# 10.11日做的事

1. 继续学习transformer源码
2. 一篇论文
3. 一篇昨天的论文（pix2pix升级版：High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs）
4. 高级体系结构的第一篇论文一点摘要（**Sparse ReRAM Engine: Joint Exploration of Activation and Weight Sparsity in Compressed Neural Networks**）
5. 有空的话跑一下gatys代码

1 一般来说 K V是差不多的，和Q不同，这就是attention，但是当QKV都一样的时候，着就成为了自注意力机制！

多头注意力内部的矩阵一定是一个方阵

用zip函数组合输入的qkv和线性层

```
query,key value= \
[model(x).view(batch_size,-1,self.head, self.d_k).transpose(1,2) for model , x
in zip(self.linears,(query, key , value))]
```

自己看视频吧方便回忆

3 该论文作者做了如下操作：

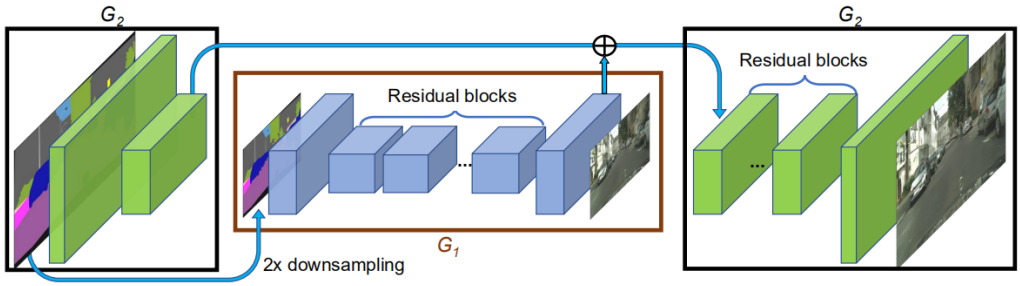


Figure 3: Network architecture of our generator. We first train a residual network $G_1$ on lower resolution images. Then, another residual network $G_2$ is appended to $G_1$ and the two networks are trained jointly on high resolution images. Specifically, the input to the residual blocks in $G_2$ is the element-wise sum of the feature map from $G_2$ and the last feature map from $G_1$.

我们的G的网络架构。我们首先在较低分辨率的图像上训练一个残差网络G1。然后，将另一个残差网络G2附加到G1上，并对高分辨率图像进行联合训练。具体来说，G2中残差块的输入是G2的特征图和G1的最后一个特征图的元素级之和。

