Danmarks Tekniske Universitet

Colorizing Grayscale Images using UNet

Lab report
by

Kristian Friis Nielsen (s204120)
Mads Andersen (s204137)
Iván Viemoes Cuevas (s205823)
Anders Nørskov (s183995)

Introduction to Intelligent Systems, 02461

Technical University of Denmark
January 22, 2021

Contents

Student	1	2	3.1	3.2	3.3	4	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8
s204120	X	X		X			X	X	(X)					
s204137	X	X				X			X	X	(X)			
s205823	X	X	X				(X)				X			X
s183995	X	X	(X)	(X)	X							X	X	

X denotes primary responsibility. (X) denotes secondary responsibility. However, everyone has contributed to all sections.

1. ABSTRACT

Restoring grayscale images, such as historically significant black and white photographs, have been a very manual and time consuming operation. Attempts at automating this process has become a highly contested field in machine learning with only specialized models being used for each unique task. In order to propose a general model we examine the scalability of adversarial networks and their ability to colorize datasets of increasing complexity. We implement a UNet with Instance Normalization as in [1] with adversarial Wasserstein loss [2], R_1 Regularization [3] and Instance Noise [4], and introduce a novel approach called Instance Smoothing much akin to Label Smoothing [5] to stabilize training further. We confirmed the agility of the model across different complexities of datasets, showing that the underlying architecture retained stability and performance as long as the UNet was scaled appropriately, achieving a FID of 16.804 on the *Athletic Fields* category of the Places 365 dataset. The potentials of adversarial networks greatly outweigh their complexity and deliver promising results when trained for extended periods of time with somewhat saturated colorization and without the same brownish tinge affecting results using the simpler MSE loss.

2. INTRODUCTION

RGB images are 3D volumes containing color intensities over three channels which are linearly transformed into a grayscale image in a lossy way – that is grayscale images do not contain any hue information, only luminosity, effectively making perfect recoloring a seemingly impossible task. We want to implement a neural network that can reverse this transformation.

Image colorization is a very model intensive task as it must both detect image features and accordingly distribute appropriate color. Additionally, object features can be arbitrarily colored according to a context and pertaining to the object itself which further increases complexity. Different methods for grayscale-colorization using neural networks have been proposed with varying results [6][7]. The objective is not to colorize objects exactly like the ground truth but rather be able to fool the human perception system.

In this report we implement a UNet with instance normalization [1] and build upon this with adversarial networks as in pix2pix [7] without the stochastic element. We implement various methods of stabilizing adversarial network training such as R_1 regularization and instance noise, and introduce a novel approach we call instance smoothing much akin to label smoothing [5]. To achieve perceptually distinguishable images for a human we train the model to predict and discriminate in the CIELAB colorspace.

We hypothesize the adversarial networks to be able to fool a human observer, and to greatly improve the results obtained by conventional colorization models which often use a per-pixel Euclidean distance measure that often leads to a desaturated, brownish tinge.

3. METHODS AND DATA

3.1. Data

To test our model we use the *MNIST* hand written digit dataset [8] and create a toy dataset to apply to our problem. We colorize the *MNIST* dataset by assigning an interval on the unit circle for each digit and uniformly sample a value from this interval and project it onto a unit square and minmax-normalize it to give us the *a and *b channels matching the CIELAB (L*a*b) colorspace with the definitions in Open Computer Vision 2 [9] for Python with each value ranging from 0 to 1. As such we sample the colors depending on the digit as follows:

$$v_\theta = 2\pi(D+u)/10 \quad \text{with} \quad u \sim \mathcal{U}[-0.5, 0.5]$$
$$(a_\theta, b_\theta) = (\sin(v_\theta),\ \cos(v_\theta))$$
$$(*a, *b) = \left(\frac{0.5a_\theta}{\max(|a_\theta|, |b_\theta|)} + 0.5,\ \frac{0.5b_\theta}{\max(|a_\theta|, |b_\theta|)} + 0.5\right)$$

with $D \in \{0, 1, 2, \ldots, 9\}$ for each digit. The L*a*b color space mimics human perception and spreads out each color such that different values of *a and *b are perceptually different. Due to the small range on the unit circle and the sequential uniform distribution of the *a and *b channels two sequential digits can be perceptually identical as seen on figure 1.a. The digits vary in quality and some are crudely written and therefore difficult for any model to predict.

In order to evaluate on more complex data, we use the *Athletic Fields* category from the Places 365 dataset [10]. This set contains 40,000 images compromised of both the standard and challenge subsets using the small 256x256 resolution image size for both. We chose the *Athletic Fields* category because it contains easily distinguishable features with mostly uniformly spread well-defined saturated colors focusing only on a single category of places. The *Athletic Fields* dataset most prominently features green terrain but it also regularly varies between red and blue as seen on some images in figure 1.b depending on what sports event is being depicted. Furthermore, it features arbitrarily colored garments and varying skin colors depending on nationality, and what sports team is playing, etc. Many of these arbitrarily colored objects are hard to distinguish in the gray colorspace, since there is some overlap between colors, e.g. yellow, cyan and white etc. due to the linear transformation. All together, this poses a substantially increased challenge for the model without being too complex compared to the whole *Places 365* dataset.

For evaluating results we use the Fréchet inception distance (FID)[11]. This metric uses the Inception v3 [12] model to define a distance between the data distributions of generated and real images - lower distances meaning the generated images have perceptually similar features to the real images.

3.2. Architecture

We use the UNet with instance normalization described by [1] and similar to [7] composed of an encoder and decoder network with skip connections from the encoder to the decoder and a bottleneck as shown in figure 2. We use learned

(a.) MNIST Handwritten Digits, colorized

(b.) Places365 Athletic Fields category

Fig. 1: The first 64 samples from both datasets.

affine instance normalization on the skip connections. The encoder network detects features while the instance normalized skip connections use the learned *style* to define the colorization of such features in the decoder network. The UNet architecture enables feature detection across scales as the encoder downsamples the input to convolve on a coarser scale while the decoder receives these features and deconstructs them into meaningful colorizations at multiple scales. This architecture fits our problem well since the spatial locations of features in the encoder and decoder are shared.

When training the model we convert each image to grayscale and input it into the network as shown in figure 2. The *MNIST* grayscale images in training are just the originally imported grayscale images. For the sake of consistency with similar literature we call the UNet "the generator $G(y)$" even though the model is fully deterministic according to some input unlike conventional generative networks – that is our model does not depend on a random latent vector z and as such we write $G(y)$ instead of $G(z|y)$ as in [13] for conditional generative networks. For all adversarial networks we use an untrained ResNet [14] model as the discriminator matching the generator in number of parameters.

