BASIC LOSS FUNCTIONS

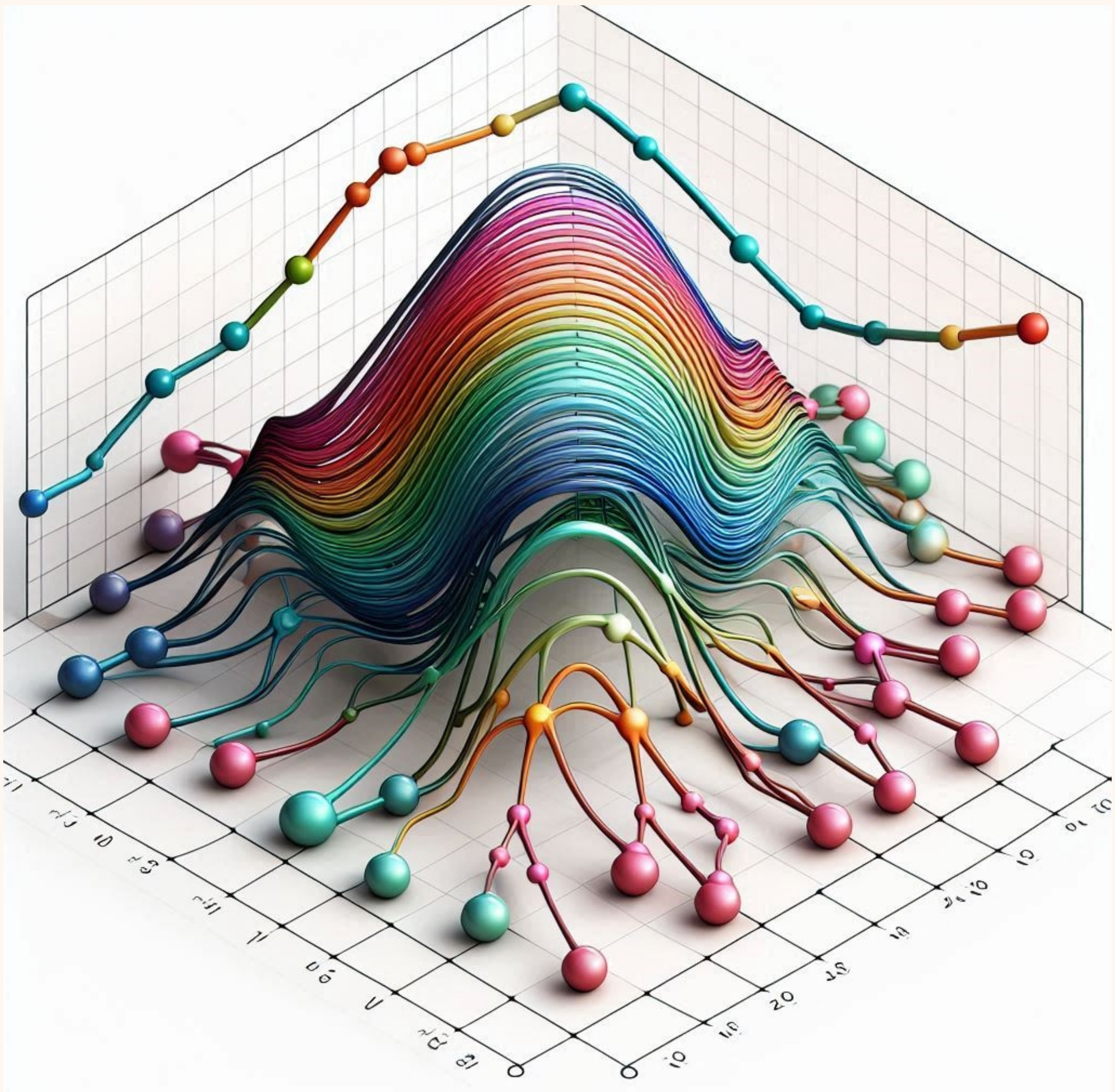
THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

https://arihara-sudhan.github.io

LOSS FUNCTION

Loss function serves as a measure of how well a model is performing. It quantifies the difference between the predicted values and the target values. We have learned of it a little bit in MLP book. The objective of our training is to reduce the loss. We also learned of optimization algorithms such as Gradient Descent, SGD, Mini Batch Gradient Descent, SGD with Momentum, AdaGrad, RMSProp and Adam. There are even more optimization algorithms focused to reduce loss. Obviously, loss function is the feedback-giver for the network by means of which the parameters are tuned. Remember, loss function is not an evaluation metric. Another term used is Cost Function. The loss function is to capture the difference between the actual and predicted values for a single datum whereas cost functions aggregate the difference for the entire training dataset.



