Generating Anime Faces Using Variational Auto-encoder (VAE)

Binyamin Reuveni, Yossef Gisser

Part 1

In this part we explore image generation using variational auto-encoder (VAE) as described in the paper called "**Generate Anime Character with Variational Auto-encoder"** written by Wuga and published in the online journal Medium.

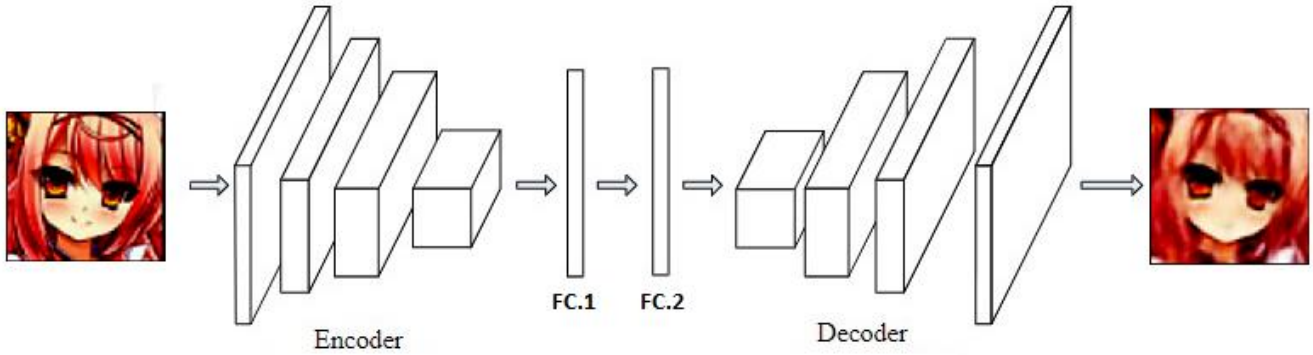
As in the paper, our dataset contains 63.6K images of anime faces, however, we extracted just 63K of them since some of them were interpolated. Our goal is to generate new anime faces using the algorithm described in the paper. We split our data to 80% train set and 20% test set.

In his paper, Wuga compared Variational Auto-encoder (VAE) with Generative Adversarial Network (GAN), and in our work we implement Wuga's method to create anime faces using only VAE.

In this part we show the results from our implementation for the algorithm described and implemented in the paper without changing any hyperparameters. The only change that has been made is to use PyTorch instead of TensorFlow V1 which has been archived and is not supported these days, this change has forced us to understand and translate each line of code from the original implementation.

In the first two parts, every train cycle (500 epochs) of the model took about 18 hours. For this reason, we were limited in our attempts.

Net's architecture:



Encoder: 4 convolutional layers with 5x5 kernel, 2x2 stride and padding with 2 pixels. After each convolutional layer we apply batch normalization and ReLU as an activation function.

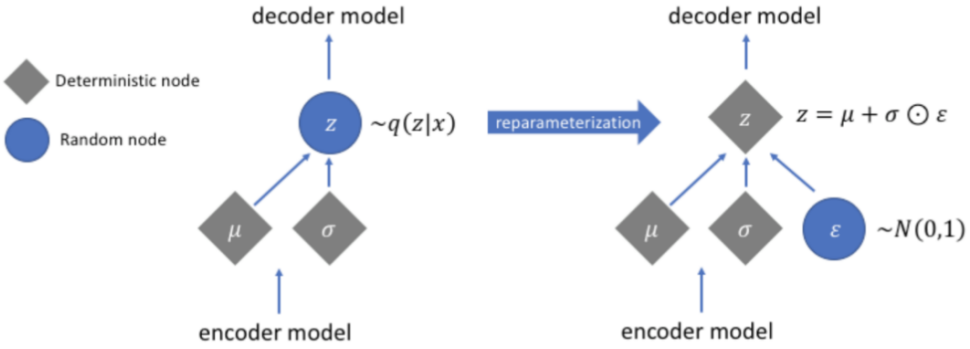
FC.1: Fully connected linear layer with batch normalization and identity function as an activation function.

FC.2: Fully connected linear layer and identity function as an activation function.

Decoder: Mirror image of the encoder and replaced convolution layers with their transposed ones.

