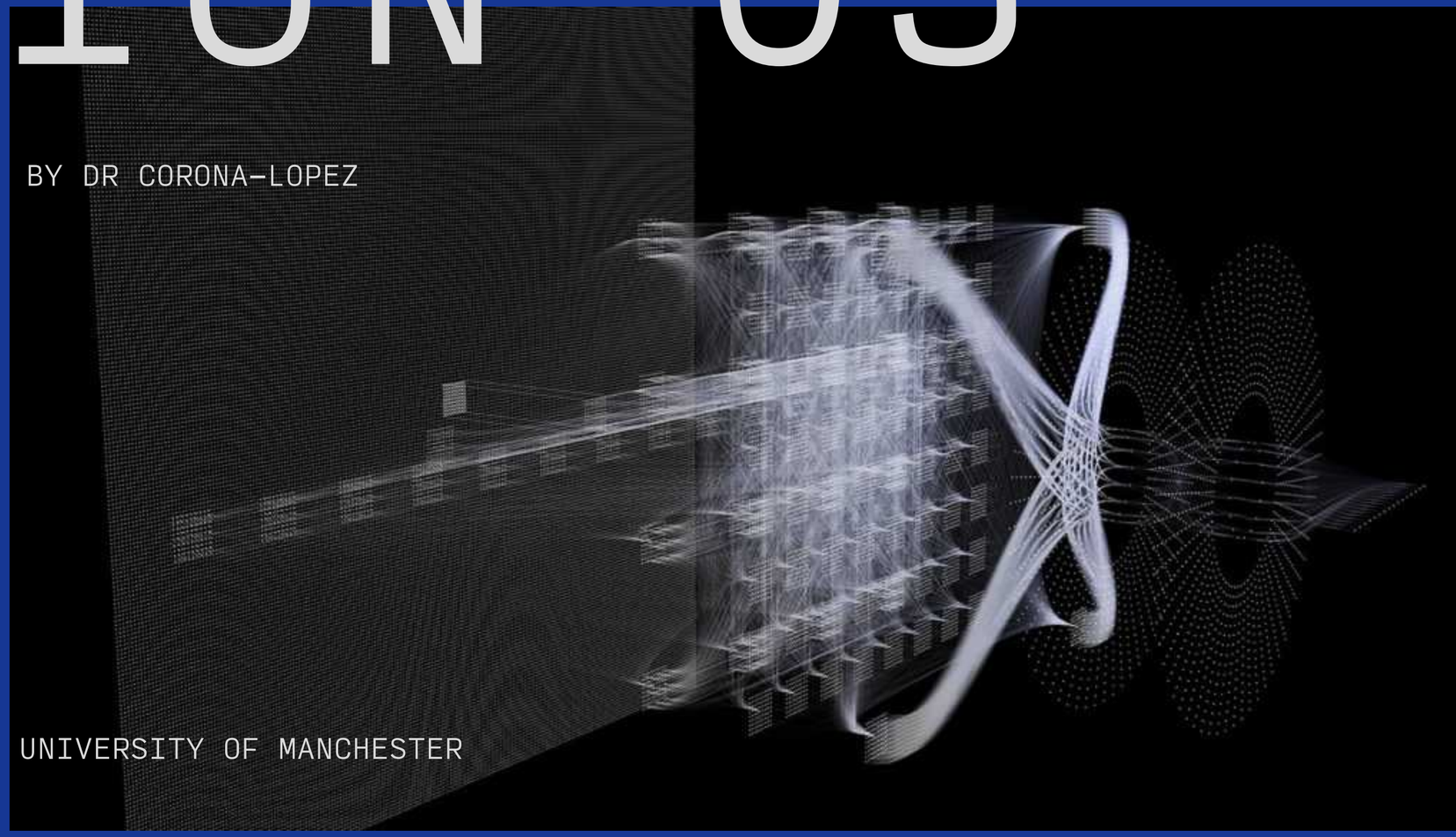
SESSION 05

WORKSHOP

MARCH 2026

TRANSFER LEARNING

BY DR CORONA-LOPEZ



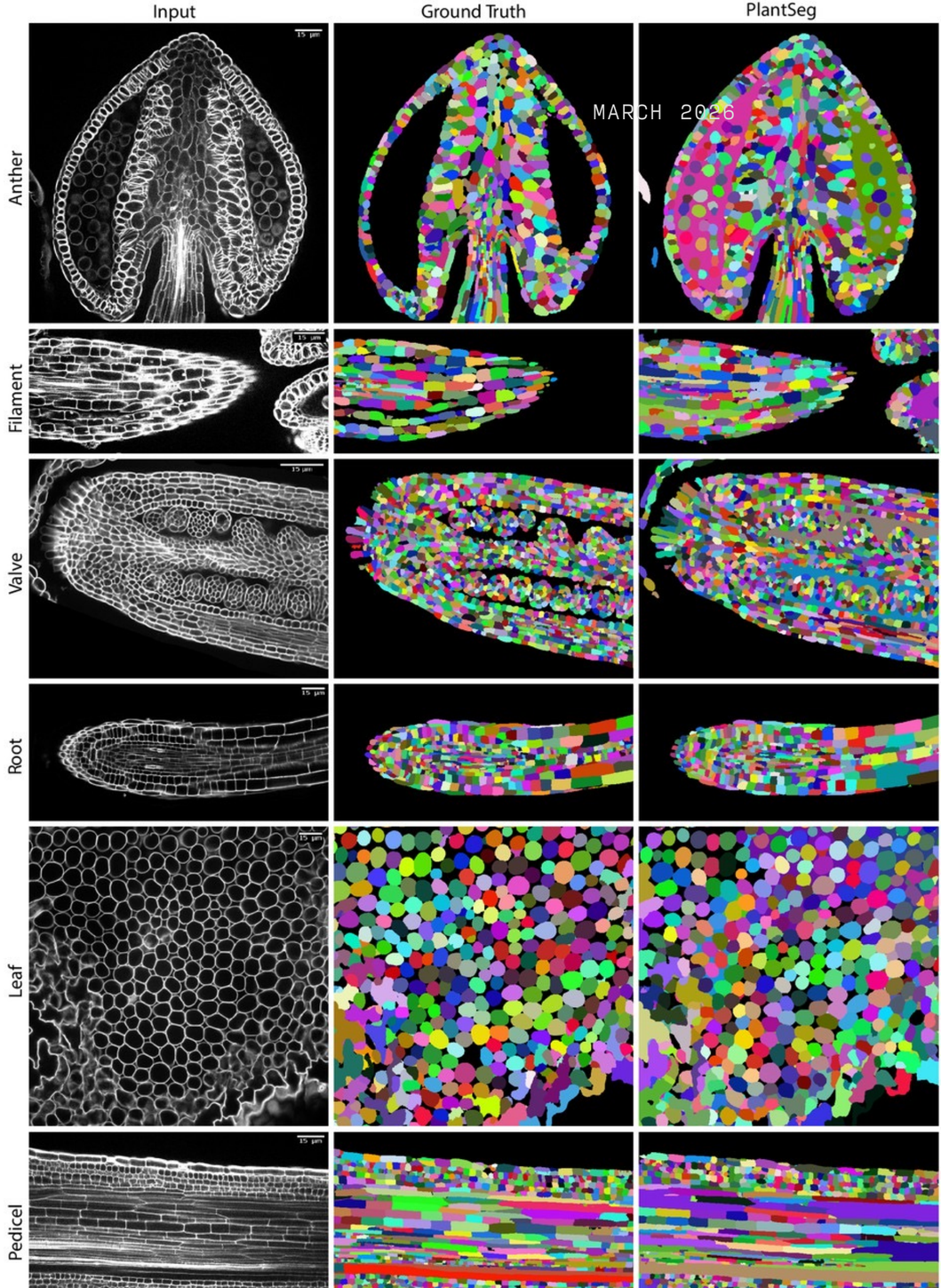
FACULTY OF SCIENCE AND ENGINEERING

UNIVERSITY OF MANCHESTER

BY DR CORONA-LOPEZ

AGENDA

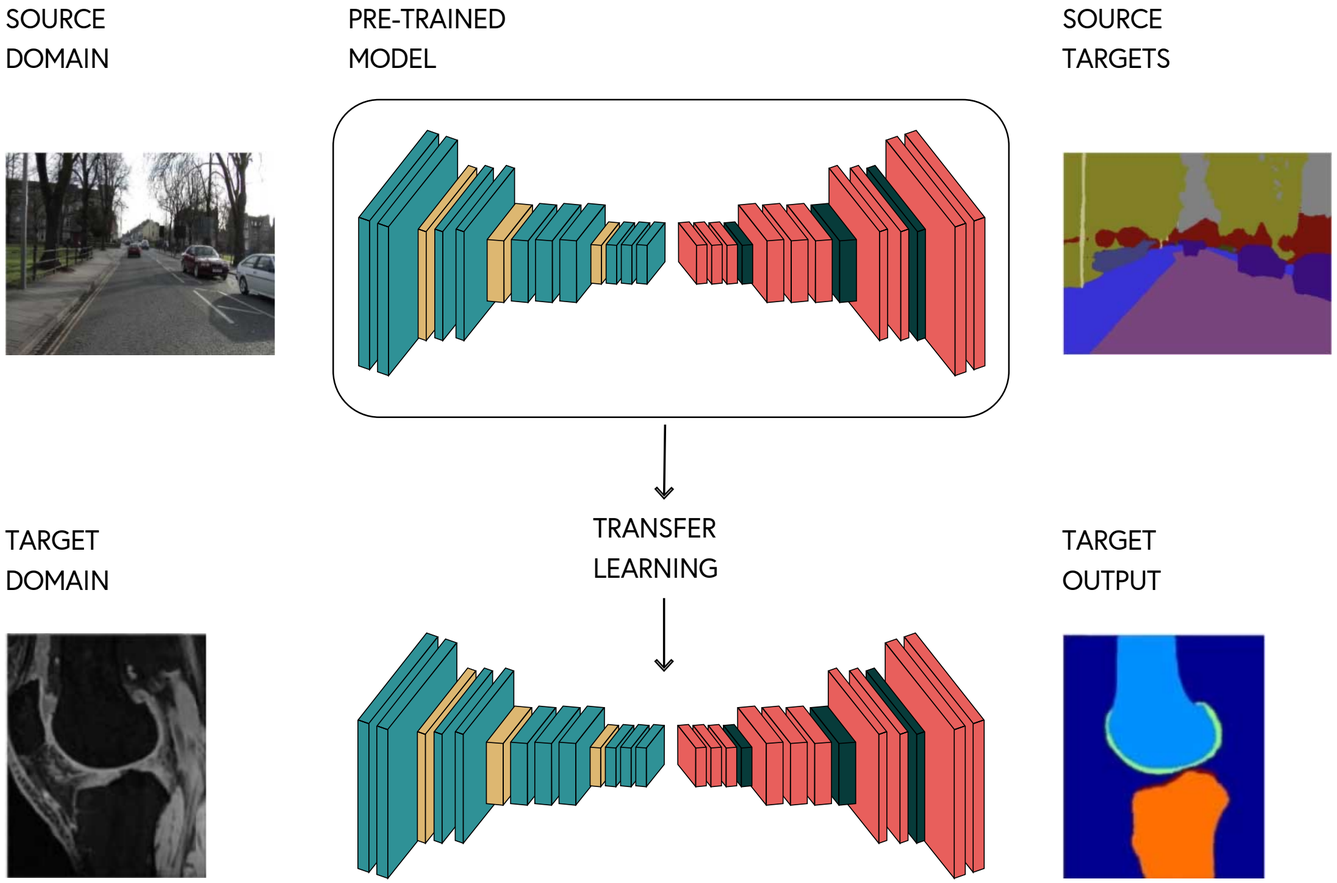
- **transfer learning:** core concepts, and when to use it
- **dataset preparation:** custom PyTorch dataset structure, computing dataset stats for normalization
