

# Tuần 4 - Phân loại khả năng mắc bệnh tim dựa vào các thông tin lâm sàng

Time-Series Team

Ngày 5 tháng 10 năm 2025

Dự án Hệ thống phân loại bài báo gồm các nội dung chính:

- *Đặt vấn đề*
- *Mở rộng của nhóm*
- *Giới thiệu chi tiết mở rộng của nhóm*

## Mục lục

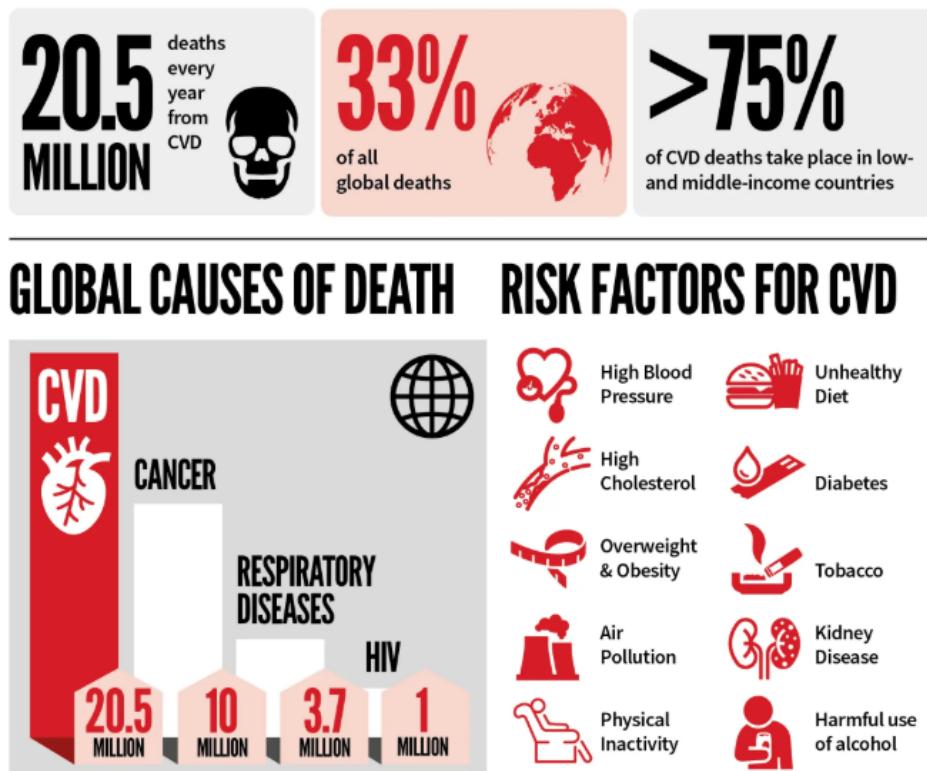
<b>1</b>	<b>Đặt vấn đề</b>	<b>3</b>
1.1	Giới thiệu bài toán . . . . .	3
1.2	Giới hạn bài toán hiện tại . . . . .	3
1.3	Hướng mở rộng . . . . .	4
1.3.1	Hướng 1 – So sánh các mô hình Ensemble Learning: Stacking và TSA trong tối ưu hóa trọng số dự đoán . . . . .	4
1.3.2	Hướng 2 – CardioFusion: Ghép đa phương thức EchoNet + Cleveland . . . . .	4
<b>2</b>	<b>Hướng mở rộng 1: So sánh các mô hình Ensemble Learning: Stacking và TSA trong tối ưu hóa trọng số dự đoán</b>	<b>5</b>
2.1	Mô tả pipeline . . . . .	5
2.2	Giới thiệu Tunicate Swarm Intelligence (TSA) . . . . .	5
2.2.1	Giới thiệu . . . . .	5
2.2.2	Nguồn gốc sinh học . . . . .	5
2.2.3	Mô hình toán học . . . . .	6
2.2.4	Định nghĩa hàm fitness trong AI . . . . .	7
2.2.5	Ví dụ: tối ưu trọng số ensemble . . . . .	7
2.2.6	Khởi tạo . . . . .	7
2.2.7	Cập nhật . . . . .	7
2.2.8	So sánh với thuật toán khác . . . . .	7
2.2.9	Ứng dụng trong AI Engineering . . . . .	7
2.2.10	Pseudocode (Python style) . . . . .	8
2.2.11	Biến thể nâng cao . . . . .	8
2.2.12	Bài toán ví dụ . . . . .	8
2.3	Giới thiệu Stacking Model . . . . .	13
2.4	Các hàm triển khai . . . . .	13
2.5	Kết quả . . . . .	17
2.6	Explainable AI . . . . .	18

<b>3 Hướng mở rộng 2: CardioFusion: Ghép đa phương thức EchoNet + Cleveland cho phân loại bệnh tim</b>	<b>23</b>
3.1 Mô tả pipeline . . . . .	23
3.1.1 Tiền xử lý & Kỹ thuật đặc trưng (Cleveland) . . . . .	23
3.2 Chuẩn hoá ảnh/video . . . . .	24
3.3 Chia tập . . . . .	25
3.4 Giới thiệu tệp dữ liệu EchoNet . . . . .	25
3.5 Tổng quan về bộ dữ liệu EchoNet-Dynamic . . . . .	25
3.5.1 Thành phần dữ liệu . . . . .	26
3.5.2 Ý nghĩa lâm sàng . . . . .	27
3.5.3 Thông kê mô tả . . . . .	27
3.5.4 Bảo mật và xử lý dữ liệu . . . . .	27
3.5.5 Ứng dụng . . . . .	28
3.6 Giới thiệu mô hình MLP . . . . .	28
3.6.1 Các thành phần MLP . . . . .	28
3.6.2 Cách thức hoạt động . . . . .	29
3.7 Giới thiệu ResNet50 và CNN . . . . .	30
3.7.1 CNN cơ bản . . . . .	30
3.7.2 LeNet-5 (1998): Ứng Dụng Đầu Tiên Thực Tế . . . . .	34
3.7.3 AlexNet (2012): Bước Nhảy Vọt Với Độ Sâu Và GPU . . . . .	36
3.7.4 ZFNet (2013): Tinh Chỉnh Và Trực Quan Hóa . . . . .	37
3.7.5 VGGNet (2014): Tăng Độ Sâu Với Kernel Nhỏ . . . . .	39
3.7.6 ResNet (2015): Mạng Siêu Sâu Với Residual Blocks . . . . .	40
3.7.7 ResNet50 Trích xuất đặc trưng . . . . .	43
3.8 Giới thiệu mô hình hợp nhất (Fusion Model) . . . . .	47
3.9 Các hàm triển khai . . . . .	48
3.9.1 Trích xuất đặc trưng ảnh với ResNet-50 . . . . .	48
3.9.2 Mô hình hợp nhất (Fusion CNN + MLP) . . . . .	49
3.9.3 Tiền xử lý & xây dựng đặc trưng (Feature Engineering) . . . . .	50
3.9.4 Chiết xuất đặc trưng từ EchoNet . . . . .	52
3.9.5 Tạo bộ dữ liệu đa phương thức (Cleveland + EchoNet) . . . . .	54
3.9.6 Huấn luyện và đánh giá mô hình Fusion . . . . .	56
3.10 Kết quả . . . . .	61
3.10.1 Nhận xét kết quả . . . . .	63
<b>4 API cho hướng mở rộng 2 - CardioFusion: Ghép đa phương thức EchoNet + Cleveland cho phân loại bệnh tim</b>	<b>63</b>
4.1 API là gì và vì sao lại cần API ? . . . . .	63
4.1.1 API có gì ? . . . . .	64
4.2 Lưu trữ Weight của mô hình cho Inference . . . . .	65
4.3 Cấu trúc folder tổng quan . . . . .	66
4.4 Kiến trúc tổng quan của API . . . . .	67
4.5 API End Points . . . . .	68
4.6 API Main Workflow (Inference workflow) . . . . .	69

## 1 Đặt vấn đề

### 1.1 Giới thiệu bài toán

Bệnh tim là nguyên nhân tử vong hàng đầu trên Thế giới và cũng là một trong những thách thức lớn nhất về sức khỏe đối với cả nam giới và phụ nữ. Hơn một nửa số người chết vì bệnh tim ra đi đột ngột mà không hề có dấu hiệu cảnh báo – và nửa còn lại thì căn bệnh đã âm thầm tồn tại trong cơ thể họ nhiều năm trước khi bùng phát. Theo thống kê của tổ chức World Heart Federation(WHF), có khoảng 20.5 triệu người tử vong mỗi năm do các bệnh liên quan đến tim mạch(CVD-Cardiovascular Diseases).



Hình 1: Thống kê của WHF về tỉ lệ tử vong do các bệnh tim mạch

Ngày nay, việc chuẩn đoán sớm chính xác và hiệu quả bệnh tim mạch dựa trên các thông tin lâm sàng trở lên cấp bách hơn bao giờ hết. Nhiều nghiên cứu đã được thực hiện và nhiều mô hình học máy khác nhau được sử dụng để phân loại và dự đoán chẩn đoán bệnh tim. Trong dự án, nhóm TimeSeries sẽ thực hiện phát triển một mô hình phân loại khả năng mắc bệnh tim.

### 1.2 Giới hạn bài toán hiện tại

Trong dự án ban đầu, các TA&STA sử dụng bộ dữ liệu Cleveland Heart Disease – gồm 303 bệnh nhân, 13 đặc trưng lâm sàng (tuổi, giới tính, huyết áp, cholesterol, nhịp tim, đau ngực, v.v.) và nhãn mục tiêu (có/không mắc bệnh tim), áp dụng các mô hình truyền thống như Naive Bayes, KNN, Decision Tree, cũng như một số mô hình ensemble (Random Forest, AdaBoost, Gradient Boosting, XGBoost). Kết quả đạt được khá khả quan, với độ chính xác lên đến 97%.

Model	Val				Test			
	Origin	FE	Origin+DT	FE+DT	Origin	FE	Origin+DT	FE+DT
Naive Bayes	.90	.90	.93	.93	.84	.84	.84	.84
KNN	.90	.90	<b>.97</b>	.87	.84	.84	.87	.84
KMeans	.70	.80	.83	.63	<b>.87</b>	<b>.87</b>	.84	.77
Decision Tree	.93	<b>.93</b>	.93	<b>.93</b>	.81	.81	.81	.81
Ensemble (KNN,DT,NB)	.87	<b>.93</b>	.90	.87	.84	.90	.84	.84
Random Forest	<b>.97</b>	.90	<b>.97</b>	.84	.90	.87	<b>.93</b>	.84
AdaBoost	<b>.97</b>	.81	<b>.97</b>	.81	<b>.97</b>	.84	<b>.93</b>	.84
Gradient Boosting	.87	.81	.87	.84	.83	.81	<b>.93</b>	.81
XGBoost	.90	.84	.93	.81	.87	.87	.90	<b>.87</b>

Hình 2: Kết quả thực nghiệm theo based-line của các TA

Tuy kết quả rất khả quan, nhưng, cách tiếp cận này vẫn tồn tại một số hạn chế:

- **Khả năng tổng quát hóa:** Kết quả cao trên dataset nhỏ có nguy cơ overfitting, chưa chắc đã áp dụng tốt trong đặc trưng thực tế lâm sàng với dữ liệu phức tạp hơn.
- **Giới hạn dữ liệu:** Cleveland dataset chỉ chứa dữ liệu bảng (tabular), quy mô nhỏ (303 mẫu), thiếu tính đa dạng.
- **Giới hạn mô hình:** Các mô hình ensemble dù mạnh nhưng chủ yếu dựa vào tabular data, khó khai thác thông tin đa phương thức (ví dụ hình ảnh/video y khoa).

### 1.3 Hướng mở rộng

Và để cải thiện khắc phục các vấn đề nêu trên, nhóm TimeSeries tiếp cận giải quyết từng vấn đề theo 2 hướng chính.

#### 1.3.1 Hướng 1 – So sánh các mô hình Ensemble Learning: Stacking và TSA trong tối ưu hóa trọng số dự đoán

- **Mục tiêu:** Tập trung cải thiện generalization và độ ổn định.
- **Phương pháp cốt lõi:**
  1. Không tạo thêm dữ liệu mới, vẫn dùng Cleveland (tabular).
  2. Thay vì gán trọng số thủ công hay voting đơn giản, sử dụng thuật toán tối ưu bầy đàn TSA để tìm trọng số tối ưu khi kết hợp các mô hình (stacking ensemble).

#### 1.3.2 Hướng 2 – CardioFusion: Ghép đa phương thức EchoNet + Cleveland

- **Mục tiêu:** Khai thác triệt để cả dữ liệu định lượng (số liệu lâm sàng) và dữ liệu trực quan (hình ảnh y khoa), tạo hệ thống dự đoán toàn diện hơn.
- **Phương pháp cốt lõi:**
  1. Sử dụng **ResNet-50** để trích đặc trưng từ EchoNet và **MLP** để trích đặc trưng từ Cleveland.
  2. Hợp nhất (fusion) đặc trưng CNN + MLP để dự đoán bệnh tim.

## 2 Hướng mở rộng 1: So sánh các mô hình Ensemble Learning: Stacking và TSA trong tối ưu hóa trọng số dự đoán

### 2.1 Mô tả pipeline

Nhận thấy rằng hướng project gốc chỉ sử dụng các mô hình cây đơn lẻ để so sánh hiệu suất, nhóm đề xuất ý tưởng đưa ra một mô hình *ensemble* để kết hợp các mô hình cây (base models) lại với nhau. Nhóm xây dựng hai hướng tiếp cận chính, được triển khai **song song và độc lập** như sau:

- **Stacking + meta model:** Phương pháp này “ráp” các mô hình cây (base models) lại với nhau, sau đó đưa đầu ra của chúng qua một *meta-model* để học cách tối ưu trọng số đóng góp của từng mô hình vào dự đoán cuối cùng. Nhóm lựa chọn Logistic Regression (LR) làm *meta-model* vì tính tuyến tính, khả năng giải thích tốt và tránh overfitting trên không gian đặc trưng đã được tóm gọn (từ các base models).
- **TSA (Tuncate Swarm Algorithm):** Cũng vẫn sử dụng stacking để kết hợp các mô hình cây, tuy nhiên thay vì đưa vào *meta-model*, nhóm sử dụng thuật toán TSA. Đây là một thuật toán meta-heuristic lấy cảm hứng từ hành vi bầy đàn của loài tuncate. TSA tối ưu trực tiếp vector trọng số  $w$  cho các mô hình cơ sở nhằm tối đa hóa một chỉ số mục tiêu (ví dụ: AUC, F1-score, AP hoặc Accuracy). Ý tưởng này khác ở chỗ không huấn luyện *meta-model*, mà sử dụng cơ chế tìm kiếm toàn cục để tìm ra trọng số tốt nhất trên tập validation.

Để đảm bảo tính khách quan, nhóm bổ sung một baseline **Equal Weights** – tức phân bổ trọng số bằng nhau cho tất cả base models – nhằm so sánh với hai phương pháp tối ưu ở trên. Qua đó, nhóm có thể quan sát hiệu suất giữa *có tối ưu và không tối ưu* trọng số trên bốn biến thể dataset. Ngoài các mô hình cây truyền thống trong project gốc, nhóm còn thêm LGBM và CatBoost nhằm tăng sự đa dạng về base models.

Toàn bộ pipeline sẽ được triển khai như sau:

### 2.2 Giới thiệu Tuncate Swarm Intelligence (TSA)

#### 2.2.1 Giới thiệu

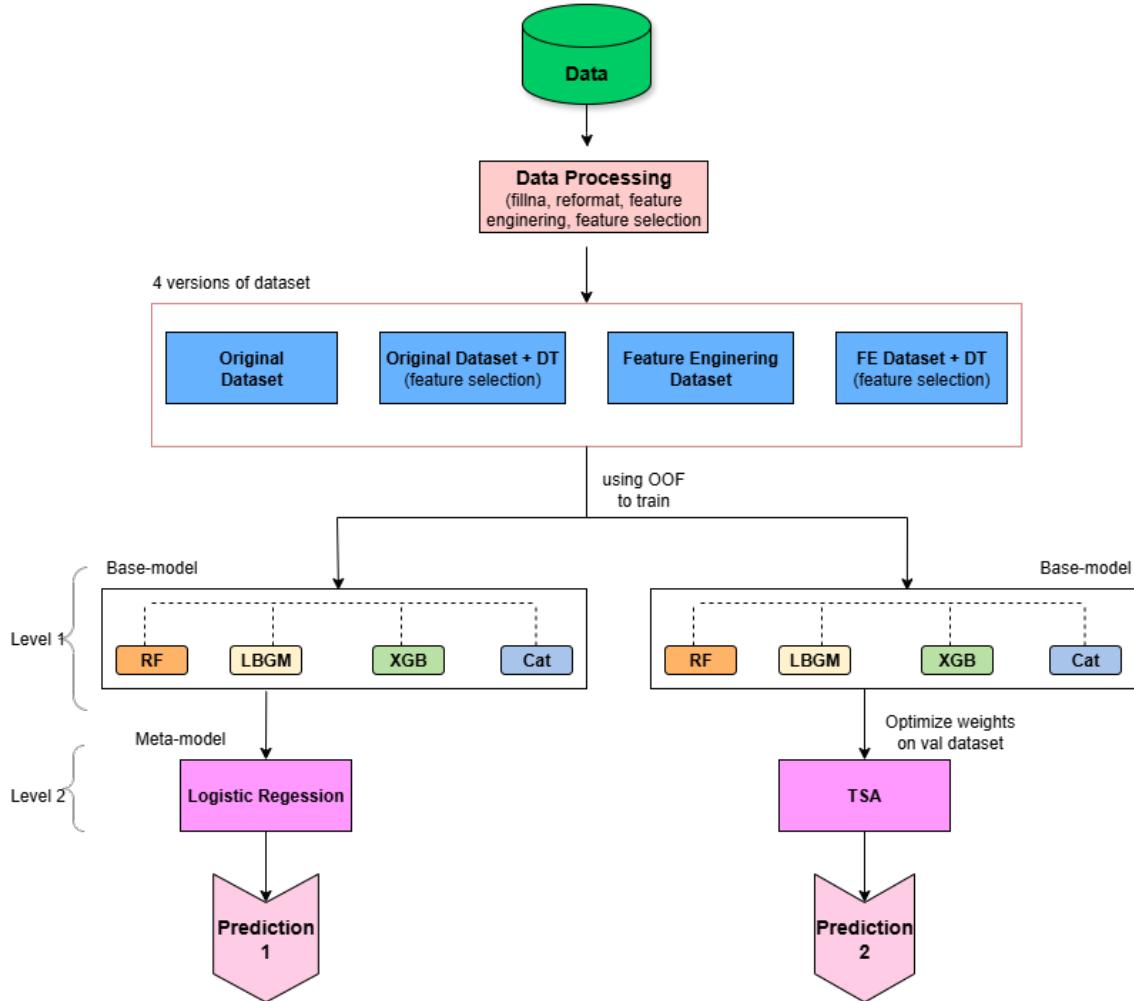
Trong trí tuệ nhân tạo, bài toán tối ưu hóa xuất hiện ở hầu hết các bước: từ huấn luyện mô hình học sâu (deep learning), chọn siêu tham số (hyperparameter tuning), tới kết hợp mô hình (ensemble learning) hay lựa chọn đặc trưng (feature selection).

Khi hàm mục tiêu không khả vi (ví dụ: tối ưu Accuracy, F1-score, AUC), các phương pháp **meta-heuristic** trở nên quan trọng. **Tuncate Swarm Intelligence (TSA)** là một meta-heuristic mới, mô phỏng hành vi bầy đàn của sinh vật biển gọi là tuncate. Thuật toán này đơn giản, dễ cài đặt, ít tham số và hiệu quả cạnh tranh với *Particle Swarm Optimization (PSO)* hay *Genetic Algorithm (GA)*.

#### 2.2.2 Nguồn gốc sinh học

Tuncate bối theo bầy với ba hành vi chính:

- **Theo leader:** các cá thể tiến về vị trí của con mạnh nhất (leader).
- **Ảnh hưởng của dòng chảy:** thêm dao động ngẫu nhiên mô phỏng tác động từ môi trường nước.
- **Tránh dồn cụm:** duy trì đa dạng quần thể, không để tất cả rơi vào một điểm duy nhất.



### 2.2.3 Mô hình toán học

Một cá thể  $X_i^t \in \mathbb{R}^d$  tại vòng lặp  $t$  được cập nhật:

$$X_i^{t+1} = X_{\text{best}}^t - A \odot (X_{\text{best}}^t - X_i^t),$$

trong đó:

- $X_{\text{best}}^t$  là nghiệm tốt nhất (leader),
- $A$  là hệ số điều khiển dựa trên tham số sinh học.

Hệ số  $A$  được xác định bởi:

$$G = c_2 + c_3 - F, \quad M = P_{\min} + c_1(P_{\max} - P_{\min}), \quad A = G \cdot M,$$

với:

- $c_1, c_2, c_3 \sim U(0, 1)$  (số ngẫu nhiên),
- $F$  giá trị fitness (càng nhỏ càng tốt nếu là bài toán minimization),
- $P_{\min}, P_{\max}$  là giới hạn tìm kiếm.

### 2.2.4 Định nghĩa hàm fitness trong AI

Trong các tác vụ AI, fitness có thể được định nghĩa như:

- $F = 1 - \text{Accuracy}$  đối với phân loại,
- $F = -\text{F1-score}$  khi dữ liệu mất cân bằng,
- $F = -\text{AUC}$  trong bài toán nhị phân,
- Validation loss khi tuning siêu tham số.

### 2.2.5 Ví dụ: tối ưu trọng số ensemble

Giả sử cần kết hợp ba mô hình: Random Forest (RF), XGBoost (XGB), và CNN với trọng số ( $w_{RF}, w_{XGB}, w_{CNN}$ ).

Ràng buộc:

$$w_i \geq 0, \quad \sum_i w_i = 1.$$

### 2.2.6 Khởi tạo

Ba vector trọng số ban đầu:

$$A = (0.33, 0.33, 0.33), \quad B = (0.45, 0.45, 0.10), \quad C = (0.20, 0.20, 0.60).$$

Kết quả kiểm thử:

$$\text{Acc}(A) = 0.83, \quad \text{Acc}(B) = 1.0, \quad \text{Acc}(C) = 0.67.$$

Fitness  $F = 1 - \text{Acc}$ , do đó  $B$  là leader.

### 2.2.7 Cập nhật

Các cá thể  $A, C$  sẽ dịch chuyển về phía  $B$  theo công thức TSA. Sau khi cập nhật, trọng số được *chuẩn hoá* lại để tổng bằng 1.

### 2.2.8 So sánh với thuật toán khác

- **PSO:** dựa vào vận tốc, nhiều tham số (inertia, cognitive, social).
- **GA:** dùng lai ghép và đột biến, quản lý quần thể phức tạp.
- **Bayesian Optimization:** hiệu quả với số chiều nhỏ, nhưng tốn kém khi nhiều biến.
- **TSA:** công thức đơn giản, ít tham số, phù hợp với các bài toán ràng buộc (ensemble weights).

### 2.2.9 Ứng dụng trong AI Engineering

- **Ensemble learning:** chọn trọng số tối ưu cho nhiều mô hình.
- **Hyperparameter tuning:** tối ưu learning rate, batch size, regularization.
- **Feature selection:** mã hoá nhị phân chọn/bỏ đặc trưng.
- **Neural Architecture Search:** tìm số lớp, số filter xấp xỉ tối ưu.

### 2.2.10 Pseudocode (Python style)

```

1 # Tuncate Swarm Algorithm (TSA)
2 initialize population X
3 evaluate fitness F(X)
4 while not stop:
5     X_best = best solution
6     for each agent X_i:
7         c1, c2, c3 = random()
8         G = c2 + c3 - F(X_i)
9         M = P_min + c1*(P_max - P_min)
10        A = G * M
11        X_i = X_best - A*(X_best - X_i)
12        normalize(X_i)
13    evaluate F(X)
14 return X_best

```

### 2.2.11 Biến thể nâng cao

Một số nghiên cứu mở rộng TSA:

- **Chaotic TSA:** thêm chuỗi hỗn loạn để tăng khả năng tìm kiếm.
- **Opposition-based TSA:** khởi tạo cả nghiệm và nghiệm đối xứng.
- **Adaptive TSA:** thay đổi  $c_1, c_2, c_3$  theo từng vòng lặp.

### 2.2.12 Bài toán ví dụ

**Mục tiêu và bối cảnh** Chúng ta tối ưu trọng số ensemble cho ba mô hình (RF, XGB, CNN) bằng meta-heuristic **Tuncate Swarm Algorithm (TSA)**. Với dữ liệu toy (6 mẫu), ta muốn tìm  $w = (w_{RF}, w_{XGB}, w_{CNN})$  sao cho dự đoán tốt hơn.

