Building a Regression Model Using PyTorch

Janani Ravi
CO-FOUNDER, LOONYCORN

www.loonycorn.com

Overview

Training a neural network using forward and backward passes

Using optimizers to update model parameters

Using layers and activation functions to train and build neural networks

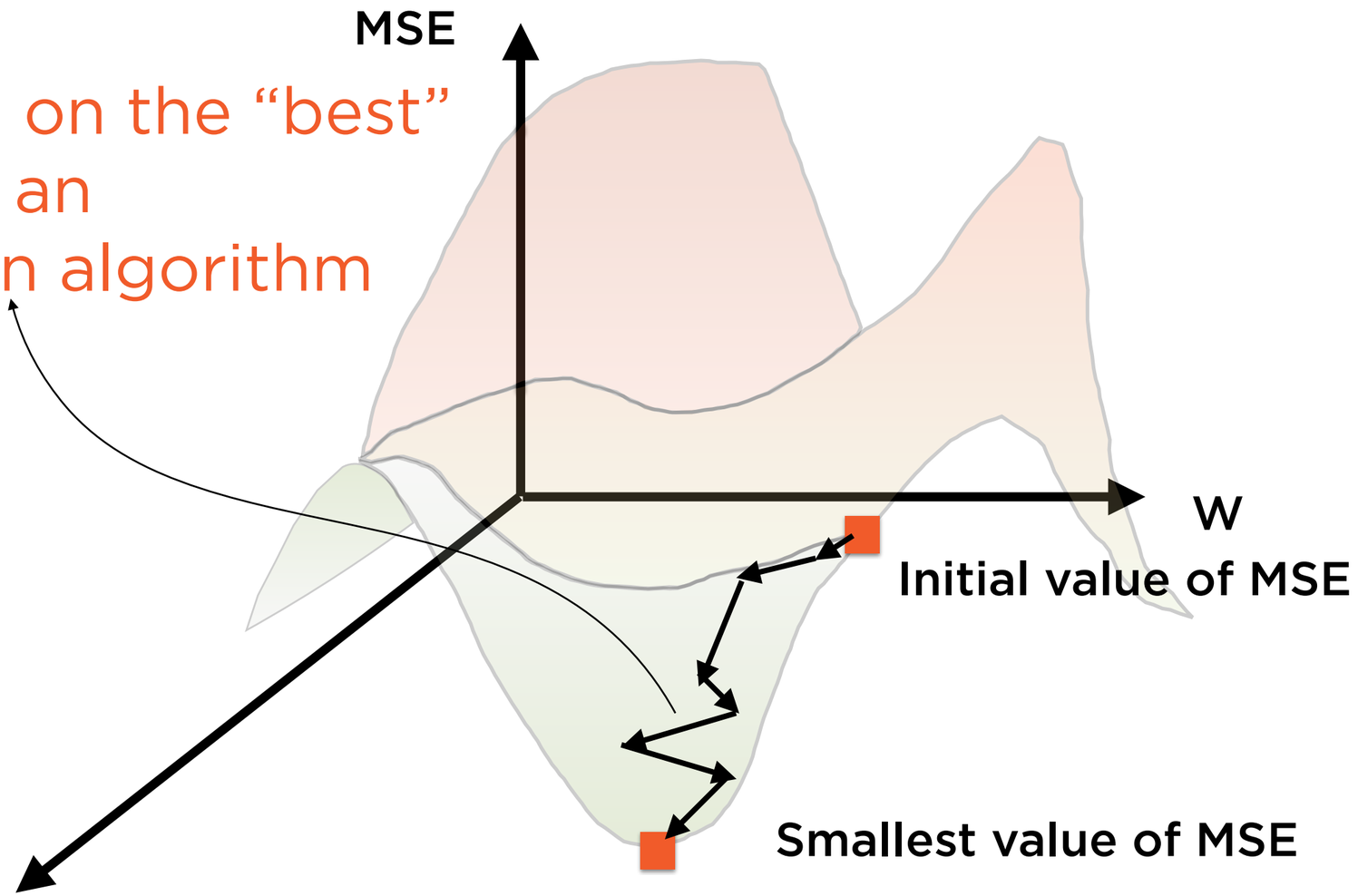
Understanding and using dropout to mitigate overfitting

