# Module 2

**Linear Regression with PyTorch**

**Linear Regression Prediction with PyTorch**

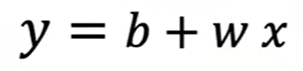
## 📌 Simple Linear Regression Prediction

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:



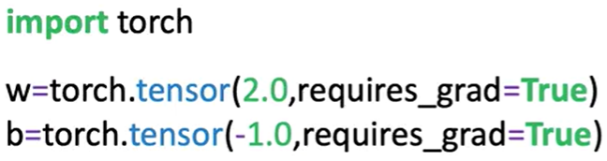
Where:

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

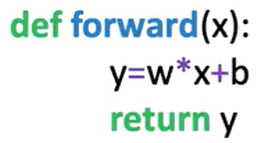
This equation defines the **linear model** that maps input values to estimated outputs. The goal of training is to determine optimal values for and .

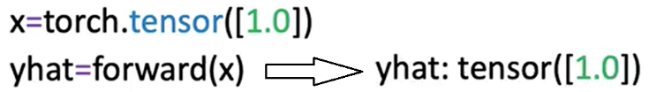
### 🔹 Prediction Using Tensors

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

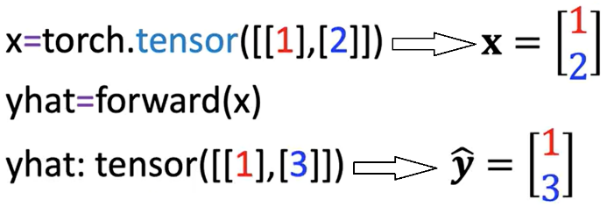
* One for the weight (slope).
* One for the bias (intercept).

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:

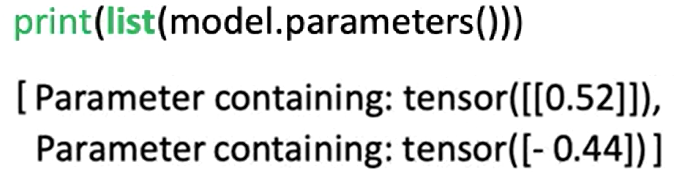
**nn.Linear(in\_features, out\_features)**.

Parameters:

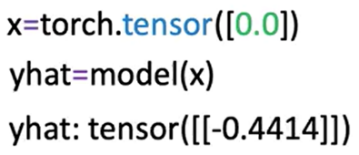
* **in\_features**: Number of input features (columns).
* **out\_features**: Number of output features.

After constructing the model:

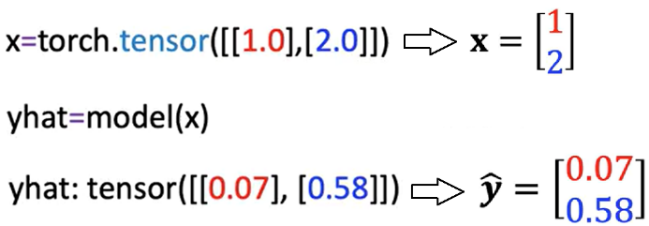
* The slope and bias are initialized randomly.
* Model parameters can be inspected using:

