# LeNet-5 Model Tutorial

Nguyễn Trường Giang

Ngày 19 tháng 1 năm 2025

## Mục lục

# 1 Giới thiệu về tài liệu này.

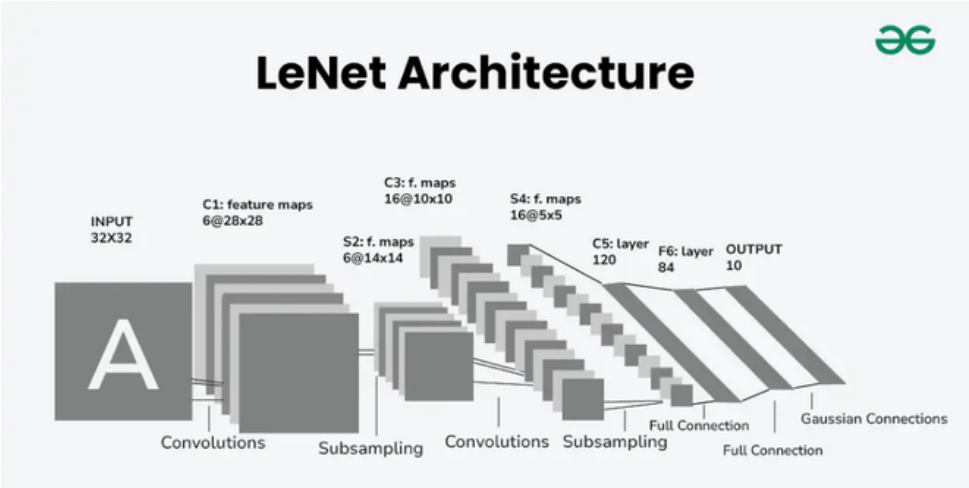
- Đây là tài liệu được tổng hợp bởi nhiều nguồn, tóm tắt và viết nên trong quá trình học của mình. Do đó nếu xảy ra sai sót mong mọi người rộng lượng bỏ qua :>

# 2 Giới thiệu về LeNet-5.

- LeNet-5 là một cấu trúc mạng nơ-ron tích chập (Convolutional Neural Network - CNN) được phát triển bởi *Yann LeCun* vào năm 1998. Đây là một trong những mô hình CNN đầu tiên, đặt nền tảng cho sự phát triển mạnh mẽ của Deep Learning trong lĩnh vực nhận dạng hình ảnh.

# 3 Cấu trúc tổng quan của LeNet-5.

- LeNet-5 được thiết kế chủ yếu để nhận dạng các chữ số viết tay và các hình ảnh nhỏ. Kiến trúc của LeNet-5 bao gồm 7 lớp (không tính các lớp input), trong đó có các lớp tích chập (**Convolutional** Layer), lớp **pooling**:gộp lại (hay lớp lấy mẫu - Subsampling Layer), và các lớp kết nối đầy đủ (Fully Connected Layer)




Hình 1: Kiến trúc của LeNet-5

## Các thành phần chính của LeNet-5:

1. Input Layer:

| Layer | | Feature Map | Size | Kernel Size | Stride | Activation |
|---|---|---|---|---|---|---|
| Input | Image | 1 | 32x32 | - | - | - |
| 1 | Convolution | 6 | 28x28 | 5x5 | 1 | tanh |
| 2 | Average Pooling | 6 | 14x14 | 2x2 | 2 | tanh |
| 3 | Convolution | 16 | 10x10 | 5x5 | 1 | tanh |
| 4 | Average Pooling | 16 | 5x5 | 2x2 | 2 | tanh |
| 5 | Convolution | 120 | 1x1 | 5x5 | 1 | tanh |
| 6 | FC | - | 84 | - | - | tanh |
| Output | FC | - | 10 | - | - | softmax |

Hình 2: Các thông tin, đặc điểm của một mô hình LeNet-5 mẫu

- Kích thước hình ảnh đầu vào: $32 \times 32 \times 1$ (1 channel $\rightarrow$ hình ảnh grayscale).

