Exploring Image Generation Techniques and Re-implementing Semantic Synthesis using SPADE (Spacially-Adaptive Denormalization)

Abstract

Using Machine Learning for image generation is very fascinating to me. It seems like it could be a way for computers to imagine. Nonetheless, they have many applications as tools for graphics designing, prototyping etc. There are many ways in which this problem has been approached. One of the most popular approaches is GAN. There are many variations, and ideas that evolved from that. This project explores three papers that deal with these, and implements one of them to study benefits and develop a better understanding. The results indicate how the theoretical ideas all come together and lead to impressive results in semantic image synthesis.

1. Background research

1.1. Papers selected

I will be reviewing the following research papers related to image generation.

1.1.1. Semantic Image Synthesis With Spatially-Adaptive Normalization

Author: Taesung Park, Ming-Yu Liu, Ting-Chun Wang, Jun-Yan Zhu

Year published: 2019

Link: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8953676

Summary: This paper by Park et al., tackles the problem of generation of imagery from a semantic layout [4]. A semantic layout is a simplistic image which defines pixel regions to belong to a certain class (like: tree, land, and sky). It basically achieves the idea of photorealistic imagination. To do this, the novel idea is that, instead of feeding the semantic layout to a deep generative model, using a ”SPacially ADaptive DEnormalization layer (SPADE)” to learn the encoded information from the layout[4]. The justifica- tion given by Park et al. is that the SPADE retains the semantic region labels throughout the generation process, whereas if it was just fed in the beginning, the information would have been somewhat lost through the way.
The new model was trained and evaluated on multiple datasets such as COCO- Stuff, ADE20K, Cityscape, and Flikr Landscapes[4]. Then the model is com- pared to three other ”semantic image synthesis” models CRN, SIMS, and pix2pixHD [4]based on three metrics: mean-IoU, pixel accuracy, Frechet inception distance. And, according to the reported value, it appears that the SPADE based model is better than the other three in almost every dataset.
Not only does the paper claim to produce these realistic images, but the style of the image can also be set by giving a style image for input.

Critique: The possibility of implementing style component along with the image generation shown in the very beginning of the paper was a great eye-catcher for me. The only thing I can think of for improving the paper would be the inclusion of an explanation of how the semantic layouts were generated in each of the datasets. It could be possible that SPADE just happens to be close to the inverse of the algorithm used to generate the layouts.

1.1.2. A Style-Based Generator Architecture for Generative Adversarial Networks

Author: Tero Karras, Samuli Laine, Timo Aila

Year published: 2019

Link: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8953766

Summary: A popular approach for synthesizing images via computers has been through Generative Adverserial Networks as described in the the paper with the same name by Goodfellow et al [1]. The paper by Karras et al claims that even though the quality of images generated by GANs is improving with time, they still severely lack explainability of features in the generated images[3].
This paper proposes a new architecture for image generation as an alternative to traditional Generative Adversarial Networks. The new model is inspired from the image style transfer architecture, which generates an image taking the content from one input image and the artistic style from the other one [2]. It utilizes the Adaptive Instance Normalization to normalize the ”style” inputs and this technique was borrowed from the style transfer paper by Huang et al [2].

Thereby, the redesigned model utilizes the style input to control stochastic aspects of the generated image.
The qualities of the generated images are evaluated using the metric, Frechet inception distance on the Celeb-A-HQ and the FFHQ datasets[3]. It is thus shown that the new-model gives a better image quality than traditional GANs. The result is that along with achieving high quality images, the new model also gives control over features of the image by varying(interpolating) the mixing val- ues, and the noise helps in creating stochastic variations (for example, position of hair, these do not change the person's identity, but are a different image)[3]. Also, in the end, a dataset of face images prepared from images scraped from Flickr is provided [3].

Critique: I find it extremely helpful that the FFHQ dataset is shared and the generation script is made open-source. One improvement in the organization of the paper I would suggest is that the ideas could be grouped more explicitly. For instance, there could have been separate sections for studying effect of mixing-inputs and noise.

