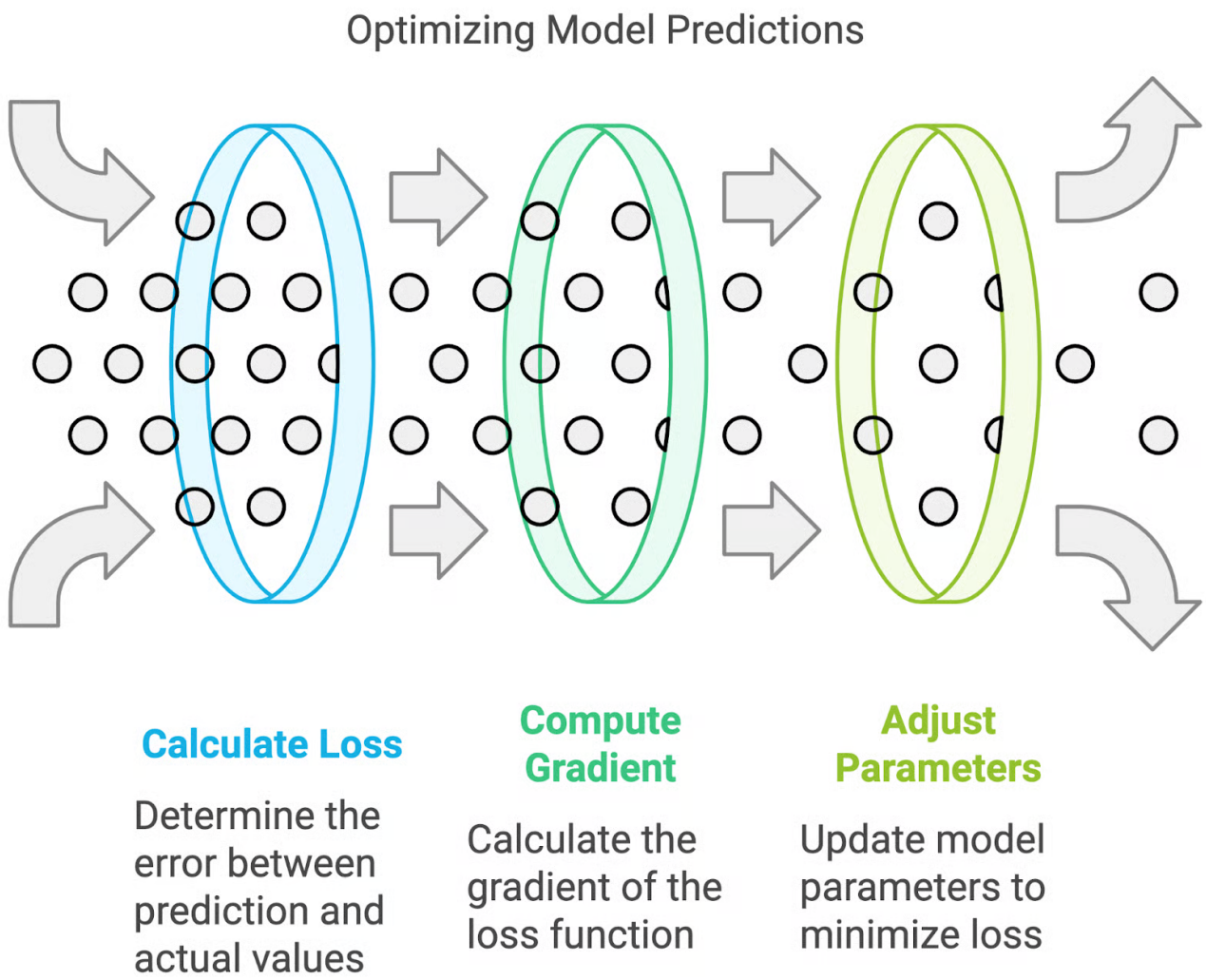
Module 6 - Tuần 2 - Loss Functions for Classification

Time-Series Team

Ngày 15 tháng 11 năm 2025

Mở đầu: Tại sao Loss Function lại quan trọng?

Trong Machine Learning, nếu kiến trúc mô hình là ”cơ thể”, dữ liệu là ”thức ăn”, thì **Hàm mất mát (Loss Function)** chính là ”la bàn”. Nó định hướng cho mô hình biết hướng đi đúng đắn để tối ưu hóa trọng số.



