Deep Learning

ANN

Contents

Introduction

1️⃣ Loss Function vs. Cost Function

📌 What are Loss Function and Cost Function?

- Loss Function: Measures how wrong the model's prediction is for a single data point.
- Cost Function: Measures how wrong the model's predictions are for the entire dataset (average of all losses).

📌 Example:

Imagine a student predicting their exam marks:

- Actual Marks: 80
- Predicted Marks: 70
- Loss: Difference between actual and predicted marks = 10
- Cost: If we check this loss for all students and take the average loss, we get the cost function.

📌 What is an Optimizer?

An optimizer adjusts the weights of the model to minimize the cost function and improve accuracy. It helps in faster learning and better performance.

📌 Types of Optimizers

🔹 1. Gradient Descent (GD)

- Moves in the direction of the lowest cost.
- Works slowly if the dataset is large.

- 🔹 2. Stochastic Gradient Descent (SGD)
- Updates weights after each training example.
- Faster than GD but fluctuates more.
- 🔹 3. Mini-Batch Gradient Descent
- Updates weights after a small batch instead of the whole dataset.
- Balances between GD & SGD.

4. Adam (Adaptive Moment Estimation) ✅ Most Used

- Adjusts learning rate automatically.
- Works well for most problems.

🔹 5. RMSprop (Root Mean Square Propagation)

- Good for recurrent neural networks (RNNs).
- Adjusts learning rates for each parameter.

Examples

📌 Example of Optimizers in Action

Imagine you are on a hill and want to reach the lowest point (minimum cost):

- GD: Walks straight but slowly.
- SGD: Runs in different directions, sometimes overshooting.
- Adam: Adjusts step size smartly and reaches the lowest point faster

💡 Final Thoughts:

- Loss Function → Measures error for one data point.
- Cost Function → Measures error for entire dataset.
- Optimizers → Help to reduce loss and improve model performance.

ANN

Regression

MAE

MSE

RMSE

HUBER LOSS

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

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

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

$$R^2 = 1 - \frac{\sum (y_i - \hat{y})^2}{\sum (y_i - \bar{y})^2}$$

Where,

$\hat{y}$ $- predicted\ value\ of\ y$

$\bar{y}$ $- mean\ value\ of\ y$

📌 Mean Absolute Error (MAE)

- **Measures the average absolute difference** between actual and predicted values.
- **Formula:**

$$MAE = \frac{1}{n} \sum |y_i - \hat{y}_i|$$

- y_i = actual value
- $\hat{y}_i$ = predicted value
- n = number of samples

✅ **Pros:** Simple, interpretable, and robust to outliers.

❌ **Cons:** Doesn't penalize large errors heavily.

🔷 **Example:** If actual values = [10, 12, 14] and predictions = [9, 13, 15], then

$$MAE = \frac{|10 - 9| + |12 - 13| + |14 - 15|}{3} = \frac{1 + 1 + 1}{3} = 1$$

📌 Mean Squared Error (MSE)

- Penalizes large errors more than small ones because it squares the differences.
- **Formula:**

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

✅ **Pros:** Amplifies large errors, making it useful when you want to reduce large deviations.

❌ **Cons:** Sensitive to outliers.

🔷 **Example:**

$$MSE = \frac{(10-9)^2 + (12-13)^2 + (14-15)^2}{3} = \frac{1+1+1}{3} = 1$$

3️⃣ Root Mean Squared Error (RMSE)

Formula:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n} \sum (y_{\text{actual}} - y_{\text{predicted}})^2}$$

✅ Advantages:

- More **interpretable** than MSE (error in original units).
- Penalizes large errors like MSE but in the original scale.

❌ Disadvantages:

- **Computationally expensive** due to the square root operation.
- Sensitive to outliers.

📌 Loss vs. Cost Function:

- **Loss Function** → Square root of squared error for **one** data point.
- **Cost Function** → Average RMSE across **all** data points.

4 Huber Loss

Used when data has **outliers** (combines MSE & MAE benefits).

Formula:

$$L = \begin{cases} \frac{1}{2}(y_{\text{actual}} - y_{\text{predicted}})^2, & \text{if } |y_{\text{actual}} - y_{\text{predicted}}| \leq \delta \\ \delta(|y_{\text{actual}} - y_{\text{predicted}}| - \frac{1}{2}\delta), & \text{if } |y_{\text{actual}} - y_{\text{predicted}}| > \delta \end{cases}$$

✅ **Advantages:**

- **Less sensitive to outliers** than MSE.
- Works well when both small and large errors need balanced handling.

