

# U-Net

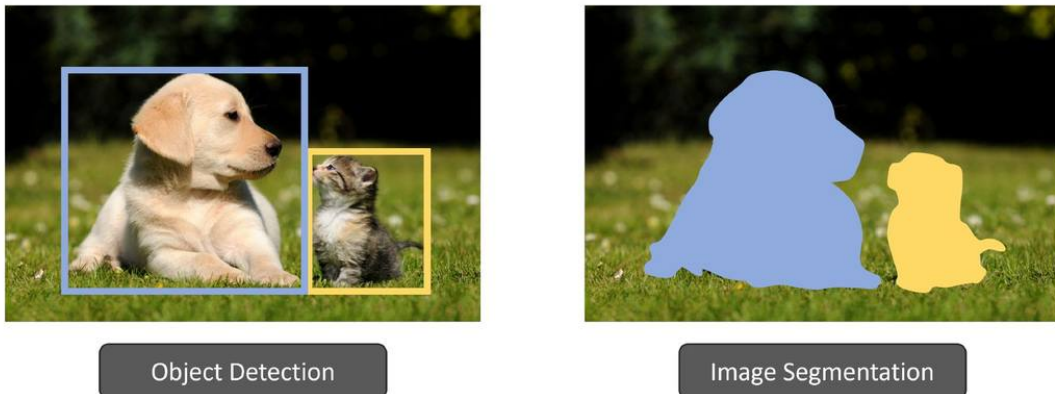
Tue-Thu Van-Dinh, Hoang-Duy Tran, Quang-Vinh Dinh

## I. Giới thiệu

### 1. Image Segmentation

**Image Segmentation (phân vùng ảnh)** là một nhiệm vụ quan trọng trong lĩnh vực thị giác máy tính, trong đó một bức ảnh được chia thành nhiều vùng khác nhau với mục tiêu đơn giản hóa hoặc thay đổi biểu diễn của bức ảnh thành một dạng có ý nghĩa hơn để dễ dàng phân tích. Image Segmentation thường được sử dụng để xác định vị trí các đối tượng hoặc đường biên, hay nói cách khác, đây là quá trình gán nhãn cho từng pixel trong ảnh, sao cho các pixel thuộc cùng một đối tượng hoặc khu vực sẽ được nhóm lại với nhau.

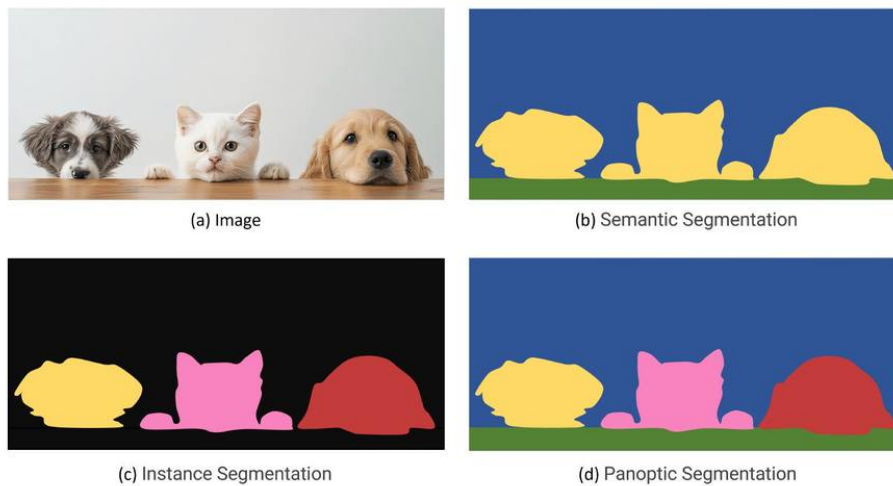
Image Segmentation và Object Detection (phát hiện đối tượng) đều có chung mục tiêu là xác định các vùng chứa đối tượng và gán nhãn cho chúng; tuy nhiên, Image Segmentation yêu cầu độ chính xác cao hơn. Khác với Object Detection, nơi đối tượng được định vị thông qua các hộp giới hạn (bounding boxes), Image Segmentation yêu cầu nhãn dự báo phải chính xác đến từng pixel trong ảnh. Điều này giúp chúng ta hiểu rõ hơn về nội dung bức ảnh, không chỉ dừng lại ở việc xác định vị trí các vật thể mà còn mô tả được hình dạng của chúng, cũng như gán từng pixel vào đúng vật thể tương ứng.



Hình 1: Minh họa phân biệt giữa Object Detection và Image Segmentation.

Image Segmentation được chia làm ba loại:

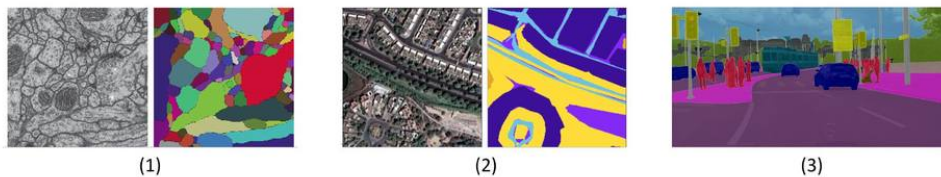
- **Semantic Segmentation:** Gán nhãn cho từng pixel trong ảnh và nhóm các pixel có cùng ngữ nghĩa vào một lớp cụ thể. Quá trình này phân đoạn các vùng ảnh theo từng lớp khác nhau mà không phân biệt giữa các đối tượng riêng lẻ thuộc cùng một lớp.
- **Instance Segmentation:** Phân đoạn các vùng ảnh chi tiết đến từng đối tượng trong mỗi lớp.
- **Panoptic segmentation:** Kết hợp giữa Semantic Segmentation và Instance Segmentation, trong đó mỗi đối tượng được phân đoạn với ranh giới rõ ràng, đồng thời danh tính và ý nghĩa của từng đối tượng trong ảnh được dự đoán chính xác.



Hình 2: Minh họa Semantic Segmentation, Instance Segmentation và Panoptic segmentation.

Image Segmentation thường được sử dụng trong nhiều ứng dụng thực tiễn như:

- Y học: Phân đoạn các tế bào, mô hoặc cơ quan trong ảnh chụp X-quang, CT, hoặc MRI.
- Xử lý ảnh vệ tinh: Phân loại vùng đất, rừng, sông ngòi từ các bức ảnh vệ tinh.
- Xe tự hành: Nhận diện đường, xe cộ, người đi bộ và các vật thể khác để hỗ trợ định vị và dẫn đường.

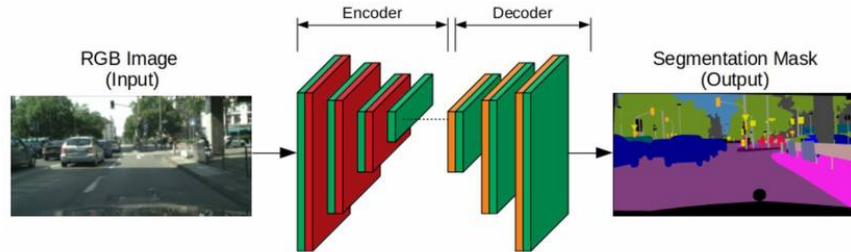


Hình 3: Minh họa ứng dụng của Image Segmentation trong (1) Y học, (2) Xử lý ảnh vệ tinh, (3) Xe tự hành.

