Coarse-to-Fine Hyper-Prior Modeling for Learned Image Compression

Hu Y, Yang W, Liu J.

2020

摘要

相较于现有的混合编解码器，基于机器学习的图像压缩方法现在在压缩率方面表现出优异的性能。能够用于图像压缩的传统基于学习的方法利用超先验和空间上下文模型来促进概率估计。然而这样的模型在对长期依赖性建模方面存在局限，并且不能充分挤压图像中的空间冗余。因此在本文中，作者提出了一个层次化的超先验粗到细框架，以对图像进行全面分析，并更有效地减少空间冗余，从而显著改善图像压缩的失真率性能。还设计了信号保持的超变换，以深入分析潜在表示，同时提出了信息聚合重建子网络，以最大限度地利用辅助信息进行重建。实验结果表明，所提出的网络能够有效地减少图像中的冗余，并显著改善速率失真性能，特别是对于高分辨率图像。

关键词：多层次超先验；信号保持超变换；信息聚合；图像压缩

1 引言

图像压缩是媒体共享和存储中最基本的技术之一。随着 8K 流媒体和虚拟现实（VR）等高分辨率视觉应用的蓬勃发展，获取高效压缩和高质量的图像/视频以适应有限的硬件资源是一个关键问题。

近年来，提出了利用神经网络的端到端优化图像压缩方法，以利用深度学习的强大建模能力来促进图像压缩方法的发展。这些方法通常利用端到端可训练模型来共同优化速率和失真。用于图像压缩的最先进网络 [6] [9] 包括广义分裂归一化（GDN）[2] 或设计非局部模型 [7] 在分析和合成变换中，以减少图像的空间冗余。熵编码中包括有超先验和上下文模型的条件熵模型。实现的学习图像压缩方法在 PSNR-比特率和 MS-SSIM-比特率度量两方面都优于现有的混合框架，如 BPG（Bellard. 2014）。

现有的基于学习的方法与上下文模型和超先验仍然忽略了一些问题。首先，在潜在表示中对待编码元素的概率的估计取决于先前解码元素的局部块，这限制了长期条件概率估计的准确性。而且，目前的大规模并行计算设备（例如 GPU）几乎无法加速这种估计。其次，由于浅层网络的限制，现有的分析和合成转换无法在保持重建质量的同时最大程度地减少空间冗余。第三，使用超先验传输的信息没有得到规范化和利用。虽然这部分信息被编码在比特流中，但并没有用于图像的重建。此外，虽然超先验和上下文模型的组合改善了整体的速率失真性能，但它们如何合作和相互作用还没有得到很好的研究。

2 相关工作

在端到端学习的图像压缩中，有两个基本的问题，首先如何设计分解变换以去相关信号，其次是如何建立有效熵编码的概率模型。过去几年中提出了许多基于深度学习的图像压缩方法，以研究这些问题的解决方案，从而改进了学习方法的压缩性能。从网络架构设计的角度来看，这些方法大致可分为两类：全卷积网络和循环模型。

2.1 循环模型（recurrent models）

循环模型方法以渐进方式对图像进行编码，在每次循环中，会生成固定维度的代码以对图像或不能在先前步骤中捕获的残余信号进行编码 [1] [5] [11] [12]。循环模型对每个步骤的表示的维数施加约束。在这种低维度中，各元素接近独立分布。此外，在 [12] 中，设计了基于神经网络的二进制化器，以对编码符号序列中的上下文依赖性进行建模，以进一步挤出冗余并促进熵编码。借助之前步骤进行预测 [1] 进一步增强了去相关的能力。然而，循环结构在复杂性方面存在问题，特别是对于更高范围的比特率。因此，更近期的工作集中在完全卷积结构上。

2.2 全卷积网络（Fully convolutional networks）

全卷积网络 [3] [4] [6] [8] [10] 受到速率失真约束的训练，以在比特率和重建质量之间进行折衷。每个训练模型对应于一个拉格朗日系数 λ，用于控制这种平衡。因此，需要训练多个模型以满足不同速率失真折衷的需求。

