

Polytech - Deep Learning - TP 9-10

Generative Adversarial Networks

Nicolas Thome

Matthieu Cord

Introduction

Jusqu'à maintenant, nous nous sommes intéressés à des **approches "discriminatives"** de la vision par ordinateur au travers de modèles de classification, c'est à dire d'un point de vue statistiques à des approches qui modélisent la distribution $P(Y | X)$. Dans ce TP, nous allons nous intéresser à des **modèles génératifs** qui modélisent traditionnellement la probabilité jointe $P(X, Y)$ et donc la vraisemblance des données X . En particulier, nous allons nous intéresser aux **Generative Adversarial Networks (GAN)**, qui, s'ils ne correspondent pas tout à fait à la définition traditionnelle des modèles génératifs, en sont très proche. Depuis leur introduction en 2014, les GAN ont un impact non négligeable sur l'apprentissage non-supervisé, nécessitant peu voir pas d'annotations. Leur capacité à modéliser les structures sous-jacentes aux données les place au coeur des techniques état de l'art en génération, complétion et modification d'images. Par exemple, sauriez-vous dire dans la figure 1 lesquelles de ces célébrités ont été générées par un GAN ?



FIGURE 1 – Lesquelles de ces célébrités ont été générées par un GAN ?

L'objectif d'un modèle GAN "classique" est d'être capable de générer une image \tilde{x} à partir d'une entrée z tirée selon une distribution pré-définie, et ce quel que soit le z choisi parmi la distribution.

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

Nous étudierons ensuite les GAN conditionnels (cGAN) dont le but est de produire une image \tilde{x} à partir d'une entrée y dont on connaît la sémantique et souvent d'un "bruit" z équivalent au z du GAN classique. Par exemple, y peut correspondre à une classe, ou à une image d'un autre domaine.

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

Partie 1 – Generative Adversarial Networks

1.1 Principe général

Dans un premier temps, nous allons nous intéresser au développement d'un réseau GAN "classique" (Goodfellow et al., 2014). En particulier, nous nous inspireront de l'architecture DCGAN (Radford et al., 2016) sur laquelle sont basés la grande majorité des modèles de GAN actuels.

L'objectif du GAN est de **générer des données réalistes**, c'est-à-dire plausibles selon une distribution de probabilité des données réelles $P(X)$. Évidemment, cette distribution est inconnue et on possède simplement des exemples de données tirées selon cette distribution $\mathcal{D}_{\text{data}} = \{\mathbf{x}_i \sim P(X)\}_{i=1..N}$: les exemples d'apprentissage de notre dataset.

Générateur. Pour résoudre cette tâche, on va définir un premier réseaux de neurones, le générateur G , dont l'objectif est de transformer n'importe quel entrée \mathbf{z} tiré selon une distribution fixées –généralement $U_{[-1,1]}$ ou $\mathcal{N}(0, I)$ – en une image $\tilde{\mathbf{x}}$ réaliste.

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

Discriminateur. La fonction de coût permettant l'apprentissage de ce générateur n'est cependant pas évidente, car la distribution $P(X)$ n'est pas connue. Pour contourner ce problème, on propose d'apprendre un second réseau de neurones, le discriminateur D . Ce réseau prendra une image en entrée et doit prédire si cette image est une image réelle \mathbf{x}^* du jeu de données ou si c'est une image $\tilde{\mathbf{x}}$ produite par le générateur.

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

Apprentissage adversaire. L'apprentissage d'un GAN est particulier, et peut être assimilé à un "jeu" (au sens de la théorie des jeux) où deux adversaires sont en compétition : le générateur apprend continuellement à tromper le discriminateur alors que le discriminateur apprend continuellement à s'adapter aux modifications du générateur pour toujours réussir à distinguer les exemples réels des exemples générés. Le coût de classification du discriminateur sera une *binary cross-entropy*.