The image before conversion is then considered the

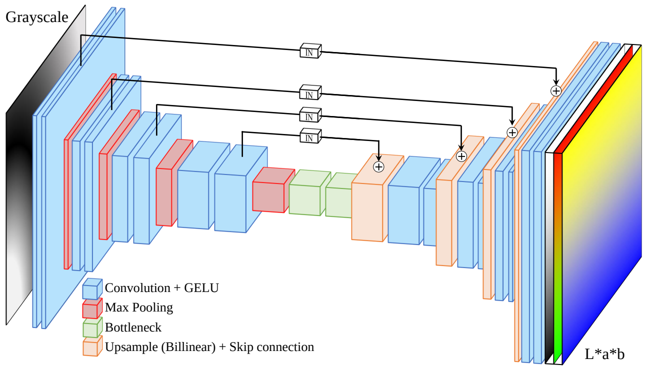

Fig. 2: UNet with instance normalization. The UNet model has 2 channels (*a,*b) in its output and the input grayscale image is concatenated as the L channel.

ground truth on which we measure the loss for training.

3.3. Reconstruction- and adversarial loss

We introduce a smoother transition for the generator by interpolating between real and generated data when evaluated by the discriminator which we call instance smoothing for image conditional GANs. For each image in a batch we sample $\alpha_s \sim \mathcal{U}[0, \beta_s]$ decreasing β_s from 0.5 to 0 over a number of iterations. We define two interpolates between real image x and a generated image $G(y)$:

$$\begin{aligned} \hat{x}_F &= \alpha_s x + (1 - \alpha_s) G(y) \quad (1) \\ \hat{x}_R &= (1 - \alpha_s) x + \alpha_s G(y) \quad (2) \\ &\text{where } x \sim p_{\text{data}}(x) \end{aligned}$$

where y is the grayscale transformation of x when training. The loss function then becomes:

$$V(D, G) = \mathbb{E}[f(D(\hat{x}_F|y)) + f(-D(\hat{x}_R|y))]$$

where

$$\begin{aligned} &\lim_{\beta_s \to 0} \mathbb{E}[f(D(\hat{x}_F|y)) + f(-D(\hat{x}_R|y))] \\ &= \mathbb{E}[f(D(G(y)|y)) + f(-D(x|y))] \end{aligned}$$

since β_s is the upper bound in the uniform distribution for α_s. This yields the original conditional GAN minimax game [13] as β_s decreases to zero. This smooth transition in training is motivated by the improvements seen in label smoothing [5] which smoothes the conditional labels of the generator where our method creates a smooth boundary between real and generated data increasing support between the two distributions. For our problem this equates to adding the correct color to the generated image to some degree. During training this addition is then tuned down at which point the generator has hopefully learned to use the correct coloring.

To stabilize training even further we implement instance noise which is defined by adding some noise signal with scale $\sigma^2_{\text{critical}} = |f'(0)|/|f''(0)|$ to the input in the discriminator. This further increases support between the real and generated distributions as described in [4].

The gradient penalty for Wasserstein GANs [2] greatly complements our method as the gradient penalty is added to the loss function along interpolated lines between real and generated data:

$$L_{\nabla \hat{x}_F} = \lambda_{\text{recon}} \mathbb{E}_{\hat{x}_F}[(\|\nabla_{\hat{x}_F} D(\hat{x}_F|y)\|_2 - 1)^2]$$

where $\hat{x}_F$ is the interpolate corresponding to the notation as defined in eq.(1). This is a soft enforcement of the 1-Lipschitz constraint required by the WGAN. This term is similar to our method of instance smoothing as they use the same interpolates. This penalty enforces the gradient to have a maximum norm of one along the interpolated lines. In our case where $\alpha_s \sim \mathcal{U}[0, \beta_s]$ for decreasing β_s the aforementioned gradient penalty tends to penalize only gradients on generated data and as such we introduce another gradient penalty operating only on real data called R_1 regularization given by:

$$R_{1\nabla x} = \frac{\gamma_{R1}}{2} E_{x \sim p_{\text{data}(x)}}[\|\nabla D(x)\|_2^2]$$

as described in [3] which also is a soft enforcement of the 1-Lipschitz constraint. Using instance noise [4] together with $R_{1\nabla x}$ regularization is further supported by [15] in which they build upon [2] by introducing exactly this form of the gradient penalty to ensure convergence towards a Nash equilibrium.

As such the final loss function (or minimax game) becomes:

$$\max_G \min_D V(D, G) = \mathbb{E}[f(D(\hat{x}_F|y)) + f(-D(\hat{x}_R|y))] + L_{\nabla \hat{x}_F} + R_{1\nabla x}$$

with $\hat{x}_R$ and $\hat{x}_F$ as defined in equation (1) and (2).

We add this adverserial loss to a standard reconstruction loss. In this report we consider Huber (Smooth L1) and Mean Squared Error (L2/MSE) loss. The reconstruction MSE loss is given by:

$$\text{MSE}(x_F, x_R) = \frac{1}{B} \sum_{n=1}^{B} (x_R - x_F)^2$$

with B being the batch size. Huber is quadratic like MSE around the origin but linear elsewhere. Huber loss has been shown to improve image saturation [16] and decrease blurring [7].

4. RESULTS

We maintained the following parameters throughout the conducted experiments and list different configurations for the two datasets in table 1. The batch size was set to 32, and a learning rate of $5 \cdot 10^{-5}$ for training on the *MNIST* data set, and 10^{-4} on the *Athletic Fields* data set. For upsampling layers we used *Billinear* and for downsampling layers we used *Maxpool*, this applies to all of our models and on both datasets. We set the $f(\cdot)$ in the loss function to the softplus function $f(x) = \log(1 + e^x)$.
Using the category *Athletic Fields* from the places data set, we randomly crop a 128x128 section of the original 256x256 image each time the image is loaded into a batch for training. For the *MNIST* data we used two decoder and encoder blocks. With the *Athletic Fields* data we used seven encoder and decoder blocks while for configuration M we only used two as in *MNIST*. Configurations K and S were trained without adversarial loss. Configuration P used dropout of 50% applied to the three first decoder layers as described in [7] which makes our model architecture only differ from *pix2pix* by applying instance normalization on the skip connections. For *MNIST* and *Athletic Fields* we used *ResNet18* and *ResNet152* respectively as the discriminator. All networks were normally initialized with $\mu = 0$, $\sigma = 0.02$ as described in the DCGAN paper [17]. We calculated the FIDs between 10,000 real and 10,000 generated images to obtain statistically sound results using *pytorch-fid* [18]. All models were trained on a Nvidia Tesla V100 SXM2 32 GB courtesy of DTU Compute.