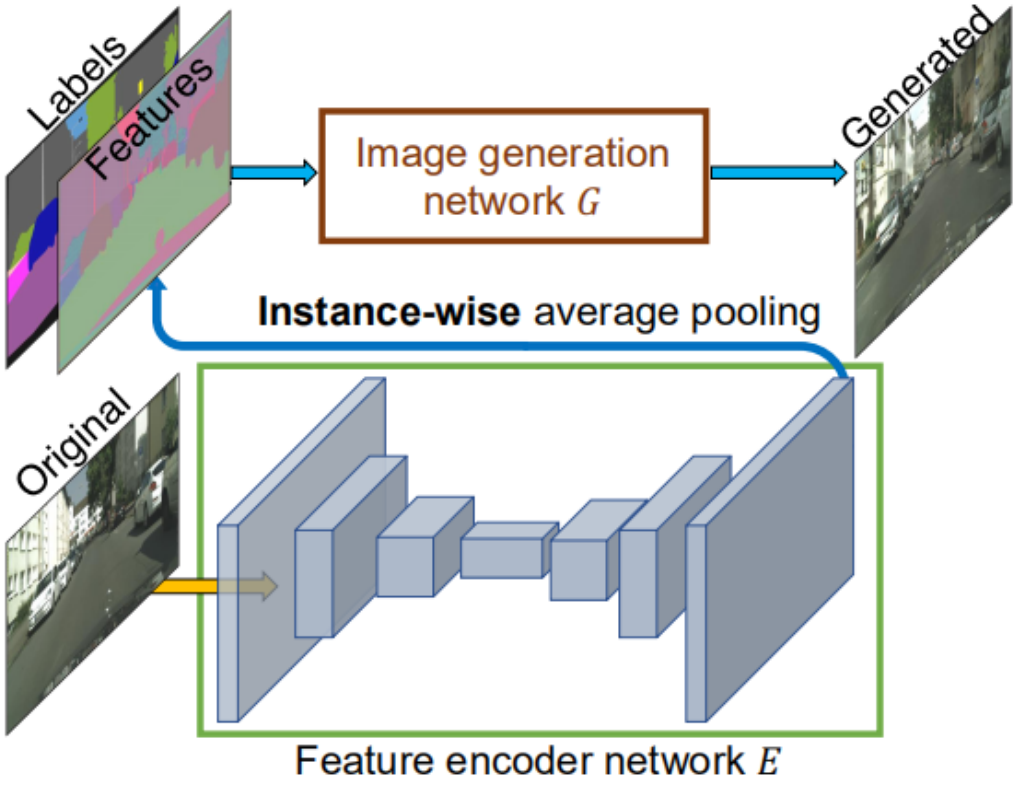


Figure 6: Using instance-wise features in addition to labels for generating images.

总结来说，这篇paper的贡献点如下：

1、很好地解决了GAN难以生成高清图片的问题，可以产生十分真实 2K 高清图片

2、可以通过调整语义分割图片以及选取不同的特征，自由生成图片。

为了使 condition GAN 很好生成高清图片，作者采用了以下方式：

1、使用多层级的 Discriminator，paper中用了3级， D1、D2、D3，对应输入的图像长宽逐级减半。这样输入尺寸大的 D1 可以更注重图像的细节真实度，而输入尺寸小的 D3 可以注重图片的整体真实度。

2、使用多层级的 Generator，在 paper 中用了两级，在 G1 生成小一些的图片后，再由 G2 在 G1 输出的基础上更进一步 refine 生成长宽翻倍的图片，当然，如果想要生成更高分辨率的图像，还可以增加 G3。

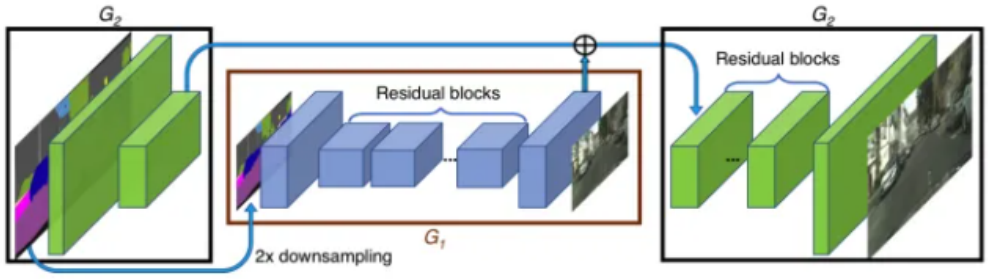


Figure 3: Network architecture of our generator. We first train a residual network $G_1$ on lower resolution images. Then, another residual network $G_2$ is appended to $G_1$ and the two networks are trained jointly on high resolution images. Specifically, the input to the residual blocks in $G_2$ is the element-wise sum of the feature map from $G_2$ and the last feature map from $G_1$.

3、Discriminator 的中间层也具有很好的参考价值，我们可以用 L1 loss 来约束 G，使得 G 产生的图像与真实图像在 Discriminator 的中间层尽可能一致。

