# Module 2

**Softmax**

## 📌Softmax

This section introduces the **Softmax function** and explains how it extends **logistic regression** to handle **multi-class classification** problems. While logistic regression assigns probabilities to two classes using the **sigmoid function**, Softmax generalizes this approach to situations where there are more than two possible output classes.

The Softmax function converts raw model outputs—often called **logits**—into probabilities that sum to one, where the predicted class corresponds to the highest probability value.

**Argmax function** is introduced as a key mechanism for selecting the predicted class from Softmax’s probability outputs.

### 🔹 Introduction to the Softmax Function

The **Softmax function** transforms a vector of unnormalized values (logits) into a normalized probability distribution. Each element of the output vector represents the probability that an input belongs to a specific class.

Mathematically, the function is defined as:

Where:

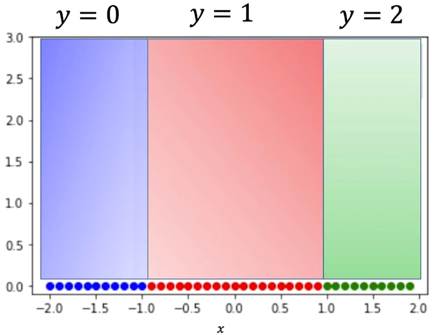
* ​ is the raw score or logit for class *i*.
* *K* is the total number of classes.
* ​ ensures all outputs are positive.
* Denominator normalizes the results so that the sum of all probabilities equals 1.

This formulation ensures that the output values can be interpreted as probabilities while maintaining differentiability, which is essential for optimization through gradient descent.

Softmax is typically used at the output layer of a neural network when performing classification with more than two classes. It enables the model to evaluate all classes simultaneously and express confidence levels for each class prediction.

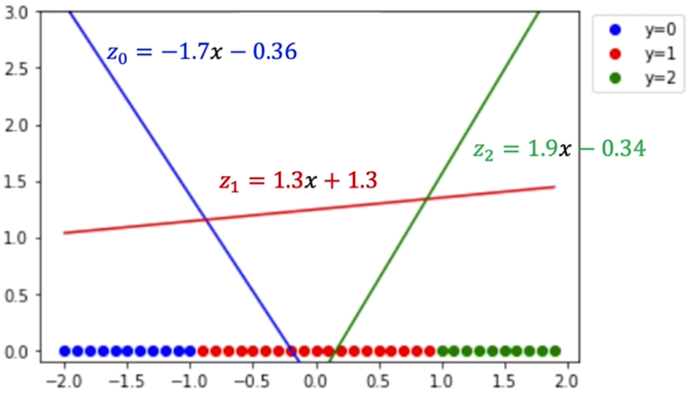
### 🔹 Softmax in One Dimension

To understand Softmax in a simple setting, consider a **1D example** with three classes, represented as blue, red, and green regions along the x-axis. Each class corresponds to a separate **linear function**, defined by its own weight (​) and bias (​):

From this plot we can observe that any point in the:

* + - * + Blue region will be classified as blue (class 0).
        + Red region will be classified as red (class 1).
        + Green region will be classified as green (class 2).

Different lines can be used to classify these points:



For each input value of x, these linear functions produce three outputs: ​. The Softmax function compares these outputs to determine which class has the highest score.

|  |  |  |
| --- | --- | --- |
| If ​, the point is classified as class 0 (blue). | If ​, the point is classified as class 1 (red). | If ​, the point is classified as class 2 (green). |

In this 1D visualization, the regions along the x-axis correspond to where each linear function dominates. These intersections represent **decision boundaries**, where the predicted class changes from one to another.

By observing how each line’s output varies with x, Softmax effectively generalizes logistic regression to support multiple output categories, ensuring every sample is assigned to the most probable class based on relative magnitudes of the logits.

### 🔹 Argmax Function

The argmax function returns the index of the largest value in a sequence of numbers.

For example:

* If , then , because the first element (100) is the largest.
* If , then , corresponding to the second value (8).

In classification, the **argmax** function identifies which class has the highest score or probability. When combined with the Softmax function, argmax is used to choose the predicted class label from the probability distribution that Softmax produces.