Configuration	GP	Recon. type	λ_{recon}	γ_{R1}	InS.†	InN.‡	FID
A MINST	✓	MSE	1			125	15.784
B MINST	✓	MSE	1			1000	18.010
C MNIST	✓	MSE	1		✓	1000	11.790
D MNIST	✓	MSE	100		✓	500	11.001
E MNIST	✓	Huber	100		✓	500	**<u>10.559</u>**
F MNIST	✓	Huber	1000		✓	500	10.635
G MNIST	✓	Huber	100	10	✓	500	10.573
H MNIST		Huber	100	10	✓	500	**<u>10.228</u>**
I MNIST		Huber	100	1	✓	500	10.550
J MNIST		Huber	100	10		500	10.301
K MNIST		Huber	100				17.613
L Ath. Fields		Huber	100	10	✓	500	25.579
M Ath. Fields - shallow		Huber	100	10	✓	500	38.454
N Ath. Fields		MSE	100	10	✓	500	25.788
O Ath. Fields		Huber	100	10		500	26.142
P Ath. Fields	✓	Huber	100		✓	500	27.127
Q Ath. Fields	✓	Huber	100	10	✓	500	**<u>25.283</u>**
R Ath. Fields - pix2pix		Huber	100	10	✓	500	26.252
S Ath. Fields - only recon.		Huber					25.296
X Ath. Fields - 4000 iter.	✓	Huber	100	10	✓	5000	16.804

Table 1: Hyperparameters and resulting FID scores for a given model trained over 1000 iterations. The models were trained on the same samples, and evaluated on the exact same images. Config. K was trained without the adversarial loss. All *Athletic Fields* models were trained using *MNIST* configuration H as a baseline. Config. M is a shallow architecture with only two blocks in the encoder and decoder. Blank field means configuration without this method. †: Instance Smoothing enabled. ‡: Linearly decrease instance noise to zero over this number of iterations.

Note that all models with adversarial loss in table 1 use instance noise. When training the adversarial networks without instance noise the models often crashed in an early iteration.

5. DISCUSSION

5.1. MNIST results

The *MNIST* dataset contains both neat- and crudely written digits. During the learning phase the model has to both detect features and colorize them appropriately. While the overwhelming majority of generated *MNIST* images are very close to the ground truth, poorly executed digits without common features are unevenly colored such as col. 10, 16 and 18 in figure 3 as the model is unsure about the exact digit-defining features. The most troublesome digits are those that are similarly indecipherable for a human, e.g. col. 19 and 20. Tricky digits such as col. 2, 10, 13 and 14 show configuration A's tendency to produce more varied results although being closer to the ground truth in col. 18.

We ascertain that the model learns to colorize only certain digit-defining features individually and struggles to make a unanimous colorization on the digit as a whole. As such some digits are unevenly colored as evident by col. 17 in figure 3 where the upper arc matches a three or a two and the lower part is only present in sevens.

Overall, configurations E, G, H, J produce visually superior results which correlates well with their FIDs. Due to the

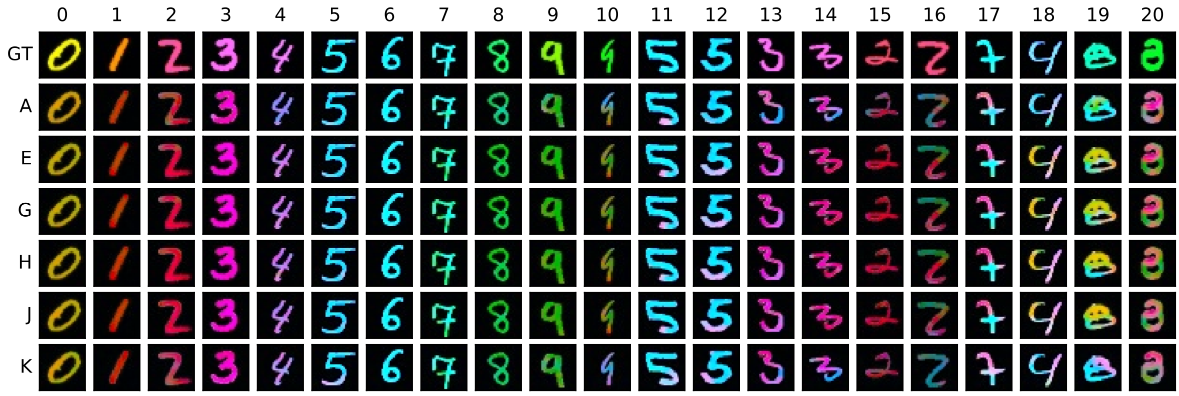

Fig. 3: Random digits from 0 to 9 and some handpicked digits for the given model configurations.

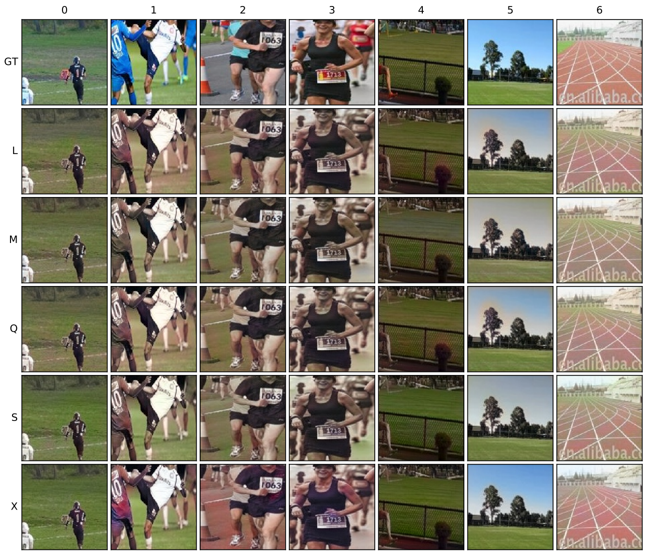

Fig. 4: Random samples from the *Athletic Fields* set and the generated image for the given model configurations on the left. Model L follows the best configuration from MNIST, M is a shallow version of L, model Q is the best performing model on *Athletic fields*, model S is trained without adversarial loss, model X is equivalent to configuration Q but trained over 4000 iterations

visual similarities between configurations K (without adversarial loss) and A (baseline) in figure 3 an unoptimized model is highly dependent on fine tuning before being able to produce the expected improvements from adversarial loss. This is further corroborated by configuration B from table 1 which shows that a wrongly tuned model underperforms compared to the baseline configuration A.

5.2. Regularization

Assessing table 1 adding instance smoothing and noise decay generally results in a significant FID decrease. This confirms the positive effect of instance smoothing as the generator learns to utilize more varying color choices matching real images and thus challenging the discriminator. Instance smoothing appears to significantly decrease FID as evident in configurations C and L compared to B and O respectively. However, comparing L and P R_1 regularization decreases FID similarly with instance smoothing. This implies that as long as either instance smoothing or R_1 regularization is used the performance and stability should improve significantly.

As evident in table 1 we achieve the best results implementing both gradient penalties on the *Athletic Fields* dataset. Both gradient penalties are soft enforcements of the 1-Lipschitz constraint. However, they differ in which data distribution to enforce the gradient penalty upon. By using both we hope to more consistently constrain the gradients of the discriminator and stabilize training. An argument can be made that the penalties do not give high enough payback from the added computational complexity. In the worst case they might also counteract each other close to equilibrium.