Có hai phương pháp chính để thực hiện Image Segmentation:

- **Phương pháp truyền thống** - dựa trên các thuật toán như:
  - Phân ngưỡng (Thresholding) để chia điểm ảnh thành các lớp dựa trên cường độ.
  - Phân cụm (Clustering) như K-means để nhóm các điểm ảnh có đặc trưng tương đồng.
  - Phân đoạn theo vùng (Region-based Segmentation) dựa vào sự tương đồng về màu sắc hoặc kết cấu.
  - Phân đoạn theo cạnh (Edge-based Segmentation) để xác định ranh giới của các đối tượng trong hình ảnh.
- **Phương pháp hiện đại:**
  - Sử dụng mạng nơ-ron tích chập (CNN) để học và trích xuất các đặc trưng trực tiếp từ dữ liệu, được huấn luyện để tự động nhận diện những đặc trưng quan trọng trong ảnh, thay vì dựa vào các hàm tùy chỉnh như trong các phương pháp truyền thống.
  - Các mạng nơ-ron dùng để phân đoạn thường áp dụng cấu trúc encoder-decoder. Encoder sẽ trích xuất các đặc trưng của ảnh qua các bộ lọc ngày càng hẹp và sâu hơn. Nếu encoder được huấn luyện sẵn trên các nhiệm vụ như nhận diện khuôn mặt hay

nhận dạng ảnh, nó sẽ tận dụng kiến thức đã học để hỗ trợ phân đoạn (học chuyển giao). Sau đó, decoder sẽ mở rộng kết quả của encoder qua nhiều lớp để tạo ra mặt nạ phân đoạn có độ phân giải pixel giống hệt ảnh đầu vào.



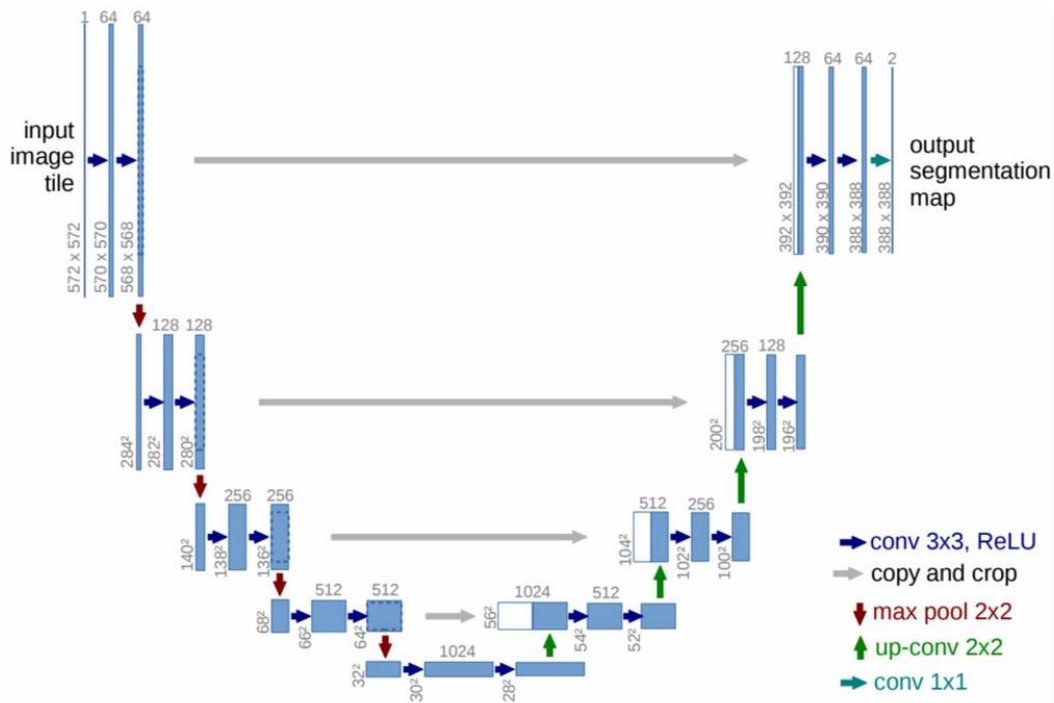
Hình 4: Cấu trúc cơ bản của mô hình mạng nơ-ron cho Image Segmentation. Nguồn: [Link](#)

Trong bối cảnh các bài toán phức tạp, các phương pháp học sâu đã trở thành lựa chọn hàng đầu nhờ khả năng xử lý chi tiết và duy trì thông tin không gian. Đặc biệt, kiến trúc mạng U-Net nổi bật với thiết kế đối xứng độc đáo và khả năng phân đoạn chính xác ở cấp độ pixel, giúp bảo tồn các chi tiết nhỏ trong ảnh. Do đó, U-Net đã trở thành giải pháp lý tưởng cho những bài toán đòi hỏi độ chi tiết cao, nơi việc duy trì thông tin không gian là vô cùng quan trọng.

Trong tài liệu này, chúng ta sẽ khám phá kiến trúc mạng U-Net và cách áp dụng nó để giải quyết bài toán về Image Segmentation.

## 2. U-Net

**U-Net** là một mạng nơ-ron tích chập được thiết kế ban đầu cho phân đoạn ảnh y sinh. Kiến trúc của U-Net bao gồm 2 phần chính là encoder và decoder đối xứng nhau với hình dạng giống chữ U.








Hình 5: Kiến trúc mạng U-Net. Nguồn: [Link](#)



Hình 5 trên mô tả kiến trúc mạng U-Net, trong đó:

- Mỗi hình chữ nhật màu xanh biểu thị một multi-channel feature map.
- Số lượng channels được ghi chú ở phía trên hình chữ nhật.
- Kích thước x-y được cung cấp ở góc dưới bên trái của hình chữ nhật.
- Các hình chữ nhật màu trắng đại diện cho các feature maps được sao chép.
- Các mũi tên biểu thị các operations khác nhau:

Mũi tên	Ý nghĩa
	Conv layer 3x3, ReLU
	Skip Connections (có crop)
	Max Pooling 2x2
	Transposed Convolution
	Conv layer 1x1

Bảng 1: Ý nghĩa các mũi tên trong kiến trúc mạng U-Net.

## 2.1. Skip Connections

U-Net lần đầu tiên giới thiệu skip connections trong Deep Learning như một giải pháp hiệu quả để khắc phục mất mát thông tin trong quá trình downsampling, thường gặp trong các khối mã hóa (encoder) của kiến trúc encoder-decoder. Skip connections là các kết nối trực tiếp từ encoder đến decoder mà không cần đi qua bottleneck, giúp giảm thiểu mất mát dữ liệu do các phương pháp aggressive pooling và downsampling gây ra.

Việc sử dụng skip connections giúp khắc phục vấn đề mất thông tin ở spatial dimension. Nếu chỉ dựa vào feature map từ bottleneck để xây dựng lại feature map có độ phân giải cao, nhiều chi tiết về vị trí pixel sẽ bị mất. Nhờ các kết nối này, thông tin từ phần encoder được truyền trực tiếp sang decoder, hỗ trợ việc tái tạo vị trí pixel chính xác hơn.

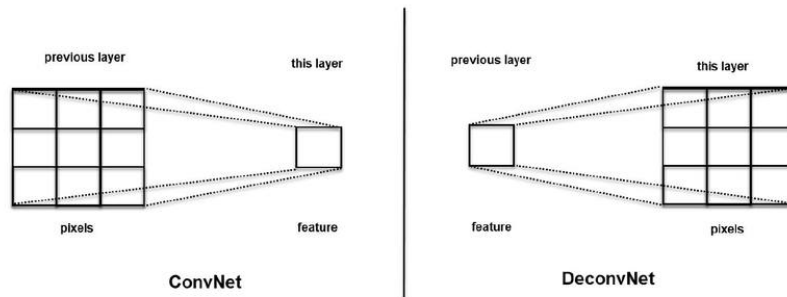
## 2.2. Transposed Convolution

Convolution là một phép toán cơ bản trong xử lý dữ liệu, đặc biệt là trong mạng nơ-ron tích chập (CNN). Phép toán này sử dụng một kernel (bộ lọc) trượt qua dữ liệu đầu vào, tính tích chập để tạo ra bản đồ đặc trưng (feature map). Kết quả thường có kích thước nhỏ hơn, giúp trích xuất các đặc trưng cục bộ quan trọng như cạnh, góc hoặc mẫu hình.

