# Module 4

**Multiple Input / Output Linear Regression**

**Multiple Linear Regression Prediction**

## 📌 Multiple LR Prediction

This section introduces how to implement multiple linear regression using PyTorch for multiple input dimensions.

The focus is on building models using both nn.Linear and custom modules through nn.Module, while exploring shape consistency, vectorized operations, and how PyTorch handles parameter initialization and predictions for multiple samples.

### 🔹 Multiple Linear Regression in Multiple Dimensions

In multiple linear regression, the model predicts an output using multiple input features.

A single prediction is obtained using a linear combination of input features:

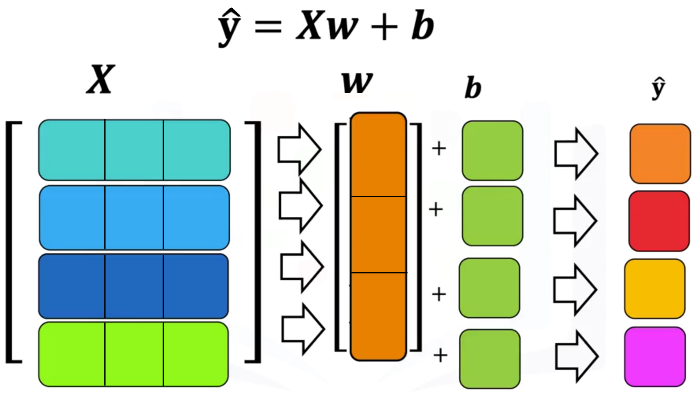
where:

* **x** is a 1 × D tensor (feature vector).
* **w** is a D × 1 tensor (weight vector).
* **b** is the bias (scalar).
* **ŷ** is the predicted value or dependent variable.

The **shape consistency** is key:

* Columns of X must match the number of weights.
* The same bias term b is added to each dot product.

For a matrix **X** with multiple rows (samples), each row is passed through the dot product with the parameter vector **w**, and the same bias **b** is added.

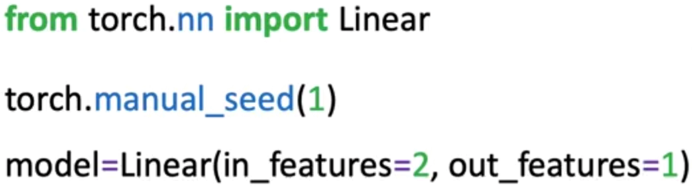
The process is repeated for each sample:

* First row ⋅ weights + bias → first prediction.
* Second row ⋅ weights + bias → second prediction.
* And so on.

The output is a vector of predictions (one per sample).

### 🔹 Linear Regression using nn.Linear

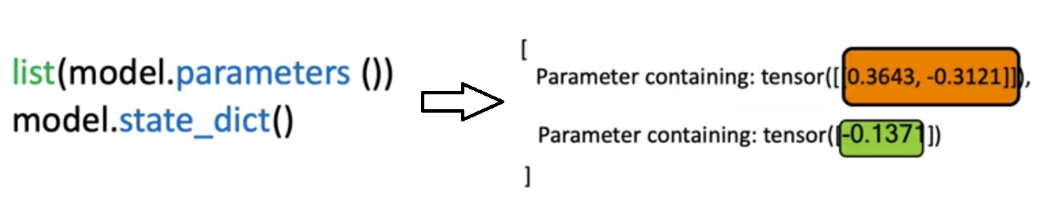
PyTorch’s built-in **nn.Linear** is used to define linear transformations.



* **in\_features** correspond to the number of input columns (features).
* **out\_features** correspond to the number of outputs (usually 1 for regression).

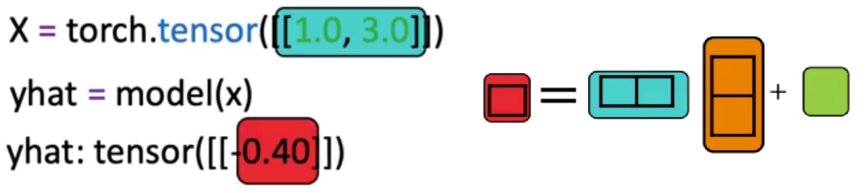
Internally, **nn.Linear**:

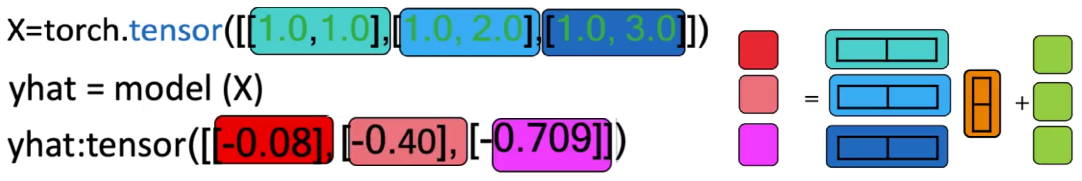
* Initializes weights and bias randomly.
* Stores parameters accessible via **.parameters()** (list() function must be applied to get an output as the method is lazily evaluated) or **.state\_dict()**.



