# Module 2

**Linear Regression with PyTorch**

**Linear Regression**

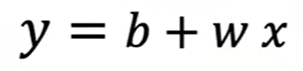
## 📌 Simple Linear Regression Prediction

This section introduces the principles of linear regression in one dimension and demonstrates how to build and use linear models in PyTorch to predict and output based on a given input.

By using functional and object-oriented approaches to define and use linear regression layers for prediction.

### 🔹 Concept of Linear Regression

Linear regression is a method used to model the relationship between an independent variable **x** (**feature**) and a dependent variable **y** (**target**). In the one-dimensional case, this relationship is represented as a straight line:



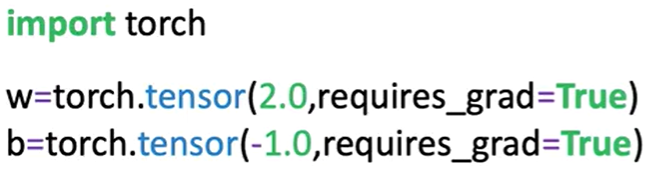
Where:

* is the predicted output (estimate).
* is the slope or weight,
* is the bias or intercept.

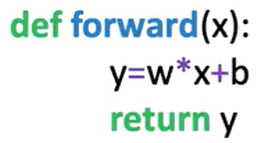
This equation defines the **linear model** that maps input values to estimated outputs. The goal of training is to determine optimal values for and .

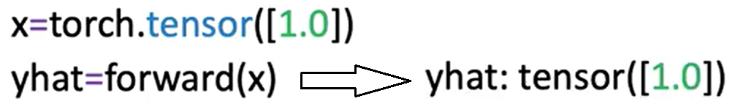
### 🔹 Prediction Using Tensors

To perform prediction manually using some arbitrary values, two tensors are created.

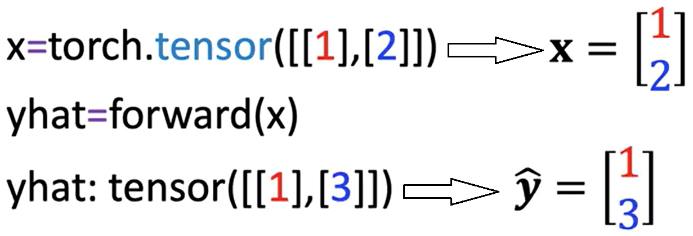
* One for the weight (slope).
* One for the bias (intercept).

Both tensors have **requires\_grad=True** set, indicating they are trainable parameters.

 A function **forward(x)** is defined to apply the linear equation.

Input values **x** are passed into this function, and the resulting tensor is the predicted output ​​.

Predictions can be made on a single input.

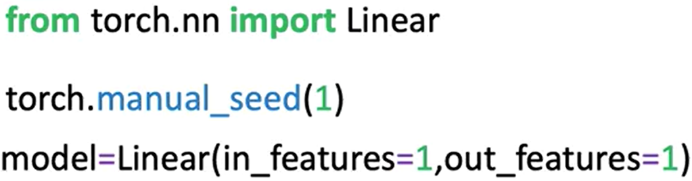


Or a tensor containing multiple rows. The linear function is applied row-wise. each row is treated as a sample.

### 🔹 Built-in Linear Model with nn.Linear

PyTorch includes a **built-in class** **nn.Linear**, which automatically handles weight and bias initialization and encapsulates the forward operation.

A linear model is created by calling:

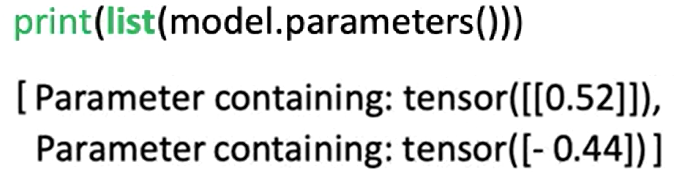
**nn.Linear(in\_features, out\_features)**.