On cherche donc à optimiser le problème suivant :

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

Concrètement, d'un point de vue pratique, on alterne entre l'apprentissage du générateur et du discriminateur, qui ont chacun une fonction objectif différente dérivée du problème ci-dessus :

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

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

Questions

1. Interprétez les équations (6) et (7). Que se passe-t-il si on utilisait seulement une des deux ?
2. Idéalement, en quoi le générateur G transforme la distribution $P(\mathbf{z})$?
3. Notez que l'équation (6) n'est pas directement dérivée de l'équation 5. Cela est justifié par les auteurs pour éviter la saturation des gradients. Quel aurait du être la "vraie" équation ici ?

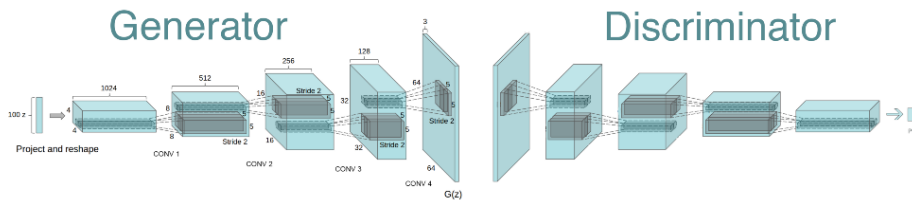
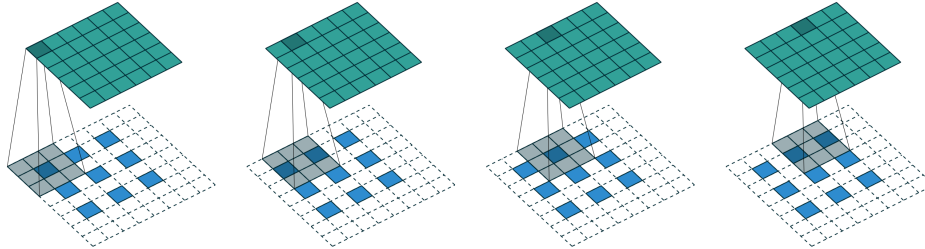


FIGURE 2 – Architecture DCGAN classique

FIGURE 3 – Illustration de la convolution transposée. L'entrée de taille 3×3 (en bas) est en quelque sorte élargie en 6×6 avant d'y appliquer du padding et une convolution classique pour obtenir une sortie de taille 6×6 (en haut)

1.2 Architectures des réseaux

La grande majorité des modèles de GAN pour la génération d'images se basent sur l'architecture Deep Convolutional GAN (DCGAN) représentée sur la figure 2 représente l'architecture proposée par Radford et al. (2016). Cette architecture propose un discriminateur et un générateur basé sur des couches de convolutions, de batch normalisation ainsi qu'un certain nombre de *tricks* et de *guidelines* pour apprendre des GAN pour la génération d'image. Dans notre cas, nous allons utiliser une variante pour générer des images 32×32 pixels.

Discriminateur. Le discriminateur a une architecture assez classique pour un modèle de "classification" d'images. Il alterne convolutions, batch normalization et activations LeakyReLU. Les particularités sont l'utilisation de stride dans les convolutions au lieu du pooling, le choix d'un kernel de taille 4, et l'utilisation de LeakyReLU dont l'objectif est de mieux transmettre les gradients. L'architecture exacte sera la suivante :

- 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

Générateur. Pour le générateur, on veut partir d'un vecteur \mathbf{z} sans information spatiale et injecter progressivement de l'information spatiale afin d'obtenir au final une image. Pour cela, l'idée classique est de chercher une opération "inverse" de la convolution avec stride, c'est à dire une opération qui transformerai par exemple un tenseur $4 \times 4 \times 64$ en un tenseur $8 \times 8 \times 32$. Cette opération s'appelle la **convolution transposée** (parfois appelée également *convolution inverse* ou *déconvolution*), dont le fonctionnement est décrit sur la figure 3. Le principe général consiste à espacer virtuellement les "pixels" des feature maps d'entrée (et donc d'augmenter spatialement la taille de l'entrée) avant d'y appliquer une convolution. Mise à part l'usage de cette couche particulière, le réseau est assez simple et son architecture exactement est la suivante :

