EM 538-001: PRACTICAL MACHINE LEARNING FOR ENGINEERING ANALYTICS

LECTURE 009

Fred Livingston, Ph.D.

LECTURE 009

- DBSCAN Worked Example
- Regression Models

EM 538-001: Practical Machine Learning for Enginering Analystics (Spring 2025)
Instructor: Fred Livingston (fjliving@ncsu.edu)

Load and Prepare Datasets

```python
from sklearn.model_selection import train_test_split
import pandas as pd

df_iris = pd.read_csv('iris.csv')

X = df_iris[['PetalLength[cm]', 'PetalWidth[cm]']]
y = df_iris['Species']
```

Python

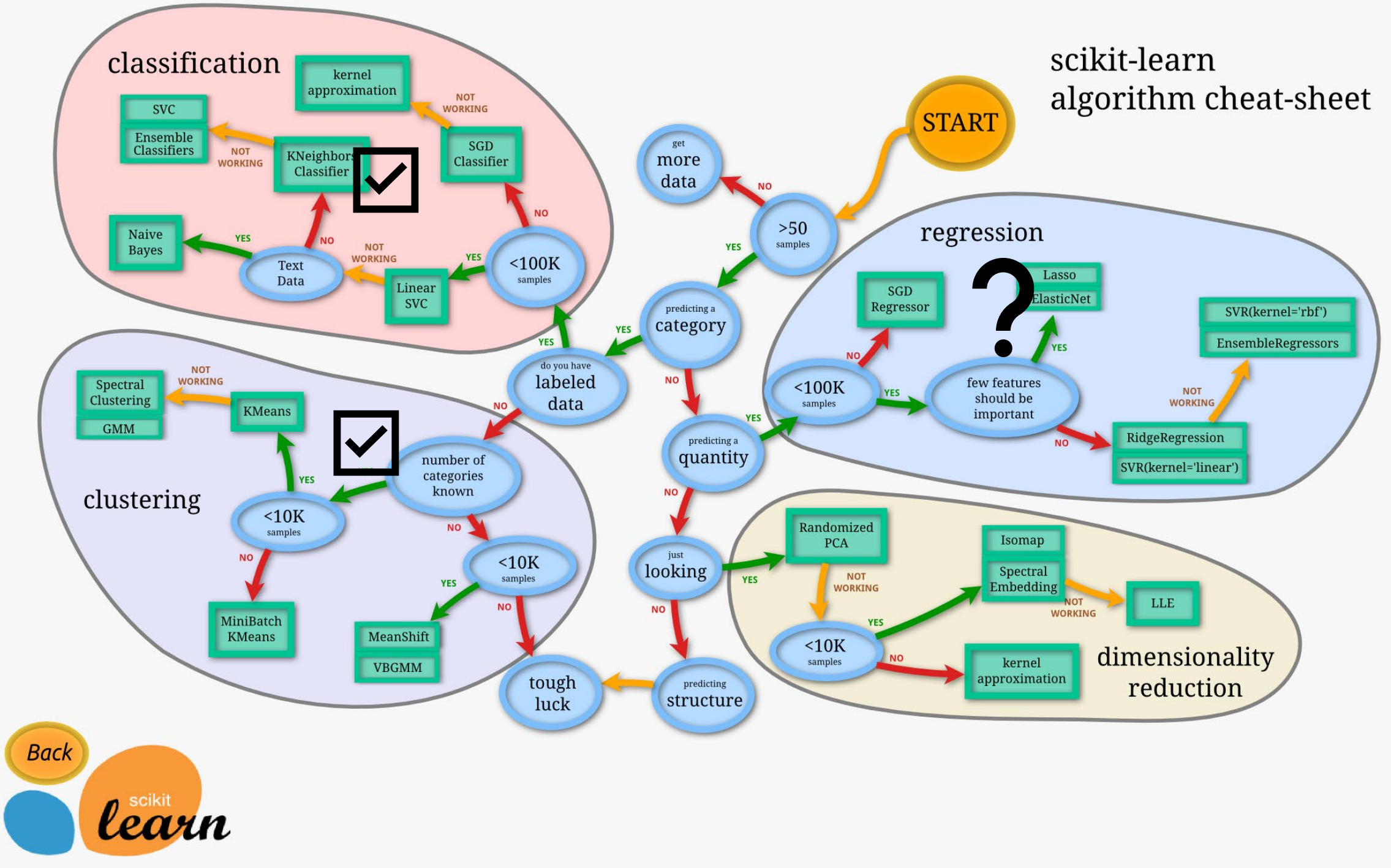
DBSCAN clustering algorithm

eps

- The maximum distance between two samples for one to be considered as in the neighborhood of the other.
- This is not a maximum bound on the distances of points within a cluster.
- This is the most important DBSCAN parameter to choose appropriately for your data set and distance function.

min_samples

- The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.


scikit-learn
algorithm cheat-sheet
START
classification
kernel approximation
NOT WORKING
SVC
Ensemble Classifiers
NOT WORKING
KNeighbors Classifier
SGD Classifier
NO
Naive Bayes
YES
NO
Text Data
NOT WORKING
YES
<100K samples
Linear SVC
NO
YES
get more data
NO
>50 samples
YES
regression
predicting a category
YES
NO
SGD Regressor
Lasso
ElasticNet
?
<100K samples
NO
YES
few features should be important
YES
NO
SVR(kernel='rbf')
EnsembleRegressors
NOT WORKING
RidgeRegression
SVR(kernel='linear')
do you have labeled data
YES
NO
predicting a quantity
YES
NO
clustering
Spectral Clustering
GMM
NOT WORKING
KMeans
YES
number of categories known
NO
<10K samples
NO
MiniBatch KMeans
<10K samples
YES
NO
MeanShift
VBGMM
just looking
YES
NO
Randomized PCA
NOT WORKING
Isomap
Spectral Embedding
NOT WORKING
LLE
YES
<10K samples
NO
kernel approximation
dimensionality reduction
tough luck
predicting structure
Back
scikit learn

REGRESSION MODELS

- Linear Regression
- Loss
- Gradient Descent
- Linear Regression Python Example
- Hyperparameters
- Linear Regression using Scikit-Learn

LINEAR REGRESSION

LINEAR REGRESSION

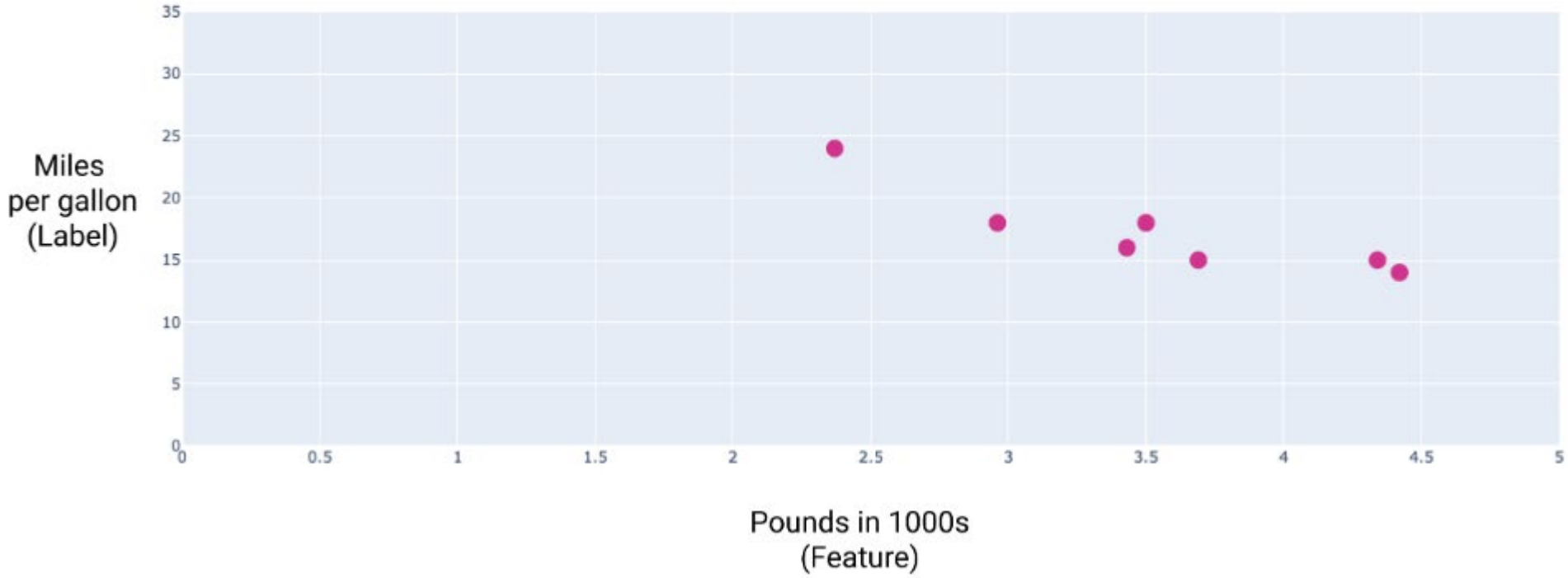
Linear regression is a statistical technique used to find the relationship between variables. In an ML context, linear regression finds the relationship between **features** and a **label**.

For example, suppose we want to predict a car's fuel efficiency in miles per gallon based on how heavy the car is, and we have the following dataset:

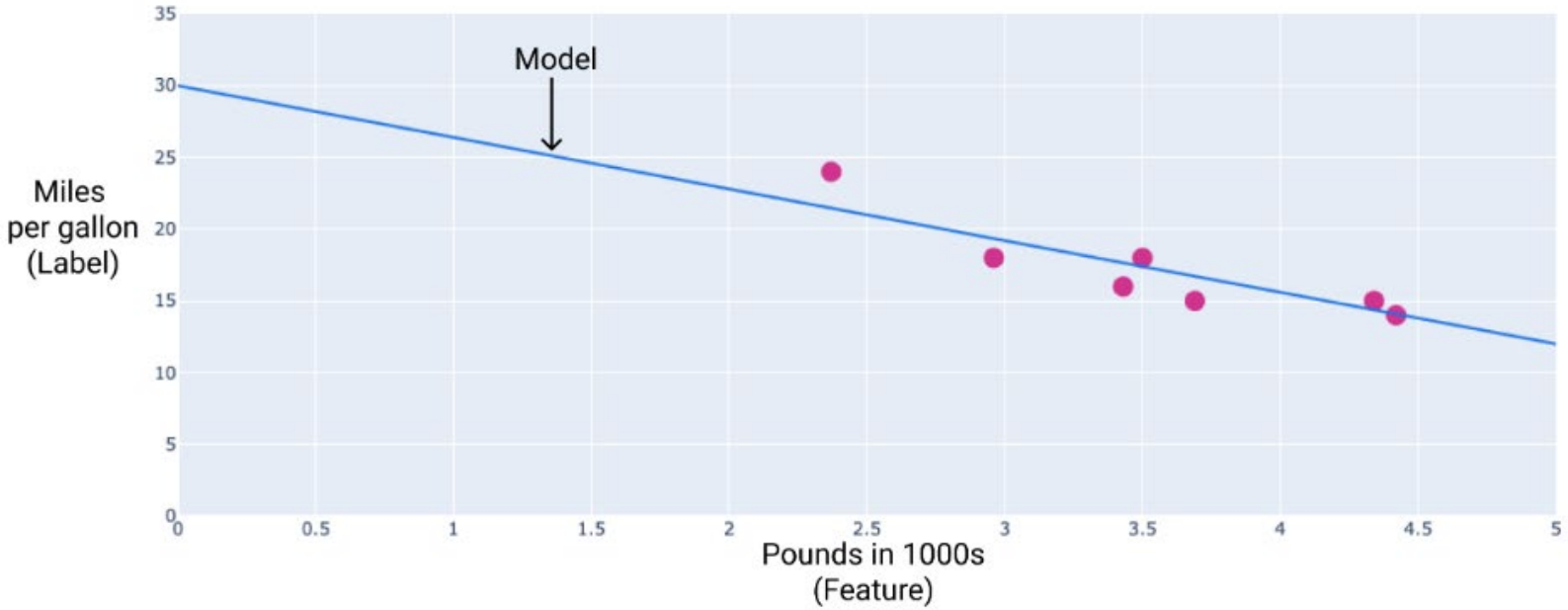
Pounds in 1000s (feature)	Miles per gallon (label)
3.5	18
3.69	15
3.44	18
3.43	16
4.34	15
4.42	14
2.37	24