Parameters:

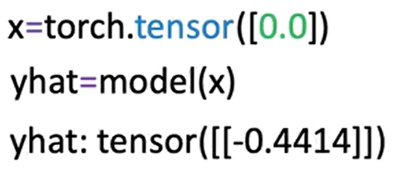
* **in\_features**: Number of input features (columns).
* **out\_features**: Number of output features.

After constructing the model:

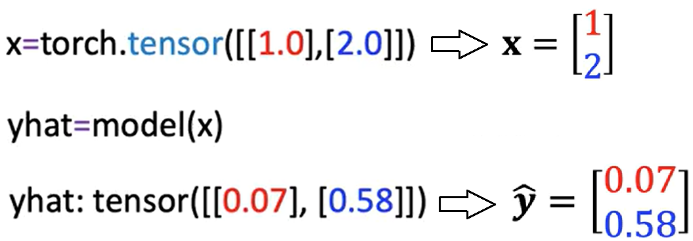
* The slope and bias are initialized randomly.
* Model parameters can be inspected using:

**model.parameters():** the first element is the slope, the second is the bias. **list()**function needs to be applied in order to get the output (because the method is lazily evaluated).

Or **model.state\_dict()**, this method is explained in detail later in this section.

To make predictions, pass the input tensor to the model directly.

There is no need to explicitly call a forward method; the object handles this internally.

Multiple input values are processed in batch format, where each row is treated as a separate input vector.

### 🔹 Building a Custom Linear Module

A custom module allows us to wrap multiple objects to make more complex workflows.

It can be defined by subclassing **nn.Module**.

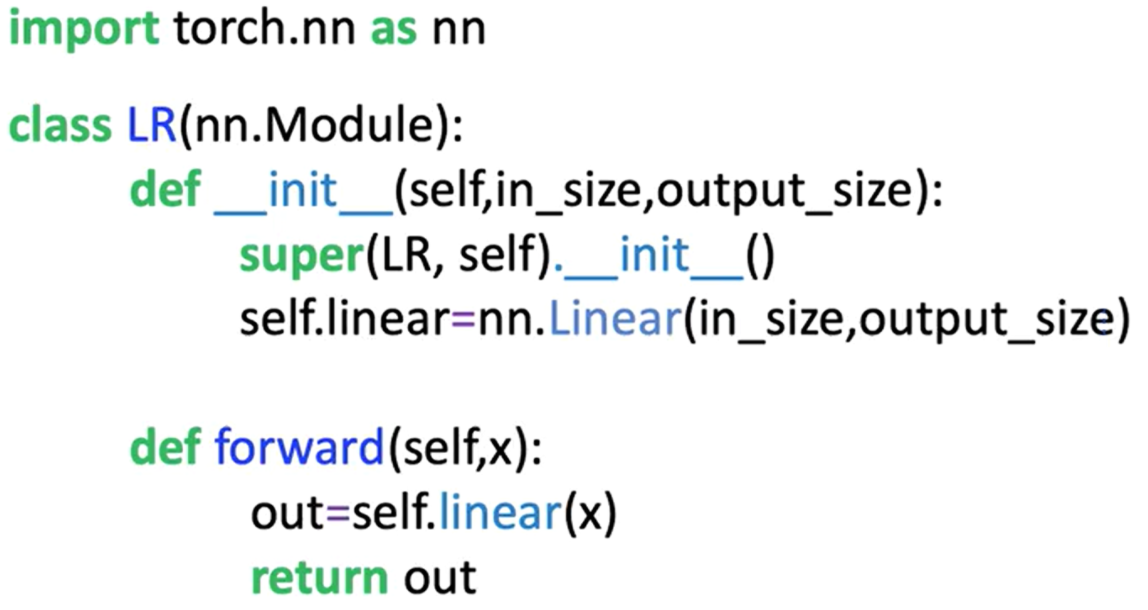
🔸 **Custom Class Structure::**

The class is a child of **nn.Module**, inheriting its methods and behavior.

