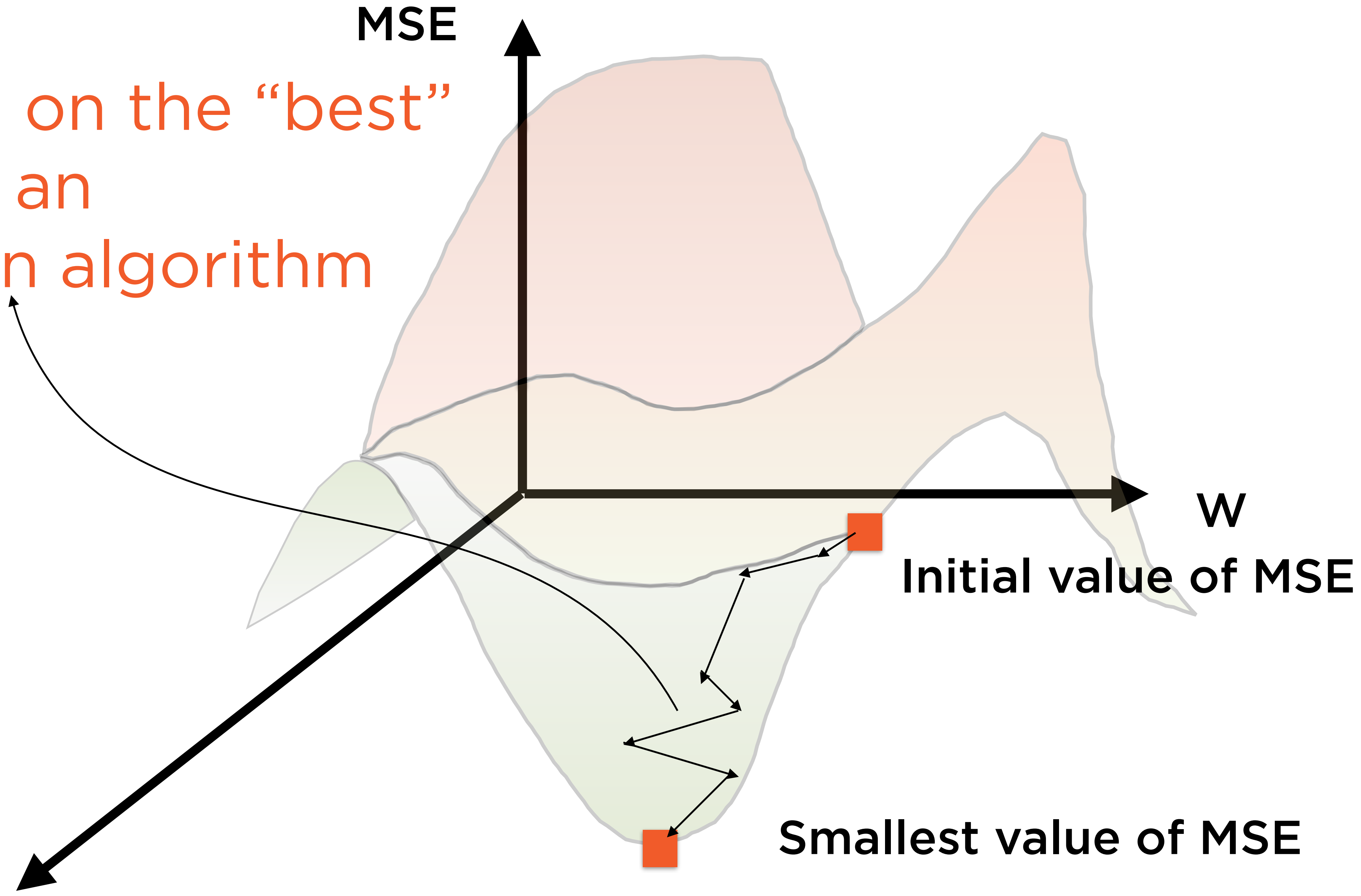


Start Somewhere



“Gradient Descent”

Converging on the “best” value using an optimization algorithm



$$\text{MSE} = \text{Mean Square Error of Loss}$$

$$\text{Loss} = \theta = y_{predicted} - y_{actual}$$

MSE

Mean Square Error (MSE) is the metric to be minimized during training of regression model

Given x, model outputs predicted value of y

$$\text{Loss} = \theta = y_{predicted} - y_{actual}$$

Loss Function θ

Loss function measures inaccuracy of model on a specific instance

Actual label, available in training data

$\text{Loss} = \theta = y_{predicted} - y_{actual}$

Loss Function θ

Loss function measures inaccuracy of model on a specific instance

In NN with 1 Neuron:

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1) = (\partial\theta/\partial W_1, \partial\theta/\partial b_1)$$

Gradient: Vector of Partial Derivatives

For a function $y = f(x_1, x_2, x_3)$, the Greek character "nabla" (∇) denotes the gradient

Partial derivative of loss w.r.t to parameter W

Holding all other parameters and the input constant - **how much does loss change when you tweak W**

```
In NN with 1 Neuron:
```

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1) = (\partial\theta/\partial W_1, \partial\theta/\partial b_1)$$

Gradient: Vector of Partial Derivatives

For a function $y = f(x_1, x_2, x_3)$, the Greek character "nabla" (∇) denotes the gradient

`In NN with 1 Neuron:`

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1) = (\partial\theta/\partial W_1, \partial\theta/\partial b_1)$$

Gradient: Vector of Partial Derivatives

For a function $y = f(x_1, x_2, x_3)$, the Greek character "nabla" (∇) denotes the gradient

`In NN with 1 Neuron:`

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1) = (\partial\theta/\partial W_1, \partial\theta/\partial b_1)$$

Gradient Descent to Minimize Loss

Find values of W_1, b_1 where loss has “lowest” gradient - i.e. minimize gradient of θ

```
Condition of minimum:

Gradient(θ) = ∇θ(W₁, b₁) = (∂θ/∂W₁, ∂θ/∂b₁) = zero
```

Gradient Descent to Minimize Loss

Find values of W_1, b_1 where loss has "lowest" gradient - i.e. minimize gradient of θ

In NN with 1 Neuron:

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1) = (\partial\theta/\partial W_1, \partial\theta/\partial b_1)$$

Gradient Descent to Minimize Loss

Find values of W_1, b_1 where loss has "lowest" gradient - i.e. minimize gradient of θ

In NN with 10,000 Neurons:

$$\text{Gradient}(\theta) = \nabla\theta(W_1,\ b_1 \ldots W_{10000},\ b_{10000})$$
$$= (\partial\theta/\partial W_1,\ \partial\theta/\partial b_1,\ \ldots \partial\theta/\partial W_{10000},\ \partial\theta/\partial b_{10000})$$

Gradient Descent for Complex Networks

The gradient vector gets very large for complex networks, need sophisticated math to calculate and optimize