DATASET

Pounds in 1000s (feature)	Miles per gallon (label)
3.5	18
3.69	15
3.44	18
3.43	16
4.34	15
4.42	14



LINEAR REGRESSION EQUATION



In algebraic terms, the model would be defined as y=m**X**+b, where

•y is miles per gallon—the value we want to predict.
•m is the slope of the line.
•x is pounds—our input value.
•b is the y-intercept.

LINEAR REGRESSION EQUATION

In ML, we write the equation for a linear regression model as follows:

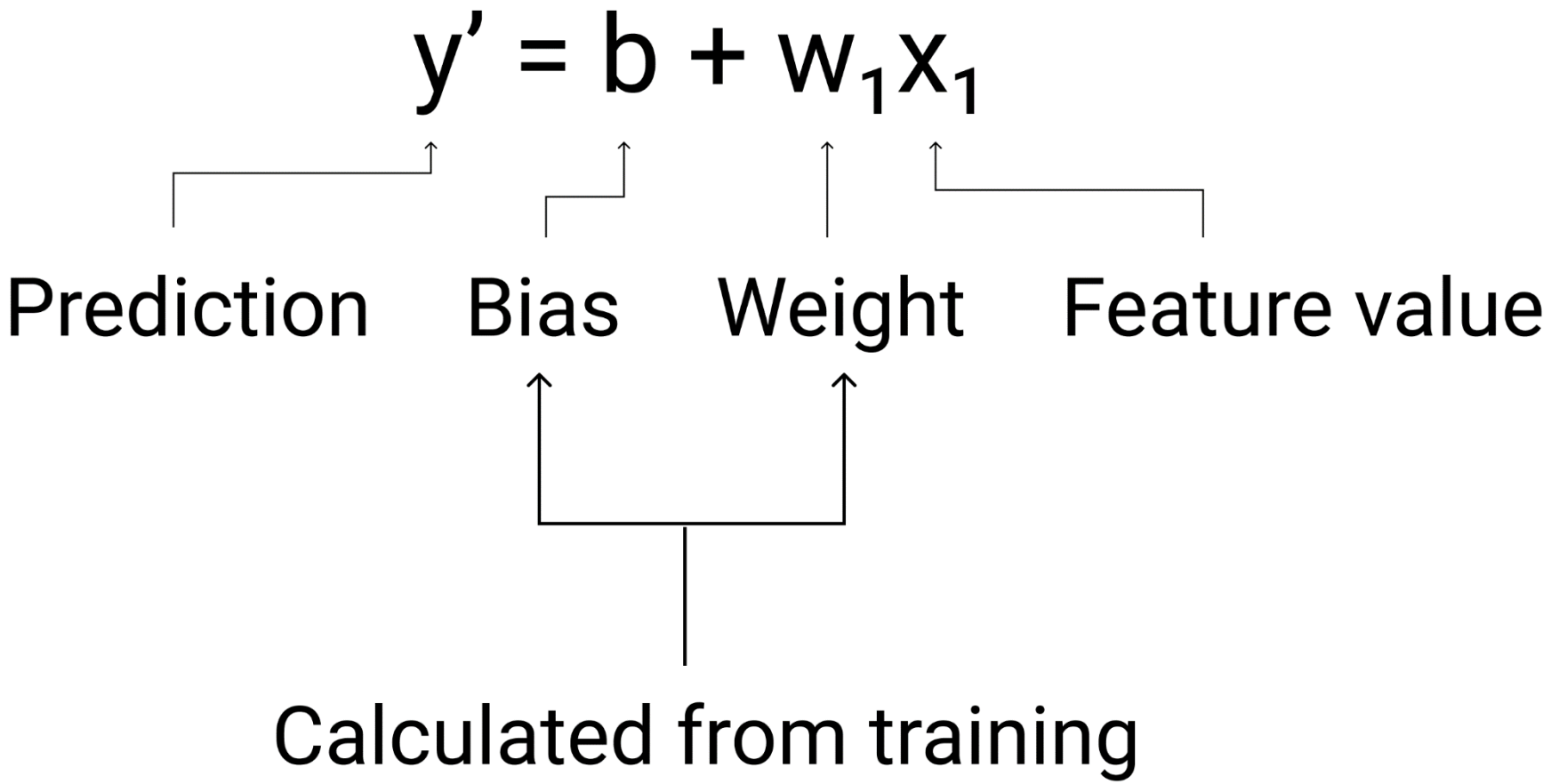
$$y'=b+w_1x_1$$

where:

- y' is the predicted label—the output.
- b is the **bias** of the model. Bias is the same concept as the y-intercept in the algebraic equation for a line. In ML, bias is sometimes referred to as w0. Bias is a **parameter** of the model and is calculated during training.
- w1 is the **weight** of the feature. Weight is the same concept as the slope m in the algebraic equation for a line. Weight is a **parameter** of the model and is calculated during training.
- x1 is a **feature**—the input.

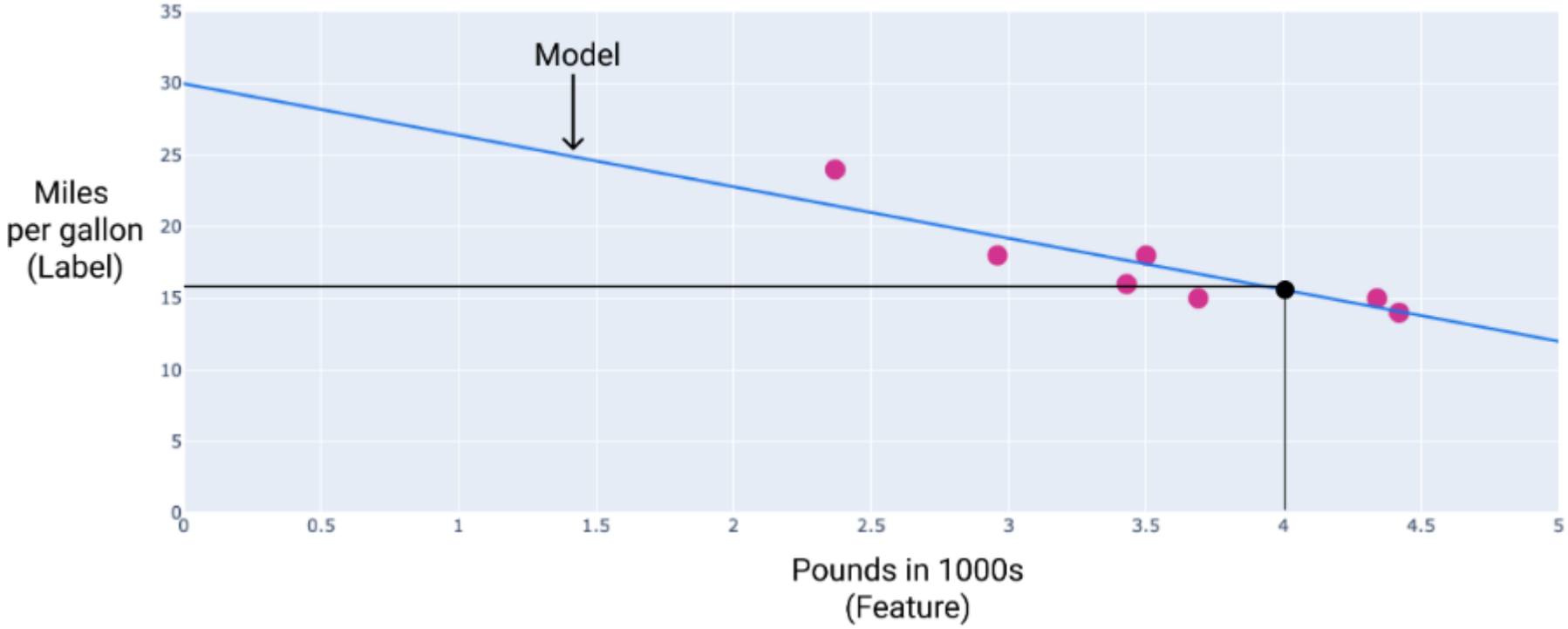
TRAINING MODEL

During training, the model calculates the weight and bias that produce the best model.



MATHEMTAICAL REPRESENTATION OF LINEAR MODEL

In our example, we'd calculate the weight and bias from the line we drew. The bias is 30 (where the line intersects the y-axis), and the weight is -3.6 (the slope of the line). The model would be defined as y'=30 + (-3.6)(x1), and we could use it to make predictions. For instance, using this model, a 4,000-pound car would have a predicted fuel efficiency of 15.6 miles per gallon.



Using the model, a 4,000-pound car has a predicted fuel efficiency of 15.6 miles per gallon.