- 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

Pratique

Suivez les instructions dans le notebook fourni (cf consignes du TME précédent pour utiliser les notebooks). Vous devrez notamment définir dans le code :

- Les architectures des deux réseaux de neurones
- Le *prior* $P(\mathbf{z})$ que l'on utilisera. Ici on choisit un *prior* gaussien $\mathcal{N}(0, I)$ de dimension n_z (un hyperparamètre que l'on pourra faire varier, par défaut 100)
- La loss de "classification" du discriminateur, une *binary cross-entropy*
- Les optimiseurs pour chaque réseau, on utilisera *Adam*
- Des parties du code d'apprentissage

Avec les hyperparamètres fournis par défaut dans le notebook, votre GAN devrait générer des visages relativement corrects après quelques centaines de batches. Une fois le code fonctionnel, on vous propose d'étudier divers aspects concernant les GAN, afin d'observer les modes de fonctionnement ou d'erreur de ce type d'architecture. Voici quelques exemples de tests possibles :

- Modifier le momentum des optimiseurs pour 0.9 par exemple, qui est la valeur par défaut utilisée le plus souvent avec Adam
- Remplacer la loss d'apprentissage du générateur par la "vraie" loss dérivée de l'équation d'origine (voir question 3)
- Modifier l'équilibrage entre les deux modèles en apprenant l'un plus souvent que l'autre (paramètre `nb_update_D/G`)
- Changer le learning rate d'un ou des deux modèles
- Apprendre plus longtemps (ex : 30 epochs) même si le modèle semble déjà générer des images correctes
- Réduire ou augmenter n_z de façon importante (ex : $n_z = 10$ ou 1000)
- Avec un GAN appris qui vous convient, prendre deux vecteurs \mathbf{z}_1 et \mathbf{z}_2 , et générez les images correspondant à de nombreuses interpolations linéaires $\alpha\mathbf{z}_1 + (1 - \alpha)\mathbf{z}_2, \alpha \in [0, 1]$
- Essayer un autre dataset (CIFAR-10 par exemple)
- Passer à une architecture pour des images 64×64 : ajouter une couche de conv+batchnorm+activation dans chaque modèle en suivant le même patterns sur la taille des filtres, avec une couche supplémentaire à $8 \cdot \text{ndf}/g$; lors du chargement du dataset, remplacer `celeba` par `celeba64` dans le code (à faire sur Google Colab pour un temps de calcul raisonnable)

Questions

- Commentez l'apprentissage du GAN avec les hyperparamètres proposés par défaut (évolution des générations, de la loss, stabilité, diversité des images, etc.)
- Commentez les diverses expériences complémentaires que vous avez réalisé (consignes ci-dessus), comment influent-elle sur la qualité de l'apprentissage (vitesse, stabilité, évolution de la loss, qualité visuelles, diversité des générations, etc.)

Partie 2 – Conditional Generative Adversarial Networks

2.1 Principe général

Les GAN peuvent être étendus aux modèles conditionnels si le générateur et le discriminateur sont conditionnés par une information supplémentaire y Mirza & Osindero (2014). Par exemple, dans le cas de notre base de données de célébrités, il est possible d'exploiter l'information d'attribut (sexe, lunettes, barbe, etc.). De cette façon, nous déciderons des attributs de la célébrité à générer (homme à lunettes, femme à barbe, etc.). La différence entre deux célébrités possédant les mêmes attributs se fera à l'aide du vecteur aléatoire z .

Pour rappel, un GAN est composé de deux réseaux : le générateur $G(z)$ et le discriminateur $D(x)$. Un cGAN est un GAN conditionné par une variable y , par exemple un vecteur d'attributs. Dans ce cas, nous allons donner y au deux réseaux, voir figures 4 et 5.

