Generating pokemon images with neural networks

Pokemonok képeinek generálása neurális hálóval

R. Kiss
First Affiliation

D. Sándor
Third Affiliation

D. Pogány
Second Affiliation

Abstract- Data generation is an important and often discussed subject in the field of Deep Learning. More specifically image generation is a topic, that showed significant improvement in the last few years, thus we decided to get an overview of the recent technologies. Our goal was to generate realistic looking pokemon images using neural networks, in this summary we will discuss possible approaches to the field. First we will give an overview of Generative Adversarial Networks, focusing on the DCGAN and the WGAN. Then we will discuss generation with Autoencoders, focusing on VAEs and AAEs. In the end we will present our results, and evaluate the modells with their strengths and weaknesses, based on the generated images.

Absztrakt- Az adatok generálása fontos és gyakran elemzett téma a Deep Learning területén. Ezen belül is a képgenerálás területe nagyfokú fejlődést mutatott az elmúlt évek során, ezért megpróbáltunk egy átfogó képet kapni a legfrissebb technológiákról. A feladatunk a következő volt: generáljunk valósnak tűnő pokemonokat neurális hálókkal, ebben az összefoglalóban a téma lehetséges megközelítéseit fogjuk bemutatni. Elsőként a Generative Adversarial Network-ökkel foglalkozunk, ezen belül a DCGAN-ra és a WGAN-ra fókuszálva. Ezután az Autoencoder típusú hálózatokkal való generálást fogjuk bemutatni, a VAE-re és az AAE-re fókuszálva. A végén bemutatjuk az eredményeinket, és kiértékeljük a modelleket a mutatott erősségeik és gyengeségeik alapján, a generált képeken keresztül.

■ INTRODUCTION TO GENERATIVE MODELING

The goal of generative models [6] are to produce new content similar to the training dataset. Generative models learn the joint probability of P(X,Y) as opposed to the traditional discriminative type which learns the conditional probability of P(Y|X=x).

■ STATE OF THE ART MODELS IN THE FIELD

Loss:

We define several distances between probability distributions to measure similarity. With these measures we can enforce our models (Pg) to learn (or approximate) the distribution of the training data (Pr). (Then if we sample from the modelled distribution, theoretically we should get outputs that are “similar” to the dataset.) First we define the *Wasserstein-1* distance for two distributions:

$$W(P_r, P_g) = inf_{\gamma \epsilon \prod(P_r,P_g)} E[||x-y||]$$

The next similarity measure is the *Kullback-Leiber* (KL) divergence:

$$D_{KL}(P_r||P_g) = \sum_i p_r(v_i)\log p_r(v_i) - p_r(v_i)\log p_g(v_i)$$

This formula gives us the non-overlapping area beneath two distributions.

Our last formula is the binary-crossentropy / log-loss:

$$H(P_r, P_g) = -\sum_i p_r(v_i)\log p_g(v_i)$$

which is equivalent to:

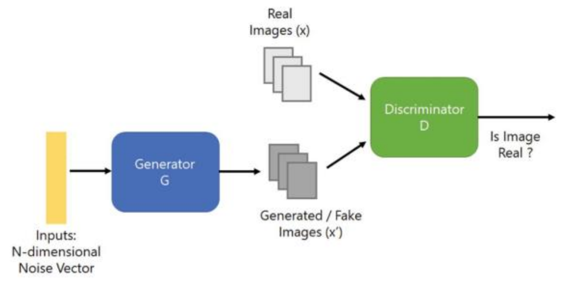
$$H(P_r, P_g) = D_{KL}(P_r||P_g) + S_r$$

Where Sr is the entropy of distribution A. The entropy is defined as follows:

$$S(v) = -\sum_i p(v_i)\log p(v_i)$$

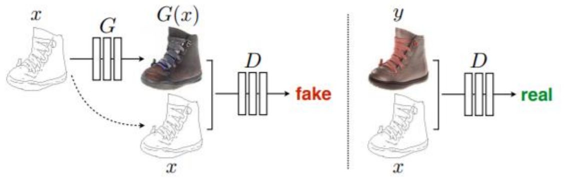
Since Pr is the training data, the Sr is a constant, so minimizing the log-loss is equivalent to minimizing the KL divergence.