Model parameters to compute value

$$x_1 \times w_1 + x_2 \times w_2 + b = Z = Y'$$

$$x_1 \times w_1 + x_2 \times w_2 + \ldots + x_m \times w_m + b = z = y'$$

Model parameters to compute value

$$x_1 \times w_1 + x_2 \times w_2 + b = Z$$

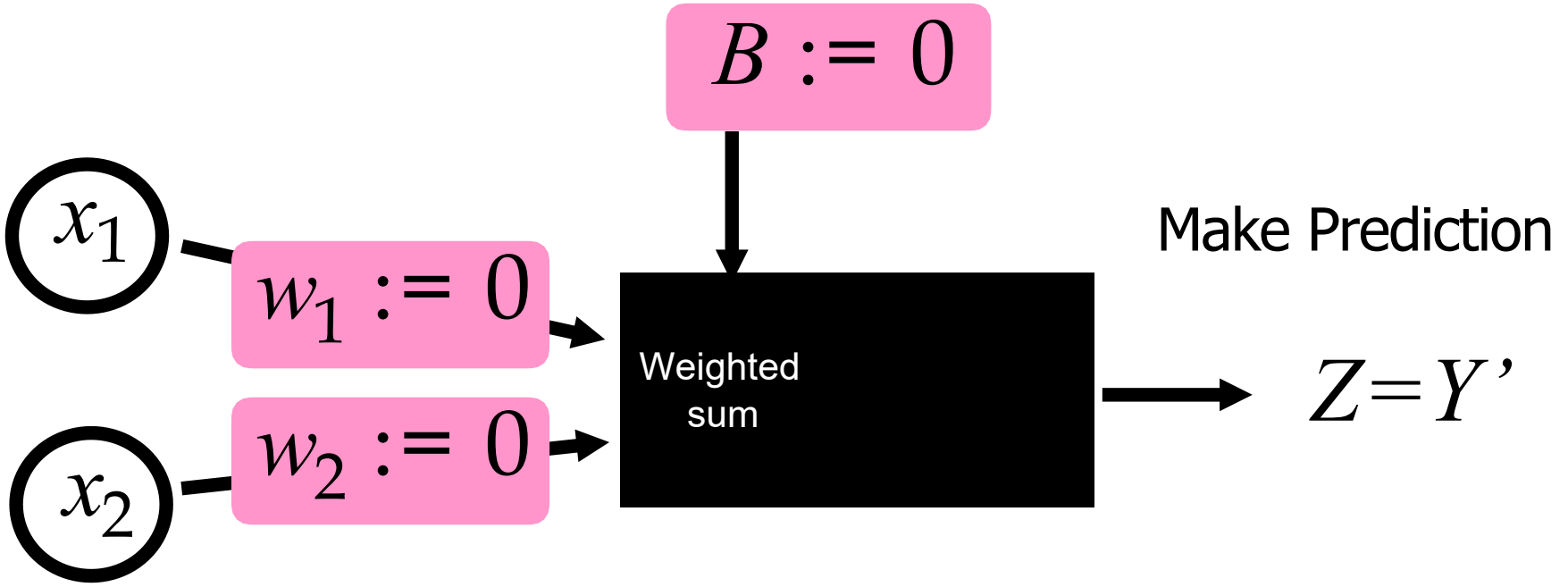
$$x_1 \times w_1 + x_2 \times w_2 + \ldots + x_m \times w_m + b = z$$

$$= b + \sum^{m} x_i \times w_i$$

1. Define training set

2. Initialize model weights and bias to zero

Pounds in 1000s (feature)	Miles per gallon (label)
3.5	18
3.69	15
3.44	18
3.43	16
4.34	15
4.42	14
2.37	24



MODEL PREDICTIONS (FORWARD PROPAGATION)

```python
import matplotlib.pyplot as plt
import matplotlib.axes as ax
from matplotlib.animation import FuncAnimation
import numpy as np

class LinearRegression:
    def __init__(self):
        self.parameters = {}

    def forward_propagation(self, train_input):
        w = self.parameters['w']
        b = self.parameters['b']
        predictions = np.multiply(w, train_input) + b
        return predictions
```

1. DEFINE TRAINING SET

2. Initialize model weights and bias to zero

3. For every training epoch:

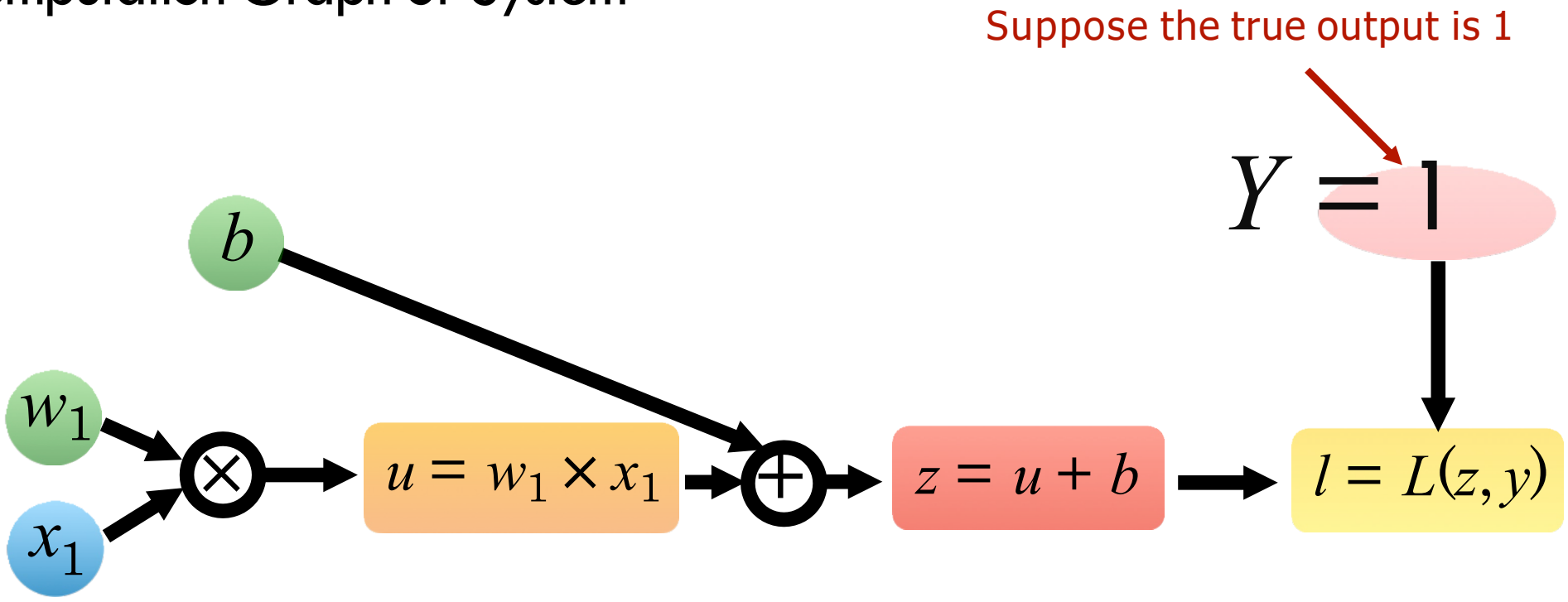
   a) For every training example $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:

      (i) Make a prediction

      (ii) Compute the error $error := y^{[i]} - \hat{y}^{[i]}$

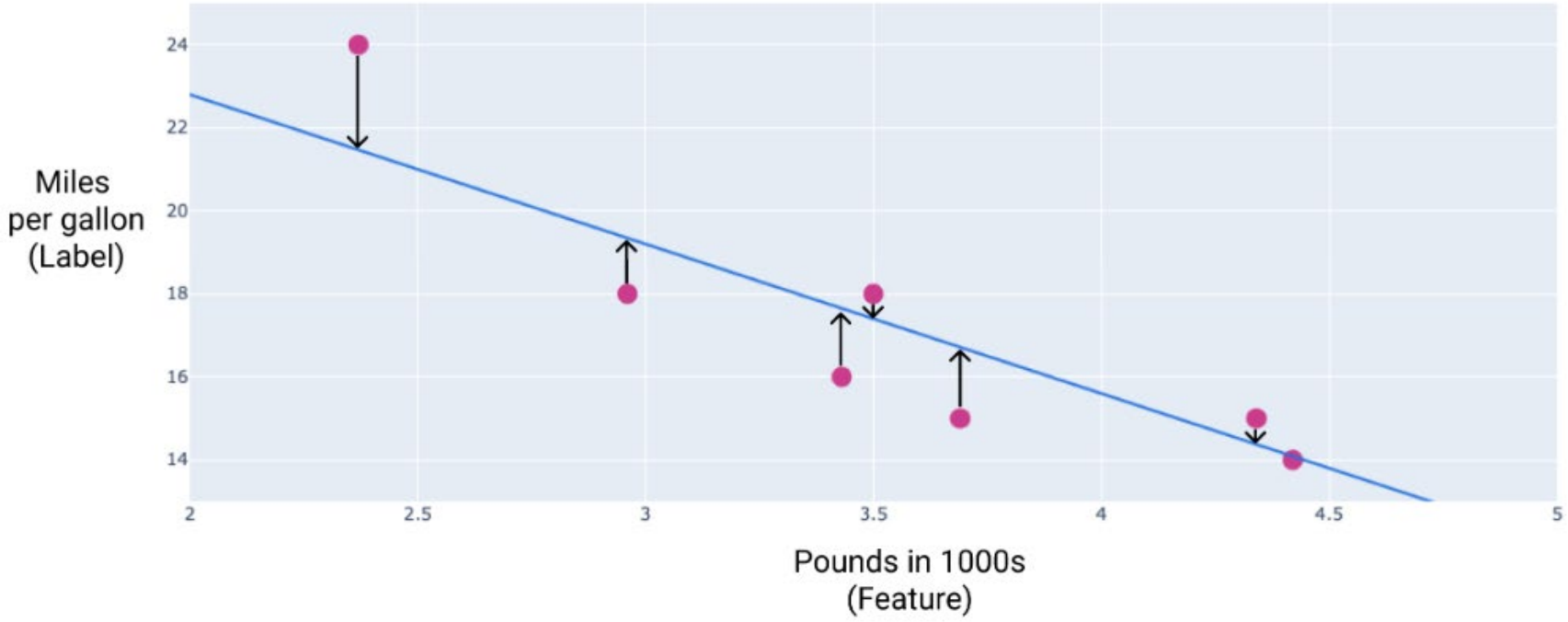
LOSS (COST)

Computation Graph of System



MODEL PREDICTIONS: LOSS

Loss is a numerical metric that describes how wrong a model's **predictions** are. Loss measures the distance between the model's predictions and the actual labels. The goal of training a model is to minimize the loss, reducing it to its lowest possible value.



The arrows show how far the model's predictions are from the actual values.

DISTANCE OF LOSS

In statistics and machine learning, loss measures the difference between the predicted and actual values. Loss focuses on the distance between the values, not the direction. For example, if a model predicts 2, but the actual value is 5, we don't care that the loss is negative -3 i.e (2-5 = -3)

Instead, we care that the distance between the values is 3.