Trong khi Convolution giảm kích thước không gian của dữ liệu, Transposed Convolution (hay còn gọi là Deconvolution) thực hiện ngược lại: nó mở rộng dữ liệu đầu vào. Đây là một kỹ thuật phổ biến trong các bài toán như Image Generation hoặc Image Segmentation, nơi cần khôi phục dữ liệu về kích thước ban đầu hoặc tạo dữ liệu có độ phân giải cao hơn.

Trong Convolution tiêu chuẩn, kích thước không gian của feature map được giảm bằng cách áp dụng kernel có kích thước cố định, kết hợp với stride và padding. Ngược lại, Transposed Convolution thực hiện chức năng mở rộng kích thước không gian của feature map bằng cách áp dụng kernel lên đầu vào, đồng thời điều chỉnh kích thước output thông qua stride và padding. Có thể hiểu, Transposed Convolution là quá trình ngược của Convolution thông thường, trong đó mỗi feature được ánh xạ thành các pixel của ảnh, thay vì ánh xạ các pixel thành feature như trong Convolution tiêu chuẩn (hình 6 dưới đây minh họa sự khác biệt giữa hai quá trình này).

So với các phương pháp upsampling truyền thống như Nearest Neighbors, Bi-Linear Interpolation hay Max-Unpooling, Transposed Convolution vượt trội hơn nhờ khả năng học các tham số từ dữ liệu, giúp khôi phục thông tin không gian một cách chi tiết và chính xác hơn.

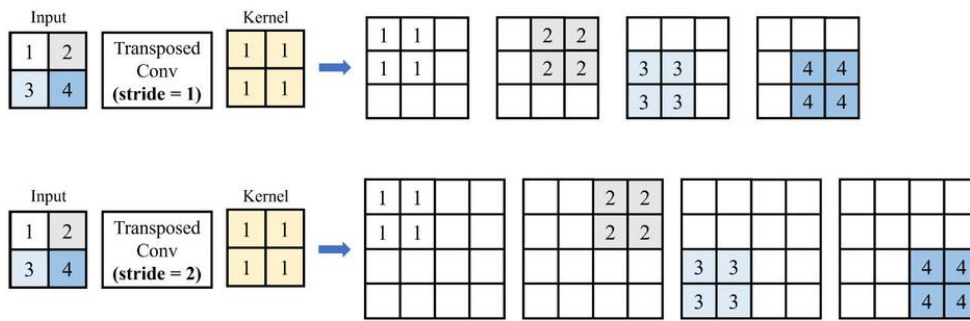


Hình 6: So sánh Convolution với Transposed Convolution. Nguồn: [Link](#)

### Quy trình thực hiện Transposed Convolution:

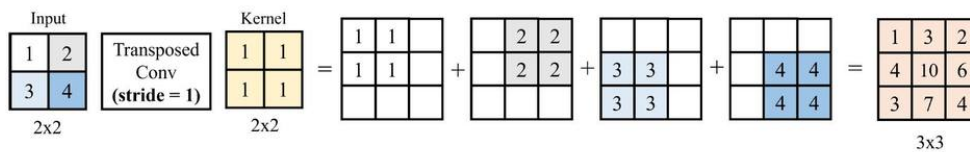
#### 1. Nhân tích chập và xử lý dịch chuyển stride:

- Mỗi phần tử trong input feature map được nhân với kernel, sau đó kết quả được đặt vào vị trí tương ứng trên output feature map.
- Với  $\text{stride} = s$ , kết quả sẽ dịch chuyển  $s$  pixel sau mỗi lần tính toán, và stride lớn sẽ làm tăng kích thước của output.



Hình 7: Minh họa bước nhân tích chập và xử lý dịch chuyển stride với hai giá trị stride bằng 1 và 2.

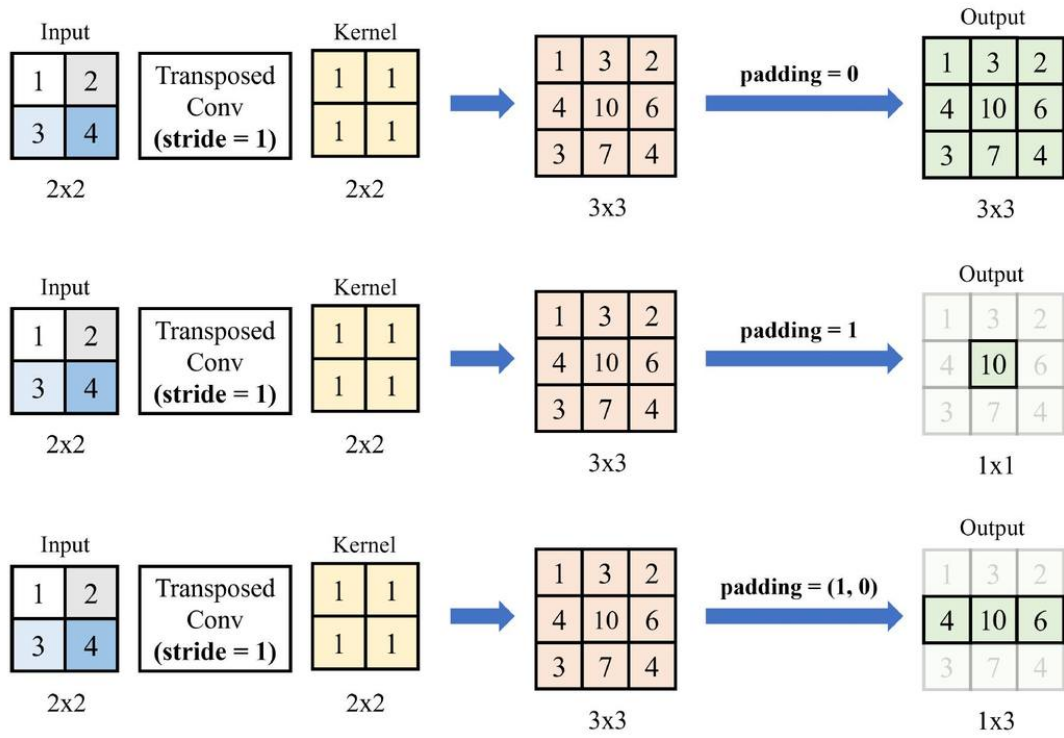
- Xử lý chồng chéo (overlap): Khi kích thước stride nhỏ hơn kích thước kernel, các vùng trên ma trận kết quả sẽ bị chồng lên nhau. Trong trường hợp này, các giá trị tại các vị trí chồng chéo sẽ được cộng lại để tạo thành giá trị cuối cùng tại vị trí đó.



Hình 8: Minh họa bước xử lý chồng chéo (overlap).

#### 3. Áp dụng padding:

- Padding được thêm vào để đảm bảo kích thước output phù hợp.
- Với  $\text{padding} = p$ , tiến hành loại bỏ  $p$  hàng và cột ở 4 cạnh (trên, dưới, trái, phải) của kết quả đầu ra. Lưu ý:
  - Khi  $\text{padding} = (p, q)$ : bỏ đi  $p$  hàng đầu tiên và cuối cùng (trên, dưới);  $q$  cột đầu tiên và cuối cùng (trái, phải) của output.
  - Khi  $\text{padding} = p$ : đây là cách viết rút gọn, tương đương với  $\text{padding} = (p, p)$ .



Hình 9: Minh họa bước áp dụng padding.

- Công thức tổng quát để xác định kích thước output:

$$Height : H_{out} = (m - 1) \cdot s - 2p + (h - 1) + 1$$

$$Width : W_{out} = (n - 1) \cdot t - 2q + (k - 1) + 1$$

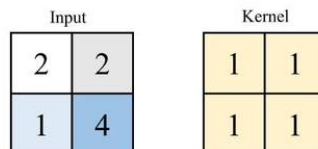
Trong đó:

- m x n: kích thước input.
- h x k: kích thước kernel.
- s x t: kích thước stride.
- p x q: kích thước padding.