**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**.

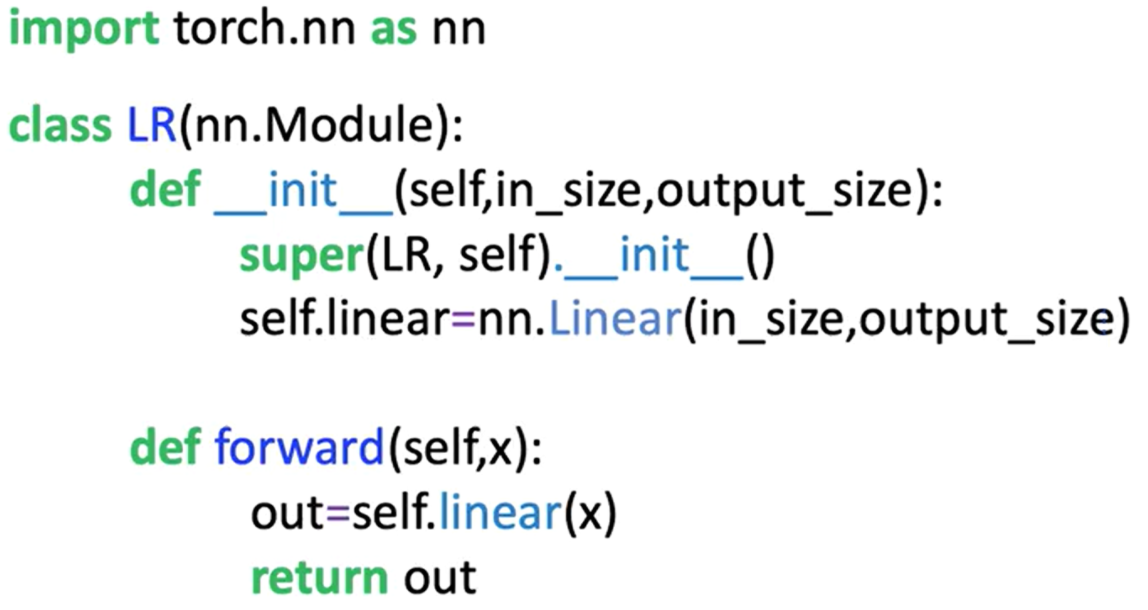
🔸 **Custom Class Structure:**

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

In the constructor:

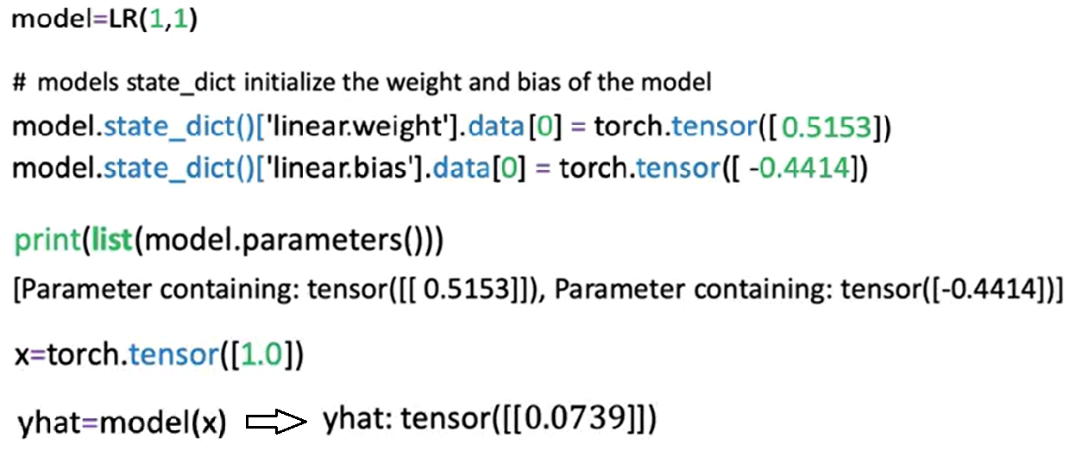
* The base class constructor is initialized using **super()**.
* In the object constructor, the argument are the **size of the input** (x, **in\_size**) and **output** (y, **output\_size**).
* A linear layer is created using **nn.Linear(input\_size, output\_size)** and stored as **self.linear**.

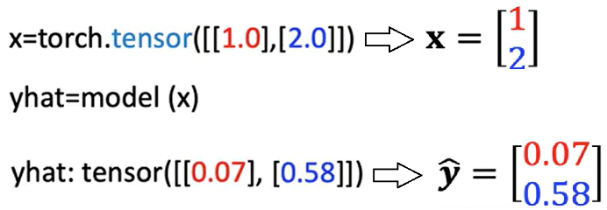
A **forward** method is defined to apply the linear transformation.



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

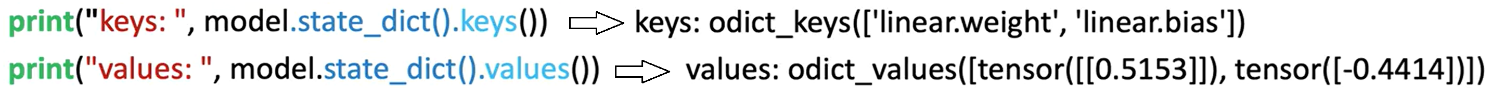
The **state\_dict()** method returns a Python dictionary containing all the learnable parameters of the model, as models get more complex this method becomes more useful.

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.