Actually Calculating Gradients

Symbolic Differentiation

Conceptually simple but hard to implement

Numeric Differentiation

Easy to implement but won't scale

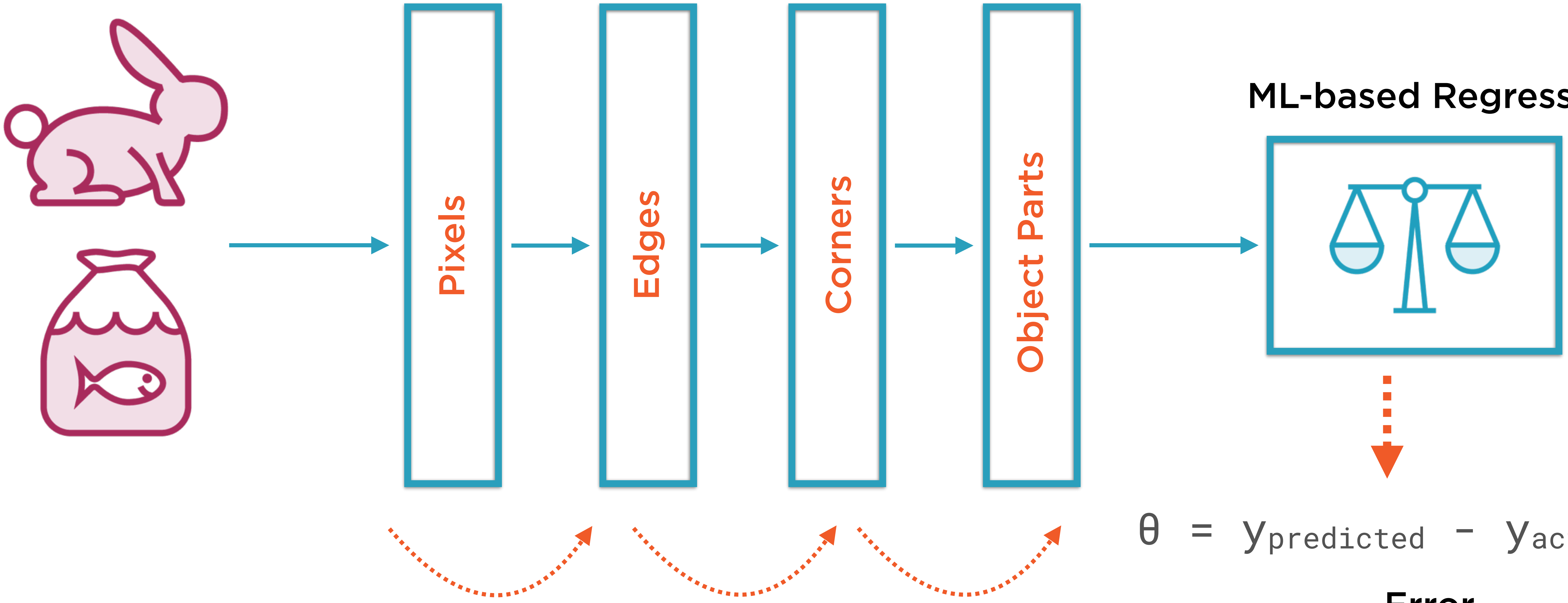
Automatic Differentiation

Conceptually difficult but easy to implement

PyTorch, TensorFlow and other packages rely on automatic differentiation

These gradients are used to update the model parameters

Forward Pass for Prediction

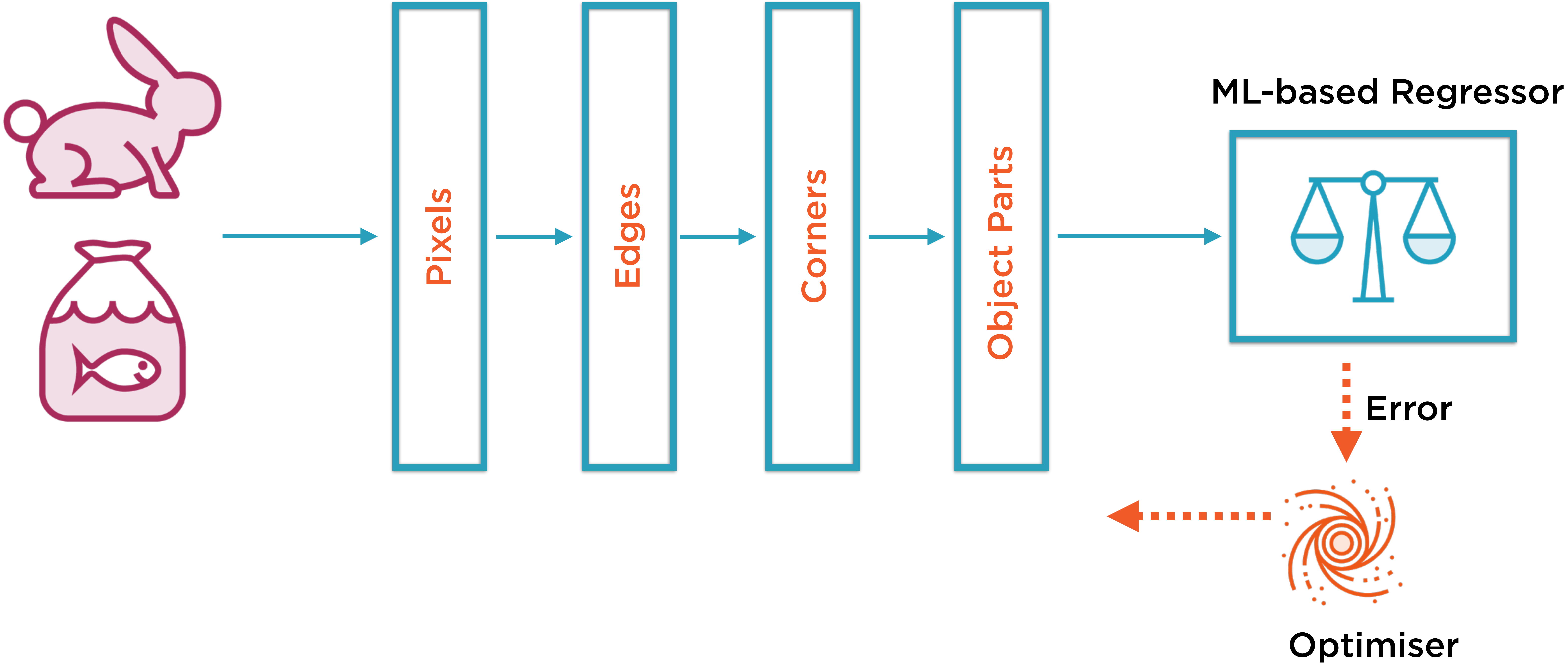


$\theta = y_{predicted} - y_{actual}$

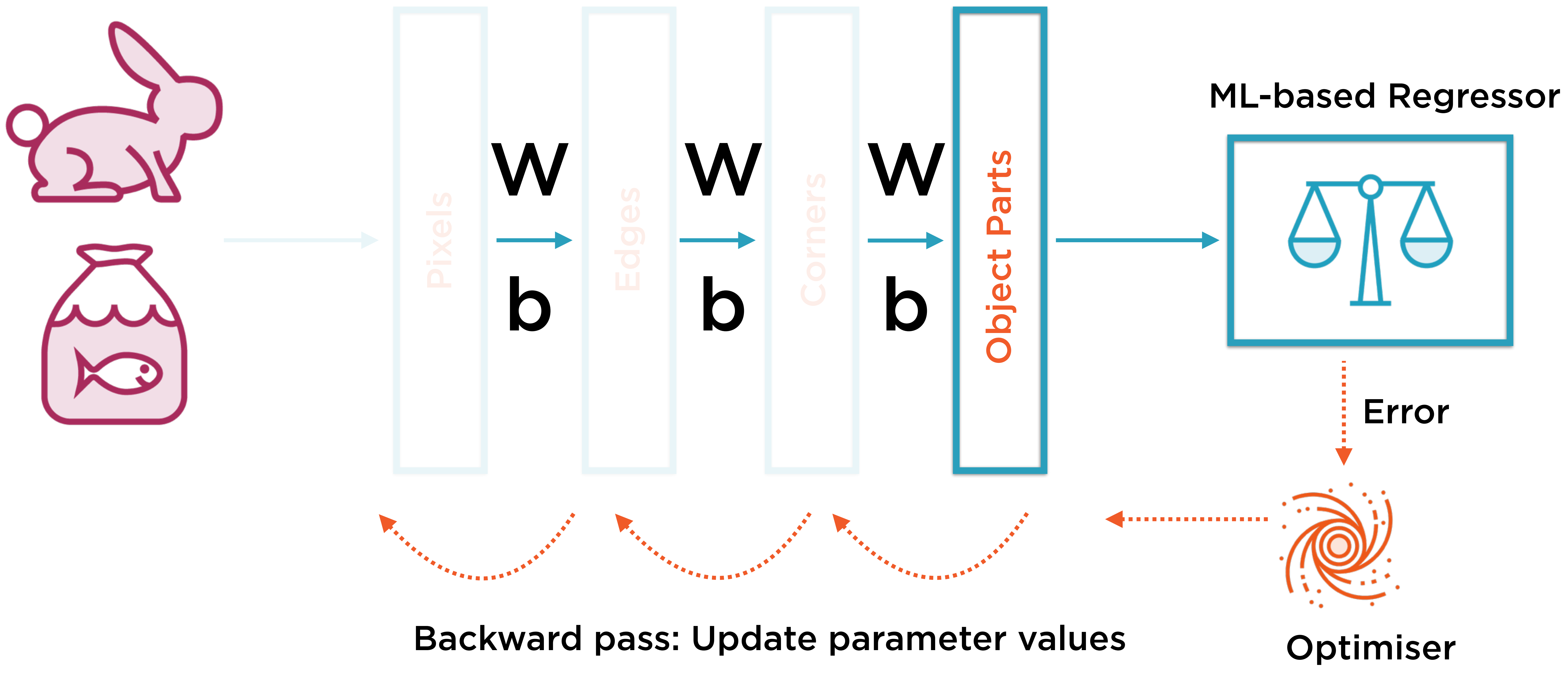
Error

Forward pass: Calculate $y_{predicted}$ and error

Backward Pass to Update Weights



Backward Pass to Update Weights



Autograd is the PyTorch package for calculating gradients for back propagation

Demo

Introducing Autograd in PyTorch

Reverse Auto-differentiation in Backpropagation

Back propagation is implemented using a technique called reverse auto-differentiation

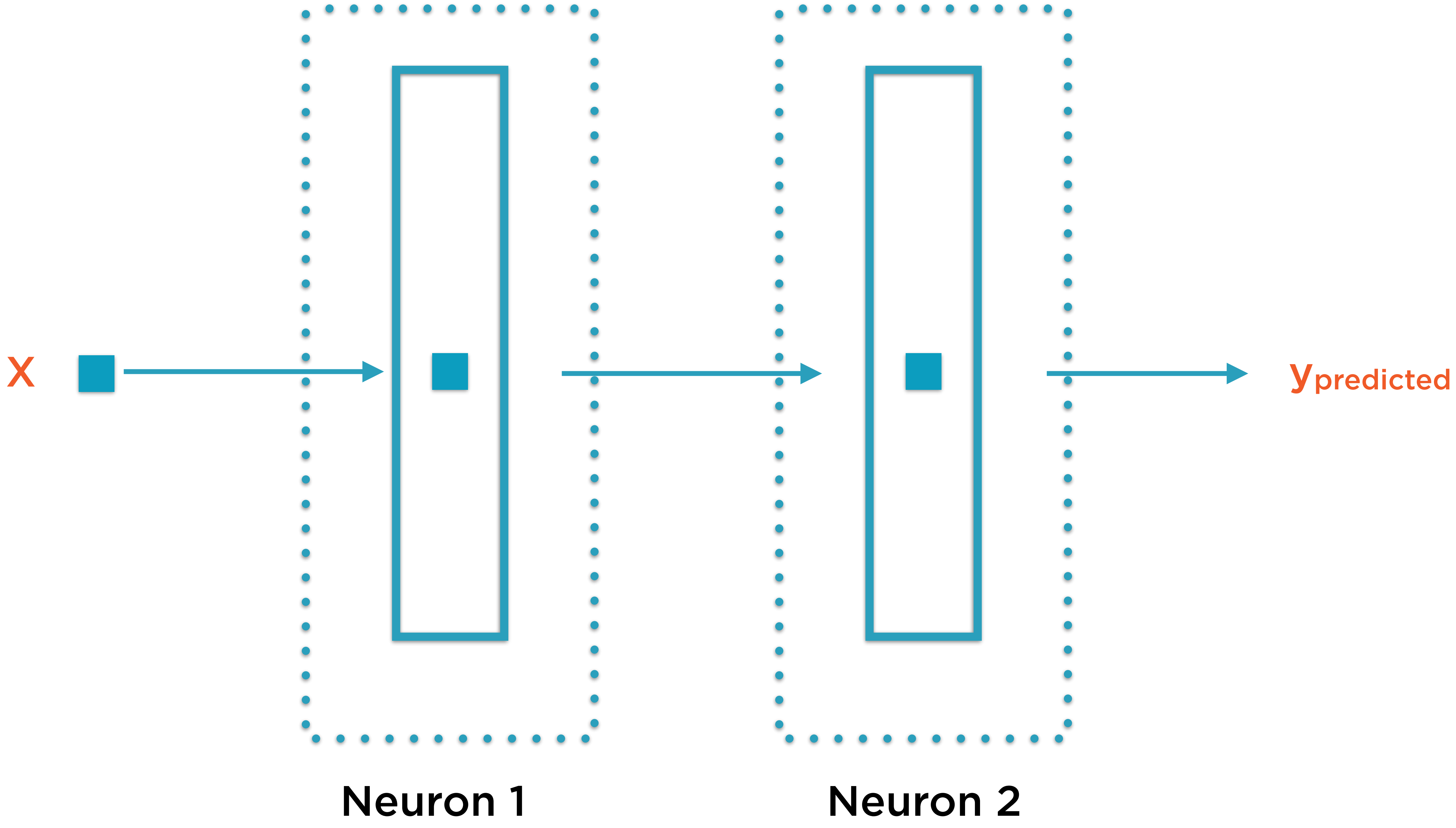
In NN with 10,000 Neurons:

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1 \ldots W_{10000}, b_{10000})$$
$$= (\partial\theta/\partial W_1, \partial\theta/\partial b_1, \ldots \partial\theta/\partial W_{10000}, \partial\theta/\partial b_{10000})$$

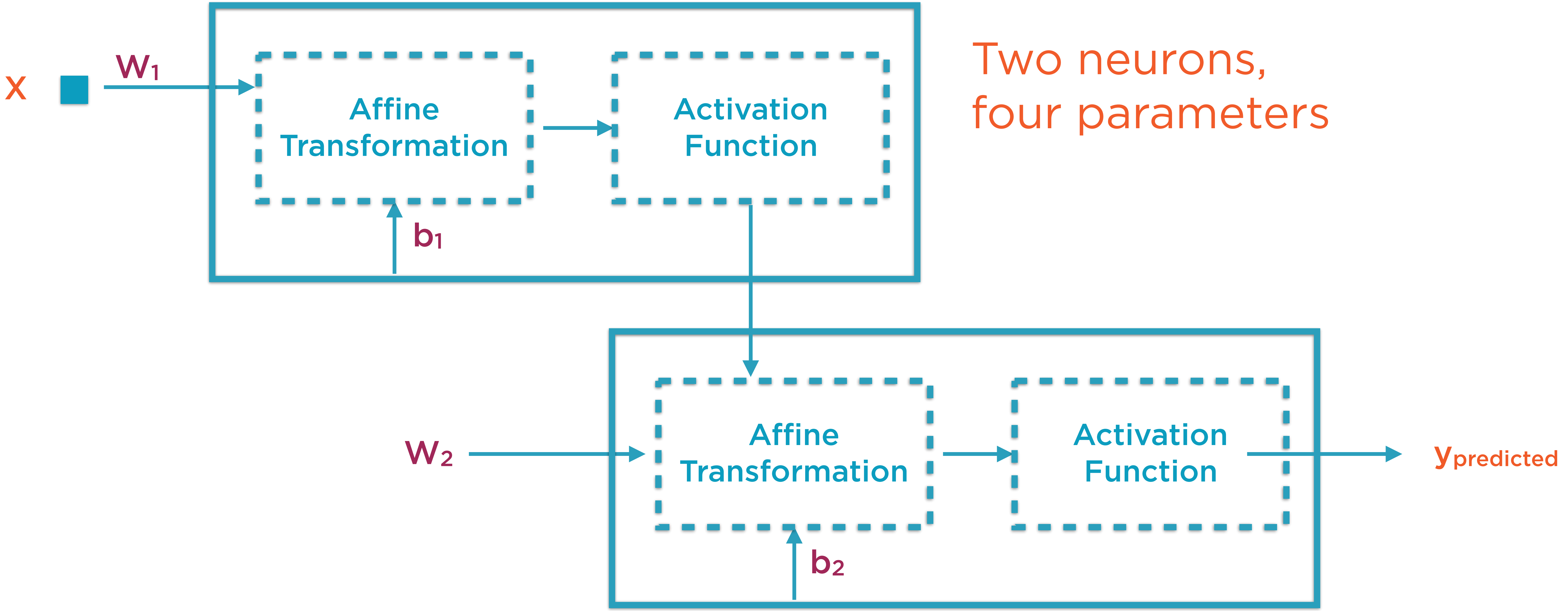
Gradient Descent for Complex Networks

The gradient vector gets very large for complex networks, need sophisticated math to calculate and optimize