❌ **Disadvantages:**

- Requires **tuning** the hyperparameter δ.

📌 **Loss vs. Cost Function:**

- **Loss Function** → Switches between MSE & MAE based on δ for **one** data point.
- **Cost Function** → Average Huber Loss across **all** data points.

Loss Function	Best For	Handles Outliers?	Interpretability	Computational Cost
MAE	Simple models, Robust data	✅ Yes	✅ Easy to interpret	🔷 Slow convergence
MSE	Normally distributed errors	❌ No	❌ Squared errors (not original scale)	🔷 High
RMSE	Large errors impact performance	❌ No	✅ More interpretable than MSE	🔷 Very High
Huber Loss	Datasets with outliers	✅ Yes	✅ Balances MAE & MSE	🔷 Requires tuning δ

📌 Loss Functions for Classification in ANN

Classification problems in **Artificial Neural Networks (ANN)** use **entropy-based loss functions**, alsc called **Log Loss (Logarithmic Loss)**. These functions measure how well a model predicts **categorical** labels (classes).

---

1 Binary Cross-Entropy (Log Loss for 2 Classes)

Used when the target variable has **two** classes (e.g., 0 & 1, "Spam" or "Not Spam").

🔷 Formula:

$$Loss = -\frac{1}{n}\sum\left(y_{\text{actual}} \cdot \log(y_{\text{predicted}}) + (1 - y_{\text{actual}}) \cdot \log(1 - y_{\text{predicted}})\right)$$

where:

- y_{actual} → True label (0 or 1)
- $y_{\text{predicted}}$ → Predicted probability (between 0 and 1)
- n → Number of data points

🔷 Example:

Let's say we are predicting whether an email is spam (1) or not spam (0).

Email	Actual Label (y_{actual})	Predicted Probability ($y_{\text{predicted}}$)	Log Loss
Email 1	1 (Spam)	0.9	-log(0.9) = **0.105**
Email 2	0 (Not Spam)	0.8	-log(1-0.8) = **0.223**

📌 Key Points:

✅ Works well for **binary** classification.

❌ Penalizes incorrect predictions **more** when the model is confident but wrong.

2️⃣ Categorical Cross-Entropy (Log Loss for Multi-Class)

Used when the target variable has **more than two** classes (e.g., Dog, Cat, Horse).

🔷 Formula:

$$Loss = -\frac{1}{n}\sum\sum y_{\text{actual},i}\log(y_{\text{predicted},i})$$

where:

- $y_{\text{actual},i}$ → 1 if class i is the true class, otherwise 0
- $y_{\text{predicted},i}$ → Probability of class i (from Softmax output)

🔷 Example:

Let's say we have a **3-class problem** (Dog, Cat, Horse).

Animal	Actual Label	Predicted Probabilities (Dog, Cat, Horse)	Log Loss
🐶 Dog	(1, 0, 0)	(0.8, 0.1, 0.1)	-log(0.8) = **0.223**
🐱 Cat	(0, 1, 0)	(0.2, 0.7, 0.1)	-log(0.7) = **0.357**
🐴 Horse	(0, 0, 1)	(0.1, 0.2, 0.7)	-log(0.7) = **0.357**

💡 Summary (Easy to Remember)

Loss Function	Used For	Works With	How it Works?	Disadvantages
Binary Cross-Entropy	**Binary Classification** (Yes/No, Spam/Not Spam)	**Sigmoid Activation**	Compares predicted probability to actual label (0 or 1)	❌ Sensitive to extreme confidence errors
Categorical Cross-Entropy	**Multi-Class Classification** (Dog/Cat/Horse)	**Softmax Activation**	Compares predicted probabilities for multiple classes	❌ Wrong confident predictions get heavily penalized

📌 Sparse Categorical Cross-Entropy

🔷 What is Sparse Categorical Cross-Entropy?

- Used for **multi-class classification** problems where labels are **integers** instead of one-hot encoded vectors.
- **Best when dealing with large categories** (e.g., classifying 1000+ objects).