一个典型的全卷积框架通常使用 GDN 来进行局部冗余移除。GDN 被应用为分析变换中多个层的激活函数 [2] [10]。这种规范化一致地减少了每个层的激活的空间冗余，以及待编码的潜在表示。与循环模型不同，对图像进行编码的比特率通过评估量化编码的联合熵并将其作为损失函数中的一个项来最小化。对于概率估计，除了基本的因子模型，[4] 中提出了一个可训练的概率密度模型，用于测量编码信息的分布以进行熵计算和算术编码。进一步假设量化编码遵循高斯分布。模型的规模由超先验编码器编码，进一步减少了空间相关性。[9] 继承并概括了概率模型的超参数，既包括量化编码，也包括相应的解码环境。这两项工作是首次利用深度神经网络架构实现了优于 BPG 4:2:0 和 BPG 4:4:4（Bellard. 2014）的性能。

3 本文方法

3.1 本文方法概述

本文旨在解决限制，并探索构建一个有效和高效的框架，该框架整合了超级级先验和图像的潜在表示，以进行高效压缩。根据作者的分析，上下文模型捕获的上下文依赖性也可以建模为更高级别的潜在表示，而后者可以完全并行化，并包括更广泛的上下文。上下文的扩展提供了减少潜在表示中的基础空间冗余能力。本文方法通过堆叠多层潜在表示，每一层都是对前一层的进一步抽象，并经过训练以逼近分解条件熵模型。通过这种方式，总比特率可以进一步减少。因此，编解码器的速率失真性能得到改善。最后作者还设计了一个信息聚合重建子网络，以充分利用不同层次的潜在表示。本文方法框架示意图如图 1所示：

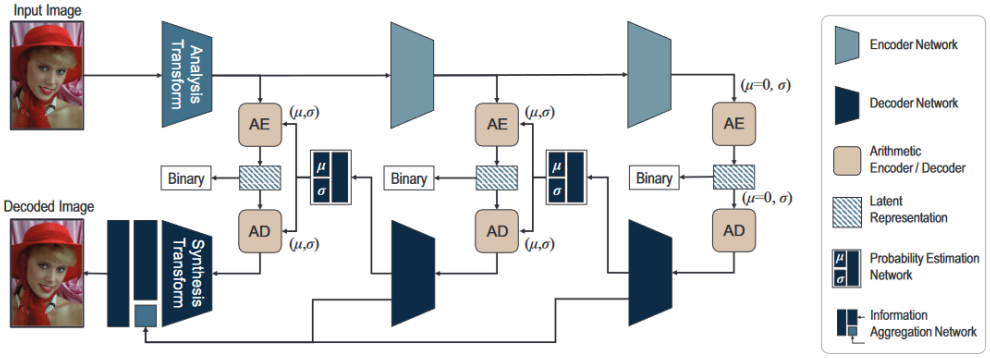

Figure 1: Overall architecture of the multi-layer image compression framework. The probability distribution of the most inner layer of hyper-prior is approximated with a zero-mean Gaussian distribution, where the scale values σ are channel-wise independent and spatially shared trainable parameters.

图 1. 方法示意图

3.2 编码流程

输入图像通过分析变换网络生成潜在表示 z_3，再将潜在表示输入两层分析变换网络产生高级表示 z_2 和超级表示 z_1，接着对 z_1、z_2、z_3 进行量化，将 z_1 的均值设为 0，方差设为 1。分别对三层分析变换网络输出的 $z_{rounded}$、均值、方差进行算术编码形成二进制文件。其算术编码使用 0 阶算术编码，每个符号的概率只与其自身有关。

3.3 解码流程

1、解码器接收编码器传输过来的二进制文件，从中读取二进制流，设置均值 0 方差 1 通过算术解码器从二进制流中恢复 $z_{1_{rounded}}$，再通过合成变换生成高级表示 h_1。