Blog này sẽ phân tích sâu các loại hàm loss phổ biến cho bài toán phân loại (Classification), từ cơ bản đến các kỹ thuật nâng cao như Label Smoothing và Ranking Loss.

Mục lục

1 Nền tảng Tư duy – Tại sao lại dùng Logarithm? (Information Theory)

Trước khi viết bất kỳ dòng code nào, ta cần trả lời câu hỏi: Tại sao máy học lại cần "ngạc nhiên"? Tại sao lại có hàm Logarit ở khắp mọi nơi?

1.1 Từ Xác suất (Probability) đến Sự ngạc nhiên (Surprise)

Trong bài toán phân loại, mô hình trả về một xác suất (ví dụ: 90% là ảnh con mèo). Nhưng làm sao để đo lường con số 90% đó tốt hay xấu so với thực tế? Các nhà khoa học dữ liệu sử dụng khái niệm "Surprise" (Sự ngạc nhiên) từ Lý thuyết thông tin.

Hãy tưởng tượng một ví dụ trực quan:

- **Trường hợp A**: Ta tung đồng xu và nó ngửa. Xác suất là 50% → không ngạc nhiên lắm.
- **Trường hợp B**: Mua vé số và trúng độc đắc. Xác suất cực thấp (ví dụ 0.00001%) → cực kỳ ngạc nhiên (High Surprise).

Từ đó, ta có mối quan hệ nghịch đảo: **Xác suất (P) càng thấp → Sự ngạc nhiên (S) càng cao**.

Ban đầu, công thức đơn giản là $S = \frac{1}{P}$. Tuy nhiên, công thức này gặp vấn đề lớn: Nếu có 2 sự kiện độc lập xảy ra liên tiếp, xác suất là phép nhân ($P(A) \cdot P(B)$), nhưng độ ngạc nhiên tự nhiên của con người lại là phép cộng (ngạc nhiên lần 1 + ngạc nhiên lần 2). Hàm $1/P$ không thỏa mãn tính chất cộng này.

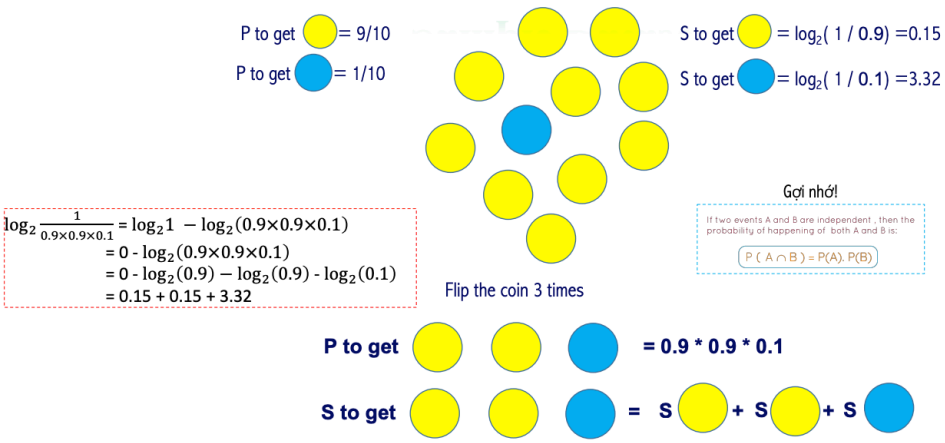
1.2 Tại sao lại dùng Logarithm? (The "Magic" of Log)

Để giải quyết vấn đề trên, Shannon (cha đẻ lý thuyết thông tin) đã đề xuất sử dụng hàm Logarithm. Đây là bước chuyển mình quan trọng nhất trong việc xây dựng hàm Loss.

Công thức đo lường sự ngạc nhiên chuẩn xác:

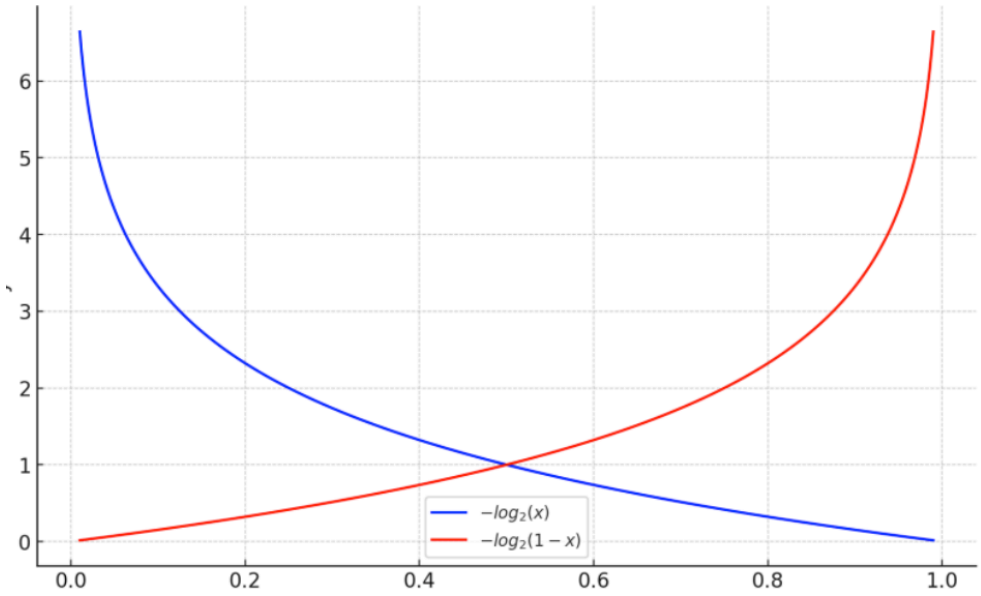
$$S = \log_2(\frac{1}{P}) = -\log_2(P)$$

Tại sao hàm Log lại ưu việt trong Machine Learning?



1. **Biến phép Nhân thành phép Cộng**: Khi tính toán xác suất của nhiều mẫu dữ liệu (như trong hình trên), xác suất tổng là tích của các xác suất thành phần ($0.9 \times 0.9 \times 0.1$). Việc nhân nhiều số nhỏ sẽ dẫn đến lỗi tràn số (underflow) trên máy tính. Logarit biến tích thành tổng ($\log(a \cdot b) = \log a + \log b$), giúp tính toán ổn định hơn rất nhiều.

2. **Trừng phạt sai lầm**: Nhìn vào đồ thị hàm $-\log(x)$ (hình dưới): Khi dự đoán đúng ($P \to 1$): Loss $\to 0$. Khi dự đoán sai ($P \to 0$): Loss $\to +\infty$. Điều này có nghĩa là mô hình sẽ bị "phạt" cực nặng nếu nó tự tin phán đoán sai (ví dụ: khẳng định chắc chắn 100% là mèo, nhưng thực tế là chó).



1.3 Entropy và KL Divergence: Thước đo khoảng cách

Hiểu được "Sự ngạc nhiên" của một sự kiện đơn lẻ, ta mở rộng ra cho toàn bộ hệ thống bằng khái niệm Entropy.

- **Entropy (H)**: Là sự ngạc nhiên trung bình. Nếu một tập dữ liệu rất hỗn loạn (ví dụ: số lượng gấu trúc đực và cái bằng nhau), Entropy sẽ đạt cực đại. Nếu dữ liệu bị lệch hẳn về một phía, Entropy sẽ thấp.

$$H(p) = -\sum p(x) \log p(x)$$

- KL Divergence (Kullback-Leibler Divergence): Đây chính là "cây thước" để đo khoảng cách giữa hai phân phối xác suất:

  1. Phân phối thật (P^*): Thực tế (Ground Truth), ví dụ con chó là [1, 0, 0].
  2. Phân phối dự đoán (P): Mô hình đoán, ví dụ [0.7, 0.2, 0.1].