#### Ví dụ minh họa:

Để hiểu rõ hơn các bước tính toán trong quá trình Transposed Convolution, chúng ta sẽ cùng thực hiện hai ví dụ với input và kernel được mô tả trong Hình 10.

- Input: Một encoded feature map kích thước 2x2.
- Kernel có kích thước 2x2.

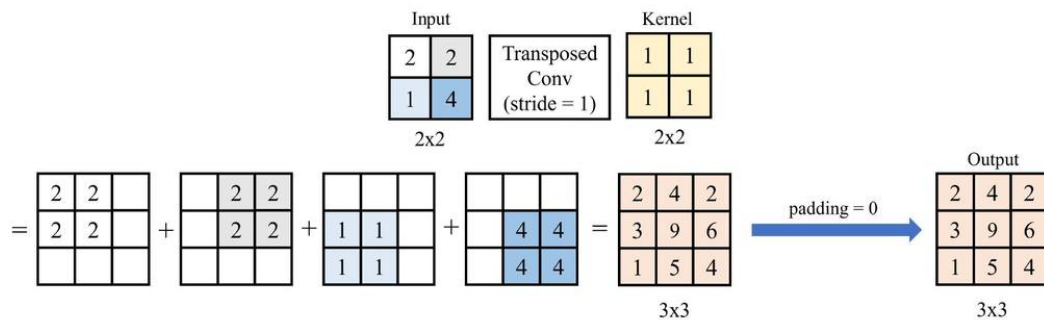


Hình 10: Giá trị Input và Kernel cho hai ví dụ tính toán quá trình Transposed Convolution.



**Ví dụ 1:** Với  $\text{stride} = 1$  và  $\text{padding} = 0$ .

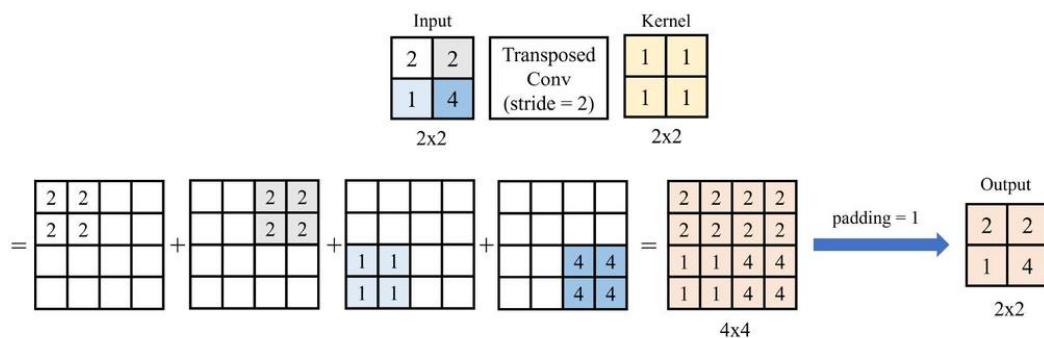
1. Đầu tiên, lấy từng pixel input nhân với kernel (nhân với từng phần tử). Với  $\text{stride} = 1$ , di chuyển kết quả của mỗi lần nhân pixel với kernel sang 1 pixel.
2. Sau khi thực hiện nhân tích chập, vì  $\text{stride}$  có kích thước nhỏ hơn kernel nên cần cộng các giá trị chồng chéo (overlap) lại với nhau. Trong trường hợp này, vì  $\text{padding} = 0$  nên không loại bỏ hàng hay cột nào ở output. Hình 11 dưới đây minh họa cho quá trình tính Transposed Convolution cho ví dụ đầu tiên:



Hình 11: Minh họa các bước tính Transposed Convolution cho ví dụ 1.

**Ví dụ 2:** Với  $\text{stride} = 2$  và  $\text{padding} = 1$ .

1. Đầu tiên, lấy từng pixel input nhân với kernel (nhân với từng phần tử). Với  $\text{stride} = 2$ , di chuyển kết quả của mỗi lần nhân pixel với kernel sang 2 pixel.
2. Sau khi thực hiện nhân chập, các giá trị không bị overlap vì kích thước của  $\text{stride}$  không nhỏ hơn kích thước kernel.
3. Với  $\text{padding} = 1$ , tiến hành loại bỏ 1 hàng và cột ở 4 cạnh (trên, dưới, trái, phải) của kết quả đầu ra. Hình 12 dưới đây minh họa trình tính Transposed Convolution cho ví dụ 2:

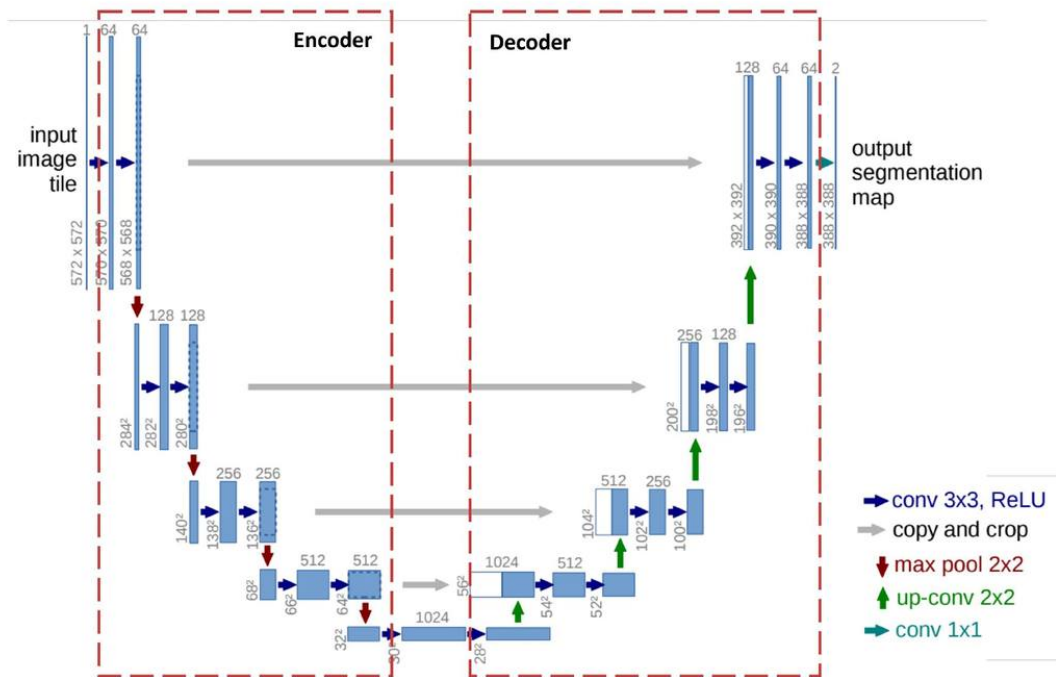


Hình 12: Minh họa các bước tính Transposed Convolution cho ví dụ 2.

### 2.3. Encoder

Dựa vào hình 13, phần encoder bao gồm các lớp Conv và MaxPooling thông thường. Khi đi từ trên xuống dưới, kích thước width x height giảm, trong khi depth tăng lên (depth của output mỗi lớp được ghi ở trên đỉnh của hình chữ nhật, còn kích thước width và height được ghi dọc theo hình chữ nhật).

Encoder được thiết kế với mục tiêu trích xuất các đặc trưng quan trọng từ ảnh đầu vào thông qua các lớp convolution và pooling. Quá trình này không chỉ giúp nhận diện các đặc trưng quan trọng mà còn giảm dần kích thước không gian của ảnh, đồng thời giữ lại những thông tin đặc trưng cần thiết cho các bước xử lý tiếp theo trong mô hình.



Hình 13: Kiến trúc U-net với 2 phần chính Encoder-Decoder.

