drawNet

AI作画，与传统的图像风格化不同，AI作画是生成整个绘画过程。先前的大多数做法都是采用迭代的方式，速度很慢，我们采用了端到端的方式，直接输出绘画过程，速度提升近百倍，并且提出了一种模拟人类作画的算法，让画笔顺序更为自然。

先前的方法：

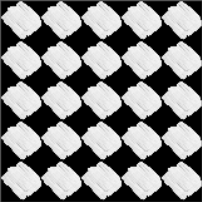
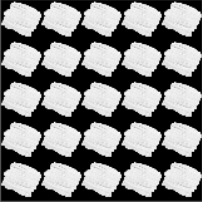
[megvii-research/ICCV2019-LearningToPaint: ICCV2019 - A painting AI that can reproduce paintings stroke by stroke using deep reinforcement learning. (github.com)](https://github.com/megvii-research/ICCV2019-LearningToPaint)

[jiupinjia/stylized-neural-painting: Official Pytorch implementation of the preprint paper "Stylized Neural Painting", in CVPR 2021. (github.com)](https://github.com/jiupinjia/stylized-neural-painting)

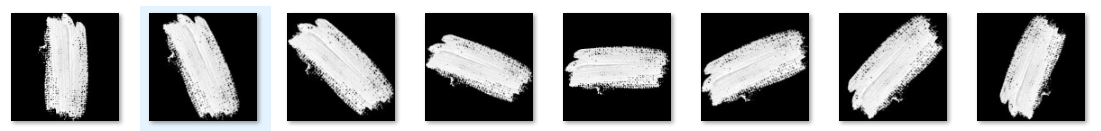
思路：

最开始打算用生成网络来直接生成每一个位置的画笔，然后用判别网络来让对抗使生成的图像真的是画笔的形状，但是效果并不好，画笔位置几乎固定不变，而且画笔也并不像。

于是我们想到了反卷积，根据反卷积的本质，如果反卷积核的参数就是画笔的图像，并且在训练的时候固定这些参数，那么输出必然是画笔的样子，如下图所示：

然后多定义几种画笔的种类，有不同的方向和粗细，实验中我们采用了8种方向和2种尺寸，如图所示：

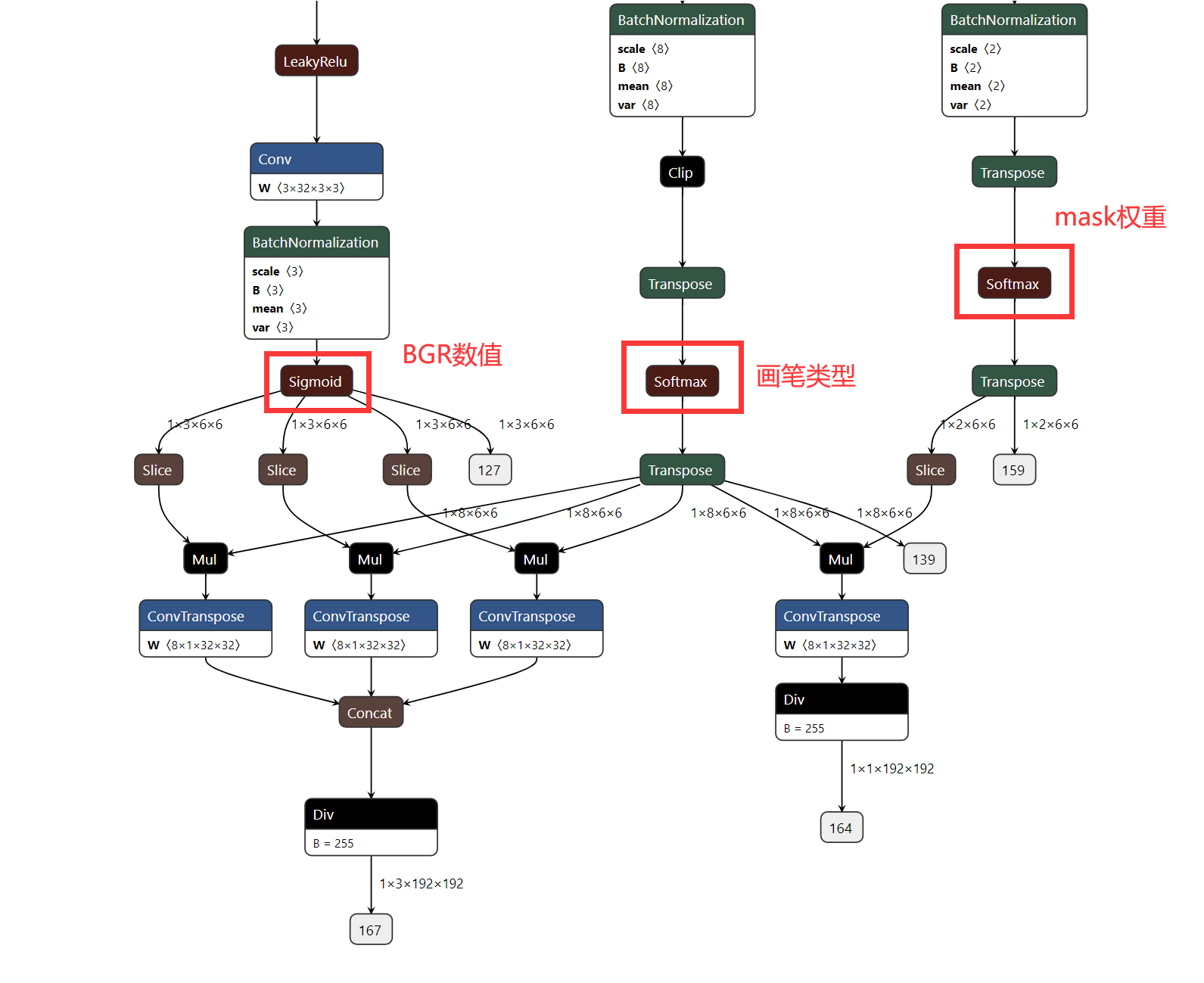


我们设计了一个绘图的模块，模块分为四部分：

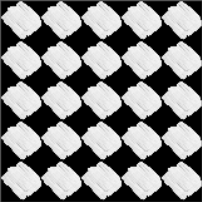
1. color，输出大小为(N,3,H,W),分别代表每个位置的BGR三种颜色。
2. draw，输出大小为(N,8,H,W),最后一层为Softmax，代表每个位置选择8种画笔的其中一个。
