COURSE 4

Introduction to Neural Networks and PyTorch

INDEX

- ***Module 1 - Tensors and Datasets*** 1
  - ***M. 1 - Section 1 - Tensors 1D & 2D*** 1
    - **Overview of Tensors** 1
      - Tensors as Building Blocks 1
      - Examples of Tensors in Neural Networks 2
      - Tensor Conversion and Compatibility 2
      - Parameters and Derivatives in PyTorch 3
      - Dataset Class in PyTorch 3
    - **Tensors 1D** 4
      - Understanding 1D Tensors 4
      - 1D Tensor Operations 4
    - **Two-Dimensional Tensors** 10
      - Understanding 2D Tensors 10
      - Creating 2D Tensors 11
      - Indexing and Slicing 11
      - Basic Operations on 2D Tensors 12
  - ***M. 1 - Section 2 - Differentiation in PyTorch*** 13
    - **Differentiation in PyTorch** 13
      - Basic Derivatives 13
      - The Backward Graph and Tensor Attributes 14
  - ***M. 1 - Section 3 -* Dataset** 16
    - **Simple Dataset** 16
      - Creating a Dataset Class 16
      - Applying Transforms to a Dataset 17
      - Composing Multiple Transforms 18
    - **Image Dataset** 20
      - Constructing an Image Dataset 20
      - Building the Custom Dataset Class 21
      - TorchVision Image Transforms 22
- ***Module 2 - Linear Regression with PyTorch*** 23
  - ***M. 2 - Section 1 - Linear Regression Prediction with PyTorch*** 23
    - **Simple Linear Regression Prediction** 23
      - Concept of Linear Regression 23
      - Prediction Using Tensors 23
      - Built-in Linear Model with `nn.Linear` 24
      - Building a Custom Linear Module 25
      - Using `state_dict` for Parameter Access 26

- ***M. 2 - Section 2 - Linear Regression*** 27
  - **Linear Regression Training** 27
    - Defining the Dataset and Learning Objective 27
    - The Noise Assumption in Regression 27
    - The Goal of Training 28
  - **Loss in Linear Regression** 29
    - Role of Loss in Model Training 29
    - Defining Loss for a Single Sample 29
    - Systematic Minimization of Loss 30
  - **Gradient Descent and Cost** 31
    - What is Gradient Descent? 32
    - Gradient Descent in Practice 32
    - Problems with Learning Rate 33
    - When to Stop Gradient Descent 33
  - **Cost** 34
    - From Loss to Cost 34
    - Cost Function as a Function of Parameters 35
    - Applying Gradient Descent to Cost 35
    - Batch Gradient Descent 37
- ***M. 2 - Section 3 - PyTorch Slope*** 38
  - **Linear Regression in PyTorch** 38
    - Gradient Descent with PyTorch Tensors 38
    - Loss Calculation and Optimization Process 39
    - Epochs, Iterations, and Loss Reduction 40
    - Monitoring Loss Across Epochs 40
- ***M. 2 - Section 4 - Linear Regression Training with PyTorch*** 42
  - **PyTorch LR Training – Slope and Bias** 42
    - Cost Surface and Parameter Space 42
    - Contour Plots and Surface Slices 43
    - Gradient Descent in PyTorch (Manual Implementation) 44
    - Gradient Vector and Direction of Optimization 45

Module 3 - Linear Regression PyTorch Way 47

- ***M. 3 - Section 1 - Stochastic & Mini-Batch Gradient Descent*** 47
  - **Stochastic Gradient Descent and Data Loader** 47
    - Stochastic Gradient Descent Overview 47
    - Manual Implementation in PyTorch 48
    - DataLoader for SGD 50
  - **Mini-Batch Gradient Descent** 51
    - Mini-Batch Gradient Descent Overview 51
    - Epochs, Batches, and Iterations 52
    - Mini-Batch Gradient Descent in PyTorch 52
    - Convergence Rate and Batch Size 53

- ***M. 3 - Section 2 - Optimization in PyTorch*** 54
  - **Optimization in PyTorch** 54
    - Optimizer Setup in PyTorch 54
    - Diagrammatic Understanding 56
- ***M. 3 - Section 3 - Training, Validation, and Test Split*** 57
  - **Training, Validation, and Test Split** 57
    - Overfitting and the Need for Splitting 57
    - Training vs. Hyperparameter Tuning 57
    - Validation vs. Training Cost 58
  - **Train, and Validate Models in PyTorch** 59
    - Data Creation and Splitting Strategy 60
    - Training Loop with Hyperparameter Evaluation 60
- ***Module 4 - Multiple Input / Output Linear Regression*** 63
- ***M. 4 - Section 1 - Multiple Output Linear Regression*** 63
  - **LR Multiple Outputs** 63
    - Multiple Output Linear Functions 63
    - Creating Custom Modules for Multi-Output Models 64
  - **Multiple Output Linear Regression Training** 66
    - Cost Function for Multiple Outputs 66
    - LR Training in PyTorch 66
- ***M. 4 - Section 2 - Multiple Linear Regression Prediction*** 69
  - **Multiple LR Prediction** 69
    - Multiple Linear Regression in Multiple Dimensions 69
    - Linear Regression using `nn.Linear` 70
    - Custom Modules 71
  - **Multiple LR Training with PyTorch** 72
    - Cost Function and Gradient Descent in Multiple Dimensions 72
    - Training the Model with PyTorch 73
- ***Module 5 - Logistic Regression for Classification*** 75
  - **Linear Classifiers** 75
    - Representing Samples and Classes 75
    - Two-Class Linear Classifiers 75
    - Logistic Regression and the Sigmoid Function 76
  - **Logistic Regression Prediction in PyTorch** 78
    - The Logistic Function in PyTorch 78
    - Building Logistic Regression Models with `nn.Sequential` 79
    - Building Custom Custom Modules with `nn.Module` 80
    - Multi-Dimensional Logistic Regression 81

📌 **Bernoulli Distribution and Maximum Likelihood Estimation** 82
- 🔹 Bernoulli Distribution 82
- 🔹 Likelihood Function 83
- 🔹 Example of Bernoulli distribution and Likelihood 83
- 🔹 Maximum Likelihood Estimation (MLE) 85
- 🔹 Connection to Logistic Regression 85

📌 **Logistic Regression Cross-Entropy Loss** 86
- 🔹 Problem with Mean Squared Error in Classification 86
- 🔹 Maximum Likelihood Estimation and Logistic Regression 88
- 🔹 Cross-Entropy Loss 90
- 🔹 Logistic Regression Training in PyTorch 90

Module 1
Tensors and Datasets

M. 1 – Section 1
Tensors 1D & 2D

📌 Overview of Tensors

Tensors are the foundational data structures used to construct and operate neural networks in PyTorch.

A neural network is fundamentally a mathematical function that accepts one or multiple inputs, processes them, and returns one or more outputs.

In PyTorch, this processing is performed using **tensor operations**, which are generalized versions of familiar mathematical operations like addition and multiplication.

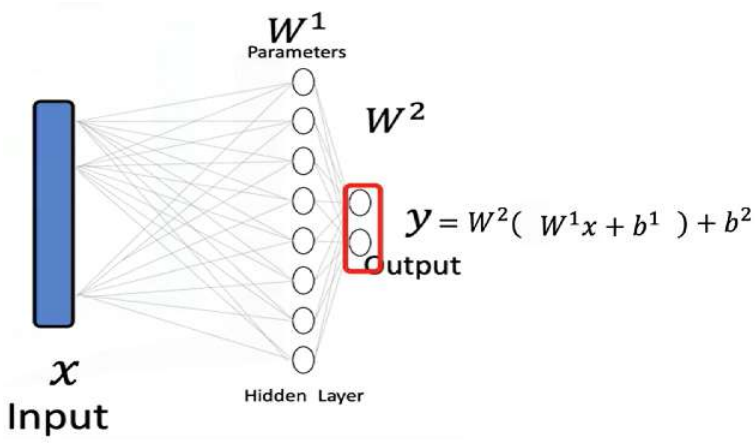
🔹 Tensors as Building Blocks

PyTorch tensors serve as the unified representation for **inputs**, **outputs**, and **parameters** within a neural network. These tensors can represent vectors, matrices, or higher-dimensional data structures, depending on the application.

Tensor operations in PyTorch form the computational backbone of how data is manipulated and learned from in neural networks.

PyTorch tensors are a generalized form of **numbers and dimensional arrays** in Python.

- The **input x** to a neural network is a tensor.
- The **output y** is also a tensor.
- The **parameters** of the model are tensors as well.
- Tensor operations allow the neural network to **transform inputs** into outputs during training and inference.

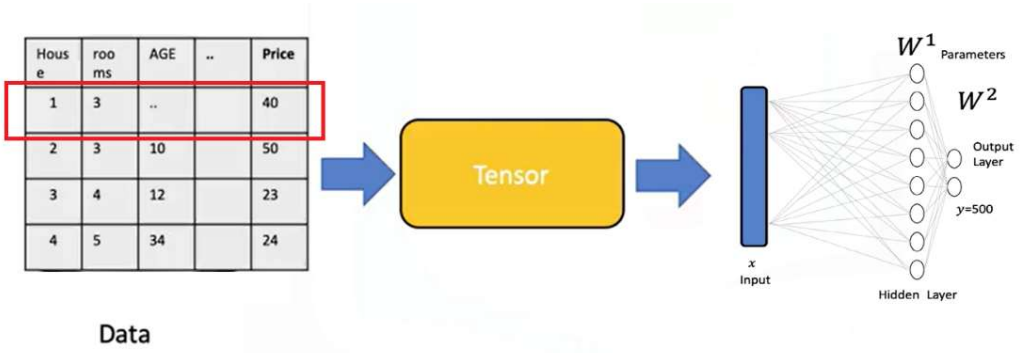


Neural networks use these tensor operations to apply **mathematical transformations**, often in the form of **vector and matrix operations**. These operations simulate the way real-world data is processed and are used throughout the course as the standard method for feeding data into neural models.

Examples of Tensors in Neural Networks

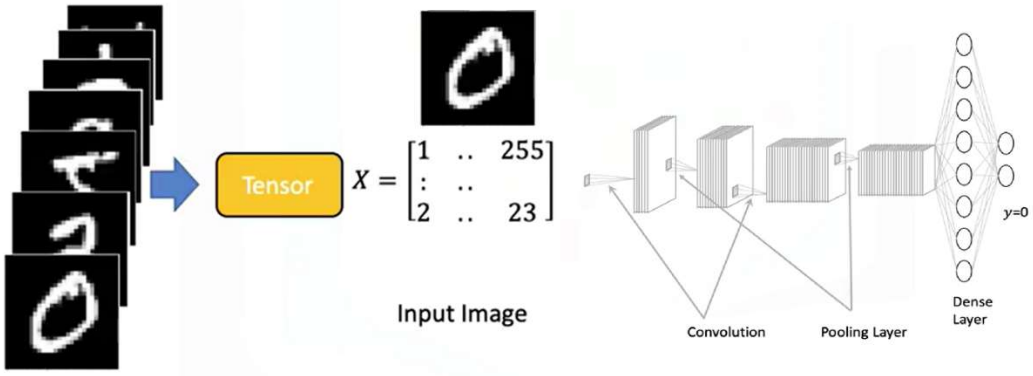
Databases can be treated as a series of tensors, where each row represents an input tensor (x) in a neural network.

A tensor is simply just a vector or a rectangular array consisting of numbers.



Images can be converted into 2D or 3D PyTorch tensors and used as input for classification tasks.

- Each tensor of the input is simply a matrix or rectangular array.
- Images are typically stored as arrays.
- Neural networks can receive these as tensors and perform classification based on the processed values.
- For instance, an image can be transformed into a tensor and classified as the digit **zero**.



Tensor Conversion and Compatibility

PyTorch tensors can be easily **converted to NumPy arrays**, and NumPy arrays can also be converted into PyTorch tensors.

This bidirectional conversion enables seamless operation within the **Python ecosystem** and allows integration with many existing Python libraries.

PyTorch also supports **GPU acceleration**, which is crucial for training large neural networks efficiently.

🔹 Parameters and Derivatives in PyTorch

Parameters in neural networks are specialized tensors that allow for the calculation of **gradients and derivatives**.

These gradients are essential for learning during training.

To enable gradient tracking, PyTorch tensors must be created with **`requires_grad=True`**.

This setting allows PyTorch to automatically compute derivatives during backpropagation.

🔹 Dataset Class in PyTorch

PyTorch provides a **Dataset class** that simplifies working with large datasets.

Using this class enables efficient data handling, transformation, and loading.

It is especially useful when building neural networks that require batch processing or data augmentation.

☑ Takeaways

☑ PyTorch tensors are the core data structures used in building and training neural networks.

☑ Inputs, outputs, and model parameters are all represented as tensors.

☑ Tensor operations in PyTorch generalize familiar mathematical operations and are essential for transforming input data.

☑ Databases and images can be represented as tensors and processed within neural networks.

☑ PyTorch integrates seamlessly with NumPy and supports GPU acceleration for scalable training.

☑ Setting requires_grad=True enables tensors to compute gradients, allowing for neural network training.

☑ The Dataset class simplifies data management and is essential for working with large-scale training data.

📌 Tensors 1D

🔹 Understanding 1D Tensors

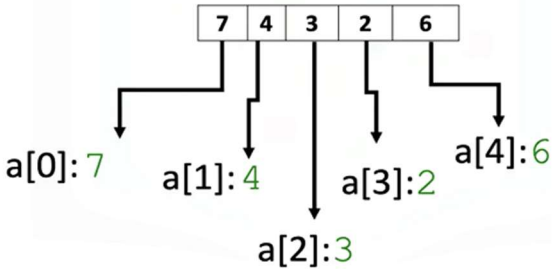
- A **0D tensor** represents a single number.
- A **1D tensor** is an array of numbers and can represent:
  - A row in a dataset
  - A vector
  - A time series
- A tensor contains elements of a **single data type**, there is a variety of different tensor types depending the data type of the elements in the tensor, such as:
  - float or double tensors (for real numbers)
  - byte tensors (for 8-bit images and unsigned integers)

Data type	dtype	Tensor types
32-bit floating point	torch.float32 or torch.float	torch.*.FloatTensor
64-bit floating point	torch.float64 or torch.double	torch.*.DoubleTensor
16-bit floating point	torch.float16 or torch.half	torch.*.HalfTensor
8-bit integer (unsigned)	torch.uint8	torch.*.ByteTensor
8-bit integer (signed)	torch.int8	torch.*.CharTensor
16-bit integer (signed)	torch.int16 or torch.short	torch.*.ShortTensor
32-bit integer (signed)	torch.int32 or torch.int	torch.*.IntTensor
64-bit integer (signed)	torch.int64 or torch.long	torch.*.LongTensor

🔹 1D Tensor Operations

- **Creating a tensor:**
  - Use `torch.tensor()` to convert the list into a tensor.
  - Data can be accessed via index.

```
import torch
a=torch. tensor([7, 4, 3, 2, 6])
```



- **Tensor Type and Data Type:**
  - Use the **`.dtype`** attribute to identify the data type stored in a tensor.
  - Use **`.type()`** to identify the tensor type.

```
import torch
a=torch. tensor([0,1,2,3,4] )
a.dtype
torch.int64
a.type()
torch.LongTensor
```

  - Explicitly set the data type using the **`dtype`** parameter.

```
a=torch. tensor( [0.0,1.0,2.0,3.0,4.0] , dtype=torch.int32 )
a.dtype ⇨ torch.int32
```

  - Using classes like **`torch.FloatTensor`**.

```
a=torch.FloatTensor( [0,1,2,3,4] )
a.type()⇨ torch.FloatTensor
a:tensor([ 0., 1., 2., 3., 4.])
```

  - Use **`.type(torch.FloatTensor)`** to convert to a float tensor.

```
a=torch. tensor( [0,1,2,3,4] )
a=a.type(torch.FloatTensor)
a.type() ⇨ torch.FloatTensor
```

- **Tensor Size and Shape:**
  - Use **`.size()`** to find the number of elements.
  - Use **`.ndimension()`** to find the number of dimensions (tensor rank).

```
a=torch.Tensor( [0,1,2,3,4] )
```

1	2	3	4	5

```
a.size() : torch.Size(5)
a.ndimension(): 1
```

- Convert a 1D tensor to 2D using **`.view(number_rows, number_cols)`**:
  - **`view(5, 1)`** turns a 1D tensor with 5 elements into a 2D column tensor.
  - Use **`view(-1,1)`** to let PyTorch infer dimensions.

```
a=torch. Tensor( [1, 2, 3, 4, 5])
```

1	2	3	4	5

```
a_col=a.view(5,1)
a_col=a.view(-1,1)
```

$$\begin{bmatrix}1\\2\\3\\4\\5\end{bmatrix}$$

Tensor Conversion with NumPy and Pandas

- Convert a NumPy array to a tensor with **`torch.from_numpy()`**
- Convert a tensor to a NumPy array using **`.numpy()`**

```
import torch
numpy_array =np. array( [0.0,1.0,2.0,3.0,4.0] )
torch_tensor =torch.from_numpy(numpy_array)
back_to_numpy=torch_tensor.numpy()
```

⚠ **Memory sharing:**
Modifying the original NumPy array affects the PyTorch tensor and vice versa.

- Convert Pandas series to tensor:
  - Use **`.values`** to get the NumPy array
  - Then apply **`torch.from_numpy()`**

```
pandas_series=pd.Series([0.1,2,0.3,10.1])
pandas_to_torch=torch.from_numpy(pandas_series.values)
```

- Convert tensor to list:
  - Use **`.tolist()`** to get a Python list

```
this_tensor=torch.tensor([0,1, 2,3])
torch_to_list=this_tensor.tolist()
torch_to_list: [0, 1, 2, 3]
```

- Convert tensor element to number:
  - Use **`.item()`** to extract a Python number from a single-element tensor

```
new_tensor=torch.tensor([5,2,6,1])

new_tensor[0]:tensor(5)
new_tensor[1]:tensor(2)

new_tensor[0].item():5
new_tensor[1].item():2
```

- **Indexing and Slicing:**
  - Access tensor elements with index (e.g., a[0])
  - Assign new values to specific elements (e.g., a[0] = 100)

```
c=torch. tensor([20,1,2,3,  4])
c:tensor([20,1,2,3,  4])

c[0]=100
c:tensor ([100,1,2,3,4])

c[4]=0
c:tensor ([100,1,2,3,0])
```

  - Slice a tensor like a list: a[1:3]
  - Assign values to slices (e.g., a[1:3] = [1, 2])

```
c:tensor ([100,1,2,3,0])
          0 | 1 | 2 | 3 | 4
d=c[1:4]
d:tensor ([1, 2, 3])
```

```
c: tensor([100,1,2,3,0])
          0 | 1 | 2 | 3 | 4
c[3:5]=torch.tensor([300.0,400.0])
c: tensor([100,1,2,300,400])
```

- **Basic Tensor Operations:**

  These operations are essential for building neural networks and understanding how tensors interact mathematically:

  - Vector Addition:

    Combine two tensors element-wise.

```
u=torch.tensor([1.0,0.0])
v= torch.tensor([0.0,1.0])

z=u+v

z: tensor([1, 1])
```

- Scalar Multiplication:
  Multiply each element of a tensor by a scalar.

```
y= torch.tensor([1,2])
z= 2*y

z: tensor([2,  4])
```

- Hadamard Product (Element-wise Multiplication)
  Multiply corresponding elements of two tensors.

```
u= torch.tensor([1,2])
v= torch.tensor([3,2])

z=u*v

z: tensor([3,  4])
```

- Dot Product
  Produces a single number that measures similarity between two vectors.

```
u=torch.tensor ([1,2])
v=torch.tensor ([3,1])

result = torch.dot(u,v)

result : 5
```

- Broadcasting
  Adding a scalar to a tensor adds it to each element.

```
u= torch.tensor([1,2,3,-1])
z=u+1

z:tensor
([2,3,4,0])
```

- **Universal Functions:**
  Apply operations across all elements:

  - **`a.mean()`**:
    Computes the average

```
a=torch.tensor([1,-1,1,-1])

mean_a=a.mean()
mean_a:0.0
```

- **`b.max()`**:
  Returns the maximum value

```
b= torch.tensor([1, -2,3,4,5])
max_b=b.max()
max_b:5
```

- Use functions like **`torch.sin()`** to apply to every element of a tensor

```
x=torch.tensor ([ 0 , np.pi/2, np.pi ] )
y= torch.sin(x)
y: tensor([ 0,1, 0])
```

- Use **`torch.linspace(start, end, steps)`** to generate evenly spaced values.

torch.linspace(-2,2, steps =5)

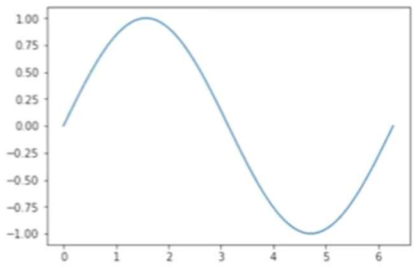
-2	-1	0	1	2

torch.linspace(-2,2,num=9)

-2	-1.5	-1	-0.5	0	0.5	1	1.5	2

- **Plotting with Tensors:**
  - Use **`matplotlib.pyplot`**
  - Use **`%matplotlib inline`** for inline notebook rendering
  - Convert tensors to NumPy before plotting: **`.numpy()`**

```
x=torch.linspace( 0 , 2*np.pi,100)
y= torch.sin(x)

import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(x.numpy(),y.numpy())
```



☑Takeaways

☑ **1D tensors** are core structures for data representation in PyTorch.