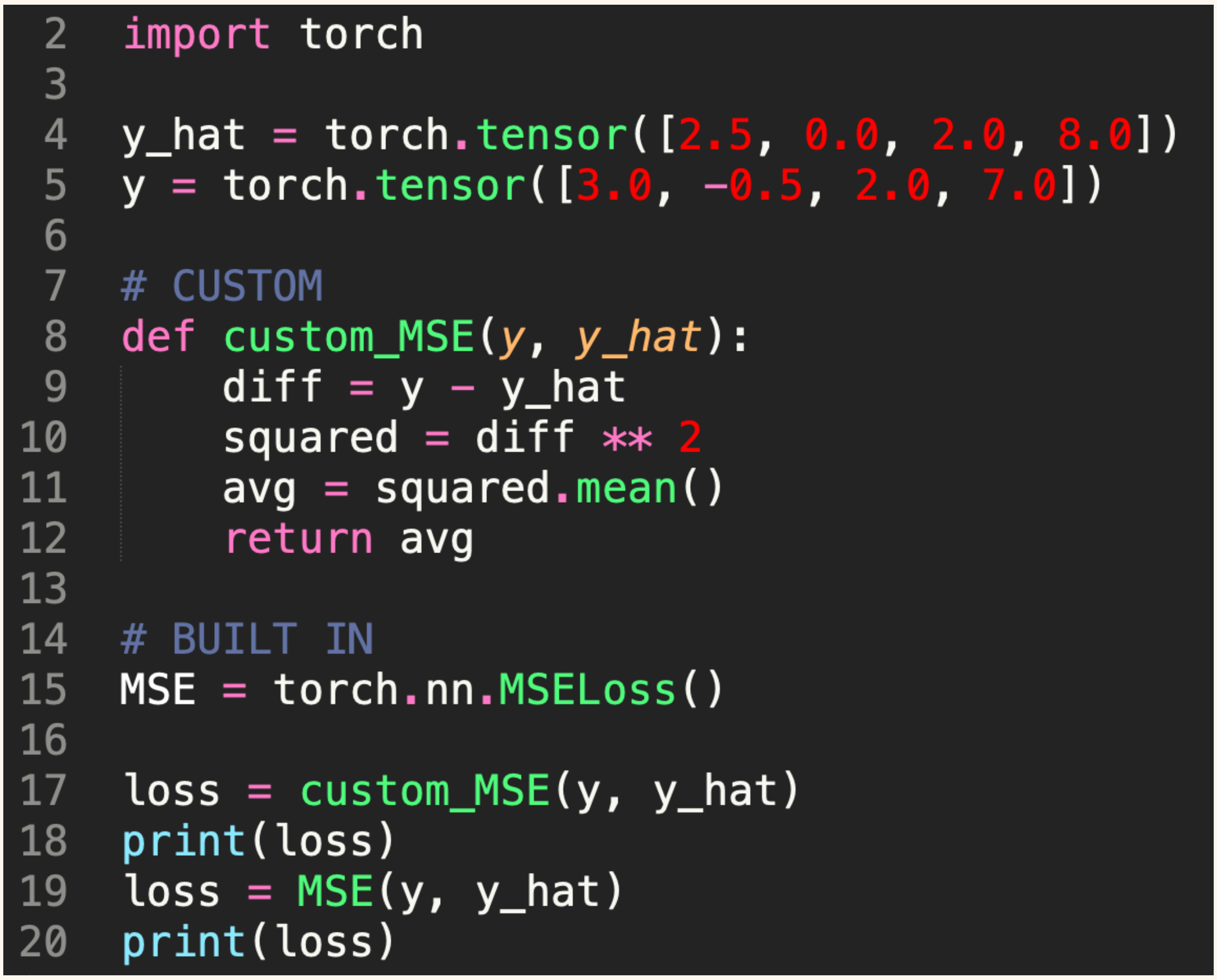
☆ MEAN SQUARED ERROR

Mean Squared Error is the one which calculates the average of the squared differences between predicted and actual values.

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

In the last MLP topic, we had a short introduction to error, on reducing which the model performs well on given data. It is okay to have (y – y_hat). But, why do we have it squared? Why do we have it average-summed? Squaring the differences between actual and predicted values makes larger errors stand out more. For instance, an error of 4 becomes 16 after squaring, while an error of 1 remains 1. This helps the model focus on reducing large errors, which are often more impactful on overall performance. Without squaring, positive and negative errors would cancel each other out. Squaring makes all errors positive, so we get a more accurate representation of total error regardless of direction (underestimation or overestimation). Averaging divides the total error by the number of data points n, resulting in a mean value that doesn't change simply because there are more (or fewer) data points. This makes the error measure comparable across datasets of any size.

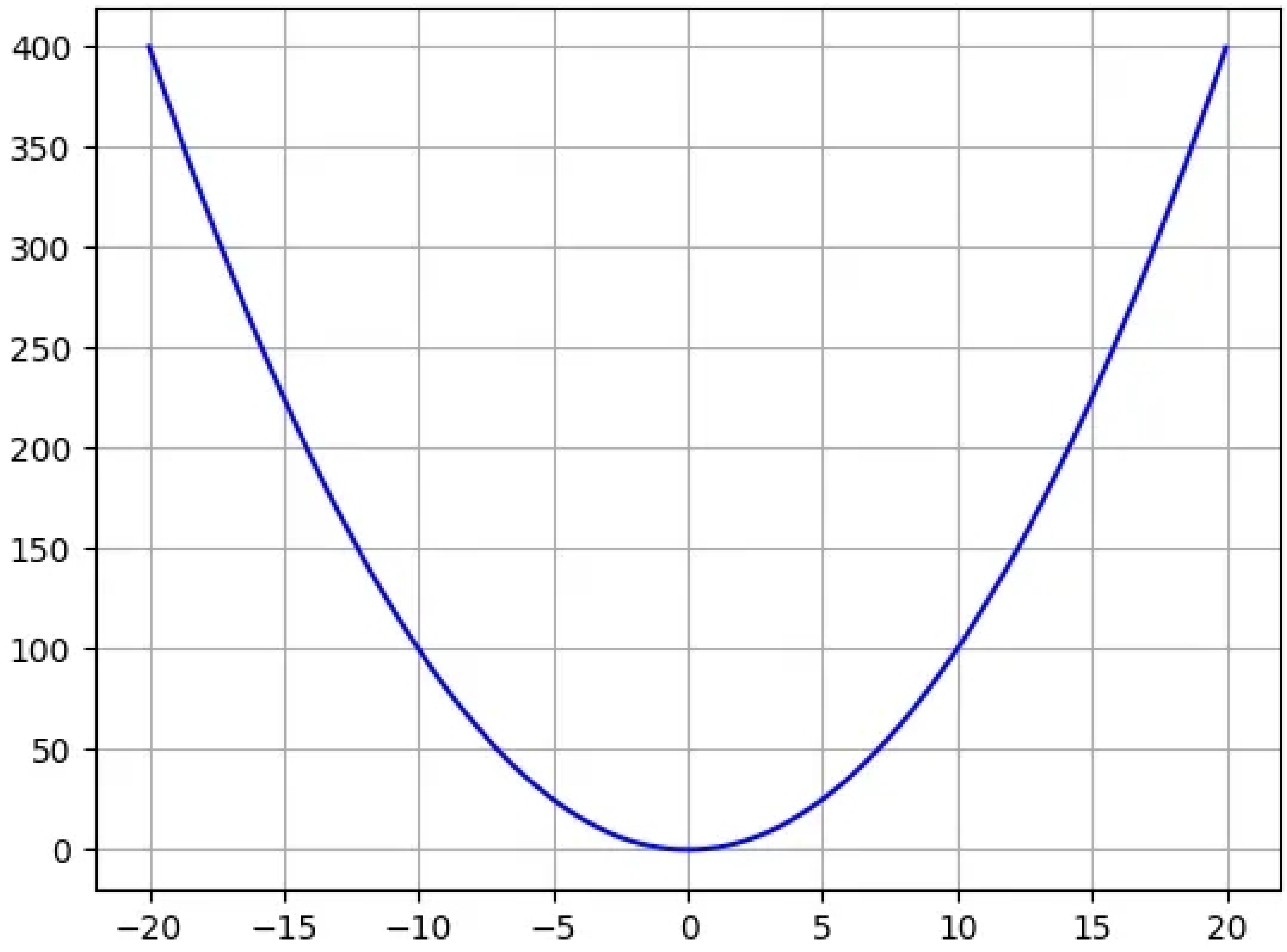
We can either implement by ourself or else use the built-in one.