Training A Neural Network

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

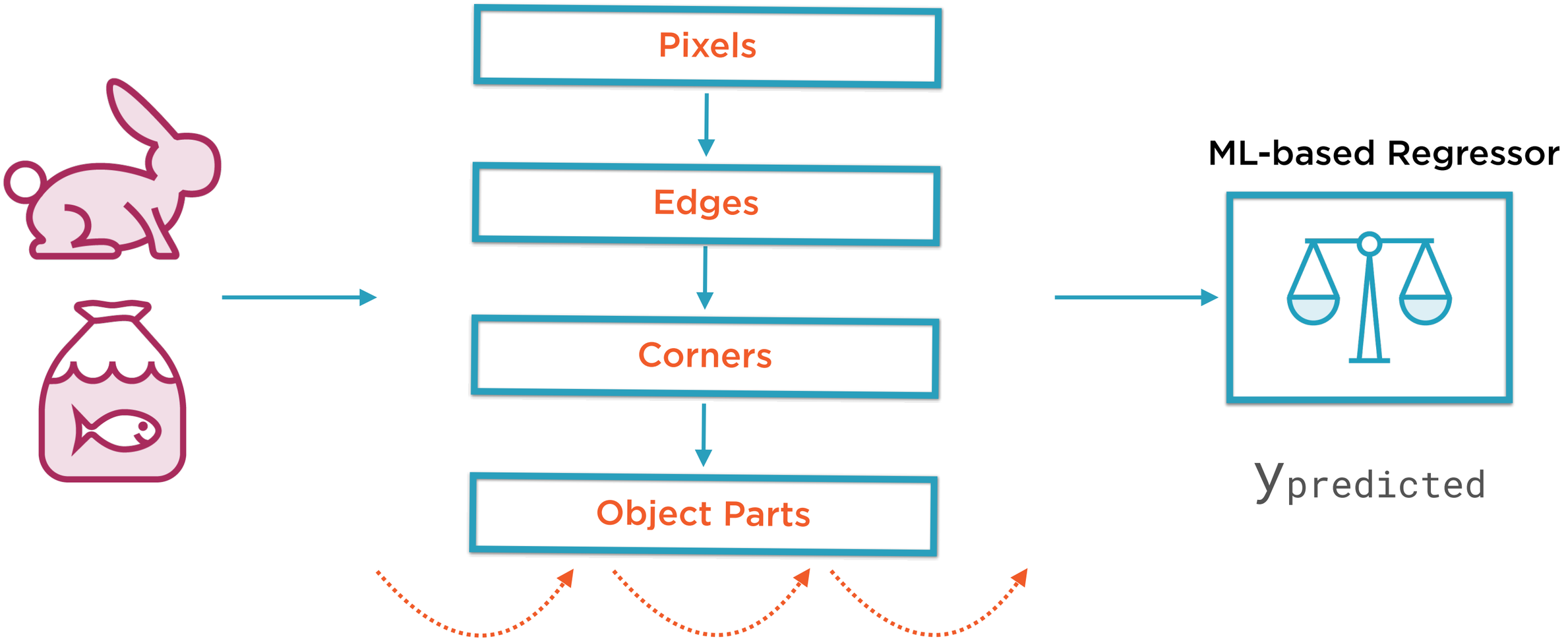
Gradient Descent

Converging on the “best” value using an optimization algorithm



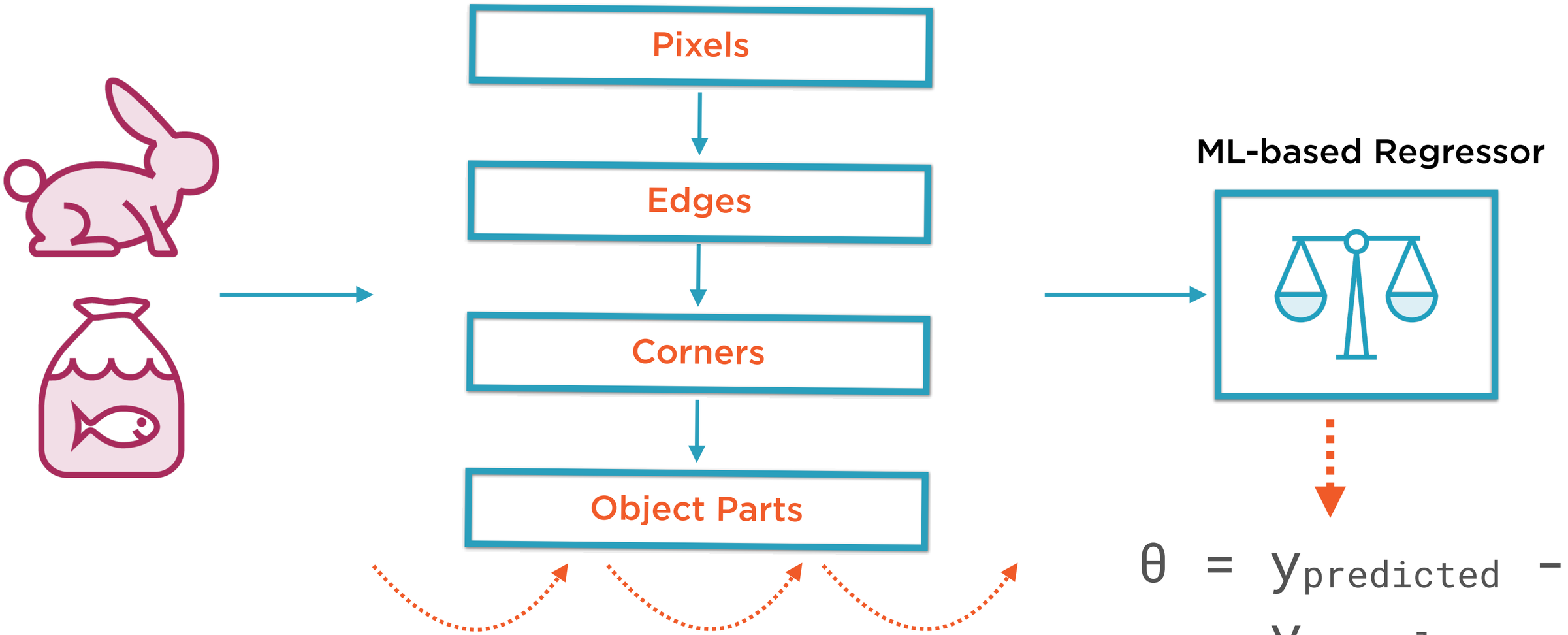
These gradients are used to update the model parameters