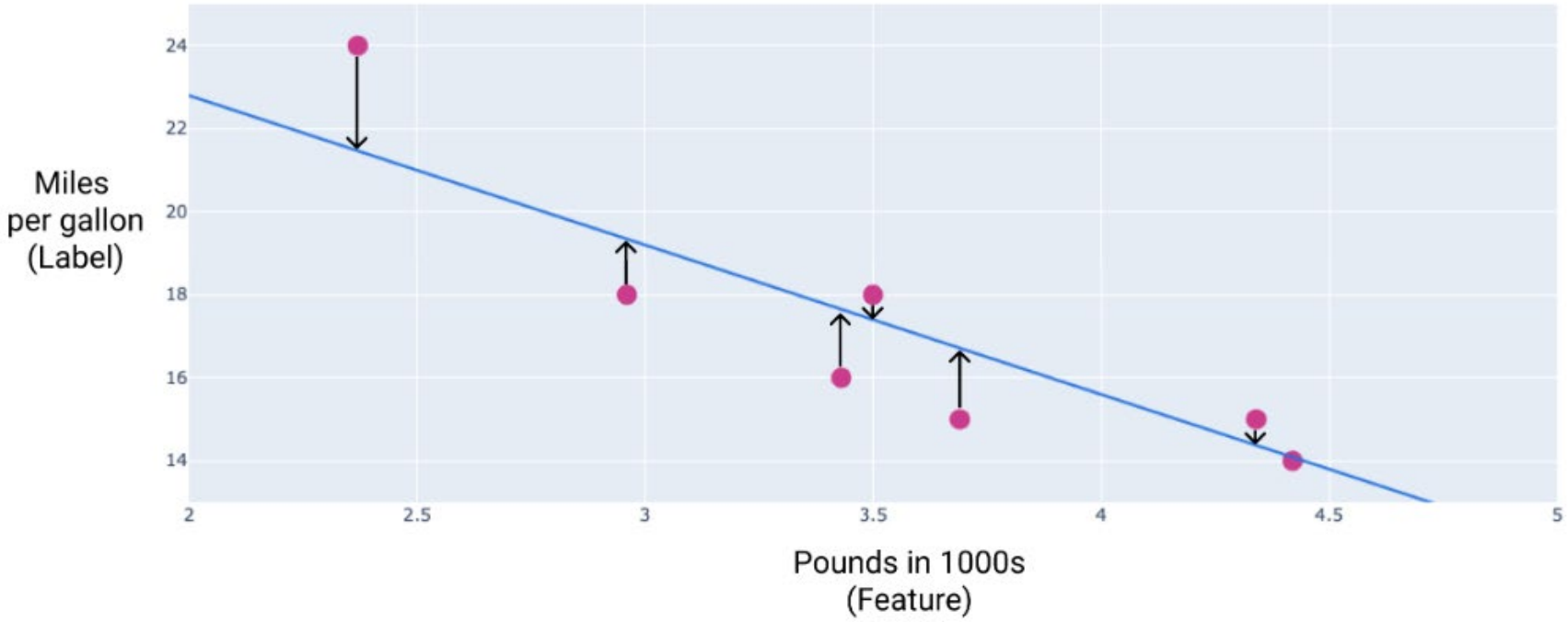
Thus, all methods for calculating loss remove the sign.

The two most common methods to remove the sign are the following:

- Take the absolute value of the difference between the actual value and the prediction.
- Square the difference between the actual value and the prediction.

CALCULATION L2 LOSS EXAMPLE

For a 2,370-pound car, **compute the L_2** loss, where the weight is -3.6 (mph per gallon, pounds) with a bias of 30 mpg



CALCULATION L2 LOSS SOLUTION

For a 2,370-pound car, **compute the L_2** loss, where the weight is -3.6 (mph per gallon, pounds) with a bias of 30 mpg

Value	Equation	Result
Prediction	$bias + (weight * feature\ value)$ $30 + (-3.6 * 2.37)$	21.5
Actual value	$label$	24
L_2 loss	$(prediction - actual\ value)^2$ $(21.5 - 24)^2$	6.25

TYPES OF LOSS

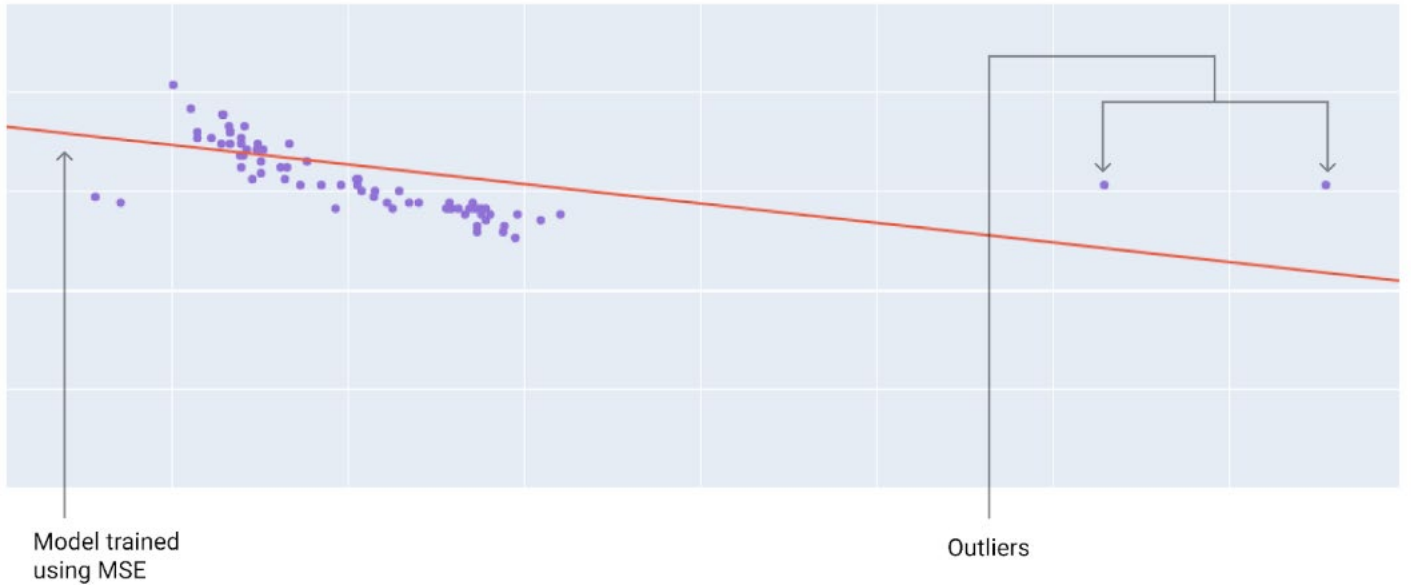
In linear regression, there are four main types of loss, which are outlined in the following table.

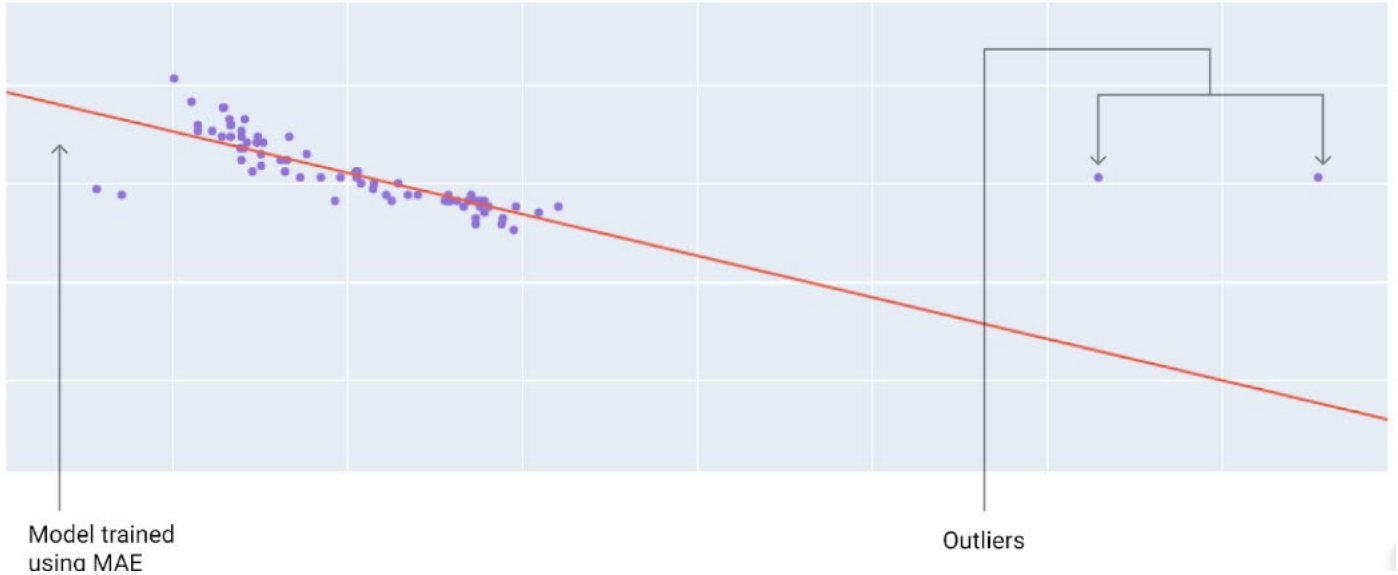
Loss type	Definition	Equation
L_1 loss	The sum of the absolute values of the difference between the predicted values and the actual values.	$\sum \lvert actual\ value - predicted\ value \rvert$
Mean absolute error (MAE)	The average of L_1 losses across a set of examples.	$\frac{1}{N} \sum \lvert actual\ value - predicted\ value \rvert$
L_2 loss	The sum of the squared difference between the predicted values and the actual values.	$\sum (actual\ value - predicted\ value)^2$
Mean squared error (MSE)	The average of L_2 losses across a set of examples.	$\frac{1}{N} \sum (actual\ value - predicted\ value)^2$

OUTLIERS

When choosing the best loss function, consider how you want the model to treat outliers. For instance, MSE moves the model more toward the outliers, while MAE doesn't. L_2 loss incurs a much higher penalty for an outlier than L_1 loss. For example, the following images show a model trained using MAE and a model trained using MSE. The red line represents a fully trained model that will be used to make predictions. The outliers are closer to the model trained with MSE than to the model trained with MAE.

