# Module 2

**Linear Regression with PyTorch**

**M. 2 – Section 1**

**Linear Regression Prediction with PyTorch**

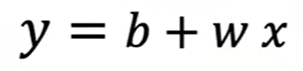
## 📌 Simple Linear Regression Prediction

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:



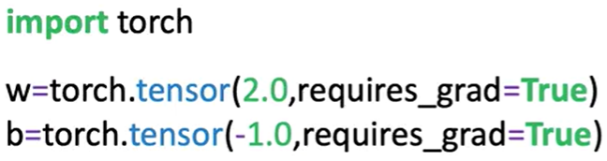
Where:

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

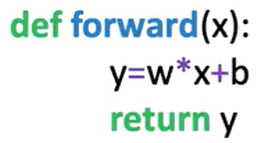
This equation defines the **linear model** that maps input values to estimated outputs. The goal of training is to determine optimal values for and .

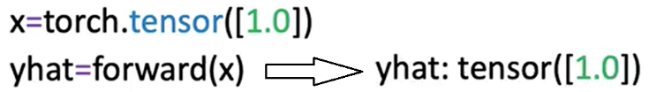
### 🔹 Prediction Using Tensors

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

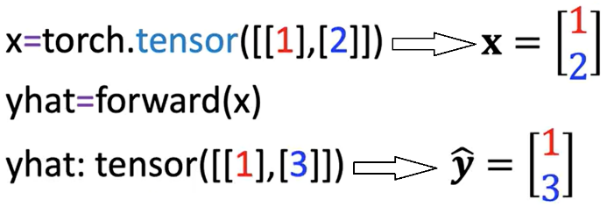
* One for the weight (slope).
* One for the bias (intercept).

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:

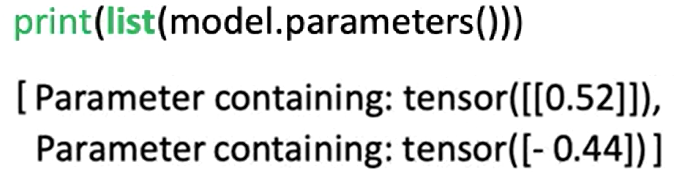
**nn.Linear(in\_features, out\_features)**.

Parameters:

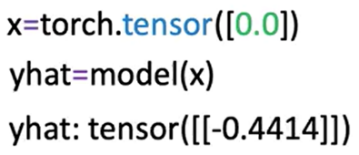
* **in\_features**: Number of input features (columns).
* **out\_features**: Number of output features.

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**.

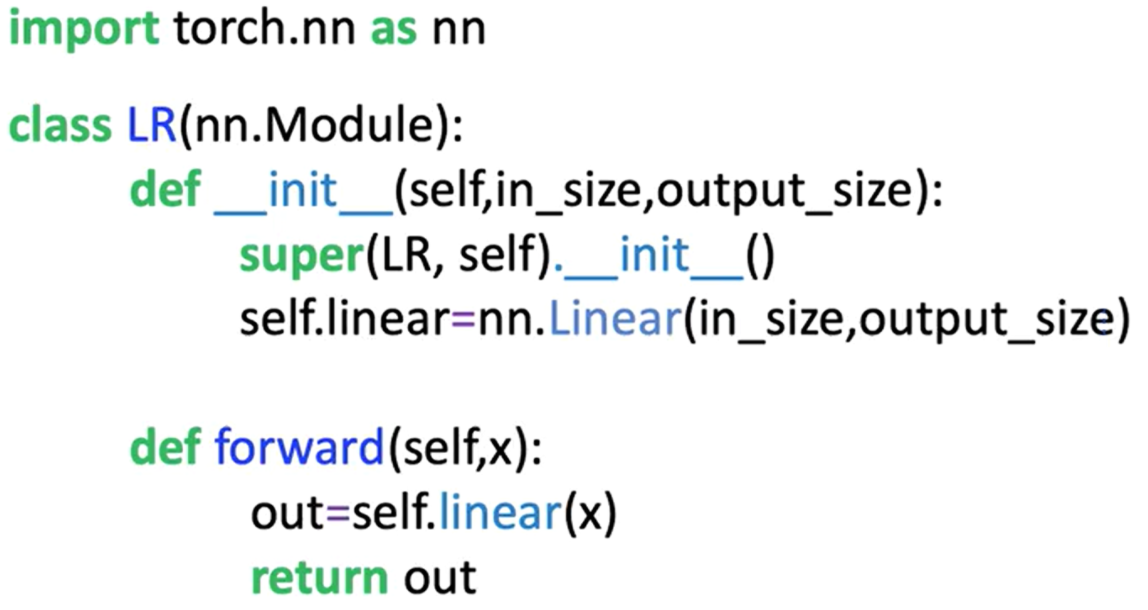
🔸 **Custom Class Structure:**

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

In the constructor:

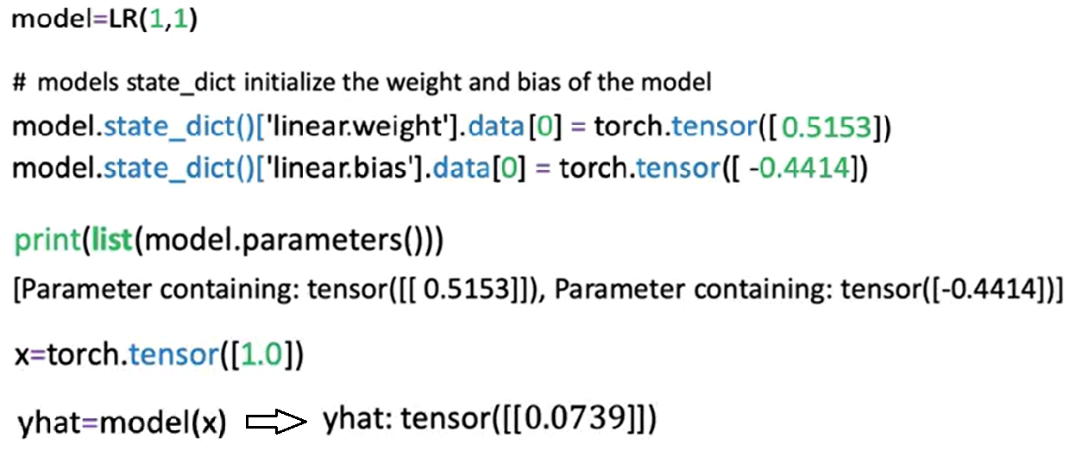
* The base class constructor is initialized using **super()**.
* In the object constructor, the argument are the **size of the input** (x, **in\_size**) and **output** (y, **output\_size**).
* A linear layer is created using **nn.Linear(input\_size, output\_size)** and stored as **self.linear**.

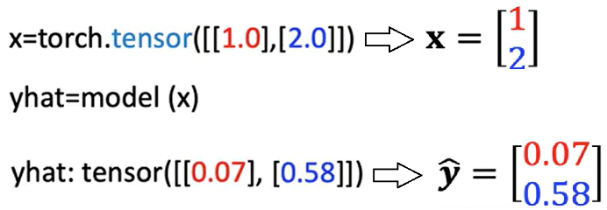
A **forward** method is defined to apply the linear transformation.



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

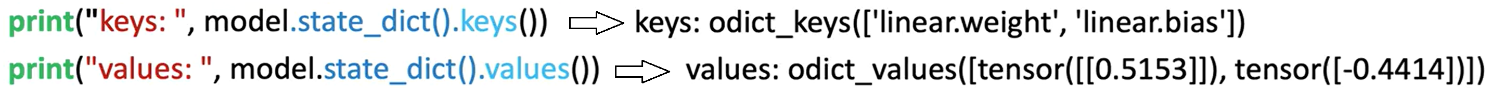
The **state\_dict()** method returns a Python dictionary containing all the learnable parameters of the model, as models get more complex this method becomes more useful.

