Deep Learning Practical Work 2-de
Generative Adversarial Networks

Data, code, and PDF version of this file are available at
`https://rdfia.github.io`

Introduction

Up until now, we have looked at "discriminative" approaches in computer vision through classification models. From a statistical point of view, these approaches model the distribution $P(Y \mid X)$. In this TP, we will look at *generative models* which traditionally model the joint distribution $P(X, Y)$ and thus the likelihood of X. In particular, we will examine **Generative Adversarial Networks (GAN)** - although they don't correspond to the traditional definition of generative models, they are very close. Since their debut in 2014, GANs have had a massive impact on unsupervised learning, requiring little or no labels. Their capacity to model the underlying structures of data have placed them in the heart of state-of-the-art methods in generation, completion, and edition of images. For example, would you have guessed that all of the faces in figure 1 are fake ? !

The goal of a "classic" GAN is to be capable of generating and image $\tilde{\mathbf{x}}$ from a noise vector $\mathbf{z}$ sampled from a predefined distribution.

$$\tilde{\mathbf{x}} = G(\mathbf{z}), \mathbf{z} \sim P(\mathbf{z}) \tag{1}$$

We can easily extend the framework to conditional GANs (cGANs), in which the goal is to produce an image $\tilde{\mathbf{x}}$ from a noise vector $\mathbf{z}$ as well as the input $\mathbf{y}$, which guides the generation process. For example, $\mathbf{y}$ can be a class label, an image, a sentence, etc.

$$\tilde{\mathbf{x}} = G(\mathbf{z}, \mathbf{y}), \mathbf{z} \sim P(\mathbf{z}) \tag{2}$$

Section 1 – Generative Adversarial Networks

1.1 General Principle

First, we will look at a "classic" unconditional GAN (Goodfellow et al., 2014). In particular, we will use the DCGAN architecture (Radford et al., 2016) as our basis, as it is the foundation of the majority of current state-of-the-art GAN models.

The objective of the GAN is to **generate realistic data**, which means plausible according to a probability distribution of real data $P(X)$. Obviously, this distribution is unknown : we only possess data sampled from this distribution $\mathcal{D}\text{ata} = \{\mathbf{x}_i \sim P(X)\}_{i=1..N}$: the examples from our dataset.

FIGURE 1 – All of these faces are completely fake. Results from Karras et al. (2021) ; trained respectively with aligned (left) and unaligned (right) face datasets.

Generator To solve this task, we will define the first neural network, G. It's objective is to transform any input $\mathbf{z}$ sampled from a fixed distribution –,generally $U_{[-1,1]}$ or $\mathcal{N}(0, I)$ – into a realistic image $\tilde{\mathbf{x}}$.

$$\tilde{\mathbf{x}} = G(\mathbf{z}), \quad \mathbf{z} \sim P(\mathbf{z}) \tag{3}$$

Discriminator. The cost function which allows to train this generator is not trivial - the distribution $P(X)$ is not known. To bypass this problem, we propose to train a second neural network : the discriminator D. This network takes an image as input and predicts whether this is a real image $\mathbf{x}^*$ from the training data or a fake image $\tilde{\mathbf{x}}$ produced by the generator.

$$D(\mathbf{x}) \in [0, 1], \quad \text{idéalement,} \begin{cases} D(\tilde{\mathbf{x}}) = 0, & \tilde{\mathbf{x}} = G(\mathbf{z}) \\ D(\mathbf{x}^*) = 1, & \mathbf{x}^* \in \mathcal{D}\text{ata} \end{cases} \tag{4}$$

Adversarial training. Training a GAN is particular, and can be seen as a "game" (under game theory) where two opponents are in competition : the generator continuously learns to fool the discriminator while the discriminator continuously adapts to the generator's modifications in order to distinguish the real examples from the fake ones. The classification cost of the discriminator will be a *binary cross-entropy* loss.