2. C1 - Convolutional Layer (Lớp tích chập):
   - 6 filters:bộ lọc, kích thước $5 \times 5$.
   - Kích thước đầu ra: $28 \times 28 \times 6$.
   - Tính toán: sử dụng convolutional giữa hình ảnh đầu vào và các filters, sau đó áp dụng **sigmoid** activation function.
3. S2 - Subsampling Layer (lớp lấy mẫu):
   - Phương pháp: Average Pooling (lấy mẫu trung bình).
   - Kích thước kernel $2 \times 2$, stride 2 (stride: độ dài mỗi bước).
   - Kích thươc đầu ra: $14 \times 14 \times 6$.
   - Mục đích: Giảm kích thước không gian của dữ liệu, giảm độ phức tạp tính toán.
4. C3 - Convolutional Layer:
   - 16 filters, size $5 \times 5$.
   - output size: $10 \times 10 \times 16$.
   - Kết nối phức tạp: KKhoong phải tất cả các map đầu ra đều được kết nối với tất cả các map đầu vào.
5. S4 - Subsampling Layer:
   - Phương pháp: *Average Pooling*.
   - Kernel size: $2 \times 2$, stride 2.
   - Output size: $5 \times 5 \times 16$.
6. C5 - Fully Connected Convolutional Layer:
   - 120 neurons (fully connected với output từ layer trước).
   - Mỗi neuron được kết nối với toàn bộ map đầu vào $5 \times 5 \times 16$.
7. F6 - Fully Connected Layer:
   - 84 neurons.
   - use the sigmoid activation function.
8. Output Layer:
   - 10 neurons (mỗi neuron đại diện cho một số từ 0 đến 9).
   - **softmax** activation function is used to convert:chuyển đổi the output into probability:xác suất. (Cấu trúc: `tobe + used + to + Vinf`: được sử dụng để làm gì...)

# 4 Một số giới hạn.

- Hoạt động không hiệu quả đối với dữ liệu lớn hoặc phức tạp (vd như ảnh RGB kích thước lớn).
- Một số các kĩ thuật tiên tiến hơn như **ResNet**, **VGG**, hay **AlexNet** đã thay thế **LeNet** trong các ứng dụng hiện đại.

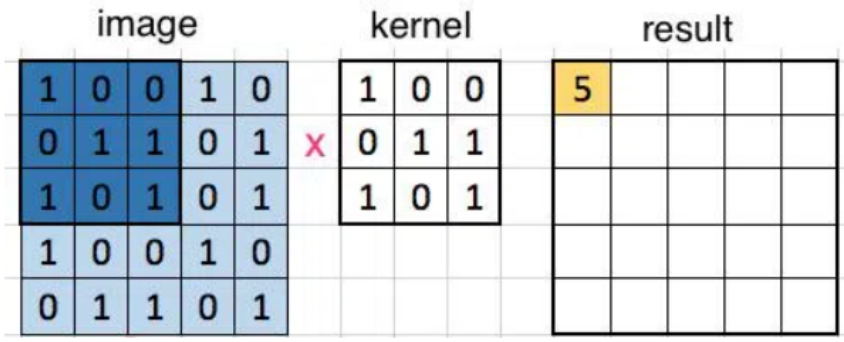
# 5 Mô tả chi tiết về các Layer.

## 5.1 Convolutional Layer.

### 5.1.1 Khái niệm:

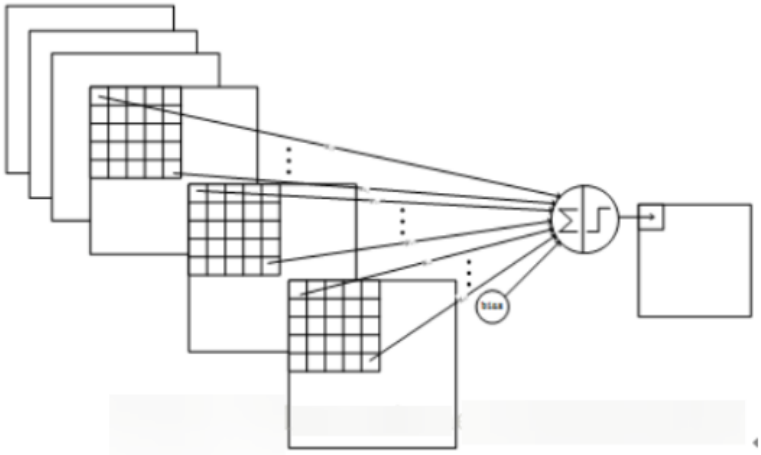
- Convolutional Layer thực hiện các phép **convolution** giữa dữ liệu đầu vào và một hoặc nhiều **filters**, còn được gọi là **kernel**. Phép toán này giúp phát hiện các mẫu hoặc đặc trưng cục bộ trong dữ liệu đầu vào.

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$




Hình 3: Mô tả cách convolutional layer hoạt động

- kernel (như ví dụ trên kernel có kích thước $3 \times 3$) sẽ trượt trên bộ dữ liệu ban đầu để cho ra một ma trận đầu ra có kích thước mới.