OUTLIERS


Model trained
using MSE
Outliers

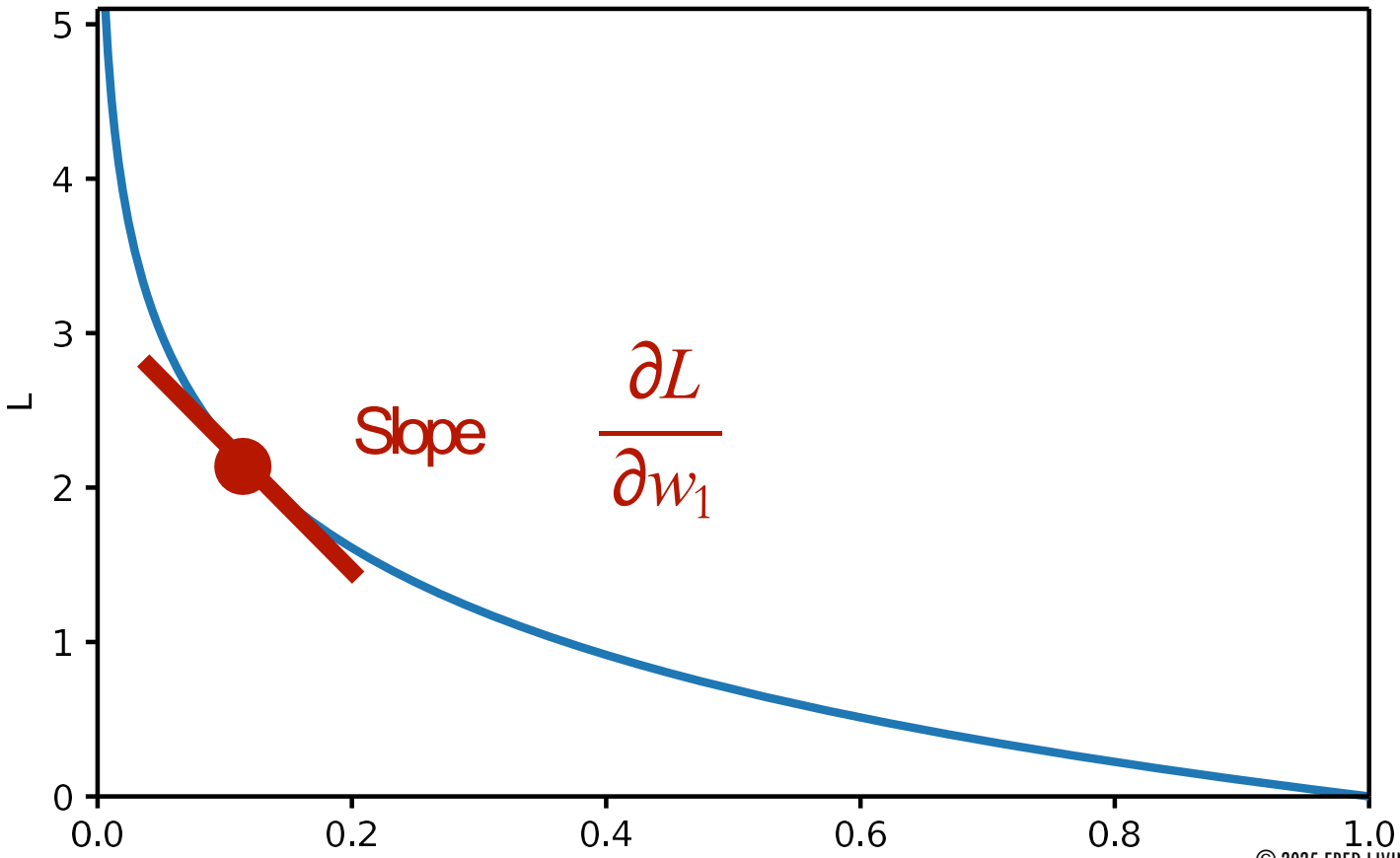

Model trained
using MAE
Outliers

LOSS/COST FUNCTION

```python
import matplotlib.pyplot as plt
import matplotlib.axes as ax
from matplotlib.animation import FuncAnimation
import numpy as np

class LinearRegression:
    def __init__(self):
        self.parameters = {}

    def forward_propagation(self, train_input): ···

    def cost_function(self, predictions, train_output):
        cost = np.mean((train_output - predictions) ** 2)
        return cost

    def backward_propagation(self, train_input, train_output, predictions): ···

    def update_parameters(self, derivatives, learning_rate): ···

    def train(self, train_input, train_output, learning_rate, iters): ···
```