3. mask，输出大小为(N,2,H,W),最后一层为Softmax，其中第二维的第1种概率代表了每个位置画笔的透明度。
4. deconv，反卷积模块，总共三个，对应BGR三个通道，输入通道数为8，输出通道数为1，输入为第二个模块draw或者第三个模块mask，理想状态想如果draw某一个位置的输出为[0,1,0,0,0,0,0,0]，即只有一位是1其他都是0，那么该位置反卷积后生成的图像就是只有一种画笔，其他种类会因为乘0而被忽略。所以为了让输出尽可能趋于只有一个1其余均为0，还特地设计了一个loss去监督，后面会提到。输入为mask的时候同理。

整个模块如下图所示，如某个位置的draw为 [0,1,0,0,0,0,0,0] ，color为(0.5,0.7,0.3)，mask为1（为0的时候就代表这一笔不用画），draw\*color则得到 B [0,0.5,0,0,0,0,0,0],

G [0,0.7,0,0,0,0,0,0], R [0,0.3,0,0,0,0,0,0],在输入反卷积后就会得到不同颜色的画笔。



我们的网络中用了10个这种绘图模块，对应着5种不同的画笔，每种画笔有两种位置，他们相互错开。可以看到，一个绘画模块会有很多缝隙，所以同一种画笔会有两个绘图模块，比如一个输入尺寸为5x5，另一个为6x6，经过交错后就可以覆盖整张画布了。

网络输入图像固定为192x192，5种尺度分别为（画笔大小x画笔数量，后面的代表错开位置的模块输入，但是输出会经过裁剪保持和前者一致）：

* 32 x 6 32 x 7
* 24 x 8 24 x 9
* 16 x 12 16 x 13
* 12 x 16 12 x 17
* 8 x 24 8 x 25

根据人类的绘画习惯，应该先用粗的画笔绘制大概轮廓，再用小画笔精修细节，所以画笔大小以此从32、24、16、12、8，这也对应着反卷积核的尺寸。

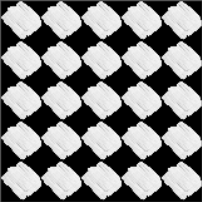
设net6为输入为32 x 6的网络，net6o为32 x 7的网络，字母o代表offset，后面以此类推。

每一个net的最后都有绘制模块，输入为(绘制的图像，画布)。这十个网络呈一个串联的关系。这里直接贴代码可能会更清晰一些。可以看到最开始的画布是一张纯黑的全为0。然后第二个网络的画布是第一个网络的输出，以此类推十个网络的输出组成了整张画。