Forward Pass

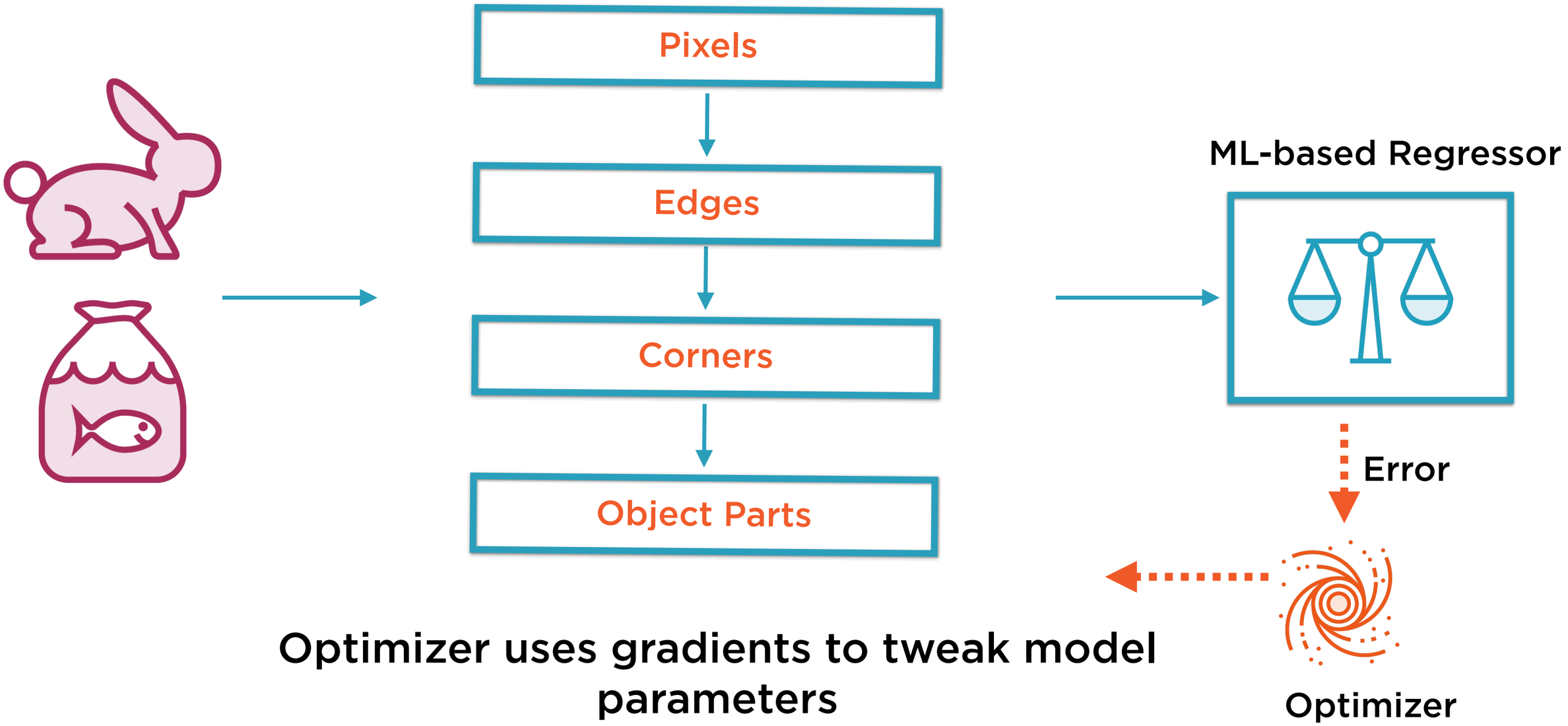


Use the current model weights and biases to make a prediction

Forward Pass

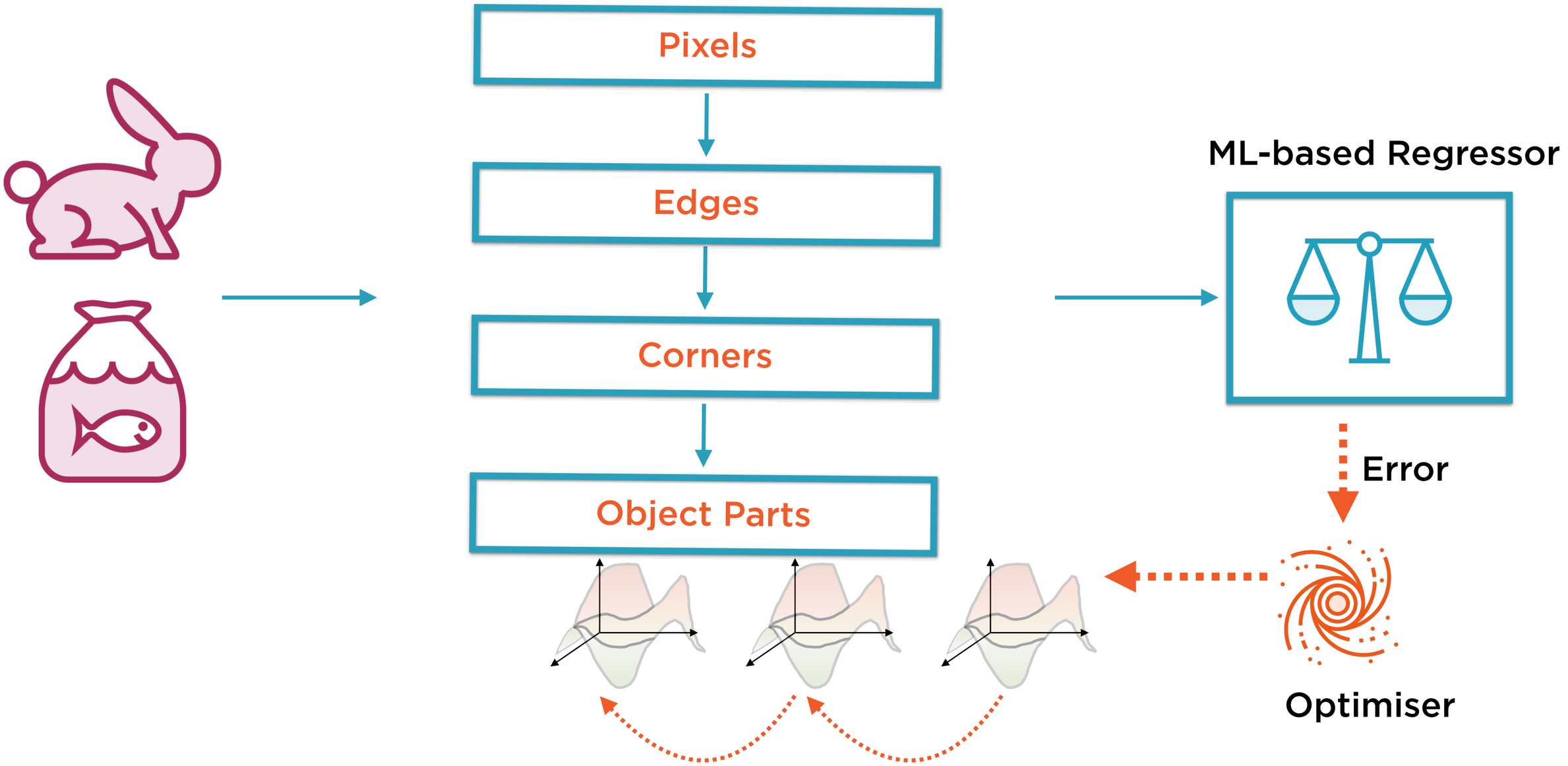


Optimizer Calculates Gradients



Optimizer uses gradients to tweak model parameters

Backward Pass



PyTorch Optimizers

Linear Regression as an Optimization Problem

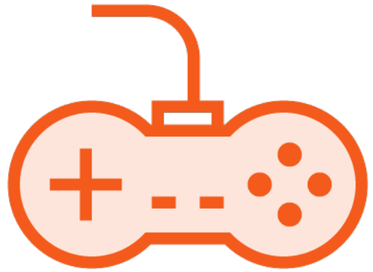
Objective Function

Minimize variance of the residuals (MSE)

Constraints

Express relationship as a straight line

$y = Wx + b$

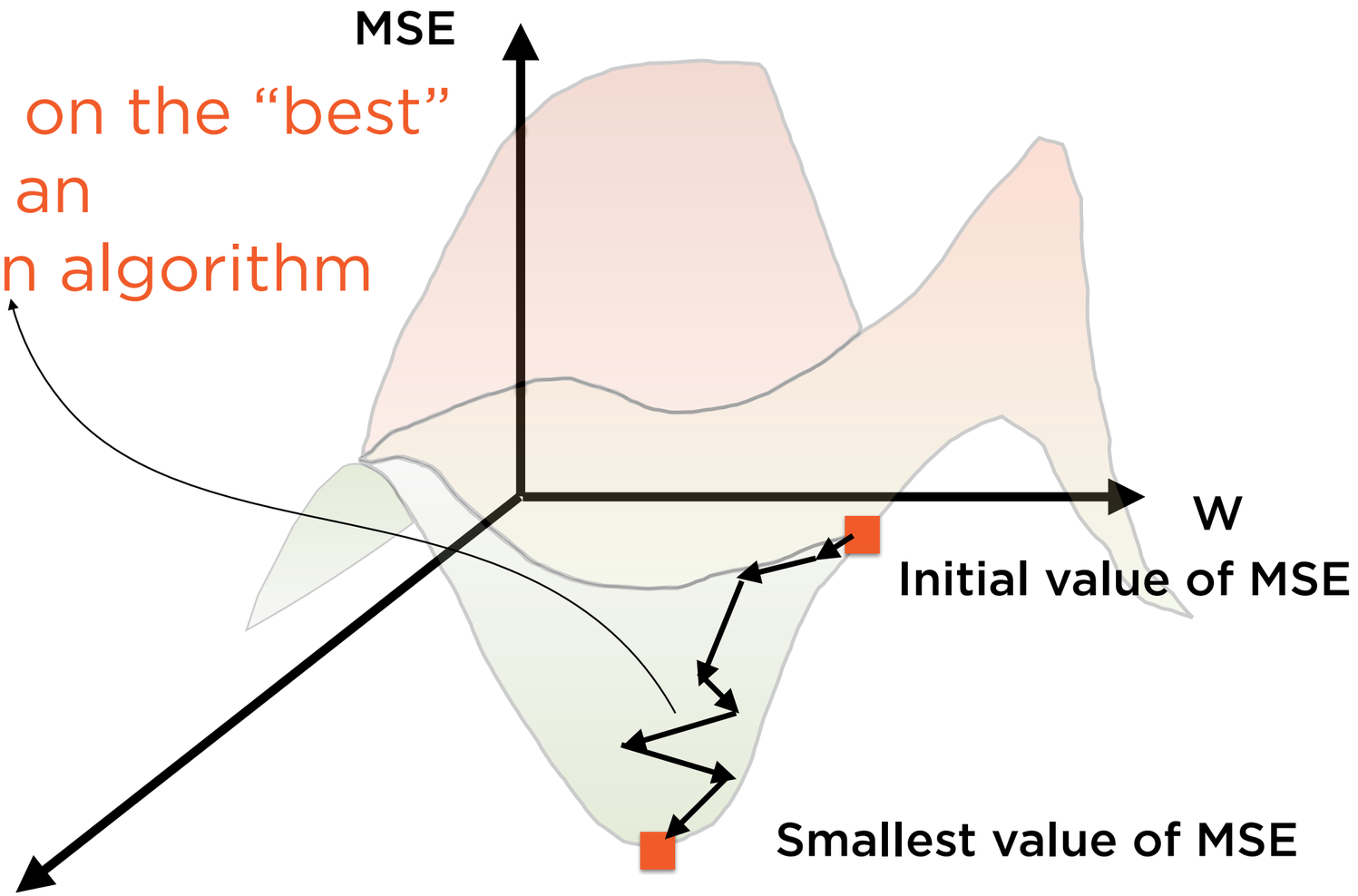


Decision Variables

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 learnable tensor

Compute Gradients

Invoke .backward()