1.1.3. FastDraw: Addressing the Long Tail of Lane Detection by Adapting a Sequential Prediction Network

Author: Jonah Philion

Year published: 2019

Link: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8953218

Summary: The goal is to have a model take an image of a road from a driver's perspective and identify the lanes from that image. For autonomous driving applications, it is essential that this process is not only accurate, but also takes very little time. The author, Philion, claims that alternate models identify pixels that are lane divisions, or segment the lanes semantically, and then the network's outputs are fed into other post processing methods to identify lanes [5]. The proposed new idea is that instead of having all the post-processing, we make the CNN predict the "lane contours" instead[5]. This essentially gives a representation of the lanes themselves as a function of y-coordinate on the image, and builds the lane post-processing into the neural network [5].
The model utilizes the ResNet50/18 architecture pretrained on the ImageNet dataset for the initial processing of the input.
The model is trained and validated on the Tusimple Lane marking challenge, and the CULane dataset [5]. It is compared to the EL-GAN model based on accuracy, false positives and false negatives. Then to test the generalizability of the model, the author used dash-cam recordings from driving around in Massachusetts. Also, then the model's propagation time is compared to other common alternatives and it is concluded that FastDraw achieves similar accuracy in much lesser times. It is able to process 90.31 frames per second as opposed to 52.6 frames per second by the next fastest model, H-Net.

Critique: I really like how the comparison with a simple ResNet50 in Table-4 are grouped by type of situation in image: crowded, shadow, crossroad etc [5]. This gives an idea of how the model perform in different scenarios. The model's speed is compared to 4 other models. However, its accuracy is not compared to all of those models. My critique would be to have both accuracy and time comparisons with all of the competing models and have the results in one table for better readability.

2. Implementation process

2.1. Implementation goal

Implementing this complicated model with multiple neural-networks seemed like a daunting task. So I divided the process into two steps.

Step 1: The first step was to define all neural networks needed: generator, discriminator, and encoder. Also, I planed to complete forward propagation from input to output (ensure dimensional correctness). The primary focus was on having the architecture built properly, instead of training or optimization. I chose this to be the goal because, once the model is defined, majority of the code is done. Training, validating and testing will require less coding and more of intelligent tweaking of hyper-parameters. So, logically it seemed like a reasonable point to break the project.

Step 2: The second step was to set-up the dataset, loss-functions, optimizers, and the training, and the evaluation script. The plan was to implement the training code and attempt training on the dataset in the same way as proposed in the original paper : same batch sizes, epochs etc. In case the training is unfeasible/too slow due to hardware limitations, I will try varying the batch sizes and/or use only a subset of the training data.

2.2. Implementation plan

Step 1: In the original paper, the authors have mentioned that the ArXiv version of their paper (https://arxiv.org/pdf/1903.07291.pdf) has additional implementation details. Its Appendix A has block diagrams for each neural network, and model architecture description. I followed those to define each model.
The big picture is that there are two parts: A discriminator network (from the PatchGAN model), and a variational auto-encoder that recreates an input image given to it. But the VAE's decoder part is made of blocks that are a modifi-

cation of the ResNet block. They take another input which is the segmented image map, and that is combined with the rest using the SPADE normalization algorithm shown immediately below.

$$\gamma^i_{c,y,x}(m)\frac{h^i_{n,c,x,y}-\mu^i_c}{\sigma^i_c}+\beta^i_{c,y,x}(m)$$

In this equation, n represents batch size, c is the number of channels. x, and y are the width and height of the image.

2.2.1. Forward propagation:

This is also described pictorially in figure 1 below.

- Training
  The actual image is fed to the encoder network, and it outputs the μ, and $\log(\sigma^2)$ encodings. They are used to scale and shift a random sampling from the standard normal distribution. This is fed into the generator (decoder) network, which also takes in the segmentation map at multiple places.

- Inference from only segmentation map input
  In this case, there is no use of the encoder network. Instead the input given to the generator is just the standard-normal gaussian noise.

- Inference from segmentation map and style image input
  If an image is given to the encoder as input which does not necessarily correspond to the segmentation map used, then the network generates an output which satisfies the segmentation map but in style of the input image.