Generative adversarial network [1][2][8] (GAN) models consist of two neural networks: a generator (g) and a discriminator (d). We call the connected model (output of g is connected to the input of d) adversarial (a). The discriminator tries to tell whether its input is real or is generated by the generator network. The generator gets noise as its input and outputs data similar to the training data format. The generator learns P(x,y), while the critic learns P(y|x). As we train the two nets together each tries to outperform the other and theoretically the generator learns to generate indistinguishable data from the real ones.

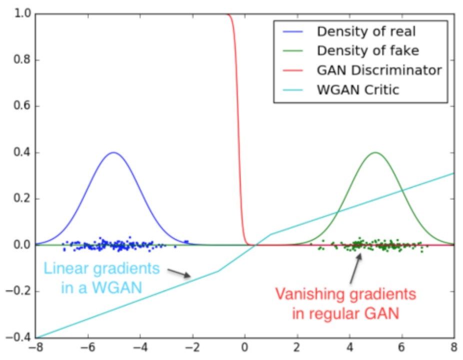

Real Images (x)
Discriminator D
Is Image Real ?
Generator G
Generated / Fake Images (x')
Inputs: N-dimensional Noise Vector

For image generation we used an architecture, called *DCGAN* [7] (deep convolutional GAN). This model has *convolutional* layers, because they can learn spatial patterns better than basic *feed-forward* layers.

We can feed extra information to our network, for example the edges of the training images, then theoretically the model learns to generate images based on the outlines (hand drawing to "real" image). This model is called cGAN [10] (conditional GAN).

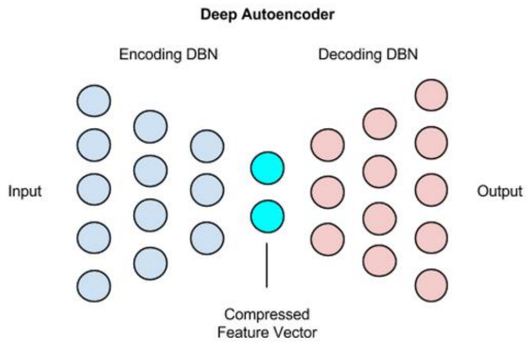


One other type of GANs is the WGAN [3] (wasserstein generative adversarial network). The inherent problem of GANs are convergence: the problem occurs, because if we train the discriminator and the generator simultaneously the discriminator will converge faster and become close to ideal in a relatively few steps, and this leads to a vanishing gradient for the generator, this process is called a collapse (which can be helped with, by adding noise to the input of the discriminator). The WGAN was introduced to solve the problem of convergence, by using the Wasserstein distance (or Earth mover's distance, because it shows how much earth is needed to be moved from one pile to another). This has a smoother gradient and thus it learns without expecting the generator to produce near perfect images.



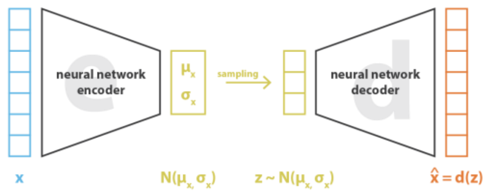
As we mentioned earlier, using the *binary-crossentropy*, we are minimizing the *KL-divergence* as well. *Wasserstein (W)* distance is much weaker than *KL-divergence*, so *W* is more likely to be continuous over the model parameters. Using *W* is much more sensible than *KL-divergence* when learning distributions supported by low dimensional manifolds.[3]

Variational autoencoders [4] (VAE) are special autoencoders. A standard autoencoder is actually a pair of two connected networks. The first one is called the encoder. The encoder is responsible for generating a latent representation of the inputs. The second network is the generator, which converts the dense latent representation back to the original input, or something very similar. The inner representation is also called the bottleneck of the network. Because the bottleneck is smaller than the original data, autoencoders can be used to data compression.



If we want to generate images similar to the training dataset with the autoencoder, we have to train it on the training data first. After that we can sample a vector from the latent representation, and the decoder generates some data, given the vector as input. But the latent space may not be continuous, hence if we sample a variation from the discontinuous latent space, the decoder will generate unrealistic data.