Hình 4: Trực quan cách convolution hoạt động

### 5.1.2 Quá trình hoạt động

1. Input:
   - Thường là một hình ảnh (size: $H \times W \times C$ với $H$ là chiều cao, $W$ là chiều rộng, $C$ là số kênh màuchannel)
   - Ví dụ: grayscale: $100 \times 200 \times 1$, hoặc RGB: $69 \times 96 \times 3$.
2. Kernel (bộ lọc):
   - là ma trận nhỏ với kích thước cố định (VD: $3 \times 3$, $5 \times 6$).
   - Mỗi kernel có *tham số học được* trong quá trình training.
   - Kernel được trượt trên toàn bộ dữ liệu đầu vào với bước nhảy **stride** để tính toán.
3. phép toán Convolution:
   - Kernel được đặt trên một vùng nhỏ của dữ liệu đầu vào.
   - Tính tích vô hướng giữa giá trị từng phần tử của kernel và vùng tương ứng trong input data.
   - Kết quả được ghi lại trên một ô của output feature map.
4. Activation function:

- Sau phép tích chập (convolution), giá trị đầu ra thường đưa qua một hàm kích hoạt (vd như ReLu) để tăng tính **phi tuyến tính**.

5. Feature Maps (bản đồ đặc trưng):
   - Mỗi kernel tạo ra một feature map, thể hiện một loại mẫu đặc trưng trong dữ liệu (vd như cạnh, góc, họa tiết, ...).

### 5.1.3 Các tham số quan trọng trong Convolutional Layer.

- Kernel size: (vd: $3 \times 3$).
- Stride: Số bước kernel di chuyển trên tập dữ liệu đầu vào.
- Padding: Thêm biên của dữ liệu đầu vào để thay đổi kích thước đầu ra.
- số lượng filters: là số kernel được áp dụng trên dữ liệu đầu vào, mỗi kernel tạo ra một feature map riêng.

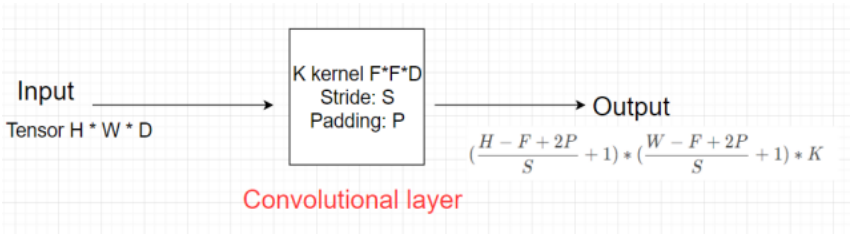
### 5.1.4 Output size của Convolutional Layer.

- Output size được tính bằng công thức:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size} + 2 \times \text{Padding}}{\text{Stride}}$$

**Ví dụ:** Giả sử input của một convolutional layer tổng quát là một tensor có size = $H \times W \times D$, Kernel có size = $F \times F \times D$ (kernel luôn có D: depth bằng depth của input và F là số lẻ, depth = channel), stride: $S$, padding $P$, Convolutional layer được áp dụng có $K$ kernel. Output là tensor 3 chiều có size:

$$(\frac{H - F + 2P}{S} + 1) \times (\frac{W - F + 2P}{S} + 1) \times K$$




Hình 5: Mô tả kĩ hơn về cách tính output size

### 5.1.5 Cú pháp:

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)
```

### 5.1.6 Ví dụ:

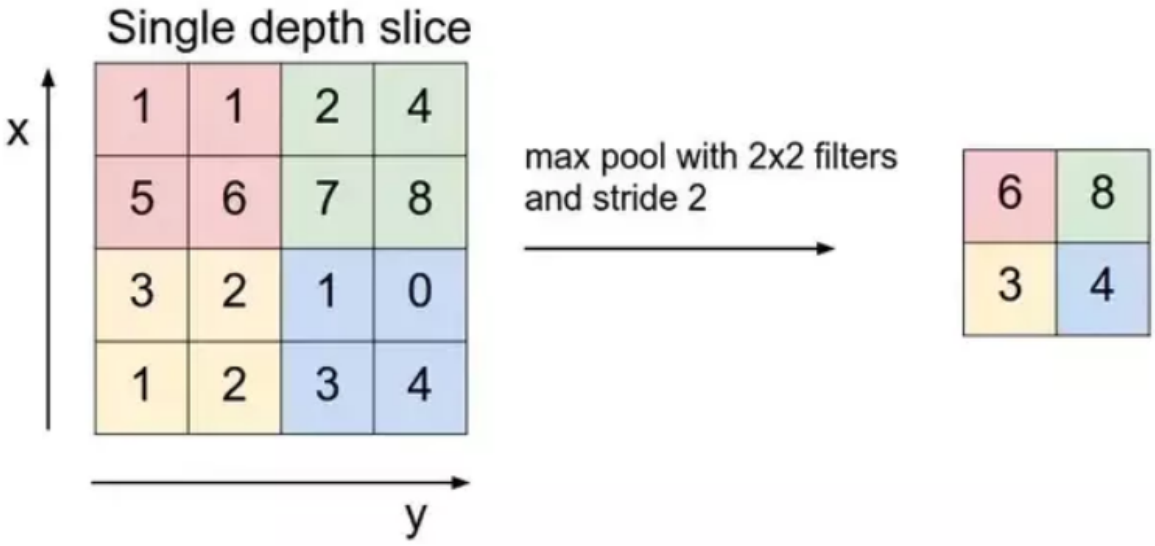
```
import torch
import torch.nn as nn

