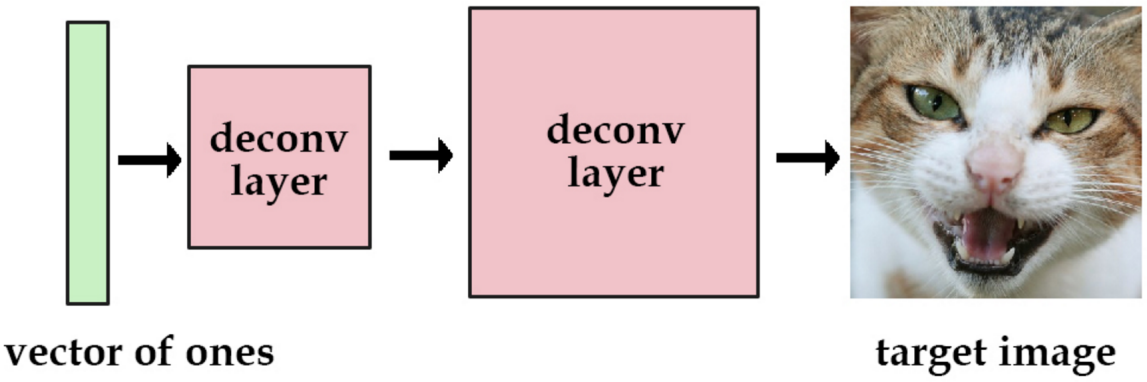
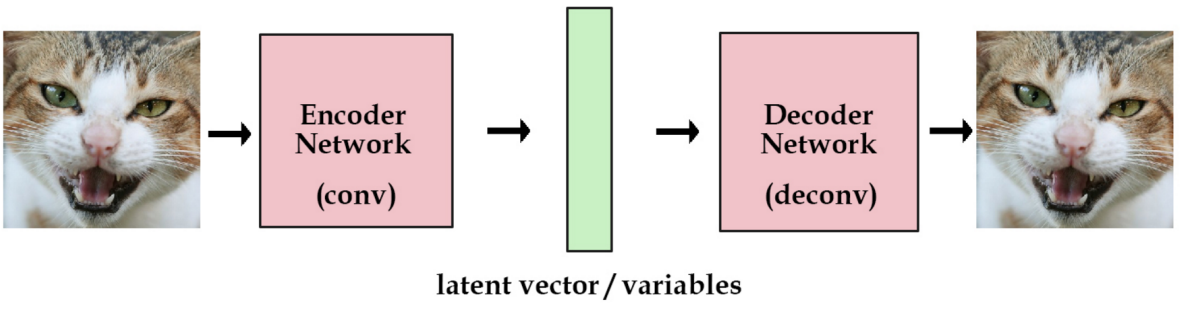
Variational Autoencoder

In order to understand VAE, we start with the simplest network first and then add additional parts step by step. Suppose we have a network with a deconvolution layer, then we set the input as a vector with all values of 1, the output as an image. The network can then be trained to reduce the mean square error between the reconstructed image and the original image. So after the training, the information about the image is retained in the parameters of the network. In this way, we regard this real value vector as a kind of encoding of the original image. For example, we can encode a cat into a vector [3.3, 4.5, 2.1, 9.8].



However, we'd rather the computer code it automatically for us. Then the AE(Autoencoder) comes out for it. We add an encoder that encodes the image into a vector. The decoder is able to reconstruct images from these vectors.



Now we get an Autoencoder model, but we can not produce new images from this model. Because every latent vector comes from original pictures.In order to achieve this goal, we can add a constraint to the encoder that forces it to produce latent vectors that obey the Gaussian distribution. Then we can just sample from the Gaussian distribution to generate random latent vectors.