- **albumentations:** synchronised image & mask augmentation
- **Baseline architecture & loss:** intro to U-Net structure, and tackling imbalance with dice loss
- **implementing transfer learning:** intro to EfficientNet architecture, building Efficient U-Net, handiling feature map mismatches, and freezing pre-trained weights



MARCH 2026

transfer learning is a technique where a model developed for one task is reused as a starting point for a model on a second task.

it leverages knowledge from pre-trained models instead of starting from scratch
it is particularly effective for deep learning models that require massive datasets and computational resources



USE CASES

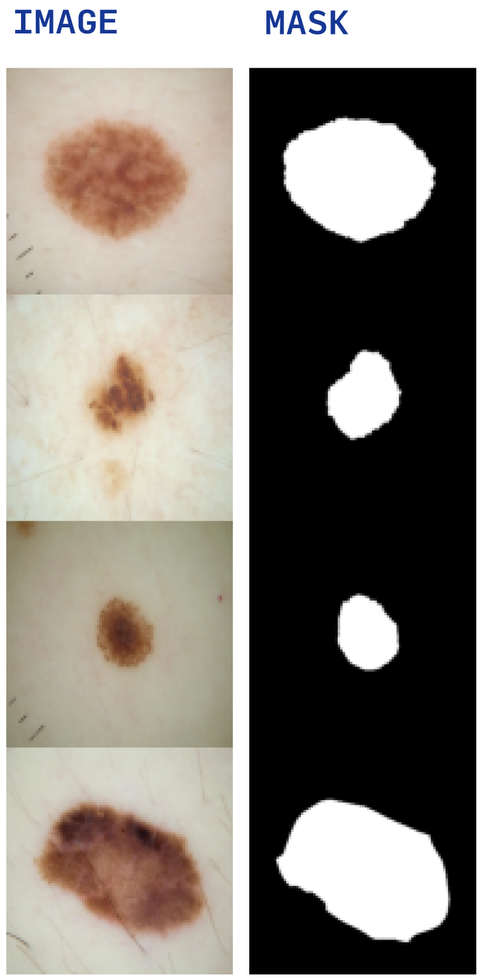
the **effectiveness** depends on similarity between source and target domains

