ASSIGNMENT 4 — PRACTICAL PART
GENERATIVE MODELS (GANS)

Samuel Laferrière[*] & Joey Litalien[†]
IFT6135 Representation Learning, Winter 2018
Université de Montréal
Prof. Aaron Courville
{samuel.laferriere.cyr,joey.litalien}@umontreal.ca

1 OVERVIEW

For this assignment, we chose to implement in PyTorch a Generative Adversarial Network (GAN) as first introduced by Goodfellow et al. (2014). The variant we used for comparison is a Wasserstein GAN (WGAN) from Arjovsky et al. (2017). The class `DCGAN` implements a Deep Convolutional GAN and contains both a `Discriminator` D and `Generator` G. Both entities can be trained independently using the methods `DCGAN.train_*`, meaning that we can customize the number of times we update each model. These training routines implement three types of loss functions: the original minmax loss, the Wasserstein loss and the least squares lost from Mao et al. (2016). The last GAN type will not be evaluated in this assignment, but generated images from our LSGAN are available in the Appendix.

Every hyperparameters related to training (*e.g.* learning rate and momentum) are set in `CelebA`, a wrapper class for the generative task at hand. To load a pre-trained generative model and sample images from its distribution, simply instantiate `DCGAN`, call `DCGAN.load_model(filename)` with the saved PyTorch weights, and run `DCGAN.generate_img(n)`.

2 ARCHITECTURE

2.1 TRAINING THE NETWORKS

We took inspiration from the DCGAN architecture from Radford et al. (2015) to build our two GANs. The architectures for the discriminator and generator are given in the tables below and are used on both our vanilla DCGAN and our WGAN. The middle columns are kernel size (k), stride (s) and zero-padding (p), respectively.

In	Out	Module	k	s	p	Normalization	Activation
100	512	Linear + Reshape				BatchNorm1D	ReLU
512	256	ConvTranspose2D	4	2	1	BatchNorm2D	ReLU
128	64	ConvTranspose2D	4	2	1	BatchNorm2D	ReLU
64	3	ConvTranspose2D	4	2	1	None	Tanh

Table 1. DCGAN Generator architecture.

As suggested in the paper, we did not apply batchnorm to G's output layer and D's input layer to avoid sample oscillation and model instability. The latent variable is first project to a $512 \times 4 \times 4 = 8192$-dimensional vector using a dense layer and then reshaped to a volume of $512 \times 4 \times 4$ for the first fractionally-strided convolution. We used a kernel size of 4 instead of 5 and removed all biases in the generator network to speed up the computations.

We further detail our choice of hyperparameters as follows.

[*]Student ID P0988904
[†]Student ID P1195712

In	Out	Module	k	s	p	Normalization	Activation
3	64	Conv2D	4	2	1	None	LeakyReLU
64	128	Conv2D	4	2	1	BatchNorm2D	LeakyReLU
128	256	Conv2D	4	2	1	BatchNorm2D	LeakyReLU
256	512	Conv2D	4	2	1	BatchNorm2D	LeakyReLU
512	1	Conv2D	4	1	0	None	None

Table 2. DCGAN Discriminator architecture.

Hyperparameter	Symbol	DCGAN	WGAN
Learning rate	α	2×10^{-4}	5×10^{-5}
Momentum	β, β^2	0.5, 0.999	None
SGD Optimizers	—	Adam	RMSProp
D/G updates ratio	n_{critic}	1	5

Table 3. Training hyperparameters for DCGAN and WGAN.

These parameters are the ones suggested by the original authors and tested on the LSUN dataset. We experimented with different values (*e.g.* Adam for WGAN) but GANs being incredibly hard to train, we settled for these values. Theoretically, the discriminator in a WGAN should be fully converged to get the best estimate of the Wasserstein distance but in practice this does not work so well. Hence, if G_{iter} is the number of times G has been updated so far in the training, we started at $n_{\text{critic}} = 20$ for $G_{\text{iter}} < 50$ (1000 minibatches) or whenever $G_{\text{iter}} + 1 \pmod{500} = 0$. This number is reduced to 5 otherwise. This is rather arbitrary but somewhat follows the heuristic described by Arjovsky himself on his GitHub repo of the original WGAN.

The dataset contains 202 599 RGB images of size $3 \times 64 \times 64$. These faces were first normalized and fed into our networks in minibatches of size 64. To train these models, we provided Shell scripts that accept a number of arguments for the optimizers, learning rate, CUDA switch, RNG seed (for debugging), and much more. These are located in `src/train`. Below are the training times for our models.

