# Module 3

**Linear Regression PyTorch Way**

**Stochastic & Mini-Batch Gradient Descent**

## 📌Stochastic Gradient Descent and Data Loader

This section introduces stochastic gradient descent (SGD) as a method for optimization and demonstrates its practical implementation using both manual iteration and PyTorch's DataLoader.

The goal is to train a model by minimizing the cost function through updates on individual samples rather than the entire dataset at once.

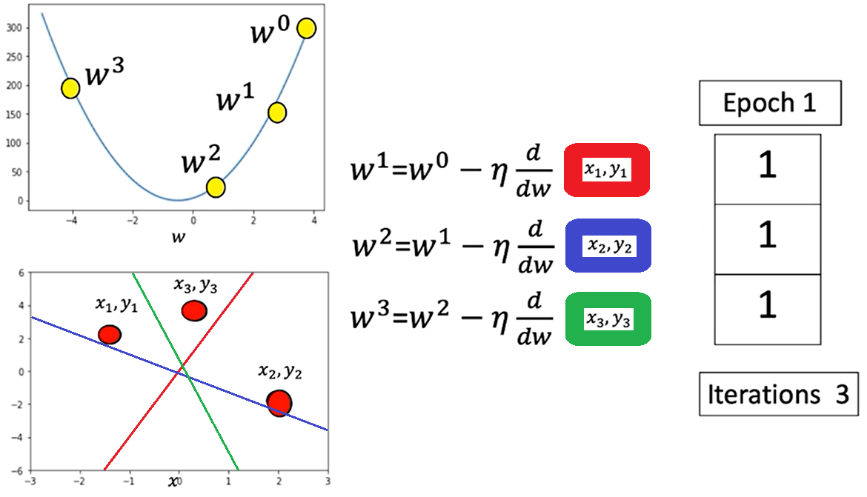
### 🔹 Stochastic Gradient Descent Overview

**Stochastic Gradient Descent** differs from batch gradient descent by **updating model parameters one sample at a time** instead of using the full dataset.

This leads to faster updates but introduces variability (fluctuations) in the cost function.

* In batch gradient descent, parameters w and b are updated by minimizing the total cost function computed across all data points.
* In SGD, each data point individually affects the parameter update. While this allows faster updates, it may result in erratic movements due to outliers or noisy samples.

During an epoch:

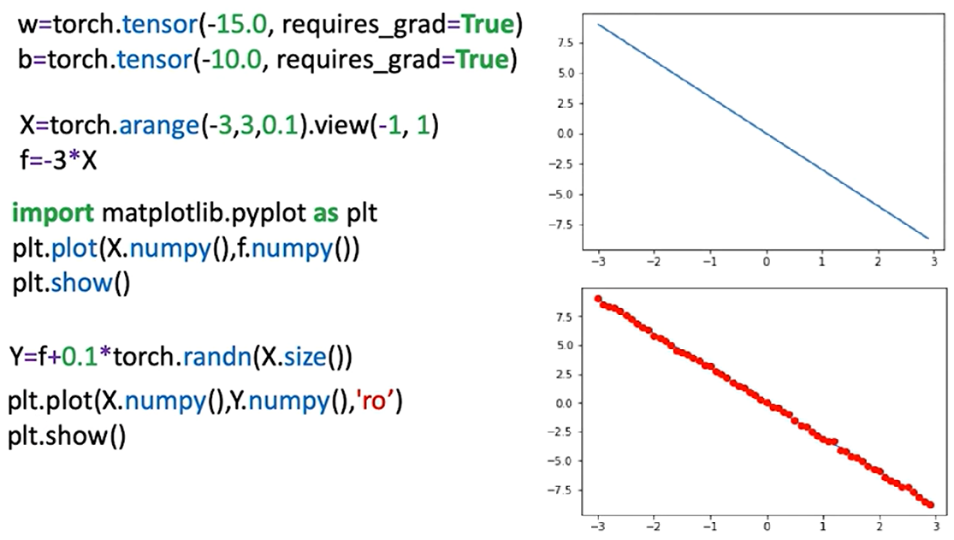
* Each data sample is processed in sequence (one iteration one process data).
* The parameter is updated based on the gradient from that single data point.
* The line (model prediction) moves closer or farther from the true data depending on the sample's influence.
* In this example (x3, y3) data point is an outlier, the loss increases drastically

This method approximates the cost function by calculating it one sample at a time, updating weights accordingly.