☑ Tensors can be **easily created, indexed, sliced, and reshaped** using intuitive syntax.

☑ PyTorch supports **type casting**, **NumPy/Pandas conversion**, and **interoperability** with Python tools.

☑ Tensor operations include **vector arithmetic**, **dot product**, **broadcasting**, and **universal functions** like mean and max.

☑ PyTorch allows mathematical functions (like sine) to be applied element-wise, enabling visualization and numerical analysis.

☑ Tools like linespace and matplotlib can be combined with PyTorch tensors for **function plotting and visualization**.

📌 Two-Dimensional Tensors

🔹 Understanding 2D Tensors

A 2D tensor is a container that holds numerical values of the same type and is typically visualized as a matrix.

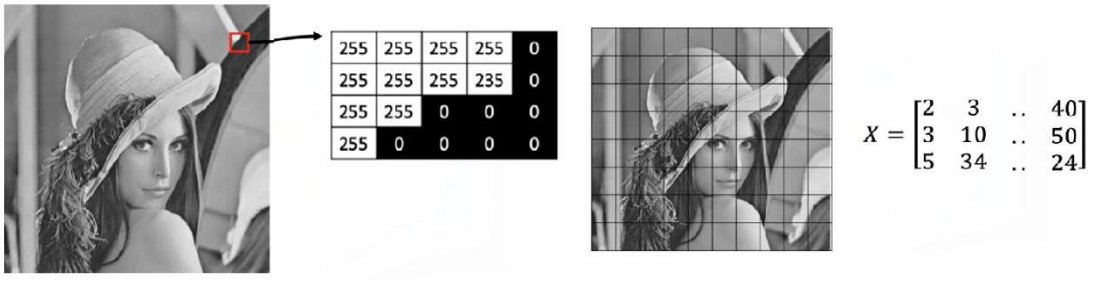
In real-world applications, 2D tensors can represent:

- **Tabular data**, where each row is a sample (e.g., a house), and each column is a feature (e.g., number of rooms, age, price).

House	rooms	AGE	..	Price
1	2	4		40
2	3	10		50
3	4	12		23
4	5	34		24

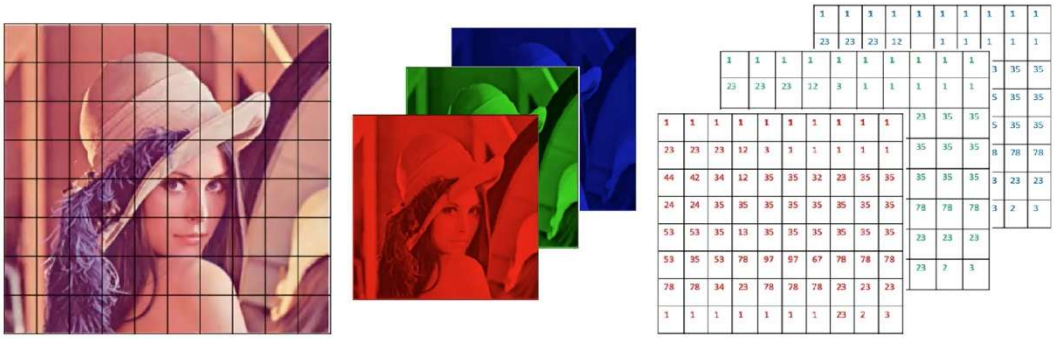
$$X = \begin{bmatrix} 2 & 4 & .. & 40 \\ 3 & 10 & .. & 50 \\ 5 & 34 & .. & 24 \end{bmatrix}$$

- **Grayscale images**, where pixel intensities range from 0 (black) to 255 (white), forming a 2D grid.



Tensors can be extended beyond two dimensions:

- **3D tensors** are used to represent color images, where each channel (red, green, blue) has its own 2D matrix of intensity values.



- Higher-dimensional tensors (e.g., 4D) are also used in deep learning.

Creating 2D Tensors

A 2D tensor can be created from a nested list, where each inner list represents a row.

The structure is interpreted as a rectangular matrix:

- The outer dimension represents rows.
- The inner dimension represents columns (eg: each element on a list, in this case since we have three elements in each list we have three columns).

Important tensor attributes:

- **Number of dimensions (rank)** can be queried to confirm the tensor structure.
- **Shape** returns the count of rows and columns.
- **Size** can be used interchangeably to obtain shape.
- **Number of elements** can be calculated by multiplying rows and columns or using a built-in method.

$a = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]$

A = torch.tensor(a)

$$A: \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

A.ndimension():2

A.shape:(3,3)
A.size() :(3,3)

A. numel() : 9

Indexing and Slicing

Indexing allows extraction of individual values, partial rows, or partial columns from the tensor for further computation or inspection.

It's performed using two indices:

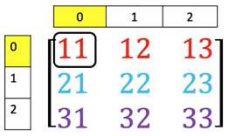
- The first index corresponds to the **row**.
- The second index corresponds to the **column**.

$A: [[A[0,0], A[0,1], A[0,2]], [A[1,0], A[1,1], A[1,2]][A[2,0], A[2,1], A\ [2,2]]]$

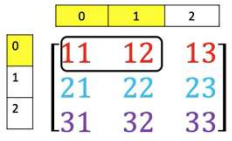
$$\begin{bmatrix} A[0,0] & A[0,1] & A\ [0,2] \\ A[1,0] & A[1,1] & A[1,2] \\ A[2,0] & A[2,1] & A[2,2] \end{bmatrix}$$

$A = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]$

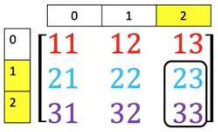
$A[0][0]$:11



$A[0,0:2]$: ([11, 12])



A[1:3,2]: ([23, 33])



Basic Operations on 2D Tensors

Addition:

- Two tensors of the same shape can be added together.
- This performs element-wise addition, similar to matrix addition in linear algebra.
- Each element in the result is the sum of the corresponding elements in the input tensors.

```
X= torch.tensor([[1,0],[0,1]])
Y= torch.tensor([[2,1][1,2]])
Z=X+Y;
Z:tensor([[3,1],
          [1,3]])
```

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$Y = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

$$Z = X + Y$$

$$Z = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

Scalar Multiplication:

- Multiplying a 2D tensor by a scalar scales each individual element.
- The resulting tensor is the same shape, but each value is multiplied by the scalar.

```
Y= torch.tensor([[2,1],[1,2]])

Z=2*Y;

Z:  tensor([[4,2],
            [2,4]])
```

$$Y = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

$$Z = 2Y = \begin{bmatrix} (2)2 & (2)1 \\ (2)1 & (2)2 \end{bmatrix}$$

$$Z = \begin{bmatrix} 4 & 2 \\ 2 & 4 \end{bmatrix}$$

Hadamard Product (Element-wise Multiplication):

- Multiplies corresponding elements of two tensors of the same shape.
- Produces a new tensor where each value is the product of the matching elements from the inputs.

```
X= torch.tensor([[1,0],[0,1]])
Y= torch.tensor([[2,1][1,2]])
Z=X*Y;
Z:  tensor([[2,0],
            [0,2]])
```

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$Y = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

$$Z = X \circ Y = \begin{bmatrix} (1)2 & (0)1 \\ (0)1 & (1)2 \end{bmatrix}$$

$$Z = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

Matrix Multiplication:

- Follows standard linear algebra rules:
  - The number of columns in matrix A must match the number of rows in matrix B.
- For each element in the resulting matrix:
  - Compute the dot product between a row from matrix A and a column from matrix B.
- The result is a new matrix with a shape defined by the row count of matrix A and the column count of matrix B.
- Matrix multiplication yields a meaningful transformation of input features, often used in neural network layers.

```
A= torch.tensor([[0,1,1],[1,0,1]])
B= torch.tensor([[1,1],[1,1],[-1,1]])
C= torch.mm(A,B);
C: tensor([[0,2],
           [0,2]])
```

☑ Takeaways

☑ 2D tensors are commonly used to represent both **structured data** (like spreadsheets or tables) and **images** (grayscale and multi-channel).

☑ Tensors can be **indexed, sliced, and reshaped** to access and manipulate specific data points or submatrices.

☑ Arithmetic operations like **addition**, **scaling**, **element-wise multiplication**, and **matrix multiplication** are supported natively and follow familiar linear algebra principles.

☑ 2D tensors provide the foundation for **layered neural network computations**, especially in the early stages of data processing and feature transformation.

☑ The structure and operations on tensors mirror real-world mathematical concepts, making them an intuitive and powerful abstraction for machine learning.

M. 1 – Section 2
Differentiation in PyTorch

📌 Differentiation in PyTorch

🔹 Basic Derivatives

A derivative represents the **rate of change of a function**.

Evaluating this derivative at a specific point (e.g., x = 2) gives the **slope of the function** at that point (2 × 2 = 4).

To compute derivatives in PyTorch:

- When creating x (a tensor) a value is specified, functions and derivatives of x are evaluated for the assigned value, in this case 2.
- When a tensor is created with **`requires_grad=True`**, PyTorch tracks all operations involving it to allow gradient computation later. It essentially tells PyTorch that the declared value will be used to evaluate functions and derivatives of x using the declared value.
- To differentiate a function defined with a tensor, **`.backward()`** function is called to trigger backpropagation.
- The result of this differentiation is stored in the **`grad`** attribute of the original input tensor, reflecting the value of the derivative at that specific input.

`x=torch.tensor(2,requires_grad=True)`	$x = 2$
`y=x**2`	$y(x) = x^2 \Rightarrow y(x=2) = 2^2$
`y.backward()`	$\frac{dy(x=2)}{dx} = 2x$
`x.grad`	$\frac{dy(2)}{dx} = 2(2)$

The Backward Graph and Tensor Attributes

PyTorch supports automatic differentiation by attaching metadata to tensors. **The backward graph is essentially composed of metadata from multiple tensors and ops**, arranged in a way that allows gradient computation.

Basically, the **backward graph** relies on tensor metadata to **compute gradients** correctly.

PyTorch constructs a **backward graph**, where tensors and operations (backward function for example) are nodes. This structure allows tracing back through computations to evaluate derivatives.

Based upon whether a particular tensor is a leaf or not in the graph, pytorch evaluates the derivative of that tensor.

If the leaf attribute for a tensor is set to True, pytorch won't evaluate its derivative.

Each tensor has important attributes:

- **Data**: Holds the actual numerical value.
- **Grad**: Stores the computed derivative once calculated.
- **Grad_fn**: Points to the function used to generate the tensor.
- **Is_leaf**: Indicates whether the tensor is a leaf node in the graph.
- **Requires_grad**: Signals that gradients should be tracked for this tensor

ℹ️ PyTorch evaluates gradients using this graph, determining how changes in input tensors affect output tensors.

x

Attributes	Values
Data	2
grad_fn	None
grad	8
is_leaf	True
requires_grad	True

$(\cdot)^2$

y

Attributes	Values
Data	4
grad_fn	PowBackwar
grad	None
is_leaf	False
requires_grad	True

- **Single Variable Differentiation:**

  PyTorch allows gradient to be computed automatically:

  - Define the tensor x with gradient tracking enabled.
  - Define z in terms of x.
  - Trigger backpropagation.
  - Access the result through the .grad attribute.

```
x = torch.tensor(2,requires_grad=True)
z = x**2 + 2*x + 1
z.backward()
x.grad = 6
```

$$z(x) = x^2 + 2x + 1$$

$$z(x = 2) = 2^2 + 2(2) + 1$$

$$\frac{dz(x = 2)}{dx} = 2x + 2$$

$$\frac{dz(2)}{dx} = 2(2) + 2$$

- **Partial Derivatives for Multivariable Functions:**

  Partial derivatives measure the change of a function with respect to one input variable, holding others constant.

  - Consider a function $\boldsymbol{f(u,v) = uv + u^2}$. The partial derivatives are:

respect to u	respect to v
$\frac{\partial f(u,v)}{\partial u} = \mathrm{v} + 2\mathrm{u}$	$\frac{\partial f(u,v)}{\partial v} = \mathrm{u}$

  - PyTorch can compute both partial derivatives by defining both input tensors with requires_grad=True, constructing the function f, calling the differentiation trigger, and then accessing the gradients for each input separately.

```
u=torch.tensor(1,requires_grad=True)
v=torch.tensor(2,requires_grad=True)
f=u*v + u**2
f.backward()
u.grad = tensor(4)
v.grad = tensor(1)
```

$$f(u = 1, v = 2) = uv + u^2$$

$$f(u = 1, v = 2) = 1(2) + 1^2 = 3$$

$$\frac{\partial f(1,2)}{\partial u}, \frac{\partial f(1,2)}{\partial v}$$

$$\frac{\partial f(u,v)}{\partial u} = \mathrm{v} + 2\mathrm{u} \Rightarrow \frac{\partial f(u = 1, v = 2)}{\partial u} = 2 + 2(1)$$

$$\frac{\partial f(u,v)}{\partial v} = \mathrm{u} \Rightarrow \frac{\partial f(u = 1, v = 2)}{\partial v} = 1$$

☑Takeaways

☑ PyTorch automates differentiation by building a **computational graph** that tracks how tensors are connected through operations.

☑ Tensors with gradient tracking enabled can be used to compute derivatives using **backward propagation**.

☑ **Single-variable derivatives** and **partial derivatives** are both supported.

☑ Gradients are accessed directly from the input tensors once calculated.

☑ Tensor attributes such as **`grad`**, **`grad_fn`**, and, **`is_leaf`** are essential for managing and understanding gradient flows.

☑ Automatic differentiation is critical for training neural networks using optimization techniques like gradient descent.

M. 1 – Section 3
Dataset

📌Simple Dataset

🔹 Creating a Dataset Class

A dataset object is created by subclassing the abstract Dataset class provided by PyTorch.

Within the constructor:

- Input features and target values are stored as tensors (x and y), each containing 100 samples. The values are created in the object constructor and assigned to the **`self.x`** and **`self.y`** tensors
- The total number of samples is stored in a length attribute.

The dataset class overrides two core methods:

- **`__len__`**: Returns the number of samples.
- **`__getitem__`**: Accepts an index and returns a tuple of feature and target tensors corresponding to that index.

```
from torch.utils.data import Dataset

class toy_set(Dataset):
    def __init__(self, length=100,transform=None):
        self.x=2*torch.ones(length,2)
        self.y=torch.ones(length,1)

        self.len= length
        self.transform = transform

    def __getitem__(self,index):
        sample= self.x[index] ,self.y[index]
            if self.transform:
                sample= self.transform(sample)
        return sample

    def __len__(self):
        return self.len
```

dataset=toy_set()

self.x	[2,2]	[2,2]	[2,2]	..	[2,2]	[2,2]
self.y	1	1	1	..	1	1
Index	0	1	2	..	98	99

- **Accessing Data Samples:**

  Individual samples are retrieved using square brackets, which act as a proxy for the __getitem__ method.

  This method returns a tuple:

  - The first element corresponds to a feature tensor.
  - The second element corresponds to a target tensor.

```
len(dataset)  ⟹  100
dataset[0]    ⟹  (tensor([ 2., 2.]), tensor([ 1.]))
```

  The dataset behaves like an iterable. It can be accessed using index notation or through iteration in a loop:

  - Iterating over the dataset triggers repeated calls to __getitem__, returning one sample per iteration.

```
for i in range(3):
    x,y=dataset[i]
    print(i,'x:',x,'y:',y)
```

⟹

```
0 x: tensor([ 2., 2.]) y: tensor([ 1.])
1 x: tensor([ 2., 2.]) y: tensor([ 1.])
2 x: tensor([ 2., 2.]) y: tensor([ 1.])
```

Applying Transforms to a Dataset

Transformations can be applied to samples using **callable classes** instead of standalone functions.

These classes define a **`__call__`** method, allowing them to behave like functions when passed to the dataset.

Custom Transform Class

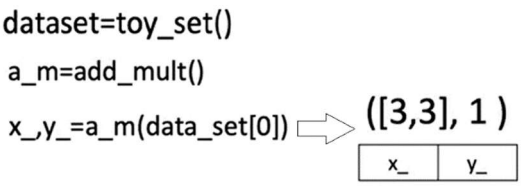
- A custom transformation class is defined with two parameters:
  - One to add a constant to the feature tensor.
  - One to multiply the target tensor by a constant.
- When a sample is passed to this transformation object, the transformation is applied and the modified tensors are returned as a tuple.

```
class add_mult(object):
    def __init__( self,addx=1,muly=1):
        self.addx=addx
        self.muly=muly

    def __call__(self, sample):
        x=sample[0]
        y=sample[1]
        x= x+self.addx
        y=y*self.muly
        sample=x,y
        return sample
```

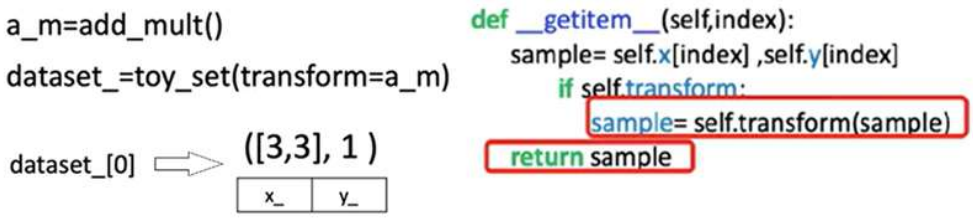
- **Applying Transforms to Dataset Samples:**

  There are two methods for applying a transformation:

  1. **Manual Application**:
     - The transformation object is created separately.
     - The object is manually applied to a sample retrieved from the dataset.
     - Only the selected sample is transformed.



  2. **Automatic Application via Constructor:**
     - The transformation object is passed to the dataset class during initialization.
     - Inside the dataset class, the transform parameter is assigned.
     - During each call to **`__getitem__`**, the transformation is applied automatically to every sample.
     - This ensures that the transformation is consistently applied across all retrieved data.



Composing Multiple Transforms

PyTorch provides a Compose class for chaining multiple transformations.

A list of transformation objects is passed to the Compose constructor.

When a sample is passed to the composed transform:

- The first transformation is applied.
- The output is passed to the second transformation.
- The final output is returned as a transformed tuple of tensors.

This compose object can be passed into the dataset class, enabling **automatic application of multiple transformations** during sample retrieval.

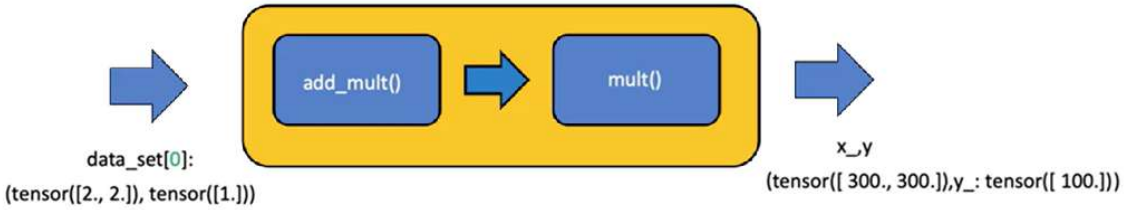
```
class mult(object):
    def __init__( self,mul=100):
        self.mul=mul

    def __call__ (self, sample):
        x=sample[0]
        y=sample[1]
        x= x *self.mul
        y=y*self.mul
        sample=x,y
        return sample
```

Let's say we would like to apply another transform, the class "**mult**" will multiply all the elements of a tensor by the value mul.