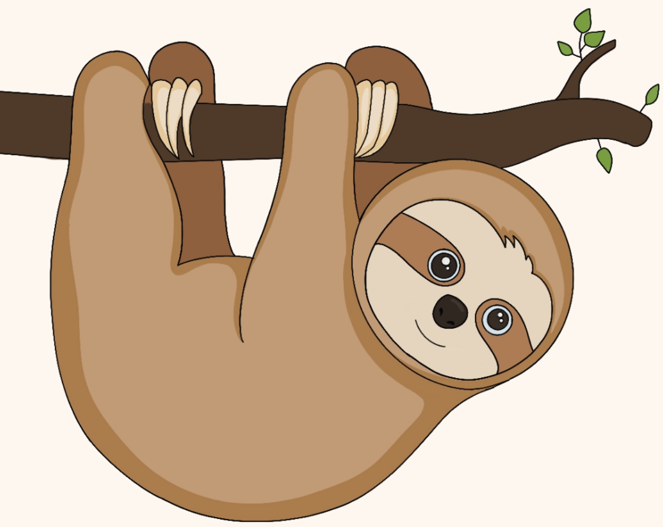


```python
import torch

y_hat = torch.tensor([2.5, 0.0, 2.0, 8.0])
y = torch.tensor([3.0, -0.5, 2.0, 7.0])

# CUSTOM
def custom_MSE(y, y_hat):
    diff = y - y_hat
    squared = diff ** 2
    avg = squared.mean()
    return avg

# BUILT IN
MSE = torch.nn.MSELoss()

loss = custom_MSE(y, y_hat)
print(loss)
loss = MSE(y, y_hat)
print(loss)
```

MSE is a convex function, meaning that it has a single global minimum and no local minima. This property makes it easier to optimize using gradient-based optimization techniques like gradient descent. MSE is differentiable everywhere, meaning its gradient (derivative) can be computed at every point. This is crucial for gradient-based optimization algorithms (such as stochastic gradient descent). MSE squares the error for each data point, which means that larger errors (outliers) have a disproportionately large impact on the loss. A single large error can significantly increase the MSE value. Thus, it becomes sensitive to outliers.

Following is the curve of MSE Loss function.