Autograd for reverse auto-differentiation

Specify Per-parameter Options

Pass iterable of dict objects

Each key a param, defines parameter group

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

Many others

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

Basic SGD Optimizer

Move each parameter value in the direction of reducing gradient

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

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

Momentum-based Optimizer

Momentum vector helps accelerate in the direction where gradient is decreasing

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

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

Momentum-based Optimizer

Momentum vector helps accelerate in the direction where gradient is decreasing

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

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

Momentum-based Optimizer

Momentum vector helps accelerate in the direction where gradient is decreasing

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

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

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 hyperparameters

Better performance

Adam ~ Adaptive Moment Estimation

Demo

Regression Using NN Layers and Optimizer

Dropout to Mitigate Overfitting

Preventing Overfitting

Regularisation - Penalise complex models

Cross-validation - Distinct training and validation phases

Dropout - Intentionally turn off some neurons during training

Dropout

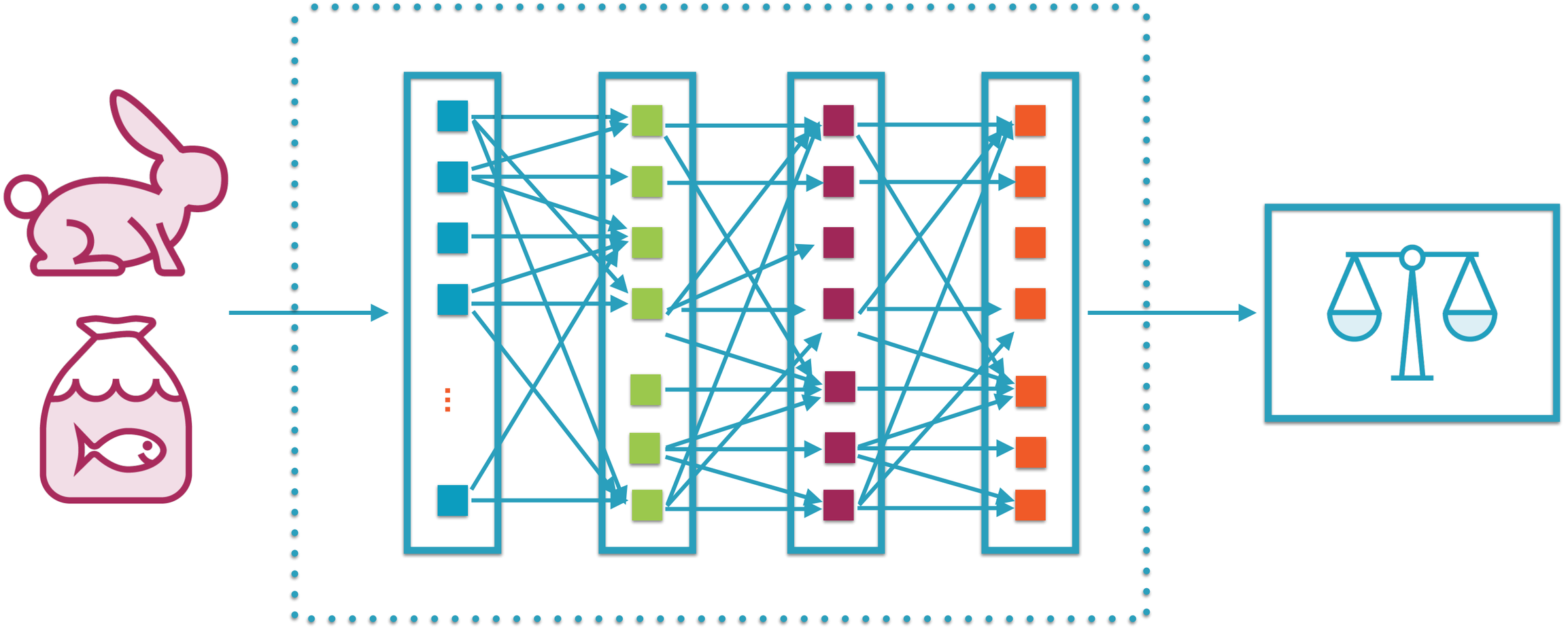
Specify a fraction of neurons that will stay off in each training step