#### Chiến lược trong mã dưới đây

- **Fitness mặc định:** log loss (nhạy hơn Accuracy, thay đổi mượt).
- **Tùy chọn:** tối ưu cả  $\tau$  (threshold) bằng cờ OPTIMIZE\_THRESHOLD.
- **Trực quan hóa:** vẽ đường hội tụ fitness và quỹ đạo trọng số tốt nhất.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.metrics import log_loss, roc_auc_score
4
5 OPTIMIZE_THRESHOLD = False
6 SEED = 0
7 ITERS = 30
8 NOISE_SCALE = 0.03
9
10 probs = np.array([
11     [0.55, 0.60, 0.80], # y=1
12     [0.40, 0.65, 0.70], # y=0
13     [0.70, 0.55, 0.74], # y=1

```

```
14     [0.35, 0.45, 0.50], # y=0
15     [0.69, 0.65, 0.85], # y=1
16     [0.43, 0.30, 0.55] # y=0
17 ], dtype=float)
18 y_true = np.array([1,0,1,0,1,0], dtype=int)
19
20 def softmax(w):
21     w = w - w.max()
22     e = np.exp(w)
23     return e / e.sum()
24
25 def sigmoid(x):
26     return 1 / (1 + np.exp(-x))
27
28 def w_to_raw(w_target):
29     w_target = np.array(w_target, dtype=float)
30     w_target = w_target / w_target.sum()
31     return np.log(w_target + 1e-12)
32
33 def ens_scores_from_raw(raw):
34     if OPTIMIZE_THRESHOLD:
35         w_raw, tau_raw = raw[:3], raw[3]
36         w = softmax(w_raw)
37         tau = sigmoid(tau_raw) # (0,1)
38     else:
39         w = softmax(raw)
40         tau = 0.5
41     scores = probs @ w
42     return scores, w, tau
43
44 def accuracy_from_raw(raw):
45     scores, w, tau = ens_scores_from_raw(raw)
46     y_pred = (scores >= tau).astype(int)
47     return (y_pred == y_true).mean()
48
49 def auc_from_raw(raw):
50     scores, _, _ = ens_scores_from_raw(raw)
51     return roc_auc_score(y_true, scores)
52
53 def logloss_from_raw(raw):
54     scores, _, _ = ens_scores_from_raw(raw)
55     scores = np.clip(scores, 1e-9, 1-1e-9)
56     return log_loss(y_true, scores)
57
58 # oChn Fitness
59 def fitness(raw):
60     return logloss_from_raw(raw)
61     # return 1.0 - accuracy_from_raw(raw) # If Fitness is Accuracy
62     # return -auc_from_raw(raw) # If Fitness is AUC
63
64 A_raw = w_to_raw([1/3, 1/3, 1/3])
65 B_raw = w_to_raw([0.45, 0.45, 0.10])
66 C_raw = w_to_raw([0.20, 0.20, 0.60])
67
68 if OPTIMIZE_THRESHOLD:
```

```

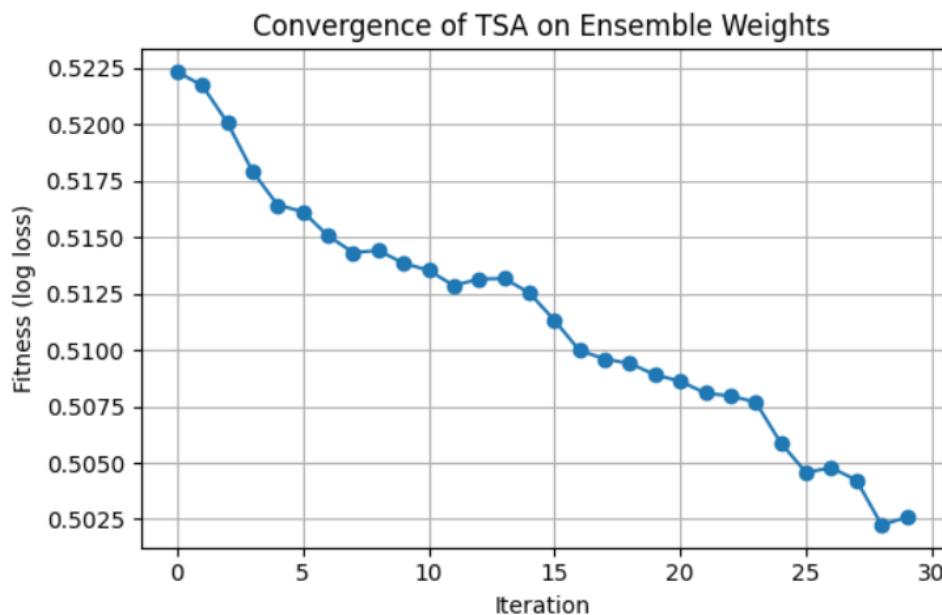
69 A_raw = np.r_[A_raw, 0.0]
70 B_raw = np.r_[B_raw, 0.0]
71 C_raw = np.r_[C_raw, 0.0]
72
73 print("== Khi àto ==")
74 print("Acc A:", round(accuracy_from_raw(A_raw), 3), "LogLoss A:", round(
    logloss_from_raw(A_raw), 4))
75 print("Acc B:", round(accuracy_from_raw(B_raw), 3), "LogLoss B:", round(
    logloss_from_raw(B_raw), 4))
76 print("Acc C:", round(accuracy_from_raw(C_raw), 3), "LogLoss C:", round(
    logloss_from_raw(C_raw), 4))
77
78 # 1 TSA round
79 rng = np.random.default_rng(SEED)
80
81 def one_tsa_iteration(pop_raw):
82     fits = [fitness(w) for w in pop_raw]
83     best_idx = int(np.argmin(fits))
84     Wbest = pop_raw[best_idx].copy()
85     scores, wbest, tau_best = ens_scores_from_raw(Wbest)
86     print("\n== ONE TSA ITERATION (manual) ==")
87     print("Best index:", best_idx, "Fitness(best):", round(fits[best_idx], 6))
88     print("Best weights:", np.round(wbest, 3), "tau:", round(tau_best, 3))
89     new_pop = []
90     for i, Wi in enumerate(pop_raw):
91         Fi = fits[i]
92         c1, c2, c3 = rng.random(3) # U(0,1)
93         G = c2 + c3 - Fi
94         M = c1
95         A = G * M
96         print(f"\n-- Agent {i} --")
97         print(" c1,c2,c3:", np.round([c1, c2, c3], 4))
98         print(" Fi:", round(Fi, 6), "> G:", round(G, 6), " M:", round(M, 6), " =>
A:", round(A, 6))
99         # update
100        Wi_new = Wbest - A * (Wbest - Wi)
101        # small noise
102        Wi_new += NOISE_SCALE * rng.normal(size=Wi.shape)
103        # in ékt åqu
104        acc_i = accuracy_from_raw(Wi_new)
105        fit_i = fitness(Wi_new)
106        scores_i, wi, tau_i = ens_scores_from_raw(Wi_new)
107        print(" weights:", np.round(wi, 3), "tau:", round(tau_i, 3))
108        print(" Acc:", round(acc_i, 3), "Fitness:", round(fit_i, 6))
109        new_pop.append(Wi_new)
110    return new_pop
111
112 population = [A_raw.copy(), B_raw.copy(), C_raw.copy()]
113 population = one_tsa_iteration(population)
114
115 # Run TSA
116 def run_tsa(pop_raw, iters=ITERS, noise_scale=NOISE_SCALE, seed=SEED):
117     rng = np.random.default_rng(seed)
118     pop = [w.copy() for w in pop_raw]
119     fit_hist = []

```

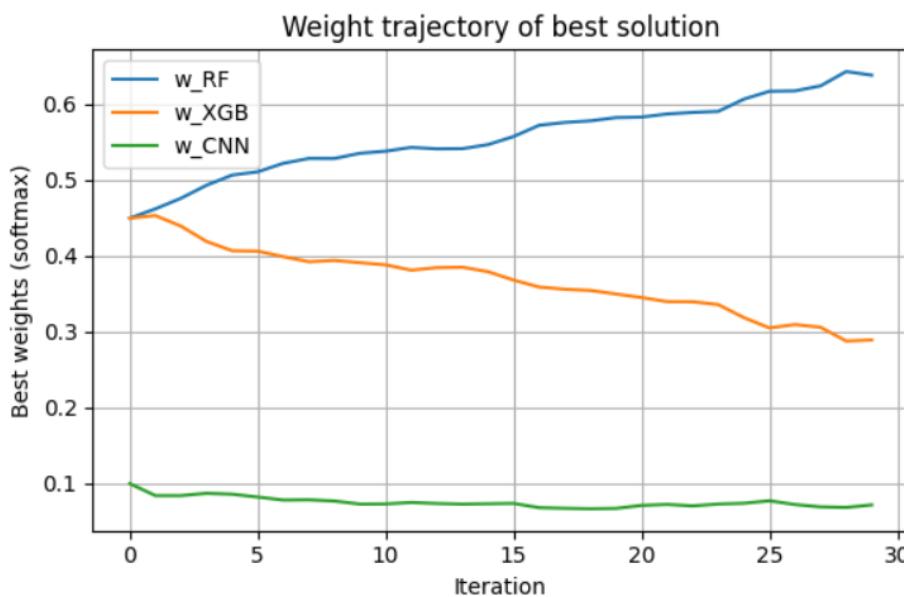
```
120 best_hist = []
121 for t in range(iters):
122     fits = np.array([fitness(w) for w in pop])
123     best_idx = int(np.argmin(fits))
124     Wbest = pop[best_idx].copy()
125     fit_hist.append(fits.min())
126     best_hist.append(Wbest)
127
128 new_pop = []
129 for Wi, Fi in zip(pop, fits):
130     c1, c2, c3 = rng.random(3)
131     G = c2 + c3 - Fi
132     M = c1
133     A = G * M
134     Wi_new = Wbest - A * (Wbest - Wi)
135     Wi_new += noise_scale * rng.normal(size=Wi.shape)
136     new_pop.append(Wi_new)
137 pop = new_pop
138
139 fits = np.array([fitness(w) for w in pop])
140 best_idx = int(np.argmin(fits))
141 return pop[best_idx], fit_hist, best_hist
142
143 best_raw, fit_hist, best_hist = run_tsa([A_raw, B_raw, C_raw], iters=ITERS)
144
145 scores_b, w_best, tau_best = ens_scores_from_raw(best_raw)
146 print("\n== Final result ==")
147 print("Best weights:", np.round(w_best, 3), " tau:", round(tau_best, 3))
148 print("Final accuracy:", round(accuracy_from_raw(best_raw), 3))
149 print("Final logloss :", round(logloss_from_raw(best_raw), 6))
150 print("Final AUC      :", round(auc_from_raw(best_raw), 3))
151
152 plt.figure(figsize=(6,4))
153 plt.plot(fit_hist, marker="o")
154 plt.xlabel("Iteration")
155 plt.ylabel("Fitness (log loss)")
156 plt.title("Convergence of TSA on Ensemble Weights")
157 plt.grid(True)
158 plt.tight_layout()
159 plt.show()
160
161 # Visualization
162 W_path = np.array([softmax(b[:3]) if not OPTIMIZE_THRESHOLD else softmax(b[:3])
163                   for b in best_hist])
164 plt.figure(figsize=(6,4))
165 plt.plot(W_path[:,0], label="w_RF")
166 plt.plot(W_path[:,1], label="w_XGB")
167 plt.plot(W_path[:,2], label="w_CNN")
168 plt.legend()
169 plt.xlabel("Iteration")
170 plt.ylabel("Best weights (softmax)")
171 plt.title("Weight trajectory of best solution")
172 plt.grid(True)
173 plt.tight_layout()
```

```
173 plt.show()
```

Code Listing 1: TSA tối ưu trọng số ensemble (có trực quan hoá)



Hình 3: TSA Fitness Log



Hình 4: Weight trajectory of best solution

### Các thành phần chính

- **Fitness = log loss:** nhạy với xác suất nên thay đổi mượt dù nhãn chưa đổi.

- **Tuỳ chọn threshold:** bật `OPTIMIZE_THRESHOLD` để TSA tối ưu thêm  $\tau \in (0, 1)$  (dùng `sigmoid` ánh xạ).
- **Cập nhật TSA:**  $X^{t+1} = X_{\text{best}} - A \cdot (X_{\text{best}} - X^t)$  với  $A = (c_2 + c_3 - F) \cdot c_1$ . Có thêm nhiều nhỏ để khám phá.
- **Ràng buộc tổng bằng 1:** dùng `softmax` để bảo đảm  $w_i \geq 0, \sum w_i = 1$ .
- **Trực quan hóa:** biểu đồ *fitness* theo vòng lặp (hội tụ), và *quỹ đạo trọng số tốt nhất* để thấy mức ảnh hưởng của từng mô hình.

### 2.3 Giới thiệu Stacking Model

Stacking là một trong những kỹ thuật ensemble phổ biến, được xây dựng theo nguyên tắc *meta-learning*. Trong bước này:

- Các mô hình cơ sở (RF, XGB, LGBM, CatBoost, ...) được huấn luyện trên tập train và tạo ra dự đoán xác suất trên tập validation (OOF – Out-of-Fold prediction).
- Các dự đoán OOF này được gom lại thành ma trận đặc trưng mới  $P$ , trong đó mỗi cột đại diện cho một mô hình cơ sở, mỗi dòng là xác suất dự đoán cho một mẫu.
- Meta-model (ở đây là Logistic Regression) sẽ được huấn luyện trên ma trận  $P$  và nhãn thật  $y$ . Quá trình này giúp LR học được trọng số tối ưu cho từng base model.

Khác với việc gán trọng số bằng nhau, Stacking cho phép hệ thống học “ngầm” rằng mô hình nào mạnh hơn sẽ có hệ số lớn hơn, mô hình nào yếu hơn sẽ bị giảm ảnh hưởng. Đồng thời, Logistic Regression cung cấp khả năng giải thích trực quan thông qua hệ số (coef), giúp xác định vai trò của từng base model trong tổ hợp.

### 2.4 Các hàm triển khai

Trong phần triển khai, nhóm xây dựng một pipeline để so sánh ba phương pháp ensemble: *Equal Weights*, *Stacking-LR*, và *Stack-TSA*. Các hàm dưới đây thể hiện cách mà hệ thống được xây dựng, huấn luyện và đánh giá.

**Base Models.** Nhóm sử dụng bốn mô hình cây làm base learners: *Random Forest*, *CatBoost*, *XGBoost*, và *LightGBM*. Việc kết hợp đa dạng các thuật toán giúp giảm variance và tăng độ tổng quát hoá.

```

1 base_models = [
2     ("rf", RandomForestClassifier(
3         n_estimators=300, max_depth=5,
4         min_samples_split=2, min_samples_leaf=1,
5         max_features='sqrt', bootstrap=True,
6         random_state=42, class_weight=None)),
7
8     ("cat", CatBoostClassifier(
9         iterations=1000, learning_rate=0.03, depth=6,
10        l2_leaf_reg=3.0, rsm=0.8, bootstrap_type="Bayesian",
11        bagging_temperature=0.5, loss_function="Logloss",
12        eval_metric="AUC", auto_class_weights="Balanced",
13        random_state=42, verbose=0, allow_writing_files=False,
14        thread_count=-1)),
15

```

```

16     ("xgb", XGBClassifier(
17         n_estimators=400, max_depth=5, learning_rate=0.1,
18         subsample=1.0, colsample_bytree=1.0, reg_lambda=1.0,
19         eval_metric="logloss", random_state=42)),
20
21     ("lgbm", LGBMClassifier(
22         n_estimators=500, max_depth=4, learning_rate=0.05,
23         subsample=0.9, colsample_bytree=0.8, reg_lambda=1.0,
24         random_state=42)),
25 ]

```

Các tham số sử dụng được tham khảo từ các bài reading của AIO mỗi tuần. Vì thời gian có hạn, nên nhóm sử dụng những tham số được khuyên dùng tương ứng với từng mô hình.

**Huấn luyện OOF và Refit.** Hàm `train_bases_oof_and_refit` sinh ra OOF (out-of-fold predictions) cho tập train, sau đó refit toàn bộ base models để suy ra xác suất trên validation và test. Điều này đảm bảo không rò rỉ dữ liệu và cho phép stacking hoạt động đúng.

```

1 def train_bases_oof_and_refit(
2     X_train, y_train, X_val, X_test,
3     n_splits=5, calibrate=True,
4     calib_method="sigmoid", calib_cv=3,
5     random_state=42
6 ):
7     skf = StratifiedKFold(
8         n_splits=n_splits, shuffle=True,
9         random_state=random_state)
10    n_models = len(base_models)
11
12    P_train_oof = np.zeros((len(X_train), n_models))
13    P_val        = np.zeros((len(X_val), n_models))
14    P_test       = np.zeros((len(X_test), n_models))
15
16    fitted_bases = []
17
18    for j, (_, base) in enumerate(base_models):
19        oof_col = np.zeros(len(X_train))
20        for tr_idx, va_idx in skf.split(X_train, y_train):
21            X_tr, y_tr = X_train.iloc[tr_idx], y_train.iloc[tr_idx]
22            X_va      = X_train.iloc[va_idx]
23
24            if calibrate:
25                model = CalibratedClassifierCV(
26                    clone(base), method=calib_method, cv=calib_cv)
27            else:
28                model = clone(base)
29
30            model.fit(X_tr, y_tr)
31            oof_col[va_idx] = model.predict_proba(X_va)[:, 1]
32
33    P_train_oof[:, j] = oof_col
34
35    if calibrate:
36        model_full = CalibratedClassifierCV(
37            clone(base), method=calib_method, cv=calib_cv)
38    else:

```

```

39     model_full = clone(base)
40
41     model_full.fit(X_train, y_train)
42     P_val[:, j] = model_full.predict_proba(X_val)[:, 1]
43     P_test[:, j] = model_full.predict_proba(X_test)[:, 1]
44     fitted_bases.append(model_full)
45
46 return P_train_oof, P_val, P_test, fitted_bases

```

**Equal Weights.** Baseline ensemble: trung bình xác suất của tất cả base models. Baseline này được dùng để so sánh với hai mô hình stacking - meta-model và stacking - TSA

```

1 def equal_weight_probs(P):
2     return P.mean(axis=1)

```

**Stacking với Logistic Regression.** Stacking-LR dùng LR làm meta-model để học cách kết hợp tuyến tính các base models.

```

1 def stack_lr_fit(P_train_oof, y_train):
2     meta = LogisticRegression(
3         C=1.0, penalty="l2", max_iter=5000, solver="lbfgs")
4     meta.fit(P_train_oof, y_train)
5     return meta

```

**TSA (Tunica Swarm Algorithm).** TSA là một thuật toán meta-heuristic được sử dụng để tối ưu hoá trọng số  $w$  bằng cách minimize hàm mục tiêu  $(1 - AUC)$ . Do đặc trưng của TSA là không dựa vào gradient để tối ưu hàm loss, ta cần định nghĩa rõ ràng một hàm mục tiêu `obj_func`. Vì TSA chỉ hỗ trợ bài toán *minimization*, nên việc dùng  $(1 - AUC)$  là phù hợp (thay vì tối đa hóa AUC trực tiếp).

Để bảo đảm nghiệm hợp lệ, các trọng số  $w$  được ràng buộc phải không âm và có tổng bằng 1. Điều này đồng nghĩa với việc sau khi tìm ra nghiệm tối ưu, ta cần chuẩn hoá vector  $w$ . Bên cạnh đó, ta thiết lập không gian tìm kiếm bằng cách đặt các giới hạn biên (`bounds`) cho từng trọng số, thường là  $[0, 1]$ .

Sau khi thiết lập bài toán, ta truyền các tham số vào bộ giải `TSA.OriginalTSA`. Thuật toán sẽ tiến hoá quần thể nghiệm và trả về nghiệm tốt nhất trong `best.solution`. Vector  $w$  này sau đó được chuẩn hoá lại và sử dụng để kết hợp các xác suất dự đoán từ các base models, từ đó tạo ra dự đoán ensemble cuối cùng.

```

1 def stack_tsa_predict_auc(P_val, y_val, P_test,
                           epoch=250, pop_size=40, seed=42):
2
3     def obj_func(w):
4         w = np.maximum(w, 0.0)
5         s = w.sum()
6         if s <= 0: return 1e9 #phat nang tong weights khi no la so am
7         w = w / s
8         p = P_val @ w
9         if (p.max() - p.min()) < 1e-12: return 1.0
10        auc = roc_auc_score(y_val, p)
11        return 1.0 - auc
12
13    n_models = P_val.shape[1]
14    bounds = [FloatVar(lb=0.0, ub=1.0) for _ in range(n_models)]
15    problem = {"obj_func": obj_func, "bounds": bounds, "minmax": "min"}
16    algo = TSA.OriginalTSA(epoch=epoch, pop_size=pop_size, seed=seed)
17    best = algo.solve(problem)

```

```

18     w = np.maximum(np.asarray(best.solution, float), 0.0)
19     w = w / (w.sum() + 1e-12)
20     p_val, p_test = P_val @ w, P_test @ w
21
22     return p_val, p_test, w

```

**Đánh giá và In kết quả.** Hàm `evaluate_version` so sánh ba phương pháp ensemble trên từng biến thể dữ liệu. Kết quả được in ra bởi `pretty_print`.

```

1 def metrics_report(y_true, p):
2     return {
3         "AUC": roc_auc_score(y_true, p),
4         "LogLoss": log_loss(y_true, p),
5         "AP": average_precision_score(y_true, p),
6     }
7
8 def evaluate_version(X_train, y_train, X_val, y_val, X_test, y_test):
9     P_train_oof, P_val, P_test, _ = train_bases_oof_and_refit(
10         X_train, y_train, X_val, X_test)
11
12     p_eq_test = equal_weight_probs(P_test)
13     meta = stack_lr_fit(P_train_oof, y_train)
14     p_stack_test = meta.predict_proba(P_val)[:, 1]
15     p_tsa_val, p_tsa_test, w_tsa = stack_tsa_predict_auc(P_val, y_val, P_test)
16
17     return [
18         {"name": "EqualWeights", **metrics_report(y_test, p_eq_test)},
19         {"name": "Stack_LR", **metrics_report(y_test, p_stack_test)},
20         {"name": "Stack_TSA", **metrics_report(y_test, p_tsa_test)},
21     ], {"w_tsa": w_tsa}
22
23 def pretty_print(title, results):
24     print(f"\n== {title} ==")
25     for r in results:
26         print(f" {r['name']}:{>12} | AUC={r['AUC']:.4f} | "
27               f"LogLoss={r['LogLoss']:.4f} | AP={r['AP']:.4f}")

```

Ở đây, ta quan tâm đến các thước đo như AUC, LogLoss, AP,... thay vì *accuracy*, vì trong các bài toán y tế việc cân bằng giữa các lớp và tối ưu hóa hiệu quả chẩn đoán quan trọng hơn so với chỉ số chính xác thô.

Cụ thể, **AUC** (Area Under the Curve) giúp đánh giá khả năng phân biệt bệnh/không bệnh trên toàn bộ các ngưỡng dự đoán, thay vì chỉ cố định ở một ngưỡng 0.5. **LogLoss** đo lường mức độ "tự tin" của mô hình trong dự đoán xác suất, một khía cạnh quan trọng trong y tế vì sai số dự đoán với xác suất cao (ví dụ dự đoán chắc chắn nhưng lại sai) thường nguy hiểm hơn. Trong khi đó, **AP** (Average Precision) phản ánh hiệu quả mô hình trong việc tìm kiếm và phát hiện các ca dương tính hiếm gặp — vốn là tình huống phổ biến trong dữ liệu y sinh học với hiện tượng *class imbalance*.

Do đó, việc tập trung vào những metric này giúp đảm bảo mô hình không chỉ đúng nhiều trường hợp, mà còn cung cấp xác suất dự đoán đáng tin cậy, có khả năng hỗ trợ bác sĩ đưa ra quyết định lâm sàng một cách an toàn hơn.

## 2.5 Kết quả

Bảng 1: So sánh hiệu suất giữa Equal Weights, Stacking (LR) và TSA trên 4 biến thể dữ liệu

Dataset	Model	AUC	LogLoss	AP
Original	EqualWeights	<b>0.9412</b>	0.3689	<b>0.9313</b>
	Stack_LR	0.9370	<b>0.3529</b>	0.9289
	Stack_TSA	0.9328	0.3713	0.9251
Ori + FE	EqualWeights	0.9202	0.3965	0.9042
	Stack_LR	<b>0.9244</b>	<b>0.3778</b>	<b>0.9066</b>
	Stack_TSA	0.9034	0.4104	0.8813
Ori + DT	EqualWeights	<b>0.9118</b>	0.4121	0.8857
	Stack_LR	<b>0.9118</b>	<b>0.4031</b>	<b>0.8984</b>
	Stack_TSA	0.9034	0.4137	0.8782
FE + DT	EqualWeights	0.9244	0.3969	0.9179
	Stack_LR	<b>0.9286</b>	<b>0.3835</b>	<b>0.9200</b>
	Stack_TSA	0.9244	0.3996	0.9179

- **Original dataset:** Equal Weights đạt AUC cao nhất (0.9412), chứng tỏ trong dữ liệu gốc việc kết hợp đơn giản (trọng số bằng nhau) đã mang lại hiệu quả rất tốt. Stacking LR có LogLoss thấp hơn (0.3529), tức dự đoán xác suất “tự tin” hơn. TSA kém hơn một chút, cho thấy tối ưu meta-heuristic chưa tạo lợi thế trong kịch bản này.
- **Ori + FE (feature engineering):** Stacking LR vượt trội hơn Equal Weights với AUC = 0.9244 và LogLoss = 0.3778. TSA lại suy giảm đáng kể (AUC = 0.9034), chứng tỏ tối ưu theo AUC trên tập validation chưa tổng quát hóa sang test.
- **Ori + DT (feature selection bằng Decision Tree):** Equal Weights và Stacking LR ngang nhau (AUC = 0.9118), trong khi TSA thấp hơn (0.9034). Điều này cho thấy khi số lượng đặc trưng bị giới hạn, lợi ích từ tối ưu hóa meta-heuristic bị hạn chế.
- **FE + DT (feature engineering + feature selection):** Đây là kịch bản tốt nhất cho Stacking LR, với AUC cao nhất toàn bộ (0.9286). Equal Weights và TSA gần như tương đương (0.9244). TSA không vượt trội, nhưng cũng duy trì kết quả ổn định.

Tóm lại:

1. Equal Weights là một baseline đơn giản nhưng mạnh, đặc biệt trên dữ liệu gốc.
2. Stacking LR thường mang lại LogLoss tốt hơn, và khi có thêm FE + DT thì đạt hiệu quả cao nhất.
3. TSA linh hoạt nhưng chưa ổn định, đôi khi không vượt qua baseline vì dễ bị overfit vào tập validation khi tối ưu trọng số.

Tóm lại, trong thí nghiệm này: **Stacking LR** là phương pháp ổn định và hiệu quả nhất; **Equal Weights** là baseline đáng tin cậy; còn **TSA** cần thêm các kỹ thuật như regularization hoặc bagging trên validation để tránh overfitting và phát huy sức mạnh.

## 2.6 Explainable AI

Trong phần này, nhóm tập trung vào việc giải thích cách các mô hình thành phần (base models) đóng góp vào quyết định cuối cùng của ensemble. Hai phương pháp được sử dụng song song là **Stacking-LR** và **TSA**, nhằm cung cấp cái nhìn ở mức *toàn cục* (global importance) và *cục bộ* (local contributions).

Ta thiết kế một hàm explain cho phương thức TSA. Hàm này nhận trọng số tối ưu  $w$  từ TSA và ma trận xác suất test ( $P_{\text{test}}$ ):

- Trước hết  $w$  được chuẩn hóa để đảm bảo tổng bằng 1. Điều này nhằm giữ ý nghĩa ensemble: xác suất dự đoán cuối cùng là tổ hợp lồi của các xác suất base models, không vượt ngoài khoảng [0, 1].
- Phần **global** vẽ bar chart các trọng số  $w$ , cho thấy mô hình nào chiếm ưu thế trong ensemble. Những base model có weight cao sẽ đóng vai trò chủ đạo trong dự đoán, trong khi các mô hình bị gán trọng số gần 0 coi như ít hoặc không đóng góp.
- Phần **local** tính đóng góp của từng base model cho một mẫu cụ thể. Cụ thể:
  - $p_{\text{row}}$  lấy vector xác suất của tất cả base models cho mẫu này.
  - $\text{contrib\_raw} = w * p_{\text{row}}$  chính là mức đóng góp tuyệt đối của từng base model sau khi nhân với trọng số đã tối ưu.

Nếu `normalize_baseline=True`, các đóng góp được chuẩn hóa bằng cách so sánh với baseline — tức giá trị trung bình có trọng số trên toàn bộ test. Điều này nhằm mục đích làm nổi bật phần *chênh lệch* mà mỗi mô hình đã tạo ra so với dự đoán điển hình, từ đó thấy rõ “ai” chịu trách nhiệm chính trong việc kéo mẫu cụ thể sang phía “nguy cơ bệnh” hoặc “không bệnh”.

- Kết quả của hàm gồm:
  1. Bảng trọng số toàn cục (global weight table) cho biết mức ảnh hưởng tổng thể của các base models.
  2. Bảng đóng góp cục bộ (local contributions table) thể hiện mức đóng góp của từng base model cho một bệnh nhân cụ thể.
  3. Xác suất dự đoán ensemble  $p_{\text{ens}}$  của mẫu đó, được tính bằng tổng các đóng góp.