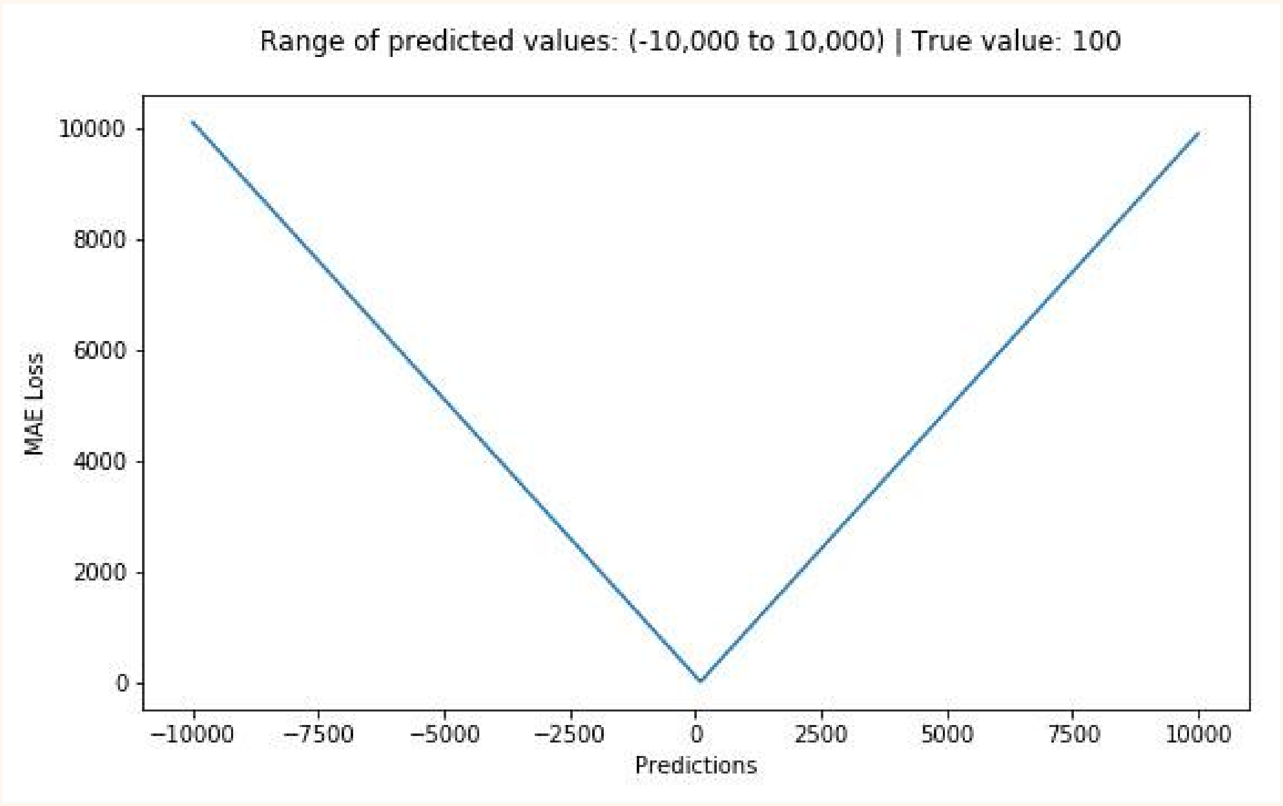


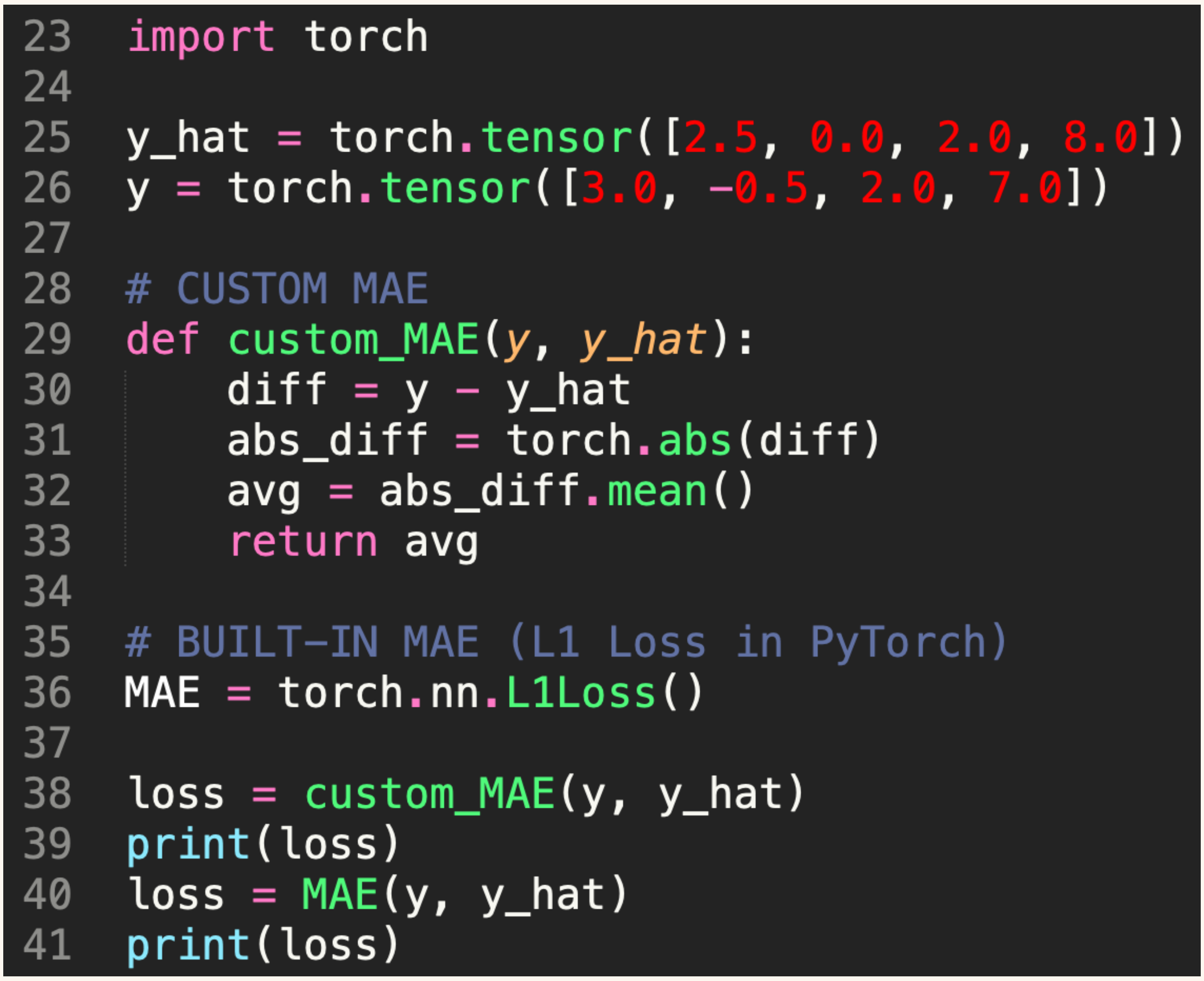
☆ MEAN ABSOLUTE ERROR

Mean Absolute Error is the one which calculates the absolute value of the average differences between predicted and actual values.

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

It gives linear penalty. A **linear penalty** means that the error term grows in direct proportion to the deviation between the predicted and actual values. This is because the MAE calculates the **absolute** difference between each predicted value and the actual value, rather than squaring the difference as in MSE. We can also say, each error contributes to the overall loss directly as it is, no matter how large or small the error, it's added linearly without amplification. If the error (difference between the predicted and actual values) is 3, then it contributes exactly 3 units to the loss. If the error is 6, it contributes 6 units to the loss. It helps make MAE less sensitive to large errors or outliers. When we say it is a Linear Penalty, the one in MSE is Quadratic Penalty. There in MSE, if the error is 6, the penalty will be like 6*6 = 36. The graph of MAE versus error is symmetrical and forms a V-shape around zero, where the minimum error is achieved when the error is zero. The problem with MAE is that, it is not differentiable some times.