Simple Example: Two Neurons, Linear Network

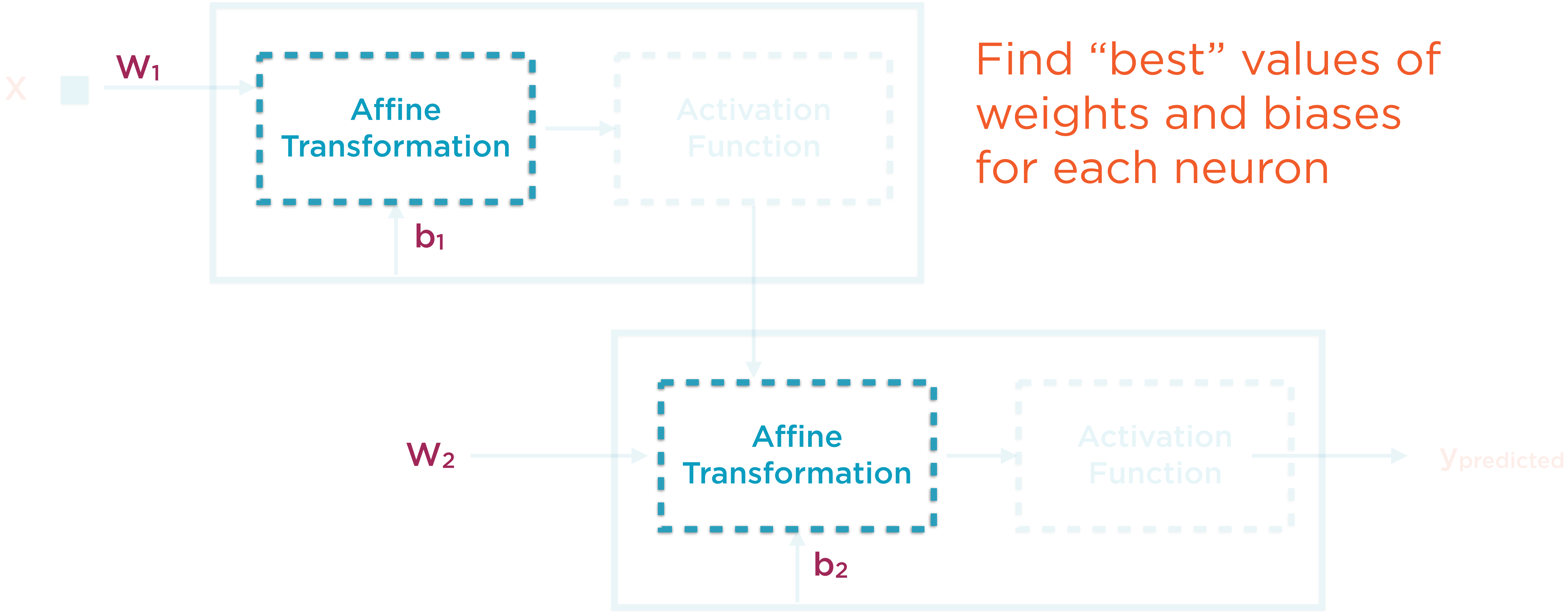


Simple Example: Two Neurons, Linear Network



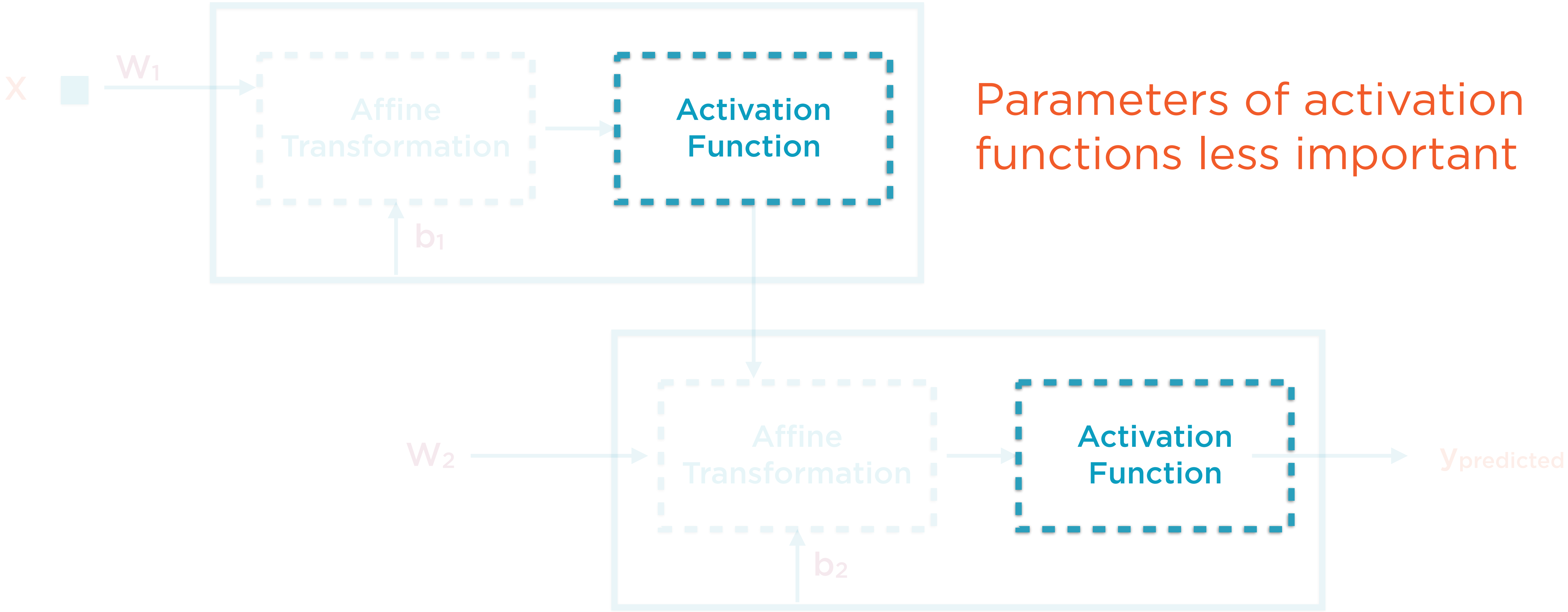
Two neurons, four parameters

Objective: "Best" Parameter Values



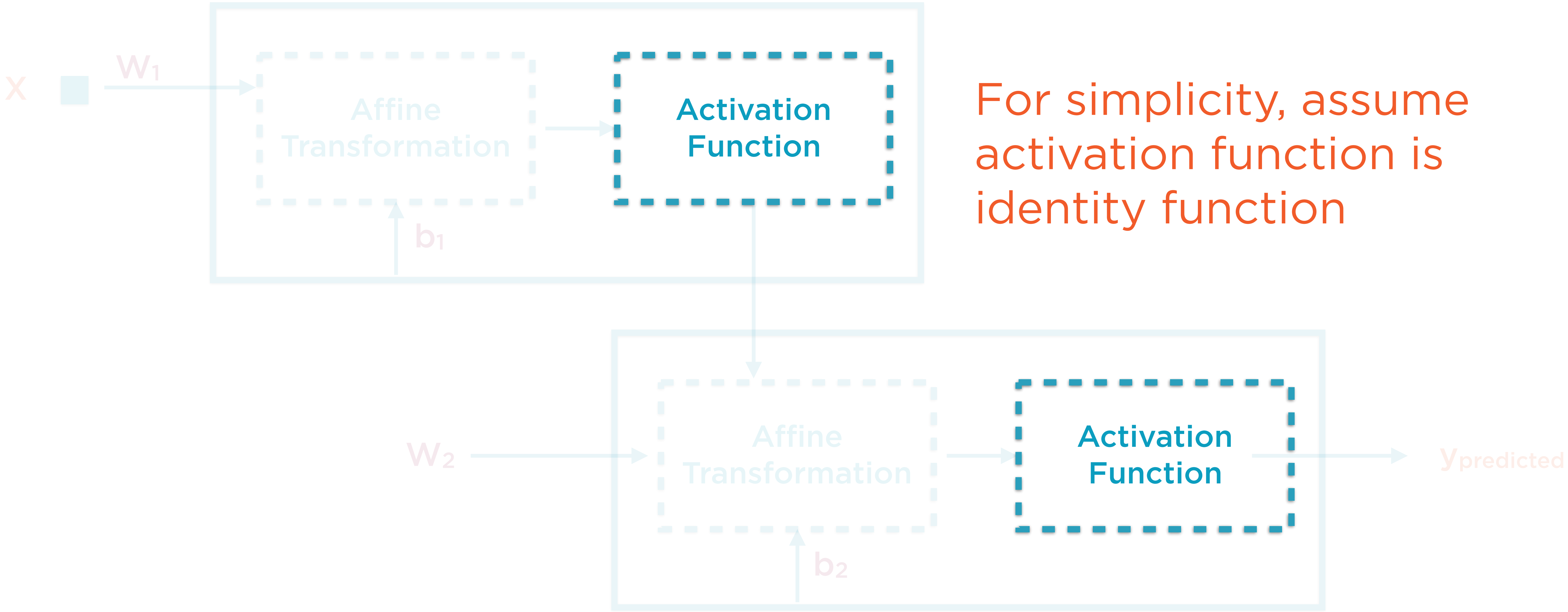
Find "best" values of weights and biases for each neuron

Objective: "Best" Parameter Values



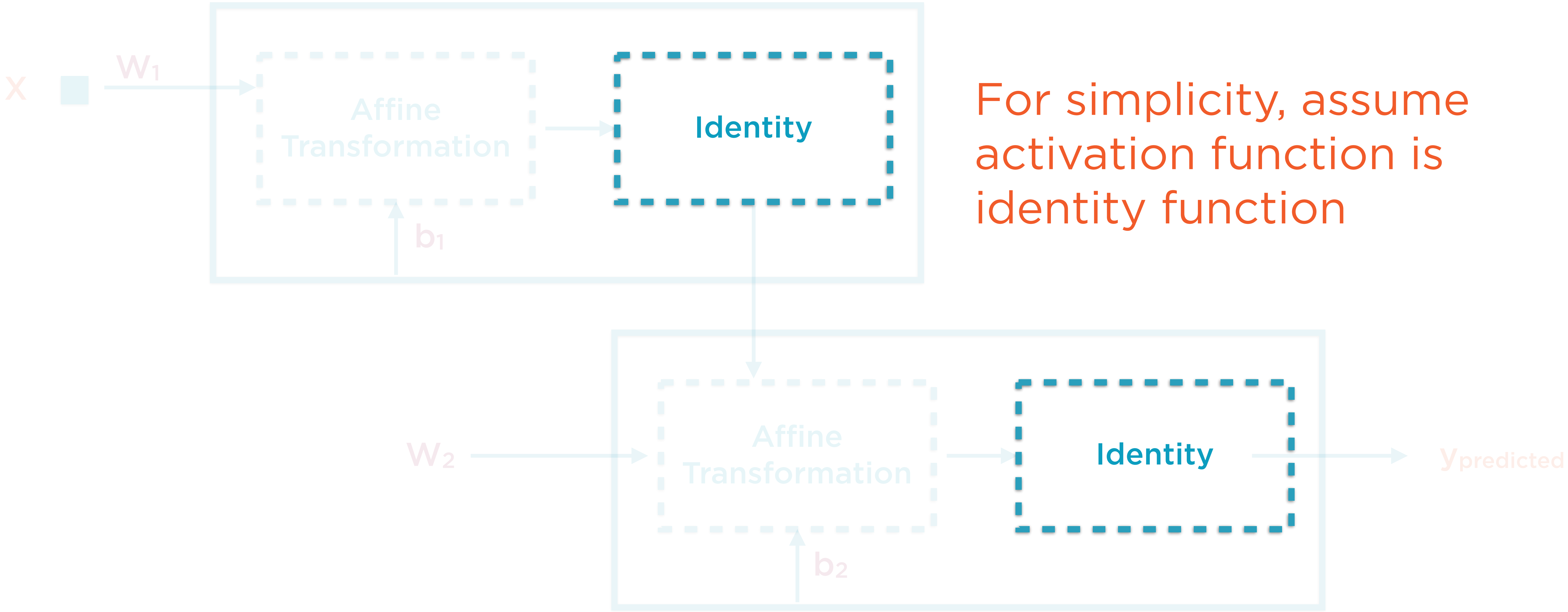
Parameters of activation functions less important

Objective: "Best" Parameter Values



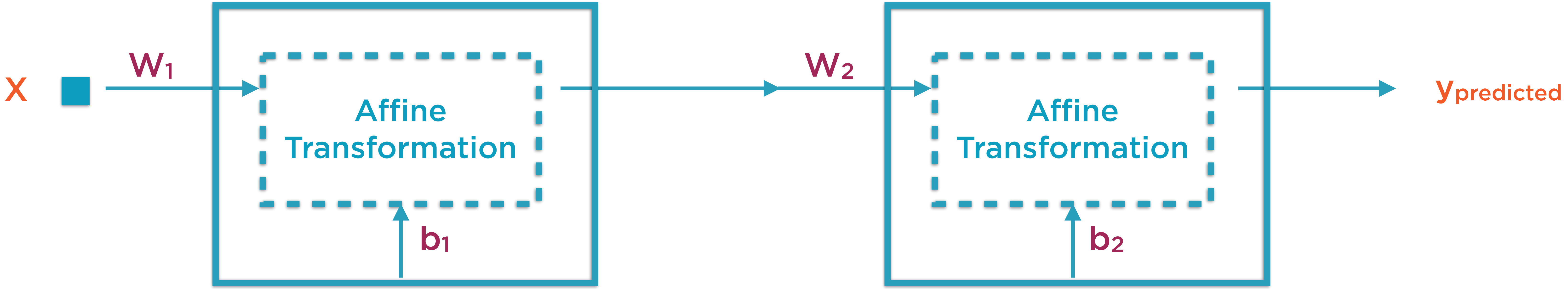
For simplicity, assume activation function is identity function

Objective: "Best" Parameter Values



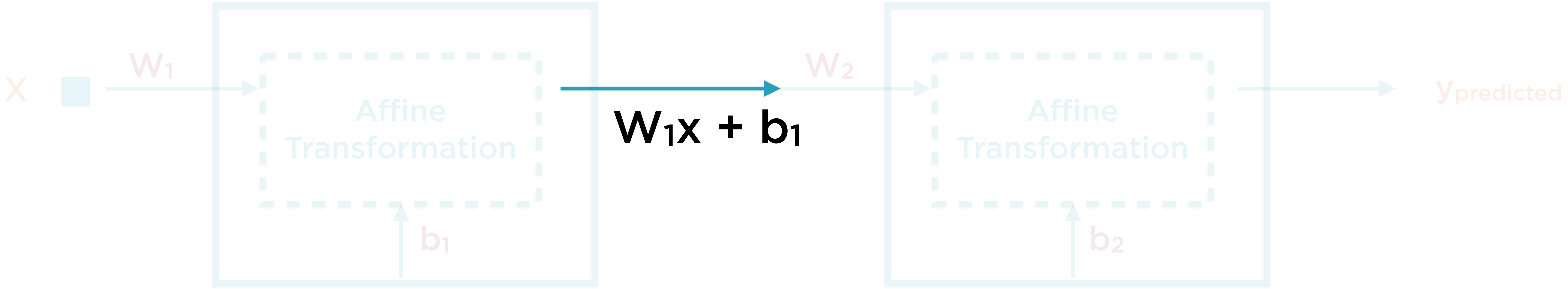
For simplicity, assume activation function is identity function