5.3. Adding depth

When ramping up to the *Athletic Fields* dataset the fully convolved UNet clearly yields the best results. This is evident when comparing the shallow model M, which uses the same amount of decoder and encoder blocks as the MNIST models, to its fully featured counterparts (e.g. model L) which otherwise share the exact same parameters. This is most likely due to the fact that we detect coarser features when increasing network depth which is allowed by the increased image resolution. Figure 4 acts as a direct visual representation of this phenomenon. Directly comparing models L (baseline) and M (shallow) the deeper network show more saturated colors particularly in large patches such as the sky or ground. Coloring is also more well defined on items such as clothing and skin. The shallow network in comparison is greatly challenged by the trees breaking up the sky, indicating a poorer ability to separate image features, as seen in col. 5. The disadvantage of increased computation time and model complexity is greatly outweighed by the superior, less bland and detailed results.

In this regard col. 6 in figure 4 works as a good perceptual measurement of the performance of the model. This image has complicated features and a deceiving perspective but still being easily conceivable by the human eye – which boils down to the model's ability to color inside the lines of the running track which model X and Q clearly does the best. Model X also performs well on arbitrarily colored objects and makes a believable colorization of the jersey in col. 1 even though the ground truth jersey is a solid blue.

5.4. Evaluation and FID

When evaluating the results of colorization some image features are more context sensitive than others. Our objective is to colorize an image in such a way that it does not remove context from the image. Objects which might have arbitrary colors *independent* of their context (e.g. T-shirts, cars) should be given a color that makes sense for the object itself – while objects that have colors that are *dependent* on context (e.g. grass, asphalt, person, sky) should be colored in a way that makes sense in a given context. Higher precision on context-dependent features with solid coloring is preferred, which are often features taking up big proportions of the image thereby increasing significance for human evaluation. Accurate colorization of smaller objects is less significant than for larger more persistent objects.

Making direct pixel-wise measurements on the performance of the model fully ignores context and can easily lead to problematic results. To this extent we utilized the FID metric which is consistent with human judgment as explored in [11] which found that human evaluation is often biased. This metric not only considers the correct color value but also takes into account the individual features of the image. The FID evaluation is more efficient than human judgement and can evaluate more images giving a better estimate of how well the model generalizes.

All models but X trained over 1000 iterations with a batch size of 32 which is 32,000 images out of the 40,000 images in the dataset meaning the models did not run for even a single epoch. No adversarial networks were plateauing or converging yet. This implies that there is room for improvement on all models without overfitting, which is also evident by model X. However, due to time limits it was not feasible to train all the models for any longer.

5.5. Datasets

As seen in figure 4 model X col. 2 our model predicts the asphalt as being a red running track. We suppose this is due to the detection of a runner, which in most cases in *Athletic Fields* are on a red surfaces, creating the impression that all runners are on red running tracks. In addition, asphalt is also underrepresented in the dataset which indicates that the trained model will not generalize well outside of *Athletic Fields* settings. By introducing greater variation in the dataset the model would have to train more to achieve the same degree of specificity and still be able to generalize.

The chosen datasets (*MNIST* and *Athlethic fields*) were selected because of the difference in complexity. Further research of the scalability of the model on even more complex datasets could be explored. This should be centered around configuration H and Q which performed the best on *MNIST* and *Athletic Fields* respectively. We conclude that the performance of using gradient penalty is dependent on the complexity of the dataset.

5.6. Generative model

In the development phase we tried implementing a way to incorporate Gaussian noise in the model to make it truly generative. However, the output ended up being highly deterministic and the generator learned to just ignore the noise – which is consistent with [7] and [19] for image conditional GANs. As a consequence we ended up removing all stochastic input for all models except model R which we trained according to the *pix2pix* model [7] using dropout of $p = 0.5$ on the first half of the decoder blocks. This model underperformed its predecessor model L which means the stochastic element of dropout only had a negative effect on the model.

For further research we propose using a mapping network from a latent vector z to an intermediate latent space $\mathcal{W}$ which controls the generator by supplying the adaptive instance normalization layers in the decoder with styles as used in StyleGAN [20]. In their paper the affine parameters of the instance normalization layers are stochastically mapped (adaptive) instead of being a learned constant as in our model. Learned constant affine parameters essentially lock the colorization to certain styles.

Using this proposed approach would make the stylization stochastic and generative compared to our constant learned stylization parameters – this would in turn allow the model to predict the color of items which can be arbitrarily colorized since the style input of the instance normalization layer is now also stochastic in nature. However, this would conflict with using a reconstruction loss such as MSE or Huber as those would penalize the model for choosing an arbitrary color that does not match a given image even though it could have been colored a different way without removing context from the image. We propose using a reconstruction loss that is independent of the hue meaning only the correct color saturation is considered although this might lead to problems with the model predicting that grass is blue which leaves it up to the discriminator to discriminate the difference.

5.7. Adversarial and reconstruction loss

In early development we found that the model usually converged to a brownish, desaturated solid color for the whole image when training a simple model using only MSE loss – which is consistent with [6], which is why we consider adversarial networks at all. In figure 4 the results for model X are somewhat saturated but still brownish in color in col. 2 and 3. However, it generally colorizes grass and skies with the correct saturation according to ground truth.

Model S without adversarial loss performed marginally worse than the best adversarial model Q according to FID metric in table 1. This could mean that the reconstruction loss has not yet plateaued on the more complex dataset *Athletic Fields*. We expect the reconstruction loss to plateau at some point while the adversarial loss would have more potential to improve the model. This is further corroborated by the results on the simpler dataset *MNIST* where the adversarial networks clearly outperformed the non-adversarial one. Since the models were still not converging after about one epoch, it is hard to predict if or when this will happen since the reconstruction loss would still be present when it plateaus. The adversarial loss would have to outweigh the reconstruction loss in order for the generator to learn.

5.8. Ethics

Since our objective is not to be exact, colorizing greyscale images such as historically significant photos could lead to loss of important specific details. Failing to colorize with historical accuracy increases risk of wrongful revisionism. On the other hand it could also contribute to an increase in public interest and immersion into historical context.

6. LEARNING OUTCOME

We have learned about the agility of convolutional neural networks and how much customization you can do to them in order to tailor them to you specific needs. We also found out that there are an overwhelming amount of possible parameters to tweak, from the structure itself to the hyperparameters. We have gained a much more robust understanding of the inner workings of convolution layers and how they detect image features - as well as the interesting implications of the skip-connection driven UNet architecture for image recognition tasks. Furthermore, we exposed ourselves to the vast potential of GANs and the adversarial interplay between generators and discriminators as a more advanced means of implementing network loss - for overcoming the shortcomings of MSE. We gathered experience on how images are represented as a 3D datastructure, and how they can be manipulated in python. We also got acquainted with how images can be defined in a colorspaces.