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.

# M. 2 – Section 2

**Linear Regression**

## 📌 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:

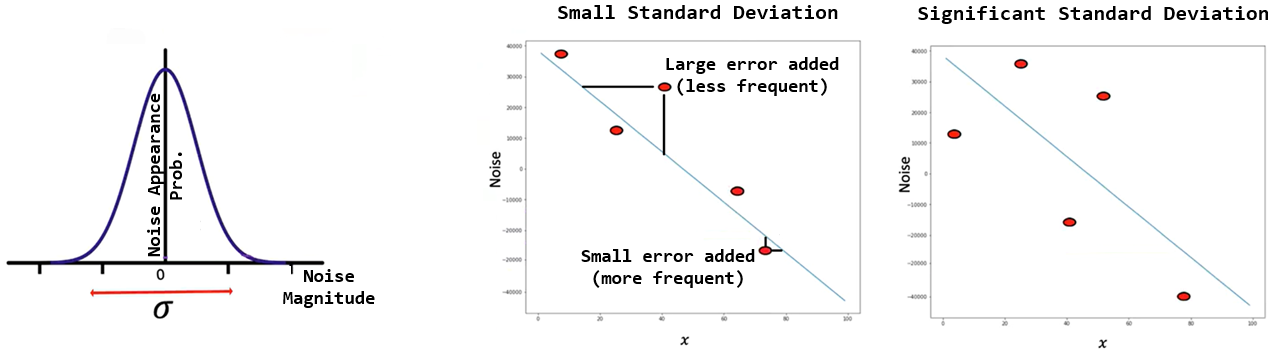
* Predicting house prices based on size.
* Estimating stock prices from interest rates.
* Modeling fuel efficiency as a function of horsepower.

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

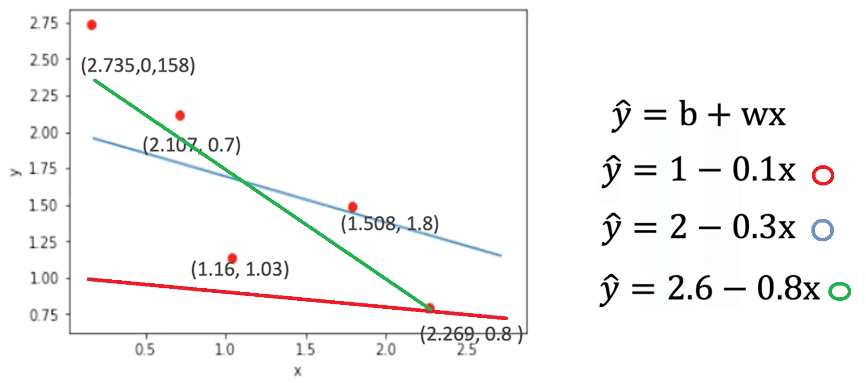
### 🔹 The Noise Assumption in Regression

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

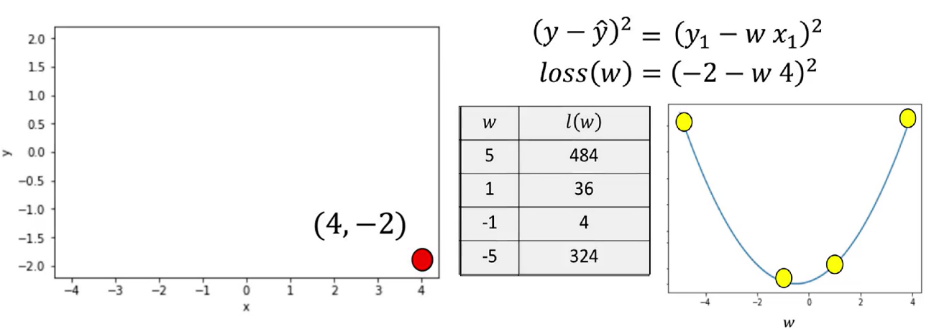
The training objective in linear regression is to learn the best model parameters (slope and bias) that result in accurate predictions for the dependent variable y given input x.

* 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.

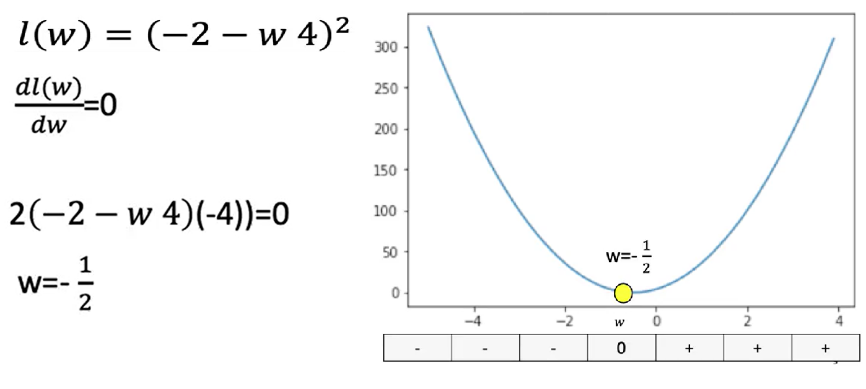
This behavior allows optimization techniques to search for the best parameter value by analyzing the derivative of the loss function.

### 🔹 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. **derivatives points in the direction of decreasing loss**.
  + 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.

### ✅ 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.

## 📌 Gradient Descent and Cost

This section introduces **gradient descent**, the fundamental optimization technique used to minimize loss functions in machine learning.

It explains how gradient descent works in one dimension and addresses challenges like learning rate selection and stopping criteria.

The process is applied to adjust model parameters iteratively to minimize prediction errors.

### 🔹 What is Gradient Descent?

Gradient descent is an **iterative algorithm** used to find the minimum of a function by adjusting its parameters in the direction of the steepest descent.

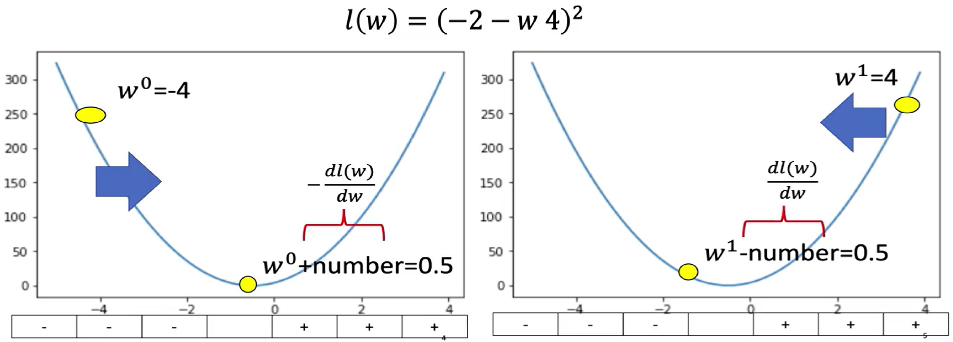
* The algorithm begins with a **random initial guess** for a model parameter (e.g., the slope in linear regression).
* It evaluates the **gradient** (i.e., the derivative) of the loss function with respect to that parameter.
* The parameter is then **updated** by subtracting a value **proportional to the derivative**:

Where:

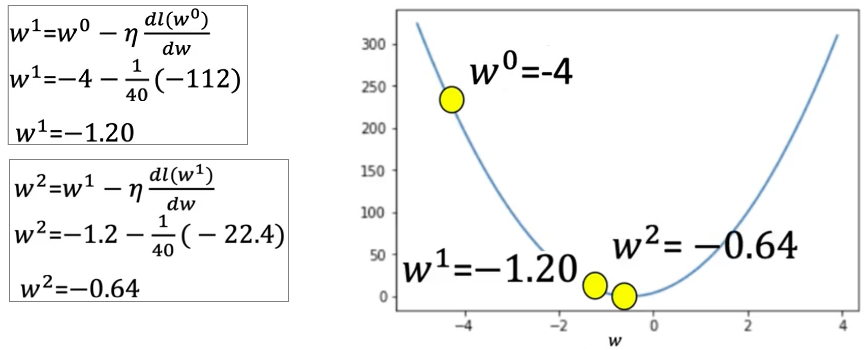
* is the current parameter value.
* is the learning rate.
* is the derivate of the loss function.