### 🔹 Manual Implementation in PyTorch

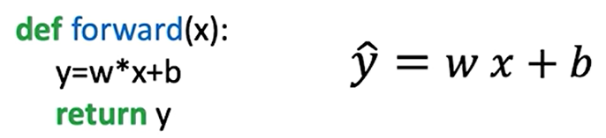
To perform manual Stochastic Gradient Descent (SGD) in PyTorch:

* A tensor for the slope is created with **requires\_grad=True** to allow automatic differentiation.
* Synthetic x values are generated and mapped to a linear function. **view()** method is used to add a dimension.
* Random noise is added to simulate realistic data.



Model structure:

* A forward function computes predictions using the line equation.



* A criterion (loss) function measures the distance between prediction and target.



* The model iterates over the data for a set number of epochs, updating the slope and bias each time

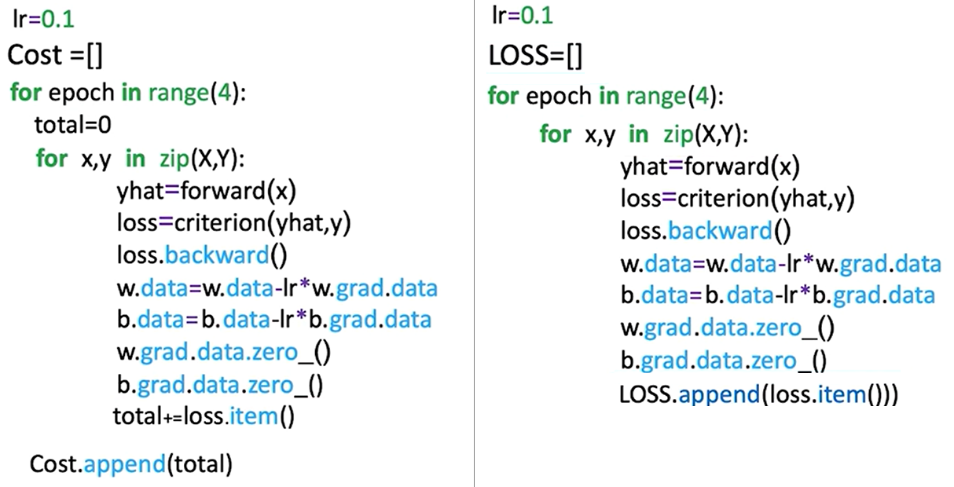
During training:

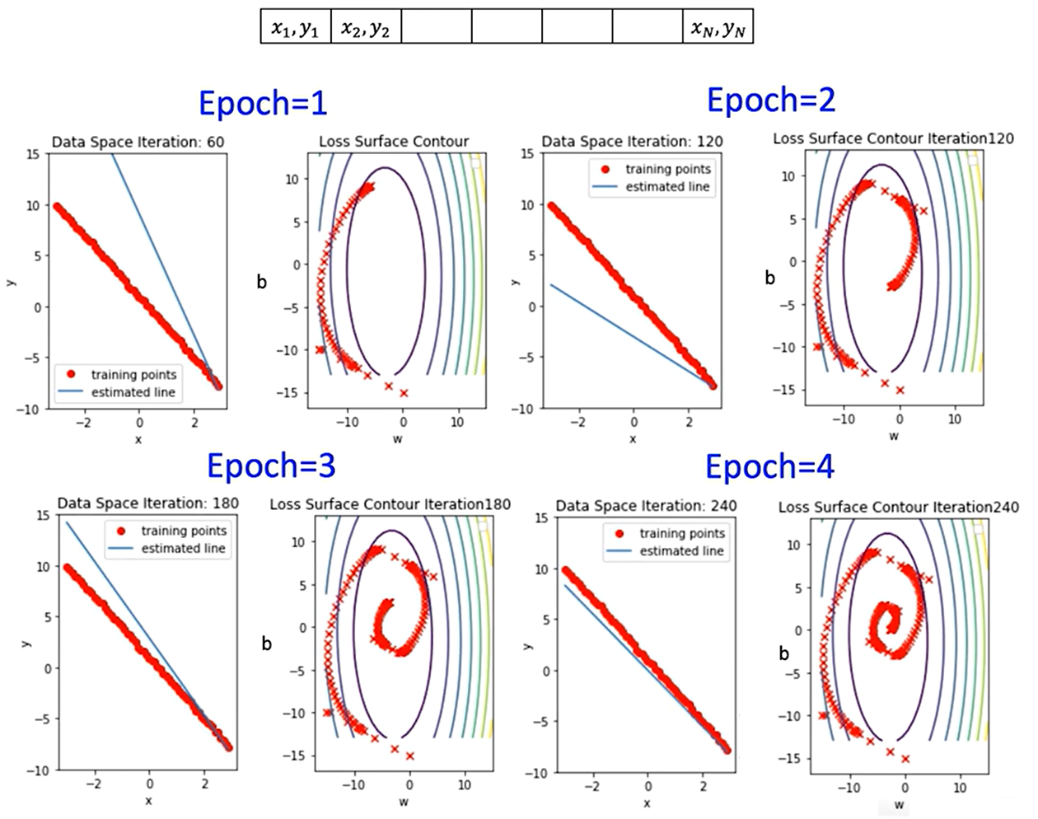
* For each sample, the loss is calculated.
* **.backward()** is called to compute gradients.
* Parameters are updated manually using learning rate and gradient.
* Gradients are reset between iterations using **.grad.zero\_()** to prevent accumulation.