7. REFERENCES

[1] Zheng Xu, Xitong Yang, Xue Li, and Xiaoshuai Sun, "The effectiveness of instance normalization: a strong baseline for single image dehazing," *CoRR*, vol. abs/1805.03305, 2018.

[2] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville, "Improved training of wasserstein gans," *CoRR*, vol. abs/1704.00028, 2017.

[3] Lars Mescheder, Sebastian Nowozin, and Andreas Geiger, "Which training methods for gans do actually converge?," in *International Conference on Machine Learning (ICML)*, 2018.

[4] Casper Kaae Sønderby, Jose Caballero, Lucas Theis, Wenzhe Shi, and Ferenc Huszár, "Amortised MAP inference for image super-resolution," *CoRR*, vol. abs/1610.04490, 2016.

[5] Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen, "Improved techniques for training gans," *CoRR*, vol. abs/1606.03498, 2016.

[6] Richard Zhang, Phillip Isola, and Alexei A. Efros, "Colorful image colorization," *CoRR*, vol. abs/1603.08511, 2016.

[7] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros, "Image-to-image translation with conditional adversarial networks," *CoRR*, vol. abs/1611.07004, 2016.

[8] Yann LeCun and Corinna Cortes, "MNIST handwritten digit database," 2010.

[9] Itseez, "Open source computer vision library," `https://github.com/itseez/opencv`, 2015.

[10] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba, "Places: A 10 million image database for scene recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[11] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Günter Klambauer, and Sepp Hochreiter, "Gans trained by a two time-scale update rule converge to a nash equilibrium," *CoRR*, vol. abs/1706.08500, 2017.

[12] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna, "Rethinking the inception architecture for computer vision," 2015.

[13] Mehdi Mirza and Simon Osindero, "Conditional generative adversarial nets," *CoRR*, vol. abs/1411.1784, 2014.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," 2015.

[15] Naveen Kodali, Jacob D. Abernethy, James Hays, and Zsolt Kira, "How to train your DRAGAN," *CoRR*, vol. abs/1705.07215, 2017.

[16] Yi Xiao, Peiyao Zhou, and Yan Zheng, "Interactive deep colorization with simultaneous global and local inputs," *CoRR*, vol. abs/1801.09083, 2018.

[17] Alec Radford, Luke Metz, and Soumith Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2016.

[18] Maximilian Seitzer, "pytorch-fid: FID Score for PyTorch," `https://github.com/mseitzer/pytorch-fid`, August 2020, Version 0.1.1.

[19] Michael Mathieu, Camille Couprie, and Yann LeCun, "Deep multi-scale video prediction beyond mean square error," 2016.

[20] Tero Karras, Samuli Laine, and Timo Aila, "A style-based generator architecture for generative adversarial networks," *CoRR*, vol. abs/1812.04948, 2018.