The sign of the gradient determines the direction of the parameter update:

* If the derivative is negative, the parameter increases.
* If the derivative is positive, the parameter decreases.
* The gradient points away from the minimum, so its negative is used to move toward the minimum.



### 🔹 Gradient Descent in Practice

To compute gradient descent:

* The process is started by selecting a random guess, and choosing the learning rate.
* Then the derivate at that point is calculated.
* Finally, the parameter is updated.

After the update, the loss decreases.

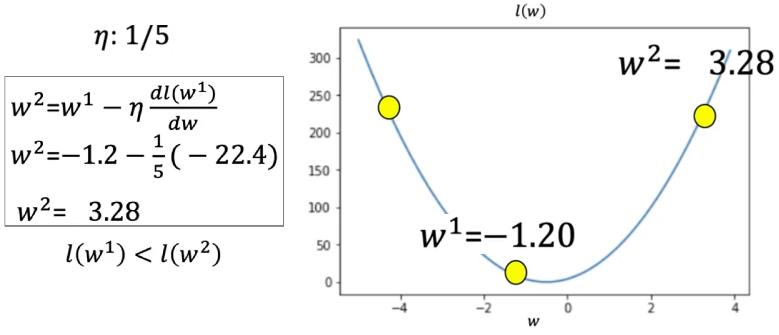
* The next iteration continues using the updated value.
* The parameter is updated again and the loss continues to decrease.

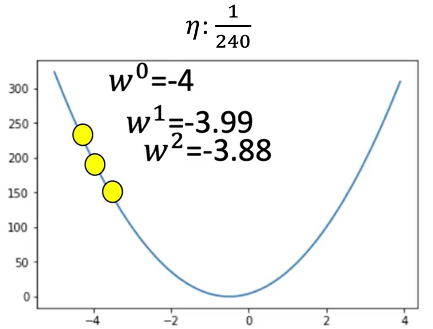
This process continues until a stop condition is reached.

### 🔹 Problems with Learning Rate

The **learning rate ()** controls the step size for parameter updates. Choosing an inappropriate value leads to problems:

**1. Learning Rate Too Large**

* The algorithm **overshoots** the minimum and the loss increases.
* This causes the algorithm to diverge or oscillate.

**2. Learning Rate Too Small**

* The algorithm makes **very slow progress**.
* This leads to excessive computation time and inefficient convergence.

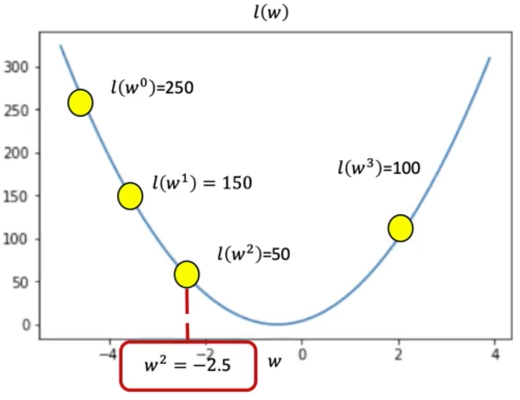
The learning rate must be chosen carefully to balance stability and speed of convergence.

### 🔹 When to Stop Gradient Descent

Several strategies are commonly used to decide **when to stop** the gradient descent process:

**1. Fixed Number of Iterations**

* Run gradient descent for a predefined number of iterations (e.g., 3 iterations).
* This is simple but may **miss the true minimum**.

For example:

* Start with a loss of 250.
* After several iterations, the loss values may be: 150 → 80 → 50 → 100.
* The stopping point is when the loss **increases** to 100.
* The best parameter is the one corresponding to a loss of 50.

**2. Monitor Loss Values**

* Continue updating until the loss stops decreasing.
* Maintain a table of loss values across iterations:
  + If the loss starts increasing or stagnates, the process stops.
  + The best parameter value is chosen from the iteration with the **lowest loss**.

### ✅ Takeaways

✅ Gradient descent is a key method for minimizing loss and learning model parameters.

✅ It updates parameters using the **negative derivative** of the loss function.

✅ The **learning rate** controls how far the parameter moves with each update:

* Too high: unstable updates, missed minimum.
* Too low: slow convergence.

✅ The update rule is repeated iteratively to approach the optimal value.

✅ The process stops either after a fixed number of iterations or when the loss stops improving.

✅ Understanding gradient descent is essential for training linear and deep learning models.

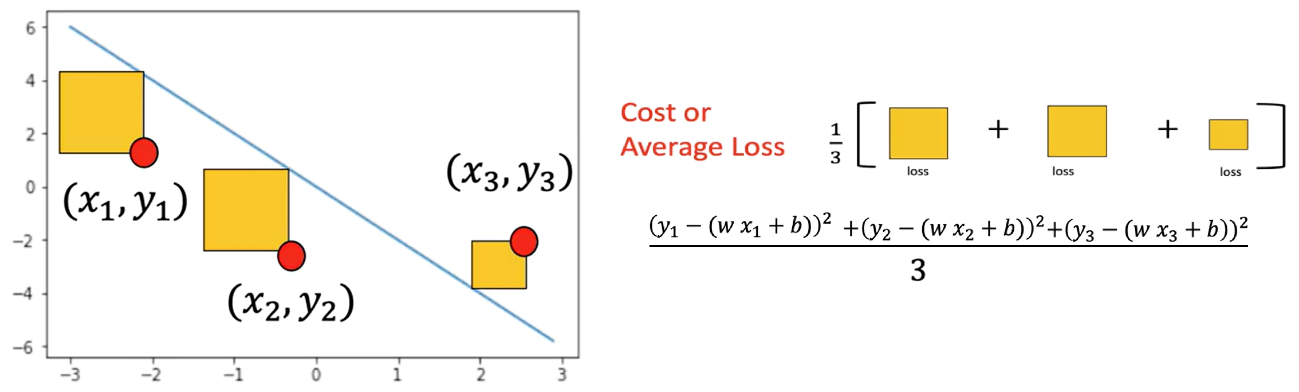
## 📌 Cost

The cost function allows the model to evaluate how well it fits **all training samples**, not just one, and serves as the basis for **batch gradient descent** optimization.

### 🔹 From Loss to Cost

The **loss function** measures prediction error for a single training example. To train a model on multiple examples, a **cost function** is defined by **summing or averaging** the losses across all samples.

* Each prediction error is squared to form a small square, visually representing the magnitude of error.
* The **cost** is the **sum of all these squared errors**, or their **average**:



Following PyTorch’s convention, the cost function is denoted as **L** (for loss), even when referring to the total over a batch.