# Make convolutional layer
conv = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)

# input data: batch_size=1, channels=3, height=32, width=32
input_tensor = torch.randn(1, 3, 32, 32)

# Aplly convolution
output = conv(input_tensor)
print(output.shape)  # Output: torch.Size([1, 16, 32, 32])
```

## 5.2 Polling layer.

### 5.2.1 Khái niệm:

- Pooling layer sẽ giảm bớt số lượng tham số trong trường hợp hình ảnh quá lớn. Các pooling có nhiều loại khác nhau:

  - Max Pooling
  - Average Pooling
  - Sum Pooling




Hình 6: Mô tả về Pooling layer loại **Max Pooling**

### 5.2.2 Cú pháp:

```
# MaxPooling Layer
torch.nn.MaxPool2d(kernel_size, stride=None, padding=0)

# AveragePooling Layer
torch.nn.AvgPool2d(kernel_size, stride=None, padding=0)
```

### 5.2.3 Ví dụ:

```
import torch
import torch.nn as nn

# Max Pooling layer with kernel_size=2, stride=2
maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

# Input tensor: batch_size=1, channels=1, height=4, width=4
input_tensor = torch.tensor([[[[1, 2, 3, 4],
                               [5, 6, 7, 8],
                               [9, 10, 11, 12],
                               [13, 14, 15, 16]]]], dtype=torch.float32)

output = maxpool(input_tensor)
print(output)
# Output: tensor([[[[ 6.,  8.],
#                   [14., 16.]]]])
```

#### 5.2.4 Ứng dụng

- **Max Pooling:** Tăng cường các đặc trưng quan trọng → Nhấn mạnh các đặc trưng nổi bật.
- **Average:** Làm mịn hoặc giảm nhiễu trong đặc trưng → Làm giảm ảnh hưởng của đặc trưng mạnh.

### 5.3 Hàm nn.Flatten()

#### 5.3.1 Khái niệm:

- Hàm nn.Flatten() trong PyTorch sử dụng để chuyển một tensor nhiều chiều thành tensor một chiều. Thường sử dụng để chuyển đổi từ các lớp tích chập (CNN layers) sáng Fully Connected layers.

#### 5.3.2 Cú pháp:

```
    torch.nn.Flatten(stard_dim = 1, end_dim = -1)
```

- **Tham số start_dim** có giá trị mặc định là 1: Chỉ định chiều đầu tiên mà quá trình flatten:làm phẳng bắt đầu (thường = 1 để chừa chiều đầu tiên là batch).
- **Tham số end_dim** có giá trị mặc định là $-1$: Nghĩa là flatten tất cả các chiều từ $start\_dim$ đến chiều cuối cùng.

#### 5.3.3 Ví dụ:

- Tensor đầu vào có kích thước $(N, C, H, W))$ (batch size, channels, height, width).
- Tensor đầu ra có kích thước $(N, C \times H \times W)$ để phù hợp khi đưa vào các lớp fully connected.

```
import torch
import torch.nn as nn

# Initialize the Flatten layer
flatten = nn.Flatten()

# Input Tensor: batch_size=2, channels=3, height=4, width=4
input_tensor = torch.randn(2, 3, 4, 4)

# Apply Flatten
output = flatten(input_tensor)
print(output.shape)
# Output: torch.Size([2, 48])  (3*4*4 = 48)
```

### 5.4 Hàm nn.Linear()

#### 5.4.1 Khái niệm:

- Hàm nn.Linear() trong PyTorch được dùng để tạo một Linear Layer, hay còn gọi là Fully Connected Layer. Giúp trích xuất và két hợp các đặc trưng từ các lớp trước đó.

#### 5.4.2 Cú pháp:

```
torch.nn.Linear(in_features, out_features, bias=True)
```

**Giải thích tham số:**

- in_features: Số chiều của dữ liệu đầu vào (vd: nếu đầu vào là một vector $x \in R^{128}$ thì in_features=128.
- out_features: Số chiều dữ liệu đầu ra.
- bias: mặc định là True.

### 5.4.3 Công thức hoạt động:

$$y = xW^T + b$$

- $x$: Dữ liệu đầu vào, size = $(N, in\_features)$.
- $W$: Ma trận trọng số học được, size = $(out\_features, in\_features)$
- $b$: Vector bias (nếu có), size = $(out\_features)$.
- $y$: dữ liệu đầu ra, size = $(N, out\_features)$.

### 5.4.4 Ví dụ:

```python
import torch
import torch.nn as nn

# Linear layer from 10 input features to 5 output features
linear = nn.Linear(10, 5)

input_tensor = torch.randn(3, 10)  # Batch size=3, 10 input features
output = linear(input_tensor)
print(output.shape)  # Output: torch.Size([3, 5])
```

# 6 Một số hàm cần thiết cho việc build LeNet-5 model.

**Ví dụ bạn đang train model với lossfn: MSELoss và optimizer: SGD:**

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Make a simple model
model = nn.Linear(1, 1)
loss_fn = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# Training data
inputs = torch.tensor([[1.0], [2.0], [3.0]])
targets = torch.tensor([[2.0], [4.0], [6.0]])

# Train model
for epoch in range(100):
    optimizer.zero_grad()  # Reset gradient
    outputs = model(inputs)  # Predict
    loss = loss_fn(outputs, targets)  # caculate loss
    loss.backward()  # caculate gradient
    optimizer.step()  # update weights

    print(f"Epoch [{epoch+1}/100], Loss: {loss.item():.4f}")
```

**Giải thích một số hàm:**

1. optimizer.zero_grad():
   - Mục đích: Để đặt lại giá trị gradient của tất cả các tham số về 0 trước đi tính gradient mới.
   - Vì trong PyTorch, gradiend được cộng dồn sau mỗi lần gọi loss.backward(). Do đó nếu không đặt lại về 0, thì lần tìm gradient sau sẽ bị ảnh hưởng.
2. loss.backkward()
   - Mục đích: tính gradient của hàm loss với từng tham số của mình.

- Pytorch sẽ tự động dựng một biểu đồ tính toán từ khi forward, sau đó dự trên quy tắc backpropagation để tìm gradient và lưu trong thuộc tính .grad.

3. optimizer.step()
   - Mục đích: Cập nhập trọng số của mô hình dựa trên gradient đã tính trước đó (do đó, phải gọi loss.backward() trước đó).
   - Opimzier sẽ lấy giá trị gradient từ .grad và áp dụng công thức tối ưu hóa để điều chỉnh trọng số.

## 6.1 Các hàm Loss function phổ biến:

### 6.1.1 nn.MSELoss (Mean Squared Error Loss

**Meaning:**

- Tính lỗi bình phương trung bình giữa predict và target.
- Thường dùng trong cacs bài toán hồi quy.

**Formula:** $MSELoss(predict, target) = \frac{1}{N}\Sigma_{i=1}^{N}(predict_i - target_i)^2$.

```
import torch
import torch.nn as nn

loss_fn = nn.MSELoss(reduction='mean')  # Or 'sum', 'none'

predicted = torch.tensor([0.5, 0.8], requires_grad=True) # predict value from model
target = torch.tensor([0.6, 1.0]) # target value

loss = loss_fn(predicted, target)
print("Loss:", loss.item())

loss.backward()  # calc gradient
```

### 6.1.2 nn.L1Loss (Mean Absolute Error Loss)

**Meaning:**

- Tính giá trị tuyệt đối trung bình giữa predict và target.
- Thường được dùng trong các bài toán hồi quy khi ta muốn giảm tác động của các ngoại lệ (outliers).

**Formula:** $L1Loss(predict, target) = \frac{1}{N}\Sigma_{i=1}^{N}|predict_i - target_i|$.

```
loss_fn = nn.L1Loss(reduction='mean')  # ặHoc 'sum', 'none'
```

### 6.1.3 nn.CrossEntropyLoss

**Meaning:**

- Kết hợp giữa Softmax và Negative Log-Likelihood Loss.
- Dược dùng nhiều trong các bài toán phân loại nhiều lớp.

