# Module 4

**Multiple Input / Output Linear Regression**

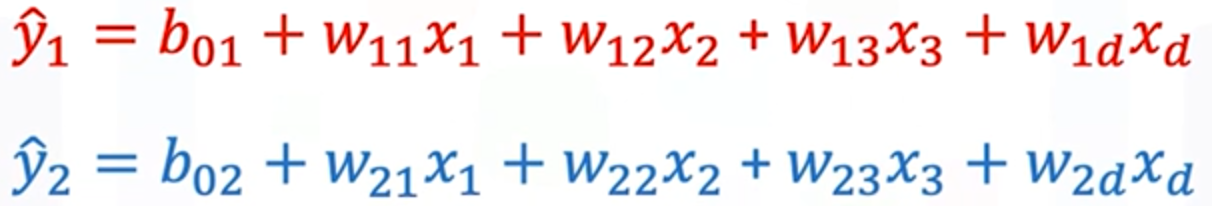
**Multiple Output Linear Regression**

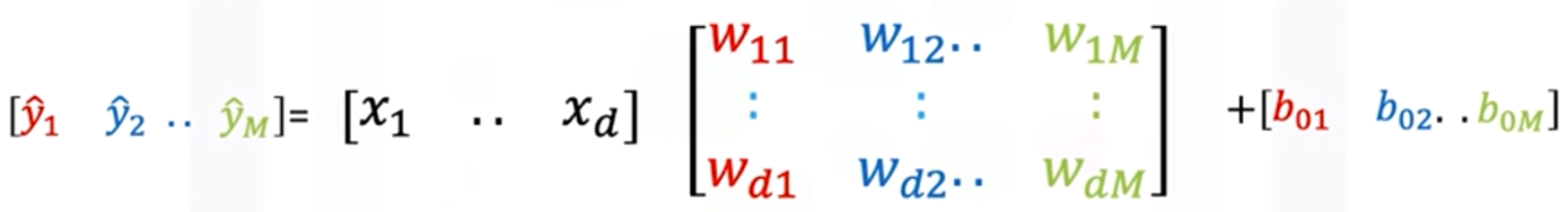
## 📌 LR Multiple Outputs

This section focuses on implementing **multiple-output linear regression** using PyTorch. It explains how to handle input tensors, matrix-based predictions, and parameter shapes when models produce multiple outputs.

### 🔹 Multiple Output Linear Functions

Linear regression with multiple outputs extends the single-output linear model by computing **multiple linear transformations** in parallel:





Each **output** corresponds to a separate linear function with its own parameters.

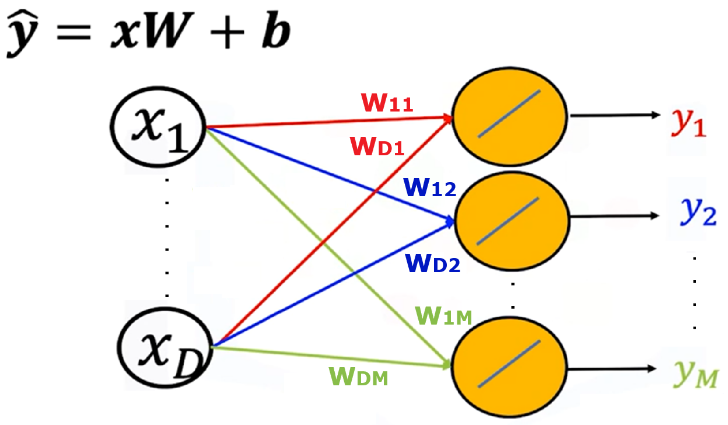
All the **weights** can be grouped into a matrix W, where:

* Each **column** in **W** represents the weights for a separate output.
* If there are **D** **input** features and **M** **outputs**:
* **W** is a **D×M** matrix.
* **b** is a **1×M bias** vector.
* Input **x** is a **1×D** vector.

The prediction ŷ is computed via:



* The result ŷ is a 1×M vector — one output per linear function.
* The computation is visually broken down as:
  + Perform the dot product of the input vector x with **each column of the matrix W**.
  + Add the corresponding **bias term**.
  + Each result is a **scalar output** for a specific linear function.