5] ✓ 0.0s

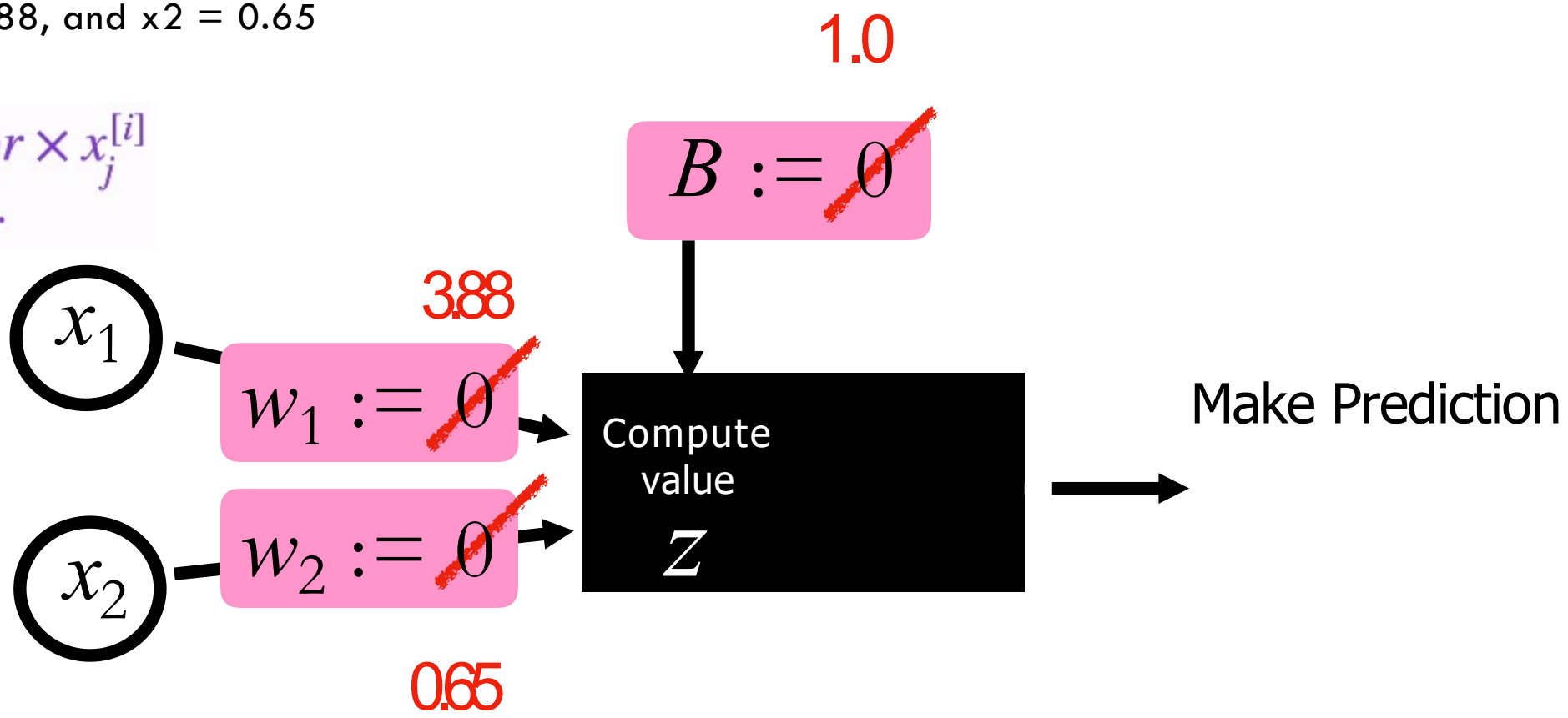
GRADIENT DESCENT



Update the weights based on the error

For where x1 = 3.88, and x2 = 0.65

$$w_j := w_j + error \times x_j^{[i]}$$
$$b := b + error$$



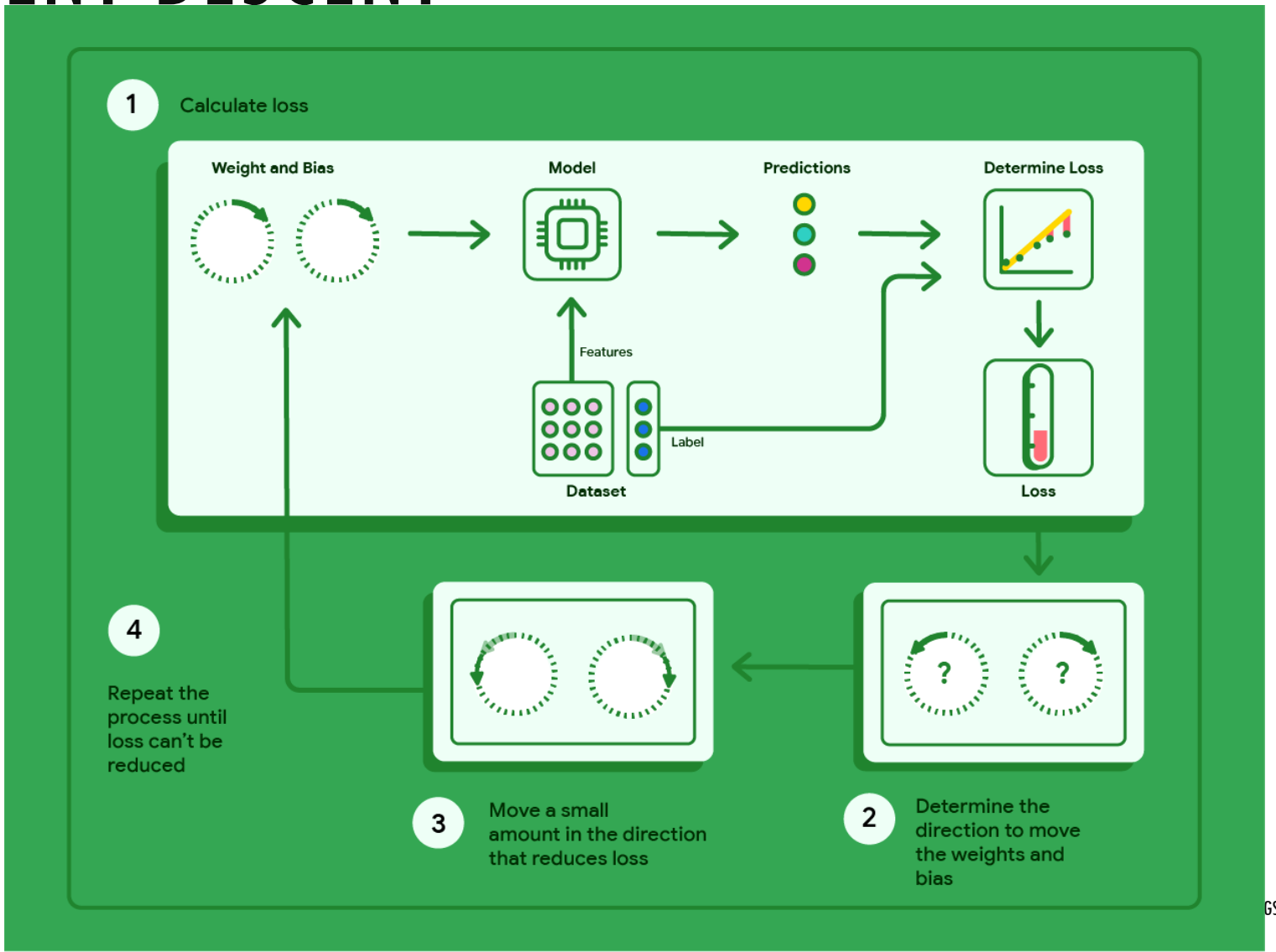
GRADIENT DESCENT

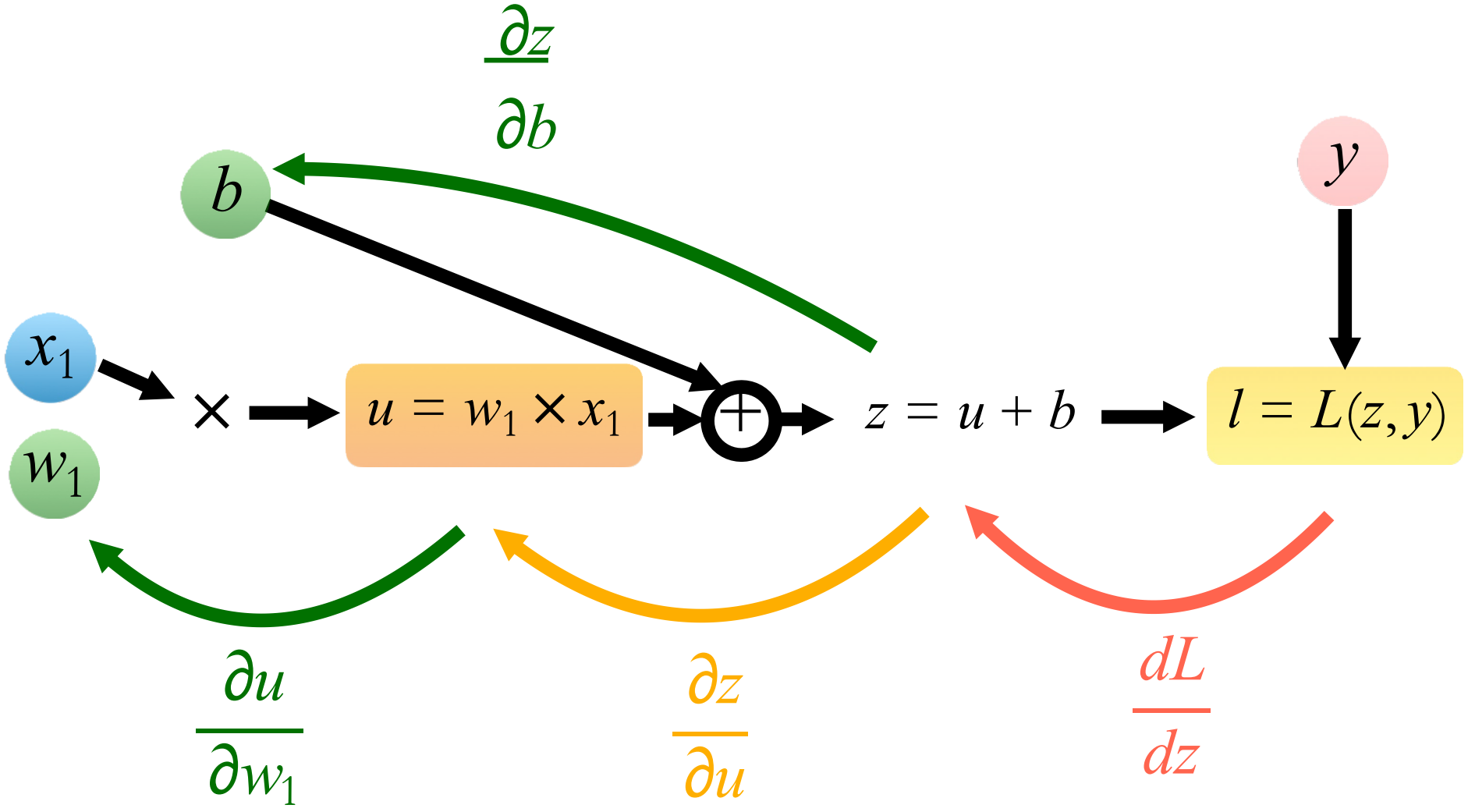
Gradient descent is a mathematical technique that iteratively finds the weights and bias that produce the model with the lowest loss. Gradient descent finds the best weight and bias by repeating the following process for a number of user-defined iterations.

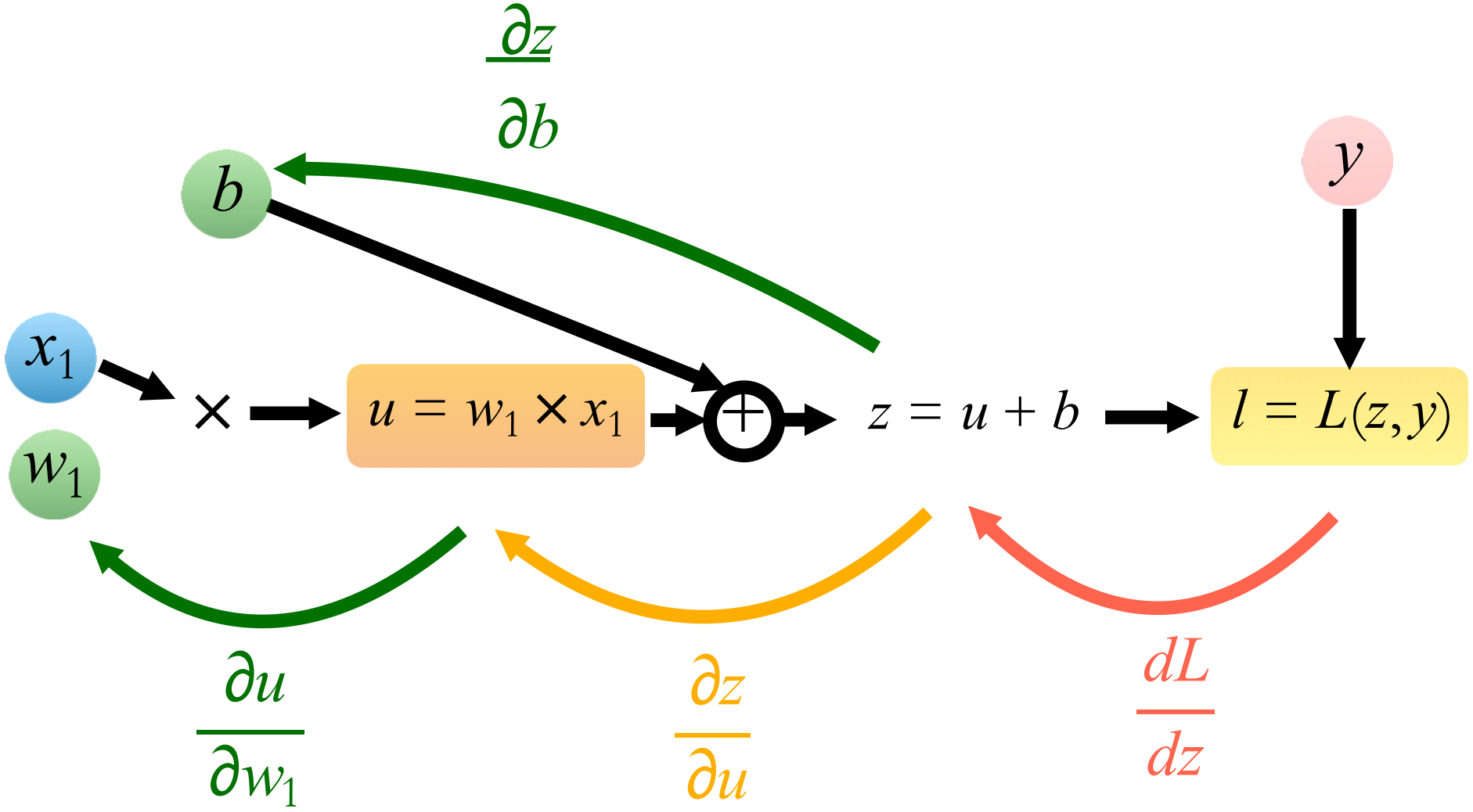
The model begins training with randomized weights and biases near zero, and then repeats the following steps:

1. Calculate the loss with the current weight and bias.
2. Determine the direction to move the weights and bias that reduce loss.
3. Move the weight and bias values a small amount in the direction that reduces loss.
4. Return to step one and repeat the process until the model can't reduce the loss any further.

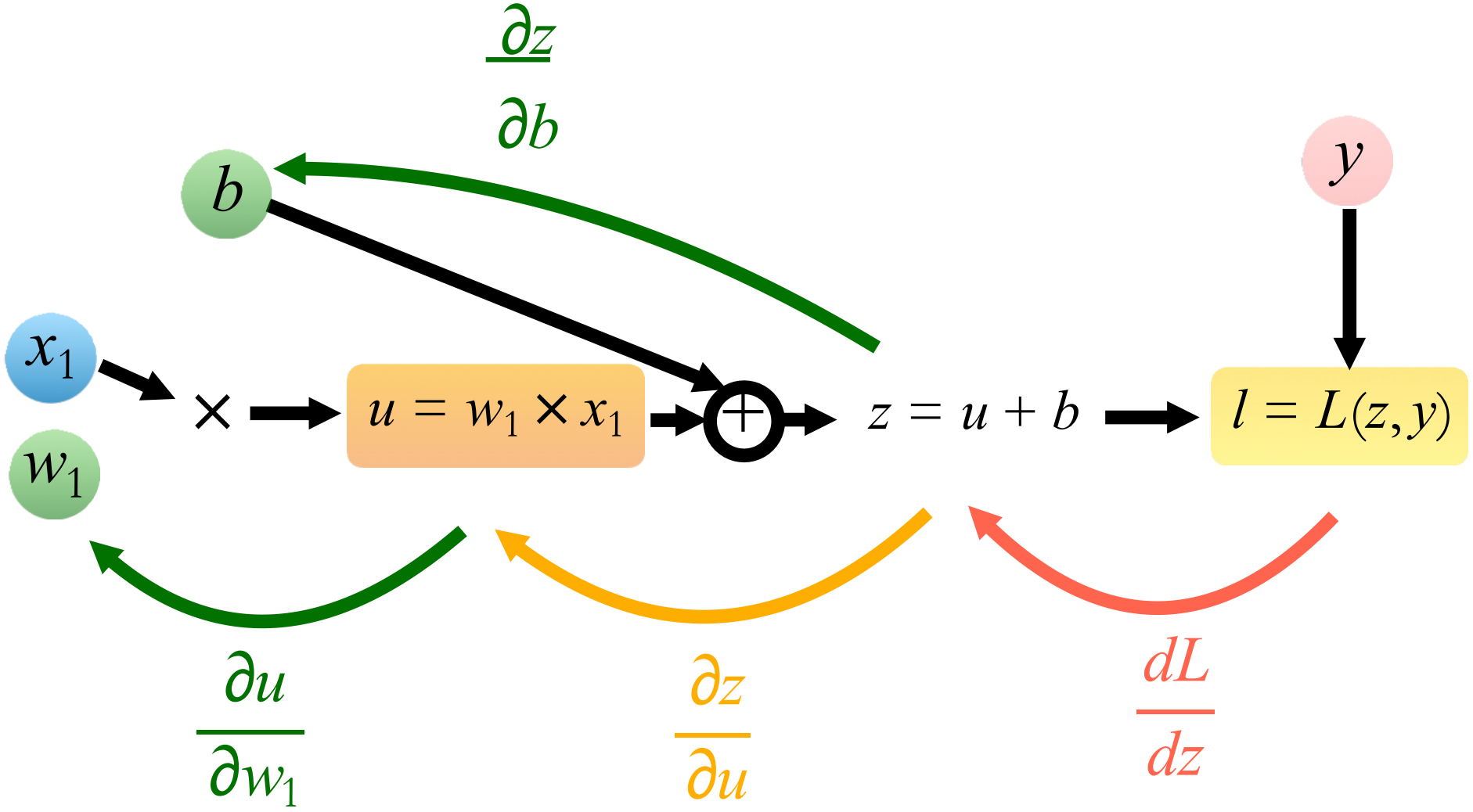
GRADIENT DESCENT




$\frac{\partial z}{\partial b}$
b
y
x_1
$\times$
$u = w_1 \times x_1$
$+$
$z = u + b$
$l = L(z, y)$
w_1
$\frac{\partial u}{\partial w_1}$
$\frac{\partial z}{\partial u}$
$\frac{dL}{dz}$