We aim to optimize the following problem :

$$\min_G \max_D \mathbb{E}_{\mathbf{x}^* \in \mathcal{D}\text{ata}} \left[\log D(\mathbf{x}^*)\right] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})} \left[\log\left(1 - D(G(\mathbf{z}))\right)\right] \tag{5}$$

Concretely, from a practical point of view, we alternate between training the generator and the discriminator, which each have a different objective function derived from the above equation :

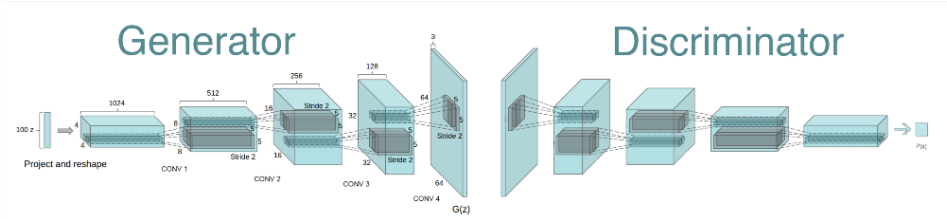


FIGURE 2 – Architecture of the classic DCGAN

$$\max_G \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})} \left[\log D(G(\mathbf{z}))\right] \tag{6}$$

$$\max_D \mathbb{E}_{\mathbf{x}^*\in\mathcal{D}\text{ata}} \left[\log D(\mathbf{x}^*)\right] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})} \left[\log\left(1 - D(G(\mathbf{z}))\right)\right] \tag{7}$$

Questions

1. Interpret the equations (6) and (7). What would happen if we only used one of the two ?
2. Ideally, what should the generator G transform the distribution $P(\mathbf{z})$ to ?
3. Remark that the equation (6) is not directly derived from the equation 5. This is justified by the authors to obtain more stable training and avoid the saturation of gradients. What should the "true" equation be here ?

1.2 Architecture of the networks

The large majority of GAN models for image generation are based on the architecture of Deep Convolutional GAN (DCGAN) proposed by Radford et al. (2016), shown in figure 2. This architecture proposes a discriminator and a generator based on convolution layers, batch normalization layers, reLU activations, as well as a certain number of *tricks* and *guidelines* to train a GAN for image generation. In our case, we will use a variant to generate images sized 32×32 pixels.

Discriminator. The discriminator has a classic architecture for an image "classification" model. It uses blocks composed of convolutions, batch normalizations and LeakyReLU activations. Implementation design includes using a stride of 2 instead of pooling, using a kernel of size 4, and using a LeakyReLU ; the objective being to better transfer the gradients. The exact architecture will be the following :

— Convolution (32 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + LeakyReLU ($\alpha = 0.2$)
— Convolution (64 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + LeakyReLU ($\alpha = 0.2$)
— Convolution (128 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + LeakyReLU ($\alpha = 0.2$)
— Convolution (1 filter, kernel 4, stride 1, padding 0, no bias) + Sigmoid activation

Generator. For the generator, we want to start from a vector $\mathbf{z}$ without spatial information and progressively inject spatial information in order to obtain the final image. For this, the idea is to use the "inverse" operation of a convolution with stride, so an operation which transforms for example a tensor sized $4 \times 4 \times 64$ into a tensor sized $8 \times 8 \times 32$. This operation is called **transposed convolution** (also known as *inverse convolution* or *deconvolution*), in which the functionality is described in figure 3. The basic idea consists in virtually spreading the "pixels" of the feature maps (and so spatially increasing the image size) before applying a convolution. Apart the use of this particular layer, the network is quite simple and its exact architecture is the following :

