# Module 3

**Linear Regression PyTorch Way**

**M. 3 – Section 1**

**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.

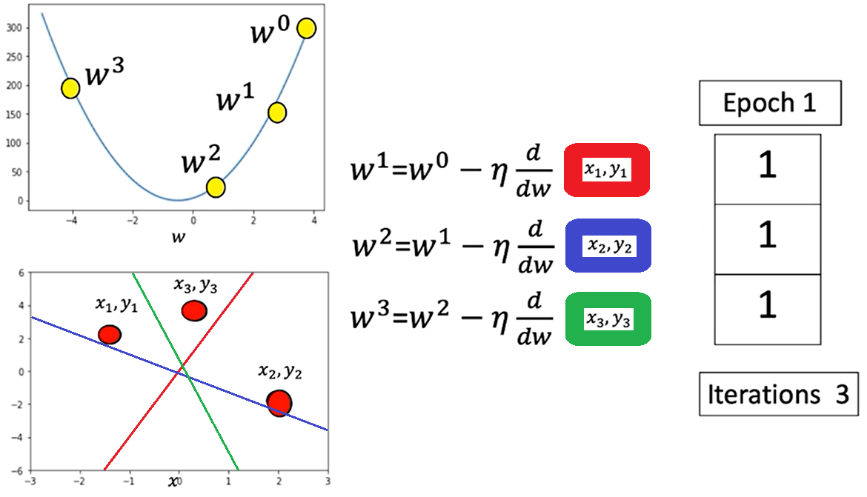
### 🔹 Stochastic Gradient Descent Overview

**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:

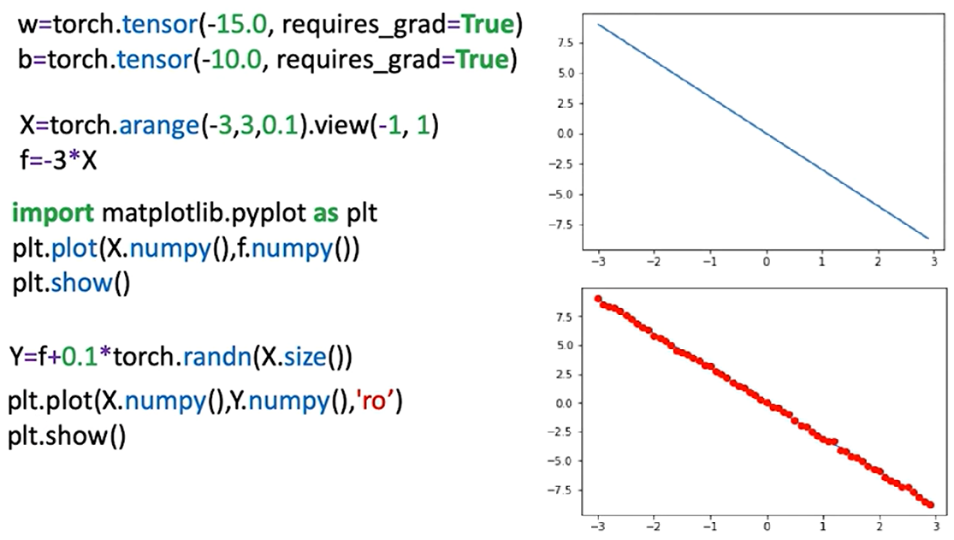
* Each data sample is processed in sequence (one iteration one process data).
* The parameter is updated based on the gradient from that single data point.
* The line (model prediction) moves closer or farther from the true data depending on the sample's influence.
* In this example (x3, y3) data point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

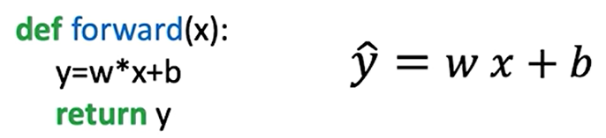
To perform manual Stochastic Gradient Descent (SGD) in PyTorch:

* 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

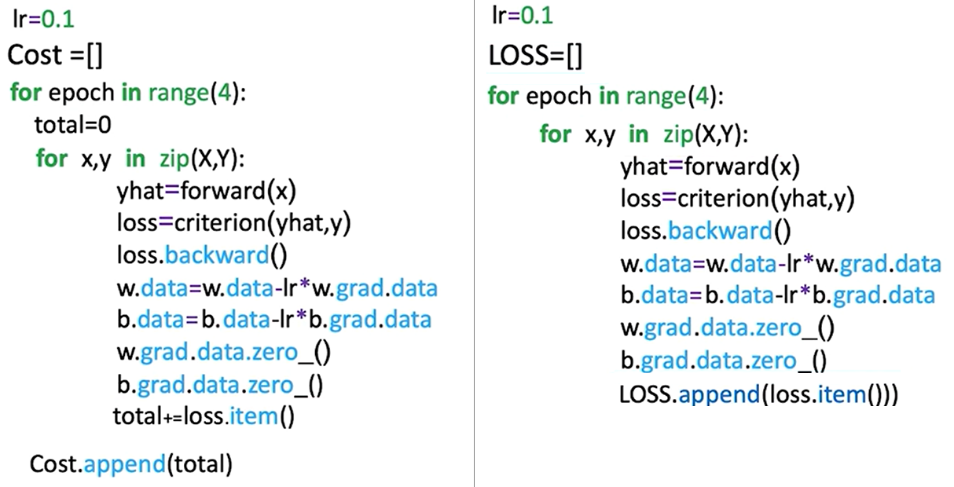
During training:

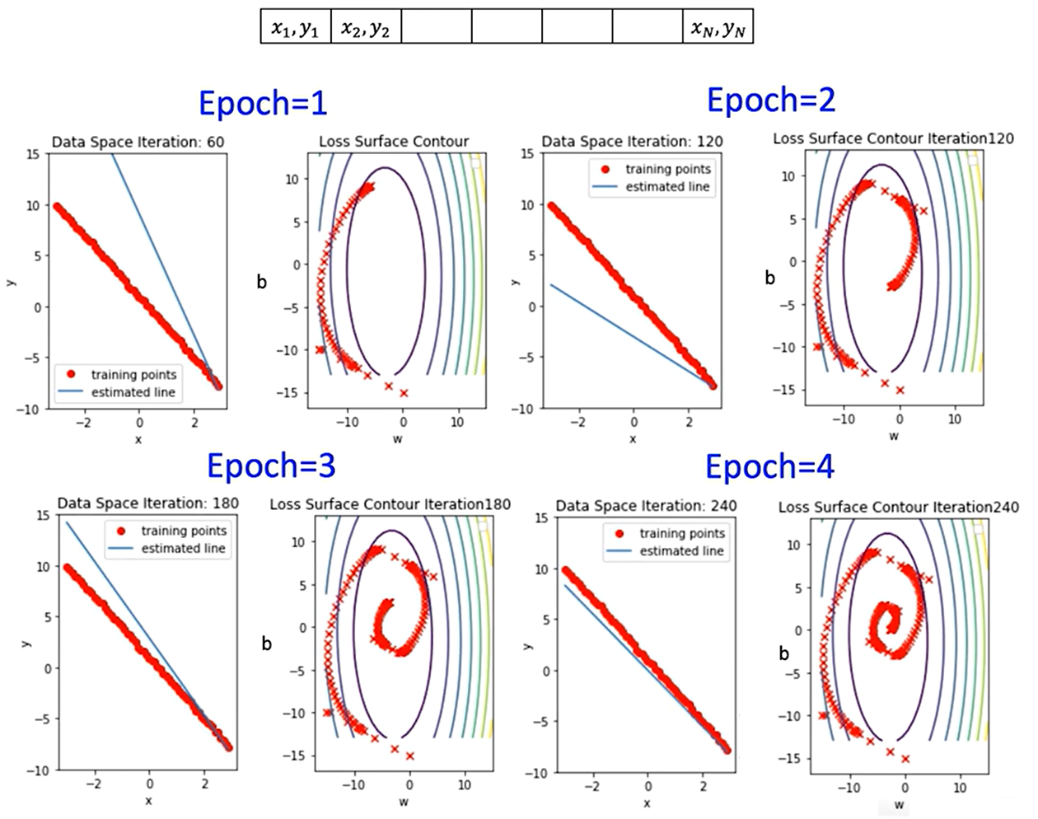
* For each sample, the loss is calculated.
* **.backward()** is called to compute gradients.
* Parameters are updated manually using learning rate and gradient.
* Gradients are reset between iterations using **.grad.zero\_()** to prevent accumulation.

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.

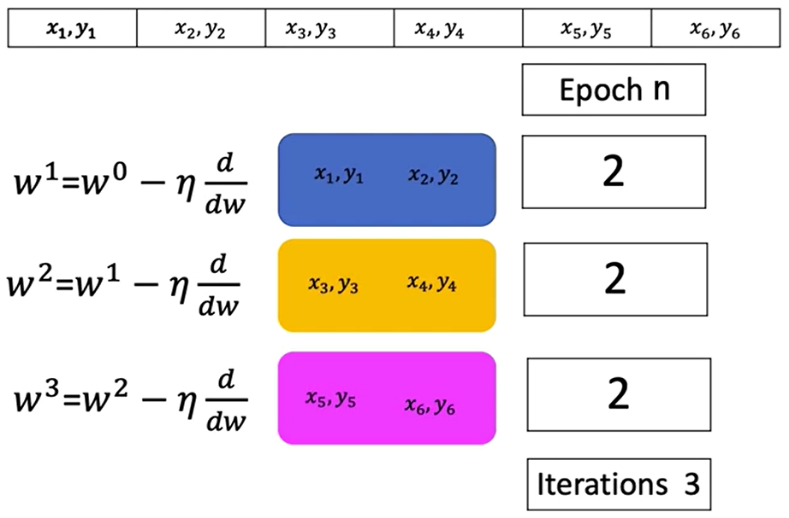
## 📌 Mini-Batch Gradient Descent

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

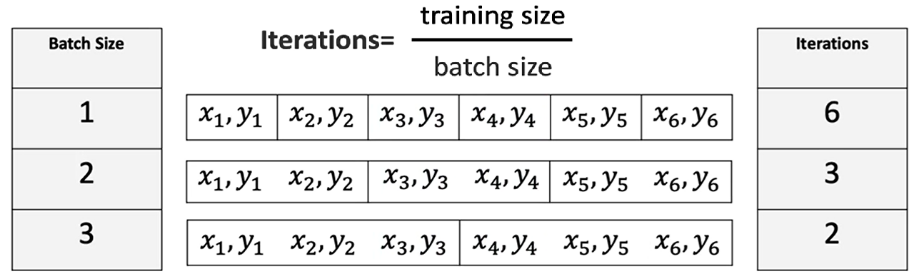
Mini-batch gradient descent uses a small group of samples in each iteration to approximate the gradient and update the model parameters. This allows the algorithm to operate on large datasets that would not otherwise fit into memory.