```

1 def explain_tsa(P_test, w, base_names, sample_idx=0, normalize_baseline=True):
2     w = np.asarray(w, dtype=float).ravel()
3     if not np.isclose(w.sum(), 1.0, atol=1e-6):
4         w = w / (w.sum() + 1e-12)
5
6     P_test = np.asarray(P_test, dtype=float)
7     base_names = list(base_names)
8
9     df_w = (pd.DataFrame({"base_model": base_names, "weight": w})
10            .sort_values("weight", ascending=False))
11
12    plt.figure(figsize=(6,3))
13    plt.bar(df_w["base_model"], df_w["weight"])
14    plt.title("TSA global weight distribution")
15    plt.tight_layout(); plt.show()
16
17    p_row = P_test[int(sample_idx), :]
18    contrib_raw = w * p_row
19    p_ens = float(contrib_raw.sum())

```

```

20
21     if normalize_baseline:
22         p_base_mean = P_test.mean(axis=0)
23         baseline = float((w * p_base_mean).sum())
24         contrib = contrib_raw - (w * p_base_mean)
25         title = f"Local TSA contributions (sample {sample_idx})..."
26     else:
27         contrib = contrib_raw
28         title = f"Local TSA contributions (sample {sample_idx})..."
29
30     df_local = pd.DataFrame({"base_model": base_names, "contribution": contrib})
31
32     plt.figure(figsize=(7, max(3, 0.4*len(w))))
33     plt.axvline(x=0.0, ls="--", lw=1)
34     plt.barch(df_local["base_model"], df_local["contribution"])
35     plt.title(title)
36     plt.tight_layout(); plt.show()
37
38     return {"weights_global": df_w,
39             "local_contributions": df_local,
40             "p_ens_sample": p_ens}

```

Ta cũng làm tương tự với phương thức stacking - meta-model, huấn luyện một Logistic Regression làm meta-model trên ma trận xác suất OOF (`P_train_oof`). Sau khi huấn luyện:

- Các hệ số `coef` được lấy ra để đo mức độ quan trọng toàn cục của từng base model. Bar chart biểu diễn giá trị tuyệt đối của các hệ số này.
- SHAP được áp dụng trên Logistic Regression, giúp tạo ra các biểu đồ beeswarm và waterfall. Beeswarm cho biết ảnh hưởng toàn cục của từng base model, còn waterfall giải thích cụ thể một ca bệnh (sample\_idx).
- Kết quả trả về gồm meta-model, bảng hệ số, explainer và shap values để dùng cho visualization.

```

1 def explain_stacking_lr(P_train_oof, y_train, P_test, base_names, sample_idx=0,
2                         max_display=12):
3     meta = LogisticRegression(C=1.0, penalty="l2", max_iter=5000, solver="lbfgs")
4     meta.fit(P_train_oof, y_train)
5
6     coef = meta.coef_.ravel()
7     imp = np.abs(coef)
8     imp_df = (pd.DataFrame({"base_model": base_names, "abs_coef": imp, "coef": coef})
9               .sort_values("abs_coef", ascending=False)
10              .reset_index(drop=True))
11
12     plt.figure(figsize=(6,3))
13     plt.bar(imp_df["base_model"].head(max_display), imp_df["abs_coef"].head(max_display))
14     plt.title("Stacking-LR | |coef| (global importance)")
15     plt.tight_layout(); plt.show()
16
17     masker = shap.maskers.Independent(P_train_oof)
18     explainer = shap.LinearExplainer(
19         model=meta,
20         masker=masker,
21     )

```

```

20     feature_perturbation="interventional",
21     link= shap.links.logit
22 )
23 sv = explainer(P_test)
24 sv.feature_names = list(base_names)
25
26 shap.plots.beeswarm(sv, max_display=min(max_display, len(base_names)), show=True)
27 shap.plots.waterfall(sv[sample_idx], max_display=max_display, show=True)
28
29 return {"meta": meta, "importance_df": imp_df, "explainer": explainer, "shap_values": sv}

```

Cuối cùng, ta tạo một hàm wrapper để gọi các hàm đã được định nghĩa:

- Gọi lại `train_bases_oof_and_refit` để thu được ma trận xác suất OOF, VAL và TEST từ các base models.
- Nếu chọn `explain_model="stack"` gọi `explain_stacking_lr`.
- Nếu chọn `explain_model="tsa"` gọi `explain_tsa`. Nếu không truyền `w_tsa` thì hàm sẽ tự tối ưu trên validation.

```

1 def explainable_ai_result(
2     X_train, y_train, X_val, y_val, X_test, *,
3     base_models, explain_model="stack", w_tsa=None, sample_idx=0
4 ):
5     P_train_oof, P_val, P_test, _ = train_bases_oof_and_refit(
6         X_train, y_train, X_val, X_test
7     )
8     base_names = [name for name, _ in base_models]
9
10    if explain_model == "stack":
11        return explain_stacking_lr(P_train_oof, y_train, P_test, base_names,
12                                    sample_idx)
13
14    elif explain_model == "tsa":
15        if w_tsa is None:
16            _, _, w_tsa = stack_tsa_predict_auc(P_val, y_val, P_test)
17        return explain_tsa(P_test, w_tsa, base_names, sample_idx)
18
19    else:
20        raise ValueError("explain_model phải là 'stack' hoặc 'tsa'")

```

Ví dụ chạy thử trên tập data gốc:

```

1 # XAI cho Stacking LR (ban Original)
2 xai_ori_stack = explainable_ai_result(
3     X_train, y_train, X_val, y_val, X_test,
4     base_models=base_models,
5     explain_model="stack",
6     sample_idx=0
7 )
8
9 # XAI cho TSA (dung w_tsa da toi uu san luc tinh evaluate_version )
10 xai_ori_tsa = explainable_ai_result(
11     X_train, y_train, X_val, y_val, X_test,

```

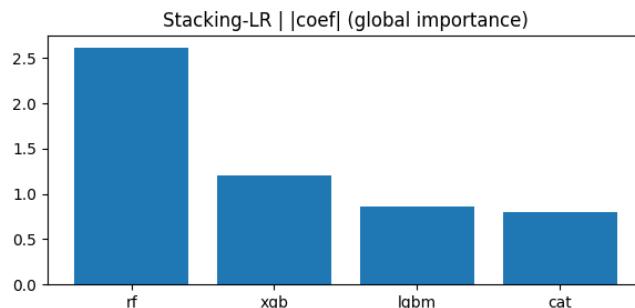
```

12 base_models=base_models,
13 explain_model="tsa",
14 w_tsa=ex_ori["w_tsa"],      # lay tu extras cua evaluate_version
15 sample_idx=0
16 )

```

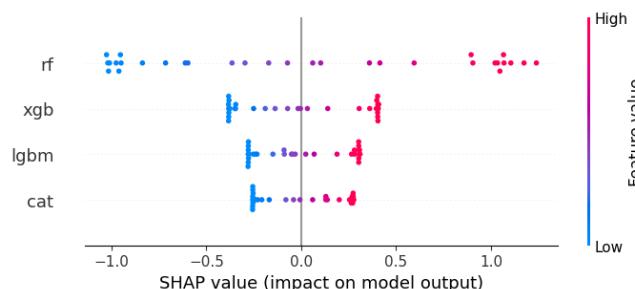
Ta thu được kết quả sau:

- Với phương pháp stacking - meta-model



Hình 5: Biểu đồ cột thể hiện mức độ đóng góp của từng mô hình

Trục x là  $|hệ số|$  của Logistic Regression khi đầu vào là vector xác suất từ các base models (rf, xgb, lgbm, cat). RF có  $|coef|$  lớn nhất  $\Rightarrow$  ở mức toàn cục, khi xác suất của RF tăng, log-odds dự đoán của ensemble thay đổi mạnh nhất. XGB kế tiếp, rồi LGBM  $\approx$  CatBoost. Đây là tầm ảnh hưởng toàn cục của từng mô hình trong ensemble, sau hiệu chỉnh xác suất (calibration), nên có thể so sánh được.

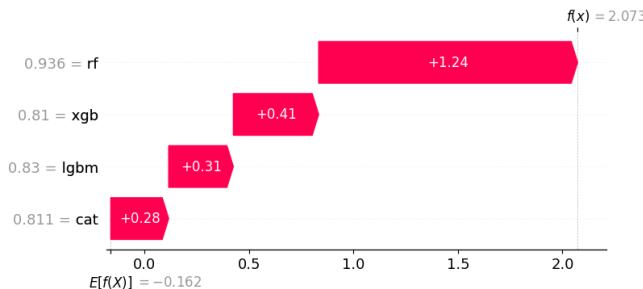


Hình 6: Beeswarm plot cho thấy đóng góp của từng mô hình khi kéo dự đoán ensemble về phía “bệnh” và “không bệnh”

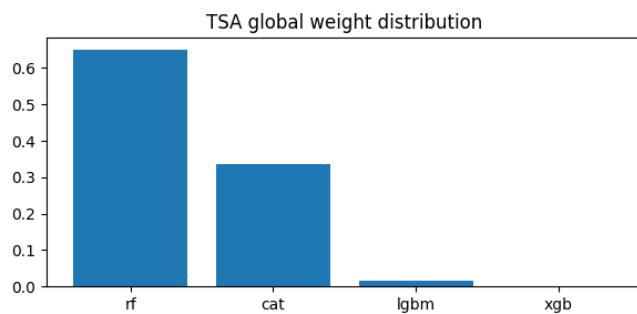
Trục hoành: SHAP value trên đầu ra của meta-LR (thang đo log-odds). Mỗi “feature” trong hình chính là xác suất dự đoán của một base model. Các điểm đỏ (giá trị “cao” của base prob) thường nằm phía dương  $\Rightarrow$  khi base model dự đoán xác suất cao, nó kéo ensemble về phía “bệnh”. Điểm xanh (xác suất thấp) kéo về phía “không bệnh”. Dải điểm của rf rộng và lệch phải hơn  $\Rightarrow$  đóng góp (dương/âm) của RF lên quyết định cuối cùng mạnh và thường xuyên hơn các base còn lại. Điều này nhất quán với bar  $|coef|$  ở trên.

$E[f(X)]$  là baseline log-odds trung bình của meta-model. Với một bệnh nhân cụ thể (sample\_idx), bốn “đặc trưng” (rf/xgb/lgbm/cat) đẩy log-odds tăng lần lượt: rf +1.24, xgb +0.41, ...  $\Rightarrow$  giải thích tại sao ca này bị ensemble xếp là “bệnh”.

- Với phương pháp stacking - TSA



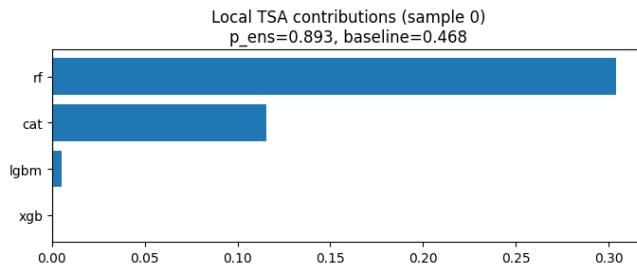
Hình 7: Waterfall plot minh họa chi tiết cách từng base model dẩy log-odds cho một bệnh nhân cụ thể



Hình 8: Biểu đồ cột thể hiện mức độ đóng góp của từng mô hình

**Trọng số toàn cục (global) của TSA:** Khi tối ưu AUC trên tập validation, TSA học ra một ensemble gần như “RF + CatBoost”, trong đó Random Forest chi phối quyết định. LGBM đóng góp rất nhỏ, XGBoost hầu như bị “tắt tiếng”.

Điễn giải: hoặc RF có khả năng phân biệt tốt nhất (AUC cao nhất/cân bằng bias-variance tốt nhất) trong các base đã hiệu chỉnh xác suất; hoặc đều ra giữa CatBoost–RF bổ sung nhau (không quá tương quan), còn LGBM/XGB trùng lắp và không thêm lợi ích nên bị dẩy trọng số xuống gần 0 khi tối ưu AUC.



Hình 9: Biểu đồ cột thể hiện chi tiết mức độ đóng góp của từng mô hình cho một bệnh nhân cụ thể

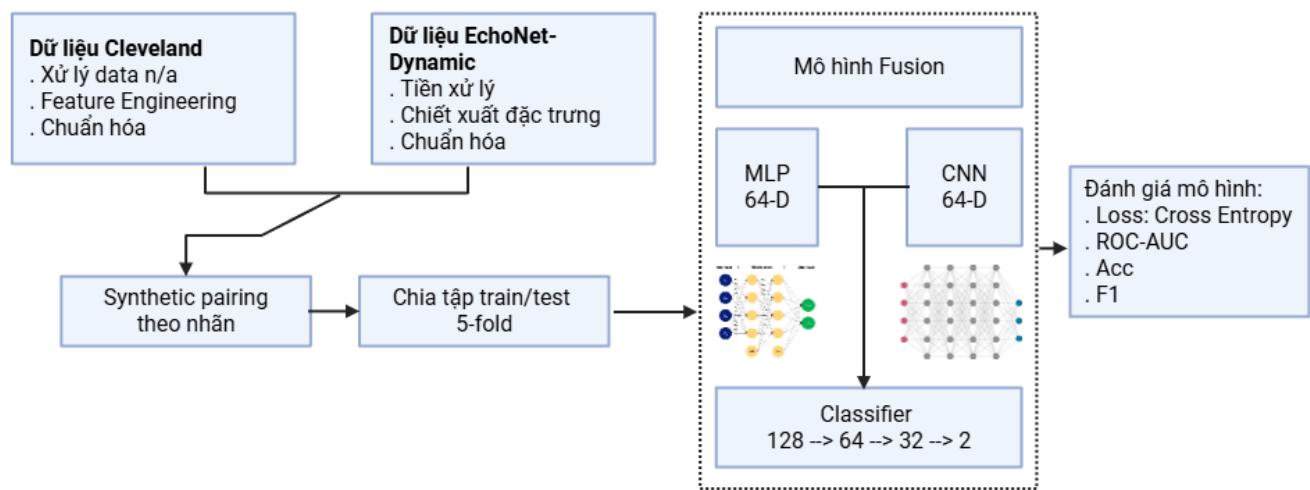
**Đóng góp cục bộ (local) cho một mẫu cụ thể:** Xét mẫu bệnh nhân đầu tiên:

Baseline = trung bình có trọng số (theo w) của xác suất các base trên toàn bộ test → “mốc” dự đoán điển hình của ensemble. p<sub>ens</sub> = xác suất dự đoán của mô hình ensemble của mẫu đó sau khi cộng các đóng góp. Trục hoành thể hiện mức độ đóng góp so với baseline. Cụ thể, Ở mẫu 0 này: RF kéo mạnh lên ( $\approx +0.30$ ), CatBoost kéo thêm ( $\approx +0.12$ ), LGBM rất nhỏ, XGB  $\approx 0 \rightarrow$  tổng cộng đẩy từ baseline 0.468 lên 0.893. Tóm lại, nếu bác sĩ muốn hiểu “ai đã đẩy ca bệnh này sang nguy cơ cao”, câu trả lời là RF là nguồn đóng góp chính, CatBoost hỗ trợ cùng chiều; hai

model còn lại không ảnh hưởng đáng kể.

### 3 Hướng mở rộng 2: CardioFusion: Ghép đa phương thức EchoNet + Cleveland cho phân loại bệnh tim

#### 3.1 Mô tả pipeline



Hình 10: Pipeline Ghép đa phương thức (Multimodal) EchoNet + Cleveland cho phân loại bệnh tim

##### 3.1.1 Tiền xử lý & Kỹ thuật đặc trưng (Cleveland)

###### Bước 1. Xử lý thiếu (NA)

- **Mặc định (ổn định)**: điền **median** theo cột số.
- **Tùy chọn (nâng cao)**: *Random Forest imputation/interpolation* cho từng cột thiếu bằng RF hồi quy/phân loại dựa trên các cột còn lại.

###### Bước 2. Feature Engineering

- **age\_group**: phân nhóm tuổi (< 40, 40–49, 50–59, 60–69, ≥ 70).
- **chol\_category**: < 200 (bình thường), 200–239 (ranh giới), ≥ 240 (cao).
- **bp\_category** (SBP): < 120 (N), 120–129 (Elev), 130–139 (S1), ≥ 140 (S2).
- **health\_index**: chỉ số tổng hợp từ tuổi/chol/BP. Ví dụ sau khi chuẩn hoá z-score:

$$\text{health\_index} = \alpha_1 z(\text{age}) + \alpha_2 z(\text{chol}) + \alpha_3 z(\text{trestbps}),$$

hoặc lấy PC<sub>1</sub> từ PCA(age,chol,trestbps).

- **cardiac\_risk**: kết hợp cp (đau ngực), thalach (nhịp tối đa), oldpeak (ST chênh). Ví dụ:

$$\text{cardiac\_risk} = \beta_1 \mathbb{1}(cp \in \{\text{typical, atypical}\}) + \beta_2 z(\text{thalach}) - \beta_3 z(\text{oldpeak}).$$

### Bước 3. Chuẩn hoá & Mã hoá

- One-hot/label-encode các biến phân loại.
- Chuẩn hoá số với **StandardScaler**:  $x' = (x - \mu)/\sigma$ .
- *Chú ý leakage*: fit imputer/scaler **chỉ trên train**, rồi áp lên val/test.

### Bước 4. Tiền xử lý & Trích xuất đặc trưng (EchoNet-Dynamic)

#### 3.2 Chuẩn hoá ảnh/video

- Lấy key-frame hoặc clip ngắn (32/64 frames), resize  $224 \times 224$ .
- Chuẩn hoá theo **ImageNet** nếu dùng backbone pretrained.
- Augmentation nhẹ: flip ngang, crop/center, chỉnh sáng/tương phản nhẹ.

### Bước 5. Backbone & Head đặc trưng ảnh

- **Backbone**: ResNet-50 *pretrained* (ImageNet).
- **Head mới**: bỏ avgpool+fc mặc định, thay bằng

AdaptiveAvgPool2d( $1 \times 1$ ) → Flatten → Linear ( $2048 \rightarrow 512$ ) → ReLU → Dropout → Linear ( $512 \rightarrow 64$ ).

- **Đầu ra nhánh ảnh**: vector **64-D**.

### Bước 6. Tạo Metadata

- Dựa meta qua MLP nhỏ in\_meta  $\rightarrow 32 \rightarrow 16 \rightarrow 8$ , concat với 64-D, rồi Linear về 64-D nếu cần.

### Bước 7. Mô hình Fusion hai nhánh

#### Nhánh tabular (Cleveland → MLP)

- Kiến trúc: in\_tab  $\rightarrow 128 \rightarrow \text{ReLU} \rightarrow \text{Dropout} \rightarrow 64 \rightarrow \text{ReLU}$ .
- Đầu ra: **64-D**.

#### Nhánh ảnh/meta (EchoNet → CNN)

- Theo mục trước, đầu ra: **64-D**.

#### Hợp nhất & Phân loại

- **Fusion**: concat([ $64_{\text{tab}}$ ,  $64_{\text{img}}$ ])  $\rightarrow 128$ .
- **Classifier**:  $128 \rightarrow 64 \rightarrow \text{ReLU} \rightarrow \text{Dropout} \rightarrow 32 \rightarrow \text{ReLU} \rightarrow 2$  (logits cho No disease / Heart disease).

### Bước 8. Synthetic Pairing theo nhãn

- Vì hai bộ *khác chủ thẻ*, tạo mini-batch bằng cách **ghép cặp theo nhãn**: chọn batch Cleveland nhãn  $y_c$ , batch EchoNet nhãn  $y_e$ , rồi bắt cặp ngẫu nhiên trong cùng nhãn ( $y_c = y_e$ ).
- Đảm bảo *stratified sampling* để giữ cân bằng lớp trong quá trình pairing.
- Cách này thay vì dùng CCA (đòi hỏi cùng nhóm chủ thẻ).

### Bước 9. Chia tập, Huấn luyện, Lịch học

#### 3.3 Chia tập

- **Stratified** train/val/test cho *mỗi nguồn*.
- Fit imputer/scaler **chỉ trên train**.

### Bước 10. Huấn luyện

- **Loss**: CrossEntropyLoss.
- **Tối ưu**: Adam (lr  $1e-3 \sim 3e-4$ ,  $L_2$   $1e-4 \sim 1e-5$ ).
- **Scheduler**: StepLR(step\_size =  $K$ ,  $\gamma = 0.1$ ) hoặc CosineAnnealingLR.
- **Epochs**: 30–100; **early-stop** theo val loss/AUROC.
- **Logging**: train/val loss, Accuracy, AUROC, F1, confusion matrix; lưu checkpoint tốt nhất theo *val AUROC*.

### Bước 11. K-fold

- 5-fold stratified riêng cho Cleveland và EchoNet; pairing theo fold; báo cáo mean $\pm$ std cho Acc/AUROC/F1.

### Bước 12. Đánh giá & Phân tích

- Trên test: Accuracy, AUROC, AUPRC (nếu mất cân bằng), F1, Sensitivity/Specificity.
- Calibration (tuỳ chọn): Brier score, reliability diagram.
- Ablation: chỉ tabular vs chỉ image vs fusion; median vs RF impute; có/không health\_index/cardiac\_risk.
- Giải thích: SHAP cho tabular; Grad-CAM cho ảnh.

#### 3.4 Giới thiệu tệp dữ liệu EchoNet

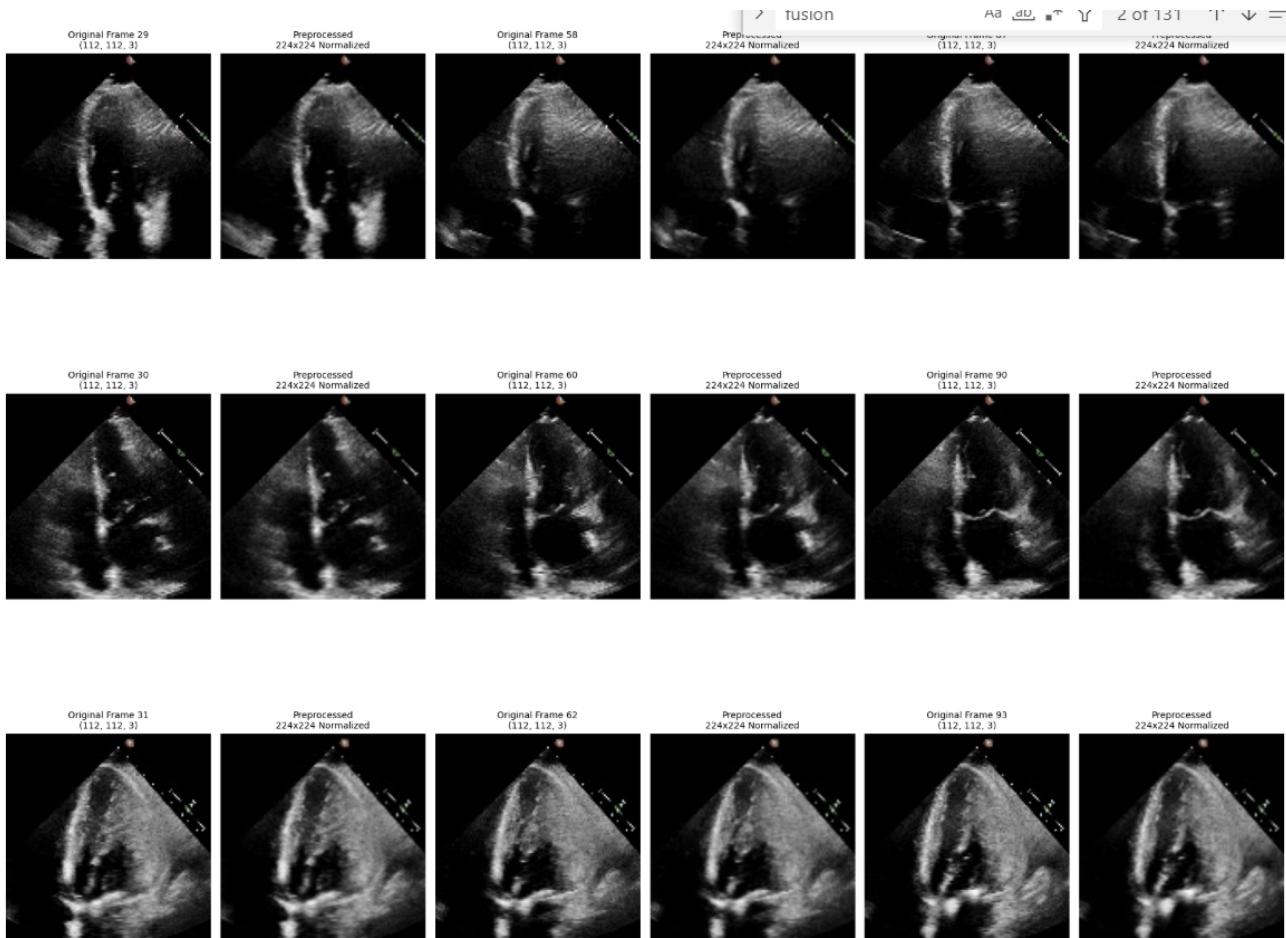
#### 3.5 Tổng quan về bộ dữ liệu EchoNet-Dynamic

Bộ dữ liệu **EchoNet-Dynamic** là một tập dữ liệu quy mô lớn về siêu âm tim (echocardiography), được xây dựng nhằm hỗ trợ các nghiên cứu về học máy và trí tuệ nhân tạo trong lĩnh vực tim mạch. Tập dữ liệu bao gồm **10.036 video siêu âm tim** (apical-4-chamber view) được thu thập từ 10.036 bệnh nhân ngẫu nhiên tại một Bệnh viện Đại học trong giai đoạn từ năm 2006 đến 2018. Mỗi video đại diện cho một cá nhân duy nhất, bảo đảm tính đa dạng và không trùng lặp.

### 3.5.1 Thành phần dữ liệu

Một nghiên cứu siêu âm tim tiêu chuẩn thường bao gồm 50–100 video và ảnh tĩnh từ nhiều góc nhìn và kỹ thuật khác nhau (ảnh 2D, Doppler mô, Doppler màu, v.v.). Trong bộ dữ liệu này, nhóm nghiên cứu đã trích xuất một video 2D dạng xám ở mặt cắt **apical-4-chamber**, gắn liền với phép đo thể tích thất trái để tính phân suất tổng máu (Ejection Fraction, EF).

Mỗi video có độ phân giải chuẩn hoá  $112 \times 112$  pixel, số khung hình dao động từ 24–1002 với tốc độ trung bình 51 fps.



Hình 11: Ví dụ về lát cảnh từ video Echonet

Các biến nhãn đi kèm (metadata) được cung cấp trong file CSV (Bảng 2), bao gồm:

- **FileName:** tên tệp đã băm, dùng để liên kết video, nhãn và annotation.
- **Age:** tuổi bệnh nhân (làm tròn đến năm gần nhất).
- **Sex:** giới tính.
- **EF:** phân suất tổng máu, tính từ ESV và EDV.
- **ESV:** thể tích cuối tâm thu (End Systolic Volume).
- **EDV:** thể tích cuối tâm trương (End Diastolic Volume).

- **Height, Width, FPS, NumFrames:** thông số kỹ thuật video.
- **Split:** phân chia tập train / validation / test cho benchmark.

Bảng 2: Các biến nhãn trong bộ dữ liệu EchoNet-Dynamic

Biến	Mô tả
FileName	Tên file đã băm để liên kết video và nhãn
Age	Tuổi (năm)
Sex	Giới tính
EF	Phân suất tổng máu (%)
ESV	Thể tích cuối tâm thu (mL)
EDV	Thể tích cuối tâm trương (mL)
FPS	Số khung hình/giây
NumFrames	Tổng số khung hình của video
Split	Phân loại train/validation/test

### 3.5.2 Ý nghĩa lâm sàng

Một chỉ số trung tâm trong đánh giá chức năng tim là **phân suất tổng máu thất trái (EF)**, được tính bằng công thức:

$$EF = \frac{EDV - ESV}{EDV} \times 100\%.$$

Chỉ số EF phản ánh khả năng bơm máu của tim, có giá trị quan trọng trong chẩn đoán bệnh cơ tim, đánh giá chỉ định cho hoá trị, và xem xét cấy ghép thiết bị tim mạch. EF thấp thường liên quan đến tiên lượng xấu hơn trong nhiều bệnh lý tim mạch.

### 3.5.3 Thống kê mô tả

Tổng cộng bộ dữ liệu có 10.036 video, trong đó chia thành tập huấn luyện (7.465, chiếm 75%), tập validation (1.289, chiếm 12,5%) và tập kiểm thử (1.282, chiếm 12,5%). Tuổi trung bình bệnh nhân là 68 năm, với 48% là nữ giới. Các đặc tính chính được trình bày ở Bảng 3.

Bảng 3: Thống kê mô tả bộ dữ liệu EchoNet-Dynamic

Chỉ số	Tổng	Train	Val	Test
Số video	10.036	7.465	1.289	1.282
Nữ giới (%)	48%	49%	44%	44%
Tuổi (năm)	68 ± 21	70 ± 22	62 ± 18	62 ± 17
FPS	50.9 ± 6.8	50.8 ± 6.7	51.0 ± 6.5	51.3 ± 7.3
Số khung hình	175 ± 57	175 ± 57	176 ± 52	176 ± 60
EF (%)	55.7 ± 12.5	55.7 ± 12.5	55.8 ± 12.3	55.3 ± 12.4
ESV (mL)	43.3 ± 34.5	43.2 ± 36.1	43.3 ± 34.5	43.9 ± 36.0
EDV (mL)	91.0 ± 45.7	91.0 ± 46.0	91.0 ± 43.8	91.4 ± 46.0

### 3.5.4 Bảo mật và xử lý dữ liệu

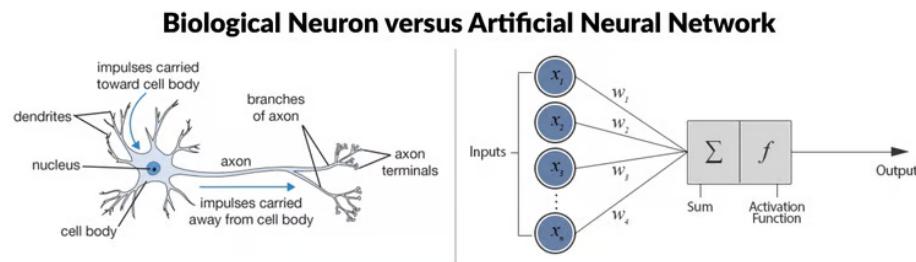
Bộ dữ liệu được khử nhận dạng theo quy trình nghiêm ngặt: loại bỏ văn bản, tín hiệu ECG và các dữ liệu ngoài vùng quét; chuyển đổi DICOM sang AVI để tránh rò rỉ thông tin cá nhân trong metadata. Các video được chuẩn hoá về kích thước  $112 \times 112$  pixel bằng phép nội suy bậc ba (cubic interpolation).

### 3.5.5 Ứng dụng

EchoNet-Dynamic được coi là một trong những bộ dữ liệu siêu âm tim công khai lớn nhất, phù hợp để huấn luyện và benchmark các mô hình deep learning trong chẩn đoán và phân loại bệnh tim.

## 3.6 Giới thiệu mô hình MLP

Mạng nơ-ron nhân tạo là cốt lõi của học máy và trí tuệ nhân tạo hiện đại. Trong số nhiều loại mạng, **perceptron đa lớp (MLP)** đóng vai trò là nền tảng cơ bản cho các hệ thống học sâu.



Hình 12: Nơ-ron sinh học so với mạng nơ-ron nhân tạo, Nguồn:[ResearchGate](#)

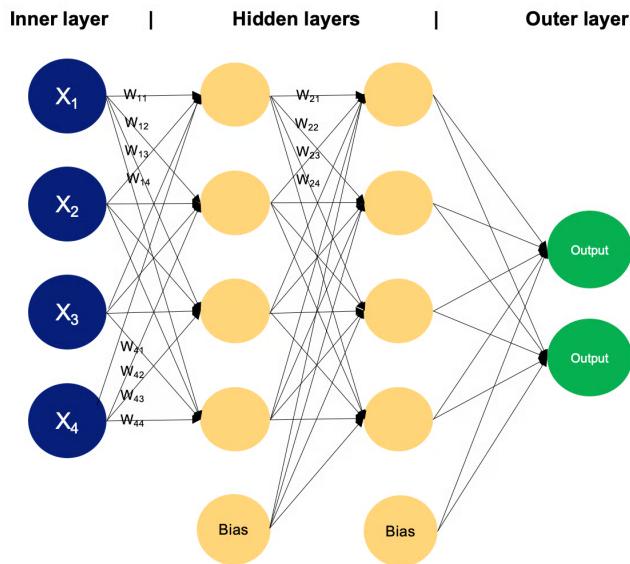
**MLP** là một loại mạng nơ-ron lan truyền thẳng bao gồm nhiều lớp nơ-ron. Các nơ-ron trong MLP thường sử dụng các hàm kích hoạt phi tuyến tính, cho phép mạng học các mẫu phức tạp trong dữ liệu. MLP rất quan trọng trong học máy vì chúng có thể học các mối quan hệ phi tuyến tính trong dữ liệu, khiến chúng trở thành những mô hình mạnh mẽ cho các tác vụ như phân loại, hồi quy và nhận dạng mẫu.

### 3.6.1 Các thành phần MLP

MLP đã được sử dụng rộng rãi trong nhiều lĩnh vực, bao gồm nhận dạng hình ảnh, xử lý ngôn ngữ tự nhiên và nhận dạng giọng nói, cùng nhiều lĩnh vực khác. Tính linh hoạt về kiến trúc và khả năng xấp xỉ bất kỳ hàm nào trong một số điều kiện nhất định khiến chúng trở thành nền tảng cơ bản trong học sâu và nghiên cứu mạng nơ-ron. Hãy cùng tìm hiểu sâu hơn về một số khái niệm chính của nó.

#### Các thành phần

- **Lớp đầu vào (Input layer):** Lớp đầu vào bao gồm các nút hoặc nơ-ron tiếp nhận dữ liệu đầu vào ban đầu. Mỗi nơ-ron đại diện cho một đặc điểm hoặc chiều của dữ liệu đầu vào. Số lượng nơ-ron trong lớp đầu vào được xác định bởi chiều của dữ liệu đầu vào.
- **Lớp ẩn (Hidden layer):** Giữa các lớp đầu vào và đầu ra, có thể có một hoặc nhiều lớp nơ-ron. Mỗi nơ-ron trong một lớp ẩn nhận đầu vào từ tất cả các nơ-ron trong lớp trước đó (lớp đầu vào hoặc một lớp ẩn khác) và tạo ra đầu ra được truyền đến lớp tiếp theo. Số lượng lớp ẩn và số lượng nơ-ron trong mỗi lớp ẩn là các siêu tham số cần được xác định trong giai đoạn thiết kế mô hình.
- **Lớp đầu ra (Output layer):** Lớp này bao gồm các nơ-ron tạo ra kết quả đầu ra cuối cùng của mạng. Số lượng nơ-ron trong lớp đầu ra phụ thuộc vào bản chất của tác vụ. Trong phân loại nhị phân, có thể có một hoặc hai nơ-ron tùy thuộc vào hàm kích hoạt và biểu diễn xác suất thuộc về một lớp; trong khi trong các tác vụ phân loại đa lớp, có thể có nhiều nơ-ron trong lớp đầu ra.



Hình 13: Ví dụ về MLP có hai lớp ẩn, Nguồn:[datacamp](#)

- **Weights:** Các neuron ở các lớp liền kề được kết nối hoàn toàn với nhau. Mỗi kết nối có một trọng số liên quan, quyết định độ mạnh của kết nối. Các trọng số này được học trong quá trình huấn luyện.
- **Bias neurons:** mỗi lớp (trừ input) thường có thêm một neuron bias cung cấp đầu vào hằng số cho tầng kế tiếp. Neuron này có trọng số riêng được học trong quá trình huấn luyện, giúp dịch chuyển hàm kích hoạt và tăng khả năng khớp dữ liệu.
- **Hàm kích hoạt:** mỗi neuron trong các lớp ẩn và lớp đầu ra áp dụng một hàm kích hoạt cho tổng trọng số các đầu vào của nó. Các hàm kích hoạt phổ biến bao gồm sigmoid, tanh, ReLU (Đơn vị tuyến tính chỉnh lưu) và softmax. Các hàm này đưa tính phi tuyến tính vào mạng, cho phép mạng học các mẫu phức tạp trong dữ liệu.
- **Feedforward and Backpropagation:** MLP được đào tạo bằng thuật toán truyền ngược, thuật toán này tính toán độ dốc của hàm mất mát theo các tham số của mô hình và cập nhật các tham số theo cách lặp đi lặp lại để giảm thiểu mất mát.

### 3.6.2 Cách thức hoạt động

Trong một MLP, các neuron xử lý thông tin theo từng bước, thực hiện các phép tính liên quan đến tổng có trọng số và các phép biến đổi phi tuyến tính.

- **Tầng Input:** nhận dữ liệu đầu vào, mỗi neuron đại diện cho một đặc trưng. Neuron ở đây chỉ truyền dữ liệu sang tầng ẩn, không tính toán.
- **Tầng Ẩn:**
  1. Mỗi neuron nhận toàn bộ đầu vào từ tầng trước, nhân với trọng số  $w$  và cộng thêm bias  $b$ .

2. công thức tổng có trọng số

$$\text{Weighted Sum} = \sum_{i=1}^n w_i x_i + b$$

3. Kết quả Weighted Sum được đưa qua hàm kích hoạt  $f(\text{Weighted Sum})$  (ReLU, sigmoid, tanh...), giúp mạng học được quan hệ phi tuyến và mô hình hóa dữ liệu phức tạp.

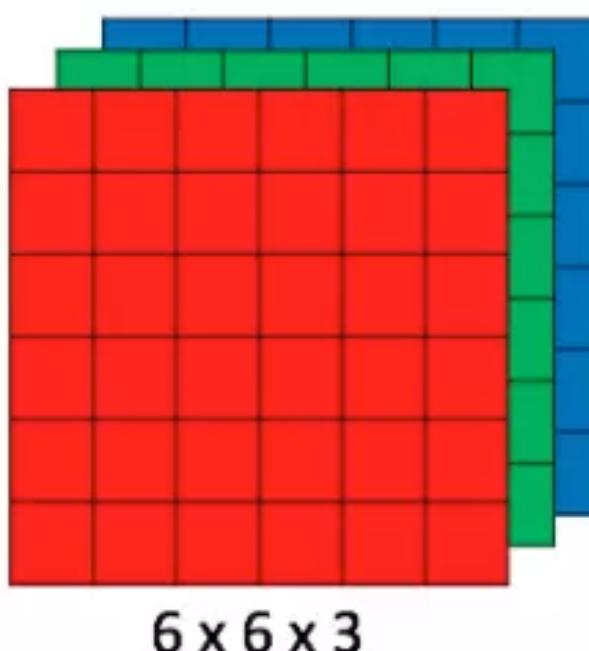
- **Tầng Output:** sinh ra dự đoán cuối cùng. Số neuron ở đây phụ thuộc bài toán: 1 neuron cho phân loại nhị phân, nhiều neuron cho phân loại đa lớp, hoặc giá trị thực cho hồi quy. Thường dùng hàm kích hoạt khác (ví dụ: sigmoid cho nhị phân, softmax cho đa lớp).
- **Training:** Trọng số  $w$  và bias  $b$  được cập nhật để giảm sai số giữa dự đoán và giá trị thực. Việc tối ưu dùng các thuật toán như Stochastic Gradient Descent (SGD), tính gradient của hàm mất mát và điều chỉnh trọng số lặp lại nhiều lần. Công thức cập nhập trọng số:

$$w_i \leftarrow w_i - \eta \frac{\partial L}{\partial w_i}$$

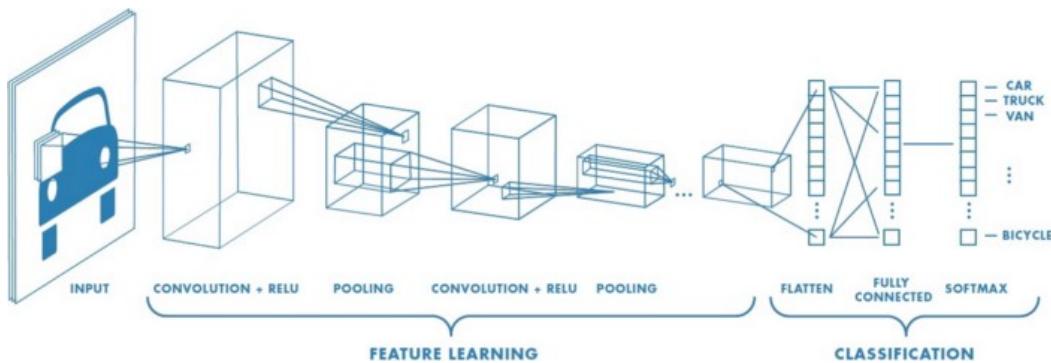
### 3.7 Giới thiệu ResNet50 và CNN

#### 3.7.1 CNN cơ bản

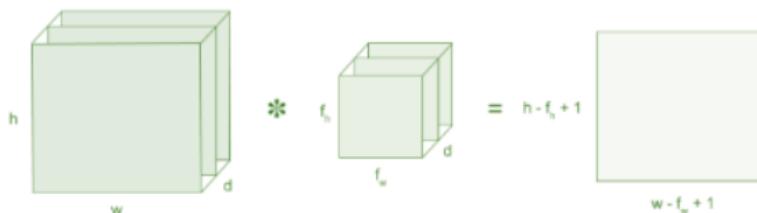
- Trong mạng neural, mô hình mạng neural tích chập (CNN) là 1 trong những mô hình để nhận dạng và phân loại hình ảnh. Trong đó, xác định đối tượng và nhận dạng khuôn mặt là 1 trong số những lĩnh vực mà CNN được sử dụng rộng rãi.
- CNN phân loại hình ảnh bằng cách lấy 1 hình ảnh đầu vào, xử lý và phân loại nó theo các hạng mục nhất định (Ví dụ: Chó, Mèo, Hổ, ...). Máy tính coi hình ảnh đầu vào là 1 mảng pixel và nó phụ thuộc vào độ phân giải của hình ảnh. Dựa trên độ phân giải hình ảnh, máy tính sẽ thấy  $H \times W \times D$  ( $H$ : Chiều cao,  $W$ : Chiều rộng,  $D$ : Độ dày). Ví dụ: Hình ảnh là mảng ma trận RGB  $6 \times 6 \times 3$  (3 ở đây là giá trị RGB).



- Về kỹ thuật, mô hình CNN để training và kiểm tra, mỗi hình ảnh đầu vào sẽ chuyển nó qua 1 loạt các lớp tích chập với các bộ lọc (Kernels), tổng hợp lại các lớp được kết nối đầy đủ (Full Connected) và áp dụng hàm Softmax để phân loại đối tượng có giá trị xác suất giữa 0 và 1. Hình dưới đây là toàn bộ luồng CNN để xử lý hình ảnh đầu vào và phân loại các đối tượng dựa trên giá trị.



**Lớp tích chập (Convolution Layer)** Tích chập là lớp đầu tiên để trích xuất các tính năng từ hình ảnh đầu vào. Tích chập duy trì mối quan hệ giữa các pixel bằng cách tìm hiểu các tính năng hình ảnh bằng cách sử dụng các ô vuông nhỏ của dữ liệu đầu vào. Nó là 1 phép toán có 2 đầu vào như ma trận hình ảnh và 1 bộ lọc hoặc hạt nhân.



- **Ma trận (volume):**  $(h \times w \times d)$

- $h$ : chiều cao
- $w$ : chiều rộng
- $d$ : chiều sâu (hoặc số kênh/điểm ảnh theo khối 3D)

- **Bộ lọc filter:**  $(f_h \times f_w \times d)$

- $f_h$ : chiều cao của filter
- $f_w$ : chiều rộng của filter
- $d$ : chiều sâu/trục sâu của filter trùng với chiều sâu của ma trận đầu vào (thường cùng số kênh)

- **Lớp tích chập:** có kích thước  $(h - f_h + 1) \times (w - f_w + 1) \times 1$  nếu chỉ một filter duy nhất và vẫn giữ nguyên chiều sâu 1 (hoặc có thể nhân bằng số lượng kênh/filters nếu có nhiều filters).

Xem xét 1 ma trận  $5 \times 5$  có giá trị pixel là 0 và 1. Ma trận bộ lọc  $3 \times 3$  như hình bên dưới.

**5 x 5 – Image Matrix**

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

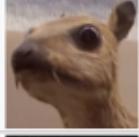
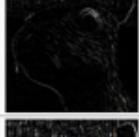
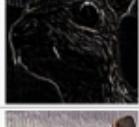
**\***

**3 x 3 – Filter Matrix**

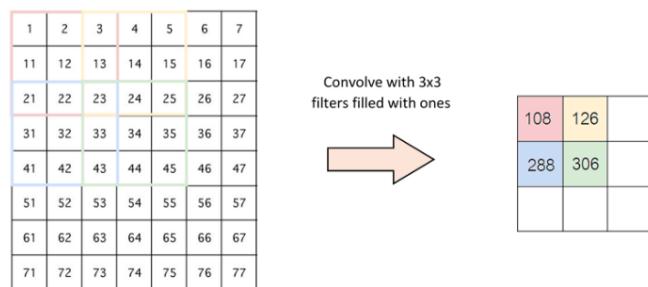
1	0	1
0	1	0
1	0	1

Sau đó, lớp tích chập của ma trận hình ảnh  $5 \times 5$  nhân với ma trận bộ lọc  $3 \times 3$  gọi là 'Feature Map' như hình bên dưới.

Sự kết hợp của 1 hình ảnh với các bộ lọc khác nhau có thể thực hiện các hoạt động như phát hiện cạnh, làm mờ và làm sắc nét bằng cách áp dụng các bộ lọc. Ví dụ dưới đây cho thấy hình ảnh tích chập khác nhau sau khi áp dụng các Kernel khác nhau.

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

**Bước nhảy (Stride)** Stride là số pixel thay đổi trên ma trận đầu vào. Khi stride là 1 thì ta di chuyển các kernel 1 pixel. Khi stride là 2 thì ta di chuyển các kernel đi 2 pixel và tiếp tục như vậy. Hình dưới là lớp tích chập hoạt động với stride là 2.



**Đường viền (Padding)** Đôi khi kernel không phù hợp với hình ảnh đầu vào. Ta có 2 lựa chọn:

- Chèn thêm các số 0 vào 4 đường biên của hình ảnh (padding).
- Cắt bớt hình ảnh tại những điểm không phù hợp với kernel.

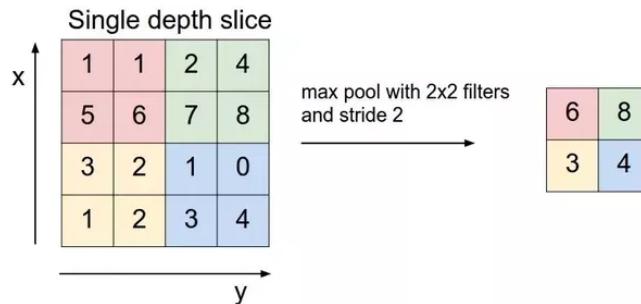
**Hàm kích hoạt (Activation Function)** Như sigmoid hoặc tanh (ReLU, ...) để thêm tính phi tuyến tính.

Vì dữ liệu trong thế giới mà chúng ta tìm hiểu là các giá trị tuyến tính không âm.

**Lớp gộp (Pooling Layer)** Lớp pooling sẽ giảm bớt số lượng tham số khi hình ảnh quá lớn. Không gian pooling còn được gọi là lấy mẫu con hoặc lấy mẫu xuống làm giảm kích thước của mỗi map nhưng vẫn giữ lại thông tin quan trọng. Các pooling có thể có nhiều loại khác nhau:

- Max Pooling
- Average Pooling
- Sum Pooling

Max pooling lấy phần tử lớn nhất từ ma trận đối tượng, hoặc lấy tổng trung bình. Tổng tất cả các phần tử trong map gọi là sum pooling



**Lớp Fully Connected** : Để phân loại cuối cùng.

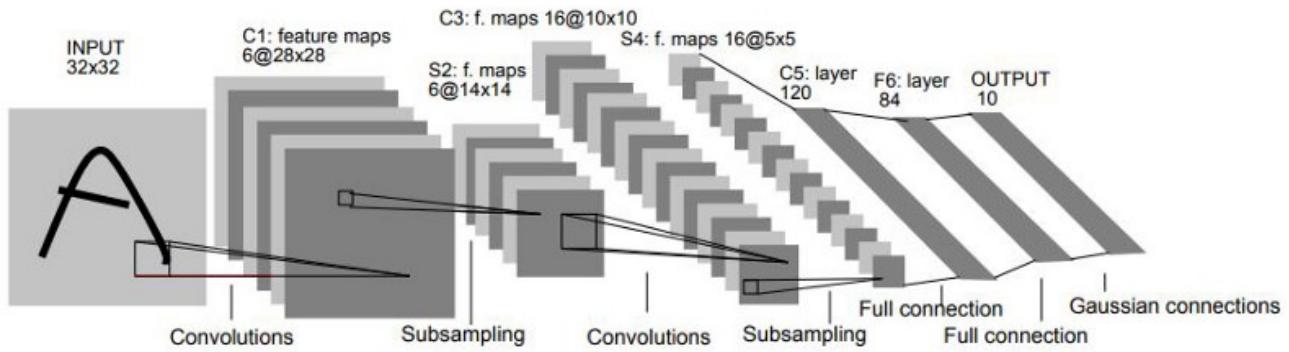
**Chuỗi I/O minh họa** Ví dụ ảnh vào  $224 \times 224 \times 3$ :

$$\begin{aligned}
 & 224 \times 224 \times 3 \xrightarrow{\text{Conv } 3 \times 3, 64, S=1, P=1} 224 \times 224 \times 64 \xrightarrow{\text{BN+ReLU}} 224 \times 224 \times 64 \\
 & \xrightarrow{\text{MaxPool } 2 \times 2, S=2} 112 \times 112 \times 64 \xrightarrow{\text{Conv } 3 \times 3, 128} 112 \times 112 \times 128 \xrightarrow{\text{BN+ReLU}} 112 \times 112 \times 128 \\
 & \xrightarrow{\text{MaxPool}} 56 \times 56 \times 128 \xrightarrow{\text{Conv } 3 \times 3, 256} 56 \times 56 \times 256 \xrightarrow{\text{BN+ReLU}} 56 \times 56 \times 256 \\
 & \xrightarrow{\text{GAP}} 1 \times 1 \times 256 \xrightarrow{\text{FC } 256 \rightarrow K} K \text{ (Softmax).}
 \end{aligned}$$

**Vấn đề ban đầu:** Huấn luyện khó với dataset lớn, vanishing gradients, và tham số nhiều. Đây là nền tảng, nhưng chưa thực tế cho ứng dụng phức tạp.

### 3.7.2 LeNet-5 (1998): Ứng Dụng Đầu Tiên Thực Tế

LeNet-5 là mô hình CNN đầu tiên được triển khai rộng rãi, do Yann LeCun phát triển cho nhận dạng chữ số handwritten (MNIST). Đây là bước nâng cấp từ CNN cơ bản bằng cách tích hợp huấn luyện end-to-end.



Những khác biệt chính của LeNet-5 so với “CNN cơ bản”

#### 1. Subsampling học được với tham số và phi tuyến tính

- **CNN cơ bản:** Pooling cố định (max/avg), không học.
- **LeNet-5:** Average pooling với  $\alpha, \beta$  học được:  $y_c = \tanh(\alpha_c \cdot \text{AvgPool}(x_c) + \beta_c)$ .
- **Lợi ích:** Linh hoạt hơn, điều chỉnh kênh theo dữ liệu, tăng biểu diễn.

#### 2. Kết nối thừa giữa feature maps (C3)

- **CNN cơ bản:** Kết nối đầy đủ giữa các kênh.
- **LeNet-5:** Bảng kết nối thừa, mỗi map chỉ kết nối tập con.
- **Lợi ích:** Giảm tham số, đa dạng đặc trưng, tránh đối xứng.

#### 3. Convolution toàn ảnh thay flatten sớm (C5)

- **CNN cơ bản:** Flatten rồi FC sau conv/pool.
- **LeNet-5:** Conv 5x5 trên 5x5 ra 1x1x120, giữ weight sharing.
- **Lợi ích:** Giảm tham số, tích hợp đặc trưng không gian tốt hơn, ít overfit.

#### 4. Thiết kế tầng tối ưu cho input 32x32

- **CNN cơ bản:** Kiến trúc linh hoạt, không khớp chặt.
- **LeNet-5:** Chuỗi lớp ăn khớp ( $32 \rightarrow 28 \rightarrow 14 \rightarrow 10 \rightarrow 5 \rightarrow 1$ ), receptive field tăng dần.
- **Lợi ích:** Ngữ cảnh mở rộng có hệ thống, tối ưu cho ký tự.

#### 5. Huấn luyện end-to-end hoàn chỉnh

- **CNN cơ bản:** Đặc trưng thủ công + classifier riêng.
- **LeNet-5:** Học đồng thời qua backprop, loss chung.
- **Lợi ích:** Đặc trưng tối ưu, chính xác cao trên MNIST.

#### 6. Hiệu quả tham số cao

- **CNN cơ bản:** MLP tương đương 310k tham số.
- **LeNet-5:** Chỉ 60k nhờ receptive field cục bộ và sharing.

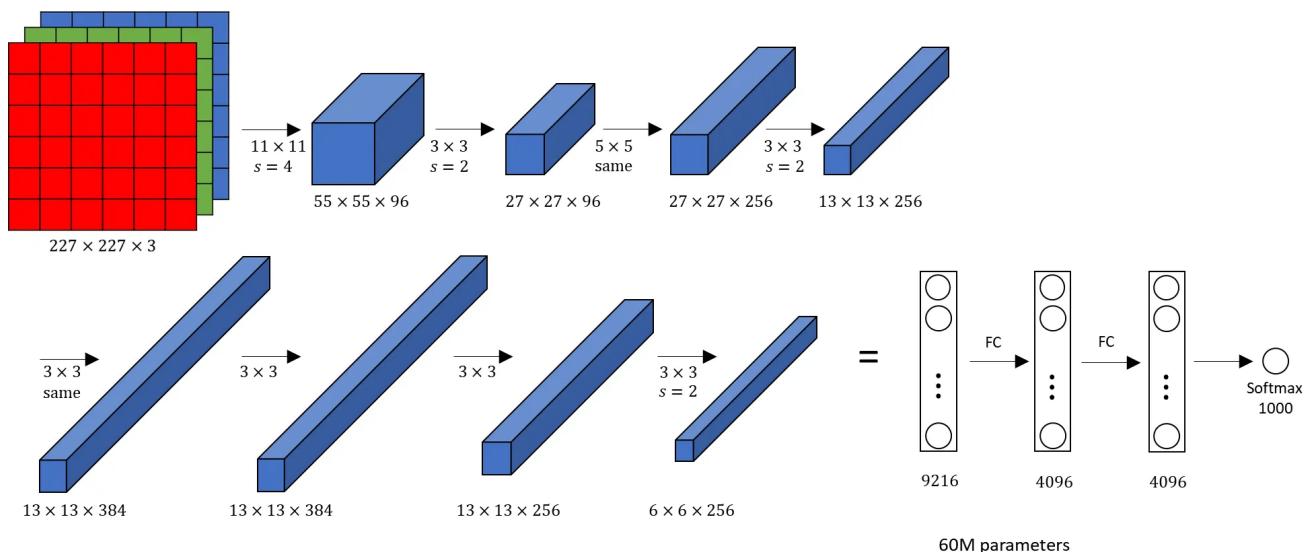
- **Lợi ích:** Nhẹ, dễ huấn luyện, biểu diễn phong phú.

## 7. Phân cấp đặc trưng và bất biến chủ đích

- **CNN cơ bản:** Phân cấp ngầm định.
- **LeNet-5:** Edges → motifs → digits, với invariance dịch chuyển.
- **Lợi ích:** Generalization tốt cho viết tay, hierarchical learning rõ ràng.

### 3.7.3 AlexNet (2012): Bước Nhảy Vọt Với Độ Sâu Và GPU

AlexNet đánh dấu sự bùng nổ của CNN, thắng cuộc ImageNet 2012. Nó nâng cấp từ LeNet bằng cách tăng độ sâu và tận dụng phần cứng hiện đại.



### Khác biệt so với LeNet-5

#### 1. Quy mô & độ sâu

- **LeNet:** ảnh  $32 \times 32$ , 2 conv nông + pooling có tham số, C5 “conv-toàn-ảnh”, rất ít kênh.
- **AlexNet:** ảnh  $227 \times 227$ , 5 conv + 3 FC, kênh lớn ( $96 \rightarrow 256 \rightarrow 384 \dots$ ), *downsample* mạnh (Conv  $11 \times 11$  stride = 4 + 3 MaxPool).

#### 2. Kích hoạt

- **LeNet:** tanh/sigmoid (dễ bão hòa).
- **AlexNet:** ReLU → học nhanh, gradient khoẻ.

#### 3. Pooling

- **LeNet:** average pooling có tham số (+ tanh).
- **AlexNet:** max-pooling chống lắn ( $3 \times 3, s=2$ ) → chọn lọc đặc trưng tốt trên ảnh tự nhiên.

#### 4. Chuẩn hóa & regularize

- **LeNet:** không BN/LRN, không dropout.
- **AlexNet:** LRN (bối cảnh thời điểm đó hữu ích), Dropout 0.5 ở 2 FC để chống overfit.

#### 5. Huấn luyện & phần cứng

- **LeNet:** CPU, dữ liệu nhỏ (MNIST).
- **AlexNet:** GPU, dữ liệu lớn (ImageNet 1.2M), *data augmentation* (crop, flip, PCA color).

#### 6. Classifier

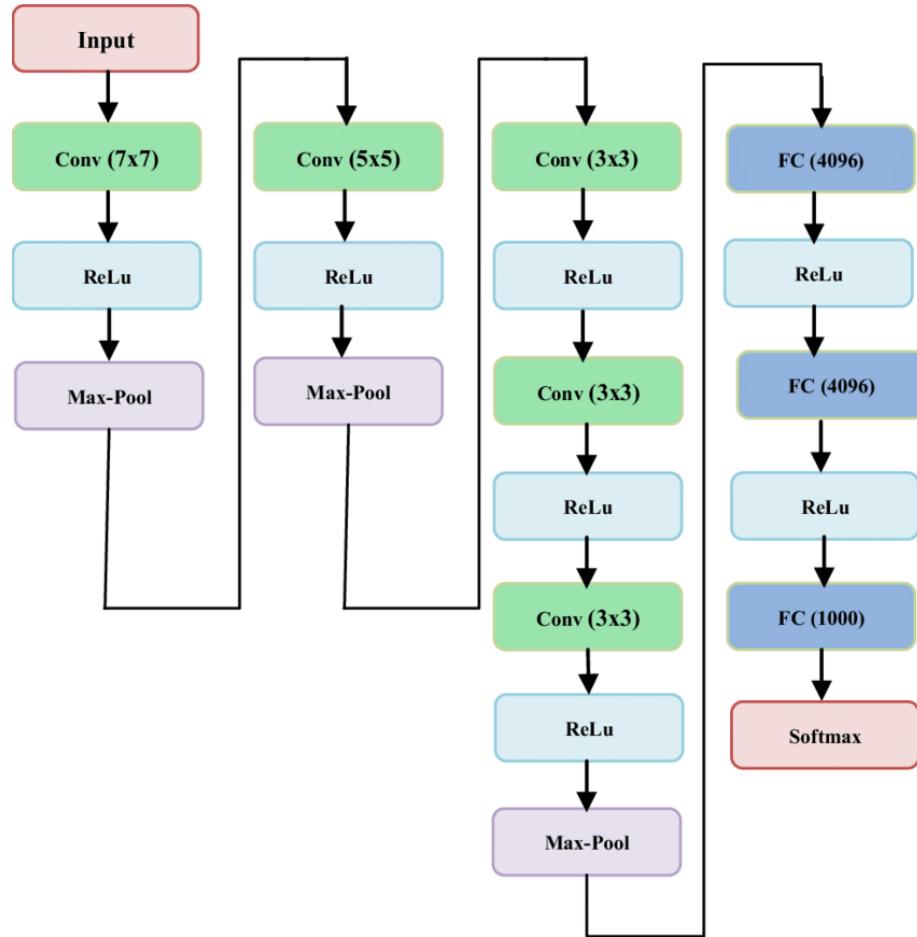
- **LeNet:** C5 (conv phủ kín) → F6 84 → 10 lớp.
- **AlexNet:** Flatten 9216 → FC4096 → FC4096 → FC1000 (“rất dày” → nhiều tham số nhưng mạnh).

#### Takeaway

- **So với LeNet-5:** AlexNet không chỉ “tăng lớp” mà là combo kiến trúc + kỹ thuật huấn luyện + GPU + dữ liệu lớn, mở ra kỷ nguyên CNN cho ảnh tự nhiên.
- **So với CNN cơ bản:** vẫn giữ khung Conv–ReLU–Pool–FC, nhưng chuẩn hóa lựa chọn (ReLU, max-pool, conv  $3 \times 3$  ở giữa, FC dày, regularization mạnh) để đạt hiệu năng ở quy mô lớn.

#### 3.7.4 ZFNet (2013): Tinh Chính Và Trực Quan Hóa

ZFNet (Zeiler & Fergus) thắng ImageNet 2013, chủ yếu “mổ xẻ” AlexNet bằng deconv/visualization để **tinh chỉnh kiến trúc**, chứ không đổi triết lý.



ZFNet làm gì?

- **Deconv/Visualization** Gắn một “*deconvnet*” vào sau từng lớp để nhìn lại vùng ảnh kích hoạt mạnh  $\Rightarrow$  chẩn đoán:
  - **Conv1**  $11 \times 11$  stride 4 của AlexNet làm mất chi tiết sớm (*aliasing*).
  - Một số cấu hình khiến feature map “đậm nhiều/ít cấu trúc”.
- **Từ chẩn đoán  $\rightarrow$  chỉnh kiến trúc**
  - **Conv1 nhỏ hơn, stride nhỏ hơn:**  $7 \times 7$ , stride 2 (giữ chi tiết tốt hơn).
  - Giữ cách “giảm dần  $H \times W$ , tăng kênh”, nhưng bớt *hung hăn* ở đầu mạng; pooling vẫn  $3 \times 3$ , stride 2 (chống lẩn).
  - Phần giữa dùng  $3 \times 3$  **đều tay** (gần “tiền thân” của VGG-style).
  - Bỏ **grouped conv 2-GPU** của AlexNet (thường train trên 1 GPU)  $\Rightarrow$  các lớp nhìn *full kênh*.
  - **Dropout** ở FC vẫn giữ; **LRN** xem như *không cần* (tác dụng nhỏ khi đã chỉnh conv/stride hợp lý).

Khác gì AlexNet & LeNet?

- So với AlexNet

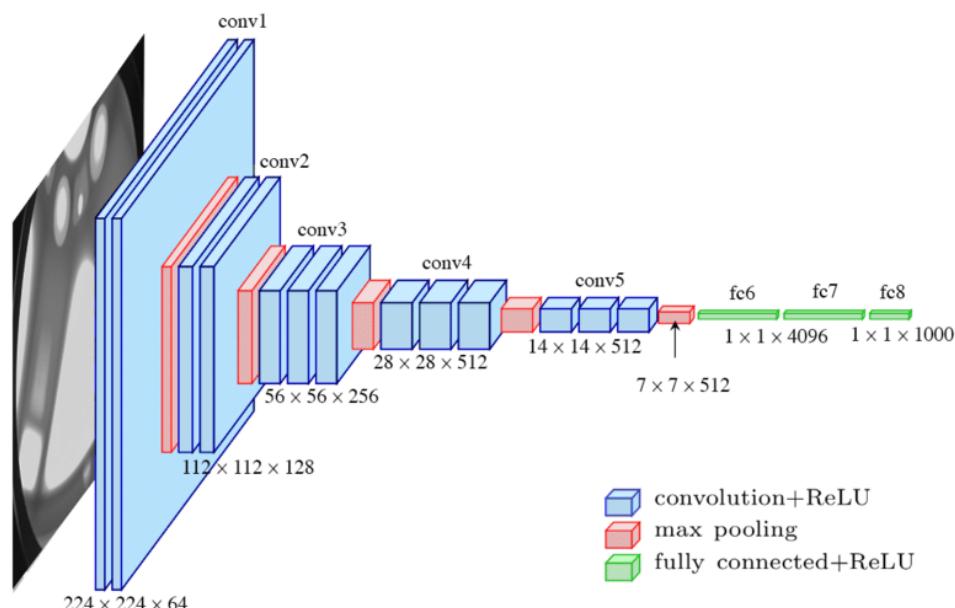
- Nhỏ kernel/stride ở đầu ( $7 \times 7, s=2$  so với  $11 \times 11, s=4$ )  $\Rightarrow$  ít mất chi tiết sớm, giảm aliasing.
- Giữ **5 conv + 3 FC**, max-pool chống lắn, ReLU, dropout — nhưng cấu hình “vừa tay” hơn  $\Rightarrow$  mAP/top-5 tốt hơn.
- Không dựa vào LRN/2-GPU split; nhấn mạnh thiết kế *conv/stride hợp lý* là chìa khoá.

- **So với LeNet**

- Làm trên ảnh tự nhiên lớn (**ImageNet**), sâu & rộng hơn nhiều; dùng **ReLU + max-pool**, không còn “avg-pool có tham số” hay kết nối thừa.

**Takeaway ZFNet = AlexNet được “bác sĩ chỉnh hình”:** dùng trực quan hoá để giảm downsample sớm, chuẩn hoá kernel/stride, bớt rườm rà  $\Rightarrow$  đặc trưng rõ rệt hơn, hiệu năng cao hơn, đặt tiền đề cho VGG (chuỗi  $3 \times 3$ ).

### 3.7.5 VGGNet (2014): Tăng Độ Sâu Với Kernel Nhỏ

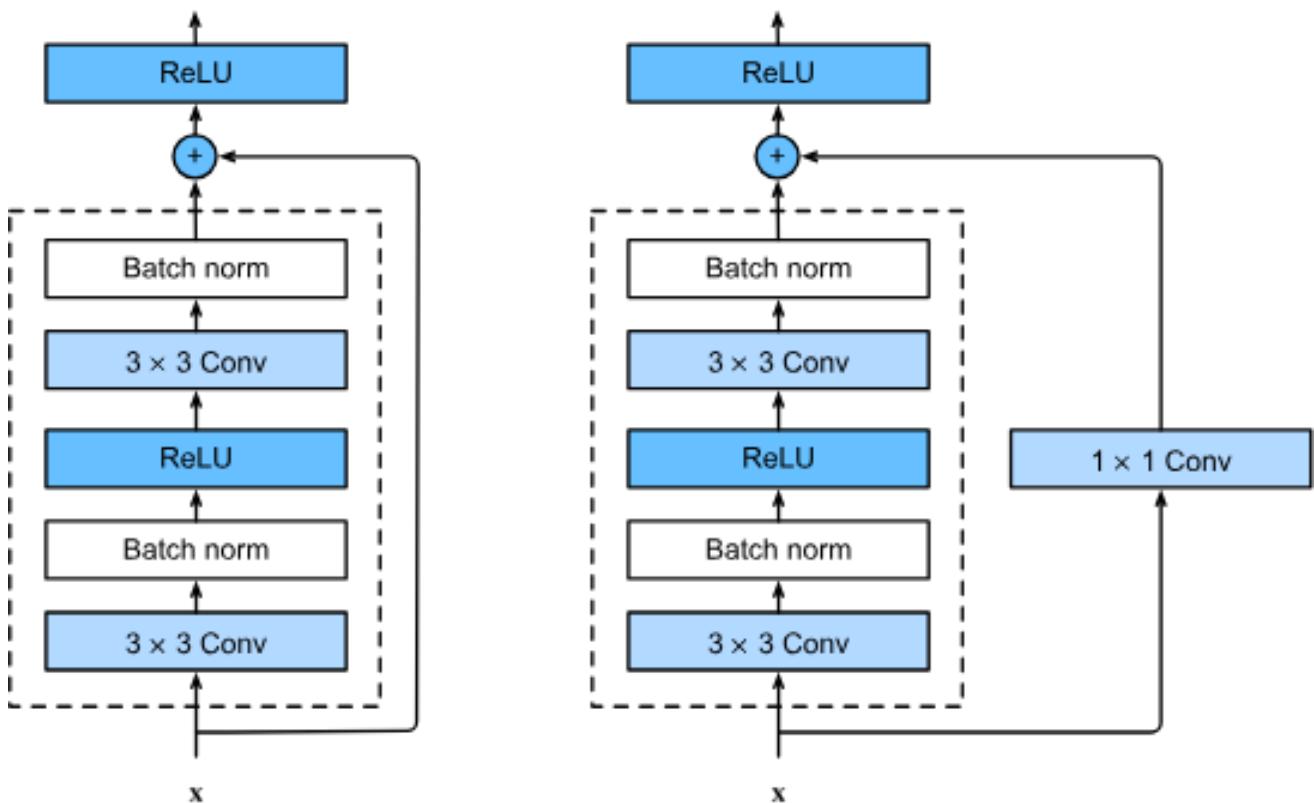


- **Ý tưởng:** kiến trúc đồng nhất, chỉ dùng Conv  $3 \times 3$  (stride = 1, padding = 1) xếp chồng; giảm kích thước chỉ bằng MaxPool  $2 \times 2$  (stride = 2).
- **Độ sâu:** VGG-16/19 có 16–19 lớp conv, sâu hơn hẳn AlexNet/ZFNet.
- **Vì sao  $3 \times 3$  xếp chồng?**  $2 \times (3 \times 3) \approx 5 \times 5$ ,  $3 \times (3 \times 3) \approx 7 \times 7$  nhưng ít tham số hơn và nhiều phi tuyến hơn  $\Rightarrow$  biểu diễn mạnh hơn.
- **Nhịp kiến trúc (VGG-16):**

$(2 \times 64) \rightarrow \text{pool} \rightarrow (2 \times 128) \rightarrow \text{pool} \rightarrow (3 \times 256) \rightarrow \text{pool} \rightarrow (3 \times 512) \rightarrow \text{pool} \rightarrow (3 \times 512) \rightarrow \text{pool} \rightarrow \text{head}$

- **BatchNorm:** bản gốc (2014) *không BN*; biến thể **VGG-BN** (2015) thêm BN giúp train ổn định hơn.
- **Ưu điểm:** độ chính xác ImageNet thời điểm đó  $\sim 7\%$  top-5 error; đặc trưng conv5 sạch, hữu dụng cho *transfer* (detection/segmentation).
- **Nhược điểm:**  $\sim 138M$  tham số (đa số ở 2 lớp FC 4096), tính toán nặng; *không có skip* nên khó tăng sâu hơn (dễ degradation/vanishing).
- **Khuyến nghị hiện đại:** dùng **VGG-BN**; thay Flatten bằng **GAP** ( $7 \times 7 \times 512 \rightarrow \text{GAP} \rightarrow 512 \rightarrow \text{FC}$ ) để giảm mạnh tham số; lấy đặc trưng từ  $\text{conv5\_3} + \text{GAP}/\text{GeM/R-MAC}$ .
- **Khác ZFNet:** VGG giữ  $H \times W$  lâu hơn và xếp chồng nhiều  $3 \times 3$  trước mỗi pool  $\Rightarrow$  đặc trưng giàu chi tiết/ngữ cảnh hơn (đổi lại nặng hơn).

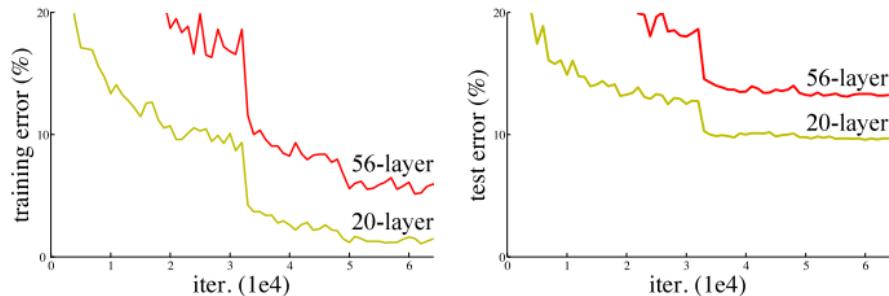
### 3.7.6 ResNet (2015): Mạng Siêu Sâu Với Residual Blocks



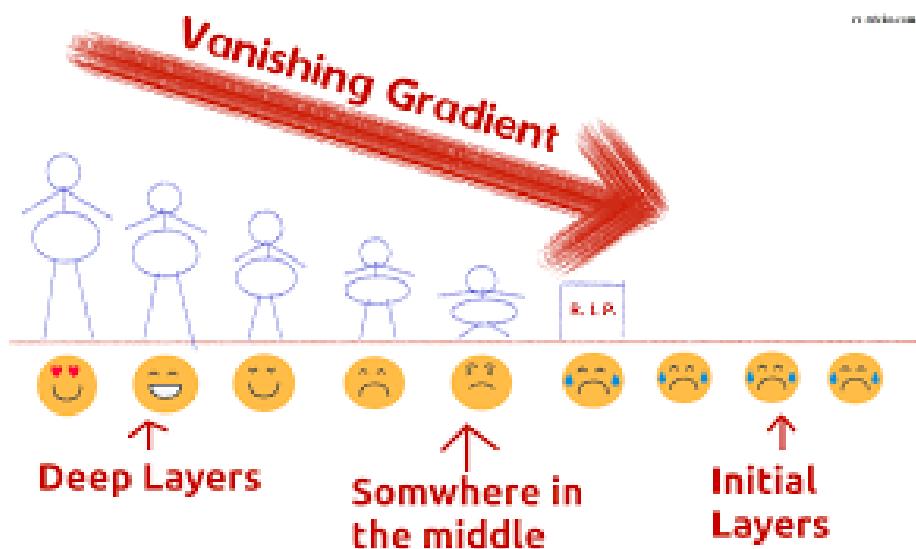
#### Động lực từ các biến thể trước

- **LeNet-5** học *end-to-end* bằng backprop nhưng còn nồng (tanh, subsampling có tham số)  $\Rightarrow$  chưa đổi mặt vấn đề tối ưu khi rất sâu.
- **AlexNet/ZFNet** đẩy quy mô (ReLU, max-pool, GPU, aug) nhưng vẫn là *mạng thắng* (không có đường tắt).
- **VGG** dùng “all- $3 \times 3$ ” để sâu (16/19 conv) song *không có skip*  $\Rightarrow$  khi tăng sâu hơn gặp **vanishing gradient & degradation** (thêm lớp làm train/test tệ hơn).

**Vanishing Gradient** Trước hết thì **Backpropagation Algorithm** là một kỹ thuật thường được sử dụng trong quá trình tranining. Ý tưởng chung của thuật toán là sẽ đi từ **output layer** đến **input layer** và tính toán gradient của cost function tương ứng cho từng parameter (weight) của mạng. Gradient Descent sau đó được sử dụng để cập nhật các parameter đó. (LeNet-5)

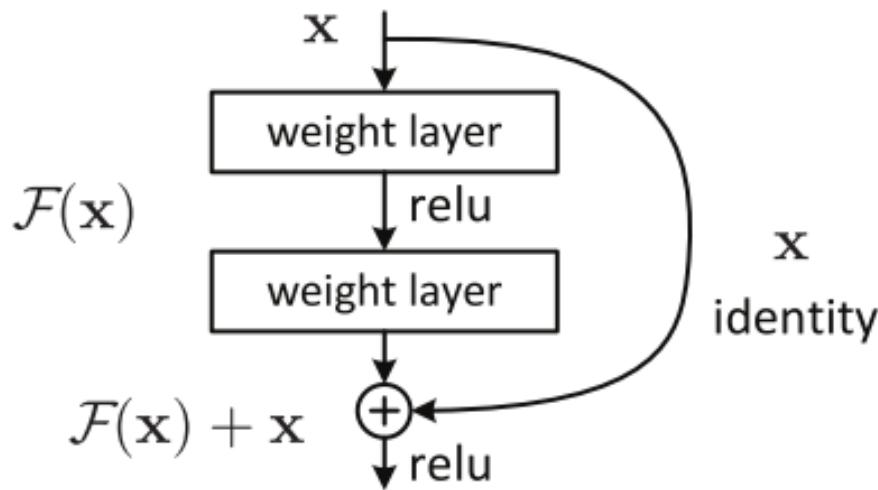


Toàn bộ quá trình trên sẽ được lặp đi lặp lại cho tới khi mà các **parameter** của network được hội tụ. Thông thường chúng ta sẽ có một **hyperparametr** (số Epoch - số lần mà tranining set được duyệt qua một lần và weights được cập nhật) định nghĩa cho số lượng vòng lặp để thực hiện quá trình này. Nếu số lượng vòng lặp quá nhỏ thì ta gấp phải trường hợp mạng có thể sẽ không cho ra kết quả tốt và ngược lại thời gian tranining sẽ lâu nếu số lượng vòng lặp quá lớn.



Tuy nhiên, trong thực tế **Gradients** thường sẽ có giá trị nhỏ dần khi đi xuống các layer thấp hơn. Dẫn đến kết quả là các cập nhật thực hiện bởi **Gradients Descent** không làm thay đổi nhiều weights của các layer đó và làm chúng không thể hội tụ và mạng sẽ không thu được kết quả tốt. Hiện tượng như vậy gọi là **Vanishing Gradients**.

**Kiến trúc mạng ResNet** Cho nên giải pháp mà ResNet đưa ra là sử dụng kết nối "tắt" đồng nhất để xuyên qua một hay nhiều lớp. Một khối như vậy được gọi là một Residual Block, như trong hình sau:



ResNet gần như tương tự với các mạng gồm có **convolution**, **pooling**, **activation** và **fully-connected layer**. Ảnh bên trên hiển thị khối dư (*residual block*) được sử dụng trong mạng. Xuất hiện một mũi tên cong xuất phát từ đầu và kết thúc tại cuối khối dư. Hay nói cách khác là sẽ bổ sung đầu vào  $X$  vào đầu ra của layer — hay chính là *phép cộng tắt* (*skip connection*) như minh họa.

Việc này giúp:

- Chống lại hiện tượng đạo hàm bằng 0 (vanishing gradient),
- Vì vẫn còn cộng thêm  $X$  vào đầu ra nên thông tin gốc không bị mất hoàn toàn.

Với  $H(x)$  là giá trị dự đoán,  $F(x)$  là giá trị thật (nhãn), ta muốn  $H(x)$  bằng hoặc xấp xỉ  $F(x)$ . Khi đó, ResNet học phần *phần dư* (*residual*) thay vì toàn bộ ánh xạ, tức là:

$$H(x) = F(x) + x$$

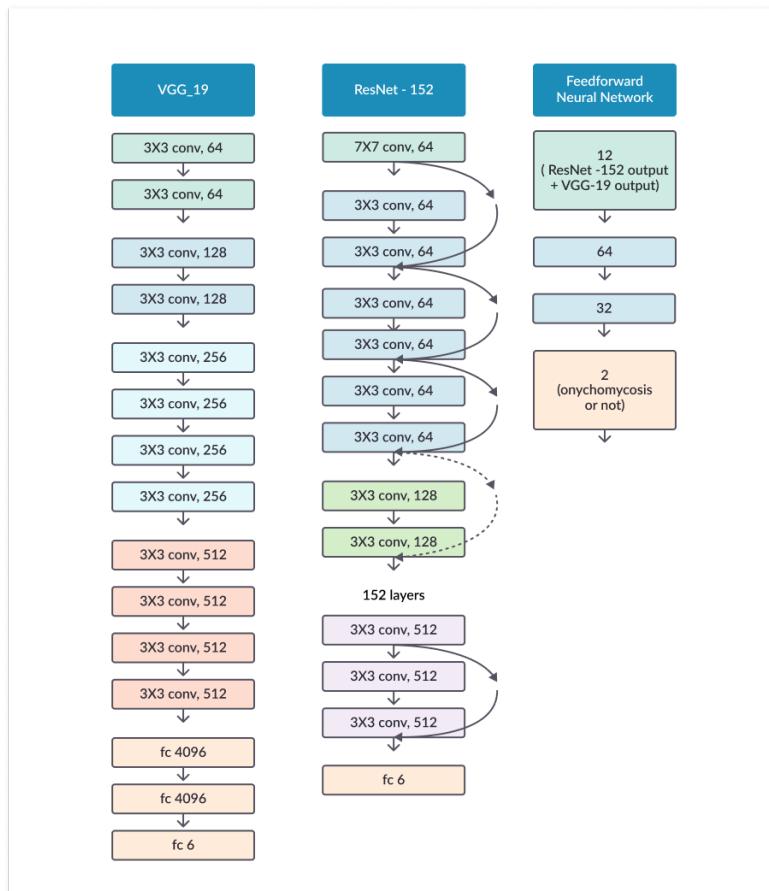
Việc  $F(x)$  có được từ  $x$  như sau:

`1 X -> weight1 -> ReLU -> weight2`

Giá trị  $H(x)$  có được bằng cách:

`1 F(x) + X -> ReLU`

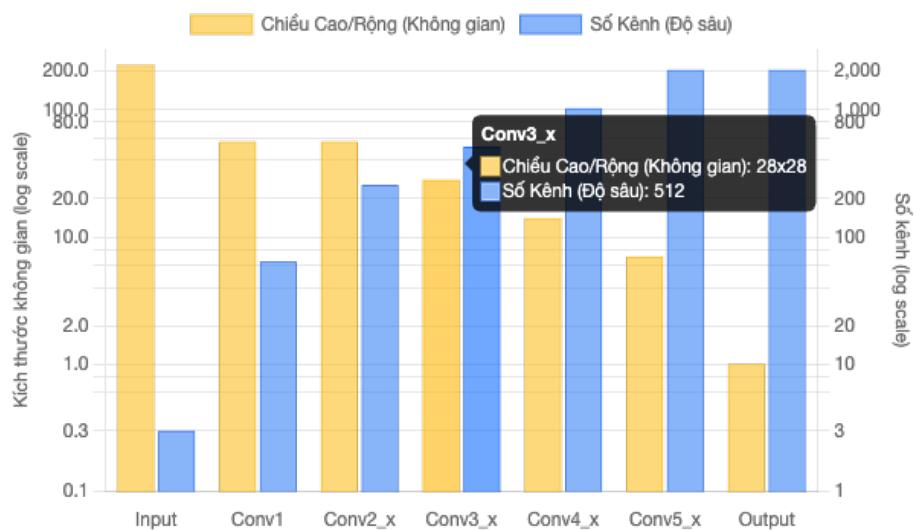
Như chúng ta đã biết việc tăng số lượng các lớp trong mạng làm giảm độ chính xác, nhưng muốn có một kiến trúc mạng sâu hơn có thể hoạt động tốt.



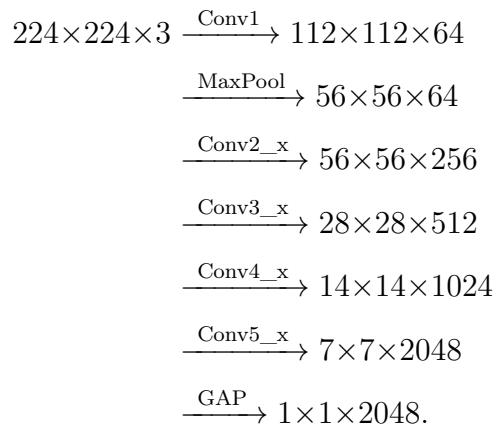
- Hình 1. VGG-19 là một mô hình CNN sử dụng kernel 3x3 trên toàn bộ mạng.
- Hình 2. ResNet dùng các kết nối tắt (skip) giữa lớp  $n$  và  $n+x$  (mũi tên cong).
- Hình 3. Dùng 12 đầu ra từ ResNet-152 và VGG-19 làm input cho mạng 2 hidden layer; đầu ra cuối qua 2 lớp ẩn.

### 3.7.7 ResNet50 Trích xuất đặc trưng

**Trực Quan Hóa Kích Thước Feature Map** Biểu đồ này thể hiện sự thay đổi kích thước của dữ liệu khi đi qua các giai đoạn của ResNet-50. Hãy chú ý cách chiều cao và chiều rộng (không gian) giảm dần, trong khi số kênh (độ sâu) tăng lên, cho phép mạng học các đặc trưng ngày càng phức tạp.



Biến đổi kích thước điển hình của ResNet-50 (output stride  $\approx 32$ ):



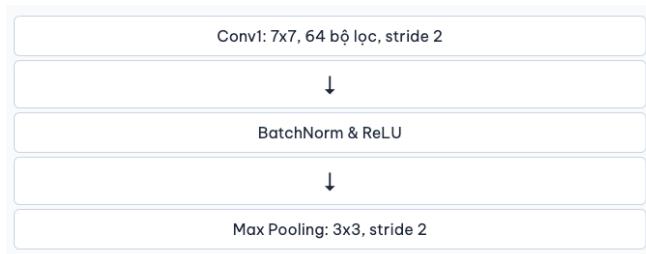
*Quy luật:*  $H \times W$  giảm theo từng stage (do stride/pool), số kênh tăng ( $64 \rightarrow 256 \rightarrow 512 \rightarrow 1024 \rightarrow 2048$ ), biểu diễn ngày càng trừu tượng/ngữ nghĩa.

**Giai đoạn 0: Ảnh Đầu Vào** Mọi thứ bắt đầu với một bức ảnh. Đối với ResNet-50, ảnh đầu vào thường được chuẩn hóa về kích thước  $224 \times 224$  pixel với 3 kênh màu (Đỏ, Lục, Lam). Đây là điểm khởi đầu của toàn bộ quá trình trích xuất đặc trưng.

- **Chuẩn hoá & chuẩn bị:** ảnh RGB  $\rightarrow$  resize/crop về  $224 \times 224 \times 3$ , chuẩn hoá theo mean/std ImageNet.
- **Chế độ trích xuất:** đặt mô hình eval để BatchNorm dùng *running stats*.

### Giai đoạn 1: Tích chập Gộp Ban Đầu (Conv1 MaxPool)

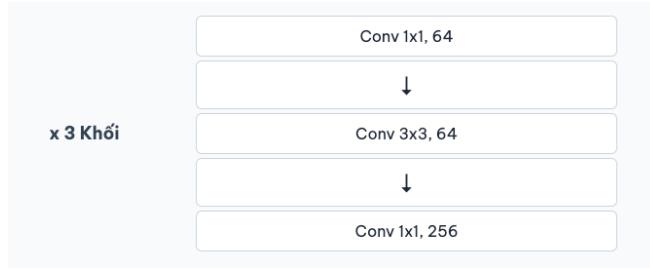
$$224 \times 224 \times 3 \xrightarrow{\text{Conv } 7 \times 7, 64, s=2} 112 \times 112 \times 64 \xrightarrow{\text{BN+ReLU}} 112 \times 112 \times 64 \xrightarrow{\text{MaxPool } 3 \times 3, s=2} 56 \times 56 \times 64.$$



*Vai trò:* trích xuất cạnh/hoa văn thô, giảm FLOPs sớm.

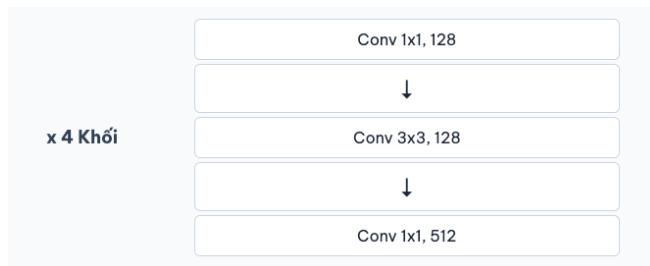
### Giai đoạn 2: Conv2\_x (3 bottleneck, out = 256)

- **Mỗi block:**  $1 \times 1$  (giảm kênh 64)  $\rightarrow 3 \times 3$  (64)  $\rightarrow 1 \times 1$  (mở kênh 256), giữa các conv có BN+ReLU; **skip identity**.
- **Kích thước:**  $56 \times 56 \times 64 \rightarrow 56 \times 56 \times 256$  (giữ  $H \times W$ ).
- **Ý nghĩa:** học mô-típ cục bộ (góc, đường cong, texture nhỏ).



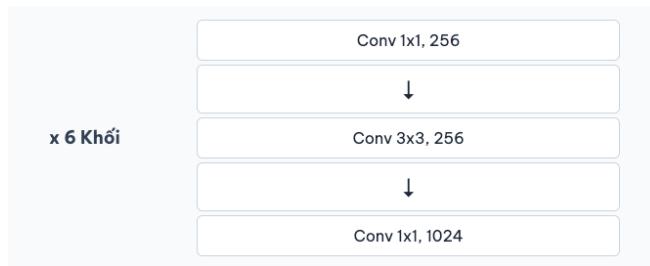
### Giai đoạn 3: Conv3\_x (4 bottleneck, out = 512, downsample)

- **Block đầu:** downsample (stride = 2 đặt ở  $1 \times 1$  hoặc  $3 \times 3$  tùy biến thể) + **projection skip**  $1 \times 1$  để khớp kênh.
- **Kích thước:**  $56 \times 56 \times 256 \rightarrow 28 \times 28 \times 512$ .
- **Ý nghĩa:** mô tả cấu trúc lớn hơn (bộ phận đơn giản của đối tượng).



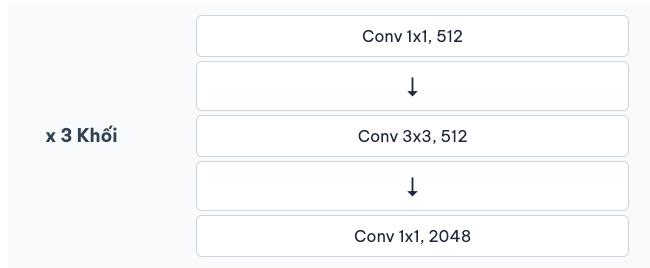
### Giai đoạn 4: Conv4\_x (6 bottleneck, out = 1024, downsample)

- **Kích thước:**  $28 \times 28 \times 512 \rightarrow 14 \times 14 \times 1024$  (downsample ở block đầu).
- **Ý nghĩa:** đặc trưng giàu ngữ nghĩa nhưng vẫn giữ tương đồng thông tin không gian; thường là “xương sống” cho det/seg.



### Giai đoạn 5: Conv5\_x (3 bottleneck, out = 2048, downsample)

- **Kích thước:**  $14 \times 14 \times 1024 \rightarrow 7 \times 7 \times 2048$ .
- **Ý nghĩa:** đặc trưng cấp vật thể/cảnh; mỗi ô  $7 \times 7$  có receptive field gần toàn ảnh.



### Giai đoạn 6: Gộp & Phân loại cuối

$7 \times 7 \times 2048 \xrightarrow{\text{Global Average Pooling}} 1 \times 1 \times 2048 \xrightarrow{\text{FC}} \text{logits}$  (vd. 1000 lớp ImageNet).

Khi trích xuất đặc trưng: dùng ở vector 2048-chiều (trước FC).

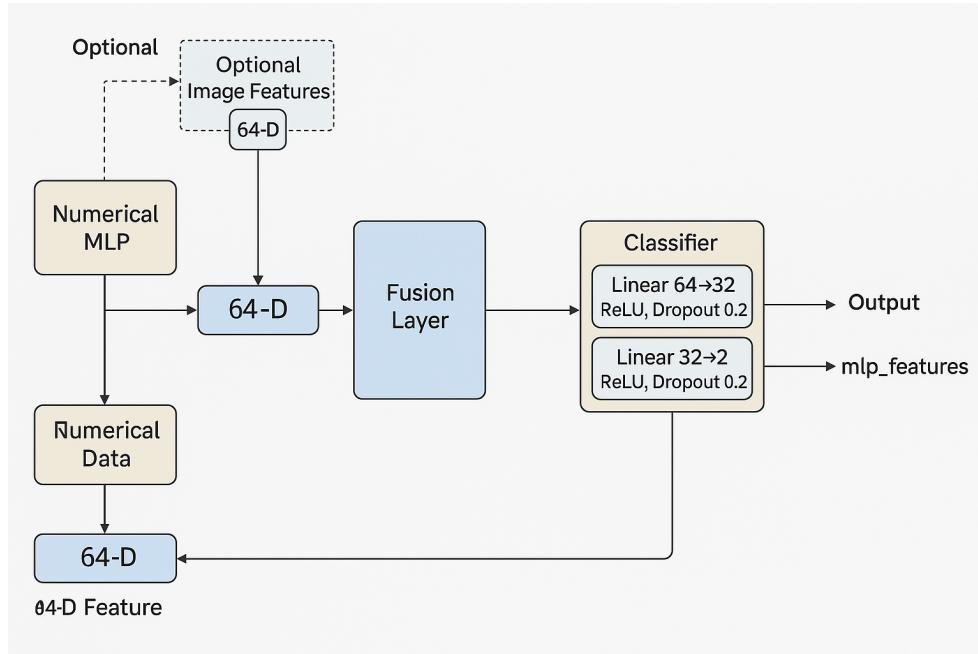


### Giai đoạn 7: Vector đặc trưng đầu ra

- **Đại diện:** vector 2048-chiều (sau GAP, trước FC) cô đọng toàn bộ ngữ nghĩa của ảnh.
- **Thực hành:**  $L_2$ -normalize để so khớp (cosine/Euclid) trong retrieval/ID; hoặc giữ *feature map*  $7 \times 7 \times 2048$  nếu cần thông tin không gian (R-MAC/GeM, attention).

**Ghi chú về residual (áp dụng cho mọi stage)** Mỗi block xuất  $y = x + F(x)$ ; skip giữ biểu diễn cũ, nhánh  $F$  chỉ học phần bổ sung  $\Rightarrow$  đặc trưng được tích lũy ổn định qua nhiều tầng; gradient lan truyền tốt khi huấn luyện. Khi đổi kích thước/kênh giữa các stage, dùng *projection skip*  $1 \times 1$  để cộng đúng kích thước.

### 3.8 Giới thiệu mô hình hợp nhất (Fusion Model)



Hình 14: Mô tả về Fusion Model

Mô hình Fusion được thiết kế để kết hợp thông tin từ hai nguồn dữ liệu khác nhau:

- **Dữ liệu bảng (tabular):** các biến số lâm sàng hoặc chỉ số y tế, ví dụ từ bộ dữ liệu Cleveland.
- **Đặc trưng ảnh (image features):** các đặc trưng trích xuất từ ảnh siêu âm tim (EchoNet-Dynamic) thông qua mạng CNN hoặc ResNet50 đã huấn luyện trước.

**Nhánh MLP (cho dữ liệu bảng).** Dữ liệu bảng đầu vào có kích thước  $d$  được đưa qua mạng MLP ba tầng:

$$d \longrightarrow 128 \longrightarrow 64,$$

tạo ra vector đặc trưng 64 chiều:

$$\mathbf{z}_{MLP} \in \mathbb{R}^{64}.$$

**Nhánh CNN (cho dữ liệu ảnh).** Ảnh siêu âm được xử lý qua ResNet50 (đã bỏ lớp phân loại cuối cùng), sau đó qua một chuỗi tầng giảm chiều:

$$\mathbf{z}_{CNN} \in \mathbb{R}^{64}.$$

**Khối Fusion.** Hai vector đặc trưng  $\mathbf{z}_{MLP}$  và  $\mathbf{z}_{CNN}$  được ghép nối (concatenate):

$$\mathbf{z}_{fusion} = [\mathbf{z}_{MLP}; \mathbf{z}_{CNN}] \in \mathbb{R}^{128}.$$

Sau đó,  $\mathbf{z}_{fusion}$  được đưa qua hai tầng ẩn để giảm chiều và học biểu diễn chung:

$$128 \longrightarrow 64.$$

**Đầu ra (Classification head).** Cuối cùng, vector 64 chiều được đưa qua tầng phân loại:

$$64 \longrightarrow 32 \longrightarrow 2,$$

để dự đoán nhãn *Heart Disease* hoặc *No Heart Disease*.

**Tóm tắt pipeline.**

$$\begin{aligned} \text{Numerical Data} &\xrightarrow{\text{MLP}} \mathbf{z}_{MLP}, & \text{Image Features} &\xrightarrow{\text{CNN}} \mathbf{z}_{CNN}, \\ [\mathbf{z}_{MLP}; \mathbf{z}_{CNN}] &\xrightarrow{\text{Fusion Layers}} \mathbf{z}_{fusion} & \mathbf{z}_{fusion} &\xrightarrow{\text{Classifier}} \hat{y}. \end{aligned}$$

**Lưu ý.** Nếu không có dữ liệu ảnh, nhánh CNN sẽ được thay thế bằng vector 0, nghĩa là chỉ sử dụng nhánh MLP để dự đoán.

### 3.9 Các hàm triển khai

#### 3.9.1 Trích xuất đặc trưng ảnh với ResNet-50

Trong bước đầu tiên, nhóm mình khởi tạo một bộ trích xuất đặc trưng ảnh dựa trên ResNet-50 đã huấn luyện trước (ImageNet). Phần *backbone* được giữ tới layer4 (2048 kênh), loại bỏ avgpool và fc. Đầu ra 2048 chiều được ánh xạ qua *projection head*  $2048 \rightarrow 512 \rightarrow 64$  (ReLU + Dropout), tạo vector 64-D tương thích với nhánh MLP trong mô hình hợp nhất. Ảnh đầu vào được resize về  $224 \times 224$  và chuẩn hoá theo thống kê ImageNet.

```

1 import torch
2 import torch.nn as nn
3 from torchvision import models, transforms
4
5 def initialize_fixed_cnn_model():
6     """Initialize FIXED ResNet-50 CNN for image feature extraction"""
7     print("Initializing FIXED ResNet-50 CNN...")
8
9     resnet = models.resnet50(pretrained=True)
10
11    class FixedCNNFeatureExtractor(nn.Module):
12        def __init__(self):
13            super(FixedCNNFeatureExtractor, self).__init__()
14            # backbone without avgpool & fc
15            self.backbone = nn.Sequential(*list(resnet.children())[:-2])
16            self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
17            self.feature_layer = nn.Sequential(
18                nn.Linear(2048, 512),
19                nn.ReLU(),
20                nn.Dropout(0.2),
21                nn.Linear(512, 64) # 64-D feature
22            )
23
24        def forward(self, x):
25            # x: (B, 3, 224, 224)
26            x = self.backbone(x)          # (B, 2048, 7, 7)
27            x = self.avgpool(x)          # (B, 2048, 1, 1)
28            x = torch.flatten(x, 1)       # (B, 2048)

```

```

29         x = self.feature_layer(x)    # (B, 64)
30         return x
31
32     cnn_model = FixedCNNFeatureExtractor()
33     cnn_model.eval()
34
35     transform = transforms.Compose([
36         transforms.Resize((224, 224)),
37         transforms.ToTensor(),
38         transforms.Normalize(mean=[0.485, 0.456, 0.406],
39                             std=[0.229, 0.224, 0.225])
40     ])
41
42     print("FIXED CNN feature extractor initialized!")
43     print("Architecture: Input(3,224,224) -> ResNet -> 2048 -> 512 -> 64")
44     return cnn_model, transform

```

Code Listing 2: initialize\_fixed\_cnn\_model(): ResNet-50 feature extractor to 64-D

### 3.9.2 Mô hình hợp nhất (Fusion CNN + MLP)

Mô hình hợp nhất kết hợp hai nhánh: (i) nhánh MLP cho dữ liệu bảng ánh xạ  $d \rightarrow 128 \rightarrow 64$  sinh vector  $\mathbf{z}_{\text{MLP}} \in \mathbb{R}^{64}$ ; (ii) nhánh CNN/ResNet sinh  $\mathbf{z}_{\text{CNN}} \in \mathbb{R}^{64}$ . Hai vector được nối  $[\mathbf{z}_{\text{MLP}}; \mathbf{z}_{\text{CNN}}] \in \mathbb{R}^{128}$ , đưa qua khối fusion  $128 \rightarrow 64$  (ReLU, Dropout). Cuối cùng là classifier  $64 \rightarrow 32 \rightarrow 2$  để xuất logits (có bệnh/không bệnh). Khi thiếu ảnh, mô hình tự động ghép thêm một vector không để vẫn sử dụng được nhánh MLP đơn kênh.

```

1 import torch
2 import torch.nn as nn
3
4 class NumericalMLP(nn.Module):
5     """Tabular MLP: in_dim -> 128 -> 64."""
6     def __init__(self, input_dim, hidden_dims=[128, 64], output_dim=64):
7         super().__init__()
8         h1, h2 = hidden_dims
9         self.net = nn.Sequential(
10             nn.Linear(input_dim, h1), nn.ReLU(), nn.Dropout(0.2),
11             nn.Linear(h1, h2), nn.ReLU()
12         )
13         self.out = nn.Linear(h2, output_dim)
14
15     def forward(self, x):
16         return self.out(self.net(x))
17
18 class FeatureFusionModel(nn.Module):
19     """CNN + MLP Feature Fusion Model - FIXED"""
20     def __init__(self, numerical_dim, fusion_dim=128, num_classes=2):
21         super(FeatureFusionModel, self).__init__()
22         self.numerical_mlp = NumericalMLP(
23             input_dim=numerical_dim, hidden_dims=[128, 64], output_dim=64
24         )
25         self.fusion_layer = nn.Sequential(
26             nn.Linear(64 + 64, fusion_dim),
27             nn.ReLU(),
28             nn.Dropout(0.4),

```

```

29         nn.Linear(fusion_dim, 64),
30         nn.ReLU(),
31         nn.Dropout(0.3)
32     )
33     self.classifier = nn.Sequential(
34         nn.Linear(64, 32),
35         nn.ReLU(),
36         nn.Dropout(0.2),
37         nn.Linear(32, num_classes)
38     )
39
40     def forward(self, numerical_data, image_features=None):
41         mlp_features = self.numerical_mlp(numerical_data) # (B, 64)
42         if image_features is not None:
43             fused_features = torch.cat([mlp_features, image_features], dim=1)
44         else:
45             fused_features = torch.cat([mlp_features, torch.zeros_like(
46                 mlp_features)], dim=1)
47         fused_features = self.fusion_layer(fused_features) # (B, 64)
48         output = self.classifier(fused_features) # (B, 2)
49
50     return output, mlp_features

```

Code Listing 3: FeatureFusionModel: MLP + CNN fusion, classifier 64-&gt;32-&gt;2

### 3.9.3 Tiềm xử lý & xây dựng đặc trưng (Feature Engineering)

Dữ liệu bảng được sao chép và tách nhãn `cardio`. Các biến liên tục (`age`, `height`, `weight`, `ap_hi`, `ap_lo`) được ép kiểu số, điền thiếu bằng trung vị; các biến phân loại (`cholesterol`, `gluc`, `gender`, `smoke`, `alco`, `active`) điền bằng giá trị phổ biến nhất. Nhóm mình sửa các mẫu có `ap_hi < ap_lo`, winsorize ở mức 1-99% để giảm ngoại lai. Từ đó tạo các đặc trưng lâm sàng: `age_years`, `bmi`, `pp`, `map`, `sbp_dbp_ratio`, cờ tăng huyết áp, cờ cholesterol/glucose cao, `risk_behaviors`; phân nhóm theo tuổi, BMI, huyết áp; và tương tác như `bmi_x_age`, `pp_x_age`, `htn_x_chol`. Cuối cùng, các biến liên tục chủ đạo được chuẩn hoá bằng `RobustScaler` để ổn định trước ngoại lai.

```

1 import numpy as np
2 import pandas as pd
3 from sklearn.impute import SimpleImputer
4 from sklearn.preprocessing import RobustScaler
5
6 def preprocess_and_engineer_features(df: pd.DataFrame):
7     """
8     Expected columns:
9     ['age', 'height', 'weight', 'gender', 'ap_hi', 'ap_lo', 'cholesterol', 'gluc',
10      'smoke', 'alco', 'active', 'cardio']
11     """
12     df = df.copy()
13
14     target_col = 'cardio'
15     assert target_col in df.columns, f"Missing target column '{target_col}'"
16     y = df[target_col].astype(int)
17     X = df.drop(columns=[target_col, "id"]).copy()
18
19     num_cols = ['age', 'height', 'weight', 'ap_hi', 'ap_lo']

```

```

20     for c in num_cols:
21         X[c] = pd.to_numeric(X[c], errors='coerce')
22
23     num_imputer = SimpleImputer(strategy='median')
24     X[num_cols] = num_imputer.fit_transform(X[num_cols])
25
26     swap = X['ap_hi'] < X['ap_lo']
27     if swap.any():
28         X.loc[swap, ['ap_hi', 'ap_lo']] = X.loc[swap, ['ap_lo', 'ap_hi']].values
29
30     for c in ['height', 'weight', 'ap_hi', 'ap_lo', 'age']:
31         lo, hi = X[c].quantile([0.01, 0.99]).values
32         X[c] = X[c].clip(lo, hi)
33
34     X['age_years'] = np.floor(X['age'] / 365.25).astype(int)
35     h_m = X['height'] / 100.0
36     X['bmi'] = X['weight'] / (h_m**2 + 1e-9)
37
38     X['pp'] = X['ap_hi'] - X['ap_lo']
39     X['map'] = X['ap_lo'] + X['pp'] / 3.0
40     X['sbp_dbp_ratio'] = X['ap_hi'] / (X['ap_lo'] + 1e-6)
41
42     X['is_hypertensive'] = ((X['ap_hi'] >= 140) | (X['ap_lo'] >= 90)).astype(int)
43     X['prehypertensive'] = (((X['ap_hi'] >= 120) & (X['ap_hi'] < 140)) |
44                               ((X['ap_lo'] >= 80) & (X['ap_lo'] < 90))).astype(
45                               int)
46     X['wide_pp'] = (X['pp'] >= 60).astype(int)
47
48     for c in ['cholesterol', 'gluc', 'gender', 'smoke', 'alco', 'active']:
49         if c in X.columns:
50             X[c] = pd.to_numeric(X[c], errors='coerce')
51
52     cat_imputer = SimpleImputer(strategy='most_frequent')
53     X[['cholesterol', 'gluc', 'gender', 'smoke', 'alco', 'active']] = cat_imputer.
54     fit_transform(
55         X[['cholesterol', 'gluc', 'gender', 'smoke', 'alco', 'active']])
56
57     X['chol_high'] = (X['cholesterol'] >= 2).astype(int)
58     X['gluc_high'] = (X['gluc'] >= 2).astype(int)
59     X['risk_behaviors'] = X['smoke'] + X['alco'] + (1 - X['active'])
60
61     X['age_bin'] = pd.cut(
62         X['age_years'], bins=[0, 39, 49, 59, 69, 120], labels=[0, 1, 2, 3, 4]
63     ).astype(int)
64     X['bmi_class'] = pd.cut(
65         X['bmi'], bins=[-np.inf, 18.5, 25, 30, np.inf], labels=[0, 1, 2, 3]
66     ).astype(int)
67     X['bp_category'] = pd.cut(
68         X['ap_hi'], bins=[-np.inf, 120, 130, 140, np.inf], labels=[0, 1, 2, 3]
69     ).astype(int)
70
71     X['bmi_x_age'] = X['bmi'] * X['age_years']
72     X['pp_x_age'] = X['pp'] * X['age_years']
73     X['htn_x_chol'] = X['is_hypertensive'] * X['chol_high']

```

```

73
74     to_scale = [
75         'age', 'age_years', 'height', 'weight', 'ap_hi', 'ap_lo',
76         'bmi', 'pp', 'map', 'sbp_dbp_ratio', 'bmi_x_age', 'pp_x_age'
77     ]
78     to_scale = [c for c in to_scale if c in X.columns]
79
80     scaler = RobustScaler()
81     X_scaled = X.copy()
82     X_scaled[to_scale] = scaler.fit_transform(X_scaled[to_scale])
83
84     print(f"Final feature shape: {X_scaled.shape}")
85     return X_scaled, y, scaler

```

Code Listing 4: preprocess\_and\_engineer\_features(): cleaning + feature engineering

### 3.9.4 Chiết xuất đặc trưng từ EchoNet

Trong bước này, nhóm mình trích xuất các đặc trưng *metadata* tiêu chuẩn từ *FileList.csv* của EchoNet—Dynamic (các biến như EF, ESV, EDV, kích thước khung hình, FPS, số khung). Nhãn nhị phân được gán theo ngưỡng EF: *heart\_disease* = (EF < 50%). Về tiền xử lý, nhóm mình diền khuyết bằng trung vị trên các cột số học, sau đó chuẩn hoá bằng *StandardScaler*. Để đồng bộ với vector đặc trưng 64 chiều từ nhánh CNN (ResNet), các đặc trưng metadata được *pad* ngẫu nhiên giá trị Gaussian (nhỏ, trung bình 0, độ lệch chuẩn 0.01) cho đủ 64 chiều hoặc cắt bớt nếu vượt quá. Ngoài ra, nhóm mình thực hiện một *sanity check* theo lô nhỏ: lấy một khung hình đầu tiên từ video .avi, áp dụng *transform* (chuẩn hoá ImageNet) và đẩy qua extractor CNN (ResNet-50) để xác nhận đường suy diễn ảnh hoạt động bình thường. Nếu không có video/không đọc được khung, pipeline sẽ chỉ dùng đặc trưng metadata.

```

1 import os
2 import cv2
3 import numpy as np
4 import pandas as pd
5 from PIL import Image
6 from collections import Counter
7 from sklearn.preprocessing import StandardScaler
8
9 import torch
10
11 def extract_echonet_features_properly(filelist_df, volume_df, videos_path,
12                                         cnn_model, transform, max_samples=50):
13     """Extract features using REAL EchoNet data - FIXED VERSION"""
14     print(f"Extracting features from EchoNet data (max: {max_samples})...")
15
16     if filelist_df is None:
17         print("No FileList data available")
18         return None, None, None
19
20     # 1) Metadata features + binary label from EF
21     print("Extracting metadata-based features from FileList.csv...")
22     feature_cols = ['EF', 'ESV', 'EDV', 'FrameHeight', 'FrameWidth', 'FPS', 'NumberOfFrames']
23     filelist_df['heart_disease'] = (filelist_df['EF'] < 50).astype(int)
24     available_cols = [col for col in feature_cols if col in filelist_df.columns]

```

```
25     print(f"Available EchoNet features: {available_cols}")
26     if not available_cols:
27         print("No suitable feature columns found!")
28         return None, None, None
29
30     sample_df = filelist_df.head(max_samples).copy()
31     features_data = sample_df[available_cols].fillna(sample_df[available_cols].
median())
32
33     # 2) (Optional) Add volume tracing features if available
34     if volume_df is not None:
35         print("Adding volume tracing features...")
36         # TODO: merge volume_df [...] theo FileName énu àcn
37         pass
38
39     # 3) Normalize and pad/crop to 64-D
40     scaler = StandardScaler()
41     features_normalized = scaler.fit_transform(features_data)
42
43     if features_normalized.shape[1] < 64:
44         padding_size = 64 - features_normalized.shape[1]
45         padding = np.random.normal(0, 0.01, (features_normalized.shape[0],
padding_size))
46         features_final = np.hstack([features_normalized, padding])
47     else:
48         features_final = features_normalized[:, :64]
49
50     labels = sample_df['heart_disease'].values
51
52     print(f"Extracted EchoNet features shape: {features_final.shape}")
53     print(f"Label distribution: {Counter(labels)}")
54     print(f"No NaN in labels: {not np.isnan(labels).any()}")
55
56     # 4) Sanity check: try reading a frame from some videos and pass through CNN
57     if videos_path is not None and cnn_model is not None:
58         print("\nAttempting video feature extraction for validation...")
59
60         video_count = 0
61         for _, row in sample_df.iterrows():
62             filename = str(row['FileName'])
63             video_file = f"{filename}.avi"
64             video_path = os.path.join(videos_path, video_file)
65
66             if os.path.exists(video_path):
67                 try:
68                     cap = cv2.VideoCapture(video_path)
69                     ret, frame = cap.read()
70                     cap.release()
71
72                     if ret and frame is not None and len(frame.shape) == 3:
73                         frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
74                         frame_pil = Image.fromarray(frame_rgb)
75                         frame_tensor = transform(frame_pil).unsqueeze(0)
76
77                     with torch.no_grad():
```

```

78             video_features = cnn_model(frame_tensor)
79             video_count += 1
80             if video_count == 1:
81                 print(f"Video processing test successful! Shape:
82 {video_features.shape}")
83             except Exception:
84                 continue
85
86             if video_count > 0:
87                 print(f"Successfully processed {video_count} video samples")
88             else:
89                 print("Video processing failed - using metadata features only")
90
91     return features_final, labels

```

Code Listing 5: extract\_echonet\_features\_properly(): trích xuất metadata + kiểm tra video frame

### 3.9.5 Tạo bộ dữ liệu đa phương thức (Cleveland + EchoNet)

Do Cleveland và EchoNet là *hai quần thể bệnh nhân khác nhau*, nhóm mình xây dựng hai chiến lược kết hợp thực tế. (i) **Synthetic pairing**: ghép cặp mẫu Cleveland và EchoNet theo *cùng nhau* (bệnh/không bệnh) để tạo cặp giả lập, cân bằng số lượng dương/tính, và trả về *group\_ids* theo chỉ số Cleveland phục vụ Group KFold (tránh rò rỉ). (ii) **Separate cohorts**: dùng hai bộ tách biệt, lấy mẫu EchoNet cắt xuống để khớp kích thước với Cleveland, *pad* chiều về cùng độ dài và huấn luyện song song khi tạo batch. Cả hai chiến lược đều ghi nhận phân bối lớp và kích thước đầu ra, đảm bảo pipeline có thể chuyển đổi linh hoạt tùy bài toán.

```

1 import numpy as np
2 from collections import Counter
3
4 def create_realistic_multimodal_dataset(cleveland_X, cleveland_y,
5                                         echonet_features=None, echonet_labels=None
6                                         ,
6                                         strategy="synthetic_pairing"):
7
8     """
9     Create realistic multimodal dataset acknowledging Cleveland and EchoNet are
10    DIFFERENT patients
11    """
12
13    print("=="*60)
14    print("REALISTIC MULTIMODAL DATASET CREATION")
15    print("=="*60)
16    print("Acknowledging: Cleveland and EchoNet represent different patient
17    populations")
18
19    if echonet_features is None:
20        print("\nStrategy: Cleveland data only")
21        return cleveland_X.values, None, cleveland_y.values, "cleveland_only"
22
23        print(f"\nDataset Statistics:")
24        print(f"- Cleveland: {cleveland_X.shape[0]} patients, {cleveland_y.mean()[:1%]} disease rate")
25        print(f"- EchoNet: {echonet_features.shape[0]} patients, {echonet_labels.mean()[:1%]} disease rate")
26        cle_pos_idx = cleveland_y[cleveland_y == 1].index.tolist()
27        cle_neg_idx = cleveland_y[cleveland_y == 0].index.tolist()

```

```
24
25     echo_pos_idx = np.where(echonet_labels == 1)[0]
26     echo_neg_idx = np.where(echonet_labels == 0)[0]
27
28     print(f"Available for pairing:")
29     print(f"  Cleveland: {len(cle_pos_idx)} positive, {len(cle_neg_idx)} negative")
30     print(f"  EchoNet: {len(echo_pos_idx)} positive, {len(echo_neg_idx)} negative")
31
32     # Balanced counts (optional cap at 60 per class)
33     max_pos = min(len(cle_pos_idx), len(echo_pos_idx))
34     max_neg = min(len(cle_neg_idx), len(echo_neg_idx))
35     n_pairs_per_class = min(max_pos, max_neg, 60)
36
37     if strategy == "synthetic_pairing":
38         paired_cleveland, paired_echonet, paired_labels = [], [], []
39         group_ids = []
40
41         # Positives
42         for i in range(max_pos):
43             cle_idx = cle_pos_idx[i % len(cle_pos_idx)]
44             echo_idx = echo_pos_idx[i % len(echo_pos_idx)]
45             paired_cleveland.append(cleveland_X.iloc[cle_idx].values)
46             paired_echonet.append(echonet_features[echo_idx])
47             paired_labels.append(1)
48             group_ids.append(cle_idx)
49
50         # Negatives
51         for i in range(max_neg):
52             cle_idx = cle_neg_idx[i % len(cle_neg_idx)]
53             echo_idx = echo_neg_idx[i % len(echo_neg_idx)]
54             paired_cleveland.append(cleveland_X.iloc[cle_idx].values)
55             paired_echonet.append(echonet_features[echo_idx])
56             paired_labels.append(0)
57             group_ids.append(cle_idx)
58
59         final_cleveland = np.array(paired_cleveland)
60         final_echonet = np.array(paired_echonet)
61         final_labels = np.array(paired_labels)
62         group_ids = np.array(group_ids)
63         return final_cleveland, final_echonet, final_labels, "synthetic_pairing",
64         group_ids
65
66     elif strategy == "separate_cohorts":
67         print("\nStrategy 2: SEPARATE COHORTS approach")
68         print("Rationale: Use both datasets independently, combine at training
69         time")
70
71         all_cleveland = cleveland_X.values
72         all_cleveland_labels = cleveland_y.values
73
74         n_echo_samples = min(len(echonet_features), len(all_cleveland))
75         echo_indices = np.random.choice(len(echonet_features), n_echo_samples,
76         replace=False)
```

```

74     selected_echonet = echonet_features[echo_indices]
75     selected_echo_labels = echonet_labels[echo_indices]
76
77     # Pad dimension to match
78     if all_cleveland.shape[1] != selected_echonet.shape[1]:
79         max_dim = max(all_cleveland.shape[1], selected_echonet.shape[1])
80         if all_cleveland.shape[1] < max_dim:
81             pad_cle = np.zeros((all_cleveland.shape[0], max_dim -
82             all_cleveland.shape[1]))
83             all_cleveland = np.hstack([all_cleveland, pad_cle])
84             if selected_echonet.shape[1] < max_dim:
85                 pad_echo = np.zeros((selected_echonet.shape[0], max_dim -
86             selected_echonet.shape[1]))
87                 selected_echonet = np.hstack([selected_echonet, pad_echo])
88
89     combined_cleveland = all_cleveland[:n_echo_samples]
90     combined_labels = all_cleveland_labels[:n_echo_samples]
91     group_ids = np.arange(len(combined_labels))
92
93     print(f"Separate cohorts approach:")
94     print(f"  Cleveland subset: {combined_cleveland.shape}")
95     print(f"  EchoNet subset: {selected_echonet.shape}")
96     print(f"  Combined labels: {Counter(combined_labels)}")
97
98     return combined_cleveland, selected_echonet, combined_labels, "separate_cohorts", group_ids
99
100 else:
101     print("Using default synthetic pairing strategy")
102     return create_realistic_multimodal_dataset(cleveland_X, cleveland_y,
103                                                 echonet_features, echonet_labels, "synthetic_pairing")

```

Code Listing 6: create\_realistic\_multimodal\_dataset(): synthetic pairing &amp; separate cohorts

### 3.9.6 Huấn luyện và đánh giá mô hình Fusion

nhóm mình huấn luyện mô hình Fusion (CNN + MLP) với  $k$ -fold cross-validation. Nếu có `group_ids`, dùng GroupKFold để tránh *data leakage*; nếu không, dùng StratifiedKFold để bảo toàn tỉ lệ lớp. Tại mỗi fold, scaler cho tabular và (nếu có) cho đặc trưng ảnh đều được *fit* trên *tập train* rồi áp dụng cho test nhằm tránh rò rỉ thông tin. Mất cân bằng lớp được xử lý bằng *class weights* trong CrossEntropyLoss. Tối ưu sử dụng Adam với `weight_decay` nhẹ và StepLR để hạ dần learning rate. Kết quả mỗi fold (loss/accuracy train, accuracy test, phân bố dự đoán) được lưu lại; sau cùng, nhóm mình gộp dự đoán để báo cáo các chỉ số tổng hợp như Accuracy, Precision, Recall, F1.

```

1 import numpy as np
2 import torch
3 import torch.nn as nn
4 import torch.optim as optim
5
6 from collections import Counter
7 from sklearn.model_selection import StratifiedKFold, GroupKFold
8 from sklearn.preprocessing import StandardScaler
9

```

```
10 def train_fusion_model(
11     aligned_numerical,
12     aligned_cnn_features,
13     aligned_labels,
14     epochs=100,
15     batch_size=7000000,
16     learning_rate=0.001,
17     groups=None,
18     seed=42,
19     n_folds=5
20 ):
21     """Train the CNN+MLP feature fusion model with 5-fold cross-validation."""
22     print("=="*50)
23     print("TRAINING FEATURE FUSION MODEL - 5-FOLD CV")
24     print("=="*50)
25
26     rng = np.random.default_rng(seed)
27     torch.manual_seed(seed)
28
29     indices = np.arange(len(aligned_numerical))
30     y_np = np.asarray(aligned_labels, dtype=int)
31
32     if groups is not None:
33         print(f"Using GroupKFold with {n_folds} folds")
34         cv_splitter = GroupKFold(n_splits=n_folds)
35         splits = cv_splitter.split(indices, y_np, groups=groups)
36     else:
37         print(f"Using StratifiedKFold with {n_folds} folds")
38         cv_splitter = StratifiedKFold(n_splits=n_folds, shuffle=True,
39                                       random_state=seed)
40         splits = cv_splitter.split(indices, y_np)
41
42     fold_results, all_y_test, all_y_pred, all_test_accuracies = [], [], [], []
43     numerical_dim = aligned_numerical.shape[1]
44
45     for fold_idx, (train_idx, test_idx) in enumerate(splits):
46         print(f"\n{'='*50}")
47         print(f"FOLD {fold_idx + 1}/{n_folds}")
48         print(f"{'='*50}")
49
50         fusion_model = create_fusion_model(numerical_dim) # factory =>
51         FeatureFusionModel(...)
52
53         # Fit scalers on TRAIN only
54         num_scaler = StandardScaler().fit(aligned_numerical[train_idx])
55         X_train_num_np = num_scaler.transform(aligned_numerical[train_idx])
56         X_test_num_np = num_scaler.transform(aligned_numerical[test_idx])
57
58         if aligned_cnn_features is not None:
59             cnn_scaler = StandardScaler().fit(aligned_cnn_features[train_idx])
60             X_train_cnn_np = cnn_scaler.transform(aligned_cnn_features[train_idx])
61         ]
62         X_test_cnn_np = cnn_scaler.transform(aligned_cnn_features[test_idx])
63         if fold_idx == 0:
```

```
61             print("Training with both numerical and CNN features (multimodal")
62         ")
63     else:
64         X_train_cnn_np = X_test_cnn_np = None
65         if fold_idx == 0:
66             print("Training with numerical features only")
67
68     # Torch tensors
69     X_train_num = torch.tensor(X_train_num_np, dtype=torch.float32)
70     X_test_num = torch.tensor(X_test_num_np, dtype=torch.float32)
71     y_train = torch.tensor(y_np[train_idx], dtype=torch.long)
72     y_test = torch.tensor(y_np[test_idx], dtype=torch.long)
73
74     if X_train_cnn_np is not None:
75         X_train_cnn = torch.tensor(X_train_cnn_np, dtype=torch.float32)
76         X_test_cnn = torch.tensor(X_test_cnn_np, dtype=torch.float32)
77     else:
78         X_train_cnn = X_test_cnn = None
79
80     print(f"Training set: {len(X_train_num)} samples")
81     print(f"Test set: {len(X_test_num)} samples")
82     print(f"Class distribution - Train: {Counter(y_train.numpy())}")
83     print(f"Class distribution - Test: {Counter(y_test.numpy())}")
84
85     # Class weights
86     class_counts = np.bincount(y_train.numpy(), minlength=2)
87     class_weights = (class_counts.sum() / (class_counts + 1e-6)).astype(np.
float32)
88     class_weights = class_weights / class_weights.mean()
89     criterion = nn.CrossEntropyLoss(weight=torch.tensor(class_weights))
90
91     optimizer = optim.Adam(fusion_model.parameters(), lr=learning_rate,
92     weight_decay=1e-4)
93     scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.5)
94
95     fusion_model.train()
96     train_losses, train_accuracies = [], []
97     n_train = X_train_num.size(0)
98     if batch_size is None or batch_size >= n_train:
99         print(f"Using FULL BATCH training (all {n_train} samples per
iteration)")
100        actual_batch_size = n_train
101    else:
102        print(f"Using MINI-BATCH training (batch_size={batch_size})")
103        actual_batch_size = batch_size
104
105    for epoch in range(epochs):
106        # Shuffle per epoch
107        perm = torch.randperm(n_train)
108        X_train_num_shuffled = X_train_num[perm]
109        y_train_shuffled = y_train[perm]
110        X_train_cnn_shuffled = X_train_cnn[perm] if X_train_cnn is not None
111    else None
112
113        total_loss, correct, total = 0.0, 0, 0
```

```
111     n_batches = (n_train + actual_batch_size - 1) // actual_batch_size
112
113     for b in range(n_batches):
114         s = b * actual_batch_size
115         e = min((b + 1) * actual_batch_size, n_train)
116         batch_num = X_train_num_shuffled[s:e]
117         batch_lbl = y_train_shuffled[s:e]
118         batch_cnn = X_train_cnn_shuffled[s:e] if X_train_cnn_shuffled is
119             not None else None
120
121         optimizer.zero_grad()
122         outputs, _ = fusion_model(batch_num, batch_cnn)
123         loss = criterion(outputs, batch_lbl)
124         loss.backward()
125         optimizer.step()
126
127         total_loss += loss.item()
128         _, pred = outputs.max(1)
129         total += batch_lbl.size(0)
130         correct += (pred == batch_lbl).sum().item()
131
132         scheduler.step()
133         epoch_loss = total_loss / n_batches
134         epoch_acc = 100.0 * correct / total
135         train_losses.append(epoch_loss)
136         train_accuracies.append(epoch_acc)
137
138         if (epoch + 1) % 20 == 0:
139             print(f"Epoch [{epoch+1}/{epochs}] Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.2f}%")
140
141     # Eval on test
142     fusion_model.eval()
143     with torch.no_grad():
144         test_outputs, test_features = fusion_model(X_test_num, X_test_cnn)
145         test_probs = torch.softmax(test_outputs, dim=1)
146         _, test_predicted = test_outputs.max(1)
147         test_accuracy = (test_predicted == y_test).float().mean().item()
148
149     # Store fold results
150     fold_result = {
151         "fold": fold_idx + 1,
152         "model": fusion_model,
153         "test_accuracy": test_accuracy,
154         "y_test": y_test.cpu().numpy(),
155         "y_pred": test_predicted.cpu().numpy(),
156         "y_probs": test_probs.cpu().numpy(),
157         "train_losses": train_losses,
158         "train_accuracies": train_accuracies,
159         "final_train_loss": train_losses[-1],
160         "final_train_acc": train_accuracies[-1],
161         "num_scaler": num_scaler,
162         "cnn_scaler": cnn_scaler if aligned_cnn_features is not None else
None
163     }
```

```
163     fold_results.append(fold_result)
164     all_y_test.extend(y_test.cpu().numpy())
165     all_y_pred.extend(test_predicted.cpu().numpy())
166     all_test_accuracies.append(test_accuracy)
167
168     print(f"\nFold {fold_idx + 1} Results:")
169     print(f"  Test Accuracy: {test_accuracy:.4f}")
170     print(f"  Final Training Loss: {train_losses[-1]:.4f}")
171     print(f"  Final Training Accuracy: {train_accuracies[-1]:.2f}%")
172
173
174 # Summary
175 print(f"\n{'='*50}")
176 print("CROSS-VALIDATION RESULTS SUMMARY")
177 print(f"{'='*50}")
178
179 mean_accuracy = np.mean(all_test_accuracies)
180 std_accuracy = np.std(all_test_accuracies)
181
182 print(f"\nTest Accuracy across {n_folds} folds:")
183 for i, acc in enumerate(all_test_accuracies):
184     print(f"  Fold {i+1}: {acc:.4f}")
185 print(f"\nMean Test Accuracy: {mean_accuracy:.4f} ± {std_accuracy:.4f}")
186 print(f"Min Test Accuracy: {np.min(all_test_accuracies):.4f}")
187 print(f"Max Test Accuracy: {np.max(all_test_accuracies):.4f}")
188
189 from sklearn.metrics import accuracy_score, precision_score, recall_score,
190 f1_score
191 overall_accuracy = accuracy_score(all_y_test, all_y_pred)
192 overall_precision = precision_score(all_y_test, all_y_pred, average='binary',
193 zero_division=0)
194 overall_recall = recall_score(all_y_test, all_y_pred, average='binary',
195 zero_division=0)
196 overall_f1 = f1_score(all_y_test, all_y_pred, average='binary',
197 zero_division=0)
198
199 print(f"\nOverall Metrics (aggregated from all folds):")
200 print(f"  Accuracy: {overall_accuracy:.4f}")
201 print(f"  Precision: {overall_precision:.4f}")
202 print(f"  Recall: {overall_recall:.4f}")
203 print(f"  F1-Score: {overall_f1:.4f}")
204
205 results = {
206     "fold_results": fold_results,
207     "mean_test_accuracy": mean_accuracy,
208     "std_test_accuracy": std_accuracy,
209     "all_test_accuracies": all_test_accuracies,
210     "all_y_test": np.array(all_y_test),
211     "all_y_pred": np.array(all_y_pred),
212     "overall_accuracy": overall_accuracy,
213     "overall_precision": overall_precision,
214     "overall_recall": overall_recall,
215     "overall_f1": overall_f1,
216     "n_folds": n_folds,
217     "best_fold": np.argmax(all_test_accuracies) + 1,
```

```

214     "best_accuracy": np.max(all_test_accuracies),
215     "worst_fold": np.argmax(all_test_accuracies) + 1,
216     "worst_accuracy": np.min(all_test_accuracies)
217   }
218   return results

```

Code Listing 7: train\_fusion\_model(): 5-fold CV, chuẩn hoá trong-fold, class weights, tổng hợp kết quả

### 3.10 Kết quả

Bảng 4 trình bày báo cáo chi tiết kết quả phân loại của mô hình trên tập kiểm thử. Mô hình đạt **accuracy = 0.87**, với **macro F1-score = 0.86** và **weighted F1-score = 0.87**. Điều này cho thấy hiệu năng tổng thể khá tốt và cân bằng giữa các lớp.

Bảng 4: Báo cáo phân loại chi tiết của mô hình

Class	Precision	Recall	F1-score	Support
No Disease	0.84	0.97	0.90	32
Heart Disease	0.94	0.74	0.83	23
Accuracy			0.87	55
Macro Avg	0.89	0.85	0.86	55
Weighted Avg	0.88	0.87	0.87	55

#### Phân tích theo từng lớp:

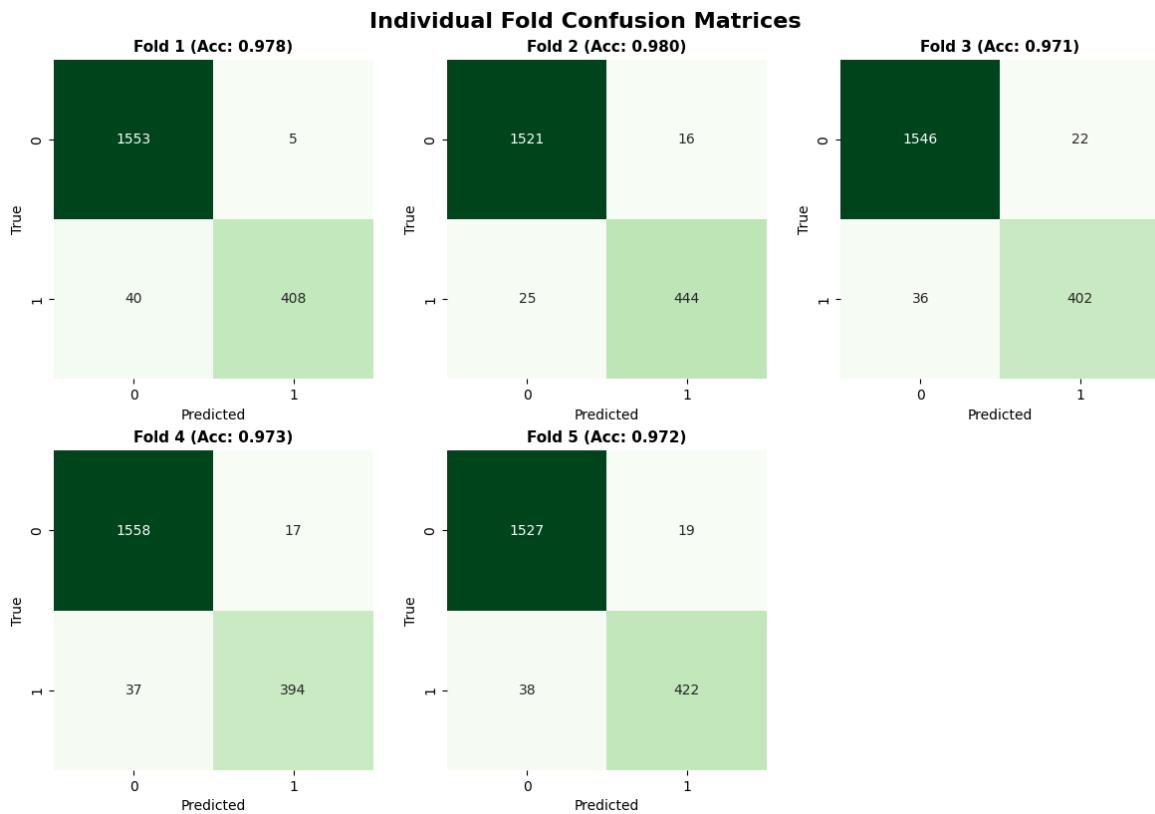
- **No Disease:** Precision = 0.84, Recall = 0.97. Mô hình hầu như không bỏ sót người khỏe, chỉ có một số ít bị dự đoán nhầm là bệnh. F1-score = 0.90, thể hiện khả năng phân loại lớp này rất tốt.
- **Heart Disease:** Precision = 0.94, cho thấy khi mô hình dự đoán có bệnh thì độ tin cậy rất cao. Tuy nhiên, Recall = 0.74, nghĩa là vẫn còn bỏ sót khoảng 26% bệnh nhân thực sự có bệnh. Đây là điểm yếu chính cần cải thiện vì trong bối cảnh y khoa, bỏ sót bệnh nhân thường nguy hiểm hơn báo động giả.

**Ý nghĩa trong chẩn đoán bệnh tim:** Mô hình hiện phù hợp làm bộ lọc sàng lọc bước đầu, vì khi dự đoán có bệnh thì độ chính xác rất cao. Tuy nhiên, cần điều chỉnh để tăng Recall ở lớp bệnh nhầm giảm nguy cơ bỏ sót bệnh nhân. Các hướng cải thiện bao gồm:

1. Điều chỉnh ngưỡng quyết định (ví dụ từ 0.5 xuống 0.4) để ưu tiên tăng Recall.
2. Sử dụng hàm mất mát có trọng số hoặc focal loss để phạt nặng các trường hợp bỏ sót bệnh.
3. Áp dụng các phương pháp hiệu chuẩn xác suất (Platt/Isotonic) và tối ưu ngưỡng trên đường cong ROC/PR.
4. Bổ sung đặc trưng từ dữ liệu lâm sàng hoặc hình ảnh để cung cấp thêm thông tin cho mô hình.

Ngay cả khi nhóm nghiên cứu triển khai với một tập dữ liệu quy mô lớn hơn 70,000 mẫu cùng 11 biến đầu vào và 1 biến mục tiêu, mô hình vẫn đạt được hiệu năng rất tốt. Kết quả của 5-fold cross-validation được thể hiện thông qua ma trận nhầm lẫn ở Hình 15, với độ chính xác

trung bình của các fold dao động từ 0.971 đến 0.980, cho thấy tính ổn định và khả năng khái quát cao của mô hình.



Hình 15: Ma trận nhầm lẫn của từng fold khi huấn luyện với hơn 70,000 mẫu dữ liệu.

Dữ liệu bao gồm ba nhóm đặc trưng chính:

- **Objective features:** thông tin khách quan như tuổi, chiều cao, cân nặng, giới tính.
- **Examination features:** kết quả thăm khám y khoa như huyết áp tâm thu, huyết áp tâm trương, cholesterol, glucose.
- **Subjective features:** thông tin chủ quan từ bệnh nhân, ví dụ như thói quen hút thuốc, sử dụng rượu bia, và mức độ hoạt động thể chất.

Chi tiết các biến trong tập dữ liệu:

Bảng 5: Mô tả chi tiết các biến trong tập dữ liệu

Tên biến	Nhóm	Mô tả	Kiểu dữ liệu
Age	Objective	Tuổi (tính bằng ngày)	int
Height	Objective	Chiều cao (cm)	int
Weight	Objective	Cân nặng (kg)	float
Gender	Objective	Giới tính (mã số)	categorical
ap_hi	Examination	Huyết áp tâm thu	int
ap_lo	Examination	Huyết áp tâm trương	int
Cholesterol	Examination	Cholesterol (1: bình thường, 2: cao, 3: rất cao)	categorical
Gluc	Examination	Glucose (1: bình thường, 2: cao, 3: rất cao)	categorical
Smoke	Subjective	Hút thuốc	binary
Alco	Subjective	Uống rượu bia	binary
Active	Subjective	Hoạt động thể chất	binary
Cardio	Target	Có bệnh tim mạch (0/1)	binary

### 3.10.1 Nhận xét kết quả

Với dữ liệu lớn và đa dạng, mô hình không chỉ đạt độ chính xác cao mà còn duy trì được sự cân bằng trong phân loại giữa hai lớp (không bệnh và bệnh tim mạch). Điều này chứng minh:

1. Tính ổn định của mô hình khi mở rộng quy mô dữ liệu.
2. Các đặc trưng y khoa cơ bản như huyết áp, cholesterol, BMI và hành vi sinh hoạt (smoking, alcohol, activity) là những yếu tố quan trọng giúp mô hình học được mối quan hệ mạnh mẽ với nguy cơ bệnh tim mạch.
3. Kết quả khẳng định tiềm năng ứng dụng của mô hình trong các hệ thống hỗ trợ quyết định lâm sàng, đặc biệt là sàng lọc và dự báo nguy cơ tim mạch trong cộng đồng.

## 4 API cho hướng mở rộng 2 - CardioFusion: Ghép đa phương thức EchoNet + Cleveland cho phân loại bệnh tim

### 4.1 API là gì và vì sao lại cần API ?

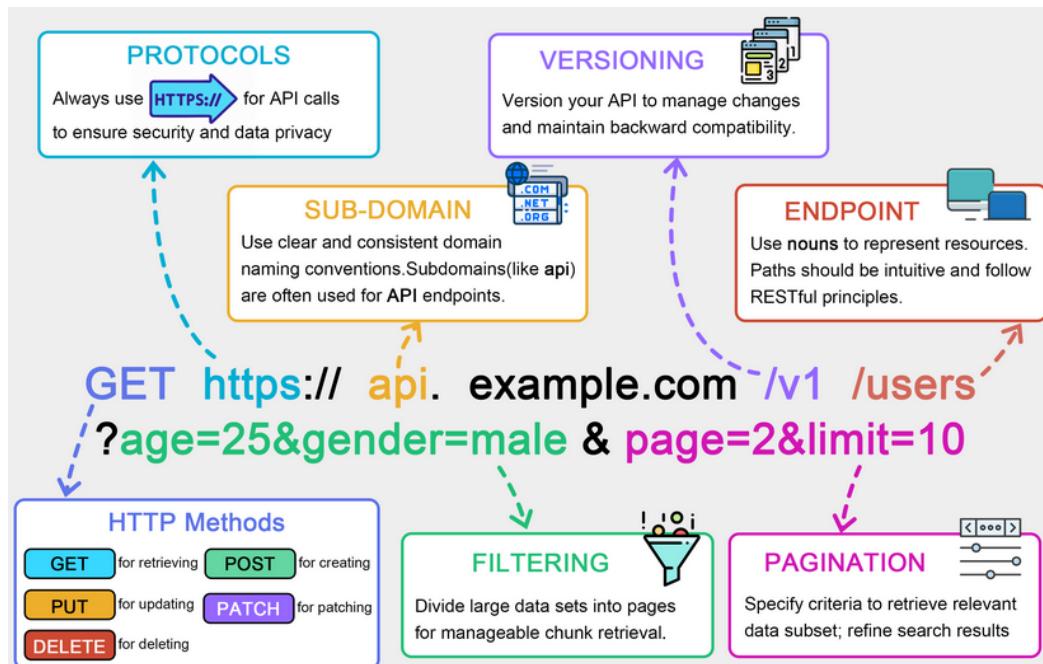
Nếu frontend là giao diện giúp đơn giản hóa trong giao tiếp giữa người và máy, thì API (Application Programming Interface) chính là giao diện của backend giúp đơn giản hóa giao tiếp giữa máy và máy qua internet. Và trong tất cả các kiến trúc API, REST là kiến trúc dễ hiểu và thông dụng nhất.

Ví dụ đơn giản cho phương thức GET của API:

```

1 from fastapi import FastAPI
2 app = FastAPI()
3
4 @app.get("/")
5 async def root():
6     return {"message": "Hello World"}
```

Code Listing 8: API cơ bản là 1 hàm nhưng được viết đè lên, giúp hàm đó có thể được truy cập ở bất cứ đâu



Hình 16: RESTfulAPI URL Structure

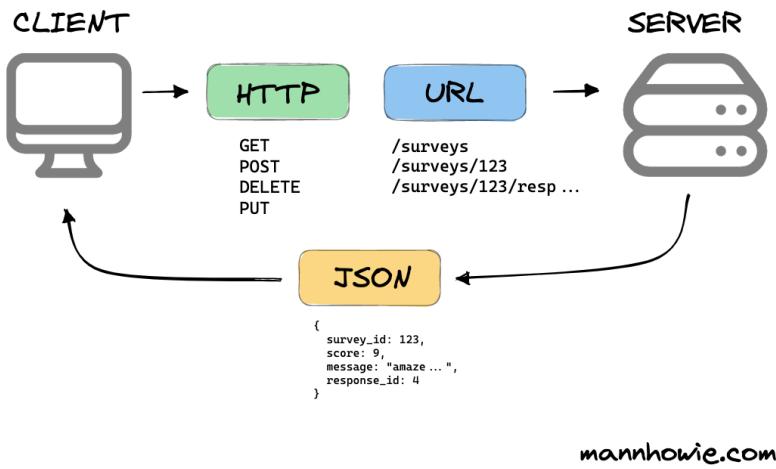
#### 4.1.1 API có gì ?

Để xử lý dữ liệu (i.e. kiểu Đối Tượng) hiệu quả, 1 hàm cần có 5 phương thức đó là Lấy (GET), Tạo (POST), Cập nhật toàn bộ (PUT), Xóa (DELETE) và cập nhật 1 số phần nhất định (PATCH). REST API có 2 đối tượng chính là làm nhiệm vụ giao tiếp với nhau đó là Client và Server, cả 2 đều là thiết bị điện tử có khả năng giao tiếp qua internet (e.g. máy tính, điện thoại). Trong đó:

**Client (có thể là 1 hoặc nhiều)** có nhiệm vụ gửi yêu cầu và nhận kết quả. Yêu cầu gửi đi có thể là 1 trong 5 phương thức GET, POST, PUT, DELETE và PATCH. Với GET và DELETE cấu trúc của 1 URL đầy đủ sẽ giống như sau:

- protocols là giao thức chung để truy cập internet có thể là http (không bảo mật) hoặc https (s cho secure, bảo mật)
- versioning định danh phiên bản khác nhau của code.
- sub-domain là 1 tên Miền độc nhất giúp định danh các trang web với nhau, giống như số đỗ.
- end-point là 1 tên Hàm độc nhất giúp định danh các Hàm trong code với nhau.
- filtering (lọc) đóng vai trò truyền dữ liệu theo cặp key:value như SQL, nếu phương thức có khả năng cập nhật dữ liệu như POST, PUT và PATCH các biến sau dấu '?' sẽ là tham số của hàm.
- pagination (phân chia trang) đóng vai trò chọn lọc số lượng dữ liệu đã được xử lý. (e.g. limit = 10 sẽ trả về 10 dữ liệu).

## WHAT IS A REST API?



Hình 17: REST API

**Server** có nhiệm vụ nhận url làm đầu vào rồi xử lý và trả về kết quả. Trong ngữ cảnh mình đang code 1 API Backend, thì máy tính của mình chính là 1 Server vì nó có khả năng nhận, xử lý và trả về dữ liệu.

Trước khi đi sâu hơn về API cho dự đoán bệnh tim trong thời gian thực, việc cách lưu trữ Weight sau huấn luyện là 1 bước không thể thiếu.

### 4.2 Lưu trữ Weight của mô hình cho Inference

Sau khi huấn luyện, mô hình lưu weights vào file .pth và scalers cho từng fold (RobustScaler cho dữ liệu số, StandardScaler cho các đặc trưng CNN) vào file .pkl bằng Pickle, giúp đảm bảo preprocessing nhất quán. Các dữ liệu được lưu bao gồm:

- Fusion Model weights (.pth) - Toàn bộ parameters đã học của mô hình Fusion (CNN + MLP)
- Numerical scaler (.pkl) - Mean/Std của 28 features (age, BMI, BP...) trong tập dữ liệu Sulianova.
- CNN feature scaler (.pkl) - Mean/Std của 64 CNN features (của frame trong mỗi video) trong tập dữ liệu EchoNet.

Khi inference, mô hình fusion có weight tốt nhất trong các fold sẽ được chọn và đồng nhất dữ liệu qua chuẩn hóa với cùng mean/median/IQR như tập huấn luyện.

```

1 if 'fusion_results' in locals() and fusion_results is not None:
2     # create a directory to save models
3     save_dir = 'models/'
4     os.makedirs(save_dir, exist_ok=True)
5
6     # save weights for each fold's model
7     for fold_result in fusion_results['fold_results']:

```

```

8     fold_num = fold_result['fold']
9     model = fold_result['model']
10
11     # save model weights (state_dict)
12     model_path = os.path.join(save_dir, f'fusion_model_fold_{fold_num}.pth')
13     torch.save(model.state_dict(), model_path)
14     print(f"Saved model weights for Fold {fold_num} to {model_path}")
15
16     # optional, save scalers for preprocessing (needed for inference)
17     if fold_result['num_scaler'] is not None:
18         scaler_path = os.path.join(save_dir, f'num_scaler_fold_{fold_num}.pkl')
19     )
20         import pickle
21         with open(scaler_path, 'wb') as f:
22             pickle.dump(fold_result['num_scaler'], f)
23         print(f"Saved numerical scaler for Fold {fold_num} to {scaler_path}")
24
25     if fold_result['cnn_scaler'] is not None:
26         scaler_path = os.path.join(save_dir, f'cnn_scaler_fold_{fold_num}.pkl')
27     )
28         with open(scaler_path, 'wb') as f:
29             pickle.dump(fold_result['cnn_scaler'], f) # type: ignore
30         print(f"Saved CNN scaler for Fold {fold_num} to {scaler_path}")
31
32     # optional, save only the best fold's model (based on highest test accuracy)
33     best_fold = fusion_results['best_fold']
34     best_model = fusion_results['fold_results'][best_fold - 1]['model']
35     best_model_path = os.path.join(save_dir, 'best_fusion_model.pth')
36     torch.save(best_model.state_dict(), best_model_path)
37     print(f"Saved best model weights (Fold {best_fold}) to {best_model_path}")
38
39     print("All model weights saved successfully!")
else:
    print("fusion_results not found. Run train_fusion_model first.")

```

Code Listing 9: initialize\_fixed\_cnn\_model(): save model weight for each fold with pickle

### 4.3 Cấu trúc folder tổng quan

Để phát triển API, nhóm mình chia thành 5 phần chính là Thủ Nghiệm (với heart\_prediction là code thử nghiệm chính), API (tổng hợp các hàm từ heart\_prediction thành các API Request), Data (dữ liệu tải về từ kaggle) và Models (chứa weight và dữ liệu được chuẩn hóa).

Trong folder chính API, folder utility tổng hợp các hàm trong heart\_prediction.ipynb và thống nhất các import qua file `__init__.py`. Với main.py là file chứa các API Request, main.py có thể truy cập các chức năng thông qua folder utility '(e.g. from utility import func1, func2,...)' chứ không phải gọi chính xác từng files một. Ngoài ra, việc dùng `__init__.py` còn giúp chuẩn hóa đường dẫn của file (file\_path) thông qua main, nghĩa là mọi file\_path sẽ bắt đầu từ main kể cả khi nó có được khai báo ở trong folder utility đi chăng nữa. Tóm lại, `__init__.py` giúp đơn giản hóa việc import lặp khai báo đường dẫn file.

```

1 Project/
2 heart_prediction.ipynb      # Notebook training & testing
3 cleveland.csv                # Dataset (backup)

```

```

4  fusion_results.json          # 5-folds result
5  requirements.txt            # Dependencies
6
7  api/                         # PRODUCTION API
8      main.py                  # FastAPI endpoints
9      utility/                # Core logic
10     __init__.py              # relative path
11     model.py                # Model architecture
12     load_model.py           # Load weights & scalers
13     load_data.py            # Load datasets
14     preprocessing.py        # Preprocessing pipeline
15     predict.py              # Prediction logic
16     feature_engineer.py    # Feature engineering
17     test/                   # Unit tests
18     start_api.bat/sh       # Startup scripts
19     Documentation/
20         API_USAGE_GUIDE.md
21         VIDEO_SELECTION_GUIDE.md
22         CODE_DOCUMENTATION_SUMMARY.md
23
24 data/                         # PREPROCESSED DATA
25     echonet_features.npy    # CNN features (10000×64)
26     echonet_labels.npy      # Ground truth labels
27     echonet_features2.npy   # Alternative features
28
29 models/                        # TRAINED MODELS Weights
30     best_fusion_model.pth  # Best model weights
31     fusion_model_fold_*.pth # 5-fold models
32     num_scaler_fold_*.pkl  # Numerical scalers
33     cnn_scaler_fold_*.pkl  # CNN feature scalers
34
35 frontend/                     # Streamlit demo (comming soon)
36 README.md

```

Code Listing 10: save model weight for each fold with pickle

#### 4.4 Kiến trúc tổng quan của API

Hệ thống API được xây dựng trên FastAPI framework, cung cấp giao diện RESTful để thực hiện dự đoán bệnh tim mạch dựa trên mô hình học sâu đa phương thức (multimodal fusion). Quy trình inference của API được thiết kế với 5 bước xử lý tuần tự:

- Khởi tạo mô hình:** Tải trọng số đã được huấn luyện tốt nhất của mô hình fusion (best\_fusion\_model.pth) cùng với các bộ chuẩn hóa dữ liệu (StandardScaler) từ thư mục /models. Mô hình fusion bao gồm hai nhánh xử lý song song: nhánh MLP cho dữ liệu số và nhánh CNN cho đặc trưng video.
- Tiền xử lý dữ liệu đầu vào:** Chuyển đổi 13 features thô từ dữ liệu bệnh nhân (tuổi, giới tính, chiều cao, cân nặng, huyết áp, cholesterol, glucose, hút thuốc, uống rượu, hoạt động thể chất) thành 28 features sau khi áp dụng feature engineering. Các features mới bao gồm BMI, MAP (Mean Arterial Pressure), PP (Pulse Pressure), các tương tác phi tuyến ( $BMI \times age$ ,  $PP \times age$ ), và các ngưỡng lâm sàng (hypertension flags, cholesterol/glucose thresholds). Sau đó, áp dụng StandardScaler để chuẩn hóa dữ liệu theo phân phối chuẩn ( $\mu = 0, \sigma = 1$ ).

**3. Trích xuất đặc trưng video:** Hệ thống cung cấp hai phương thức để người dùng chọn video siêu âm tim từ bộ dữ liệu EchoNet-Dynamic (10,030 videos):

- GET /videos: Liệt kê danh sách video với phân trang (pagination), mỗi video kèm metadata như EF (Ejection Fraction), số khung hình, FPS, và thông tin phân chia tập dữ liệu (train/val/test).
- GET /videos/search?query=<pattern>: Tìm kiếm video theo tên file sử dụng khớp chuỗi con không phân biệt hoa thường (case-insensitive substring matching).

Tên video được người dùng cung cấp (ví dụ: 0X1005D03EED19C65B.avi) sẽ được ánh xạ sang chỉ số video\_index thông qua hàm `get_video_index_from_filename()`. Đặc trưng CNN 64 chiều của video tương ứng sẽ được trích xuất từ file đã lưu sẵn (`echonet_features.npy`) để tránh tính toán lại, giảm thời gian phản hồi từ vài giây xuống milliseconds.

**4. Dự đoán qua mô hình:** API nhận yêu cầu POST tại endpoint `/predict` với đầu vào JSON chứa dữ liệu bệnh nhân và `video_index/video_filename`. Mô hình fusion thực hiện forward pass với hai đầu vào song song:

- Numerical features (28 dims) → MLP [128, 64] → 64-dim embedding
- CNN features (64 dims) → Identity mapping → 64-dim embedding

Hai embeddings được ghép (concatenate) thành vector 128 chiều, qua lớp fusion [128, 64], và cuối cùng qua classifier [64, 32, 2] để đưa ra dự đoán nhị phân (Disease/No Disease).

**5. Trả về kết quả:** API trả về response dưới dạng JSON có cấu trúc chuẩn theo Pydantic model `PredictionResponse`:

- `prediction`: Nhãn dự đoán (0=No Disease, 1=Disease)
- `prediction_label`: Nhãn dễ đọc ("Disease"/"No Disease")
- `disease_probability`: Xác suất mắc bệnh  $P(Y = 1|X)$  từ softmax layer
- `confidence`: Độ tin cậy =  $\max(p, 1 - p)$  với  $p$  là `disease_probability`
- `patient_id, video_index`: Metadata để truy vết
- `ground_truth`: Nhãn thực tế (nếu có) để đánh giá

## 4.5 API End Points

```
endpoints
/predict      "POST - Make a prediction"
/predict/batch "POST - Make batch predictions"
/videos        "GET - List available videos in the echo dataset"
/videos/search "GET - Search videos by filename pattern (like regex)"
/health         "GET - Check API health (api available or not)"
/docs          "GET - Interactive API documentation"
```

Chức năng nổi bật - endpoint predict:

```
1 {
2     "patient_data": {
3         "id": 1, # Patient ID
4         "active": 1, # Physical activity
```

```

5      "age": 18, # Age in year
6      "alco": 0, # Alchoho usage 0=no, 1=yes
7      "ap_hi": 110, # Systolic blood pressure (huyet áp tâm thu)
8      "ap_lo": 80, # Diastolic blood pressure (huyet áp tâm truong)
9      "cardio": 0, # Cardiovascular disease (0=no, 1=yes)
10     "cholesterol": 1, # Cholesterol level (1=normal, 2=above normal, 3=well
11        above normal)
12     "gender": 2, # Gender (1=female, 2=male)
13     "gluc": 1, # Glucose level (1=normal, 2=above normal, 3=well above normal
14        )
15     "height": 168, # Height in cm
16     "smoke": 0, # Smoking (0=no, 1=yes)
17     "weight": 62 # Weight in kg
18   },
19   "video_index": 2           # Option 1: choose video by index
20 # OR
21   "video_filename": "0X10...avi" # Option 2: choose by video filename which
22     map back to video_id
23 }
```

Code Listing 11: High Light Feature

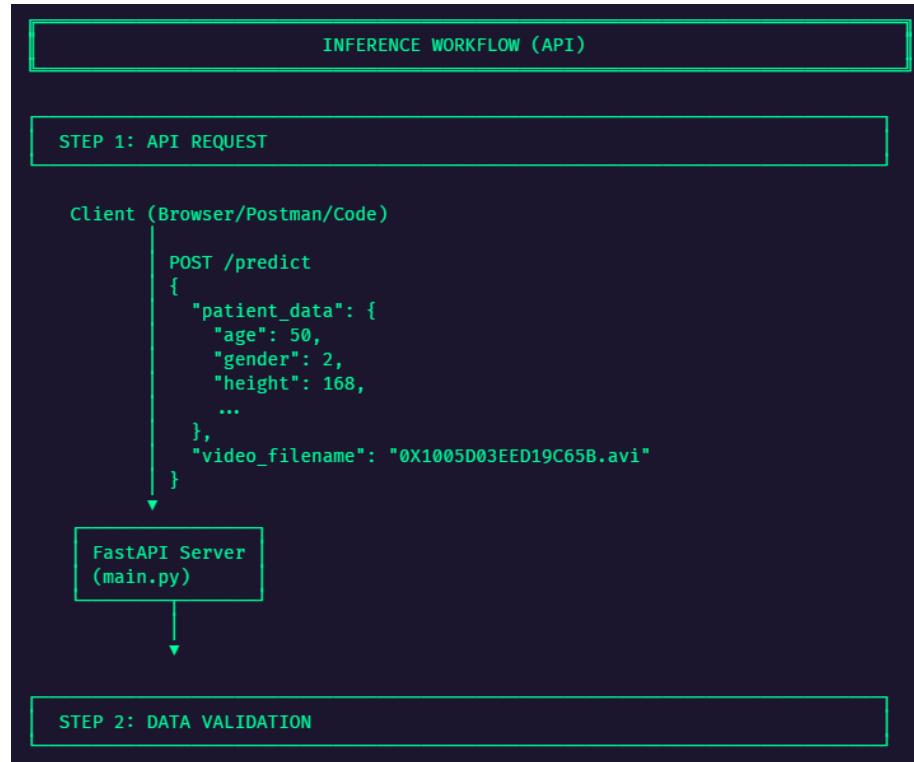
Có patient\_data là một Pydantic Object (i.e. chuỗi JSON) dùng để lưu trữ thông tin đầu vào trước khi đưa vào dự đoán. **Models**

- **PatientData:** 13 trường dữ liệu tương tự như trong dataset huấn luyện
- **PredictionRequest, video\_index:** Metadata để truy vết
- **PredictionResponse:** Nhận dự đoán (0=No Disease, 1=Disease)

Note: Đầu vào /predict có thể sử dụng 'video\_index' (0-10029) hoặc 'video\_filename' (e.g. '0X1005D03EED19C65B.avi')"

## 4.6 API Main Workflow (Inference workflow)

### Bước 1: API Request



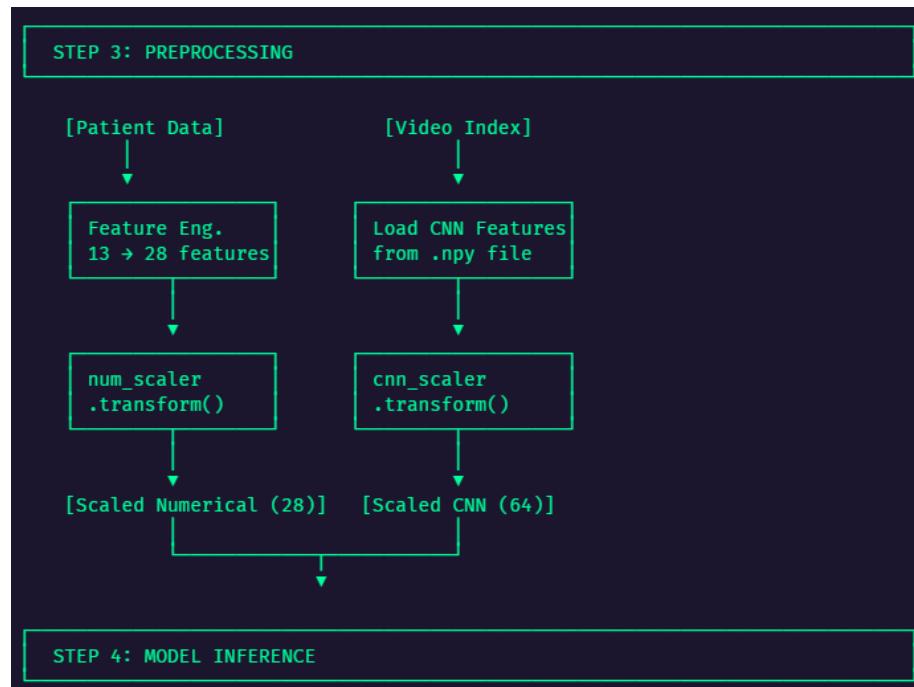
Đầu vào là 1 POST request với đầu vào là 1 Padantic Model (i.e. chuỗi JSON) gồm 13 trường dữ liệu do người dùng nhập từ giao diện API Swagger làm tham số cho phương thức (method) /predict.

## Bước 2: Data Validation



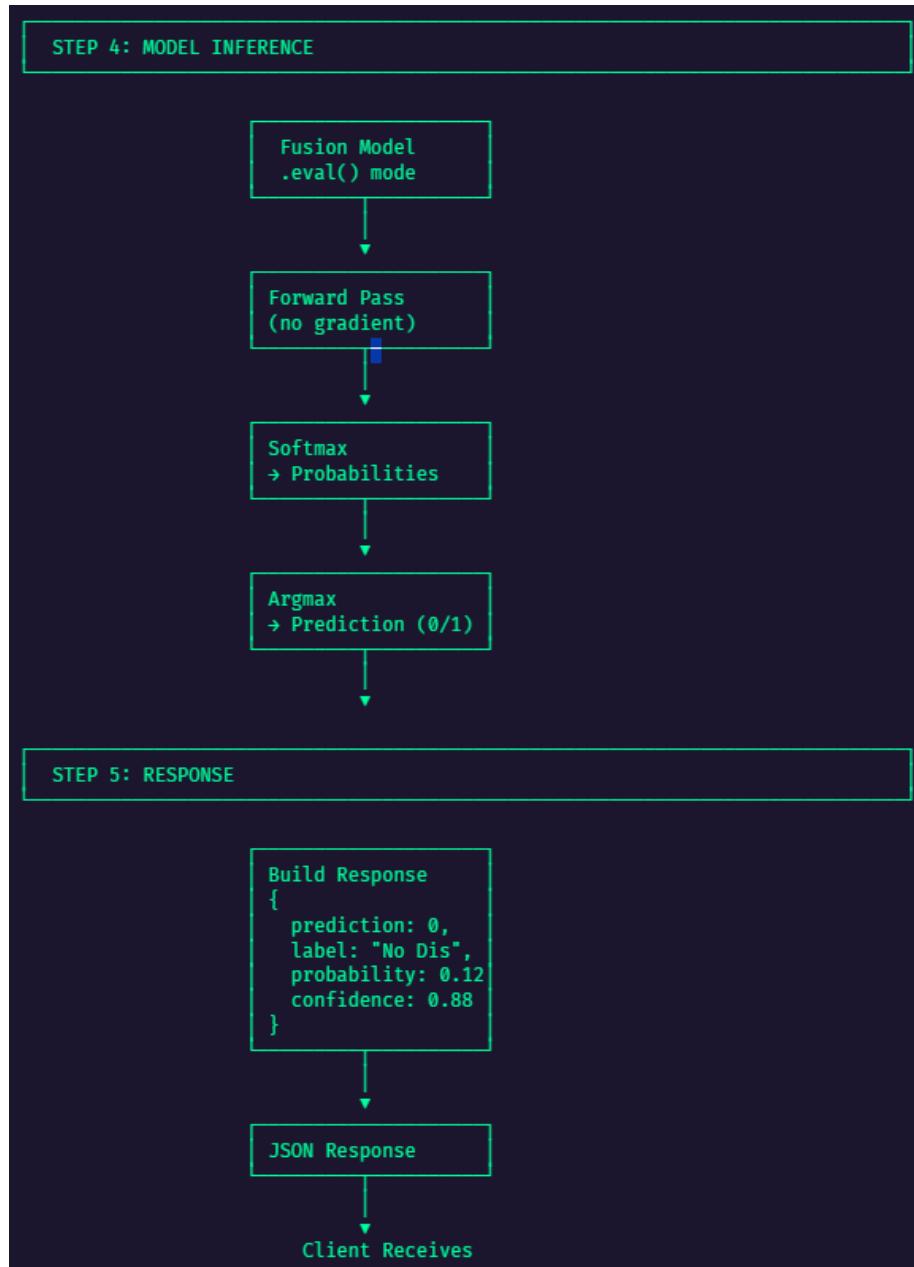
Trước khi đi vào xử lý, dữ liệu trước tiên sẽ được xử lý với Age từ số năm sang số ngày tuổi, kiểm tra các trường dữ liệu có nằm trong khoảng giá trị hợp lý hay không. Nếu file video được chọn thay vì nhập video trực tiếp, tên file sẽ được ánh xạ về video id.

### Bước 3: Preprocessing



Trong bước tiền xử lý để đảm bảo tính nhất quán trong inference và training, num\_scaler cho dữ liệu số trong Pandas Object và cnn\_scaler cho dữ liệu ảnh (nếu có) được lưu khi huấn luyện trên 1 fold sẽ được lấy để chuẩn hóa dữ liệu đầu vào.

## Bước 4 + 5: Model Inference and Response



Trong chạy dự đoán, các tham số được đưa vào Fusion Model rồi thực hiện forward pass để ghép các đặc trưng từ dữ liệu bảng (i.e. Pandatic Object) và dữ liệu ảnh (Echo dataset's image) nếu có, đầu ra là xác suất dự đoán cho 2 nhãn có hoặc không, nhãn có xác suất lớn hơn sẽ được chọn để kết quả dự đoán trong JSON đầu ra.

Cấu trúc JSON đầu ra có bao gồm:

```

1  {
2      "prediction": 0,
3      "prediction_label": "No Disease",
4      "disease_probability": 0.14276719093322754,

```

```
5     "confidence": 0.8572328090667725 ,
6     "patient_id": 1,
7     "video_index": 4,
8     "ground_truth": 0,
9     "numerical_features_shape": [
10         1,
11         28
12     ],
13     "cnn_features_shape": [
14         1,
15         64
16     ],
17     "device": "cuda"
18 }
```

Code Listing 12: /predict endpoint output