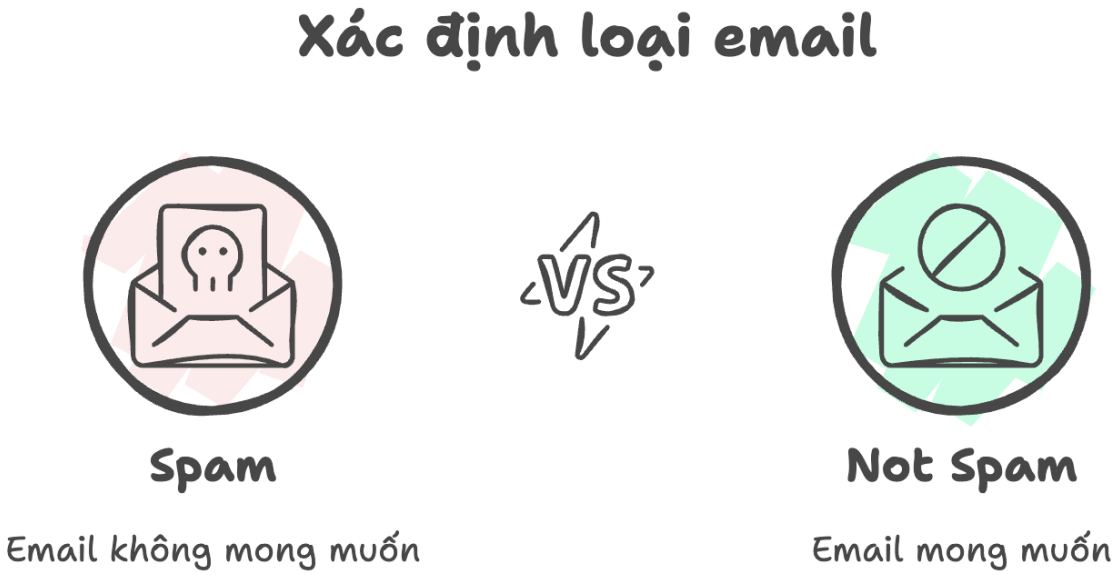
  Mục tiêu của việc huấn luyện (Training) chính là kéo phân phối P lại gần P^* nhất có thể. Công thức KL Divergence:

$$D_{KL}(P^*||P) = \sum P^*(x) \log \frac{P^*(x)}{P(x)}$$

  .

2 Binary Classification – Cuộc Chiến Giữa Xác Suất và Biên Độ

Bài toán kinh điển: Ta nhận được một email. Mô hình cần quyết định đây là **Spam (1)** hay **Not Spam (0)**.



2.1 Binary Cross-Entropy (BCE) Loss: Tiêu chuẩn của Deep Learning

BCE Loss (hay Log Loss) hoạt động dựa trên cơ chế phạt độ lệch về xác suất. Nó đặc biệt "nhạy cảm" với độ tự tin của mô hình.

2.1.1 Công thức và Cơ chế hoạt động

$$L = -[y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})]$$

Trong đó:

- y: Nhãn thật (0 hoặc 1).
- $\hat{y}$: Xác suất dự đoán (output của hàm **Sigmoid**, $0 \leq \hat{y} \leq 1$).

Mổ xẻ công thức: Công thức này thực chất là một câu lệnh *if-else* toán học:

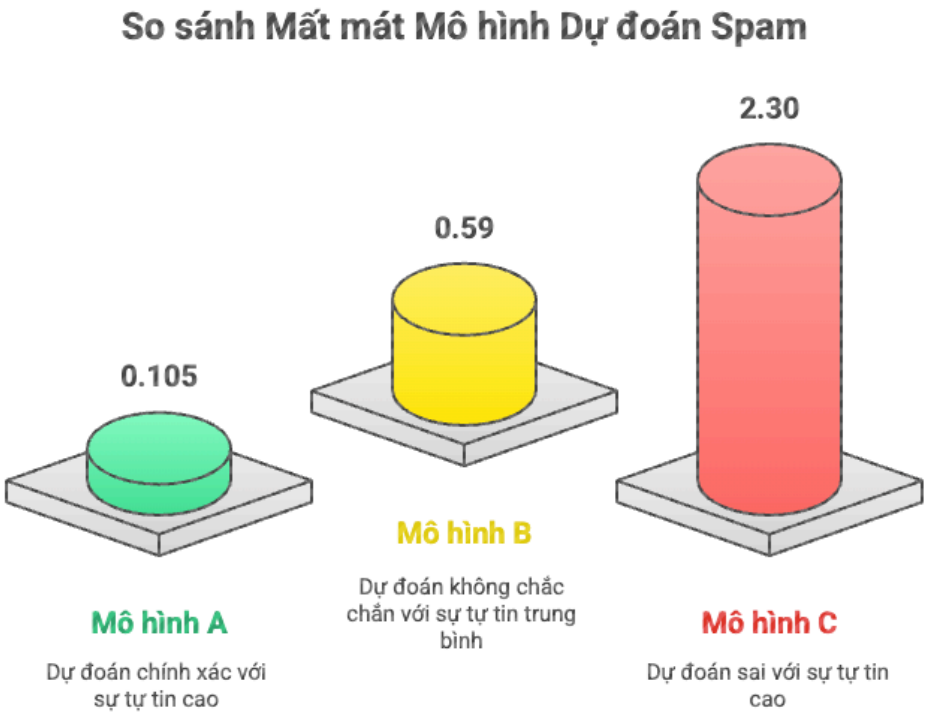
- Nếu nhãn thật là 1 ($y = 1$): Công thức còn lại $-\log(\hat{y})$. Ta muốn $\hat{y}$ càng gần 1 càng tốt (Loss $\approx$ 0).
- Nếu nhãn thật là 0 ($y = 0$): Công thức còn lại $-\log(1 - \hat{y})$. Ta muốn $\hat{y}$ càng gần 0 càng tốt để $(1 - \hat{y})$ gần 1.