### 🔹 Cost Function as a Function of Parameters

The cost function depends on model parameters, especially the **slope (weight)** and **bias**:

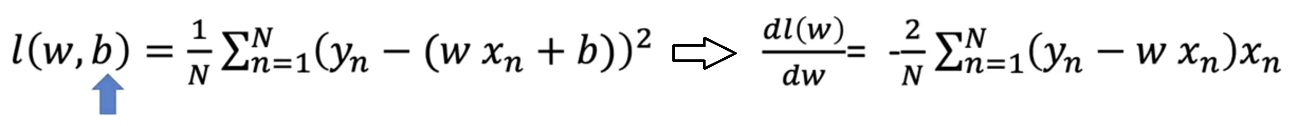
* The slope determines the steepness of the predicted line. It controls the relationship between x and y.
* The bias controls the horizontal offset.

The **goal** is to find the values of slope and bias that **minimize the cost**, which means achieving the best possible fit for the data.

### 🔹 Applying Gradient Descent to Cost

The same **gradient descent** technique used for single-sample loss is applied to the **cost function** to optimize parameters across **multiple samples**.

* For multiple data points, the derivative of the cost with respect to the slope is the **sum of the derivatives** for each sample.



* The process involves:
  1. Computing the cost using all samples.
  2. Calculating the gradient (derivative) of the cost.
  3. Updating parameters using the gradient.

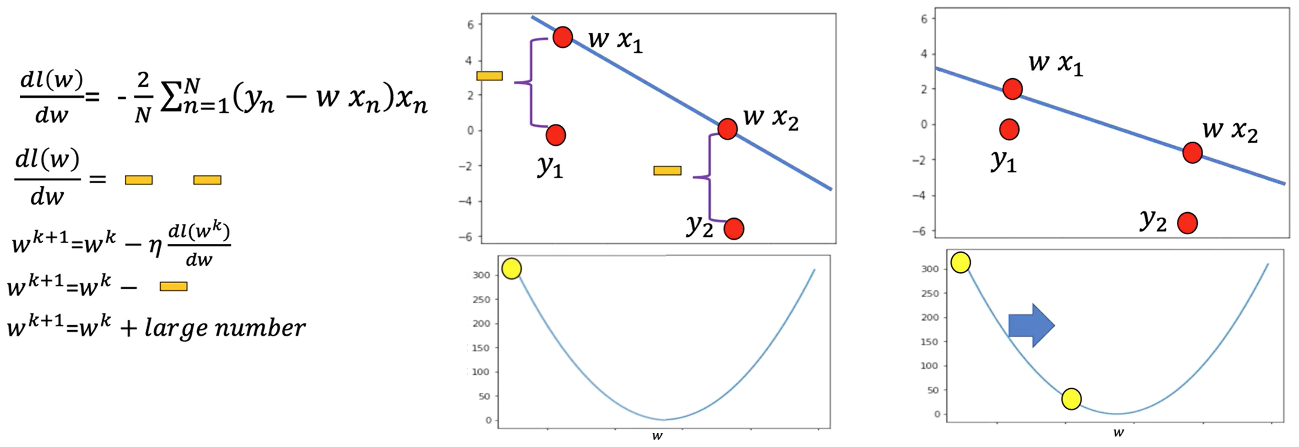
This process is repeated iteratively to improve the model fit.

Several scenarios illustrate how sample distribution affects the derivative of gradient descent with just the slope:

🔸 **Case 1 - All Samples on the Same Side:**

If both data points lie **below** the current prediction line:

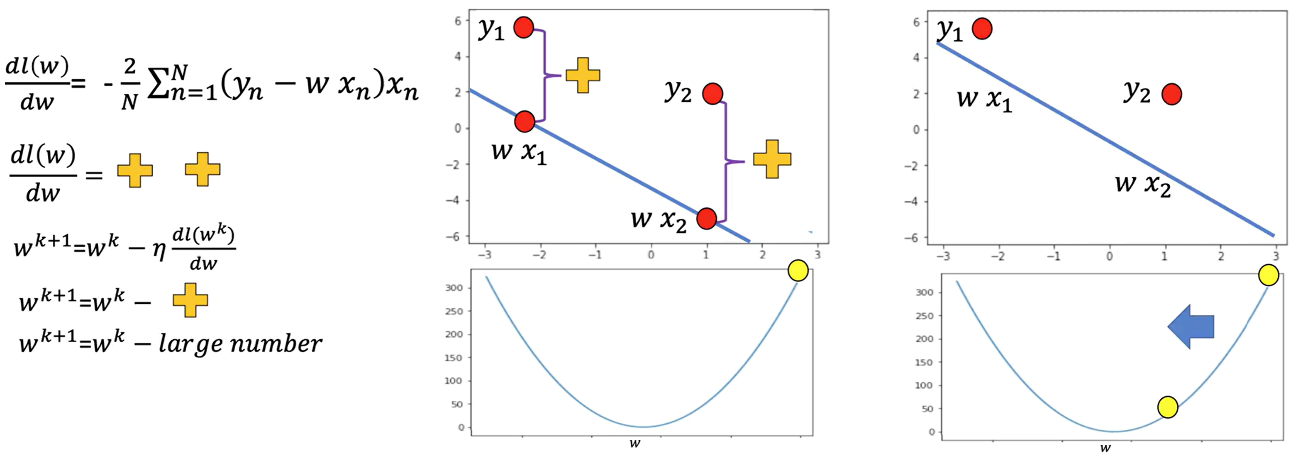
* The derivative is **strongly negative**.
* The update step adds a **large positive value** to the parameter.
* The prediction line moves **closer to the data**.



🔸 **Case 2 - All Samples on the Opposite Side:**

If both points lie **above** the prediction line:

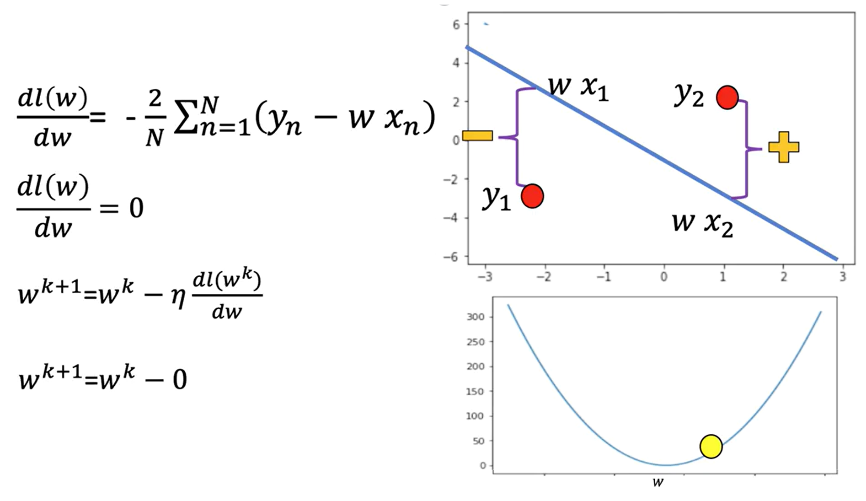
* The derivative is **strongly positive**.
* The update subtracts a **large value**, moving the line down.



🔸 **Case 3 - Mixed Sides:**

If one sample is **above** and the other **below** the line:

* The positive and negative derivatives **cancel each other out**.
* The resulting derivative is **near zero**.
* The update step is **very small**, and the line changes little.



### 🔹 Batch Gradient Descent

When the **entire dataset** is used to calculate the cost and gradient at each step, the method is called **Batch Gradient Descent**:

* The "batch" refers to the full training set.
* All samples are used to:
  + Compute the total cost.
  + Calculate the overall derivative.
  + Perform the parameter update.

