U-Net for Image Classification

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Project Overview

This project explores the implementation and application of the U-Net architecture, initially developed for image segmentation tasks. Our objective is to first apply U-Net to segment images effectively in various datasets, such as medical and satellite imagery. Subsequently, we adapt the U-Net architecture for image classification by modifying its network structure. The goal is to demonstrate the versatility of U-Net in handling different types of deep learning tasks in computer vision, assessing its performance in both segmentation and classification scenarios.

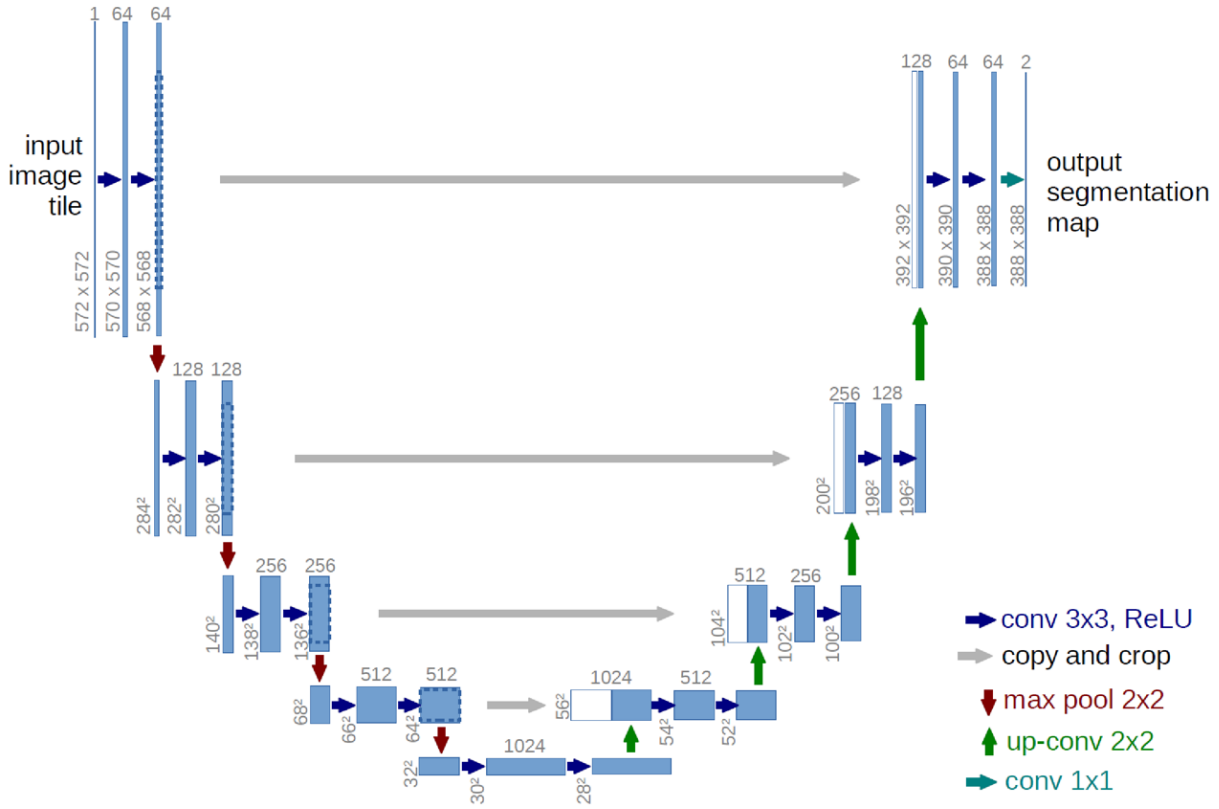
In [15]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

In [18]:
```
# Specify the path to your image file
image_path = 'UNET_architecture.png'

# Load the image using matplotlib's imread function
img = mpimg.imread(image_path)

# Create a larger figure with higher DPI
plt.figure(figsize=(10, 8), dpi=300)

plt.imshow(img, interpolation='nearest')
plt.axis('off')
plt.show()
```



2. Build U-Net Architecture

The U-Net model is constructed using PyTorch, featuring an encoder-decoder structure with skip connections. Key components include:

- **Encoder (Downsampling Path):** Sequential convolutional blocks reduce the spatial dimensions while increasing the feature depth.
- **Bottleneck:** The lowest resolution is processed here, allowing the network to learn the most abstract features.
- **Decoder (Upsampling Path):** Transposed convolutions are used to increase spatial dimensions, paired with skip connections from the encoder to preserve high-resolution details.
- **Final Layer:** The output of the last upsampling step is passed through a convolutional layer to map the deep feature maps to the desired output channels.