In fact, there is also a trade-off between the accuracy of the reconstructed image and the fit of the Gaussian distribution. We can let the networks decide the trade-off by themselves. On the one hand, we can use the mean square error to measure the reconstruction error of images. On the other hand, we can use the KL-divergence to measure the difference between the distribution of our latent vectors and the Gaussian distribution.

$$\mathcal{L}_{\text{VAE}} = -\mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})}{q(\boldsymbol{z}|\boldsymbol{x})}\right] = \mathcal{L}_{\text{llike}}^{\text{pixel}} + \mathcal{L}_{\text{prio}}$$

with

$$\mathcal{L}_{\text{llike}}^{\text{pixel}} = -\mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}\left[\log p(\boldsymbol{x}|\boldsymbol{z})\right]$$
$$\mathcal{L}_{\text{prior}} = D_{\text{KL}}(q(\boldsymbol{z}|\boldsymbol{x})\|p(\boldsymbol{z})),$$

Generative Adversarial Networks

GAN(Generative Adversarial Networks) consist of a generator network and a discriminator network. The generator network takes a random sample from the latent space as input, and its output should imitate the true sample in the training set

as much as possible. The input of the discriminator network is the real sample or the output of the generated network, and its purpose is to distinguish the output of the generated network from the real sample as much as possible. The generator network should deceive the discriminator network as much as possible.

VAE/GAN

The paper provides a model combining the VAE and GAN. It learns to encode, generate, and discriminate data sets simultaneously.

The VAE uses L2 loss to compute the reconstruction error of the image, that is to say, minimize MSE. This way will end up with blurred reconstructed pictures. However, GAN does not propose any assumption for the loss function. Instead, it uses a discriminator to tell us whether the sample comes from the expected probability distribution. The signal from the discriminator is fed back to the generator, which then learns to produce a better sample. The trick here is that the GAN is not harmed by an hand-engineering loss function (such as the L1 or L2 loss), which may not be sufficiently accurate to measure the error in the distributions in the data. That's one of the reasons why GAN was able to produce a clearer sample.

This model adds a discriminator network to the origin VAE model to discriminate the output of the decoder images. From the GAN’s perspective, the decoder is regarded as the generator network. So this model optimizes Encoder, Decoder(Generator) and discriminator simultaneously, in order to make the latent space generated by the encoder satisfying the Gaussian distribution as much as possible. In other words,

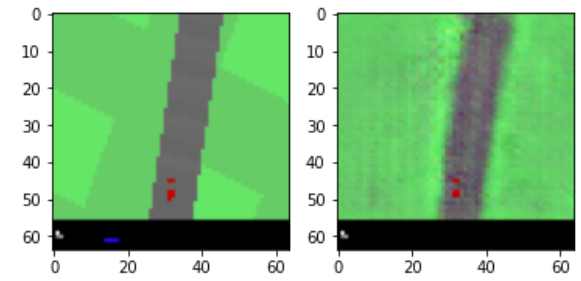
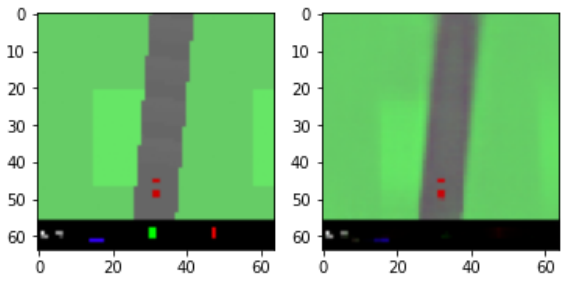
Make images before and after VEA as similar as possible. The decoder (generator) generated images as far as possible real, and made the discriminator as far as possible to distinguish real and fake images.

Implementation

I use the structure in this https://github.com/leoHeidel/vae-gan-tf2 repo. Firstly, I found I can't start training since it will be killed by the system. So I reduce the scale of the network that is the variable of DEPTH and also reduce the batch size. At the start of the training, the loss increased at some epoches. So I changed some parameters of the loss function. For example the log likelihood of the GAN is combined with three parts: true image, false image and random generator image. I increase the weights of log likelihood of the true image. However we trained it for a long time, but the results of the generated image have some bad results, such as the road disappearing in the reconstruction image, especially the encounter with the curve road.

Performance comparison

In the VAE part, the GAN version has more noisy information, and it seems that the Original one can better extract the major feature(the road).

GAN	Original
	

In the RNN part, the GAN version has a higher loss, but not a lot.

In the controller part, the GAN version has lower performance in the same training environment, and it seems that it can't move the car for a lap. While the original one does a good job, it can turn and accelerate in a straight line. Though sometimes, it may drive out of the race track, it would find the right way and go back to the track after a while.

For this is a gif, you can see it in "/res/GAN_out.jpg" and "/res/ORIGINAL_out.jpg".

Because of the limited time and CPU,GPU performance, the controller doesn't reach its upper limit, and it still has a space for progress. We generated a gif in the '/res' folder to show the level of the controller.