For example, if the batch size is 3:

* The model uses all 3 data points to compute the cost and update.
* This ensures stability and directionally correct updates.

### ✅ Takeaways

✅ The cost function aggregates prediction errors across all training samples.

✅ It serves as the objective function for training linear models.

✅ Gradient descent applied to the cost function is **batch gradient descent**.

✅ Sample distribution affects the gradient and update size.

✅ Proper gradient accumulation across the batch ensures consistent parameter updates.

# M. 2 – Section 3

**PyTorch Slope**

## 📌 Linear Regression in PyTorch

This section introduces a hands-on implementation of gradient descent in PyTorch **using only the slope (no bias)** to develop a deeper conceptual understanding of model optimization.

The training process is built step by step using raw PyTorch operations without relying on higher-level modules or abstractions.

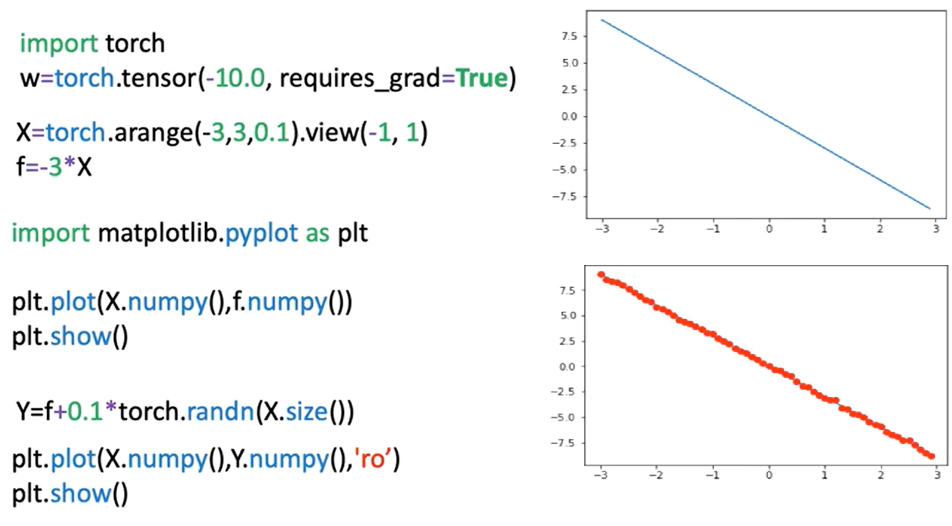
### 🔹 Gradient Descent with PyTorch Tensors

To perform gradient descent manually, a PyTorch tensor is used to represent the model parameter — the slope of the line.

The tensor is initialized with **requires\_grad=True** to allow PyTorch to automatically compute gradients during backpropagation.

The procedure involves the following:

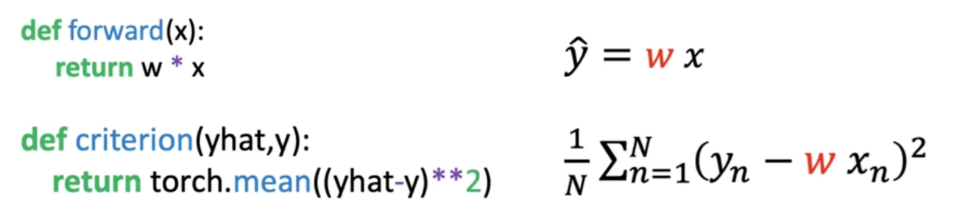
* Creating a tensor for the slope parameter w
* Generating sample X values and corresponding Y values using a known slope, **view()** function is used to add an additional dimension.
* Adding random noise to simulate real-world data variability
* Visualizing the initial data and the true line using matplotlib



### 🔹 Loss Calculation and Optimization Process

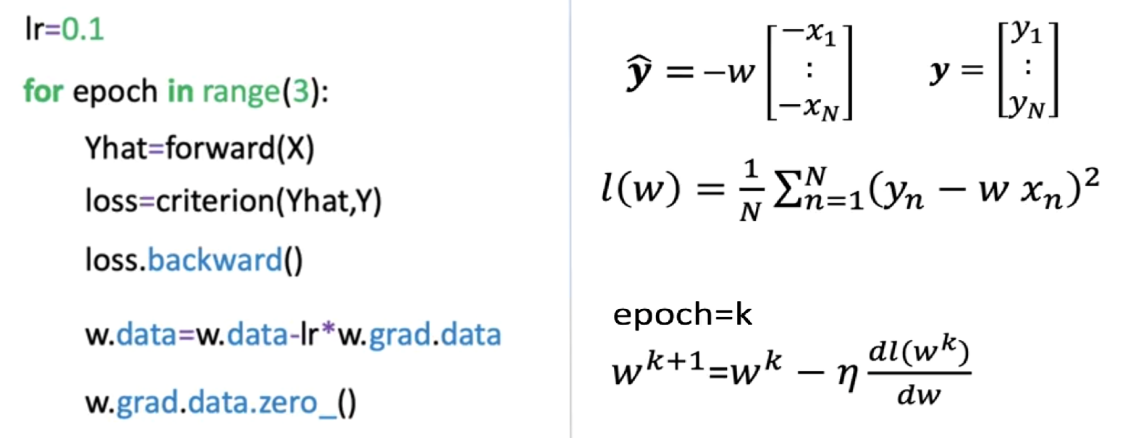
The model is defined using a **forward()** function that performs a simple linear transformation.

The cost is computed using a loss function that evaluates how far the predicted values are from the actual target values. In this example, Mean Squared Error (MSE) is used as the loss criterion. Though it represents the cost, it is referred to as "loss" to align with PyTorch's terminology.



The key training steps are:

* Compute predictions using the forward function
* Evaluate loss using the defined cost function
* Perform backpropagation using **loss.backward()**, the method backwards on the loss calculates the derivative with respect to all the variables in the loss function (PyTorch will be able to differentiate variables as the parameter **requires\_grads** is set to **True**).
* Access gradients with **w.grad**, this method gives the derivate at the point -10.
* Update the slope (parameter) w using the derivative and the learning rate. The attribute **.data()** gives access to the data contained in the variable.
* Reset gradients to zero with **w.grad.zero\_()** for the next iteration, this is due to the fact that PyTorch calculates the gradient in a iterative manner.



### 🔹 Epochs, Iterations, and Loss Reduction

The process is repeated over multiple **epochs**, where one epoch equals one full pass over the dataset. Each iteration involves:

* Updating the model parameter (slope)
* Observing the gradual decrease in loss
* Adjusting the predicted line to better fit the noisy data

Visualization is used to track:

* The current parameter estimate (as a red dot in the cost function plot)
* The fit of the predicted line (blue) against actual data points (red)
* The loss trend over time

The gradient magnitude determines the size of the parameter update:

* In early epochs, a steep gradient leads to large changes in the slope.
* In later epochs, as the model approaches optimal values, the gradient and parameter updates become smaller.

To better understand this slowdown, it’s helpful to look at the tangent line at the points for different iterations. The tangent line slope is equal to the derivative.

* For the first point, the slope is large as such the jump is large.
* For the third iteration the slope is much smaller so the decrease of the average loss is much smaller.