“Dropout” neurons chosen at random

Different neurons off in each training step

In effect, each training step builds different network configuration

Densely Connected Neural Network

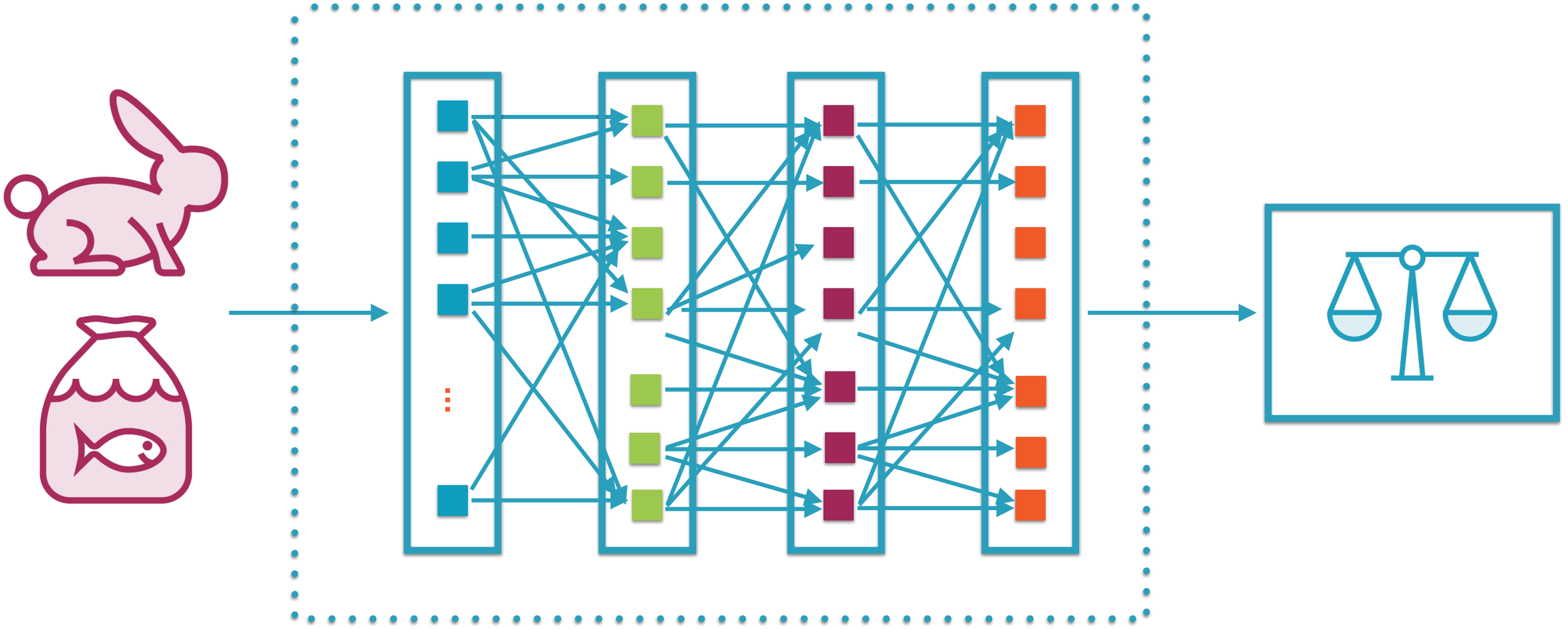


Corpus of Images

High risk of overfitting during training due to dense, complex network

ML-based Classifier

Dropout = 50%

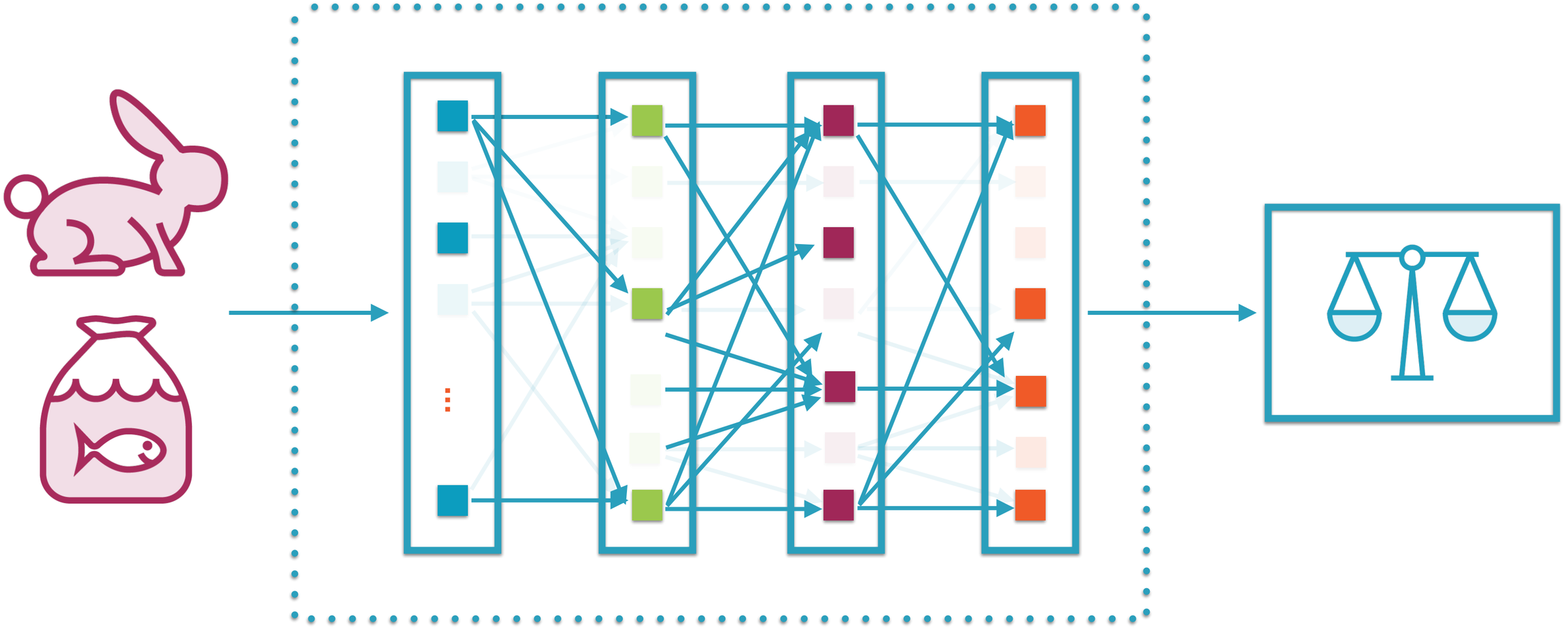


Corpus of Images

Randomly switch off say 50% of neurons in each training step