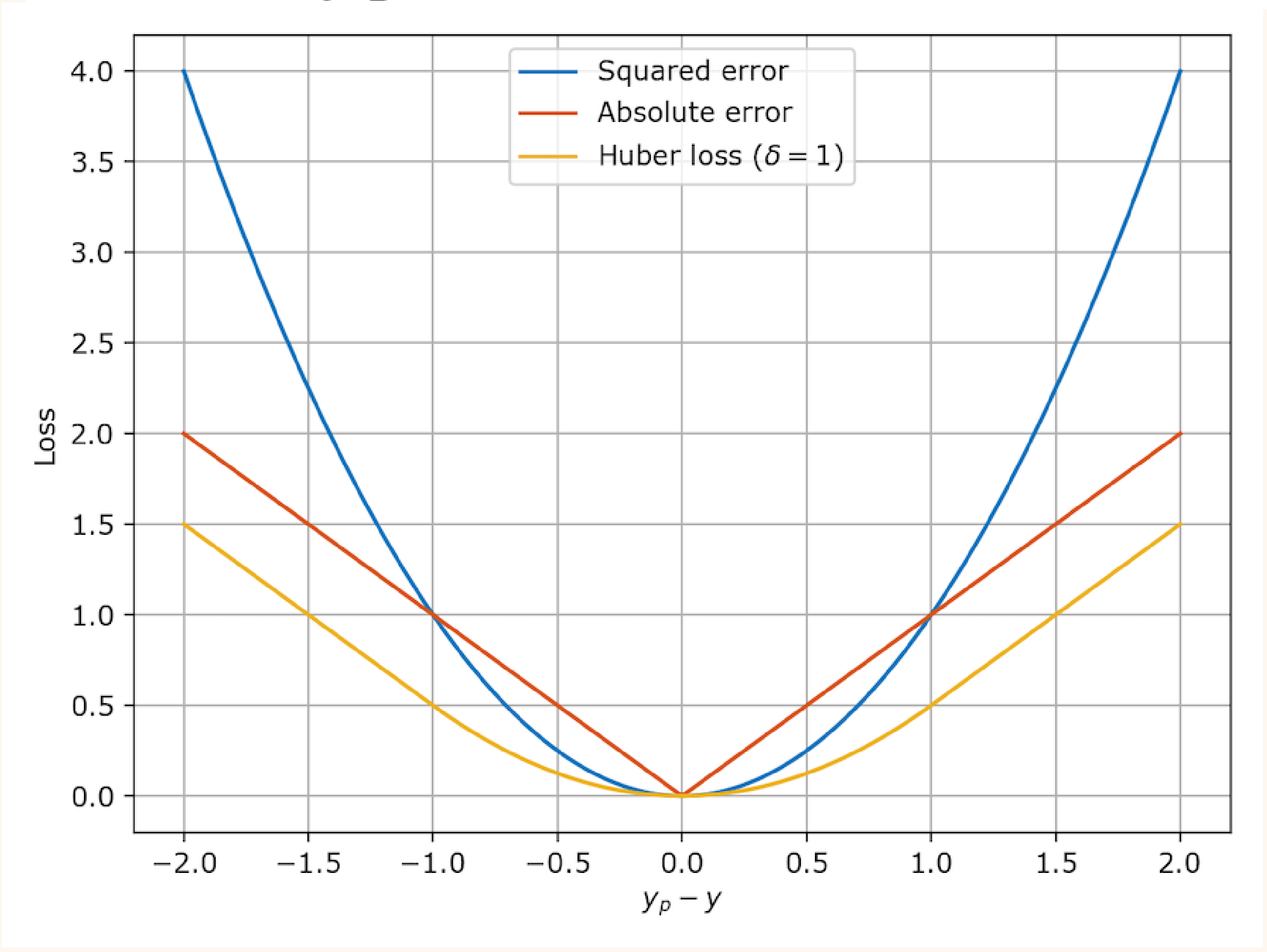
```python
import torch

y_hat = torch.tensor([2.5, 0.0, 2.0, 8.0])
y = torch.tensor([3.0, -0.5, 2.0, 7.0])

# CUSTOM MAE
def custom_MAE(y, y_hat):
    diff = y - y_hat
    abs_diff = torch.abs(diff)
    avg = abs_diff.mean()
    return avg

# BUILT-IN MAE (L1 Loss in PyTorch)
MAE = torch.nn.L1Loss()

loss = custom_MAE(y, y_hat)
print(loss)
loss = MAE(y, y_hat)
print(loss)
```

☆ HUBER LOSS

Huber Error combines the essence of both Mean Squared Error and Mean Absoluter Error. It is linear for large errors and quadratic for small errors. It is also differentiable. It has parameter delta to switch between MSE and MAE.

$$Huber = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(y_i - \hat{y}_i)^2 \qquad |y_i - \hat{y}_i| \le \delta$$

$$Huber = \frac{1}{n}\sum_{i=1}^{n}\delta\left(|y_i - \hat{y}_i| - \frac{1}{2}\delta\right) \qquad |y_i - \hat{y}_i| > \delta$$



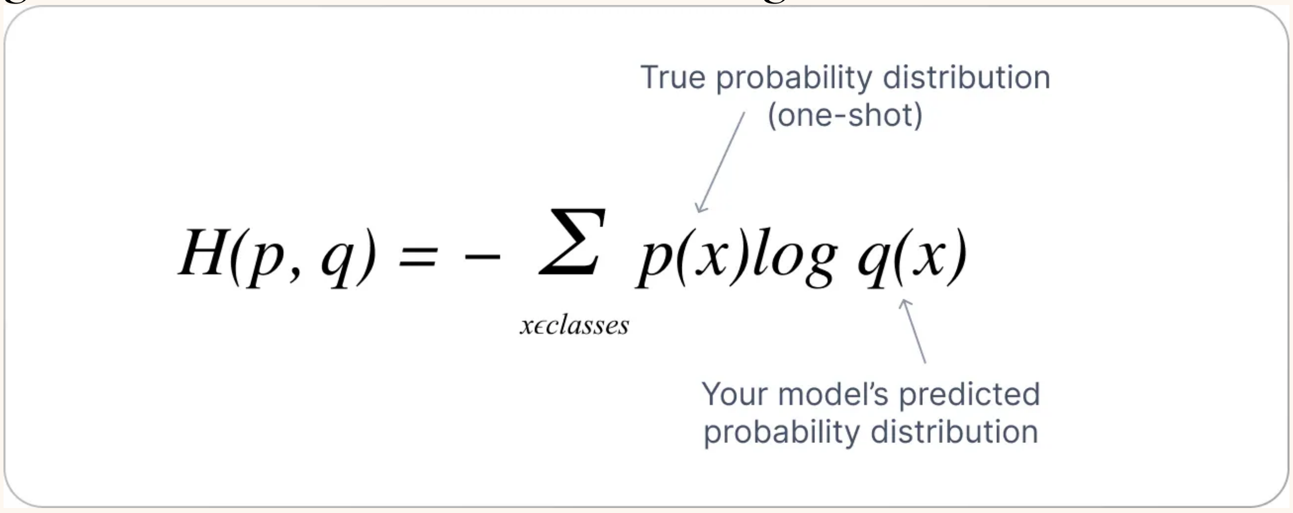
Huber Loss is smooth, as there is no abrupt transition between the quadratic and linear regions.