* + 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.

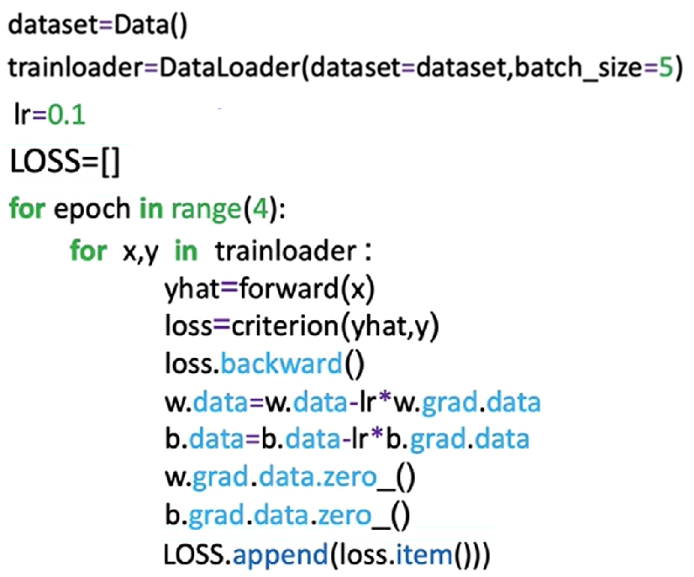
### 🔹 Mini-Batch Gradient Descent Overview

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

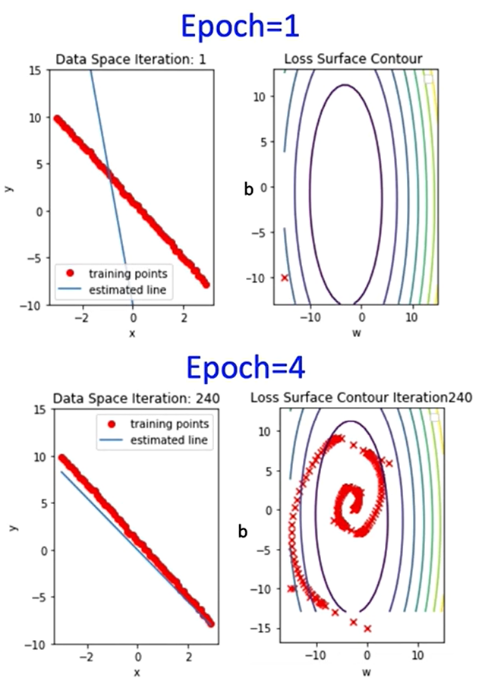
Implementing mini-batch gradient descent in PyTorch closely resembles the process for stochastic gradient descent, with a key change in batch size configuration.

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:**

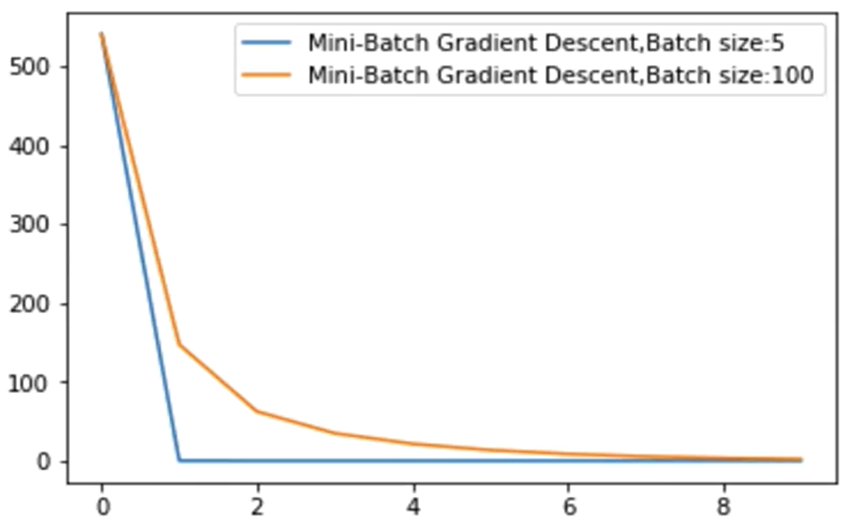
* + Store the loss after each iteration to monitor training progress.
  + This provides an approximation of the overall cost trend.

### 🔹 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.

# M. 3 – Section 2

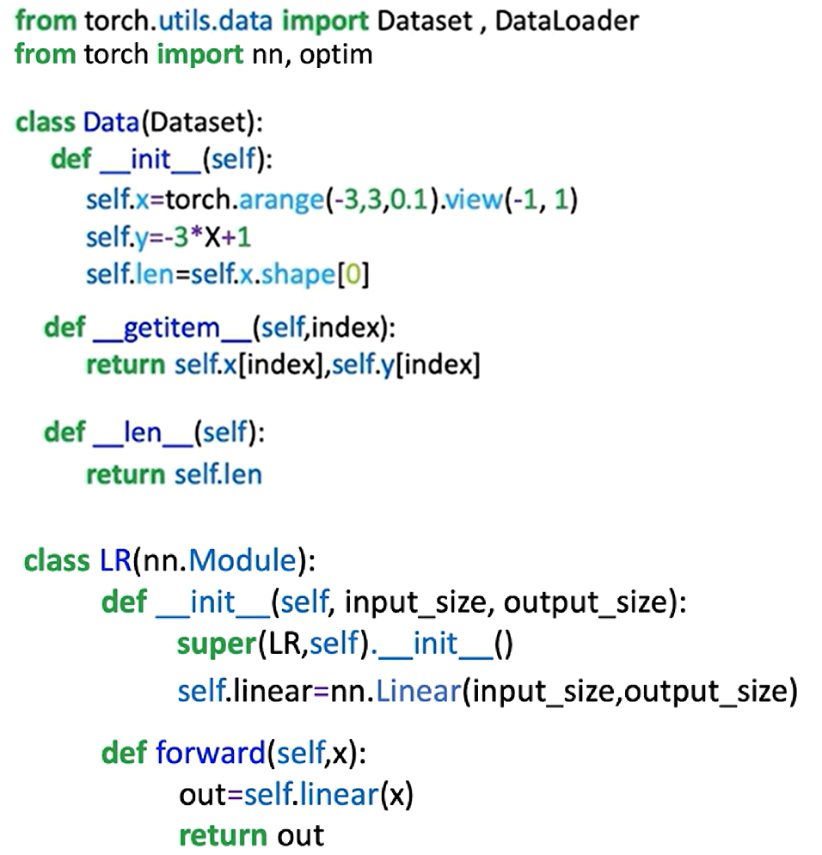
**Optimization in PyTorch**

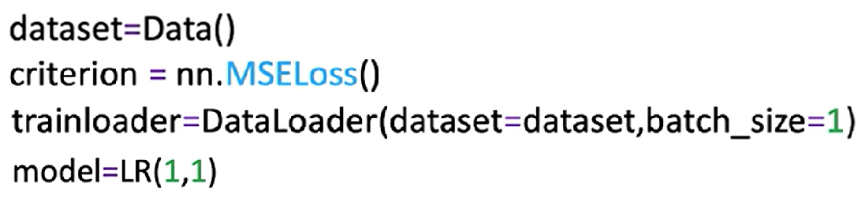
## 📌Optimization in PyTorch

This section introduces how to use PyTorch’s built-in optimizer for structured model training, using a standard PyTorch workflow, including model definition, parameter management, and iterative parameter updates with gradient information.

### 🔹 Optimizer Setup in PyTorch

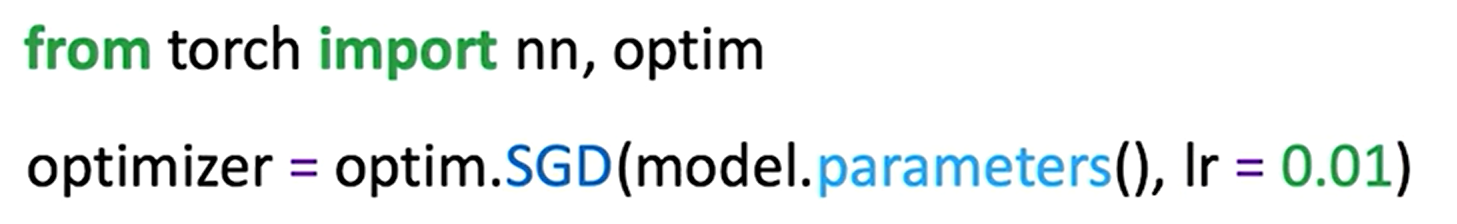
The PyTorch optimizer is responsible for managing the learning parameters of a model and applying gradient updates during training. This approach standardizes how different optimization algorithms are implemented and used in practice.

* A dataset object is first created to store the training data.
* A custom model class is defined by subclassing **nn.Module**. This model includes all learnable parameters such as weights and biases.
* A criterion function (or loss function) is defined using **nn.MSELoss()** or another predefined function from **torch.nn**. This function computes the cost between predicted outputs and ground truth labels.
* A **DataLoader** object is used to batch and shuffle data for more efficient iteration during training.

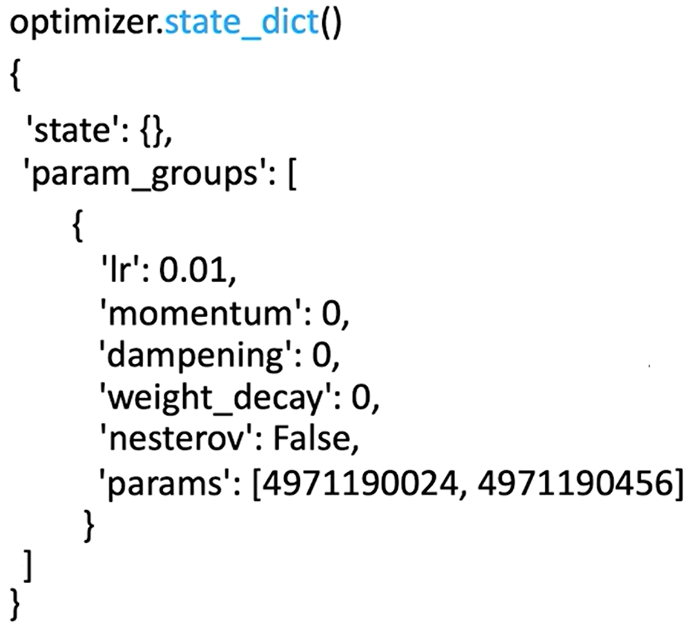


**🔸** **Constructing the Optimizer:**

* The optimizer is imported from **torch.optim**, e.g., **torch.optim.SGD**.
* The optimizer object is constructed by passing two arguments:
  + The learnable parameters of the model using **model.parameters()**.
  + A **learning rate**, which defines the update step size.



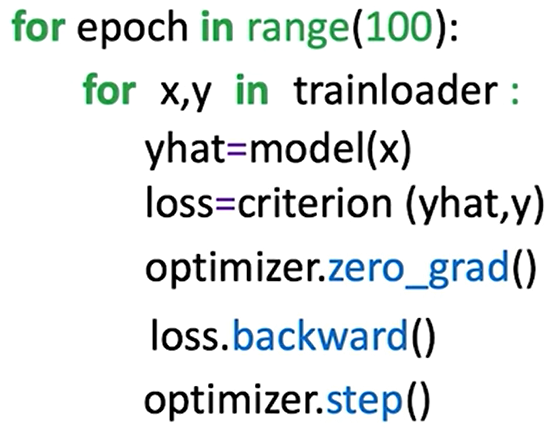
* The optimizer maintains a **state dictionary** (**optimizer.state\_dict()**), which includes both hyperparameters and internal states necessary for training.



This structure ensures the optimizer is aware of which parameters to update and how to update them across training steps.

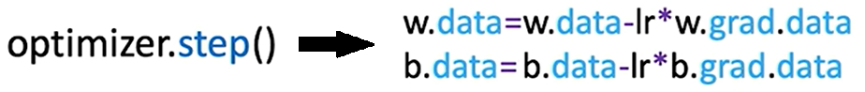
**🔸** **Optimization Workflow:**

Each epoch follows a standard training routine that involves:

1. Iterating over batches in the **trainloader**.
2. Performing a **forward pass** using the model to generate predictions.
3. Calculating the **loss** between predictions and targets.
4. Calling **optimizer.zero\_grad()** to reset gradients from previous steps.
5. Running **loss.backward()** to compute gradients via automatic differentiation.
6. Calling **optimizer.step()** to update model parameters using the gradients.

These steps must be performed in sequence every epoch to ensure proper learning behavior.

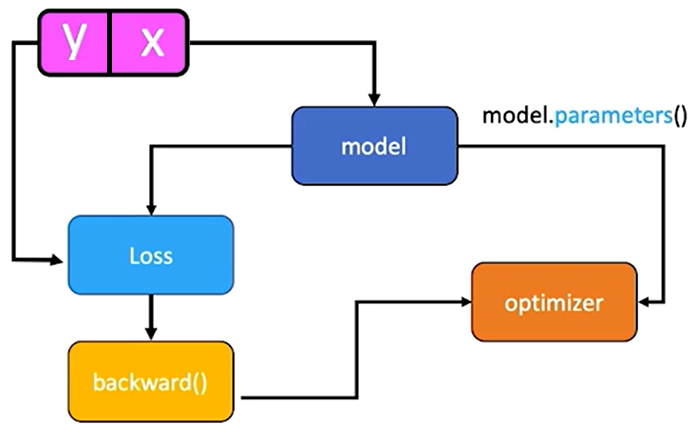
ℹ️ Note: Calling **optimizer.step()** essentially updates the parameters like we in manual way.



### 🔹 Diagrammatic Understanding

A conceptual flow of how components interact in PyTorch training:

* The **optimizer** is initialized with the model’s parameters.
* For each batch:
  + The model computes **ŷ** from input **x**.
  + The loss function computes the cost using ŷ and ground-truth y.
  + **loss.backward()** computes gradients.
  + **optimizer.step()** uses those gradients to adjust weights and biases.



Even though no explicit connection is coded between the optimizer and the loss, PyTorch’s autograd engine internally tracks operations and connects them via the computational graph.

This methodology forms the basis for most training procedures in PyTorch.

### ✅ Takeaways

✅ PyTorch optimizers like SGD handle the learning step in gradient descent.

✅ The **optimizer.step()** method abstracts and manages parameter updates.

✅ A standard training loop includes forward pass, loss computation, gradient reset, backpropagation, and parameter update.

✅ The optimizer operates over the computational graph created by PyTorch’s autograd system.

✅ This training methodology scales naturally to more complex models and optimizers.

# M. 3 – Section 3

**Training, Validation, and Test Split**

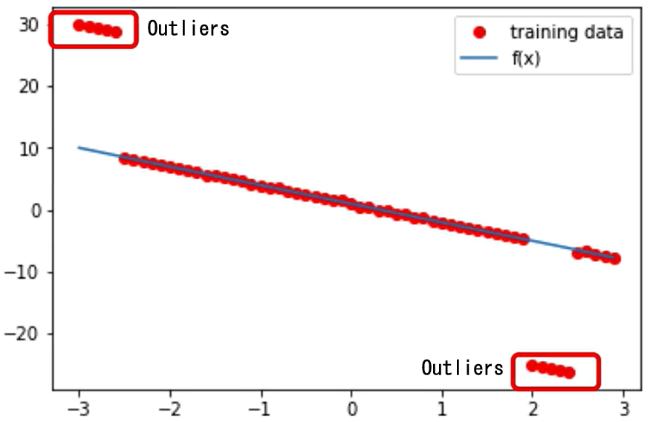
## 📌 Training, Validation, and Test Split

This section introduces the fundamental process of splitting a dataset into three distinct parts—training, validation, and test data—and describes their specific purposes in model development and evaluation.

### 🔹 Overfitting and the Need for Splitting

Overfitting happens when a model learns patterns specific to the training data, including noise or outliers, but fails to generalize to unseen data.

This behavior is common in complex models that perform very well on training data but poorly on data they haven't seen before.



To address this, the dataset is split into three subsets:

* **Training data** is used to learn model parameters (e.g., slope and bias).
* **Validation data** is used to evaluate different hyperparameter settings.
* **Test data** is used for final evaluation, simulating how the model performs in real-world scenarios.

The data split is often done randomly, but in demonstrations, it may be deterministic to ensure clearer understanding.

### 🔹 Training vs. Hyperparameter Tuning

Model training involves adjusting parameters like weights and biases through optimization techniques such as gradient descent.

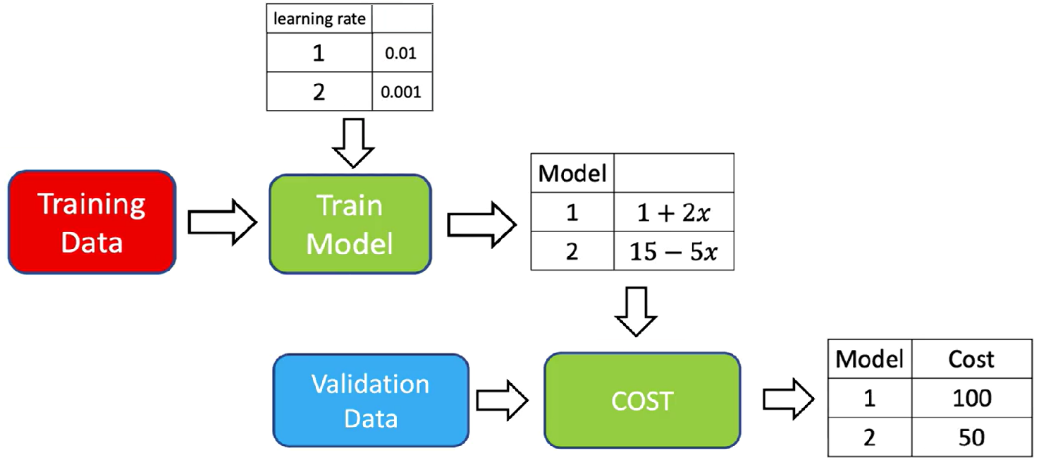
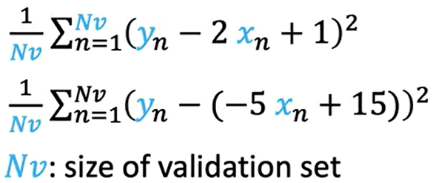
Hyperparameters like learning rate and mini-batch size are not learned, they are manually set and significantly affect the training process.

To find optimal hyperparameters:

* Train the model multiple times with different hyperparameter settings.
* Evaluate each resulting model using the **validation data**.
* Select the hyperparameter set that minimizes the validation cost.

For example, if two learning rates are tested:

* Each rate produces a different model.
* The validation loss is calculated for both.
* The model with the **lowest validation loss** is chosen—even if it doesn't minimize training loss.

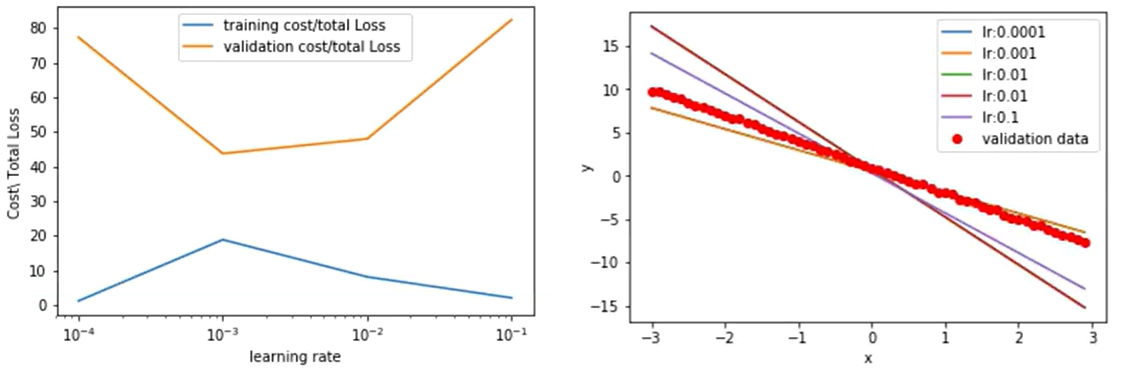


### 🔹 Validation vs. Training Cost

When choosing hyperparameters such as the learning rate, it's essential to evaluate model performance using **validation cost**, not just training cost. Relying on training cost can be misleading, as it only reflects how well the model fits the training data—not how well it generalizes.

This example demonstrates the importance of validation loss when tuning hyperparameters. The best model is the one that minimizes **validation cost**, not necessarily the one that fits training data most closely.

* The **left graph** shows total loss on the **training set** and **validation set** across different learning rates.
* The **right graph** shows the fitted regression lines (by learning rate) over the **validation data points**.



* + A learning rate of **0.1** yields the **lowest training loss**, but its validation loss is the **highest**, and the corresponding fitted line deviates significantly from the red validation points. This is a classic sign of **overfitting**.
  + A learning rate of **0.001** gives a **higher training loss** but results in the **lowest validation loss**, and the estimated line aligns well with the validation data. This indicates **better generalization**.

ℹ️ Selecting a model purely based on minimizing training loss may lead to choosing a model that performs **poorly on new data**.

### ✅ Takeaways

✅ Overfitting occurs when models perform well on training data but poorly on unseen data.

✅ Datasets are split into **training**, **validation**, and **test** sets to prevent overfitting and support robust evaluation.

✅ **Training data** is used to learn model parameters.

✅ **Validation data** is used to choose hyperparameters like learning rate and batch size.

✅ **Test data** is used only after finalizing the model to assess generalization.

✅ Hyperparameter tuning must rely on validation performance, not training performance.

✅ Proper data splitting and evaluation practices ensure model performance reflects real-world scenarios.

## 📌 Train, and Validate Models in PyTorch

This section outlines a structured approach to training, validating, and saving models using PyTorch.

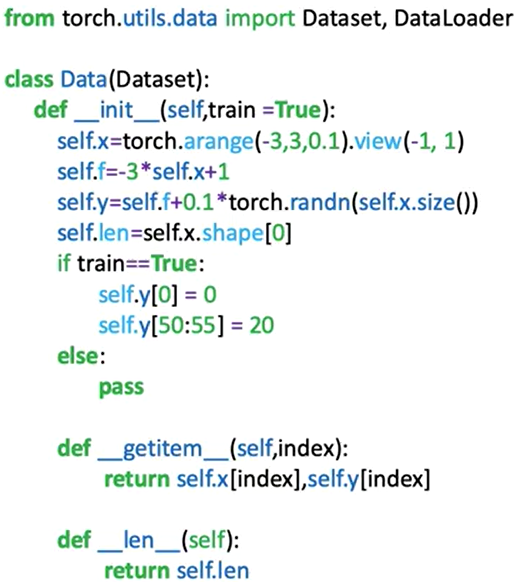
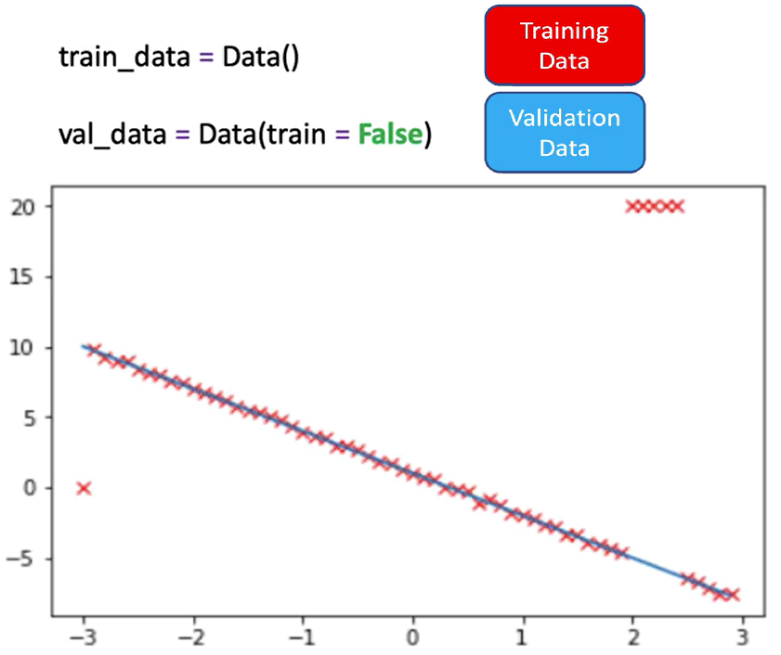
It emphasizes how learning rate selection, validation loss monitoring, and deterministic data splitting can be used to optimize model generalization.

The goal is to construct a model that fits well despite outliers, select the best performing configuration using validation data, and store the trained model for future use.

This process is critical for understanding hyperparameter tuning, validation-based selection, and scalable evaluation strategies.

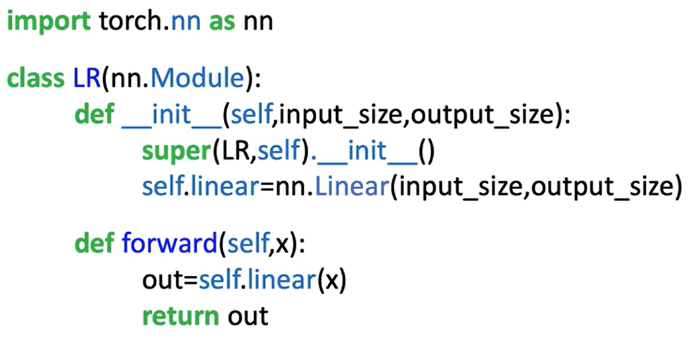
### 🔹 Data Creation and Splitting Strategy

* + Artificial data is generated using a custom Dataset class.
  + The dataset includes an **option to return training or validation data**, depending on the initialization flag.
  + **Training data includes outliers**, to intentionally simulate overfitting.
  + Two Dataset objects are created:
  + **train\_data** (with outliers)
  + **val\_data** (clean, used for evaluating generalization)
* These objects are visualized by plotting red training points over the original linear function to highlight the deviation caused by outliers.



### 🔹 Training Loop with Hyperparameter Evaluation

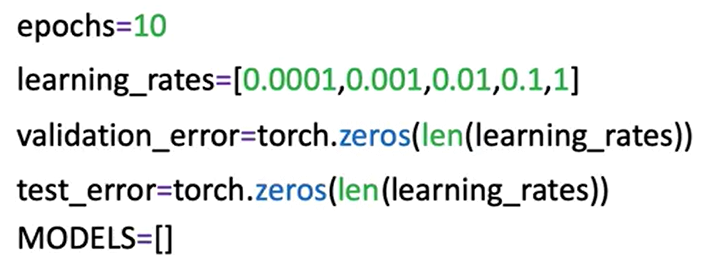
* A custom module is created to define the linear regression model.



* A **criterion** (loss function) and a **trainloader** are instantiated using PyTorch tools.



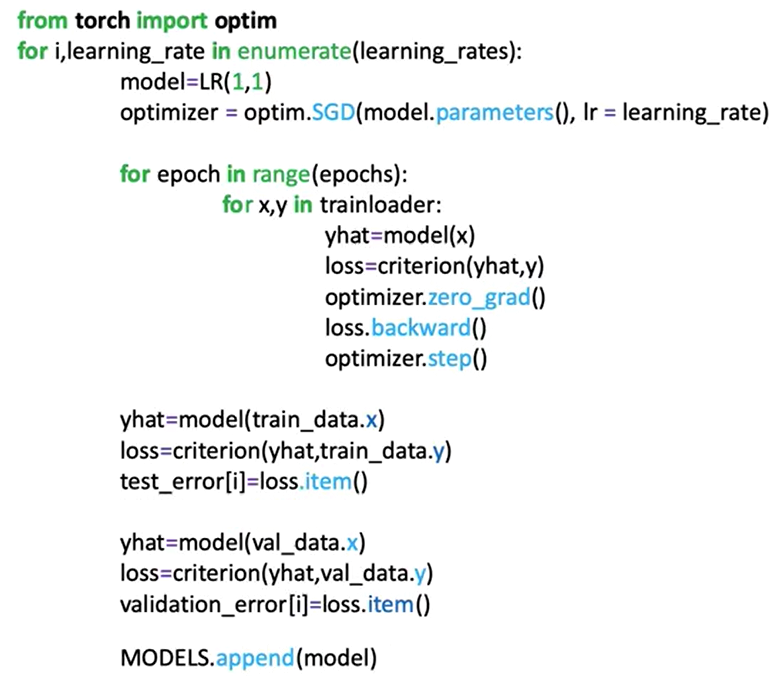
* **Only the learning rate is varied** in this procedure.
* Multiple hyperparameter trials are defined:
* A **list of learning rates** is created.
* Two tensors (**train\_cost**, **val\_cost**) are used to track losses for each learning rate.
* A list models is used to store each trained model instance.

****

**🔸** **Training Process:**

A **for** loop iterates through the learning rate list. For each learning rate:

* A new **model** and **optimizer** (SGD) are initialized.
* The model is trained for **10 epochs** using the training data.
* The **training loss is calculated** and stored:
* **train\_data.x** and **train\_data.y** are used to make predictions and compute the cost.
* **.item()** is used to extract a scalar from the PyTorch loss tensor.
* The **validation loss is calculated** on the full validation dataset. **val\_data.x**, and **val\_data.y** methods assumes all validation data can be loaded in memory. For large datasets, a DataLoader loop should be used instead.
* The trained model is appended to the **models** list.



**🔸** **Evaluation and Best Model Selection:**

After all learning rates are evaluated:

* Training and validation losses for each learning rate are **plotted**.
* The learning rate that results in the **lowest validation loss** is selected as the best.



Each model’s predictions on the validation data are visualized:

* The **optimal model line** is the one that fits the validation data most closely.
* This illustrates how the correct learning rate allows the model to generalize well despite noisy training data.



### ✅ Takeaways

✅ PyTorch optimizers like SGD handle the learning step in gradient descent.

✅ The **optimizer.step()** method abstracts and manages parameter updates.

✅ A standard training loop includes forward pass, loss computation, gradient reset, backpropagation, and parameter update.

✅ The optimizer operates over the computational graph created by PyTorch’s autograd system.