Model	Training time	Avg epoch time	Avg batch time
Vanilla GAN	3 hrs 34 mins	4 mins 15 s	56 ms
WGAN	2 hrs 38 mins	3 mins 9 s	35 ms

Table 4. Training our vanilla GAN and Wassterstein GAN on an NVIDIA GTX 1080.

Animations of the 50 epochs with fixed latent variables can be found in `src/checkpoints` under each model's `video` folder.

2.2 INCREASING THE FEATURE MAP SIZE

To illustrate the three upsampling schemes, we reuse a toy example from Odena et al. (2016). Here, we used a stride of $s = 1$ (in white) and a constant kernel $k = (a, b, c) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ of size 3 (in blue). The input vector $\mathbf{x}$ (in pink) is a discretized color gradient, *i.e.* $\mathbf{x} = (9, 7, 5, 3, 1) \odot 10^{-1}\mathbf{1}_5$. The upsampled vector $\mathbf{y}$ is obtained from the matrix $\mathbf{C}^\top$ and $\hat{\mathbf{x}}$.

Given an input vector, a fractionally-strided convolution first adds a stride inbetween elements, and then perform a convolution (in blue). This is done in Figure 1, where the kernel is flipped and $\mathbf{C}^\top$ is a 3×4 sparse matrix. Note the *checkerboard effect* present in $\mathbf{y}$ due to kernel overlapping.

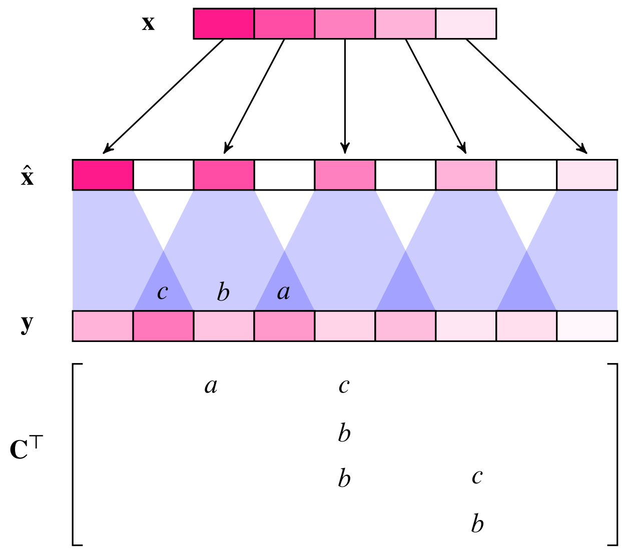


Figure 1. Fractionally-strided convolution.

In the case of a nearest neighbor (NN) convolution, we first expand $\mathbf{x}$ by duplicating the elements to obtain $\hat{\mathbf{x}}$ and then perform the convolution. Doing the arithmetic gives us the matrix $\mathbf{C}^\top$ depicted in Figure 2 below. We observe that the upsampled vector $\mathbf{y}$ is much more smoother and is free of any artifact other than at its extremes.

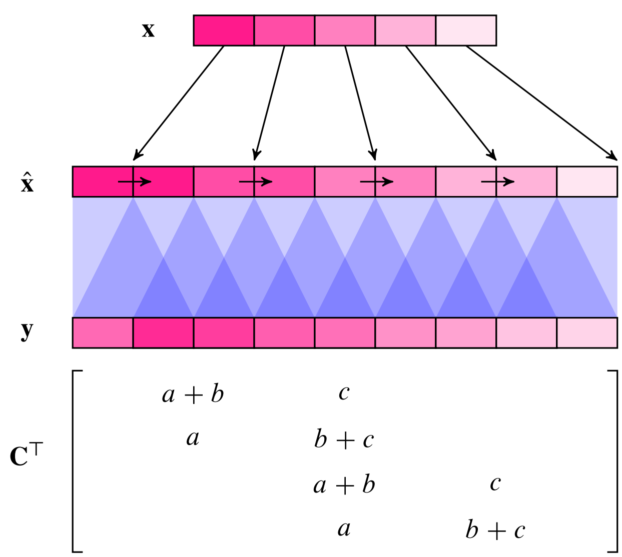
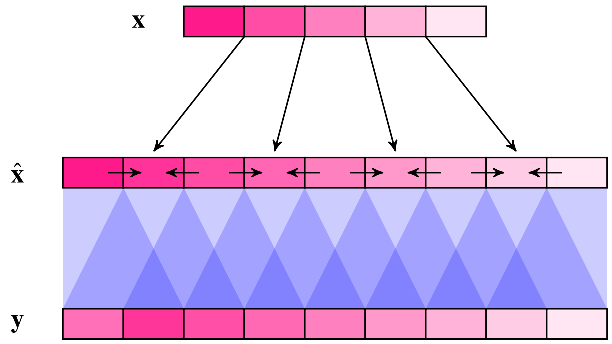


Figure 2. Nearest neighbor convolution.