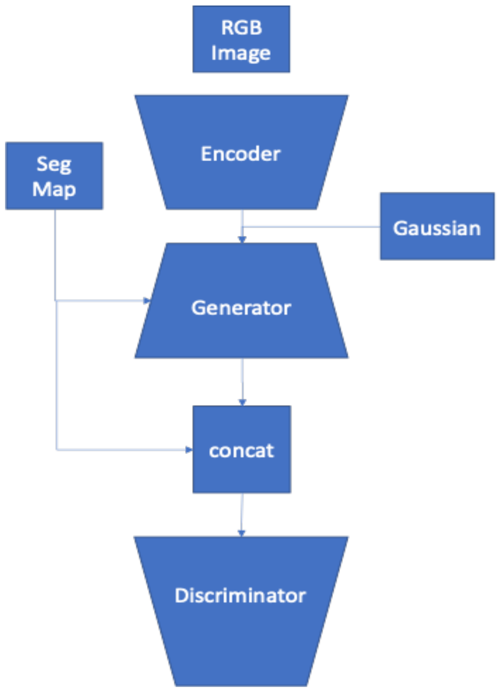


Figure 1: Model architecture

Step 2: Now that I have completed the forward propagation, we move on to the rest of the concepts. This model computes the generator's loss in two parts: KL-Divergence, and Hinge loss.

$$D_{KL}(P||Q)=\sum_i P(i)\times ln(\frac{P(i)}{Q(i)})$$

Equation for KL-Divergence

The original paper trained the model for "200 epochs on Cityscapes and ADE20K datasets, 100 epochs of training on the COCO-Stuff dataset, and 50 epochs of training on the Flickr Landscapes dataset". They train on 8 GPUs using a batch size of 32. This is not feasible for me to recreate given the time and memory constraints on the free version of Google Colab. So I chose 1000 image-annotation pairs at random from the COCO-stuff dataset and used a batch size of 4.

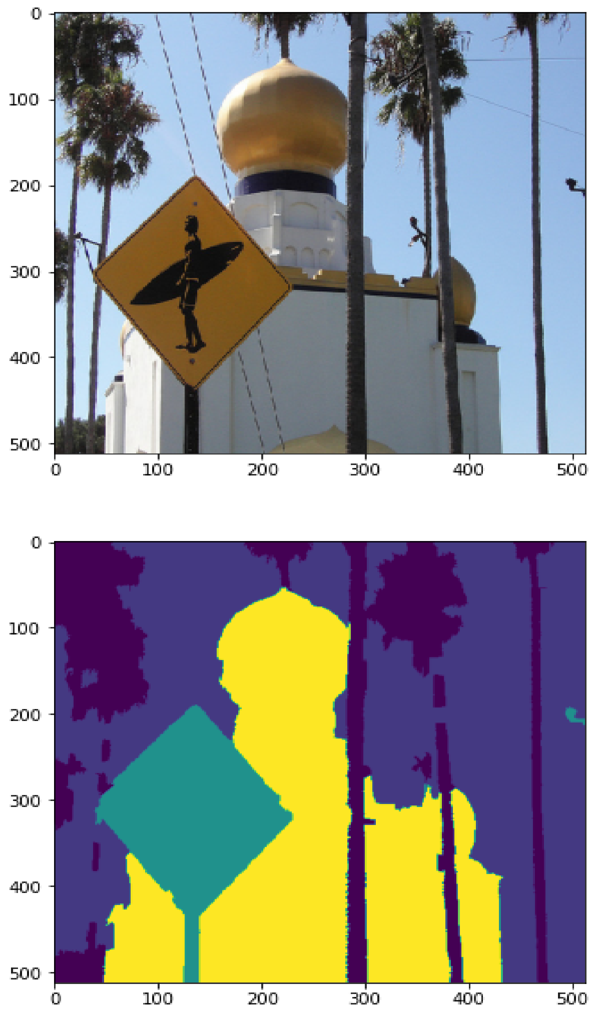


Figure 2: Example image along with segmentation map from the COCO-stuff dataset

The COCO-stuff dataset has RGB images of Common objects and a json file describing the segmentation regions. It has 182 classes in the annotation maps. Here is a samples from the dataset shown in Figure 2.

2.3. Preliminary implementation

Step 1: I have implemented the model architecture. It only involved following the block diagrams, because everything including the number of intermediate channels between convolutions has been explicitly described in the paper.

Owing to the complexity of the model, it wasn't possible for every simple implementation detail to be discussed in the paper, so, I worked on understanding how that works from the authors' implementation shared in their repository on Github (`https://www.github.com/NVlabs/SPADE`).