#### 2.4. Decoder

Phần này được thiết kế ngược lại với encoder vì nó làm tăng kích thước width x height, đồng thời giảm depth. Để thực hiện điều này, ta sử dụng Transposed Convolution. Tại mỗi giai đoạn của decoder, layer đối xứng tương ứng của encoder sẽ được crop và concatenate lại. Từ hình 13, ta có thể thấy rõ cách thức hoạt động của decoder, với các bước thực hiện tăng dần kích thước không gian và tái tạo lại thông tin đã bị nén trong encoder.

Ngoài ra, kiến trúc U-Net không sử dụng lớp fully connected (FC) ở phần cuối, điều này tạo ra sự khác biệt so với các mô hình Deep Learning truyền thống. Trong các mô hình end-to-end thông thường, lớp FC sẽ kết nối các đặc trưng đã được trích xuất qua các lớp convolution để đưa ra kết quả dự đoán cuối cùng. Tuy nhiên, U-Net, việc kết nối các đặc trưng và tạo ra kết quả dự đoán được thực hiện thông qua phần up-sampling ở nửa thứ hai của "chữ U" (decoder).

#### 2.5. Loss Function

Đây là bài toán phân lớp cho các pixels nên loss function sẽ bằng tổng cross entropy của các pixels trong ảnh.



## II. Thực Hành

Trong phần thực hành, chúng ta sẽ sử dụng kiến trúc U-Net để giải quyết bài toán Image Segmentation trên các bộ dataset cụ thể. Quá trình này bao gồm việc huấn luyện mô hình và đánh giá hiệu suất trên dữ liệu thực tế.

### Bài toán: Segmentation cho bộ dữ liệu chó mèo

Đối với bài toán này, chúng ta sẽ xây dựng một mô hình Segmentation nhằm phân đoạn hình ảnh thú cưng (chó/mèo) từ tập dữ liệu **Oxford-IIIT Pet Dataset**. Mục tiêu của bài toán là tạo ra một mặt nạ (mask) phân đoạn, trong đó mỗi pixel được gán vào một trong ba lớp: **Background** (nền), **Pet** (vùng chứa thú cưng), **Contour** (đường viền phân tách giữa thú cưng và nền).

Bài toán segmentation này không chỉ giúp xác định rõ ràng vùng thú cưng trong ảnh mà còn là nền tảng để giải quyết các bài toán phức tạp hơn, như tách nền tự động, phân tích hình dáng, hoặc nhận diện giống loài.

#### 1. Import các thư viện cần thiết

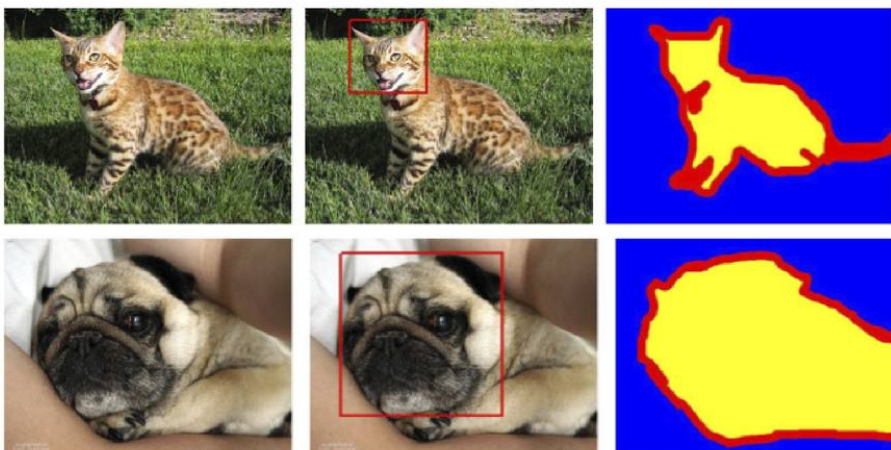
Đầu tiên, chúng ta sẽ import các thư viện cần thiết để xử lý dữ liệu và xây dựng mô hình.

```
1 import copy
2 import numpy as np
3 import torch
4 import torch.nn as nn
5 import torch.nn.functional as F
6 import matplotlib.pyplot as plt
```

#### 2. Xử lý dữ liệu

##### 2.1. Import và tiền xử lý dữ liệu

**Oxford-IIIT Pet Dataset** là một tập dữ liệu phổ biến, được thiết kế dành cho các bài toán **Image Classification** và **Image Segmentation**. Bộ dữ liệu này do nhóm nghiên cứu tại Đại học Oxford phát triển, cung cấp cả nhãn phân loại chó mèo và mặt nạ phân đoạn chi tiết.



Hình 14: Một số mẫu dữ liệu trong Oxford-IIIT Pet Dataset.

Trong phần này, chúng ta sẽ xử lý dữ liệu từ **Oxford-IIIT Pet Dataset** bằng cách áp dụng các phép biến đổi (transforms) cho ảnh và mặt nạ phân đoạn. Ảnh sẽ được chuẩn hóa, resize về kích thước đồng nhất và chuyển đổi thành tensor, trong khi mặt nạ phân đoạn được xử lý để đảm bảo các giá trị pixel phù hợp với định dạng yêu cầu của mô hình.

Dữ liệu đã được chia sẵn thành hai phần: tập huấn luyện (**trainval**) và tập kiểm tra (**test**) để sử dụng cho quá trình huấn luyện và đánh giá mô hình một cách hiệu quả.

```

1 from torchvision import transforms
2 from torchvision.datasets import OxfordIIITPet
3 from torch.utils.data import DataLoader
4
5 img_size = (128, 128)
6 num_classes = 3
7
8 transform = transforms.Compose([
9     transforms.Resize(img_size),
10    transforms.ToTensor(),
11    transforms.Normalize((0.485, 0.456, 0.406),
12                        (0.229, 0.224, 0.225))
13 ])
14
15 def target_transform(target):
16     img = transforms.Resize(img_size)(target)
17     img = transforms.functional.pil_to_tensor(img).squeeze_()
18     img = img - 1
19
20     img = img.to(torch.long)
21     return img
22
23
24 train_set = OxfordIIITPet(root="pets_data", split="trainval", target_types="segmentation",
25                          transform=transform,
26                          target_transform=target_transform,
27                          download=True)
28
29 test_set = OxfordIIITPet(root="pets_data", split="test", target_types="segmentation",
30                         transform=transform,
31                         target_transform=target_transform,
32                         download=True)
33
34 batch_size = 64
35 train_loader = DataLoader(train_set,
36                          batch_size=batch_size,
37                          shuffle=True,
38                          num_workers=6)
39 test_loader = DataLoader(test_set,
40                         batch_size=batch_size,
41                         num_workers=6)

```

## 2.2. Hậu xử lý dữ liệu

Quá trình de-normalize được thực hiện để đưa dữ liệu về phạm vi giá trị ban đầu, giúp hiển thị ảnh rõ ràng hơn để có thể kiểm tra và đánh giá mô hình.

```

1 def de_normalize(img,
2                 mean=(0.485, 0.456, 0.406),
3                 std=(0.229, 0.224, 0.225)):
4     result = img * std + mean
5     result = np.clip(result, 0.0, 1.0)
6
7     return result

```

### 3. Xây dựng Model

Ở phần này, chúng ta sẽ bắt đầu xây dựng kiến trúc U-Net để giải quyết bài toán phân đoạn ảnh.

Đầu tiên, ta thiết kế khối Convolutional (ConvBlock), đây là thành phần cốt lõi của U-Net. Khối này được sử dụng để trích xuất đặc trưng từ ảnh đầu vào thông qua các lớp tích chập. Sau đó, ConvBlock sẽ được sử dụng để xây dựng các thành phần chính trong U-Net như Encoder và Decoder.