其中y表示该尺度下的所有位置的画笔，大小为(N,3,192,192)



m是该尺度下的所有位置的画笔的mask，大小为(N,3,192,192)



m\_代表前面提到的mask模块的输出，大小为(N,2,H,W)

d代表前面提到的draw模块的输出，大小为(N,8,H,W)

canvas = img.new\_zeros(img.size()).detach()

y6,m6,d6,m\_6,c6 = self.net6(img,canvas)

pred6 = canvas \* (1 - m6) + y6 \* m6

canvas = pred6\*1

y6o,m6o,d6o,m\_6o,c6o = self.net6o(img,canvas)

pred6o = canvas \* (1 - m6o) + y6o \* m6o

canvas = pred6o\*1

y8,m8,d8,m\_8,c8 = self.net8(img,canvas)

pred8 = canvas \* (1 - m8) + y8 \* m8

canvas = pred8\*1

y8o,m8o,d8o,m\_8o,c8o = self.net8o(img,canvas)

pred8o = canvas \* (1 - m8o) + y8o \* m8o

canvas = pred8o\*1

y12,m12,d12,m\_12,c12 = self.net12(img,canvas)

pred12 = canvas \* (1 - m12) + y12 \* m12

canvas = pred12\*1

y12o,m12o,d12o,m\_12o,c12o = self.net12o(img,canvas)

pred12o = canvas \* (1 - m12o) + y12o \* m12o

canvas = pred12o\*1

y16,m16,d16,m\_16,c16 = self.net16(img,canvas)

pred16 = canvas \* (1 - m16) + y16 \* m16

canvas = pred16\*1

y16o, m16o, d16o,m\_16o,c16o = self.net16o(img, canvas)

pred16o = canvas \* (1 - m16o) + y16o \* m16o

canvas = pred16o\*1

y24, m24, d24,m\_24,c24 = self.net24(img, canvas)

pred24 = canvas \* (1 - m24) + y24 \* m24

canvas = pred24\*1

y24o, m24o, d24o,m\_24o,c24o = self.net24o(img, canvas)

pred24o = canvas \* (1 - m24o) + y24o \* m24o

Loss函数设计：

训练的loss函数有七个，分别代表不同的含义。

loss1，让每一个尺度的输出尽可能和目标图像相似，pixelloss为每个位置像素L1误差的均值。

loss1 = self.pixelloss(img,pred6)\

+self.pixelloss(img,pred6o)\

+self.pixelloss(img,pred8)\

+self.pixelloss(img,pred8o)\

+self.pixelloss(img,pred12)\

+self.pixelloss(img,pred12o)\

+self.pixelloss(img,pred16)\

+self.pixelloss(img,pred16o)\

+self.pixelloss(img,pred24)\

+self.pixelloss(img,pred24o)

loss2，让每一个尺度的所有画笔和图像尽可能相似，但是越小的画笔最后画，所以权重也最大，注意这里要乘以mask。

loss2 = self.pixelloss(img\*m6, y6\*m6)\*0.2 \

+ self.pixelloss(img\*m6o, y6o\*m6o)\*0.2 \

+ self.pixelloss(img\*m8, y8\*m8)\*0.4 \

+ self.pixelloss(img\*m8o, y8o\*m8o)\*0.4\

+ self.pixelloss(img\*m12, y12\*m12)\*0.6 \

+ self.pixelloss(img\*m12o, y12o\*m12o)\*0.6\

+ self.pixelloss(img\*m16, y16\*m16)\*0.8 \

+ self.pixelloss(img\*m16o, y16o\*m16o)\*0.8 \

+ self.pixelloss(img\*m24, y24\*m24) \

+ self.pixelloss(img\*m24o, y24o\*m24o)

loss3，让每一个mask尽可能地小，意思就是用最少的画笔来绘制这幅图。

loss3 = torch.mean(m\_6[:, :1, :, :]) \

+ torch.mean(m\_6o[:, :1, :, :]) \

+ torch.mean(m\_8[:, :1, :, :]) \

+ torch.mean(m\_8o[:, :1, :, :]) \

+ torch.mean(m\_12[:, :1, :, :]) \

+ torch.mean(m\_12o[:, :1, :, :])\

+ torch.mean(m\_16[:, :1, :, :]) \

+ torch.mean(m\_16o[:, :1, :, :]) \

+ torch.mean(m\_24[:, :1, :, :]) \

+ torch.mean(m\_24o[:, :1, :, :])