ML-based Classifier

Dropout = 50%

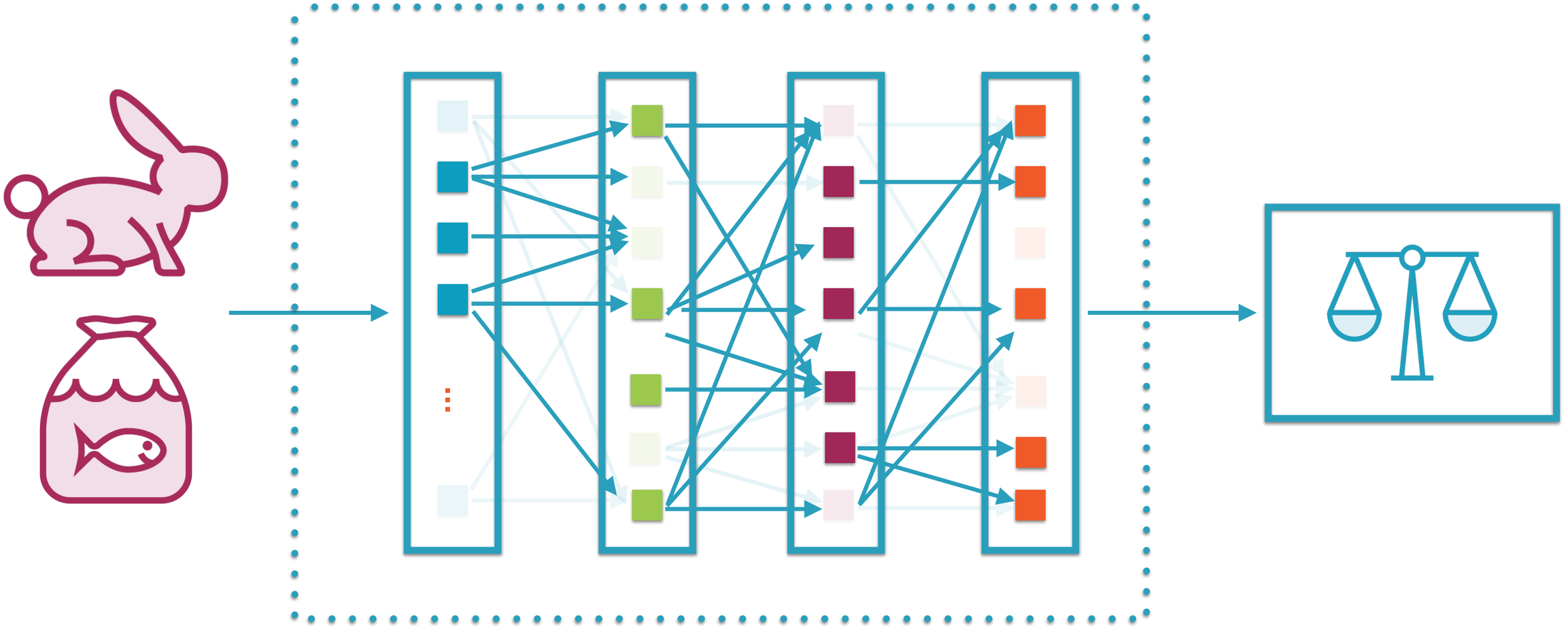


Corpus of Images

Training forced to rely on a much simpler neural network

ML-based Classifier

Dropout = 50%

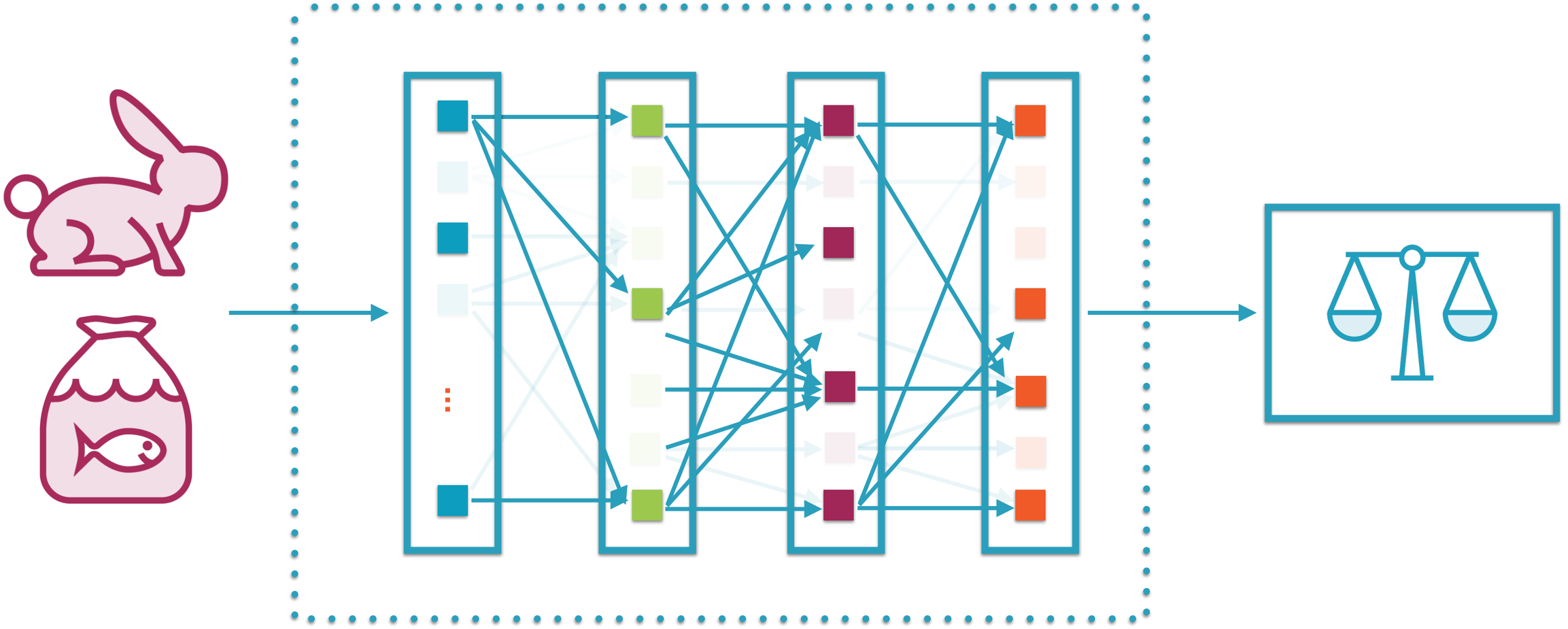


Corpus of Images

Each training step will build a different configuration

ML-based Classifier

Dropout = 50%

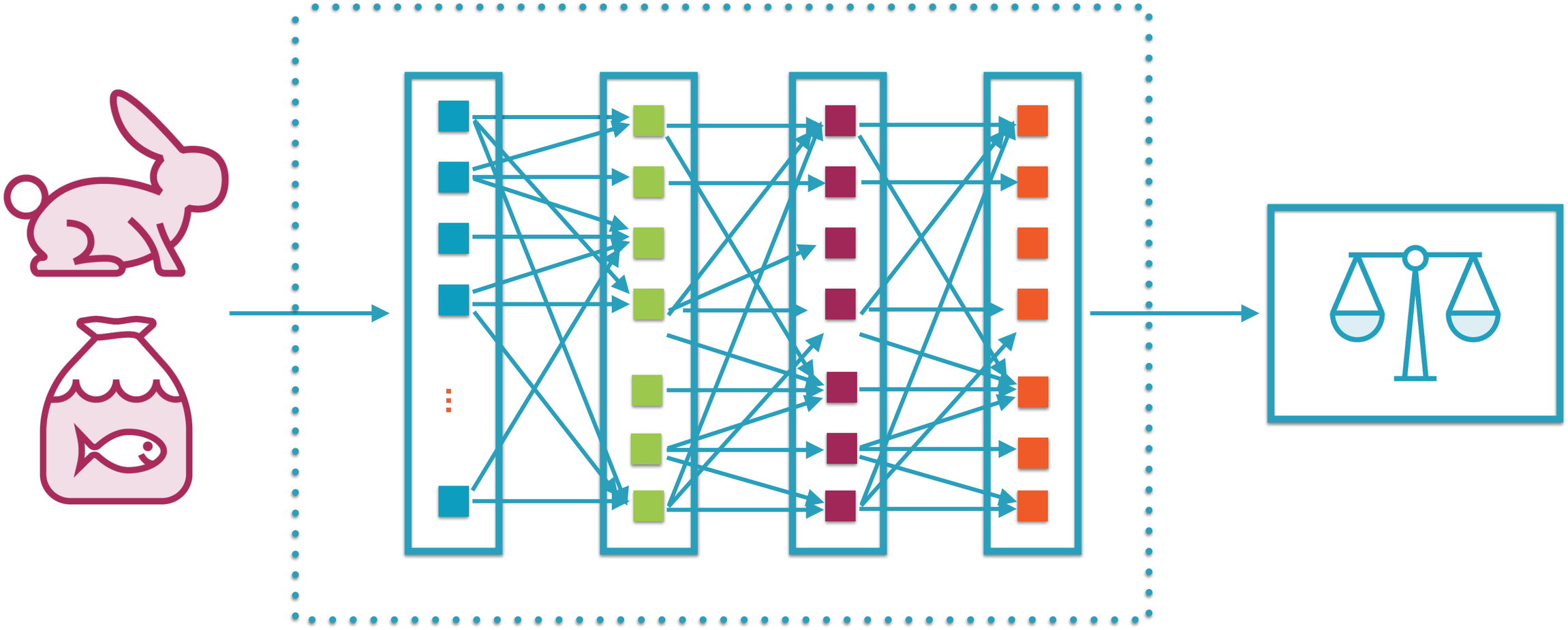


Corpus of Images

Each training step will build a different configuration