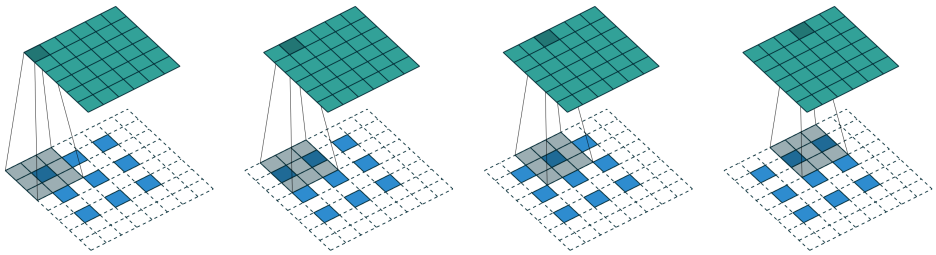


FIGURE 3 – Illustration of the transposed convolution. The 3×3 input (bottom) is in a way stretched to 6×6 before applying padding and a classical convolution to obtain the result 6×6 (above)

— Convolution Transpose (128 filters, kernel 4, stride 1, padding 0, no bias) + Batch Norm + ReLU
— Convolution Transpose (64 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + ReLU
— Convolution Transpose (32 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + ReLU
— Convolution Transpose (3 filters, kernel 4, stride 2, padding 1, no bias) + Tanh activation

Implementation

Follow the instructions in the provided notebook. You will notably define :

— The architectures of the two networks.
— The *prior* $P(\mathbf{z})$ which we will use. Here we choose a gaussian *prior* guassien $\mathcal{N}(0, I)$ with dimension n_z (a hyperparameter which we can vary, 100 by default)
— The loss of the "classification" of the discriminateur, a *binary cross-entropy* loss.
— The optimizers for each network. We will use *Adam*
— Code for training the networks.

With the hyperparameters given by default in the notebook, your GAN should generate digits which should start taking form after a few hundred batches, and should start looking more or less realistic after 1500 batches. Once the code works, we propose that you study different aspects concerning GANs, to observe the functionality and the drawbacks of this type of architecture. Here are some possible tests :

— Modify ngf or ndf. In particular, reduce or increase one of the two significantly.
— Replace the custom weight initialization with pytorch's default initialization.
— Replace the training loss of the generation with the "true" loss derived from the original equation (see question 3).
— Change the learning rate of on or both models.
— Learn for longer (ex : 30 epochs) even if it seems that the model already generates correct images.
— Reduce or increase significantly n_z (ex : $n_z = 10$ ou 1000).
— Using a learned GAN, take 2 noise vectors $\mathbf{z}_1$ and $\mathbf{z}_2$ and generate the images corresponding to several linear interpolations $\alpha\mathbf{z}_1 + (1 - \alpha)\mathbf{z}_2, \alpha \in [0, 1]$.
— Try to generate 64×64 images by adding another block in your models (resize your real data accordingly when defining the dataloader).
— Try another dataset, celeba or CIFAR-10, for example.

Questions

4. Comment on the training of of the GAN with the default settings (progress of the generations, the loss, stability, image diversity, etc.
5. Comment on the diverse experiences that you have performed with the suggestions above. In particular, comment on the stability on training, the losses, the diversity of generated images, etc.

Section 2 – Conditional Generative Adversarial Networks

2.1 General Principle

GANs can be extended to conditional models if the generator and the discriminator are conditioned by a supplementary condition $\mathbf{y}$ Mirza & Osindero (2014). For example, in the case of MNIST, we can supply the class label into the networks. In the case of faces, we could supply attribute information (wearing glasses, hair color, etc). The condition can equally be text or an image. See this cool tool to play with a state-of-the-art conditional GAN by providing either a sketch, a segmentation map, or a text description (or all three !). See figures 4a and 4b