**Formula:** $CrossEntropyLoss(predict, target) = \frac{1}{N}\Sigma_{i=1}^{N} log\frac{e^{predict_{i,target_i}}}{\Sigma_j e^{predict_{i,j}}}$.

```
loss_fn = nn.CrossEntropyLoss(weight=None, ignore_index=-100, reduction='mean')
```

## 6.2 Các thuật toán optimizer phổ biến:

### 6.2.1 optim.SGD

```
import torch
import torch.optim as optim

# Let's say you already have a model
optimizer = optim.SGD(
    model.parameters(),  # Model parameters need to be optimized
    lr=0.01,             # learning rate
    momentum=0.9,        # momentum(ddong luong)
    weight_decay=0.0001  # Regularization (L2)
)
```

- optim.SGD là một trong những thuật toán tối ưu hóa cơ bản và phổ biến nhất trong học máy. Thuật toán này sử dụng Gradient Descent ngẫu nhiên (Stochastic Gradient Descent) để cập nhật trọng số của mô hình dựa trên gradient của hàm loss đối với các tham số mô hình.

### 6.2.2 optim.Adam (Adaptive Moment Estimation)

- optim.Adam là một thuật toán tối ưu hóa tiên tiến, kết hợp giữa hai kỹ thuật:

- Momentum: Tăng tốc hội tụ bằng cách tích lũy gradient trung bình.
- RMSProp: Điều chỉnh learning rate theo từng tham số dựa trên trung bình bình phương của gradient.

**Meaning:**

- Tự động điều chỉnh learning rate trong suốt quá trình tối ưu hóa.
- Phù hợp với nhiều bài toán khác nhau, đặc biệt là mạng nơ-ron sâu, vì nó không yêu cầu tinh chỉnh learning rate quá nhiều.

**Cú pháp:**

```
import torch.optim as optim

optimizer = optim.Adam(
    model.parameters(),  # Model parameters need to be optimized
    lr=0.001,            # Learning rate
    betas=(0.9, 0.999),  # momentum
    eps=1e-08,           # Avoid dividing by zero
    weight_decay=0       # Regularization
)
```

### 6.2.3 optim.AdamW

- Thường được sử dụng trong các mô hình lớn như Transformer (tạm thời skip).

# 7 Một số Activation function phi tuyến phổ biến.

## Khái quát:

1. **ReLU:** Lớp ẩn, mô hình phổ biến, bài toán nhận diện hình ảnh, xử lý văn bản.
2. **Leaky ReLU:** Lớp ẩn, khi ReLU gặp vấn đề ”dead neurons”.
3. **Sigmoid:** Lớp đầu ra cho bài toán phân loại nhị phân (xác suất nằm trong [0, 1]).
4. **Tanh:** Lớp ẩn, khi dữ liệu cần trung tâm hóa xung quanh 0; bài toán RNN hoặc chuỗi thời gian.
5. **Softmax:** Lớp đầu ra cho bài toán phân loại đa lớp (xác suất từng lớp).

- Sử dụng ReLU hoặc Leaky ReLU cho các lớp ẩn.
- Chọn Softmax hoặc Sigmoid cho lớp đầu ra tùy thuộc vào loại bài toán phân loại.
- Sử dụng Tanh nếu cần giá trị trung tâm hóa xung quanh 0, đặc biệt trong các mô hình RNN.

## 7.1 ReLu activation function:

**Phạm vi đầu ra:**

$[0, \infty)$.

**Khi nào nên dùng:**

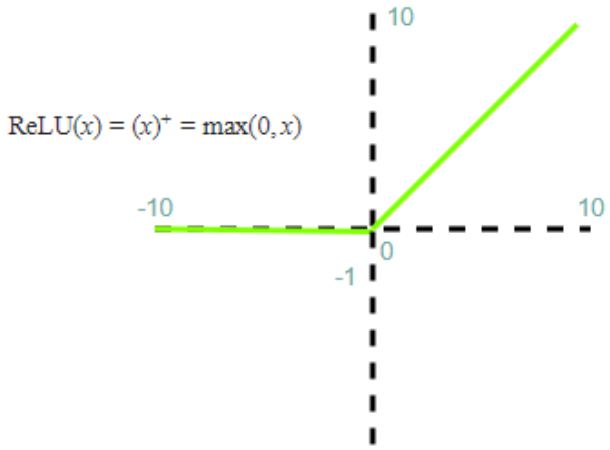
- Lựa chọn phổ biến cho hidden layer trong mạng neurons do tính đơn giản và hiệu quả.
- Thích hợp khi bạn cố tình muốn một số neurons bị **dead neurons** nếu giá trị đầu vào là âm.
- thường sử dụng trong bài toán nhận diện hình ảnh, xử lý văn bản, xử lý chuỗi thời gian.

**Ưu điểm:**

Tính toán đơn giản và hiệu quả. Giảm vấn đề gradient vanishingbiến mất so với hàm Sigmoid/Tanh.

**Nhược điểm:**

Gặp vấn đề **dead neurons** khi gradient trở về 0 vĩnh viễn.