Finally, the bilinear resize convolution maps $\mathbf{x}$ to $\hat{\mathbf{x}}$ by interpolating between the values. This introduces $\frac{1}{2}$ terms in the deconvolution matrix which needs to be 4×4 in this case. This is shown in Figure 3.

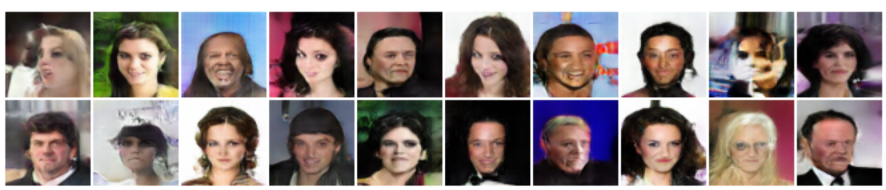


$$\mathbf{C}^\top \quad \begin{bmatrix} a+\frac{1}{2}b & \frac{1}{2}b+c & & \\ \frac{1}{2}a & \frac{1}{2}a+b+\frac{1}{2}c & \frac{1}{2}c & \\ & a+\frac{1}{2}b & \frac{1}{2}b+c & \\ & \frac{1}{2}a & \frac{1}{2}a+b+\frac{1}{2}c & \frac{1}{2}c \end{bmatrix}$$

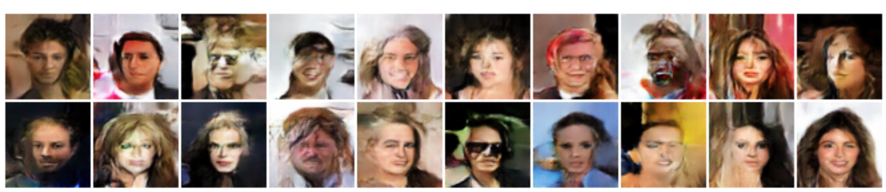
Figure 3. Bilinear resize convolution.

3 QUALITATIVE EVALUATIONS

3.1 DCGAN VS. WGAN VISUAL COMPARISON



(a) Vanilla GAN



(b) WGAN

Figure 4. Comparing our vanilla GAN (*top*) with its Wasserstein counterpart (*bottom*). More images can be found in the Appendix.

Against all odds, it seems like our GAN outperforms our WGAN. Even if the latter does not show any sign of mode collapse as compared to the former (see Appendix), the quality of the samples produced by the DCGAN are clearly superior. Our GAN's outputs are more crisp and less blurry. In particular, the Wasserstein GAN samples look more like oil paintings and very few samples could be confused with an actual human face.

3.2 LATENT SPACE EXPLORATION

To explore the face manifold in latent space, we iterated over all 10^2 dimensions of two given $\mathbf{z}$'s and changed the value to $\pm 3\sigma^2 = \pm 3$ to amplify the signal. We selected the dimensions we thought gave the most notable differences and plot them (Fig. 5). Among the visual variations we observed are open/close mouth, skin color, hair fringe, hair colour, cheek and jaw shape, background, and gender changes. Interestingly, we also have a semblance of face orientation change in the woman case for GAN (first and fourth from the right).

(a) Vanilla GAN

(b) WGAN

Figure 5. Changing a single dimension in latent space for different generative models. Original face $\mathbf{x} = G(\mathbf{z})$ is the left-most image.

TODO: Discuss main differences in quality

3.3 Screen and Latent Space Interpolation

Interpolating in screen space (Fig. 6) obviously gave very poor results as we simply interpolate between pixels and completely disregard the underlying structure of the generator G. Only the first and last images look like actual human faces; anything in the middle has ghosting artifacts since it is just a blend of RGB channels.

Interpolating in latent space (Fig. 7), however, gives pleasant results. Indeed, any $\mathbf{z}'$ seems to yield a plausible celebrity face. This is because when we interpolate the latent variables, we "walk" on the face manifold of G and so any value should somewhat give a realistic face.