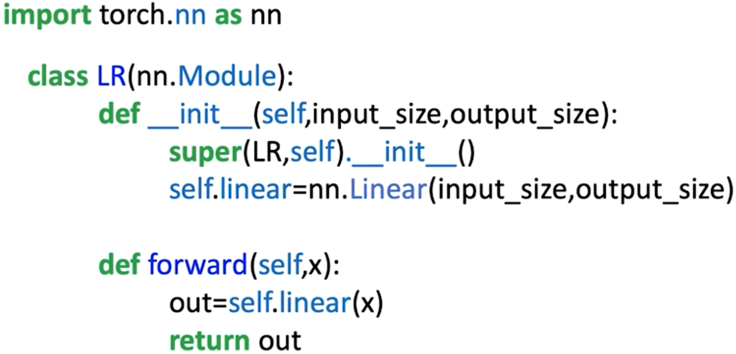
Directed graphs are used to represent:

* **Input features** as nodes.
* **Weights and biases** as edges.
* **Multiple outputs** as final nodes produced by distinct linear paths.

This structure generalizes cleanly to any number of outputs and features, enabling efficient multi-output regression.

### 🔹 Creating Custom Modules for Multi-Output Models

The custom module structure remains the same as before, but the parameter will be altered when the object model is created:

**input\_dim**: number of features.

**output\_dim**: number of outputs.

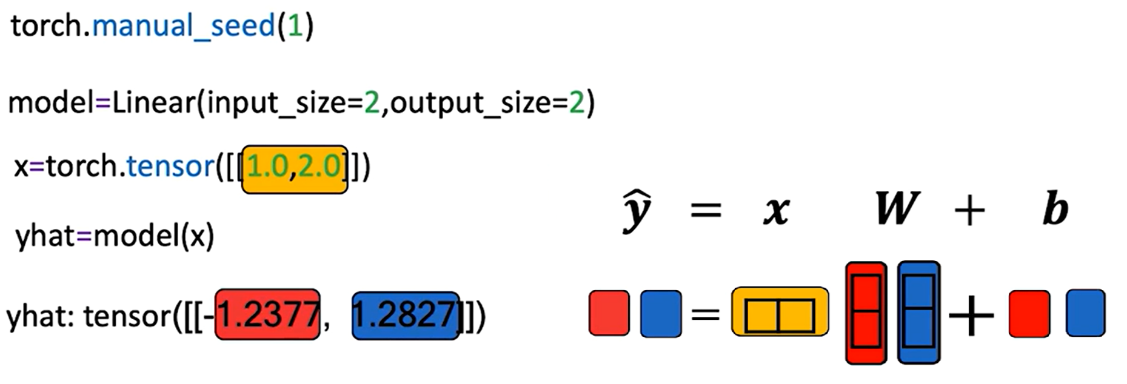
**nn.Linear(input\_dim, output\_dim)** is used to define the linear transformation layer.

The **.forward()** method handles the prediction call with **self.linear(x)**.

The model behaves like **nn.Linear**, but allows full customization and extension when needed.

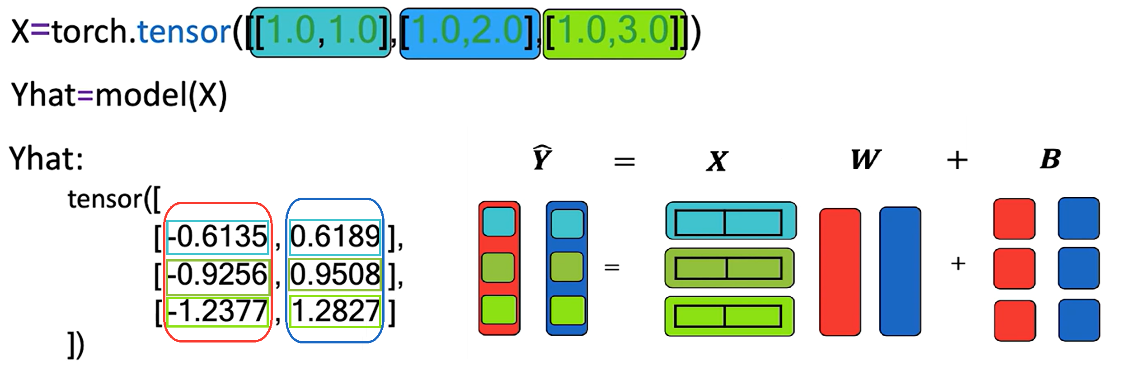
**🔸** Predictions for a **single input sample**:

* Input: 1×D tensor
* Output: 1×M tensor



**🔸**Predictions for **multiple sample**:

* Input: N×D tensor (N samples with D features)
* Output: N×M tensor (N predictions with M outputs each)



Each row in the output matrix Y corresponds to a sample.