### 🔹 Monitoring Loss Across Epochs

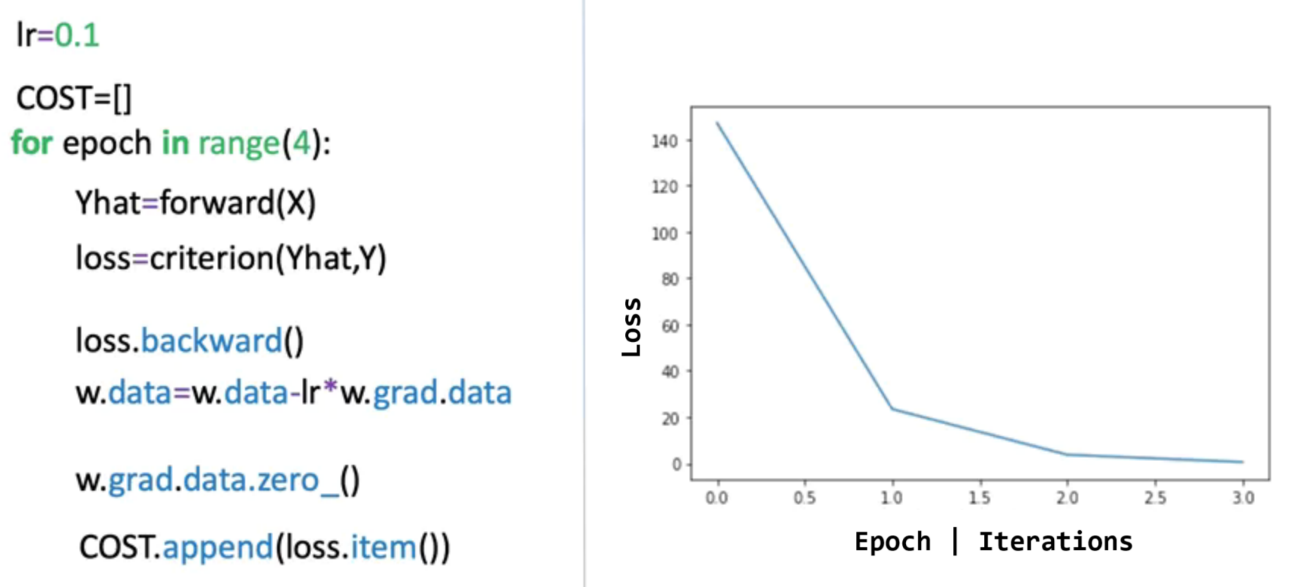
As our models get more complicated, it gets more difficult to plot the COST or average loss for each parameter, one alternative is to look at the COST for every iteration.

To visualize training progress:

* Loss values are appended to a list on each iteration.
* **.item()** is used to convert PyTorch tensors into native Python numbers.
* Loss is plotted over iterations to verify convergence.

The plotted graph shows:

* A consistent decrease in loss as training progresses
* Smoother model convergence
* Correlation between parameter updates and improvements in model performance



### ✅ Takeaways

✅ PyTorch enables low-level manual control over gradient descent and parameter updates using raw tensors.

✅ Setting **requires\_grad=True** allows automatic gradient computation.

✅ Loss is computed and used to update model parameters through **.backward()** and gradient subtraction.

✅ The learning rate controls the step size in each iteration; gradients guide the direction of parameter updates.

✅ Visualization of loss per epoch and model fit provides insight into convergence and optimization efficiency.

✅ This process builds foundational intuition for deeper PyTorch training workflows and prepares for future modules using higher-level abstractions.

# M. 2 – Section 4

**Linear Regression Training with PyTorch**

## 📌 PyTorch LR Training – Slope and Bias

This section focuses on training a linear regression model in PyTorch by learning both slope (weight) and bias using gradient descent.

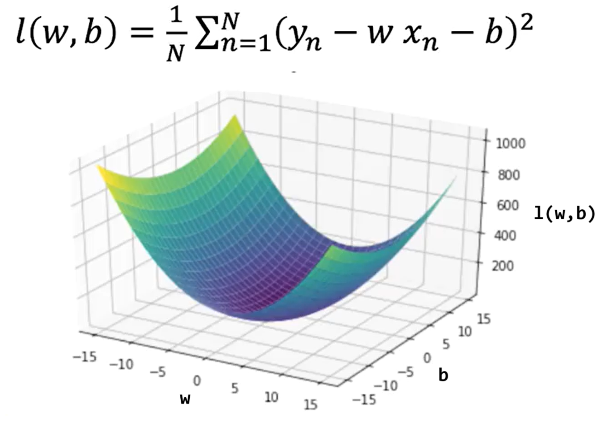
The objective is to minimize a cost surface defined by these parameters and understand how the model iteratively adjusts them to fit the training data.

### 🔹 Cost Surface and Parameter Space

The cost function in linear regression is defined as the **average loss** across training samples. It depends on two parameters:

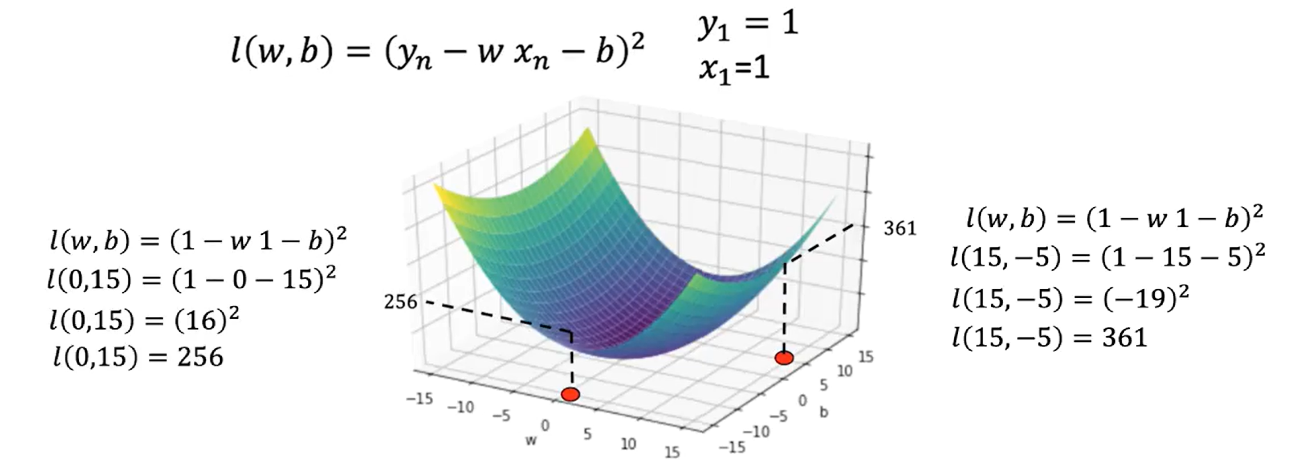
* **Slope (w)**: Determines the relationship between input x and output y.
* **Bias (b)**: Controls the vertical offset of the line.

When considering both parameters simultaneously, the cost function becomes a **two-dimensional function**, and it can be visualized as a **cost surface**:

* One axis represents the slope.
* One axis represents the bias.
* The vertical dimension (height) represents the value of the cost.

This surface visualization allows us to understand how the cost changes depending on the parameter combinations.

High-cost values indicate poor model fit, while regions with low-cost values indicate better predictions.

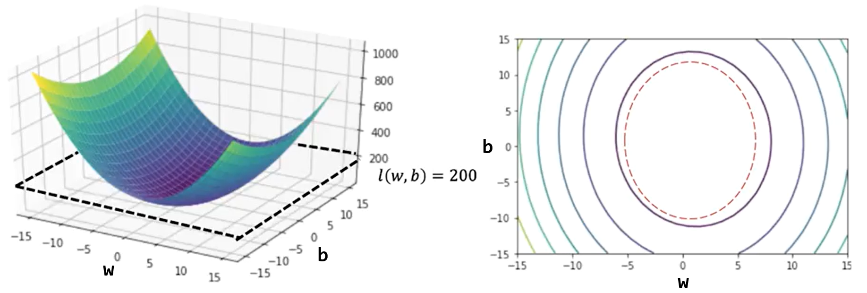


### 🔹 Contour Plots and Surface Slices

A contour plot is a useful tool for understanding cost surface. It’s a birds-eye view of the surface.