Variational autoencoders have continuous latent space, allowing easy random sampling and interpolation. To reach this we have to change the latent representation. The encoder generates two vectors instead of one, the first one contains the means, and the second one contains the standard deviations. The decoder samples a vector from these parameters, and use it as input.



If we alter the loss function, we can force the encoder to learn any probability distribution as latent space. If we want the encodings to be evenly distributed around the center of the latent space, we can add Kullback-Liebler divergence to the loss.

$$\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$$

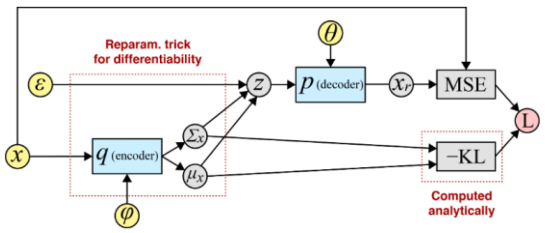
(divergence between our representation and the standard normal).

If we optimize the reconstruction loss and the Kullback-Liebler divergence together, the generator will produce similar outputs from nearby encodings. This means we can now randomly generate latent representation (moreover we can interpolate between them).

One more thing with the sampling. To have a differentiable network, we have to change the sampling, and it is called the reparametrization trick. We have the means (g(x)) and covariances (h(x)), the sampled data (z) will be:

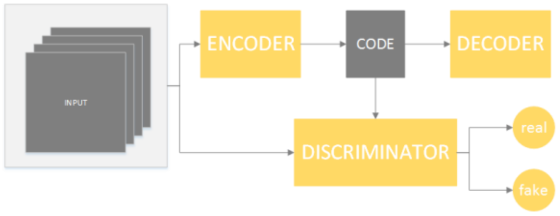
$$z = g(x) + h(x)*N$$

where N is a random variable N(0,1).



Now we can backpropagate the error on g(x) and h(x), so we can train the encoder.

Adversarial autoencoders [[5]] (AAE) have the same aim as VAE. The difference is that the network has a prior distribution to control the output of the encoder, and it uses an adversarial process instead of KL divergence. To force the encoder to generate similar representations to the prior distribution, the network has a generator and a discriminator part, as we have just seen it in the GAN.



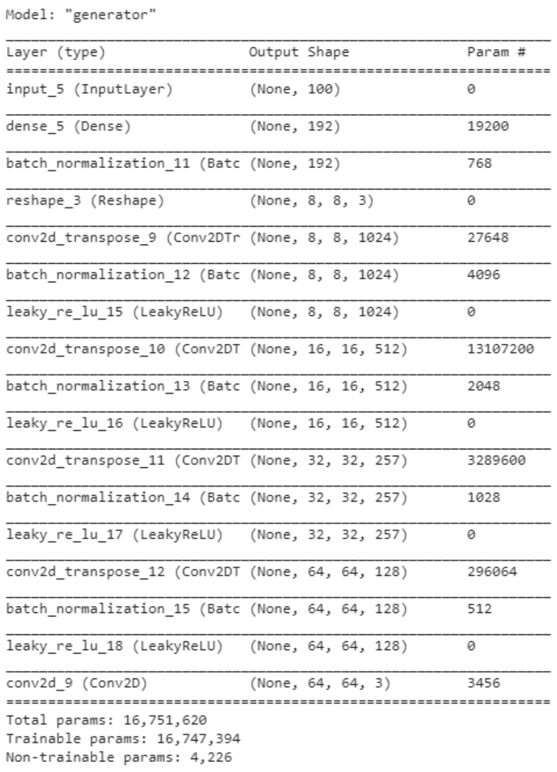
The generator here is the encoder, and the discriminator tries to tell, whether its input is a generated representation, or it is sampled from the prior distribution.

We train the network in two phases. At first we train the autoencoder, to minimize the reconstruction loss, it is called the reconstruction phase. After that, in the regulation phrase we train the discriminator.