SCENARIO	EXAMPLE	BENEFIT
LIMITED TRAINING DATA	medical imaging with few samples	pre-trained features compensate for data scarcity
SIMILAR DOMAINS	from natural images to satellite imagery	underlying features (edges, textures) transfer well
TIME CONSTRAINTS	rapid prototyping needs	accelerates model development cycle
HARDWARE LIMITATIONS	training with limited gpu access	reduces computational requirements
PREVENTING OVERFITTING	small dataset applications	regularization effect from pre-trained weights

ISIC 2016

medical datasets like ISIC are typically smaller than general computer vision datasets, making transfer learning particularly valuable

- contains dermoscopic images of skin lesions with expert-annotated segmentation masks
- 900 training images and 379 test images with corresponding binary masks
- critical for developing automated diagnostic tools for early melanoma detection
- challenging due to varying lesion sizes, shapes, colours, and skin types



DATA NORMALIZATION

```python
# Conceptual loop for computing statistics
channel_sum = torch.tensor([0.0, 0.0, 0.0])
channel_squared_sum = torch.tensor([0.0, 0.0, 0.0])
num_batches = 0

for images, _ in dataloader:
    # images shape: [batch_size, 3, height, width]
    channel_sum += torch.mean(images, dim=[0, 2, 3])
    channel_squared_sum += torch.mean(images ** 2, dim=[0, 2, 3])
    num_batches += 1

mean = channel_sum / num_batches
std = (channel_squared_sum / num_batches - mean ** 2) ** 0.5
```

pixel values typically range from 0 to 255. neural networks struggle with large, unscaled inputs, leading to slow and unstable training.

the solution: we normalize the dataset using z-score normalization: $x_{normalized} = \frac{x-\mu}{\sigma}$

while we can use the generic imagenet statistics (mean: [0.485, 0.456, 0.406]), medical images (like pink/brown skin lesions) have a completely different color distribution.

calculating the exact dataset statistics yields better results.

how it works: we iterate over the entire training set to calculate the sum and squared sum of pixel values across the red, green, and blue channels.

DATA AUGMENTATION & SYNC

```python
import albumentations as A

# Albumentations handles BOTH image and mask simultaneously
transform = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.VerticalFlip(p=0.5),
    A.RandomRotate90(p=0.5)
])

# Usage in __getitem__:
# augmented = transform(image=image_array, mask=mask_array)
```

in standard image classification, if you rotate a picture of a cat 90 degrees, the label remains "cat".

the segmentation way (sync): in segmentation, the "label" is a pixel-perfect mask. if you flip the image, you must flip the mask in the exact same way. if they fall out of sync, the network learns garbage.

the solution: standard torchvision struggles with synced mask augmentation.

this is why we switch to the albumentations library, which handles synchronized (image, mask) transforms natively.

DATA AUGMENTATION & SPLITTING

```
# Inside the updated ISICDataset __getitem__ method:
    if self.transform:
        # Albumentations expects named arguments for sync transforms!
        augmented = self.transform(image=image_np, mask=mask_np)
        image = augmented['image']
        mask = augmented['mask']

# --- Splitting the Data ---
from torch.utils.data import random_split

total_size = len(full_dataset)
train_size = int(0.8 * total_size)
val_size= total_size - train_size

train_dataset, val_dataset = random_split(full_dataset,
                                          [train_size, val_size])
```

