Comparison of Different Loss Functions

Loss Function	Formula	Derivative	When to Use	When to Avoid	Algorithms That Use It
Mean Squared Error (MSE)	$\frac{1}{N}\sum(y_i - \hat{y}_i)^2$	$\frac{dL}{d\hat{y}} = -\frac{2}{N}(y - \hat{y})$	Regression problems with normally distributed errors	When data has outliers (sensitive to large errors)	Linear Regression, Ridge Regression
Mean Absolute Error (MAE)	(\frac{1}{N} \sum	y_i - \hat{y}_i	)	$\frac{dL}{d\hat{y}} = \text{sign}(\hat{y} - y)/N$	Regression with outliers (robust loss)
Huber Loss	Quadratic for small errors, Linear for large errors	Piecewise function: Linear for small errors, quadratic for large errors	Regression with moderate outliers (balances MSE and MAE)	When outliers are very extreme or negligible	Robust Regression, Outlier-sensitive Models
Log Loss (Binary Cross-Entropy)	$-\frac{1}{N}\sum[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$	$\frac{dL}{d\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$	Binary classification (e.g., logistic regression, neural networks)	When classes are highly imbalanced	Logistic Regression, Neural Networks

Categorical Cross-Entropy	$-\frac{1}{N}\sum\sum y_{ij}\log(\hat{y}_{ij})$	$\frac{dL}{d\hat{y}} = -\sum \frac{y_{ij}}{\hat{y}_{ij}}$	Multi-class classification (e.g., softmax classifiers)	When labels are not one-hot encoded	Neural Networks (Softmax), Deep Learning
Hinge Loss	$\sum \max(0, 1 - y_i\hat{y}_i)$	$\frac{dL}{d\hat{y}} = -y_i$ if $y_i\hat{y}_i < 1$, else 0	Classification problems with SVMs (maximizing margin)	When probabilistic outputs are needed	Support Vector Machines (SVM)
KL Divergence	$\sum y_i \log \frac{y_i}{\hat{y}_i}$	$\frac{dL}{d\hat{y}} = -\sum \frac{y_i}{\hat{y}_i}$	When comparing probability distributions	When data distributions do not need comparison	Bayesian Models, Variational Inference

1. Mean Squared Error (MSE)

Formula

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

- Penalizes large errors quadratically, making it sensitive to outliers.

Derivative

$$\frac{dL}{d\hat{y}_i} = -\frac{2}{N}(y_i - \hat{y}_i)$$

- The gradient decreases as the error gets smaller.

Likelihood (Assuming Gaussian Noise)

$$p(y|X, w) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - \hat{y})^2}{2\sigma^2}\right)$$

- Assumes that errors follow a normal distribution.

Log-Likelihood

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

- Minimizing negative log-likelihood leads to MSE.

```python
import numpy as np

class MSELoss:
    """ Implements Mean Squared Error (MSE) Loss """

    def loss(self, y, y_pred):
        """ Compute the MSE loss

        Parameters:
        - y: Actual values (numpy array)
        - y_pred: Predicted values (numpy array)

        Returns:
        - Mean squared error
        """
        return np.mean((y - y_pred) ** 2)

    def gradient(self, y, y_pred):
        """ Compute the gradient (derivative) of MSE loss

        Parameters:
        - y: Actual values (numpy array)
        - y_pred: Predicted values (numpy array)

        Returns:
        - Gradient of MSE
        """
        return -2 * (y - y_pred) / len(y)

# Example Usage
y_true = np.array([3.0, -0.5, 2.0])
y_pred = np.array([2.5, 0.0, 2.0])

mse = MSELoss()
print("MSE Loss:", mse.loss(y_true, y_pred))
print("Gradient:", mse.gradient(y_true, y_pred))
```

2. Mean Absolute Error (MAE)

Formula

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

- Penalizes errors linearly instead of quadratically, making it robust to outliers.

Derivative

$$\frac{dL}{d\hat{y}_i} = \frac{1}{N} \text{sign}(\hat{y}_i - y_i)$$

- The gradient is constant and does not depend on the error size.

Likelihood (Assuming Laplace Noise)

$$p(y|X, w) = \frac{1}{2b} \exp\left(-\frac{|y - \hat{y}|}{b}\right)$$

- Assumes that errors follow a Laplace distribution.

Log-Likelihood

$$\log L = -\sum_{i=1}^{N} \frac{|y_i - \hat{y}_i|}{b}$$

- Minimizing negative log-likelihood leads to MAE.

Fully Commented Implementation

python

```python
class MAELoss:
    """ Implements Mean Absolute Error (MAE) Loss """

    def loss(self, y, y_pred):
        """ Compute the MAE loss

        Parameters:
        - y: Actual values (numpy array)
        - y_pred: Predicted values (numpy array)

        Returns:
        - Mean absolute error
        """
        return np.mean(np.abs(y - y_pred))

    def gradient(self, y, y_pred):
        """ Compute the gradient (derivative) of MAE loss

        Parameters:
        - y: Actual values (numpy array)
        - y_pred: Predicted values (numpy array)

        Returns:
        - Gradient of MAE
        """
        return np.sign(y_pred - y) / len(y)

# Example Usage
mae = MAELoss()
print("MAE Loss:", mae.loss(y_true, y_pred))
print("Gradient:", mae.gradient(y_true, y_pred))
```

3. Binary Cross-Entropy (Log Loss)

Formula

$$L(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

- Used for binary classification tasks.

