# Module 2

**Linear Regression with PyTorch**

**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.

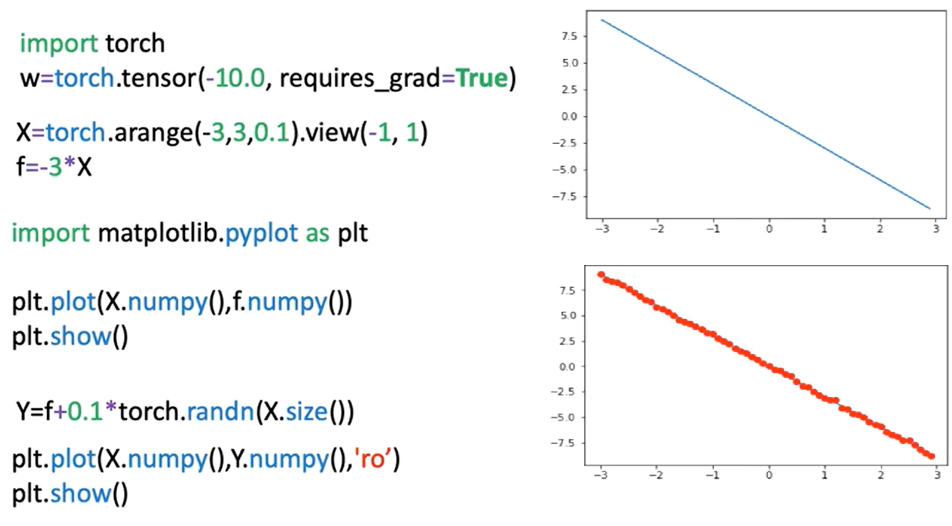
### 🔹 Gradient Descent with PyTorch Tensors

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:

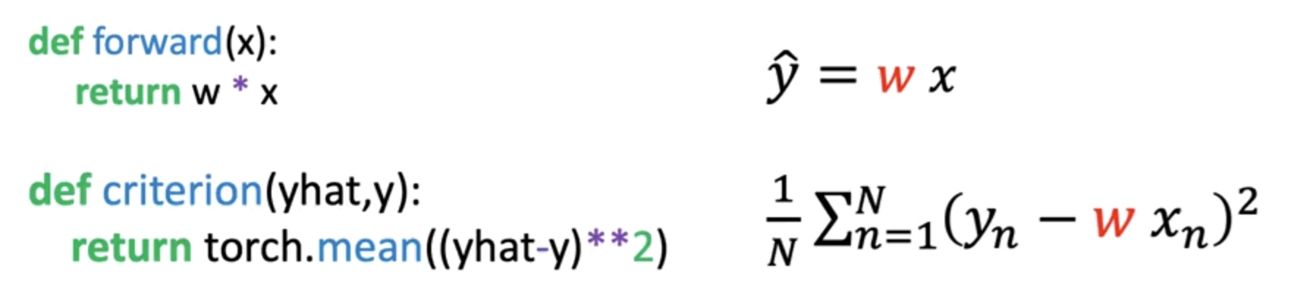
* Creating a tensor for the slope parameter w
* Generating sample X values and corresponding Y values using a known slope, **view()** function is used to add an additional dimension.
* Adding random noise to simulate real-world data variability
* Visualizing the initial data and the true line using matplotlib



### 🔹 Loss Calculation and Optimization Process

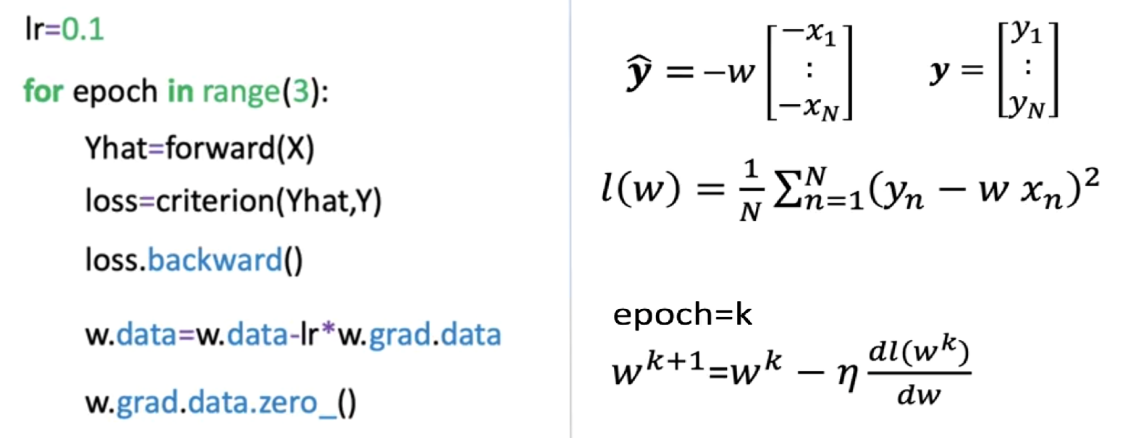
The model is defined using a **forward()** function that performs a simple linear transformation.

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.