■ ARCHITECTURAL OVERVIEW

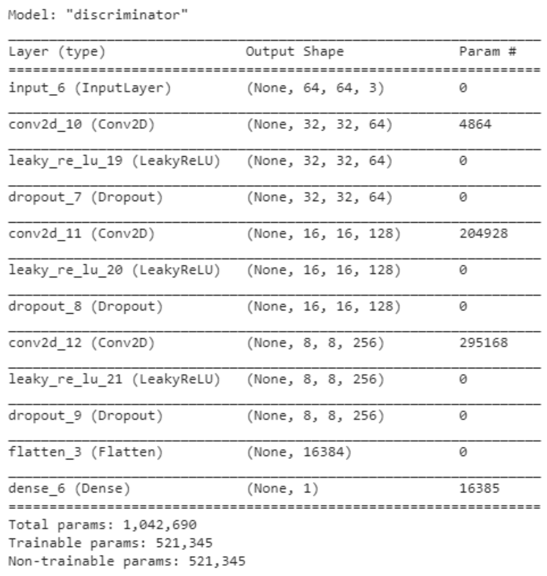
GAN:

The generator model consists of one dense layer, we feed the noise vector to this layer first, then 4 {Conv2DTranspose, BatchNormalization, LeakyReLU} units (each scales up the tensors size by 4) and a Conv2D layer is connected to the end with *tanh* activation. This layers output is the final output of the model, with the same dimensions as the original images.



```
Model: "generator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_5 (InputLayer)         (None, 100)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 192)               19200     
_________________________________________________________________
batch_normalization_11 (Batc (None, 192)               768       
_________________________________________________________________
reshape_3 (Reshape)          (None, 8, 8, 3)           0         
_________________________________________________________________
conv2d_transpose_9 (Conv2DTr (None, 8, 8, 1024)        27648     
_________________________________________________________________
batch_normalization_12 (Batc (None, 8, 8, 1024)        4096      
_________________________________________________________________
leaky_re_lu_15 (LeakyReLU)   (None, 8, 8, 1024)        0         
_________________________________________________________________
conv2d_transpose_10 (Conv2DT (None, 16, 16, 512)       13107200  
_________________________________________________________________
batch_normalization_13 (Batc (None, 16, 16, 512)       2048      
_________________________________________________________________
leaky_re_lu_16 (LeakyReLU)   (None, 16, 16, 512)       0         
_________________________________________________________________
conv2d_transpose_11 (Conv2DT (None, 32, 32, 257)       3289600   
_________________________________________________________________
batch_normalization_14 (Batc (None, 32, 32, 257)       1028      
_________________________________________________________________
leaky_re_lu_17 (LeakyReLU)   (None, 32, 32, 257)       0         
_________________________________________________________________
conv2d_transpose_12 (Conv2DT (None, 64, 64, 128)       296064    
_________________________________________________________________
batch_normalization_15 (Batc (None, 64, 64, 128)       512       
_________________________________________________________________
leaky_re_lu_18 (LeakyReLU)   (None, 64, 64, 128)       0         
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 64, 64, 3)         3456      
=================================================================
Total params: 16,751,620
Trainable params: 16,747,394
Non-trainable params: 4,226
```

The discriminator model has 3 {Conv2D, LeakyReLU, Dropout} units, and after flattening, a single dense unit at the end with sigmoid activation.

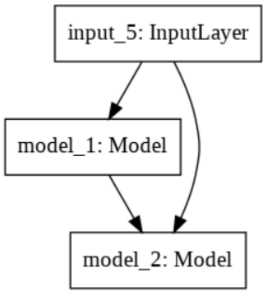


```
Model: "discriminator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         (None, 64, 64, 3)         0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 32, 32, 64)        4864
_________________________________________________________________
leaky_re_lu_19 (LeakyReLU)   (None, 32, 32, 64)        0
_________________________________________________________________
dropout_7 (Dropout)          (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 16, 16, 128)       204928
_________________________________________________________________
leaky_re_lu_20 (LeakyReLU)   (None, 16, 16, 128)       0
_________________________________________________________________
dropout_8 (Dropout)          (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 8, 8, 256)         295168
_________________________________________________________________
leaky_re_lu_21 (LeakyReLU)   (None, 8, 8, 256)         0
_________________________________________________________________
dropout_9 (Dropout)          (None, 8, 8, 256)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 16384)             0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 16385
=================================================================
Total params: 1,042,690
Trainable params: 521,345
Non-trainable params: 521,345
```

As a loss function we used *binary-crossentropy*.