Code: Training on MNIST and Athletic Fields

Also found on: `https://github.com/andersxa/ColorizingGrayscaleImages/blob/main/Colorizing_Grayscale_Images_Colored.ipynb`

```
from IPython.display import display, clear_output
from matplotlib import pyplot as plt
#plt.style.use('dark_background')
import numpy as np
import torch
import cv2
from tqdm import tqdm
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import torchvision.models as models
from glob import glob
import os
#from PIL import Image
#from skimage import color
torch.manual_seed(0)
np.random.seed(0)

# %%
#@title Parameters {display-mode: "form"}
#@markdown ---
batch_size = 64  #@param {type: "number"}
disc_iters = 5  #@param {type: "number"}
gen_iters = 1  #@param {type: "number"}
plot_iter = 50  #@param {type: "number"}
#@markdown ---
D_lr = 5e-5  #@param {type: "number"} #1e-4 er optimalt
G_lr = 5e-5  #@param {type: "number"} #1e-4 er optimalt

beta_1 = 0.9  #@param {type: "number"}
beta_2 = 0.999  #@param {type: "number"}
G_betas = D_betas = (beta_1, beta_2)  #ONLY FOR ADAM
#@markdown ---
training_opt = "GP"  #@param ["None","R1_reg", "GP", "R1_GP"]
R1_reg = 0.0  #@param {type: "slider", min: 0.0, max: 100.0, step: 0.1}
instance_noise_iter = 1000  #@param {type: "slider", min: 0.0, max: 20000, step: 100}
instance_smoothing = False  #@param {type: "boolean"}
#@markdown ---
#Training options for reconstruction loss
recon_type = 'mse'  #@param ["None", "huber", "mse"]
recon_lambda = 1.0  #@param [0.1,0.2,0.2,0.4,0.5,0.6,0.7,0.8,0.9,1.0,1.1,1.2,1.5,2.0,5.0,10.0,25.0,50.0,100.0,200.0] {ty
#@markdown ---
downsample_type = "MaxPool"  #@param ["Conv", "MaxPool"]

model_name = "MNIST-1mse-no-smooth-1000ins-noise"

# %%
hyper_params = {}
hyper_params['batch_size'] = batch_size
hyper_params['disc_iters'] = disc_iters
hyper_params['gen_iters'] = gen_iters
hyper_params['plot_iter'] = plot_iter
hyper_params['D_lr'] = D_lr
hyper_params['G_lr'] = G_lr
hyper_params['beta_1'] = beta_1
hyper_params['beta_2'] = beta_2
hyper_params['training_opt'] = training_opt
hyper_params['R1_reg'] = R1_reg
hyper_params['instance_noise_iter'] = instance_noise_iter
hyper_params['instance_smoothing'] = instance_smoothing
if recon_type in ['mse', 'huber']:
  hyper_params['recon_type'] = recon_type
  hyper_params['recon_lambda'] = recon_lambda
hyper_params['downsample_type'] = downsample_type

# %%
# For athletic field
# min_size = 256
# crop_size = 128
# if not os.path.exists(f'./{min_size}_sized_images.txt'):
#   with open(f'./{min_size}_sized_images.txt', 'w+') as f:
#     for p in tqdm(glob('data_256/a/athletic_field/outdoor/*.jpg')):
#       print(p, file=f)

# import multiprocessing
# resize_crop_transform = transforms.Compose([transforms.Resize(min_size),transforms.RandomCrop(crop_size)])
# class CustomDataset(torch.utils.data.Dataset):
#     def __init__(self, transform):
#       with open(f'./{min_size}_sized_images.txt', 'r') as f:
#         self.image_paths = f.read().splitlines()
#       self.transform = transform

#     def __len__(self):
#       return len(self.image_paths)

#     def __getitem__(self, idx):
```

```
#       im = cv2.imread(self.image_paths[idx])
#       #im_RGB = torch.tensor(cv2.cvtColor(im, cv2.COLOR_BGR2RGB) / 255.0, dtype=torch.float).permute(2,0,1)
#       im_Lab = torch.tensor(cv2.cvtColor(im, cv2.COLOR_BGR2Lab) / 255.0, dtype=torch.float).permute(2,0,1)
#       #im = self.transform(torch.cat([im_Lab, im_RGB],0))
#       im = self.transform(im_Lab)
#       return im

# dataset = CustomDataset(transform=resize_crop_transform)
# data_loader = torch.utils.data.DataLoader(dataset,
# batch_size=batch_size, shuffle=True, num_workers=multiprocessing.cpu_count(), pin_memory=True)
# print(len(dataset))

# %%
import multiprocessing


class ColorMNIST(torch.utils.data.Dataset):

  def __init__(self, dataset):
    self.data = dataset.data
    self.targets = dataset.targets

  def __len__(self):
    return len(self.data)

  def __getitem__(self, idx):
    pi = 3.1415927410125732
    image, label = self.data[idx], self.targets[idx]
    image = image / 255.0
    label = float(label) + torch.empty(1).uniform_(-0.5, 0.5)
    a_const = torch.sin(2 * pi * label / 10.0)
    b_const = torch.cos(2 * pi * label / 10.0)
    sample = torch.stack([
        image,
        0.5 * a_const / max(abs(a_const), abs(b_const)) * image + 0.5,
        0.5 * b_const / max(abs(a_const), abs(b_const)) * image + 0.5
    ])
    return sample


dataset = ColorMNIST(
    torchvision.datasets.MNIST('data', train=True, download=True))
data_loader = torch.utils.data.DataLoader(dataset,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          num_workers=0,
                                          pin_memory=True)
print(len(dataset))


# %%
def get_samples(batch_size):
  idxs = np.random.randint(len(dataset), size=(batch_size,))
  return idxs, torch.stack([dataset[i] for i in idxs], 0)


# %%
def Lab2RGB(im):
  if im.shape[0] == 3:
    im = im.permute(1, 2, 0)
  im = 255.0 * im
  im = im.to(torch.uint8).numpy()
  return cv2.cvtColor(im, cv2.COLOR_Lab2RGB)


# %%
plt.figure(figsize=(4.25, 4.25), dpi=180)
plt.imshow(Lab2RGB(
    torchvision.utils.make_grid([dataset[i] for i in range(8 * 8)],
                                8).cpu()),
           interpolation='nearest')
plt.axis('off')


# %%
def weights_init(m):
  classname = m.__class__.__name__
  if classname.find('Conv2d') != -1:
    nn.init.normal_(m.weight.data, 0.0, 0.02)


class ConvBlock(torch.nn.Module):

  def __init__(self,
               in_channels,
               out_channels,
               n_intermediate_layers=1,
               direction='in'):
    super(ConvBlock, self).__init__()
    if direction == 'out':
      self.BlockLayers = nn.Sequential(
          nn.Conv2d(in_channels,
                    in_channels,
```

```
                    kernel_size=3,
                    stride=1,
                    padding=1), nn.GELU(), *[
                        l for _ in range(n_intermediate_layers)
                        for l in [
                            nn.Conv2d(in_channels,
                                      out_channels,
                                      kernel_size=3,
                                      stride=1,
                                      padding=1),
                            nn.GELU()
                        ]
                    ])
    else:
      self.BlockLayers = nn.Sequential(
          nn.Conv2d(in_channels,
                    out_channels,
                    kernel_size=3,
                    stride=1,
                    padding=1), nn.GELU(), *[
                        l for _ in range(n_intermediate_layers)
                        for l in [
                            nn.Conv2d(out_channels,
                                      out_channels,
                                      kernel_size=3,
                                      stride=1,
                                      padding=1),
                            nn.GELU()
                        ]
                    ])

  def forward(self, x):
    x = self.BlockLayers(x)
    return x


class EncoderBlock(torch.nn.Module):

  def __init__(self,
               in_channels,
               out_channels,
               n_intermediate_layers=1):
    super(EncoderBlock, self).__init__()
    self.EncoderBlock = ConvBlock(in_channels,
                                  out_channels,
                                  n_intermediate_layers,
                                  direction='in')
    self.DownSample = nn.MaxPool2d(
        2) if downsample_type == 'MaxPool' else nn.Sequential(
            nn.Conv2d(out_channels,
                      out_channels,
                      kernel_size=3,
                      stride=2,