Hình 7: ReLu activation function

**Formula:** $ReLU(x) = max(0, x)$

```python
import torch
import torch.nn as nn

# defining relu
r = nn.ReLU()

# Creating a Tensor with an array
input = torch.Tensor([1, -2, 3, -5])

# Passing the array to relu function
output = r(input)

print(output) # Output: tensor([1., 0., 3., 0.])
```

## 7.2 Leaky ReLU acvitation function:

**Phạm vi đầu ra:**

$(-\infty, \infty)$.

**Khi nào nên dùng:**

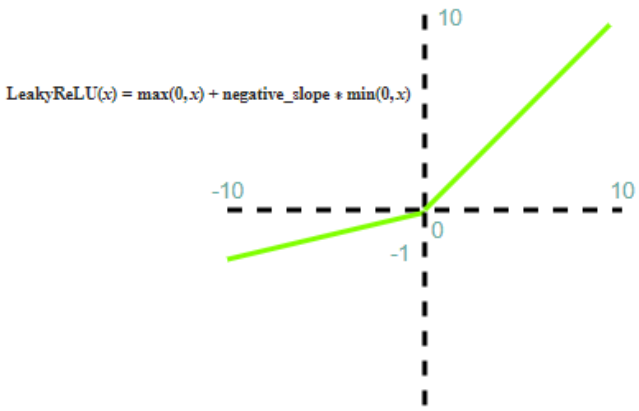
- Khi đang dùng ReLU nhưng gặp vấn đề **dead neurons**.
- Trong những mô hình phức tạp, hoặc khi gradient bị mất trong một số neurons.
- Thường dùng trong **Deep Convolutional Neural Networks (CNNS)**.

**Ưu điểm:**

Giảm thiểu vấn đề **dead neurons**, tính toán đơn giản như ReLU.

**Nhược điểm:**

Cần phải điều chỉnh hệ số $\alpha$ cho phù hợp với bài toán.




Hình 8: Leaky ReLU activation function

**Formula:** $LeakyReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$

```python
import torch
import torch.nn as nn

# defining Lrelu and the parameter 0.2 is passed to control the negative slope ; a
    =0.2
r = nn.LeakyReLU(0.2)

# Creating a Tensor with an array
input = torch.Tensor([1,-2,3,-5])

output = r(input)

print(output) # Output: tensor([ 1.0000, -0.4000,  3.0000, -1.0000])
```

## 7.3 Sigmoid activation function:

**Phạm vi đầu ra:**

$[0, 1]$

**Khi nào nên dùng:**

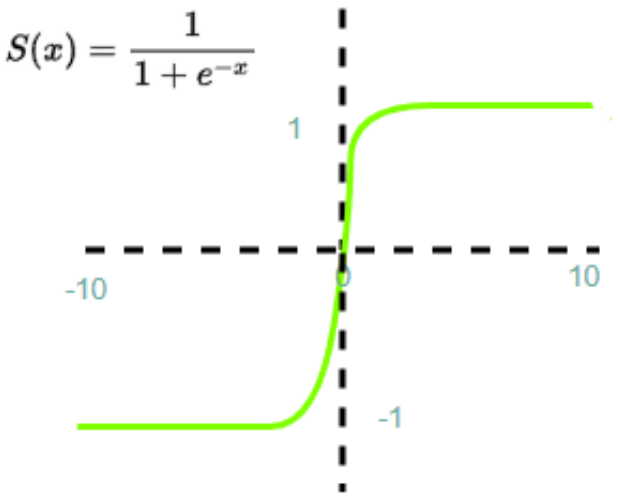
- Thích hợp cho lớp đầu ra trong các bài toán phân loại nhị phân.
- Khi cần **xác suất hóa** đầu ra trong khoảng $[0, 1]$.

**Ưu điểm:**

Phì hợp cho các bài toán nhị phân vì đầu ra dễ hiểu (xác xuất).

**Nhược điểm:**

- Gradient vanishingbiến mất khi $x$ quá lớn hoặc quá nhỏ.
- Không đối xứng qua gốc 0 (có khả năng gây mất cân bằng gradient).