DCGAN:

Our DCGAN architecture is basically the same as the GAN models, however we introduced a new input layer to each model (the edges of the original images).



We introduced some convolutional layers, and flattened their output to the generator.
For the discriminator, we simply put the edges layer as a fourth channel to the input.

WGAN:

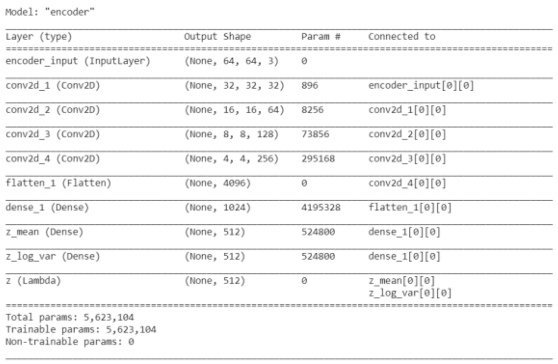
The difference to our GAN model is the loss function. This model uses the *Wasserstein* distance as a loss function: To calculate the Wasserstein loss is difficult to implement thus we introduce the Lipshitz function.

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2|$$

The new model has to find the (1 - Lipschitz function), which is achievable by an architecture similar to the the DCGAN's discriminator, we call the new model a critic. The differences are the following: We change the sigmoid output to a linear function, thus it does not predict whether an image is real or not, instead it assigns a rating to each image, based on confident it is that the image is real. Another difference is, that since its optimizing the 1 - Lipschitz function, to achieve this we introduce a weight clip after every epoch, to a certain range. For generator network we can still use the one used in the DCGAN, but with the new loss function it is scored based on how well it makes the discriminator predict the wrong way.

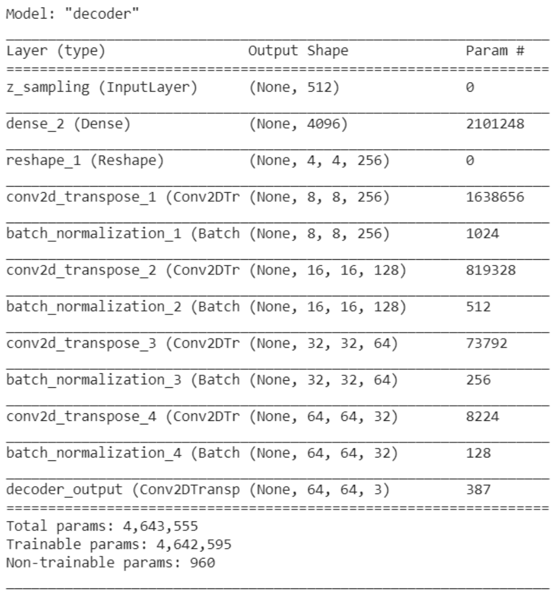
VAE:

The encoder has 4 convolutional layer, followed by a flatten and dense layer. After that it has a z_mean and a z_log_var layer (g(x) and h(x)), and finally a z layer is sampled from them. With 512 latent size and 64 as image size it has 5.6 million trainable parameters.



```
Model: "encoder"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
encoder_input (InputLayer)      (None, 64, 64, 3)    0
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 32, 32, 32)   896         encoder_input[0][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 16, 16, 64)   8256        conv2d_1[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 8, 8, 128)    73856       conv2d_2[0][0]
__________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 4, 4, 256)    295168      conv2d_3[0][0]
__________________________________________________________________________________________________
flatten_1 (Flatten)             (None, 4096)         0           conv2d_4[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 1024)         4195328     flatten_1[0][0]
__________________________________________________________________________________________________
z_mean (Dense)                  (None, 512)          524800      dense_1[0][0]
__________________________________________________________________________________________________
z_log_var (Dense)               (None, 512)          524800      dense_1[0][0]
__________________________________________________________________________________________________
z (Lambda)                      (None, 512)          0           z_mean[0][0]
                                                                 z_log_var[0][0]
==================================================================================================
Total params: 5,623,104
Trainable params: 5,623,104
Non-trainable params: 0
__________________________________________________________________________________________________
```