Single-sample and batch predictions:

* A 1×D input returns a 1×1 output.
* An N×D input matrix returns an N×1 output (one prediction per sample).

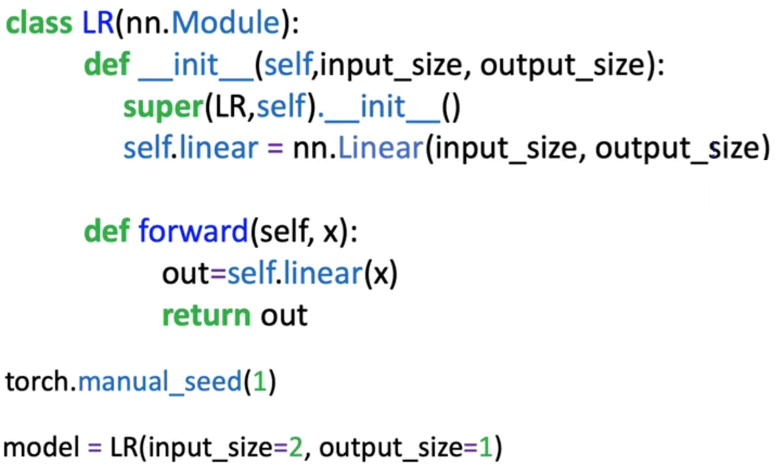




### 🔹 Custom Modules

For extensibility, PyTorch allows defining custom models using nn.Module.

The custom class inherits from **nn.Module** and defines:

* A constructor where the internal **nn.Linear** is created. **super().**\_\_**init()**\_\_ ensures **nn.Module** methods and attributes are inherited.
* A **forward** method to compute predictions.

Once defined, the object behaves like **nn.Linear**:

* It can be called on a tensor directly (e.g., **model(x)**).
* It supports both single-sample and batch inputs.

This setup will be crucial when constructing more complex neural networks.

### ✅ Takeaways

✅ Multiple linear regression generalizes to D-dimensional inputs using dot products and bias addition.

✅ Shape alignment is required: input features must match parameter dimensions.

✅ **nn.Linear** handles linear transformations efficiently and supports parameter access via **.parameters()** and **.state\_dict()**.

✅ Batch input handling returns a prediction per sample via matrix operations.

✅ Custom models built using **nn.Module** are essential for flexibility and will be used throughout neural network design.

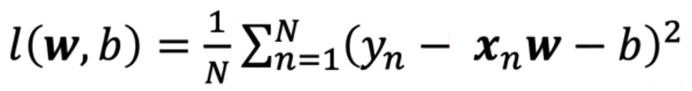
## 📌 Multiple LR Training with PyTorch

This section focuses on implementing the training procedure for multiple linear regression using PyTorch.

It introduces the full pipeline to train a model on 2D input data using PyTorch's **nn.Linear** class, **DataLoader**, and **autograd** functionality.

### 🔹 Cost Function and Gradient Descent in Multiple Dimensions

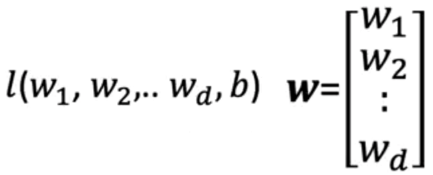
In multiple linear regression, the cost function measures the discrepancy between **predicted values ŷ** and **true targets y**. The cost is computed using the squared error:



Here, **ŷ** (xn x w – b) is computed as a **dot product** between the input feature vector x and weight vector w, plus a bias term b.

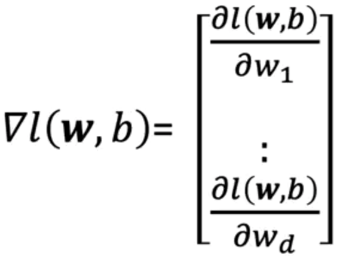
For an input of ***d*** dimensions:

* d 🡪 d weights + 1 bias = d + 1 parameters.