4、加入 Perceptual Loss：Uno Whoiam：Perceptual Loss: 提速图像风格迁移1000倍

5、输入除了使用语义分割，还使用了实例分割的信息，具体使用方法是使用将 instance 的信息以带有类别信息的物体边缘图的方式concat进输入。

perceptual loss 的 pytorch 实现

```
import torch
import torch.nn as nn
from torchvision import models
from PIL import Image
from style_transfer_perceptual_loss.image_dataset import get_transform
from src.utils.train_utils import get_device


class Vgg16(nn.Module):
    def __init__(self):
        super(Vgg16, self).__init__()
        features = models.vgg16(pretrained=True).features
```

```
        self.to_relu_1_2 = nn.Sequential()
        self.to_relu_2_2 = nn.Sequential()
        self.to_relu_3_3 = nn.Sequential()
        self.to_relu_4_3 = nn.Sequential()

        for x in range(4):
            self.to_relu_1_2.add_module(str(x), features[x])
        for x in range(4, 9):
            self.to_relu_2_2.add_module(str(x), features[x])
        for x in range(9, 16):
            self.to_relu_3_3.add_module(str(x), features[x])
        for x in range(16, 23):
            self.to_relu_4_3.add_module(str(x), features[x])

        # don't need the gradients, just want the features
        for param in self.parameters():
            param.requires_grad = False

    def forward(self, x):
        h = self.to_relu_1_2(x)
        h_relu_1_2 = h
        h = self.to_relu_2_2(h)
        h_relu_2_2 = h
        h = self.to_relu_3_3(h)
        h_relu_3_3 = h
        h = self.to_relu_4_3(h)
        h_relu_4_3 = h
        out = (h_relu_1_2, h_relu_2_2, h_relu_3_3, h_relu_4_3)
        return out


def gram(x):
    (bs, ch, h, w) = x.size()
    f = x.view(bs, ch, w * h)
    f_T = f.transpose(1, 2)
    G = f.bmm(f_T) / (ch * h * w)
    return G


class PerceptualLoss:
    def __init__(self, args):
        self.content_layer = args.content_layer
        device = get_device(args)
        self.vgg = nn.DataParallel(Vgg16())
        self.vgg.eval()
        self.mse = nn.DataParallel(nn.MSELoss())
        self.mse_sum = nn.DataParallel(nn.MSELoss(reduction='sum'))
        style_image = Image.open(args.style_image).convert('RGB')
        _, transform = get_transform(args)
        style_image = transform(style_image).repeat(args.batch_size, 1, 1,
1).to(device)

        with torch.no_grad():
            self.style_features = self.vgg(style_image)
            self.style_gram = [gram(fmap) for fmap in self.style_features]
```

```
        pass

    def __call__(self, x, y_hat):
        b, c, h, w = x.shape
        y_content_features = self.vgg(x)
        y_hat_features = self.vgg(y_hat)

        recon = y_content_features[self.content_layer]
        recon_hat = y_hat_features[self.content_layer]
        L_content = self.mse(recon_hat, recon)

        y_hat_gram = [gram(fmap) for fmap in y_hat_features]
        L_style = 0
        for j in range(len(y_content_features)):
            _, c_l, h_l, w_l = y_hat_features[j].shape
            L_style += self.mse_sum(y_hat_gram[j], self.style_gram[j]) /
float(c_l * h_l * w_l)

        L_pixel = self.mse(y_hat, x)

        # calculate total variation regularization (anisotropic version)
        # https://www.wikiwand.com/en/Total_variation_denoising
        diff_i = torch.sum(torch.abs(y_hat[:, :, :, 1:] - y_hat[:, :, :, :-1]))
        diff_j = torch.sum(torch.abs(y_hat[:, :, 1:, :] - y_hat[:, :, :-1, :]))
        L_tv = (diff_i + diff_j) / float(c * h * w)

        return L_content, L_style, L_pixel, L_tv
```

## 三、Perceptual Loss

要想对图像进行风格迁移，首先必须要做的是定义风格是什么？

风格是什么，作为一个对世界有着丰富而敏感的人，当然可以感受得到，可要想用言语精确描述达到可以量化的程度，我想是难以做到的。于是问题就来了，如果我们无法精确地量定义风格是什么，又该如何去教机器去了解什么是风格呢？

