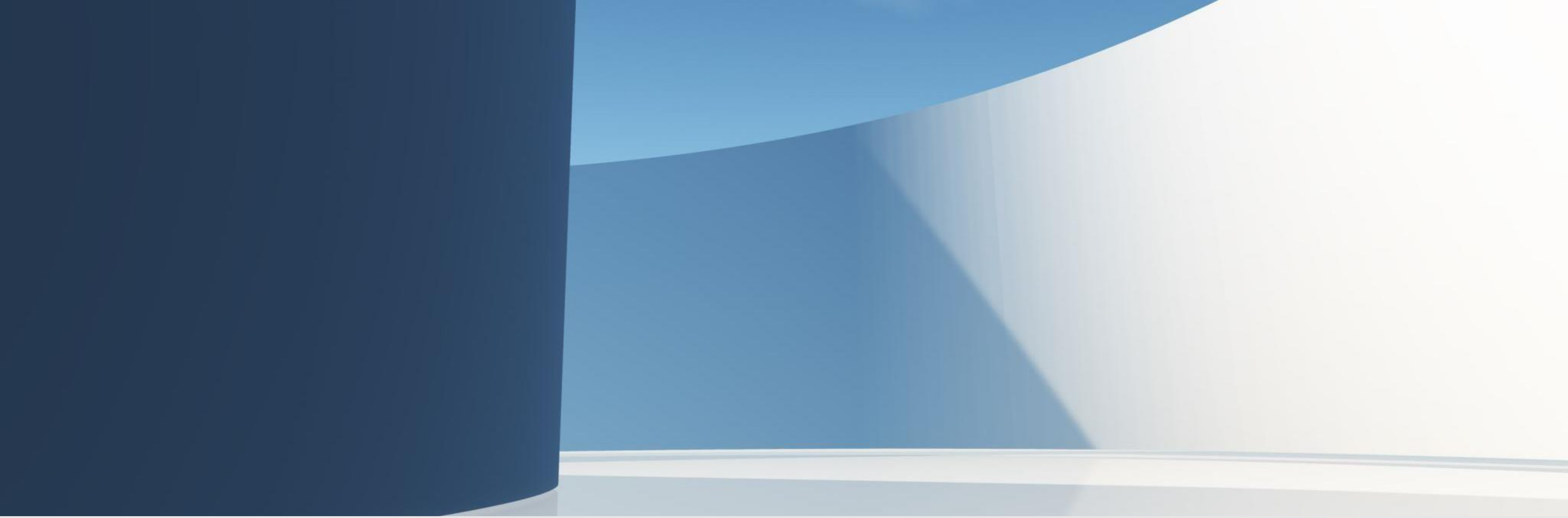

Optimizers, Cost Functions

Loss/ Cost functions

- Cost functions (or loss functions) in deep learning measure the error between the model's predictions and the actual target values.
- The choice of cost function depends on the task (classification, regression, etc.).
- **1. Mean Squared Error (MSE)**

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

  - **How it works**: Measures the average of the squares of the errors between predicted (y^i) and actual values (Yi).
  - **When to use**: Best suited for **regression tasks** where the output is continuous, like predicting house prices or stock prices. MSE penalizes larger errors more, leading to smoother predictions.

- **2. Mean Absolute Error (MAE)**

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

  - **How it works**: Calculates the average of the absolute differences between the predicted and actual values.
  - **When to use**: Also used for **regression**, especially when outliers are present. MAE treats all errors equally, so it's more robust to outliers than MSE.

- **3. Binary Cross-Entropy (Log Loss)**

$$\text{Loss} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

  - **How it works**: Measures the difference between the actual label yi∈{0,1} and the predicted probability y^i for **binary classification** tasks.
  - **When to use**: For **binary classification problems**, such as spam detection, fraud detection, or any task where there are two possible outcomes (0 or 1).

- 4. **Categorical Cross-Entropy**

$$\text{Loss} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

  - **How it works**: Similar to binary cross-entropy but for multi-class classification. Here, yi is a one-hot encoded true label vector, and y^i is the predicted probability for each class.
  - **When to use**: For **multi-class classification problems** where there is one correct class out of many, such as image classification or text classification tasks (e.g., predicting if an image is a cat, dog, or bird).

- **5. Sparse Categorical Cross-Entropy**
  - **How it works**: Similar to categorical cross-entropy, but instead of one-hot encoding the true labels, the true labels are integers representing the class index.
  - **When to use**: When your target labels are integers instead of one-hot encoded vectors. Useful for **multi-class classification** problems with a large number of classes, as it saves memory and computation.