A **contour plot** is a two-dimensional representation that shows slices of the cost surface at fixed cost values. Each contour line connects parameter combinations that yield the **same cost**.

* The horizontal axis corresponds to slope (w).
* The vertical axis corresponds to bias (b).



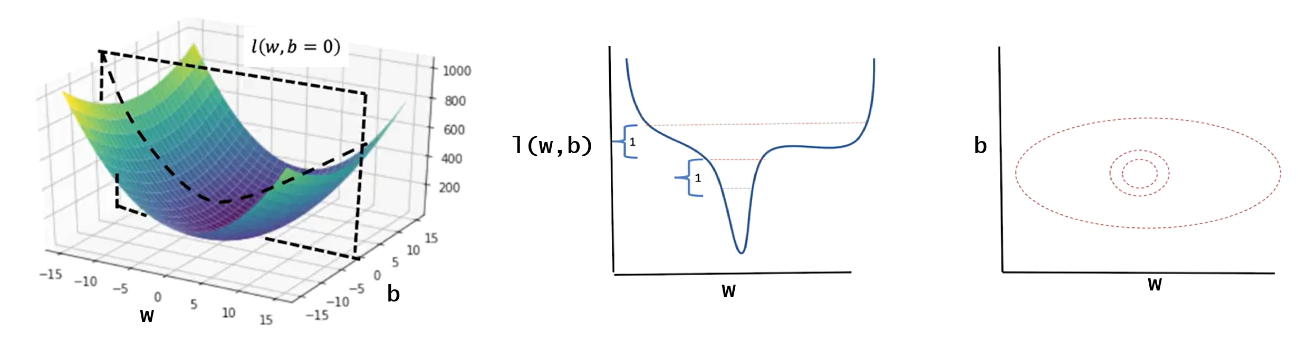
The **spacing between contour lines** reflects how steep or flat the surface is:

* Closely spaced contours indicate steep changes in cost (high gradient).
* Widely spaced contours indicate shallow regions (low gradient

Vertical and horizontal slices of the surface show how cost changes when varying one parameter at a time:

* A **vertical slice** (e.g., fixing b = 0) shows how cost varies with slope.
* A **horizontal slice** shows how cost varies with bias.
* These cross-sections visually demonstrate how curvature affects the update magnitude in gradient descent.

As you move away from the center (the minimum), the rate of change (gradient) increases, which informs the direction and size of parameter updates.



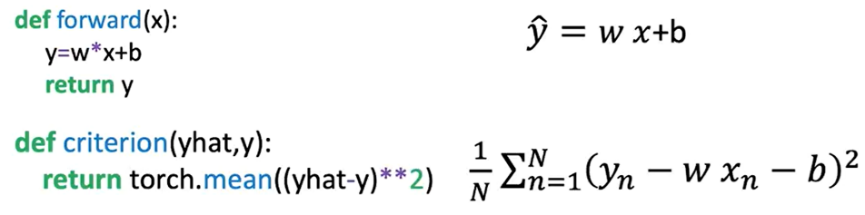
### 🔹 Gradient Descent in PyTorch (Manual Implementation)

ℹ️ There are several ways to minimize the cost function, the following methos is considered the “hard way” and usually is not implemented.

The training process is performed using PyTorch tensors and manual gradient descent.

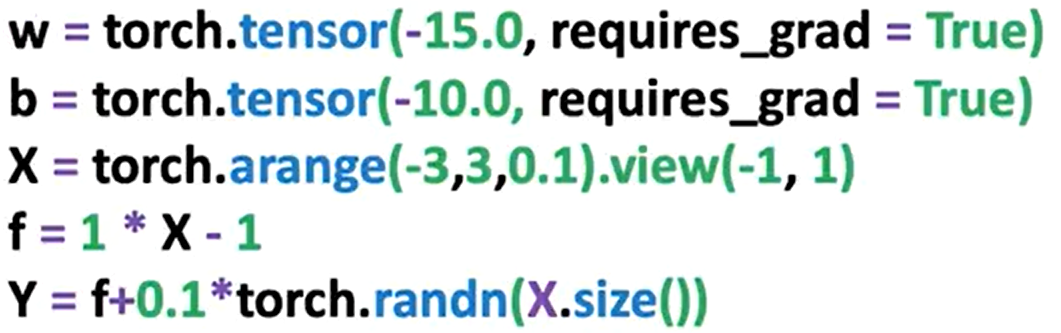
The following are key aspects of the training process:

* The model uses a **forward function** that defines de prediction using the equation of a line (includes both slope and bias).
* A criterion (cost) function is defined to calculate the mean squared error, representing the cost between predicted and actual values.



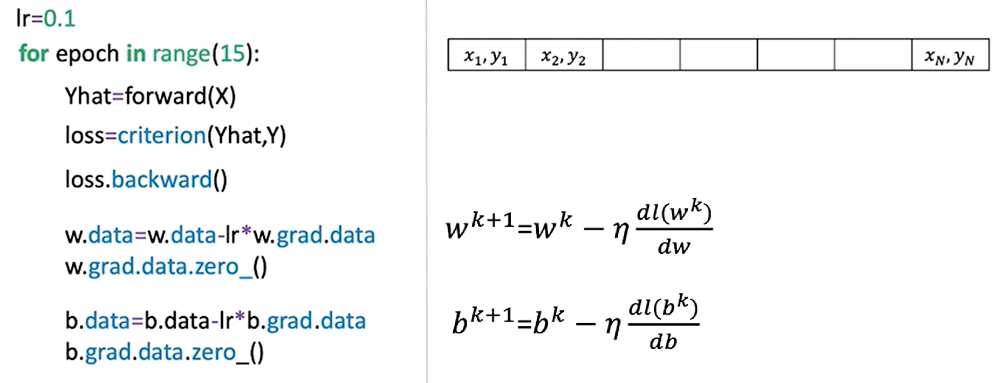
Tensors for:

* **w** (slope) and **b** (bias) are initialized with **requires\_grad=True** so PyTorch tracks their gradients.
* **X** (input data) and **Y** (target values) are created for the regression task.