2、h1 通过概率估计网络获得 z2 的均值和方差，通过算术解码器从二进制流中恢复 $z_{2_{rounded}}$，再通过合成变换生成高级表示 h_2。

3、h2 通过概率估计网络获得 z3 的均值和方差，通过算术解码器从二进制流中恢复 $z_{3_{rounded}}$，再通过合成变换生成主要表示 p_f。

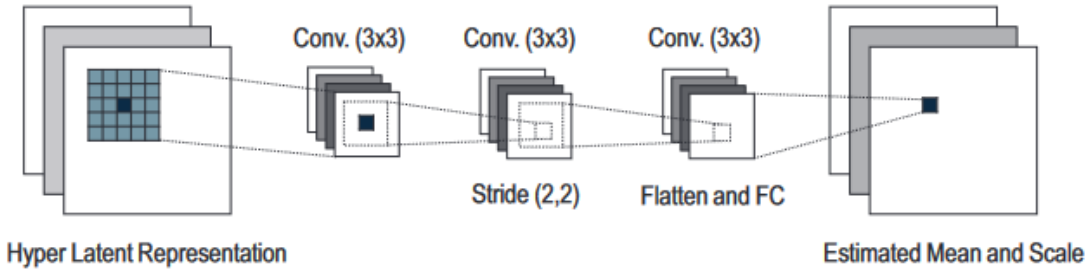

图 2. 概率估计网络结构示意图

4、最后使用信息聚合网络利用 p_f、h_2、h_1 进行图像重建。

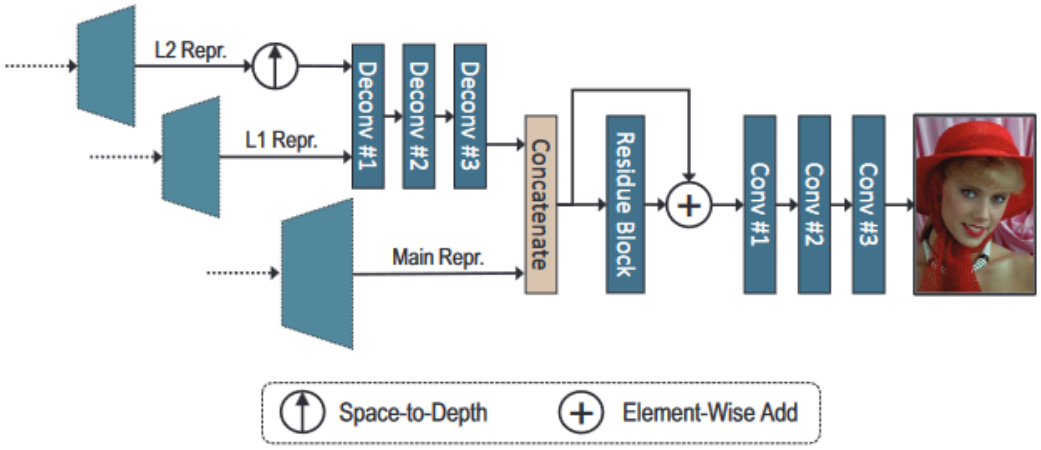

图 3. 信息聚合网络结构示意图

4 复现细节

4.1 与已有开源代码对比

本论文提供了模型源代码，通过阅读论文和思考后，我认为论文中的变换网络模型结构简单，可以尝试通过修改变换网络和高级变换网络的结构对模型进行改进，以期实现提高本论文所提出的方法框架在图像压缩方面的性能提升。因此在下面我将修改前的模型代码和修改后的模型代码展示出来。

4.1.1 分析变换模型和合成变换模型

在论文方法中，分析变换模型被用于在编码阶段最开始的时候将输入图像分析变换为最基础的潜在表示 z_3，在分析变换过程中将原始输入图像映射到潜在表示中使得图像中的大量冗余信息被压缩，提取出最重要的特征表示。合成变换模型被用在解码阶段最后的时候从解码器输出的潜在表示重建为输出图像。以下是原文中的分析变换模型和合成变换模型的具体代码细节。

```
    # Main analysis transform model with GDN
class analysisTransformModel(nn.Module):
    def __init__(self, in_dim, num_filters, conv_trainable=True):
        super(analysisTransformModel, self).__init__()
        self.t0 = nn.Sequential(
        )
        self.transform = nn.Sequential(
            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(in_dim, num_filters[0], 5, 2, 0),
            GDN(num_filters[0]),

            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(num_filters[0], num_filters[1], 5, 2, 0),
            GDN(num_filters[1]),

```

```
            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(num_filters[1], num_filters[2], 5, 2, 0),
            GDN(num_filters[2]),

            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(num_filters[2], num_filters[3], 5, 2, 0),
        )

    def forward(self, inputs):
        x = self.transform(inputs)
        return x

# Main synthesis transform model with IGDN
class synthesisTransformModel(nn.Module):
    def __init__(self, in_dim, num_filters, conv_trainable=True):
        super(synthesisTransformModel, self).__init__()
        self.transform = nn.Sequential(
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.ConvTranspose2d(
                in_dim, num_filters[0], 5, 2, 3, output_padding=1),
            IGDN(num_filters[0]),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.ConvTranspose2d(
                num_filters[0], num_filters[1], 5, 2, 3, output_padding=1),
            IGDN(num_filters[1]),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.ConvTranspose2d(
                num_filters[1], num_filters[2], 5, 2, 3, output_padding=1),
            IGDN(num_filters[2])
        )
        # Auxiliary convolution layer: the final layer of the synthesis model.
        # Used only in the initial training stages, when the information
        # aggregation reconstruction module is not yet enabled.
        self.aux_conv = nn.Sequential(
          nn.ZeroPad2d((1,0,1,0)),
          nn.ConvTranspose2d(num_filters[2], num_filters[3], 5, 2, 3,
          output_padding=1)
        )

    def forward(self, inputs):
        x = self.transform(inputs)
        y = self.aux_conv(x)
        return x, y
```