In the constructor:

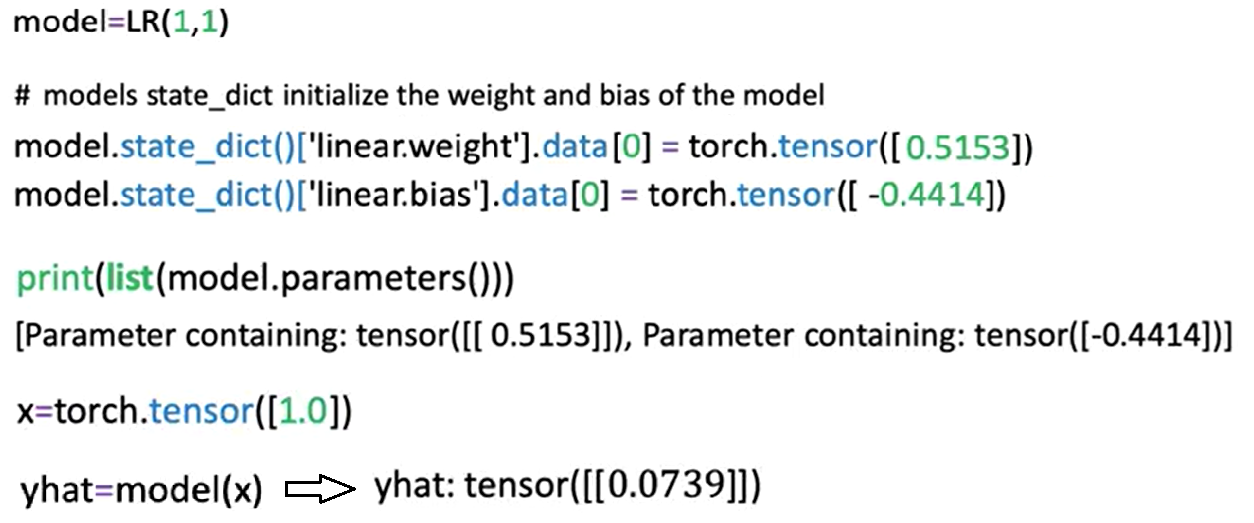
* The base class constructor is initialized using **super()**.
* In the object constructor, the argument are the **size of the input** (x, **in\_size**) and **output** (y, **output\_size**).
* A linear layer is created using **nn.Linear(input\_size, output\_size)** and stored as **self.linear**.

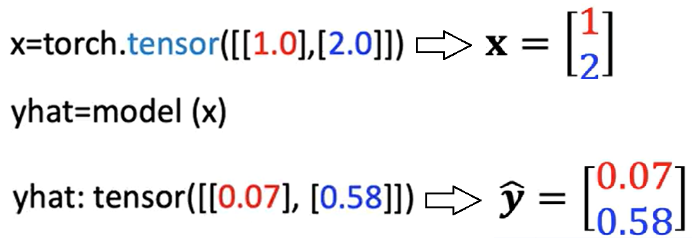
A **forward** method is defined to apply the linear transformation.



Once the custom class is defined:

* A model object is created by passing input/output size arguments.
* Model parameters are available through inherited methods:
  + **model.state\_dict()** is used to initialize the weight and bias of the model.
  + **model.parameters()** for inspecting layer-specific weights and biases.
* Predictions are made by calling the model with the input tensor, the method **forward** do not have to be called explicitly.



The initialized custom model can be used to make multiple predictions as seen before; the object maps every row in the tensor.