1. **Manual Application**:

   In the constructor, we place a list. The first element of the list is the constructor for the first transform, the second element of the list is the constructor for the second transform.

```
from torchvision import transforms
data_transform = transforms.Compose([add_mult(),mult()])
x_,y_=data_transform(data_set[0])
```



2. **Automatic Application via Constructor:**

   The compose object can be applied directly in the dataset constructor, each time a sample is retrieved, the original tensor is passed to the compose object (the first transform is applied, then the second transform is applied).

```
data_set_tr=toy_set(transform=data_transform)
```

☑ Takeaways

☑Custom dataset objects can be built by subclassing PyTorch's Dataset class and implementing the length and indexing methods.

☑Data stored in tensors can be accessed, indexed, and iterated over in a structured and repeatable way.

☑Transformations can be implemented as callable classes for better modularity and reuse.

☑Applying transformations during dataset construction enables efficient preprocessing at the data loading stage.

☑Multiple transformations can be composed using PyTorch's Compose utility, allowing sequential data processing in a clean and scalable manner.

📌 Image Dataset

This section explains how to build a custom dataset class for image data using PyTorch, how to preprocess image inputs using TorchVision transforms, and how to work with TorchVision's built-in datasets.

🔹 Constructing an Image Dataset

To construct an image dataset, the process begins by importing libraries from PyTorch, Pandas, and TorchVision.

```
from PIL import Image
import pandas as pd
import os
from matplotlib.pyplot import imshow
from torch.utils.data import Dataset, DataLoader
```

The dataset is built using Zalando's **Fashion-MNIST** training set, which contains: 60,000 grayscale images, with 28 × 28 pixels resolutions, and 10 distinct classes representing types of clothing.

The dataset is provided in the form of folder of image files (a CSV file mapping each image file to a class label):

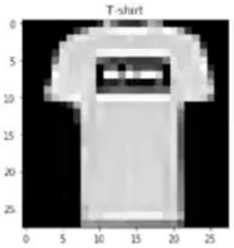
- The first column contains the clothing label (class).
- The second column image file name (the image file path is constructed by combining the base directory with the image file name).

```
directory="/resources/data"
csv_file ='index.csv'
csv_path=os.path.join(directory,csv_file)

data_name = pd.read_csv(csv_path)
data_name.head()
```

	category	image
0	Ankle boot	img/fashion0.png
1	T-shirt	img/fashion1.png
2	T-shirt	img/fashion2.png
3	Dress	img/fashion3.png
4	T-shirt	img/fashion4.png

Images in the dataset can be loaded using **`Image.open(path)`** and stored in a variable.

```
image_name =data_name.iloc[1, 1]
image_path=os.path.join(directory,image_name)

image = Image.open(image_path)
plt.imshow(image,cmap='gray', vmin=0, vmax=255)
plt.title(data_name.iloc[1, 0])
plt.show()
```



🔹 Building the Custom Dataset Class

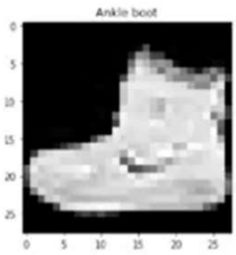
The image dataset class follows the same structure as a PyTorch Dataset subclass:

- In the **constructor**:
  - The CSV file is loaded.
  - The image names and labels are stored as a DataFrame attribute (self.data_names).
- The `__getitem__` method is responsible for:
  - Receiving an index.
  - Retrieving the image name and label from the DataFrame.
  - Building the full image path.
  - Loading the image using the path.
  - Assigning the class label to y.
  - Returning a tuple of (image, y).

```
class Dataset(Dataset):
  def __init__(self, csv_file, data_dir, transform=None):
      self.transform = transform
      self.data_dir=data_dir
      data_dircsv_file=os.path.join(self.data_dir,csv_file)
      self.data_name= pd.read_csv(data_dircsv_file)
      self.len=self.data_name.shape[0]

  def __len__(self):
      return self.len

  def __getitem__(self, idx):
      img_name=os.path.join(self.data_dir,self.data_name.iloc[idx, 1])
      image = Image.open(img_name)
      y = self.data_name.iloc[idx, 0]
      if self.transform:
          image = self.transform(image)
    return image, y
```

⚠️ This approach avoids loading all images into memory at once, making it scalable to large datasets.

```
dataset = Dataset(csv_file=csv_file, data_dir=directory)
y=dataset[0][1] ⇨ y:'Ankle boot'
image=dataset[0][0]
plt.imshow(image,cmap='gray', vmin=0, vmax=255)
plt.title(data_name.iloc[19, 0])
plt.show()
```



🔹 TorchVision Image Transforms

TorchVision includes a powerful module of image transforms used during data preprocessing. Transforms are applied to modify images before passing them to a neural network.

```
import torchvision.transforms as transforms

transforms.CenterCrop(20)

transforms.ToTensor()
```

- 🔸 **Composing Transforms:**

  Multiple transforms can be combined into a sequence using **`transforms.Compose`**.

  A Compose object accepts a list of transforms, when a sample is passed through the Compose object:

  - The first transform is applied.
  - The result is passed to the next transform.
  - The final transformed image is returned.

  The composed transform is passed into the dataset's constructor and applied automatically when samples are retrieved via **`__getitem__`**.

  ℹ️ After transformation, image tensors have an extra dimension representing the batch or channel axis, required for model compatibility.

```
import torchvision.transforms as transforms

croptensor_data_transform = transforms.Compose([transforms.CenterCrop(20), transforms.ToTensor()])

dataset = Dataset(csv_file=csv_file , data_dir=directory,transform=croptensor_data_transform )

dataset[0][0].shape ⇨ torch.Size([1, 20, 20])
```

☑Takeaways

☑Image datasets can be built using a CSV file mapping image names to class labels.

☑Data should be loaded **on demand** in the __getitem__ method to conserve memory and enable scalability.

☑TorchVision transforms such as cropping and tensor conversion are essential for preparing image inputs for model training.

☑The Compose utility allows multiple preprocessing steps to be applied sequentially and efficiently.

Module 2
Linear Regression with PyTorch

M. 2 – Section 1
Linear Regression Prediction with PyTorch

📌 Simple Linear Regression Prediction

This section introduces the principles of linear regression in one dimension and demonstrates how to build and use linear models in PyTorch to predict and output based on a given input.

By using functional and object-oriented approaches to define and use linear regression layers for prediction.

🔹 Concept of Linear Regression

Linear regression is a method used to model the relationship between an independent variable **x** (**feature**) and a dependent variable **y** (**target**). In the one-dimensional case, this relationship is represented as a straight line:

$$y = b + w\,x$$

Where:

- $\hat{y}$ is the predicted output (estimate).
- w is the slope or weight,
- b is the bias or intercept.

This equation defines the **linear model** that maps input values to estimated outputs. The goal of training is to determine optimal values for w and b.

🔹 Prediction Using Tensors

To perform prediction manually using some arbitrary values, two tensors are created.

- One for the weight (slope).
- One for the bias (intercept).

Both tensors have **`requires_grad=True`** set, indicating they are trainable parameters.

A function **`forward(x)`** is defined to apply the linear equation.

Input values **x** are passed into this function, and the resulting tensor is the predicted output $\hat{y}$.

Predictions can be made on a single input.

```
import torch

w=torch.tensor(2.0,requires_grad=True)
b=torch.tensor(-1.0,requires_grad=True)

def forward(x):
        y=w*x+b
        return y

x=torch.tensor([1.0])
yhat=forward(x)  ⇒  yhat: tensor([1.0])
```

Or a tensor containing multiple rows. The linear function is applied row-wise. each row is treated as a sample.

```
x=torch.tensor([[1],[2]]) ⇨ x = [1; 2]
yhat=forward(x)
yhat: tensor([[1],[3]]) ⇨ ŷ = [1; 3]
```

◆ Built-in Linear Model with `nn.Linear`

PyTorch includes a **built-in class `nn.Linear`**, which automatically handles weight and bias initialization and encapsulates the forward operation.

A linear model is created by calling:

`nn.Linear(in_features, out_features)`.

```
from torch.nn import Linear

torch.manual_seed(1)
model=Linear(in_features=1,out_features=1)
```

Parameters:

- **`in_features`**: Number of input features (columns).
- **`out_features`**: Number of output features.

After constructing the model:

- The slope and bias are initialized randomly.
- Model parameters can be inspected using:

  `model.parameters()`: the first element is the slope, the second is the bias. **`list()`** function needs to be applied in order to get the output (because the method is lazily evaluated).

  Or **`model.state_dict()`**, this method is explained in detail later in this section.

```
print(list(model.parameters()))

[ Parameter containing: tensor([[0.52]]),
  Parameter containing: tensor([- 0.44]) ]
```

To make predictions, pass the input tensor to the model directly.

There is no need to explicitly call a forward method; the object handles this internally.

```
x=torch.tensor([0.0])
yhat=model(x)
yhat: tensor([[-0.4414]])
```

Multiple input values are processed in batch format, where each row is treated as a separate input vector.

```
x=torch.tensor([[1.0],[2.0]]) ⇨ x = [1; 2]
yhat=model(x)
yhat: tensor([[0.07], [0.58]]) ⇨ ŷ = [0.07; 0.58]
```

Building a Custom Linear Module

A custom module allows us to wrap multiple objects to make more complex workflows.

It can be defined by subclassing **`nn.Module`**.

Custom Class Structure:

The class is a child of **`nn.Module`**, inheriting its methods and behavior.

In the constructor:

- The base class constructor is initialized using **`super()`**.
- In the object constructor, the argument are the **size of the input** (x, **`in_size`**) and **output** (y, **`output_size`**).
- A linear layer is created using **`nn.Linear(input_size, output_size)`** and stored as **`self.linear`**.

A **`forward`** method is defined to apply the linear transformation.

```
import torch.nn as nn

class LR(nn.Module):
        def __init__(self,in_size,output_size):
                super(LR, self).__init__()
                self.linear=nn.Linear(in_size,output_size)

        def forward(self,x):
                out=self.linear(x)
                return out
```

Once the custom class is defined:

- A model object is created by passing input/output size arguments.
- Model parameters are available through inherited methods:
  - **`model.state_dict()`** is used to initialize the weight and bias of the model.
  - **`model.parameters()`** for inspecting layer-specific weights and biases.
- Predictions are made by calling the model with the input tensor, the method **`forward`** do not have to be called explicitly.

```
model=LR(1,1)

# models state_dict initialize the weight and bias of the model
model.state_dict()['linear.weight'].data[0] = torch.tensor([ 0.5153])
model.state_dict()['linear.bias'].data[0] = torch.tensor([ -0.4414])

print(list(model.parameters()))
[Parameter containing: tensor([[ 0.5153]]), Parameter containing: tensor([-0.4414])]

x=torch.tensor([1.0])

yhat=model(x) ⇨ yhat: tensor([[0.0739]])
```

The initialized custom model can be used to make multiple predictions as seen before; the object maps every row in the tensor.

```
x=torch.tensor([[1.0],[2.0]]) ⇨ x = [1; 2]
yhat=model (x)
yhat: tensor([[0.07], [0.58]]) ⇨ ŷ = [0.07; 0.58]
```

◆ Using `state_dict` for Parameter Access

The **`state_dict()`** method returns a Python dictionary containing all the learnable parameters of the model, as models get more complex this method becomes more useful.

One Function is to map the relationship of the linear layers to its parameters.

- Each key corresponds to a named parameter (e.g., **`linear.weight`**, **`linear.bias`**).
- Each value is the tensor containing the current value of the parameter.

This dictionary is useful for:

- Inspecting parameter values
- Debugging model initialization
- Saving and loading model weights in more advanced use cases

```
print("keys: ", model.state_dict().keys()) ⇨ keys: odict_keys(['linear.weight', 'linear.bias'])
print("values: ", model.state_dict().values()) ⇨ values: odict_values([tensor([[0.5153]]), tensor([-0.4414])])
```

☑Takeaways

☑Linear regression models define a simple mapping between input and output using a linear equation.

☑In PyTorch, models can be implemented manually using tensors or more efficiently using **`nn.Linear`**.

☑The **`nn.Linear`** class handles weight and bias internally and can be used directly for predictions.

☑Custom modules can be built by subclassing **`nn.Module`** and defining a forward method.

☑Once constructed, model objects behave like callable functions and do not require explicit calls to the forward method.

☑Model parameters and their initialization can be accessed using **`.parameters()`** and **`.state_dict()`**.

☑These foundational practices set the stage for training models and scaling to more complex architectures.

M. 2 – Section 2
Linear Regression

📌 Linear Regression Training

This section introduces the training process for linear regression in PyTorch. It defines what constitutes a dataset, explains the noise assumption behind regression models, and presents the objective of learning model parameters by minimizing the mean squared error.

The focus is on how a model learns from examples by fitting a line that best captures the relationship between the input and output variables.

🔹 Defining the Dataset and Learning Objective

Linear regression aims to model the relationship between a feature (independent variable x) and a target (dependent variable y).

The goal of training is to **learn the best** values for the model **parameters**—slope and bias—that define a linear function capable of estimating y given x.

- A dataset is composed of **N** pairs of values: (x_1, y_1), (x_2, y_2), ..., (x_n, y_n).
- Each x_i and y_i pair is related through a linear function plus a small amount of random noise.
- This process is known as **supervised learning**, where known input-output pairs are used to fit a model.

Examples of real-world applications of simple linear regression include:

- Predicting house prices based on size.
- Estimating stock prices from interest rates.
- Modeling fuel efficiency as a function of horsepower.

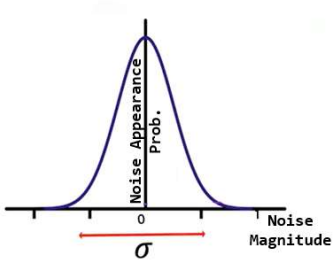
In all cases, x is the feature and y is the predicted output.

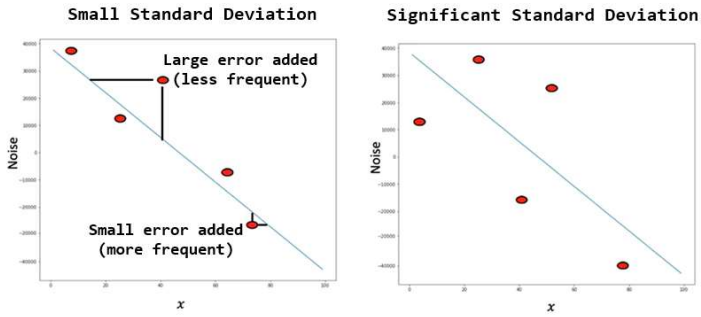
🔹 The Noise Assumption in Regression

Even when the relationship between variables is approximately linear, real-world data is not perfectly aligned on a straight line. This is because of **random noise**, which reflects measurement errors or unmodeled effects.

- The noise is assumed to be **Gaussian-distributed** with a mean of zero.
- The horizontal axis of the Gaussian curve represents the magnitude of the added noise.
- The vertical axis represents the probability of observing that value.
- Most of the noise values are close to zero, with only occasional large deviations.

- The more significant the standard deviation or, the more disperse the distribution is, the more the samples deviate from the line.

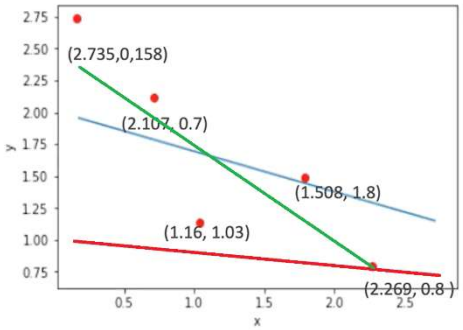




The Goal of Training

The objective of training a linear regression model is to **find the line that best fits the dataset**.

- In practice, several candidate lines may be drawn through the data points.
- Visually inspecting lines can suggest better or worse fits, but a mathematical method is needed for objective evaluation.



$\hat{y} = b + wx$

$\hat{y} = 1 - 0.1x$

$\hat{y} = 2 - 0.3x$

$\hat{y} = 2.6 - 0.8x$

To formalize the training process, a **cost function** is introduced:

- The cost function is the **Mean Squared Error (MSE)**:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

Where:

- $\hat{y}_i$ is the predicted output
- y_i is the actual output
- n is the number of data points

- The MSE depends on the **slope and bias** of the model.
- Different parameter values lead to different MSE values.
- The best-fitting line is the one that **minimizes** this cost function.

Minimizing the mean squared error ensures that, on average, the model's predictions are as close as possible to the true values of y.

☑ Takeaways

☑ A linear regression model learns to map x to y by fitting a line to a set of input-output pairs.

☑ Datasets consist of ordered pairs of numeric values, where each pair defines a single example.

☑ Real data contains noise, modeled as Gaussian-distributed random variation added to each observation.

☑ The goal of training is to identify model parameters (slope and bias) that minimize the prediction error.

☑ The prediction error is quantified using the **mean squared error**, which forms the basis of the cost function used during optimization.

📌 Loss in Linear Regression

This section introduces the concept of **loss** as a fundamental building block in model training.

Loss quantifies the difference between the model's prediction and the true value, and serves as the foundation for the **cost function**, which is used to guide parameter optimization.

🔹 Role of Loss in Model Training

The training objective in linear regression is to learn the best model parameters (slope and bias) that result in accurate predictions for the dependent variable y given input x.

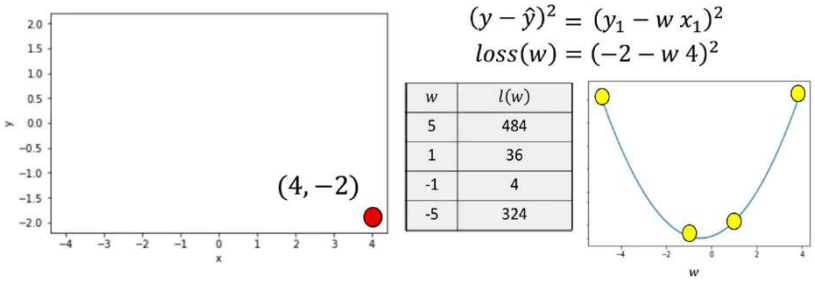
- For any input-output pair (x, y), the model produces an estimate $\hat{y}$ using the linear function.
- Loss measures how far this prediction $\hat{y}$ is from the actual value y.
- The **smaller** the loss, the **better** the prediction.

🔹 Defining Loss for a Single Sample

To understand how the model adjusts parameters, a simplified example is used with only one sample:

- Suppose x = -2 and y = 4.
- A model prediction is computed using a candidate slope value w.
- The prediction error is calculated as:

$$Loss = (\hat{y} - y)^2 = (w.x - y)^2$$



- Selecting a **slope of 5**, the line is far from the data point. In the data space, the value of the loss function is relatively large,
- Selecting a **slope of 1**, the value for the loss is near the minimum of the parameter space.
- Selecting a **slope of -1**, the result gets much closer to the minimum of the loss function, closer to the loss curve.
- A **slope of -5** the line is much farther away from the data point.

The squared difference captures how far the prediction is from the actual value and ensures that positive and negative errors do not cancel each other out.

- Since the true values of x and y are fixed during training, the loss becomes a function of the model parameter (slope).

- The loss function is also called the **criterion function**.
- It outputs a numerical value that reflects how good or bad a model's prediction is.
- When visualized, the loss function appears as a **concave bowl**, or **parabola**, in the parameter space.

This shape has key properties:

- **Minimum point** corresponds to the best slope value.
- **Left of the minimum**: the derivative (slope of the loss curve) is negative.
- **Right of the minimum**: the derivative is positive.
- **At the minimum**: the derivative is zero.

This behavior allows optimization techniques to search for the best parameter value by analyzing the derivative of the loss function.

◆ Systematic Minimization of Loss

Instead of testing random parameter values, a **systematic method** is preferred to minimize the loss:

- Visualizing loss for different slope values shows:
  - Poor parameter choices result in high loss.
  - Optimal parameter values bring the predicted line closer to the actual data point, reducing loss.

- The **best slope** can be found algebraically by:
    - Taking the **derivative** of the loss function with respect to the slope. **derivatives points in the direction of decreasing loss**.
    - Setting the derivative equal to zero.
    - Solving for the slope value.

This technique finds the slope that minimizes loss for the given data point.

We can actually find the best value for the slope by setting the derivative = 0.