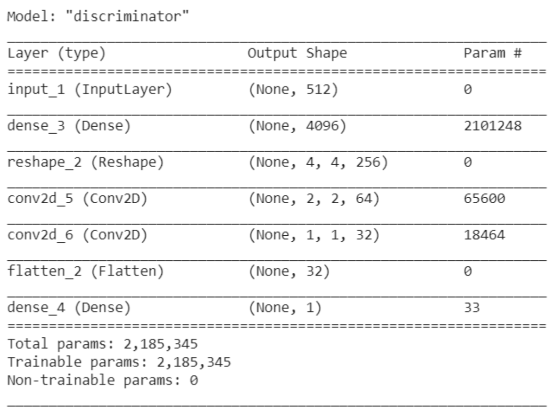
The decoder is a mirror image of the encoder with 4 transpose convolutional layers and batch normalization between them. It has 4.6 million trainable parameters.



```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
z_sampling (InputLayer)      (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 4096)              2101248
_________________________________________________________________
reshape_1 (Reshape)          (None, 4, 4, 256)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 8, 8, 256)         1638656
_________________________________________________________________
batch_normalization_1 (Batch (None, 8, 8, 256)         1024
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 16, 16, 128)       819328
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 32, 32, 64)        73792
_________________________________________________________________
batch_normalization_3 (Batch (None, 32, 32, 64)        256
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 64, 64, 32)        8224
_________________________________________________________________
batch_normalization_4 (Batch (None, 64, 64, 32)        128
_________________________________________________________________
decoder_output (Conv2DTransp (None, 64, 64, 3)         387
=================================================================
Total params: 4,643,555
Trainable params: 4,642,595
Non-trainable params: 960
_________________________________________________________________
```

AAE:

The autoencoder part is the same as in the VAE. The discriminator is a network with 2 convolutional layers between 2 dense layers, and it has 2.1 million trainable parameters.



```
Model: "discriminator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 4096)              2101248
_________________________________________________________________
reshape_2 (Reshape)          (None, 4, 4, 256)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 2, 2, 64)          65600
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 1, 1, 32)          18464
_________________________________________________________________
flatten_2 (Flatten)          (None, 32)                0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 33
=================================================================
Total params: 2,185,345
Trainable params: 2,185,345
Non-trainable params: 0
_________________________________________________________________
```

■ OUR IMPLEMENTATION

DATA:

We used the pokemon image dataset, which consists of 819 256x256 rgba images. We downscaled them to 64x64 rgb images. This is not that much for training the networks, so we used augmentation to increase the training data size. The augmented dataset contains 16380 images.

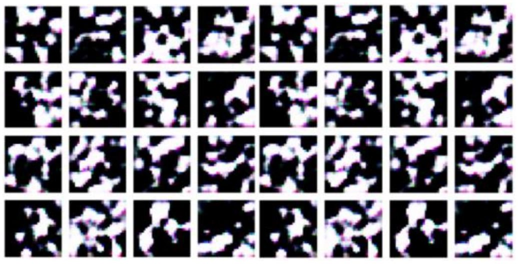
GAN:

As noise input to the generator, we used Gaussian noise (N(0,1)).

We tried multiple training strategies:

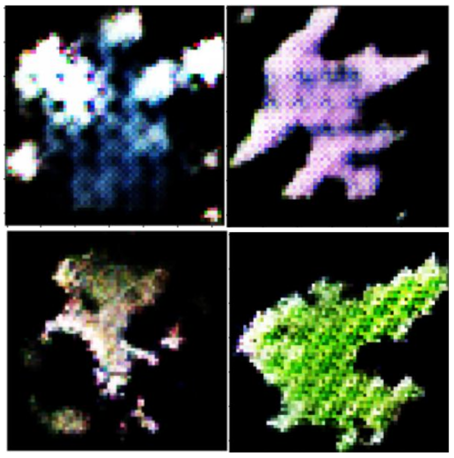
- Which model to train?
  - simply train g and d in every epoch
  - only train the weaker net (which has the worse accuracy)
- Weights
  - all weights are trainable
  - the discriminator weights are fixed when training the generator

We tried to pretrain the generator (train with an AE) and got results like:

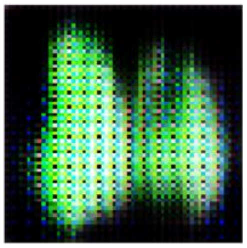


This strategy did not work well. It seems the network managed to learn shapes, but it lacks colors (the weights to each rgb channel may be very similar).