🔷 Formula

$$Loss = -\sum y \log(\hat{y})$$

Where:

- y = actual class label (integer form)
- $\hat{y}$ = predicted probability

🔷 Example

If we have three classes: **Dog (0), Cat (1), and Bird (2)**

- True label = `1` (Cat)
- Model predicts probabilities: **[0.2, 0.7, 0.1]**
- Loss = $-\log(0.7)$

Batch Gradient Descent (BGD)

- Uses **all data points** to compute the gradient before updating weights.
- **Stable but slow** for large datasets.

📌 **Formula:**

$$W = W - \eta \cdot \frac{1}{N} \sum \nabla L(W, x, y)$$

Where:

- W = weights
- η = learning rate
- N = total dataset size
- ∇L = gradient of the loss function

✅ **Pros:** More stable updates, guaranteed convergence.
❌ **Cons: Very slow** for large datasets.

Mini-Batch Gradient Descent (MBGD) – Best of Both! 🚀

- **Uses small batches** of data (e.g., batch size = 32, 64, 128).
- Balances between **SGD (fast but noisy)** and **BGD (stable but slow)**.
- **Most commonly used in Deep Learning!**

📌 **Formula:**

$$W = W - \eta \cdot \frac{1}{m} \sum \nabla L(W, x, y)$$

Where:

- m = mini-batch size (e.g., 32, 64).

✅ **Pros:** Efficient, stable, works well for large datasets.

❌ **Cons:** Needs **tuning** of batch size.

1️⃣ SGD (Stochastic Gradient Descent)

📌 **How it works:**

- Adjusts weights step by step using **gradient descent**.
- Uses **random small batches** of data (stochastic = random).
- **Good for small datasets** but **slow and noisy** for deep networks.

📌 **Formula:**

$$W = W - \eta \cdot \nabla L$$

Where:

- W = weights
- η = learning rate (step size)
- ∇L = gradient of the loss function

✅ **Best for:** Simple models, small datasets

❌ **Downside:** Can get stuck in local minima, slow

2️⃣ Momentum SGD 🚀 (Faster SGD)

📌 **How it works:**

- Similar to **SGD**, but **adds momentum** (like rolling a ball down a hill).
- Helps move **faster in the right direction** and avoids getting stuck.

📌 **Formula:**

$$v_t = \gamma v_{t-1} + \eta \nabla L$$

$$W = W - v_t$$

Where:

- v_t = momentum term (helps smooth updates)
- γ = momentum factor (e.g., 0.9)

✅ **Best for:** Faster convergence in deep networks

❌ **Downside**: Needs tuning of **momentum factor**

3️⃣ RMSprop (Root Mean Square Propagation)

📌 How it works:

- Adjusts the learning rate **for each weight individually**.
- **Reduces big jumps** by dividing by the moving average of past gradients.
- Great for **RNNs and deep networks**.

📌 Formula:

$$W = W - \frac{\eta}{\sqrt{v_t + \epsilon}} g_t$$

Where:

- v_t = moving average of past gradients
- ϵ = small number to prevent division by zero

✅ **Best for:** RNNs, speech recognition

❌ **Downside:** May not generalize well for all tasks

4️⃣ Adagrad (Adaptive Gradient Descent)

📌 **How it works:**

- **Adapts learning rates** for each weight.
- Works well for **sparse data (NLP, recommendation systems)**.
- But the learning rate **keeps decreasing**, making it slow over time.

📌 **Formula:**

$$W = W - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

Where:

- G_t = sum of squared gradients
- η = learning rate

✅ **Best for**: NLP, sparse data problems

❌ **Downside:** Learning rate becomes too small over time

5️⃣ Adam (Adaptive Moment Estimation) – Most Used 🔥

📌 **How it works:**

- **Combines Momentum and RMSprop** for the best results.
- Adjusts the learning rate **for each weight** based on past gradients.
- Works well for **most deep learning problems**.

📌 **Formula:**

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$W = W - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t$$

Where:

- m_t = moving average of gradients (Momentum)
- v_t = moving average of squared gradients (RMSprop)

✅ **Best for: Most deep learning models** (CNNs, RNNs, Transformers)
❌ **Downside:** Uses more memory

🔷 Summary: Which Optimizer to Use?

Optimizer	Best For	Good Choice?
SGD	Small datasets, simple models	❌ (Too slow for deep learning)
Momentum SGD	Faster convergence	✅ (Better than SGD)
RMSprop	RNNs, speech recognition	✅ (Works well for sequential data)
Adagrad	Sparse data, NLP	⚠️ (Good, but slows over time)
Adam	Almost everything	🔥 **Best overall choice**