ML-based Classifier

Dropout During Training Only



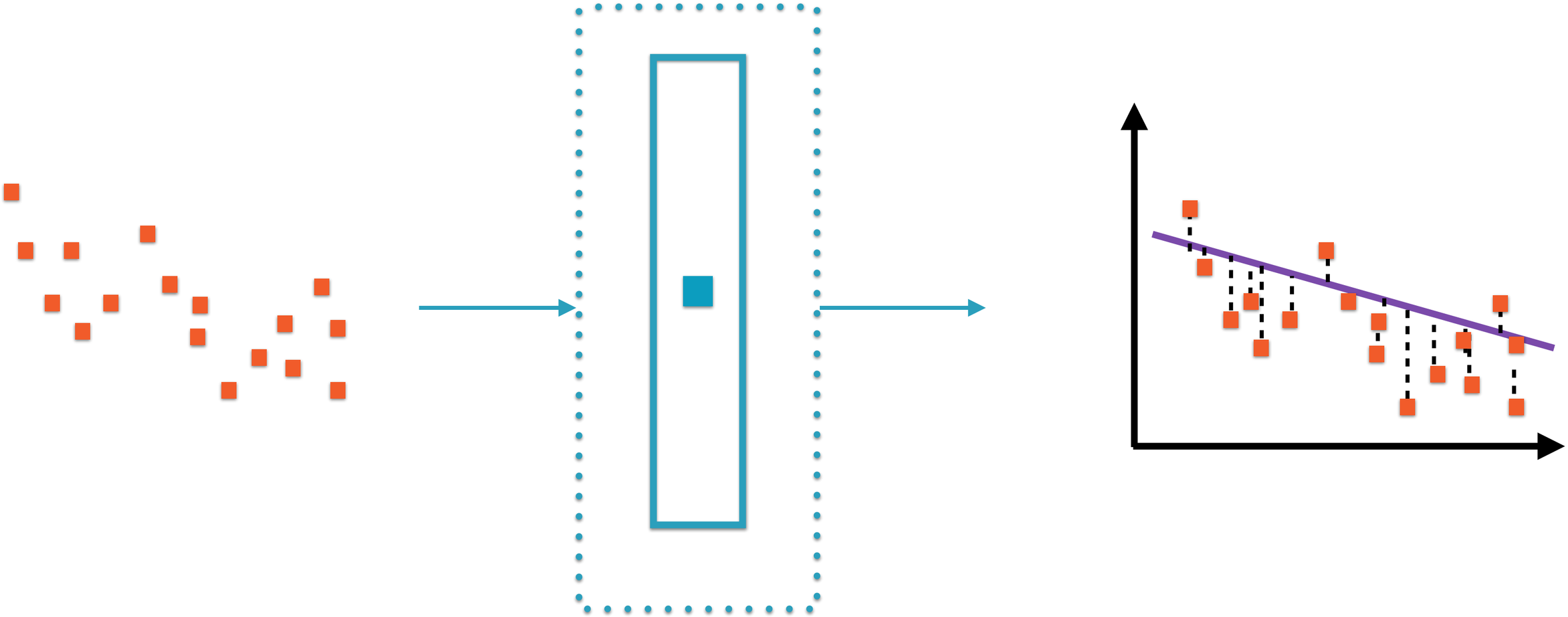
Corpus of Images

During actual usage in test mode, full dense neural network is used

ML-based Classifier

Gradient Descent and Batches

Regression: The Simplest Neural Network

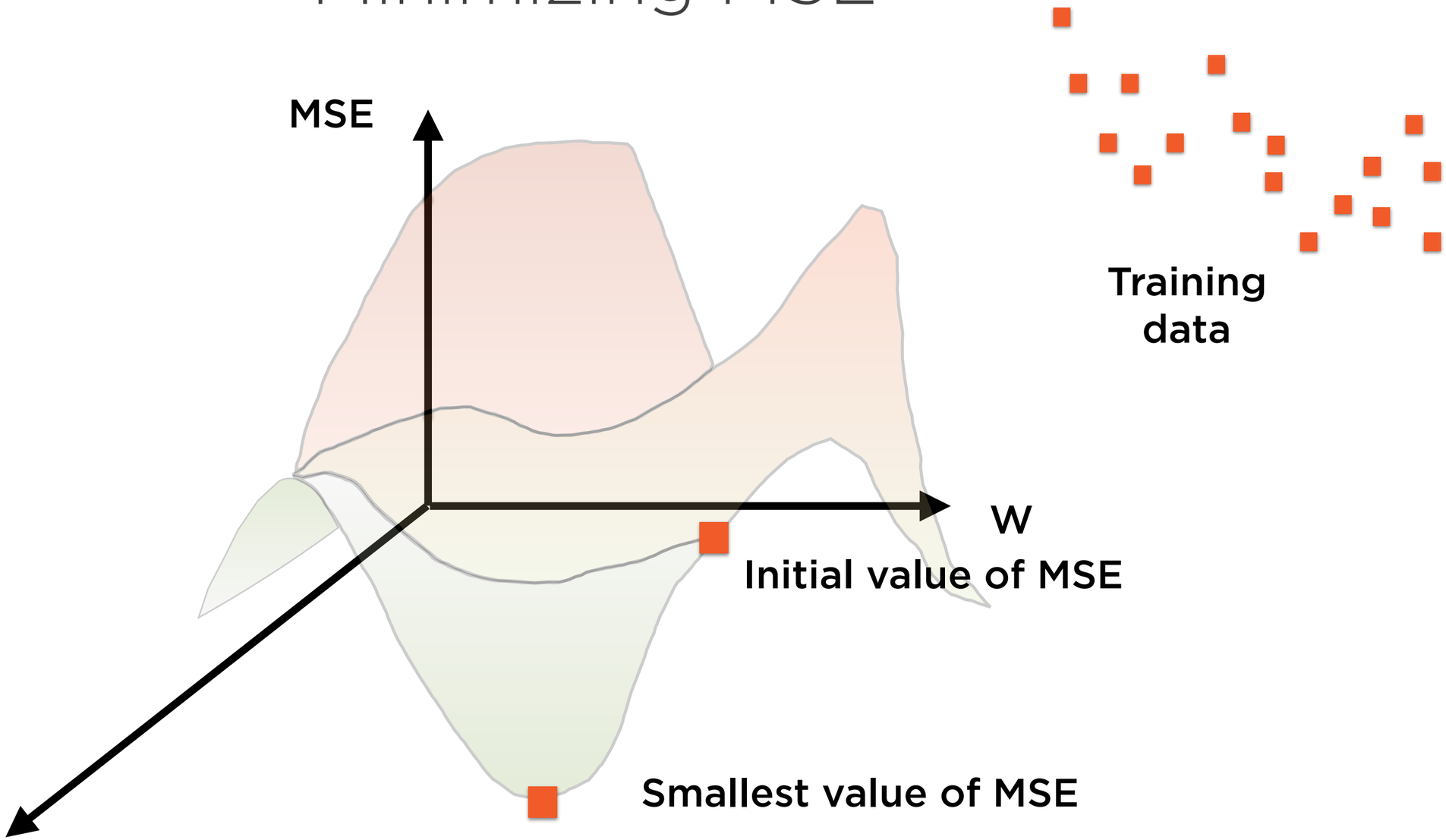


Training data

Single Neuron

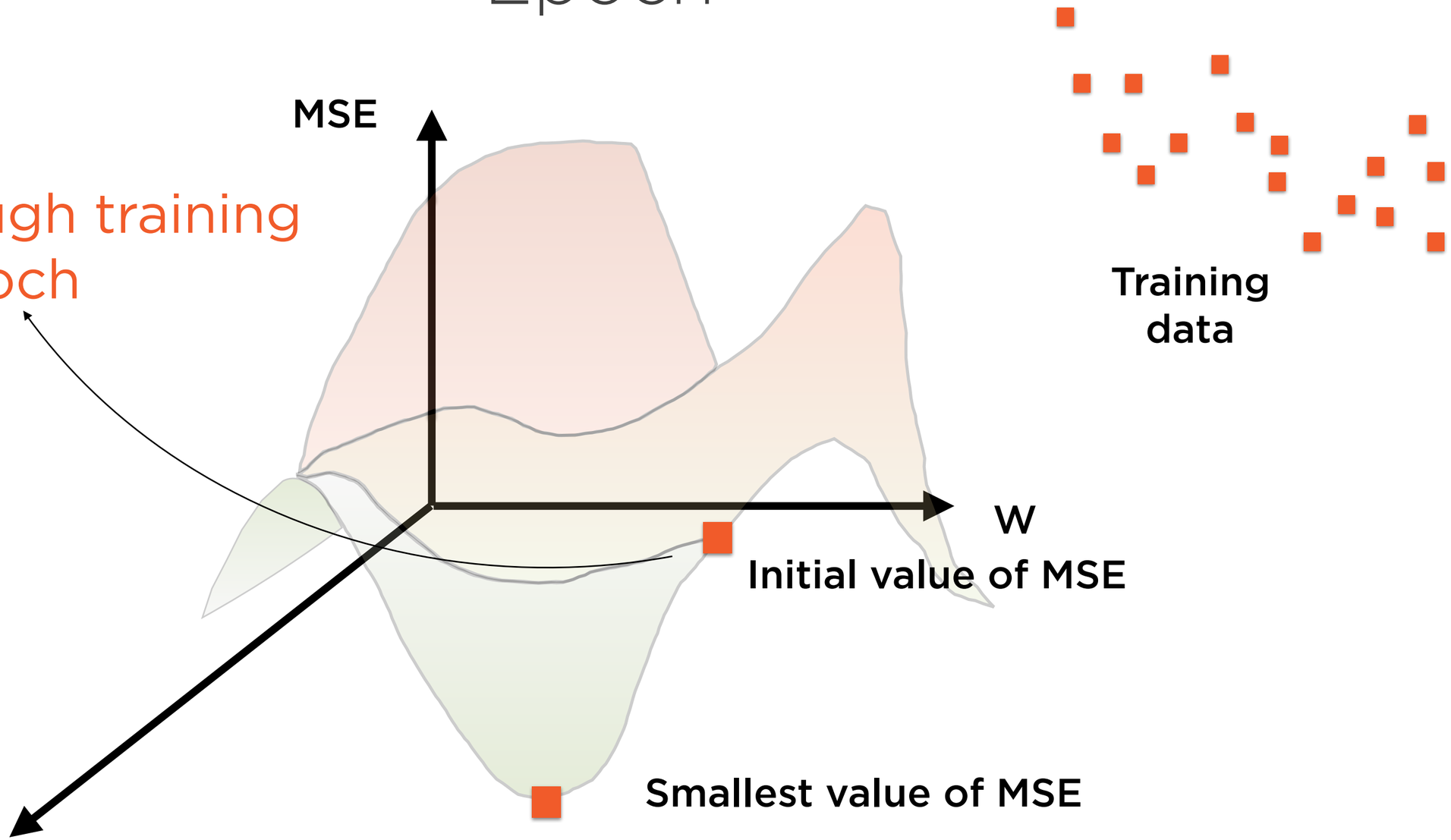
Regression Line

Minimizing MSE

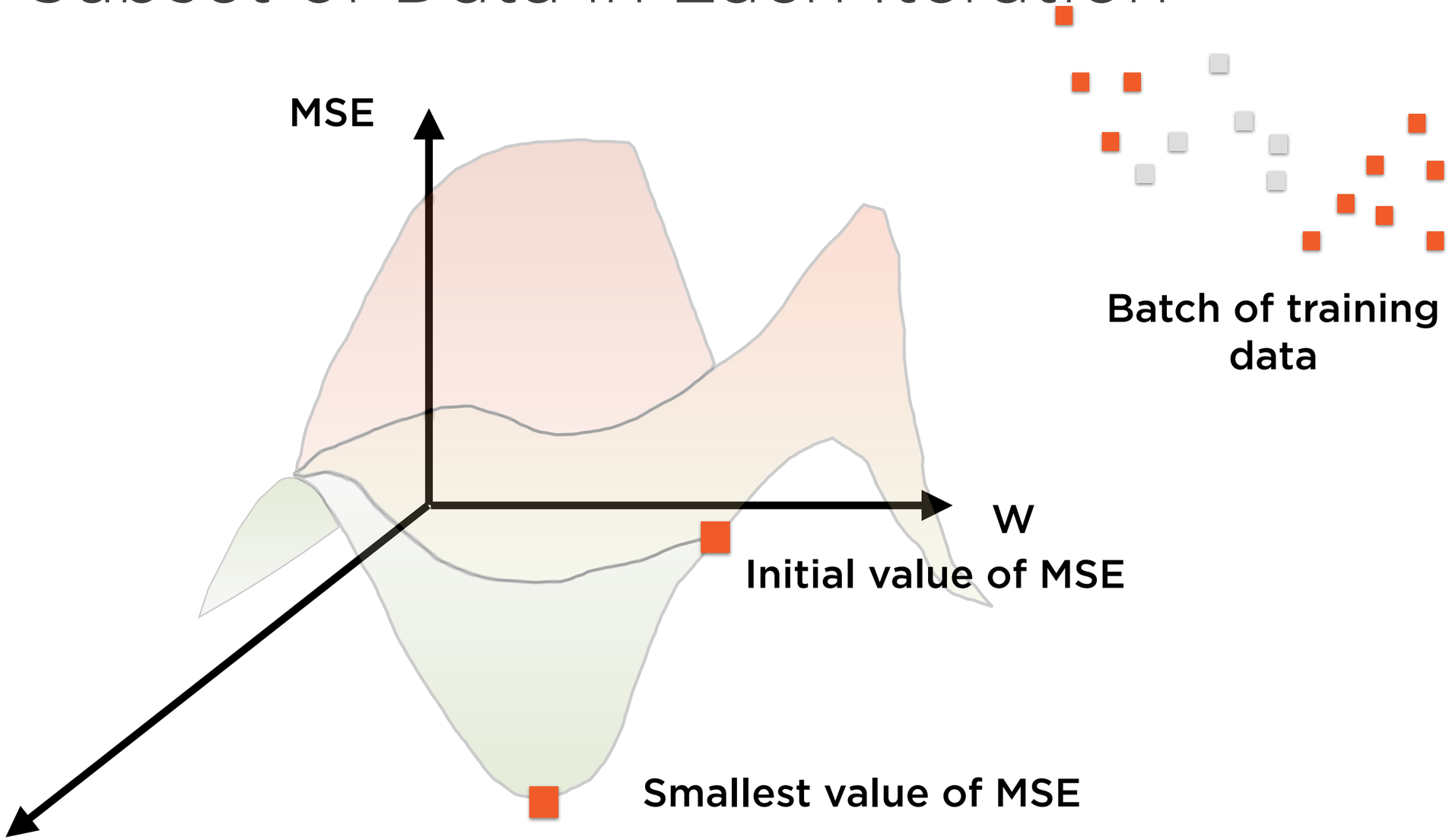


“Epoch”