Step 2: The optimizer I used is exactly what the papers described, Adam with beta1 = 0.0, beta2 = 0.999, and an initial learning rate of 0.0002. The implementation on the Github repository has various user friendly options for tweaking the normalization algorithms between BatchNorm, InstanceNorm etc. and they also have ways of tweaking other hyper-parameters such as the learning rates and number of intermediate channels in the convolution layers. My implementation does not have that level of customizability and user-friendliness, it directly uses the values that were shown to be effective in the original paper.

2.4. Preliminary results OR significant discussion of implementation problems

Step 1: Upon making a rough implementation, I compared my logic to the repository by the authors. There are some differences between the two: For instance, when computing the SPADE normalization, the authors multiplied the inputs by $(1 + \gamma)$ instead of by γ, as mentioned in the paper. I tried training the model for the two epochs using both pathways, and both of them yielded similar results. It seems as if the model just learns the parameters according to what constants are added with no apparent issue.

Step 2: The COCO-stuff dataset also has an unknown class among its segmentation annotation maps. This needed to be handled separately and required manually adding 1 to the number of channels in the input because the input actually had 1 more channel than the number of classes.

3. Thoughts and results

3.1. Results obtained

All losses were recorded every 4 batches. So the actual batch indices are 4 times the values shown in the plots 3a, 3b, and 3c. Upon training for 2 epochs, over 1000 images, the KL-Divergence on the training dataset dropped down from around 70 to around 5. Looking at the plot showing this in figure 3a, it seems as if it is not improving further. However, according to the TTUR (Two-Time scale Update Rule) method of updating the learning rates described in the paper, the next epoch would run with a learning rate half of the current one for the generator, and double the current value for the discriminator.

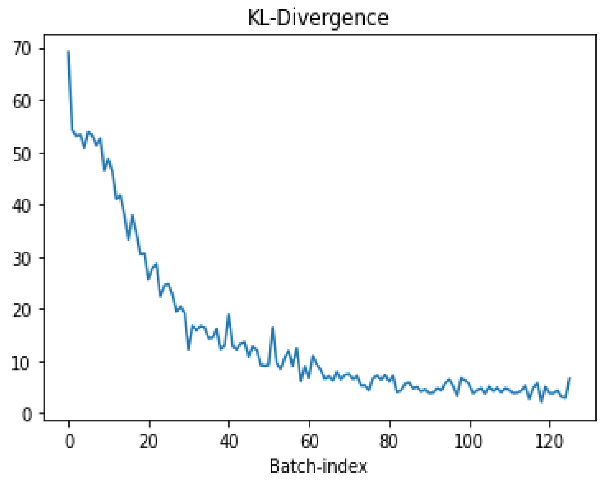


Figure 3a): KL-Divergence over training iterations

The training object for a GAN works on minimizing the generator loss. In this case the generator's loss includes both the KL-Divergence and the Hinge loss. The KL-Divergence happened to be significantly greater than the Hinge loss(GAN-Loss), therefore it did not drop as obviously as the KL divergence. If it were permissible to train for more epochs, then, they would both decrease now that both have reached the same order of magnitude.
This is also evident in the figure 3b shown below.

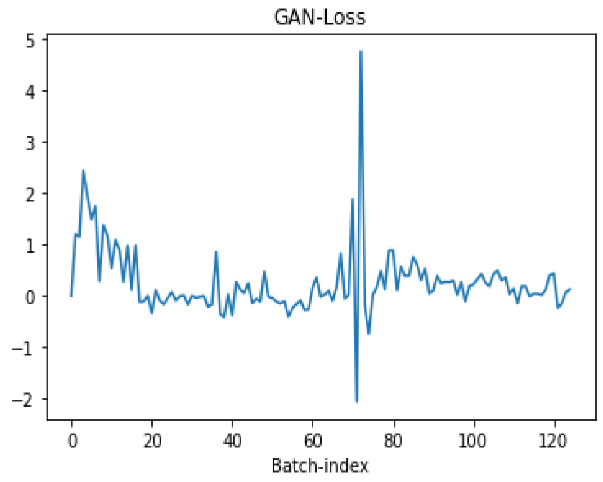


Figure 3b): Generator's hinge loss over training iterations

The discriminator loss is maximized, it appears to hover pretty close to around 1 throughout the training process. This can also be observed in the figure 3c below.

An interesting thing to note is the spike in the losses around the batch with index number 72*4.