In [19]:
```
import torch
import torch.nn as nn
import torchvision.transforms.functional as TF
```

In [20]:
```
class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DoubleConv, self).__init__()
        self.conv = nn.Sequential(
```

```
            nn.Conv2d(in_channels, out_channels, 3, 1, 1, bias=False), #
            nn.BatchNorm2d(out_channels), # NOT IN THE PAPER! (batch norm
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self,x):
        return self.conv(x)
```

In [21]:

```
class UNET(nn.Module):
    def __init__(self, in_channels=3, out_channels=1, features=[64, 128,
        super(UNET, self).__init__()
        self.ups = nn.ModuleList()
        self.downs = nn.ModuleList()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Down part of UNET
        for feature in features:
            self.downs.append(DoubleConv(in_channels, feature))
            in_channels = feature

        # Up part of UNET
        for feature in reversed(features):
            self.ups.append(
                nn.ConvTranspose2d(
                    feature*2, feature, kernel_size=2, stride=2,
                )
            )
            self.ups.append(DoubleConv(feature*2, feature))

        # Bottleneck
        self.bottleneck = DoubleConv(features[-1], features[-1]*2)

        # Final layer
        self.final_conv = nn.Conv2d(features[0], out_channels, kernel_siz

    def forward(self, x):
        skip_connections = []
        for down in self.downs:
            x = down(x)
            skip_connections.append(x)
            x = self.pool(x)

        x = self.bottleneck(x)
        skip_connections = skip_connections[::-1]

        for idx in range(0, len(self.ups), 2): # Up & DoubleConv is a sin
            x = self.ups[idx](x)
            skip_connection = skip_connections[idx//2]

            if x.shape != skip_connection.shape:# reshape
```

```
                x = TF.resize(x, size=skip_connection.shape[2:]) # only n

            concat_skip = torch.cat((skip_connection, x), dim=1)
            x = self.ups[idx+1](concat_skip)

        return self.final_conv(x)
```

One problem that could occur: Lets say the input layer is 161x161. The max pool will going to floor the div by 2 and create an 80x80. Then, during the up sample it will create an output of 160x160. This means we won't be able to concatenate the two.

To solve this, always choose an input that is divisible by 16.

In [22]:
```
def check_size():
    x = torch.randn((3, 1, 161, 161))
    model = UNET(in_channels=1, out_channels=1)
    preds = model(x)
    print(preds.shape)
    print(x.shape)
    assert preds.shape == x.shape

if __name__ == "__main__":
    check_size()
```

```
torch.Size([3, 1, 161, 161])
torch.Size([3, 1, 161, 161])
```

2. Try Image Segmentation

Using the U-Net model, perform image segmentation on a dataset. The process involves three main steps:

Dataset Preparation

- **Objective:** Load and preprocess the dataset suitable for segmentation tasks.
- **Details:** Choose datasets that are commonly used for segmentation tasks, such as medical images (e.g., MRI or CT scans) or satellite images.
- **Preprocessing Steps:** Include normalization, resizing images to a consistent dimension, and possibly augmenting the dataset to improve model robustness.

Training

- **Objective:** Train the U-Net model on the prepared dataset.
- **Loss Function:** Use an appropriate loss function for segmentation. Common choices include:
  - **Dice Loss:** Useful for dealing with class imbalance in images.
  - **Cross-Entropy Loss:** Standard loss for segmentation tasks.

- **Optimization:** Use optimizers like Adam or SGD to minimize the loss function.

Evaluation

- **Objective:** Validate the model's performance on a separate test set.
- **Metrics:** Evaluate the model using segmentation-specific metrics such as:
  - **IoU (Intersection over Union):** Measures the overlap between the predicted segmentation and the ground truth.
  - **Dice Coefficient:** Similar to IoU, useful for assessing the quality of the segmentation.