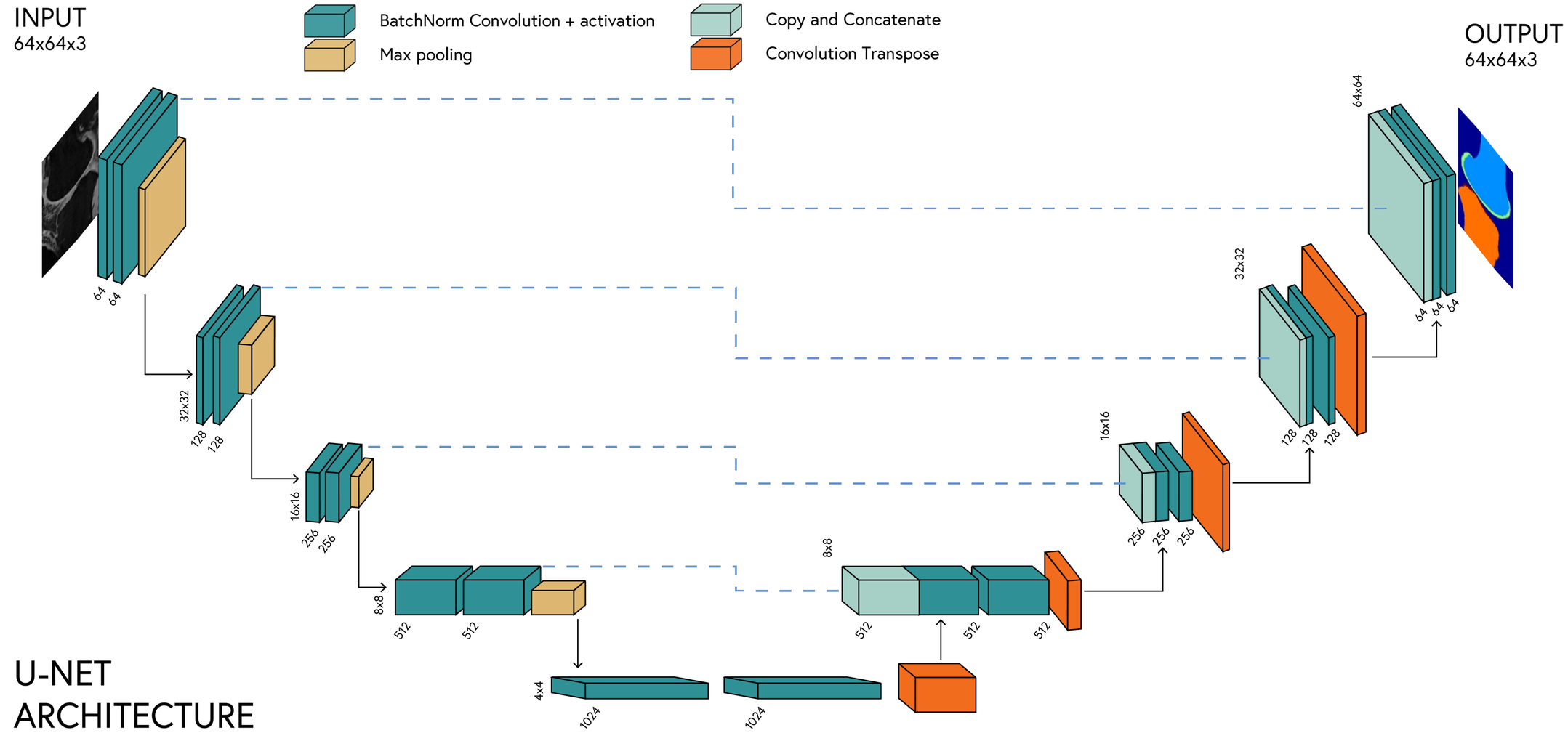
the wrapper: because albumentations requires numpy arrays and returns a dictionary, we need to update our dataset's __getitem__ method to properly pass both the image and mask to the transform function.

train/validation split: we never train on all our data. we must reserve a portion (usually 20%) to validate that our model is actually learning generalizable features, not just memorizing the training images.

pytorch random_split: a built-in utility that randomly divides the dataset into non-overlapping subsets.

U-NET

- **encoder-decoder architecture** with **skip** connections
- captures both context and localisation information
- widely used for biomedical image segmentation

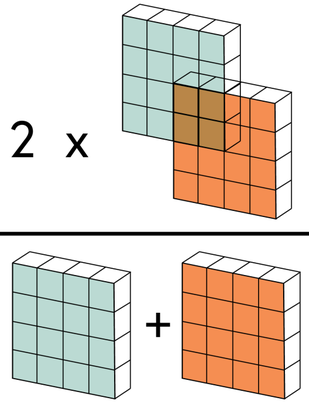


08 U-NET ARCHITECTURE

DICE LOSS

the class imbalance problem: in medical imaging, the region of interest (e.g., a skin lesion) often occupies only 5% of the image, while healthy skin/background is 95%.

$$\text{DICE} = \frac{2 \times \text{OVERLAP}}{\text{TOTAL AREA}} = \frac{2 \times}{+}$$



dice loss:

- measures **overlap** between predicted and ground truth masks
- handles **class imbalance** better than binary cross-entropy

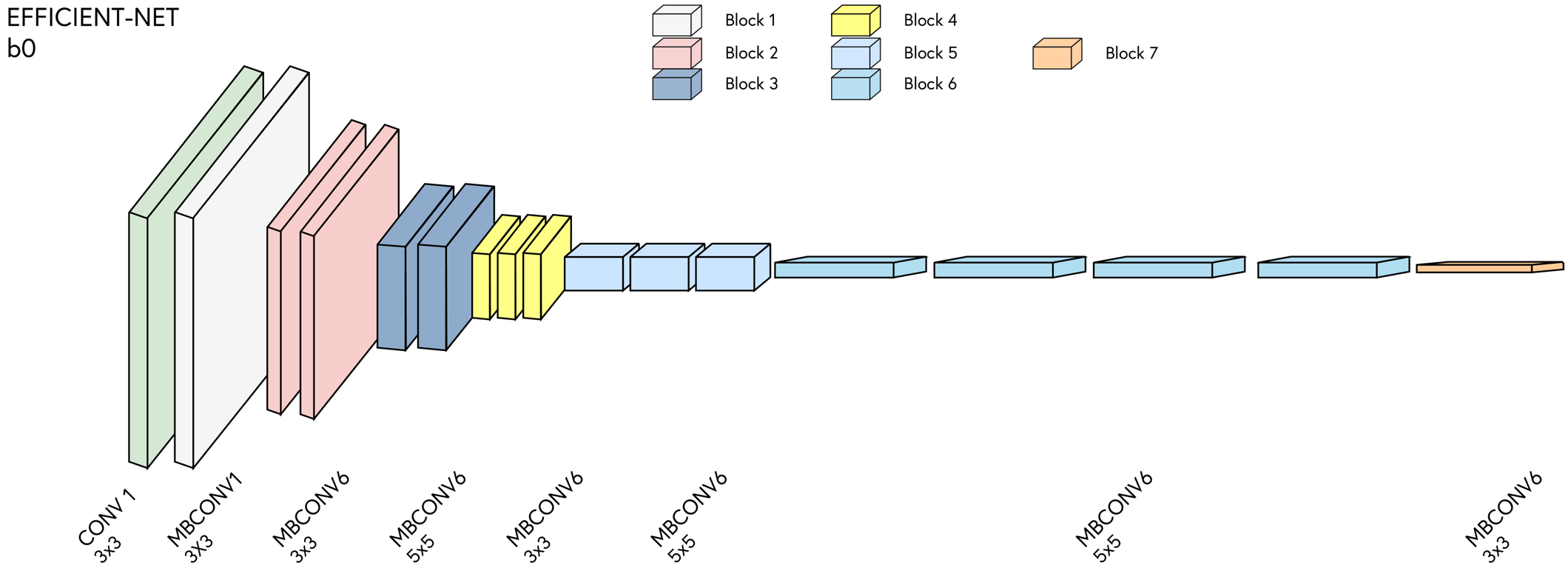
$$\text{DiceLoss} = 1 - \frac{2|X \cap Y|}{|X| + |Y|}$$

```python
class DiceLoss(nn.Module):
    def __init__(self, smooth=1e-6):
        super().__init__()
        self.smooth = smooth # Prevents division by zero

    def forward(self, pred, target):
        # Flatten the tensors
        pred = pred.view(-1)
        target = target.view(-1)

        # Calculate Intersection and Union
        intersection = (pred * target).sum()
        dice = (2. * intersection + self.smooth) / (pred.sum() + target.sum() + self.smooth)

        return 1 - dice # Return the loss to be minimized
```