#### 3.1. Xây dựng khối Convolutional

```
1 class ConvBlock(nn.Module):
2     def __init__(self, in_channels, out_channels) -> None:
3         super().__init__()
4         self.conv_block = nn.Sequential(
5             nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1, bias=False),
6             nn.BatchNorm2d(out_channels),
7             nn.ReLU(inplace=True),
8             nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False),
9             nn.BatchNorm2d(out_channels),
10            nn.ReLU(inplace=True),
11        )
12
13    def forward(self, x):
14        x = self.conv_block(x)
15        return x
```

#### 3.2. Xây dựng khối Encoder

```
1 class Encoder(nn.Module):
2     def __init__(self, in_channels, out_channels) -> None:
3         super().__init__()
4         self.encoder = nn.Sequential(
5             nn.MaxPool2d(2),
6             ConvBlock(in_channels, out_channels)
7         )
8
9     def forward(self, x):
10        x = self.encoder(x)
11        return x
```

#### 3.3. Xây dựng khối Decoder

```
1 class Decoder(nn.Module):
2     def __init__(self, in_channels, out_channels) -> None:
3         super().__init__()
4         self.conv_trans = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=4, stride
5         =2, padding=1)
6         self.conv_block = ConvBlock(in_channels, out_channels)
7
8     def forward(self, x1, x2):
9         x1 = self.conv_trans(x1)
10        x = torch.cat([x2, x1], dim=1)
11        x = self.conv_block(x)
12        return x
```



### 3.4. Xây dựng Model

```

1 class UNet(nn.Module):
2     def __init__(self, n_channels, n_classes) -> None:
3         super().__init__()
4         self.n_channels = n_channels
5         self.n_classes = n_classes
6
7         self.in_conv = ConvBlock(n_channels, 64)
8
9         self.enc_1 = Encoder(64, 128)
10        self.enc_2 = Encoder(128, 256)
11        self.enc_3 = Encoder(256, 512)
12        self.enc_4 = Encoder(512, 1024)
13
14        self.dec_1 = Decoder(1024, 512)
15        self.dec_2 = Decoder(512, 256)
16        self.dec_3 = Decoder(256, 128)
17        self.dec_4 = Decoder(128, 64)
18
19        self.out_conv = nn.Conv2d(64, n_classes, kernel_size=1)
20
21    def forward(self, x):
22        x1 = self.in_conv(x)
23
24        x2 = self.enc_1(x1)
25        x3 = self.enc_2(x2)
26        x4 = self.enc_3(x3)
27        x5 = self.enc_4(x4)
28
29        x = self.dec_1(x5, x4)
30        x = self.dec_2(x, x3)
31        x = self.dec_3(x, x2)
32        x = self.dec_4(x, x1)
33
34        x = self.out_conv(x)
35
36        return x

```

## 4. Train và evaluate Model

### 4.1. Xây dựng hàm để hiển thị kết quả dự đoán

```

1 @torch.inference_mode()
2 def display_prediction(model, image, target):
3     model.eval()
4     img = image[None,...].to(device)
5     output = model(img)
6     pred = torch.argmax(output, axis=1)
7
8     plt.figure(figsize=(10, 5))
9
10    plt.subplot(1,3,1)
11    plt.axis('off')
12    plt.title("Input Image")
13    plt.imshow(de_normalize(image.numpy().transpose(1,2,0)))
14
15    plt.subplot(1,3,2)
16    plt.axis('off')
17    plt.title("Prediction")
18    plt.imshow(pred.cpu().squeeze())
19
20    plt.subplot(1,3,3)
21    plt.axis('off')
22    plt.title("Ground Truth")
23    plt.imshow(target)
24
25    plt.show()

```

## 4.2. Xây dựng hàm evaluate

```

1 def evaluate(model, test_loader, criterion):
2     model.eval()
3     test_loss = 0.0
4     with torch.no_grad():
5         for inputs, labels in test_loader:
6             inputs, labels = inputs.to(device), labels.to(device)
7
8             outputs = model(inputs)
9             loss = criterion(outputs, labels)
10
11             test_loss += loss.item()
12
13     test_loss = test_loss / len(test_loader)
14     return test_loss

```

## 4.3. Khai báo các tham số cho Model

```

1 device = "cuda" if torch.cuda.is_available() else "cpu"
2 max_epoch = 30
3 LR = 0.001
4 criterion = nn.CrossEntropyLoss()
5 optimizer = torch.optim.Adam(model.parameters(), lr=LR)

```

## 4.4. Model Training và Evaluation

Trong phần này, chúng ta sẽ thực hiện huấn luyện mô hình U-Net để giải quyết bài toán phân đoạn ảnh. Sau mỗi epoch, kết quả đánh giá sẽ được in ra màn hình cùng với hình ảnh cùng với hình ảnh dự đoán và mặt nạ thực tế để dễ dàng theo dõi và trực quan hóa hiệu suất của mô hình.

```

1 test_index = 80
2 display_image = test_set[test_index][0]
3 display_target = test_set[test_index][1]
4
5 train_losses = []
6 test_losses = []
7 best_loss = float('inf')
8 best_model_wts = copy.deepcopy(model.state_dict())
9
10 model.to(device)
11 for epoch in range(max_epoch):
12     model.train()
13     running_loss = 0.0
14     for inputs, targets in train_loader:
15         inputs, targets = inputs.to(device), targets.to(device)
16
17         optimizer.zero_grad()
18         outputs = model(inputs)
19         loss = criterion(outputs, targets)
20         running_loss += loss.item()
21         loss.backward()
22         optimizer.step()
23     epoch_loss = running_loss / len(train_loader)
24     test_loss = evaluate(model, test_loader, criterion)
25     if test_loss < best_loss:
26         best_loss = test_loss
27         best_model_wts = copy.deepcopy(model.state_dict())
28     print(f"Epoch [{epoch + 1}/{max_epoch}], Trainning loss: {epoch_loss:.4f}, Test Loss: {test_loss:.4f}")
29     print(f"Test image_{test_index} after epoch {epoch+1}: ")
30     display_prediction(model, display_image, display_target)
31
32 train_losses.append(epoch_loss)
33 test_losses.append(test_loss)

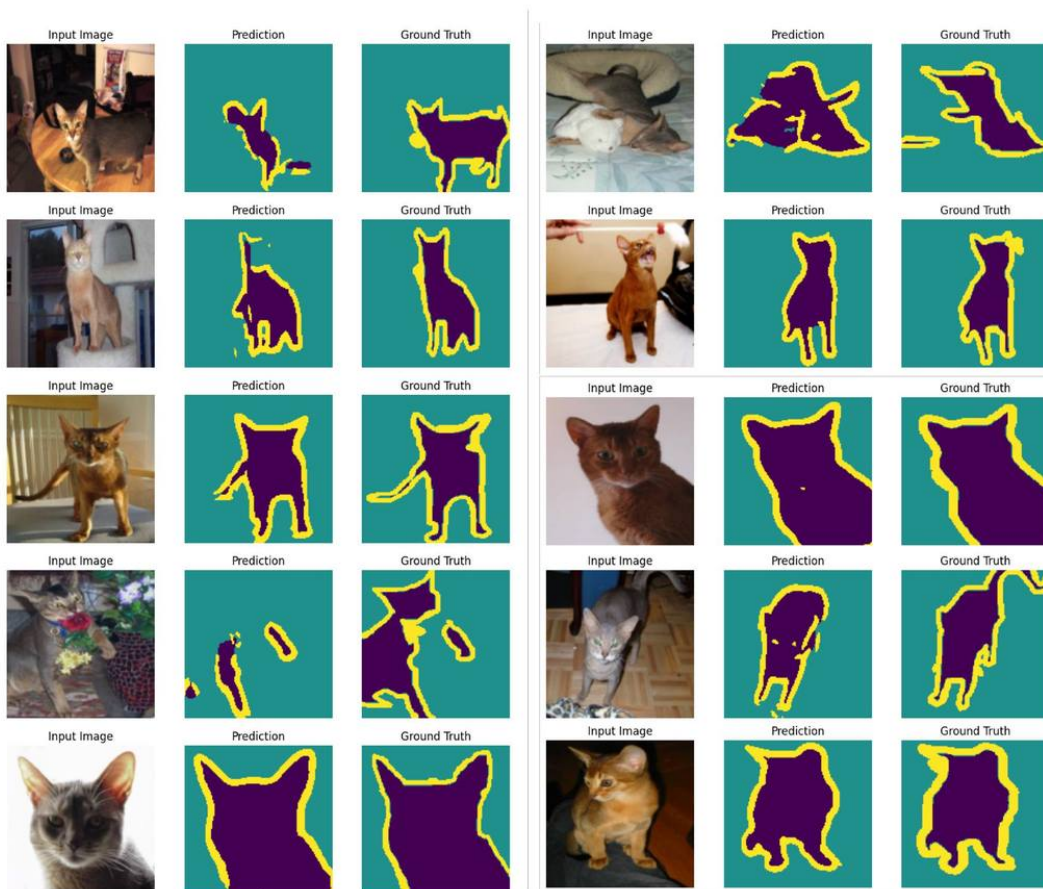
```