### 🔹 Combining Argmax and Softmax

The synergy between **Softmax** and **argmax** forms the basis of multi-class prediction.

1. The **Softmax function** converts each raw output ​ into a probability ​.
2. The **argmax function** identifies the class index with the highest probability.

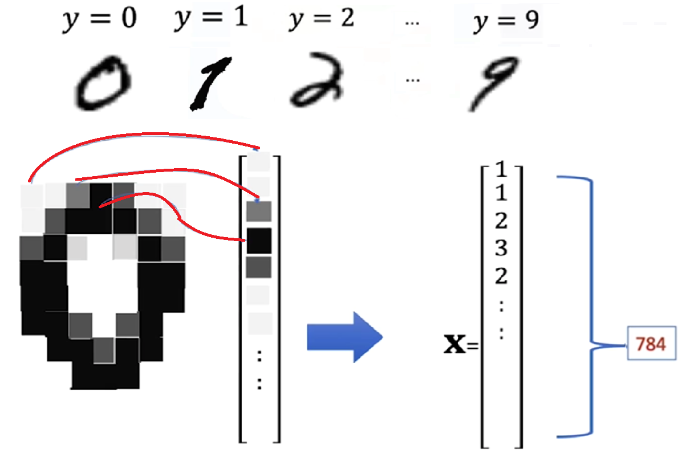
This combination allows the model to both express confidence in each class and make a discrete final decision.

Consider the previous example, each line produces an output score (logit) for a given value of x:

|  |  |  |
| --- | --- | --- |
| Input of x = -1.5 | Input of x = 0.5 | Input of x = 1.5 |

### 🔹 Softmax General (2D)

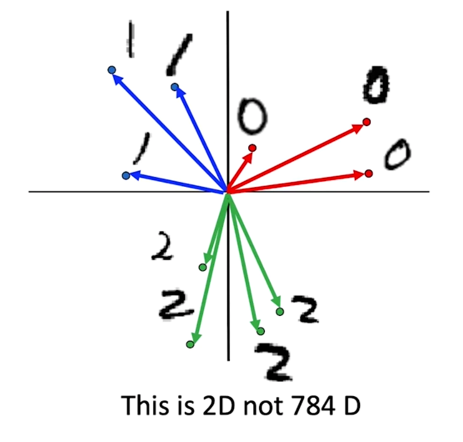
The general use case of softmax is when the input is multidimensional.

To illustrate how it works in a 2D feature space, the **MNIST** dataset will be used. MNIST is used for classifying handwritten digits into different classes ranging from 0 to 9.

Each image is a grayscale image:

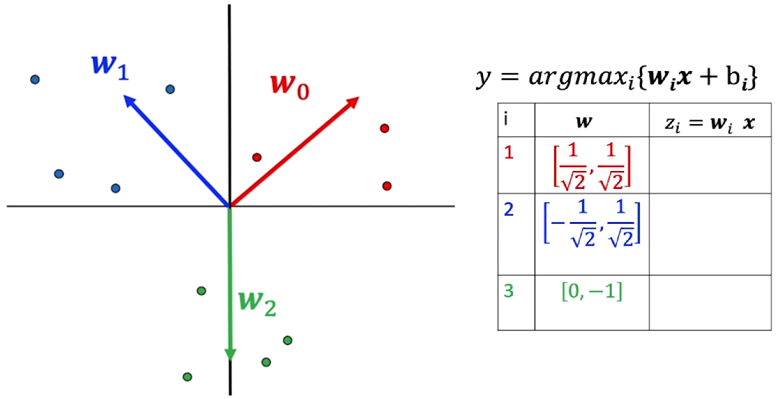
* Contains 784 pixels values (28x28), which can be treated as a high-dimensional vector.
* For each pixel the intensity values can range from 0 to 255

To simplify the visualization, the explanation is reduced to a 2D case (because visualizing and plotting 784 dimensions would be difficult), where each data point is represented as a vector .



These vectors represent the parameters of softmax in 2D. The softmax function is used for finding the points nearest to each parameter vector.

