# 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

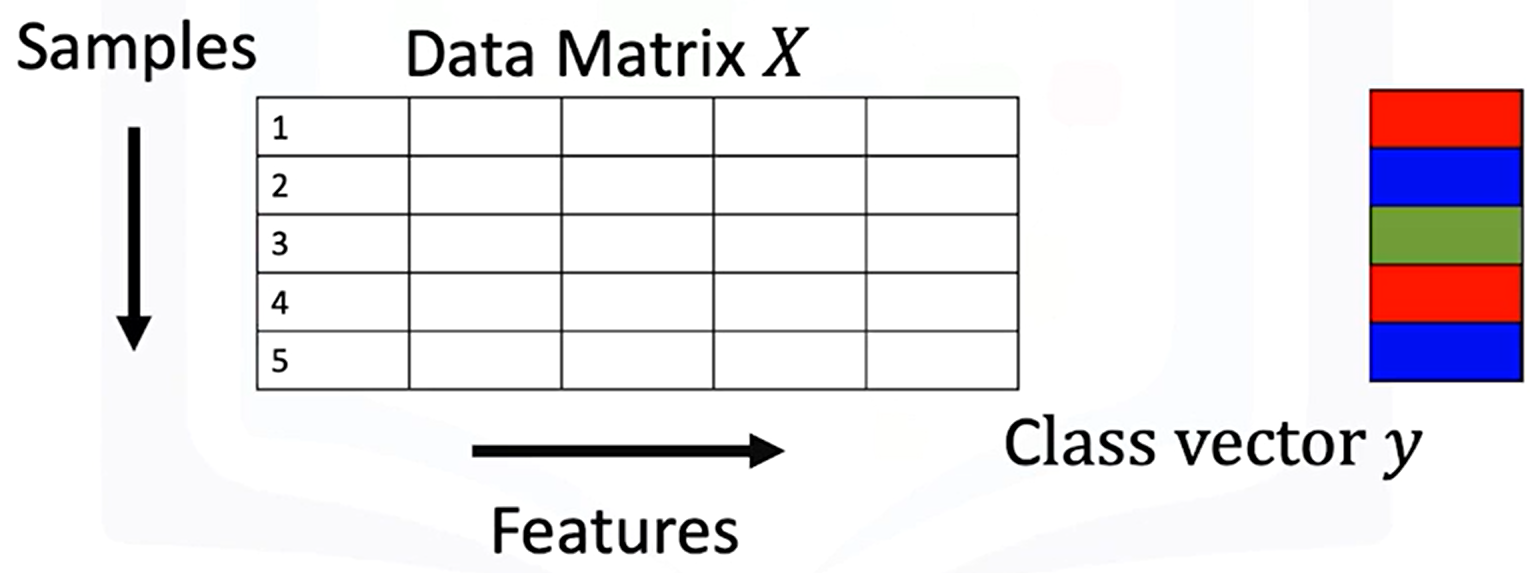
Logistic regression is about predicting which class a particular sample belongs to based on their features.

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

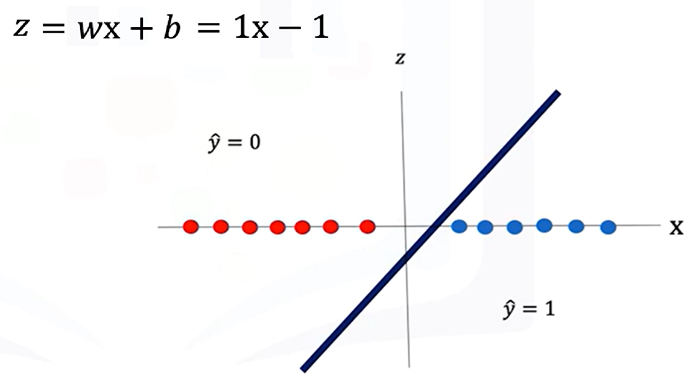
A **linear classifier** separates data points from two classes using a line (or hyperplane in higher dimensions).



|  |  |
| --- | --- |
| **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).

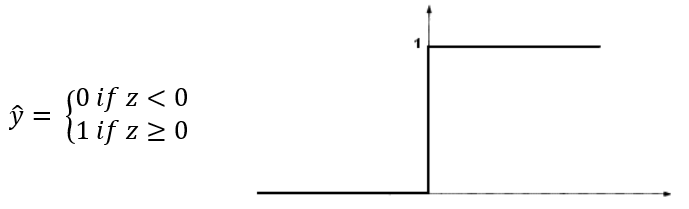
**Example:**

* If , classify as class 1.
* If , 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:**

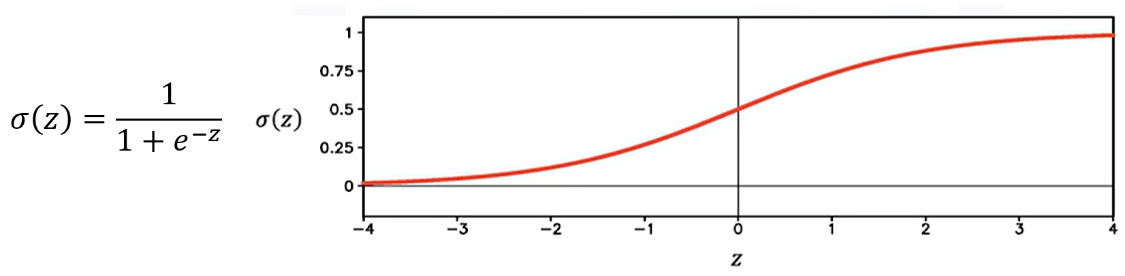
A **threshold function** maps continuous z values into discrete classes:



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**:



* 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 **σ(z)>0.5**, predict 1.
* If **σ(z)≤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:



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

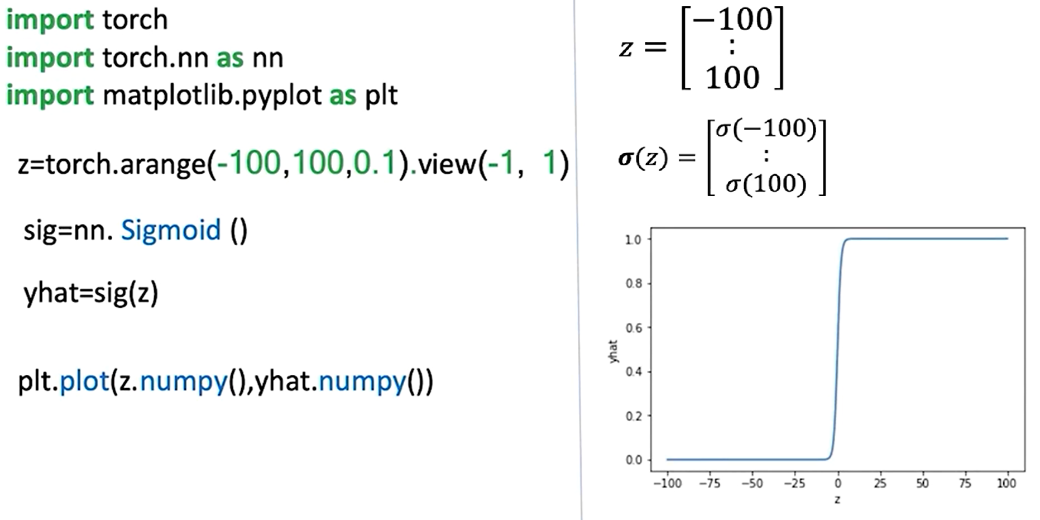
The logistic function (also called the **sigmoid function**) takes a real-valued number and compresses it into a range between 0 and 1. This is critical for classification tasks because the output can be interpreted as a **probability**.

In logistic regression, the **linear function** output is passed through the **logistic (sigmoid) function** to produce the prediction ​.

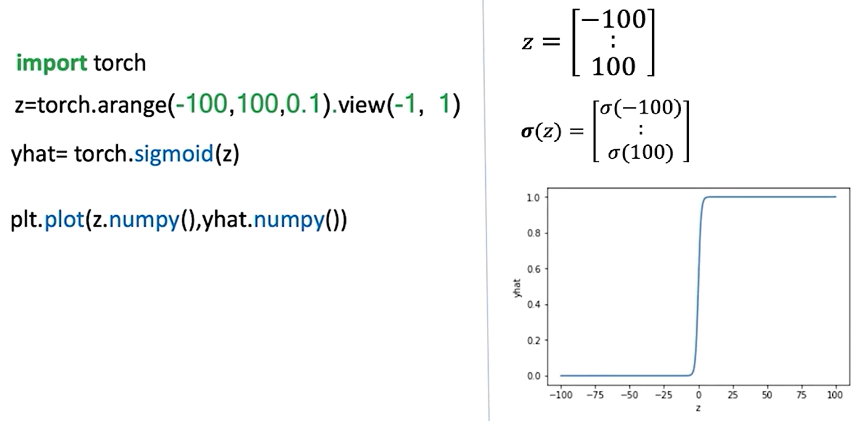
* The output is still **one-dimensional** even when the input involves vector operations (e.g., dot products).

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.



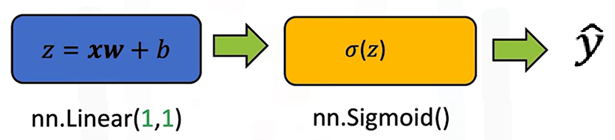
1. **Using torch.sigmoid() function**
   * Call **torch.sigmoid(tensor)** directly to compute the logistic transformation.



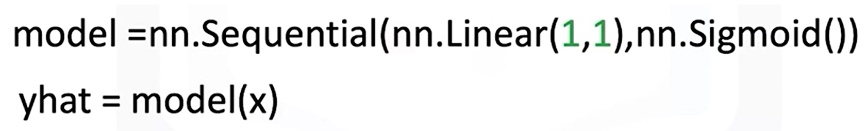
### 🔹 Building Logistic Regression Models with nn.Sequential