The smoothness property allows for more stable training compared to non-smooth loss functions like MAE. Huber Loss is convex, meaning that it has a single global minimum. This property ensures that optimization algorithms like gradient descent will converge to the optimal solution.

```python
import torch

y_hat = torch.tensor([2.5, 0.0, 2.0, 8.0])
y = torch.tensor([3.0, -0.5, 2.0, 7.0])

# CUSTOM HUBER LOSS
def custom_huber_loss(y, y_hat, delta=1.0):
    diff = y - y_hat
    abs_diff = torch.abs(diff)

    loss = torch.where(abs_diff < delta,
                       0.5 * (diff ** 2), #MSE
                       delta * (abs_diff - 0.5 * delta)) #MAE

    avg_loss = loss.mean()
    return avg_loss

# BUILT-IN HUBER LOSS
huber_loss_fn = torch.nn.SmoothL1Loss()

loss = custom_huber_loss(y, y_hat, delta=1.0)
print("Custom Huber Loss:", loss)
loss_builtin = huber_loss_fn(y_hat, y)
print("Built-in Huber Loss:", loss_builtin)
```

☆ CROSS ENTROPY LOSS

Cross-Entropy Loss measures how well the predicted probability distribution matches the true distribution. When the predicted probability for the true class is high, the loss is low. If the predicted probability is low, the loss is high, indicating that the model's prediction is far from the true label. Cross-Entropy Loss is non-linear but convex when dealing with one-hot encoded target distributions, which makes it suitable for optimization with gradient-based methods like stochastic gradient descent (SGD).


True probability distribution (one-shot)
Your model's predicted probability distribution

$$H(p, q) = -\sum_{x \epsilon classes} p(x) log\ q(x)$$

```python
import torch
import torch.nn.functional as F

class CustomCrossEntropyLoss(torch.nn.Module):
    def __init__(self):
        super(CustomCrossEntropyLoss, self).__init__()

    def forward(self, outputs, labels):
        probs = F.softmax(outputs, dim=1)
        log_probs = torch.log(probs)
        loss = -torch.sum(labels * log_probs) / labels.size(0)
        return loss

import torch.nn as nn
outputs = torch.tensor([[1.5, 2.0, 3.0],
                       [0.2, 1.1, 0.8],
                       [2.4, 1.6, 3.5]])
labels = torch.tensor([2, 1, 0])
criterion = nn.CrossEntropyLoss()
loss = criterion(outputs, labels)
print(f"Loss: {loss.item()}")
```

☆ BINARY CROSS ENTROPY LOSS

Binary Cross-Entropy Loss is a specific form of Cross-Entropy Loss used for binary classification tasks. It measures the dissimilarity between the predicted probability of the positive class and the true label. If the model predicts a probability close to 1 for the correct class, the loss is minimal. Conversely, if the prediction is far from the correct class, the loss increases, indicating that the model's prediction deviates from the true label. Binary Cross-Entropy Loss is convex and works well with gradient-based optimization methods like stochastic gradient descent (SGD) for training binary classifiers. For binary classification, we typically want to model the probability of one class (usually the positive class) using a single output neuron with a sigmoid activation function. This gives us a probability value between 0 and 1, which we then compare against the true label (0 or 1). In contrast, Cross-Entropy Loss for multi-class classification assumes that each class has its own output neuron, and it compares the predicted probability distribution across all classes. This would not be suitable for binary classification where only two outcomes (0 or 1) are considered.

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

```
import torch
import torch.nn.functional as F
y_true = torch.tensor([1, 0, 1, 1], dtype=torch.float32)
y_pred = torch.tensor([0.9, 0.1, 0.8, 0.7], dtype=torch.float32)
loss = F.binary_cross_entropy(y_pred, y_true)
```

☆ BALANCED CROSS ENTROPY

Balanced Cross-Entropy Loss is an extension of the Binary Cross-Entropy Loss that takes into account class imbalance. It assigns different weights to the positive and negative classes, allowing the model to focus more on the underrepresented class.

$$L = -\frac{1}{N}\sum_{i=1}^{N}[w_1 \cdot y_i \log(p_i) + w_2 \cdot (1 - y_i)\log(1 - p_i)]$$

The weights w1 and w2 are typically set inversely proportional to the class frequencies to balance the impact of each class. For example, if the dataset is highly imbalanced with more negatives than positives, you might set w1 (positive class weight) higher than w2 (negative class weight). We can implement it like the following:

```
loss = F.binary_cross_entropy(y_pred, y_true, weight=torch.tensor([w_pos, w_neg]))
```

☆ FOCAL LOSS

Balanced Cross-Entropy Loss adjusts the loss by assigning a higher weight to the underrepresented class, which helps mitigate class imbalance to some extent. However, the problem is that it still treats both easy and hard examples in the same way.