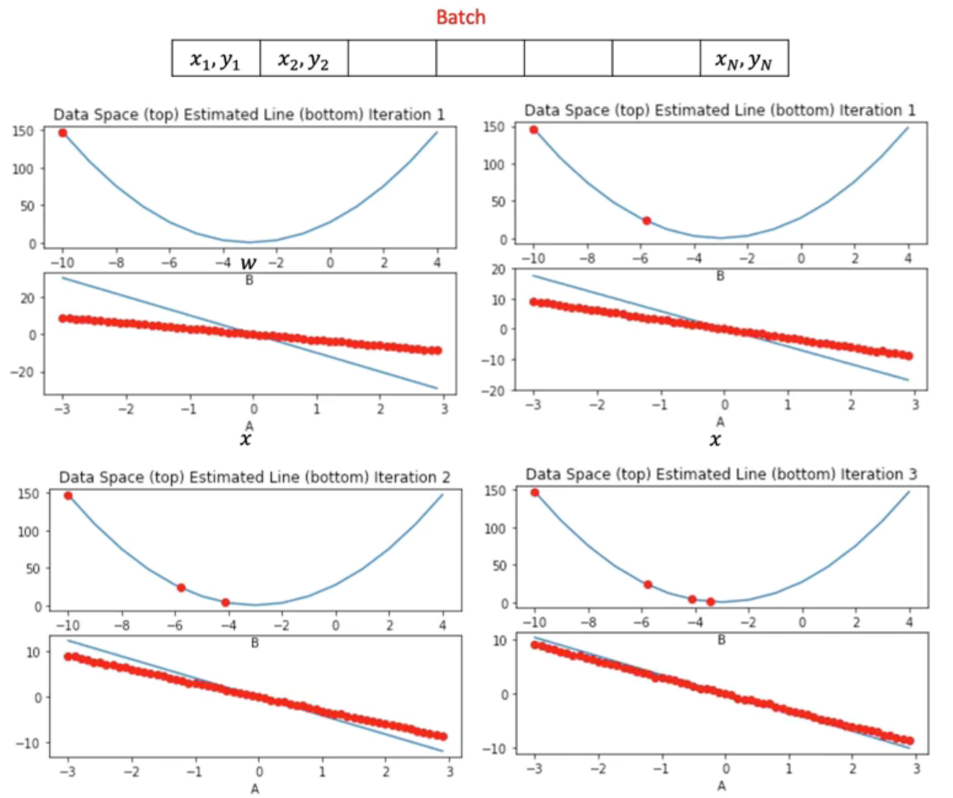
The key training steps are:

* 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

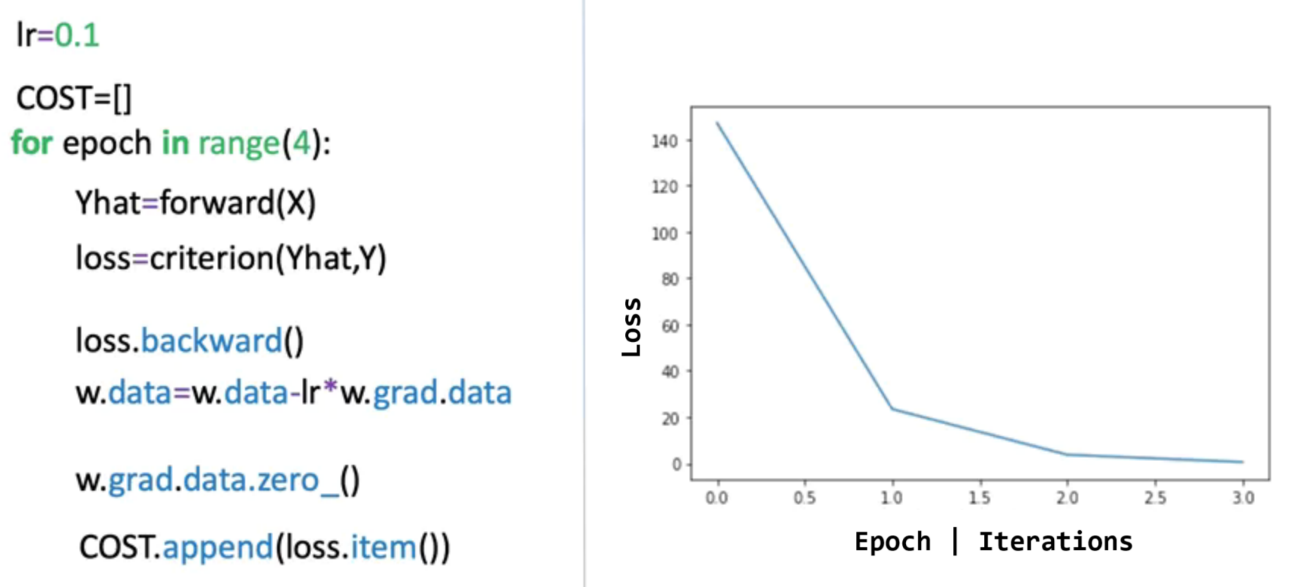
As our models get more complicated, it gets more difficult to plot the COST or average loss for each parameter, one alternative is to look at the COST for every iteration.

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.