2.1.2 Tại sao BCE lại "tàn nhẫn"?

Xem xét ví dụ phân loại Spam sau để thấy BCE trừng phạt sai lầm nặng nề như thế nào:

Giả sử nhãn thật là Not Spam ($y = 1$):

1. Mô hình A (Khá tốt): Dự đoán 90% là Not Spam ($\hat{y} = 0.9$).Loss $= -\ln(0.9) \approx 0.105$. Phạt rất nhẹ.
2. Mô hình B (Phân vân): Dự đoán 55% là Not Spam ($\hat{y} = 0.55$).Loss $= -\ln(0.55) \approx 0.59$. Phạt trung bình.
3. Mô hình C (Sai nghiêm trọng): Dự đoán 10% là Not Spam ($\hat{y} = 0.1$) $\rightarrow$ Tức là nó tin 90% đây là Spam!Loss $= -\ln(0.1) \approx 2.30$. Phạt cực nặng.



Bài học rút ra: **BCE Loss** không chỉ quan tâm đúng hay sai, mà còn quan tâm dự đoán tự tin đến mức nào. Sai mà còn "cố chấp" (tự tin cao) sẽ nhận hình phạt khổng lồ.

2.1.3 Convexity (Tính lồi) – Tại sao không dùng MSE?

Trong **Linear Regression**, ta dùng **Mean Squared Error (MSE)**. Nhưng trong bài toán phân loại với hàm **Sigmoid**, nếu dùng **MSE**, hàm **loss** sẽ trở nên "gồ ghề" (non-convex), khiến thuật toán Gradient Descent dễ bị kẹt ở các vũng trũng cục bộ (local minima). Ngược lại, hàm Log Loss (BCE) đảm bảo hàm mất mát là **Convex (Lồi)**, giúp **Gradient Descent** trượt trơn tru xuống đáy (Global Minimum).

2.1.4 Code PyTorch: Lưu ý quan trọng

Trong PyTorch, có 2 cách dùng, nhưng khuyến khích nên dùng **cách 2** để ổn định tính toán (numerical stability):

Cách 1: `nn.BCELoss` (Đầu vào phải qua **Sigmoid** trước)

```
# [cite: 836-837]
criterion = nn.BCELoss()
loss = criterion(torch.sigmoid(logits), y_true)
```

Cách 2: `nn.BCEWithLogitsLoss` (Khuyên dùng - Tích hợp sẵn Sigmoid). Cách này tránh lỗi tràn số khi log tính toán với các giá trị xác suất quá nhỏ (≈ 0)

```
# [cite: 849-850]
criterion = nn.BCEWithLogitsLoss()
loss = criterion(y_pred_logits, y_true)
```

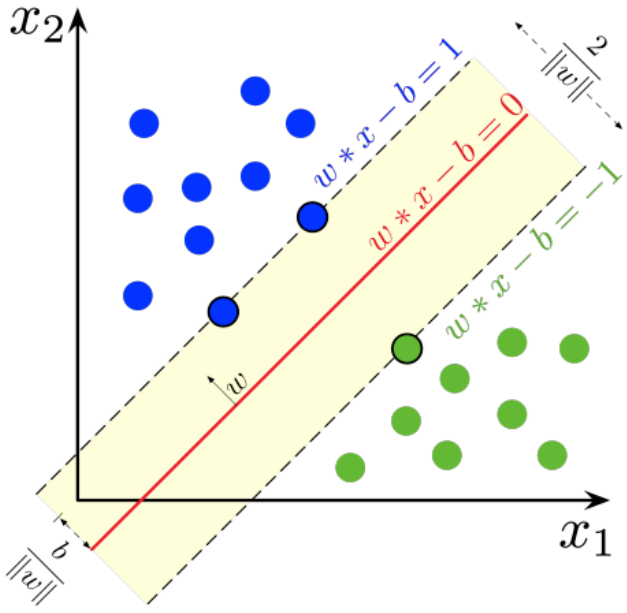
2.2 Hinge Loss: "Vệ sĩ" của SVM

Nếu **BCE** là một nhà toán học tính toán xác suất chi li, thì **Hinge Loss** là một vệ sĩ chỉ quan tâm đến khoảng cách an toàn.

