UNET-A Annotated Guide

> Based on: U-Net: Convolutional Networks for Biomedical Image Segmentation

> By :- Olaf Ronneberger, Philipp Fischer, Thomas Brox

> Venue :- MICCAI (Medical Image Computing and Computer-Assisted Intervention)

Table of Contents

Abstract

Image segmentation plays a crucial role in various fields, particularly in biomedical imaging. Traditional methods often struggle with limited generalization and precision, especially when working with small datasets. The U-Net architecture, introduced in 2015, has revolutionized this domain by offering a fully convolutional network optimized for pixel-level classification. Its unique "U"-shaped design combines an encoder-decoder structure with skip connections, preserving spatial information and ensuring precise localization.

This project implements and evaluates the U-Net model for the Carvana Image Masking Challenge dataset, a benchmark for high-resolution image segmentation. An annotated implementation of the U-Net architecture is developed, providing a step-by-step explanation of each component, including its encoder-decoder paths, skip connections, and training mechanisms. The annotated code serves as a valuable resource for beginners and practitioners aiming to understand the model's design, functionality, and application.

The project explores the architecture's strengths, such as its ability to handle limited training data and produce real-time segmentation, while addressing its limitations, including high computational requirements and sensitivity to input dimensions. Recommendations for potential improvements, such as attention mechanisms, multi-scale feature fusion, and domain adaptation, are also discussed.

Through this annotated implementation and practical evaluation, the project not only highlights U-Net's transformative impact but also empowers learners and professionals to adopt and extend the model for diverse segmentation tasks.

Background

Traditional image segmentation techniques like thresholding, region growing, and edge detection were used before deep learning. These methods relied heavily on human prior knowledge and had limited generalization ability.During the rise of CNN,breakthrough came with convolutional neural networks (CNNs) achieving pixel-level classification for image segmentation. Deep CNNs demonstrated a strong ability to extract features and rapidly developed for computer vision applications.Fully Convolutional Networks was an important milestone was the development of FCNs, which adapted classification networks like VGG for segmentation by converting fully connected layers to convolutional layers. This allowed end-to-end training for pixel-wise prediction.

The U-Net architecture was introduced in 2015, U-Net became a seminal architecture for biomedical image segmentation. Its key innovation was the symmetric expanding path that enables precise localization, combined with skip connections to preserve fine details.

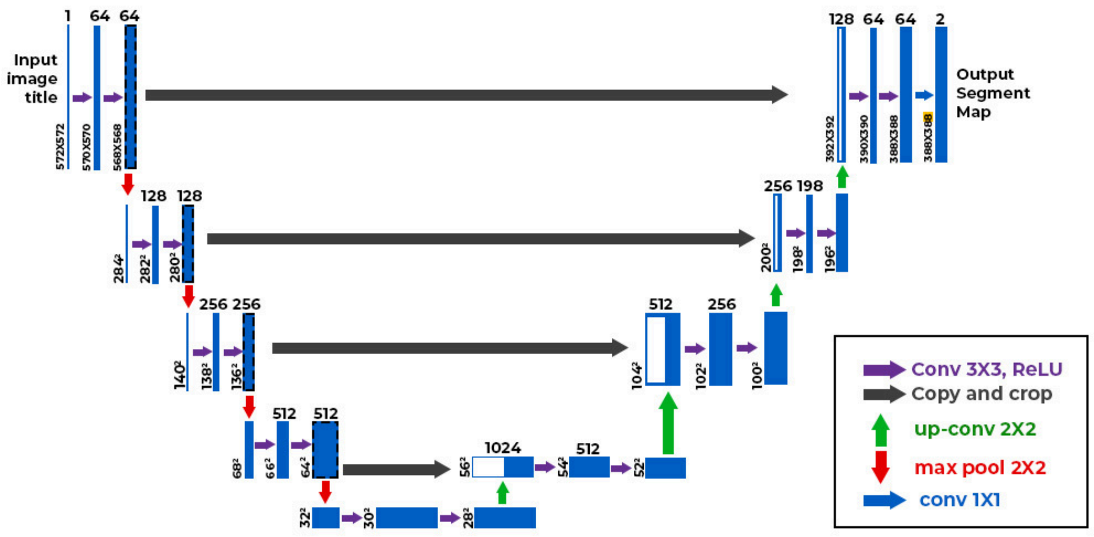
Challenges in biomedical image segmentation

The U-Net paper was developed to address several key challenges in biomedical image segmentation:

1. **Limited training data:** There was a need for a method that could perform well with very few annotated training samples, as obtaining large annotated datasets in the biomedical field is often difficult and time-consuming.
2. **Precise localization:** Biomedical image segmentation requires high precision in localizing structures and boundaries. The U-Net architecture was designed to capture both context and precise localization.
3. **Speed requirements:** Fast segmentation was necessary for practical applications in biomedical imaging. The U-Net architecture aimed to provide quick segmentation results.
4. **Versatility:** There was a need for a single architecture that could be applied to various biomedical segmentation tasks without significant modifications.
5. **Performance improvement:** The authors aimed to outperform existing methods, particularly the sliding-window convolutional network approach that was considered state-of-the-art at the time.

Overview of U-Net architecture

U-Net is a type of convolutional neural network (CNN). which is used for image segmentation tasks mostly in medical imaging. the architecture is developed by Olaf Ronneberger, Philipp Fischer, and Thomas Brox, is famous for its symmetric, "U"-shaped design, which consists of an encoder-decoder structure.



The U shape architechture consists of a **contracting path** and an **expansive path**.

1. The contracting path contains encoder layers that reduces the spatial resolution of the input and capture contextual information from the input .
2. The expansive path contains decoder layers that takes the information from the contracting path via skip connections to generate a segmentation map therefore,

decoding the encoded data.

The architecture overcomes the above challenges\

1. **Limited Training Data:** The model uses data augmentation and its effective architecture design helps in achieving high accuracy even with small annotated datasets.
2. **Precise Localization:** The U-Net uses skip connections which preserves the spatial information, making precise localization of structures within images.
3. **Speed Requirements:** U-Net's efficient encoder-decoder (U-shape) structure allows for fast and real-time segmentation.
4. **Versatility:** The model flexible design allows it to be applied to various biomedical segmentation tasks without significant mofification.
5. **Performance Improvement:** U-Net model outperforms using sliding-window convolutional network approach by segmenting the complete images at once with better accuracy.

Key breakthroughs of U-Net that address segmentation challenges

U-Net introduced several advancements that tackled key challenges in image segmentation, particularly in the biomedical field:

1. **Symmetric Encoder-Decoder Design**
   U-Net features a unique "U"-shaped architecture comprising a contracting (encoder) path and an expanding (decoder) path. The encoder extracts context through successive convolutions and downsampling, while the decoder reconstructs the spatial resolution. This balanced design enables the model to capture both global context and fine-grained details, essential for precise segmentation tasks.
2. **Skip Connections for Spatial Precision**
   A critical innovation in U-Net is the use of skip connections, which directly transfer feature maps from the encoder to the decoder. This integration helps retain spatial details that are often lost during downsampling, ensuring accurate boundary detection and structure localization in the segmentation output.
3. **Effective Use of Limited Training Data**
   The architecture is optimized to perform well even with small annotated datasets, a common limitation in biomedical imaging. Techniques such as data augmentation (e.g., rotation, flipping, and elastic deformation) are integrated to artificially increase the diversity of training data, improving generalization.
4. **Pixel-Level Predictions**
   U-Net excels at generating dense, pixel-wise predictions, making it ideal for

applications requiring detailed segmentation. The fully convolutional nature of the network ensures that every pixel in the input image contributes to the final output, enhancing segmentation accuracy.

5. **Real-Time Segmentation Capability**
   The efficient design of U-Net allows for faster computations compared to traditional sliding-window methods. This makes it suitable for real-time or near real-time applications, a crucial requirement in fields like medical diagnostics.

6. **Adaptability Across Applications**
   U-Net's flexible design supports a variety of segmentation tasks with minimal architectural modifications. Its robustness makes it applicable to tasks ranging from organ segmentation in medical imaging to object segmentation in satellite images.

7. **Improved Performance Over Traditional Methods**
   Unlike earlier approaches like sliding-window convolutional networks, which process small patches of an image, U-Net segments the entire image in one pass. This holistic approach improves both accuracy and computational efficiency.

These breakthroughs have made U-Net a cornerstone model for tasks like medical image segmentation, satellite image analysis, and more. Its design principles have influenced numerous segmentation architectures developed since its introduction.

Model Architecture

Double Convolution Layer

The Double Convolution layer in U-Net is a component that applies two convolutional layers back-to-back. Each layer uses filters to detect features in the image, like edges or textures.

The purpose of using two convolutions is to make the model better at capturing detailed information from the image. After each convolution, a ReLU activation is applied to introduce non-linearity, helping the network learn more complex patterns.

In simple terms, the DoubleConv layer helps the U-Net model understand both fine details and broader patterns in the image, making it effective for accurately identifying and segmenting objects.

The below code implementation does the following tasks:

1. **Two 3x3 Convolutions with Padding:** Both convolutions use a 3x3 kernel with padding of 1, preserving the spatial dimensions of the feature map i.e a 512x512 input results inputs in 512x512 outputs.

2. **First Convolution Layer:** Increases the number of channels (e.g., from 3 to 64), capturing initial features from the input.

3. **Second Convolution Layer:** Maintains the same number of channels (e.g., 64 to 64) and refines the feature map, enriching the details learned.

4. **ReLU Activation after Each Convolution:** Adds non-linearity, helping the model learn complex patterns.

5. **Forward Function:** It provides the forward data flow in the convolution block

Purpose: The two-layer setup refines feature extraction, enabling U-Net to capture detailed spatial information essential for accurate image segmentation.

In [ ]:
```
import torch
import torch.nn as nn

class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv_op = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1
            nn.ReLU(inplace=True),  # Activation function
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=
            nn.ReLU(inplace=True)  # Activation function
        )

    def forward(self, x):
        return self.conv_op(x)  # Forward pass through two convolution la
```

Contracting Path (Encoder):

The DownSample class represents a single block in the encoder. It has two main jobs

1. Extract important features from the image
2. Reduce the size of the image so the model can focus on more global details

The below code implementation does the following tasks:

1. **Double Convolution Layer (self.conv):** Applies two convolution operations, helping the model pick up details and patterns from the input image.
2. **Max Pooling Layer (self.pool):** This reduces the image size by half i.e from 512x512 to 256x256 , making it smaller and focusing on large patterns
3. **Forward Method (forward function):** This provides the flow of the data in the block, the DownSample block first extracts detailed features from the input image, creating a feature map (down). It then downsamples this map to a smaller size (p). Both outputs are returned: down is saved for skip connections to retain details, and p continues through the network.

> **Purpose:** The encoder's job is to capture image details while reducing the size, allowing the model to see both small and large patterns. The saved feature maps from each layer will later help the decoder part recreate the image accurately during segmentation.

In [ ]:
```
class DownSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = DoubleConv(in_channels, out_channels)  # Double convo
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)  # Max pooling

    def forward(self, x):
        down = self.conv(x)  # Apply two convolutions to extract features
        p = self.pool(down)  # Apply max pooling to reduce spatial dimens
        return down, p  # Return both the feature map and pooled output
```

Expansive Path (Decoder):

The UpSample class is responsible for upsampling the feature maps and combining them with the saved feature maps from the encoder.

The below code implementation does the following tasks:

1. **Upsampling:**The first layer, self.up, uses a transposed convolution (also called a deconvolution) to increase the size of the input feature map. For example, if the input is 64x64, this layer would upsample it to 128x128.

This step reverses the downsampling done in the encoder, allowing the model to reconstruct the image's spatial resolution progressively.

2. **Skip Connection:** After upsampling, the upsampled feature map (x1) is concatenated with the corresponding feature map (x2) from the encoder (contracting path). This is known as a skip connection.

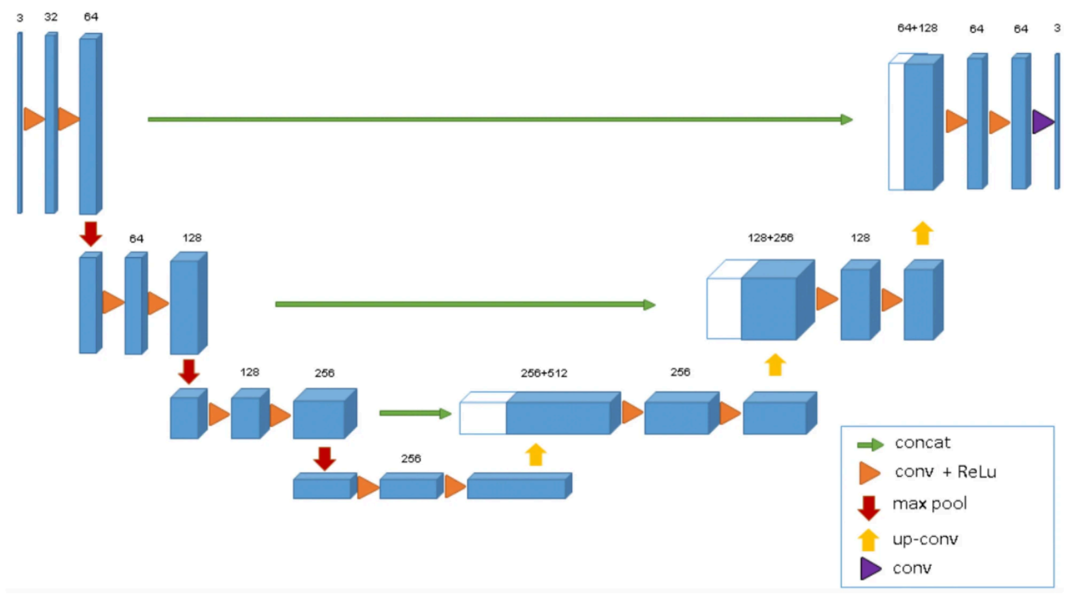
The skip connection allows the model to bring back important details from the encoder, which helps retain spatial and edge information lost during downsampling.

3. **Double Convolution:** The concatenated feature map (combining x1 and x2) is passed through two convolution layers (self.conv) to refine the features.

This step further processes the combined features, allowing the model to merge high-level details (from upsampling) with the finer details (from the encoder) effectively.

> **Purpose:** In the expansive path, each UpSample block helps reconstruct the original image size by upsampling and refining the features, while the skip connections ensure that important details from the encoder are preserved. This setup allows the U-Net to achieve precise segmentation with accurate boundary details.

In [ ]:
```
class UpSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels // 2, kerne
        self.conv = DoubleConv(in_channels, out_channels)  # Two convolut

    def forward(self, x1, x2):
        x1 = self.up(x1)  # Upsamples the input
        x = torch.cat([x1, x2], 1)  # Combines with encoder's feature map
        return self.conv(x)  # Refines combined features
```



Skip Connections: Role and function within the network

Skip connections link feature maps from the encoder (downsampling path) directly to the decoder (upsampling path). This allows the model to retain important details from the original image as it reconstructs the segmented output.

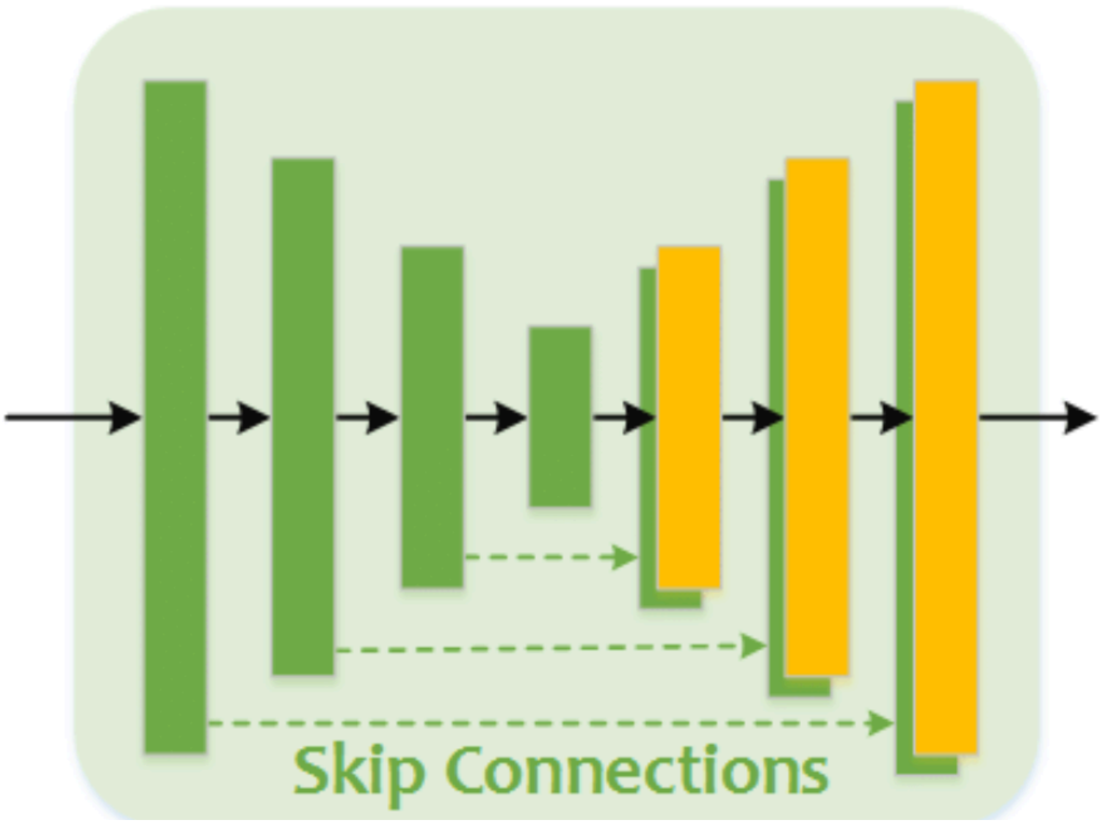
The below code implementation does the following tasks:

1. **Encoder Output Storage:** As the input image goes through each layer in the encoder, feature maps are created and stored (e.g., down_1, down_2, etc.).

These feature maps contain fine details and patterns that would otherwise be lost when the image is downsized 2. **Passing to the Decoder:** Each saved feature map from the encoder is passed to a corresponding layer in the decoder.

For example, down_4 is used in up_convolution_1, down_3 in up_convolution_2, and so on. 3. **Concatenation in UpSample:** Inside each UpSample layer, the saved feature map from the encoder is combined (concatenated) with the upsampled feature map from the decoder.

This combination gives the decoder both the fine details from the encoder and the broader patterns it learns as it upsamples, helping it produce a more accurate segmentation.



> **Purpose:** Skip connections help preserve spatial details and high-resolution features from the encoder, enabling the decoder to produce more accurate segmentation outputs. They effectively combine both low-level and high-level features, making U-Net particularly powerful for tasks like image segmentation.

Bottleneck:(The Bridge)

In this U-Net forward method, the bottleneck sits between the encoder (downsampling) and decoder (upsampling) parts. It processes the **most compressed** and **detailed features** from the encoder and prepares them for reconstruction in the decoder.

1. p4: This is the smallest, deepest feature map output from the encoder. After several downsampling steps, p4 has a small spatial size (height and width are reduced), but it has a high channel depth, capturing complex and abstract features from the input image.
2. self.bottle_neck: This is a DoubleConv layer (two convolutional layers with ReLU activations). It applies additional processing to p4 to refine these high-level features further.

Purpose:

- **Condensed Feature Representation:** The bottleneck layer applies two convolutions to extract the most important features, which are critical to understanding the entire image structure. These features contain both fine details and broader patterns that the encoder has accumulated.

- **Transition from Encoder to Decoder:** The bottleneck acts as a bridge between the encoder and decoder. While the encoder reduces spatial information and focuses on feature depth, the decoder will expand spatial dimensions again to reconstruct the image.

By refining the features at this stage, the bottleneck prepares a compact and meaningful representation for the decoder to use during upsampling and reconstruction.

- **Setting Up for Accurate Segmentation:** The bottleneck ensures that the decoder receives high-quality, condensed information, allowing it to accurately recreate the image and achieve precise segmentation.

Final Layer: Purpose and impact on segmentation output

Purpose of the Final Layer

1. **Reduces Channels:** The nn.Conv2d layer with kernel_size=1 reduces the channels from 64 to num_classes.If we're doing binary segmentation (e.g., foreground vs. background), num_classes would be 1.This step adjusts the model's output to match the number of segmentation classes needed.

2. **Classifies Each Pixel:** This 1x1 convolution layer operates on each pixel independently, assigning a probability (or confidence) for each class. The result is a segmentation map where each pixel is labeled according to the predicted class.

Impact on Segmentation Output

1. **Creates the Segmentation Map:** The out variable becomes the final segmentation output, with each pixel indicating its predicted class.

2. **Pixel-Level Accuracy:**By applying a 1x1 convolution, each pixel is classified independently, which allows U-Net to achieve high accuracy in segmenting fine details.

In the U-Net model, the final layer produces the segmentation map by reducing the channels to the number of classes. This step generates a detailed output where each pixel is assigned a class, enabling precise image segmentation.

In [ ]:
```
import torch
import torch.nn as nn

class UNet(nn.Module):
    def __init__(self, in_channels, num_classes):
        super().__init__()

        # Contracting path (encoder): progressively downsamples the input
        self.down_convolution_1 = DownSample(in_channels, 64)  # First do
        self.down_convolution_2 = DownSample(64, 128)         # Second do
        self.down_convolution_3 = DownSample(128, 256)        # Third dow
        self.down_convolution_4 = DownSample(256, 512)        # Fourth do

        # Bottleneck layer: connects the encoder and decoder, capturing h
        self.bottle_neck = DoubleConv(512, 1024)

        # Expansive path (decoder): progressively upsamples the feature m
        self.up_convolution_1 = UpSample(1024, 512)           # First ups
        self.up_convolution_2 = UpSample(512, 256)            # Second up
        self.up_convolution_3 = UpSample(256, 128)            # Third ups
        self.up_convolution_4 = UpSample(128, 64)             # Fourth up

        # Final output layer: 1x1 convolution to reduce channels to num_c
        self.out = nn.Conv2d(in_channels=64, out_channels=num_classes, ke
```

> In the **contracting path**, the input is progressively downsampled, with intermediate feature maps saved for skip connections.

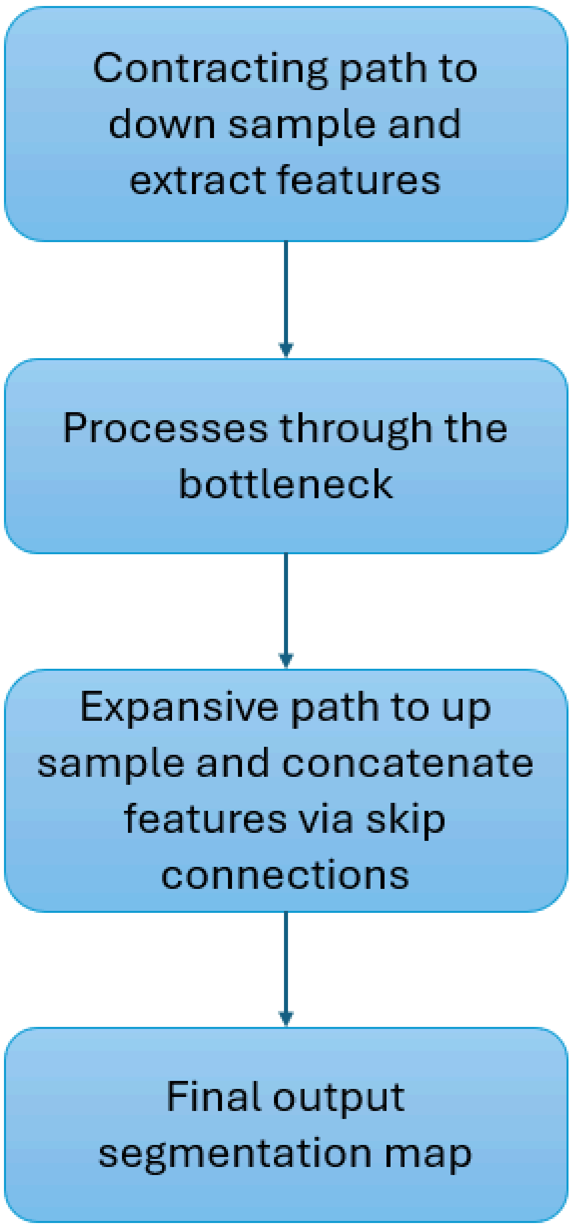
> The **bottleneck layer** processes the most compressed representation of the data.

> During the **expansive path**, the feature maps are progressively upsampled and combined with the corresponding saved feature maps from the contracting path using skip connections.

In [ ]:
```
def forward(self, x):
        # Contracting path (encoder)
        down_1, p1 = self.down_convolution_1(x)    # down_1 is saved for
        down_2, p2 = self.down_convolution_2(p1)   # down_2 is saved for
        down_3, p3 = self.down_convolution_3(p2)   # down_3 is saved for
        down_4, p4 = self.down_convolution_4(p3)   # down_4 is saved for

        # Bottleneck
        b = self.bottle_neck(p4)

        # Expansive path (decoder) with skip connections
        up_1 = self.up_convolution_1(b, down_4)  # Skip connection with d
        up_2 = self.up_convolution_2(up_1, down_3)  # Skip connection wit
        up_3 = self.up_convolution_3(up_2, down_2)  # Skip connection wit
        up_4 = self.up_convolution_4(up_3, down_1)  # Skip connection wit

        # Final output
        out = self.out(up_4)
        return out
```

Synthesis of Architecture: How All Parts Come Together

The code defined above shows the full architecture in the UNet class's forward method, which organizes the flow through contracting, bottleneck, and expansive paths.


Contracting path to down sample and extract features
Processes through the bottleneck
Expansive path to up sample and concatenate features via skip connections
Final output segmentation map

Model Training

Data Loading and Preprocessing for Model Training

The CustomCarvanaDataset class is responsible for loading the image and mask pairs from the dataset folders, resizing them to a uniform size, converting them to PyTorch tensors, and making them accessible for training or testing the model. This ensures the data is prepared and standardized for use in a deep learning pipeline.

The below code is trying to achieve the following tasks with the dataset:

1. **Initialization (__init__)**: Loads file paths for images and masks from the specified dataset directory (base_dir).

Differentiates between training and test data using the is_test flag.

Defines a set of transformations to resize the images and masks to 512x512

Converts them into PyTorch tensors.

2. **Get Item (__getitem__):** Fetches the image and corresponding mask at the given index (idx).

Coverts The image to RGB (3-channel format) and mask to grayscale (L mode) (1-channel format).

Applies transformations to standardize the image and mask, resizes them to 512x512 & converts them to PyTorch tensors.

In [ ]:
```
import os
from PIL import Image
from torch.utils.data import Dataset
from torchvision.transforms import Compose, Resize, ToTensor

class CustomCarvanaDataset(Dataset):
    def __init__(self, base_dir, is_test=False):
        # Initialize the dataset with paths for images and masks
        self.base_dir = base_dir

        if is_test:
            # Load test image and mask paths
            self.image_paths = sorted([os.path.join(base_dir, "manual_tes
            self.mask_paths = sorted([os.path.join(base_dir, "manual_test
        else:
            # Load training image and mask paths
            self.image_paths = sorted([os.path.join(base_dir, "train", fn
            self.mask_paths = sorted([os.path.join(base_dir, "train_masks

        # Define transformations: resize to 512x512 and convert to tensor
        self.transforms = Compose([
            Resize((512, 512)),  # Resize all images/masks to 512x512
            ToTensor()           # Convert images/masks to PyTorch tensor
        ])

    def __getitem__(self, idx):
        # Load image and mask at the specified index
        image = Image.open(self.image_paths[idx]).convert("RGB")  # Conve
        mask = Image.open(self.mask_paths[idx]).convert("L")      # Conve

        # Apply transformations and return as tensors
        return self.transforms(image), self.transforms(mask)

    def __len__(self):
        # Return total number of image-mask pairs
        return len(self.image_paths)
```

Training Loop and Sub-Parts

Training Loop: The training loop runs for a specified number of epochs, iterating over the training dataset to update the model weights.

```
for epoch in tqdm(range(EPOCHS)):
    model.train()  # Sets the model to training mode
    train_running_loss = 0
    for idx, img_mask in
enumerate(tqdm(train_dataloader)):
        img = img_mask[0].float().to(device)  # Load and
prepare input image
        mask = img_mask[1].float().to(device)  # Load and
prepare corresponding mask

        y_pred = model(img)  # Forward pass through the
model
        optimizer.zero_grad()  # Clear previous gradients
        loss = criterion(y_pred, mask)  # Compute the loss
        train_running_loss += loss.item()
        loss.backward()  # Backward pass to compute
gradients
        optimizer.step()  # Update model parameters
```

Validation Loop: Validates the model on the validation dataset without updating weights.

```
model.eval()  # Sets the model to evaluation mode
with torch.no_grad():  # No gradient computation for
validation
    for idx, img_mask in enumerate(tqdm(val_dataloader)):
        img = img_mask[0].float().to(device)
        mask = img_mask[1].float().to(device)
        y_pred = model(img)  # Forward pass
        loss = criterion(y_pred, mask)  # Compute
validation loss
```

Optimizer: AdamW optimizer for training the model, using the model's parameters and a specified learning rate. AdamW is a variant of Adam that includes weight decay for better regularization. This updates model weights to minimize the loss.

```
optimizer = optim.AdamW(model.parameters(),
lr=LEARNING_RATE)
```

Loss Function: Calculates the error between predicted and true masks for binary segmentation tasks.

```
criterion = nn.BCEWithLogitsLoss()
```

The combined code for training segment,validation , optimizer & Loss function can be found below:

Hyperparameter Tuning for GPU config

In [ ]:
```
# important Libraies imported
import torch
from torch import optim, nn
from torch.utils.data import DataLoader, random_split
from tqdm import tqdm
import matplotlib.pyplot as plt
```

> This function visualizes the training and validation losses over the course of the training process. It plots the losses for each epoch using different markers to distinguish training and validation trends.

In [ ]:
```
# Function to plot training and validation losses over epochs
def plot_loss(train_losses, val_losses, epochs):
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, epochs + 1), train_losses, label="Training Loss", m
    plt.plot(range(1, epochs + 1), val_losses, label="Validation Loss", m
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Training and Validation Loss Over Epochs")  #
    plt.legend()
    plt.grid()
    plt.show()
```

> **Learning Rate:** A learning rate of 3e-4 was chosen for a stable balance between convergence speed and loss minimization

> **Batch Size:** Batch size of 32 was used to accommodate memory constraints on the available GPU.

> **Epochs:** 6 epochs were selected based on early stopping criteria observed during experimentation

In [ ]:
```
if __name__ == "__main__":
    # Hyperparameters
    LEARNING_RATE = 3e-4  # Learning rate for the optimizer
    BATCH_SIZE = 32  # Batch size for training and validation
    EPOCHS = 6  # Number of epochs for training
    DATA_PATH = "/content"  # Path to the dataset
    MODEL_SAVE_PATH = "/content/drive/MyDrive/Deep_Learning_Project/model

    # Device configuration (GPU if available, else CPU)
    device = "cuda" if torch.cuda.is_available() else "cpu"

    # Load dataset and split into training and validation sets
    train_dataset = CustomCarvanaDataset(DATA_PATH)  # Load custom datase
    generator = torch.Generator().manual_seed(42)  # Set random seed for
    train_dataset, val_dataset = random_split(train_dataset, [0.8, 0.2],

    # Create DataLoader for training and validation
    train_dataloader = DataLoader(dataset=train_dataset, batch_size=BATCH
    val_dataloader = DataLoader(dataset=val_dataset, batch_size=BATCH_SIZ

    # Initialize the model, optimizer, and loss function
    model = UNet(in_channels=3, num_classes=1).to(device)  # UNet model f
    optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)  # Adam
    criterion = nn.BCEWithLogitsLoss()  # Binary cross-entropy loss with

    # Lists to store losses for visualization
    train_losses = []
    val_losses = []
```

> This code iterates over the dataset in mini-batches, performs a forward pass to compute predictions, and calculates the loss between predictions and ground truth masks.

> Gradients are computed using backpropagation, and the optimizer updates the model's parameters to minimize the loss. The average loss for the epoch is calculated and stored for monitoring training progress.

In [ ]:
```
# Training and validation loop
    for epoch in tqdm(range(EPOCHS)):
        # Training phase
        model.train()  # Set model to training mode
        train_running_loss = 0  # Initialize running loss for training
        for idx, img_mask in enumerate(tqdm(train_dataloader, desc=f"Trai
            img = img_mask[0].float().to(device)  # Load image batch to d
            mask = img_mask[1].float().to(device)  # Load mask batch to d

            y_pred = model(img)  # Forward pass
            optimizer.zero_grad()  # Zero the gradients
            loss = criterion(y_pred, mask)  # Compute loss
            train_running_loss += loss.item()  # Accumulate running loss

            loss.backward()  # Backpropagation
            optimizer.step()  # Update model parameters

        train_loss = train_running_loss / (idx + 1)  # Compute average tr
        train_losses.append(train_loss)  # Append to training loss list
```

> This code set to evaluation mode, and gradient computation is disabled to save memory and speed up inference.

> The validation loop computes the loss for each batch without updating the model, accumulating it to calculate the average validation loss for the epoch.

> The training and validation losses are printed for progress tracking, and the trained model is saved, followed by a plot of the loss trends over epochs for analysis.

In [ ]:
```
# Validation phase
        model.eval()  # Set model to evaluation mode
        val_running_loss = 0  # Initialize running loss for validation
        with torch.no_grad():  # Disable gradient computation for validat
            for idx, img_mask in enumerate(tqdm(val_dataloader, desc=f"Va
                img = img_mask[0].float().to(device)  # Load image batch
                mask = img_mask[1].float().to(device)  # Load mask batch

                y_pred = model(img)  # Forward pass
                loss = criterion(y_pred, mask)  # Compute loss
                val_running_loss += loss.item()  # Accumulate running los

            val_loss = val_running_loss / (idx + 1)  # Compute average va
            val_losses.append(val_loss)  # Append to validation loss list

        # Print losses for the current epoch
        print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Validatio

    # Save the trained model to the specified path
    torch.save(model.state_dict(), MODEL_SAVE_PATH)

    # Plot training and validation losses
    plot_loss(train_losses, val_losses, EPOCHS)
```

Model Evaluation

Minimal Working Example (MWE-CPU)

1. **Dataset Used**: The Carvana Image Masking Dataset is a popular dataset designed for image segmentation tasks, particularly car segmentation. It consists of high-resolution car images (1918x1280 pixels) and corresponding binary masks that identify the car regions in the images. The dataset was originally used in a Kaggle competition to predict accurate masks for cars under varying lighting conditions and angles.

2. **Tuning made for MWE-CPU**: Since this is a gpu intesnive task , preparing for a CPU version is challenging. The following adjustments we made to the whole process to accomodate a CPU centric training model.

   - Subset the Dataset: Used 10% of the dataset to take a smaller sample of the training dataset and validation dataset to reduce computational load.
   - Reduced Batch Size: Reduced BATCH_SIZE to 4 for lighter memory usage.
   - Fewer Epochs: Reduced the number of EPOCHS to 2 for testing purposes.

Evaluation with adjusted U-Net Model

The provided code is designed to perform semantic segmentation on a single input image using a trained U-Net model. It preprocesses the image, loads the model weights, performs inference to generate a predicted segmentation mask, and visualizes both the original image and the predicted mask side by side.

Below is the specific tasks perfomed by the code:

1. **Model Loading:** Loads the pretrained U-Net model and its weights from a specified file path.

Moves the model to the appropriate device (CPU or GPU) for computation. 2. **Image Preprocessing:**Resizes the input image to 512x512 pixels.

Converts the image into a tensor format compatible with PyTorch models 3. **Inference:** Passes the preprocessed image through the U-Net model to predict the segmentation mask.

4. **Post-Processing:** Processes the input image and predicted mask for visualization, including normalizing the predicted mask to a binary format (0 and 1).

5. **Visualization:** Displays the original image and the predicted segmentation mask side by side using Matplotlib.

In [ ]:
```
# important Libraies imported
import torch
import matplotlib.pyplot as plt
from torchvision import transforms
from PIL import Image
```

> This function performs inference using a trained U-Net model on a single image and visualizes the input image alongside the predicted segmentation mask. It loads the model and applies preprocessing, including resizing and converting the image into a tensor. The function performs inference to generate the predicted mask, post-processes both the input and mask for visualization, and displays them side by side using matplotlib for easy comparison and analysis.

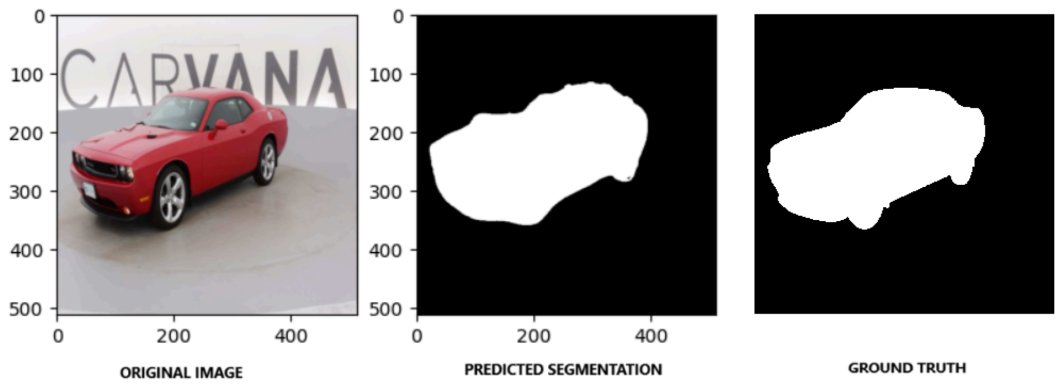
In [ ]:
```
# Function for performing inference on a single image and visualizing the
def single_image_inference(image_pth, model_pth, device):
    # Load the trained U-Net model and its weights
    model = UNet(in_channels=3, num_classes=1).to(device)
    model.load_state_dict(torch.load(model_pth, map_location=torch.device

    # Define transformations to preprocess the input image
    transform = transforms.Compose([
        transforms.Resize((512, 512)),  # Resize the image to 512x512
        transforms.ToTensor()  # Convert the image to a PyTorch tensor
    ])

    # Open the image, apply transformations, and move it to the specified
    img = transform(Image.open(image_pth)).float().to(device)
    img = img.unsqueeze(0)  # Add batch dimension (1, C, H, W) for model

    # Perform inference to get the predicted mask
    pred_mask = model(img)

    # Post-process the input image for visualization
    img = img.squeeze(0).cpu().detach()  # Remove batch dimension and mov
    img = img.permute(1, 2, 0)  # Rearrange tensor dimensions to (H, W, C

    # Post-process the predicted mask for visualization
    pred_mask = pred_mask.squeeze(0).cpu().detach()  # Remove batch dimen
    pred_mask = pred_mask.permute(1, 2, 0)  # Rearrange tensor dimensions
    pred_mask[pred_mask < 0] = 0  # Set negative values to 0
    pred_mask[pred_mask > 0] = 1  # Set positive values to 1 (binary mask

    # Create a figure to display the input image and the predicted mask s
    fig = plt.figure()
    for i in range(1, 3):  # Loop to create two subplots
        fig.add_subplot(1, 2, i)  # Add a subplot (1 row, 2 columns)
        if i == 1:
            plt.imshow(img, cmap="gray")  # Display the input image in th
        else:
            plt.imshow(pred_mask, cmap="gray")  # Display the predicted m
    plt.show()  # Show the figure
```

> This sets up and executes the inference process for a single image using a trained U-Net model. Finally, it calls the `single_image_inference` function to process the image, generate a segmentation mask, and visualize the results.

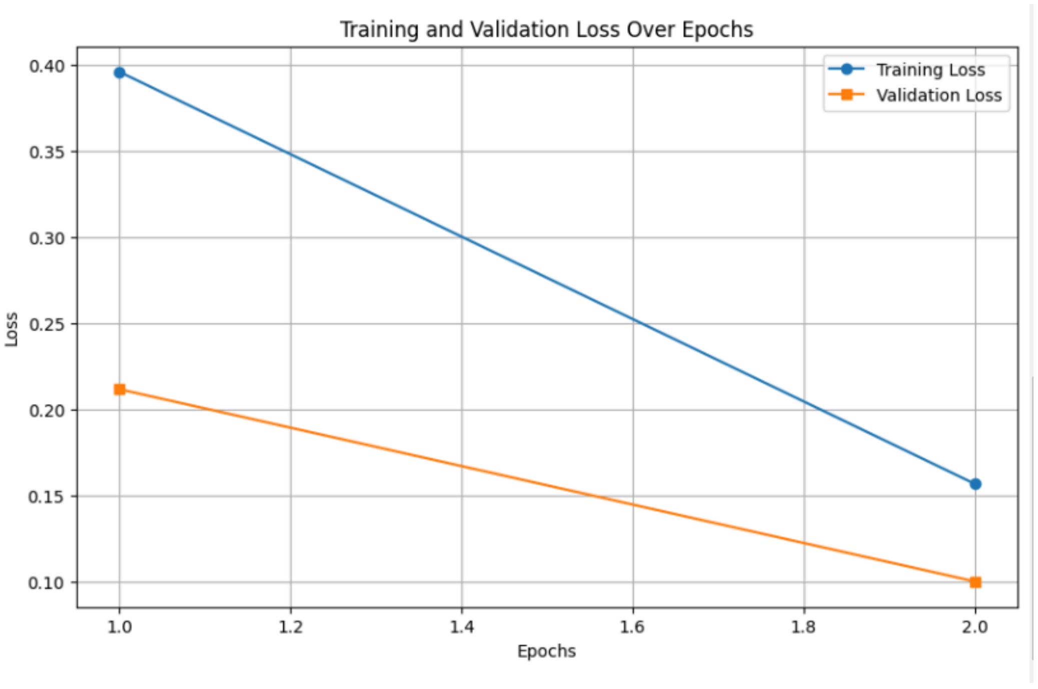
In [ ]:
```
if __name__ == "__main__":
    # Define file paths for the input image and model weights
    SINGLE_IMG_PATH = "/content/drive/MyDrive/Deep_Learning_Project/sampl
    DATA_PATH = "./data"
    MODEL_PATH = "/content/drive/MyDrive/Deep_Learning_Project/models_tra

    # Determine the device for computation (GPU if available, else CPU)
    device = "cuda" if torch.cuda.is_available() else "cpu"

    # Perform inference on the single image and visualize results
    single_image_inference(SINGLE_IMG_PATH, MODEL_PATH, device)
```

Experiments and Results

Output from the adjusted minimal model using CPU under partial load

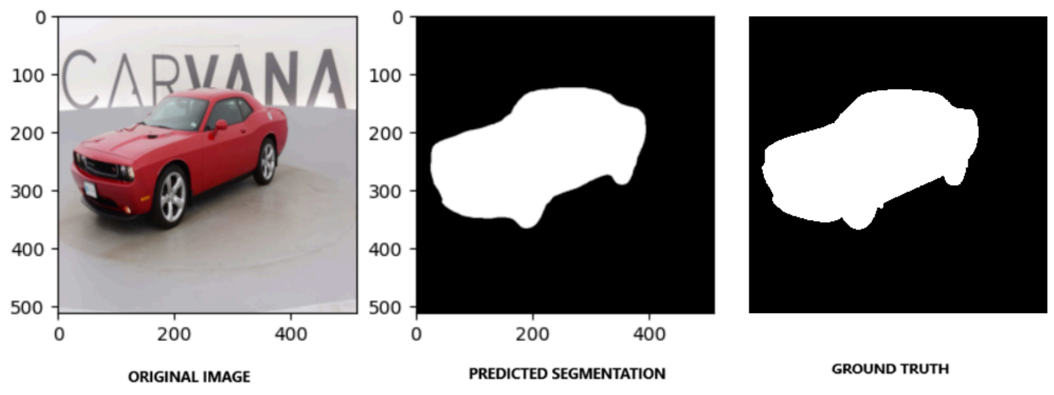


Performance metric visualisaton for the adjusted minimal model using CPU under partial load

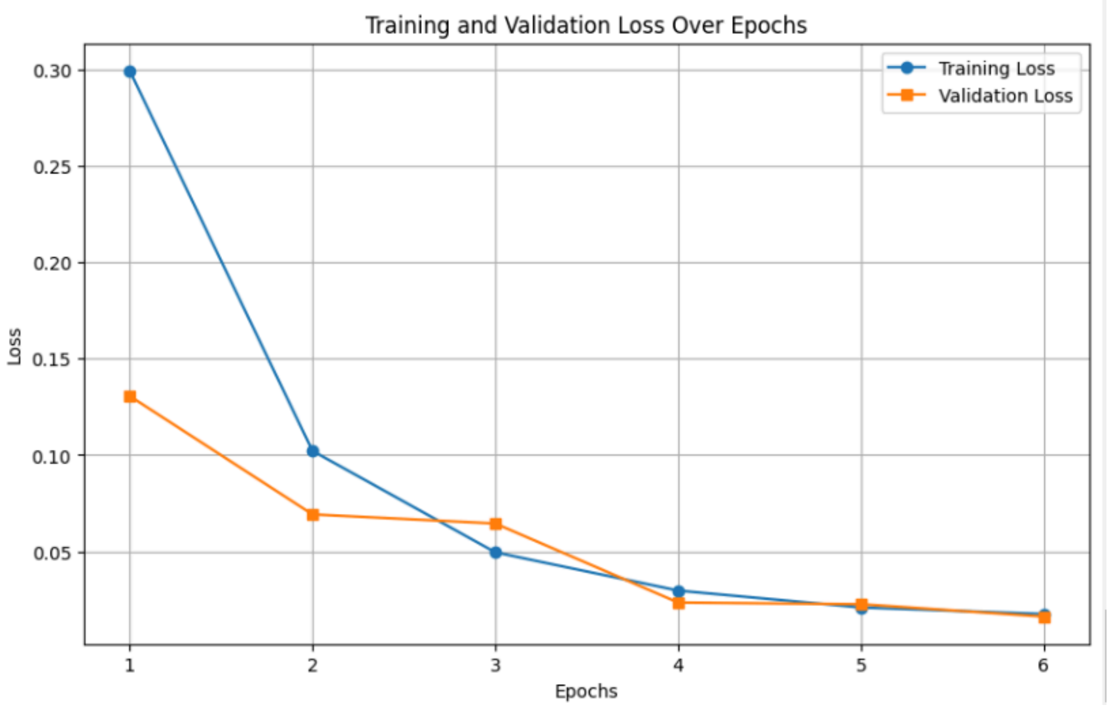


> The training loss(blue line), decreases steadily, indicating the model's improved performance on the training dataset. Similarly, the validation loss(orange line), also decreases, suggesting that the model is generalizing well to unseen data. The consistent downward trend reflects effective training under the given conditions.

Output from the original model using GPU under full load



Performance metric visualisaton for the adjusted minimal model using GPU under full load



> The training loss (blue line) and validation loss (orange line) consistently decrease, demonstrating effective learning and model generalization. By the final epochs, the losses converge, suggesting that the model is stable and performing well without overfitting.

Observed Limitations of CPU Training

1. **Slower Training Speed**: CPUs process computations sequentially, leading to significantly longer training times compared to GPUs. This can make it impractical to experiment with large models or multiple configurations within a reasonable timeframe.

2. **High Memory Usage**: Training deep learning models on a CPU often consumes more RAM due to its inefficiency in parallel processing. This can lead to system memory bottlenecks, especially when working with larger datasets or complex architectures.

3. **Limited Scalability**: CPUs struggle to handle large datasets or complex models, limiting their use in scalable training workflows. They are not ideal for distributed training across multiple devices, making it harder to scale for industrial workloads.

4. **Suboptimal Model Performance**: Extended training durations and slower optimization can impact the overall performance of the model. The slower iteration cycles can also delay hyperparameter tuning and model improvements.

5. **Increased Inference Time**: CPUs require more time to make predictions, making real-time applications impractical. This limitation is especially critical for

applications like autonomous vehicles or medical imaging, where speed is crucial.

6. **Difficulty in Handling High-Resolution Images**: High-resolution image processing is resource-intensive, making it challenging for CPUs to manage efficiently. This can lead to prolonged computation times or even system crashes when processing very large images.

Limitations and Weaknesses of U-Net

1. **Memory and Computational Requirements**
U-Net's encoder-decoder design with multiple layers and skip connections can demand significant computational resources, particularly for high-resolution images. This makes it less accessible for systems with limited hardware capabilities.

2. **Overfitting on Small Datasets**
While U-Net is designed to perform well with limited data, it can still overfit on very small datasets, especially when the data lacks sufficient variability. This requires careful tuning of regularization techniques and augmentation strategies.

3. **Lack of Multi-Scale Feature Integration**
U-Net processes features at different scales, but its direct skip connections may not fully integrate multi-scale contextual information. This can lead to suboptimal performance on tasks requiring a more nuanced understanding of hierarchical features.

4. **Sensitivity to Input Size**
U-Net's fully convolutional nature allows for input of arbitrary sizes, but mismatched input dimensions can still cause issues with padding and resizing, potentially leading to artifacts in the segmentation map.

5. **Limited Generalization to Non-Biomedical Domains**
While highly effective in biomedical imaging, U-Net's architecture may require substantial adaptations for non-biomedical tasks, such as those involving complex textures or irregular shapes.

6. **Imbalanced Class Segmentation**
U-Net can struggle with datasets where one class dominates (e.g., background), leading to poor segmentation of smaller or less represented classes.

Future Directions and Potential Improvements

1. **Incorporating Attention Mechanisms**
Adding attention layers to U-Net could improve its ability to focus on relevant regions, enhancing segmentation accuracy, especially in cases with complex or noisy backgrounds.

2. **Efficient Architectures for Resource-Constrained Environments**
   Developing lightweight variants of U-Net, such as pruning or compressing layers, can make the model more suitable for devices with limited computational power, such as mobile or edge devices.

3. **Multi-Scale Feature Fusion**
   Introducing advanced multi-scale feature integration methods, such as pyramid pooling or dilated convolutions, could allow U-Net to better capture global and local features simultaneously.

4. **Handling Imbalanced Datasets**
   Improved loss functions, such as focal loss or weighted Dice loss, can address class imbalance by emphasizing underrepresented classes during training.

5. **Semi-Supervised and Unsupervised Learning**
   Leveraging semi-supervised or unsupervised learning approaches can reduce dependence on large labeled datasets, making U-Net more applicable in scenarios where annotations are scarce.

6. **Domain Adaptation Techniques**
   Enhancing U-Net with domain adaptation capabilities can improve its generalization to tasks outside biomedical imaging, such as autonomous driving or agricultural monitoring.

7. **3D and Temporal Extensions**
   Extending U-Net to handle 3D data or temporal sequences could open new opportunities in applications like video segmentation, volumetric imaging, and motion tracking.

8. **Integration of Transformers**
   Incorporating transformer-based components could enhance U-Net's ability to model long-range dependencies, providing a complementary perspective to its localized convolutional operations.

By addressing these limitations and exploring innovative extensions, U-Net can continue to evolve, maintaining its relevance and effectiveness across a broader range of segmentation tasks.