* Each **row** of X is multiplied by W and then **biases** are added (broadcasted over all rows).
* The result is a matrix Y where:
  + **Rows** → input samples
  + **Columns** → output dimensions

The model uses **dot product and broadcasting** to generate predictions across all outputs and samples efficiently.

### ✅ Takeaways

✅ Multiple-output linear regression models produce **M parallel predictions** using a single matrix of weights and a bias vector.

✅ PyTorch’s nn.Linear and custom modules can easily accommodate multiple outputs by setting out\_features > 1.

✅ Matrix operations (dot products and broadcasting) are essential to compute predictions efficiently across samples and output dimensions.

✅ Custom modules maintain flexibility while maintaining compatibility with PyTorch's model API.

✅ The output of the linear transformation is a **2D tensor**, where rows represent samples and columns represent outputs.

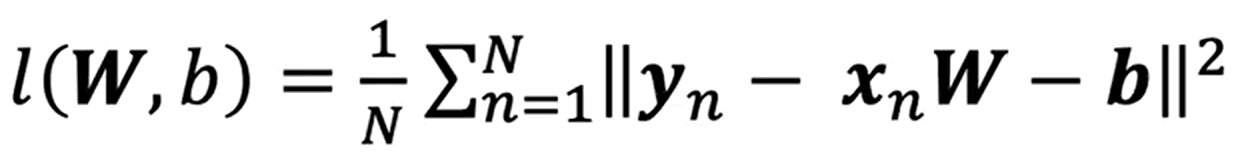
## 📌 Multiple Output Linear Regression Training

This section explains how to train a multiple-output linear regression model in PyTorch.

### 🔹 Cost Function for Multiple Outputs

When the model has **multiple outputs**, both the target values y and the predictions ŷ are vectors.

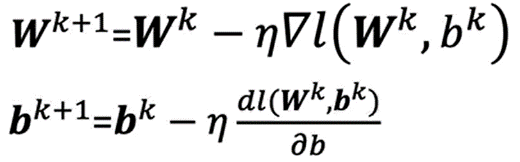
* The cost function is the **sum of squared distances** between the prediction vector and the target vector across all outputs:



This formulation accounts for all output dimensions simultaneously.

The weight parameter W is a **matrix** (one column per output), and the bias term b is a **vector** (one element per output).

The gradient descent update equations remain the same in principle, but the operations are **vectorized** to handle multiple outputs in parallel.