2.2.1 Công thức và Tư duy Margin (Biên độ - Lề)

$$L = \max(0, 1 - y \cdot f(x))$$

- y: Nhãn thật (quy ước là -1 hoặc +1 thay vì 0 và 1).
- $f(x)$: Điểm số đầu ra (Score) của mô hình (chưa qua Sigmoid).

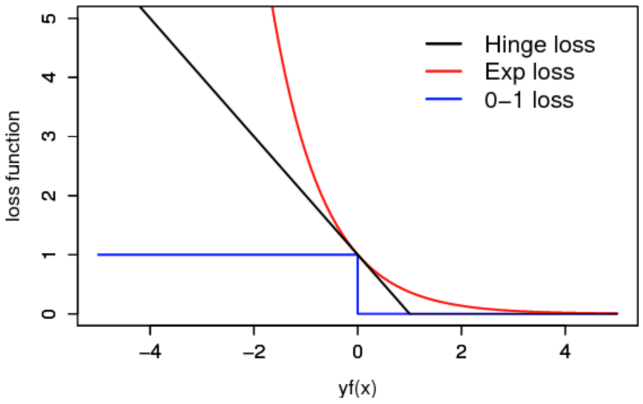


Cơ chế "Margin" (Biên độ)

- Vùng an toàn ($y \cdot f(x) \geq 1$): Mô hình dự đoán đúng và điểm số đủ lớn (vượt qua biên 1). Loss = 0. Không bị phạt.
- Vùng nguy hiểm ($y \cdot f(x) < 1$):
  - Dự đoán đúng nhưng điểm thấp ($0 < y \cdot f(x) < 1$): Vẫn bị phạt nhẹ vì chưa đủ "an toàn".
  - Dự đoán sai ($y \cdot f(x) < 0$): Phạt nặng tịnh tiến theo mức độ sai.

2.2.2 Ví dụ minh họa

Nhìn vào biểu đồ so sánh



- Đường màu đen (Hinge Loss) bắt đầu tăng giá trị phạt ngay khi $y \cdot f(x) < 1$.
- Điều này ép mô hình không chỉ phân loại đúng mà còn phải đẩy các điểm dữ liệu ra xa ranh giới quyết định (Decision Boundary) ít nhất là 1 đơn vị. Đây là lý do SVM có khả năng tổng quát hóa tốt.

2.2.3 Code PyTorch triển khai Hinge Loss

PyTorch không có hàm `HingeLoss` trực tiếp cho classification nhị phân kiểu này, ta thường tự viết hoặc dùng `MultiMarginLoss` cho đa lớp. Dưới đây là cách tự viết đơn giản:

```
# [cite: 1051-1053]
def hinge_loss(y_pred, y_true):
    # y_true ảphi là -1 ặhoc 1
    return torch.mean(torch.clamp(1 - y_pred * y_true, min=0))
```

So sánh Binary Cross-Entropy và Hinge Loss

Đặc điểm	Mô hình	Bản chất	Đầu ra	Ưu điểm
Binary Cross-Entropy (BCE)	Deep Learning, Logistic Regression	Xác suất	Giá trị xác suất	Hàm loss mượt
Hinge Loss	Support Vector Machines (SVM)	Biên độ	Giá trị thực	Ranh giới phân loại rõ ràng

3 Multi-class Classification – Khi Thế Giới Không Chỉ Có "Đúng" và "Sai"

Trong **Binary Classification**, ta chỉ có 2 cửa (0 hoặc 1). Nhưng trong **Multi-class**, chúng ta có C cửa. Thách thức lớn nhất ở đây là làm sao để các xác suất này "cạnh tranh" nhau một cách công bằng.

3.1 Cross-Entropy (CE) Loss: "Người trọng tài" Softmax

Để giải quyết bài toán đa lớp, ta cần một cơ chế để biến đổi các điểm số thô (logits) từ mô hình thành một phân phối xác suất hợp lệ (tổng bằng 1). Đó chính là vai trò của **Softmax**.