通过对这两个模型的结构进行改进，稍微提高模型的复杂程度，使得分析变换模型能够更好的获取输入图像中的更高维度的信息，以此尝试提高模型压缩的能力。使得合成变换模型能够最大限度地使用潜在表示中的信息，提高模型重建图像的能力。修改后的分析变换模型和合成变换模型如下所示。

```
    class analysisTransformModel(nn.Module):
    def __init__(self, in_dim, num_filters, conv_trainable=True):
        super(analysisTransformModel, self).__init__()
        self.transform = nn.Sequential(
            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(in_dim, num_filters[0], 5, 2, 0),
            GDN(num_filters[0]),

            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(num_filters[0], num_filters[0], 5, 2, 0),
            GDN(num_filters[0]),

            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(num_filters[0], num_filters[1], 5, 2, 0),
            GDN(num_filters[1]),

            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(num_filters[1], num_filters[1], 5, 2, 0),
            GDN(num_filters[1]),

            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(num_filters[1], num_filters[2], 5, 2, 0),
            GDN(num_filters[2]),

            nn.ZeroPad2d((1, 2, 1, 2)),
            nn.Conv2d(num_filters[2], num_filters[3], 5, 2, 0),
        )

    def forward(self, inputs):
        x = self.transform(inputs)
        return x

class synthesisTransformModel(nn.Module):
    def __init__(self, in_dim, num_filters, conv_trainable=True):
        super(synthesisTransformModel, self).__init__()
        self.transform = nn.Sequential(
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.ConvTranspose2d(
                in_dim, num_filters[0], 5, 2, 3, output_padding=1),
            IGDN(num_filters[0]),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.ConvTranspose2d(
                num_filters[0], num_filters[0], 5, 2, 3, output_padding=1),
            IGDN(num_filters[0]),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.ConvTranspose2d(
                num_filters[0], num_filters[1], 5, 2, 3, output_padding=1),
            IGDN(num_filters[1]),
```

```
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.ConvTranspose2d(
                num_filters[1], num_filters[1], 5, 2, 3, output_padding=1),
            IGDN(num_filters[1]),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.ConvTranspose2d(
                num_filters[1], num_filters[2], 5, 2, 3, output_padding=1),
            IGDN(num_filters[2])
        )
        self.aux_conv = nn.Sequential(
          nn.ZeroPad2d((1,0,1,0)),
          nn.ConvTranspose2d(num_filters[2], num_filters[3], 5, 2, 3,
          output_padding=1)
        )

    def forward(self, inputs):
        x = self.transform(inputs)
        y = self.aux_conv(x)
        return x, y
```

4.1.2 信号保持超变换模型

由于单层分析变换在所提出的架构中存在两个方面的问题：

1）分析变换固定通道数并对特征图进行下采样，降低了维度。

2）在分析变换的开头或综合转换的结尾结合大的卷积核和 ReLU 会丢失一些未经转换的信息，限制了容量。

因此作者添加了一个信号保持超变换通过保留信息以进行粗细分析，以促进多层结构的实现。以下是原工作中的信号保持超变换模型，这两个模型结构使用了对称设计，能够输出作为模型图形重建过程中的外层的先决条件和重建的辅助信息。

```
    # Hyper analysis transform (w/o GDN)
class h_analysisTransformModel(nn.Module):
    def __init__(self, in_dim, num_filters, strides_list, conv_trainable=True):
        super(h_analysisTransformModel, self).__init__()
        self.transform = nn.Sequential(
            nn.Conv2d(in_dim, num_filters[0], 3, strides_list[0], 1),
            Space2Depth(2),
            nn.Conv2d(num_filters[0]*4, num_filters[1], 1, strides_list[1], 0),
            nn.ReLU(),
            nn.Conv2d(num_filters[1], num_filters[1], 1, 1, 0),
            nn.ReLU(),
            nn.Conv2d(num_filters[1], num_filters[2], 1, 1, 0)
        )

    def forward(self, inputs):
        x = self.transform(inputs)
        return x