### 🔹 LR Training in PyTorch

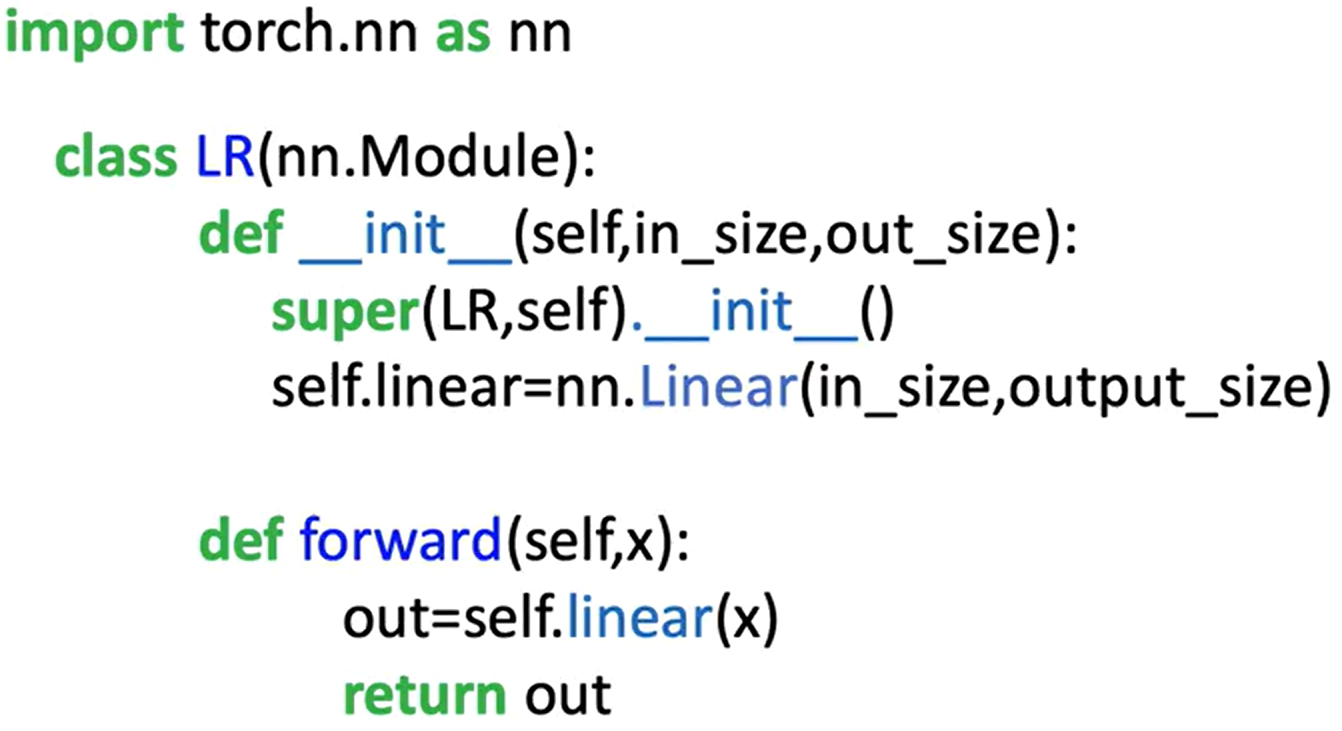
**🔸Model structure:**

The **custom module** or class for the model remains unchanged from the single-output case.

The key difference is in the parameters passed when creating the model:

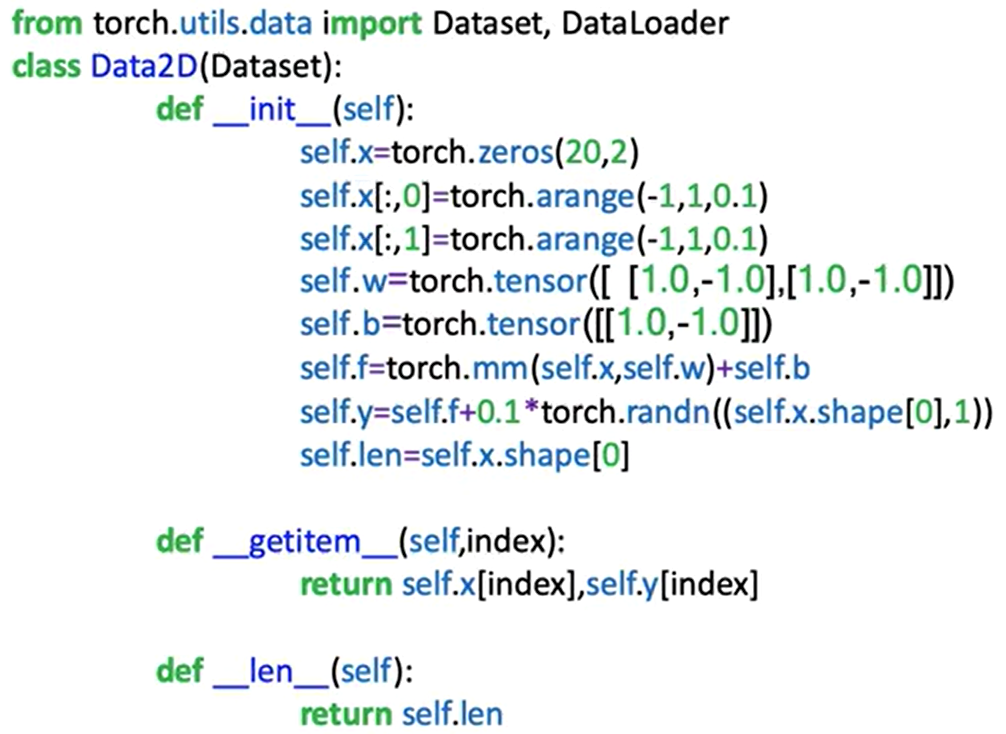
* + **in\_features** → number of input features.
  + **out\_features** → number of outputs.