**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.



* **Processing flow:**

1. Input tensor → Linear layer → intermediate output z
2. z → Sigmoid function → ​

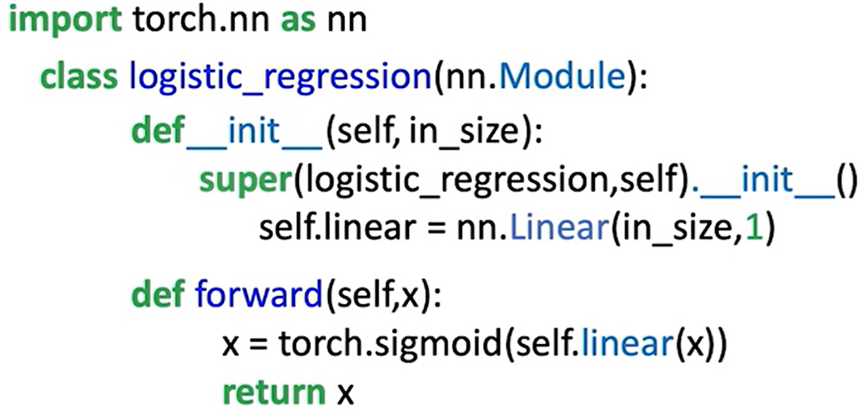
This approach is compact, and PyTorch automatically connects the layers in the sequence.

### 🔹 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.

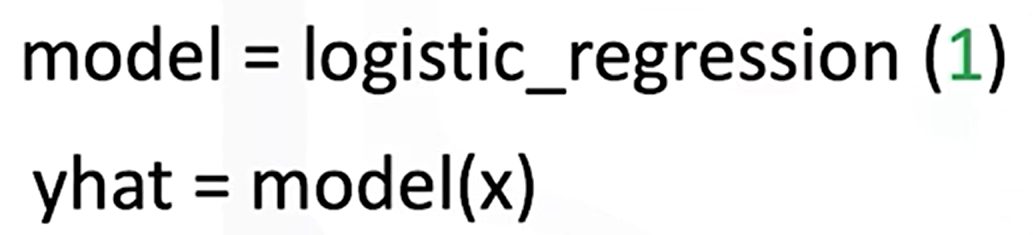
**Structure of the custom class:**

* 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 ​​.



**Key difference from linear regression:**

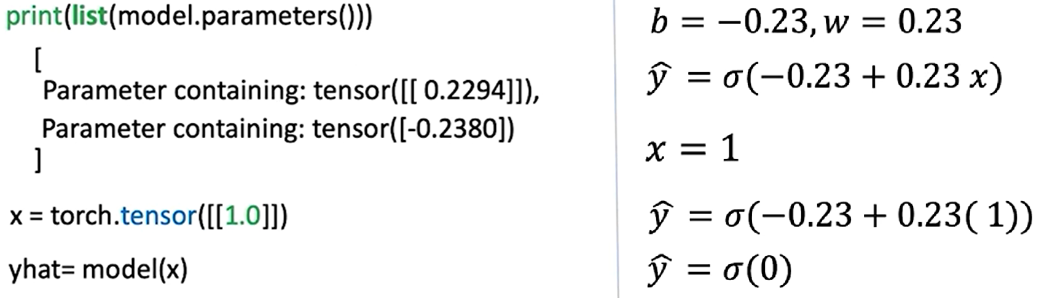
The sigmoid activation is included directly in the output step.



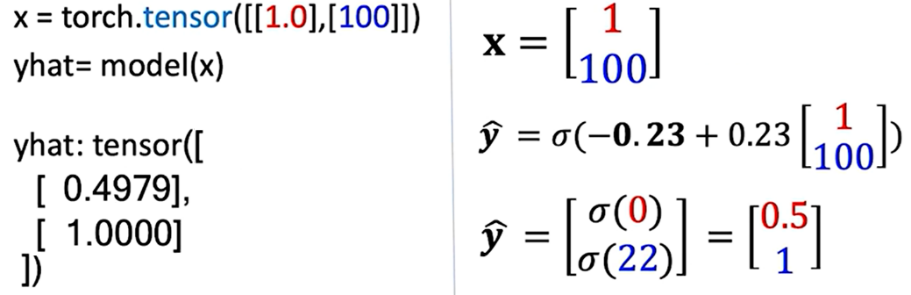
**Custom module** is more explicit and easier to expand, while **sequential** is faster for straightforward.

**🔸Making predictions:**

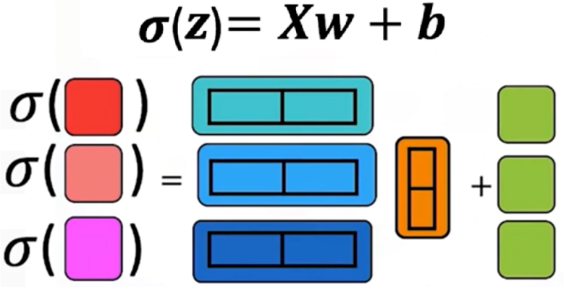
* 1D Single-Sample Prediction:



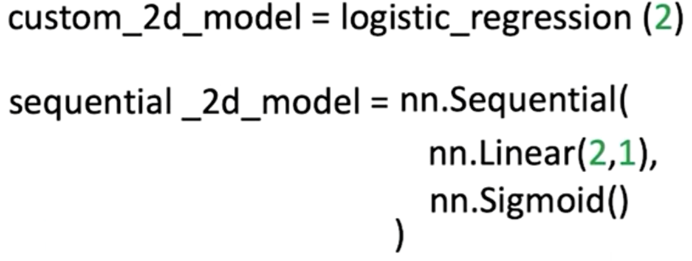
* 1D Multi-Sample Prediction:



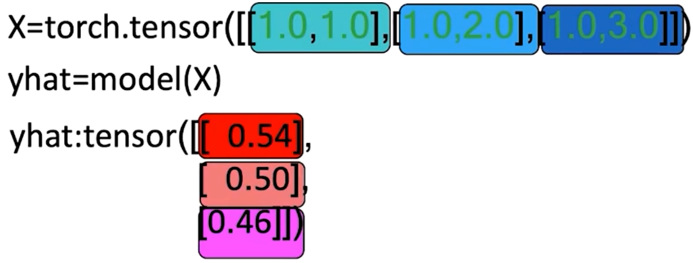
### 🔹 Multi-Dimensional Logistic Regression



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.

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.

### ✅ 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.

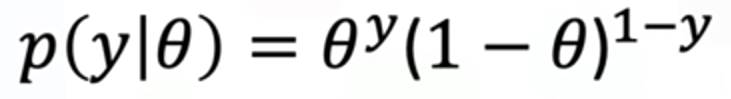
### 🔹 Bernoulli Distribution

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 − θ.

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:



Here:

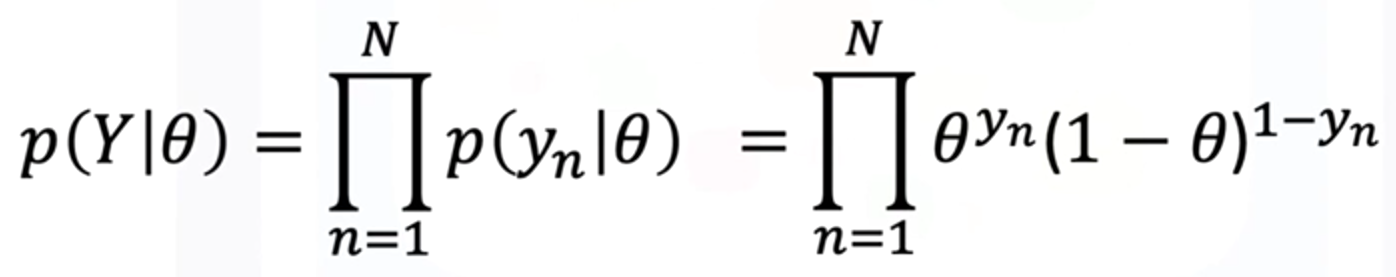
* **y** is the observed outcome, which can only take the values 0 or 1.
* If **y=1**, the expression simplifies to θ.
* If **y=0**, the expression simplifies to 1−θ.

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:



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

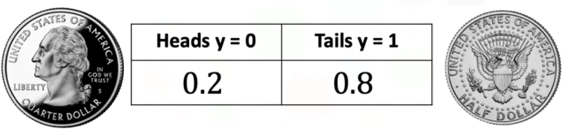
Different values of θ will produce different likelihood values, and the goal is to find the one that makes the observed data most probable.

### 🔹 Example of Bernoulli distribution and Likelihood

**🔸Example when theta is known:**

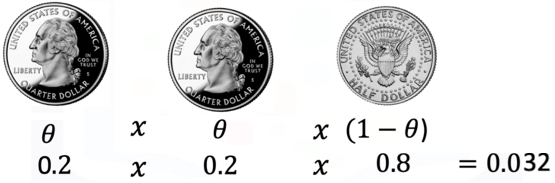
In order to demonstrate these concepts, consider a **biased** coin flip, the probability of head and tail is given by the Bernoulli parameter (θ).

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



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

Considering a sequence of three events:



* + - 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).

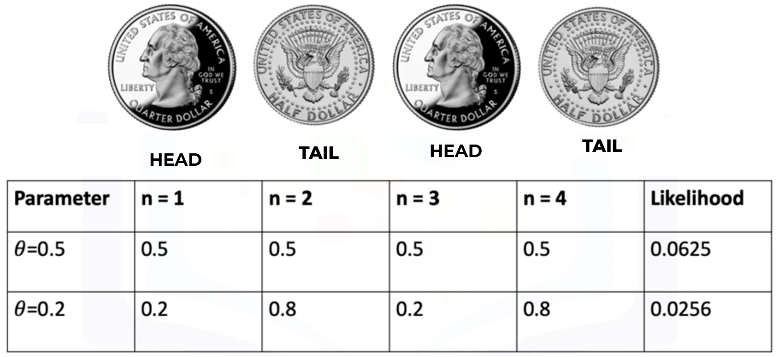
The likelihood of these events is obtained by simply multiplying the probabilities.

**🔸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:



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**.

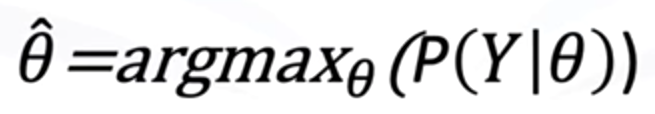
⚠️The actual parameter θ can be estimated by considering parameter values that **maximize the likelihood**.