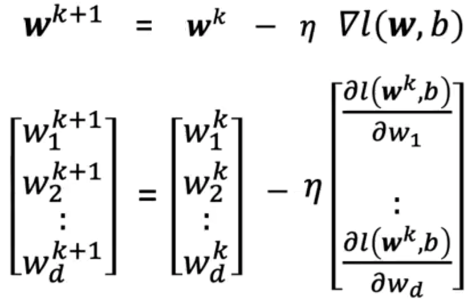


The gradient of the loss with respect to:

* Bias (b) is a scalar.
* Weights (w) is a vector of partial derivatives.



Weights update rules in vector form:



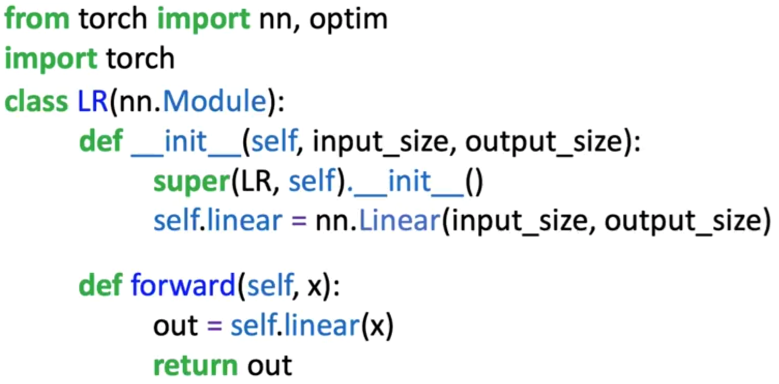
* η is the learning rate.

### 🔹 Training the Model with PyTorch

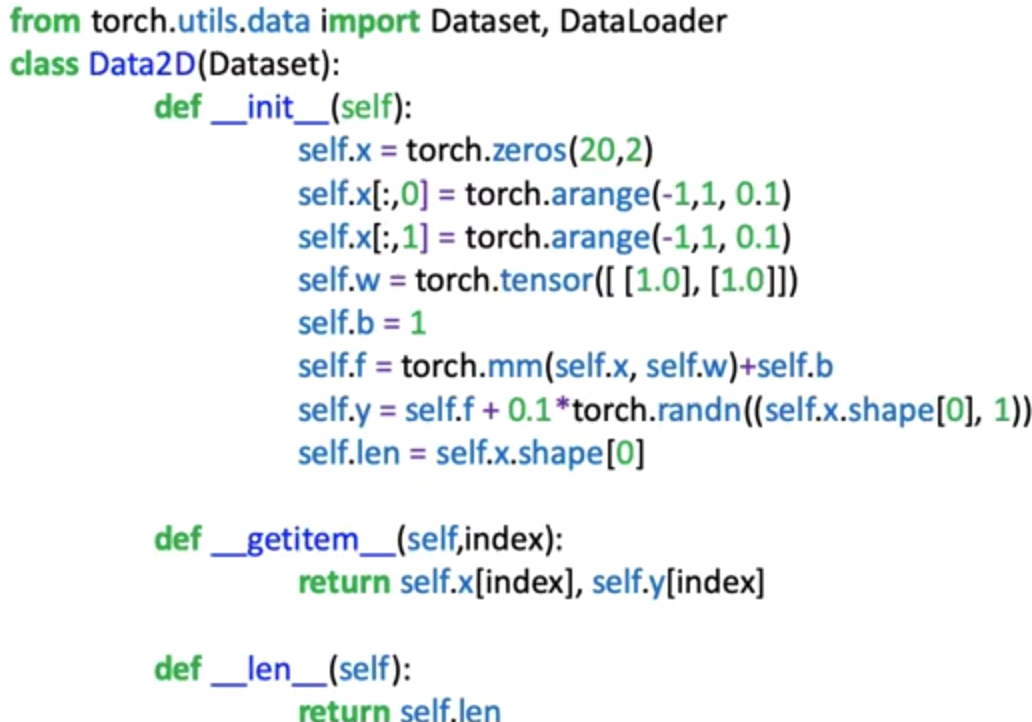
The training procedure follows the same structure as single-variable regression, in this example 2D input tensors will be handled.