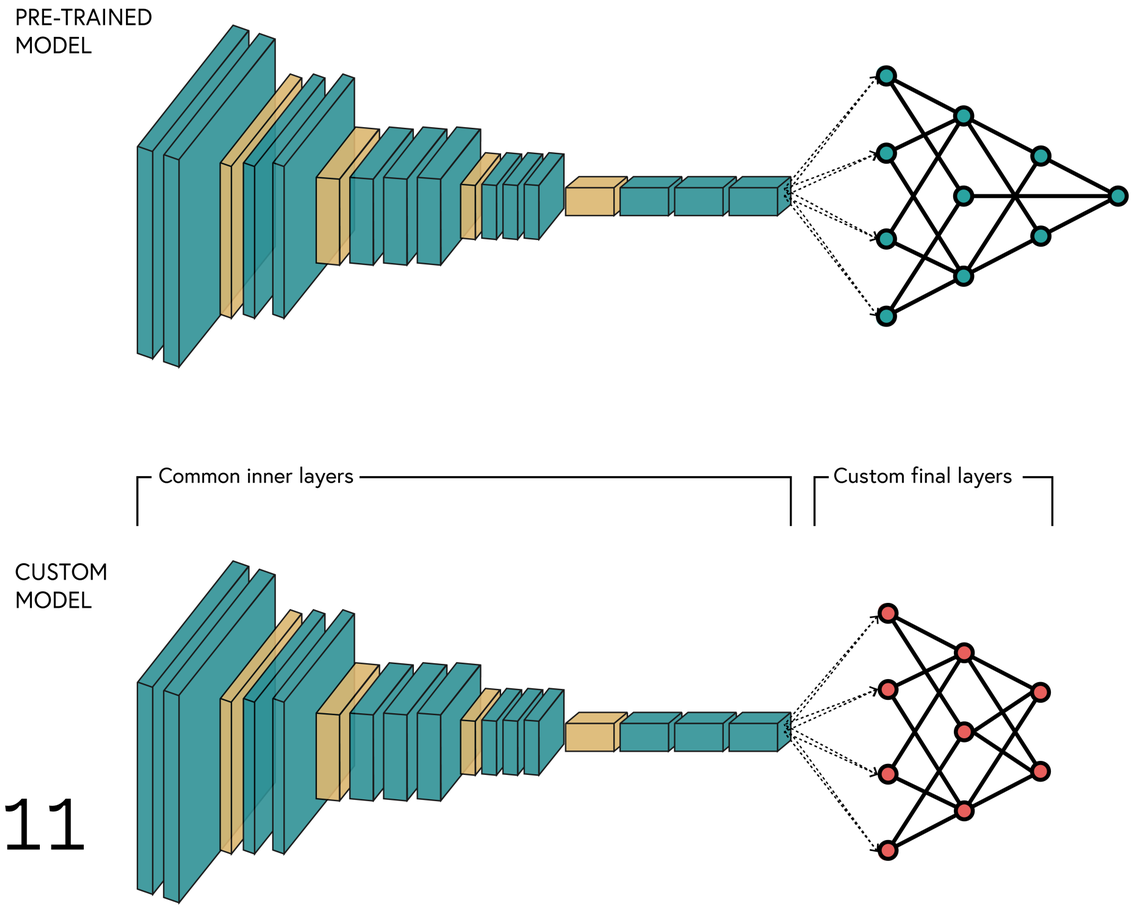
EFFICIENT-NET

introduced by google. achieves state-of-the-art accuracy while being significantly smaller and faster than previous models like ResNet or VGG.

trained on millions of images (image-net), it already knows how to detect edges, textures, and complex shapes perfectly.



TRANSFER LEARNING



Step 1: Select a Pre-Trained Model

Choose a model architecture that has already been trained on a massive, diverse dataset

Step 2: Modify the Architecture

Strip away the original "head" of the model.

Step 3: Freeze the Base Layers

Lock the weights of the pre-trained layers (requires_grad = False).

Step 4: Add and Train New Layers

Attach your new, untrained layers tailored to your specific task (our U-Net Decoder).

Step 5: Fine-Tuning (Optional but powerful)

Once the new layers are stable, "unfreeze" the entire model. Train for a few more epochs using a very small learning rate.

PRE-TRAINED WEIGHTS

the torchvision.models API: pytorch makes it incredibly simple to download pre-trained models.

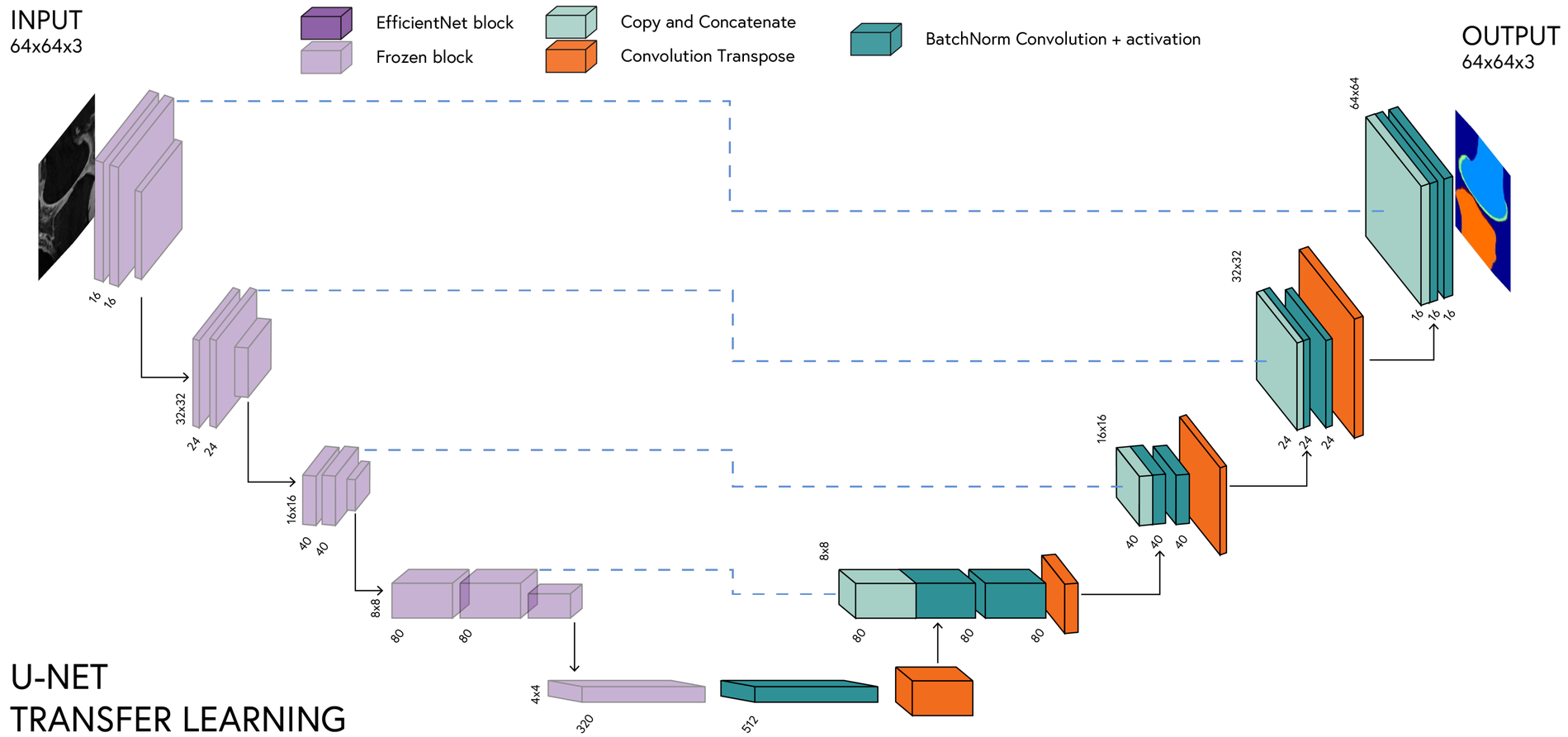
the weights object: we specify IMAGENET1K_V1 to ensure we get the model that has already learned from the massive imagenet dataset.