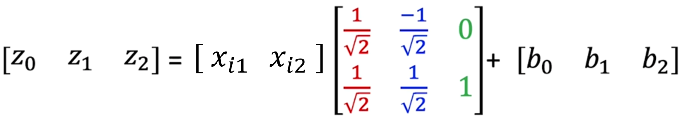
In this geometric view:

* **Each weight vector ​** defines a direction in the feature space.
* **Each sample** is classified based on which weight vector it is most aligned with.
* Points near ​ are classified as class 0 (blue).
* Points near ​ are classified as class 1 (red).
* Points near **​** are classified as class 2 (green).
* **Decision boundaries** emerge where two class scores ​ and ​ are equal, dividing the space into regions corresponding to each class.

For the sake of simplicity, the vectors used in the example represent their respective digit in 2D.

|  |  |
| --- | --- |
|  |  |

Each ***class i*** has an associated **weight vector wi** and **bias bi**. The output score for each class is computed by performing the dot product of with each of the **w vectors**.



The **softmax** transformation then converts these scores into probabilities, one for each class. Then by using the **argmax** function on the computed probabilities, the corresponding class is obtained, by obtaining the index with the highest value.

### ✅ Takeaways

✅ **Softmax Generalization:** Extends logistic regression to multi-class problems by transforming logits into probabilities that sum to one.

✅ **1D to 2D Intuition:** Builds understanding of class boundaries through geometric visualization, from simple one-dimensional data to two-dimensional representations.

✅ **Argmax Role:** Selects the class with the highest probability, forming the final classification decision.

✅ **Decision Boundaries:** Softmax creates smooth and differentiable boundaries, allowing effective gradient-based optimization.

✅ **Visualization Accuracy:** Correct region boundaries, line alignment, and labeling are crucial for accurately interpreting Softmax behavior.

✅ **Foundation for Implementation:** This conceptual understanding prepares for the next section, where Softmax will be implemented in PyTorch and integrated with cross-entropy loss for practical classification tasks.

## 📌Softmax PyTorch

This section demonstrates how to structure a Softmax-based classifier, define its **input and output dimensions**, configure the **loss function and optimizer**, and monitor **classification accuracy**.

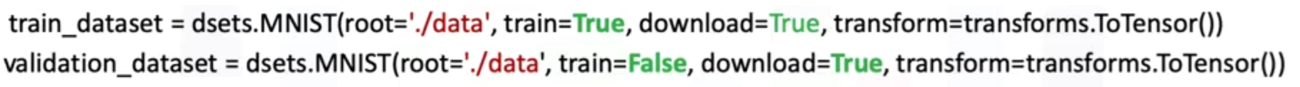
The process also emphasizes how PyTorch automates Softmax computation within **cross-entropy loss**, allowing efficient training for image classification tasks.

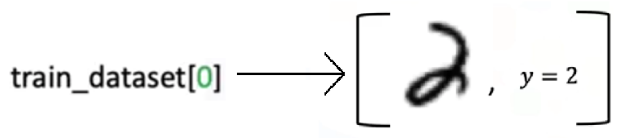
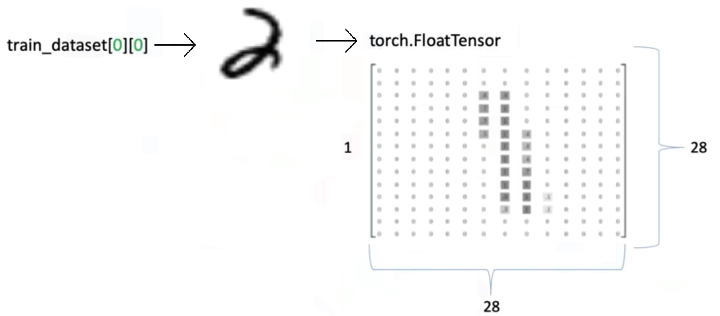
By using the **MNIST dataset**, full workflow is illustrated —from **data loading** and **model construction** to **training**, **validation**, and **evaluation**.

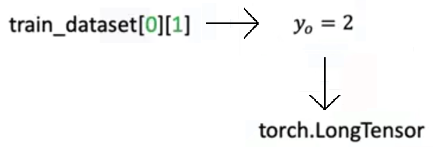
### 🔹 Loading and Preparing the Dataset

To demonstrate softmax workflow the **MNIST dataset will be used**, it contains grayscale images of handwritten digits (0–9), each with dimensions **28 × 28 pixels**.