3.1.1 Từ Softmax đến Cross-Entropy

- **Softmax**: Ép các giá trị đầu ra z_i vào khoảng $(0, 1)$ và đảm bảo tổng của chúng bằng 1.

$$p_i = \text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

- **Cross-Entropy Loss**: Sau khi có xác suất p, ta so sánh nó với nhãn thật y (dạng one-hot vector).

$$CE = -\sum_{i=1}^{C} y_i \log(p_i)$$

3.1.2 Tại sao CE chỉ quan tâm đến lớp đúng?

Hãy nhìn vào công thức trên. Giả sử ta đang phân loại ảnh con chó (lớp 0). Nhãn one-hot sẽ là $y = [1, 0, 0, 0]$. Khi thay vào công thức tổng:

$$CE = -[1 \cdot \log(p_{ch}) + 0 \cdot \log(p_{mo}) + 0 \cdot \log(p_{nga}) + \dots]$$
$$CE = -\log(p_{ch})$$

Bài học cốt lõi: Trong bài toán **Multi-class** tiêu chuẩn (dùng one-hot), hàm **Loss** chỉ quan tâm đến việc dự đoán xác suất cho lớp đúng là bao nhiêu. Nó không trực tiếp phạt việc bạn dự đoán sai các lớp khác, miễn là xác suất lớp đúng đủ cao.

3.1.3 Code PyTorch: Đừng nhầm lẫn!

Một sai lầm kinh điển của người mới dùng **PyTorch**:

- Mô hình không cần lớp `Softmax` ở cuối cùng.
- Hàm `nn.CrossEntropyLoss` đã tự động thực hiện `LogSoftmax` + `NLLLoss` bên trong.

```
# [cite: 1316-1323]
import torch.nn as nn
# Logits dau ra tu mo hinh (chua qua Softmax)
logits = torch.tensor([[2.0, 1.0, 0.1]])
# Nhan dung la lop 0
labels = torch.tensor([0])

criterion = nn.CrossEntropyLoss()
loss = criterion(logits, labels) # Tu dong tinh Softmax -> Log -> Loss
```

3.2 Sparse Categorical Cross-Entropy (SCCE): Tiết kiệm là quốc sách

Về mặt toán học, **SCCE** và **CE** là một. Kết quả tính ra giống hệt nhau. Sự khác biệt nằm ở kỹ thuật lập trình và tối ưu bộ nhớ.

3.2.1 Vấn đề của One-hot Encoding

Giả sử, ta đang huấn luyện mô hình dự đoán từ vựng tiếng Anh với 50,000 từ.

- Nếu dùng **CE** thường: Mỗi nhãn là một vector chứa 49,999 số 0 và 1 số 1 → Lãng phí bộ nhớ khủng khiếp để lưu trữ những số 0 vô nghĩa.

3.2.2 Giải pháp SCCE

Thay vì lưu vector `[0, 0, 1, 0, ...]`, ta chỉ cần lưu số nguyên (index) `2`.

- **CE**: Input nhãn dạng [`batch_size`, `num_classes`].
- **SCCE**: Input nhãn dạng [`batch_size`].

  Lưu ý: Trong **PyTorch**, hàm `nn.CrossEntropyLoss` mặc định hỗ trợ chuẩn SCCE (nhận nhãn là index class (long tensor)) nên không cần hàm riêng biệt. Trong **TensorFlow/Keras** mới phân biệt rõ `CategoricalCrossentropy` và `SparseCategoricalCrossentropy`.

3.3 Label Smoothing: Kỹ thuật chống "Ảo tưởng sức mạnh" (Overfitting)

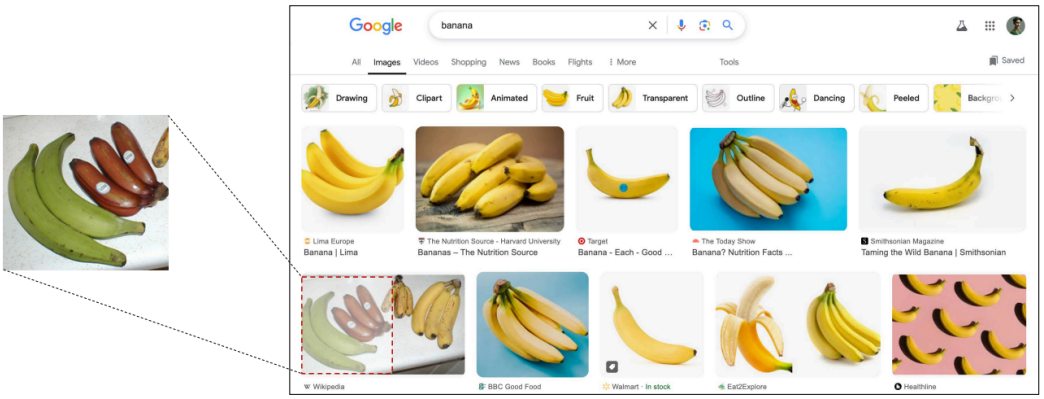
Đây là phần nâng cao thú vị nhất trong bài toán **Multi-class**.