Simple Example: Two Neurons, Linear Network



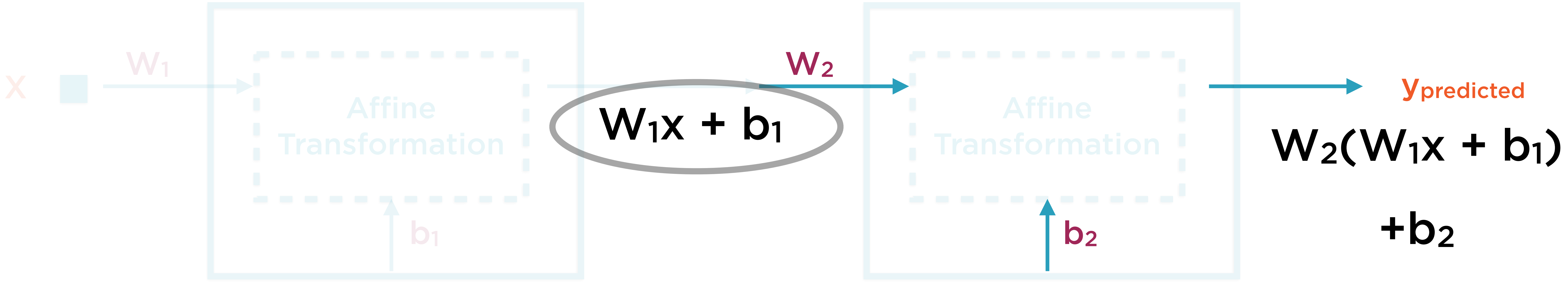
Greatly simplifies our little example

Simple Example: Two Neurons, Linear Network



Output of first neuron = $W_1x + b_1$

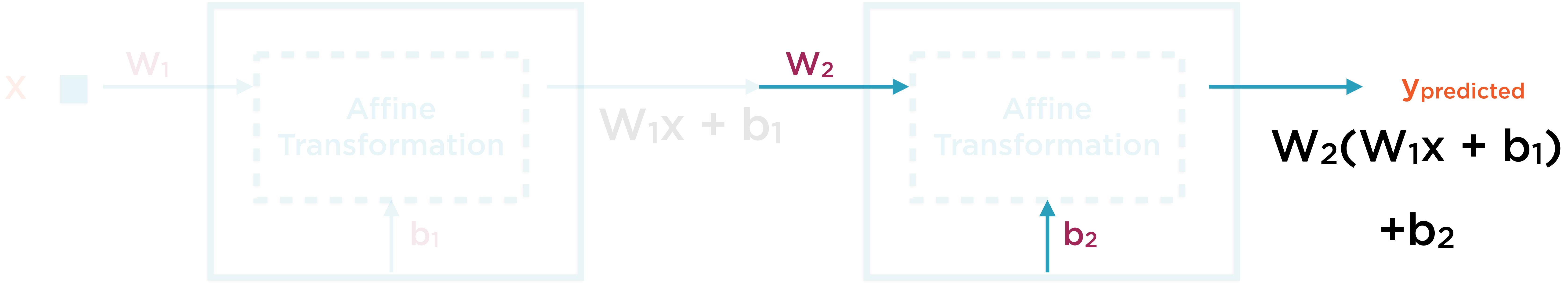
Simple Example: Two Neurons, Linear Network



Output of second neuron = W_2 x Output of first neuron

$+ b_2$

Simple Example: Two Neurons, Linear Network



Output of second neuron = $W_2(W_1x + b_1) + b_2$

$y_{predicted} = W_2(W_1x + b_1) + b_2$

MSE = Mean Square Error of Loss

$\text{Loss} = \theta = y_{predicted} - y_{actual}$

$y_{predicted} = W_2(W_1x + b_1) + b_2$

Loss Function θ

Loss function measures inaccuracy of model on a specific instance

$$\theta = y_{predicted} - y_{actual} = W_2(W_1x + b_1) + b_2 - y_{actual}$$

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1, W_2, b_2)$$
$$= (\partial\theta/\partial W_1, \partial\theta/\partial b_1, \partial\theta/\partial W_2, \partial\theta/\partial b_2)$$

Gradient: Vector of Partial Derivatives

For a function $y = f(x_1, x_2, x_3)$, the Greek character “nabla” (∇) denotes the gradient

$$\theta = y_{predicted} - y_{actual} = W_2(W_1x + b_1) + b_2 - y_{actual}$$

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1, W_2, b_2)$$

$$= (\partial\theta/\partial W_1, \partial\theta/\partial b_1, \partial\theta/\partial W_2, \partial\theta/\partial b_2)$$

$$\partial\theta/\partial W_1 = W_2x$$

Differentiate θ with respect to W_1, assuming all other values are constant

$$\theta = y_{predicted} - y_{actual} = W_2(W_1x + b_1) + b_2 - y_{actual}$$

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1, W_2, b_2)$$
$$= (\partial\theta/\partial W_1, \partial\theta/\partial b_1, \partial\theta/\partial W_2, \partial\theta/\partial b_2)$$

$$\partial\theta/\partial W_2 = W_1x + b_1$$

Differentiate θ with respect to W_2, assuming all other values are constant

$$\theta = y_{predicted} - y_{actual} = W_2(W_1x + b_1) + b_2 - y_{actual}$$

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1, W_2, b_2)$$
$$= (\partial\theta/\partial W_1, \partial\theta/\partial b_1, \partial\theta/\partial W_2, \partial\theta/\partial b_2)$$

$$\partial\theta/\partial b_1 = W_2$$

Differentiate θ with respect to b_1, assuming all other values are constant

$$\theta = y_{predicted} - y_{actual} = W_2(W_1x + b_1) + b_2 - y_{actual}$$

$$\text{Gradient}(\theta) = \nabla\theta(W_1, b_1, W_2, b_2)$$
$$= (\partial\theta/\partial W_1, \partial\theta/\partial b_1, \partial\theta/\partial W_2, \partial\theta/\partial b_2)$$

$$\partial\theta/\partial b_2 = 1$$

Differentiate θ with respect to b_2, assuming all other values are constant

$$
\begin{aligned}
\text{Gradient}(\theta) &= \nabla\theta(W_1,\ b_1,\ W_2,\ b_2) \\
&= (\partial\theta/\partial W_1,\ \partial\theta/\partial b_1,\ \partial\theta/\partial W_2,\ \partial\theta/\partial b_2) \\
&= (W_2 x,\ W_2,\ W_1 x + b_1,\ 1)
\end{aligned}
$$

Gradient: Vector of Partial Derivatives

Have calculated the gradient using high-school calculus - “symbolic differentiation”

$$\text{Gradient}(\theta^t) = (W_2^t x,\ W_2^t,\ W_1^t x + b_1^t,\ 1)$$

Gradient: Vector of Partial Derivatives

These gradients apply to a specific time t

$$\text{Gradient}(\theta^t) = (W_2^t x,\ W_2^t,\ W_1^t x + b_1^t,\ 1)$$

Gradient: Vector of Partial Derivatives

These gradients apply to a specific time t

$$\text{Parameters}^{t+1} = \text{Parameters}^{t} - \text{learning\_rate} \times \text{Gradient}(\theta)^{t}$$

For Next Time Step: Update Parameter Values

Move each parameter value in the direction of reducing gradient

Exact math and mechanics are complex and will vary by optimization algorithm

$$\text{Parameters}^{t+1} = \text{Parameters}^{t} - \text{learning\_rate} \times \text{Gradient}(\theta)^{t}$$

For Next Time Step: Update Parameter Values

Move each parameter value in the direction of reducing gradient

$$\text{Parameters}^{t+1} = \text{Parameters}^{t} - \text{learning\_rate} \times \text{Gradient}(\theta)^{t}$$

For Next Time Step: Update Parameter Values

Move each parameter value in the direction of reducing gradient

$$Parameters^{t+1} = Parameters^{t} - learning\_rate \times Gradient(\theta)^{t}$$

For Next Time Step: Update Parameter Values

Move each parameter value in the direction of reducing gradient

Calculated in backward pass
of time t

$$\text{Parameters}^{t+1} = \text{Parameters}^{t} - \text{learning\_rate} \times \text{Gradient}(\theta)^{t}$$

For Next Time Step: Update Parameter Values

Move each parameter value in the direction of reducing gradient

Updated in backward pass
of time t...

$$\text{Parameters}^{t+1} = \text{Parameters}^{t} - \text{learning\_rate} \times \text{Gradient}(\theta)^{t}$$

For Next Time Step: Update Parameter Values

Move each parameter value in the direction of reducing gradient

...then used in forward pass
of time t+1

$$\text{Parameters}^{t+1} = \text{Parameters}^{t} - \text{learning\_rate} \times \text{Gradient}(\theta)^{t}$$

For Next Time Step: Update Parameter Values

Move each parameter value in the direction of reducing gradient

Why Two Passes?

Symbolic Differentiation

Conceptually simple but hard to implement

Numeric Differentiation

Easy to implement but won’t scale

Automatic Differentiation

Conceptually difficult but easy to implement

Because reverse auto-differentiation needs two passes

Automatic Differentiation

Conceptually difficult but easy to implement

Reverse-mode auto-differentiation

Used in TF, PyTorch

Two passes in each training step

- Forward step: Calculate loss
- Backward step: Update parameter values

Actually Calculating Gradients

Symbolic Differentiation

Conceptually simple but hard to implement

Numeric Differentiation

Easy to implement but won't scale

Automatic Differentiation

Conceptually difficult but easy to implement

PyTorch, TensorFlow and other packages rely on automatic differentiation

Actually Calculating Gradients

Symbolic Differentiation

Conceptually simple but hard to implement

Numeric Differentiation

Easy to implement but won't scale

Automatic Differentiation

Conceptually difficult but easy to implement

Symbolic Differentiation

Conceptually simple but hard to implement

Actually calculate each element of gradient vector

(Approach adopted in example above)

Easy to understand, hard to implement

- complex neural networks
- Some activation functions are hard to differentiate
- Output function may be non-differentiable

Actually Calculating Gradients

Symbolic Differentiation

Conceptually simple but hard to implement

Numeric Differentiation

Easy to implement but won't scale

Automatic Differentiation

Conceptually difficult but easy to implement

Numeric Differentiation

Easy to implement but won't scale

Trivial to implement

$y = f(x)$

Add small "perturbation" ∂x to x

$y + \partial y = f(x + \partial x)$

$\partial y/\partial x = [f(x + \partial x) - f(x)]/\partial x$

Numeric Differentiation

Easy to implement but won't scale

Problem: need to do for each parameter

Will not scale to complex networks

(May have thousands of parameters)

Actually Calculating Gradients

Symbolic Differentiation

Conceptually simple but hard to implement

Numeric Differentiation

Easy to implement but won’t scale

Automatic Differentiation

Conceptually difficult but easy to implement

Automatic Differentiation

Conceptually difficult but easy to implement

Relies on a mathematical trick

Based on Taylor's Series Expansion

Allow fast approximation of gradients

Automatic Differentiation

Conceptually difficult but easy to implement

Add a dual number to each parameter at each step

The dual number is very very small…

…but not negligible

By Taylor series, get gradient value in single operation

Automatic Differentiation

Conceptually difficult but easy to implement

Two flavors

- Forward-mode
- Reverse-mode

Automatic Differentiation

Conceptually difficult but easy to implement

Forward-mode ~ numeric differentiation

Suffers from same flaw…

…Requires one pass per parameter

Will not scale to complex networks

Automatic Differentiation

Conceptually difficult but easy to implement

Reverse-mode used in TF, PyTorch…

Two passes in each training step

- Forward step: Calculate loss
- Backward step: Update parameter values

Automatic Differentiation

Conceptually difficult but easy to implement

Reverse auto-differentiation used everywhere

- TensorFlow: autodiff
- PyTorch: autograd

Back propagation is only required during training: to do so in PyTorch, invoke the .backward() method

Demo

Linear model to calculate gradients using autograd

Update model weights in the backward pass

PyTorch NN Library

PyTorch NN Library

Several types of layers

- Convolutional
- Recurrent
- Normalization
- Padding
- Pooling
- Linear
- Dropout
- Vision

PyTorch NN Library

Several types of activation functions

- ReLU
- Sigmoid
- Tanh
- ...

PyTorch NN Library

Parameters

Containers

Loss functions

Distance functions

PyTorch NN Library

DataParallel layers for distributed multi-GPU support

Easier (currently) to leverage GPUs in PyTorch than in TensorFlow

Demo

Build a fully connected sequential model for automobile price prediction

PyTorch Optimizers

Linear Regression as an Optimization Problem

Objective Function

Minimize variance of the residuals (MSE)

Linear Regression as an Optimization Problem

Objective Function

Minimize variance of the residuals (MSE)

Constraints

Express relationship as a straight line

$$y = Wx + b$$

Linear Regression as an Optimization Problem

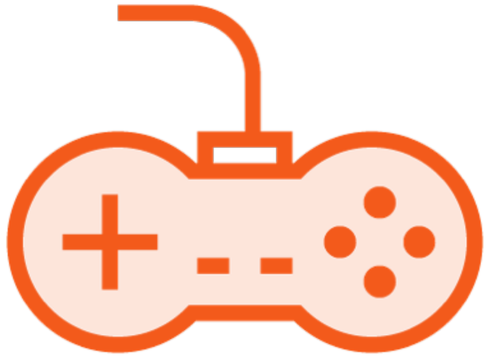
Objective Function

Minimize variance of the residuals (MSE)