### 🔹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.

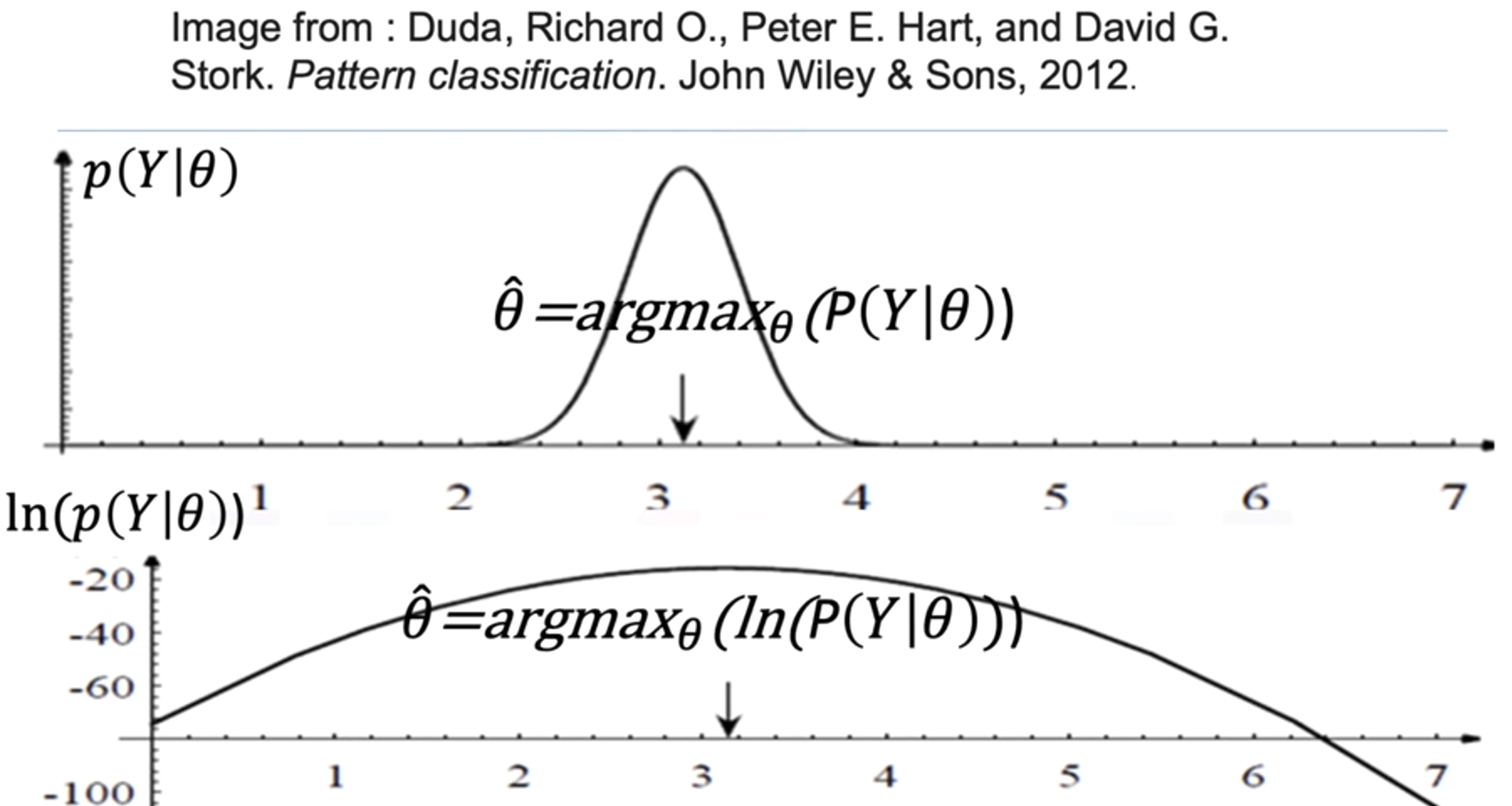
Mathematically:



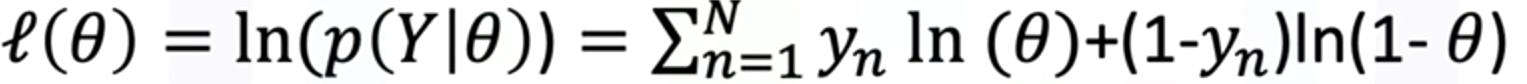
Since maximizing a product of probabilities (likelihood function) can be mathematically inconvenient, it is common to instead maximize the **log-likelihood function**.

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.



The log-likelihood is given by:



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 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.

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.

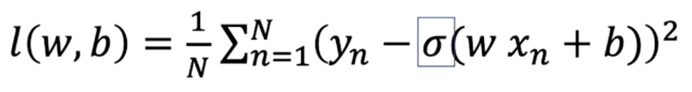
### 🔹 Problem with Mean Squared Error in Classification

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

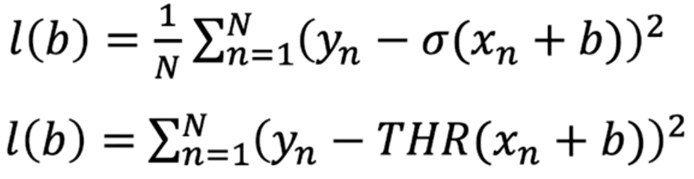
In **classification**, however, predictions must represent discrete classes. If MSE is applied to classification, the resulting cost surface becomes flat in certain regions. This flatness creates gradients equal to zero, preventing the model parameters from updating properly during training. As a result, the model can get stuck misclassifying samples, unable to adjust further.