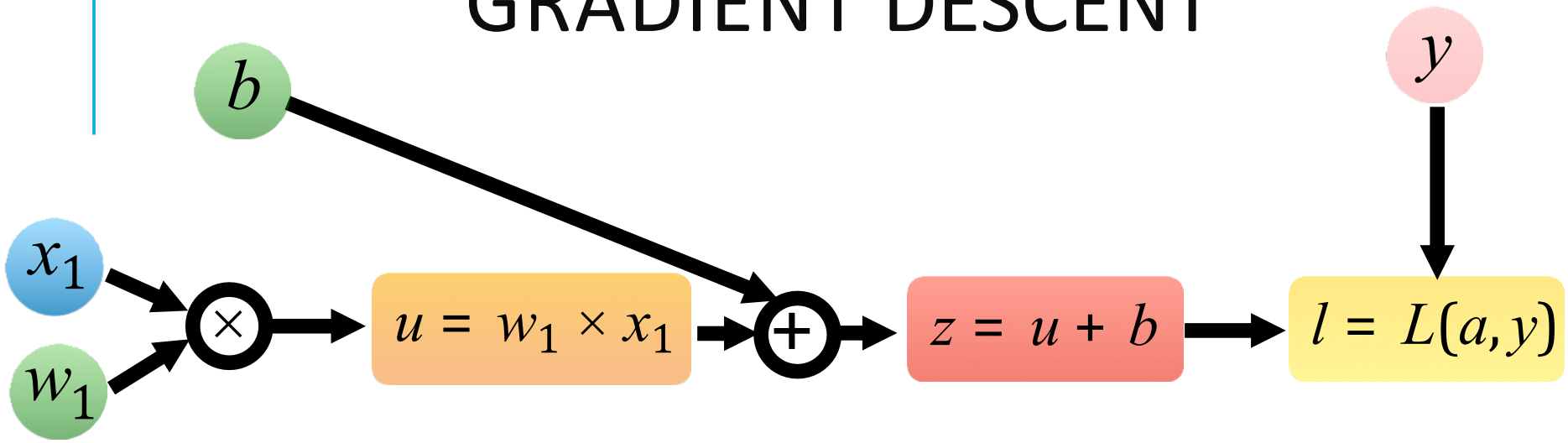


$$\frac{\partial L}{\partial w_1} = \frac{\partial u}{\partial w_1} \times \frac{\partial z}{\partial u} \times \frac{dL}{dz}$$



$$\frac{\partial L}{\partial b} = \frac{\partial z}{\partial b} \times \frac{dL}{dz}$$

MINIMIZING THE LOSS USING GRADIENT DESCENT



$$\frac{\partial L}{\partial w_1} = \frac{\partial u}{\partial w_1} \times \frac{\partial z}{\partial u} \times \frac{dL}{dz}$$

$$\frac{\partial L}{\partial b} = \frac{\partial z}{\partial b} \times \frac{dL}{dz}$$

$$\nabla \boldsymbol{L} = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial b} \end{bmatrix}$$

GRADIENT DESCENT

```python
class LinearRegression:
    def __init__(self):
        self.parameters = {}

    def forward_propagation(self, train_input): ···

    def cost_function(self, predictions, train_output): ···

    def backward_propagation(self, train_input, train_output, predictions):
        derivatives = {}
        df = (predictions-train_output)
        # dw= 2/n * mean of (predictions-actual) * input
        dw = 2 * np.mean(np.multiply(train_input, df))
        # db = 2/n * mean of (predictions-actual)
        db = 2 * np.mean(df)
        derivatives['dw'] = dw
        derivatives['db'] = db
        return derivatives

    def update_parameters(self, derivatives, learning_rate): ···

    def train(self, train_input, train_output, learning_rate, iters): ···
```

✓ 0.0s

UPDATE WEIGHTS AND BIAS

```
import matplotlib.pyplot as plt
import matplotlib.axes as ax
from matplotlib.animation import FuncAnimation
import numpy as np

class LinearRegression:
    def __init__(self):
        self.parameters = {}

    def forward_propagation(self, train_input): ...

    def cost_function(self, predictions, train_output): ...

    def backward_propagation(self, train_input, train_output, predictions): ...

    def update_parameters(self, derivatives, learning_rate):
        self.parameters['w'] = self.parameters['w'] - learning_rate * derivatives['dw']
        self.parameters['b'] = self.parameters['b'] - learning_rate * derivatives['db']

    def train(self, train_input, train_output, learning_rate, iters): ...
```

[5] ✓ 0.0s

EM 538-001: Practical Machine Learning for Enginering Analystics (Spring 2025)
Instructor: Fred Livingston (fjliving@ncsu.edu)

Load and Prepare Datasets

[17]
```python
import pandas as pd
chicago_taxi_dataset = pd.read_csv("https://download.mlcc.google.com/mledu-datasets/chicago_taxi_train.csv")
```
Python

```python
chicago_taxi_dataset.head()
```
Python

```python
import seaborn as sns
sns.scatterplot(data=chicago_taxi_dataset, y="FARE", x="TRIP_MILES")
```
Python

[20]
```python
from sklearn.model_selection import train_test_split

X = chicago_taxi_dataset["TRIP_MILES"]
y = chicago_taxi_dataset["FARE"]

X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=0.2,
                         shuffle=True, random_state=123)
```
Python

SGD HYPERPARAMTERS

SGD HYPERPARAMTERS

Hyperparameters are variables that control different aspects of training. Three common hyperparameters are:

- **Learning rate**
- **Batch size**
- **Epochs**

In contrast, **parameters** are the variables, like the weights and bias, that are part of the model itself. In other words, hyperparameters are values that you control; parameters are values that the model calculates during training.

LEARNING RATE

Learning rate is a floating point number you set that influences how quickly the model converges. If the learning rate is too low, the model can take a long time to converge. However, if the learning rate is too high, the model never converges, but instead bounces around the weights and bias that minimize the loss. The goal is to pick a learning rate that's not too high nor too low so that the model converges quickly.

The learning rate determines the magnitude of the changes to make to the weights and bias during each step of the gradient descent process. The model multiplies the gradient by the learning rate to determine the model's parameters (weight and bias values) for the next iteration. In the third step of **gradient descent**, the "small amount" to move in the direction of negative slope refers to the learning rate.

The difference between the old model parameters and the new model parameters is proportional to the slope of the loss function.

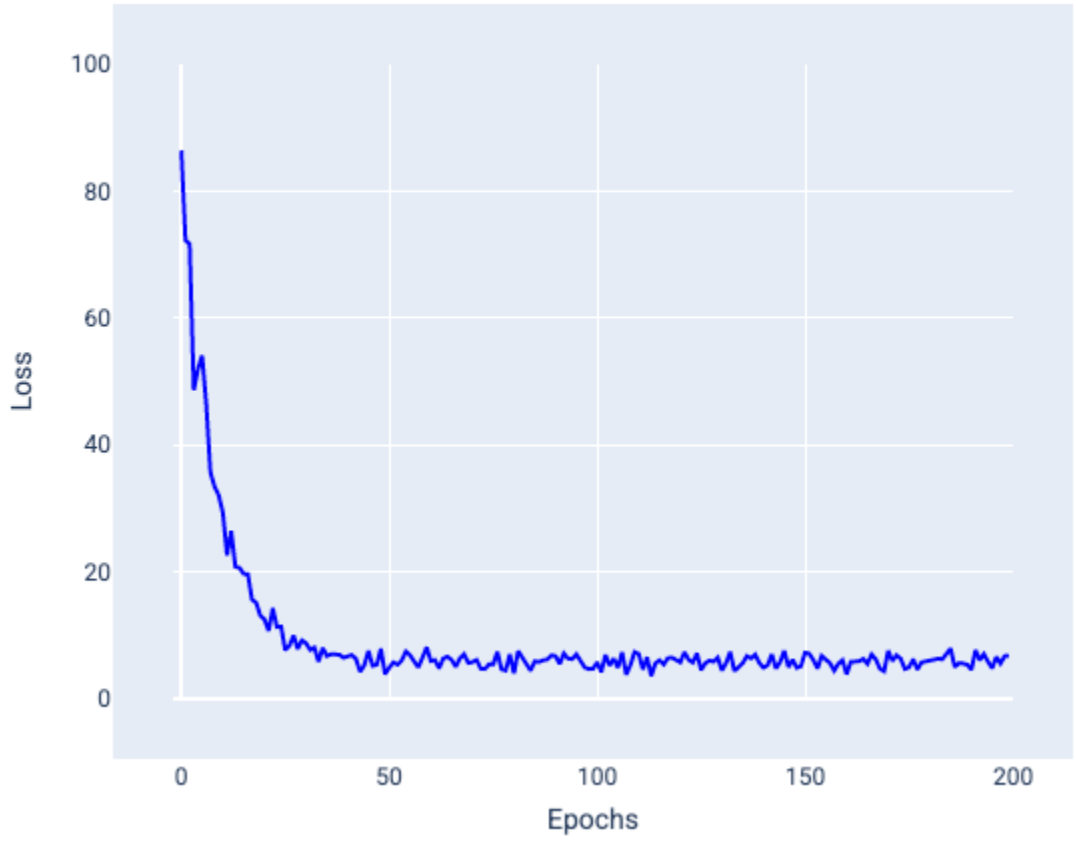
BATCH SIZE

Batch size is a hyperparameter that refers to the number of examples the model processes before updating its weights and bias. You might think that the model should calculate the loss for *every* example in the dataset before updating the weights and bias. However, when a dataset contains hundreds of thousands or even millions of examples, using the full batch isn't practical.