First, we are going to download the dataset for Carvana Image Masking Challenge

In [51]:
```
import os
from PIL import Image
from torch.utils.data import Dataset
import numpy as np
import torchvision
```

In [63]:
```
class CarvanaDataset(Dataset):
    def __init__(self, image_dir, mask_dir, transform=None):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.transform = transform
        self.images = os.listdir(image_dir)

    def __len__(self):
        return len(self.images)

    def __getitem__(self, index):
        img_path = os.path.join(self.image_dir, self.images[index])
        mask_path = os.path.join(self.mask_dir, self.images[index].replac
        image = np.array(Image.open(img_path).convert("RGB"))
        mask = np.array(Image.open(mask_path).convert("L"), dtype=np.floa
        mask[mask == 255.0] = 1.0 # becuase we are going to use sigmoid a

        if self.transform is not None:
            augmentations = self.transform(image=image, mask=mask)
            image = augmentations["image"]
            mask = augmentations["mask"]

        return image, mask
```

In [64]:
```
import torch
import albumentations as A
from albumentations.pytorch import ToTensorV2
from tqdm import tqdm
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
```

In [91]:
```
def save_checkpoint(state, filename="my_checkpoint.pth.tar"):
    print("=> Saving Checkpoint")
    torch.save(state, filename)

def load_checkpoint(checkpoint, model):
    print("=> Loading checkpoint")
    model.load_state_dict(checkpoint["state_dict"])
```

In [92]:
```
def get_loaders(
    train_dir,
    train_maskdir,
    val_dir,
    val_maskdir,
    batch_size,
    train_transform,
    val_transform,
    num_workers=4,
    pin_memory=True,
):
    train_ds = CarvanaDataset(
        image_dir=train_dir,
        mask_dir=train_maskdir,
        transform=train_transform,
    )

    train_loader = DataLoader(
        train_ds,
        batch_size=batch_size,
        num_workers=num_workers,
        pin_memory=pin_memory,
        shuffle=True,
    )

    val_ds = CarvanaDataset(
        image_dir=val_dir,
        mask_dir=val_maskdir,
        transform=val_transform,
    )

    val_loader = DataLoader(
        val_ds,
        batch_size=batch_size,
        num_workers=num_workers,
        pin_memory=pin_memory,
        shuffle=False,
    )

    return train_loader, val_loader
```

In [93]:
```
def check_accuracy(loader, model, device="cuda"):
    num_correct = 0
    num_pixels = 0
```

```
    dice_score = 0
    model.eval()

    with torch.no_grad():
        for x, y in loader:
            x = x.to(device)
            y = y.to(device).unsqueeze(1)
            preds = torch.sigmoid(model(x))
            preds = (preds > 0.5).float()
            num_correct += (preds == y).sum()
            num_pixels += torch.numel(preds)
            dice_score += (2 * (preds * y).sum()) / (
                (preds + y).sum() + 1e-8
            )

    print(
        f"Got {num_correct}/{num_pixels} with acc {num_correct/num_pixels
    )
    print(f"Dice score: {dice_score/len(loader)}")
    model.train()
```

In [94]:
```
def save_predictions_as_imgs(loader, model, folder="saved_images/", devic
    model.eval()
    for idx, (x, y) in enumerate(loader):
        x = x.to(device=device)
        with torch.no_grad():
            preds = torch.sigmoid(model(x))
            preds = (preds > 0.5).float()
        torchvision.utils.save_image(
            preds, f"{folder}/pred_{idx}.png"
        )
        torchvision.utils.save_image(y.unsqueeze(1), f"{folder}{idx}.png"

    model.train()
```

In [83]:
```
import torch
import albumentations as A
from albumentations.pytorch import ToTensorV2
from tqdm import tqdm
import torch.nn as nn
import torch.optim as optim
```

In [95]:
```
# Hyperparameters etc.
LEARNING_RATE = 1e-4
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
BATCH_SIZE = 16
NUM_EPOCHS = 3
NUM_WORKERS = 2
IMAGE_HEIGHT = 160  # 1280 originally
IMAGE_WIDTH = 240  # 1918 originally
PIN_MEMORY = True
LOAD_MODEL = False
TRAIN_IMG_DIR = "drive/MyDrive/data/train_images/"
```

```
TRAIN_MASK_DIR = "drive/MyDrive/data/train_masks/"
VAL_IMG_DIR = "drive/MyDrive/data/val_images/"
VAL_MASK_DIR = "drive/MyDrive/data/val_masks/"
```