Constraints

Express relationship as a straight line

$y = Wx + b$

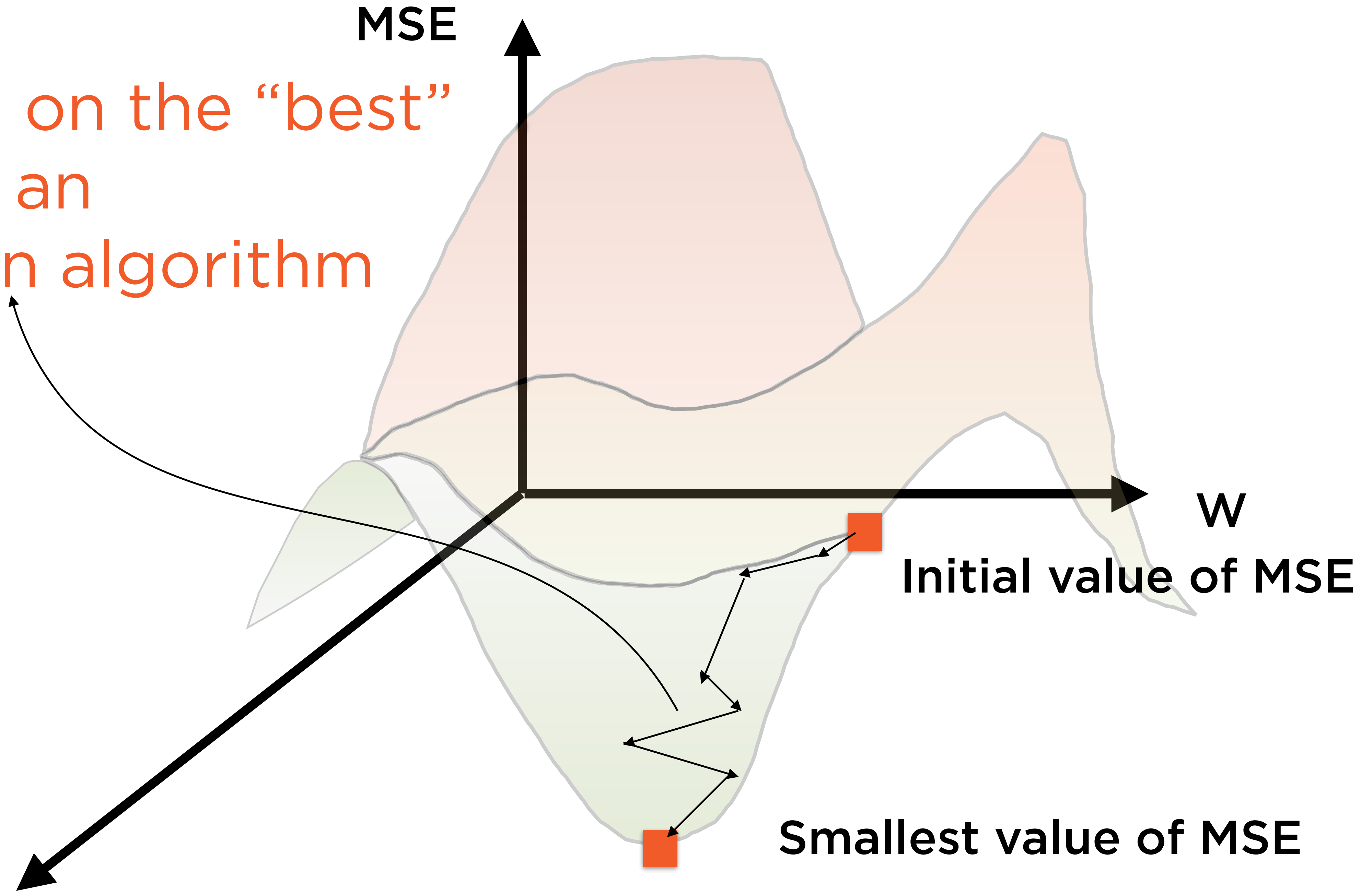


Decision Variables

Values of W and b

“Gradient Descent”

Converging on the “best” value using an optimization algorithm



Using an Optimizer in PyTorch

Construct Optimizer Object

Pass iterable of all parameters

Each parameter should be a Variable

Specify Per-parameter Options

Pass iterable of dict objects

Each key a param, defines parameter group

Compute Gradients

Invoke .backward()

Autograd for reverse auto-differentiation

Take an Optimization Step

Invoke optimizer.step()

Overloaded version takes in a closure (advanced)

torch.optim

torch.optim.Optimizer

torch.optim.Adadelta

torch.optim.Adagrad

torch.optim.Adam

...

$$\text{Parameters}^{t+1} = \text{Parameters}^{t} - \text{learning\_rate} \times \text{Gradient}(\theta)^{t}$$

Basic SGD Optimizer

Move each parameter value in the direction of reducing gradient

$$\text{momentum\_vec}^{t} = \text{momentum\_coeff} + \text{learning\_rate} \times \text{Gradient}(\theta)^{t}$$

$$\text{Parameters}^{t+1} = \text{Parameters}^{t} - \text{momentum\_vec}$$

Momentum-based Optimizer

Momentum vector helps accelerate in the direction where gradient is decreasing

$$\text{momentum\_vec}^{t} = \text{momentum\_coeff} + \text{learning\_rate} \times \text{Gradient}(\theta)^{t}$$

$$\text{Parameters}^{t+1} = \text{Parameters}^{t} - \text{momentum\_vec}$$

Momentum-based Optimizer

Gradients at each step weighted by those in previous step

Benefit: Faster convergence

$$momentum\_vec_{t} = momentum\_coeff + learning\_rate \times Gradient(\theta)_{t}$$