1 pass through training data = 1 epoch

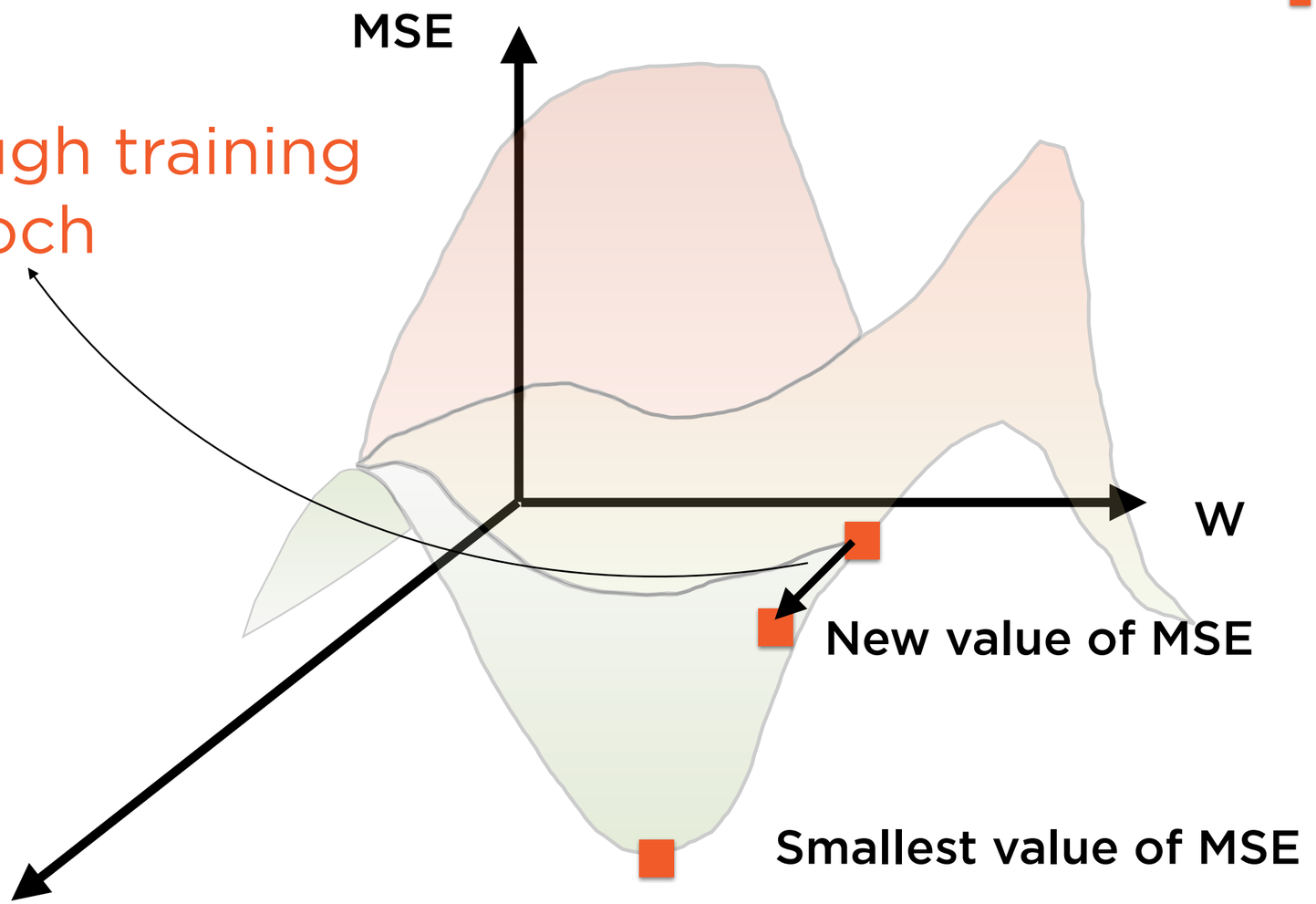


Subset of Data in Each Iteration



Epoch ~ Multiple Batches

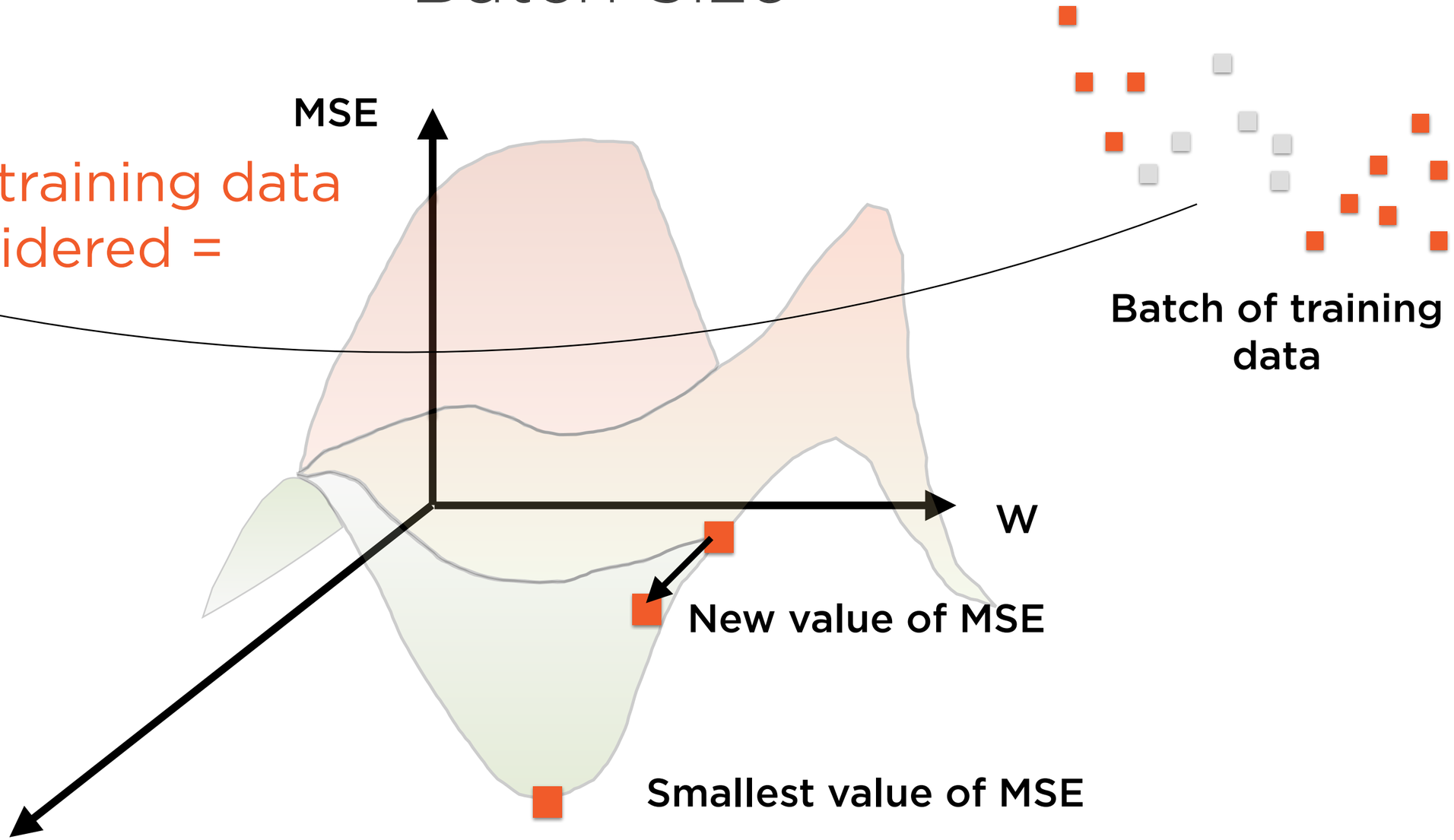
1 pass through training data = 1 epoch



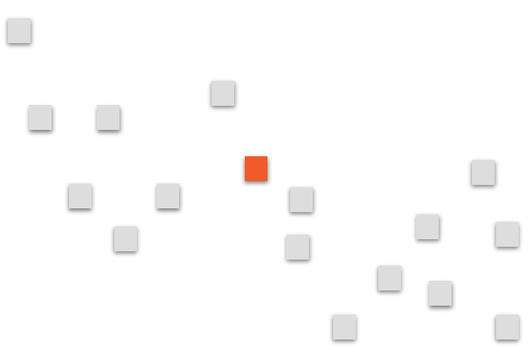
Batch of training data

“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

Demo

Building a regression model for demand prediction

Summary

Training a neural network using forward and backward passes

Using optimizers to update model parameters

Using layers and activation functions to train and build neural networks

Understanding and using dropout to mitigate overfitting