### 🔹 Using state\_dict for Parameter Access

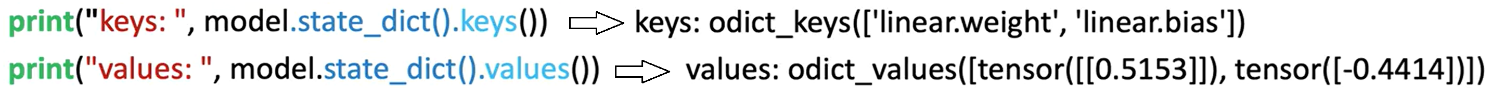
The **state\_dict()** method returns a Python dictionary containing all the learnable parameters of the model, as models get more complex this method becomes more useful.

One Function is to map the relationship of the linear layers to its parameters.

* Each key corresponds to a named parameter (e.g., **linear.weight**, **linear.bias**).
* Each value is the tensor containing the current value of the parameter.

This dictionary is useful for:

* Inspecting parameter values
* Debugging model initialization
* Saving and loading model weights in more advanced use cases



### ✅ Takeaways

✅Linear regression models define a simple mapping between input and output using a linear equation.

✅In PyTorch, models can be implemented manually using tensors or more efficiently using **nn.Linear**.

✅The **nn.Linear** class handles weight and bias internally and can be used directly for predictions.

✅Custom modules can be built by subclassing **nn.Module** and defining a forward method.

✅Once constructed, model objects behave like callable functions and do not require explicit calls to the forward method.

✅Model parameters and their initialization can be accessed using **.parameters()** and **.state\_dict()**.

✅These foundational practices set the stage for training models and scaling to more complex architectures.

## 📌 Linear Regression Training

This section introduces the training process for linear regression in PyTorch. It defines what constitutes a dataset, explains the noise assumption behind regression models, and presents the objective of learning model parameters by minimizing the mean squared error.

The focus is on how a model learns from examples by fitting a line that best captures the relationship between the input and output variables.

### 🔹 Defining the Dataset and Learning Objective

Linear regression aims to model the relationship between a feature (independent variable x) and a target (dependent variable y).

The goal of training is to **learn the best** values for the model **parameters**—slope and bias—that define a linear function capable of estimating y given x.

* A dataset is composed of **N** pairs of values: (x₁, y₁), (x₂, y₂), …, (xₙ, yₙ).
* Each xᵢ and yᵢ pair is related through a linear function plus a small amount of random noise.
* This process is known as **supervised learning**, where known input-output pairs are used to fit a model.

Examples of real-world applications of simple linear regression include:

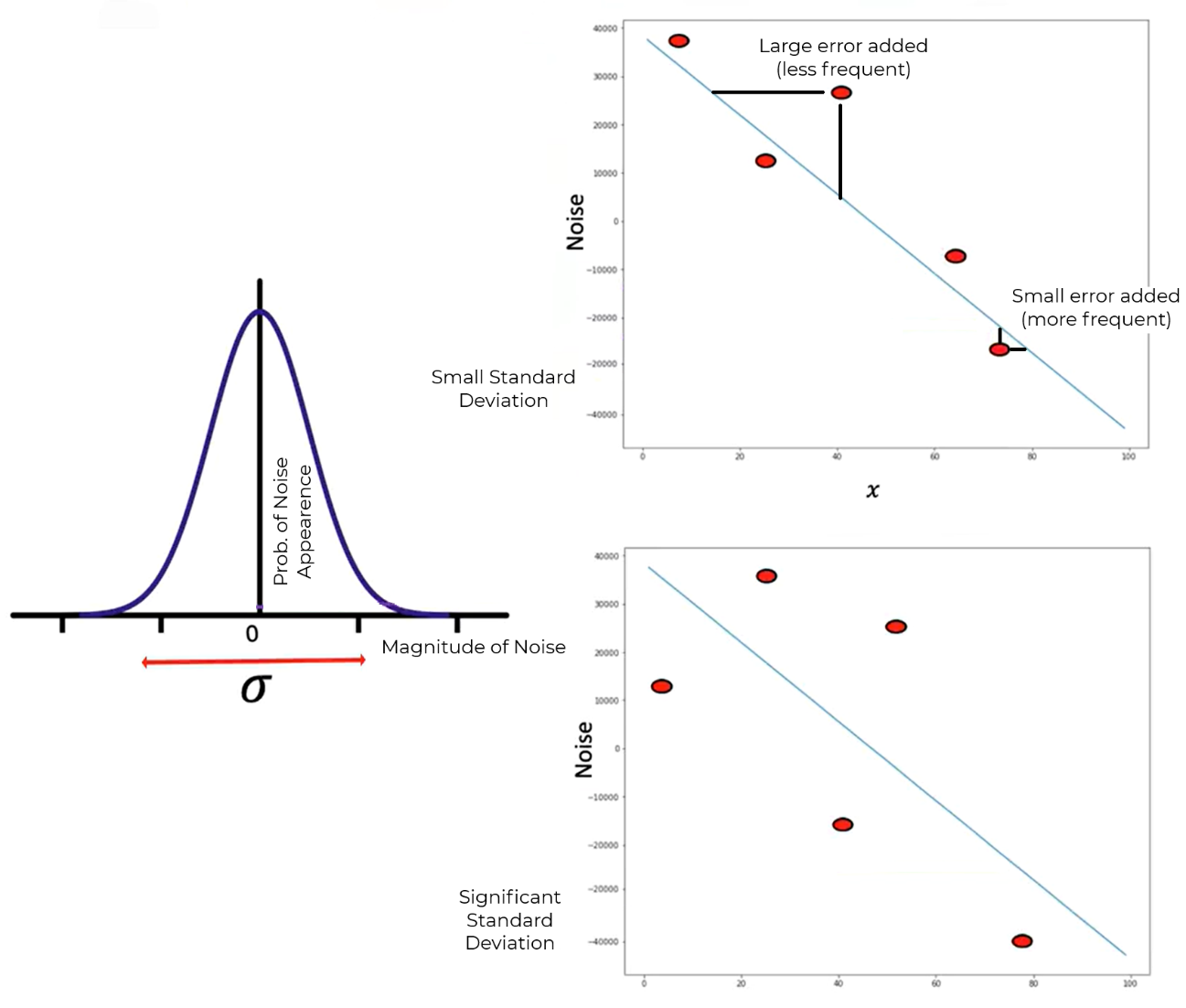
* Predicting house prices based on size.
* Estimating stock prices from interest rates.
* Modeling fuel efficiency as a function of horsepower.

In all cases, x is the feature and y is the predicted output.

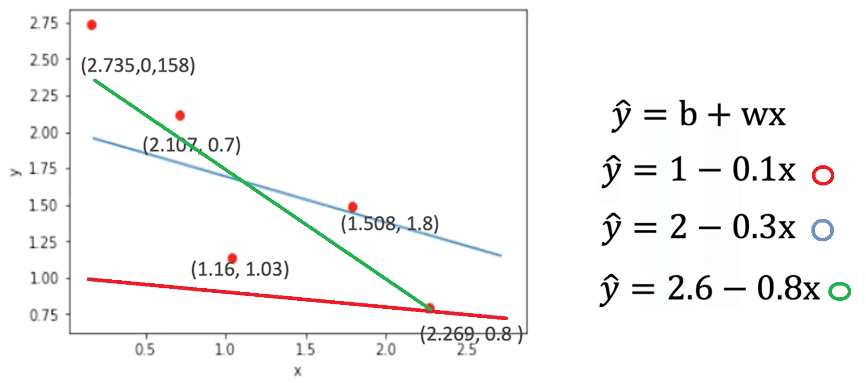
### 🔹 The Noise Assumption in Regression

Even when the relationship between variables is approximately linear, real-world data is not perfectly aligned on a straight line. This is because of **random noise**, which reflects measurement errors or unmodeled effects.