$$Parameters_{t+1} = Parameters_{t} - momentum\_vec$$

Momentum-based Optimizer

Need a momentum coefficient, between 0 and 1 to prevent overshooting

Advanced Optimizers

Many variants of optimizers

Increasing complexity

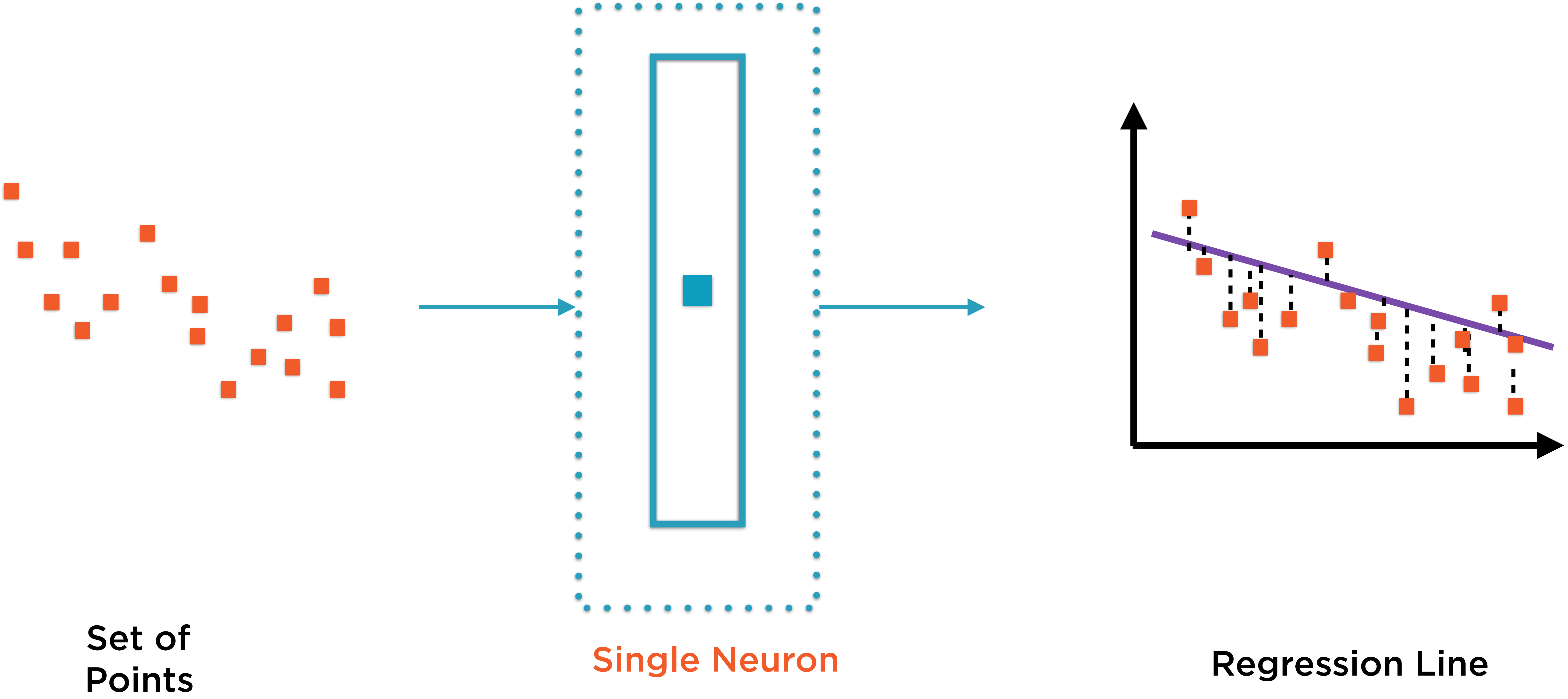
More hyper parameters

Better performance

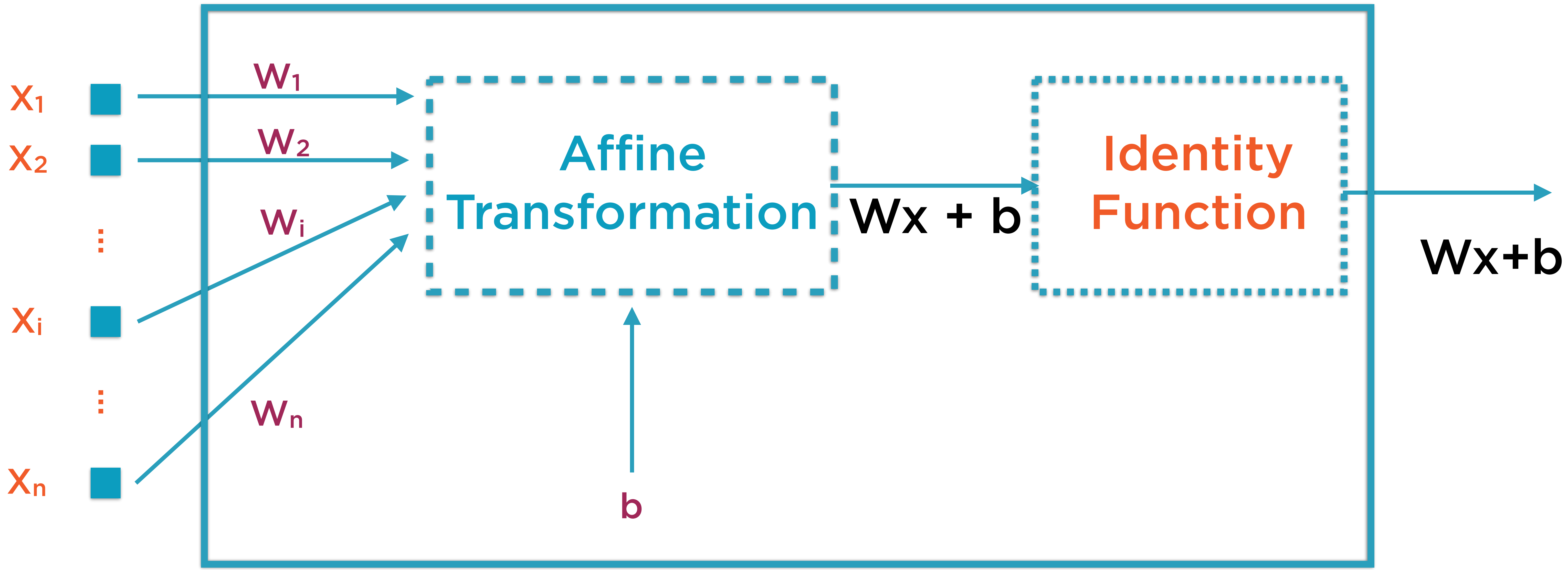
Adam ~ Adaptive Moment Estimation

Classification Using Neural Networks

Linear Regression with One Neuron



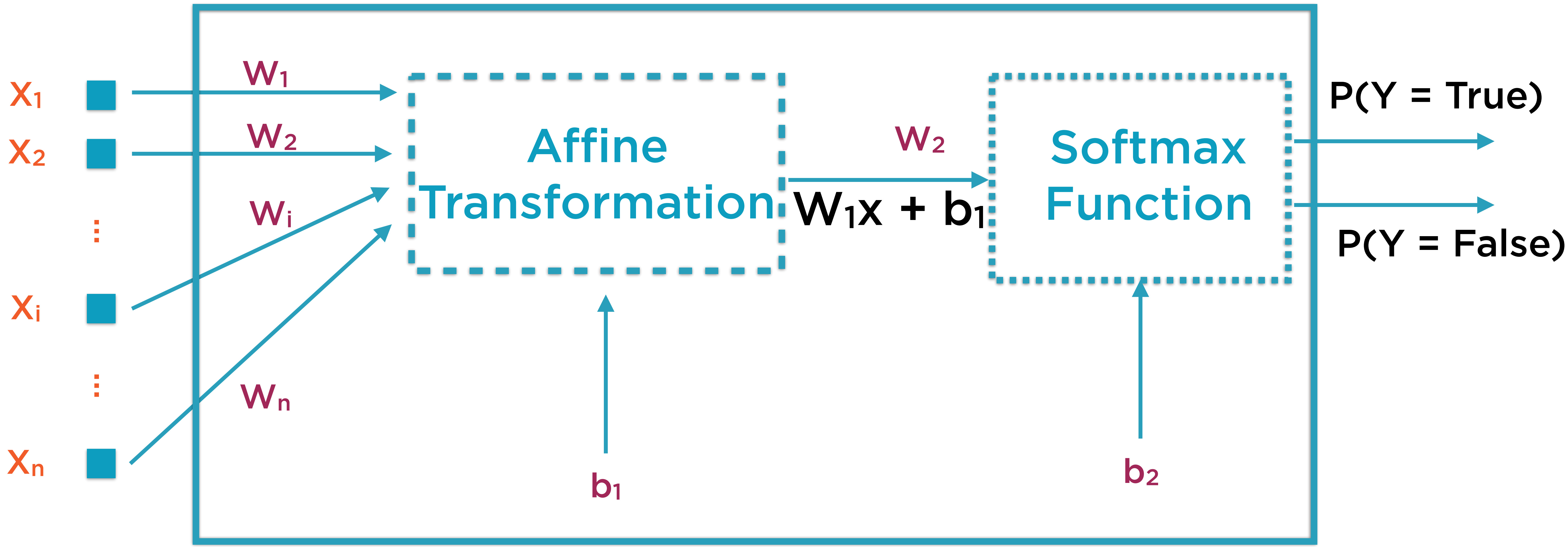
Linear Regression with One Neuron



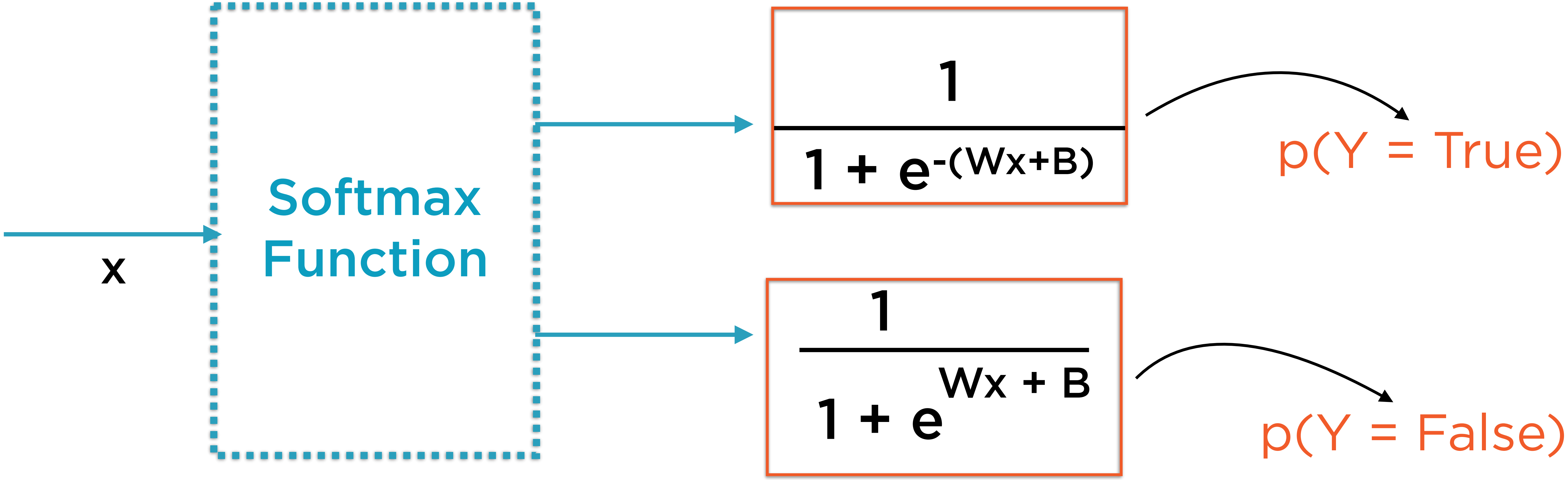
Linear Regression with One Neuron

Linear Classification with One Neuron

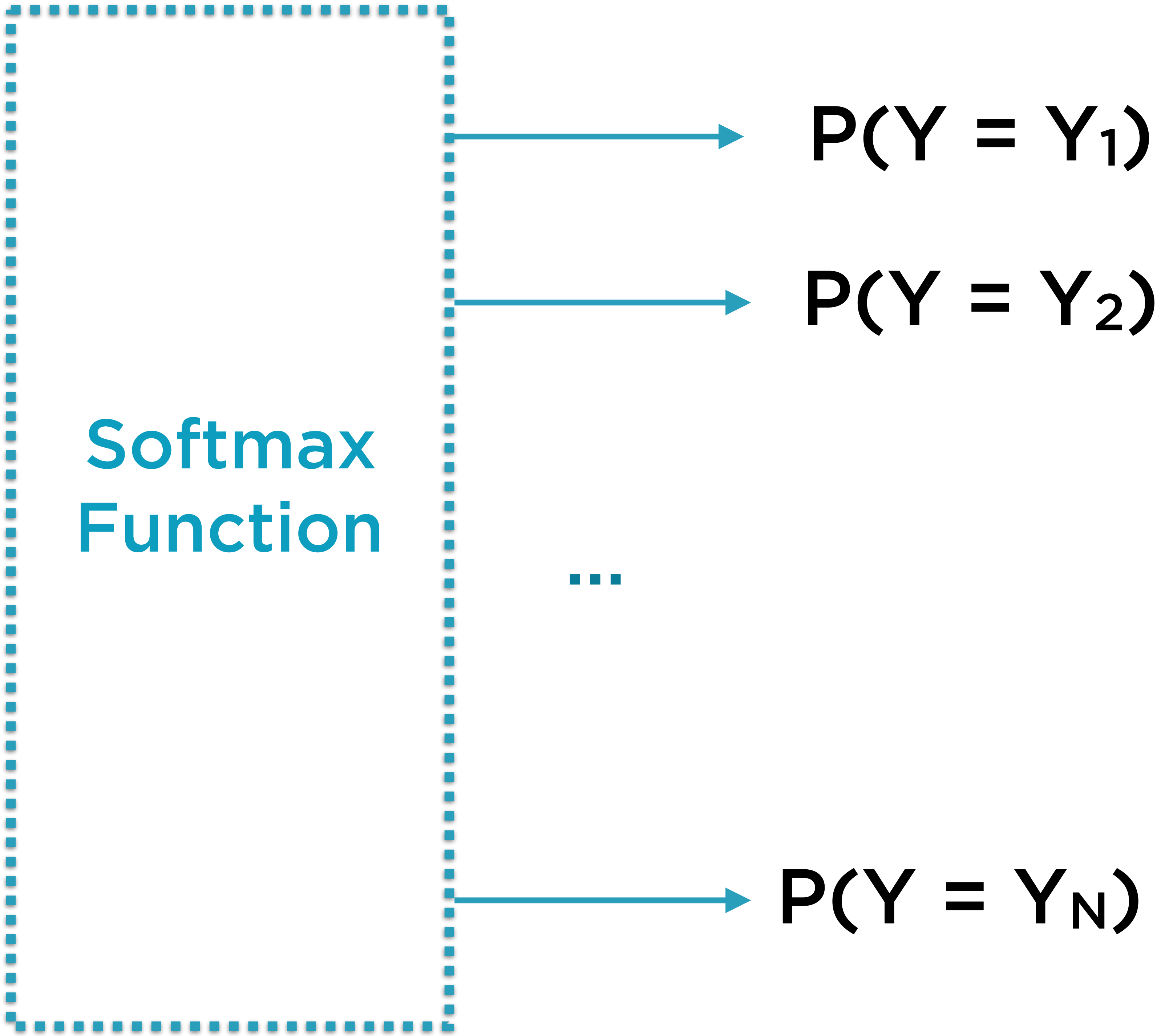
Linear Classification with One Neuron



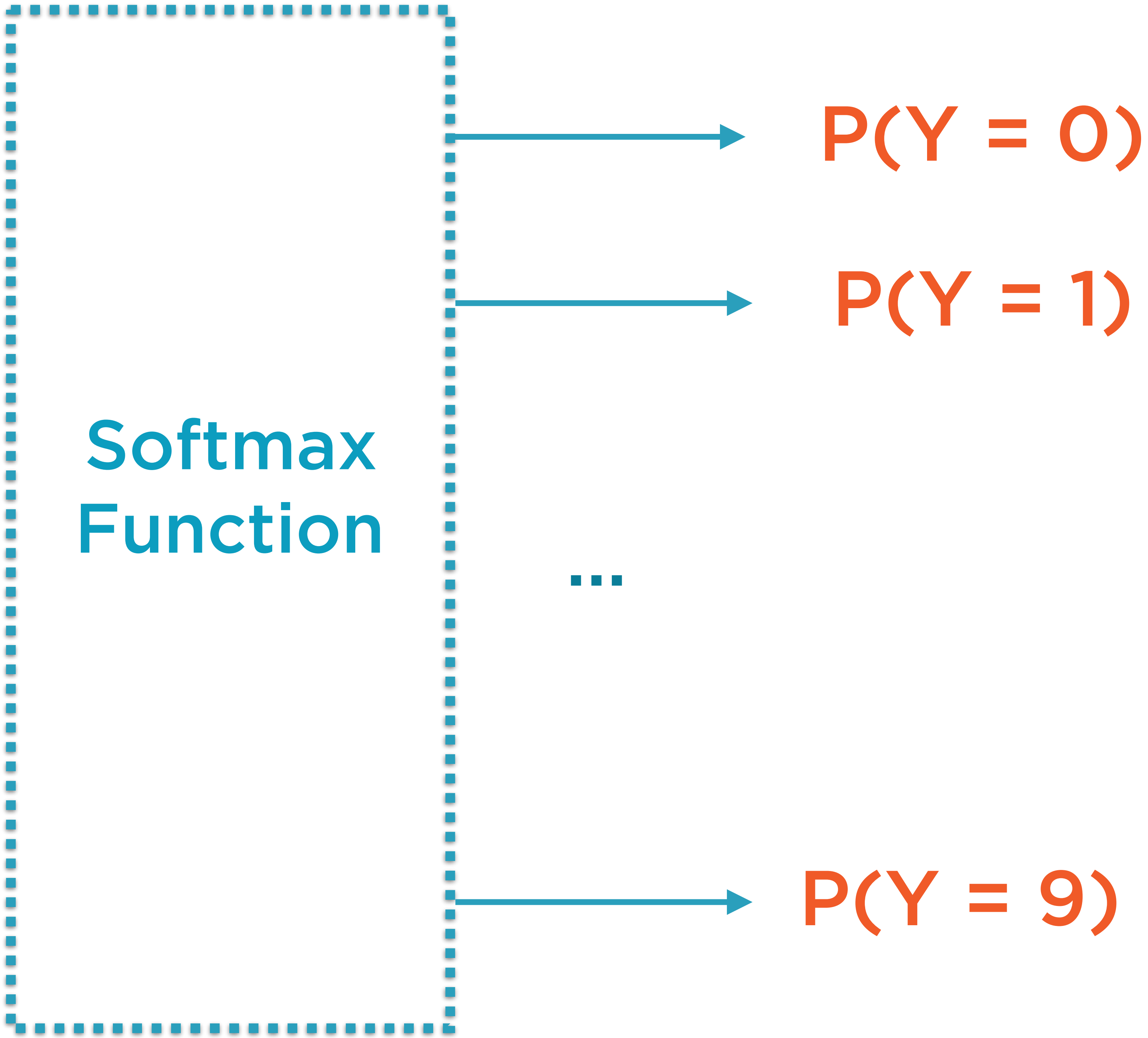
SoftMax for True/False Classification



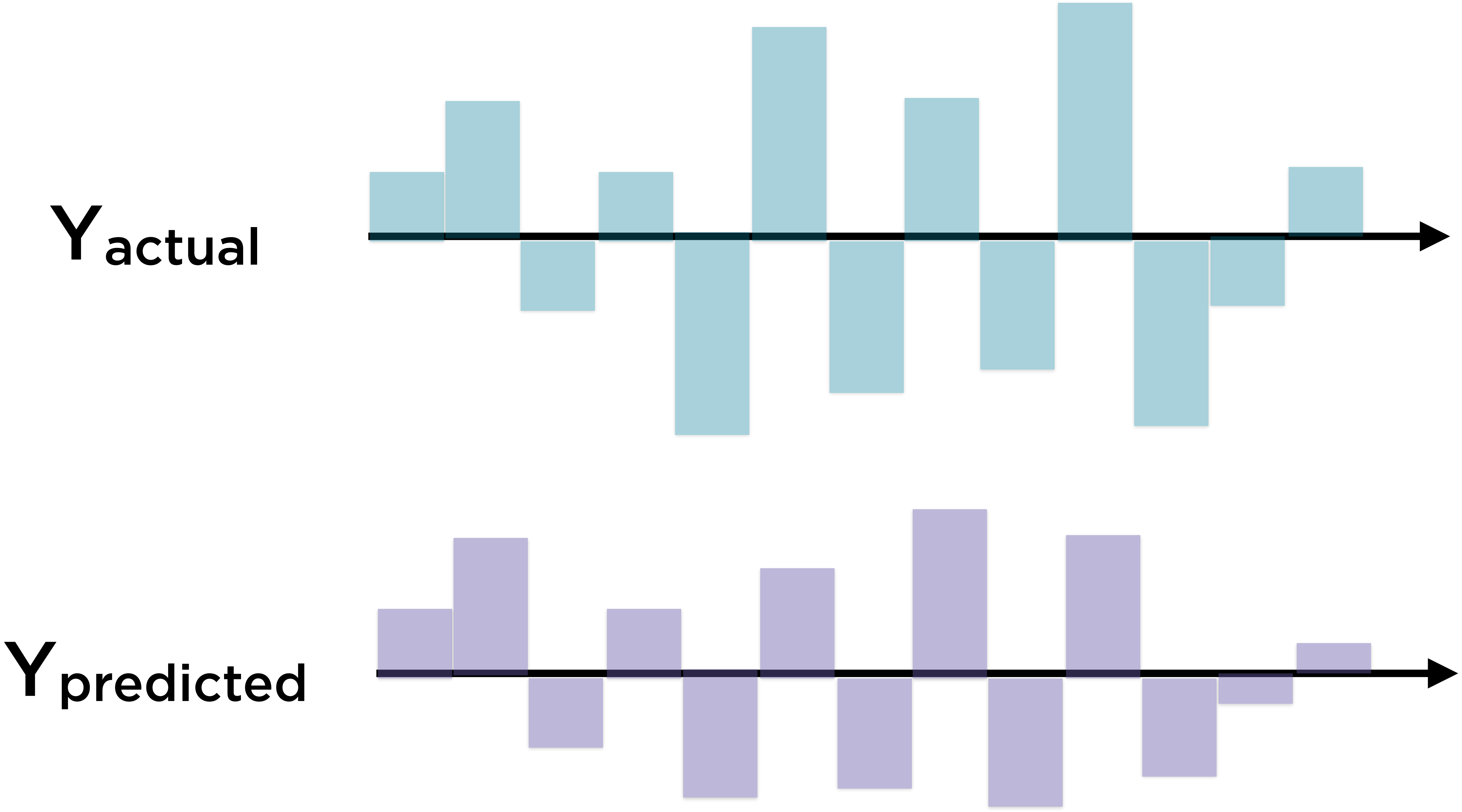
SoftMax N-category Classification



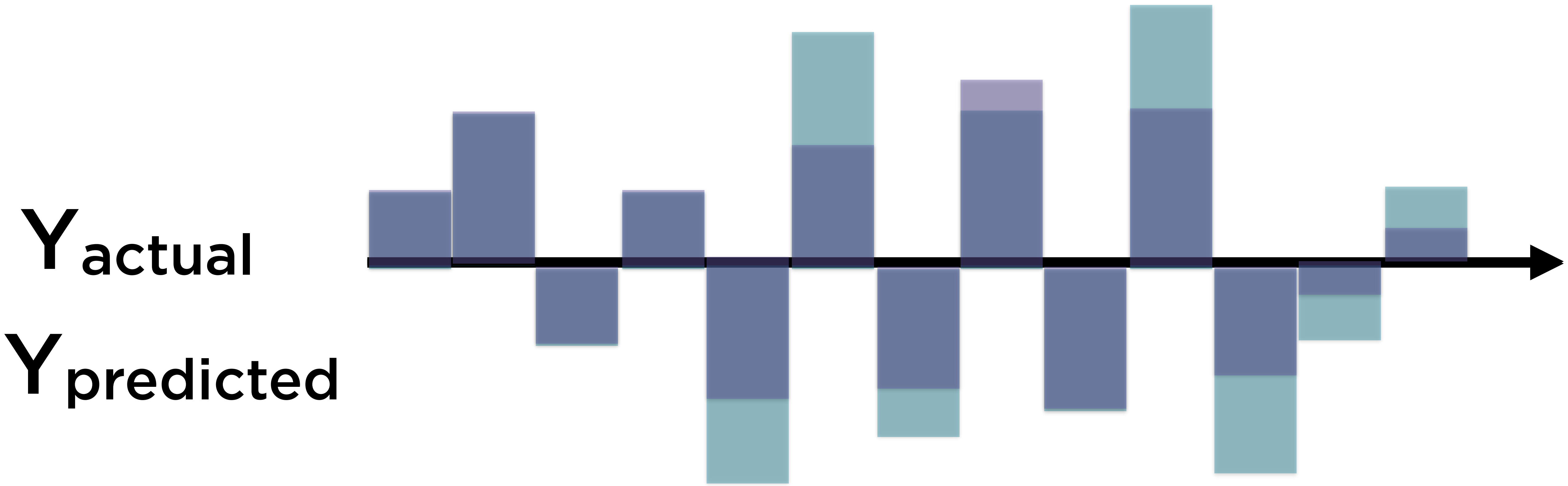
SoftMax for Digit Classification



Intuition: Low Cross Entropy

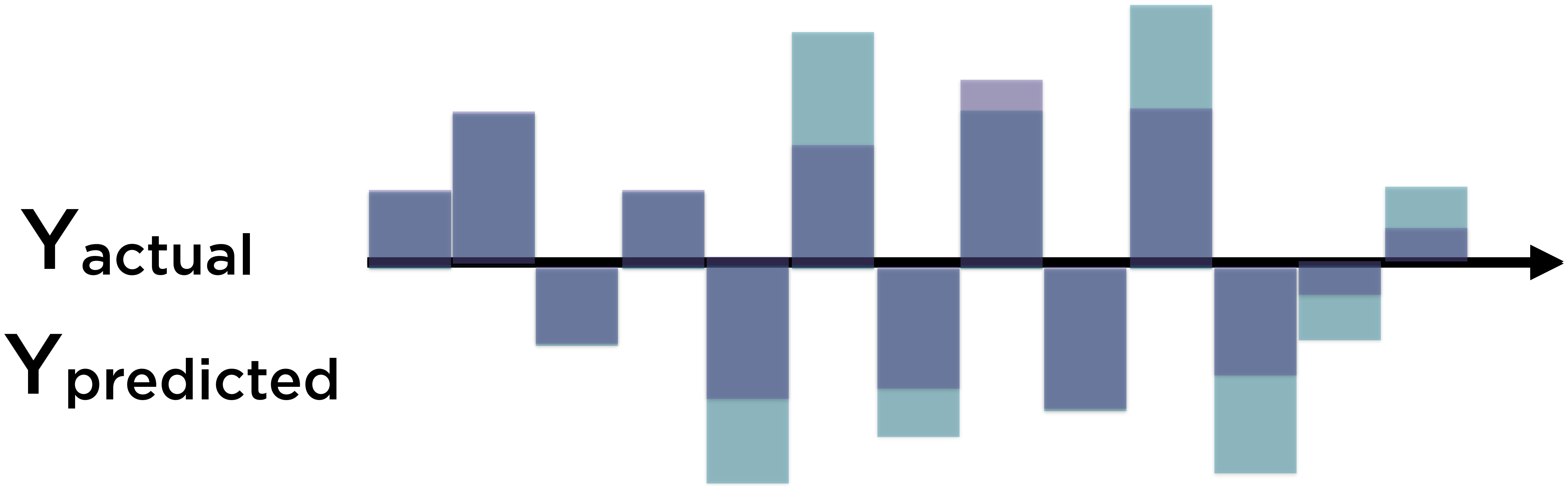