Derivative

$$\frac{dL}{d\hat{y}_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)}$$

- Used in logistic regression training.

Likelihood (Bernoulli Distribution)

$$p(y|X, w) = \prod_{i=1}^{N}\hat{y}_i^{y_i}(1 - \hat{y}_i)^{(1-y_i)}$$

- Assumes binary labels with probabilities.

Log-Likelihood

$$\log L = \sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

- Minimizing negative log-likelihood leads to BCE.

Fully Commented Implementation

python

Copy Edit

```python
class BinaryCrossEntropyLoss:
    """ Implements Binary Cross-Entropy Loss """

    def loss(self, y, y_pred):
        """ Compute Binary Cross-Entropy loss

        Parameters:
        - y: Actual binary values (numpy array)
        - y_pred: Predicted probabilities (numpy array)

        Returns:
        - BCE loss
        """
        eps = 1e-9  # Small epsilon to prevent log(0)
        y_pred = np.clip(y_pred, eps, 1 - eps)  # Clip predictions to avoid log(0)
        return -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
```

```python
    def gradient(self, y, y_pred):
        """ Compute the gradient (derivative) of BCE loss

        Parameters:
        - y: Actual binary values (numpy array)
        - y_pred: Predicted probabilities (numpy array)

        Returns:
        - Gradient of BCE
        """
        eps = 1e-9  # Prevent division by zero
        y_pred = np.clip(y_pred, eps, 1 - eps)
        return (y_pred - y) / (y_pred * (1 - y_pred))

# Example Usage
y_true_class = np.array([1, 0, 1])
y_pred_class = np.array([0.9, 0.2, 0.8])

bce = BinaryCrossEntropyLoss()
print("BCE Loss:", bce.loss(y_true_class, y_pred_class))
print("Gradient:", bce.gradient(y_true_class, y_pred_class))
```

2. Categorical Cross-Entropy (CCE)

Formula

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} y_{ij} \log(\hat{y}_{ij})$$

- Used for **multi-class classification problems.**
- C is the number of classes.
- y_{ij} is **1 if the sample belongs to class j, otherwise 0** (one-hot encoding).
- $\hat{y}_{ij}$ is the **predicted probability for class j (from softmax output).**

Derivative

$$\frac{dL}{d\hat{y}_{ij}} = -\frac{y_{ij}}{\hat{y}_{ij}}$$

- The gradient is large for small probabilities **(forcing the model to adjust incorrect predictions quickly)**.

Likelihood (Multinomial Distribution)

$$p(y|X, W) = \prod_{i=1}^{N} \prod_{j=1}^{C} \hat{y}_{ij}^{y_{ij}}$$

- Each label follows a **categorical probability distribution.**

Log-Likelihood

$$\log L = \sum_{i=1}^{N} \sum_{j=1}^{C} y_{ij} \log(\hat{y}_{ij})$$

- **Minimizing negative log-likelihood** gives **CCE loss**.

Fully Commented Implementation of CCE

python

Copy Edit

```python
class CategoricalCrossEntropyLoss:
    """ Implements Categorical Cross-Entropy Loss """

    def loss(self, y, y_pred):
        """ Compute Categorical Cross-Entropy loss

        Parameters:
        - y: Actual one-hot encoded labels (numpy array)
        - y_pred: Predicted probabilities (numpy array)

        Returns:
        - Categorical Cross-Entropy loss
        """
        eps = 1e-9  # Small epsilon to prevent log(0)
        y_pred = np.clip(y_pred, eps, 1 - eps)  # Clip predictions
        return -np.mean(np.sum(y * np.log(y_pred), axis=1))

    def gradient(self, y, y_pred):
        """ Compute the gradient (derivative) of CCE loss

        Parameters:
        - y: Actual one-hot encoded labels (numpy array)
        - y_pred: Predicted probabilities (numpy array)

        Returns:
        - Gradient of CCE
        """
        eps = 1e-9  # Prevent division by zero
        y_pred = np.clip(y_pred, eps, 1 - eps)
        return -y / y_pred
```

```python
# Example Usage
y_true = np.array([
    [1, 0, 0],  # Class 0
    [0, 1, 0],  # Class 1
    [0, 0, 1]   # Class 2
])

y_pred = np.array([
    [0.7, 0.2, 0.1],  # Probabilities for class 0, 1, 2
    [0.1, 0.6, 0.3],
    [0.2, 0.3, 0.5]
])

cce = CategoricalCrossEntropyLoss()
print("CCE Loss:", cce.loss(y_true, y_pred))
print("Gradient:", cce.gradient(y_true, y_pred))
```

BCE vs. CCE – When to Use Which

Loss Function	Used For	Output Layer Activation	Typical Algorithms
Binary Cross-Entropy (BCE)	Binary classification (2 classes)	Sigmoid ($\sigma(z)$)	Logistic Regression, Binary Neural Networks
Categorical Cross-Entropy (CCE)	Multi-class classification (>2 classes)	Softmax ($\frac{e^{z_j}}{\sum_k e^{z_k}}$)	Deep Learning, Image Classification (e.g., CNNs)

- **Use BCE** if your problem is **binary classification**.
- **Use CCE** if your problem has **more than two classes** and labels are **one-hot encoded**.