loss4，让每种画笔的概率尽可能的远离0.5，但由于是softmax，所以结果会趋近于只有一个数值是1，其他的都是0。loss4是越大越好。

loss4 = torch.mean(torch.abs(d6-0.5))\

+torch.mean(torch.abs(d6o-0.5))\

+torch.mean(torch.abs(d8-0.5))\

+torch.mean(torch.abs(d8o-0.5))\

+torch.mean(torch.abs(d12-0.5))\

+torch.mean(torch.abs(d12o-0.5))\

+torch.mean(torch.abs(d16-0.5))\

+torch.mean(torch.abs(d16o-0.5))\

+torch.mean(torch.abs(d24-0.5))\

+torch.mean(torch.abs(d24o-0.5))\

loss5，让每种画笔mask的概率尽可能的远离0.5，原理同loss4，loss5也是越大越好。

loss5 = torch.mean(torch.abs(m\_6-0.5))\

+torch.mean(torch.abs(m\_6o-0.5))\

+torch.mean(torch.abs(m\_8-0.5))\

+torch.mean(torch.abs(m\_8o-0.5))\

+torch.mean(torch.abs(m\_12-0.5))\

+torch.mean(torch.abs(m\_12o-0.5))\

+torch.mean(torch.abs(m\_16-0.5))\

+torch.mean(torch.abs(m\_16o-0.5))\

+torch.mean(torch.abs(m\_24-0.5))\

+torch.mean(torch.abs(m\_24o-0.5))\

loss6，让在单张画的不同位置和不同画的相同位置尽可能让画笔的方向不一样，使draw的输出在不同的轴方差最大化，意思就是尽可能地选择不同方向的画笔，loss6也是越大越好。

在实验中添加loss6之后在训练一段时间后会出现NaN现象，经过分析是因为该loss是为了让方差尽可能地打，这里面的d前面有softmax，所以训练会让exp前的数值越来越大，因为有指数运算，所以很容易导致越界，所以在softmax前面做了一个[-10,10]的数值截断。

loss6 = torch.mean(torch.std(d6 \* m\_6[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d6o \* m\_6o[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d8 \* m\_8[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d8o \* m\_8o[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d12 \* m\_12[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d12o \* m\_12o[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d16 \* m\_16[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d16o \* m\_16o[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d24 \* m\_24[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d24o \* m\_24o[:, :1, :, :], [2, 3])) \

+ torch.mean(torch.std(d6 \* m\_6[:, :1, :, :], 0)) \

+ torch.mean(torch.std(d6o \* m\_6o[:, :1, :, :], 0)) \

+ torch.mean(torch.std(d8 \* m\_8[:, :1, :, :], 0)) \

+ torch.mean(torch.std(d8o \* m\_8o[:, :1, :, :], 0)) \

+ torch.mean(torch.std(d12 \* m\_12[:, :1, :, :], 0)) \

+ torch.mean(torch.std(d12o \* m\_12o[:, :1, :, :], 0)) \

+ torch.mean(torch.std(d16 \* m\_16[:, :1, :, :], 0)) \

+ torch.mean(torch.std(d16o \* m\_16o[:, :1, :, :], 0)) \

+ torch.mean(torch.std(d24 \* m\_24[:, :1, :, :], 0)) \

+ torch.mean(torch.std(d24o \* m\_24o[:, :1, :, :], 0))

loss7，contentLoss，出自图像风格化，将预测图像和目标图像都用预训练好的分类网络来提取浅层特征，使浅层特征尽可能地相似，和loss1的pixelloss不同，contentLoss更多的是对边缘之类的特征更加敏感，可以加速收敛。这里采用了mobilenet和resnet50的浅层特征。

loss7 = closs(pred24o,img)

最终loss组成如下，因为loss4、loss5、loss6都是越大越好，所以采用了w/(loss+1)的形式，其中w=1。

loss= loss1\*5+loss2\*5+loss3\*0.1+w/(loss4+1)+5/(loss5+1)+w/(loss6+1)+loss7

绘制

网络可以直接输出最终的图像，也可以输出每一个画笔的顺序，这里设计了一个算法来模仿人类的绘图过程。首先是画笔从大到小，同种尺度的画笔优顺序逻辑如下：

设：误差p=pixelloss(目标图像p,画布p)-pixelloss(目标图像p,画布p+画笔p) p代表位置。先选择误差p最大的位置作为起始点，绘制完当前位置后，选择位置p周围8个位置中误差最大的作为下一个起始点，如果周围的都已经绘制过了，则再次选择误差p最大的位置作为下一个起始点，直到所有位置都绘制完成。这种绘制逻辑就是优先选择让画布更像目标图像的位置，然后前后画笔尽可能的相邻，这种方式比较符合人类的逻辑。

性能

测试环境：CPU：i7-9700k，GPU：2070s

只输出最终图像的话，单张192x192在40毫秒左后，计算绘制路径大概需要350-400毫秒，抛开保存图片的耗时，生成单张图片的绘制流程耗时在1秒左右，而之前的两个工程的速度，一个在70多秒左右，一个是1.5秒左右(输出为128x128)。在速度性能上大幅度提升。最终结果差异也比较大。下图均为单张输出结果。



Learning To Paint

Stylized Neural Painting Ours