## 5. Load trọng số và dự đoán

Phần này là bước cuối cùng trong quy trình, mô hình thực hiện dự đoán 10 ảnh đầu tiên trên tập kiểm tra, giúp đánh giá hiệu suất mô hình và xác nhận rằng mô hình hoạt động tốt nhất với trọng số đã lưu.

```
1 model.load_state_dict(best_model_wts)
2
3 n_test_points = 10
4
5 for i in range(n_test_points):
6     img, gt = test_set[i]
7
8     display_prediction(model, img, gt)
```

Sau khi huấn luyện xong, kết quả đạt được của mô hình như sau:



Hình 15: Kết quả dự đoán của mô hình sau khi huấn luyện.



## Bài toán 2: Segmentation cho Kvasir-SEG dataset

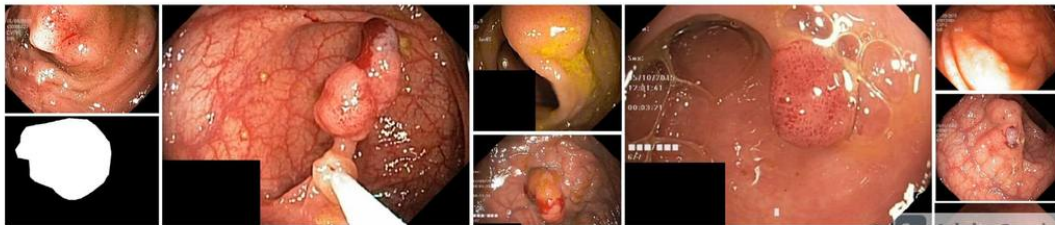
Trong bài toán này, chúng ta sẽ xây dựng một mô hình Segmentation nhằm phân đoạn hình ảnh nội soi từ tập dữ liệu **Kvasir-SEG**. Mục tiêu chính của bài toán là tạo ra một mặt nạ (mask) phân đoạn, trong đó mỗi pixel được gán vào một trong hai lớp: **Background** (nền) hoặc **Polyp** (vùng tổn thương có hình dáng giống một khối u).

### 1. Import các thư viện cần thiết

```
1 import os
2 import cv2
3 import copy
4 import torch
5 import random
6 import numpy as np
7 from PIL import Image
8 from glob import glob
9 import torch.nn as nn
10 import torch.nn.functional
11 from torchvision import transforms
12 from matplotlib import pyplot as plt
13 from torch.utils.data import Dataset, DataLoader
14 from sklearn.model_selection import train_test_split
```

### 2. Xử lý dữ liệu

**Kvasir-SEG** là một bộ dữ liệu chuẩn trong lĩnh vực phân đoạn y tế, đặc biệt được sử dụng cho các nhiệm vụ phân đoạn đa dạng như phát hiện và phân đoạn tổn thương trong nội soi tiêu hóa. Bộ dữ liệu này do Simula Research Laboratory phát triển và công bố, nhằm hỗ trợ nghiên cứu về phân tích hình ảnh nội soi.



Hình 16: Một số mẫu dữ liệu trong Kvasir-SEG dataset.

#### 2.1. Tải dữ liệu

```
1 !wget https://zenodo.org/record/1003777/files/Kvasir-SEG.zip
2 !unzip Kvasir-SEG.zip
```

#### Tạo đường dẫn

```
1 ROOT_PATH = '/content/Kvasir-SEG/Kvasir-SEG'
2 images_path = os.path.join(ROOT_PATH, 'images')
3 masks_path = os.path.join(ROOT_PATH, 'masks')
```

## 2.2. Xây dựng các hàm để xử lý dữ liệu

Ở phần này chúng ta sẽ xử lý dữ liệu nhằm chuẩn bị đầu vào phù hợp, trong đó có việc chuyển đổi ảnh về ảnh xám (gray scale), với ảnh nội soi trong bộ dữ liệu Kvasir-SEG thường không yêu cầu thông tin màu sắc đầy đủ (RGB) để phân đoạn vùng tổn thương. Do đó, việc chuyển đổi sang ảnh xám là một cách tối ưu hóa tài nguyên, đồng thời giữ lại thông tin cần thiết phục vụ cho mô hình.

```

1  img_size = (128, 128)
2
3  image_transform = transforms.Compose([
4      transforms.Resize(img_size),
5      transforms.Grayscale(num_output_channels=1),
6      transforms.ToTensor(),
7      transforms.Normalize(mean=[0.485], std=[0.229])
8  ])
9
10 def mask_transform(mask):
11     mask = transforms.Resize(img_size, interpolation=Image.NEAREST)(mask)
12     mask = transforms.functional.pil_to_tensor(mask).squeeze(0)
13     mask = mask / 255.0
14     mask = torch.round(mask).long()
15     return mask

```

## 2.3. Xây dựng các hàm tăng cường dữ liệu

Mặc dù bộ dữ liệu Kvasir-SEG cung cấp hình ảnh và nhãn chất lượng cao, số lượng mẫu trong tập dữ liệu này vẫn còn hạn chế (chỉ khoảng 1.000 ảnh). Điều này dễ dẫn đến overfitting, đặc biệt khi sử dụng các mô hình học sâu với số lượng tham số lớn.

Vì vậy, ta sẽ áp dụng các phương pháp data augmentation để củng cố lượng dữ liệu cho mô hình. Cụ thể:

1. **Elastic Transform (Biến dạng đàn hồi):** Áp dụng biến dạng không gian mềm mại để mô phỏng sự thay đổi tự nhiên trong hình ảnh nội soi, giúp mô hình nhận diện đặc trưng tốt hơn khi hình ảnh bị biến dạng.
2. **Horizontal Flip (Lật ngang):** Lật ảnh và mask theo chiều ngang, tăng khả năng nhận diện đặc trưng đối xứng, hữu ích với các tổn thương có tính đối xứng trái-phải.
3. **Vertical Flip (Lật dọc):** Lật ảnh và mask theo chiều dọc, bổ sung sự đa dạng, giúp mô hình thích nghi với góc nhìn khác nhau.
4. **Combined Flip (Kết hợp lật và biến dạng đàn hồi):** Kết hợp Elastic Transform, Horizontal Flip, và Vertical Flip để tạo các biến thể dữ liệu phong phú nhất, tăng khả năng học đặc trưng và cải thiện tổng quát hóa của mô hình.

```

1  def elastic_transform(image, mask, alpha_affine=10):
2      random_state = np.random.RandomState(None)
3      shape = image.size[::-1] # (H, W)
4
5      center_square = np.float32(shape) // 2
6      square_size = min(shape) // 3
7      pts1 = np.float32([center_square + square_size,
8                          [center_square[0] + square_size, center_square[1] - square_size],
9                          center_square - square_size])
10     pts2 = pts1 + random_state.uniform(-alpha_affine, alpha_affine, size=pts1.shape).astype(np.float32)
11
12     M = cv2.getAffineTransform(pts1, pts2)
13     image = cv2.warpAffine(np.array(image), M, shape[::-1], borderMode=cv2.BORDER_REFLECT_101)
14     mask = cv2.warpAffine(np.array(mask), M, shape[::-1], borderMode=cv2.BORDER_REFLECT_101)
15
16     return Image.fromarray(image), Image.fromarray(mask)
17