Two common techniques to get the right gradient on *average* without needing to look at every example in the dataset before updating the weights and bias are **stochastic gradient descent** and **mini-batch stochastic gradient descent**:

STOCHASTIC GRADIENT DESCENT (SGD)

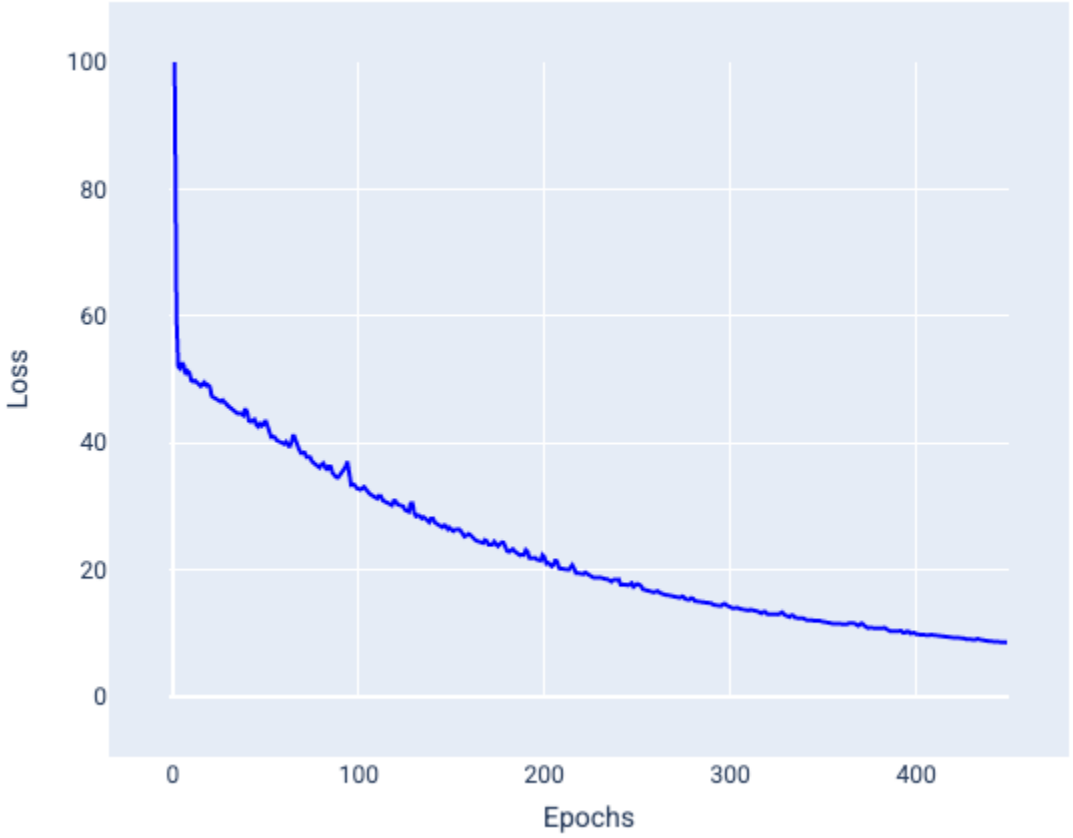
Stochastic gradient descent uses only a single example (a batch size of one) per iteration. Given enough iterations, SGD works but is very noisy. "Noise" refers to variations during training that cause the loss to increase rather than decrease during an iteration. The term "stochastic" indicates that the one example comprising each batch is chosen at random. Notice in the following image how loss slightly fluctuates as the model updates its weights and bias using SGD, which can lead to noise in the loss graph:



MINI-BATCH STOCHASTIC GRADIENT DESCENT

Mini-batch stochastic gradient descent is a compromise between full-batch and SGD.
For N number of data points, the batch size can be any number greater than 1 and less than N. The model chooses the examples included in each batch at random, averages their gradients, and then updates the weights and bias once per iteration.

Determining the number of examples for each batch depends on the dataset and the available compute resources. In general, small batch sizes behaves like SGD, and larger batch sizes behaves like full-batch gradient descent.



EPOCHS

During training, an **epoch** means that the model has processed every example in the training set *once*. For example, given a training set with 1,000 examples and a mini-batch size of 100 examples, it will take the model 10 **iterations** to complete one epoch.

Training typically requires many epochs. That is, the system needs to process every example in the training set multiple times.

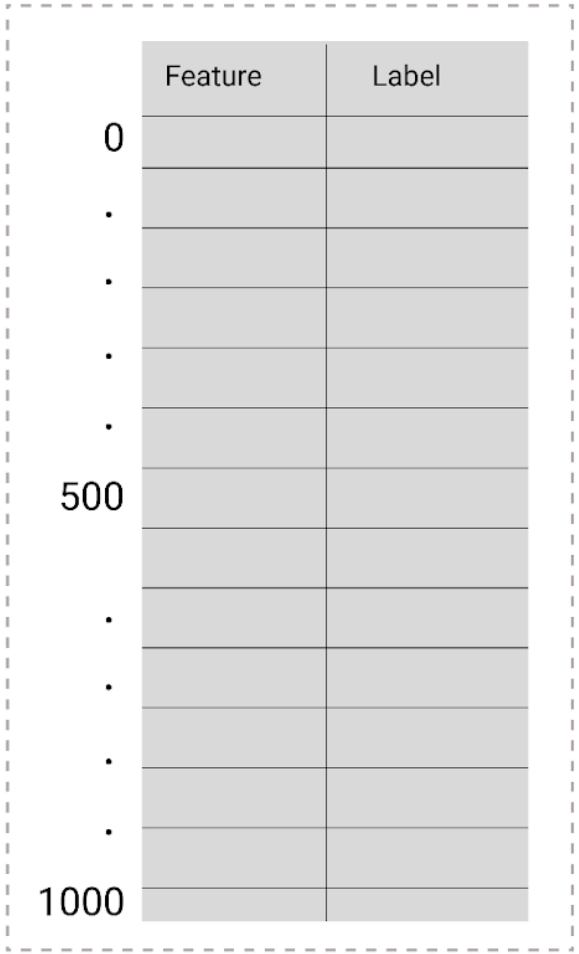
The number of epochs is a hyperparameter you set before the model begins training. In many cases, you'll need to experiment with how many epochs it takes for the model to converge. In general, more epochs produces a better model, but also takes more time to train.

FULL BATCH VS MINI BATCH

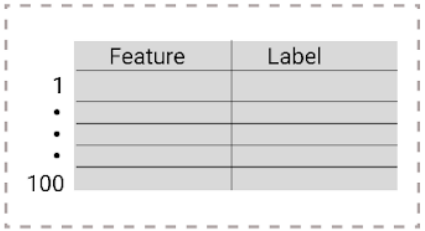
Batch type	When weights and bias updates occur
Full batch	After the model looks at all the examples in the dataset. For instance, if a dataset contains 1,000 examples and the model trains for 20 epochs, the model updates the weights and bias 20 times, once per epoch.
Stochastic gradient descent	After the model looks at a single example from the dataset. For instance, if a dataset contains 1,000 examples and trains for 20 epochs, the model updates the weights and bias 20,000 times.
Mini-batch stochastic gradient descent	After the model looks at the examples in each batch. For instance, if a dataset contains 1,000 examples, and the batch size is 100, and the model trains for 20 epochs, the model updates the weights and bias 200 times.

BATCH

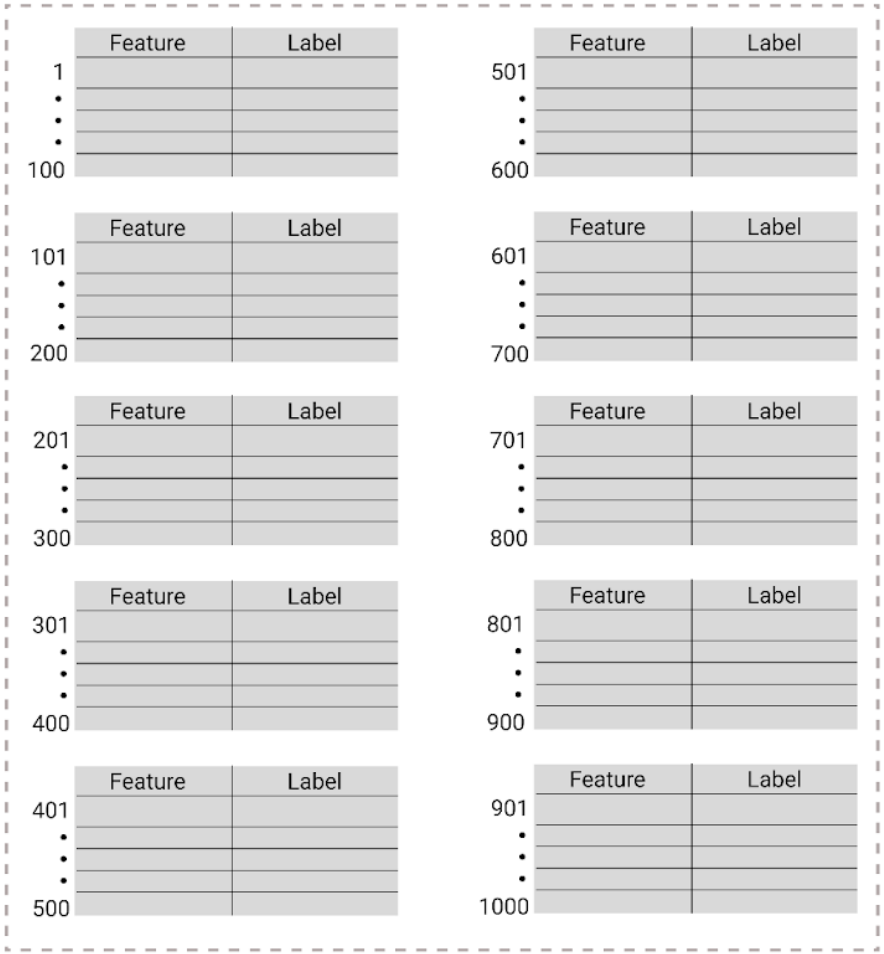
Full batch: an entire dataset



Single batch: a mini batch



One epoch: a full batch composed of ten mini batches



Python

```python
from sklearn.model_selection import train_test_split

X = chicago_taxi_dataset[["TRIP_MILES"]].values
y = chicago_taxi_dataset["FARE"].values

X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=0.2,
                         shuffle=True, random_state=123)
```

[4] Python

```python
from sklearn.linear_model import SGDRegressor

sgd_model = SGDRegressor(alpha=0.001, penalty='l2', max_iter=1000)
sgd_model.fit(X_train, y_train)
```

Python

```python
sgd_model.score(X_train, y_train)
```

Python

```python
print("Coefficients: \n", sgd_model.coef_)
```

Python

```python
print("Bias: \n", sgd_model.intercept_)
```

Python

Q/A?