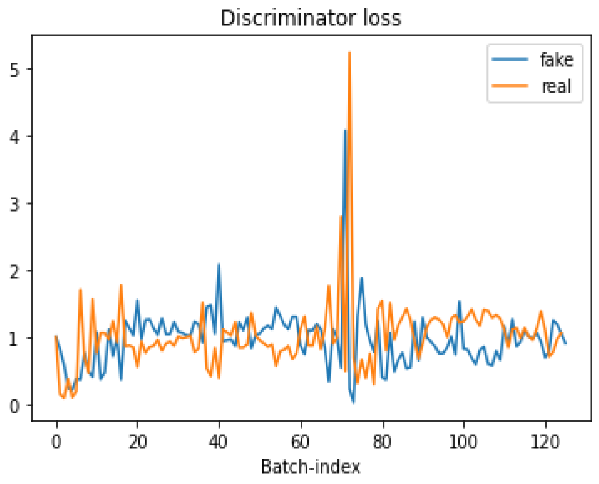


Figure 3c): Discriminator loss over training iterations for both real and fake images

I feel that this spike is occurring due to a couple of batches of very distinct images from the rest that are causing these unreasonable spikes in Hinge losses.

The following figure (figure 4) presents the output of the obtained model for 7 randomly chosen images from the COCO-stuff dataset that the model was not trained on. The first column is the segmentation map inputted to the model for making the generations shown in the middle column. The rightmost column contains the original images that the segmentation maps were based on.

3.2. Explanation for obtained results

Given that the authors trained the model on datasets with more than 100 times the number of images that I used. Also, they trained the model for more than 100 times the number of epochs I did, it is unreasonable to expect similar quality of results.

Therefore it is futile to compute performance metrics such as mIoU on the model that I trained. However, it is interesting to note that even with such limited training iterations and data, the model learned to produce images that follow the edges of the segmentation maps of the images fairy accurately.

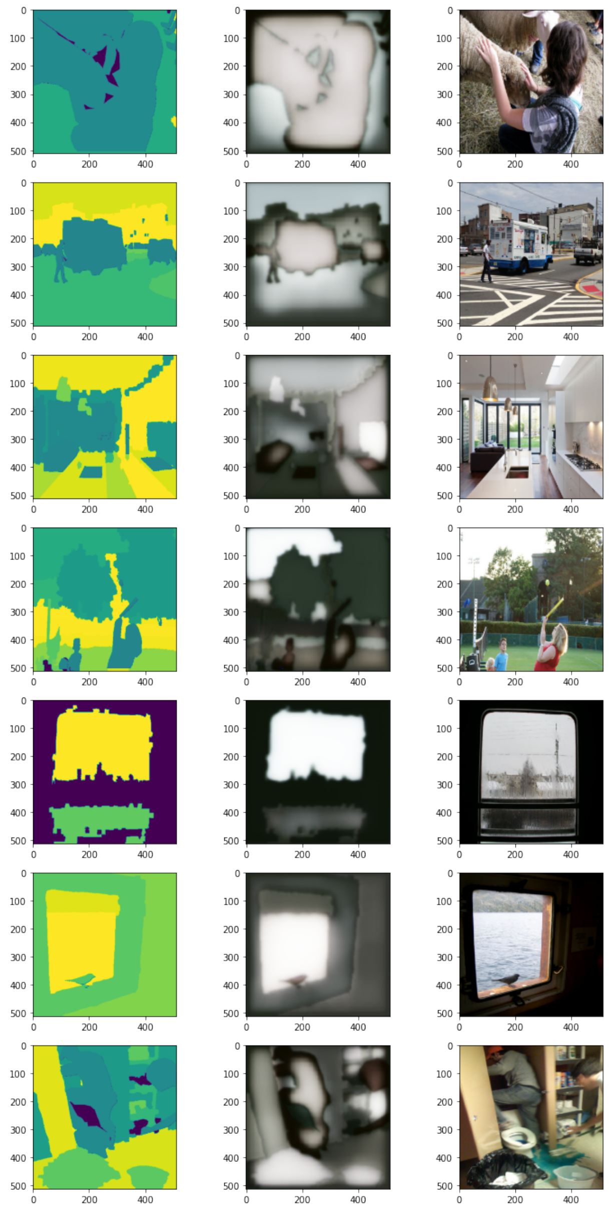
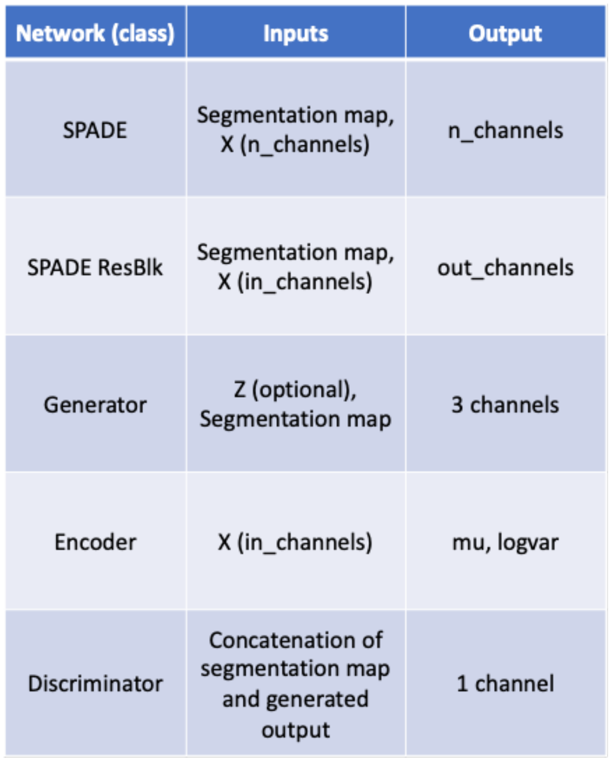