The encoder takes the input image and encodes it into a normal distribution latent space, then the decoder sample from that distribution and reconstruct the input image. We want to train the model end-to-end, but sampling operation is not differentiable, so instead of sampling we inject noise without changing the desired probability distribution. Therefore, between the fully connected layers there is one of the most important parts that later allows us to take the decoder part and generate new images with it, this part is called the Reparameterization Trick. Every normal distribution can be represented by a unit gaussian $\mu^{(i)} + \sigma^{(i)} \odot \epsilon \ where \ \epsilon \sim N(0,1)$

Reparameterization Trick:



Another important thing is that we initialize the weights of the network as Wuga did, convolution layers with weights sampled from $N(0, 0.02^2)$ and batch normalization layers with weights sampled from $N(1, 0.02^2)$.

Loss function:
The loss function used by Wuga is a combination of

1. Gaussian negative log likelihood loss (GNLL or Reconstruction loss):

$$0.5\frac{\sum_{i=0}^{n}(x-\hat{y})^2}{var^2+\log(var)}$$

Where n is number of features, x is the original data, $\hat{y}$ is the data after we run it through the net and var is the variance of observation.
Minimizing reconstruction loss term: create image as vivid/real as possible. Minimize the error between real image with the generated image.

2. KL divergence:
The KL divergence has a closed formula (with a prior of unit Gaussian):

$$-0.5\sum_{i=0}^{n}1+logvar-mu^2-e^{logvar}$$

Where n is number of features, mu is the first half (with length of latent dimension) of the encoded tensor and $logvar$ is the second half (with length of latent dimension) of the encoded tensor.

Minimizing KL term: drag P(z|x) distribution to N(0,1). We want to generate the images by sampling through N(0,1), so it is better to let latent distribution as close to the standard norm as possible.

Both loss functions have the same weight. Wuga uses the var parameter from GNLL as hyperparameter that scale GNLL, bigger var scale down the GNLL loss so it would be closer to the KL divergence loss.

As explained in the paper:
“It is easy to see that balancing those two components is very critical to let VAE work.
If we completely ignore KL term, the Variational Auto-encoder converges back to standard Auto-encoder, which vanish any stochastic part of the objective function. So, the VAE could not generate new image but only remember and display the training data (or create pure noise since there is no image encoded in that latent position!). The optimal result, if you are lucky enough, is kernel PCA!
If we somewhat ignore the reconstruction term, the latent distribution collapses into standard normal distribution. So, no matter what the input is, you always get similar output.

Now we understand the trick:

- We want the VAE generate reasonable image, but we do not what it to display training data.
- We want to sample from $N(0,1)$, but we don't want to see same image again and again.

We want the model to create very different images.
Then, how do we balance them? We set standard derivation of observation as a hyper parameter!)."

Hyper parameters:

GNLL Var	0.01
Learning rate	0.0001
Latent dimension	64
Batch size	64
Number of epochs	500

The results:

- The model obtained the minimal test loss at epoch 382
- The reconstruction of random 8 images at epoch 382:

- A random sample of 8 images at epoch 382:

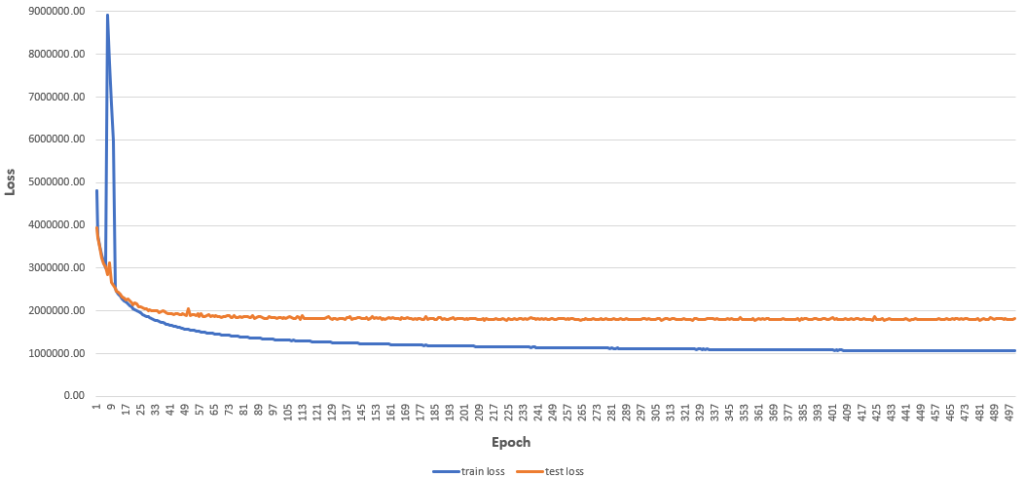
- Learning fast forwards:

- The reconstruction of random 8 images at epoch 500 (the last epoch):

- A random sample of 8 images at epoch 500 (the last epoch):

As seen in the pictures above, the model creates very blurry images, and the model does not improve since the 382 epoch.