```

```
# Hyper synthesis transform (w/o GDN)
class h_synthesisTransformModel(nn.Module):
    def __init__(self, in_dim, num_filters, strides_list, conv_trainable=True):
        super(h_synthesisTransformModel, self).__init__()
        self.transform = nn.Sequential(
            nn.ConvTranspose2d(in_dim, num_filters[0], 1, strides_list[2], 0),
            nn.ConvTranspose2d(
                num_filters[0], num_filters[1], 1, strides_list[1], 0),
            nn.ReLU(),
            nn.ConvTranspose2d(
                num_filters[1], num_filters[1], 1, strides_list[1], 0),
            nn.ReLU(),
            Depth2Space(2),
            nn.ZeroPad2d((0, 0, 0, 0)),
            nn.ConvTranspose2d(
                num_filters[1]//4, num_filters[2], 3, strides_list[0], 1)
        )

    def forward(self, inputs):
        x = self.transform(inputs)
        return x
```

通过对这两个模型的结构进行改进，稍微提高模型的复杂程度，使得高级分析变换模型能够更好的获取潜在表示中的更高维度的信息，以此尝试提高模型压缩的能力。也能够使得高级合成变换模型能够更好地利用输入的高级潜在表示重建输出。修改后的高级分析变换模型和高级合成变换模型如下所示。

```
    class h_analysisTransformModel(nn.Module):
    def __init__(self, in_dim, num_filters, strides_list, conv_trainable=True):
        super(h_analysisTransformModel, self).__init__()
        self.transform = nn.Sequential(
            nn.Conv2d(in_dim, num_filters[0], 3, strides_list[0], 1),
            nn.ReLU(),
            nn.Conv2d(num_filters[0], num_filters[0], 3, strides_list[0], 1),
            nn.ReLU(),
            Space2Depth(2),
            nn.Conv2d(num_filters[0]*4, num_filters[1], 1, strides_list[1], 0),
            nn.ReLU(),
            nn.Conv2d(num_filters[1], num_filters[1], 1, 1, 0),
            nn.ReLU(),
            nn.Conv2d(num_filters[1], num_filters[1], 1, 1, 0),
            nn.ReLU(),
            nn.Conv2d(num_filters[1], num_filters[2], 1, 1, 0)
        )

    def forward(self, inputs):
        x = self.transform(inputs)
        return x

```

```
class h_synthesisTransformModel(nn.Module):
    def __init__(self, in_dim, num_filters, strides_list, conv_trainable=True):
        super(h_synthesisTransformModel, self).__init__()
        self.transform = nn.Sequential(
            nn.ConvTranspose2d(in_dim, num_filters[0], 1, strides_list[2], 0),
            nn.ReLU(),
            nn.ConvTranspose2d(num_filters[0], num_filters[0],
            1, strides_list[2], 0),
            nn.ReLU(),
            nn.ConvTranspose2d(num_filters[0], num_filters[1],
            1, strides_list[1], 0),
            nn.ReLU(),
            nn.ConvTranspose2d(num_filters[1], num_filters[1],
            1, strides_list[1], 0),
            nn.ReLU(),
            nn.ConvTranspose2d(num_filters[1], num_filters[1],
            1, strides_list[1], 0),
            nn.ReLU(),
            Depth2Space(2),
            nn.ZeroPad2d((0, 0, 0, 0)),
            nn.ConvTranspose2d(num_filters[1]//4, num_filters[2],
            3, strides_list[0], 1)
        )

    def forward(self, inputs):
        x = self.transform(inputs)
        return x
```

5 实验结果分析

在原论文方法模型进行上述修改后，通过使用 DIV2K 数据集来训练网络，该数据集包含未经有损压缩的高分辨率图像。还以 0.5 的尺度额外对图像进行降采样来增广训练数据集。

网络经过三个阶段进行训练。在第一阶段预训练主要的分析和合成变换，以实现对图像的良好重建。主要的分析和合成变换是通过嵌入的上下文模型进行预训练的。在第二阶段冲随机初始化超变换网络，并以端到端的方式训练整个模型，不使用上下文模型。最后，原始合成变换的最后一层被信息聚合网络替换，并同时训练整个网络。最后通过结果分析和对比发现除训练时间较原论文方法模型稍长外，在 Kodak 数据集和 Tecnick 数据集上的压缩性能表现相当，并没有明显差距。

