NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DIOGO FILIPE BARATA SOUSA

BSc in Computer Science and Engineering

VIDEO MATTING FOR LIVE SPORTS BROADCASTING

ARTIFICIAL INTELLIGENCE FOR REAL-TIME INFERENCE

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
January, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

VIDEO MATTING FOR LIVE SPORTS BROADCASTING

ARTIFICIAL INTELLIGENCE FOR REAL-TIME INFERENCE

DIOGO FILIPE BARATA SOUSA

BSc in Computer Science and Engineering

Adviser: Guilherme Franco
AI Developer, wTVision

Co-adviser: Sofia Cavaco
Assistant Professor, NOVA School of Science and Technology

Examination Committee

Chair: Ricardo Gonçalves
Assistant Professor, NOVA School of Science and Technology

Rapporteur: David Semedo
Assistant Professor, NOVA School of Science and Technology

Adviser: Guilherme Franco
AI Developer, wTVision

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
January, 2024

Video Matting for Live Sports Broadcasting
Artificial Intelligence for real-time inference

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.0.3) [23].

Abstract

Nowadays, many applications in the television and movie industry make foreground detection against a well-established background with a distinguishable solid color. By making this distinction, it is possible in a post-process phase, to delete the background while keeping the foreground. This usage of a solid and distinguishable color for the background works to a certain extent. For real and complex images or videos where no distinguishable background can be selected, this approach is not suitable. This problem is referenced as **alpha matting**.

To solve more complex cases of alpha matting, a novel approach is needed. Many state-of-the-art solutions make use of deep learning architectures that are specialized in detecting foregrounds in complex scenarios.

For the architecture specialization in sports, a new synthetic dataset was created. This dataset, composed of multiple sports videos is used to train and validate alpha matting networks.

A deep learning architecture was studied and trained for the sports context. A new study on a custom hybrid model was proposed to compete with state-of-the-art architectures already available.

To use the capabilities of neural networks during a sports broadcast in the professional market, an integration was developed that aims to achieve the highest frame rate possible.

Resumo

Atualmente, diversas aplicações da indústria televisiva e cinematográfica usam a deteção do primeiro plano de uma imagem relativamente ao seu plano de fundo de cor sólida e distinguível. Desta forma é possível alterar apenas o plano de fundo mantendo o primeiro plano.

A utilização de um plano de fundo distinguível funciona em certas circunstâncias, contudo para imagens ou vídeos num contexto real pode não haver essa possibilidade. O problema de separar o plano de fundo do primeiro plano designa-se por **alpha matting**.

Para resolver os casos mais complexos de alpha matting é necessário aplicar novas técnicas. As soluções atuais baseiam-se em arquiteturas de inteligência artificial especializadas em detetar primeiros planos.

Para a especialização destas arquiteturas no contexto de desporto, um dataset composto por múltiplos desportos foi criado para realizar treinos e validações destas redes.

Uma arquitetura de aprendizagem profunda foi estudada e especializada no caso particular do desporto. Um novo estudo sobre uma nova rede híbrida foi proposto para competir com as atuais soluções disponíveis.

Para tirar partido destas redes durante transmissões desportivas no mercado profissional, uma implementação foi realizada de raiz para atingir o melhor desempenho possível com o intuito de integrar um produto final num motor de renderização.

Contents

List of Figures

List of Tables

Listings

Acronyms

AI Artificial Inteligence *(pp. 3, 51, 80, 84)*
API Application Programming Interface *(pp. 3, 42, 47, 56, 60, 64, 66, 68, 69, 71, 74–76)*
ASPP Atrous Spatial Pyramid Pooling *(pp. 14, 18, 56)*

CNN Convolutional Neural Network *(pp. viii, xi, 2, 3, 7, 9, 21, 23, 25, 30–32, 34–36, 47, 50, 58–60, 66, 80, 81)*
COM Component Object Model *(p. 74)*
ConvGRU Convolutional Gated Recurrent Units *(pp. 18, 19, 60)*
CPU Central Processing Unit *(pp. xi, 1, 47, 51, 56, 66–68, 73, 85)*
CSPP Center-Surround Pyramid Pooling *(p. 14)*

DLL Dynamic-Link Library *(pp. 74, 75)*
DLSS Deep Learning Super Sampling *(p. 84)*

FCN Fully Convolutional Network *(p. 19)*
FHD Full High Definition *(pp. 3, 19, 38, 45, 57, 59, 78, 84)*
FPS Frames Per Second *(pp. 45, 49, 67, 78, 80, 84)*

GCL Gated Convolutional Layers *(p. 17)*
GPU Graphics Processing Unit *(pp. xi, xii, 2, 3, 31, 47, 49, 51, 62, 64–69, 71–74, 76, 78, 84, 85)*

HRDB High-Resolution Detail Branch *(pp. 15, 17)*

JIT Just-In-Time *(pp. 65, 66)*

LFPNet Long-Range Feature Propagation Network *(pp. viii, 13)*
LSTM Long-Short Term Memory *(p. 60)*

MHSA Multi-Head Self-Attention *(pp. viii, 20, 22–25, 49, 50)*
MSE Mean Squared Error *(p. 52)*

NLP Natural Language Processing *(p. 20)*

OOM Out-Of-Memory *(pp. 31, 55)*

PAST Prior-Attentive Swin Transformer *(pp. viii, 25, 26)*
PNG Portable Network Graphics *(p. 37)*
PPM Pyramid Pooling Module *(pp. 15–17)*

RAM Random Access Memory *(pp. 47, 66, 67)*
ReLU Rectified Linear Unit *(pp. 10, 17, 19, 23)*
RGB Red, Green and Blue *(pp. 1, 7, 12, 39, 50, 64, 69)*
RNN Recurrent Neural Network *(p. 3)*
RVM Robust Video Matting *(pp. viii–xi, 17–19, 47–51, 56–62, 64, 66, 71, 73, 74, 76–78, 80, 81, 85, 86)*

SAD Sum of Absolute Differences *(pp. xi, 46)*
SCB Semantic Context Branch *(pp. 15, 17)*
SST NOVA School of Science and Technology *(pp. ix, 71)*

UHD Ultra High Definition *(pp. 3, 19, 29, 31, 43, 45, 57, 59, 78, 82, 84)*
UI User Interface *(pp. ix, 75, 76)*

ViT Vision Transformer *(pp. 21, 22, 34)*

WPF Windows Presentation Foundation *(p. 75)*

1 Introduction

1.1 Context

In the film and streaming industry it is common to filter images as to change some specific information of the recorded footage (for example changing the background). The typical approach used by these industries is to mask the undesired features with a green/blue material, so it can be post-processed and replaced.

The processing of an image and determination of what is the background and the foreground is called **alpha matting**. By definition, alpha matting assumes that each image is a linear combination between a foreground and a background layer defined by an alpha value.

The following formula defines the standard formula for the alpha matting problem:

$$C_i = \alpha_i \mathcal{F}_i + (1 - \alpha_i)\mathcal{B}_i \tag{1.1}$$

In this formula the C_i, F_i and B_i are: the color of pixel i in the source image, the foreground layer and the background layer respectively. α is a value between 0 and 1 that defines the blending between the foreground and background layer. So, the alpha matting problem is based on solving all 7 unknown variables: 3 for the Red, Green and Blue (RGB) values of each F_i and B_i and the alpha value α.

Nowadays, many algorithms exist to accurately filter green/blue screen from videos. However, other solutions are needed in situations where it is not possible to filter the background using a solid color filter.

Many state-of-the-art solutions take advantage of neural networks to read an image and predict the relation between the background and foreground. Using neural networks as the main solver for alpha matting brings some problems, two of them are performance and accuracy. Neural networks have a high complexity, which means that they cannot process images at a high frame rate with the typical CPU hardware. Also, for the network models to achieve accurate alpha mattes, it is typically required a deep architecture which may not be possible to implement in real-time due to hardware limitations in terms of memory and speed.

1.2 Motivation

As previously referred, one of the many uses of the alpha matting procedure is live-streaming, mainly used by streamers in different online platforms using a simple green screen. But alpha matting can be applied in other circumstances not commonly seen, for example in the broadcasting of sports.

Similar to the live-streaming industry, professional live-broadcast also has a similar use case for the application of alpha matting. During live-broadcast of sports, images captured by a camera go through a rendering engine to draw a template accordingly (the scores of each team alongside sponsorships are an example). From these templates it is pretty common to avoid drawing the template over a journalist or the players during a match. The alpha matte comes to solve this problem. If the designers have an alpha matting solution that works in any type of outdoor sport it can generate different layers within a video and different effects can be added to the post-process phase in the rendering pipeline.

Alpha matting during broadcast can be applied not only to sports but also in other contexts, like podcasts and interviews, so the solution should be as robust as possible.

At the time of writing, no alpha matting solution available that applies to sports was found during the research for this project. The datasets publicly available do not include human figures in a sports context, alongside their respective equipment. The main purpose of this project is to provide a solution to the lack of alpha matte datasets for sports (players and equipment). Since there is no available network tuned for sports, a network was selected and trained with this created dataset.

To create a new standard in the alpha matting research, a new custom architecture was also developed using state-of-the-art technologies to create the best results with the lowest inference speeds to satisfy the high performance requirements.

To achieve the high performances required for the broadcasting industry a new optimized integration was proposed and developed from scratch for this project. This implementation leverages the capabilities of a fully GPU accelerated pipeline to make the generation of alpha mattes as fast as possible. It also has the capacity of integrating any type of neural network in a rendering pipeline.

1.3 Objectives and Contributions

For this project, the line of work followed this structure:

- **Background and Related Work**: define what are the current matting algorithms that attempt to solve the alpha matting problem. Not only neural networks but also more classical methods, alongside their advantages and disadvantages. Various neural network architectures are referred and explained. This search also makes a comparison between the transformer architectures and the CNN approach;

- **Sports Oriented Dataset**: since this application of alpha matting is targeted at multiple sports, a dataset needs to be generated to fine-tune or train the network that will be used in the final product. This dataset aims at being the first dataset to implement alpha matting specifically directed at sports;
- **Architecture Fine-Tuning**: an architecture is selected as a component for the final product to integrate in the broadcasting pipeline. The selected model is fine-tuned with the generated dataset;
- **The Search for Transformers**: a new custom implementation is studied and designed to achieve new state-of-the-art results in alpha matting while leveraging the power of the transformers together with CNNs and Recurrent Neural Network (RNN)s.
- **Rendering Integration**: the newly fine-tuned model is accelerated by the GPU to reduce the latency of the operations and integrated in the broadcast engine. This implementation should guarantee at least Full High Definition (FHD) with 60 images per second;

With the implementation presented in this document it will be possible to not only use a neural network architecture in real-time to generate an alpha matte, but also to make the most use of the GPU computing capabilities to optimize the integration of the network in a rendering pipeline. *DirectX-11* and the *CUDA* Application Programming Interface (API) are used to process incoming frames at a lower latency using the processing capabilities of *NVIDIA* GPUs.

A general model framework was created to integrate any type of neural network. This framework was wrapped using *C++* language with integration with a *C#* environment.

The research made during this project not only contributed to a high performance implementation but also provided a path to integrate other types of neural networks in the broadcasting industry. The use of transformers in real-time was also a case study.

In the end, it should be possible for designers to make use of Artificial Inteligence (AI), select which elements belong to the foreground and background layers and blend these elements with the alphas generated by the network. All while satisfying the performance requirements defined by the broadcasting industry of 60 images per second with multiple resolutions, from FHD to Ultra High Definition (UHD) (1920 per 1080 pixels and 3840 per 2160 pixels respectively).

2

Background and Related Work

The different alpha matting techniques are exposed in this chapter. This chapter's objective is to answer how existent alpha matting solutions approach the problem and how different deep learning architectures are structured to achieve state-of-the-art results.

The neural networks presented use common concepts of deep learning, to clarify this, the most used basic concepts are also explained in this chapter.

2.1 Matting Techniques

2.1.1 Sampling Methods

These methods take as input an image and a trimap of that image. A trimap is an image with the same size as the original one, but with only 3 colors: black, white, and gray. The black pixels are the pixels known to belong to the background, the white pixels belong to the foreground, and the gray pixels are named the unknown pixels. The unknown pixels are the ones that are not known if they belong to the foreground or background. These trimaps are used in many matting techniques, but typically they require to be manually created.

Sampling methods assume that some pixels from the foreground and the background are known, so they rely on sampling those pixels to find relations between them and estimate the best foreground/background combination in the image. One of the differences between the existent sampling methods is how they sample and find relations between pixels. For example, some metrics for sampling can be defined by following the nearest pixel, or by ray casting from the pixel's position up to a certain limit.

These methods are very fast, but they rely on a trimap as input to find the relations between known and unknown pixels, and they do not provide accurate alpha mattes in many circumstances for natural images.

2.1.2 Propagation Methods

Propagation methods have numerous details in common with sampling methods. These methods reformulate the alpha formula 1.1 with the objective of propagating the alpha

values from the known to the unknown regions.

This propagation of the alpha values to the unknown regions is done by solving equations (like the geodesic distance [1]). Sampling methods can be applied after the application of the propagation method to achieve better results.

2.1.3 Deep Learning Methods

Deep learning solutions are the newer methods to solve the alpha matting problem. These methods make use of deep learning networks. Deep learning networks offer a more robust solution than the previous methods. Their robustness comes from the fact that they can learn high-level features, like patterns, which can then be used to infer the alpha matte of an image.

In this category multiple architectures exist, some receive as input an image and a trimap, similar to the previous methods, while others do not rely on a trimap to calculate the alpha matte. These trimap-free methods are more robust in theory, since they do not rely on already defined data, but they are more computationally expensive and error-prone.

With the use of these architectures, as with any deep learning technique, there is a necessary training phase. This training needs to be done using a dataset of images or videos with their corresponding alpha mattes. Nonetheless, this by itself is a complex problem since creating an alpha matte is a complex task for the user. Many datasets have been created to train these networks, for example the Adobe dataset [35] that contains 50300 synthetic images, 49300 for training and 1000 for testing purposes.

One of many dilemmas with these datasets is that the generation of the alpha matte is a cumbersome process to create, it needs a pixel precise alpha to make the training accurate. This ends with most datasets relying on data augmentation techniques (like fusing two images in which one of them is considered the foreground) to generate more data for training. This ends up with images that may have a hard boundary between foreground and background, which can make the network extrapolate wrong information and generate incorrect results in a real scenario.

2.2 Sampling Methods

2.2.1 Shared-Sampling

The *Shared-Sampling* [10] algorithm is a sampling method that takes a trimap alongside an image as input. It is based on the assumption that, for small neighborhoods, pixels tend to share similar attributes. Taking this assumption into consideration, the unknown regions of a trimap have a lot of redundant work if we take into account the individual pixels. To reduce the redundant work, these techniques are divided in three steps: **Expansion of unknown regions**, **Sample Selection and Matte Computation** and finally **Local Smoothing**.

2.2.1.1 Expansion of unknown regions

Two distances are defined for this step: the image space distance $D_{image}(p, q)$ and the color space distance $D_{color}(p, q)$. The image space distance is calculated by the spatial distance between two pixels (p and q) in the image. The color space distance is given by the difference between the colors of two pixels.

The expansion process begins by checking, for each of the pixels in the unknown region, whether there is another labeled pixel that satisfies the following conditions: the image space distance is less than or equal to a value ($D_{image}(p, q) \leq k_i$), the color space distance is less than or equal to a value ($D_{color}(p, q) \leq k_c$) and the distance between them is minimal. If a labeled pixel satisfies all the conditions, then the unknown pixel is labeled with that class (foreground/background).

2.2.1.2 Sample Selection and Matte Computation

From the previous step, not all unknown pixels end up with defined labels. To solve this problem, each unknown pixel defines which are the best pairs of foreground and background values from a set of samples in the neighborhood. The set of samples must be disjoint from other neighboring pixels and must have a fixed global size (called k_g). The sample selection process is defined by the following steps: each unknown pixel creates k_g line segments, each segment is defined by an initial angle and inside each 3 by 3 window, pixels have different values for this initial angle (as depicted in 2.1). The first labeled pixel in each line segment is chosen as the sample for the pixel.

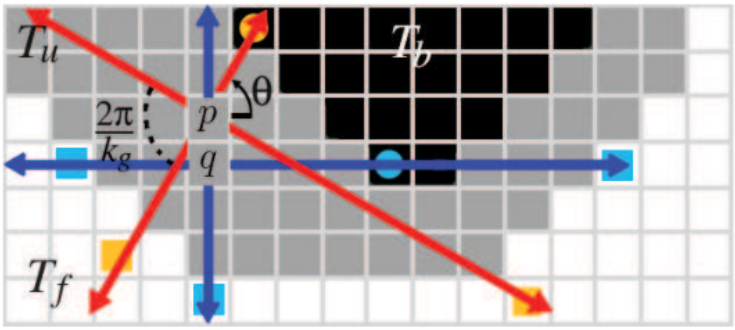

Figure 2.1: Sampling selection procedure (image from the original paper [10])

Thus, as for any sampling method, from the samples selected, statistics and functions are applied to select the best alpha values. In the Shared-Sampling method, functions are defined to take into account the minimization of chromatic distortion and image space statistics (such as the image distance between the pixel and the samples).

2.2.1.3 Local Smoothing

The application of the functions in the **Matte Computation** phase does not guarantee the absence of discontinuities in the alpha values. The discontinuities are removed by

applying weighted averages of the closest neighbors of a pixel. To ensure that erroneous values are not propagated to neighbors, each pixel takes into consideration their confidence values.

At the end of this last step, all the unknown pixels will have alpha values determined and smoothed.

2.3 Basic Concepts of Deep Learning

Deep learning, as previously referred, is one of the possible methods to solve the alpha matting problem. In this section the base concepts applied to the neural networks are explained.

2.3.1 CNNs and Convolutional Layers

CNN (also referred as ConvNet) are one of the most used network models to solve computer vision tasks. These models take an image (or multiple images) as input and provides a multitude of information, for example: segmentation, visual recognition and even medical imaging. These networks are mainly composed of **convolutional layers**.

Convolution layers use multiple convolutions to process images. Convolutions are defined by a kernel (also called filter) and a stride. The kernel is a matrix which is multiplied by the input in each corresponding position. After each multiplication the kernel slides through the image, the sliding is defined by the stride. Kernels are also used in other image processing methods like the Gaussian Blur. The stride determines the distance in the x and y axes that the kernel should move in each iteration. An example of a convolution operation is presented in 2.2

In a CNN structure, the first convolutional layers extract lower level features of an image (like edges and color gradients), but with more layers stacked, the architecture starts to extract higher level features, like objects and faces. For an image that has more than one channel (like a RGB image), the kernel is applied to all the channels and the result is the sum of all channels into a one channel feature map.

There are other types of convolutions. One of them is the **Depthwise Convolution**, the main difference of this convolution is that, instead of applying the same kernel to all the channels it is able to apply different kernels for each channel of the image. A particular case of Depthwise Convolution that uses a 1 by 1 kernel is called **Pointwise Convolution**. There is also **Dilated Convolution** where the kernel is defined with spaces between elements, this convolution tends to increase the receptive field.

Receptive field is an important concept in image processing and also applied to convolutions. It is defined by the size of the region in the input that will be used to generate the features. A practical example that shows the importance of selecting a good receptive field for the task is object detection of large objects. In this case a small size of

the receptive field cannot process the data needed to recognize the objects. Depending on the context the receptive fields may need adjustments.

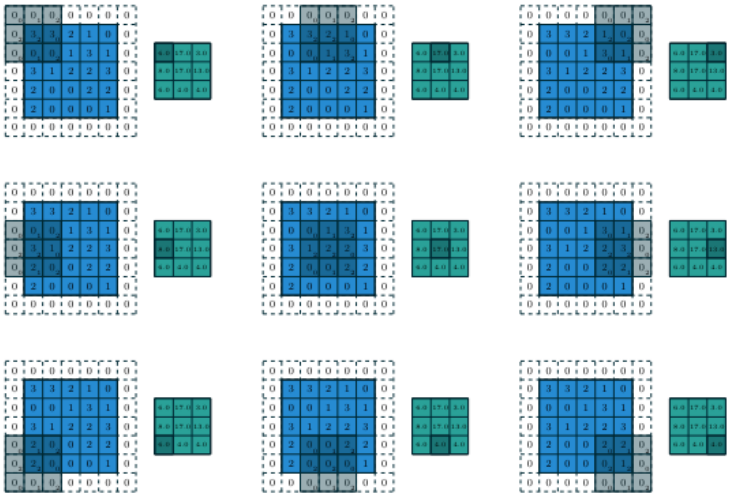

Figure 2.2: The representation of a convolution operation of a 3 by 3 kernel with 0 padding taken from the convolution arithmetic paper [7]

2.3.2 Pooling Operations

Pooling operations come with the purpose of reducing the size of the input image or feature map. The pooling operation is defined by a kernel similar to the convolution operation. The main difference between pooling operations and standard convolutions is that, from multiple image pixels, only a value is kept, reducing the overall size of the feature map.

Many pooling operations exist for selecting what is the value kept from a set of pixels. The *max pooling*, presented in 2.3 keeps, from a group of pixels, the higher number preserved in the final output. Another common used pooling operation is the *average pooling*, where an average is applied to a set of pixels and stored in the output.

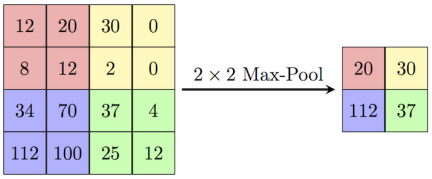

Figure 2.3: Presentation of the max pooling operation

2.3.3 Encoders and Decoders

Encoders and Decoders are two separate neural networks which work cooperatively to process data. The encoder receives an input that can be a sequence of words, images or any other input according to the context of use and outputs features (usually in the form

of numbers) extracted from that input. The decoder does the opposite process, receives features and extracts an output that can be used. In the context of computer vision it is pretty common for the encoder to receive an image (or multiple images) as input, extract features from that input, and the decoder takes those features and produces another image with important data inferred (for example a mask with alpha values). This relation between encoder and decoder is depicted in 2.4.

Figure 2.4: The standard encoder-decoder structure (source: https://d2l.ai/chapter_recurrent-modern/encoder-decoder.html)

2.3.4 Training Networks

One important basic concept to introduce is how does a network extrapolate information from data to achieve results according to the problem.

The total data available for the training, testing and validation is referenced as dataset. The iteration over all the training values present in the dataset is referenced as an epoch.

During training, the input data and the ground truth values are sequentially loaded as needed (if they do not fit in memory). The ground truth values define what is the expected output of the network after the forward pass. A forward pass is the execution, from end-to-end, of a network to generate output values from some input.

From the output data originated from the network and the ground truth expected values a loss function is applied. A loss function describes where the network missed the expected values. When the loss is calculated, a backward pass is executed. The backward pass iterates the network in the opposite direction of the forward pass and updates the corresponding weights of each layer to approximate the output of the network to those ground truth values.

The updates applied to the weights of the layers are based on gradients. Gradients are the derivative of a function that indicates the local slope of that function. These gradients make it possible to predict where the next point in the function will be. Within the context of neural networks training, they are used to determining how much the weights should vary in each backward pass, in order to minimize the error between the network's predicted and correct values. The higher the gradient, the more the weights vary.

2.3.5 Inductive Bias

A neural network during training is always limited by a number of samples present in the dataset. The inductive bias allows a neural network to learn a solution over another based on the ground truth values. In CNNs the inductive bias is implemented through locality, where the relation between neighbor pixels provide features which can then be

related using a hierarchy of resolutions. Batch normalization layers, activation functions and optimizers are also inductive biases that guide the network to a specific solution.

2.3.6 Activation Functions

Neural networks have three different types of layers: the input layers, the hidden layers and the output layers. The input layers feed the hidden layers with the input data. Hidden layers receive information from input layers or other hidden layers and applies weights using activation functions (which are typically the same among them). The output layers provide the final prediction of the model based on the information provided by the hidden layers.

Activation functions are used between hidden layers and output layers to define which values are important to forward to the next layer.

The typical activation function present in the hidden layers are: Rectified Linear Unit (ReLU), Sigmoid, and hyperbolic Tangent. All these functions are non-linear, so it is possible to calculate the derivative function of each one to understand how the weights relate to the input. In the linear activation functions, like the binary step, the derivatives are constant values, this implies that it is not possible to correlate the values to the input.

2.3.7 ResNets

Architectures with a high amount of layers tends to increase the error during training sessions, this happens not because of over-fitting but because gradients start to approach values close to zero (vanishing gradients) or keep getting bigger (exploding gradients). In both situations the architecture never converges to optimal values. To solve this problem, **Residual Networks** [13] were created. These networks have residual blocks with **skip connections**. These skip-connections make activations jump layers. With the application of these residual blocks (depicted in 2.5) it is possible to implement deeper architectures without increasing the training and test error.

One of the most popular *ResNet* architecture is named *ResNet-50*, which is composed of 50 layers where 48 are convolutional layers, one Max Pooling Layer and another layer for average pooling.

2.3.8 U-Net

The *U-Net* [26] architecture is designed for semantic segmentation. It has two paths: the contracting path and the expansive path. In the contracting path repeated convolutions are applied while downsampling the image into lower feature map scales. These feature maps are used in the expansive path concatenated with the obtained upsampled features. This simple structure makes it possible to add more layers in each path of the architecture to achieve specific results. The two paths of the architecture are depicted in 2.6.

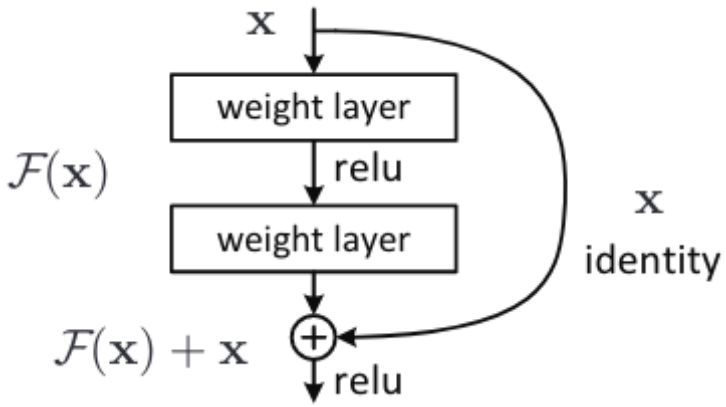

Figure 2.5: The representation of the residual block from the ResNet original paper [13]

The structure of the U-Net is based on the encoder-decoder structure, but instead of giving the output of the encoder to the decoder, the encoder (referred as contracting path) has several intermediate connections to the decoder (here referred as the up-sampling path) to help the decoder get the global context of the image.

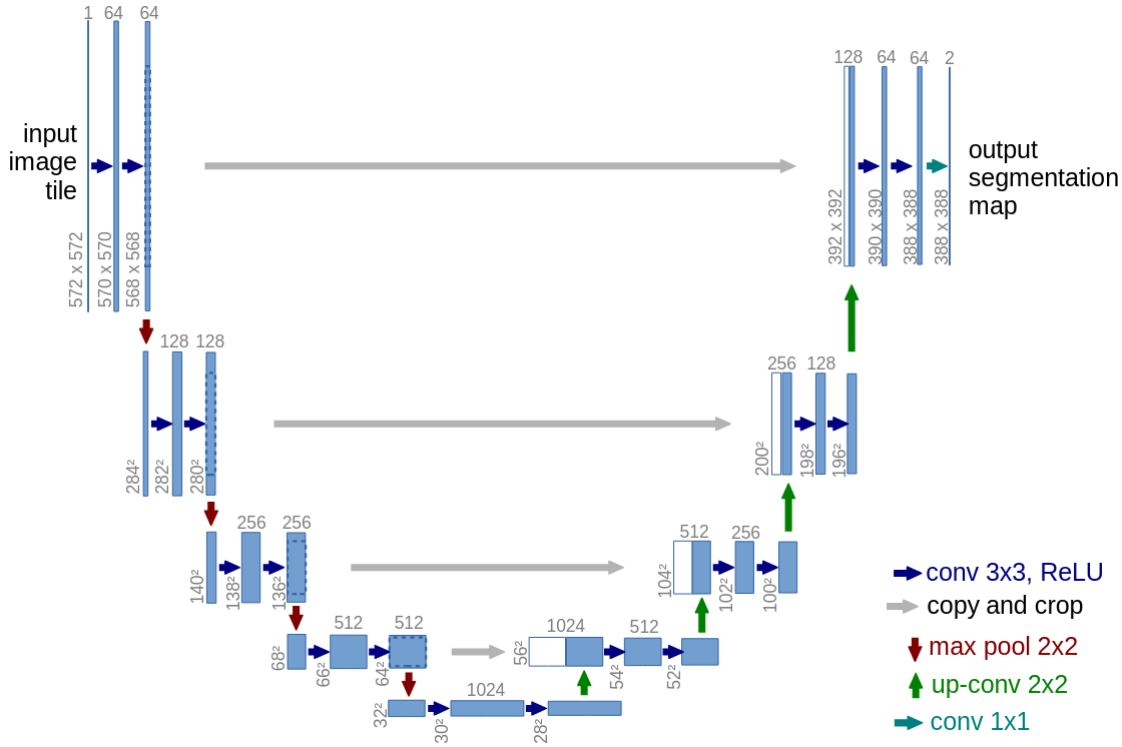

Figure 2.6: A representation of a U-Net architecture from the original paper [26]

2.4 CNN-Based Models

2.4.1 Deep Image Matting

Deep Image matting [35] is one of the first projects that started the use of neural networks to solve the alpha matting problem. The network takes an image and a trimap as input. Its structure (represented in 2.7) is composed of two stages. The first stage is a deep convolutional encoder-decoder network that computes an alpha matte. The second stage, called *Matting refinement stage* is a fully convolutional network that serves the main purpose of refining the alpha prediction from the first stage.

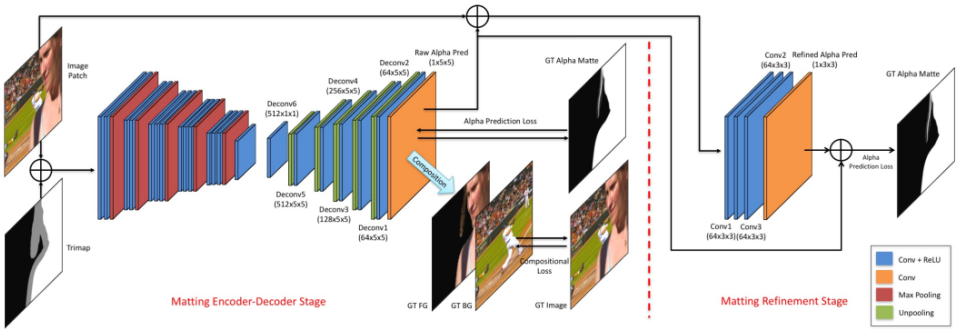

Figure 2.7: Deep Image Matting architecture (image from [35])

2.4.1.1 Encoder-Decoder Network

The input at this stage is an image patch and the corresponding trimap of that patch which are concatenated generating a 4-channel input, 3 RGB components for the image and 1 gray component for the alpha values.

The encoder part of the network receives these patches and transforms them into down-sampled feature maps using 14 convolutional layers and 5 max pooling layers. The decoder reverses the process of the encoder by up-sampling the feature maps using 6 convolutional layers and 5 unpooling layers. At the end of the decoder an alpha prediction layer is used to define the alpha values for each patch received.

2.4.1.2 Matting Refinement Stage

By receiving the concatenation between the image patch and the predicted alpha values, in this phase 4 convolutional layers are applied. This stage's main purpose is to refine and sharpen the alpha prediction from the first stage.

This architecture is a very robust solution compared to the sampling or propagation-based methods because the network can learn the structure and the semantics of a great number of images and apply this knowledge to infer the alpha matte of new images.

2.4.1.3 Training Losses

For training, two losses are applied the first stage. The first one is the absolute difference between the ground-truth alpha values and the predicted alpha values, this loss is called the *alpha prediction loss*. The second loss is the *compositional loss*, defined by the absolute difference between the ground-truth masked image and the predicted image.

2.4.2 Long-Range Feature Propagation Network

The LFPNet [20] follows a patch based approach to process the image and extract features. The chosen design of this network, showcased in 2.9, relies on the fact that the context of a patch in an image is important to generate an accurate alpha matting solution.

When a trimap and its image are given as input, both are divided into patches. The goal of the network is to calculate the alpha values of each patch (similar to the architecture referred in Section 2.4.1). These patches (depicted in 2.8) are divided into an internal center image patch and a context image patch to generate the long-range features. The network is composed of two modules, the propagating module and the matting module, described below.

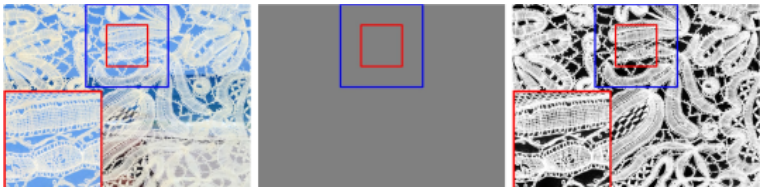

Figure 2.8: Representation of the patches in LFPNet, in red the inner center image patch, and blue is the context image patch (image from [20])

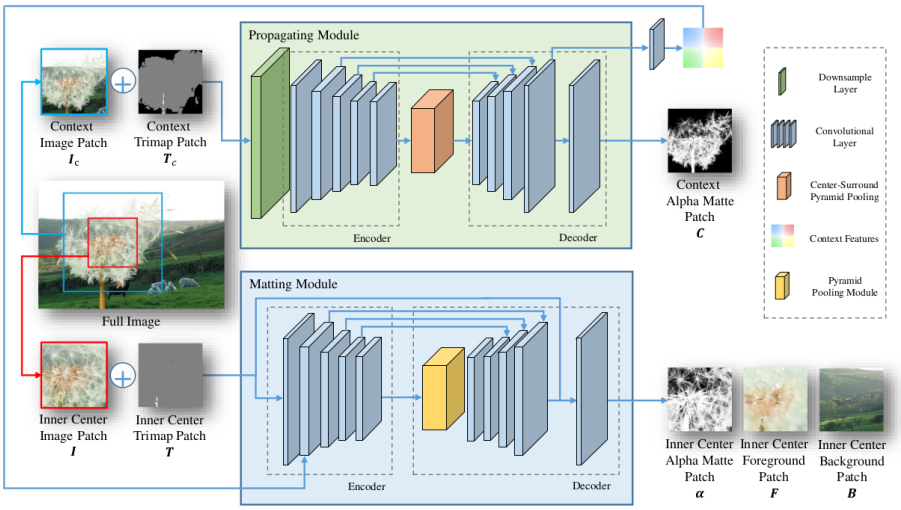

Figure 2.9: Architecture of the LFPNet (image from the paper [20])

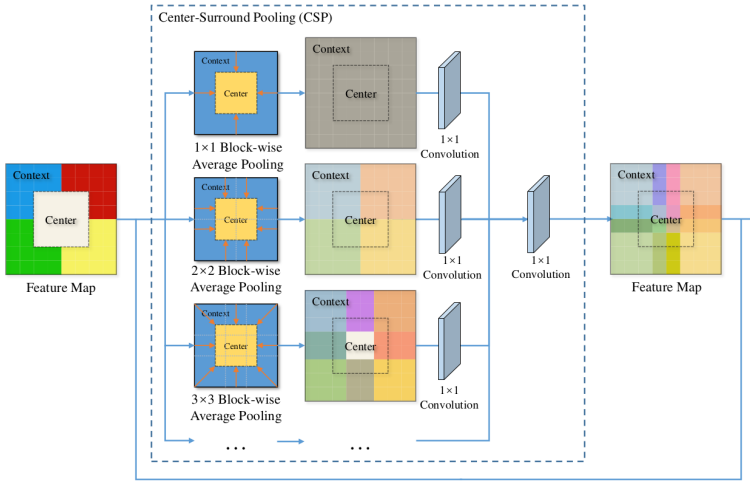

Figure 2.10: Image from the original paper [20] that represents the architecture of the Center-Surround Pooling

2.4.2.1 Propagating Module

The propagating module's structure allows for the extraction of long-range features of the context image patch using an encoder-decoder structure with the application of a Center-Surround Pyramid Pooling (CSPP) layer between them.

The **encoder** phase uses the context image patches, which are then down-sampled by bicubic interpolation. The backbone is used for extracting context features from the image patches. The backbone structure follow the same architecture as the ResNet-50 with the only change being in the block 3 and 4 of the structure where the convolutional layers are changed to dilated convolutional layers to give more receptive field.

The CSPP is responsible for propagating long-range features at multiple scales. To achieve this functionality, the CSPP is composed of a Center-Surround Pooling followed by a Atrous Spatial Pyramid Pooling (ASPP) module.

The **Center-Surround Pooling** splits the context feature map into blocks and applies a multiscale average pooling for each block to obtain the average features of the respective block. Thus, since the context image patch is larger than the center image patch, the context features are propagated to the inner center image patch. After this process, a 1 by 1 convolutional layer is applied and then concatenated with the context feature map. This process is described in 2.10.

The ASPP [3] module smooths the output of the Center-Surround Pooling and extends the receptive fields. This module consists of 6 parallel layers, a 1 by 1 convolutional layer, a global average pooling layer followed by 1 by 1 convolutional layer, and 4 dilated convolutions of 3 by 3 kernels with different dilation rates. The outputs of the 6 layers are concatenated and given as input to the decoder.

The **decoder** up-samples the context features to the original size of the patch, and at the same time recovers the low-level features lost during the downsampling. The decoder is

composed of 4 convolutional layers and 4 up-sampling layers. The output of this decoder is the alpha matte of the context image patch.

2.4.2.2 Matting Module

The Matting Module is an adaptation of the previous described architecture 2.4.1. Adopting a U-Net architecture, this module has an encoder to extract features and a decoder to estimate the respective values of alpha, foreground and background.

The **encoder** of the matting module extracts features of the image at multiple scales from both patches. The backbone is also a ResNet-50 architecture with the previous modifications as the ones referred before.

The **decoder** adopts a Pyramid Pooling Module (PPM) to fuse semantic information extracted at multiple scales. After the features are fused, up-sampling is applied using residual blocks to match the input resolution of the image patches. Skip-connections are used to add the low-level features to the up-sampled feature maps. 3 convolutional layers are stacked to estimate the alpha matte and the respective foreground and background colors.

2.4.3 PP-Matting

Most deep architectures rely on a trimap as input to calculate the alpha matte of an image or video. This makes the problem of calculating the background and the foreground much simpler, since it only needs to account for the pixels in the unknown class of the trimap. Nevertheless, generating a trimap is by itself a complex problem since it needs user assistance to define most of the background and foreground.

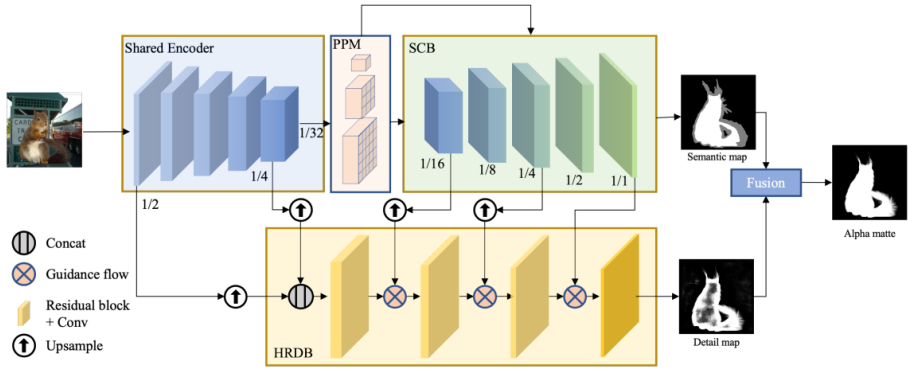

Figure 2.11: The architecture of the PP-Matting taken from the original paper [5]

PP-Matting [5] is an architecture that aims to solve the alpha matting problem for natural images with high levels of accuracy without the need of a trimap. The network of this trimap-free approach is composed of parallel branches, one called Semantic Context Branch (SCB) and the other High-Resolution Detail Branch (HRDB), with an encoder in common (as depicted in 2.11).

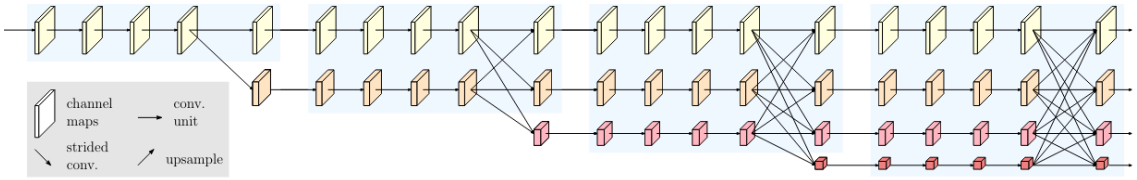

Figure 2.12: Main body of *HRNet* from the original paper [30]

2.4.3.1 Encoder architecture

The encoder is strongly based on *HRNet48* [30] architecture. The HRNet48 is used because most of the deep convolutional networks gradually use a reduced size of the feature maps to be able to process high resolution images. After this reduction of size, it may be needed to recover a high-resolution representation based on the low resolution. The HRNet48 is a type of architecture that maintains a high-resolution representation of an image throughout the layers.

The encoder is composed of different stages: First the image passes by 2 convolutions with stride 2 and a 3 by 3 kernel, thus decreasing the resolution by a factor of $\frac{1}{4}$. After this point, the result of this operation is given to a set of layers which is referenced in the paper by the *main body*.

This main body consists of several stages. The first stage is composed by high-resolution convolutions. The following stages add parallel streams gradually from high to low resolutions. So, each subsequent stage has all resolutions from the previous plus a lower one (illustrated in 2.12). At the end of each stage, a fusion module is included to exchange information between the multiple resolutions. So the output of the main body has the same resolution as the input given.

This encoder can maintain a high resolution as it benefits from parallelism between the multiple resolutions, granting a better overall performance. The output of the encoder is then sent to the next two parallel branches that will be described below.

2.4.3.2 Semantic Context Branch

Before entering into this branch the input passes by a PPM. The PPM [39] is a module used to extract global context from an image. In computer vision tasks, the size of the receptive field is important to extract as much detail as possible from an image. The PPM works on solving this task by fusing feature maps from different pyramid scales.

From a feature map of an image, this module generates different feature maps in different pyramid scales, afterwards all the feature maps are up-sampled to match the size of the feature map received as input. Finally, the different levels of features are concatenated, including the feature map received as input. The output is a global feature map generated with different levels of detail and serves to improve the semantic context from the data given by the shared encoder.

From the PPM the semantic context can fall into ambiguity due to the absence of a trimap. The SCB solves this problem by extracting global semantics present in the image. Composed of 5 blocks, with each one being composed of 3 layers and each layer in the block composed by a convolutional layer, a batch normalization, a ReLU and one bilinear up-sampler. For context, bilinear upsampler/downsampler uses two linear interpolations of color, in the x and y directions to calculate the color of a pixel.

As input to this branch, the architecture makes use of features at $\frac{1}{32}$ of the original image's resolution, which is provided by the shared encoder.

2.4.3.3 High-Resolution Detail Branch

The structure of the HRDB is composed of 3 residual blocks and 1 convolutional layer at the end. The input given is a composition of low and high level features given by the shared encoder. Between each two blocks, a *Guidance Flow* module is applied.

The Guidance Flow acts as a bridge between the two branches to make the semantic features help on the prediction of details in the HRDB. This bridge is composed of multiple Gated Convolutional Layers (GCL). The GCL works by concatenating the semantic context to the detail features and applying 2 blocks of operations. The first block is composed of a convolution, a batch normalization and a ReLU. The second block applies a convolution and a batch normalization followed by a sigmoid function. The output of these blocks is called a *guidance map*. The generation of this map can be expressed by the following formula:

$$g = \sigma(\mathsf{C}_{1x1}(s \| d)) \tag{2.1}$$

where s is the semantic data of the SCB and d the details created by the HRDB. The σ denotes the sigmoid function and the C_{1x1} a normalized 1 by 1 convolution. After the creation of the guidance map by following the previous formula, the new detail feature map is calculated by:

$$\widehat{d} = (d \odot g + d)^T w \tag{2.2}$$

where $\widehat{d}$ is the new feature map, the d is the feature map defined by the HRDB, g is the guidance map and w a weighting kernel. These multiple GCL can be applied on any block of the SCB. In the original paper, 3 GCL are applied with the first, third and last block of the SCB.

At the end of HRDB, with the application of the Guidance Flow between blocks, the output is a detail map. The focus of this map is the details in the transition regions of the predicted foreground and background.

2.4.4 Robust Video Matting

In the previous sections, the presented architectures aim to solve the problem by predicting the alphas of an image. The RVM [18] architecture (represented in 2.13) aims to solve the alpha predictions for a video by making use of temporal information.

With temporal information it is possible for an architecture to "see" multiple frames and its predictions and features. In situations where an image could have ambiguity, for example when a foreground color temporarily becomes similar to the background, it is possible to rely on previous frames to solve these problems. RVM is a trimap-free architecture, so for a given video it is not needed to provide any type of additional input.

This architecture follows a recurrent paradigm where an encoder extracts features of an individual frame and a recurrent decoder joins temporal information. At the end of the encoder-decoder structure, a *Deep Guided Filter* is applied to generate the final output.

The network can receive multiple frames at the same time. During training the batch normalization can compute statistics to guarantee consistent normalization. Nonetheless, for live video streaming it is possible to use only 1 frame at a time but batching multiple frames can take advantage of the parallel capabilities of the hardware to process all the batched frames at the same time.

2.4.4.1 Encoder

This encoder is based on state-of-the-art segmentation networks. Using these types of networks, it is possible to improve the detection of human figures in a frame. The encoder can be any type of network but for the original paper the encoder is strictly based on *MobileNetV3-Large* [15] which improves the overall performance of the network and can be applied to mobile devices. Nevertheless, there is a version of the architecture that uses RestNet-50 as the encoder.

The MobileNetV3-Large is defined by an initial full convolution of the input and the application of depthwise separable convolutions (briefly explained in 2.3.1). In these networks, the standard convolution layer applied is a combination of depthwise convolutions and a pointwise convolution. This factorization of the standard convolution reduces the computational costs of the operation. And since the network follows a simple structure, it is possible to be used in different contexts and in different topologies.

The encoder extracts features at different scales to the recurrent decoder. By following the MobileNet [14] paper, it is applied a Lite Reduced ASPP to the output of the encoder to optimize the performance and to extract denser features.

2.4.4.2 Recurrent Decoder

In a decoder where temporal information is needed, it is common to remove old information to store new one. This approach is limited since it may be needed a lot of old information in a video to accurate predict new alpha values. This architecture solves this problem by applying a recurrent mechanism making it possible to store long and short-term temporal information in a video. To bundle all this information the decoder makes intensive use of Convolutional Gated Recurrent Units (ConvGRU) at multiple scales. In terms of composition, the decoder is divided in 3 major blocks:

The **Bottleneck block** grabs the $\frac{1}{16}$ from the feature scale of the encoder and applies a ConvGRU layer on half of the channels.

The **Up-sampling block** is applied at $\frac{1}{8}$, $\frac{1}{4}$ and $\frac{1}{2}$ feature scales in sequential order.

The last major block is called **Output block**, this block firstly merges the bilinear up-sampled output of the previous block with the input image. Then it applies sequences of operations consisting of 2 convolutions, batch normalization and ReLU strictly in this order to extract final hidden features. These features are converted to the output that include 1-channel alpha prediction, 3-channel foreground prediction and 1-channel segmentation prediction.

The use of the ConvGRU in half of the channels helps it focus on aggregating temporal information, while the other half is forwarded as spatial features of the current frame.

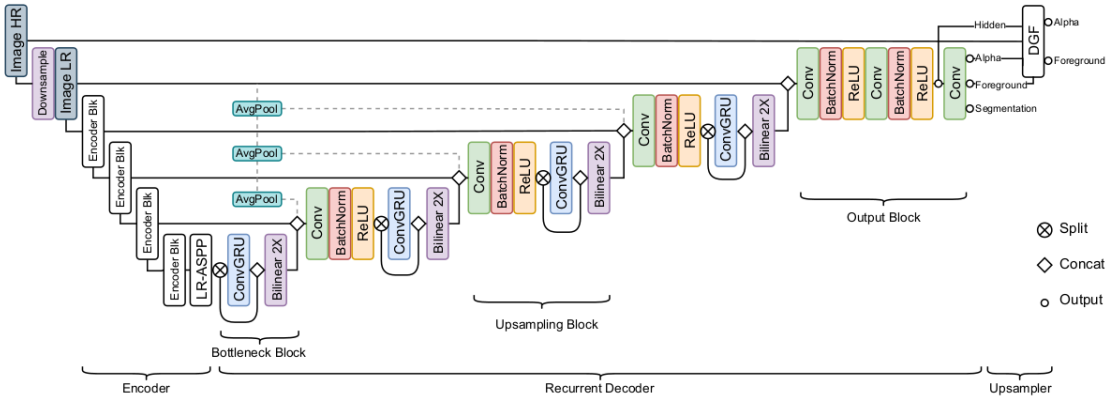

Figure 2.13: Architecture of the RVM from the original paper [18]

2.4.4.3 Deep Guided Filter Module

The frame is initially down-sampled when passing through the network. The main purpose of the *Deep Guided Filter* [32] is to generate high resolution alpha, foreground and features from the low-resolution data predicted and the original high resolution input image. This module is optional, it is only necessary if the input frame has a bigger resolution (UHD or FHD) than what the output predictions provide.

The Deep Guided Filter module is a network composed of Fully Convolutional Network (FCN) and a guided filtering layer [33] that receives the low-resolution data from the FCN and generates the high resolution data, as showcased in 2.14.

In the figure 2.14 it is depicted the standard use of a guided filtering layer. I_h is the input image, I_l is a down-sampled of I_h and the O_l is the output generated from an arbitrary network. The guided filtering layer takes the down-sampled images alongside the original image and extrapolates the new output image, referred as O_h with the same original input resolution

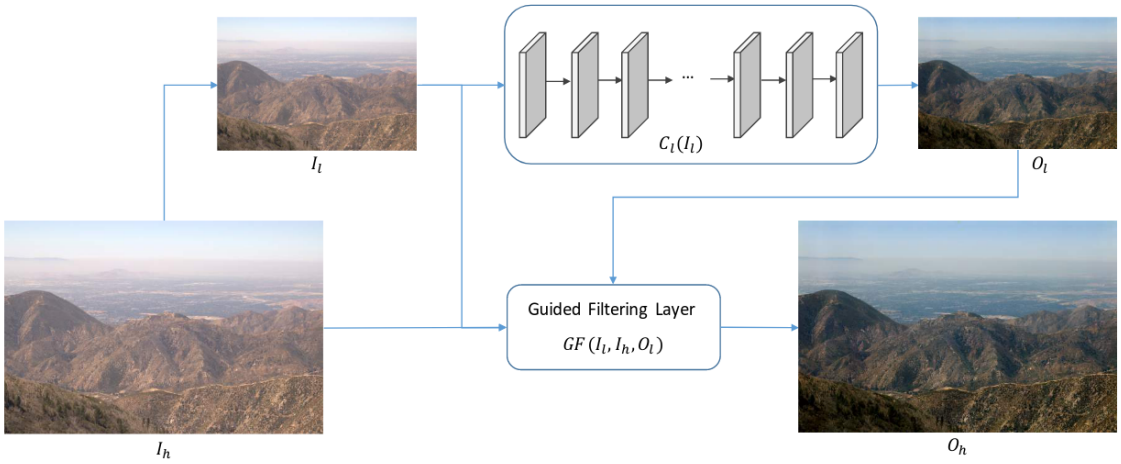

Figure 2.14: The guided filtering layer structure (image from the paper [33])

2.4.4.4 Training Approach

This architecture, different from the previous ones referred, is strongly based on segmentation tasks to solve the alpha matting problem. This is so because human matting is closely related to human segmentation, since both rely on detecting the human in an image. It is possible to take advantage of a network trained to semantically detect human subjects. Another big advantage of this fusion of tasks is the amount of data available. There are more datasets for segmentation tasks than for matting, and since segmentation datasets tend to be more complex, this prevents the model from over-fitting to synthetically generated matting data that does not correspond to real use cases.

2.5 Transformers

The use of transformers became mainstream in Natural Language Processing (NLP). This was so because transformers, with the MHSA [29] block (shown in 2.15) can easily grab the global context of a set of sentences and find relations between them. With the popularity of the transformer-based architectures, computer vision researchers took advantage of these novel architectures to achieve better results.

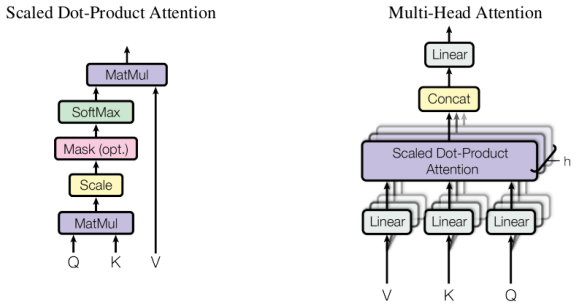

Figure 2.15: The standard block of MHSA presented in [29]

2.5.1 Multi-Head Self-Attention

The self-attention mechanism, presented in [29], takes as input three matrices referenced by Q, K and V. These matrices are originated from three different linear layers with their respective weights. The input matrices are logically split among the different attention heads. Inside each attention head the following formula is applied:

$$att_h = softmax(((Q_h * K_h^T + mask)/\sqrt{querysize})) * V_h \tag{2.3}$$

Where Q_h, K_h and V_h are the split input matrices for an input head h. $querysize$ is a hyperparameter that defines the size of the linear layers to produce the Q, K, V matrices.

This equation creates the attention scores for each attention head. At the end of the process, all the attention scores are merged together and forward to the next layers.

2.5.2 Vision Transformer

A Vision Transformer (ViT) [6] works by applying the mechanism of self-attention to computer vision tasks. This model flattens the image into several patches, which serve as input to the transformer. These architectures that use vision transformers can remove completely the need for convolution operations as a part of their structure to extract features.

A hybrid architecture can also be applied with this block, in these architectures feature maps are extracted using a CNN and served as input to the ViT based models.

The ViT block and others modified ViT blocks are used as a baseline for multiple state-of-the-art networks.

2.5.3 Swin Transformer

The *Swin Transformer* [21] (one of the many modified transformers based on ViT) was developed as a general-purpose ViT backbone by Microsoft for computer vision tasks.

The name *Swin* comes from the fact that the transformer makes use of shifted windows (depicted in 2.16) to apply self-attention in an image. The base architecture implements the same patching technique as the ViT, where an image is split into a number of patches. A window is defined by a set of patches, and the shifting is performed between layers. This shifting is the mechanism that ensures connections between different windows. This mechanism is applied to the Swin Transformer block. Between two Swin Transformer blocks, a patch merging layer is applied to the data to reduce the number of tokens by concatenation of patches.

2.5.3.1 Video Swin Transformer

The *Video Swin Transformer* [22] is an adaptation of the Swin Transformer mentioned earlier, the main difference is that it in consideration the temporal information of the videos. To find relations between pixels, a spatio-temporal distance is calculated, which means that

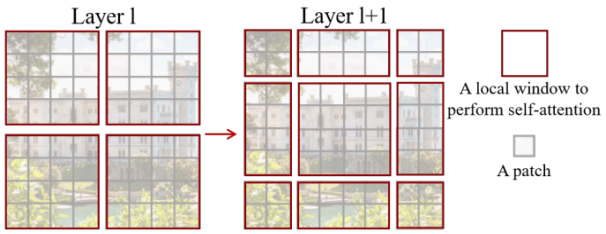

Figure 2.16: The shifted windows mechanism presented in the original paper [21]

pixels closer to each other in spatio-temporal terms have a higher probability of having some relation between them.

The Video Swin Transformer changes the self-attention layer using the shifted window mechanism to support the extra dimension of the temporal data (as shown in 2.17).

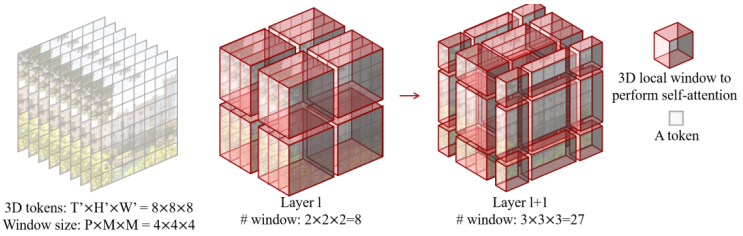

Figure 2.17: The shifted windows mechanism applied to video [22]

2.5.4 MobileViTv2

All the previous methods referred make use of the self-attention block present in the first ViT 2.5.2 implementation. Although these solutions give state-of-the-art results in many benchmarks, the main problem turns out to be the latency of processing each image. The MHSA block has $O(k^2)$ time complexity where k is the number of patches of the image. *MobileViTv2* [24] comes to solve this problem, reducing costly operations and the memory requirements of typical transformer architectures.

MobileViTv2 defines a block called *Separable Self-Attention*, in which, similar to MHSA, the input x (which has k by d dimensions) is divided in three different branches: input, key and value. The input branch applies a linear projection using a set of trainable weights, that serve as a *latent token*. The linear projection computes the distance between the input x and the latent token resulting in a k-dimensional vector.

A softmax operation generates the context scores c_s, composed of k dimensions, based on the previous k-dimensional vector. Using this softmax to generate the context scores, instead of generating the context scores for each token for all the k tokens, reduced the time complexity of $O(k^2)$ to O(k).

After the generation of the context scores, a context vector is calculated based on the following formula:

$$c_v = \sum_{n=1}^{k} c_s(i) x_K(i) \tag{2.4}$$

where x_K is a linear projection of the input to a d-dimensional space using the key branch with trainable weights. This linear projection creates a composition of k by d dimensions.

In parallel, the input x is linearly projected using the value branch with trainable weights, creating an output of d by d dimensions. This output goes through a ReLU layer producing a k by d dimensions vector referenced as x_v.

$$x_v = ReLU(x * W_v) \tag{2.5}$$

where the W_v are trainable weights.

After all these steps, another linear layer is applied to make a relationship between the contextual information present in c_v and the values in x_v, so the final output is composed of k by d dimensions.

Separable Self-Attention is described as:

$$y = (c_v * x_v) * W_O \tag{2.6}$$

where c_v is the formula in 2.4, the x_v is represented in 2.5.

The MobileViTv2 architectures take advantage of the Separable Self-Attention block and convolution operations to create a hybrid network directed at mobile-devices where memory and processing power is a constraint. As such, these architectures focus on inference speed instead of high accuracies. The results comparing multiple architectures are provided by the authors in 2.18

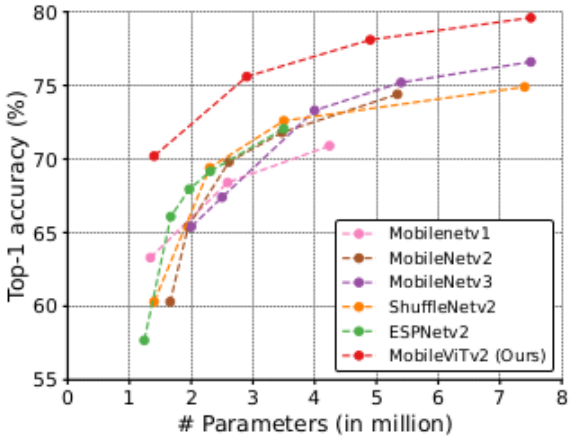

Figure 2.18: Accuracy between CNNs and MobileViTv2 presented in [24]

2.5.5 Swiftformer

Alongside the previous MobileViTv2, more research was made to extract global context from images in computer vision without needing the $O(k^2)$ time complexity present in the typical MHSA. Another example of an efficient transformer block came from the *Swiftformer* [27] architecture.

As seen in MobileViTv2, the typical transformer architectures use 3 different branches generated from the input: the value, key and query branch. In the case of MobileViTv2, linear projections were applied with trainable weights to generate interactions between each branch. In the *Swiftformer* architecture a new transformer block called *Efficient Additive Attention* removes the interaction between the key and value branches. The authors of Swiftformer believe that making query-key interactions using linear projection layers is enough to learn relationships between different tokens.

In the Efficient Additive Attention block the input starts by being transformed into the query and key branches using their respective trainable weights. The calculated query matrix is then multiplied by a set of learnable parameters, referred to as w_a followed by a softmax operation. This operation is defined by the following formula 2.7

$$\alpha = softmax(Q * w_a / \sqrt{d}) \tag{2.7}$$

Where Q is the query matrix generated from the input using learnable weights and w_a the parameter vector of d dimensions.

From the previous step, a global query vector referred as q is generated using:

$$q = \sum_{i=1}^{n} \alpha_i * Q_i \tag{2.8}$$

At this point the only operation left is the relationship between this global query vector q and the key matrix. To encode these relationships, an element-wise product operation produces the global context similar to the attention matrix present in the typical MHSA block. Finally, the output of the Efficient Additive Attention block is defined by the equation 2.9.

$$y = Q_n + T(K * q) \tag{2.9}$$

Where Q_n is the normalized query matrix, and T a simple linear transformation used in an element-wise product of the global query vector and the key matrix $K * q$.

The Swiftformer architectures (presented in 2.19) take advantage of this Efficient Additive Attention block alongside convolution encoders to learn global and local representations of the input image. These networks are another example of the trade-off between fast inference speeds and accurate results. According to the authors, the architectures are presented with 4 different sizes, referred to as *Swiftformer-XS*, *Swiftformer-S*, *Swiftformer-L1*, *Swiftformer-L3*. Also, according to the authors *Swiftformer-XS* can achieve latencies of 0.7 milliseconds on an iPhone 14 using a custom neural engine.

2.6 Transformer-Based Models

2.6.1 Matteformer

The *Matteformer* [25] makes use of a transformer-based architecture to solve the alpha matting problem. This architecture takes a trimap and an image forwards that information

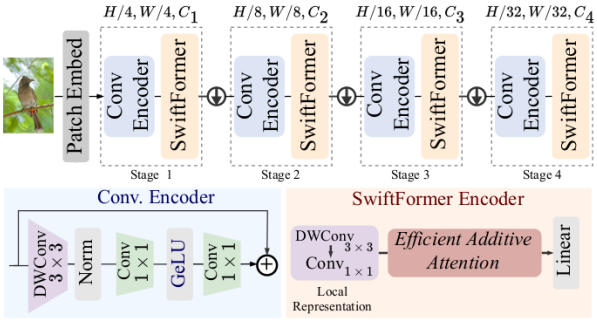

Figure 2.19: Swiftformer's base architecture presented in [27]

to the transformer. This architecture follows a typical encoder-decoder structure. For the decoder part, a simple set of convolutions and up-sampling layers are applied, as for the encoder, a more complex structure was chosen.

The encoder unit is composed by an altered version of the previous mentioned Swin Transformer block, along with the patch merging layer to reduce the number of tokens. This altered version of the Swin Transformer block, called PAST block (presented in 2.20), as the name suggests, takes information of prior tokens to compensate the low receptive field of local windows at an initial state, which gives the lower layers a too small of an attention region.

Prior tokens are generated for the three regions of the trimap: the foreground, background and unknown. This is calculated by averaging all the tokens belonging to the respective regions (as presented in 2.10).

$$p^q = \frac{1}{N_q} \sum_{i=1}^{N} r_i^q \cdot z_i \tag{2.10}$$

where N is the number of tokens, q is the respective query (foreground, background or unknown), i is the token index, the z_i is the feature of the token, r_i^q is a number that confirms if the token i belongs to the region q (0 if it does not belong, 1 if belongs to the region).

Therefore, by applying these prior tokens, the self-attention layer can not only calculate features from the local window, but also grab global features from these tokens accumulated from the previous blocks to use in the calculations. As the prior tokens are concatenated with the spatial tokens in the layer, a MHSA is applied following the common structure of a transformer.

The Matteformer architecture is showcased in 2.21.

2.6.2 VM-Former

Designed to solve the video matting problem, the network in [16] (showcased in 2.22), named *VM-Former*, starts by receiving video frames as input to a backbone architecture, which is applied to extract feature maps from the frames. Following the backbone network (which is typically a CNN), a Transformer encoder and decoder are applied.

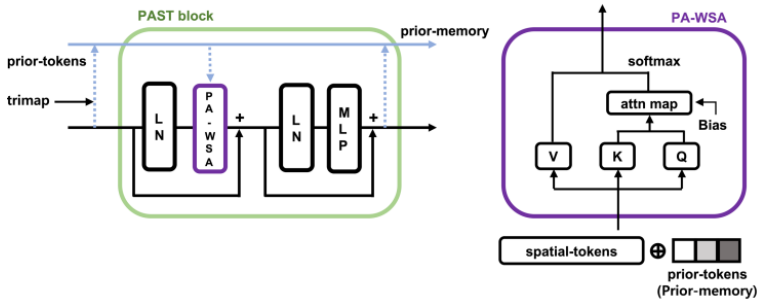

Figure 2.20: Matteformer's PAST block structure from [25]

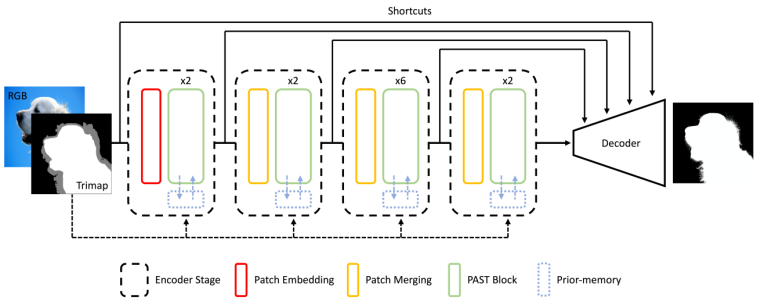

Figure 2.21: Matteformer's base architecture presented in [25]

2.6.2.1 Transformer encoder

This encoder has a variable number of blocks, with each block being made of a self-attention layer and a multi-layer perceptron. Layer normalization and residual connections are applied to the output of the previous layers. The conjunction of these layers outputs a feature sequence for each frame. If multiple frames are given, this encoder concatenates all the calculated feature sequences.

A convolution layer is applied to the feature sequence of the previous frame to obtain a short range of features with temporal information between two frames. Then, the output passes through a *Feature Pyramid* network, referred in Section 2.6.2.2. This phase is described in the paper as *feature modeling*.

2.6.2.2 Feature Pyramid Network

The Feature Pyramid Network [19] works as a feature extractor, it generates feature maps on multiple scales. This feature extractor leverages the convolution network to extract features from low to high scales.

This architecture has two pathways: the Bottom-up path calculates features at different levels, and each new scale's features are used to calculate the next one. The Top-down path starts by up-sampling the higher pyramid levels, which have the lowest scales. While up-sampling from the lowest scales, lateral connections are made with the layers of the Bottom-up path to merge feature maps of the same spatial size.

The next phase is used for query modelling. The queries of the first step are generated having the same length of the generated feature sequences. In each decoder block the

queries go through a self-attention layer, a multi-layer perceptron and a layer normalization, similar to the encoder block. Applying a Cross-Attention module to this new generated data, relations can then be found between the corresponding feature sequences using a softmax function.

After Cross-Attention, a new module is applied with purpose of generating long range and query-based temporal information. The predicted alpha mattes are calculated simply by matrix multiplication of feature maps and the queries.

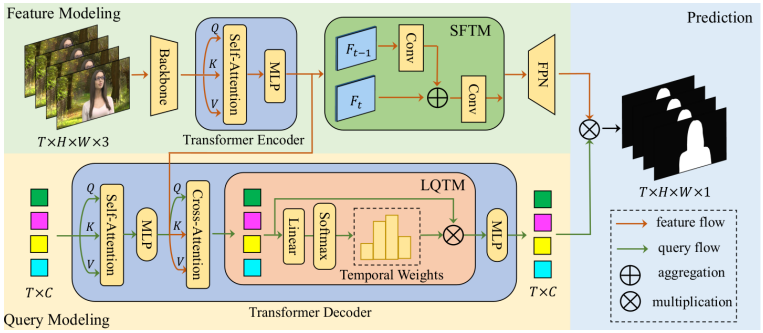

Figure 2.22: VM-Former's architecture from the original paper [16]

2.7 Refiners

As previously referred in 2.4.4.3, refiners are used to match the output of a network to the input resolution. Refiners are needed because typically to improve the inference speed the network, the image is downscaled before entering the network. If the output size differs from the original image size (before downscaling), a refiner needs to be applied in order to scale up the output.

2.7.1 Deep Guided Filter

Guided filters are a set of methods built to extract from a low-resolution image a high-resolution version. This set of operations is typically a linear model represented in 2.11.

$$\widehat{O}_l^i = A_l^k I_l^i + b_l^k, \forall i \in \omega_k \tag{2.11}$$

Where ω_k is a local window of a low resolution image I_l given as input, the element i is a pixel inside ω_k. A_l and b_l are obtained by minimizing a reconstruction error.

The high-resolution output is produced by upsampling the learned values of A_l and b_l and applying a linear model to the high resolution input I_h as to generate the high resolution output O_h.

$$O_h = A_h * I_h + b_h \tag{2.12}$$

The paper [32] takes the previous implementation and creates a differentiable layer to be integrated in any type of network model.

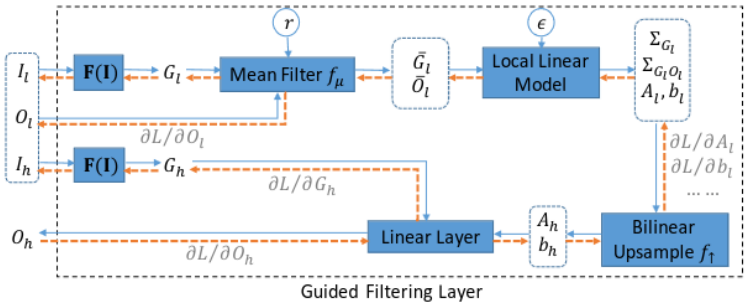

Figure 2.23: Image from [32] that represents the computation graph of the Deep Guided Filter

2.7.1.1 Convolutional Guided Filter

Since the differentiable guided filter has few parameters, *Wu Huikai et Al* [32] created a convolutional guided filter that has the capacity to perform well in different scenarios. The mean filter is replaced by dilated convolutions, and pointwise convolutions are applied instead of the local linear model. The computation graph of the convolution-based version is depicted in 2.24.

Applying these changes makes it possible for the model to be used in various applications and trained for a specific task when integrated with a model.

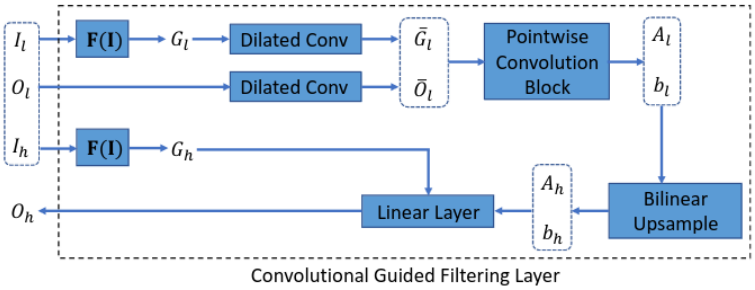

Figure 2.24: Image from [32] that represents the computation graph of the convolutional guided filter layer

2.7.2 Spatio-Temporal Refiner

The *Spatio-Temporal refiner* [34] explores the use of spatial and temporal features to improve the downscaling and upscaling accuracy of a video. The referred paper proposes a space-time video downsampling method merged with upsampling.

This approach to the downsampling/upsampling flow avoids using the typical bicubic interpolation for downsampling a frame as a single operation, instead, interprets the process of downsampling as a part of the network with the ability of being differentiable.

The spatio-temporal refiner also proposes 2 new modules: a deformable temporal propagation and space-time pixel shuffle (depicted in 2.26) to achieve a high reconstruction performance of each downsampled video frame. The paper also defines an end-to-end network with the previous modules integrated. This network is depicted in 2.25.

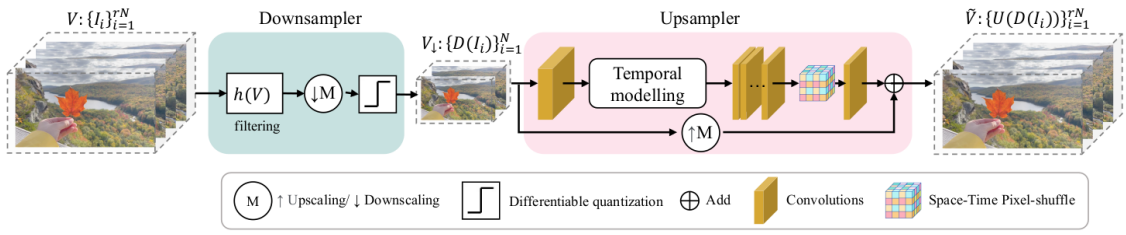

Figure 2.25: Image provided from the spatio-temporal paper [34] that depicts the training framework with upsampler and downsampler merged in a network

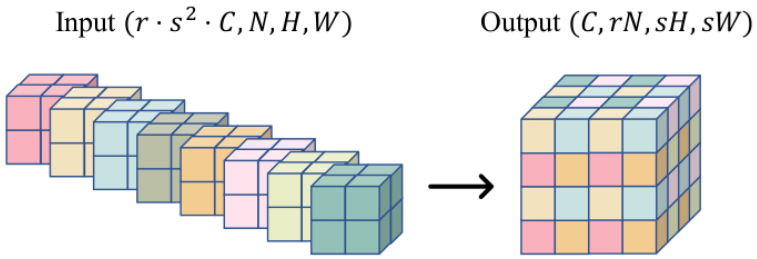

Figure 2.26: The Space-Time shuffle proposed in [34]

2.7.3 SparseMat

SparseMat [28] aims to solve the memory consumption of matting networks, by focusing on achieving UHD matting without upsampling/downsampling artifacts. This refiner defines two types of redundant computation: spatial and temporal redundancy. Both type of redundancies are put into operation by two concepts: spatial and temporal sparsity.

Spatial redundancy is defined, inside a frame, where large regions of redundant work can be avoided (for example large regions of solid pixels). For establishing the spatial redundant work a spatial sparsity map is created. This map is defined by a dilation and an erosion operations applied to the output of a low resolution alpha matting network (composed by foreground colors and alpha values) in order to define the boundaries where there is a transition between foreground and background.

Temporal redundancy is described by the redundant information of consecutive frames that have little pixel-wise difference between them. To create the temporal sparsity as to define the redundant work the alpha matte of two consecutive frames are subtracted, and a binary map is created to indicate where the divergent pixels are located. The intersection between the spatial sparsity and the temporal sparsity defines the spatio-temporal sparsity map used in the next phases.

The sparsity map defines which pixels are active and should be considered during the reconstruction of the alpha matte at a higher resolution. This reconstruction using the active pixels is applied by the *Sparse High-Resolution* module.

2.7.3.1 Sparse High-Resolution Module

The Sparse High-Resolution module is a CNN implemented using only sparse operations, which only consider the active pixels. Before the decoder, global information is extracted from the bottleneck part (where more features are generated), and a masked Squeeze-and-Excitation layer is applied (referred in the paper as *Masked Squeeze-And-Excitation* or Masked-SE).

The Masked-SE applies global average pooling with the foreground and background layers generated from the alpha matting network. At the end of each average pooling, a fully connected layer is applied for each individual image layer (foreground and background). Both image layers are merged, and a fully connected layer followed by a sigmoid function is used. The structure of these operations is depicted in 2.27.

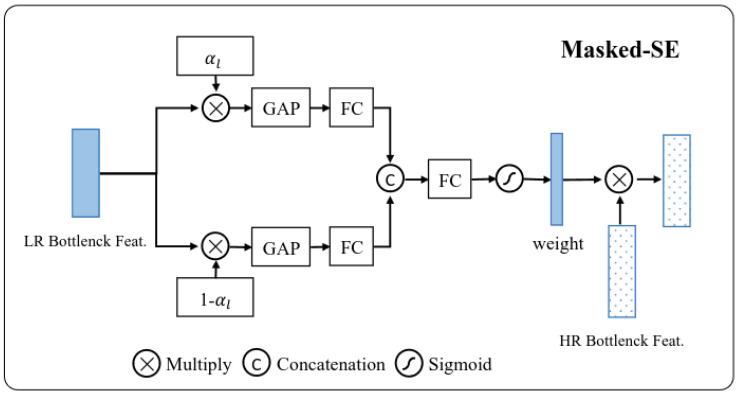

Figure 2.27: The Masked Squeeze-And-Excitation operation (image from [28])

After the sigmoid function, a channel-attention weight is created which is then multiplied by the features generated from the bottleneck block. After this multiplication the network continues with the sparse decoder and creates a sparse high-resolution alpha matte as output.

Then, the sparse high-resolution alpha matte is converted to a dense structure. The formula in 2.13 provides the calculations to generate a dense alpha matte of the frame i that matches the resolution in the input.

$$\alpha_h^i = (1 - M) \cdot (1 - M_s) \cdot \alpha_{l\uparrow}^i + (1 - M) \cdot M_s \cdot \alpha_h^{i-1} + M \cdot \alpha_{ph}^i \tag{2.13}$$

Where M is the spatio-temporal sparsity map previously generated, with M_s being the spatial component of the sparsity map. The $alpha_{l\uparrow}^i$ is the low resolution alpha matte upsampled to match the input resolution using standard bilinear upscaling. The α_h^{i-1} is the high resolution alpha matte of a previous frame.

At the end of the sparse network, and the conversion of the alpha matte to a dense representation, the proposed framework ends with a high resolution output based on a low resolution prediction of a generic framework.

2.7.3.2 Out-Of-Memory Problems

The paper also adds that, when working with UHD images, the number of active pixels that the network needs to take in consideration can end up causing Out-Of-Memory (OOM) errors in some GPUs.

To avoid OOM errors the paper suggest cascading inference. The previous algorithm is applied in multiple low resolutions until it reaches the same resolution as the input. At each resolution the number of active pixels is reduced from the previous generated alpha mattes. This implementation tends to decrease inference speed but improves memory usage. The process of cascading inference is depicted in 2.28.

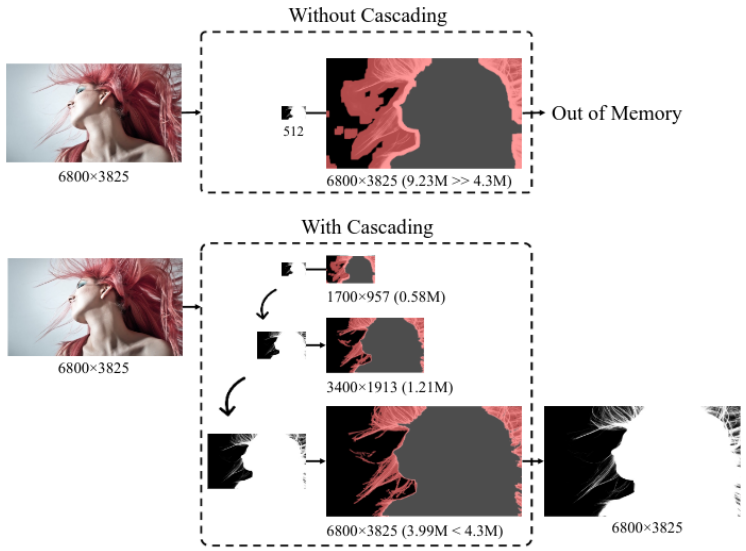

Figure 2.28: Cascading inference to reduce the number of active pixels between resolutions (image from [28])

2.8 Trimap Generation

Up to this point, multiple architectures have been discussed based on their original papers. Some architectures use the transformer structure, others still use the CNN approach. But many architectures like the aforementioned Deep Image Matting, or Matteformer, need an important input to give results with high accuracy: a **trimap**. Using this approach human effort is needed to draw the trimap before entering the network. In a video, if a trimap is required for each frame, this methodology proves to be almost unfeasible. To mitigate this problem, different methodologies have been designed to automate this process.

2.8.1 Automatic Trimap Generation

To generate a trimap, *Gupta et al* [11] rely on the fact that in an image, a salient object is present on the scene. From this point, the framework is defined by three phases to generate

the trimap of the images: over-segmentation and feature description, identification of background and foreground superpixels, trimap generation.

2.8.1.1 Over-segmentation and feature description

In this phase the image is segmented into N superpixels. Superpixels are groups of pixels that have relations between them. From each superpixel, features are extracted into patches of 13 by 13 pixels, these features are robust to illumination changes, contrasts and geometric distortions.

From the superpixel features, saliency maps are calculated using multiple independent methods. Three were presented in the original paper [11]. The first relies on supervised learning to integrate regional features to form the saliency map. The second follows a score approach where a set of object candidates are generated from segmentation and then an algorithm is applied to rank the different regions based on a saliency score. The third makes use of global and local context by using a CNN architecture to extract a saliency model of the image. These three outputs are combined to obtain the saliency map.

2.8.1.2 Identification of Background and Foreground Superpixels

This phase takes the calculated saliency map and classifies the superpixels into salient or non-salient. This classification is determined by obtaining an average value of the saliency for the superpixel. The superpixel is classified as salient if its value exceeds some threshold, otherwise is non-salient. As an initial step, the foreground pixels are the ones classified as salient and the background pixels are all the superpixels that are classified as non-salient.

This identification of foreground and background pixels can lead to inaccuracies in the results. For that purpose, a clustering algorithm of the extracted features of the superpixels is applied. Thus, ten clusters are generated, five for the foreground pixels and the other five for the background pixels. For each superpixel, a Euclidean distance is calculated between the superpixel and the cluster center of the opposed class. In the case that the calculated distance is less than a predefined threshold, the class of the pixel is changed to the class of that cluster. The saliency map is then updated with the new classes of the superpixels.

2.8.1.3 Trimap generation

After the classification of the superpixels comes the phase responsible for the generation of the trimap. The saliency map is modified to provide binary values instead of continuous values of saliency by following a threshold approach. This new binary map is then eroded and dilated. These two operations are stored in two separate maps, an eroded and a dilation map. The eroded map is then subtracted from the dilated map. This subtraction defines the unknown region, which will then be calculated by a trimap-based alpha matting network.

2.8.2 Smart Scribbles

Scribbles are also a method of trimap generation. Scribble methods always require user input (depicted in 2.29). The standard workflow for the scribbles is defined by: the user suggests where the foreground, background and unknown are located by drawing on top of the pixels accordingly. The Smart Scribbles [37] architecture takes this idea of drawing a few scribbles in an image and adds learning methods behind it to extrapolate information drawn by the user to other pixels. This work tries to reduce the overall complexity of drawing a trimap pixel by pixel while extrapolating accurate information from the image and the data given by the user.

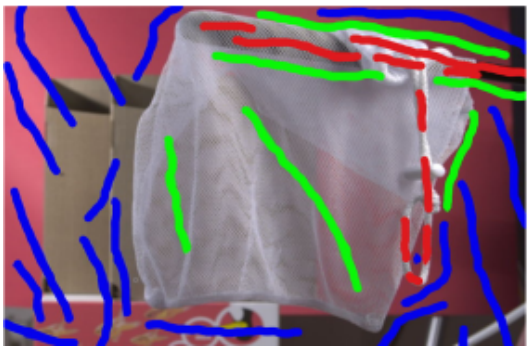

Figure 2.29: Traditional scribbles (blue for background, red for foreground and green for unknown)

This framework, similar to the Automatic Trimap Generation, starts with segmenting the image to obtain superpixels. Operating at the superpixel level helps correlate color and texture between different regions and decreases the overall computational cost. These generated superpixels pass through a region division, these regions are defined by regular rectangles that divide the image into M by M groups of values. Each region's information is defined by the superpixels that correspond to that region.

Areas of the image that contain single colors or smooth textures are often useless for drawing scribbles. The architecture can propagate the data drawn by the user to those areas. To define which regions have the most impact on the propagation of the alpha matte, a set of metrics need to be defined to reduce the user input requirements. Regions with great impact on this propagation are called *informative regions*.

As previously stated, each region has information derived from superpixels data, so from that data it is important to define which of them are informative regions. A number of criteria are defined to solve this problem:

1. High similarity with neighborhood;
2. High color and texture diversity;
3. Include foreground and background;
4. Located on a boundary of an object;

The information for a region is calculated by the following formula:

$$Info = \Upsilon + \Gamma + \Lambda + \Delta \tag{2.14}$$

where Υ represents the similarity with the region's neighbors. Γ is the color and texture diversity within the region. Λ is the foreground, background and unknown distribution in the region. Δ is the edge score that suggests the transition between objects in the image.

After the calculation of the information for each region and the definition of the informative regions, the region with the maximum information is suggested for drawing. After the user input, this information is propagated to the other superpixels, making use of local and low-level features with a re-modeled Markov Process [9] but also applying a CNN to extract global features. Following this propagation of information, a probability matrix is defined. Each element in position (i, j) represents the transfer probability between superpixel i and superpixel j. Finally, the CNN uses the auxiliary probability matrix to infer which label the remaining superpixels belong to.

2.9 Transformer-Based versus CNN-Based

Ever since deep learning techniques started being used to solve computer vision tasks, the CNN architecture as been the most common one. When the transformer model was developed and its use in the context of computer vision (described in Section 2.5.2) started, it is debatable whether this new model achieves better results compared to CNNs.

This section takes the results and the details from the ViT [6] paper and a unified set of tests and conclusions referred in [31] to compare the CNN-based architectures with the transformer-based ones.

2.9.1 Robustness

One of the main advantages of applying transformers to computer vision tasks is the robustness to out-of-distribution samples.

When faced with samples that fall out of the training scope, transformers tend to have better results when compared to CNN-based architectures.

Due to the application of the self-attention layer combined with layers such as the patch embedding layer where the image is divided into patches, transformers tend to inherently achieve high robustness as they can find relations between distant data. According to [31] it may be possible to achieve higher robustness from CNN models by using the following techniques:

1. Divide the image into patches similarly to the conversion made by transformers;
2. Apply higher kernel sizes to the convolutions to increase the scope of the features;
3. Remove the normalization layers and activation functions.

These steps not only increases the speed of training but also the robustness of the network. Applying these changes to the CNN architectures results on a higher robustness and completely outperform some transformer architectures under the same conditions.

Figure 2.30 exhibits different robustness evaluations on multiple datasets. The *Robust-ResNet* (in blue) is a CNN architecture based on the *RestNet-50* (in red) with the three aforementioned changes applied to its design. The *DeiT-Small* (in yellow) is a transformer architecture.

These results can confirm that the robustness to out-of-distribution samples is not bound to the self-attention layer, but to the capacity of extracting global features from the patches of images.

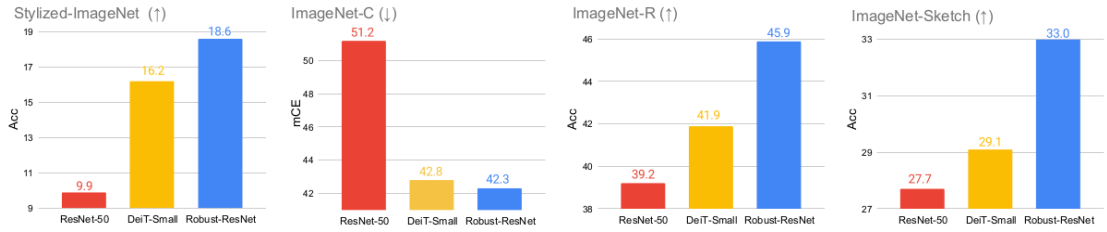

Figure 2.30: Robustness evaluation of different architectures from the paper [31]

2.9.2 Training

The previous section only compared architectures in terms of their robustness by applying the same training to all architectures. Since alpha matting datasets are difficult to generate, the training dataset size needs to be taken into consideration when choosing the architecture.

Using the ViT [6] model, a test set is run on different pre-training sizes. It appears that transformer-based architectures need a larger quantity of data for training to achieve the same accuracy as the CNN-based models (as shown in 2.31). This makes it possible to conclude that transformer-based architectures tend to overfit on smaller datasets, so the convolutional inductive bias represents a good advantage on smaller datasets to avoid overfitting.

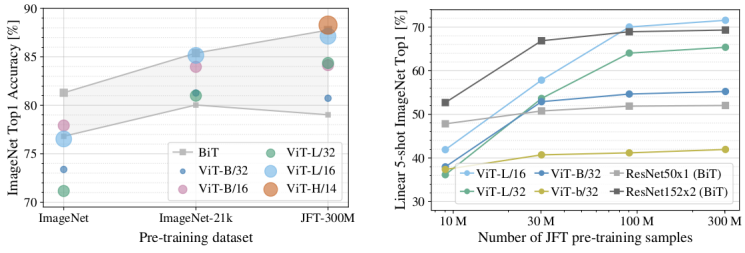

Figure 2.31: Analysis of accuracy between ResNets (*BiT*) and ViT architectures with different training sizes (taken from original paper [6])

2.9.3 Performance

Many changes can be made to the CNN architecture to improve the overall performance. The main culprit for the performance bottleneck in these types of architectures is the convolution operation applied to the images. This is one of the main advantages of the transformer-based architectures, since self-attention has a high level of parallelism, it is hardly possible to achieve the same amount of operations per second using CNNs. Nevertheless, it is possible to achieve better performances with hybrid models, unless the hybrid model increases its size and so the difference between the transformer and hybrid architectures starts to fade.

3

Sports Oriented Dataset

This chapter showcases the process of creating the proposed sports dataset to help tune alpha matting networks to sports environments.

3.1 Dataset Approaches

Alpha matting datasets are composed by multiple foreground videos and/or images, their alpha masks and backgrounds. Different algorithms may be used for the generation of this data. The core idea is to take an image with alpha values (usually in Portable Network Graphics (PNG) format) and blend those values using the previous defined formula 1.1 with a random background image.

For the video matting problem, the dataset needs to have coherence between consecutive frames so that neural networks can extract temporal and/or spatial information. For the generation of these types of videos the most standard approach is applying chroma keying, where a scene is recorded with a green background and an algorithm extracts the levels of green in the recorded video. When the greens are removed it is assumed that any remaining color belongs to the foreground layer, and so the alpha masks are generated based on those colored pixels.

There are already examples of publicly available datasets for the alpha matting training, *Composition-1K*[35], a dataset composed of 49300 training and 1000 testing images, respectively. The *VideoMatte-240K*[17] is composed of 480 green screen videos of which their alpha mattes and respective foregrounds were extracted, this dataset also comes with random images to be used as background.

3.2 Dataset Generation

The available alpha matting networks tend to be very accurate on detecting the human figure. This makes them very good for sports applications. Nevertheless, sports have a number of important information that should be considered as part of the foreground that these datasets do not include, for example a basket or soccer ball should be detected

by the network and considered as part of the foreground. This of course can extend to other sports.

Due to the lack of a sports-oriented dataset for alpha matting, a number of green screen videos were recorded, making a total of 210794 foreground frames for training which includes different sports: basket, tennis, football and volleyball. The dataset is also composed by 14397 different background images in different sports fields. These backgrounds can be used during the training or validation phase. The validation data is composed by 17847 foreground frames and their respective alphas (one video for each sport). All the remaining data is considered to be part of the training set.

The foreground videos, recorded at FHD interlaced followed a post-processing phase where each video is edited to correct some noise in the recordings. In this editing phase, videos were upscaled or downscaled to provide different resolutions.

3.2.1 Recordings

Each video (up to 2 hours long), specific for each sport is divided in small recordings of up to 5 minutes. The recordings focus on a specific layout of information, for example, in the tennis video, the player adopts different angles in each recording to guarantee uniqueness between every recording. With this workflow the dataset ends up with a set of unique angles and movements for each sport. Selecting the training, validation and testing dataset turns into an easy task when long videos are divided in multiple small recordings.

These recordings were made using an *ATEM* switcher. This equipment can change what is sent by the device at a specific output. In the selected output, this device displayed a frame composed of a set of *SMPTE* color bars. These bars are typically used for testing television signal but in this context are used as a dividing mark between recordings.

3.2.2 Frame Splitting

Since each video has various and unique recordings separated by *SMPTE* color bars, a script is applied to each video to split the dataset without manual work. This script follows this guideline:

1. Store the frames of the video until the splitter is found;
2. If the splitter was found, write the frames to disk;
3. If the next subsequent frames are still the splitter does not write any frame;
4. When the frames being read stop being the splitter, go back to step 1;

Among the many steps of the algorithm the most important one is the detection of the splitter. For that, an image of the splitter is stored in persistent storage, and it is loaded as needed. The typical algorithm for frame matching in the *Open-CV* library [2] is sliding through both images (in this case, the frame of the video and the SMPTE color bars) and

calculate the maximum likelihood (or the minimum value of the difference) between both images. The implementation used is present in 3.1, where $T(x, y)$ denotes the pixel of the template image (in this case the splitter) in the position x and y and the $I(x, y)$ represents the original image pixel in the position x and y.

$$R(x,y) = \frac{\sum_{x',y'}(T(x',y') \cdot I(x+x', y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x', y+y')^2}}$$

Figure 3.1: The normalized likelihood calculation

The expression presented in 3.1 determines, for each pixel x and y of the original image, each pixel of the template image, referred as x' and y' image is iterated and the color of the template is multiplied by the colors of the original image in the x and y position. The division normalizes the result values.

The algorithm implemented in Open-CV results in a matrix of probabilities with width equal to $W - w + 1$,being W and w the width of the original image and the width of the template, respectively. The height follows the same expression, being H the height of the original image and h the height of the template, the height of the matrix of probabilities is given by $H - h + 1$.

To check if the image in the video frame is the bar splitter, the only operation needed is checking if at any position of the matrix there are high enough values of probability so that the image is the splitter. In this particular case, checking for probability above 90% gave good results, avoiding most of the false negative and false positives.

3.2.3 Green Screen Filtering

The post-processing phase uses the green screen filtering algorithm to generate the alpha matting for the respective videos.

Starting with the color space switch from RGB to LAB, which is also composed by 3 channels: lightness, red/green intensities and blue/yellow intensities. This conversion is done following a set of formulas. Assuming that the videos are composed by 8-bit frames (which means that each channel has 8 bits), the RGB values are converted to floating point and normalized to the range 0 to 1, this normalization is simple as dividing all the values by 255. Then a weighted multiplication matrix, also referred as conversion matrix (presented in 3.2) is applied on the normalized values which then results in XYZ components. From this point, the X component represents the red/green axis, the Y represents the luminance and Z the blue/yellow axis. The X and Z components are divided by the reference white values of the color space. In the Open-CV implementation [2] it is used 0.950456 for the X_n and 1.088754 for the Z_n.

The next remaining step is to convert from XYZ to LAB color space, using the formulas presented in 3.3.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow X/X_n, \text{where} X_n = 0.950456$$

$$Z \leftarrow Z/Z_n, \text{where} Z_n = 1.088754$$

Figure 3.2: The matrix multiplication to convert RGB to XYZ values [2]

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \leq 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$

$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

Figure 3.3: The matrix multiplication to convert XYZ to LAB values

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t \leq 0.008856 \end{cases}$$

Figure 3.4: Definition of f(t) used in 3.3

The values after this conversion will be rescaled to the range of 0 to 255, keeping the same 8 bits per channel.

3.2.4 Thresholding

After the conversion of color space, the A channel is extracted and binary thresholded using Otsu's binarization. The Otsu's binarization method determines automatically the optimal threshold value for an image. This method is defined by the following formulas in 3.5.

$$\sigma_w^2(t) = q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t)$$

$$q_1(t) = \sum_{i=1}^{t} P(i) \quad \& \quad q_2(t) = \sum_{i=t+1}^{I} P(i)$$

$$\mu_1(t) = \sum_{i=1}^{t} \frac{iP(i)}{q_1(t)} \quad \& \quad \mu_2(t) = \sum_{i=t+1}^{I} \frac{iP(i)}{q_2(t)}$$

$$\sigma_1^2(t) = \sum_{i=1}^{t} [i - \mu_1(t)]^2 \frac{P(i)}{q_1(t)} \quad \& \quad \sigma_2^2(t) = \sum_{i=t+1}^{I} [i - \mu_2(t)]^2 \frac{P(i)}{q_2(t)}$$

Figure 3.5: Otsu's binarization mathematical definition

Even though the formulas appear complex, Otsu's binarization is simple. The method looks at the peaks of histograms, which relates the number of pixels to the color in the

image, finding what is the color that has the higher amount of pixels and chooses values that lie between peaks where the variance is minimal. At the end of this phase, a mask is generated but still needs refinement, according to the Open-CV documentation [2], Otsu's thresholding method achieves better results when a Gaussian blur is applied. In this case, a Gaussian blur with a small 3 by 3 kernel was applied. So after thresholding, the algorithm produces a binary mask, but this method ends up creating, in some situations, some noise in the generated masks, so a new step of refinement is needed.

3.2.5 Refinement

The subsequent refinement may take place before or during training since it consists of a number of steps to guarantee the consistency and correctness of the alpha values. The steps include applying an opening operation and blur.

The erosion and dilation operations are, similar to Gaussian Blur, a kernel based operation, which means that a kernel slides through the pixels of an image and using the values in the kernel, pixel operations are applied.

In the case of erosion, if a pixel with a 0 value inside the kernel exists, the original pixel respective to that position is changed to 0, the pixel is eroded. In the case of all pixels having 1 inside the kernel, the respective pixel stays with its value.

The dilation operation is the opposite operation of erosion, if there is at least one pixel with value 1 inside the kernel the pixel respective to the position of the kernel is changed to 1, this means the pixel was dilated. If all values are 0 the pixel stays with the zero value. The example of the Open-CV library is presented in 3.6.

Figure 3.6: Original, eroded and dilated image respectively

The opening operation is a morphological operation consisting of an erosion after a dilation of an image. The application of the opening operation reduces the noise in the alpha masks because the dilation can omit false negatives inside boundaries of the human figure. Applying the dilation makes an increase on the outline of the foreground, resulting in background colors being considered as foreground. To avoid this, the erosion is then applied to those pixels to maintain the original boundaries.

This noise cancellation technique is only possible since the generated dataset only contains strictly opaque objects (objects with alpha values of 1).

After the masks' refinement, it is possible to apply the alpha matting operation to filter the background of the raw video to store only the foreground colors of the video. Since every object is strictly opaque a channel-wise multiplication of the raw image with the alpha values is sufficient to obtain the needed foreground values. An example of the final output of these operations is present in 3.7

After all these phases the mask and the foreground video are stored in *MPEG-4* format using the *Open-CV video writer* API.

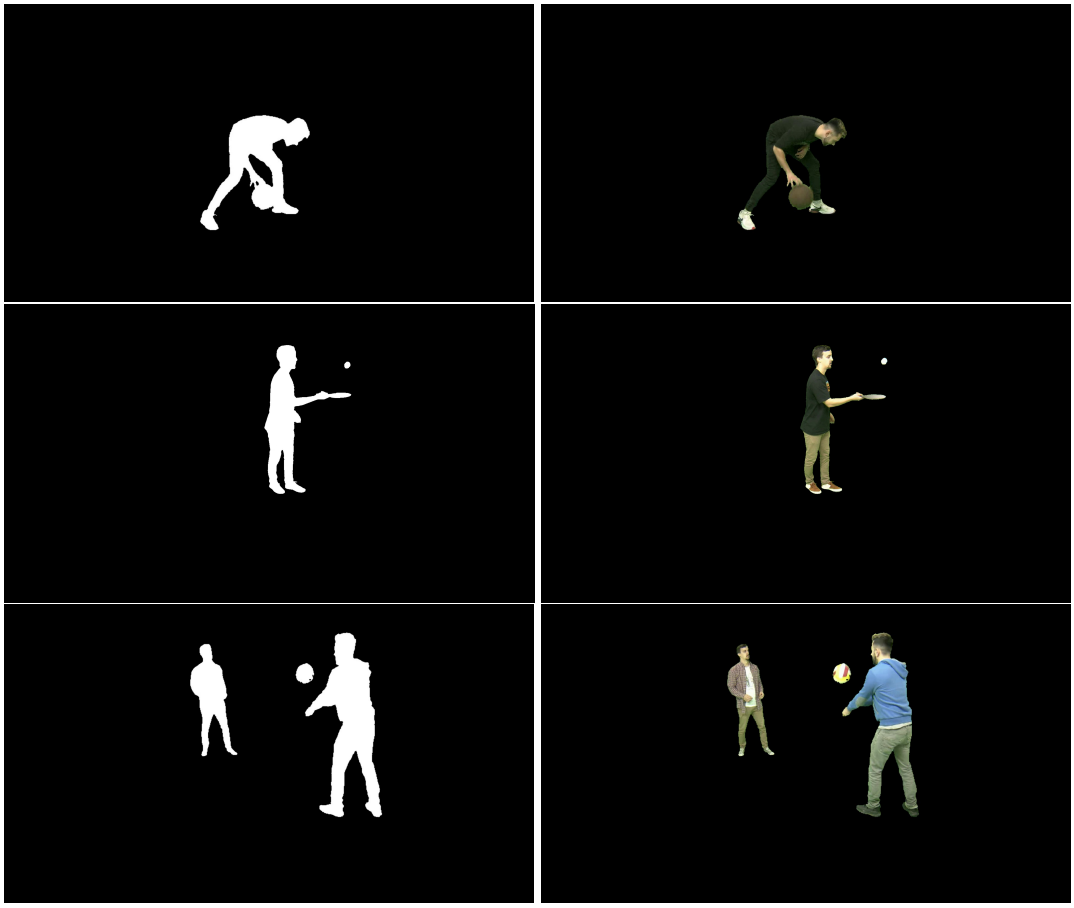

Figure 3.7: Dataset examples with the mask's final values and respective foreground

Since the dataset has alpha values tending to a binary scope, due to the objects used being very opaque and distinguished from the background, a small Gaussian blur was applied during training to add a smoother blending between the foreground and background layers.

During training the foreground and alpha images were downscaled. These operations were applied to avoid having human figures and equipments being very big when compared to the randomly selected surroundings.

During the dataset review it appeared that rescaling between 0.6 to 0.8 of the original foreground and alpha resolutions had a better and more realistic blending result. Not

because data turned more accurate but because most of the scale of the background images did not match the scale of the foreground objects. To achieve this, the foreground image and respective mask were rescaled and padded afterwards to maintain the original resolution. To avoid tampering the dataset, this operation was applied only during training. This allows for the same foreground to be used for various backgrounds, where a different foreground scale may be more appropriate. Also, if this dataset is to be used in the future, other operations are possible using the original scale. The resizing of the images was applied using a bilinear downsampling algorithm.

3.2.6 Background Recordings

As for the backgrounds, most publicly available data can also be used to improve the capacity of the network to detect sports related features. Nevertheless, a number of backgrounds were recorded and photographed to have some new additions to the dataset. Both videos and photographs taken of sports fields were composed of different resolutions to add some variety to the dataset.

3.2.7 Contextual Annotations

The presented dataset can be used in other types of training. Having that in mind, a file with contextual data about each recording was stored. The contextual information included the height at which the camera was recording, the specific sport and the angle in which the player was positioned.

For the recorded backgrounds, some information was also stored (what sport it relates to, the angle of the camera, and the height of the recordings). As concluded in the alpha matting training, where this information was queried to match some sports field to their corresponding values, it turned out irrelevant and was discarded.

3.3 Results and Discussions

The dataset provided a good amount of data to train alpha matting networks with different and dynamic backgrounds. The foreground footage, even though it has hours of content, has a small variety since it was not possible to record more videos and include more people and equipment. There are also impractical sports to record, kayaking is one of them, since it is impossible to provide the same complex environment in a green screen studio.

Another problem later arose, which is the unbalanced content. The dataset provides videos with multiple resolutions to use during training and validation, but it appears that only the tennis related videos are in UHD resolution. This can bring problems when training networks since they can overfit to this particular sport. The reason why is explained in the Architecture Fine-Tuning Chapter.

Nevertheless, at the time of writing, the dataset generated in this project is one of the biggest datasets for the video/image matting training. In the Future Work and Final Remarks Chapter other solutions are suggested to improve and grow the content of this dataset.

4

Architecture Fine-Tuning

From a number of architectures already designed and tested to solve the alpha matting problem, the purpose of this chapter is to filter and select an architecture that will be fine-tuned with the generated dataset and integrated in a broadcast final product.

4.1 The Right Solution

The matting applied in the broadcasting streaming has some particularities. First the streaming process starts with a camera producing images at a high frequency. Those images are then fed into a rendering engine that will add all the effects necessary (footers, headers and all information needed for the end user). From this point the implementation used will need to be able to read an image and try to relate previous data.

The system needs to have long-term temporal capacity to find relations between previous frames and new ones. The solution will work with frames coming in a high frequency from 30 to 60 Frames Per Second (FPS). This means that the rendering pipeline will need to dispatch previous frames from memory in order to read incoming ones, so the temporal capacity should not come with the need of having previous frames loaded in memory, since managing previous stored frames could cause a bottleneck.

From this point, multiple architectures were studied and tested, from inference speeds to temporal context capacity. For the empirical testing of the networks, a video was recorded and, frame by frame, results were compared. Also, multiple resolutions, from FHD to UHD, were tested.

4.1.1 Research

As seen in the Background and Related Work Chapter, multiple public solutions for solving the alpha matting problem exist. From sampling methods to neural network solutions, determining the best methods demands testing some examples and analyse how well a solution delivers its values, not only the performance but also the results.

4.1.1.1 Sampling versus Neural Networks

In the table 4.1.1.1 it is possible to analyse the overall SAD error for multiple available alpha matting solutions The SAD error calculates the difference between the predicted image matting and the ground truth matting.

Architecture names	Overall SAD errors
TMFNet	3.4
LFPNet	4.8
Deep Image Matting	18.5
DCNN Matting	23.1
Shared Matting	41.3
Global Sampling Matting	43.8
KNN Matting	46.2

Table 4.1: SAD values from `alphamatting.com`

In table 4.1.1.1, *KNN Matting* [4], *Global Sampling Matting* [12] and *Shared Sampling* [10] have higher SAD errors. As for neural network solutions like the *TMFNet* [40], they achieve a SAD error as low as 3.4, about 12 times less than the best sampling method in the table. This leads to the conclusion of going forward with a neural network solution, has it appeared more robust when predicting alpha values of images.

4.1.1.2 The State-of-Art Models

As exposed in the Background and Related Work Chapter, multiple architectures are available to be tested. Architectures directed at solving the image matting based on an image and its respective trimap and others solve the image matting without any extra input besides the image. There are also a number of architectures designed to work on video matting, that take a sequence of in-memory images and generate their respective mattes with the help of temporal information.

From the multiple architectures available, the ones with smaller SAD errors tend to have auxiliary input to help the matting problem, like the *TMFNet* [40] in which depth information is required. Also, most of the architectures are not projected to work on a high performance environment where images are fed to the network at high frame rates.

Starting with trimap-based architectures, these networks have accurate mattes, but the problem lies in their input requirements. In a broadcast environment the image is generated by an external device, typically a camera broadcasting to other devices. From these external devices, images are forwarded to a rendering engine to process the images, as can be expected, there is no possible step to provide a trimap to the network. This impossibility in real-time makes these networks invalid in these contexts.

Nevertheless, there are architectures that can provide results without any type of trimap input. Here three are referenced: PP-Matting, Robust Video Matting and VM-Former. PP-Matting is an architecture designed to solve the natural image matting problem

without any type of extra input, even though this network is able to solve the image matting problem, it is incapable of providing some important functionality.

Typically, architectures similar to PP-Matting which are designed to solve a single image matting tend to have great results but have slower inference speeds. There are more downsides to these architectures, for example, they are not capable of storing temporal information from a sequence of images. This feature is important in this context where the architecture will be solely applied to videos.

For the purpose of matting videos, the valid architectures are reduced to Robust Video Matting and VM-Former.

4.1.2 RVM versus VM-Former

Both architectures are designed to solve the video matting problem. The RVM is CNN based, VM-Former has a CNN backbone for feature extraction followed by layers based on existing transformer encoders and decoders.

Both architectures have temporal capacity and good results for the alpha mattes in the video used during the test. The problem is that, at the same resolution both architectures have a big difference in inference speeds. To confirm these facts, a series of tests were performed.

4.1.3 Information about the tests

4.1.3.1 Hardware used

- Operating System: Windows 11 Pro
- GPU: *NVIDIA RTX 3070 Ti*
- CPU: *12th Gen Intel(R) Core(TM) i7-12700H*
- System's Random Access Memory (RAM): 32 GB
- Dedicated Video-RAM: 8 GB

The manufacturer's stock cooling system was used. This system was deliberately chosen because it matches common hardware used by designers during a broadcast service. If omitted, tests in this and next chapters are executed with the same hardware.

The framework used in this context was the PyTorch 2.x framework. Both models were put into evaluation mode where some layers, for example Batch Normalization, were disabled. The model was also accelerated using the CUDA API 11.8. The gradient calculation was also disabled to reduce memory consumption while inferencing.

Note that, for the tests in 4.1 and 4.1.3.1 no refiner or upsampler technology was used to match the input resolution. The tests were made guaranteeing that the full image was used throughout the networks and their output matches the input resolution.

The test 4.1 was made with the previous referred system with increasing input sizes. With this test it is possible to analyse how the networks' performance varies with the increase of the image size.

As for the 4.1.3.1 the image was kept at a low resolution of 640 per 360 pixels, and was horizontally increased. This means that the number of images fed into the network was gradually increased to evaluate the capacity of processing and relate a variable number of images at the same time.

The inference times were extracted executing 5000 iterations with the respective input. At the end of the set of the iterations the time was averaged and rounded to the lowest number.

Both models have different version sizes, in the case of VM-Former the architecture used has fewer parameters and is trained to work with typically 512 by 288 resolutions. As for the RVM, the architecture has two versions which only differ in the utilized backbone, being them RestNet or MobileNetV3. The *MobileNetV3* was the backbone used in these tests.

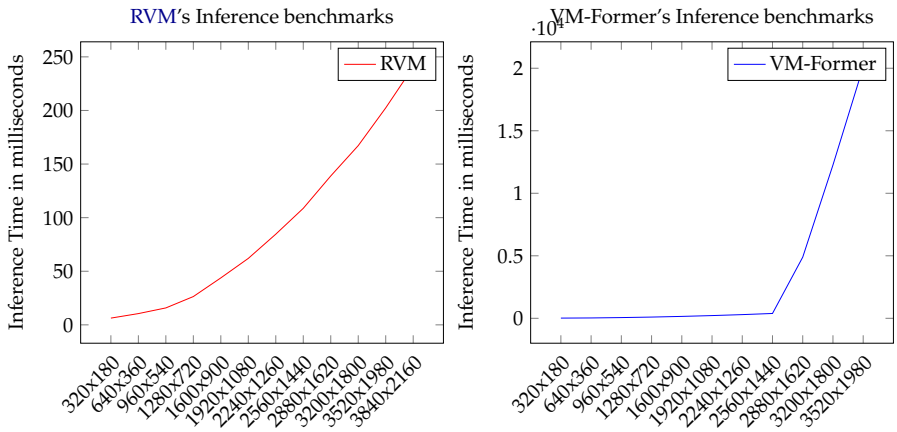

Figure 4.1: Tests made to RVM and VM-Former with multiple input resolutions

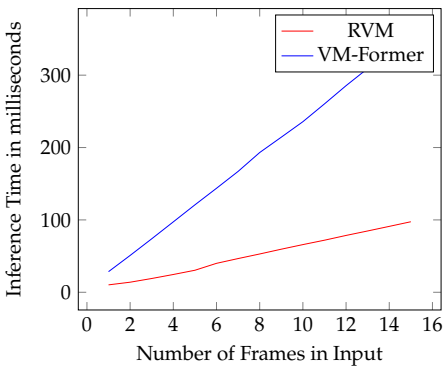

Figure 4.2: Test with increasing batch size for each of the networks

Figure 4.3: Comparison between masks generated by RVM (left) and VM-Former (right)

4.1.4 Notes on VM-Former

From the values depicted in 4.1 and 4.1.3.1 the VM-Former can achieve high performances, but the original images need to be in a lower resolution than the RVM to achieve the required speeds of 60 FPS (16 milliseconds per image). In higher resolutions the architecture took around 2 days to execute all the tests required.

The architecture went as far as achieving thermal throttling in the GPU. This thermal throttling was achieved when reaching images of 2880 by 1620 pixels.

From the tests it is possible to assume that the network has good capacity for alpha matting but only at very low resolutions, so, before entering the network the image needs to lose a lot of data with downscaling operation. The vertical scaling capability (increasing resolutions) of the network is very low, since it tends to exponentially need more computation time.

The use of the GPU tends to be very high. This is so because the typical transformer architecture makes use of a MHSA block. MHSA has very high parallelism, so it is possible to extract more computing power from the GPU with the possibility of scaling with the capacity of the hardware.

Even so, the architecture can achieve usable performances for the real-time mark when in low resolution, but it does not provide the full solution for the alpha matting problem. The structure of the network is designed to provide the alpha masks without any foreground colors. This makes it impossible to correctly blend semi-transparent objects in the scene. As such, when using the VM-Former it is only possible to extract the correct blending from strictly opaque foreground objects.

Also, as referenced before, the tests were made without any type of refiner to match

the output of the network to the input. This is to guarantee that throughout the networks all the image data is processed, but also to even out the tests between networks, since VM-Former does not provide the capacity of upsampling or downsampling the input image, so the output will always match the input.

4.1.5 Notes on RVM

The RVM appears to have smaller inference times overall compared to VM-Former, apparently because it uses operations of typical CNNs and VM-Former uses standard MHSA block which end up being a bottleneck in terms of inference times due to its quadratic temporal complexity.

As for the alpha matting values, RVM has good empirical results overall, as seen in the figure 4.3. The network matches the overall human figure, having a hard time making a good boundary between the foreground and background.

Contrary to the VM-Former, RVM gives all the data needed to make a perfect blending of semi-opaque objects, the output of the network is composed by two images, one with the alpha values and another with the RGB colors of the foreground layer. So, it is possible from this data to achieve the background colors using the following formula:

$$b = (src - \alpha * fgr)/(1 - \alpha) \tag{4.1}$$

Where b is the RGB values of the background color, src is the source image given as input to the network, fgr the RGB values of the foreground color, given by the network, and α the alpha values also generated by the network.

As noted in the Robust Video Matting section, the RVM model comes with an integrated refiner. This refiner is only used when the network downsamples the input image to work with smaller resolutions. The downsampling is a tunable parameter tweaked by the user of the network, but independently of how much the network downsamples the input image at the beginning, the refiner will always match the original input resolution. Using this parameter it is possible to achieve higher performances with smaller resolutions since at the end it is possible to extract a high resolution version of the predicted values.

One of the main criteria needed for using a neural network in real-time during broadcast is the high performance and stability of the network more than its results. Having an end-user with a high frame stutter is the most unwanted problem. For that matter, with the capacity of matching high resolution input while using low resolution predictions and the ability of blending semi-opaque objects, RVM was the selected network to be integrated in the system since it outperforms VM-Former.

4.2 TensorFlow versus PyTorch

The RVM architecture has two available versions for final deployment using different frameworks: the *TensorFlow* and *PyTorch*.

Image Resolution	TensorFlow-GPU	PyTorch-GPU
1920x1080	120.1 ms	62.0 ms
3840x2160	453.4 ms	240.6 ms

Table 4.2: Inference time in milliseconds of RVM in different frameworks

Both frameworks provide generic wrappers to run AI models using the C++ language without needing a Python environment to integrate them. Nevertheless, both frameworks have big differences between them.

An important note to make also about TensorFlow is that, its native GPU execution in the Windows operating system is no longer supported. The last version with native support was version 2.10. The only possible counter-measure to solve this problem was running the network in a Linux container and writing to a mounted volume. To measure the GPU times of TensorFlow the official container was used, and the overhead of running the model inside a container, if any, was totally ignored.

From the table in 4.2, it is possible to conclude that RVM has a higher performance using the PyTorch framework. As to the reason why, it appears that, during inference, TensorFlow automatically transfers the results of the network from GPU to CPU. Making this transfer of information between a device and the host causes a bottleneck during inference. Due to these facts the TensorFlow framework was discarded.

4.3 Architecture Training

The RVM was previously trained by the authors to detect the human figure with an incredible accuracy, detecting the alphas and the colors of the human subject, with the capacity of detecting their reflections in the environment.

As can be seen in 4.3, the architecture does not have the capability to detect gym equipments like the kettlebell. This also occurs in other sports, where balls and rackets are not detected as foreground. This is due to the prioritization of the human figure during the training of the architecture.

The main objective of this fine-tuning is to make the neural network have a higher accuracy and better alpha mattes in the context of sports where it can have features like a ball present as part of the foreground layer.

To fine-tune the architecture the custom dataset referred in Sports Oriented Dataset Chapter was used. This fine-tuning follows the authors' training procedure.

4.3.1 Losses

In each step, multiple losses are calculated to determine the gradient and execute a backward pass to update the weights of the network.

4.3.1.1 L1 Loss

The typical loss used during training is the *L1* loss. Provided by the PyTorch framework, this loss measures the mean absolute error between the predicted data and the ground truth data. This loss may have different reduction types, for example, the mean reduction, where the difference between the predicted values and the ground truth per pixel in the image is given, and the mean value is calculated based on those differences. Following the PyTorch documentation the L1 loss is implemented following the formula in 4.4.

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = |x_n - y_n|$$

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

Figure 4.4: The L1 Loss formula used by the PyTorch framework

In 4.4 the N value is the batch size, x is the predicted values generated by the network and y the ground truth values of the dataset. In the training loss formula 4.6, this loss is referred to as $\mathcal{L}_{L1}$

4.3.1.2 Temporal Coherence Loss

The *temporal coherence* loss tries to force the network to not make big changes in consecutive frames with the purpose of decreasing the amount of flickering and to make the data consistent throughout a video.

To calculate this loss, the following mathematical formula is followed:

$$\mathcal{L}_c(x, y) = \frac{\partial x}{\partial t} - \frac{\partial y}{\partial t} \tag{4.2}$$

In this equation the x are the predicted values, the y is the ground truth data and dt refers to the temporal change of each set of values. So, basically the temporal coherence loss is defined by the difference between the temporal change of the predicted values and the temporal change of the ground truth values.

The coherence loss is calculated from the batch during training in PyTorch. First it is determined the difference between each image in the predicted values x. The same operation is applied to the images in the ground truth y. From those differences an *MSE* loss is applied. The following snippet of code is the "pythonic" approach using the integrated loss Mean Squared Error (MSE) loss provided by PyTorch to calculate a value for the temporal coherence loss.

Listing 4.1: Temporal coherence loss implementation

```
1 from torch.nn import functional as F
2 # x is the predicted values
3 # y is the ground truth values
4 # Assuming the shape (B,C,H,W)
```

```
dx = x[1:] - x[:-1]
dy = y[1:] - y[:-1]
coherence_loss = F.mse_loss(dx,du)
```

4.3.1.3 Pyramid Laplacian Loss

Laplacian is an operation used to find edges on an image. This operation has the ability to define the orientation of edges based on their gradient.

The Laplacian operator is a derivative operator, it extracts the gradients of the image at each pixel. In mathematical terms, Laplacian can be applied as 4.3 for any function f with a number i of x variables.

$$\Delta f = \sum_i \frac{d^2 f}{dx_i^2} \tag{4.3}$$

For images, a Laplacian filter is used to approximate the Laplacian operator. This Laplacian filter is defined as: $M = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

The *Laplacian* loss is calculated by applying a convolution operation (referred as D) with the previously referred filter to images and calculating the mean-squared differences of those Laplacians. The formula for a typical Laplacian loss is as follows:

$$L_{lap}(x, y) = \sum_{ij} (D(x) - D(y))_{ij}^2 \tag{4.4}$$

This last equation is applied at each pixel ij, being D the application of the convolution operation with the Laplacian filter and x and y two arbitrary images, in this case, x are the predicted values of the network and y the ground truth values.

From the concept of the Laplacian loss, *Pyramid Laplacian* loss is the application of Laplacian loss extended to different scales of the image. By creating a Gaussian pyramid using down sampling and up sampling operations to images. Then, using adjacent levels and calculating their difference the result is a low-frequency residual of the image corresponding to that Gaussian level. This process of down sampling, up sampling and generating Laplacian pyramids from a Gaussian Pyramid is depicted in 4.5.

Based on the definition of a Laplacian pyramid, the Pyramid Laplacian loss is defined by the sum of differences between each level of the pyramid. The following equation defines the Pyramid Laplacian loss:

$$\mathcal{L}_{lap}(x, y) = \sum_{i=1}^{l} 2^{i-1} \left\| L^i(x) - L^i(y) \right\|_1 \tag{4.5}$$

In this equation l represents the levels of the Laplacian pyramid, L^i is the Laplacian of an image at level i and the x and y represents the predicted values and the ground truth values as all the previous losses. For this loss, the "pythonic" approach was achieved by using the code in 4.2.

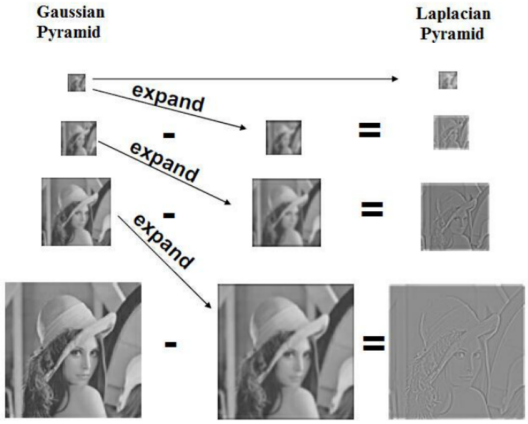

Figure 4.5: Image taken from [8] that relates the Gaussian pyramid to the Laplacian pyramid

Listing 4.2: Laplacian loss implementation

```
from torch.nn import functional as F
# levels is the number of levels present
# in the laplacian pyramid
# kernel the Laplacian Filter used in the Gaussian pyramid
# laplacian_pyramid is function that generates the Laplacian
# pyramid for each image according to the number of levels

pred_pyramid = laplacian_pyramid(x, kernel, levels)
true_pyramid = laplacian_pyramid(y, kernel, levels)

for level in range(levels):
    loss_level = (2 ** level)
    loss_level *= F.l1_loss(pred_pyramid[level],
                            true_pyramid[level])
    loss += loss_level
loss /= levels
```

The *laplacian_pyramid* function presented in the Python code is presented in A.1.

4.3.1.4 Training loss

The training loss is composed by the previous referred losses. This composition is made from the summation of all the losses.

$$\mathcal{L}_t = (\mathcal{L}_{lap}(\alpha, \alpha_{gt}) + \mathcal{L}_c(\alpha, \alpha_{gt}) * 5 + \mathcal{L}_c(fgr, fgr_{gt}) * 5 + \mathcal{L}_{L1}(\alpha, \alpha_{gt}) + \mathcal{L}_{L1}(fgr, fgr_{gt}))/5 \tag{4.6}$$

The α refers to the predicted alpha values and α_{gt} is the ground truth provided by the dataset. The foreground values are referred to as fgr and fgr_{gt} for the predicted and ground truth values respectively. The total loss is then averaged by the total number of losses used, in this case the total loss will be divided by 5.

4.3.2 Augmentations

It is expected that makes the network overfit to the data provided. Augmentations should be applied to add diversity to the training values seen by the network. All the augmentations that will be referred were applied to the foreground layer and the respective backgrounds. For the alpha masks, the only augmentations applied are affine transformations (augmentations that change the color of the pixels are not applied as to not compromise the ground truth data).

Since the train is made using video with sequential data, augmentations should not create abrupt and unrealistic changes in sequential images. To avoid creating absurd augmentations, **motion transformations** take all the images in a batch and interpolates a value to achieve a smooth and gradual augmentation throughout the sequence of frames.

These motion transformations include gradually changing brightness, contrast, saturation and hue of an image, as with the linear interpolation of rotation, translation and scaling. An important operation also present in videos is the motion blur. To add to the motion blur already present in the dataset generated, motion blur was interpolated among the batched frames with a Gaussian filter.

The motion augmentations between the batched frames were applied using mostly linear interpolations. Using a rotation augmentation as an example, the augmentation defines two angles of rotation, the first and the final angle. This can be done using a typical pseudo-random function that can be limited to avoid giving big values and unviable angle values. With a first and final angle, the rotation value for an image at the i^{th} position of the batch is calculated by the equation 4.7.

$$rot^{i}_{img} = f * A + (1 - f)B \tag{4.7}$$

Where A is the first angle and B is the final angle. f is the interpolation factor which has 0 value for the first image in the batch and 1 for the final image.

It is using this formula that all the motion augmentations are applied along a set of frames present in a batch.

4.3.3 Training Step

The model was executed for 13 epochs, each epoch iterates through all the videos correspondent to the training phase. Each training step is composed by two steps, the low-resolution pass and the high-resolution pass. The low-resolution pass is achieved by down sampling the input image when entering the network. Then, when returning the alpha matte results, the predicted values enter the refiner after the decoding phase alongside the input image to match the input resolution. The down sampling ratio was dynamically changed according to the resolution of the input image as to avoid OOM errors due to high batch sizes.

In the high-resolution pass there is no down sampling of the image, similar to the previous tests made. The image is used in its full resolution throughout the network, and

the results from the decoder match the input resolution, so there is no pass in the refiner stage.

The model provided by the authors had already good alpha matting results, as depicted in 4.3. The network was trained with segmentation datasets alongside matting datasets and used a pre-trained MobileNetV3 architecture with a high capacity of feature extraction.

With pre-train applied to this network, the main objective of this training is to provide better results when it comes to the context of sports. In this context multiple players can be on-screen with their respective equipment and other types of elements can be added as belonging to the foreground layer.

To maintain the already high capacity of the network, the typical approach to fine-tune the model. The fine-tuning is based on freezing the bottom layers, typically the encoders, and unfreeze top layers. To freeze a layer is to stop its weights from being updated during the backward pass. With this procedure the models can extract the same generic features in the backbone, but in the decoder phase it can change the relevance of each feature to match the specific task.

In RVM, 3 parts were frozen: the backbone network, the ASPP module and the projection layer's segmentation part. So the decoder, the projection responsible for generating the alpha matte, and the refiner were trained with the new data. In this way it is possible to maintain the robustness already provided by the network and use the same architecture for the specific task of alpha matting in the sports context.

All the data was read from the video files provided by the sports-oriented dataset during training. When reading the dataset some operations were made to guarantee that the alpha values were not binary and had continuous values in the range of 0 and 1 in the boundary of the foreground and background. The reading of the videos and images were executed using the OpenCV API [2]. One advantage of having the data stored in video format is the smaller size of the dataset in permanent memory, but results in higher CPU usage during training since it needs to decode the video making its time longer.

4.4 Validation Results and Discussions

During the validation step the model was changed to evaluation mode to disable Batch Normalization and Dropout layers. No augmentations were applied to the dataset images, only the blending was applied to merge the foreground with the background according to the ground truth alpha mattes to generate the input image.

At each iteration over the validation data, the dataset is fed to the network in the same order with the purpose of having a side-by-side comparison between each trained model.

The validation errors are displayed in 4.4. It is noticeable that the original weights of RVM displayed a good overall performance, but it was possible to improve it by fine-tuning the model using the new generated sports dataset.

The training stopped at epoch 13 because the loss values of the validation data started to converge and, the following training epochs started to change very slightly.

Training Epoch	Validation Loss
Original	0.02983
Epoch 12	0.00776
Epoch 13	0.00813

Table 4.3: The validation loss of the RVM training

One error made throughout the training process was the use of the non-normalized losses that do not take into consideration the total number of pixels. This meant that, for UHD videos, the architecture converged faster than the FHD videos. This was even more pronounced because the dataset is unbalanced as stated in 3.

Contrary to the validation results, the empirical results on generic images were worsened. The newly generated dataset hadn't sufficient variety of data to fine-tune the model culminated with an overfit to the data provided.

Figure 4.6: Empirical results of the fine-tuned RVM after epoch 13 (left) and original model (right)

5

The Search for Transformers

In this chapter, it will be discussed how and why a new custom model was implemented, which parts were tested, improved and how the solution works in terms of inference speeds. The work on this architecture will continue in the 7 Chapter. Note that the work on this project is still in progress at the time of writing, but the research made in this field may contribute to implement faster transformers that can be used in real-time during broadcasting and streaming.

5.1 Motivation

Up to this point, a number of architectures exist for the image matting problem. Most of the networks achieve good results, mostly because in the image matting context there is no performance concern, so the architecture of the network can be as deep as possible to achieve higher robustness and better results.

In terms of video matting, the available architectures are very reduced, and most of them are designed for post-process integration, where multiple frames can be loaded in memory and the network works on those frames without worrying about inference speeds.

At the time of writing, the only architectures that can be used in a real-time environment, while keeping an acceptable frame rate, are the Robust Video Matting and VM-Former, as discussed in 4.1.

With this lack of alternatives in the video matting world, this chapter aims to provide a new custom architecture. This architecture will be designed to fulfill all the performance requirements while achieving new state-of-the-art results.

5.2 The Base Architecture

The RVM architecture uses a CNN backbone for feature extraction alongside a decoder to extract the alpha mattes from those features. This same structure is applied to the VM-Former. This architecture uses a CNN backbone to extract features and uses another

encoder and decoder composed by transformers to produce an alpha matte. Both architectures have many similarities, but none of them is designed for using any other method for feature extraction besides CNN based backbones.

Our base architecture takes the same backbone concept and applies it as an encoder. The main difference of this backbone is that, instead of being composed by convolution operations, the backbone has transformers inserted between convolutions to extract more features from an image. The decoder of the architecture is based on the decoder provided by RVM with a single modification to support a bigger number of features from the backbone. The refiner is also a matter for research, which will be referenced in the following sections.

5.2.1 Backbone

As referred, typically the backbone has an exclusive CNN-based structure, but recently the search for faster transformers has been improving. In the Background and Related Work Chapter two transformer architectures and their respective transformer blocks were explained, the MobileViTv2 and Swiftformer, both architectures try to be as close as possible to the CNN backbones, like the MobileNetV3 used in RVM. Even though it cannot achieve the same inference speed as this architecture is very similar. In the table 5.1 the inference speeds of the Swiftformer-XS and MobileViTv2-1.0 encoders operating with FHD and UHD images are presented and compared with the MobileNetV3 encoder of RVM.

Encoders	1920x1080	3840x2160
RVM Encoder	18.3 ms	70.1 ms
Swiftformer-XS Encoder	20.0 ms	75.2 ms
MobileViTv2-1.0 Encoder	71.8 ms	278.9 ms

Table 5.1: The performance of transformer and CNN encoders in specific resolutions

The tests were performed by extracting the respective encoders from the raw models made available through the *HuggingFace* community. Each of the encoders were split into multiple feature levels to integrate directly into the RVM decoder. An important note to take is that the MobileViTv2 encoder extracts more features than the Swiftformer-XS in this configuration. MobileViTv2 extracts 4 feature levels, these feature levels are composed of 64,128,256 and 512 features respectively. As for the Swiftformer-XS encoder, it still extracts 4 feature levels but composed of 48, 56, 112 and 220 features.

With this data it is proven that it is possible to use transformers as part of a backbone while keeping good overall performance, of course, with more restrictions in terms of input resolution.

From the tests provided, between the different transformers tested, the Swiftformer-XS hybrid encoder apparently has the best performance in terms of inference speeds, which ended being the better choice to integrate in the backbone of the network.

One advantage of having a hybrid encoder is, as referred in Transformer-Based versus CNN-Based Section, smaller hybrid networks tend to have better robustness than a fully CNN or transformer-based architecture. The execution environment favors smaller networks as the solution to achieve higher performances, since they deliver a good trade-off between results and speeds.

5.2.2 Refiner

As seen in the RVM architecture, the use of the Deep Guided Filter provides good overall results with some drawbacks regarding the object's boundaries. The SparseMat refiner claims to provide a better solution for the image upscaling process.

The SparseMat architecture at the time of integrating and testing in the created architecture was deprecated. To solve this, the architecture went through a reimplementation to be integrated with new versions of the CUDA API. With this new re-implementation, the refiner showed an overall bad performance taking up almost 80% of the total inference time and up to 2 seconds to execute. Due to this bad real-time performance the Deep Guided Filter architecture was selected to be integrated in the new model.

5.2.2.1 Temporal Refiner

Since the Deep Guided Filter reached the required inference speeds, a new refiner based on this architecture was created. This refiner is capable of acquiring temporal information throughout the inference phase. This refiner was created to check the improvements of upscaling a sequence of images with the aggregating features of previous ones. With this new functionality the temporal refiner still reaches similar speeds to Deep Guided Filter.

To implement the aggregation of temporal data similar to Long-Short Term Memory (LSTM), the refiner takes the Deep Guided Filter architecture and instead of applying the typical convolution layers, uses ConvGRU to aggregate hidden states throughout the inference.

5.3 Architecture Training

The architecture's training is based on the RVM training. It is designed to have the capacity of inferring on segmentation data as an initial step to locate and define the human figure in a variety of images.

5.3.1 Datasets

The training is programmed to have the capacity of reading a number of publicly available datasets. Starting with *YouTube-VIS* segmentation dataset [36], composed of 2883 YouTube videos, 2238 training videos, 304 validation videos and 343 test videos, all composed of a different number of objects including humans.

The *COCO* dataset is also integrated in the training, and is composed of 123287 images with their respective segmentations, from which 66808 have human figures in different contexts.

Continuing with segmentation datasets, one of the more frequently used ones is the *ADE20k* dataset, containing 25574 images for training and 2000 images for validation. This dataset is one of the biggest ones available to use, where not only people and objects are segmented, but also parts of the same objects are defined as a different segmentation.

In terms of matting datasets, the *VideoMatte240K* provided by the authors of RVM, consists of 484 videos, being 5 of them used for validation purposes, which translates to 240709 different images with their corresponding foregrounds and alpha values. The custom dataset described in Chapter 3 was also integrated in the training steps.

5.3.2 Losses

Most of the losses used in the custom model training are similar to the ones used in the fine-tuning of RVM model, with one new loss: *Composition Loss*.

The *Composition loss* function computes the original source image from the predicted matting values. The formula for the composition loss is as follows:

$$comp_{pred} = fgr_{pred} + (1 - \alpha_{pred}) * bgr_{gt} \tag{5.1}$$

$$com_{gt} = fgr_{gt} + (1 - \alpha_{gt}) * bgr_{gt} \tag{5.2}$$

$$comp_{loss} = \ell(com_{pred}, com_{gt}) \tag{5.3}$$

In the previous equations, fgr_{pred} represents the predicted color values of the foreground, the bgr_{gt} is the ground truth background values, fgr_{gt} is the ground truth foreground colors and ℓ is the L1 loss function.

5.4 Results and Discussions

This chapter dealt with the research and definition of a brand-new custom architecture with a hybrid structure to achieve state-of-the-art alpha matting results while trying to maintain the high performance criteria. First, a hybrid backbone was integrated for feature extraction. With these backbones it is possible to extract more features from an input image.

For the decoder part, a modified version of the standard RVM decoder was used in order to support more features from the backbone. In the upsampling part, a temporal refiner was created to test how a refiner capable of aggregating temporal information can improve the alpha matte results in the final output.

Nevertheless, this new architecture, at the time of writing, was not tested in terms of results besides the inference speeds with random images. But a training is in course and the network is already capable of predicting some alpha values in images.

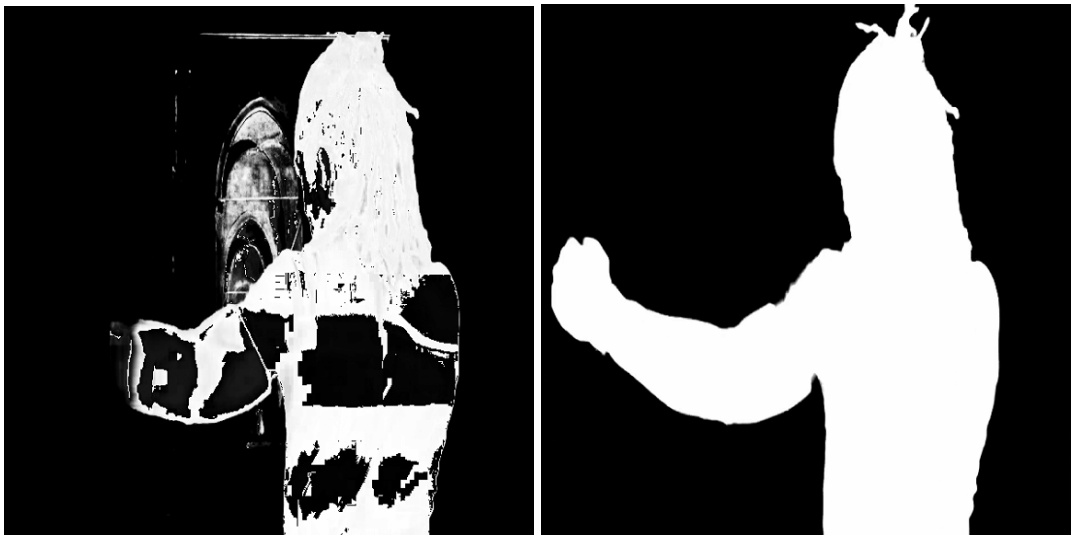

Figure 5.1: Example of the intermediate results of the custom model's training using the Swiftformer encoder

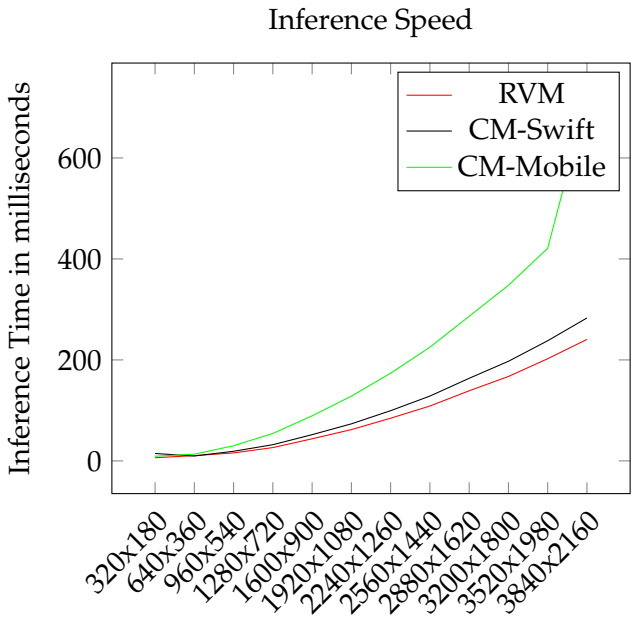

Figure 5.2: Inference times of different models

As presented in 5.2, *CM* refers to the custom model created and its respective backbone used as feature extractor, which can be the Swiftformer-XS or the MobileViTv2 encoders.

The versions of the RVM are the same as the previous tests in 4.1. And similarly to the previous tests, no refiner was applied during inference, so the input resolutions were used throughout the network.

Comparing the custom model with the architecture of RVM, it was possible to note that, at the same resolution, the custom network not only is slower, but also the utilization of the GPU is higher, achieving close to 100% of the NVIDIA RTX 3070 Ti capacity. Nevertheless, with the same computational resources the custom models, outperformed,

in terms of inference speeds, the VM-Former architecture, and it is configured to support semi-transparent objects in the scene.

6

Rendering Integration

In this chapter the focus is on the total integration of RVM in a rendering pipeline supported by the DirectX technology and on the post-train improvements of the inference time.

6.1 Deployment Environment

6.1.1 Rendering APIs

Every render engine needs to be supported by a API capable of processing and drawing images into the display. These types of APIs typically require a set of resources, and it is common practice to have them stored in buffers.

The buffers of resources can be made of vertex data with their respective transformations but also images in a specific format to accelerate the indexing of RGB values. Each API has its respective order of operations to render images into a frame buffer. The frame buffer usually exists in all rendering APIs, and it is where the final rendered frame will be written to.

As seen in 6.1 the programmable rendering pipeline is composed by multiple stages. Typically, these stages can run functions, called shaders, written by the programmer to process and correctly render the data to the frame buffer.

In the used rendering engine, the flow of image processing is as follows:

1. A camera or an external device generates an image;
2. The image is fed into a broadcasting board;
3. The broadcasting board feeds the image data into the engine;
4. The engine uploads the image to the GPU using a platform and API dependent implementation;
5. The rendering API uses the image as a resource, reading and writing to create effects on top of the raw image;

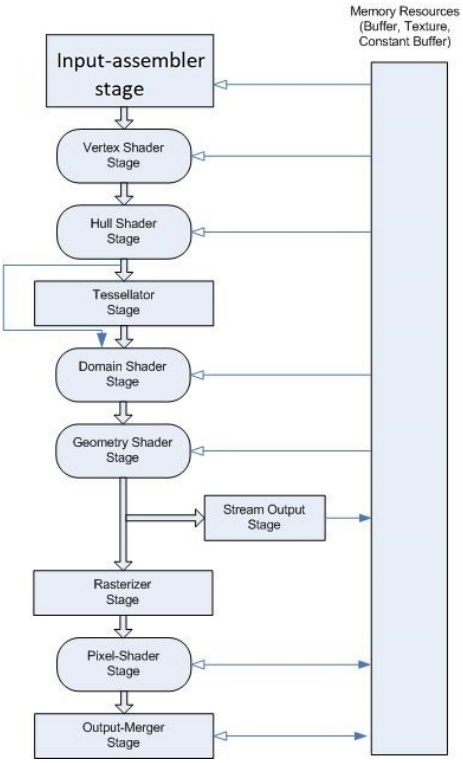

Figure 6.1: The rendering pipeline offered by *DirectX* 11

6. The engine uploads this data to the output channel of the broadcasting board;

6.1.2 TorchScript Compiler

Typically, when a neural network is intended for use in commercial applications, the entire environment of execution needs to be considered. Up to this point, all the code and tests made with the network were executed using the PyTorch framework in a Python environment. In real-time applications, the network will be executed using the C# language with the .NET framework. It is a possibility that the available system has no integrated Python interpreter, and it may be the case that it is not feasible to execute it using the Python compiler.

From this necessity of having a form of execution of the model that is easily integrated with a generic environment, the Torch community developed a generic language that can execute in almost every environment. This language is called TorchScript.

TorchScript works by doing a forward pass of the network. A forward pass is an iteration throughout the network from the input to the output. During this forward pass the compiler generates a graph of operations. The graph of operations stores the weights of each layer, all the operations and the order that they are executed, including operations that are inside control flows (if statements).

From the graph of operations, the TorchScript Just-In-Time (JIT) compiler analyses the operations and tries to fuse them in kernels. A kernel in this context is a function, typically developed with low-level languages, that is written to run inside the GPU. The fusing of

operations tends to increase the speed of a model since there is some added overhead on launching multiple kernels. Nevertheless, there are operations that cannot be fused and those require distinct kernels.

To create this TorchScript version of the RVM model, PyTorch 2.x provides a method `torch.jit.script(<model>)`, where `<model>` is the network composed by `nn.Modules`. The `torch.jit.script(<model>)` returns a scripted model optimized using the JIT-compiler.

At first, the scripted model tends to have a slower start, even slower than the raw implementation in PyTorch but while iterating through the network, the compiler fuses kernels and optimizes operations until it reaches the maximum speed that it can achieve. The number of iterations necessary to achieve the highest speed-ups are different between models. The performance speeds of the scripted model are presented in 6.4.

The scripted version accelerated by the compiler provided by PyTorch accelerates the model. This acceleration is even more noticeable when reaching high resolution images where speed-ups can achieve 1.32, which means that the new scripted model can achieve as much as 30% more inference speed.

At the end of this step the model is compiled in an agnostic language that can be run without the need of a Python interpreter. Nevertheless, to run this model the Torch API is still needed. From this point onward the use of the network outside Python is supported with the Torch C++ implementation referenced as *LibTorch*.

6.2 Rendering Pipeline

In any type of network, mainly in networks that are deep and have many image operations, like the CNNs, their inference should be executed using a highly parallel implementation accelerated by hardware, typically a GPU. In 6.2 it is possible to compare the inference times for specific resolutions using the model on a CPU and GPU. The CPU displayed poor performance when compared to the GPU accelerated version of the model, which achieved at least 33% more performance.

Image Resolution	CPU RVM	GPU RVM
1920x1080	1842.7 ms	55.8 ms
3520x1980	5857.5 ms	202.4 ms

Table 6.1: Inference time in milliseconds of RVM running on different hardware

All the GPU operations used by the Torch framework are implemented using the *NVIDIA CUDA* API, this API provides a set of functions alongside a compiler that enables a programmer to use the GPU for general purposes with the objective of accelerating parallel algorithms.

The problems of executing the model inside the GPU hardware start by making data available inside the respective device. Typically, the system's RAM memory is available

only to the CPU, so the CPU needs to make data transfers to the respective device memory to be used. In the opposite part, when the GPU processes and outputs predicted values of the network, those values need to be available for the CPU to use on the application that needs it. PyTorch provides this functionality by calling the respective `torch.tensor.cuda` and `torch.tensor.cpu` to transfer tensor data to GPU and CPU respectively.

Along with these data transfers, the typical network may not be ready to read some type of values, some networks only work with normalized values, which, depending on the application may need to be pre-processed to match the network specifications.

The inverse problem is also valid, when the GPU outputs some values, it may happen that those values are not in a format that the application is able to read and use.

With this, the integration of a generic network must be divided three phases:

1. Processing inputs: convert data types, structures and transfers to specific device;

2. Inference phase: totally dependent on the network;

3. Processing outputs: reading and processing the network output for the application.

The inference phase is totally dependent on the forward pass of the network, when the architecture is designed and trained this phase stays fixed. The processing of the inputs and outputs is totally application dependent, which means that the programmer needs to make sure that the data is available to use by the Torch framework in a tensor format and available on the corresponding device (GPU or CPU) where the network will run, in this case it will run in the GPU to achieve a lower inference times.

Image Resolution	Uploading Time	Downloading Time
1920x1080	7.7 ms	6.7 ms
3840x2160	16.6 ms	27.2 ms

Table 6.2: The time, in milliseconds, taken to provide the GPU model with CPU data (uploading time) and vice-versa (downloading time)

To provide the values in the tests 6.2 50 executions were made in each resolution to calculate an average of the time taken for each operation using the previous referred PyTorch methods: `torch.tensor.cuda` and `torch.tensor.cpu`. Also, to match the worst case scenario the cache of the CUDA environment was cleared between tests.

With these measurements it is possible to check that, when an image is in the system's RAM the uploading to GPU takes some time. The total time that takes to transfer data between hardware makes integrating the network unfeasible with higher resolutions to run at 60 FPS unless the image is highly downsampled, losing a lot of information during this process. Making the transfer of CPU to GPU an avoidable solution.

Avoiding the device (GPU) to/from host (CPU) transfers takes this responsibility for the C++ wrapper to come with the capacity of running the network using the GPU while

avoiding this type of memory transfers. To understand this functionality an introduction to how the rendering engine is implemented is needed.

6.2.1 Integration

As previously stated, the image data should avoid being copied from CPU to GPU, since these copies already happen when mapping the values from a broadcasting board to be used as a DirectX resource. Taking into consideration that the data is already present on GPU's memory, the main problem that needs to be addressed is how to take advantage of a DirectX resource and read it as a tensor in the Torch framework. The answer to this question is the previous referred NVIDIA CUDA API.

The NVIDIA CUDA API equips the programmer with the capability of using NVIDIA GPUs for general purpose computing. The Torch framework integrates all this functionality to extract a higher performance from the models. Since Torch works with this API, it is possible to index the DirectX resources responsible for storing the raw image from the broadcast board into the context of the NVIDIA CUDA API.

For indexing a DirectX resource, the NVIDIA CUDA API provides a number of functions. These functions take a DirectX resource and index it in the CUDA environment as a CUDA resource. CUDA resources are similar to the DirectX resources, as they can index any type of data, like a buffer with vertex values or an image in texture format.

With the capacity of indexing resources from other APIs, the CUDA API works as a middle-man between the DirectX API that uses images as textures to index and draw into frame buffers, and the Torch framework that treats images as tensors with a specific data type and layout.

6.2.1.1 Implementation

First, the Torch framework allocates a contiguous slice of memory where the image will be written to. This slice should match the total resolution of the image and the data type should be indexed according to the type of DirectX texture, in this case, a byte tensor, which means that each element is composed by 8 bits.

Previous to the allocation of an empty tensor in the GPU's memory, the engine already allocated the DirectX resource that will be used to write new data into. The C++ wrapper receives the pointer for the respective resource and a pointer to a new allocated resource where the output of the network will be written into. When receiving the pointers, the wrapper tries to register the DirectX into the CUDA API. If any operation fails, the engine is alerted so that it will not be able to do inference with this resource, aborting the execution of the model.

When the CUDA API successfully registered all the input and output DirectX resources, it is ready to start using them. When a new image is received and the engine asks the C++ wrapper to infer about the image, the CUDA API maps the input and output resource into its context. If the mapping was successful, the DirectX should not apply any type

of operation while CUDA is indexing that data. This may lead to unexpected results according to the documentation given by the NVIDIA CUDA API. From this point on, assuming that the mapping was successful, the data must be copied to the previous reserved memory of Torch present in the GPU.

The memory copy inside the GPU is needed because resources mapped by CUDA have no pointer available for the programmer since indexing data on GPU can vary depending on the GPU's model and drivers to provide better cache hits when indexing those resources. For that reason, the CUDA API wraps those driver-dependent implementations with a `cudaArray` structure.

Even though the `cudaArray` structure does not provide any type of memory pointer, the NVIDIA CUDA API provides the capabilities of copying that data into a contiguous block of memory. This functionality is provided by `cudaMemCpyFromArray`, which accepts the destination address of the GPU's memory where the data should be written to, the array structure that will be the source of the data, and the size of the elements in that array.

After this memory copy inside the GPU, the Torch framework which had an empty tensor is now filled with the data from the DirectX resource. The only operations left before entering the inference phase are converting the values into floats, and normalizing the data between 0 and 1. All these operations are integrated directly using the Torch framework with the filled tensor. The input is ready to be fed into the model, ending the input processing phase.

After the inference of the network, two images are created, one with the RGB colors of the foreground layer, and another one with the alpha values predicted by the model. Both of the tensors were concatenated to create a 4-channel image which will be the final output image. This image must be forced by Torch to be contiguous in GPU's memory for the following memory copies. For the final operation, the NVIDIA CUDA had already mapped the output resource to another `cudaArray` that matches the same type present in the final image. In this case, DirectX is expecting a 4-channel image where each channel has a normalized float value. Since the tensor is contiguous in memory, the CUDA API can use the function `cudaMemCpyToArray`, which receives a pointer of the GPU's memory where the data is stored, the `cudaArray` where to write the data, and the respective size of the data. After all these operations, the NVIDIA CUDA API can unmap the resources. This marks the end of the output processing stage, making the output available for the DirectX API to use.

The calculation of the background layer can be implemented using the same pipeline, where Torch takes the predicted alphas by the model, the foreground layer and the input image and creates a new tensor with the respective background colors concatenated with the inverse of the alpha values. But this makes processing the output expensive. Also, it adds the need for creating another 4-channel DirectX resource to be filled with the respective background image and its alphas. This solution would demand more memory consumption and processing power, not only in memory copies inside the GPU, but also

in the rendering stage of the engine where the shader needed to index another texture.

To avoid this overhead of having another texture to be indexed by the rendering pipeline, the implementation took advantage of the shader stage to provide this functionality.

As previously referred, shaders are small functions that run in a specific step of the rendering pipeline. Since the resources of the foreground and alpha predicted values alongside the raw image are already created, a simple shader can access the color data of the texture to extract the background layer. Using the shader stage, it is possible to switch from foreground to background highlight with a simple press of a button without changing any code behind the DirectX resource management and CUDA indexing.

With this, the engine uses a render pass to run the shaders of the alpha matting. These shaders have access to the raw image and the predicted values of the network and, depending on the mode, blend the foreground objects or the background objects. The background colors are calculated based on the formula in 4.1. This solution is typically thread-safe and does not interfere with the performance, since in each render call, the already compiled shader will be executed according to the option selected by the user. The shader for blending the artificial background selected by the user is presented in 6.1.

Listing 6.1: Background blending shader

```
float4 WPixelShader(PixelInputType input) : SV_TARGET
{
    // the alpha matte generated by the AI
    float4 alphaMatte = AlphaBlending.Sample(Sampler,input.tex);
    // the raw video input
    float4 videoInput = RawImage.Sample(Sampler,input.tex);
    // the artificial background defined by the user
    float4 backgroundLayer = BackgroundLayer.Sample(Sampler,input.tex);
    // the background of the raw video defined by the AI
    float3 background = float3(videoInput.rgb*(1-alphaMatte.a) * (1-backgroundLayer.a))
        ↪ + backgroundLayer.rgb * (1-alphaMatte.a);
    // the foreground based on the alphas and foreground colors generated
    float3 foreground = alphaMatte.rgb * alphaMatte.a;
    // the final blend which is a sum of the background and the foreground
    return float4(foreground + background,1.0f);
}
```

The following steps need to be taken a priori:

1. Create a DirectX a 4-channel image resource where the matting data will be written to;

2. Register the DirectX resources to be mapped by CUDA;

For processing the image input of the network:

1. Map the DirectX resources to the CUDA context;

2. Copy the data from the input resource into the input Torch tensor;

3. Prepare the data using the capabilities of the Torch framework, changing the data to float values, normalizing the data and permuting the format of the tensor to have its channels as its first dimension;

For processing the output of the network:

1. Concatenate the foreground colors with the alpha values from the network;
2. Make the tensor contiguous in GPUs memory;
3. Copy the contiguous memory block to the indexed DirectX resource using the CUDA indexing;
4. Unmap all the DirectX resources to be used by the application when rendering to avoid unpredicted behavior;

In 6.2 it is possible to see the RVM architecture integrated in the broadcasting engine.

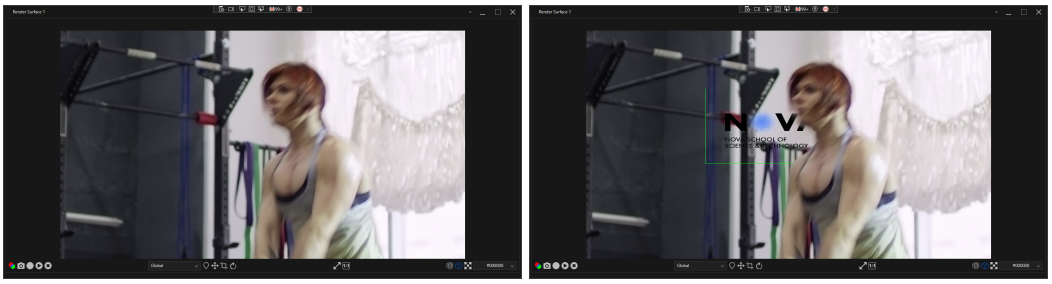

Figure 6.2: Example of the broadcasting engine running with RVM with no background (left), and with the SST logo as background (right)

6.2.2 Alternative Implementations

6.2.2.1 Input Copies

The previous implementation provided a final and working solution for the proposed problem. Nevertheless, as it can be seen, there are multiple hidden memory copies inside the GPU. First, the input resource is copied to a Torch tensor with the correct image format. Since the engine stores the initial image as a byte image, the tensor needs to be composed of byte types, but the model typically works with normalized float values. The conversion of these values requires a memory copy, the Torch allocates a new tensor of the same shape but composed by float elements. This is needed as a way of keeping the byte tensor always allocated, as to be used in a subsequent memory copies, so Torch it does not need to allocate new memory for the input tensor every time there is a conversion.

The problem with this implementation is that it requires two memory copies: one to copy the data indexed by the CUDA API, and another to change the data type of the values, creating a new tensor. The normalization is implemented as an in-place operation, meaning that no new memory is allocated, and the normalization is applied to the already

created float tensor. In this implementation, two memory copies are made inside the GPU: one explicitly made by the `cudaMemCpyFromArray` and another to change the data type of the tensor data.

To solve the double memory copy, a custom CUDA kernel was created. CUDA kernels, similar to shaders are programs that run on the GPU, those kernels are launched in a multithreaded environment provided by the hardware.

The kernel receives a pointer to the GPU's memory where Torch allocated the tensor, and a `cudaTextureObject`. This object, similar to `cudaArray`, is a wrapper around a DirectX resource that can be indexed as a texture. When CUDA creates a texture based on a `cudaArray` it is possible to access the texture data from the kernels inside the GPU.

With the data indexed as a texture, the custom kernel accesses that data, converts it to match the data type that the model requires, and writes that converted data to the tensor indexed by Torch. With this implementation, it is possible to have less memory copies being executed inside the GPU.

Using a custom kernel to execute the memory copies, two implementations are possible, the only difference being how threads work with the data and how many threads are needed. In a first implementation, each thread processes a line of the image, so when starting the threads, the number of threads is equivalent to the height of the image. This kernel is implemented in 6.2.

Listing 6.2: Custom kernel with a line per GPU thread

```
__global__ void memcpy_tensor_row( cudaTextureObject_t image,
                                   half* tensor,
                                   unsigned int width,
                                   unsigned int height)
{
  unsigned int x = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int y = blockDim.y * blockIdx.y + threadIdx.y;
  unsigned int idx = x * width * 3;
  if (x > height - 1)
    return;

  for (unsigned int i = 0; i < width; i++)
  {
    float4 text = tex2D<float4>(image, i, x);
    tensor[idx++] = half(text.z);
    tensor[idx++] = half(text.y);
    tensor[idx++] = half(text.x);
  }
}
```

There are alternatives to this specific kernel, since the two most common resolutions of images are 1920 or 3840 pixels wide, it is possible to create two custom kernels with the `width` variable being a fixed value. With the width as a fixed value it is possible for

the compiler to unroll the loop and make the operations slightly faster. The unrolled loop alternative was also benchmarked in following tests (presented in 6.3).

Another possible implementation for the custom kernel is making each GPU thread process a single pixel of the image, loading its data from the `cudaTextureObject`, converting it and writing into the respective position of the Torch tensor. This implementation is presented in 6.3.

Listing 6.3: Custom kernel with a pixel per GPU thread

```
__global__ void memcpy_tensor( cudaTextureObject_t input_image,
                               half* tensor,
                               unsigned int width,
                               unsigned int height)
{
    unsigned int x = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int y = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int idx = x * width * 3 + y * 3;
    if (x > height - 1 || y > width - 1)
        return;
    float4 text = tex2D<float4>(input_image, y, x);
    tensor[idx++] = half(text.z);
    tensor[idx++] = half(text.y);
    tensor[idx] = half(text.x);
}
```

All the implementations referred until now are valid implementations of a final product to run the network without CPU to GPU copies. But among all these valid implementations, some showed better performances than others. On table 6.3, it is possible to check the computing time of the RVM model with different input memory copies.

Copy Method	1920x1080	3840x2160
Cuda Copy	8 ms	22 ms
Thread/Line Rolled	11 ms	33 ms
Thread/Line Unrolled	12 ms	33 ms
Thread/Element	8ms	22 ms

Table 6.3: Inference time in milliseconds of running RVM with different input processing algorithms

The implementation of an element per GPU thread appears to be close to the standard use of the `cudaMemCpyFromArray`, since both to take the same time. Even though the custom kernel implements the conversion of data from byte to float and the channel switch operation, it seems that the implementations provided by the Torch framework achieve a similar performance. The major advantage of using a custom kernel implementation with a thread per pixel is to reduce of memory usage, since a single Torch float tensor suffices to be filled with all the image data for the network.

6.2.2.2 Output Copies

Apart from the ones necessary for the input, there are also memory copies necessary the for processing the output. As referenced in the initial implementation, the model of the network outputs two tensors, one with the foreground colors and another one with the alpha values. From those two tensors, a third one is created with the values of both merged into a single one, with the purpose of creating the 4-channel image to be used as a texture in the DirectX pipeline. The concatenation typically creates a memory copy, where the two tensors are written to a third one capable of storing all the data from both. To avoid this concatenation process there is a need to dig deeper on how the model creates the output data.

Analyzing the code of RVM architecture, it is possible to see that there is a split in the data located between the decoder and the refiner, that separates the foreground from the alpha values. From this point, the refiner needs to make a number of concatenation operations to recreate the original predicted image. So, by removing the data split in the middle of the network and removing all the concatenation operations inserted in the refiner due to that splitting, the model was slightly optimized. The network got its overhead diminished, and there was no need for a concatenation process to occur inside the engine, since that data is already indexed by a 4-channel Torch tensor throughout the entire network. From this, the only requirement is to make sure that the data is contiguous within the GPU's memory that is going to be copied to the DirectX resource using the CUDA API.

6.3 C++ to C#

Up until this point, the C++ API was used in multiple implementations of the integration. But in the context of this project, the engine used was written in the C# language. This creates a problem for libraries that are directed at the C++ language without out-of-the-box bindings for the C# environment. To bind the C++ functionality to C#, an interface was provided.

6.3.1 C++ Interface

In the *.NET* framework provided by Microsoft, the interaction between managed code (typically C#) and unmanaged code (usually C++ and C) is common in various applications. The Component Object Model (COM) defines how managed and unmanaged code interacts in the system and how one can use the other one's functionalities. The only requirement to use the COM is the ability to pass pointers and call functions using pointers between languages. It was through COM specifications that a C# wrapper was developed to run C++ code.

An interface was provided to the C# as an external Dynamic-Link Library (DLL). This DLL provides multiple functions to work with the model using different types of

resolutions.

The interface to the C# language is composed by the function presented in 6.4.

Listing 6.4: The C++ interface for C#

```
#define ALPHA_MATTING_API __declspec(dllexport)

extern "C" ALPHA_MATTING_API
    void* RVM_New(char* path);

extern "C" ALPHA_MATTING_API
    bool RVM_Set_Image_Data(void* model,
                            float downsample_ratio,
                            int height,
                            int width);
extern "C" ALPHA_MATTING_API
    bool RVM_Set_D3_Input_Resource (void* model,
                                    vod* image_res_input,
                                    void* mask_res);
extern "C" ALPHA_MATTING_API
    bool RVM_Infer(void* model);
extern "C" ALPHA_MATTING_API
    void RVM_Dispose(void* model);
```

Starting with the `RVM_New` function, its responsibility is loading a TorchScript accelerated model from a file specified by the `path` variable. The `RVM_Set_Image_Data` is responsible for initializing all the necessary data to start running the model with the given image resolution and downsample ratio. The `RVM_Set_D3_Input_Resource` serves to register the DirectX resources to the CUDA context. The two last functions must be called before trying to infer values from an image. If both functions are not called to allocate the necessary data, the inference may end up with unexpected results. The `RVM_Infer` is the method called to properly infer about an image. There is no image data given to the function since at this stage the CUDA context is already indexing the correct resources. `RVM_Dispose` cleans all the Torch resources and unregisters all the DirectX data from CUDA.

This DLL does not provide any type of locking system, so it is implied that the application calling this functionality has no thread race to the DirectX or any other dependent API.

6.3.2 User Interface

For deploying a final product to be used by designers a UI was provided to interact with the parameters of the model and the layers that will be used for blending in the rendering engine. This UI was written using the Windows Presentation Foundation (WPF) framework using the designs that match the remaining UIs of the engine.

The foreground and background layer parameters existent layers, rendered by the engine, that can contain multiple primitives. Based on the user the rendered image of

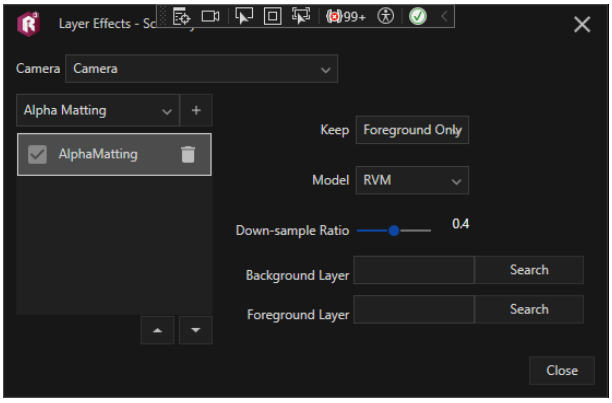

Figure 6.3: UI of the alpha matting integration on engine

the selected layer will be read by the shader to correctly blend the color based on the generated alpha matte.

6.4 Accelerating the Model

As previously referred, executing the model using TorchScript optimizations produces a valuable speed-up of the inference time. In this section, the objective is to add to the speed-up of the base TorchScript model with the help of compilers and different quantization methods.

6.4.1 Torch Inductor

Torch Inductor is a compiler backend used by the method `torch.compile` to accelerate models using optimized kernels, different from the `torch.jit.script`. The model generated by this method is compiled with a fixed input and always requires compilation when a new format of input is provided. The speed-ups of the base RVM model are presented in 6.4.

The `torch.compile` also gives the choice of changing which backend to use in the compilation. One of the backends available is the *TensorRT* API provided by NVIDIA to optimize models to their GPUs.

The major issue with this compilation procedure is that it is only supported in the Linux systems.

6.4.2 Frozen graph

Another available operation included in PyTorch that attempts to increase the speed-up of the scripted base model is the `torch.jit.freeze` function. This function clones the given scripted model and tries to inline submodules, parameters and attributes and convert them to constants in the graph of execution.

6.4.3 Quantizations

Quantization is useful when deploying an architecture since it is possible to extract good results with less computing power. By lowering the precision of the data (including the weights of the layers in the model), it is possible to reduce the memory requirements and inference times. According to PyTorch documentation it is possible to reduce up to 4 times the memory requirements of a model.

Multiple types of quantization exist. Typically, two are referred: the *Quantization-Aware-Train* and *the Post-Train Quantization*. In Quantization-Aware-Train, the model is trained with quantization errors so, when the model is quantized, it loses less information. In the Post-Train Quantization, the model is already trained and a quantization is applied afterwards. The previous one is the quantization process that was tested, since the selected architecture had already trained weights.

The PyTorch framework also has the ability of clamping a generic model from 32 bits float to 16 bits float. This reduces the precision as well as the overhead of the model, not only inside the model but also in its integration with the broadcasting engine (since the data is smaller, the memory copies take less time).

PyTorch provides other methods of quantization. One of them, referenced as *FX Graph Mode*, is a Post-Training Quantization solution. This quantization is an automatic process that creates an intermediate representation of the model and executes quantization passes to decrease the precision of the model. This implementation only works in code that is traceable for Torch on a forward pass of the model. In the case of control flow, it may happen that part of the code is not quantized. With some changes to the model it was possible to apply this quantization, but it was not possible to convert it to TorchScript, so it could not be integrated in the runtime environment.

Nevertheless, for unknown reasons the FX Graph applied to RVM displayed a slower performance compared to the base model. FX-Graph is still under prototyping which may be the cause of the slower performance in some models.

In the plot 6.4 it is possible to compare multiple versions of the RVM model with different quantization methods and optimizations applied. The different quantized and optimized models had a first 5000 iteration warm-up phase and another 5000 iterations for each resolution to average the inference times.

6.5 Results and Discussions

From the final integrated product it is possible to evaluate how well the solution behaves in realistic scenarios. These results rely solely on an empirical evaluation of the values given by the network and how well the integration can run during the standard use of the engine.

The network was tested in a final environment running a football stream. The first detail noticed is that, even though the architecture can detect most of the players' silhouettes, it

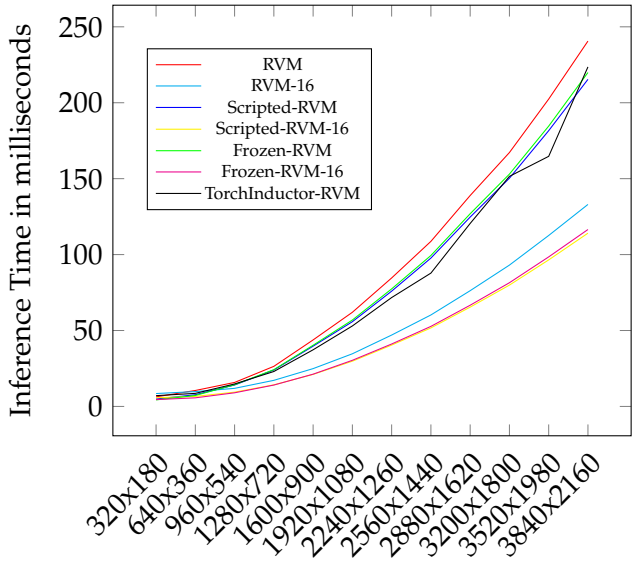

Figure 6.4: Inference times with multiple versions of RVM at different resolutions

leaves out many other players in the field. This is typically noticeable when a team has a similar color with one that is present in the surrounding environment.

A problem also displayed in the tested football match is the flickering that occurs with high camera variations. This occurs because the network cannot extract good information from the sequence of frames where the camera is rapidly changing, making it lose important information in the next few frames.

In terms of performance, the architecture provides the needed performance of 60 FPS at not only FHD but also UHD. Of course, when running in UHD it may be necessary to increase the downsample ratio of the input image inside the network. When using the network, the engine can maintain a good performance overall, but this creates new problems. Not only is it needed to increase the capacity of the hardware, but also the probability of adding a GPU bottleneck is very high. This is so because, in big projects where animations are being played, masks are being calculated and a lot of primitives are being drawn, meaning the GPU has a big amount of rendering information to handle and also needs to allocate resources to execute the neural network.

In the engine used to integrate the network, a scene is composed by multiple scene layers, which are composed by multiple primitives. To test the empirical results, a scene composed by only a scene layer was created, and inside this scene layer there was only a quad to render the video output where the network would infer. With this simple setup, with no other primitive or post-process effect, the engine consumed an overall of 30% of the NVIDIA RTX 3070 Ti capacity. This means that, in some big project it may be applied, but the system can turn out very unstable and even crash.

The use of the GPU is even more noticeable when working with transformers. In the VM-Former and the custom architecture presented in The Search for Transformers Chapter, the execution of the networks in a simple environment where random images

are being feed to the network using PyTorch, NVIDIA RTX 3070 Ti reaches 90% overall use. This means that, at the time of writing, unless using a very small input image, it may be unusable in a real-time environment.

7

Future Work and Final Remarks

In this chapter there will be presented improvements not only to the dataset, but also to the RVM and custom architecture design and training. New AI technologies are also referred with the purpose of improving future broadcast productions.

7.1 Architecture Enhancements

7.1.1 Custom Model's Training

The created architecture displayed great performance in terms of inference speeds, achieving 60 FPS at almost the same resolution as the typical CNNs available for image and video matting. But even though the inference speeds match the requirements, there was no training applied to the network, which could not show how capable the network is.

The training of newly created network, despite not providing proper results, follows the baseline described in the The Search for Transformers Chapter. With multiple datasets, the train initially aims at making the network capable of detecting the human figure in different contexts. Finally, the network is going to be improved end-to-end with alpha matting datasets to improve the outlines of humans. Finally, similar to the workflow applied to the RVM architecture, the architecture will be fine-tuned for sports contexts.

7.1.2 Coherent Encoder

One important point to discuss the RVM is that the used backbone is a pre-trained network available for public use. The architecture is designed from scratch for video matting. Knowing that data is sequential, RVM was trained using the temporal coherence loss with data present in the batch. Following the same idea, the training step of the custom architecture extracts the multi-level features and tries to improve the coherence of those features. This means that the encoder will also be trained to provide features leveraging the fact that data is temporarily sequential.

One goal also added to future work is the addition of temporal capacity to the encoder and not only the decoder. With this, it may be possible to create a more solid alpha matte throughout sequential frames, avoiding flickering during broadcast.

7.1.3 Hybrid Decoder

The custom architecture decoder followed the Robust Video Matting decoder provided. By only changing a few parameters, it is possible to integrate the decoder with any type of encoder as long as the encoder provides multi-level features. Even so, this decoder does not add any new functionality.

One of the objectives is to work on a decoder that has low impact on the performance while also making use of the transformer capabilities. This implementation is mainly based around VM-Former decoder, with the exception that instead of a raw Vision Transformer or Swin Transformer, the transformer blocks are switched to the Efficient Additive Attention of Swiftformer, and convolutions operations are added to extract local information.

The use of a full hybrid architecture, according to Transformer-Based versus CNN-Based Section, tends to have better robustness than full CNN or transformer-based models.

7.1.4 Learnable Downsampling and Upsampling

An important feature that was not implement, similar to RVM, is the fact that the custom architecture has the possibility of providing a downsampling ratio parameter to define how much the network should downsample the image. Based on this parameter, the network applies a static algorithm, typically bilinear downsampling to reduce the image size.

After the prediction, the network upscales the images using any type of upscaling technology (similar to Deep Guided Filter) to match the original resolution. So, there is no relation between the downsampling and the upsampling process.

To improve this, the downsampling and upsampling process should be related. In this way it is possible to downsample and upsample in a more consistent way. Spatio-Temporal Refiner provides an initial implementation of how to relate both operations. With learnable parameters, it is possible to make the network learn how to downsample and upsample an image, decreasing the loss of information throughout the network.

7.1.5 Audio Guidance

One important feature that it was not implemented and for consequence, untested, is using the audio as a feature generator in the matting network. Typically, the broadcasting of sports comes with audio, so the user can receive feedback on what is happening, this can also be integrated in the network.

The idea behind this implementation is, the network receives audio data and tries to extract important information out of it. The new generated data could provide the network with new situational awareness of the context that is working with, which could help guide the alpha matting.

7.2 Dataset Improvement

7.2.1 Increasing Data

Another future goal is generating more data following the same approach presented in 3, since as can be seen in Architecture Fine-Tuning, it is possible to achieve a more robust model if the dataset has more variety. One problem with the created dataset is that even though multiple resolutions were recorded, those resolutions turned out unbalanced between the different sports, as the tennis-related footage was the only sport generated at UHD resolution.

In the Sports Oriented Dataset Chapter there was no reference to test that data against. This was so because, due to lack of time and data, the testing set was not created. However, it is appointed to be released in next versions of the dataset.

7.2.2 Synthetic Data

The generated dataset followed the most common solution for generating alpha matting values. The problem with this solution is that it is very limited, since it requires recording multiple videos, with different equipment and human figures. It is not possible to create enough content to properly specialize a network in some sports. One example of its limitation is the ability to generate kayaking data. In kayaking, human figures are recorded on water inside kayaks. In a typical green screen studio it is close to impossible to reproduce the same environment.

Light effects on surfaces are also important to take into consideration. Alpha matting networks typically come with the limitation of not being capable of detecting foreground objects when subjected to very specular light effects.

With this in mind, a future project exists to start generating synthetic data using rendering software similar to *Blender*. Using these types of software it is possible to create highly realistic scenarios and generate the masks and foreground colors in an easy workflow. It is possible to implement very dense elements to be integrated in the matting dataset since it is also possible to match the transparency of each object in the scene individually.

7.2.3 Automated Generation

One matter of research taking advantage of multiple networks that are known to give the best results and integrate those networks in an automatic generation of data for the dataset.

The proposed design uses a very robust segmentation network on a random sport's image with the purpose of detecting human figures and their equipment. From the segmentation given by the network, a trimap is generated using a dilation and an erosion to define the unknown pixels (similar to 2.8.1). A trimap-based alpha matting network is applied to generate the final alpha matte for that image. With this procedure, it is possible

to automatize the process of generating good ground truth alpha matte, assuming that all the networks give overall good results.

7.2.4 Web Application

Another future work that has already started development is a web application to provide data to the alpha matting. The objective of this web application is to produce accurate alpha mattes and their respective foreground colors for a natural image.

The difference in this application is that, instead of drawing the corresponding alpha matte and foreground colors per pixel, the user is guided by a neural network that will help extrapolate information to generate the needed data. This implementation borrows ideas from the referenced Smart Scribbles section. By drawing scribbles, the user can guide the network, without the cumbersome process of working with individual pixels, thus reducing the overall manual work.

One improvement to this app is, since trimap-based architectures tend to have better overall results for alpha mattes, the web application can have those networks work in the background.

7.2.5 Blending Images

Until this moment, the general workflow for providing an image to an alpha matting network follows these steps:

1. Select a random foreground video or image and the respective alpha matte;

2. Select a random or specific background video or image;

3. Merge the two using an elementwise multiplication and the alpha matting formula in each channel.

These steps create a valid source image that is read by the network. The problem with this simple blending algorithm is that it does not reproduce believable lighting effects, and the source image ends with an obvious salient object.

The proposed solution to help mitigate this problem of a raw and simplistic blending technique is adding a pass through a blending network to blend the two images (background and foreground), according to the respective alpha values. In figure 7.1 it is possible to check the results of a blending network.

As can be seen by the car example present in 7.1 it is possible to not only realistically blend two images, but also reproduce artistic styles.

Figure 7.1: Example of image blending from the paper [38]

7.3 Broadcasting Enhancements

Up to this point, the rendering pipeline in broadcast is being used to create effects in existent video frames. With the project presented on this document it was proven that it is possible to add layers to the rendering pipeline that increase its functionality.

Using AI integrated with a rendering pipeline brought great contributions to the market, like Deep Learning Super Sampling (DLSS) from NVIDIA and AMD's *FidelityFX*. Both can be integrated in rendering pipelines to upscale the resolution of an output frame.

With the previous solutions available for use, with hardware acceleration provided by manufacturers, it is possible to allocate resources to use GPU-accelerated neural networks with little overhead without bottlenecking the rendering pipeline.

The research made for this project proved that it is possible to run a real-time broadcast at the 60 FPS mark using AI. Adding to this, the tendency and desire for the markets to include, in a number of contexts, GPU-accelerated neural networks, the interest on applying other neural network models with different purposes arose.

7.3.1 Video Super Resolution

Starting with new state-of-the-art technology used in games, video super resolution works similar to a refiner. The main objective is to upscale a video to a specific resolution to provide a better overall image quality. The video super resolution has a major difference in the refiners used in the tested networks: the typical Deep Guided Filter receives as input a low resolution image usually provided by a neural network and the original source image used by that network to match the resolution. However, in video super resolution there is no image with the original resolution to be used as reference. The network needs to infer on how to upscale the low resolution image to match a given resolution without any other references.

This implementation displayed great interest in the broadcasting industry, since it is possible, at low cost, to receive a FHD video and convert it to UHD resolution with high quality.

It could also improve the implementation of other networks. For example the alpha matting needs a refiner to match the input resolution to the output resolution. With the

video super resolution integrated, the network could work in a dowmsampled resolution and would not need to integrate a refiner, since the video would be upsampled later in the pipeline.

7.3.2 Fully CUDA Accelerated Pipeline

Currently, when it comes to processing image information that does not include rendering, most of the algorithms used in the broadcasting pipeline are projected to run on a CPU. With the research made for this project, a fully functional CUDA-accelerated pipeline appears to be possible since most of the data is eventually uploaded to the GPU for rendering purposes. Having capabilities to process data indexed by the CUDA environment, it is possible to create custom kernels to accelerate algorithms. One of the applications of the GPU capabilities is the processing of tracking data.

When a camera records an image of a stadium, the image data is processed to find out how the camera is positioned in that environment to match the position of camera in the virtual environment of the engine. This implementation is typically processed in CPU. But having an implementation accelerated by CUDA can enable higher performances and avoids CPU bottlenecks.

This logic is being applied not only to the processing of tracking data, but all high parallel algorithms that belong to the rendering pipeline of broadcast data.

7.4 Final Remarks

The proposed project's main objective was to create a solution to use an alpha matting network in the broadcasting context, mainly in sports. The solution involved the use of CUDA accelerated operations to help integrate a generic network in the rendering engine to achieve the fastest implementation possible.

For the model, RVM was optimized and used in the final product. A fine-tuned version of the network was created using the newly generated dataset showcased in 3, this step ended with relative bad results. It appeared that the generated dataset did not have enough diversity to correctly improve the model. As seen in the Chapter 4, the model provided better results in the validation data, but the validation data was very similar to the data provided during training. This similarity ended with great validation results, but poor empirical results in real videos. It turned out that using the original weights of RVM provided a better solution than the fine-tuned version.

As for the transformer development (referred in The Search for Transformers Chapter), at the time of writing it is difficult to find hardware capable of handling transformers in real-time without eventually throttling, but in long term, transformers will be a possible solution to integrate in high performance applications.

At the end of the project, the broadcast engine finished with a workable solution to generate alpha mattes. As a final example of what the product turned out, the image

7.2 displays a football match being broadcasted using the RVM neural network with the result background blended with a custom logo.

Figure 7.2: An example of the final product running during a football match

Bibliography

[1] X. Bai and G. Sapiro. "Geodesic Matting: A Framework for Fast Interactive Image and Video Segmentation and Matting". In: *International Journal of Computer Vision* 82.2 (2009-04), pp. 113–132. ISSN: 1573-1405. DOI: 10.1007/s11263-008-0191-z. URL: https://doi.org/10.1007/s11263-008-0191-z (cit. on p. 5).

[2] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000) (cit. on pp. 38–41, 56).

[3] L.-C. Chen et al. *DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs*. 2016. DOI: 10.48550/ARXIV.1606.00915. URL: https://arxiv.org/abs/1606.00915 (cit. on p. 14).

[4] Q. Chen, D. Li, and C.-K. Tang. "KNN matting". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 35.9 (2013-09), pp. 2175–2188. DOI: 10.1109/TPAMI.2013.18 (cit. on p. 46).

[5] P. Contributors. *PaddleSeg, End-to-end image segmentation kit based on PaddlePaddle*. https://github.com/PaddlePaddle/PaddleSeg. 2019 (cit. on p. 15).

[6] A. Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2020. DOI: 10.48550/ARXIV.2010.11929. URL: https://arxiv.org/abs/2010.11929 (cit. on pp. 21, 34, 35).

[7] V. Dumoulin and F. Visin. *A guide to convolution arithmetic for deep learning*. 2016. DOI: 10.48550/ARXIV.1603.07285. URL: https://arxiv.org/abs/1603.07285 (cit. on p. 8).

[8] J. L. EPIPHANY and s. Sutha. "Design of FIR Filters for Fast Multiscale Directional Filter Banks". In: *International Journal of u- and e-Service, Science and Technology* 7 (2014-10). DOI: 10.14257/ijunesst.2014.7.5.20 (cit. on p. 54).

[9] M. Gasparini. "Markov Chain Monte Carlo in Practice". In: *Technometrics* 39.3 (1997), pp. 338–338. DOI: 10.1080/00401706.1997.10485132. eprint: https://www.tandfonline.com/doi/pdf/10.1080/00401706.1997.10485132. URL: https://www.tandfonline.com/doi/abs/10.1080/00401706.1997.10485132 (cit. on p. 34).

[10] E. S. L. Gastal and M. M. Oliveira. "Shared Sampling for Real-Time Alpha Matting". In: *Computer Graphics Forum* 29.2 (2010-05). Proceedings of Eurographics, pp. 575–584 (cit. on pp. 5, 6, 46).

[11] V. Gupta and S. Raman. *Automatic Trimap Generation for Image Matting*. 2017. DOI: 10.48550/ARXIV.1707.00333. URL: https://arxiv.org/abs/1707.00333 (cit. on pp. 31, 32).

[12] K. He et al. "A global sampling method for alpha matting". In: *CVPR 2011*. 2011, pp. 2049–2056. DOI: 10.1109/CVPR.2011.5995495 (cit. on p. 46).

[13] K. He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV] (cit. on pp. 10, 11).

[14] A. Howard et al. *Searching for MobileNetV3*. 2019. DOI: 10.48550/ARXIV.1905.02244. URL: https://arxiv.org/abs/1905.02244 (cit. on p. 18).

[15] A. G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. DOI: 10.48550/ARXIV.1704.04861. URL: https://arxiv.org/abs/1704.04861 (cit. on p. 18).

[16] J. Li et al. "VMFormer: End-to-End Video Matting with Transformer". In: *arXiv preprint* (2022) (cit. on pp. 25, 27).

[17] S. Lin et al. *Real-Time High-Resolution Background Matting*. 2020. arXiv: 2012.07810 [cs.CV] (cit. on p. 37).

[18] S. Lin et al. *Robust High-Resolution Video Matting with Temporal Guidance*. 2021. arXiv: 2108.11515 [cs.CV] (cit. on pp. 17, 19).

[19] T.-Y. Lin et al. *Feature Pyramid Networks for Object Detection*. 2016. DOI: 10.48550/ARXIV.1612.03144. URL: https://arxiv.org/abs/1612.03144 (cit. on p. 26).

[20] Q. Liu, B. Z. Haozhe Xie Shengping Zhang, and R. Ji. "Long-Range Feature Propagating for Natural Image Matting". In: *ACM Multimedia*. 2021 (cit. on pp. 13, 14).

[21] Z. Liu et al. "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021 (cit. on pp. 21, 22).

[22] Z. Liu et al. "Video Swin Transformer". In: *arXiv preprint arXiv:2106.13230* (2021) (cit. on pp. 21, 22).

[23] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[24] S. Mehta and M. Rastegari. *Separable Self-attention for Mobile Vision Transformers*. 2022. arXiv: 2206.02680 [cs.CV] (cit. on pp. 22, 23).

[25] G. Park et al. "MatteFormer: Transformer-Based Image Matting via Prior-Tokens". In: *arXiv preprint arXiv:2203.15662* (2022) (cit. on pp. 24, 26).

[26] O. Ronneberger, P. Fischer, and T. Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. DOI: 10.48550/ARXIV.1505.04597. URL: https://arxiv.org/abs/1505.04597 (cit. on pp. 10, 11).

[27] A. Shaker et al. *SwiftFormer: Efficient Additive Attention for Transformer-based Real-time Mobile Vision Applications*. 2023. arXiv: 2303.15446 [cs.CV] (cit. on pp. 23, 25).

[28] Y. Sun, C.-K. Tang, and Y.-W. Tai. "Ultrahigh Resolution Image/Video Matting With Spatio-Temporal Sparsity". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2023-06, pp. 14112–14121 (cit. on pp. 29–31).

[29] A. Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762 (cit. on pp. 20, 21).

[30] J. Wang et al. *Deep High-Resolution Representation Learning for Visual Recognition*. 2019. DOI: 10.48550/ARXIV.1908.07919. URL: https://arxiv.org/abs/1908.07919 (cit. on p. 16).

[31] Z. Wang et al. "Can CNNs Be More Robust Than Transformers?" In: *arXiv preprint arXiv:2206.03452* (2022) (cit. on pp. 34, 35).

[32] H. Wu et al. "Fast End-to-End Trainable Guided Filter". In: *CVPR*. 2018 (cit. on pp. 19, 27, 28).

[33] H. Wu et al. *Fast End-to-End Trainable Guided Filter*. 2018. DOI: 10.48550/ARXIV.1803.05619. URL: https://arxiv.org/abs/1803.05619 (cit. on pp. 19, 20).

[34] X. Xiang et al. *Learning Spatio-Temporal Downsampling for Effective Video Upscaling*. 2022. arXiv: 2203.08140 [cs.CV] (cit. on pp. 28, 29).

[35] N. Xu et al. *Deep Image Matting*. 2017. DOI: 10.48550/ARXIV.1703.03872. URL: https://arxiv.org/abs/1703.03872 (cit. on pp. 5, 12, 37).

[36] L. Yang, Y. Fan, and N. Xu. *The 4th Large-scale Video Object Segmentation Challenge - video instance segmentation track*. 2022-06 (cit. on p. 60).

[37] X. Yang et al. "Smart Scribbles for Image Matting". In: *ACM Transactions on Multimedia Computing, Communications, and Applications* 16.4 (2020-11), pp. 1–21. DOI: 10.1145/3408323. URL: https://doi.org/10.1145%2F3408323 (cit. on p. 33).

[38] L. Zhang, T. Wen, and J. Shi. *Deep Image Blending*. 2019. DOI: 10.48550/ARXIV.1910.11495. URL: https://arxiv.org/abs/1910.11495 (cit. on p. 84).

[39] H. Zhao et al. *Pyramid Scene Parsing Network*. 2016. DOI: 10.48550/ARXIV.1612.01105. URL: https://arxiv.org/abs/1612.01105 (cit. on p. 16).

[40] W. Zhou et al. "TMFNet: Three-Input Multilevel Fusion Network for Detecting Salient Objects in RGB-D Images". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 6.3 (2022), pp. 593–601. DOI: 10.1109/TETCI.2021.3097393 (cit. on p. 46).

A

Appendix

Listing A.1: Laplacian loss helper functions

```
1
2
3  def laplacian_pyramid(img, kernel, max_levels):
4      current = img
5      pyramid = []
6      for _ in range(max_levels):
7          current = crop_to_even_size(current)
8          down = downsample(current, kernel)
9          up = upsample(down, kernel)
10         diff = current - up
11         pyramid.append(diff)
12         current = down
13     return pyramid
14
15 def gauss_kernel(device='cpu', dtype=torch.float32):
16     kernel = torch.tensor([[1, 4, 6, 4, 1],
17                            [4, 16, 24, 16, 4],
18                            [6, 24, 36, 24, 6],
19                            [4, 16, 24, 16, 4],
20                            [1, 4, 6, 4, 1]], device=device, dtype=dtype)
21     kernel /= 256
22     kernel = kernel[None, None, :, :]
23     return kernel
```