为了解决这个问题，学者们采用了一个十分取巧的方法，那就是，既然我们无法定义风格是什么，那么不妨定义一下：风格不是什么？

**那么风格不是什么呢?**

风格绝对不是内容，即同样一副美术作品的内容，是可以用不同风格来表达的，而具有同样风格的作品，可以具有完全不同的内容。更具体地说，风格是一种特征，这种特征具有**位置不敏感性。**

于是，我们可以借用训练好的 DCNN，前向传播提取图片的特征图 $F \in \mathbb{R}^{C\times H\times W}$ 后，对于每个点的特征 $F_{h,w} \in \mathbb{R}^{C}$ 求其 Gram 矩阵得到 $G_{h,w} = F_{h,w}F_{h,w}^{T} \in \mathbb{R}^{C\times C}$，然后将每个点的 Gram 矩阵相加 $G = \sum_{h,w} G_{h,w} = \sum_{h,w} F_{h,w}F_{h,w}^{T} \in \mathbb{R}^{C\times C}$ 这个 Gram 矩阵最大的特点就是具有**位置不敏感性**，所以，我们可以将这个 G 当做衡量一张图篇风格的量化描述，考虑到一个卷积神经网络中间有多层 特征图，对于每层特征图都可以得到 Gram 矩阵，所以我们可以使用 $\{G_1, G_2, \dots G_l\}$ 来更为全面地描述一张图的风格。相应的损失函数则为：

$$L_{style}^{l} = ||\frac{1}{C_l H_l W_l}(G_l(\hat{y}) - G_l(y))||_2^2$$

搞定了对风格的量化描述，接下来就要对图片的**内容**进行量化描述了，不过这个比较简单，直接用每一层的特征图来描述即可：$\{F_1, F_2, \dots F_l\}$，相应的损失函数为

$$L_{content}^{l} = ||\frac{1}{C_l H_l W_l}(F_l(\hat{y}) - F_l(y))||_2^2$$

最后，为了保持风格转换后的低层的特征，还引入了两个简单的 Loss

$$L_{pixel} = \frac{1}{CHW}||\hat{y} - y||_2^2$$

即耳熟能详的的 MSELoss。还有 total variation loss，这个 loss 的目的是为了提高图像的平滑度：

$$L_{tv} = Mean||\sum_{h,w}(\hat{y}_{h+1,w} - y_{h,w}) + (\hat{y}_{h,w+1} - y_{h,w})||_2^2$$

整体的 Loss 就出来了：

$$L = \lambda_{style}L_{style} + \lambda_{content}L_{content} + \lambda_{pixel}L_{pixel} + \lambda_{tv}L_{tv}$$

## 四、Perceptual Loss Pytorch 实现

*在这项工作中，我们生成了2048个×1024视觉上吸引人的结果，具有一个新的对抗性损失，以及新的多尺度生成器和鉴别器架构。*此外，我们还将我们的框架扩展到具有两个附加特性的交互式视觉操作。首先，我们合并了对象实例分割信息，这支持对象操作，如删除/添加对象和更改对象类别。其次，我们提出了一种方法，在相同的输入下生成不同的结果，允许用户交互地编辑对象外观。人类意见研究表明，我们的方法显著优于现有的方法，提高了深度图像合成和编辑的质量和分辨率。

以下是本文采用的方法：

**Instance-Level Image Synthesis**

- **3.2. Improving Photorealism and Resolution**
- - **Coarse-to-fifine generator**
  - **Multi-scale discriminators**
  - **Improved adversarial loss**
- **Improved adversarial loss**
- - 相反，我们认为实例映射提供的最关键的信息是对象边界，而这在语义标签映射中是不可用的。例如，当同一个类的对象彼此相邻时，仅查看语义标签映射并不能区分它们。对于街景尤其如此，因为许多停放的汽车或行走的行人经常彼此相邻，如图4a所示。但是，使用实例映射，分离这些对象成为一个更容易的任务。因此，为了提取这一信息，我们首先计算实例边界图（图4b）。在我们的实现中，如果实例边界映射中的一个像素的对象ID与它的任何4个（KNN算法！）邻居不同，则该像素为1，否则为0。然后，将实例边界映射与语义标签映射的一个热向量表示连接起来，并输入生成器网络。类似地，鉴别器的输入是实例边界映射、语义标签映射和真实/合成图像的通道连接。图5b显示了一个演示通过使用对象边界进行改进的示例。
- **Learning an Instance-level Feature Embedding**