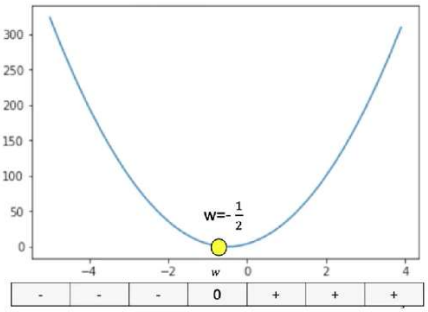
⚠️ However, this exact method is impractical for more complex models (e.g., deep learning), where explicit derivatives are difficult or impossible to compute algebraically.

$$l(w) = (-2 - w\ 4)^2$$

$$\frac{dl(w)}{dw}=0$$

$$2(-2 - w\ 4)(-4))=0$$

$$w=-\frac{1}{2}$$



☑Takeaways

☑ Loss is a numeric measure of how well a model prediction matches the actual target value.

☑ For linear regression, loss is commonly defined as the **squared difference** between prediction and target.

☑ The loss function is treated as a function of the model parameters (e.g., slope).

☑ The objective of training is to **minimize the loss** to improve prediction accuracy.

☑ The loss function has a clear geometric interpretation: its **minimum** represents the best-fitting model.

☑ Derivatives indicate how to update parameters and are foundational to gradient descent and training in neural networks.

📌Gradient Descent and Cost

This section introduces **gradient descent**, the fundamental optimization technique used to minimize loss functions in machine learning.

It explains how gradient descent works in one dimension and addresses challenges like learning rate selection and stopping criteria.

The process is applied to adjust model parameters iteratively to minimize prediction errors.

What is Gradient Descent?

Gradient descent is an **iterative algorithm** used to find the minimum of a function by adjusting its parameters in the direction of the steepest descent.

- The algorithm begins with a **random initial guess** for a model parameter (e.g., the slope in linear regression).
- It evaluates the **gradient** (i.e., the derivative) of the loss function with respect to that parameter.
- The parameter is then **updated** by subtracting a value **proportional to the derivative**:

$$w_{n+1} = w_n - \eta . \frac{dL}{dw}$$

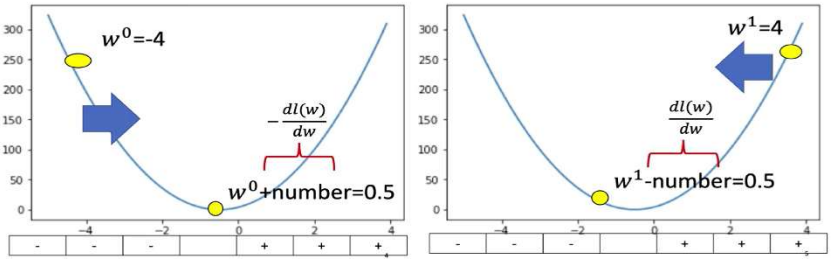
Where:

- w_n is the current parameter value.
- η is the learning rate.
- $\frac{dL}{dw}$ is the derivate of the loss function.

The sign of the gradient determines the direction of the parameter update:

- If the derivative is negative, the parameter increases.
- If the derivative is positive, the parameter decreases.
- The gradient points away from the minimum, so its negative is used to move toward the minimum.

$$l(w) = (-2 - w\ 4)^2$$



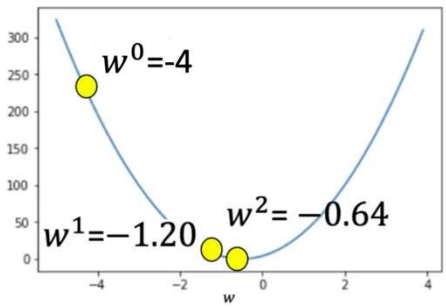
Gradient Descent in Practice

To compute gradient descent:

- The process is started by selecting a random guess, and choosing the learning rate.
- Then the derivate at that point is calculated.
- Finally, the parameter is updated.

$$w^1 = w^0 - \eta \frac{dl(w^0)}{dw}$$
$$w^1 = -4 - \frac{1}{40}(-112)$$
$$w^1 = -1.20$$

$$w^2 = w^1 - \eta \frac{dl(w^1)}{dw}$$
$$w^2 = -1.2 - \frac{1}{40}(-22.4)$$
$$w^2 = -0.64$$



After the update, the loss decreases.

- The next iteration continues using the updated value.
- The parameter is updated again and the loss continues to decrease.

This process continues until a stop condition is reached.

Problems with Learning Rate

The **learning rate (η)** controls the step size for parameter updates. Choosing an inappropriate value leads to problems:

1. Learning Rate Too Large

- The algorithm **overshoots** the minimum and the loss increases.
- This causes the algorithm to diverge or oscillate.

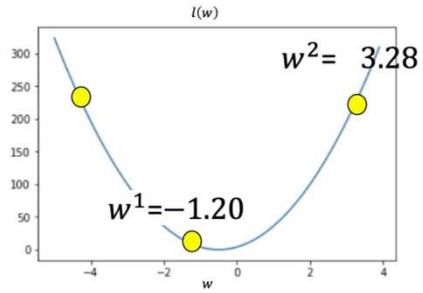
η: 1/5

$$w^2 = w^1 - \eta \frac{dl(w^1)}{dw}$$

$$w^2 = -1.2 - \frac{1}{5}(-22.4)$$

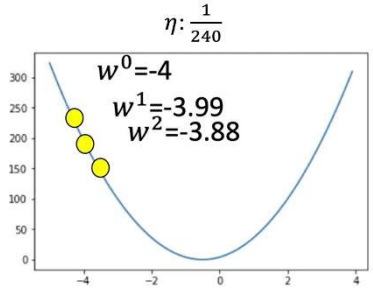
$$w^2 = 3.28$$

$$l(w^1) < l(w^2)$$



2. Learning Rate Too Small

- The algorithm makes **very slow progress**.
- This leads to excessive computation time and inefficient convergence.



The learning rate must be chosen carefully to balance stability and speed of convergence.

When to Stop Gradient Descent

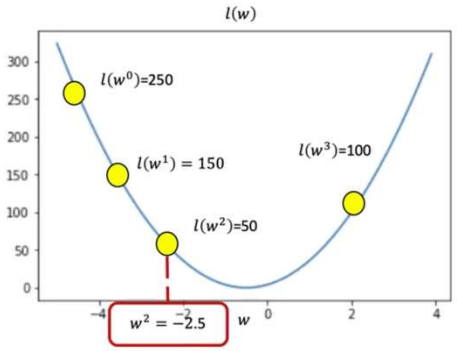
Several strategies are commonly used to decide **when to stop** the gradient descent process:

1. Fixed Number of Iterations

- Run gradient descent for a predefined number of iterations (e.g., 3 iterations).
- This is simple but may **miss the true minimum**.

For example:

- Start with a loss of 250.
- After several iterations, the loss values may be: 150 → 80 → 50 → 100.
- The stopping point is when the loss **increases** to 100.
- The best parameter is the one corresponding to a loss of 50.



2. Monitor Loss Values

- Continue updating until the loss stops decreasing.
- Maintain a table of loss values across iterations:
  - If the loss starts increasing or stagnates, the process stops.
  - The best parameter value is chosen from the iteration with the **lowest loss**.

Loss	Keep Going
250>150	Yes
150>50	Yes
50>100	NO

☑Takeaways

☑ Gradient descent is a key method for minimizing loss and learning model parameters.

☑ It updates parameters using the **negative derivative** of the loss function.

☑ The **learning rate** controls how far the parameter moves with each update:

- Too high: unstable updates, missed minimum.
- Too low: slow convergence.

☑ The update rule is repeated iteratively to approach the optimal value.

☑ The process stops either after a fixed number of iterations or when the loss stops improving.

☑ Understanding gradient descent is essential for training linear and deep learning models.

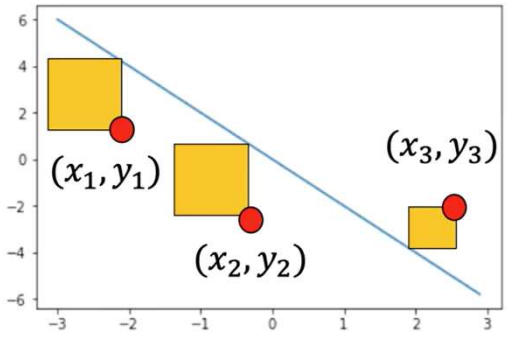
📌Cost

The cost function allows the model to evaluate how well it fits **all training samples**, not just one, and serves as the basis for **batch gradient descent** optimization.

🔹From Loss to Cost

The **loss function** measures prediction error for a single training example. To train a model on multiple examples, a **cost function** is defined by **summing or averaging** the losses across all samples.

- Each prediction error is squared to form a small square, visually representing the magnitude of error.
- The **cost** is the **sum of all these squared errors**, or their **average**:
  - $Total\ cost = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
  - $Average\ cost = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$



Cost or Average Loss $\frac{1}{3}\left[\text{loss} + \text{loss} + \text{loss}\right]$

$$\frac{(y_1 - (w\,x_1 + b))^2 + (y_2 - (w\,x_2 + b))^2 + (y_3 - (w\,x_3 + b))^2}{3}$$

Following PyTorch's convention, the cost function is denoted as **L** (for loss), even when referring to the total over a batch.

Cost Function as a Function of Parameters

The cost function depends on model parameters, especially the **slope (weight)** and **bias**:

- The slope determines the steepness of the predicted line. It controls the relationship between x and y.
- The bias controls the horizontal offset.

The **goal** is to find the values of slope and bias that **minimize the cost**, which means achieving the best possible fit for the data.

Applying Gradient Descent to Cost

The same **gradient descent** technique used for single-sample loss is applied to the **cost function** to optimize parameters across **multiple samples**.

- For multiple data points, the derivative of the cost with respect to the slope is the **sum of the derivatives** for each sample.

$$l(w,b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - (w\,x_n + b))^2 \Rightarrow \frac{dl(w)}{dw} = -\frac{2}{N}\sum_{n=1}^{N}(y_n - w\,x_n)x_n$$

- The process involves:
  1. Computing the cost using all samples.
  2. Calculating the gradient (derivative) of the cost.
  3. Updating parameters using the gradient.

This process is repeated iteratively to improve the model fit.

Several scenarios illustrate how sample distribution affects the derivative of gradient descent with just the slope:

- **Case 1 - All Samples on the Same Side:**

  If both data points lie **below** the current prediction line:

  - The derivative is **strongly negative**.
  - The update step adds a **large positive value** to the parameter.
  - The prediction line moves **closer to the data**.

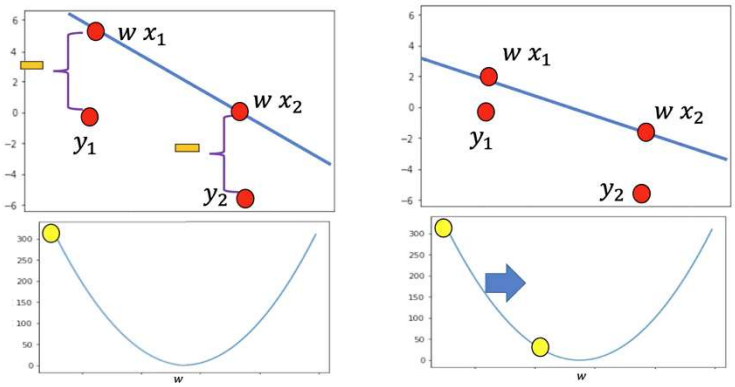
$$\frac{dl(w)}{dw} = -\frac{2}{N}\sum_{n=1}^{N}(y_n - w\,x_n)x_n$$

$$\frac{dl(w)}{dw} = - \; -$$

$$w^{k+1} = w^k - \eta\frac{dl(w^k)}{dw}$$

$$w^{k+1} = w^k - \; -$$

$$w^{k+1} = w^k + large\ number$$



- **Case 2 - All Samples on the Opposite Side:**

  If both points lie **above** the prediction line:

  - The derivative is **strongly positive**.
  - The update subtracts a **large value**, moving the line down.

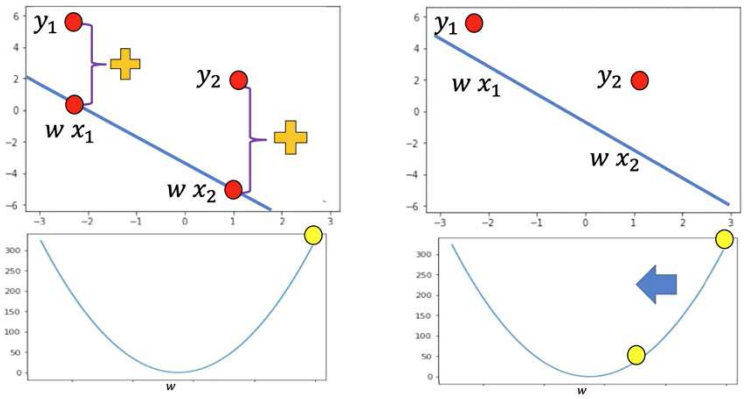
$$\frac{dl(w)}{dw} = -\frac{2}{N}\sum_{n=1}^{N}(y_n - w\,x_n)x_n$$

$$\frac{dl(w)}{dw} = + \; +$$

$$w^{k+1} = w^k - \eta\frac{dl(w^k)}{dw}$$

$$w^{k+1} = w^k - \; +$$

$$w^{k+1} = w^k - large\ number$$



- **Case 3 - Mixed Sides:**

  If one sample is **above** and the other **below** the line:

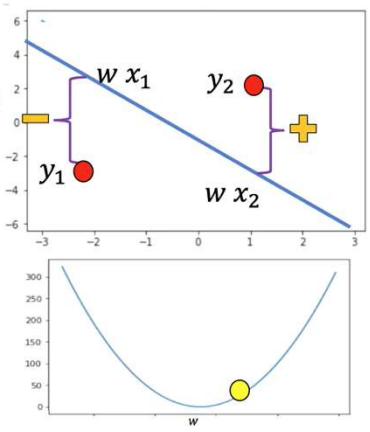
  - The positive and negative derivatives **cancel each other out**.
  - The resulting derivative is **near zero**.
  - The update step is **very small**, and the line changes little.

$$\frac{dl(w)}{dw} = -\frac{2}{N}\sum_{n=1}^{N}(y_n - w\,x_n)$$

$$\frac{dl(w)}{dw} = 0$$

$$w^{k+1} = w^k - \eta\frac{dl(w^k)}{dw}$$

$$w^{k+1} = w^k - 0$$



Batch Gradient Descent

When the **entire dataset** is used to calculate the cost and gradient at each step, the method is called **Batch Gradient Descent**:

- The "batch" refers to the full training set.
- All samples are used to:
  - Compute the total cost.
  - Calculate the overall derivative.
  - Perform the parameter update.

For example, if the batch size is 3:

- The model uses all 3 data points to compute the cost and update.
- This ensures stability and directionally correct updates.

☑ Takeaways

☑ The cost function aggregates prediction errors across all training samples.

☑ It serves as the objective function for training linear models.

☑ Gradient descent applied to the cost function is **batch gradient descent**.

☑ Sample distribution affects the gradient and update size.

☑ Proper gradient accumulation across the batch ensures consistent parameter updates.

M. 2 – Section 3
PyTorch Slope

📌 Linear Regression in PyTorch

This section introduces a hands-on implementation of gradient descent in PyTorch **using only the slope (no bias)** to develop a deeper conceptual understanding of model optimization.

The training process is built step by step using raw PyTorch operations without relying on higher-level modules or abstractions.

🔹 Gradient Descent with PyTorch Tensors

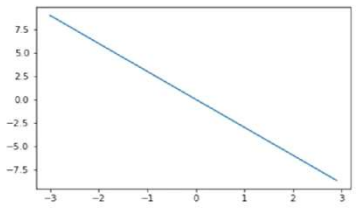
To perform gradient descent manually, a PyTorch tensor is used to represent the model parameter — the slope of the line.

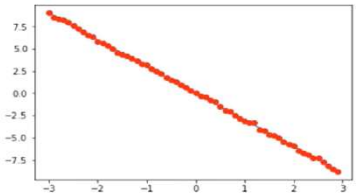
The tensor is initialized with **`requires_grad=True`** to allow PyTorch to automatically compute gradients during backpropagation.

The procedure involves the following:

- Creating a tensor for the slope parameter w
- Generating sample X values and corresponding Y values using a known slope, **`view()`** function is used to add an additional dimension.
- Adding random noise to simulate real-world data variability
- Visualizing the initial data and the true line using matplotlib

```
import torch
w=torch.tensor(-10.0, requires_grad=True)

X=torch.arange(-3,3,0.1).view(-1, 1)
f=-3*X

import matplotlib.pyplot as plt

plt.plot(X.numpy(),f.numpy())
plt.show()

Y=f+0.1*torch.randn(X.size())
plt.plot(X.numpy(),Y.numpy(),'ro')
plt.show()
```





Loss Calculation and Optimization Process

The model is defined using a **`forward()`** function that performs a simple linear transformation.

The cost is computed using a loss function that evaluates how far the predicted values are from the actual target values. In this example, Mean Squared Error (MSE) is used as the loss criterion. Though it represents the cost, it is referred to as "loss" to align with PyTorch's terminology.

```
def forward(x):
    return w * x

def criterion(yhat,y):
    return torch.mean((yhat-y)**2)
```

$$\hat{y} = w\,x$$

$$\frac{1}{N}\sum_{n=1}^{N}(y_n - w\,x_n)^2$$

The key training steps are:

- Compute predictions using the forward function
- Evaluate loss using the defined cost function
- Perform backpropagation using **`loss.backward()`**, the method backwards on the loss calculates the derivative with respect to all the variables in the loss function (PyTorch will be able to differentiate variables as the parameter **`requires_grads`** is set to **True**).
- Access gradients with **`w.grad,`** this method gives the derivate at the point -10.
- Update the slope (parameter) w using the derivative and the learning rate. The attribute **`.data()`** gives access to the data contained in the variable.
- Reset gradients to zero with **`w.grad.zero_()`** for the next iteration, this is due to the fact that PyTorch calculates the gradient in a iterative manner.

```
lr=0.1
for epoch in range(3):
    Yhat=forward(X)
    loss=criterion(Yhat,Y)
    loss.backward()
    w.data=w.data-lr*w.grad.data
    w.grad.data.zero_()
```

$$\hat{y} = -w\begin{bmatrix} -x_1 \\ \vdots \\ -x_N \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

$$l(w) = \frac{1}{N}\sum_{n=1}^{N}(y_n - w\,x_n)^2$$

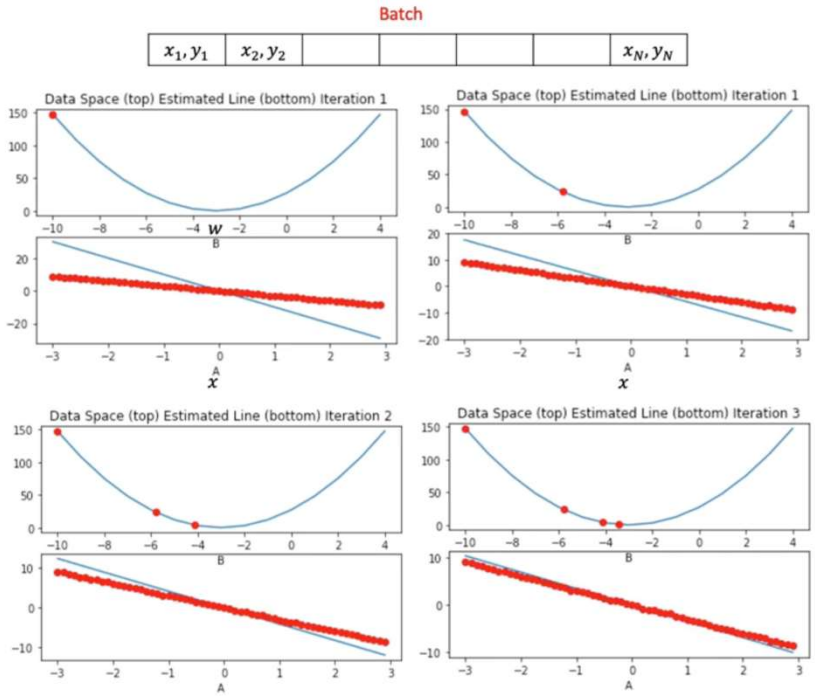
epoch=k

$$w^{k+1} = w^k - \eta\frac{dl(w^k)}{dw}$$

Epochs, Iterations, and Loss Reduction

The process is repeated over multiple **epochs**, where one epoch equals one full pass over the dataset. Each iteration involves:

- Updating the model parameter (slope)
- Observing the gradual decrease in loss
- Adjusting the predicted line to better fit the noisy data



Visualization is used to track:

- The current parameter estimate (as a red dot in the cost function plot)
- The fit of the predicted line (blue) against actual data points (red)
- The loss trend over time

The gradient magnitude determines the size of the parameter update:

- In early epochs, a steep gradient leads to large changes in the slope.
- In later epochs, as the model approaches optimal values, the gradient and parameter updates become smaller.