Here is the graph that shows the loss as function of the epoch:



Part 2

In this part we tried to change and test the network with different parameters in order to produce less blurry and more detailed images.

At first, we tried to increase the Latent dimension so the model will learn more features from the original images and thus generating less blurry images. Increasing the latent dimension worked as expected, the model creates less blurry images and more detailed ones.

Changing the weights: We decide to try multiply different weights to each of GNLL and KL for scaling instead of changing the GNLL var. We choose numbers such that GNLL and KL be closer to each other because in the original version $GNLL >> KL$, so we change it to $KL * 10$, $GNLL / 100$. We thought that our algorithm doesn't give enough importance to KL and that the reason that images are blurry. As can be seen below its work pretty good.

Hyper parameters:

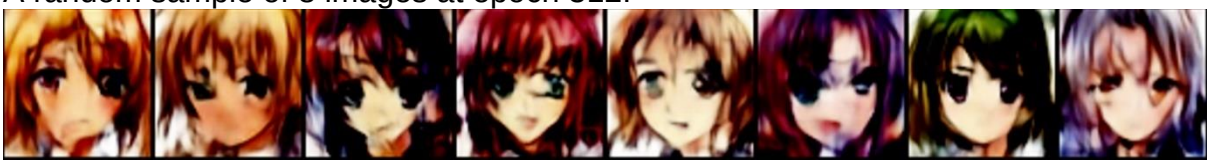
GNLL Var	0.01
Learning rate	0.0001
Latent dimension	512
Batch size	64
Number of epochs	500

The results:

- The model obtained the minimal test loss at epoch 312

- The reconstruction of random 7 (not 8 because of modesty reasons) images at epoch 312:

- A random sample of 8 images at epoch 312:



As seen in the pictures above, blurriness has been improved dramatically and more detailed image all around.

After we saw that scaling the weights, so the loss functions are closer, resulted with better quality images, we increased the KL weight, $KL * 50$, $GNLL / 100$. All other hyperparameters are the same.

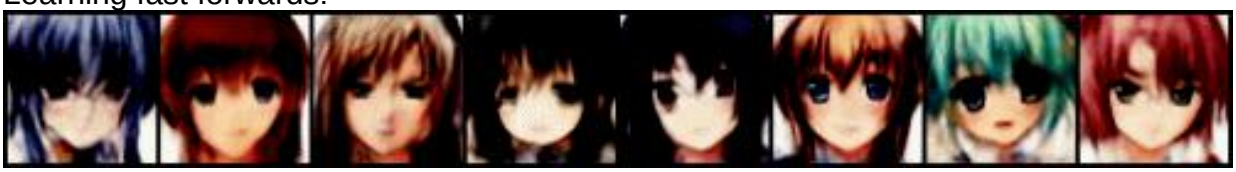
The results:

- The model obtained the minimal test loss at epoch 466

- The reconstruction of random 8 images at epoch 466:

- A random sample of 8 images at epoch 466:

- Learning fast forwards:

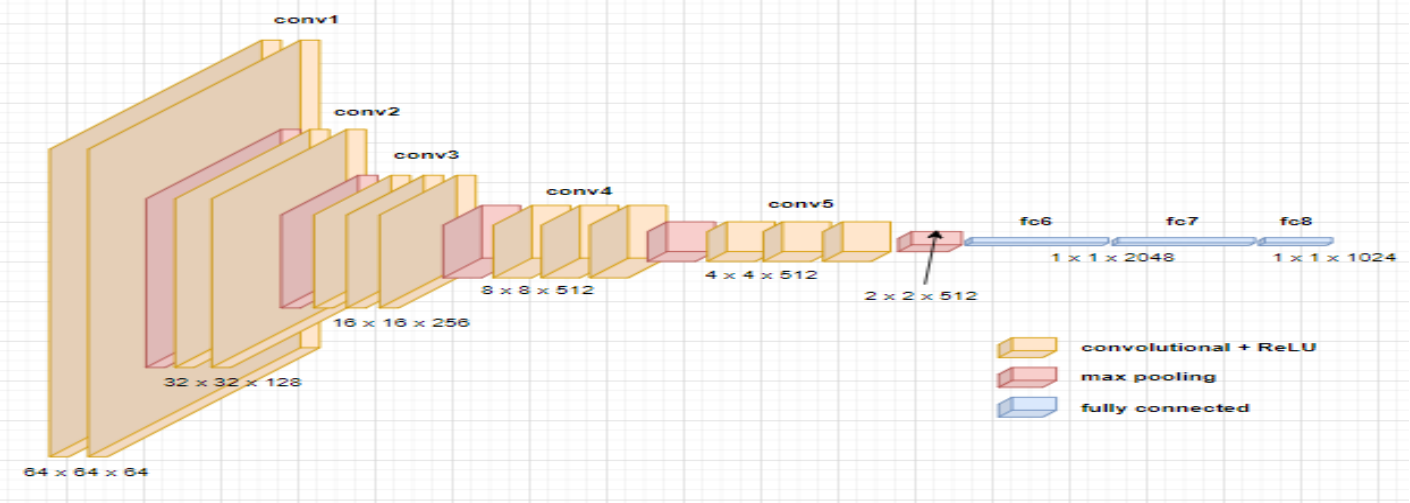


As expected, we got more improvement in this try.

Part 3

Our final **creative part** uses a VGG16 network as the encoder and mirrored VGG16 network as decoder. The idea came from a paper called “Generative Face Completion”. In this paper the researchers use VGG16 architecture to decode and encode images in order to complete celebrity faces, some part of each image was removed, and the algorithm success was measured by 2 discriminators: local discriminator that compare the reality of the generated part of the image to the part that was removed (original part), and a global discriminator that compare the reality of the fully generated image and the original image.
The model from the above paper shows great results in completing missing parts of real face images in high resolution, so, we thought that this encoder/decoder architecture will be able handle our goal in learning to generate new anime faces in much less resolution.

The following is VGG16 architecture which we used in our implementation: (with modifications for input image 64x64)



The first problem we have encountered was “exploding gradients” after only a few epochs. In order to find out what causing the gradients to explode we tracked the loss functions, the results showed that both, KL-divergence and GNLL, caused the gradients to explode asynchronously, each time, randomly, one of the loss functions got huge first and the train stage was unable to proceed. The solution for GNLL was to enlarge the var hyperparameter parameter until the issue has been solved. For the KL-divergence the solution was to choose smaller learning rate, because the train started with small loss that came from KL function and after a few optimizing steps the loss cause the gradients to explode.

Training the following model with the virtual machine grunted to us was a challenge, because it took about 30 hours for the algorithm to reach epoch 500, and the VM had sudden shutdowns after 20 hours of train.

Hyper parameters:

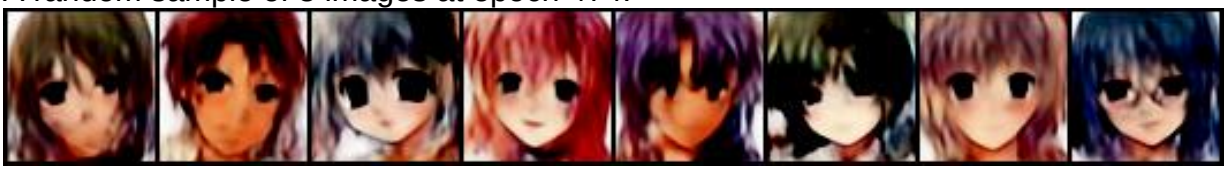
GNLL Var	0.1
Learning rate	0.00001
Latent dimension	512
Batch size	64
Number of epochs	500

The results:

- The model obtained the minimal test loss at epoch 474

- The reconstruction of random 8 images at epoch 474:

- A random sample of 8 images at epoch 474:



- Learning fast forwards:

As seen in the result, the new encoder/decoder did manage to reconstruct and create nice generated images. Thou it has one issue, visible in both generated and reconstructed images, is that the model did not capture the eyes correctly, the eyes appear to be black with no pupils.

Conclusions:

In the first part, we were unable to restore exactly what Wuga did, because of the difference from Tensorflow.v1 (which is an archived library) and PyTorch. In addition, it seems that Wuga's paper and Wuga's code have some differences. We assume that those two things are the reason why we did not achieve the same results as Wuga.

In the second part, in both of attempts we change latent dimension from 64 to 512 what leads the model to capture more details and be more accurate. Also, as we say and show above, when we gave different weights to GNLL and KL for scaling them we got more accurate images.

In the third part, if we compare the results from the model based on VGG16 to our models in the second part, we can see that its surely better than the first attempt in the second part, especially the random sample, and it look like it is even better than the second attempt in the second part, surely not worse.