**方法：**

1. **网络结构和目标函数：**

**Coarse-to-fine generator:**

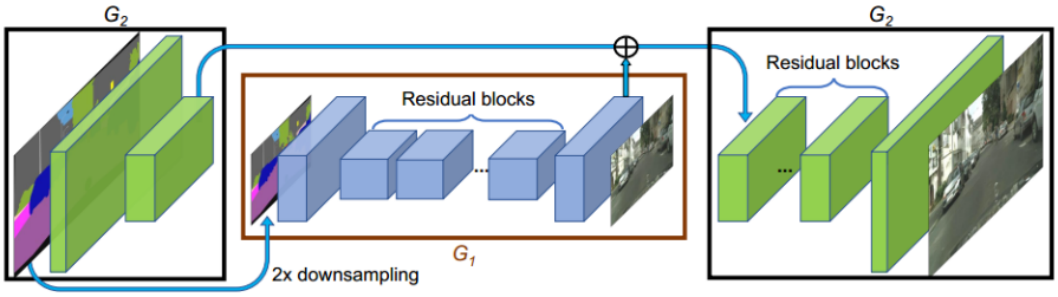


Figure 3: Network architecture of our generator. We first train a residual network $G_1$ on lower resolution images. Then, another residual network $G_2$ is appended to $G_1$ and the two networks are trained jointly on high resolution images. Specifically, the input to the residual blocks in $G_2$ is the element-wise sum of the feature map from $G_2$ and the last feature map from $G_1$.

$G_1$ 网络执行的分辨率为1024*512，$G_2$ 执行的分辨率是$G_1$ 的4倍，若要得到更高的分辨率可以额外增加一个局部增强生成器，同样是之前的4倍（纹理特征是否跟上？）。

$G_1$ 的输入是1024*512的语义标签map，通过$G_1$ （包含：一个卷积front-end、一系列残差块和一个反卷积back-end）得到分辨率为1024*512的图像。

G2的输入为两个特征maps的元素级和：$G_2$ 的输出特征map、$G1$ $G_1$ 网络中back-end的最后一个特征map。这有助于整合$G_1$ 到$G_2$ 的全局信息。

**Multi-scale discriminator**

由$D_1 D_2 D_3$ 三个判别器组成。具体来说，就是对真实和合成的高分辨率图片执行2和4倍下采样得到一个3scales的图像金字塔。在coarset scale的图像有最大的接受域，能知道生成器生成全局一致的内容。在finest scale的图像激励生成器产生finer细节。

**Improved adversarial loss**

GAN loss中加入了feature matching loss，可以稳定训练。

$$\min_G \left( \left( \max_{D_1,D_2,D_3} \sum_{k=1,2,3} \mathcal{L}_{\text{GAN}}(G, D_k) \right) + \lambda \sum_{k=1,2,3} \mathcal{L}_{\text{FM}}(G, D_k) \right)$$ ，其中

$$\mathcal{L}_{\text{FM}}(G, D_k) = \mathbb{E}_{(\mathbf{s},\mathbf{x})} \sum_{i=1}^{T} \frac{1}{N_i} [||D_k^{(i)}(\mathbf{s}, \mathbf{x}) - D_k^{(i)}(\mathbf{s}, G(\mathbf{s}))||_1],$$ ，$\lambda$ 控制两项的重要程度。而最终的损失函数中还加入了感知损失。

1. **Instance Maps**

为了进一步提升生成图像的质量，首先计算了instance boundary map，然后和语义标签map的one-hot向量表示进行级联输入到生成器中。效果如下：

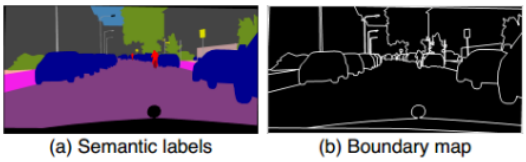


Figure 4: Using instance maps: (a) a typical semantic label map. Note that all connected cars have the same label, which makes it hard to tell them apart. (b) The extracted instance boundary map. With this information, separating different objects becomes much easier.