**🔸Example**

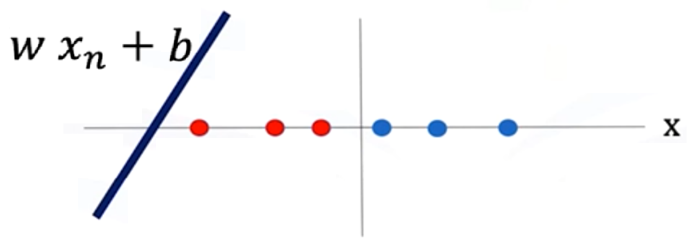
The cost function is given by:



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.



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



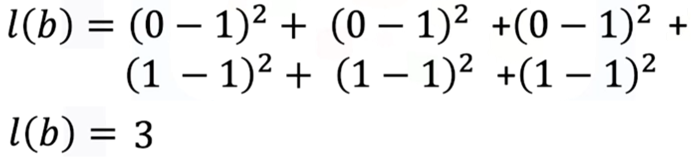
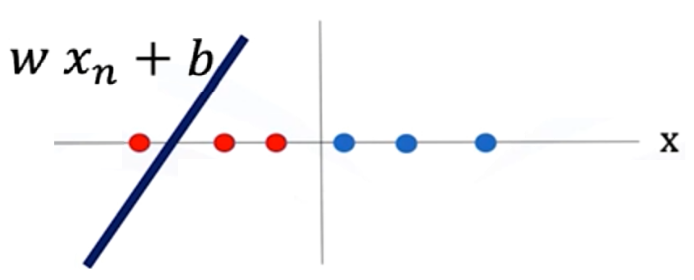
In the mathematical representation for the loss in this example:

yn = 0 for the red samples.

yn = 1 for the blue samples.

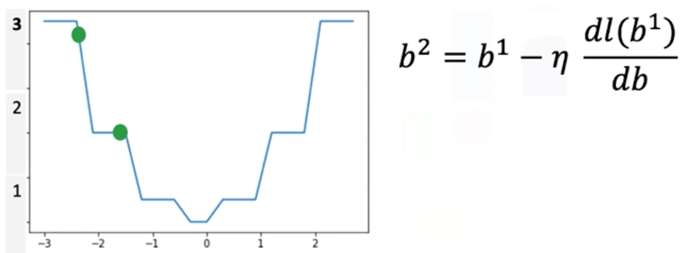
THR() = 1 for all of these samples.

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



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

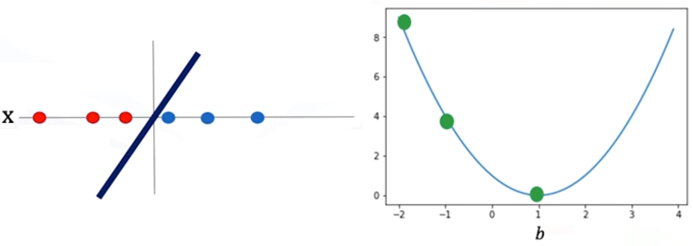
By seeing the plot of the cost, of the threshold function, the gradient descent is obtained from the bias parameter b.



The value for the cost for a specific line is given by the green ball in the plot, as the 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 getting updated for the classifier.

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.

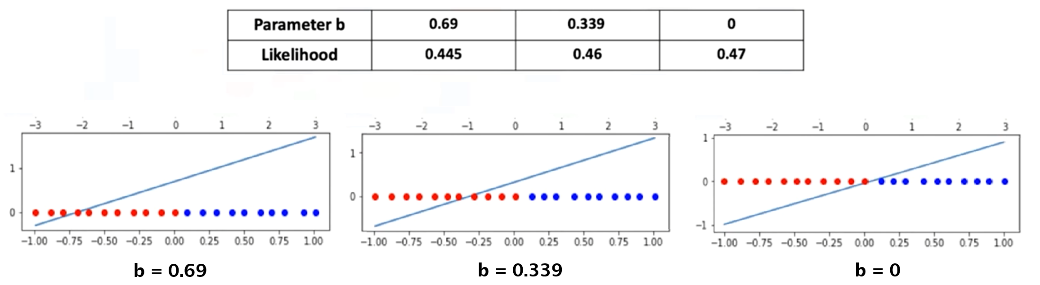


Unlike the abrupt jumps of thresholding, the sigmoid curve ensures that parameter updates are always possible, guiding the model closer to the correct decision boundary.

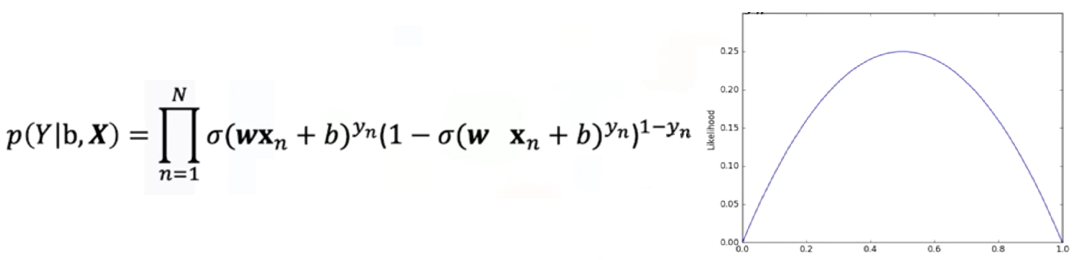
### 🔹 Maximum Likelihood Estimation and Logistic Regression

Logistic regression is built on the probabilistic framework of maximum likelihood estimation (MLE). Each sample in the dataset belongs to a class 𝑦, where 𝑦 ∈ {0, 1}. The logistic function provides the estimated probability of a sample belonging to a particular class.