                      padding=1), nn.GELU())

  def forward(self, x):
    x = self.EncoderBlock(x)
    skip = x
    x = self.DownSample(x)
    return x, skip


class DecoderBlock(torch.nn.Module):

  def __init__(self,
               in_channels,
               out_channels,
               n_intermediate_layers=1):
    super(DecoderBlock, self).__init__()
    if upsample_type == 'PixelShuffle':
      self.UpSample = PixelShuffleUpsample(in_channels, 2)  #
    else:
      self.UpSample = nn.Upsample(scale_factor=2,
                                  mode='bilinear',
                                  align_corners=False)
    self.DecoderBlock = ConvBlock(in_channels,
                                  out_channels,
                                  n_intermediate_layers,
                                  direction='out')

  def forward(self, x, skip):
    x = self.UpSample(x)
    x = self.DecoderBlock(x + skip)
    return x


# %%
#Unet Basic
class GeneratorUNet(torch.nn.Module):

  def __init__(self):
    super(GeneratorUNet, self).__init__()
    #Encoder Mapping
```

```
    self.enc_block1 = EncoderBlock(1, 256, 1)
    self.enc_block2 = EncoderBlock(256, 256, 1)

    #Skip connection Instance Norms
    self.skip_IN1 = nn.InstanceNorm2d(256, affine=True)
    self.skip_IN2 = nn.InstanceNorm2d(256, affine=True)

    #Bottleneck
    self.bot_block1 = ConvBlock(256, 512, 1, direction='in')
    self.bot_block2 = ConvBlock(512, 256, 1, direction='out')

    #Decoder
    self.dec_block1 = DecoderBlock(256, 256, 1)
    self.dec_block2 = DecoderBlock(256, 256, 1)

    #Out
    self.out_conv = nn.Conv2d(256,
                              2,
                              kernel_size=1,
                              stride=1,
                              padding=0,
                              bias=True)

  def forward(self, gray):
    x = gray
    #x = torch.cat([x, torch.randn(x.size(0), 2, x.size(2), x.size(3), device=x.device)], 1)
    x, skip1 = self.enc_block1(x)
    x, skip2 = self.enc_block2(x)

    #Bottleneck
    x = self.bot_block1(x)

    #Skip AdaINs
    skip1 = self.skip_IN1(skip1)
    skip2 = self.skip_IN2(skip2)

    x = self.bot_block2(x)

    #Decoder
    x = self.dec_block1(x, skip2)
    x = self.dec_block2(x, skip1)

    #Out
    out = torch.sigmoid(self.out_conv(x))
    out = torch.cat([gray, out], 1)
    return out


GNet = GeneratorUNet().to(device)
GNet.apply(weights_init)
print(GNet)
print("Generator trainable parameters:",
      sum(p.numel() for p in GNet.parameters() if p.requires_grad))


# %%
#Unet Basic
class DiscriminatorNet(torch.nn.Module):

  def __init__(self):
    super(DiscriminatorNet, self).__init__()
    #resnet = models.resnet34()
    #self.resnet = nn.Sequential(resnet.conv1, resnet.bn1, resnet.relu, resnet.maxpool,
    # resnet.layer1, resnet.layer2, resnet.layer3, resnet.layer4, resnet.avgpool)

    # #Encoder
    self.enc_conv1 = nn.Conv2d(3,
                               128,
                               kernel_size=3,
                               stride=1,
                               padding=1)
    self.enc_conv2 = nn.Conv2d(128,
                               128,
                               kernel_size=3,
                               stride=1,
                               padding=1)
    self.enc_down1 = nn.Conv2d(128,
                               256,
                               kernel_size=3,
                               stride=2,
                               padding=1)

    self.enc_conv3 = nn.Conv2d(256,
                               256,
                               kernel_size=3,
                               stride=1,
                               padding=1)
    self.enc_conv4 = nn.Conv2d(256,
                               256,
                               kernel_size=3,
                               stride=1,
                               padding=1)
    self.enc_down2 = nn.Conv2d(256,
```

```
                                512,
                                kernel_size=3,
                                stride=2,
                                padding=1)

    self.enc_conv5 = nn.Conv2d(512,
                                512,
                                kernel_size=3,
                                stride=1,
                                padding=1)
    self.enc_conv6 = nn.Conv2d(512,
                                512,
                                kernel_size=3,
                                stride=1,
                                padding=1)

    #Out
    self.out_conv = nn.Conv2d(512,
                               1024,
                               kernel_size=1,
                               stride=1,
                               padding=0)

  def forward(self, x):
    x = res = F.gelu(self.enc_conv1(x))  #
    x = res + F.gelu(self.enc_conv2(x))  #
    x = res = F.gelu(self.enc_down1(x))

    x = res = res + F.gelu(self.enc_conv3(x))  #
    x = res = res + F.gelu(self.enc_conv4(x))  #
    x = res = F.gelu(self.enc_down2(x))

    x = res = res + F.gelu(self.enc_conv5(x))  #
    x = res + F.gelu(self.enc_conv6(x))  #

    x = F.gelu(self.out_conv(x))
    return x


DNet = DiscriminatorNet().to(device)
DNet.apply(weights_init)
print(DNet)
print("Discriminator trainable parameters:",
      sum(p.numel() for p in DNet.parameters() if p.requires_grad))


# %%
def Lab2RGB(im):
  if im.shape[0] == 3:
    im = im.permute(1, 2, 0)
  im = 255.0 * im
  im = im.to(torch.uint8).numpy()
  return cv2.cvtColor(im, cv2.COLOR_Lab2RGB)


def smooth(scalars, weight):
  last = scalars[0]
  smoothed = list()
  for point in scalars:
    smoothed_val = last * weight + (1 - weight) * point
    smoothed.append(smoothed_val)
    last = smoothed_val
  return smoothed


def anneal(val, target):
  return max((target - val) / target, 0)


# %%
from torch.cuda.amp.autocast_mode import autocast
#from torch.cuda.amp import GradScaler

from collections import defaultdict

G_optimizer = torch.optim.Adam(GNet.parameters(),
                                lr=G_lr,
                                betas=G_betas)
D_optimizer = torch.optim.Adam(DNet.parameters(),
                                lr=D_lr,
                                betas=D_betas)

losss_print = defaultdict(lambda: [])
print_to_legend = {
    'Div.': r'Div.',
    'G.i+1': r'$D(G_{i+1}(z|y))$',
    'D.': r'$D(x)$',
    'G.i': r'$D(G_{i}(z|y))$'
}

mse_loss_fn = nn.MSELoss()
huber_loss_fn = nn.SmoothL1Loss(beta=0.5)
iteration = -1
```

```
#Linear annealing

# %%

with tqdm(range(1000),
          unit_scale=(gen_iters + disc_iters) * batch_size,
          unit='img',
          ncols=200) as tqdm_bar:  #, autocast():#
  for iteration in tqdm_bar:
    GNet.train()
    DNet.train()
    #https://arxiv.org/pdf/1610.04490.pdf & https://arxiv.org/pdf/1701.04862.pdf
    instance_noise_annealing = anneal(iteration, instance_noise_iter)
    DNet.zero_grad()
    #---start disc loop---
    disc_real_avg = 0.0
    disc_fake_avg = 0.0
    for _ in range(disc_iters):
      idxs, real = get_samples(batch_size)
      real = real.to(device).requires_grad_(True)
      gray = real[:, :1]

      fake = GNet(gray)

      #Training Discriminator

      if instance_smoothing:
        alpha = 0.5 * instance_noise_annealing * torch.rand(
            real.size(0), 1, 1, 1, device=real.device).expand_as(real)
        real_input = (1 - alpha) * real + alpha * fake.detach()
      else:
        real_input = real

      real_noise = (instance_noise_annealing *
                    np.sqrt(2)) * torch.randn_like(real,
                                                   device=device)

      disc_real = DNet(real_input + real_noise)

      Dreal_loss = F.softplus(-disc_real).mean()

      if instance_smoothing:
        alpha = 0.5 * instance_noise_annealing * torch.rand(
            real.size(0), 1, 1, 1, device=real.device).expand_as(real)
        fake_input = alpha * real + (1 - alpha) * fake.detach()
      else:
        fake_input = fake.detach()

      fake_noise = (instance_noise_annealing *
                    np.sqrt(2)) * torch.randn_like(real,
                                                   device=device)

      disc_fake = DNet(fake_input + fake_noise)

      Dfake_loss = F.softplus(disc_fake).mean()

      #https://arxiv.org/pdf/1801.04406.pdf - R1 Reguilarization
      grad_penalty = 0.0
      if training_opt in ['R1_reg', 'R1_GP']:
        real_grad = torch.autograd.grad(outputs=disc_real.sum(),
                                        inputs=real,
                                        create_graph=True,
                                        only_inputs=True)[0]
        grad_penalty += (R1_reg / 2) * real_grad.view(
            real_grad.size(0), -1).square().sum(-1).mean()

      #https://arxiv.org/pdf/1704.00028.pdf afsnit 4 - gradient penalty
      if training_opt in ['GP', 'R1_GP']:
        alpha = torch.rand(real.size(0), 1, 1, 1,
                           device=real.device).expand_as(real)
        interp = alpha * real + (1 - alpha) * fake.detach()
        disc_interp = DNet(interp)
        interp_grad = torch.autograd.grad(outputs=disc_interp.sum(),
                                          inputs=interp,
                                          create_graph=True,
                                          only_inputs=True)[0]
        grad_penalty += 10.0 * F.relu(
            interp_grad.view(interp_grad.size(0), -1).norm(
                2, dim=1).subtract(1.0)).square().sum(-1).mean()

      D_loss = Dreal_loss + Dfake_loss + grad_penalty
      D_loss.backward()

      disc_real_avg += disc_real.mean().item()
      disc_fake_avg += disc_fake.mean().item()
    #---end disc loop---
    D_optimizer.step()
    losss_print['D.'].append(disc_real_avg / disc_iters)
    losss_print['G.i'].append(disc_fake_avg / disc_iters)

    #Training Generator
    GNet.zero_grad()
    #---Start generator loop---
```

```
    disc_gen_avg = 0.0
    for _ in range(gen_iters):
      idxs, real = get_samples(batch_size)
      real = real.to(device).requires_grad_(True)
      gray = real[:, :1]

      fake = GNet(gray)

      if instance_smoothing:
        alpha = 0.5 * instance_noise_annealing * torch.rand(
            real.size(0), 1, 1, 1, device=real.device).expand_as(real)
        gen_input = alpha * real.detach() + (1 - alpha) * fake
      else:
        gen_input = fake

      gen_noise = (instance_noise_annealing *
                   np.sqrt(2)) * torch.randn_like(real, device=device)

      disc_gen = DNet(gen_input + gen_noise)

      G_loss = F.softplus(-disc_gen).mean(
      )  #criterion(output, torch.full_like(output, 1.0, dtype=torch.float, device=device))

      recon_loss = 0.0
      if recon_type == 'mse':
        recon_loss = mse_loss_fn(fake[:, 1:], real[:, 1:])
      elif recon_type == 'huber':
        recon_loss = huber_loss_fn(fake[:, 1:], real[:, 1:])
      (G_loss + recon_lambda * recon_loss).backward()

      disc_gen_avg += disc_gen.mean().item()
    #---end generator loop---
    G_optimizer.step()
    losss_print['G.i+1'].append(disc_gen_avg / gen_iters)

    with torch.no_grad():
      tqdm_bar.set_postfix_str(
          f"Iteration {iteration}, I.N.A.: {instance_noise_annealing:.3f}, "
          + ", ".join([
              f"{ns}: {ls[-1]:.4f}" for ns, ls in losss_print.items()
          ]))
      if iteration % plot_iter == 0:
        clear_output(wait=True)
        #evaluate
        fig, axs = plt.subplots(4, 11, figsize=(22, 8), dpi=100)
        gs = axs[0, 5].get_gridspec()
        for axs_y in axs[:, 5:]:
          for ax in axs_y:
            ax.remove()
        fig.suptitle(f"{model_name}, Iteration {iteration}, " +
                     ", ".join([
                         f"{k}: {v}" +
                         ("\n" if (i + 1) % 7 == 0 else "")
                         for i, (k,
                                 v) in enumerate(hyper_params.items())
                     ]))
        axbig = fig.add_subplot(gs[:, 5:])
        axbig.tick_params(direction='in', pad=-22)
        legend = []
        for ns, ls in losss_print.items():
          axbig.plot(ls)
          legend.append(print_to_legend[ns])
        axbig.legend(legend)
        im_pred_ = fake[:, -3:].cpu().detach()
        im_gray_ = gray.cpu().detach()
        im_true_ = real[:, -3:].cpu().detach()
        for i in range(4):
          axs[i][0].imshow(
              Lab2RGB(im_pred_[i].permute(1, 2, 0).float() *
                      torch.tensor([0.0, 1.0, 1.0]).view(1, 1, 3) +
                      torch.tensor([0.5, 0.0, 0.0]).view(1, 1, 3)),
              interpolation='nearest',
              aspect='equal',
              vmin=0,
              vmax=1)
          axs[i][1].imshow(
              Lab2RGB(im_true_[i].permute(1, 2, 0).float() *
                      torch.tensor([0.0, 1.0, 1.0]).view(1, 1, 3) +
                      torch.tensor([0.5, 0.0, 0.0]).view(1, 1, 3)),
              interpolation='nearest',
              aspect='equal',
              vmin=0,
              vmax=1)
          axs[i][2].imshow(im_gray_[i].permute(1, 2,
                                               0).squeeze().float(),
                           cmap='gray',
                           interpolation='nearest',
                           aspect='equal',
                           vmin=0,
                           vmax=1)
          axs[i][3].imshow(Lab2RGB(im_pred_[i].permute(1, 2,
                                                       0).float()),
                           interpolation='nearest',
```

```
                            aspect='equal',
                            vmin=0,
                            vmax=1)
            axs[i][4].imshow(Lab2RGB(im_true_[i].permute(1, 2,
                                                        0).float()),
                            interpolation='nearest',
                            aspect='equal',
                            vmin=0,
                            vmax=1)
            axs[i][0].set_axis_off()
            axs[i][1].set_axis_off()
            axs[i][2].set_axis_off()
            axs[i][3].set_axis_off()
            axs[i][4].set_axis_off()
            axs[i][2].set_title(f"ID:{idxs[i]}")
        #fig.savefig(f"I_{iteration}_results.png", dpi=180)
        plt.show()
        #torch.save({'gnet':GNet.state_dict(), 'dnet':DNet.state_dict()}, 'save_athletic_field.h')

# %%
torch.save({
    'gnet': GNet.state_dict(),
    'dnet': DNet.state_dict()
}, f'{model_name}.h')

# %%
#s_dict = torch.load(f"{model_name}.h")

# %%
#s_dict = torch.load(f"{model_name}.h")
#GNet.load_state_dict(s_dict['gnet'])

# %%
#for i in range(20):
#    real_sample = get_samples(512)
#    for i_r, r in enumerate(real_sample):
#        im_rgb = Lab2RGB(r)
#        cv2.imwrite(f"MNIST_real/{i*512+i_r}.jpg", im_rgb)

# %%
get_ipython().run_cell_magic('bash', '-s "$model_name"', 'mkdir "$1"')

# %%
real_paths = list(glob("MNIST_real/*.jpg"))
with torch.no_grad():
  for i in range(20):
    im_paths = real_paths[i * 512:(i + 1) * 512]
    gray = torch.stack([
        torch.tensor(cv2.imread(f, cv2.IMREAD_GRAYSCALE),
                     device=device).unsqueeze(0) / 255.0
        for f in im_paths
    ])
    fake_sample = GNet(gray)
    for i_f, f in enumerate(fake_sample):
      im_rgb = Lab2RGB(f.cpu())
      cv2.imwrite(f"{model_name}/{os.path.basename(im_paths[i_f])}",
                  im_rgb)

# %%
get_ipython().run_cell_magic(
    'bash', '-s "$model_name"',
    'python3 -m pytorch_fid --device cuda:0 "MNIST_real" "$1"')

# %%
model_name
```