* The noise is assumed to be **Gaussian-distributed** with a mean of zero.
* The horizontal axis of the Gaussian curve represents the magnitude of the added noise.
* The vertical axis represents the probability of observing that value.
* Most of the noise values are close to zero, with only occasional large deviations.
* The more significant the standard deviation or, the more disperse the distribution is, the more the samples deviate from the line.



### 🔹 The Goal of Training

The objective of training a linear regression model is to **find the line that best fits the dataset**.

* In practice, several candidate lines may be drawn through the data points.
* Visually inspecting lines can suggest better or worse fits, but a mathematical method is needed for objective evaluation.

To formalize the training process, a **cost function** is introduced:

* The cost function is the **Mean Squared Error (MSE)**:

Where:

* is the predicted output
* is the actual output
* is the number of data points
* The MSE depends on the **slope and bias** of the model.
* Different parameter values lead to different MSE values.
* The best-fitting line is the one that **minimizes** this cost function.

Minimizing the mean squared error ensures that, on average, the model's predictions are as close as possible to the true values of y.

### ✅ Takeaways

✅ A linear regression model learns to map x to y by fitting a line to a set of input-output pairs.

✅ Datasets consist of ordered pairs of numeric values, where each pair defines a single example.

✅ Real data contains noise, modeled as Gaussian-distributed random variation added to each observation.

✅ The goal of training is to identify model parameters (slope and bias) that minimize the prediction error.

✅ The prediction error is quantified using the **mean squared error**, which forms the basis of the cost function used during optimization.

## 📌 Loss in Linear Regression

This section introduces the concept of **loss** as a fundamental building block in model training.

Loss quantifies the difference between the model’s prediction and the true value, and serves as the foundation for the **cost function**, which is used to guide parameter optimization.

### 🔹 Role of Loss in Model Training

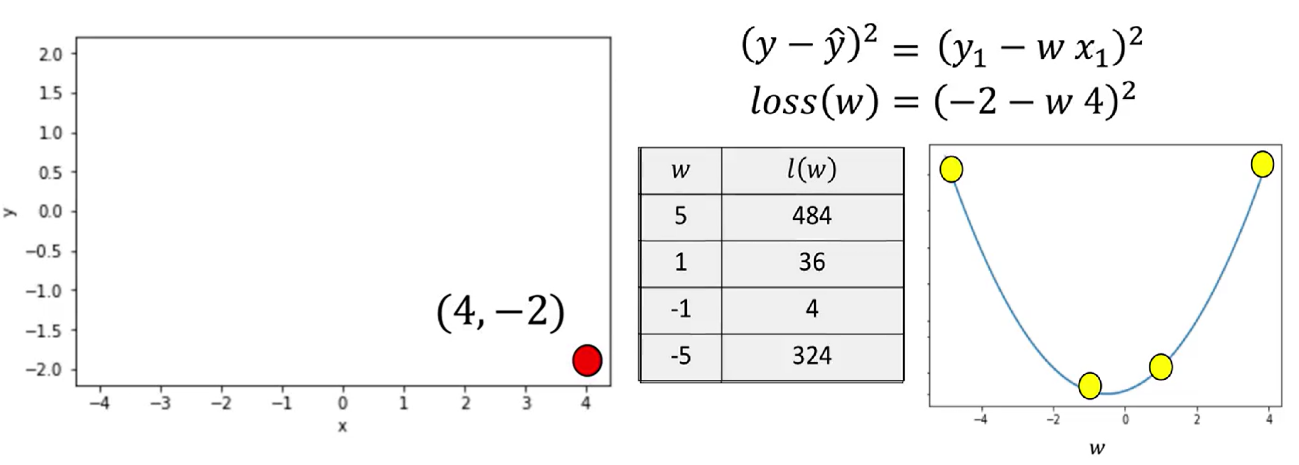
The training objective in linear regression is to learn the best model parameters (slope and bias) that result in accurate predictions for the dependent variable y given input x.