This ensures the model’s **weight matrix W** and **bias vector b** are sized appropriately for multi-output predictions.



**🔸** **Dataset and DataLoader:**

The **dataset class** is adapted to generate **two target values** per sample instead of one.



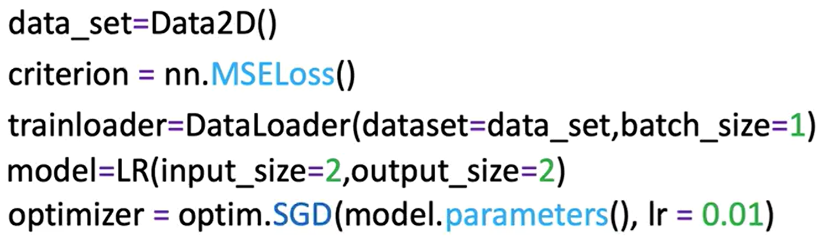
A dataset object is instantiated with both the feature matrix X and the multi-output target matrix Y.

**🔸** **Training Loop:**

The training procedure follows the standard PyTorch workflow:

1. **Initialize:**

* Create the model object with two input features and two output features.
* Define the cost function (criterion).
* Create the optimizer (e.g., SGD) with a specified learning rate.

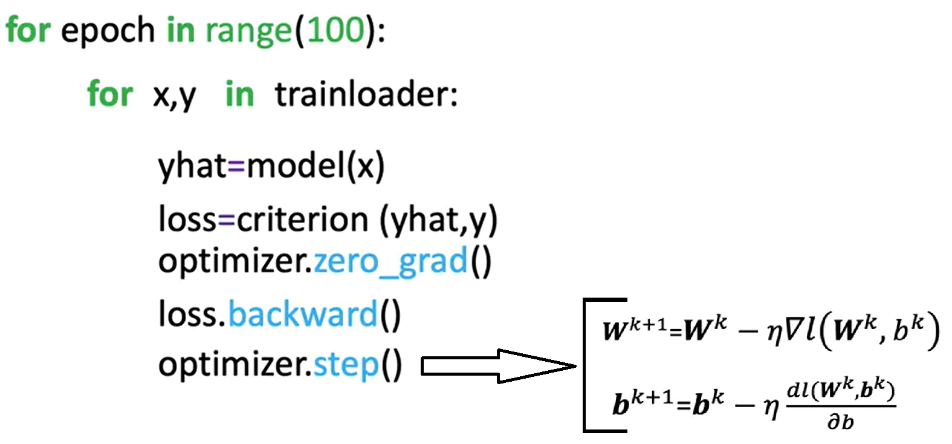


1. **Iterate over epochs**:

For each epoch:

* Loop over the **DataLoader** batches.
* **Forward pass**: Input batch → model → predictions (**ŷ**).
* **Compute loss**: Compare **ŷ** with target **y** using the cost function.
* **Zero gradients**: **optimizer.zero\_grad()** to reset gradient accumulation.
* **Backward pass**: **loss.backward()** to compute gradients of all parameters.
* **Update parameters**: **optimizer.step()** to adjust weights and biases.

The update step applies **matrix operations** that adjust all weights and biases for all outputs in one step.



### ✅ Takeaways

✅ In multi-output regression, the target y and predictions ŷ are vectors, requiring the cost function to sum squared differences across all outputs.

✅ The weight parameter W becomes a **matrix**, and the bias b becomes a **vector** when moving from single to multiple outputs.

✅ The PyTorch training loop for multiple outputs is essentially identical to the single-output case, with changes only in tensor shapes and dimensions.

✅ Vectorized updates allow all output parameters to be optimized simultaneously in each training step.

✅ The same custom model architecture can be reused for single or multiple outputs by adjusting the out\_features parameter when instantiating the model.