Intuition: Low Cross Entropy



The labels of the two series are in-synch

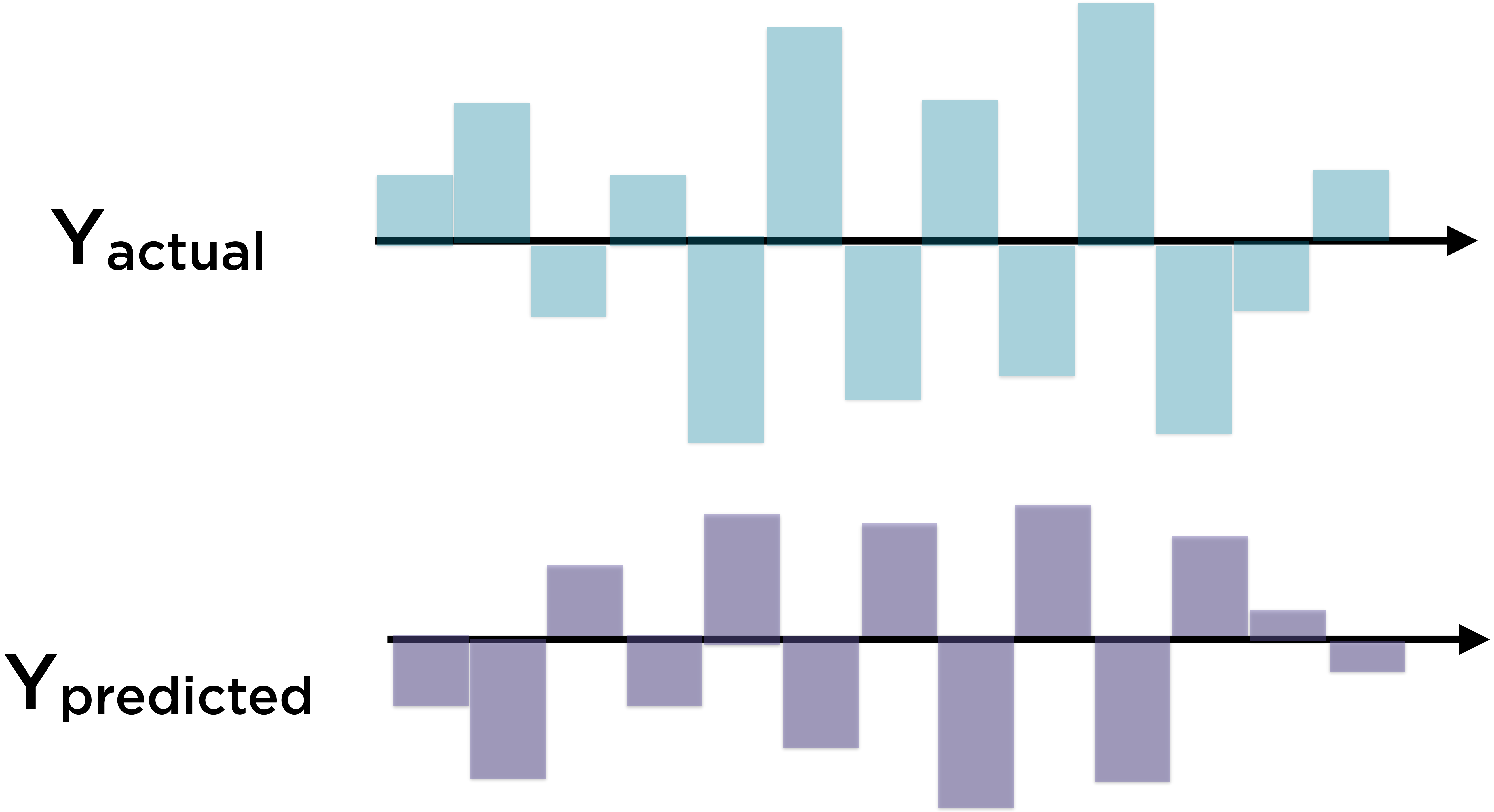
Intuition: Low Cross Entropy



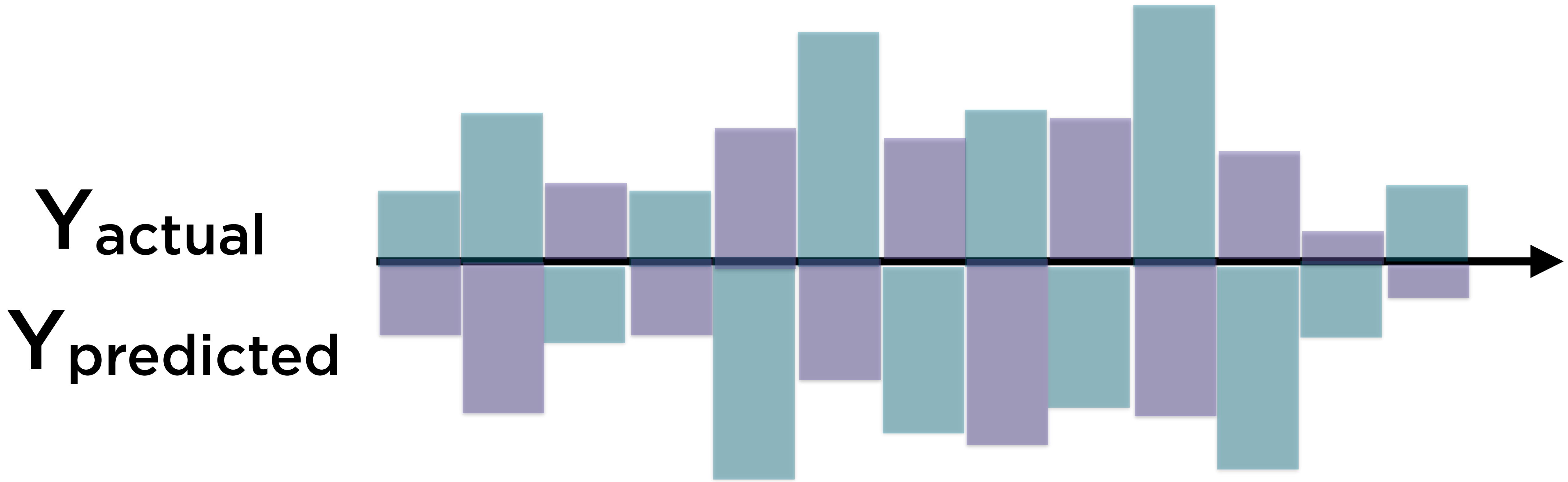
$\boxed{-Sum(\ Y_{actual} * \log [\ Y_{predicted}]\ )}$ will be small

Cross Entropy

Intuition: High Cross Entropy



Intuition: High Cross Entropy

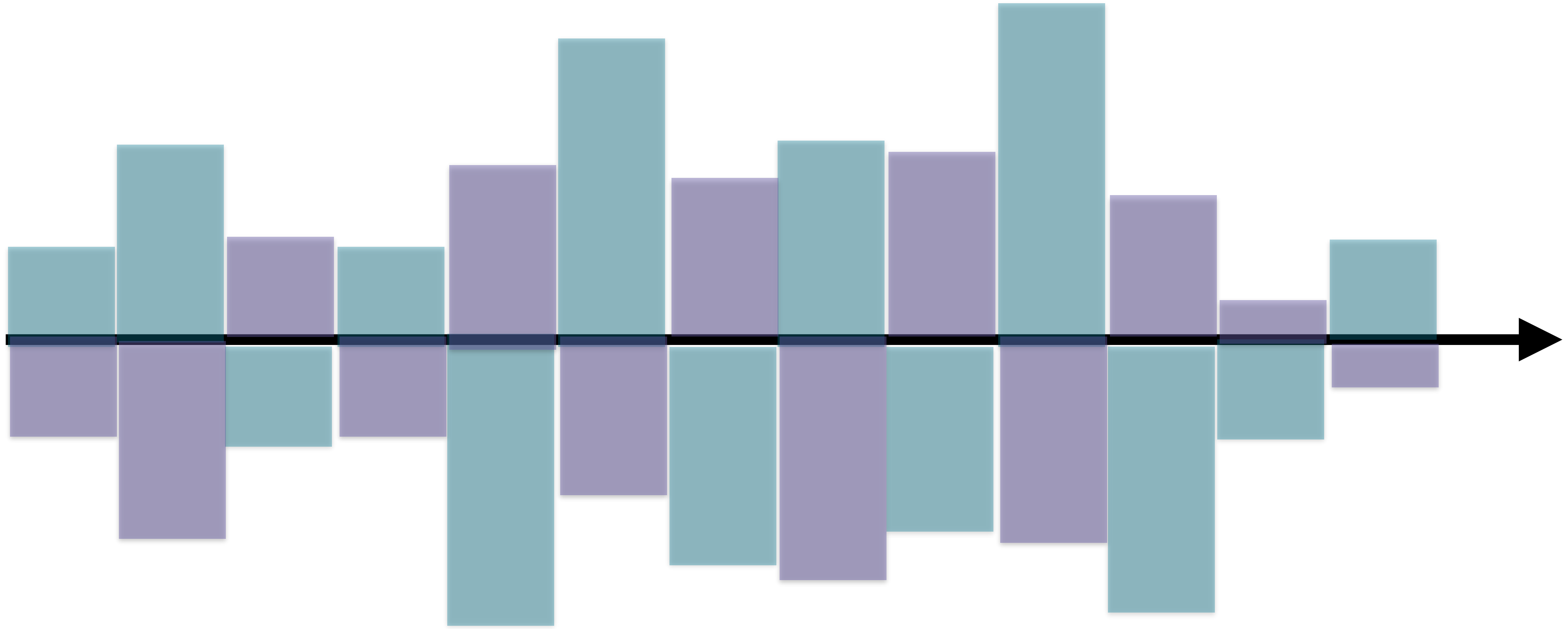


The labels of the two series are out-of-synch

Intuition: High Cross Entropy

Y_{actual}

$Y_{predicted}$



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

Cross Entropy

Linear Classification as an Optimization Problem

Linear Classification as an Optimization Problem

Objective Function

Minimize cross-entropy between Y_{actual} and $Y_{predicted}$

Linear Classification as an Optimization Problem

Objective Function

Minimize cross-entropy between Y_{actual} and $Y_{predicted}$

Constraints

Express relationship as an exponential one

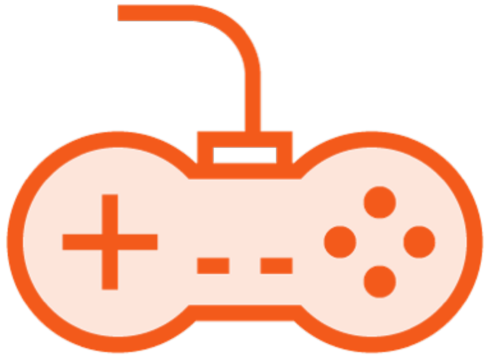
Linear Classification as an Optimization Problem