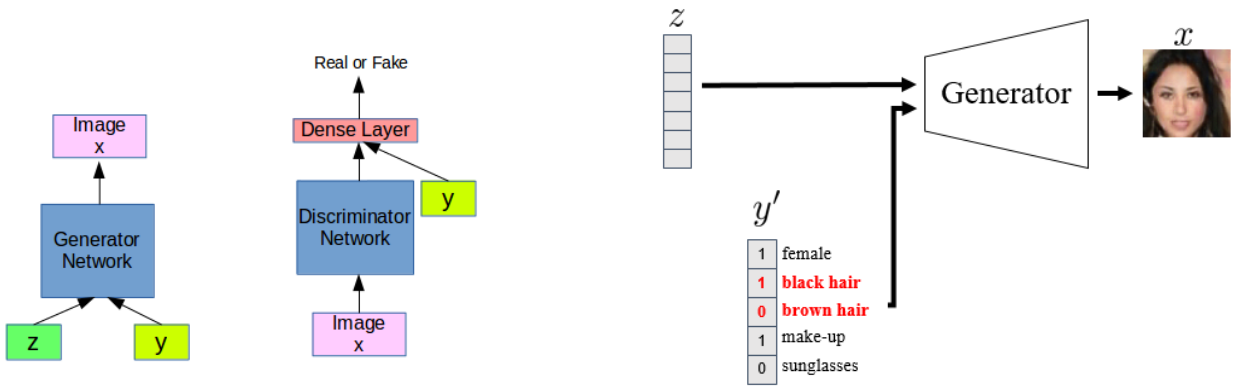
In our case, our networks will be defined as follows :

Generator. The conditional generator $cG(\mathbf{z}, \mathbf{y})$ will generate an image from a noise vector $\mathbf{z}$ and an attribute vector $\mathbf{y}$ associated with the image $\mathbf{x}^* \in \mathcal{D}\text{ata}$.

$$\tilde{\mathbf{x}} = cG(\mathbf{z}, \mathbf{y}), \quad \mathbf{z} \sim P(\mathbf{z}), \quad \mathbf{y} = attribute(\mathbf{x}^*) \tag{8}$$

Discriminator. To train our conditional generator, we define the conditional discriminator $cD(\mathbf{x}, \mathbf{y})$ which predicts if the image $\mathbf{x}$ possessing the attributes $\mathbf{y}$ is real $\mathbf{x}^*$ or generated $\tilde{\mathbf{x}}$.

$$D(\mathbf{x}, \mathbf{y}) \in [0, 1], \quad \text{ideally,} \begin{cases} D(\tilde{\mathbf{x}}, \mathbf{y}) = 0, & \tilde{\mathbf{x}} = G(\mathbf{z}, \mathbf{y}) \\ D(\mathbf{x}^*, \mathbf{y}) = 1, & \mathbf{x}^* \in \mathcal{D}\text{ata} \end{cases} \tag{9}$$



(a) Illustration of a cGAN

(b) Generation example of a cGAN

FIGURE 4 – Conditional GAN

2.2 cDCGAN Architectures for MNIST

We will extend our previous architecture to accommodate an extra condition : the label of the digit we wish to generate or discriminate. Here, $\mathbf{x}$ is the image and $\mathbf{y}$ the associated image. $\mathbf{y}$ will be represented as a one-hot vector of size 10 (ex : 3 $\rightarrow$ [0,0,0,1,0,0,0,0,0,0]). We wish to train a generator which takes as input a noise vector $\mathbf{z}$ and a label $\mathbf{y}$ in order to generate an image $\hat{\mathbf{x}}$ sized 32×32 corresponding to the label $\mathbf{y}$.

We will continue to use our generator and discriminator blocks. However, we need a method to fuse the label with the noise vector or the image before applying a standard DCGAN architecture. Figure 5 details the fusing process.