* + 1. **Importing required modules:**
* **torch** for tensor computation.
* **torchvision.transforms** for image preprocessing.
* **torchvision.datasets** for accessing prebuilt datasets like MNIST.
  + 1. **Applying transformations:**
* Dataset is splitted into training (**train = True**) and validation (**train = False**).



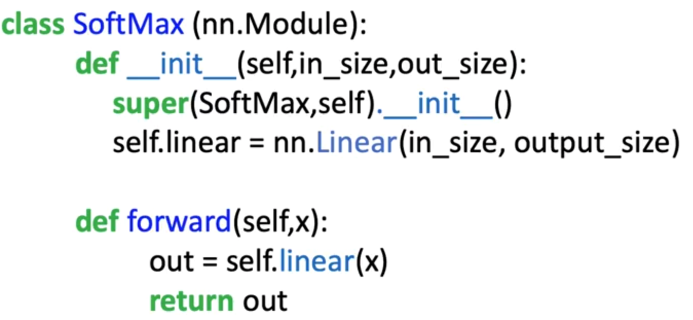
* The transformation **transforms.ToTensor()** converts the images from PIL format to PyTorch tensors. Each pixel intensity is normalized to the range [0,1], and the resulting tensor has a shape of [1, 28, 28] per image.
* Each element on the datasets contains a tuple.
* **First** element is the actual MNIST image, loaded as a tensor of shape [28, 28].
* **Second** element tuple is the label or actual class of the image.

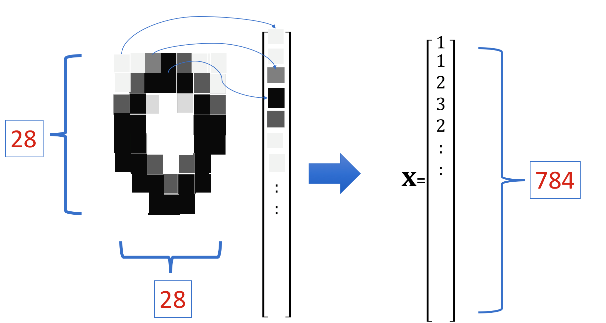


### 🔹 Defining the Softmax Classifier

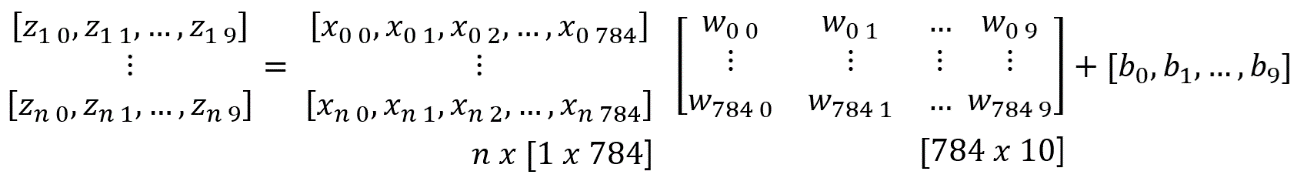
A **custom Softmax module** can be implemented in PyTorch by subclassing nn.Module. This mirrors the structure of logistic regression but includes a configurable output size to accommodate multiple classes.

**🔸Implementation outline:**

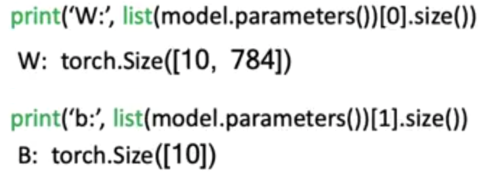
1. **Constructor:**
   * Accepts two parameters:
     + Input size (number of features).
     + Output size (number of classes).
   * Defines a linear layer using nn.Linear(in\_features, out\_features).
2. **Forward Pass:**
   * Passes the input tensor through the linear transformation:
   * Returns z, **raw logits**, without applying a logistic or sigmoid function.