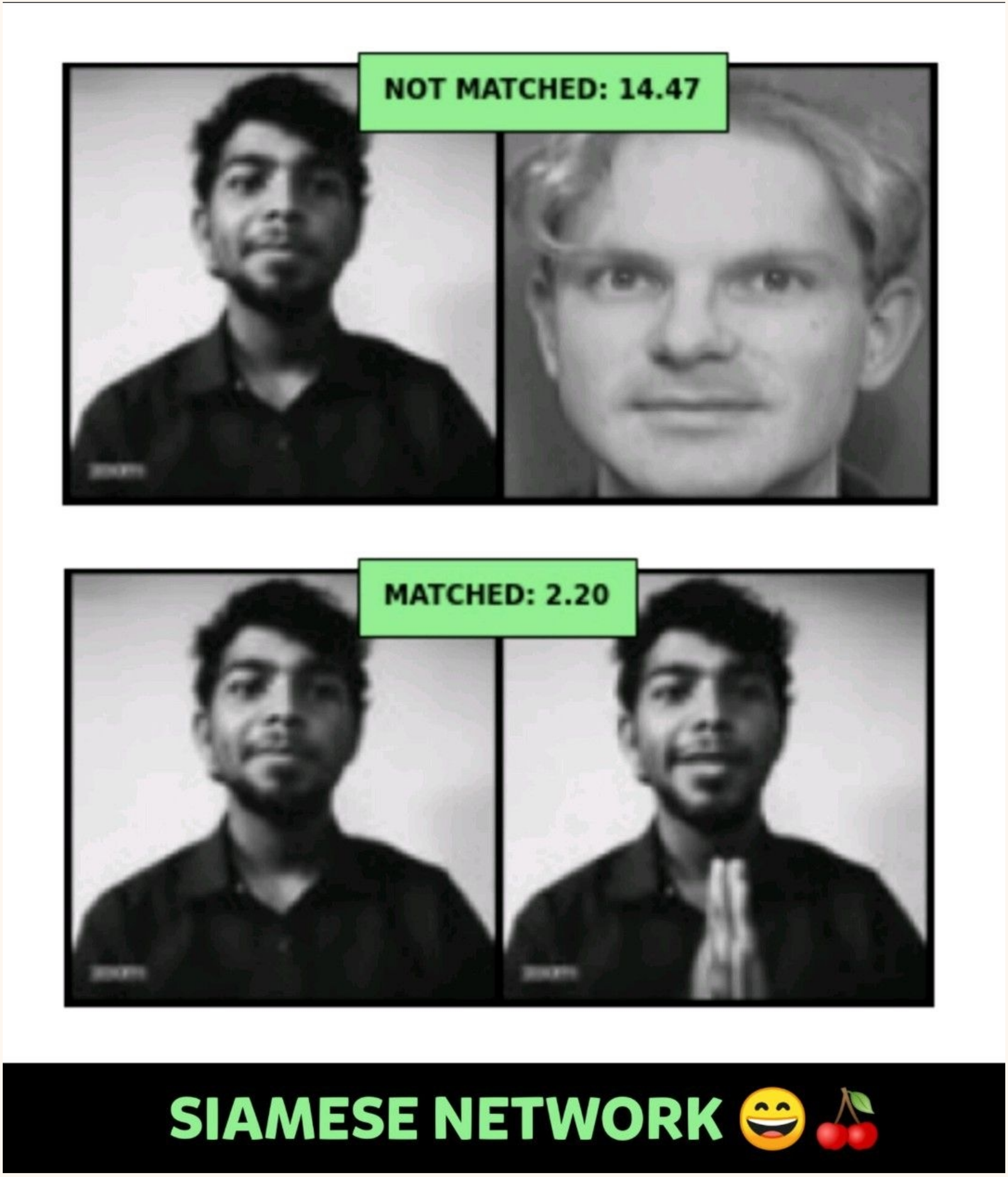
Focal Loss modifies the standard cross-entropy loss by adding a factor to **down-weight the loss** for well-classified examples and **focus more** on the misclassified examples, especially the hard examples. It is defined as:

$$FL(p_t) = -\alpha_t(1 - p_t)^{\gamma}\log(p_t)$$

Focal Loss achieves this by **focusing more on hard examples** and down-weighting easy examples, allowing the model to better learn from rare or difficult examples, thus handling extreme class imbalance more effectively.

☆ CONTRASTIVE LOSS

The core idea behind **Contrastive Loss** is to encourage the model to output a small distance for similar pairs and a large distance for dissimilar pairs. This is typically used in problems like face verification, where you want the model to learn to distinguish between similar and dissimilar instances (e.g., whether two faces belong to the same person).



$$L = \frac{1}{2N} \sum_{i=1}^{N} \left[y_i \cdot \mathrm{d}(x_i, x_i')^2 + (1 - y_i) \cdot \max(0, m - \mathrm{d}(x_i, x_i'))^2 \right]$$

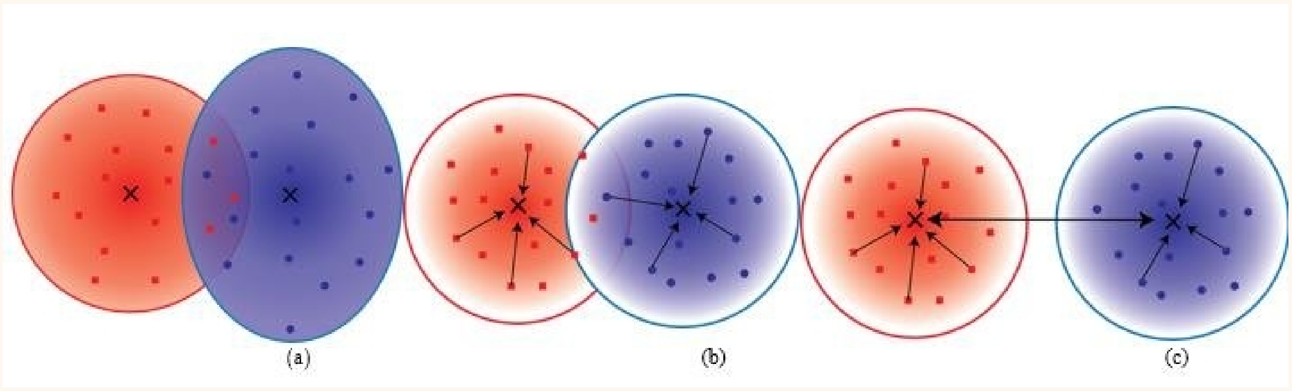
For **similar pairs** (yi=1), the loss is proportional to the square of the Euclidean distance between the two samples, i.e., we want to reduce the distance for similar pairs. For **dissimilar pairs** (yi=0), the loss is proportional to the square of the difference between the margin m and the Euclidean distance, ensuring the distance between dissimilar pairs exceeds the margin m. If the distance exceeds m, the loss is zero.

```python
import torch
import torch.nn.functional as F

def contrastive_loss(y_true, y_pred, margin=1.0):
    euclidean_distance = F.pairwise_distance(y_pred[0], y_pred[1])
    loss = torch.mean(
        y_true * torch.pow(euclidean_distance, 2) +
        (1 - y_true) * torch.pow(torch.clamp(margin - euclidean_distance, min=0), 2)
    )
    return loss

y_true = torch.tensor([1, 0, 1], dtype=torch.float32)
y_pred = torch.tensor([[0.2, 0.3], [0.4, 0.5], [0.1, 0.8]], dtype=torch.float32)
loss = contrastive_loss(y_true, y_pred, margin=1.0)
```