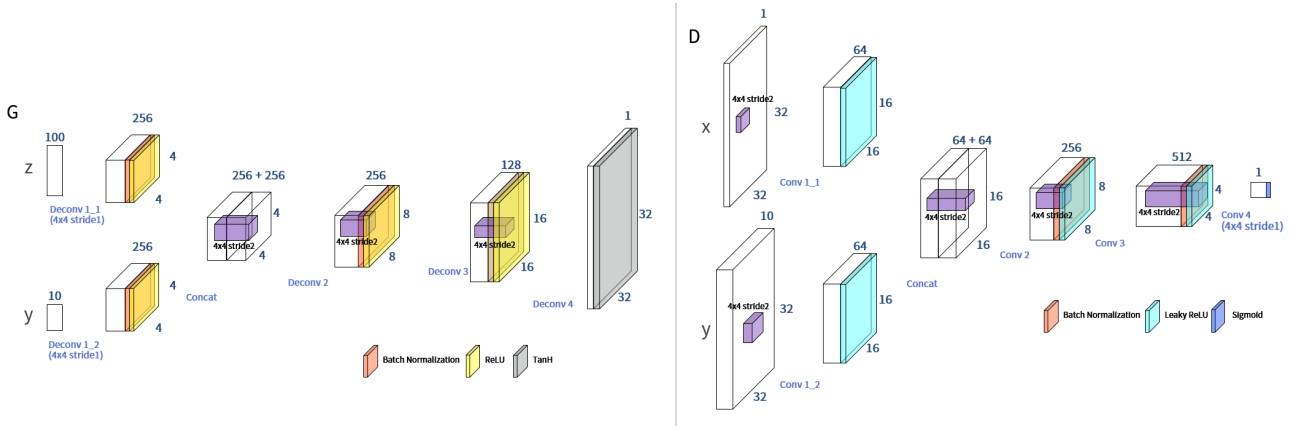


FIGURE 5 – The cDCGAN to implement.

Discriminator. Our discriminator is very similar as before, but we need to fuse the image with the one-hot-vector digit information. The full architecture is described here :

Branch 1 : Projection of the image $\mathbf{x}$

— Conv. (64 filters, kernel 4, stride 2, padding 1) + LeakyReLU ($\alpha = 0.2$)

Branch 2 : Projection of the label $\mathbf{y}$

— Expansion of $\mathbf{y}$ into a tensor sized $10 \times 32 \times 32$

— Conv. (64 filters, kernel 4, stride 2, padding 1) + LeakyReLU ($\alpha = 0.2$)

Concatenation and classification

— Concatenate branches 1 and 2 (64 + 64 channels)

— Conv. (256 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + LeakyReLU ($\alpha = 0.2$)

— Conv. (512 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + LeakyReLU ($\alpha = 0.2$)

— Conv. (1 filter kernel 4, stride 1, padding 0)

— Sigmoid activation

Generator. Likewise, the generator needs to fuse the label $\mathbf{y}$ with the noise vector. We perform a similar concatenation as before, detailed as follows :

Branch 1 : Projection of the vector $\mathbf{z}$

— Transpose Convolution (256 filters, kernel 4, stride 1, padding 0, no bias) + Batch Norm + ReLU

Branch 2 : Projection of the label $\mathbf{y}$

— Transpose Convolution (256 filters, kernel 4, stride 1, padding 0, no bias) + Batch Norm + ReLU

Concatenation and Image generation

— Concatenate branches 1 and 2 (256 + 256 channels)

— Transpose Convolution (256 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + ReLU

— Transpose Convolution (128 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + ReLU

— Transpose Convolution (1 filters, kernel 4, stride 2, padding 1)

— Tanh

Pratique

Follow the instructions in the provided notebook (complete ConditionalGenerator and ConditionalDiscriminator classes, as well as the code needed for training.)

With the hyperaparameters provided by default in the notebook. The images should start taking form quite quickly, and should look almost perfect by 2000 batches.

Questions

6. Comment on your experiences with the conditional DCGAN.
7. Could we remove the vector $\mathbf{y}$ from the input of the discriminator (so having $cD(\mathbf{x})$ instead of $cD(\mathbf{x}, \mathbf{y})$) ?
8. Was your training more or less successful than the unconditional case ? Why ?
9. Test the code at the end. Each column corresponds to a unique noise vector z. What could z be interpreted as here ?

Références

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, 2014.

Tero Karras, Miika Aittala, Samuli Laine, Erik Härkönen, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Alias-free generative adversarial networks. In *Proc. NeurIPS*, 2021.

Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv :1411.1784*, 2014.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *ICLR*, 2016.