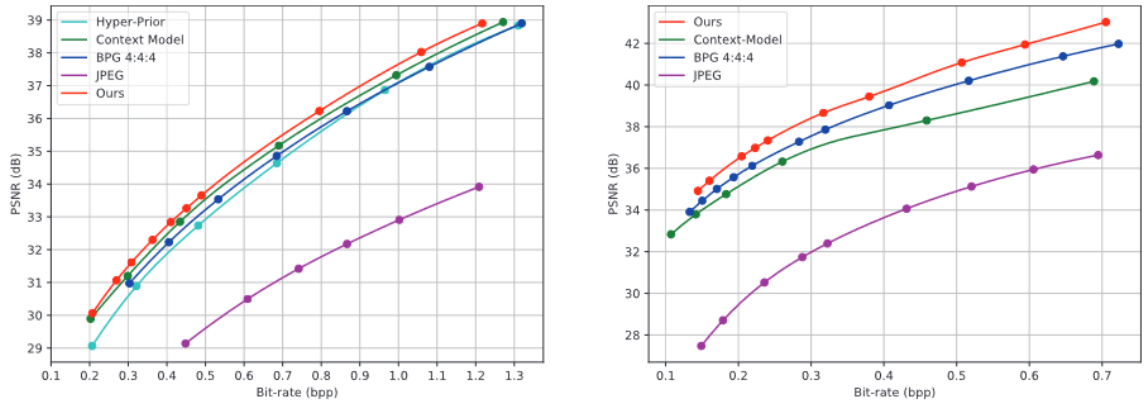

图 4. 实验结果示意

6 总结与展望

在本论文中设计了一种粗到细超先验引导的自动编码器用于图像压缩。该框架旨在将图像分解为潜在表示，其中潜在表示中的元素可以更高效地编码（对于最内层）或有条件地建模（对于其他支持的层）。通过这种方式能更好地近似所要编码图像中像素的联合分布，并分配适当的位来表示图像在位流中。作者提出了信号保持超变换来构建粗到细的框架。通过用提出的结构替换原始变换，减少了信息损失，并允许堆叠更多层的超表示，从而增强了更好地挤压空间冗余的能力。它还促进了信息聚合重建子网络，以充分利用来自所有层的位流来提高重建质量。

虽然尝试对模型结构进行改进的结果没有取得成效，但是我认为这一修改方向是可以对模型性能有所提高的。而修改结果差距不大的原因可能是因为修改后的模型结构相比于原论文方法模型结构在冗余信息压缩能力上并没有很大提升，因此可以进一步改用提取特征能力更强的网络结构作为变换模型和超变换模型，可以有效提升此模型图像压缩的能力。

参考文献

[1] Mohammad Haris Baig, Vladlen Koltun, and Lorenzo Torresani. Learning to inpaint for image compression. *Advances in Neural Information Processing Systems*, 30, 2017.

[2] Johannes Ballé. Efficient nonlinear transforms for lossy image compression. In *2018 Picture Coding Symposium (PCS)*, pages 248–252. IEEE, 2018.

[3] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704*, 2016.

[4] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. *arXiv preprint arXiv:1802.01436*, 2018.

[5] Nick Johnston, Damien Vincent, David Minnen, Michele Covell, Saurabh Singh, Troy Chinen, Sung Jin Hwang, Joel Shor, and George Toderici. Improved lossy image compression with priming and spatially adaptive bit rates for recurrent networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4385–4393, 2018.

[6] Jooyoung Lee, Seunghyun Cho, and Seung-Kwon Beack. Context-adaptive entropy model for end-to-end optimized image compression. *arXiv preprint arXiv:1809.10452*, 2018.

[7] Haojie Liu, Tong Chen, Peiyao Guo, Qiu Shen, Xun Cao, Yao Wang, and Zhan Ma. Non-local attention optimized deep image compression. *arXiv preprint arXiv:1904.09757*, 2019.

[8] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Conditional probability models for deep image compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4394–4402, 2018.

[9] David Minnen, Johannes Ballé, and George D Toderici. Joint autoregressive and hierarchical priors for learned image compression. *Advances in neural information processing systems*, 31, 2018.

[10] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017.

[11] George Toderici, Sean M O'Malley, Sung Jin Hwang, Damien Vincent, David Minnen, Shumeet Baluja, Michele Covell, and Rahul Sukthankar. Variable rate image compression with recurrent neural networks. *arXiv preprint arXiv:1511.06085*, 2015.

[12] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 5306–5314, 2017.