Hình 9: Sigmoid activaion function

**Formula:** $S(x) = \frac{1}{1+e^{-x}}$

```
import torch
import torch.nn as nn

# Calling the sigmoid function
sig = nn.Sigmoid()

# Defining tensor
input = torch.Tensor([1,-2,3,-5])

# Applying sigmoid to the tensor
output = sig(input)
```

```
print(output) # Output: tensor([0.7311, 0.1192, 0.9526, 0.0067])
```

## 7.4 Tanh activation function:

**Phạm vi đầu ra:**

(-1, 1)

**Khi nào nên dùng:**

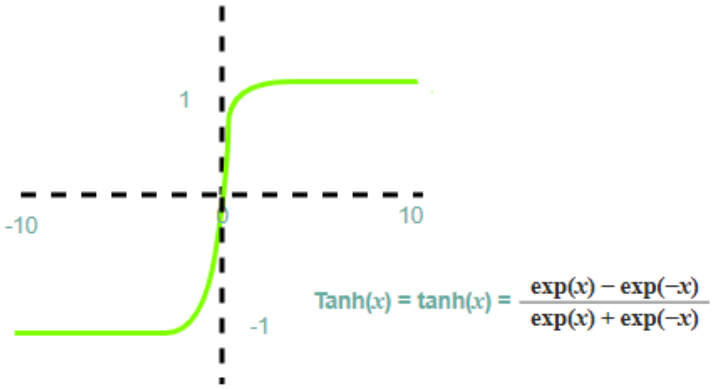
- Dùng trong hidden layer khi dữ liệu cần được chuẩn hóa xung quanh 0.
- Thường dung kkhi muốn trung tâm hóa dữ liệu, giúp gradient lan truyền tốt hơn.
- Hiệu quả trong các mạng RNNs/bài toán chuỗi thời gian.

**Ưu điểm:**

Đối xứng qua gốc 0, giúp cân bằng gradient.

**Nhược điểm:**

Gradient vanishingbiến mất giống sigmoid khi x rất lớn hoặc rất nhỏ.




Hình 10: Tanh activation function

**Formula:** $Tanh(x) = \frac{e^x - e^{-x}}{e^x - e^{-x}}$

```
import torch
import torch.nn as nn

# Calling the Tanh function
t = nn.Tanh()

# Defining tensor
input = torch.Tensor([1,-2,3,-5])

# Applying Tanh to the tensor
output = t(input)
print(output) # Output: tensor([ 0.7616, -0.9640,  0.9951, -0.9999])
```

### 7.5 Softmax activation function:

**Phạm vi đầu ra:**

(0, 1), tổng các đầu ra = 1.

**Những khi nên dùng:**

- Sử dụng ở lớp đầu ra của mạng neurons trong bài toán phân loại đa lớp.
- Khi cần xác xuất hóa đầu ra cho từng lớp.

**Ưu điểm:**

Chuyển đổi điểm số đầu ra thành xác suất, dễ hiểu, dễ sử dụng.

**Nhược điểm:**

Tính toán tốn kém hơn các hàm khác, không phù hợp cho hidden layer. **Formula:** $Softmax(x) = \frac{e^{x_i}}{\Sigma_j e^{x_j}}$

```python
import torch
import torch.nn as nn

# Calling the Softmax function with
# dimension = 0 as dimension starts
# from 0
sm = nn.Softmax(dim=0)

# Defining tensor
input = torch.Tensor([1,-2,3,-5])

# Applying function to the tensor
output = sm(input)
print(output) # Output: tensor([0.7311, 0.1192, 0.9526, 0.0067])
```

# 8 Ví dụ về một LeNet-5 phân loại thời trang.

- Dưới đây là code ví dụ về mô hình LeNet-5 sử dụng thư viện PyTorch nhằm nhận diện các loại trang phục trong tập dữ liệu **Fashion-MNIST**.
- Fashion-MNIST là một tập sữ liệu với các hình ảnh $28 \times 28$ đại diện cho các loại trang phục như áo thun, giày, túi xách, v.v.

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Define LeNet-5 model
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=2)  # Output:
            6x28x28
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)  # Output: 6x14x14
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)  # Output: 16x10x10
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)  # Output: 16x5x5
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # Fully Connected Layer 1
        self.fc2 = nn.Linear(120, 84)  # Fully Connected Layer 2
        self.fc3 = nn.Linear(84, 10)  # Output Layer (10 classes)

    def forward(self, x):
        x = torch.tanh(self.conv1(x))
        x = self.pool1(x)
```

```
        x = torch.tanh(self.conv2(x))
        x = self.pool2(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
        x = self.fc3(x)
        return x

# Download and prepare Fashion-MNIST data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # normalize data to [-1, 1]
])

train_dataset = datasets.FashionMNIST(root='./data', train=True, download=True,
    transform=transform)
test_dataset = datasets.FashionMNIST(root='./data', train=False, download=True,
    transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)

# Initialize the model, loss function and optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = LeNet5().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train model
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        # Forward
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch␣{epoch+1}/{num_epochs},␣Loss:␣{running_loss/len(train_loader):.4f}
        ")

# Evaluate model
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Accuracy␣on␣test␣set:␣{100␣*␣correct␣/␣total:.2f}%")
```