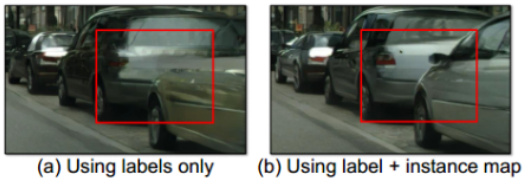


Figure 5: Comparison between results without and with instance maps. It can be seen that when instance boundary information is added, adjacent cars have sharper boundaries.

1. **Learing an Instance-level Feature embedding**

为了生成多样性的图片并且允许实例级控制，论文提出了在生成器的输入中添加一个低维度的特征通道。

为生成低维特征，用一个编码器E找到一个和图片中每个instance相关的ground truth 特征向量。为了保证特征和每个instance一致，对E的输出添加了一个instance-wise 平均池化层，然后将平均特征广播到实例的所有像素位置（什么意思？）。

的输出添加了一个instance-wise 平均池化层。然后将平均特征广播到实例的所有像素位置（什么意思？）。

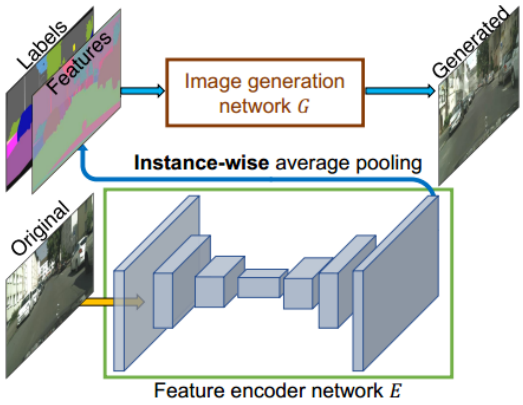


Figure 6: Using instance-wise features in addition to labels for generating images.

**实验结果分析：**

4 **Sparse ReRAM Engine: Joint Exploration of Activation and Weight Sparsity in Compressed Neural Networks**

利用模型稀疏性减少无效计算是DNN推理加速器实现能量效率的常用方法。然而，由于crossbar结构的紧密耦合，利用基于ReRAM的神经网络加速器的稀疏性是一个较少探索的领域。基于ReRAM的NN加速器的现有体系结构研究假设，整个纵横制阵列可以在一个周期内激活。然而，出于推理精度的考虑，矩阵向量计算在实践中必须以更小的粒度进行，称为运算单元（OU）。基于OU的体系结构为利用DNN稀疏性创造了新的机会。在本文中，我们提出了第一个实用的稀疏ReRAM引擎，它利用了权重和激活稀疏性。我们的评估表明，所提出的方法在消除无效计算方面是有效的，并且提供了显著的性能改进和节能。

CONCLUSION:在本文中，我们研究了基于ReRAM的DNN加速器稀疏性探索的设计挑战，并证明了一个实用的基于ou的ReRAM加速器为有效利用DNN稀疏性开辟了新的机会。我们提出了第一个实用的稀疏ReRAM引擎（SRE），它利用细粒度的基于ou的计算来共同利用权值和激活稀疏性。我们对广泛的神经网络模型进行的评估显示，在没有利用稀疏性的基线上，SRE提供了显著的性能加速（高达42.3倍）和能源节约（高达95.4%）。此外，SRE成功地实现了实际的基于ReRAM的DNN加速器设计，考虑到ReRAM单元可靠性的限制，实现了令人满意的推理精度，同时提供了与过度理想化对应的性能相当的性能和能源效率。

5 跑了一下gatys的代码，顺利运行了，有空去读一下源码，实现是tf1