Steps:

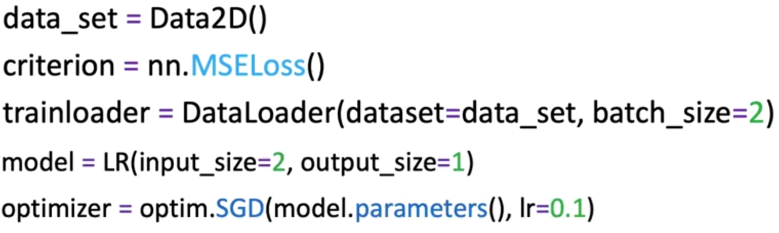
* 1. Create utility classes:
* Create linear custom model that inherits from **nn.Module** and behaves like **nn.Linear**.



* Define a dataset class to create the dataset object. It has two dimensions for the input x.

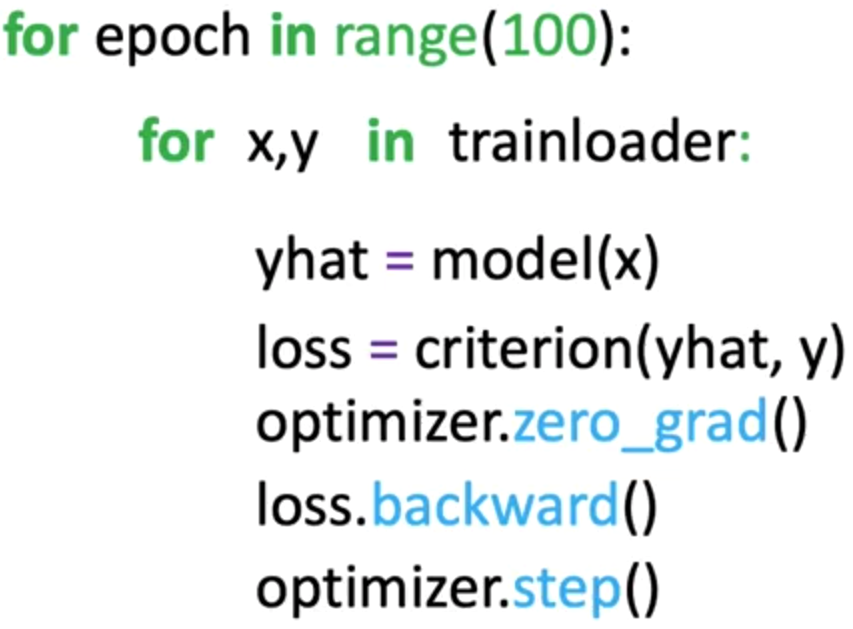


* 1. Define training parameters:
* Use the pre-defined class to create a dataset.
* Define the loss function or criterion.
* Create **DataLoader** object for mini-batch training.
* Instantiate the custom model.
* Define the **optimizer**.

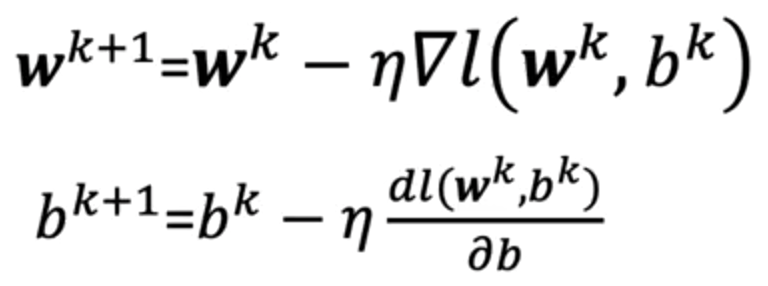


* 1. Loop through every epoch to update parameters:
* For each epoch:

1. Loop through each mini-batch of samples.
2. Perform a **forward pass**: compute predictions.
3. Compute the **loss** between predictions and ground-truth labels.
4. Set gradients to zero using **optimizer.zero\_grad()** to prevent accumulation.
5. Perform **backward pass**: compute gradients using **loss.backward()**.
6. Update model parameters with **optimizer.step()**.



* This process performs a **vectorized update** to all weights and bias simultaneously based on the computed gradients.



### ✅ Takeaways

✅ In multiple linear regression, each feature has a corresponding weight, and the model includes a bias term.

✅ The cost function generalizes to higher dimensions by summing squared errors across all feature combinations.

✅ Training with PyTorch involves:

* Defining a model using nn.Linear,
* Creating a DataLoader for batch processing,
* Using optimizer.step() and loss.backward() for gradient-based updates.

✅ After enough epochs, the model improves its ability to track the training data using the learned parameters.