* For any input-output pair (x, y), the model produces an estimate ​ using the linear function.
* Loss measures how far this prediction ​ ​ is from the actual value y.
* The **smaller** the loss, the **better** the prediction.

### 🔹 Defining Loss for a Single Sample

To understand how the model adjusts parameters, a simplified example is used with only one sample:

* Suppose x = -2 and y = 4.
* A model prediction is computed using a candidate slope value .
* The prediction error is calculated as:



* Selecting a **slope of 5**, the line is far from the data point. In the data space, the value of the loss function is relatively large,
* Selecting a **slope of 1**, the value for the loss is near the minimum of the parameter space.
* Selecting a **slope of -1**, the result gets much closer to the minimum of the loss function, closer to the loss curve.
* A **slope of -5** the line is much farther away from the data point.

The squared difference captures how far the prediction is from the actual value and ensures that positive and negative errors do not cancel each other out.

* Since the true values of x and y are fixed during training, the loss becomes a function of the model parameter (slope).
* The loss function is also called the **criterion function**.
* It outputs a numerical value that reflects how good or bad a model’s prediction is.
* When visualized, the loss function appears as a **concave bowl**, or **parabola**, in the parameter space.

This shape has key properties:

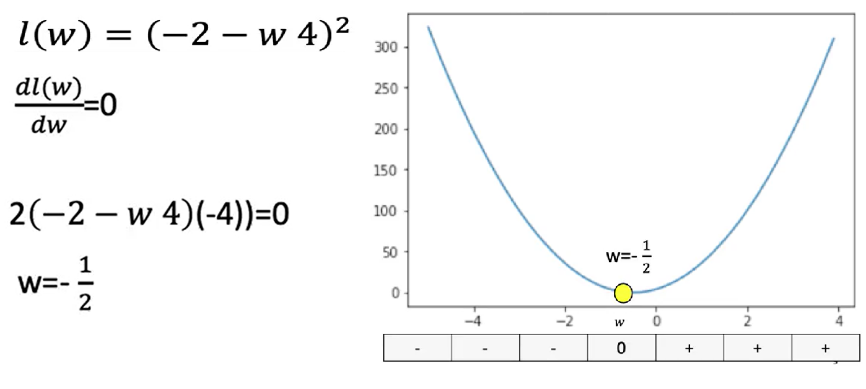
* **Minimum point** corresponds to the best slope value.
* **Left of the minimum**: the derivative (slope of the loss curve) is negative.
* **Right of the minimum**: the derivative is positive.
* **At the minimum**: the derivative is zero.

This behavior allows optimization techniques to search for the best parameter value by analyzing the derivative of the loss function.

### 🔹 Systematic Minimization of Loss

Instead of testing random parameter values, a **systematic method** is preferred to minimize the loss:

* Visualizing loss for different slope values shows:
  + Poor parameter choices result in high loss.
  + Optimal parameter values bring the predicted line closer to the actual data point, reducing loss.
* The **best slope** can be found algebraically by:
  + Taking the **derivative** of the loss function with respect to the slope.
  + Setting the derivative equal to zero.
  + Solving for the slope value.

This technique finds the slope that minimizes loss for the given data point.

We can actually find the best value for the slope by setting the derivative = 0.

⚠️ However, this exact method is impractical for more complex models (e.g., deep learning), where explicit derivatives are difficult or impossible to compute algebraically.

Still, the insight that **derivatives point in the direction of decreasing loss** is crucial for gradient-based optimization.

### ✅ Takeaways

✅ Loss is a numeric measure of how well a model prediction matches the actual target value.

✅ For linear regression, loss is commonly defined as the **squared difference** between prediction and target.

✅ The loss function is treated as a function of the model parameters (e.g., slope).

✅ The objective of training is to **minimize the loss** to improve prediction accuracy.

✅ The loss function has a clear geometric interpretation: its **minimum** represents the best-fitting model.

✅ Derivatives indicate how to update parameters and are foundational to gradient descent and training in neural networks.