In training models with **Contrastive Loss**, the key idea is to select pairs of data points — these are typically **positive pairs** (similar) and **negative pairs** (dissimilar). Proper pair selection is crucial for the model's performance because the learning task is built around how well the model learns to distinguish between similar and dissimilar instances.

☆ TRIPLET LOSS

Triplet Loss is another powerful loss function used in **metric learning**, particularly to learn an embedding space where similar items are close together, and dissimilar items are far apart. It's commonly used in tasks like **face verification**, **image retrieval**, and **few-shot learning**. The goal of **Triplet Loss** is to minimize the distance between an **anchor** sample and a **positive** sample (same class), while maximizing the distance between the **anchor** and a **negative** sample (different class). This is achieved by ensuring that the anchor-positive pair is closer in the embedding space than the anchor-negative pair by a given margin.

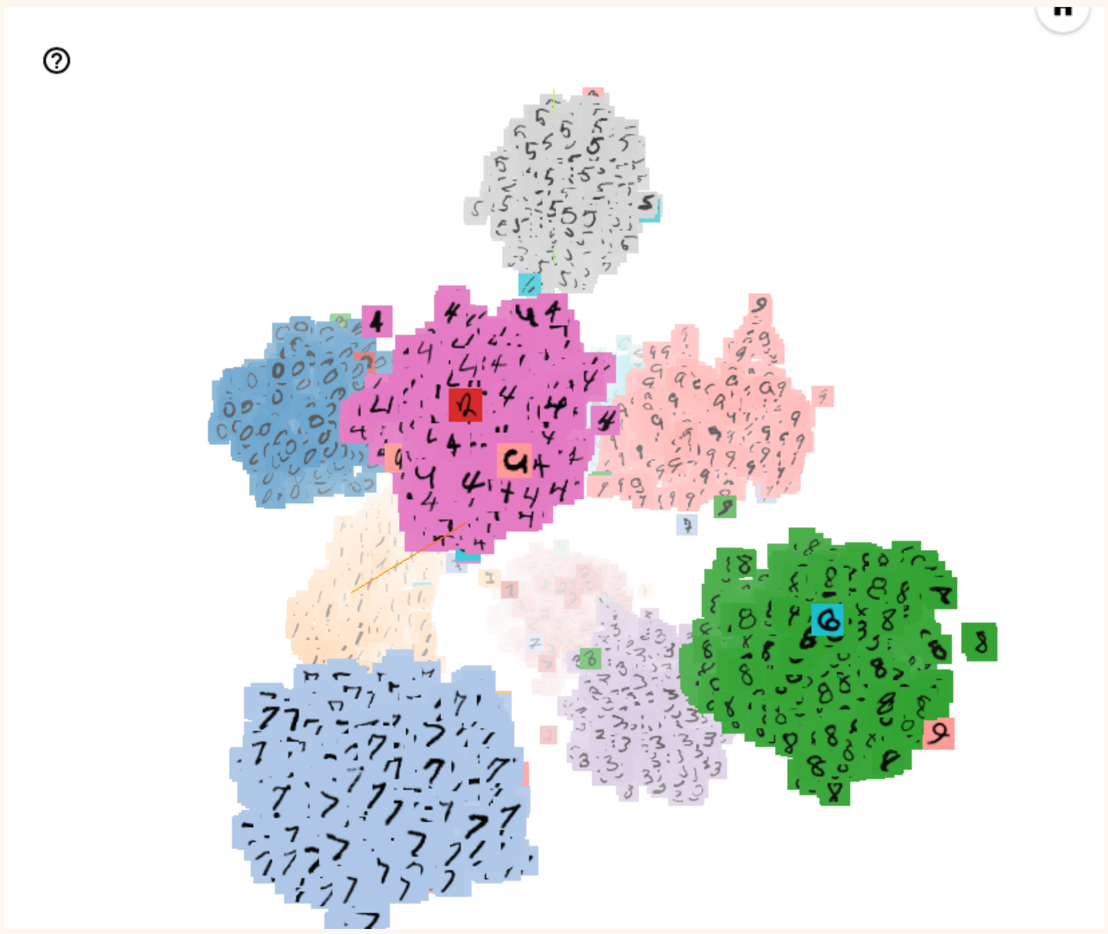


Given a triplet (A,P,N):

- **A**: Anchor sample
- **P**: Positive sample (same class as the anchor)
- **N**: Negative sample (different class from the anchor)

$$L(A, P, N) = \max\left(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0\right)$$

f(x) represents the embedding function of the model for a sample x. ‖f(A)−f(P)‖ is the Euclidean distance between the anchor and positive samples. ‖f(A)−f(N)‖ is the Euclidean distance between the anchor and negative samples. alpha is a **margin** that ensures the negative sample is sufficiently far away from the anchor (this margin is typically a hyperparameter).



The Clusters should come out like this.

MERCI