Objective Function

Minimize cross-entropy between Y_{actual} and $Y_{predicted}$

Constraints

Express relationship as an exponential one



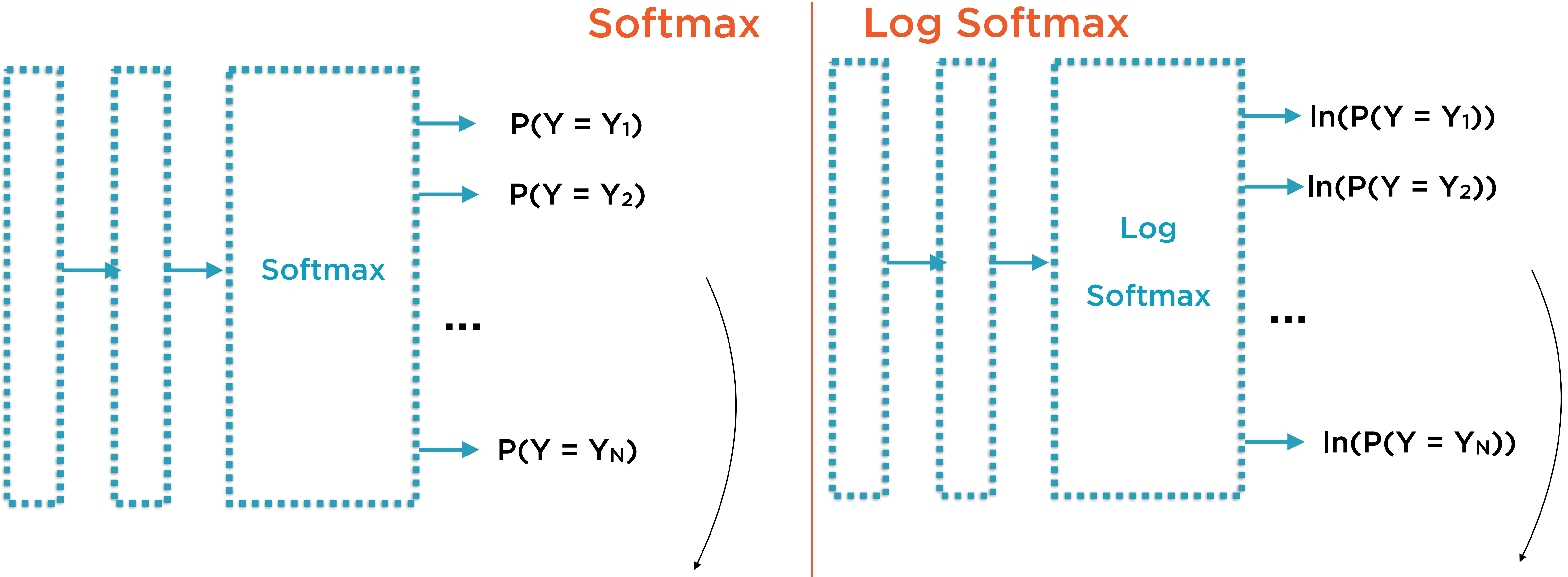
Decision Variables

Find “best” values for parameters

Softmax or Log Softmax as Output Layer?

Softmax	Log Softmax
Output layer of NN is Softmax	Output layer of NN is Log Softmax
Slightly less stable	Slightly more stable, nicer properties
Use cross-entropy as loss function	Use NLL (negative log likelihood) as loss
Objective is to minimize cross-entropy	Mathematically equivalent (almost)
Need just 1 output layer (Softmax)	Might need additional output layer (log), but in PyTorch just use LogSoftMax

Softmax or Log Softmax as Output?

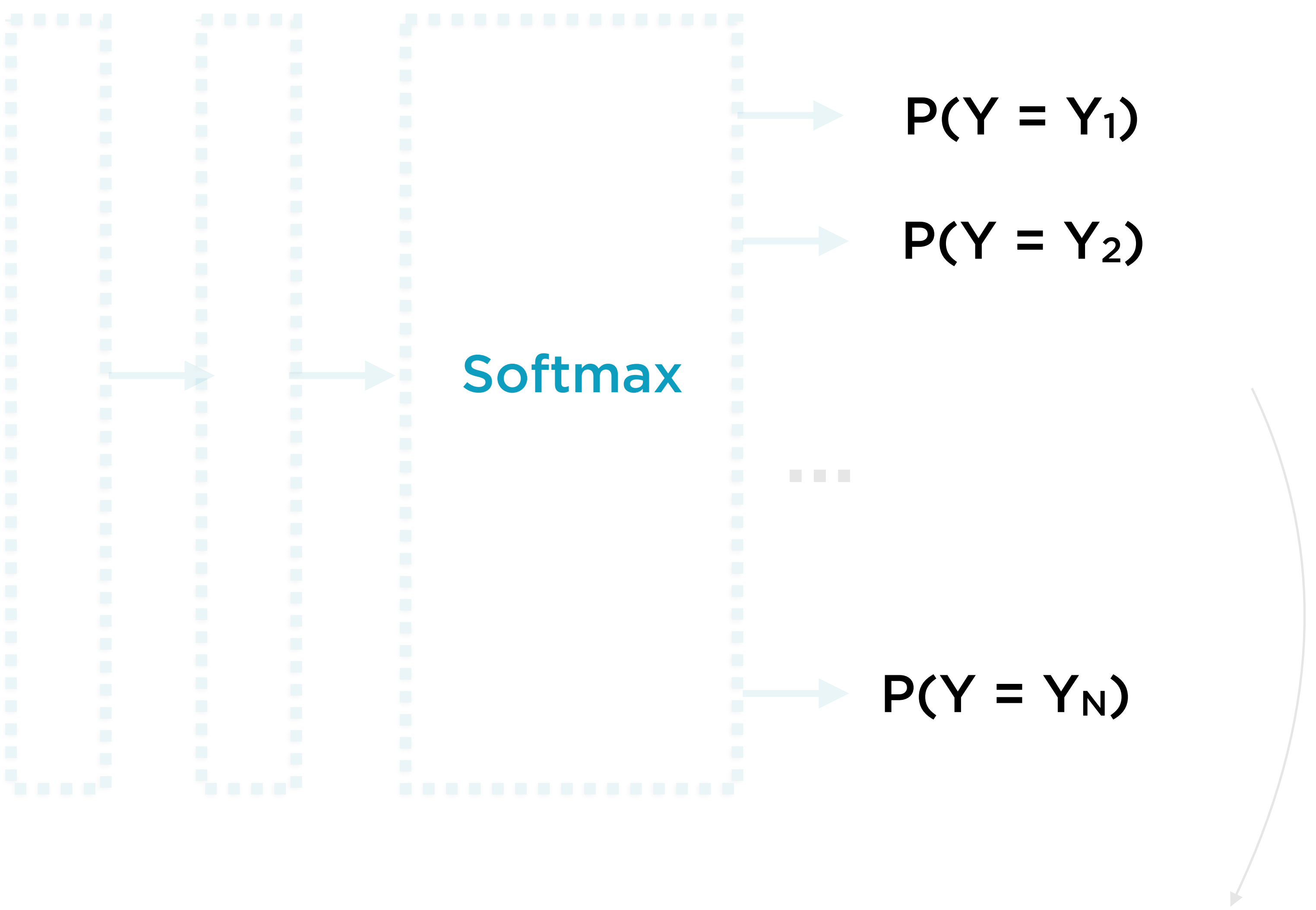


Minimize cross-entropy loss

Minimize NLL loss

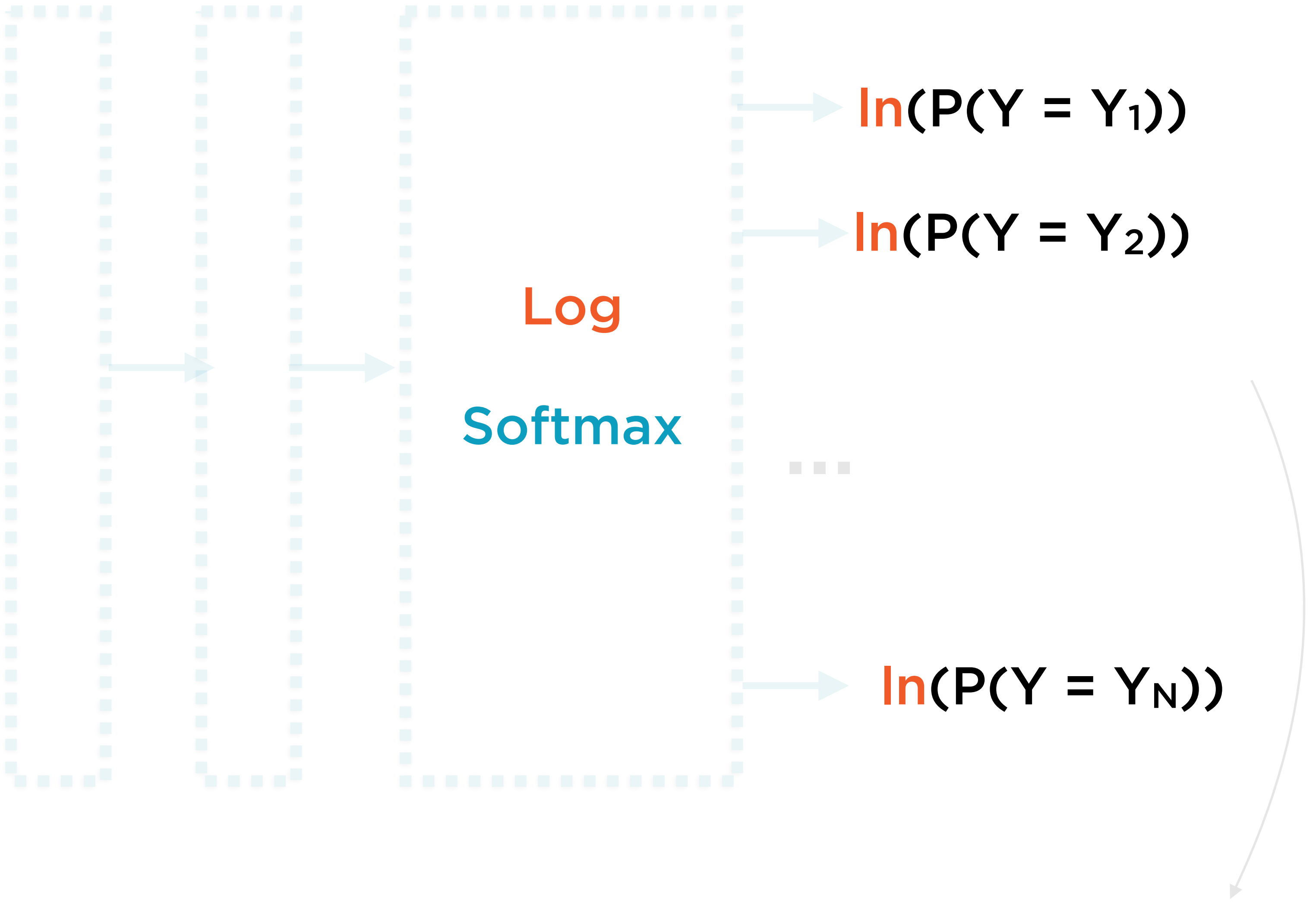
Softmax or Log Softmax as Output?

Softmax



Minimize cross-entropy loss

Log Softmax



Minimize NLL loss

Using (Output Layer = LogSoftmax and Loss = NLL) is equivalent* to using (Output Layer = Softmax and Loss = Cross-entropy)

Demo

Predicting survival probabilities on the Titanic

One-hot Encoding

Sunday

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

One-hot Encoding

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Monday	0	1	0	0	0	0	0
Thursday	0	0	0	0	1	0	0
Saturday	0	0	0	0	0	0	1

Summary

PyTorch uses the Autograd library for backpropagation during training

Autograd relies on reverse-mode automatic differentiation

Conceptually similar to autodiff in TensorFlow

Explore NN functionality for classification

NLL as loss function and LogSoftMax as output layer