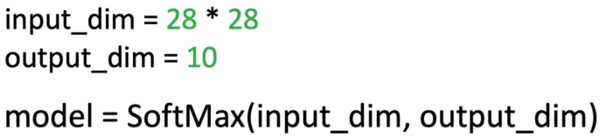
**🔸Considerations:**

* **Images:**
  + Each 28x28 image will be reshaped into **1D vector** of 784 elements.   
    Thus, the **input dimension** is **784** (each pixel is represented as a independent feature).
* **Output dimension:**
  + MNIST has **10 digits** classes (0-9), the **output dimension** of the softmax classifier is 10.   
    Each **output neuron** corresponds to one possible digit class.
* **Model parameters:**
  + Each of the 10 outputs classes has its own weight vector and bias parameter.
  + Each weight vector contains **784 elements**, one for each pixel.
  + The total parameters include 10 weight vectors (one per class) and 10 bias values.



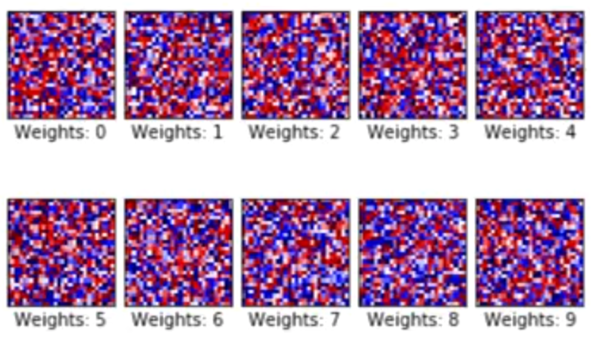
**🔸Parameter Initialization and Visualization:**

1. ****Input and output dimension are declared to initialize the model

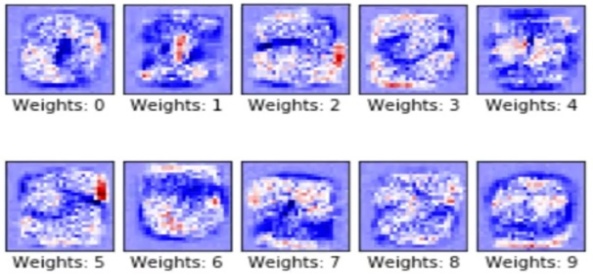


* + Initially, the weight matrices are randomly initialized, appearing as random noise when visualized.

The weights will evolve during training to form distinguishable patterns representing digits 0–9.



Initialized weights



Trained weights

Each weight vector is learning to recognize specific features corresponding to one of the digit categories, effectively forming a **template representation** for each class.

### 🔹 Configuring the Loss Function and Optimizer

🔸 **Loss Function:** Cross-Entropy Loss:



The **cross-entropy loss** function combines two operations:

* It **applies Softmax internally** to the model’s raw outputs (logits).
* It then computes the **negative log-likelihood loss** between predicted probabilities and actual class labels.

Thus, there is **no need to explicitly apply Softmax** in the forward pass when using this loss function.

⚠️The input for the loss function (criterion) **needs** to be a **long tensor** with dimension **n**, instead of a nx1 (used for linear regression).

🔸 **Optimizer Configuration:**

The optimizer used is **Stochastic Gradient Descent (SGD)**, defined as:



This optimizer updates weights based on gradient descent after each batch iteration.

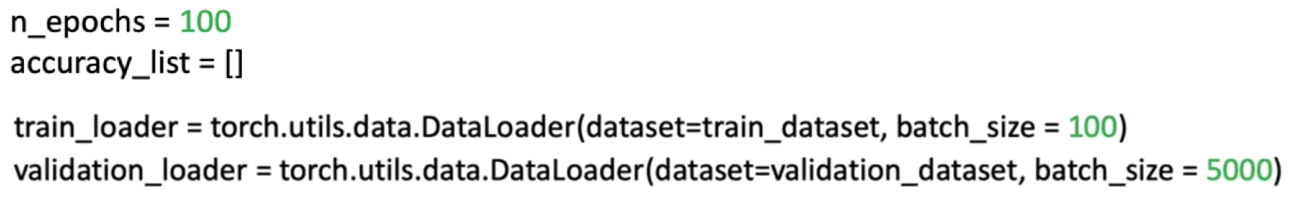
### 🔹 Training the model:

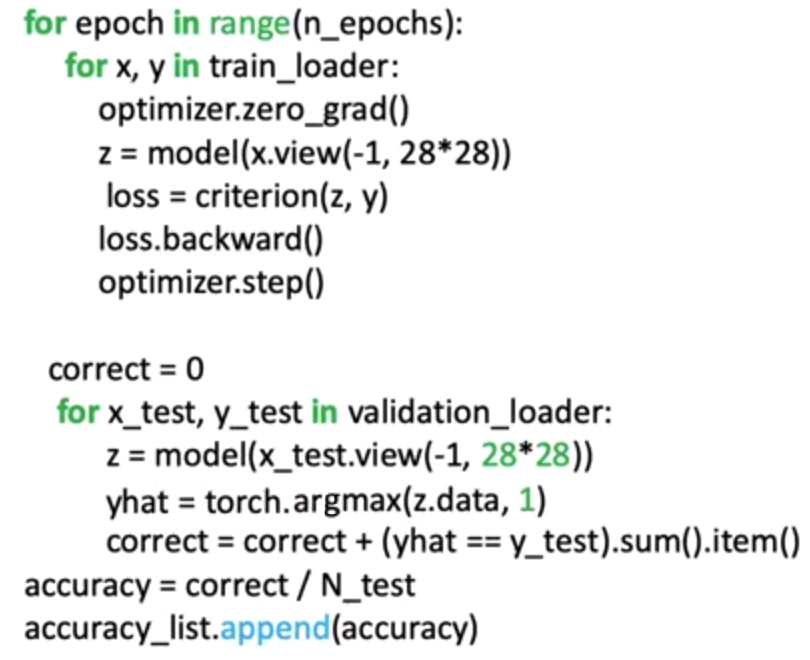
The **training loop** executes across multiple epochs, following these steps:

🔸 **Batch processing:**

Define batch training parameters as:

* **Epochs:** Total number of full passes through the training dataset.
* **Accuracy tracking variables:** Used to monitor misclassified and correctly classified samples per epoch.
* **Divide dataset into batches:** Using **DataLoader** create the train and validation loader.





🔸 **Training loop:**

* + - 1. **Forward pass →** z = model(x.view(-1,28\*28))

Input batch *x* is passed to the model to obtain the logits *z*.

⚠️ Each batch is reshaped using **.view(-1, 28\*28)** to flatten images into 1D vectors. Converts the **rectangle** tensor in the batch to a **row** tensor.

1. **Loss computation →** loss = criterion (z, y)

Cross-entropy loss is computed between model predictions and actual labels.

1. **Backward pass and Optimization**

optimizer.zero\_grad()**→** Compute gradients.

loss.backward()**→** Update Parameters.

optimizer.step()**→** Reset gradients before next iteration.

1. **Training phase completion**

After all the batches are processed, one epoch is completed. The model then is evaluated on the validation dataset with the learned parameters.

🔸 **Model evaluation loop:**

* + - 1. **Prediction**

z = model(x\_test.view(-1,28\*28)) **→** Outputs logits are computed for the validation batch.

yhat = torch.argmax(z.data, 1) **→** Class indexes are determined.

1. **Comparison with True Labels →** correct = correct + (yhat == y\_test).sum().item()

The predictions yhat are compared to the true labels y\_test. All correct classified labels are sum up, to obtain the number of correct predictions.

1. **Accuracy calculation →** accuracy = correct / n\_test

Model performance is obtained by dividing the correct predictions against total number of samples.

Accuracy value for the epoch is stored in a list to evaluate model performance in every epoch.

### ✅ Takeaways

✅ **Softmax as a Classifier:** Converts linear model outputs (logits) into normalized probability distributions across multiple classes, enabling multi-class prediction.

✅ **Cross-Entropy Integration:** Combines Softmax and log-likelihood into a single differentiable loss function that guides optimization effectively.

✅ **Representation Learning:** Each weight vector in the Softmax model learns to represent distinct class features, evolving from random noise to structured patterns resembling digits.

✅ **Gradient-Based Optimization:** Training with stochastic gradient descent adjusts model parameters by minimizing classification error through iterative backpropagation.

✅ **Probabilistic Interpretation:** Model outputs represent class probabilities, making predictions interpretable and measurable in terms of confidence.