Loss Functions in Deep Learning

💡 **Loss functions measure how wrong a model's predictions are.**

- They guide training by telling the optimizer which direction to adjust weights.

- Small value of loss function → Algorithm is performing great

- Loss Function in DL, works same as it does in ML.

Regression

1. MSE
2. MAE
3. Huber Loss

Classification

1. Binary cross entropy
2. Categorical cross entropy
3. Hinge Loss (SVM)

Autoencoders

- KL Divergence

GAN

1. Discriminator Loss
2. MinMax
3. GAN Loss

Object Detection

- Focal Loss

Embeddings

- Triplet Loss

Loss Function (Error Function) vs Cost Function

- LF is calculated on a single training example
  - It's like saying, "Oops! My prediction for *this* one example was off by *this much*." 📏
- CF is calculated on Entire training dataset (or a batch)
  - Typically an average of loss functions over the data
  - It's like saying, "On average, how bad are my predictions for *all* the examples I've seen?" 📊

📉 Common Loss Functions

Mean Squared Error (MSE)

- Also known as → **L2 Loss, Squared Loss**

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

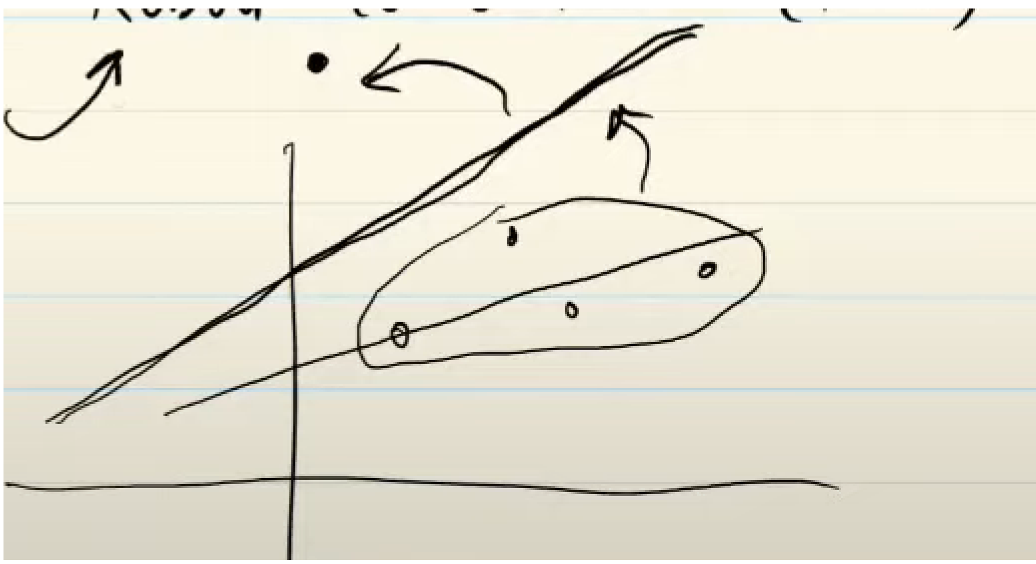
- **Use Case**: Regression (predicting continuous values like house prices).

Pros:

- Simple
- Differentiable

Cons:

- Unit is squared
- Sensitive to outliers.



💡 In DL, in order to apply MSE, the **activation function of the last neuron should be** → `Linear`

Mean Absolute Error (MAE)

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- Also known as → **L1 Loss**

- **Use Case**: **Regression** (robust to outliers).

Pros:

- Robust to outliers
- Same unit

Cons:

- Slower convergence (non-smooth at zero).
- Graph is **not differentiable**
  - You cannot apply gradient descent
  - You have to use sub-gradient

Huber Loss

$$L_\delta = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

δ is hyperparameter. You can adjust its value.

- **Use Case**: Regression (combines **MSE and MAE**).

- For small errors,
  - it behaves like MSE
- for larger errors (outliers)
  - it behaves like MAE.
- Ex. when you have 25% outliers, neither MAE nor MSE will perform well.
  - In such case, you use Huber Loss

Binary Cross-Entropy (Log Loss)

$$\mathrm{BCE} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

- **Use Case**: **Binary classification** (e.g., spam detection).
- It penalizes the model more when the prediction is far from the true class.

> 💡 **!!The activation function in output layer must be → Sigmoid**

Pros:

- Works well with probabilities.
- Differentiable

Cons:

- Multiple local minimas

- Unstable for extreme predictions (log(0) = -∞).

Categorical Cross-Entropy

$$\text{Categorical Cross-Entropy} = -\sum_{i=1}^{n} y_{\text{true}}^{(i)} \log(y_{\text{pred}}^{(i)})$$

- **Use Case**: Multi-class classification (e.g., MNIST digits).

> **No. of neurons in the output layer = No. of categories**

> 💡 **!!The activation function in output layer must be → Softmax**

- **Pros**: Optimized for probabilities.
- **Cons**: Requires **one-hot encoded labels**.

Sparse Categorical Cross-Entropy

- **Used for**: Multi-class classification problems with **integer labels** (instead of one-hot encoded vectors).
- When Output column is Numeric
  - **Example**: `y_true = [2, 0, 1]` (instead of one-hot).

$$\text{Sparse Categorical Cross-Entropy} = -\sum_{i=1}^{n} \log\left(p_{y_{\text{true}}^{(i)}}\right)$$

- **Explanation**: This is similar to categorical cross-entropy, but the target labels are integers (e.g., 0, 1, 2 for 3 classes) instead of one-hot encoded vectors. It is more memory-efficient when dealing with large datasets.
- **When to use**: In multi-class classification tasks where labels are integers, not one-hot encoded.