Figure 4: Model performance results on 7 random images from COCO-stuff. Column 1: Segmentation map inputs, Column 2: Model outputs, Column 3: Original image

4. Code structure and concluding thoughts

4.1. Code structure

Since I implemented all the code in one Jupyter notebook (.ipynb file) on colab, I relied primarily on cells to separate and organize different aspects of the code. Whereas, the authors' implementation has a structure following modules defined in python scripts (.py source files).



Network (class)	Inputs	Output
SPADE	Segmentation map, X (n_channels)	n_channels
SPADE ResBlk	Segmentation map, X (in_channels)	out_channels
Generator	Z (optional), Segmentation map	3 channels
Encoder	X (in_channels)	mu, logvar
Discriminator	Concatenation of segmentation map and generated output	1 channel

Figure 5: Networks and their inputs, outputs

Classes to represent the networks

- SPADE
  This is the simplest block that combines the image input and the segmentation map,
- SPADE ResBlk
  This consists of multiple SPADE blocks and can have different number of channels in input and output.
- Generator
  This is the decoder part of the variational auto-encoder.
- Encoder
  This is the encoder, which is a simple convolutional neural network that outputs the mean and variance for encoding the original image.
- Discriminator
  This is based on the PatchGAN network and has a Conv layer in the end followed by the Leaky ReLu activation.

4.2. Concluding Thoughts

This experience of implementing an actual research paper sparked a deep appreciation in me for the level of detail down to which researches have to try different possibilities and make design decisions to get a working model. One such instance was thinking about resizing images to match the dimensions. For the actual images, you could use a bilinear interpolation transform to resize. However, for the segmentation map, that would lead to catastrophic results as the class values are discrete values. So, they must be resized using nearest neighbor interpolation. Once someone reads this, it seems obvious, but thinking about these while designing is very fascinating to me.

References

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial networks, 2014. URL `https://arxiv.org/abs/1406.2661`.

Huang, X. and Belongie, S. Arbitrary style transfer in real-time with adaptive instance normalization, 2017. URL `https://arxiv.org/abs/1703.06868`.

Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. Image-to-image translation with conditional adversarial networks, 2016. URL `https://arxiv.org/abs/1611.07004`.

Karras, T., Laine, S., and Aila, T. A style-based generator architecture for generative adversarial networks, 2018. URL `https://arxiv.org/abs/1812.04948`.

Park, T., Liu, M.-Y., Wang, T.-C., and Zhu, J.-Y. Semantic image synthesis with spatially-adaptive normalization. 2019. doi: 10.48550/ARXIV.1903.07291. URL `https://arxiv.org/abs/1903.07291`.

Philion, J. Fastdraw: Addressing the long tail of lane detection by adapting a sequential prediction network, 2019. URL `https://arxiv.org/abs/1905.04354`.