To better visualize the interpolation process, we created looping GIFs (and MP4 videos to avoid compression artifacts) over 50 frames (*i.e.* $\alpha = i/50$). These are located in the `explore/screen_space` and `explore/latent_space` directories of the project source. You can create an interpolating sequence between two random seeds by running the provided Python script with a pretrained generative model. A series of arguments such as model type and number of frames can be specified. For more info, simply run the program with the help flag.

(a) Vanilla GAN

(b) WGAN

Figure 6. Interpolating in screen space for different generative models. Each image is generated using two fixed latent variables and computing $\mathbf{x}' = \alpha\mathbf{x_0} + (1-\alpha)\mathbf{x}_1,\ \alpha \in [0, 1]$.

4 QUANTITATIVE EVALUATIONS

4.1 RESULTS FOR INCEPTION AND MODE SCORE

We computed the Inception score and Mode score for both models using $N = 4096$ generated images. The results are shown in the bar graph below.

4.2 DISCUSSION ON SCORING METHODS

TODO

(a) Vanilla GAN

(b) WGAN

Figure 7. Interpolating in latent space for different generative models. Each image is generated using a different latent variable $\mathbf{z}' = \alpha \mathbf{z}_0 + (1-\alpha)\mathbf{z}_1,\ \alpha \in [0, 1]$.

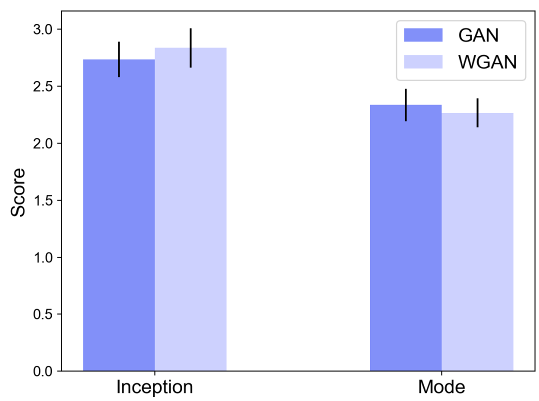


Figure 8. Evaluating the performance of our trained GAN and WGAN using different metrics.

References

Martín Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pp. 214–223, 2017.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pp. 2672–2680, 2014.

Xudong Mao, Qing Li, Haoran Xie, Raymond Y. K. Lau, and Zhen Wang. Multi-class generative adversarial networks with the L2 loss function. *CoRR*, abs/1611.04076, 2016.

Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

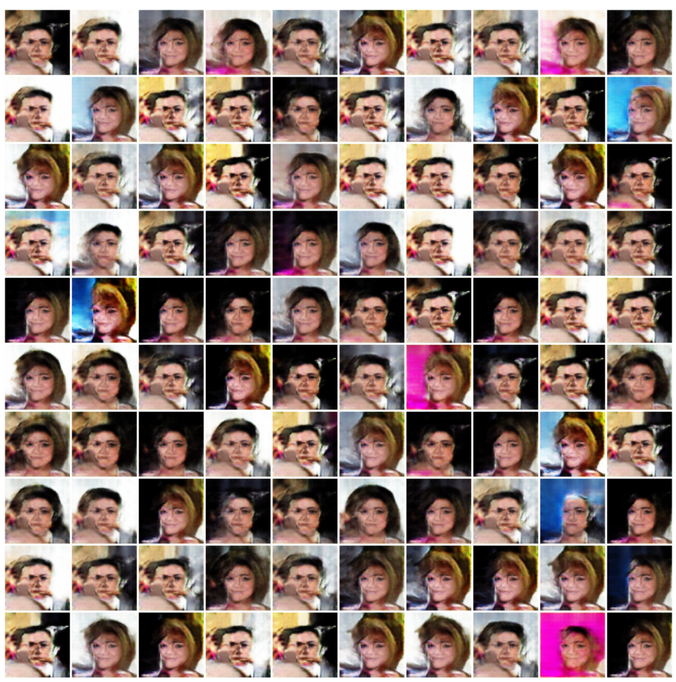
A Generated Examples

(a) Vanilla GAN

(b) WGAN

Figure 9. 100 generated images using both models trained for 50 epochs. All images look more or less plausible, but we can note a small mode collapse in the case of the vanilla GAN as multiple values of $\mathbf{z}$ map to the same image (1,2).

(a) LSGAN after 44 epochs



(b) LSGAN after 50 epochs

Figure 10. 100 generated images using our trained LSGAN using the same hyperparameters as the vanilla GAN. Total mode collapse occurs at the end of training time.