Progress:

* Parameter values are tracked per epoch.
* Loss values are stored in a list to monitor convergence. Loss can be stored in two ways in order to track model progress:
  + **Each training step (batch):** The loss value can be stored in a list in each iteration, this can be thought of as an approximation of the cost.
  + **Each Epoch:** The cost can be calculated by storing in a list the accumulated loss in total for each epoch.

The aggregated loss over the entire dataset is calculated (the **average loss** can be calculated as well) and the cost value for each epoch is appended.





### 🔹 DataLoader for SGD

Creating a **custom dataset class** is necessary when using PyTorch’s **DataLoader** in order to define **how the data should be accessed**.

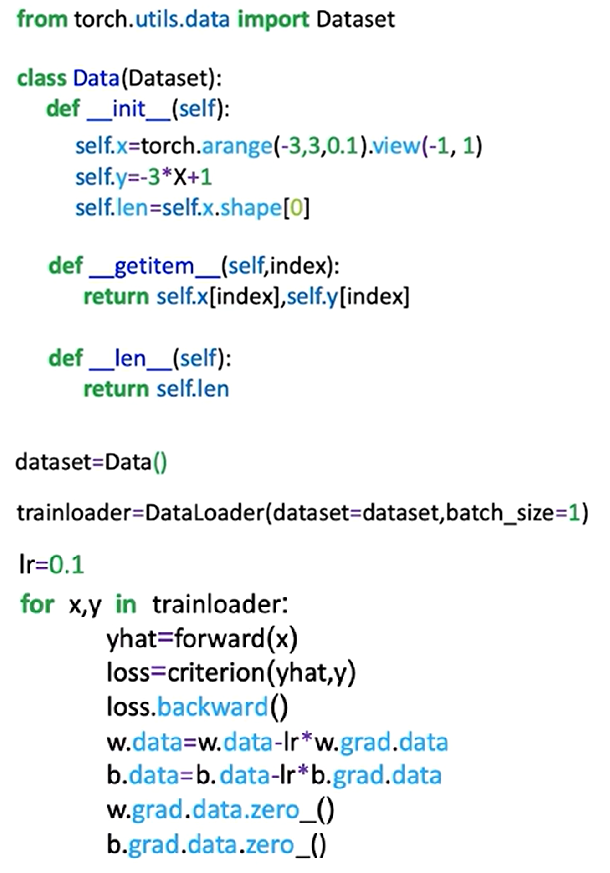
The DataLoader expects a dataset object that implements **two methods**:

1. **\_\_len\_\_()** – returns the total number of samples
2. **\_\_getitem\_\_(idx)** – returns a single sample (usually as a tuple: (input, label))

When passed to **DataLoader**, the dataset:

* Gets indexed with **\_\_getitem\_\_(i)** during each iteration.
* Is batched automatically if **batch\_size > 1**.
* Can be shuffled, parallelized (via **num\_workers**), and more.

The custom dataset class is created using:

* The **\_\_init\_\_** method initializes features (x) and targets (y) as tensors.
* The **\_\_len\_\_** method returns dataset length.
* The **\_\_getitem\_\_** method retrieves samples by index.

The PyTorch **DataLoader** simplifies iteration:

* Accepts a dataset object.
* **batch\_size=1** is used to simulate stochastic gradient descent.
* Returns mini-batches (in this case, one sample per iteration).
* The **DataLoader** allows consistent, batched access to training data, supporting shuffling and multiprocessing.

### ✅ Takeaways

✅ Stochastic Gradient Descent updates weights per sample, offering fast, incremental learning but potentially unstable convergence.

✅ Manual implementation in PyTorch demonstrates gradient calculation, parameter updates, and tracking performance.

✅ DataLoader provides a more scalable and standardized method for batch-wise sample iteration.

✅ Storing and visualizing loss across epochs is critical for monitoring training progress and convergence.

✅ Parameter updates using PyTorch’s gradient tracking system mirror low-level optimization logic, preparing for deeper models.