feature extraction (features): we don't want the whole classification model. we specifically target the .features module of efficientnet to extract the intermediate feature maps needed for our u-net skip connections.

```python
import torchvision.models as models
from torchvision.models import EfficientNet_B0_Weights

# 1. Download the pre-trained EfficientNet-B0 model
weights = EfficientNet_B0_Weights.IMAGENET1K_V1
efficientnet = models.efficientnet_b0(weights=weights)

# 2. Isolate the feature extractor (ignoring the classification head)
encoder_features = efficientnet.features

# Note: We will tap into specific blocks (e.g., layers 1, 3, 5)
# to act as our skip connections for the U-Net Decoder!
```

EFFICIENT U-NET

the hybrid architecture: we are fusing two ideas: the powerful feature extraction of efficientnet (encoder) + the precise spatial localization of u-net (decoder).



U-NET
TRANSFER LEARNING

FREEZING

the danger of random initialization: our new u-net decoder has completely random weights. if we start training immediately, the massive errors from the random decoder will backpropagate into our pre-trained encoder.

the solution -> "freezing": we must turn off gradient tracking (requires_grad = false) for the efficientnet encoder during the initial training phase.

the workflow: train only the decoder for a few epochs until it learns how to use the encoder's features. then, optionally "unfreeze" the encoder and train everything together with a tiny learning rate (fine-tuning).

```
class EfficientUNet(nn.Module):
    def __init__(self):
        super().__init__()
        # Load pre-trained encoder
        self.encoder = models.efficientnet_b0(
          weights=EfficientNet_B0_Weights.IMAGENET1K_V1).features

        # FREEZE the encoder weights!
        for param in self.encoder.parameters():
            param.requires_grad = False

        # Define the randomly initialized Decoder...
        # self.decoder_block1 = ...

    def forward(self, x):
        # ...
```

SMART-UP

standard u-nets are perfectly symmetrical.

however, when you swap in an external pre-trained encoder like efficientnet, the spatial dimensions (height/width) of the skip connections might not perfectly match your decoder's upsampled feature maps due to different padding or strides in the efficientnet blocks.

the error: if you try to concatenate two tensors with different spatial sizes (e.g., a 56x56 map with a 55x55 map), pytorch will throw a runtime shape error.

the solution: after the standard transposed convolution, we check if the sizes match. if they don't, we dynamically resize the upsampled tensor to perfectly match the skip connection using bilinear interpolation.

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

class SmartUpBlock(torch.nn.Module):
    # ... (__init__ defines self.up and self.conv) ...

    def forward(self, x1, x2):
        # x1: from previous decoder layer
        # x2: skip connection from encoder

        x1 = self.up(x1) # Initial upsampling

        # Check spatial dimensions (ignoring batch and channel dims)
        if x1.size()[2:] != x2.size()[2:]:
            # Force x1 to match the spatial size of x2
            x1 = F.interpolate(
                x1,
                size=x2.size()[2:],
                mode='bilinear',
                align_corners=False # Prevents edge artifacts
            )

        # Now it is safe to concatenate along the channel dimension!
        x = torch.cat([x2, x1], dim=1)

        return self.conv(x)
```