```

```

18 def hflip_transform(image, mask):
19     return image.transpose(Image.FLIP_LEFT_RIGHT), mask.transpose(Image.FLIP_LEFT_RIGHT)
20
21 def vflip_transform(image, mask):
22     return image.transpose(Image.FLIP_TOP_BOTTOM), mask.transpose(Image.FLIP_TOP_BOTTOM)
23
24 def flip_transform(image, mask):
25     return image.transpose(Image.FLIP_LEFT_RIGHT).transpose(Image.FLIP_TOP_BOTTOM), mask.
        transpose(Image.FLIP_LEFT_RIGHT).transpose(Image.FLIP_TOP_BOTTOM)

```

## 2.4. Xây dựng PyTorch Dataset

```

1 class KvasirDatasetAugmented(Dataset):
2     def __init__(self, images_path, masks_path, transform=None, target_transform=None,
        augmentations=None):
3         self.images_path = sorted(os.listdir(images_path))
4         self.masks_path = sorted(os.listdir(masks_path))
5         self.images_dir = images_path
6         self.masks_dir = masks_path
7         self.transform = transform
8         self.target_transform = target_transform
9         self.augmentations = augmentations if augmentations else []
10
11         self.data = self._generate_augmented_data()
12
13     def _generate_augmented_data(self):
14         augmented_data = []
15         for img_file, mask_file in zip(self.images_path, self.masks_path):
16             img_path = os.path.join(self.images_dir, img_file)
17             mask_path = os.path.join(self.masks_dir, mask_file)
18
19             image = Image.open(img_path).convert("RGB")
20             mask = Image.open(mask_path).convert("L")
21
22             augmented_data.append((image, mask))
23
24             for aug_func in self.augmentations:
25                 aug_image, aug_mask = aug_func(image, mask)
26                 augmented_data.append((aug_image, aug_mask))
27
28         return augmented_data
29
30     def __len__(self):
31         return len(self.data)
32
33     def __getitem__(self, idx):
34         image, mask = self.data[idx]
35
36         if self.transform:
37             image = self.transform(image)
38         if self.target_transform:
39             mask = self.target_transform(mask)
40
41         return image, mask

```

## 2.5. Khởi tạo dataset với augmentation

```

1 augmentations = [elastic_transform, hflip_transform, vflip_transform, flip_transform]
2 dataset = KvasirDatasetAugmented(images_path, masks_path, transform=image_transform,
        target_transform=mask_transform, augmentations=augmentations)

```



## 2.6. Tiến hành phân chia dữ liệu

```

1 train_size = int(0.7 * len(dataset))
2 val_size = int(0.15 * len(dataset))
3 test_size = len(dataset) - train_size - val_size
4 train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size,
    val_size, test_size])

```

## 2.7. Khai báo PyTorch Dataloader

```

1 batch_size = 64
2
3 train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=2)
4 val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=2)
5 test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, num_workers=2)

```

# 3. Xây dựng Model

Ở phần này chúng ta sẽ tiến hành xây dựng mô hình U-Net để Segmentation cho bộ dữ liệu Kvasir-SEG, nhưng vì cấu trúc mô hình không thay đổi nên chúng ta sẽ tái sử dụng cấu trúc mô hình U-Net từ bài toán trước.

## 3.1. Khai báo các tham số cho Model

```

1 device = "cuda" if torch.cuda.is_available() else "cpu"
2 model = UNet(n_channels=1, n_classes=2).to(device)
3 criterion = nn.CrossEntropyLoss()
4 optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

```

## 3.2. Model Training và Evaluation

```

1 best_loss = float('inf')
2 best_model_wts = copy.deepcopy(model.state_dict())
3
4 for epoch in range(30):
5     model.train()
6     train_loss = 0.0
7     for inputs, targets in train_loader:
8         inputs, targets = inputs.to(device), targets.to(device)
9         optimizer.zero_grad()
10        outputs = model(inputs)
11        loss = criterion(outputs, targets)
12        loss.backward()
13        optimizer.step()
14        train_loss += loss.item()
15    train_loss = train_loss / len(train_loader)
16
17    val_loss = evaluate(model, val_loader, criterion)
18
19    if val_loss < best_loss:
20        best_loss = val_loss
21        best_model_wts = copy.deepcopy(model.state_dict())
22
23    print(f"Epoch {epoch + 1}, Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}")

```

### 3.3. Xây dựng hàm để hiển thị kết quả dự đoán

```

1 def display_prediction(model, image, ground_truth, device):
2     image = image.unsqueeze(0).to(device)
3     with torch.no_grad():
4         prediction = model(image)
5         prediction = torch.argmax(prediction, dim=1).squeeze(0).cpu().numpy()
6
7     image_np = image.squeeze(0).permute(1, 2, 0).cpu().numpy()
8     ground_truth_np = ground_truth.cpu().numpy()
9
10    fig, axes = plt.subplots(1, 3, figsize=(15, 5))
11    axes[0].imshow(image_np, cmap='gray')
12    axes[0].set_title("Input Image")
13    axes[1].imshow(ground_truth_np, cmap='gray')
14    axes[1].set_title("Ground Truth")
15    axes[2].imshow(prediction, cmap='gray')
16    axes[2].set_title("Prediction")
17    for ax in axes:
18        ax.axis("off")
19    plt.show()

```

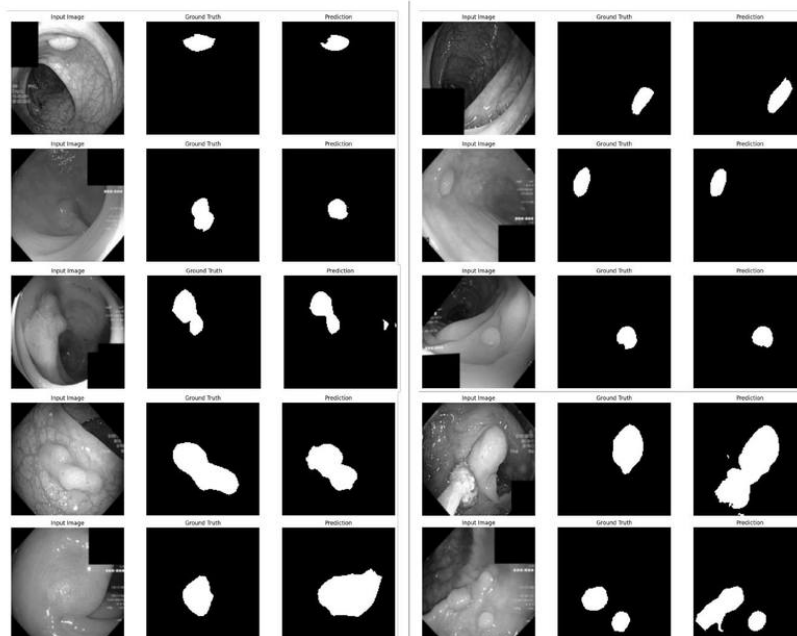
## 4. Load trọng số và dự đoán

Mô hình được sử dụng để dự đoán 10 ảnh đầu tiên từ tập kiểm tra nhằm đánh giá hiệu suất tổng thể của mô hình và đảm bảo rằng nó đạt được kết quả tốt nhất với trọng số đã lưu.

```

1 model.load_state_dict(best_model_wts)
2
3 n_test_points = 10
4 model.eval()
5
6 test_samples = [test_dataset[i] for i in range(n_test_points)]
7
8 for i in range(n_test_points):
9     img, gt = test_samples[i]
10    display_prediction(model, img, gt, device)

```



Hình 17: Kết quả dự đoán của mô hình sau khi huấn luyện.

- Hết -