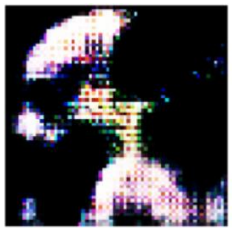
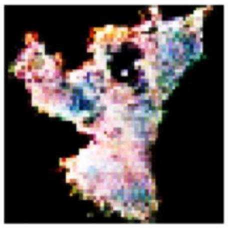
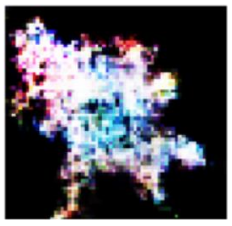
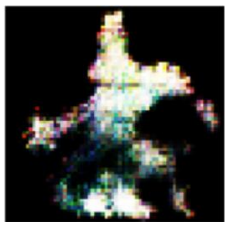
Without pretraining, the best results:



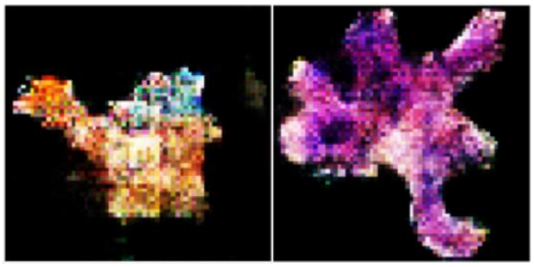
On the top right image we can see that the generator learned to shade the object “correctly”. (The left edges are brighter while the right ones are darker)



As hyperparameter optimization we tried Adam as optimizer, this produced the worst results.

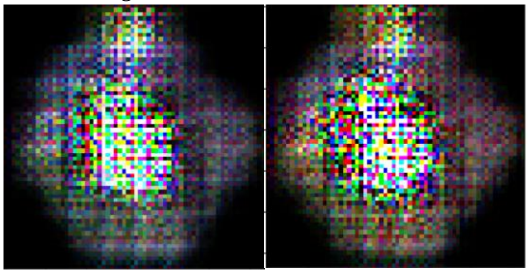
	Not fixed	Fixed weights
Train each		
Train weaker		

Not fixing the discriminator weights can result better images, so we tried to generate pokemons with a bigger generator model (7 million weights) and with not fixed discriminator. The results were not better.



CGAN:

For the extra input data we gave the model the edges of original training images. We tried multiple times and after ~2500 epochs the generator stucks at a global minima, and we were not able to get it out from there. It learns to draw something in the middle and near the edges, the images are black (like in all the training images). It generates images like these:

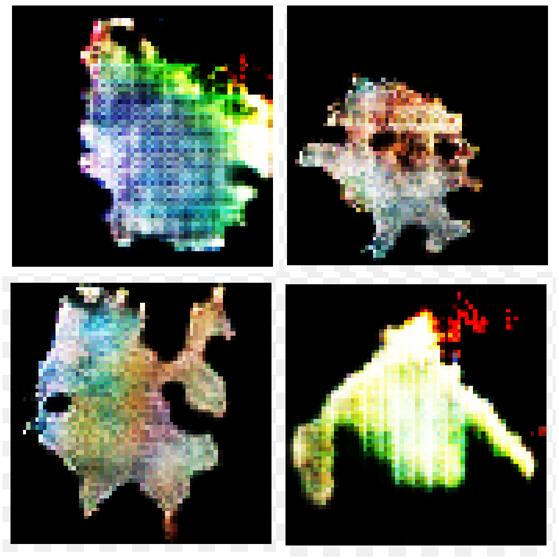


WGAN:

To implement the WGAN we used the DCGAN as base and switched the discriminator to a critic, with linear activation and a weight clip of (-0.1,0.1). The loss function was the following:

```
from keras import backend as K
def wasserstein_loss(y_true, y_pred):
    return K.mean(y_true * y_pred)
```

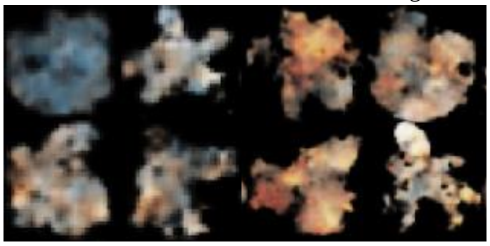
Since we used keras another challenge was the implementation of the loss, and instead of the traditional, we opted for the loss above, with the labels of the images being 1 for real and -1 for generated images, wich gives the same result as the original implementation. We used the same number of layers as in the DCGAN, for optimizer we used RMSProp, and trained the generator five times for every training of the discriminator.
The best results for the WGAN:



Interesting observations: On the first and the last image we can see fire like patterns, which occurred on multiple images, and on the second and third images we see black dots which could be interpreted as eyes (although on some images there were a lot more, but that is a common problem with GANs), therefore we can see that it was able to learn distinguishing features.

VAE:

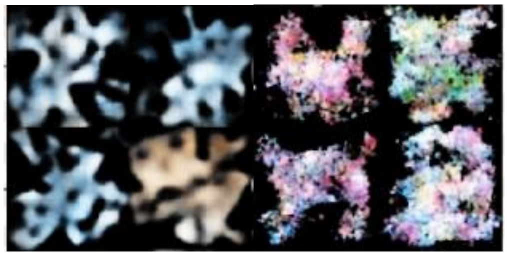
We made the VAE with size 512 latent dimension, and used the 64x64 images, but tried it with resizing to 32x32 which gave considerably better results, but unfortunately they were too small (which is a known property of the VAE). We used four Convolutional layers for the encoder and four Transpose layers for the decoder with batch normalization between them. We used the Adam optimizer.
The best results for size 32x32 and 64x64 images:



AAE:

For the AAE we used the same structure as the VAE with a discriminator with one dense and two convolutional layers. For optimizer we used Adam. We generated images of size 64x63 and 128x128.

The generated images (64x64) and (128x128):



■ CONCLUSION AND FUTURE WORK

It seems so, GANs can generate higher resolution images without getting blurry. However images by the AEs are more pokemon-like in our opinion (They tend to be more colorful, like real pokemons).
It would be interesting to find out why the cGAN gets stuck during training.
There are other GAN models, we have not tried yet, for example the EBGAN [[9]] (Energy Based GAN), which is basically a GAN with an autoencoder discriminator.

■ REFERENCES

[1] Pattanayak, S., Pattanayak, & John, S. (2017). Pro Deep Learning with TensorFlow. Apress.

[2] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In Advances in neural information processing systems (pp. 2672-2680).

[3] Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein gan. arXiv preprint arXiv:1701.07875.

[4] Doersch, C. (2016). Tutorial on variational autoencoders. arXiv preprint arXiv:1606.05908.

[5] Makhzani, A., Shlens, J., Jaitly, N., Goodfellow, I., & Frey, B. (2015). Adversarial autoencoders. arXiv preprint arXiv:1511.05644.

[6] Elton, D. C., Boukouvalas, Z., Fuge, M. D., & Chung, P. W. (2019). Deep learning for molecular design-a review of the state of the art. Molecular Systems Design & Engineering.

[7] Fang, W., Zhang, F., Sheng, V. S., & Ding, Y. (2018). A method for improving CNN-based image recognition using DCGAN. Comput., Mater. Continua, 57(1), 167-178.

[8] Goodfellow, I. (2016). NIPS 2016 tutorial: Generative adversarial networks. arXiv preprint arXiv:1701.00160.

[9] Zhao, J., Mathieu, M., & LeCun, Y. (2016). Energy-based generative adversarial network. arXiv preprint arXiv:1609.03126.

[10] Isola, P., Zhu, J. Y., Zhou, T., & Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1125-1134).