# Module 3

**Linear Regression PyTorch Way**

**Training, Validation, and Test Split**

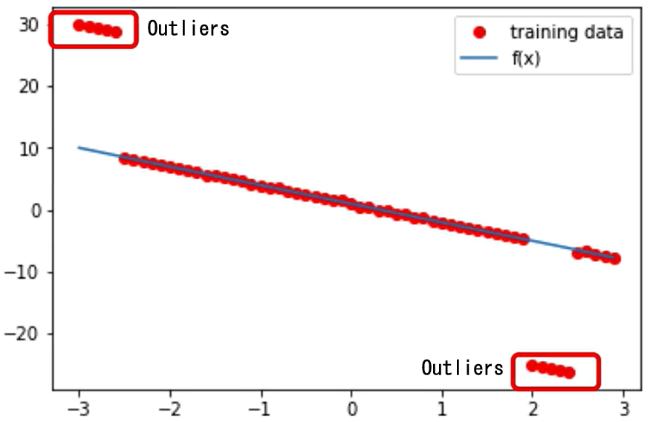
## 📌 Training, Validation, and Test Split

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.

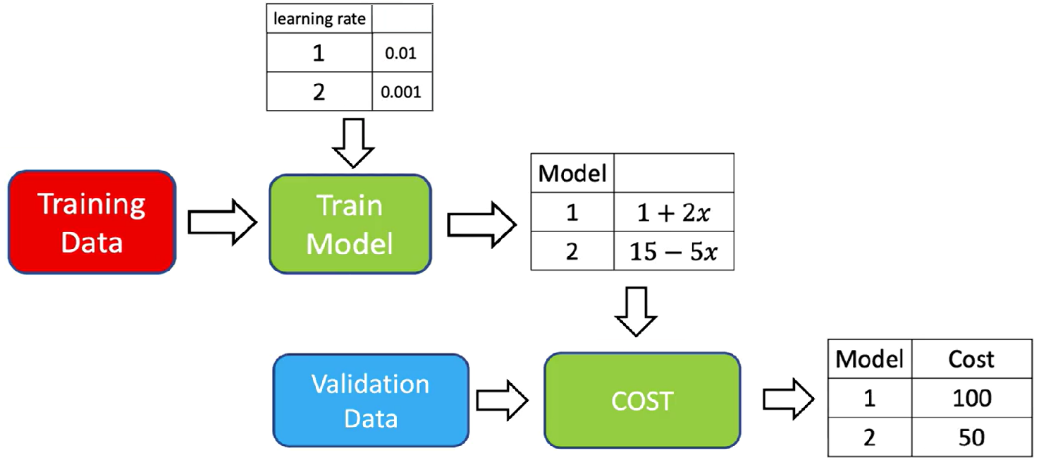
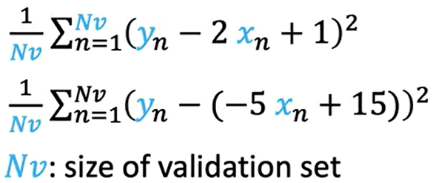
Hyperparameters like learning rate and mini-batch size are not learned, they are manually set and significantly affect the training process.

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.

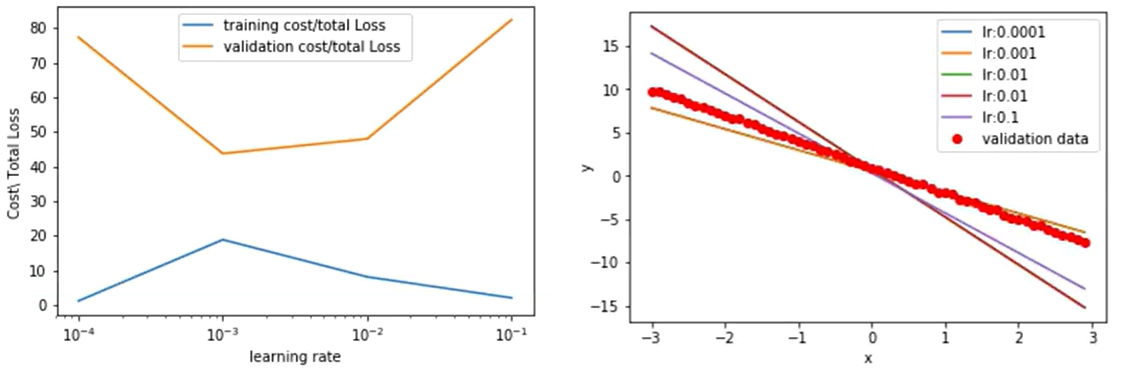


### 🔹 Validation vs. Training Cost

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.