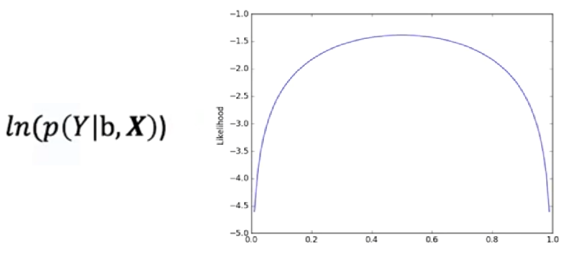
The likelihood function is constructed by multiplying the predicted probabilities across all samples in the training set.



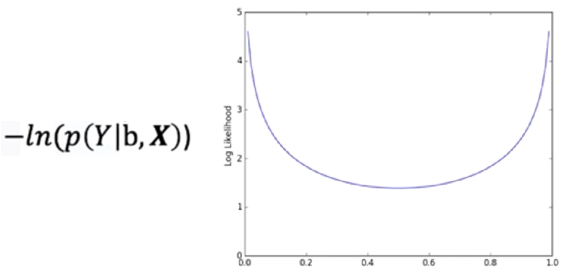
Maximizing this likelihood is equivalent to finding the set of weights and bias that best explain the observed data.



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.



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.



This negative log-likelihood is the basis of the cross-entropy loss.

### 🔹 Cross-Entropy Loss

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

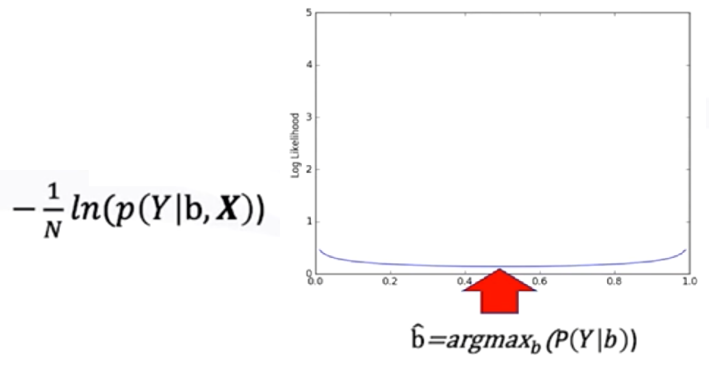
For logistic regression, the loss is defined as:

Here:

* ​ is the true class label of the ith sample.
* ​ 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.

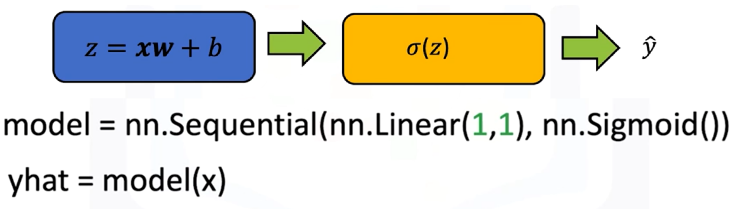


### 🔹 Logistic Regression Training in PyTorch

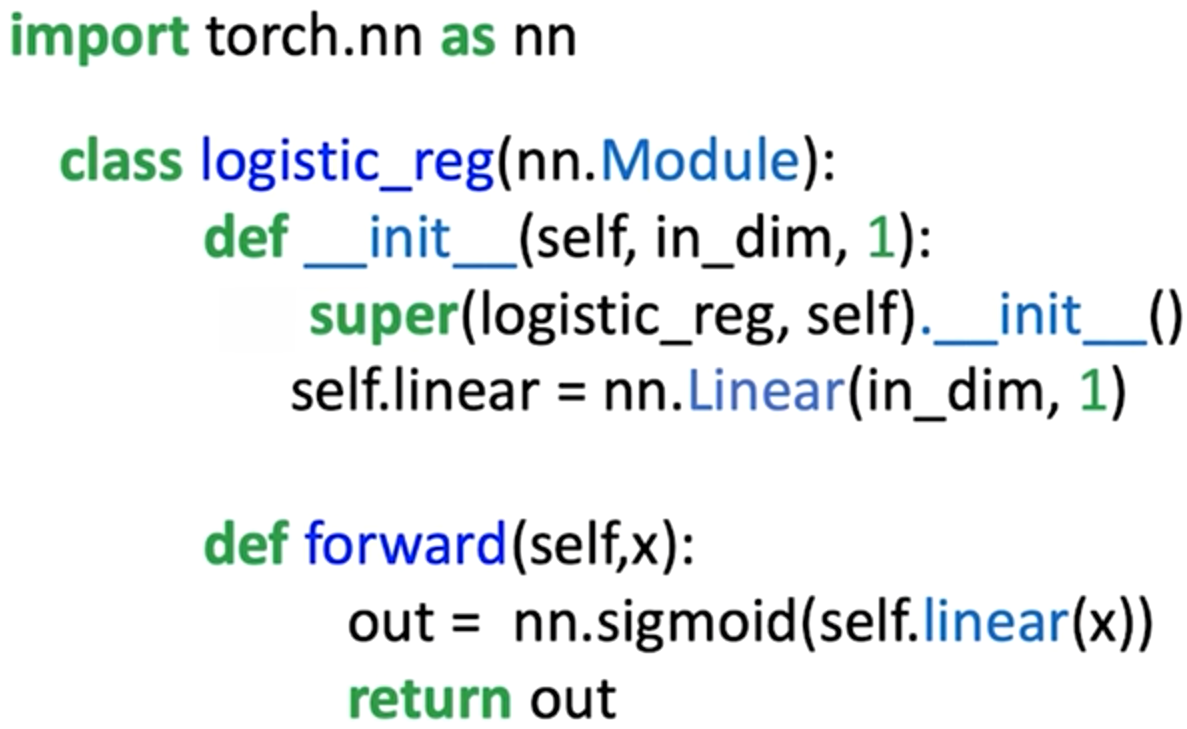
Implementing logistic regression in PyTorch involves the following key components:

1. **Model creation:**

🛠 A logistic regression model can be created using **nn.Sequential**, combining a linear layer with a sigmoid activation.



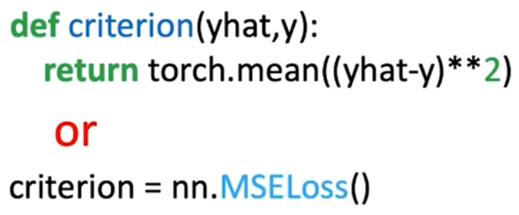
🛠 Alternatively, a custom model can be defined by subclassing **nn.Module**, explicitly including the linear transformation and sigmoid activation in the forward pass.

****

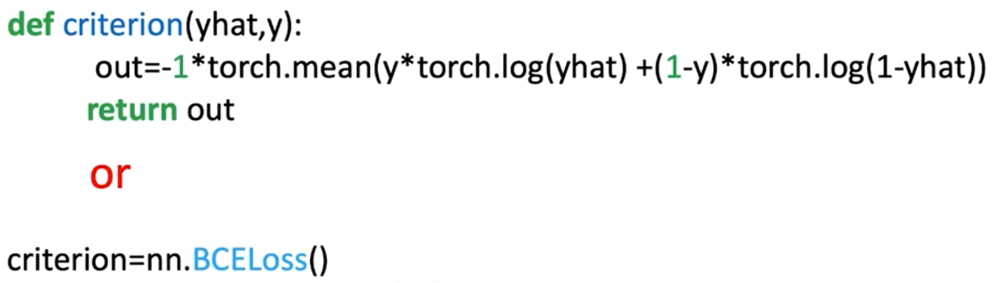
1. **Loss function:**

Loss function is used for updating the weight and bias of the model.

🔧 While PyTorch provides **nn.MSELoss**, it is not ideal for classification.

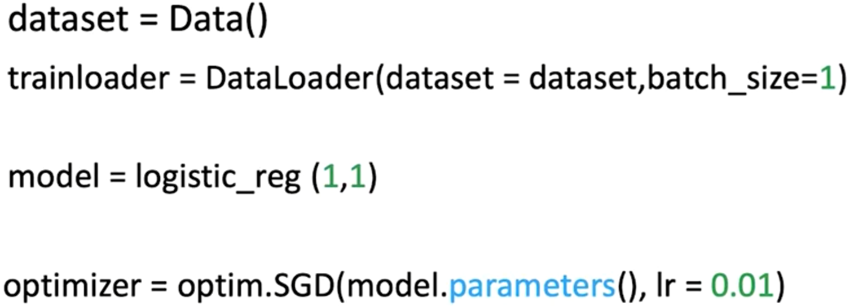


🔧 Instead, the preferred function is **nn.BCELoss**, which directly implements the cross-entropy formulation for binary classification.

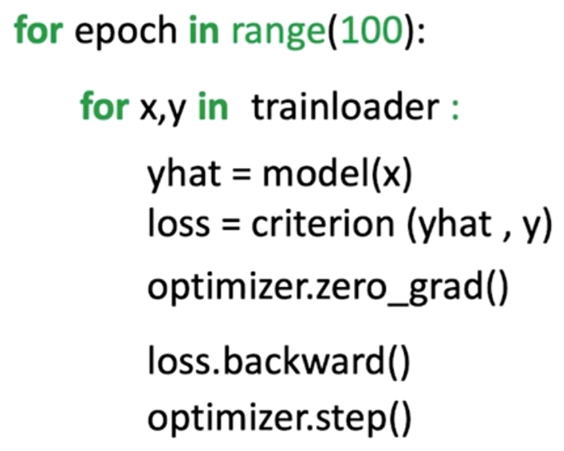
****

1. **Define training parameters:**

🔗 Process is started by loading the dataset, creating the logistic regression model, and selecting the optimizer (for updating the model parameters).



1. **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.

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.