To better understand this slowdown, it's helpful to look at the tangent line at the points for different iterations. The tangent line slope is equal to the derivative.

- For the first point, the slope is large as such the jump is large.
- For the third iteration the slope is much smaller so the decrease of the average loss is much smaller.

Monitoring Loss Across Epochs

As our models get more complicated, it gets more difficult to plot the COST or average loss for each parameter, one alternative is to look at the COST for every iteration.

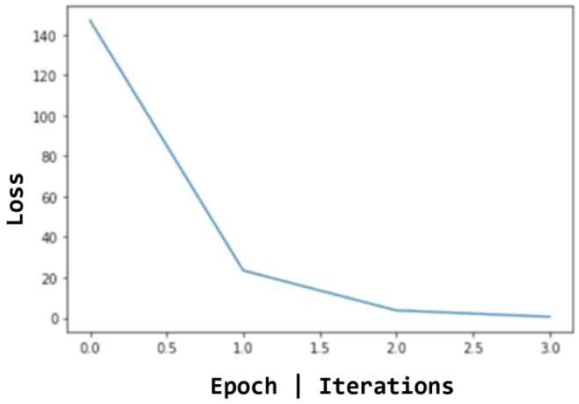
To visualize training progress:

- Loss values are appended to a list on each iteration.
- `.item()` is used to convert PyTorch tensors into native Python numbers.
- Loss is plotted over iterations to verify convergence.

The plotted graph shows:

- A consistent decrease in loss as training progresses
- Smoother model convergence
- Correlation between parameter updates and improvements in model performance

```
lr=0.1
COST=[]
for epoch in range(4):
    Yhat=forward(X)
    loss=criterion(Yhat,Y)

    loss.backward()
    w.data=w.data-lr*w.grad.data

    w.grad.data.zero_()

    COST.append(loss.item())
```



☑ Takeaways

☑ PyTorch enables low-level manual control over gradient descent and parameter updates using raw tensors.

☑ Setting **`requires_grad=True`** allows automatic gradient computation.

☑ Loss is computed and used to update model parameters through **`.backward()`** and gradient subtraction.

☑ The learning rate controls the step size in each iteration; gradients guide the direction of parameter updates.

☑ Visualization of loss per epoch and model fit provides insight into convergence and optimization efficiency.

☑ This process builds foundational intuition for deeper PyTorch training workflows and prepares for future modules using higher-level abstractions.

M. 2 – Section 4
Linear Regression Training with PyTorch

📌 PyTorch LR Training – Slope and Bias

This section focuses on training a linear regression model in PyTorch by learning both slope (weight) and bias using gradient descent.

The objective is to minimize a cost surface defined by these parameters and understand how the model iteratively adjusts them to fit the training data.

🔹 Cost Surface and Parameter Space

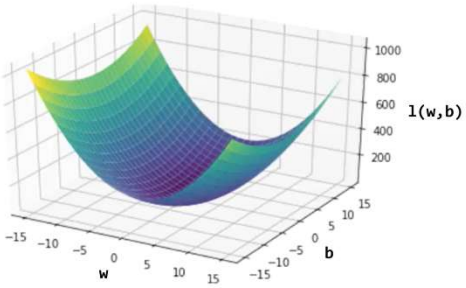
The cost function in linear regression is defined as the **average loss** across training samples. It depends on two parameters:

- **Slope (w)**: Determines the relationship between input x and output y.
- **Bias (b)**: Controls the vertical offset of the line.

When considering both parameters simultaneously, the cost function becomes a **two-dimensional function**, and it can be visualized as a **cost surface**:

- One axis represents the slope.
- One axis represents the bias.
- The vertical dimension (height) represents the value of the cost.

$$l(w,b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - w\,x_n - b)^2$$

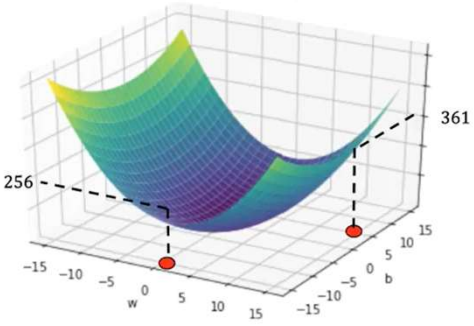


This surface visualization allows us to understand how the cost changes depending on the parameter combinations.

High-cost values indicate poor model fit, while regions with low-cost values indicate better predictions.

$$l(w,b) = (y_n - w\,x_n - b)^2 \qquad \begin{matrix} y_1 = 1 \\ x_1 = 1 \end{matrix}$$

$$l(w,b) = (1 - w\,1 - b)^2$$
$$l(0,15) = (1 - 0 - 15)^2$$
$$l(0,15) = (16)^2$$
$$l(0,15) = 256$$



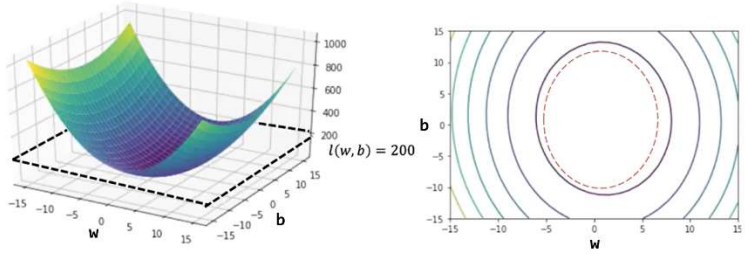
$$l(w,b) = (1 - w\,1 - b)^2$$
$$l(15,-5) = (1 - 15 - 5)^2$$
$$l(15,-5) = (-19)^2$$
$$l(15,-5) = 361$$

Contour Plots and Surface Slices

A contour plot is a useful tool for understanding cost surface. It's a birds-eye view of the surface.

A **contour plot** is a two-dimensional representation that shows slices of the cost surface at fixed cost values. Each contour line connects parameter combinations that yield the **same cost**.

- The horizontal axis corresponds to slope (w).
- The vertical axis corresponds to bias (b).



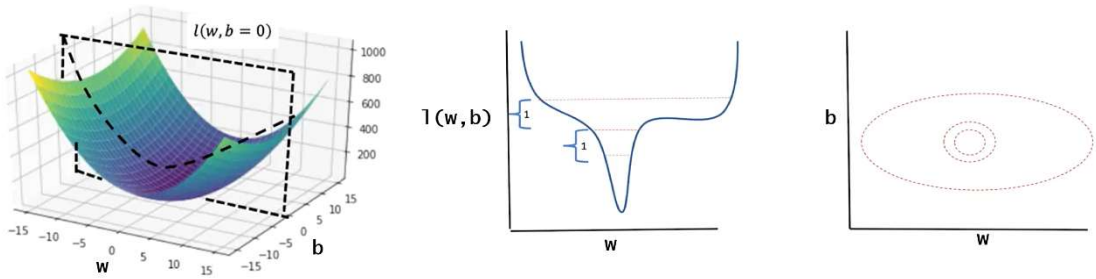
The **spacing between contour lines** reflects how steep or flat the surface is:

- Closely spaced contours indicate steep changes in cost (high gradient).
- Widely spaced contours indicate shallow regions (low gradient

Vertical and horizontal slices of the surface show how cost changes when varying one parameter at a time:

- A **vertical slice** (e.g., fixing b = 0) shows how cost varies with slope.
- A **horizontal slice** shows how cost varies with bias.
- These cross-sections visually demonstrate how curvature affects the update magnitude in gradient descent.

As you move away from the center (the minimum), the rate of change (gradient) increases, which informs the direction and size of parameter updates.



◆ Gradient Descent in PyTorch (Manual Implementation)

ℹ️ There are several ways to minimize the cost function, the following methos is considered the “hard way” and usually is not implemented.

The training process is performed using PyTorch tensors and manual gradient descent.

The following are key aspects of the training process:

- The model uses a **forward function** that defines de prediction using the equation of a line (includes both slope and bias).
- A criterion (cost) function is defined to calculate the mean squared error, representing the cost between predicted and actual values.

```
def forward(x):
    y=w*x+b
    return y

def criterion(yhat,y):
    return torch.mean((yhat-y)**2)
```

$$\hat{y} = w\,x+b$$

$$\frac{1}{N}\sum_{n=1}^{N}(y_n - w\,x_n - b)^2$$

Tensors for:

- **w** (slope) and **b** (bias) are initialized with **`requires_grad=True`** so PyTorch tracks their gradients.
- **X** (input data) and **Y** (target values) are created for the regression task.

```
w = torch.tensor(-15.0, requires_grad = True)
b = torch.tensor(-10.0, requires_grad = True)
X = torch.arange(-3,3,0.1).view(-1, 1)
f = 1 * X - 1
Y = f+0.1*torch.randn(X.size())
```

The procedure is identical as seen in last section, except we also update the bias term. During each epoch:

- The model predicts output using the forward function.
- The loss is computed.
- **`.backward()`** is called on the loss tensor to calculate gradients.
- The **`.grad`** attribute is accessed for both parameters to update them manually.
- **`.data`** is used to apply the update using the learning rate.
- **`.zero_()`** is called to reset gradients for the next iteration.

```
lr=0.1
for epoch in range(15):
    Yhat=forward(X)
    loss=criterion(Yhat,Y)
    loss.backward()
    w.data=w.data-lr*w.grad.data
    w.grad.data.zero_()
    b.data=b.data-lr*b.grad.data
    b.grad.data.zero_()
```

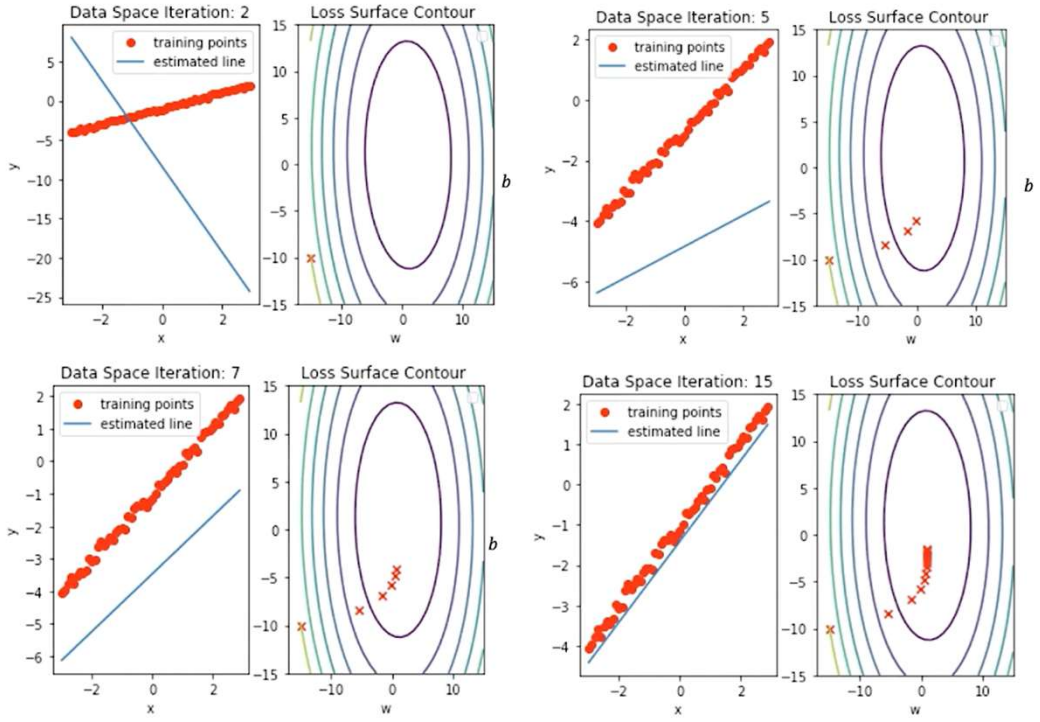
x_1, y_1	x_2, y_2					x_N, y_N

$$w^{k+1}=w^k - \eta\frac{dl(w^k)}{dw}$$

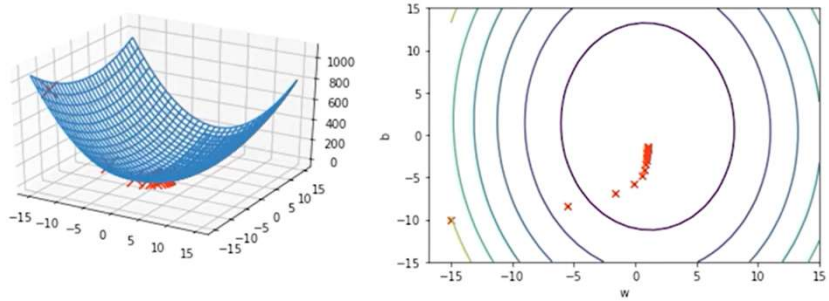
$$b^{k+1}=b^k - \eta\frac{dl(b^k)}{db}$$

The plot on the right shows the loss or cost function for different values of the parameter; the red x's represent parameter values at a given iteration.

The plot on the left shows the estimated function using the parameter values, and the red dots show the training points.



In this image we can see the correspondence between the contour lines and the loss surface.



This manual procedure gives insight into how PyTorch handles gradients internally and how gradient descent is applied in practice.

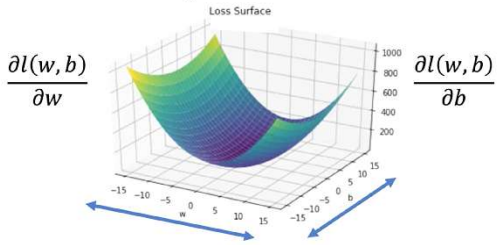
Gradient Vector and Direction of Optimization

In general, the derivative with respect to a multivariate function is called a partial derivative.

If we put the partial derivative with respect to each variable into a vector, we get the gradient, hence the name gradient descent.

The gradient vector, points in the direction of the steepest increase of the cost function.

$$l(w,b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - w\,x_n - b)^2$$

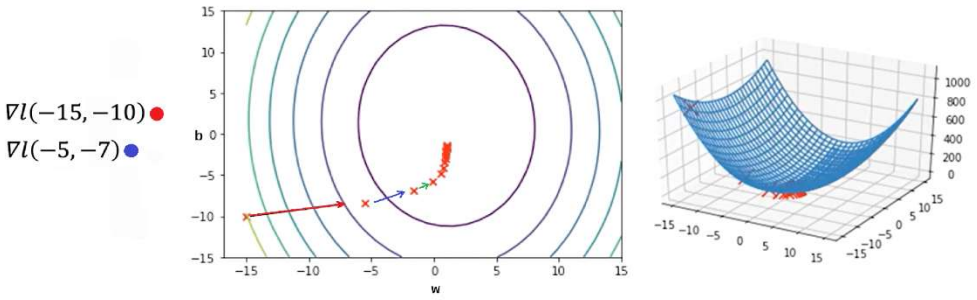


Gradient

$$\nabla l(w,b) = \begin{bmatrix} \frac{\partial l(w,b)}{\partial w} \\ \frac{\partial l(w,b)}{\partial b} \end{bmatrix}$$

The gradient is always **perpendicular to the contour lines** and guides the updates during training.

The gradient also points to the direction of greatest change, hence points to the direction of the next iteration, the same is true for the next iteration, and so on.



☑ Takeaways

☑ The cost function in linear regression with slope and bias defines a **surface** representing model error across parameter combinations.

☑ **Contour plots and surface slices** help visualize how parameter updates impact model performance.

☑ **Gradient descent** is used to iteratively update both slope and bias, reducing the model's loss with each epoch.

☑ PyTorch allows full manual implementation of gradient-based optimization using tensors, gradient tracking, and **`.backward()`** computations.

☑ The **gradient vector** directs the updates and always points toward the direction of greatest increase, so moving opposite to it reduces the loss.

☑ After several epochs, the predicted line aligns closely with the data, and parameters converge to values that minimize the cost.

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

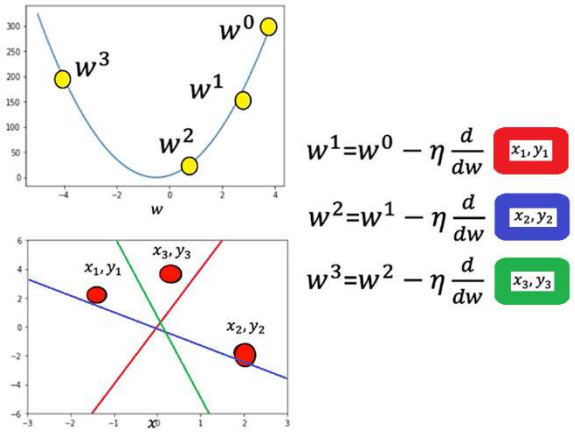
Stochastic Gradient Descent differs from batch gradient descent by **updating model parameters one sample at a time** instead of using the full dataset.

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



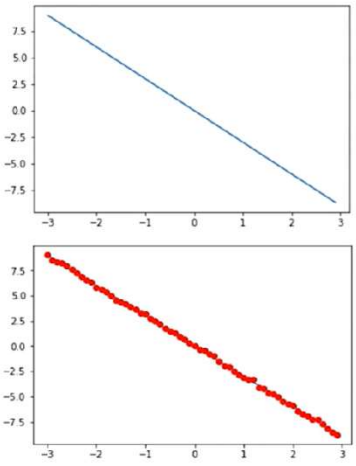
This method approximates the cost function by calculating it one sample at a time, updating weights accordingly.

◆ 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.

```
w=torch.tensor(-15.0, requires_grad=True)
b=torch.tensor(-10.0, requires_grad=True)

X=torch.arange(-3,3,0.1).view(-1, 1)
f=-3*X

import matplotlib.pyplot as plt
plt.plot(X.numpy(),f.numpy())
plt.show()

Y=f+0.1*torch.randn(X.size())
plt.plot(X.numpy(),Y.numpy(),'ro')
plt.show()
```



Model structure:

- A forward function computes predictions using the line equation.

```
def forward(x):
    y=w*x+b
    return y
```

$$\hat{y} = w\,x + b$$

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

```
def criterion(yhat,y):
    return torch.mean((yhat-y)**2)
```

$$\frac{1}{N}\sum_{n=1}^{N}(y_n - w\,x_n + b)^2$$

- 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:

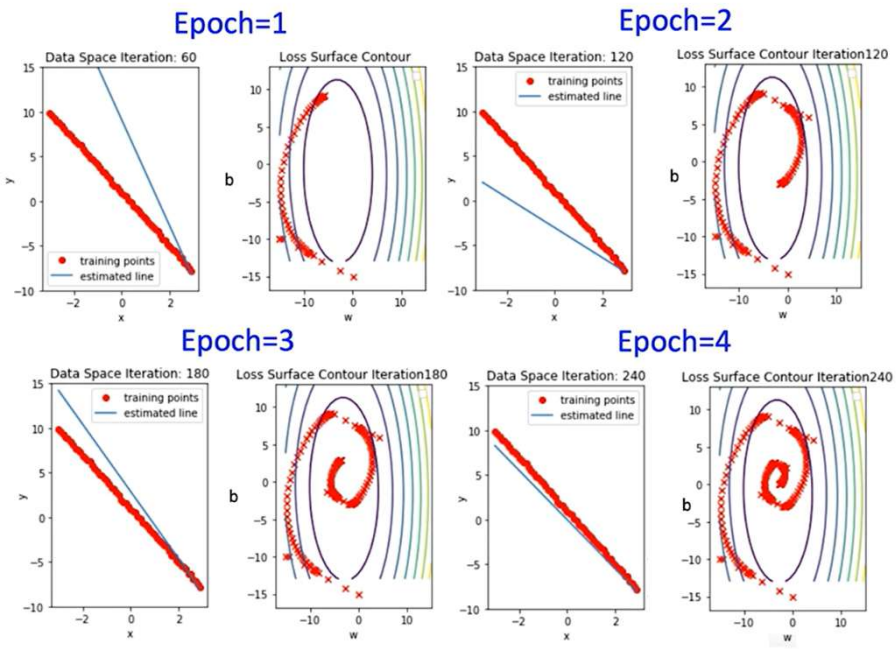
  ✓ **Each training step (batch):** The loss value can be stored in a list in each iteration, this can be thought of as an approximation of the cost.

  ✓ **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.

```
lr=0.1
Cost =[]
for epoch in range(4):
    total=0
    for x,y in zip(X,Y):
        yhat=forward(x)
        loss=criterion(yhat,y)
        loss.backward()
        w.data=w.data-lr*w.grad.data
        b.data=b.data-lr*b.grad.data
        w.grad.data.zero_()
        b.grad.data.zero_()
        total+=loss.item()
    Cost.append(total)
```

```
lr=0.1
LOSS=[]
for epoch in range(4):
    for x,y in zip(X,Y):
        yhat=forward(x)
        loss=criterion(yhat,y)
        loss.backward()
        w.data=w.data-lr*w.grad.data
        b.data=b.data-lr*b.grad.data
        w.grad.data.zero_()
        b.grad.data.zero_()
        LOSS.append(loss.item())
```

x_1, y_1	x_2, y_2					x_N, y_N



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.

```
from torch.utils.data import Dataset

class Data(Dataset):
    def __init__(self):
        self.x=torch.arange(-3,3,0.1).view(-1, 1)
        self.y=-3*X+1
        self.len=self.x.shape[0]

    def __getitem__(self,index):
        return self.x[index],self.y[index]

    def __len__(self):
        return self.len

dataset=Data()

trainloader=DataLoader(dataset=dataset,batch_size=1)

lr=0.1
for x,y in trainloader:
        yhat=forward(x)
        loss=criterion(yhat,y)
        loss.backward()
        w.data=w.data-lr*w.grad.data
        b.data=b.data-lr*b.grad.data
        w.grad.data.zero_()
        b.grad.data.zero_()
```

☑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.

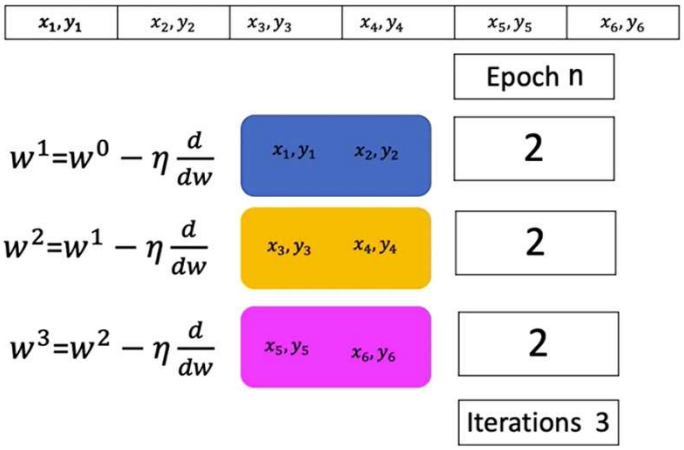
This approach reduces memory consumption and improves training performance compared to full-batch or stochastic gradient descent.

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.

Epochs, Batches, and Iterations

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

$$\text{Iterations} = \frac{\text{training size}}{\text{batch size}}$$

Batch Size		Iterations
1	x_1, y_1 | x_2, y_2 | x_3, y_3 | x_4, y_4 | x_5, y_5 | x_6, y_6	6
2	x_1, y_1 x_2, y_2 | x_3, y_3 x_4, y_4 | x_5, y_5 x_6, y_6	3
3	x_1, y_1 x_2, y_2 x_3, y_3 | x_4, y_4 x_5, y_5 x_6, y_6	2

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

Mini-Batch Gradient Descent in PyTorch

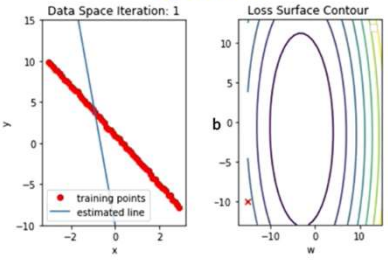
Implementing mini-batch gradient descent in PyTorch closely resembles the process for stochastic gradient descent, with a key change in batch size configuration.

Steps:

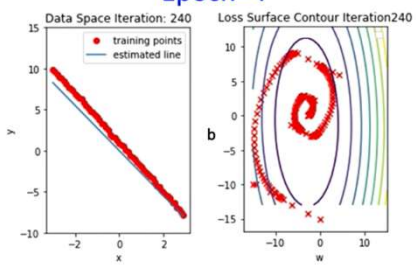
- **Create Dataset Object:**
  - This object holds the training data.
  - It enables access to individual samples or batches using slicing and indexing.
- **Create the `DataLoader`:**
  - Pass the dataset to **`DataLoader`** and specify the desired **`batch_size`**.
  - This object enables efficient 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.

```
dataset=Data()
trainloader=DataLoader(dataset=dataset,batch_size=5)
lr=0.1
LOSS=[]
for epoch in range(4):
    for x,y in trainloader :
        yhat=forward(x)
        loss=criterion(yhat,y)
        loss.backward()
        w.data=w.data-lr*w.grad.data
        b.data=b.data-lr*b.grad.data
        w.grad.data.zero_()
        b.grad.data.zero_()
        LOSS.append(loss.item())
```

Epoch=1



Epoch=4

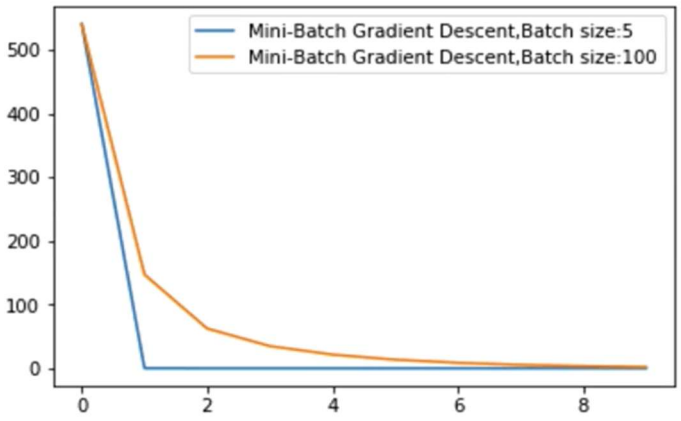


🔹 Convergence Rate and Batch Size

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.

```
from torch.utils.data import Dataset , DataLoader
from torch import nn, optim

class Data(Dataset):
    def __init__(self):
        self.x=torch.arange(-3,3,0.1).view(-1, 1)
        self.y=-3*X+1
        self.len=self.x.shape[0]

    def __getitem__(self,index):
        return self.x[index],self.y[index]

    def __len__(self):
        return self.len

class LR(nn.Module):
        def __init__(self, input_size, output_size):
                super(LR,self).__init__()
                self.linear=nn.Linear(input_size,output_size)

        def forward(self,x):
                out=self.linear(x)
                return out
```

```
dataset=Data()
criterion = nn.MSELoss()
trainloader=DataLoader(dataset=dataset,batch_size=1)
model=LR(1,1)
```

- **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.

```
from torch import nn, optim

optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

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

```
optimizer.state_dict()
{
 'state': {},
 'param_groups': [
     {
       'lr': 0.01,
       'momentum': 0,
       'dampening': 0,
       'weight_decay': 0,
       'nesterov': False,
       'params': [4971190024, 4971190456]
      }
 ]
}
```

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.

```
for epoch in range(100):
    for x,y in trainloader :
        yhat=model(x)
        loss=criterion (yhat,y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

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

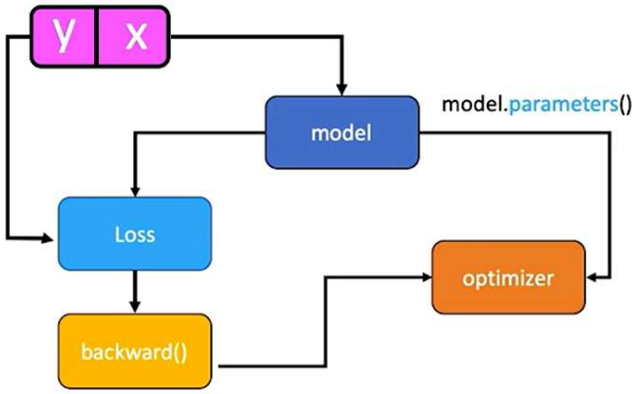
ℹ️ Note: Calling **`optimizer.step()`** essentially updates the parameters like we in manual way.

optimizer.step() ➡ w.data=w.data-lr*w.grad.data
b.data=b.data-lr*b.grad.data

🔹 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

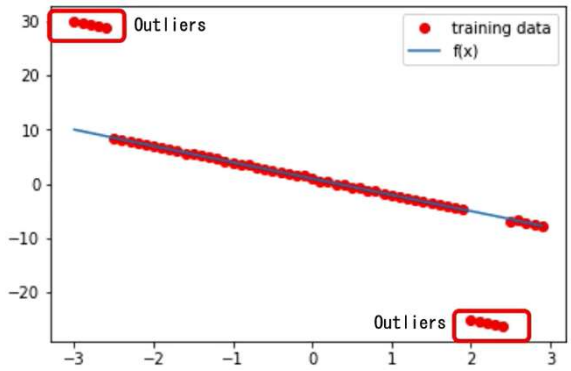
📌 Training, Validation, and Test Split

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:

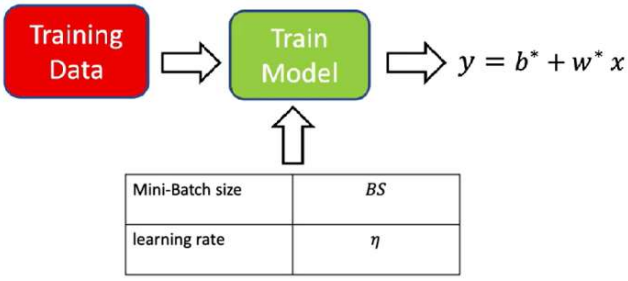
- **Training data** is used to learn model parameters (e.g., slope and bias).
- **Validation data** is used to evaluate different hyperparameter settings.
- **Test data** is used for final evaluation, simulating how the model performs in real-world scenarios.

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.

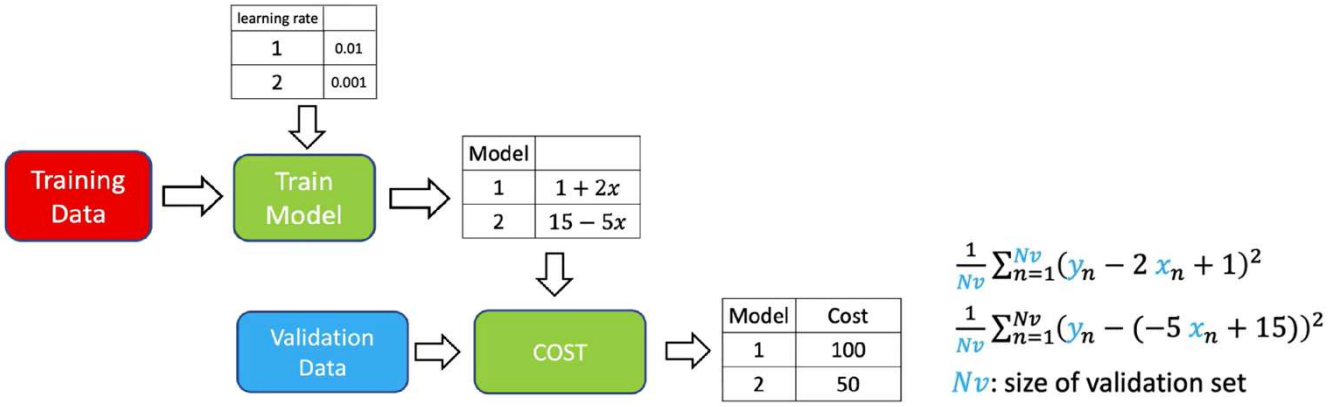


To find optimal hyperparameters:

- 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.

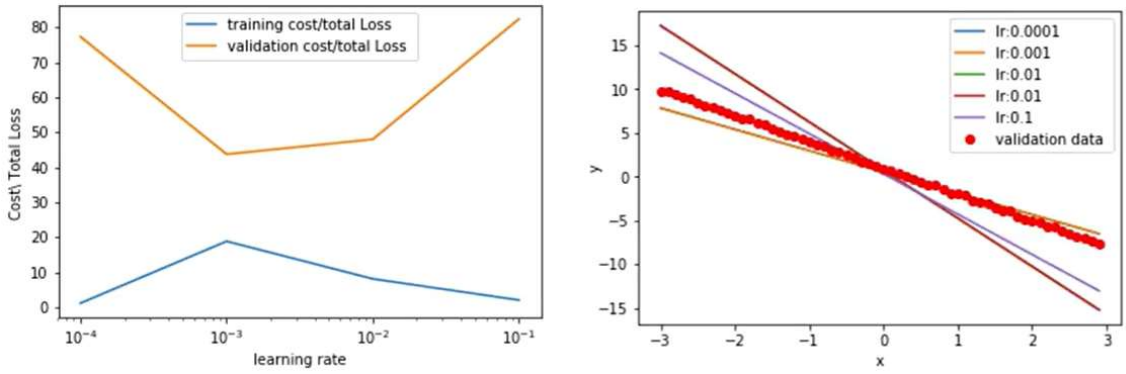


Validation vs. Training Cost

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.

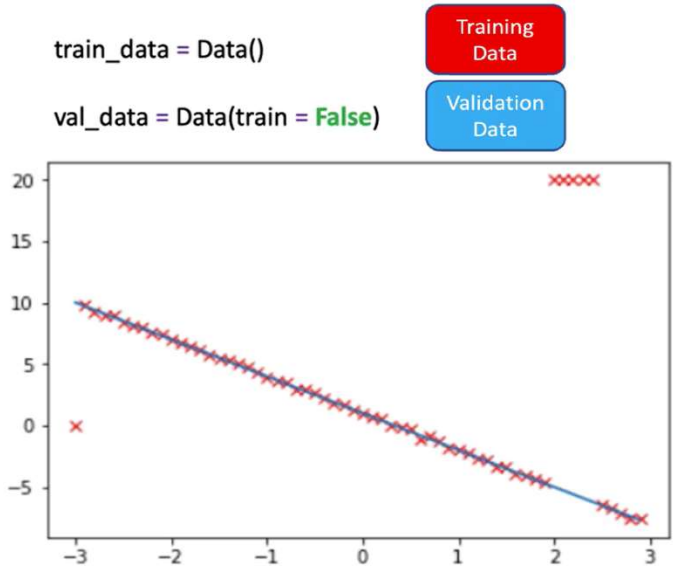
Data Creation and Splitting Strategy

- 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.

```
from torch.utils.data import Dataset, DataLoader

class Data(Dataset):
    def __init__(self,train =True):
        self.x=torch.arange(-3,3,0.1).view(-1, 1)
        self.f=-3*self.x+1
        self.y=self.f+0.1*torch.randn(self.x.size())
        self.len=self.x.shape[0]
        if train==True:
            self.y[0] = 0
            self.y[50:55] = 20
        else:
            pass

        def __getitem__(self,index):
            return self.x[index],self.y[index]

        def __len__(self):
            return self.len
```

```
train_data = Data()

val_data = Data(train = False)
```



Training Loop with Hyperparameter Evaluation

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

```
import torch.nn as nn

class LR(nn.Module):
    def __init__(self,input_size,output_size):
        super(LR,self).__init__()
        self.linear=nn.Linear(input_size,output_size)

    def forward(self,x):
        out=self.linear(x)
        return out
```

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

```
criterion = nn.MSELoss()
trainloader=DataLoader(dataset=train_data,batch_size=1)
```

- **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.

```
epochs=10
learning_rates=[0.0001,0.001,0.01,0.1,1]
validation_error=torch.zeros(len(learning_rates))
test_error=torch.zeros(len(learning_rates))
MODELS=[]
```

- **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.

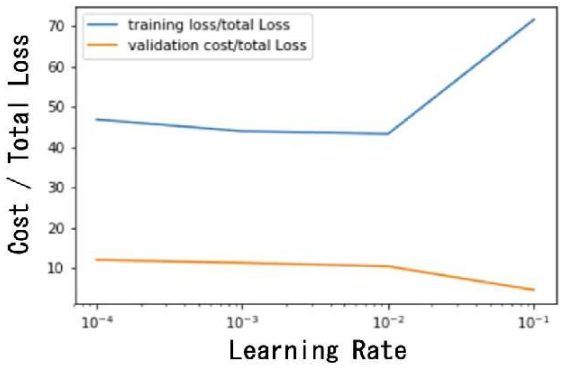
```
from torch import optim
for i,learning_rate in enumerate(learning_rates):
        model=LR(1,1)
        optimizer = optim.SGD(model.parameters(), lr = learning_rate)

        for epoch in range(epochs):
                for x,y in trainloader:
                        yhat=model(x)
                        loss=criterion(yhat,y)
                        optimizer.zero_grad()
                        loss.backward()
                        optimizer.step()

        yhat=model(train_data.x)
        loss=criterion(yhat,train_data.y)
        test_error[i]=loss.item()

        yhat=model(val_data.x)
        loss=criterion(yhat,val_data.y)
        validation_error[i]=loss.item()

        MODELS.append(model)
```

🔸 **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.

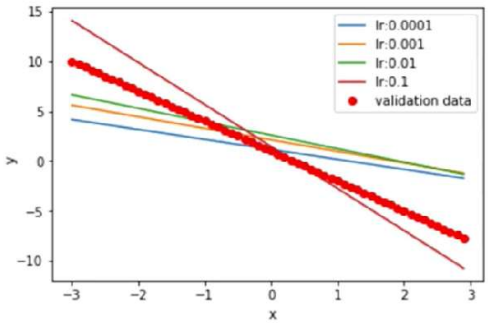
```
plt.semilogx(np.array(learning_rates),validation_error.numpy(),label='training cost/total Loss')
plt.semilogx(np.array(learning_rates),test_error.numpy(),label='validation cost/total Loss ')
plt.ylabel('Cost\ Total Loss ')
plt.xlabel('learning rate')
plt.legend()
plt.show()
```



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.

```
for model,learning_rate in zip(MODELS, learning_rates):
    yhat=model(val_data.x)
    plt.plot(val_data.x.numpy(),yhat.detach().numpy(),label='lr:'+str(learning_rate))

plt.plot(val_data.x.numpy(),val_data.y.numpy(),'or',label='validation data ')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```



☑ 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.

Module 4
Multiple Input / Output Linear Regression

M. 4 – Section 1
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:

$$\hat{y}_1 = b_{01} + w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{1d}x_d$$

$$\hat{y}_2 = b_{02} + w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + w_{2d}x_d$$

$$[\hat{y}_1 \quad \hat{y}_2 \,..\, \hat{y}_M] = [x_1 \quad .. \quad x_d] \begin{bmatrix} w_{11} & w_{12}.. & w_{1M} \\ : & : & : \\ w_{d1} & w_{d2}.. & w_{dM} \end{bmatrix} + [b_{01} \quad b_{02}..b_{0M}]$$

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:

$$\hat{y} = xW + b$$

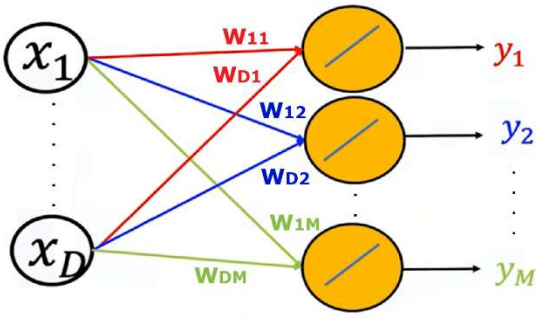
- The result ŷ is a 1×M vector — one output per linear function.
- The computation is visually broken down as:
  - Perform the dot product of the input vector x with **each column of the matrix W**.
  - Add the corresponding **bias term**.
  - Each result is a **scalar output** for a specific linear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

$$\hat{y} = xW + b$$



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)`**.

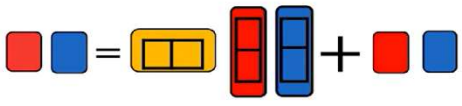
```
import torch.nn as nn

class LR(nn.Module):
    def __init__(self,input_size,output_size):
        super(LR,self).__init__()
        self.linear=nn.Linear(input_size,output_size)

    def forward(self,x):
        out=self.linear(x)
        return out
```

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

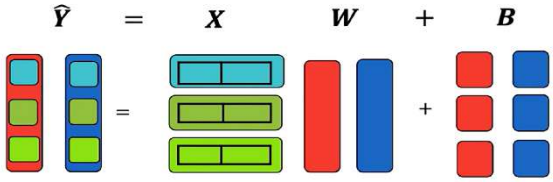
```
torch.manual_seed(1)

model=Linear(input_size=2,output_size=2)
x=torch.tensor([[1.0,2.0]])
yhat=model(x)
yhat: tensor([[-1.2377,  1.2827]])
```

$$\hat{y} = x \quad W + \quad b$$



- Predictions for **multiple sample**:
  - Input: N×D tensor (N samples with D features)
  - Output: N×M tensor (N predictions with M outputs each)

```
X=torch.tensor([[1.0,1.0],[1.0,2.0],[1.0,3.0]])
Yhat=model(X)
Yhat:
tensor([
        [-0.6135, 0.6189],
        [-0.9256, 0.9508],
        [-1.2377, 1.2827]
])
```



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:

$$l(\boldsymbol{W}, b) = \frac{1}{N}\sum_{n=1}^{N}\|\boldsymbol{y}_n - \boldsymbol{x}_n\boldsymbol{W} - \boldsymbol{b}\|^2$$

  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.

$$\boldsymbol{W}^{k+1} = \boldsymbol{W}^{k} - \eta\nabla l\left(\boldsymbol{W}^{k}, b^{k}\right)$$

$$\boldsymbol{b}^{k+1} = \boldsymbol{b}^{k} - \eta\frac{dl(\boldsymbol{W}^{k}, \boldsymbol{b}^{k})}{\partial b}$$

🔹 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.

```
import torch.nn as nn

class LR(nn.Module):
    def __init__(self,in_size,out_size):
        super(LR,self).__init__()
        self.linear=nn.Linear(in_size,output_size)

    def forward(self,x):
        out=self.linear(x)
        return out
```

- **Dataset and DataLoader:**

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

```
from torch.utils.data import Dataset, DataLoader
class Data2D(Dataset):
        def __init__(self):
                self.x=torch.zeros(20,2)
                self.x[:,0]=torch.arange(-1,1,0.1)
                self.x[:,1]=torch.arange(-1,1,0.1)
                self.w=torch.tensor([ [1.0,-1.0],[1.0,-1.0]])
                self.b=torch.tensor([[1.0,-1.0]])
                self.f=torch.mm(self.x,self.w)+self.b
                self.y=self.f+0.1*torch.randn((self.x.shape[0],1))
                self.len=self.x.shape[0]

        def __getitem__(self,index):
                return self.x[index],self.y[index]

        def __len__(self):
                return self.len
```

  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. **<u>Initialize:</u>**
     - 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.

```
data_set=Data2D()
criterion = nn.MSELoss()
trainloader=DataLoader(dataset=data_set,batch_size=1)
model=LR(input_size=2,output_size=2)
optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

2. **Iterate over epochs**:

   For each epoch:

   - Loop over the `DataLoader` batches.
   - **Forward pass**: Input batch → model → predictions ($\hat{\mathbf{y}}$).
   - **Compute loss**: Compare $\hat{\mathbf{y}}$ with target $\mathbf{y}$ 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.

```
for epoch in range(100):
    for x,y  in  trainloader:
        yhat=model(x)
        loss=criterion (yhat,y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

$$\begin{cases} W^{k+1}=W^k-\eta\nabla l\left(W^k,b^k\right) \\ b^{k+1}=b^k-\eta\frac{dl(W^k,b^k)}{\partial b} \end{cases}$$

☑ 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.

M. 4 – Section 2
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:

$$\hat{y} = x_1 w_1 + x_2 w_2 + \cdots + x_D w_D + b$$

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:

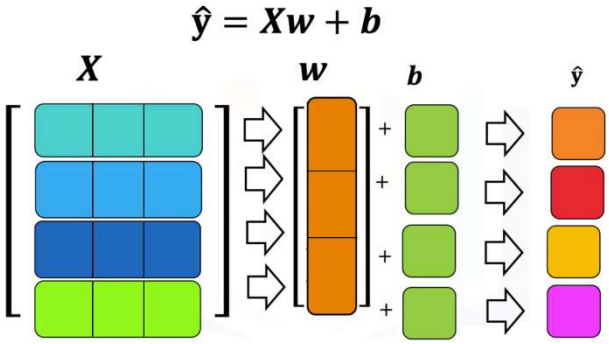
- Columns of X must match the number of weights.
- The same bias term b is added to each dot product.

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`

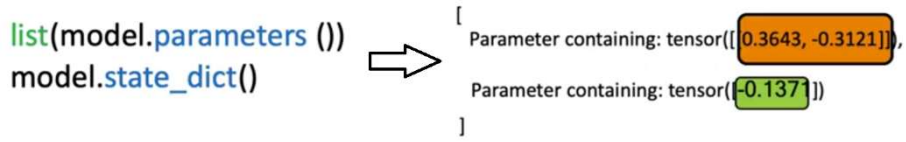
PyTorch's built-in **`nn.Linear`** is used to define linear transformations.

```
from torch.nn import Linear

torch.manual_seed(1)

model=Linear(in_features=2, out_features=1)
```

- **`in_features`** correspond to the number of input columns (features).
- **`out_features`** correspond to the number of outputs (usually 1 for regression).

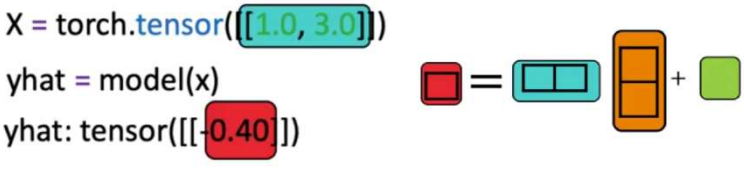
Internally, **`nn.Linear`**:

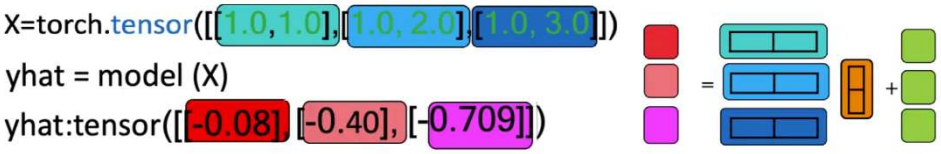
- Initializes weights and bias randomly.
- Stores parameters accessible via **`.parameters()`** (`list()` function must be applied to get an output as the method is lazily evaluated) or **`.state_dict()`**.



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.

```
class LR(nn.Module):
    def __init__(self,input_size, output_size):
        super(LR,self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        out=self.linear(x)
        return out

torch.manual_seed(1)

model = LR(input_size=2, output_size=1)
```

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:

$$l(\boldsymbol{w}, b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - \boldsymbol{x}_n\boldsymbol{w} - b)^2$$

Here, $\hat{\mathbf{y}}$ ($x_{n\,x}\,w - b$) 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.

$$l(w_1, w_2, .. w_d, b) \quad \boldsymbol{w}=\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}$$

The gradient of the loss with respect to:

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

$$\nabla l(\boldsymbol{w}, b) = \begin{bmatrix} \frac{\partial l(\boldsymbol{w},b)}{\partial w_1} \\ \vdots \\ \frac{\partial l(\boldsymbol{w},b)}{\partial w_d} \end{bmatrix}$$

Weights update rules in vector form:

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^{k} - \eta\ \nabla l(\boldsymbol{w}, b)$$

$$\begin{bmatrix} w_1^{k+1} \\ w_2^{k+1} \\ \vdots \\ w_d^{k+1} \end{bmatrix} = \begin{bmatrix} w_1^{k} \\ w_2^{k} \\ \vdots \\ w_d^{k} \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial l(\boldsymbol{w}^k,b)}{\partial w_1} \\ \vdots \\ \frac{\partial l(\boldsymbol{w}^k,b)}{\partial w_d} \end{bmatrix}$$

- 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`**.

```
from torch import nn, optim
import torch
class LR(nn.Module):
    def __init__(self, input_size, output_size):
        super(LR, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        out = self.linear(x)
        return out
```

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

```
from torch.utils.data import Dataset, DataLoader
class Data2D(Dataset):
    def __init__(self):
        self.x = torch.zeros(20,2)
        self.x[:,0] = torch.arange(-1,1, 0.1)
        self.x[:,1] = torch.arange(-1,1, 0.1)
        self.w = torch.tensor([ [1.0], [1.0]])
        self.b = 1
        self.f = torch.mm(self.x, self.w)+self.b
        self.y = self.f + 0.1*torch.randn((self.x.shape[0], 1))
        self.len = self.x.shape[0]

    def __getitem__(self,index):
        return self.x[index], self.y[index]

    def __len__(self):
        return self.len
```

2. 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**.

```
data_set = Data2D()
criterion = nn.MSELoss()
trainloader = DataLoader(dataset=data_set, batch_size=2)
model = LR(input_size=2, output_size=1)
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

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()`**.

```
for epoch in range(100):
    for x,y  in  trainloader:
        yhat = model(x)
        loss = criterion(yhat, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

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

$$w^{k+1}=w^k-\eta\nabla l\left(w^k,b^k\right)$$

$$b^{k+1}=b^k-\eta\frac{dl(w^k,b^k)}{\partial b}$$

☑ 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.

Module 5
Logistic Regression for Classification

📌 Linear Classifiers

This section introduces linear classifiers and the concept of logistic regression for classification.

Explains how samples and their features are represented, how classification boundaries are defined, and how the logistic function can be used to transform continuous outputs into class probabilities.

🔹 Representing Samples and Classes

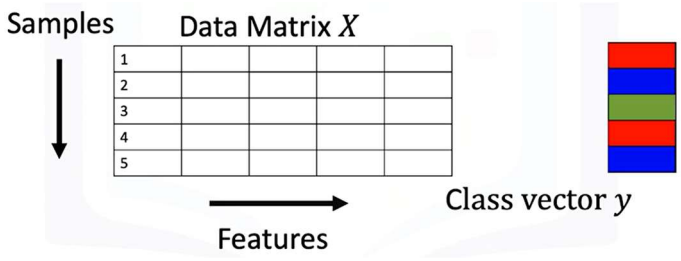
Logistic regression is about predicting which class a particular sample belongs to based on their features.

Each sample in a dataset contains a set of features stored in a **data matrix** X, where:

- **Rows** represent different samples.
- **Columns** represent different features.

A separate **class vector** y contains the discrete class labels for each sample.

- For example, in a three-class scenario, class labels may be 0 (red), 1 (blue), or 2 (green).
- Each element in y corresponds to a row in X.



🔹 Two-Class Linear Classifiers

A **linear classifier** separates data points from two classes using a line (or hyperplane in higher dimensions).

$$z = wx + b$$

Equation in 1D:

- **w:** weight term
- **b:** bias term

Equation in multiple dimensions:

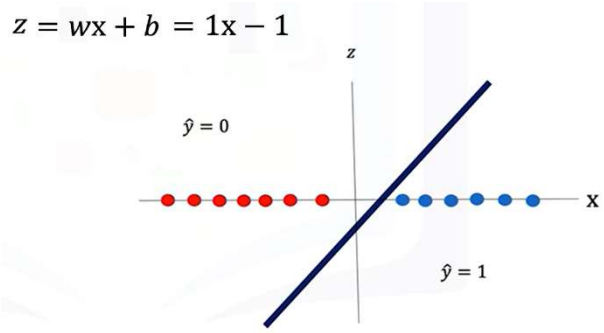
- **w** and **x** are vectors, and the decision boundary is defined where **z=0**.

Linear separability:

- A dataset is **linearly separable** if a single line (or hyperplane) can separate samples from different classes.
- Samples on one side of the boundary yield positive z values, while samples on the other side yield negative z values.

If a line is used to calculate the class of the points, it will always returns real numbers (such as -1, 3, -2).

$$z = wx + b = 1x - 1$$



Example:

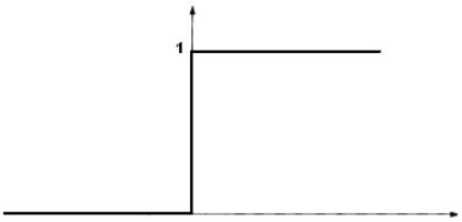
- If $z > 0$, classify as class 1.
- If $z < 0$, classify as class 0.

We need the class to be 0 or 1, so to convert the numbers to a class we use an activation function.

- **Threshold Function for Binary Classification:**

  A **threshold function** maps continuous z values into discrete classes:

$$\hat{y} = \begin{cases} 0 \; if \; z < 0 \\ 1 \; if \; z \geq 0 \end{cases}$$

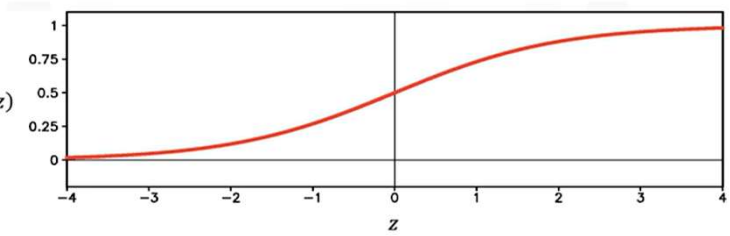


  This simple approach works for perfectly separable datasets but is limited when data is noisy or not perfectly separable.

Logistic Regression and the Sigmoid Function

Logistic regression replaces the threshold function with the **sigmoid function**:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



- Outputs values between 0 and 1.
- Approximates **0** for large negative z and **1** for large positive z.
- For intermediate values of z, the output lies between 0 and 1.

Classification decision:

- If $\sigma(z)>0.5$, predict 1.
- If $\sigma(z)\leq 0.5$, predict 0.

Interpretation:

- Values near 0.5 indicate **low certainty** in classification.
- Values near 0 or 1 indicate **high certainty**.

Applying the **sigmoid function** to z:

- Converts the continuous output to a probability.
- Applying a threshold determines the predicted class label.

◆ **Threshold Function for Binary Classification:**

Logistic regression outputs can be interpreted as probabilities:

- $\mathbf{P}(\hat{y} = 1 \mid x) = \sigma(z)$
- $\mathbf{P}(\hat{y} = 0 \mid x) = 1 - \sigma(z)$

This allows classification models to provide **confidence scores** along with class predictions.

☑ Takeaways

☑ **Linear classifiers** define a decision boundary using a line or hyperplane to separate classes.

☑ **Threshold function** converts continuous scores into discrete class predictions.

☑ **Logistic regression** uses the sigmoid function for smoother decision boundaries and probabilistic interpretation.

☑ **Certainty of classification** increases with distance from the decision boundary.

☑ In higher dimensions, logistic regression generalizes the separation to planes or hyperplanes.

☑ Logistic regression outputs can be directly interpreted as probabilities for class membership.

📌 Logistic Regression Prediction in PyTorch

This section explains how logistic regression is implemented in PyTorch for prediction tasks.

Introduces the logistic function, describes two ways to construct it in PyTorch, using nn.Sequential and custom modules.

Finally, it demonstrates how to perform predictions with single and multiple samples in both one-dimensional and multi-dimensional cases.

🔹 The Logistic Function in PyTorch

The logistic function (also called the **sigmoid function**) takes a real-valued number and compresses it into a range between 0 and 1. This is critical for classification tasks because the output can be interpreted as a **probability**.

In logistic regression, the **linear function** output is passed through the **logistic (sigmoid) function** to produce the prediction $\hat{y}$.

- The output is still **one-dimensional** even when the input involves vector operations (e.g., dot products).

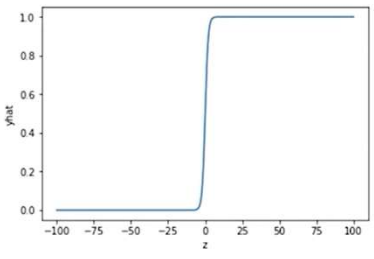
There are two main ways to implement the logistic function in PyTorch, both methods produce the same result, mapping inputs to a range between 0 and 1:

1. **Using `torch.nn.Sigmoid`**
   - Create a sigmoid object with **`nn.Sigmoid()`**.
   - Pass an input tensor to this object to obtain the transformed output.

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

z=torch.arange(-100,100,0.1).view(-1,  1)

sig=nn. Sigmoid ()

yhat=sig(z)


plt.plot(z.numpy(),yhat.numpy())
```

$$z = \begin{bmatrix} -100 \\ \vdots \\ 100 \end{bmatrix}$$

$$\sigma(z) = \begin{bmatrix} \sigma(-100) \\ \vdots \\ \sigma(100) \end{bmatrix}$$

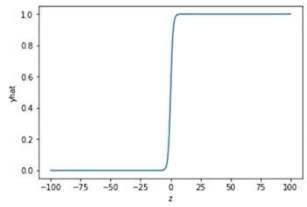


2. **Using `torch.sigmoid()` function**
    - Call **`torch.sigmoid(tensor)`** directly to compute the logistic transformation.

```
import torch
z=torch.arange(-100,100,0.1).view(-1, 1)
yhat= torch.sigmoid(z)

plt.plot(z.numpy(),yhat.numpy())
```

$$z = \begin{bmatrix} -100 \\ \vdots \\ 100 \end{bmatrix}$$

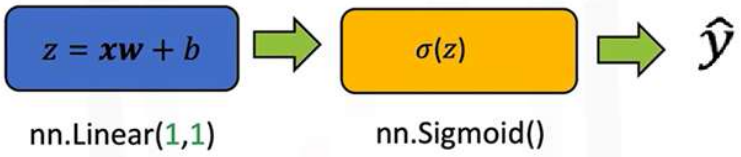
$$\boldsymbol{\sigma}(z) = \begin{bmatrix} \sigma(-100) \\ \vdots \\ \sigma(100) \end{bmatrix}$$



◆ Building Logistic Regression Models with `nn.Sequential`

`nn.Sequential` provides a **concise and fast way** to define logistic regression models by stacking layers and activation functions in order.

A **sequential constructor** is used to produce the model, it starts with a linear constructor and then pass the output to the sigmoid constructor.



- **Example construction for 1D input:**
    - First element: **`nn.Linear(in_features=1, out_features=1)`**
    - Second element: **`nn.Sigmoid()`**
    - The **`Sequential`** container processes the input through the linear transformation and then the sigmoid function.

```
model =nn.Sequential(nn.Linear(1,1),nn.Sigmoid())
yhat = model(x)
```

- **Processing flow:**
    1. Input tensor → Linear layer → intermediate output z
    2. z → Sigmoid function → $\hat{y}$

This approach is compact, and PyTorch automatically connects the layers in the sequence.

Building Custom Custom Modules with `nn.Module`

Logistic regression models can also be created by subclassing **`nn.Module`**. It gives more control and flexibility for defining custom behavior.

Structure of the custom class:

- In the constructor (**`__init__`**):
  - Define a **`nn.Linear()`** transformation with the input and output size.
- In the forward pass:
  - Apply the linear transformation to the input.
  - Pass the result through the sigmoid function.
  - Return the final prediction $\hat{y}$.

```
import torch.nn as nn
  class logistic_regression(nn.Module):
        def __init__(self, in_size):
            super(logistic_regression,self).__init__()
                self.linear = nn.Linear(in_size,1)
        def forward(self,x):
            x = torch.sigmoid(self.linear(x))
            return x
```

Key difference from linear regression:

The sigmoid activation is included directly in the output step.

```
model = logistic_regression (1)
yhat = model(x)
```

Custom module is more explicit and easier to expand, while **sequential** is faster for straightforward.

Making predictions:

- 1D Single-Sample Prediction:

```
print(list(model.parameters()))
  [
   Parameter containing: tensor([[ 0.2294]]),
   Parameter containing: tensor([-0.2380])
  ]

x = torch.tensor([[1.0]])

yhat= model(x)
```

$$b = -0.23, w = 0.23$$

$$\hat{y} = \sigma(-0.23 + 0.23\,x)$$

$$x = 1$$

$$\hat{y} = \sigma(-0.23 + 0.23(\,1))$$

$$\hat{y} = \sigma(0)$$

- 1D Multi-Sample Prediction:

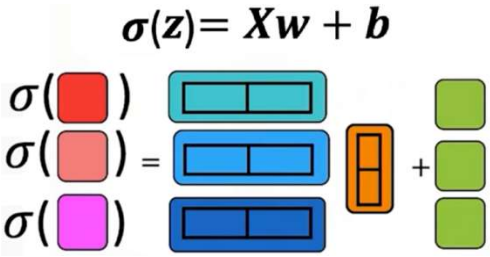
```
x = torch.tensor([[1.0],[100]])
yhat= model(x)

yhat: tensor([
  [ 0.4979],
  [ 1.0000]
])
```

$$\mathbf{x} = \begin{bmatrix} 1 \\ 100 \end{bmatrix}$$

$$\hat{y} = \sigma(-\mathbf{0.23} + 0.23 \begin{bmatrix} 1 \\ 100 \end{bmatrix})$$

$$\hat{y} = \begin{bmatrix} \sigma(0) \\ \sigma(22) \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}$$

Multi-Dimensional Logistic Regression



A multi-dimensional model behaves similar to 1D models. Instead of passing 1 as **`in_features`** parameter, the input of the model will be the number of features. For example:

- The model is defined with **`in_features=2`** (Input is a vector with two features).
- Steps:
  - Linear transformation using weight vector and bias term.
  - Sigmoid applied to intermediate output.
  - Final prediction is a single probability.

```
custom_2d_model = logistic_regression (2)

sequential _2d_model = nn.Sequential(
                          nn.Linear(2,1),
                          nn.Sigmoid()
                       )
```

For prediction:

- Input is a tensor containing multiple 2D samples.
- Each row in the tensor represents one sample.
- The model processes each row independently:
  - Linear transformation → Sigmoid → Probability.
- The result is a column of outputs, each corresponding to one input sample.

```
X=torch.tensor([[1.0,1.0],[1.0,2.0],[1.0,3.0]])
yhat=model(X)
yhat:tensor([[ 0.54],
             [ 0.50],
             [0.46]])
```

☑Takeaways

☑ The **logistic function** compresses outputs to the [0, 1] range, making it suitable for binary classification.

☑ PyTorch offers **two ways** to implement it: nn.Sigmoid (object) and torch.sigmoid() (function).

☑ **nn.Sequential** provides a compact way to define models by chaining layers and activations.

☑ **Custom modules** subclassing nn.Module are more explicit and flexible, though equivalent in output.

☑ Logistic regression predictions follow a clear flow:

☑ Linear function → Sigmoid function → Probability.

☑ Models handle both **single-sample and multi-sample inputs**, in **1D and 2D cases**.

📌 Bernoulli Distribution and Maximum Likelihood Estimation

This section introduces the **Bernoulli distribution** and the concept of **maximum likelihood estimation (MLE)**, which are essential to understanding the probabilistic foundation of logistic regression.

The Bernoulli distribution models binary events.

The maximum likelihood estimation provides a systematic way to infer parameters that best explain observed data.

🔹 Bernoulli Distribution

The Bernoulli distribution models a binary outcome with two possible values, such as success/failure or head/tail. It is parameterized by a single value θ, known as the **Bernoulli parameter**.

- The probability of the outcome being **1** (e.g., “success” or “tail”) is θ.
- The probability of the outcome being **0** (e.g., “failure” or “head”) is $1 - \theta$.

This compact representation means that a single parameter θ is enough to describe the full distribution of outcomes.

The probability mass function of the Bernoulli distribution is expressed as:

$$p(y|\theta) = \theta^{y}(1-\theta)^{1-y}$$

Here:

- **y** is the observed outcome, which can only take the values 0 or 1.
- If **y=1**, the expression simplifies to θ.
- If **y=0**, the expression simplifies to 1−θ.

This general form allows us to compute probabilities for any binary outcome using a single equation.

Likelihood Function

When observing multiple independent events, we want to quantify how likely it is that a given parameter θ explains the entire sequence. This is done using the **likelihood function**, which is the product of the probabilities of all observed outcomes.

If we observe ***n*** independent samples *y1,y2,...,yn* the likelihood function is:

$$p(Y|\theta) = \prod_{n=1}^{N} p(y_n|\theta) = \prod_{n=1}^{N} \theta^{y_n}(1-\theta)^{1-y_n}$$

This formulation captures the probability of observing the entire dataset under a particular choice of parameter θ.

Different values of θ will produce different likelihood values, and the goal is to find the one that makes the observed data most probable.

Example of Bernoulli distribution and Likelihood

Example when theta is known:

In order to demonstrate these concepts, consider a **biased** coin flip, the probability of head and tail is given by the Bernoulli parameter (θ).

The probability of heads is given by θ, and the probability of tails is given by 1−θ.

Heads y = 0	Tails y = 1
0.2	0.8

The **Likelihood** of a sequence of events can be calculated by multiplying the probability of each individual event.

Considering a sequence of three events:

$$\theta \quad x \quad \theta \quad x \quad (1-\theta)$$
$$0.2 \quad x \quad 0.2 \quad x \quad 0.8 \quad = 0.032$$

- 1st flip = head: θ = 0.2 (probability of observing a head).
- 2nd flip = head: θ = 0.2.
- 3rd flip = tail: 1-θ = 0.8 (probability of observing a tail).

The likelihood of these events is obtained by simply multiplying the probabilities.

- **Example when theta is unknown:**

To start a **sample value of theta** is considered, for example:

- θ = 0,5 (probability of observing a head).
- θ = 0,2 (probability of observing a head).

Consider that the coin is tossed n = 4 times:

Parameter	n = 1	n = 2	n = 3	n = 4	Likelihood
θ=0.5	0.5	0.5	0.5	0.5	0.0625
θ=0.2	0.2	0.8	0.2	0.8	0.0256

For the following sequence, the likelihood values for the two parameters equal **0.0625** and **0.0256**, respectively.

Notice that amongst the two values of the likelihood, the value of likelihood corresponding to the parameter theta equals 0.5 is larger compared to the other value.

This intuitively makes sense. In the real world, if you flip a coin, the probability of getting a head or tail is equally likely.

The **value of the likelihood** given by the parameter theta equals **0.5** is **more likely to occur**.

⚠️ The actual parameter θ can be estimated by considering parameter values that **maximize the likelihood**.

Maximum Likelihood Estimation (MLE)

Maximum Likelihood Estimation is the method used to determine the parameter θ that maximizes the likelihood function.

The value of θ that achieves this maximum is considered the best estimate of the true parameter.

Mathematically:

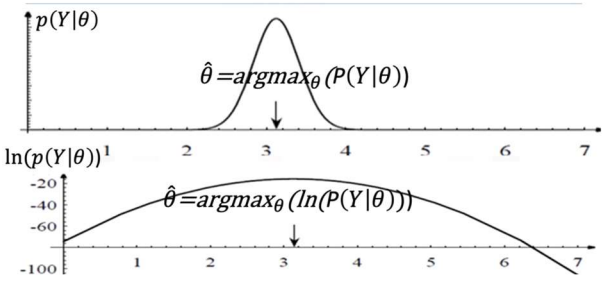
$$\hat{\theta} = argmax_{\theta}\,(P(Y|\theta))$$

Since maximizing a product of probabilities (likelihood function) can be mathematically inconvenient, it is common to instead maximize the **log-likelihood function**.

The logarithm is a monotonic transformation, meaning it preserves the location of the maximum while simplifying computation.

As the log function is monotonically increasing, the location of the maximum value of the parameter remains in the same position.

Image from : Duda, Richard O., Peter E. Hart, and David G. Stork. *Pattern classification*. John Wiley & Sons, 2012.



The log-likelihood is given by:

$$\ell(\theta) = \ln(p(Y|\theta)) = \sum_{n=1}^{N} y_n \ln(\theta) + (1-y_n)\ln(1-\theta)$$

This formulation is much easier to differentiate and optimize, and it forms the basis of parameter estimation in logistic regression.

Connection to Logistic Regression

In logistic regression, the Bernoulli distribution serves as the underlying probabilistic model for the binary outcomes. The likelihood function is constructed from the predicted probabilities of the model, and maximum likelihood estimation is used to adjust the weights and bias so that the model's predictions maximize the probability of the observed training data.

This means that training a logistic regression model is equivalent to finding the parameters that maximize the log-likelihood of the Bernoulli distribution over the dataset.

☑Takeaways

☑ The **Bernoulli distribution** models binary outcomes using a single parameter θ.

☑ The **likelihood function** aggregates probabilities of observed outcomes to measure how well a parameter explains the data.

☑ **Maximum Likelihood Estimation (MLE)** finds the parameter value $\hat{\theta}$ that maximizes this likelihood.

☑ The **log-likelihood** simplifies optimization while preserving the same maximum.

☑ Logistic regression training is based on maximizing the log-likelihood of the Bernoulli distribution with respect to model parameters.

📌 Logistic Regression Cross-Entropy Loss

This section explains why cross-entropy is used as the loss function for logistic regression instead of mean squared error. Addresses the limitations of **mean squared error (MSE)** when applied to classification problems, explaining why cross-entropy offers a more suitable alternative for optimizing models that output class probabilities.

It connects maximum likelihood estimation to the derivation of cross-entropy loss, explains the limitations of threshold-based loss functions, and demonstrates how PyTorch implements logistic regression training with cross-entropy.

🔹 Problem with Mean Squared Error in Classification

In **linear regression**, **mean squared error** (MSE) is effective because outputs are continuous and minimizing squared deviations directly improves predictions.

In **classification**, however, predictions must represent discrete classes, so the objective is to minimize classification errors rather than numerical distance errors.

If MSE is applied to classification, the resulting cost surface becomes flat in certain regions (because it treats the output as a continuous variable instead of a probability). This flatness creates gradients equal to zero (vanishing gradient problem), preventing the model parameters from updating properly during training (the model can get stuck misclassifying samples, unable to adjust further).

- **Example**

The cost function is given by:

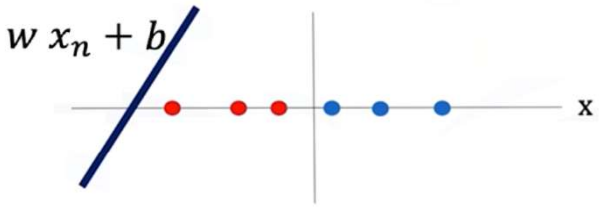
$$l(w,b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - \sigma(w\,x_n + b))^2$$

The following will be a simplified example to demonstrate this problem, so a simplified version of the cost function is used, focusing on the bias term.

$$l(b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - \sigma(x_n + b))^2$$

$$l(b) = \sum_{n=1}^{N}(y_n - THR(x_n + b))^2$$

The three **red** samples have been misclassified, and the **blue** samples have been correctly classified



In the mathematical representation for the loss in this example:

- y_n = 0 for the red samples.
- y_n = 1 for the blue samples.
- THR() = 1 for all of these samples.

Plugging the values of Y_n and the threshold function for all these samples in the loss function:

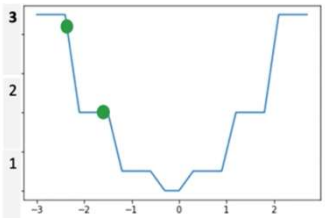
$$l(b) = (0-1)^2 + (0-1)^2 + (0-1)^2 + (1-1)^2 + (1-1)^2 + (1-1)^2$$

$$l(b) = 3$$

Generalizing for this particular threshold function, if two misclassified samples, the cost would be 2, and for 0 misclassified samples 0 is the cost.

Misclassified	3	2	0
Cost	3	2	0

By seeing the plot of the cost, of the threshold function, the gradient descent is obtained from the bias parameter b.

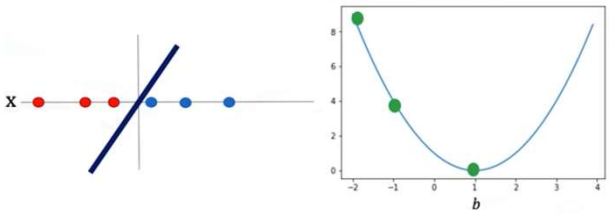


$$b^2 = b^1 - \eta \frac{dl(b^1)}{db}$$

The value for the cost for a specific line is given by the green ball in the plot, as the threshold line moves, the misclassified samples are reduced, and the loss is reduced as well.

But, as the line moves something interesting happens also, the value of the cost function falls in a region with a flat line, the **gradient in this region is 0**, when this happens the parameter will get stuck in this region resulting in none of the parameter values (weights and biases) getting updated for the classifier, so the model cannot effectively adapt to reduce misclassifications.

To address the limitation seen on the example, classification problems replace the threshold function with the **sigmoid function**, which provides smooth gradients across all regions.



Unlike the abrupt jumps of thresholding, the sigmoid curve ensures that parameter updates are always possible, guiding the model closer to the correct decision boundary.

To overcome the flat gradient issue, the loss function must provide smooth gradients that reflect how confidently the model predicts correct versus incorrect labels. This requirement motivates the transition from MSE to **cross-entropy loss**, derived mathematically from Maximum Likelihood Estimation.

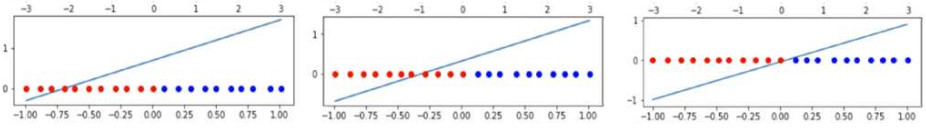
◆ Maximum Likelihood Estimation and Logistic Regression

Logistic regression is built on the probabilistic framework of **Maximum Likelihood Estimation (MLE)**, a statistical approach used to estimate parameters that make the observed data most probable under a model.

Each sample in the dataset belongs to a class y, where $y \in \{0, 1\}$. The model predicts probabilities using the **sigmoid function**, which maps linear combinations of inputs to values between 0 and 1 (probability of a sample belonging to a particular class).

The likelihood function is constructed by multiplying the predicted probabilities across all samples in the training set.

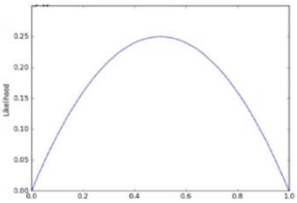
Parameter b	0.69	0.339	0
Likelihood	0.445	0.46	0.47



b = 0.69 b = 0.339 b = 0

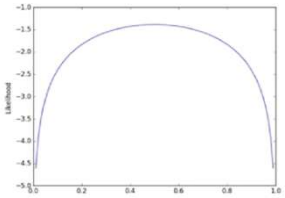
Maximizing this likelihood is equivalent to finding the set of weights and bias that best explain the observed data.

$$p(Y|\mathrm{b},\boldsymbol{X}) = \prod_{n=1}^{N} \sigma(\boldsymbol{w}\mathbf{x}_n + b)^{y_n}(1-\sigma(\boldsymbol{w}\ \ \mathbf{x}_n + b)^{y_n})^{1-y_n}$$



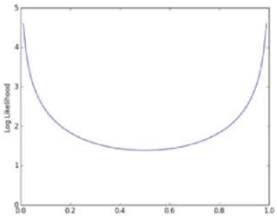
Because direct maximization of the likelihood is inconvenient, the log-likelihood is used instead. This transformation preserves the location of the maximum while simplifying optimization, for numerical stability, resulting in the **log-likelihood function**:

$$ln(p(Y|\mathrm{b},\boldsymbol{X}))$$



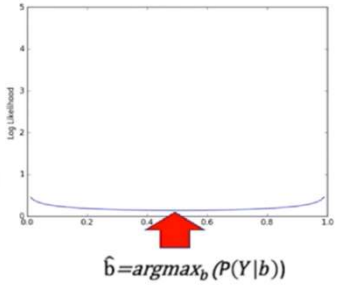
When the log-likelihood is multiplied by -1, the problem **converts** from **maximization to minimization**, which aligns with how optimization algorithms like gradient descent are implemented.

$$-ln(p(Y|\mathrm{b},\boldsymbol{X}))$$



This negative log-likelihood is the basis of the cross-entropy loss. And if the function is average out, the minimum of this functions corresponds to the maximum value of the likelihood.

$$-\frac{1}{N}ln(p(Y|\mathrm{b},\boldsymbol{X}))$$



$$\hat{\mathrm{b}} = argmax_b\,(P(Y|b))$$

Cross-Entropy Loss

The cross-entropy loss quantifies the difference between the predicted class probabilities and the true class labels.

For logistic regression, the loss is defined as:

$$l(\theta) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \,.\ln(\hat{y}_i) + (1-y_i)\,.\ln(1-\hat{y}_i)]$$

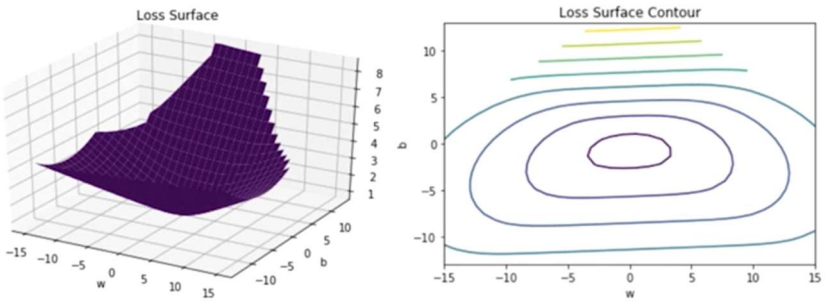
$$l(\theta) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \,.\ln(\sigma(w.x_i+b)) + (1-y_i)\,.\ln(1-\sigma(w.x_i+b))]$$

Here:

- y_i is the true class label of the i[th] sample.
- $y_i = \sigma(w.x_i + b)$ is the predicted probability of belonging to class 1.
- θ represents the model parameters (weights and bias).

This loss function penalizes confident but incorrect predictions more heavily than less confident ones. As a result, the model is encouraged to adjust its parameters toward probabilities that align with the true distribution of classes.

Unlike MSE, the cross-entropy cost surface remains smooth and convex, ensuring stable gradient values throughout training. This property allows gradient descent to converge effectively toward parameter values that minimize misclassification.

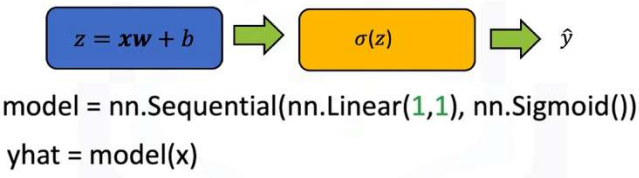


Logistic Regression Training in PyTorch

Implementing logistic regression in PyTorch involves the following key components:

1. **Model creation:**

   🛠️ A logistic regression model can be created using **`nn.Sequential`**, combining a linear layer with a sigmoid activation.




```
model = nn.Sequential(nn.Linear(1,1), nn.Sigmoid())
yhat = model(x)
```

🛠️ Alternatively, a custom model can be defined by subclassing **`nn.Module`**, explicitly including the linear transformation and sigmoid activation in the forward pass.

```
import torch.nn as nn

class logistic_reg(nn.Module):
    def __init__(self, in_dim, 1):
        super(logistic_reg, self).__init__()
        self.linear = nn.Linear(in_dim, 1)

    def forward(self,x):
        out =  nn.sigmoid(self.linear(x))
        return out
```

2. **Loss function:**

   Loss function is used for updating the weight and bias of the model.

   🔧 While PyTorch provides **`nn.MSELoss`**, it is not ideal for classification.

```
def criterion(yhat,y):
    return torch.mean((yhat-y)**2)

or

criterion = nn.MSELoss()
```

   🔧 Instead, the preferred function is **`nn.BCELoss`**, which directly implements the cross-entropy formulation for binary classification.

```
def criterion(yhat,y):
    out=-1*torch.mean(y*torch.log(yhat) +(1-y)*torch.log(1-yhat))
    return out

or

criterion=nn.BCELoss()
```

3. **Define training parameters:**

   🔗 Process is started by loading the dataset, creating the logistic regression model, and selecting the optimizer (for updating the model parameters).

```
dataset = Data()
trainloader = DataLoader(dataset = dataset,batch_size=1)

model = logistic_reg (1,1)

optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

4. **Training loop:**

   - Input data is passed to the model to produce predictions.
   - The loss between predictions and true labels is calculated using cross-entropy.
   - Gradients of the loss with respect to model parameters are computed via **`loss.backward()`**.
   - Parameters are updated using optimizers such as stochastic gradient descent (SGD) with a defined learning rate.
   - Repeating this process for multiple epochs gradually reduces loss and improves classification accuracy.

```
for epoch in range(100):
    for x,y in  trainloader :
        yhat = model(x)
        loss = criterion (yhat , y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

By the end of training, the logistic regression model outputs probabilities between 0 and 1.

To assign a final class label, a threshold (typically 0.5) is applied: predictions greater than or equal to 0.5 are classified as class 1, while predictions less than 0.5 are classified as class 0.

☑Takeaways

☑ Mean squared error is unsuitable for classification, it creates flat cost surfaces that block parameter updates.

☑ Logistic regression is derived from maximum likelihood estimation, which seeks parameters that maximize the probability of the observed data.

☑ Cross-entropy loss is the negative log-likelihood of the Bernoulli distribution, providing a smooth and convex cost function for classification tasks.

☑ In PyTorch, cross-entropy loss is efficiently implemented using **`nn.BCELoss`**, and logistic regression can be trained using standard optimization procedures with SGD.

☑ The use of cross-entropy ensures that logistic regression models learn parameter values that minimize misclassifications and generalize effectively to new data