In [96]:
```
def train_fn(loader, model, optimizer, loss_fn, scaler): # going to do 1
    loop = tqdm(loader)

    for batch_idx, (data, targets) in enumerate(loop):
        data = data.to(device=DEVICE)
        targets = targets.float().unsqueeze(1).to(device=DEVICE)

        # forward
        with torch.cuda.amp.autocast():
            predictions = model(data)
            loss = loss_fn(predictions, targets)

        # backward
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        # update tqdm loop
        loop.set_postfix(loss=loss.item())
```

In [97]:
```
def main():
    train_transform = A.Compose( # train data
        [
            A.Resize(height=IMAGE_HEIGHT, width=IMAGE_WIDTH),
            A.Rotate(limit=35, p=1.0),
            A.HorizontalFlip(p=0.5),
            A.VerticalFlip(p=0.1),
            A.Normalize(
                mean=[0.0, 0.0, 0.0],
                std=[1.0, 1.0, 1.0],
                max_pixel_value=255.0, # value between 0 and 1
            ),
            ToTensorV2(),
        ],
    )

    val_transforms = A.Compose( # validation data
        [
            A.Resize(height=IMAGE_HEIGHT, width=IMAGE_WIDTH),
            A.Normalize(
                mean=[0.0, 0.0, 0.0],
                std=[1.0, 1.0, 1.0],
                max_pixel_value=255.0,
            ),
            ToTensorV2(),
        ],
    )
```

```
    model = UNET(in_channels=3, out_channels=1).to(DEVICE)
    loss_fn = nn.BCEWithLogitsLoss() # change for multiclass
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

    train_loader, val_loader = get_loaders(
        TRAIN_IMG_DIR,
        TRAIN_MASK_DIR,
        VAL_IMG_DIR,
        VAL_MASK_DIR,
        BATCH_SIZE,
        train_transform,
        val_transforms,
        NUM_WORKERS,
        PIN_MEMORY,
    )

    if LOAD_MODEL:
        load_checkpoint(torch.load("my_checkpoint.pth.tar"), model)


    check_accuracy(val_loader, model, device=DEVICE)
    scaler = torch.cuda.amp.GradScaler()

    for epoch in range(NUM_EPOCHS):
        train_fn(train_loader, model, optimizer, loss_fn, scaler)

        # save model
        checkpoint = {
            "state_dict": model.state_dict(),
            "optimizer":optimizer.state_dict(),
        }
        save_checkpoint(checkpoint)

        # check accuracy
        check_accuracy(val_loader, model, device=DEVICE)

        # print some examples to a folder
        save_predictions_as_imgs(
            val_loader, model, folder="drive/MyDrive/", device=DEVICE
        )
```

In [98]:
```
main()
```

```
Got 30820348/39091200 with acc 78.84
Dice score: 0.0
100%|██████████| 255/255 [02:45<00:00,  1.54it/s, loss=0.147]
=> Saving Checkpoint
Got 38312900/39091200 with acc 98.01
Dice score: 0.9542471170425415
100%|██████████| 255/255 [02:49<00:00,  1.51it/s, loss=0.0913]
=> Saving Checkpoint
Got 38717418/39091200 with acc 99.04
Dice score: 0.9777600765228271
```

```
100%|██████████| 255/255 [02:48<00:00,  1.51it/s, loss=0.0623]
=> Saving Checkpoint
Got 38839229/39091200 with acc 99.36
Dice score: 0.9848598837852478
```

3. Modify the Code for Image Classification

To adapt the U-Net model for image classification tasks, follow these modifications:

Adapt Network Structure

- **Objective:** Modify the U-Net model to output class probabilities.
- **Modifications:**
  - Add a global average pooling layer after the last convolutional layer to reduce each feature map to a single number.
  - Append a fully connected layer (or layers) that outputs the probability for each class.

Change Loss Function

- **Objective:** Utilize a suitable loss function for classification.
- **Loss Function:** For multi-class classification, use **Cross-Entropy Loss**, which is well-suited for discrete probability distributions.

Output Adjustments

- **Objective:** Ensure the model outputs match the number of target classes.
- **Details:**
  - Adjust the final layer to have as many outputs as there are classes.
  - Use a softmax activation function in the final layer to ensure the output values represent probabilities.

In [ ]:

In [ ]: