中国科学技术大学

学士学位论文

视频显著性检测及其 FFmpeg 集成

作者姓名：　韩佳乐

学科专业：　计算机科学与技术

导师姓名：　陈晓辉　尹华锐

完成时间：　二〇二一年六月九日

University of Science and Technology of China

A dissertation for bachelor's degree

Video Saliency Detection and its application in FFmpeg

Author: Han Jiale

Speciality: Computer Science and Technology

Supervisors: Associate Prof. Chen Xiaohui, Associate Prof. Yin Huarui

Finished time: June 9, 2021

致　谢

在研究学习期间，我有幸得到陈晓辉老师，尹华锐老师和姜文彬学长的指导与帮助。

依稀记得，当初我因考研导致论文开题延期。在教务系统上，大概 6 个人选择了陈老师的这个课题，但陈老师最后选择相信我。实验开始时，我对神经网络完全是零基础，面对 ffplay 的 3000 多行源码更是感到力不从心，多亏了姜文彬学长的帮助，才理解 ffplay 的工作流程，并开始入门 pytorch。实验进行到一半，我错误的将显著性检测理解为物体检测，还好陈老师及时提醒和指正，从而让我避免在错误的方向上花更多时间，除此之外，陈老师还细心教导我们论文引用部分的写作方法。到了实验后期，我成功将显著性检测模型嵌入到 ffplay 中，但是视频帧率比较低，不如人意，陈老师这时又热心指导我尝试使用 GPU 加速神经网络和 DMA 方式转移数据，并耐心告诫我们论文写作时的注意事项。就这样，一步一步，从无到有，我顺利地完成了毕业设计，此时的心情是欣慰和感激。

“好雨知时节，当春乃发生；随风潜入夜，润物细无声”，非常感谢陈晓辉老师，尹华锐老师和姜文彬学长的细心栽培和无私帮助。祝愿陈晓辉老师和尹华锐老师日后能培养出更多杰出学子，桃李满天下；也祝研二的姜文彬学长能早早发出优秀的 paper，百尺竿头，更进一步！

目 录

中文内容摘要

FFmpeg 作为先进的跨平台的开源视频处理软件，正在越来越多地被使用。但其缺少对显著性检测的支持，难以实现与显著性检测等常用图像处理算法的结合。随着 GPU 的飞速发展，大量优秀的基于神经网络的图像处理算法也纷纷涌现。如果能够实现这些算法与 FFmpeg 的结合，可以充分拓展 FFmpeg 的应用场景。本文旨在探究在 FFmpeg 中嵌入神经网络算法的可行途径，并嵌入显著性检测算法。但 FFmpeg 源码工程量巨大，每次测试编译，时间久，效率低。本文提出了一种巧妙的办法，将神经网络算法嵌入 FFmpeg 的播放工具 ffplay 中并单独编译。实验结果显示可以正常运行，速度可达 12fps，并且此方法通用性较好，亦可适用于其他基于神经网络的图像处理算法。工程代码地址在 https://github.com/hanjialeOK/YOLOv3-in-FFmpeg/tree/saliency

关键词：显著性检测；ffmpeg；poolnet；libtorch

Abstract

As an advanced cross-platform and open-source video processing software, FFmpeg is being used more and more widely. However, it lacks the support of saliency detection, so it is difficult to combine with saliency detection and other common image processing algorithms. With the rapid development of GPU, a large number of excellent image processing algorithms based on neural network have emerged. If these algorithms can be combined with FFmpeg, the application of FFmpeg can be fully expanded. This paper aims to explore the feasible way of putting neural network algorithm in FFmpeg, such as the saliency detection. But the source code of FFmpeg is a huge amount of work, each test compilation would take a long time with low efficiency. This paper presents an ingenious method, which embeds the neural network algorithm into ffplay and compiles it separately. The experimental result show that the system can operate normally, and the speed can reach 12 fps. In addition, This method has good generality and can also be applied to other image processing algorithms based on neural network. Code can be found at https://github.com/hanjialeOK/YOLOv3-in-FFmpeg/tree/saliency.

Key Words: saliency detection; ffmpeg; poolnet; libtorch

第一章　研究背景及意义

传统的显著性目标检测是基于人工设计特征，但由于缺乏高级语义信息导致难以应用于复杂场景。伴随着机器学习的春风，显著性目标检测也开始基于深度学习来提取显著性区域，并取得了不错的效果，如 PoolNet[1]，BASNet[2]，AFNet[3] 等。由于能够模仿人类视觉找到显著性物体，显著性目标检测在计算机视觉方面发挥着重大作用，显著性目标检测可以用于视频压缩编码：对图像的非显著区域设置较高的压缩比率，对图像的显著区域设置较低的压缩比率，这种模拟人类的视觉注意来设置视频图像的压缩比率，具有广泛的适用性。此外，还可以应用于医学影像诊断，机器人避障等。

FFmpeg 作为先进的跨平台的开源视频处理软件，能够完成视频封装与解封装，编码，解码，格式转换，本地播放，在线播放等功能，除此之外，FFmpeg 还有非常强大的滤镜功能，能够在后期对音视频进行多种多样的编辑。可谓是当代音视频处理领域的集大成者。windows 上的 MPC（media player classic），Linux 平台上 MPlayer，以及以跨平台著称的后起之秀 VLC，这三种多媒体播放框架均是基于 FFmpeg。

尽管 FFmpeg 功能十分强大，但其中缺少对基于神经网络的图像处理算法的支持。我在知乎上① 看到刘歧前辈② 曾经指导学生做过基于神经网络实现对图像进行去雨点、去雾的功能，并最终将其加入了 FFmpeg。本文以此为出发点，旨在探究在 FFmpeg 中嵌入基于神经网络的图像处理算法的方式，并提出了一种简单可行的方案。

首先，FFmpeg 源码文件工程量巨大，使用 configure 脚本进行管理，而且单次编译耗时较久。倘若直接在源码中修改，那么每次测试时都需要重新编译整个目录，效率非常低。为了解决这个问题，我的方案是先正常安装 FFmpeg，然后把神经网络模型集成到播放器源码 ffplay.c 中，单独编译修改后的 ffplay.c 时，只需要链接 FFmpeg 的动态库以及头文件，这可以使用 CMake 进行管理。工程的整体结构如图 1.1 所示，具体工程的创建步骤和细节在本文的第四章有详细介绍。在这种方式下，每次编译测试仅需数秒的时间。

其次，嵌入的神经网络模型来自于 PyTorch。当下机器学习的主流框架是 TensorFlow 和 PyTorch，前者由谷歌开发，2015 年发布，后者由 FaceBook 开发，

① https://zhuanlan.zhihu.com/p/106192748

② FFmpeg 官方源代码维护者，资深音视频技术专家

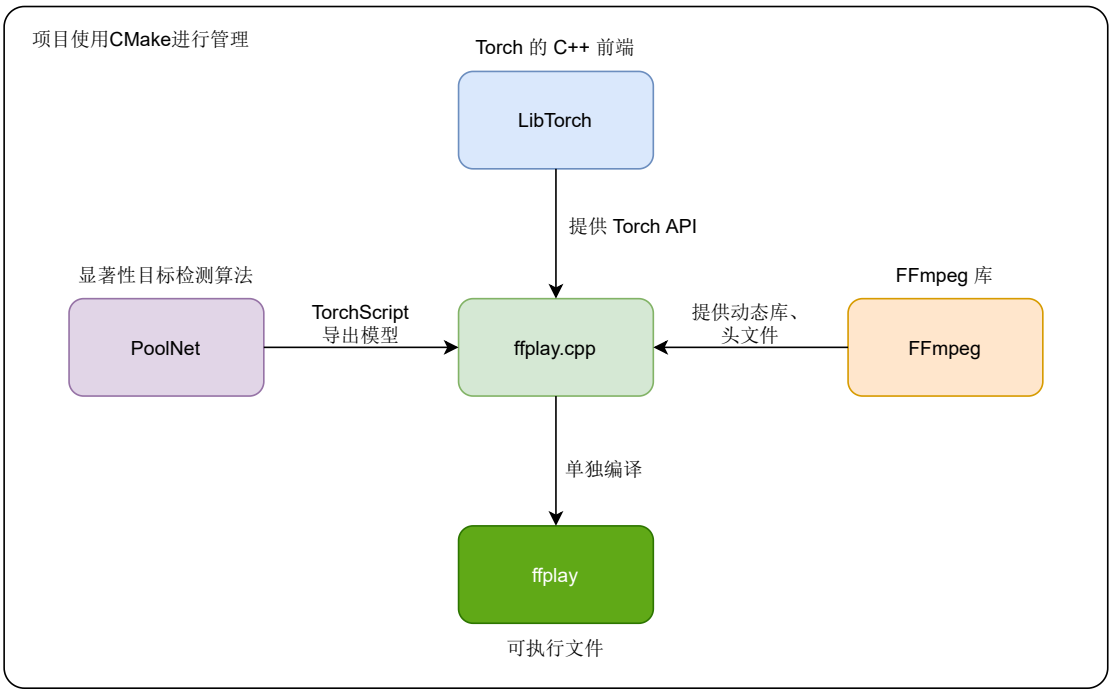


图 1.1　工程整体结构

2017 年发布。PyTorch 虽然发布时间较晚，但因其简洁和易于使用的特点，目前已经和 TensorFlow 平分秋色。PyTorch 提供了 TorchScript 工具，用于将神经网络模型部署到生产环境中。除此之外，PyTorch 还提供了 C++ 前端的 API，称之为 LibTorch，以支持开发者在 C++ 环境下搭建神经网络。

在实验中，我尝试了两种调用神经网络的方式：一是使用 TorchScript 工具将模型导出，在 C++ 前端直接调用模型；二是使用 LibTorch 将神经网络重新构建为类对象，在 C++ 前端把类对象实例化。我在第五章结果与总结中对比了两种方法的效果。总体来说，实验结果差强人意，对于输入为 256x256 的图片，在单个 GeForce GTX 1080 Ti 上运行时可以达到 12fps。

第二章　图像显著性检测

第一节　图像显著性检测简介

场景中的前景对象与其背景形成明显的视觉差异时，能够快速吸引视觉注意力，称为显著区域。人类视觉系统会自动从复杂的场景中选择性地关注显著区域，并为之分配注意资源，这被称为视觉注意机制。有学者将这种机制引入图像分析领域，从而诞生了显著性检测（Saliency Detecion）技术，可以从海量的图像信息中快速发现关键的信息，优先分配计算资源进行处理。显著性检测具体可分为视觉注视点预测（Visual Saliency Prediction）和显著性目标检测（Saliency Object Detecion）。本文主要讨论的是显著性目标检测，其目的是检测和分割图像中的显著性目标，以像素级的标签标记目标的完整区域。显著性目标检测的结果称为显著图（Saliency Map），显著图为灰度图，如图 2.1 所示，黑色为背景，白色为显著物体，像素的灰度值反映显著度的大小。

(a) 原图　　(b) 显著图

图 2.1　显著性目标检测结果示例

传统的具有代表性的视觉显著性目标检测算法有很多。IT[4] 算法采用自底向上的思想，将图像中各像素点在颜色、亮度、方向方面与相邻像素点的对比值来计算显著度的大小。SR[5] 算法是基于频域分析的纯数学计算方法，无需特征提取，只需分析图像的对数频谱。LRMR[6] 算法将图像分为稀疏矩阵（显著区域）和低秩矩阵（背景区域），图像经过稳健主元分析技术进行低秩矩阵恢复算法，差异部分得到的即为显著区域。然而，由于缺乏高级语义信息，导致这些算法在复杂场景难以有效实现前景目标和背景图像的分离。

近年来，随着机器学习的强势崛起，越来越多的致力于显著性目标检测的学

者开始把目光投向深度学习，并取得了瞩目的成果。深度神经网络的结构采用端到端的方式，分为多层，上一层的输出直接作为下一层的输入，每层为卷积，池化或非线性激活函数 ReLU①。经过数据集训练后，可以自适应地提取高级语义特征，从而克服人工提取特征的局限性。当前基于深度学习的显著性目标检测面临到两个问题，一是如何融合多层次深度特征，二是显著性物体的边界质量容易受到损失函数的影响。学者们纷纷致力于探索更加有效的神经网络结构以解决这些问题。BASNet[2] 使用一个编码-解码网络（即 U 型网络）进行显著性预测，再通过一个精心设计的残差模块来细化预测边界。AFNet[3] 在 U 型网络的基础上，在每层都嵌入了注意力反馈模块，并采用边界强化损失作为损失函数。PoolNet[1] 在 U 型网络的基础上提出全局引导模块和特征整合模块控制多级特征图与高级特征语义无缝融合。

第二节　PoolNet 简介

如图 2.2 所示，论文依旧使用 U 型网络，但在此基础上提出了两个亮眼的模块：一是全局引导模块（GGM），目的是为不同特征级别的特征图提供高级语义信息。二是特征整合模块（FAM），目的是使粗略高级语义信息与上采样路径中的细微特征无缝融合。由于这两个模块均是依赖于池化技巧，因此命名为 PoolNet。下面将围绕 U 型网络，全局引导模块和特征整合模块三个方面展开介绍。

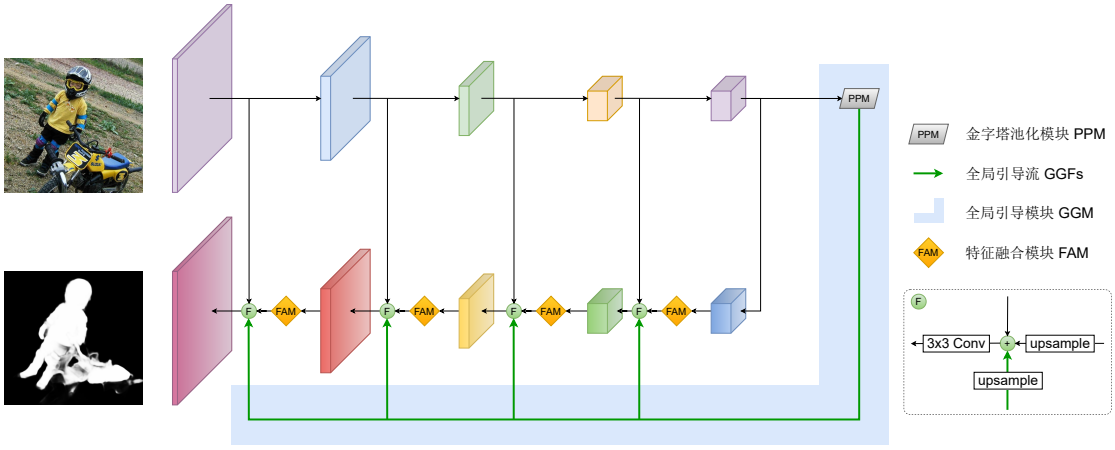


图 2.2　PoolNet 整体架构

①https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

一、U 型网络

U 型网络，又称编码器-解码器网络，最早由 Olaf Ronneberger[7] 等人提出并应用于医学图像分割。U 型网络是一种语义分割网络，包含了一个用于提取高层语义信息的收缩网络和一个用于输出定位的上采样网络，来自收缩网络中的高分辨率特征与上采样输出相结合，可以使定位更加精确。该网络在少量数据训练的情况下可以获得精确的分割结果。

PoolNet 继续沿用了 U 型网络，收缩网络使用经典架构 deeplab_resnet50[8][9]，上采样网络使用双线性插值，以尺寸为 300x400 的 3 通道图片作为输入，中间特征图的变化如图 2.3 所示。

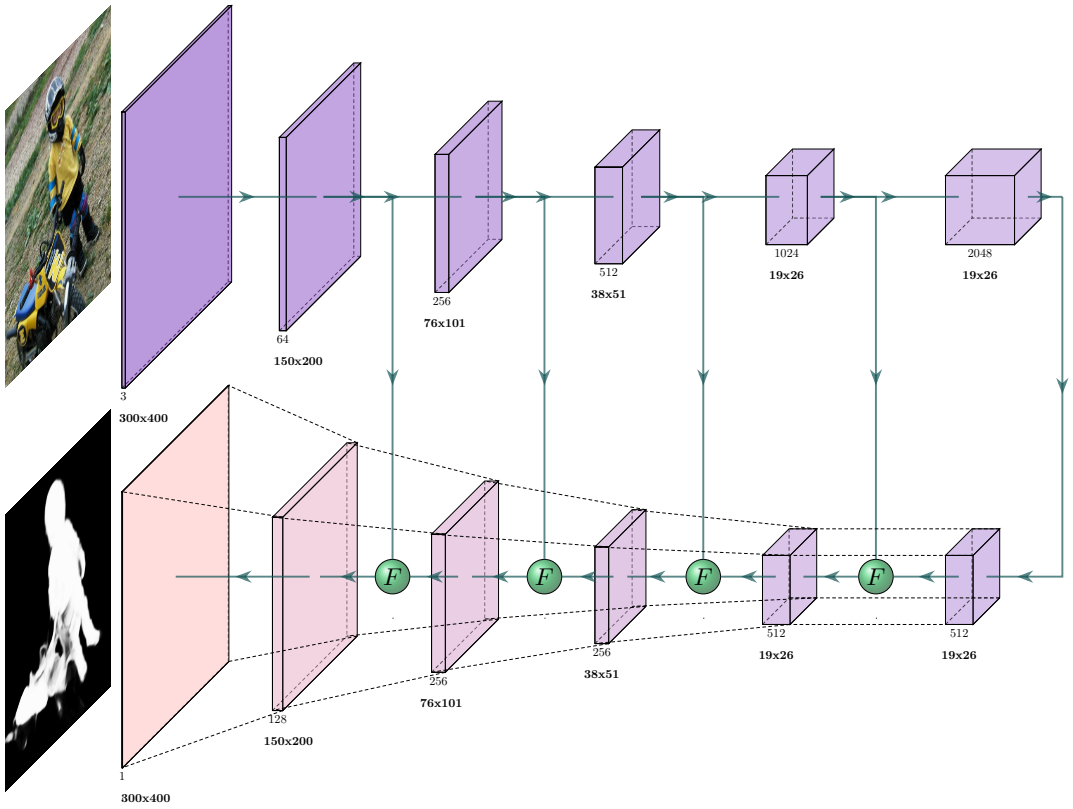


图 2.3　U 型网络

尽管 U 型网络取得了很好的效果，但是依旧存在很大的提升空间。首先，在上采样过程中，U 型网络底层获得的高级语义信息不断向上层传播，导致逐渐被稀释。其次，正如 Zhao[10] 等人指出的那样，CNN 网络的感受野并非和网络的深度成比例关系。针对这两个问题，PoolNet 提出了 GGM 和 FAM 模块

二、全局引导模块（GGM）

高层语义特征对定位显著物体的具体位置有很大作用，但是网络骨架获得的高级语义信息在上采样过程中会不断被稀释，导致其产生的作用会被限制。除此之外，在实际的实验中[10][11]，CNNs 网络的感受野比理论值要小得多，这一点对于深层网络非常明显，导致网络难以捕捉全局信息。考虑到对于上采样过程中高级语义信息的缺失，PooNet 引入了全局引导模块 GGM，包含了一个改进的金字塔池化模块 PPM 和一系列全局引导流 GGFs，目的是为了使每一级的特征图获得显著物体的位置信息。

1. 金字塔池化模块（PPM）

该 PPM 模块融合了 4 种不同尺度的特征图。以 300x400 的输入尺寸为例，通过 backbone 得到 19x26 的特征图（图 2.4 左边蓝色块），这是尺度最大的特征。该特征图通过 AdaptiveAvgPool2d① 分别池化为 1x1，3x3，5x5 三种小尺寸特征，经过卷积层后，再经过上采样将尺寸变回 19x26。然后将这 4 种特征 concat 到一起，由于通道数会变为原来的 4 倍，所以还要通过一个卷积层恢复至原通道数。

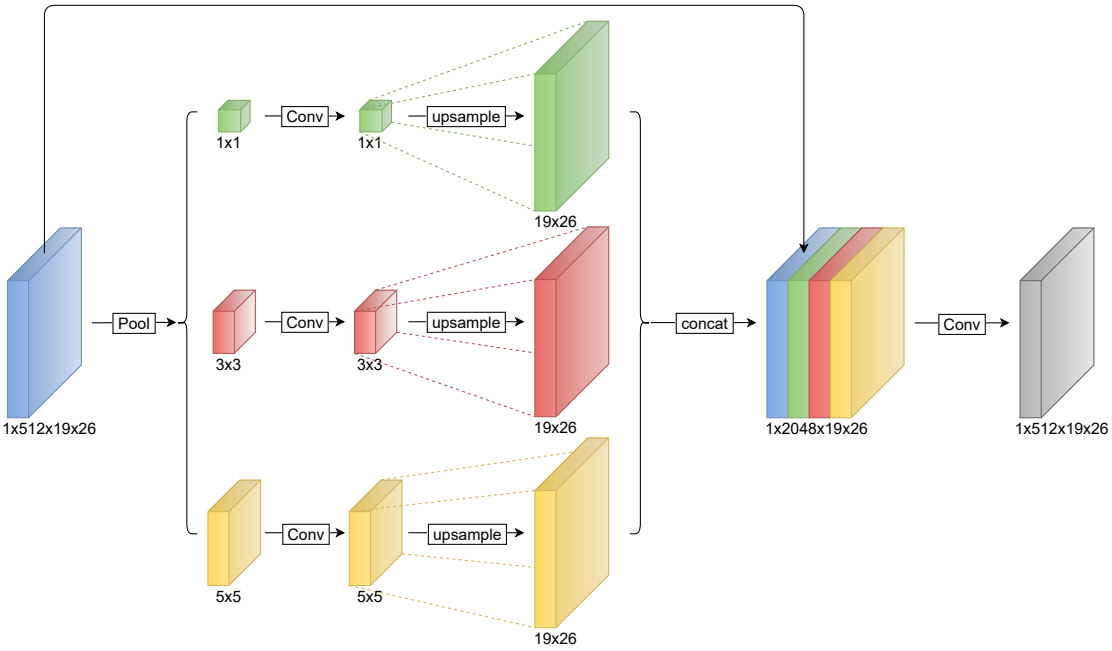


图 2.4　金字塔池化模块（PPM）

2. 全局引导流（GGFs）

现在需要做的是思考从 PPM 模块获得的高层语义信息（图 2.5 右上角灰色块）如何融入到上采样过程不同级别的特征图中。首先，将高级语义信息特征图分别进行比例为 1，2、4、8 的上采样，把尺寸调整为 19x26，38x51，76x101，

①https://pytorch.org/docs/stable/generated/torch.nn.AdaptiveAvgPool2d.html

150x200，对应了不同级别特征图的尺寸；然后分别通过卷积层把通道数分别调整为 512，256，256，128，对应了不同级别特征图的通道。现在，可以进行融合了！图 2.5 中自下而上箭头部分表示在不同级别的特征图中加入高层语义信息。

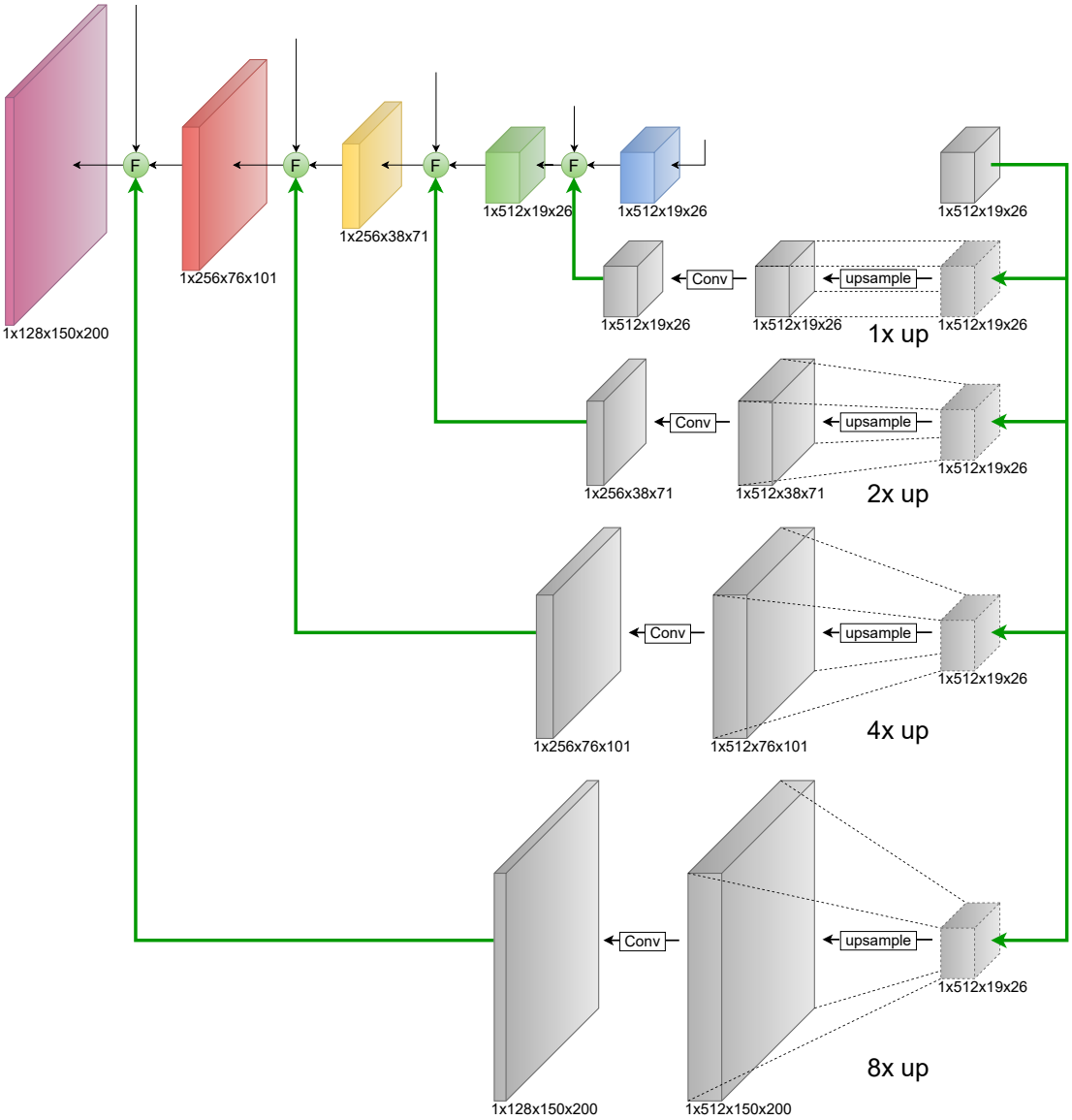


图 2.5　全局引导流（GGFs）

三、特征整合模块（FAM）

前面提到的 GGM 模块是在不同级别的特征图中加入高级语义信息，那么新的问题是如何使来自 GGM 模块的粗糙的高层语义信息和不同尺度的特征图无缝融合？在传统的 U 型网络上采样过程中，相邻特征图间放缩的比例为 2，与来自收缩网络各层的中低级语义信息融合后，再通过一个 3x3 卷积层可以很好地降低上采样带来的混叠效应。但是，GGFs 模块为了使高层语义信息的尺寸和特征图相同，采用了较大比例的上采样（4 倍，8 倍），这很容易导致高级语义信息

和特征图之间产生难以逾越的鸿沟。

为了解决这个问题，PoolNet 提出了特征整合模块（FAM）。如图 2.6 所示，FAM 模块包含 4 个分支，输入特征图首先经过 AvgPool2d[①] 按照 2，4，8 三种比例进行下采样，经过 3x3 卷积层后，再经过对应比例的上采样（双线性插值）恢复到原尺寸。然后，这 4 个分支直接相加，再经过 3x3 卷积层后就做好了与高级语义信息融合的准备。

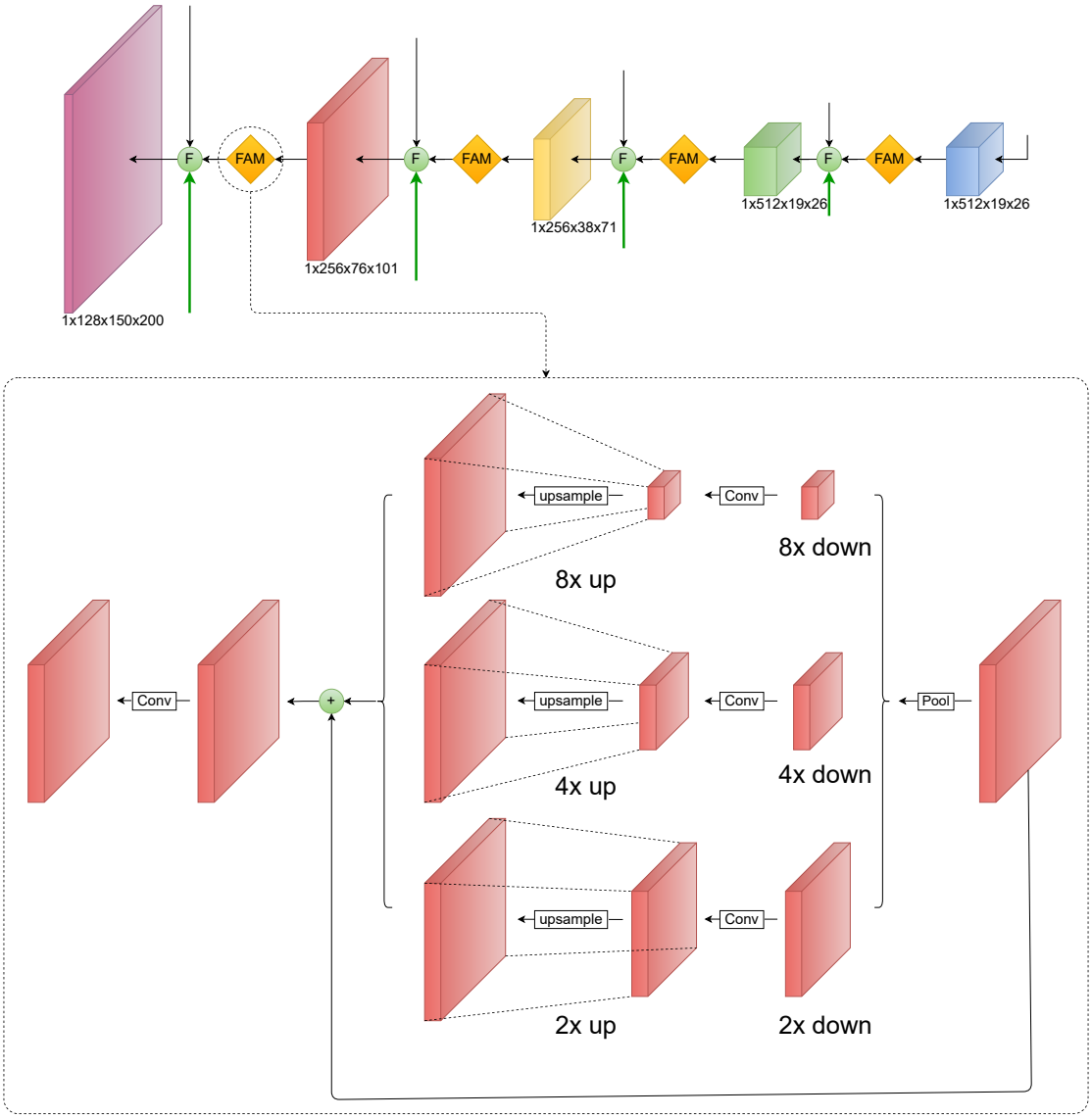


图 2.6　特征整合模块（FAM）

总的来说，FAM 的作用有两点。一是帮助模型降低了上采样导致的混叠效应，尤其是当上采样比例较大时（如 4 倍，8 倍）。二是从不同尺度上刻画显著物体的空间位置，增加了整个网络的感受野。

[①] https://pytorch.org/docs/stable/generated/torch.nn.AvgPool2d.html

第三章　FFmpeg

第一节　FFmpeg 简介

一、FFmpeg 基本组成

FFmpeg 是领先的多媒体框架，“FF”表示“Fast Forward”。FFmpeg 基本组成包含 libavformat，libavcodec，libavfilter，libavdevice，libavutil，libswresample，libswscale 等模块库。libavformat 库能够实现视频格式封装，解封装和转封装，包含了大多数媒体封装格式，如 MP4，FLV，KV，TS 等文件封装格式，还有 RTMP，RTSP，MMS，HLS 等网络协议封装格式。libavcodec 库能够实现视频编码，解码和转码，包含了常用的编解码格式，如 MPEG4，AAC，MJPEG 等。libavfilter 库实现了非常强大的的音视频滤镜功能，可用于多媒体后期的处理与编辑，如添加水印，绿幕抠图，视频裁剪、截图等。libavdevice 是一个包含输入和输出设备的库，可以从常见的媒体输入框架读取数据并将数据渲染到媒体输出框架中。libavutil 库包含了很多简化实用工具，其中包括随机数生成器，数据结构，核心多媒体实用程序等。libswscale 库能够实现图像格式转换，如图像缩放和像素格式转换。libswresample 库提供了高级别的音频重采样 API。

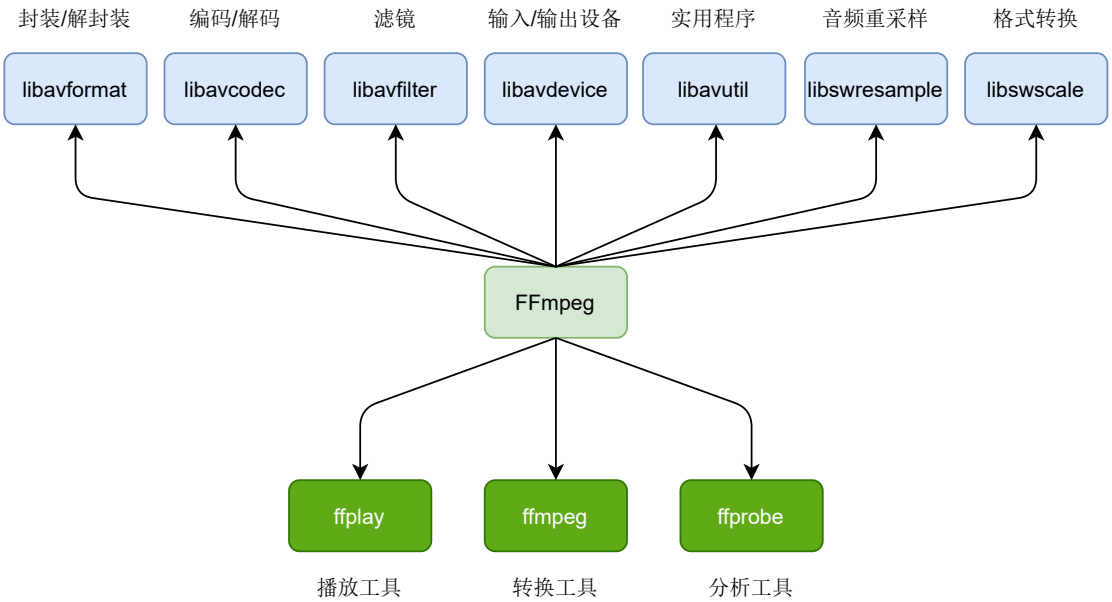


图 3.1　FFmpeg 基本组成

FFmpeg 有着强大的跨平台特性，可以在 Linux，Mac OS X，Microsoft Windows 等平台下编译。编译后会产生上面提到的七个库文件和三种工具。三种工具分别

是 ffmpeg，ffprobe，ffplay。其中，ffmpeg 是多媒体处理工具，可以进行视频转码和转封装，ffprobe 是多媒体信息查看工具，fflay 则是一个简单播放器。

第二节　ffplay 工作流程

一、解码

ffplay 使用多线程进行工作，其中一个是读进程，还有三个分别是视频、音频、字幕线程。如图 3.2 所示，读进程（read_thread()）首先对视频文件进行解封装，并找到媒体流信息，然后循环调用 av_read_frame() 获取压缩编码数据 av_packet，并调用 packet_queue_put() 根据编码数据类型的不同（视频、音频、字幕），将 av_packet 放到对应的 packet_queue 中。视频线程主要有两个函数，get_video_frame() 和 queue_picture()。前者调用 packet_queue_get() 从 packet_queue 中取出压缩编码数据，并使用 libavcodec 库进行解码，然后再调用 avcodec_receive_frame() 接收解码后的帧，后者负责把接收到的帧加入到队列。

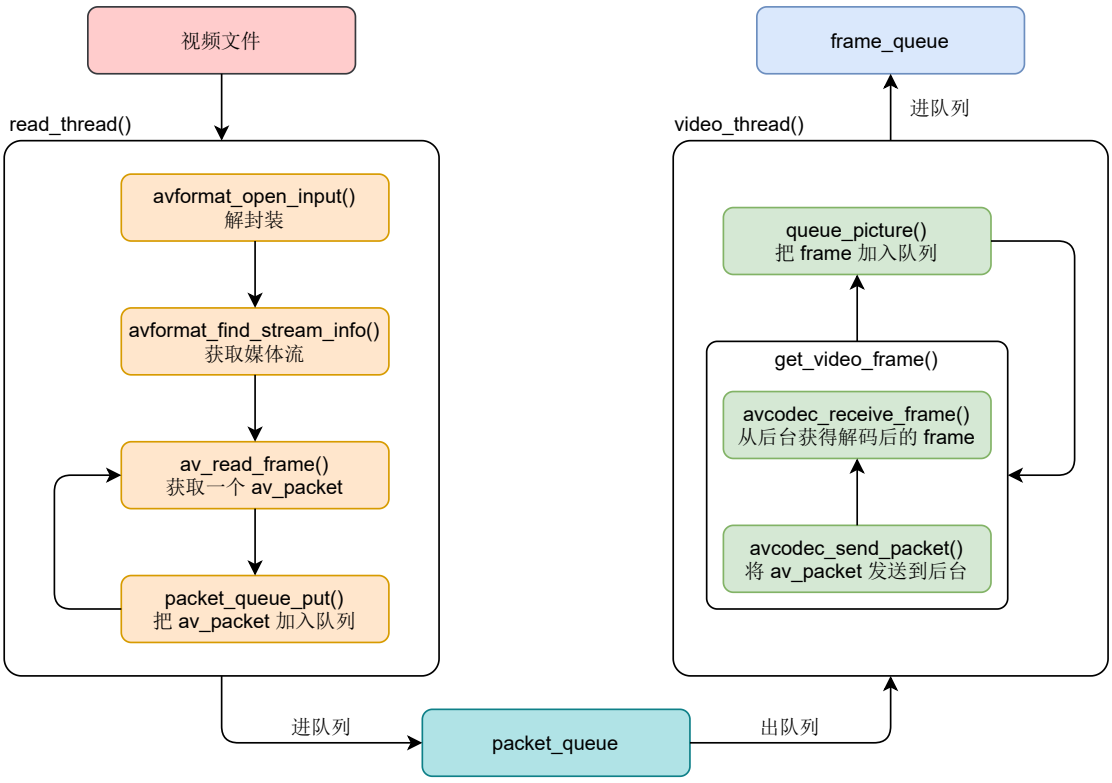


图 3.2　视频解码流程

二、播放

ffplay 调用 video_display() 进行视频播放，核心流程如图 3.3 所示。video_display() 首先调用 SDL_RenderClean() 清空渲染器，然后进入

video_image_display() 函数内部。frame_queue_peek_last() 从解码得到的帧队列中获取帧数据。calculate_display_rect() 计算播放窗口的大小，以保证窗口被拉伸时，视频依旧保证纵横比。upload_texture() 负责把视频帧的像素数据更新到纹理区域。SDL_RenderCopy() 则是把纹理数据复制到渲染器中。最后 SDL_RenderPresent() 将渲染器中的内容在播放窗口中显示。

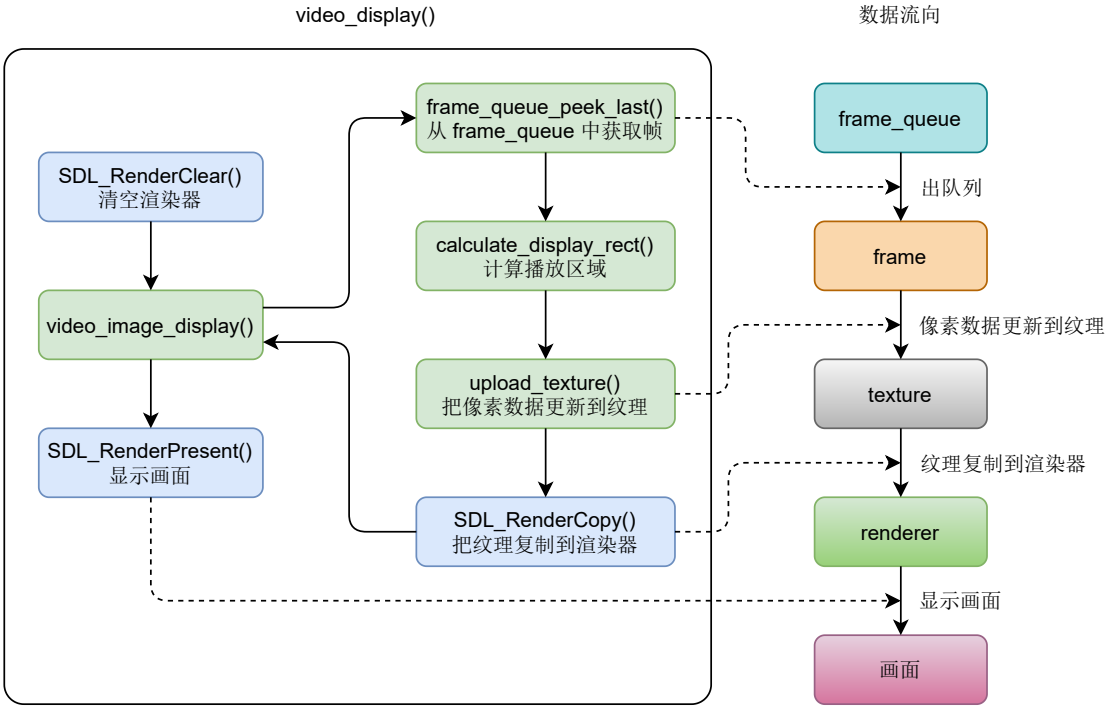


图 3.3　视频播放流程

第四章　神经网络模型嵌入 FFmpeg

第一节　环境搭建

环境为 ubuntu-20.04，机器配置为 nvidia-460+cuda-11.2，FFmpeg 版本为 4.3.1。LibTorch 版本为 1.7.1+cu110。如图 1.1 所示，FFmpeg 提供库文件和头文件，LibTorch 提供 Torch 的 C++ 前端 API。nvidia 驱动和 cuda 则提供 GPU 加速环境。

一、通过源码安装 FFmpeg-4.3.1

1. 安装依赖库

安装 SDL2。ffplay 依赖 SDL，如果没有 SDL，将不会编译产生可执行文件 ffplay。SDL2 可以下载源码编译安装，也可以通过包管理工具 apt 安装（推荐）。

```
sudo apt install libsdl2-dev
```

安装 yasm。yasm 用于对汇编优化的支持，若不需要汇编优化的支持，可在编译选项中关闭 yasm 即可（-disable-yasm）。

```
sudo apt install yasm
```

2. 安装 FFmpeg

下载 ffmpeg-4.3.1 源码并解压。

```
wget http://ffmpeg.org/releases/ffmpeg-4.3.1.tar.gz
tar zxf ffmpeg-4.3.1.tar.gz
```

为了不污染源码，在根目录下新建 build 文件夹，在里面执行 configure 和 make 操作。–enable-shared 是为了生成 so 动态库。这里的 build 文件夹下会生成一些 config 头文件，这是源码里没有的。这些头文件在后面单独编译 ffplay 源码时会用得到。

```
cd ffmpeg-4.3.1
mkdir build && cd build
./../configure --prefix=/usr/local/ffmpeg --enable-
   shared
make
sudo make install
```

然后使用 vim 修改环境变量。

```
// 打开配置文件
vim ~/.bashrc
// 在文件末尾添加
export PATH=/usr/local/ffmpeg/bin:$PATH
export LIBRARY_PATH=/usr/local/ffmpeg/lib:$LIBRARY_PATH
export LD_LIBRARY_PATH=/usr/local/ffmpeg/lib:
   $LD_LIBRARY_PATH
// 使配置文件生效
source ~/.bashrc
```

测试是否安装成功。

```
ffmpeg -version
ffplay -version
```

二、安装 nvidia-460 和 cuda-11.2

配置 GPU 加速环境，可以参考 https://blog.csdn.net/weixin_43742643/article/details/115355545，安装过程不予详述。

三、安装 libtorch

libtorch 是 pytorch 的 C++ 前端，官网提供了编译好的最新版本。我也总结了一份所有版本的集合，地址为 https://blog.csdn.net/weixin_43742643/article/details/114156298 。需要注意的是，libtorch 在 1.5.0 版本之前使用 CXX 11 标准编译，从 1.5.0 版本开始使用 CXX 14 标准编译。这里选择下载 libtorch-1.7.1 的 cu110 版本

```
wget https://download.pytorch.org/libtorch/cu110/
   libtorch-shared-with-deps-1.7.1%2Bcu110.zip
unzip libtorch-shared-with-deps-1.7.1%2Bcu110.zip
mv libtorch libtorch-1.7.1
```

第二节　单独编译 ffplay

CMake 是一个跨平台的开源工具，用于构建，测试和打包项目，CMake 使用独立于编译器的配置文件来控制项目的编译过程，可以生成 makefile 文件。pytorch 官网推荐使用 cmake 工具对 C++ 前端项目进行管理。值得注意的是，ffplay 源码是用 C 语言写的，但是需要在其中加入 C++ 前端模型，因此需要将源码文件 ffplay.c 改为 ffplay.cpp。

一、创建工程

首先，创建根目录 MYPLAY-ROOT，其目录结构如下所示。其中 CMakeLists.txt 即为 cmake 配置文件，其余三个文件均来自源码 ffmpeg-4.3.1/fftools/ 文件夹下。

```
.
├── CMakeLists.txt
├── cmdutils.c
├── cmdutils.h
└── ffplay.cpp
```

二、编辑 CMakeLists.txt

libtorch 作为第三方库需要手动添加。使用 PATHS 关键字指定路径。

```
find_package(Torch REQUIRED PATHS ~/libtorch-1.7.1)
```

ffplay 源码提供了主要的头文件，build 文件夹则提供了一些编译以后生成的 config 头文件。

```
set(FFMPEG_SOURCE ~/ffmpeg-4.3.1)
include_directories(${FFMPEG_SOURCE})
set(FFMPEG_BUILD ~/ffmpeg-4.3.1/build)
include_directories(${FFMPEG_BUILD})
```

添加编译选项 -fpermissive 是因为 ffplay.c 转为 cpp 文件后会严格检查类型转换，导致出现大量 error，该选项可以把此类 error 降为 warning。

```
set(CMAKE_CXX_FLAGS "-fpermissive -w -g")
```

编译可执行文件时链接 ffmpeg 动态库

```
target_link_libraries(myplay ${TORCH_LIBRARIES} -
   lavcodec -lavdevice -lavfilter -lavformat -lavutil -
   lswresample -lswscale -lm -lSDL2
)
```

使用 CXX 14 标准进行编译

```
set_property(TARGET myplay PROPERTY CXX_STANDARD 14)
```

三、修改 cmdutils.h

定位到 249 行，这个函数的形参和 C++ 关键字 class 重名，把形参 class 改成 myclass 即可。

```
# 修改前
void show_help_children(const AVClass *class, int flags
    );
# 修改后
void show_help_children(const AVClass *myclass, int
    flags);
```

四、修改 ffplay.cpp

首先增加 Torch 头文件

```
#include <torch/torch.h>
#include <torch/script.h>
```

ffmpeg 是用 C 写的，由于现在是 cpp 文件，需要将与 ffmpeg 相关的头文件使用关键字 extern C 包裹。头文件 cmdutils.h 也要包裹进去。

```
#include "libavutil/avstring.h"
#include "libavutil/eval.h"
#include "libavutil/mathematics.h"
#include "libavutil/pixdesc.h"
#include "libavutil/imgutils.h"
#include "libavutil/dict.h"
#include "libavutil/parseutils.h"
#include "libavutil/samplefmt.h"
#include "libavutil/avassert.h"
#include "libavutil/time.h"
#include "libavutil/bprint.h"
#include "libavformat/avformat.h"
#include "libavdevice/avdevice.h"
#include "libswscale/swscale.h"
#include "libavutil/opt.h"
#include "libavcodec/avfft.h"
#include "libswresample/swresample.h"

#if CONFIG_AVFILTER
# include "libavfilter/avfilter.h"
# include "libavfilter/buffersink.h"
# include "libavfilter/buffersrc.h"
#endif

#include "cmdutils.h"
}
```

枚举类型 SHOW_MODE 定义在结构体 VideoState 内部，由于现在是 cpp 文

件，使用时需要加上前缀 VideoState::ShowMode:: 。为了方便起见，直接添加宏定义如下（位置必须放在结构体 VideoState 定义之后）：

```
#define ShowMode VideoState::ShowMode
#define SHOW_MODE_NONE ShowMode::SHOW_MODE_NONE
#define SHOW_MODE_VIDEO ShowMode::SHOW_MODE_VIDEO
#define SHOW_MODE_WAVES ShowMode::SHOW_MODE_WAVES
#define SHOW_MODE_RDFT ShowMode::SHOW_MODE_RDFT
#define SHOW_MODE_NB ShowMode::SHOW_MODE_NB
```

定位到函数 opt_show_mode()，其内部调用的函数 parse_number_or_die() 的返回值是 double 类型，C++ 无法将其转化为 VideoState::ShowMode 枚举类型。但 C++ 是可以把整形识别为枚举类型的，因此可以强制类型转换把 double 转为 int。

```
show_mode = !strcmp(arg, "video") ? SHOW_MODE_VIDEO :
   !strcmp(arg, "waves") ? SHOW_MODE_WAVES :
   !strcmp(arg, "rdft" ) ? SHOW_MODE_RDFT :
   (int)parse_number_or_die(opt, arg, OPT_INT, 0,
      SHOW_MODE_NB-1);
```

第三节　在 ffplay 中嵌入模型

这部分的主要内容是介绍将神经网络模型嵌入到 ffplay.cpp 中。ffplay 把视频分解为帧，每一帧的处理都在 video_image_display() 函数中，该函数又调用 upload_texture() 将帧的像素数据更新到纹理。如图 4.1 所示，可以在 upload_texture() 函数中调用神经网络模型，对帧画面进行处理，神经网络模型的初试化则可以放在 main() 中。

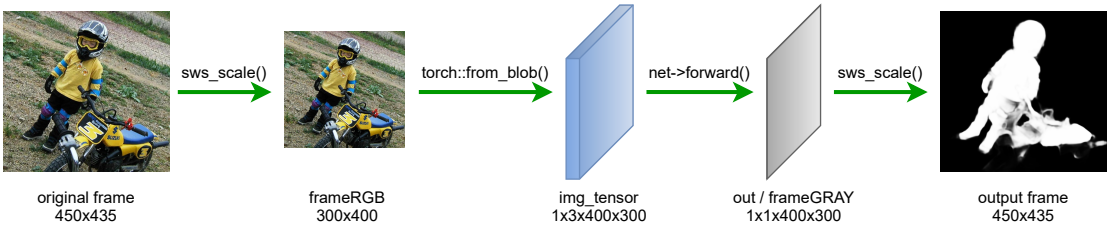


图 4.1　帧数据处理过程

一、神经网络模型初始化

神经网络模型的初始化在 main() 中完成，这里介绍两种方式，见图 4.2。第一种是用 TorchScript 工具导出模型 poolnet.pt，模型中不仅包含各参数的值，还包括神经网络各层的结构，因此可以直接在 C++ 中生成神经网络。第二种是使用 C++ 前端 API 重新构建神经网络类，然后把类对象实例化，不过其参数权重

依旧从 poolnet.pt 中导入。这种方法工程量相对较多。

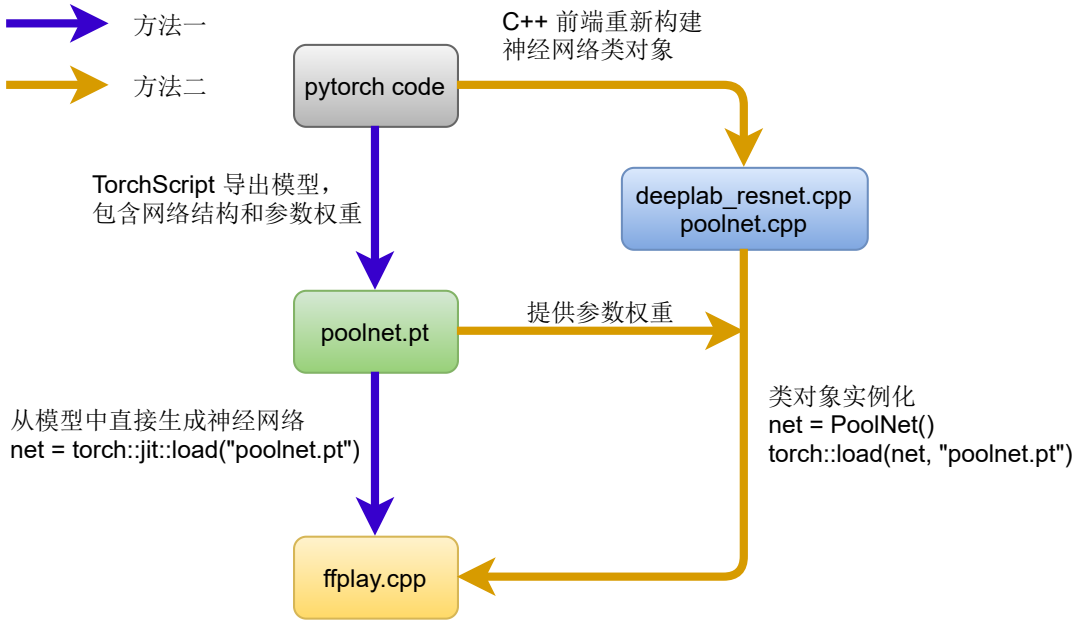


图 4.2　在 ffplay 中嵌入模型

1. 从参数模型导入神经网络

使用 jit::load 直接从参数模型 poolnet.pt 中导入神经网络。

```
net = torch::jit::load("../models/poolnet.pt");
net.to(torch::kCUDA);
net.eval();
```

2. 神经网络类实例化

PoolNet() 是使用 Torch 的 C++ 前端 API 复现的神经网络类。通过 load 可以加载训练好的参数模型。

```
net = PoolNet();
torch::load(net, "../models/poolnet.pt");
net->to(torch::kCUDA);
net->eval();
```

二、原始帧数据转为 RGB 格式

视频帧的格式一般为 YUV 格式，而神经网络模型的输入一般为三通道的 RGB 数据，因此，需要将原始帧数据转为 RGB 格式。frameRGB 的类型是 AVFrame *，是帧指针，使用 av_frame_alloc() 进行初始化。frameRGB 的宽和高应满足神经网络模型的要求，其格式为 RGB。buffer 申请的空间用于 frameRGB 存储数据。这样即完成了 RGB 帧的构建。

```
frameRGB = av_frame_alloc();
frameRGB->width = input_image_size;
frameRGB->height = input_image_size;
frameRGB->format = AV_PIX_FMT_RGB24;
numBytes = avpicture_get_size(frameRGB->format,
   frameRGB->width, frameRGB->height);
buffer = (uint8_t *)av_malloc(numBytes*sizeof(uint8_t))
   ;
avpicture_fill((AVPicture *)frameRGB, buffer, frameRGB
   ->format, frameRGB->width, frameRGB->height);
```

frame 是原始帧，frameRGB 帧用来存储转换结果，*img_convert_ctx 用来存储转换前后的相关信息。sws_scale() 函数则负责转换。

```
*img_convert_ctx = sws_getCachedContext(*
   img_convert_ctx,
   frame->width, frame->height, frame->format,
   frameRGB->width, frameRGB->height, frameRGB->format,
   sws_flags, NULL, NULL, NULL);
if (*img_convert_ctx != NULL) {
   uint8_t *pixels[4];
   int pitch[4];
   if (!SDL_LockTexture(*tex, NULL, (void **)pixels,
      pitch)) {
      sws_scale(*img_convert_ctx, (const uint8_t *
         const *)frame->data, frame->linesize,
         0, frame->height, frameRGB->data, frameRGB->
            linesize);
      SDL_UnlockTexture(*tex);
   }
}
```

三、从 RGB 帧中读取数据

frameRGB->data[0] 为帧数据首地址，from_blob() 函数可以从外部内存创建一个张量。premute() 用于调整张量形状为 {1,3,frameRGB->height,frameRGB->width}，toType() 再把类型转为 float。

```
torch::Tensor img_tensor = torch::from_blob(frameRGB->
   data[0], {1, frameRGB->height, frameRGB->width, 3},
   torch::kByte).to(torch::kCUDA);
img_tensor = img_tensor.permute({0,3,1,2}).toType(torch
   ::kFloat);
```

四、利用神经网络进行推理

如果神经网络是从参数模型导入，

```
torch::Tensor out = net.forward({img_tensor}).toTensor
    ();
```

如果神经网络是从类对象构造，

```
torch::Tensor out = net->forward(img_tensor);
```

五、处理输出结果

out 的形状是 {1,1,frameRGB->height,frameRGB->width}，需要转化为{frameRGB->height,frameRGB->width}。to(torch::kCPU) 负责把张量从 GPU 转移到 CPU 上。

```
out = out.squeeze().sigmoid().mul(255.0).toType(torch::
    kByte).to(torch::kCPU);
```

六、将结果转换到原始帧中

out.data_ptr() 是张量数据的首地址，memcpy() 将数据从张量复制到 frameRGB 帧，然后再通过 sws_scale() 转换为原始帧的大小和格式。

```
auto frameGRAY = out;
int frameGRAY_width = frameRGB->width;
int frameGRAY_height = frameRGB->height;
*img_convert_ctx = sws_getCachedContext(*
    img_convert_ctx,
    frameGRAY_width, frameGRAY_height, AV_PIX_FMT_GRAY8,
    frame->width, frame->height, frame->format,
    sws_flags, NULL, NULL, NULL);
if (*img_convert_ctx != NULL) {
    uint8_t *pixels[4], *frameGRAY_data[8];
    int pitch[4], frameGARY_linesize[8];
    frameGRAY_data[0] = out.data_ptr();
    frameGARY_linesize[0] = frameGRAY_width;
    if (!SDL_LockTexture(*tex, NULL, (void **)pixels,
        pitch)) {
        sws_scale(*img_convert_ctx, (const uint8_t *
            const *)frameGRAY_data, frameGARY_linesize,
            0, frameGRAY_height, frame->data, frame->
                linesize);
        SDL_UnlockTexture(*tex);
    }
}
```

七、运行

新建目录 MYPLAY-ROOT/build，执行

```
cmake ..
make
myplay -v quiet ../videos/test.gif
```

运行结果如图 4.3 所示

图 4.3　运行结果

第五章　结果与结论

第一节　结果

对于尺寸为 256x256 的视频输入，两种方法处理前 10 帧画面的时间如图 5.1 所示。其中方法一表示使用 TorchScript 工具将模型导出，在 C++ 前端直接调用模型；方法二是使用 LibTorch 将神经网络重新构建为类对象，在 C++ 前端把类对象实例化。可以看出，方法一（蓝色）在处理前两帧画面时花费的时间较久，第一帧花费 250 ms，第二帧则需要 1000 ms，之后稳定在 84 ms。方法二（橙色）第一帧花费 120 ms，第二帧需要 100 ms，之后稳定在 86 ms。至于原因可能涉及到 TorchScript 底层代码的实现，暂未得知。

由于两种方法的网络权重参数均是从同一个文件中加载，因此输出的图像完全相同。

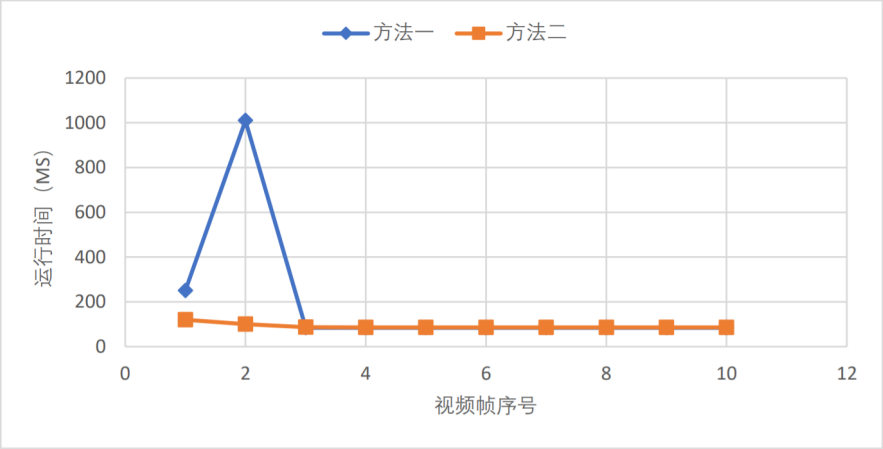
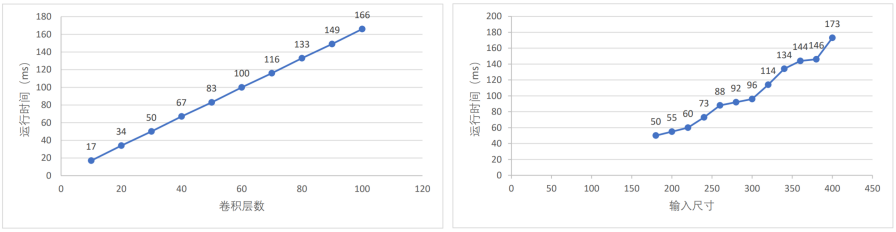


图 5.1　前 10 帧画面运行时间

第二节　分析

每帧图像的处理时间主要与网络结构深度和输入图像尺寸有关。如图 5.2a，对于固定尺寸的输入，随着神经网络深度的加深，处理时间几乎是线性增长；另一方面，如图 5.2b，对于固定深度的神经网络，随着输入尺寸的增加，处理时间也在逐渐增长。对于本文中嵌入的网络 PoolNet 来说，其卷积层数为 92，输入尺寸大小为 256x256，因此每帧处理时间较慢。

人眼识别连贯图像的速度是每秒 24 帧，也就是说每帧处理的时间应低于 40ms。就目前的 256x256 的输入来看，每帧的处理时间只能达到 84ms，人眼观看视频时会有明显的卡顿。虽然可以尝试缩小图像尺寸以缩短运行时间，但是像



(a) 处理时间与卷积层数的关系。　(b) 处理时间与输入尺寸的关系。

图 5.2　处理时间的影响因素

素数据的丢失会导致显著性检测算法难以捕捉全局信息。如图 5.3 所示，随着图片尺寸的缩小，运行效果越来越差。

(a) 原图　(b) 260　(c) 240　(d) 220　(e) 200　(f) 180

图 5.3　缩小尺寸后的运行结果

第三节　总结与展望

为了增加 FFmpeg 对基于神经网络的图像处理算法的支持，本文尝试了将显著性检测算法嵌入到 FFmpeg 的播放工具 ffplay 中并单独编译 ffplay 的方法，并介绍了两种调用神经网络的方式。由于神经网络深度和输入尺寸的限制因素，算法每帧的处理时间达到 84ms，帧率为 12fps 左右。本次实验存在下面几点不足，可以继续探究和改进。

第一点，PoolNet 在 pytorch 环境下，对于 300x400 的输入可以达到 30fps。但是，在 C++ 环境下，对于 256x256 的输入，仅能达到 12fps。可以继续深入探究产生这种差别的原因。

第二点，实验室中的设备有 3 颗 GPU，但运行时仅使用一颗。之后可以尝试将图像切分，使用多颗 GPU 并行处理，然后再把结果合在一起。理论上这样可以加快处理时间。

第三点，本文的工作是将神经网络算法嵌入到 FFmpeg 的播放工具 ffplay 中，虽然方便测试并且能够运行，但是不够灵活，导致 ffplay 的功能比较单一。之后可以考虑把多种不同的神经网络算法集成到 FFmpeg 的滤镜功能中，从而根据不同的目的，使用对应的命令行参数调用对应的算法，更加方便灵活。

参 考 文 献

[1] LIU J J, HOU Q, CHENG M M, et al. A simple pooling-based design for real-time salient object detection[C/OL]//The IEEE Conference on Computer Vision and Pattern Recognition. 2019: 3912-3921. DOI: 10.1109/CVPR.2019.00404.

[2] QIN X, ZHANG Z, HUANG C, et al. Basnet: Boundary-aware salient object detection[C/OL]//The IEEE Conference on Computer Vision and Pattern Recognition. 2019: 7471-7481. DOI: 10.1109/CVPR.2019.00766.

[3] FENG M, LU H, DING E. Attentive feedback network for boundary-aware salient object detection[C/OL]//The IEEE Conference on Computer Vision and Pattern Recognition. 2019: 1623-1632. DOI: 10.1109/CVPR.2019.00172.

[4] ITTI L, KOCH C, NIEBUR E. A model of saliency-based visual attention for rapid scence analysis[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1998, 20(11): 1254-1259.

[5] HOU X, ZHANG L. Saliency detection: A spectral residual approach[C/OL]// IEEE Conference in Computer Vision and Pattern Recognition: volume 2007. 2007. DOI: 10.1109/CVPR.2007.383267.

[6] OLSHAUSEN B A, FIELD D J. Emergence of simple-cell receptive field properties by learning a sparse code for natural images[J]. Nature, 1996, 381(6583): 607-609.

[7] RONNEBERGER O, FISCHER P, BROX T. U-net: Convolutional networks for biomedical image segmentation[C/OL]//International Conference on Medical Image Computing and Computer-Assisted Intervention: volume 9351. 2015: 234-241. DOI: 10.1007/978-3-319-24574-4_28.

[8] CHEN L C, PAPANDREOU G, KOKKINOS I, et al. Semantic image segmentation with deep convolutional nets and fully connected crfs[C]//International Conference on Learning Representations. 2015.

[9] HE K, ZHANG X, REN S, et al. Deep residual learning for image recognition[C]// IEEE Conference on Computer Vision and Pattern Recognition. 2016: 770-778.

[10] ZHAO H, SHI J, QI X, et al. Pyramid scene parsing network[C/OL]//IEEE Conference on Computer Vision and Pattern Recognition. 2017: 6230-6239. DOI:

10.1109/CVPR.2017.660.

[11] ZHOU B, KHOSLA A, LAPEDRIZA A, et al. Object detectors emerge in deep scene cnns[J]. CoRR. arXiv, 2014.