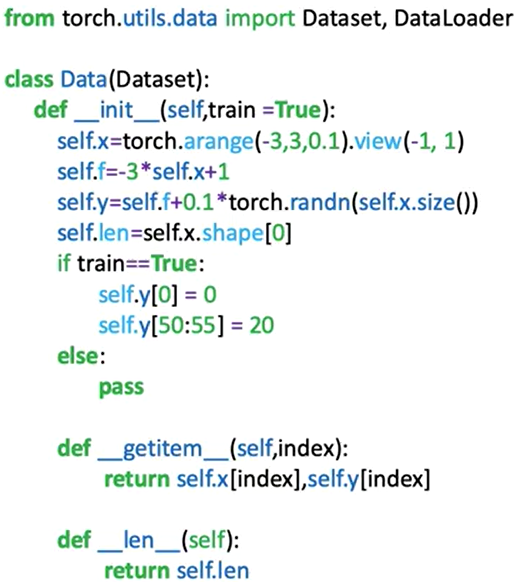
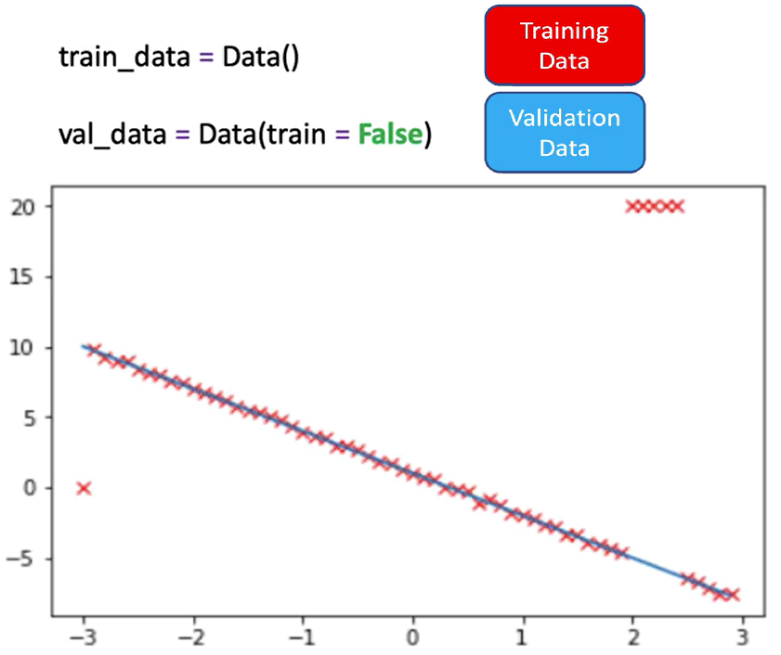
It emphasizes how learning rate selection, validation loss monitoring, and deterministic data splitting can be used to optimize model generalization.

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.

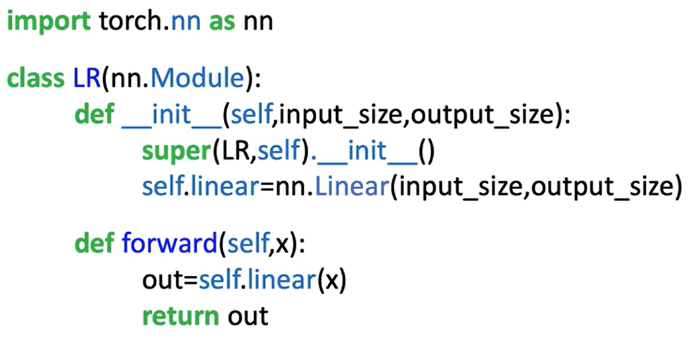
### 🔹 Data Creation and Splitting Strategy

* + 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

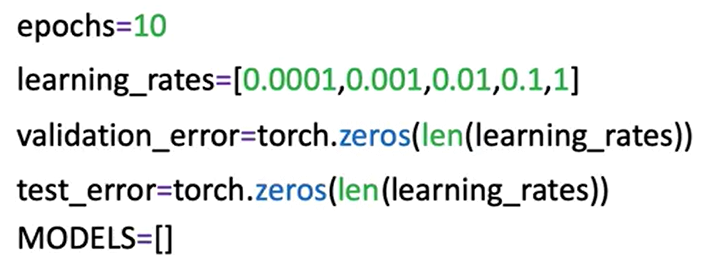
* A custom module is created to define the linear regression model.



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



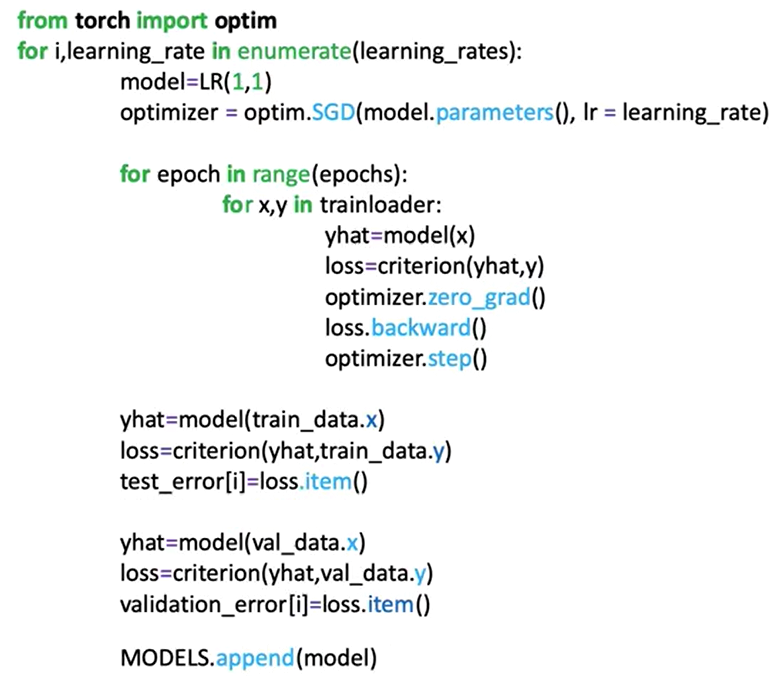
* **Only the learning rate is varied** in this procedure.
* Multiple hyperparameter trials are defined:
* A **list of learning rates** is created.
* Two tensors (**train\_cost**, **val\_cost**) are used to track losses for each learning rate.
* A list models is used to store each trained model instance.

****

**🔸** **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:**

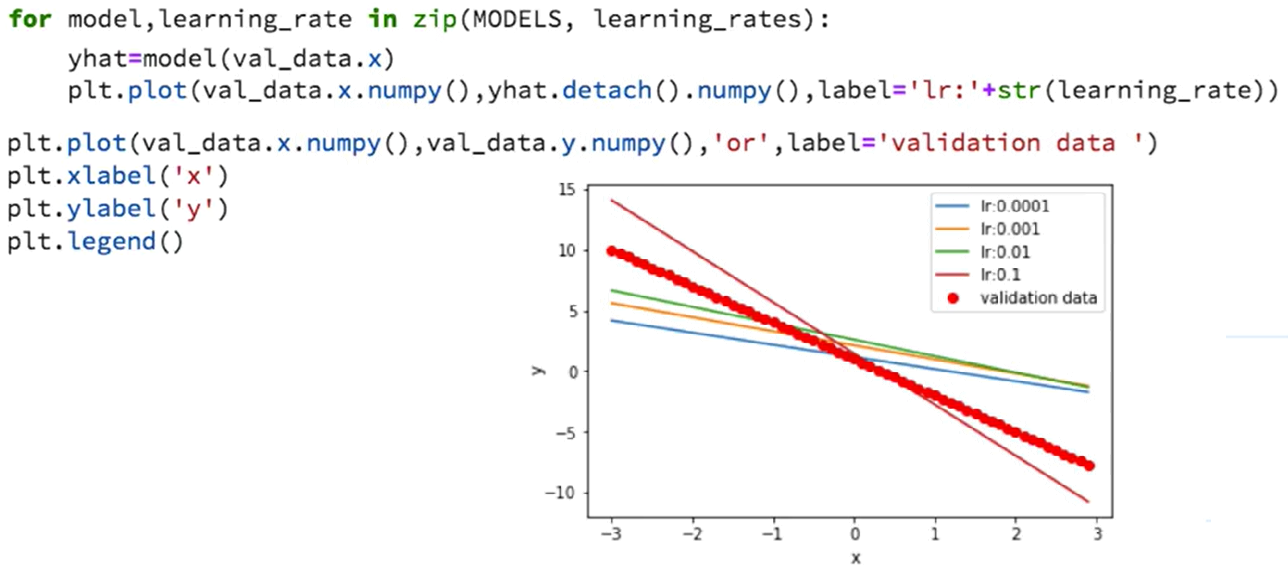
After all learning rates are evaluated:

* 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.

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