- **6. Kullback-Leibler Divergence (KL Divergence)**

Formula:

$$D_{\text{KL}}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

- **How it works**: Measures how one probability distribution P diverges from a second distribution Q.
- **When to use**: Often used in **unsupervised learning**, particularly in models like variational autoencoders (VAE) and when comparing distributions, e.g., reinforcement learning where you compare the policy of the agent.

- **7. Hinge Loss**

$$\text{Loss} = \max(0, 1 - y_i \cdot \hat{y}_i)$$

  - **How it works**: Used for "maximum-margin" classification like support vector machines (SVMs). Penalizes incorrect classifications and correct classifications that are not confidently correct.
  - **When to use**: Best for **binary classification** tasks where margin maximization is important, often used in tasks requiring **SVMs** or in deep learning applications with SVM-like behavior.

Summary of Cost Functions and Their Use-Cases

- **MSE/MAE/Huber**: For **regression** tasks (continuous output).
- **Binary Cross-Entropy**: For **binary classification** (two possible outcomes).
- **Categorical Cross-Entropy/Sparse Categorical Cross-Entropy**: For **multi-class classification**.
- **KL Divergence**: For comparing **probability distributions**, often in **unsupervised learning** or **reinforcement learning**.
- **Hinge Loss**: For **binary classification** using SVM-like models.
- **Poisson Loss**: For **count-based regression** tasks (such as number of customers, number of emails received in a day).

Optimizers

- In deep learning, optimizers are algorithms used to adjust the weights of neural networks to minimize the loss function.
- common optimizers and when to use them:
  - **Stochastic Gradient Descent (SGD)**
    - Basic optimization, good for large datasets.
    - **When to use**: Works well when data is plentiful and simple.
    - Often used in large-scale applications like image recognition.
    - Slow convergence but leads to good generalization.
  - **SGD with Momentum**
    - For faster, smoother convergence.
    - **When to use:** Use when the optimization is slow or gets stuck in local minima. It speeds up convergence, especially on complex tasks.

- **RMSprop (Root Mean Square Propagation):**
  - **When to use**: Great for mini-batch training and noisy data (e.g., recurrent neural networks)
  - Helps with faster convergence.

- **Adam (Adaptive Moment Estimation):** General-purpose, widely effective across tasks.
  - The most popular optimizer for most deep learning models.
  - **When to use :**It generally performs well across various tasks, like natural language processing and computer vision, due to fast convergence.

- **Adagrad**: Best for sparse data and features.
  - **When to use**: Useful for sparse data and when features are very different in scale, like in natural language processing.
  - However, learning rates can get too small, slowing down training.

- **Adadelta**: Improved Adagrad, useful for preventing slowdowns.
- **Nadam**: When you need fast convergence with high accuracy.

Appendix

- **1. Stochastic Gradient Descent (SGD)**
  - **How it works**: Updates weights by calculating the gradient of the loss function with respect to the model parameters.
  - **When to use**: Works well when data is plentiful and simple. Often used in large-scale applications like image recognition. Slow convergence but leads to good generalization.

- **2. SGD with Momentum**
  - **How it works**: Accelerates the gradient vectors in the right direction by adding a fraction of the previous update to the current update.
  - **When to use:** Use when the optimization is slow or gets stuck in local minima. It speeds up convergence, especially on complex tasks.

- **3. RMSprop (Root Mean Square Propagation)**
  - **How it works**: Adapts the learning rate for each parameter by maintaining a moving average of the squared gradient.
  - **When to use**: Effective for problems with noisy gradients (e.g., recurrent neural networks) or for training on mini-batches. Helps with faster convergence.
- **4. Adam (Adaptive Moment Estimation)**
  - **How it works**: Combines the benefits of both Momentum and RMSprop by computing adaptive learning rates for each parameter and using moving averages of both the gradients and their squares.
  - **When to use**: The most popular optimizer for most deep learning models. It generally performs well across various tasks, like natural language processing and computer vision, due to fast convergence.
- **5. Adagrad**
  - **How it works**: Adjusts the learning rate for each parameter individually, scaling it inversely with the sum of the squares of the past gradients.
  - **When to use**: Useful for sparse data and when features are very different in scale, like in natural language processing. However, learning rates can get too small, slowing down training.

- **6. Adadelta**
  - **How it works**: A refinement of Adagrad that limits the learning rate from shrinking too much by focusing on a window of past updates.
  - **When to use**: Suitable for cases where Adagrad would cause the learning rate to decrease too much, like in NLP tasks.
- **7. Nadam (Nesterov-accelerated Adam)**
  - **How it works**: An extension of Adam that incorporates Nesterov momentum, offering faster convergence.
  - **When to use**: Good for models that require fast convergence and high accuracy. Often used in computer vision and language models.