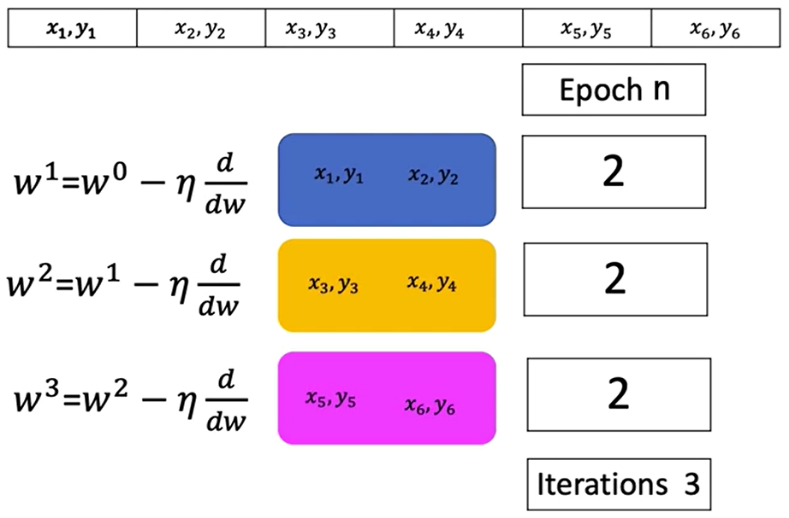
## 📌 Mini-Batch Gradient Descent

Mini-batch gradient descent enables efficient training of models on large datasets by processing multiple samples at once.

This approach reduces memory consumption and improves training performance compared to full-batch or stochastic gradient descent.

It divides the dataset into manageable subsets (batches), each used to perform a parameter update during training.

### 🔹 Mini-Batch Gradient Descent Overview

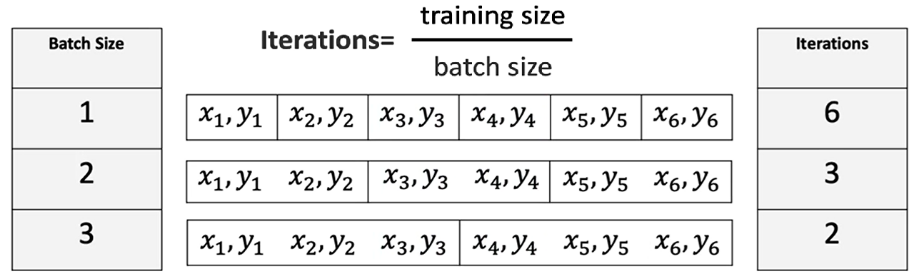
Mini-batch gradient descent uses a small group of samples in each iteration to approximate the gradient and update the model parameters. This allows the algorithm to operate on large datasets that would not otherwise fit into memory.

* + Each iteration computes the cost using a subset of samples rather than the entire dataset.
  + The cost for each iteration corresponds to the average loss over the current mini-batch.
  + Multiple mini-batch iterations form one epoch, which represents a full pass through the dataset.

In contrast to stochastic gradient descent (which uses a batch size of 1), mini-batch gradient descent uses more than one sample per iteration. This helps reduce the high variance in parameter updates, leading to more stable and efficient convergence.

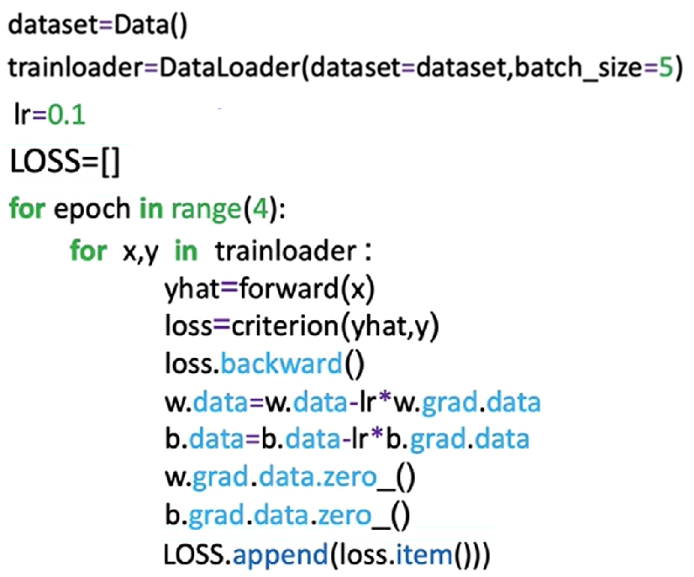
### 🔹 Mini-Batch Gradient Descent Overview

The number of iterations in an epoch is determined by the total number of samples divided by the batch size:



Each iteration within an epoch updates the model parameters based on the loss computed for that batch.