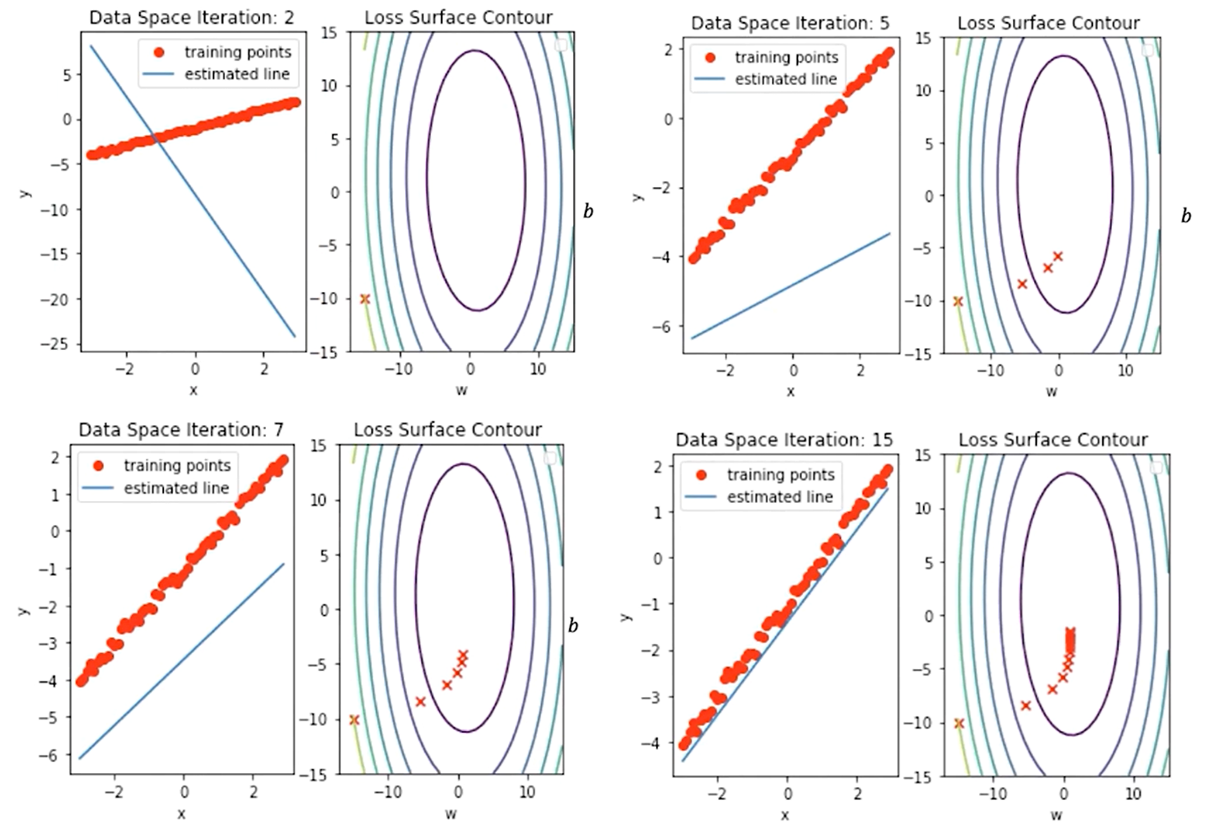
The procedure is identical as seen in last section, except we also update the bias term. During each epoch:

* The model predicts output using the forward function.
* The loss is computed.
* **.backward()** is called on the loss tensor to calculate gradients.
* The **.grad** attribute is accessed for both parameters to update them manually.
* **.data** is used to apply the update using the learning rate.
* .**zero\_()** is called to reset gradients for the next iteration.

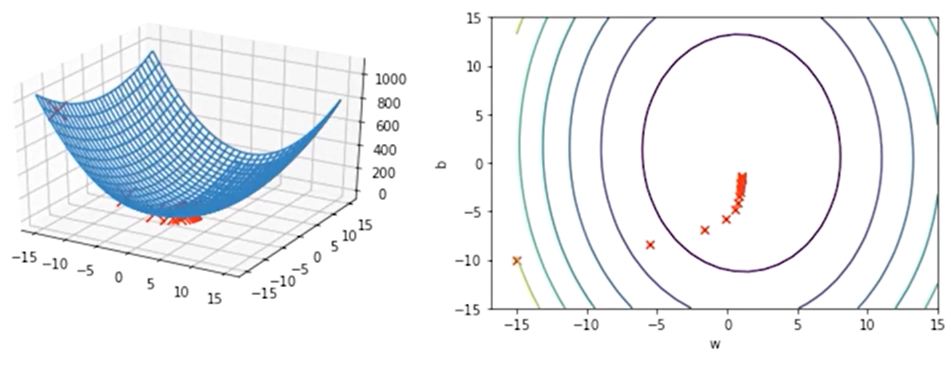


The plot on the right shows the loss or cost function for different values of the parameter; the red x’s represent parameter values at a given iteration.

The plot on the left shows the estimated function using the parameter values, and the red dots show the training points.



In this image we can see the correspondence between the contour lines and the loss surface.



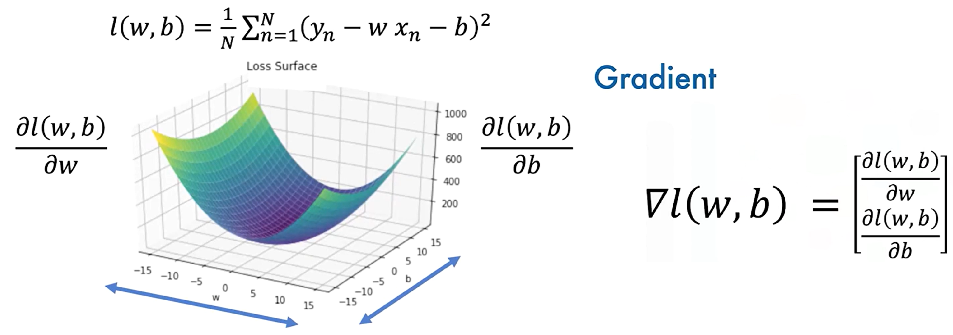
This manual procedure gives insight into how PyTorch handles gradients internally and how gradient descent is applied in practice.

### 🔹 Gradient Vector and Direction of Optimization

In general, the derivative with respect to a multivariate function is called a partial derivative.

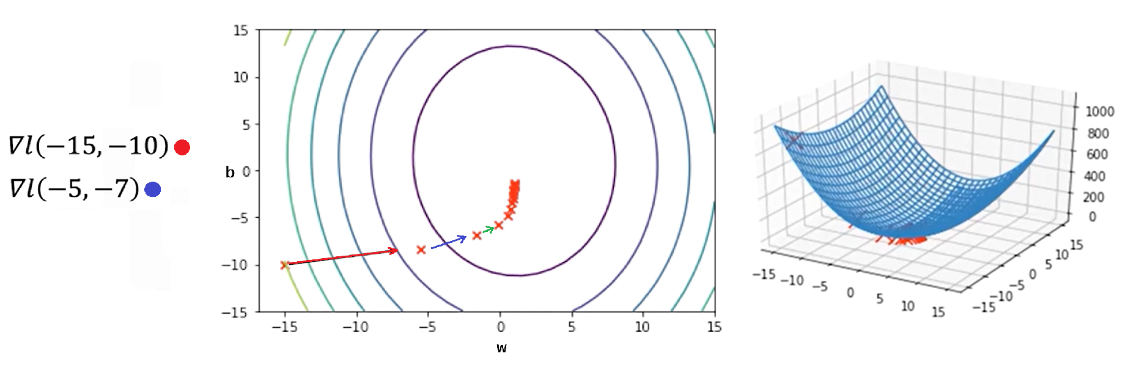
If we put the partial derivative with respect to each variable into a vector, we get the gradient, hence the name gradient descent.

**The gradient vector**, points in the direction of the steepest increase of the cost function.



The gradient is always **perpendicular to the contour lines** and guides the updates during training.

The gradient also points to the direction of greatest change, hence points to the direction of the next iteration, the same is true for the next iteration, and so on.



### ✅ Takeaways

✅ The cost function in linear regression with slope and bias defines a **surface** representing model error across parameter combinations.

✅ **Contour plots and surface slices** help visualize how parameter updates impact model performance.

✅ **Gradient descent** is used to iteratively update both slope and bias, reducing the model’s loss with each epoch.

✅ PyTorch allows full manual implementation of gradient-based optimization using tensors, gradient tracking, and **.backward()** computations.

✅ The **gradient vector** directs the updates and always points toward the direction of greatest increase, so moving opposite to it reduces the loss.

✅ After several epochs, the predicted line aligns closely with the data, and parameters converge to values that minimize the cost.