3.3.1 Vấn đề: "Hard Label" quá cực đoan

Khi ta gán nhãn One-hot (ví dụ: Chó = 1, Mèo = 0), ta đang ép mô hình phải tin tuyệt đối rằng "đây chắc chắn là chó, không thể có 0.0001% nào là mèo". Điều này dẫn đến 2 hậu quả:

- **Overfitting**: Mô hình học vẹt trên dữ liệu train để đạt xác suất 1.0, nhưng kém trên dữ liệu mới.
- **Overconfidence**: Mô hình quá tự tin vào dự đoán sai.

Hãy xem ví dụ quả chuối sau:



Hầu hết chuối màu vàng, nhưng vẫn có chuối màu xanh hoặc đỏ. Nếu ép nhãn Yellow=1, mô hình sẽ gặp khó khăn khi gặp quả chuối lạ.

3.3.2 Giải pháp: "Làm mềm" nhãn (Label Smoothing)

Thay vì dạy mô hình: "Đây là Chó (100%)", ta dạy: "Đây là Chó (90%), nhưng hãy để dành 10% khả năng cho các con vật khác".

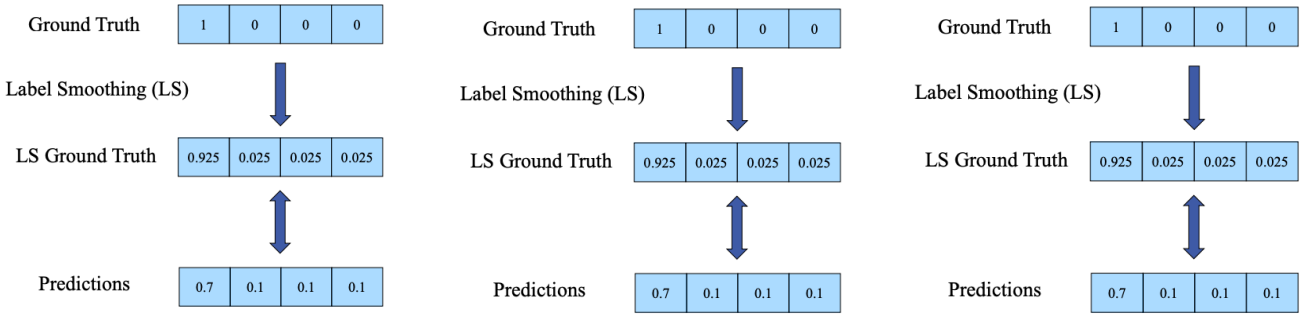
Công thức biến đổi nhãn:

$$\tilde{y}_i = (1 - \alpha)y_i + \frac{\alpha}{C}$$

- y_i: Nhãn one-hot gốc (0 hoặc 1).
- α: Hệ số làm mềm (thường chọn 0.1).
- C: Số lượng lớp.

Ví dụ minh họa: Với 4 lớp, $\alpha = 0.1$.

- Nhãn gốc (Chó): `[1, 0, 0, 0]`.
- Nhãn mềm:
  - Tại vị trí Chó: $(1 - 0.1) * 1 + 0.1/4 = 0.925$
  - Tại vị trí khác: $(1 - 0.1) * 0 + 0.1/4 = 0.025$
  - Nhãn mới: `[0.925, 0.025, 0.025, 0.025]`.



3.3.3 Tác dụng

Label Smoothing giúp mô hình tạo ra các cụm **(clusters)** trong không gian đặc trưng chặt chẽ hơn và tách biệt tốt hơn, giúp mô hình tổng quát hóa (**generalization**) tốt hơn nhiều trên dữ liệu kiểm thử (test set).