### 🔹 Mini-Batch Gradient Descent Overview

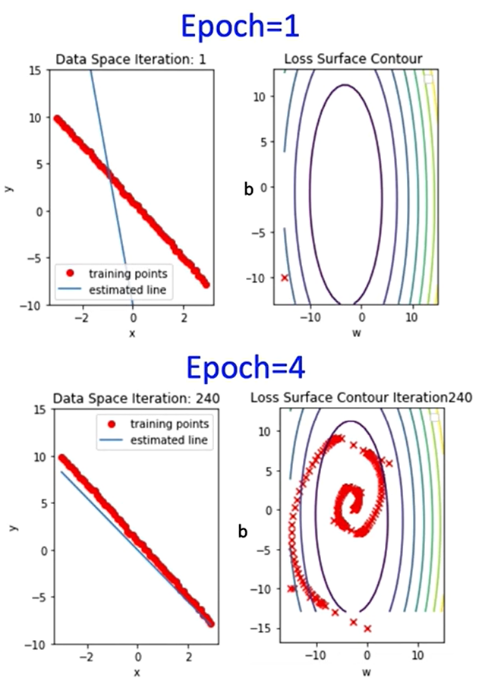
Implementing mini-batch gradient descent in PyTorch closely resembles the process for stochastic gradient descent, with a key change in batch size configuration.

Steps:

**🔸Create Dataset Object:**

* This object holds the training data.
* It enables access to individual samples or batches using slicing and indexing.

**🔸Create the DataLoader:**

* Pass the dataset to **DataLoader** and specify the desired **batch\_size**.
* This object enables efficient and automatic batching of data during training.

**🔸Training Loop:**

* Iterate over the DataLoader.
* For each batch:
* Compute the forward pass.
* Compute the loss.
* Perform backpropagation and update parameters using the optimizer.

**🔸Loss Tracking:**

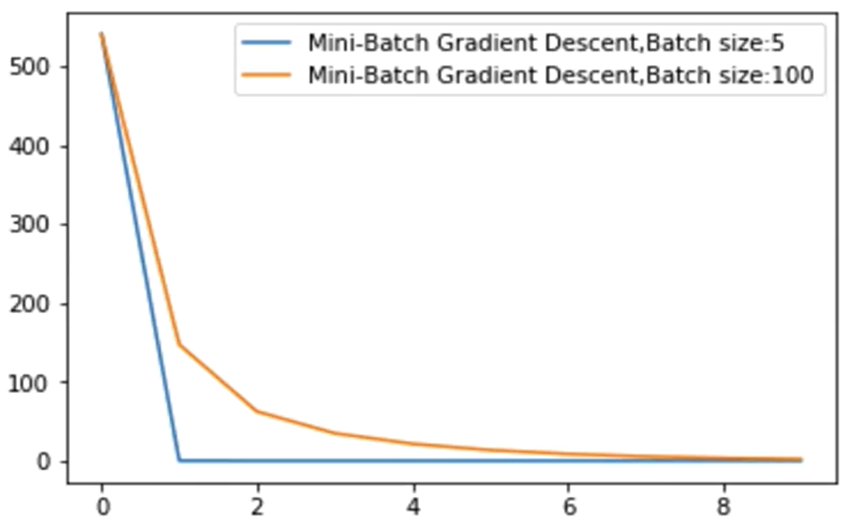
* + Store the loss after each iteration to monitor training progress.
  + This provides an approximation of the overall cost trend.

### 🔹 Convergence Rate and Batch Size

The convergence rate refers to how quickly the loss or cost function approaches its minimum value.

Different batch sizes influence this rate:

* **Smaller batches** tend to update parameters more frequently, which may introduce noise but can lead to faster learning in early stages.
* **Larger batches** provide more stable updates but may converge more slowly and require more computation per iteration.



A plot of cost vs. iteration count across different batch sizes illustrates how the learning dynamics shift depending on batch size, and how quickly or slowly models converge under different configurations.

### ✅ Takeaways

✅ Mini-batch gradient descent improves memory efficiency and training stability by processing small subsets of the data at each iteration.

✅ The number of iterations per epoch is inversely proportional to the batch size.

✅ PyTorch supports mini-batch training via the **DataLoader** by specifying the **batch\_size** parameter.

✅ Different batch sizes impact convergence rate, affecting training duration and learning behavior.

✅ Storing and analyzing the cost at each iteration helps visualize training progress and optimize training strategies.