Générateur. On définit le générateur conditionnel $cG(z, y)$ qui génère une image à partir d'un vecteur aléatoire z et d'un vecteur d'attributs y associé à l'image $x^* \in \mathcal{Data}$.

$$\tilde{x} = cG(z, y), \quad z \sim P(z), \quad y = \text{attribut}(x^*) \quad (8)$$

Discriminateur. Pour apprendre notre générateur conditionnel, on définit le discriminateur conditionnel $cD(x, y)$ qui prédit si l'image x possédant les attributs y est une image réelle x^* ou une image générée \tilde{x} .

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

Questions

6. Formellement dans le cas cGAN, quel est le problème qu'on cherche à optimiser ? (réécrire l'équation (6) et (7) avec un discriminateur et générateur conditionnels)
7. A quelle(s) variable(s) le générateur de la figure 6 pourrait-il être conditionné, sachant que ce modèle permet de modifier des images existantes ?
8. A quelle(s) variable(s) le générateur de la vidéo ci-après pourrait-il être conditionné ?
<https://twitter.com/HumanVsMachine/status/1068909241405251584>
9. A quelle(s) variable(s) le générateur du début de la vidéo ci-après pourrait-il être conditionné ?
<http://youtu.be/ayPqjPekn7g>

2.2 Architectures cDCGAN pour MNIST

Nous allons étendre l'architecture Deep Convolutional GAN (DCGAN) au cas conditionnel pour le cas de la génération de chiffre MNIST. La base de données MNIST est composée d'images de chiffres entre 0 et 9. Nous considérerons ici que x est l'image et y le chiffre qui lui est associé. y sera représenté sous la forme d'un vecteur one-hot de taille 10 (ex : 3 \rightarrow [0,0,0,1,0,0,0,0,0]). De fait, nous voulons apprendre un générateur prenant en entrée un bruit z et un label y afin de générer une image \hat{x} de taille 32×32 pixels associée à au label y .

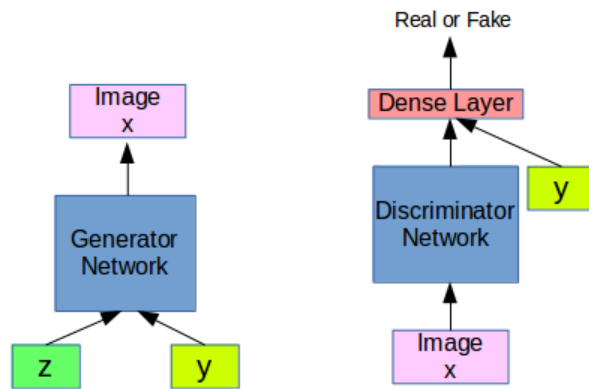


FIGURE 4 – Illustration d'un cGAN.

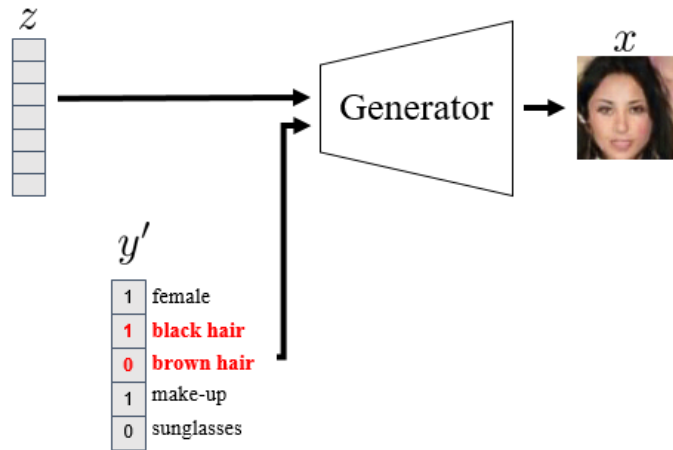


FIGURE 5 – Exemple de génération par un cGAN.



FIGURE 6 – Extension du cGAN avec un Fader Network permettant d'effectuer une transition d'attributs à partir d'une image réelle.

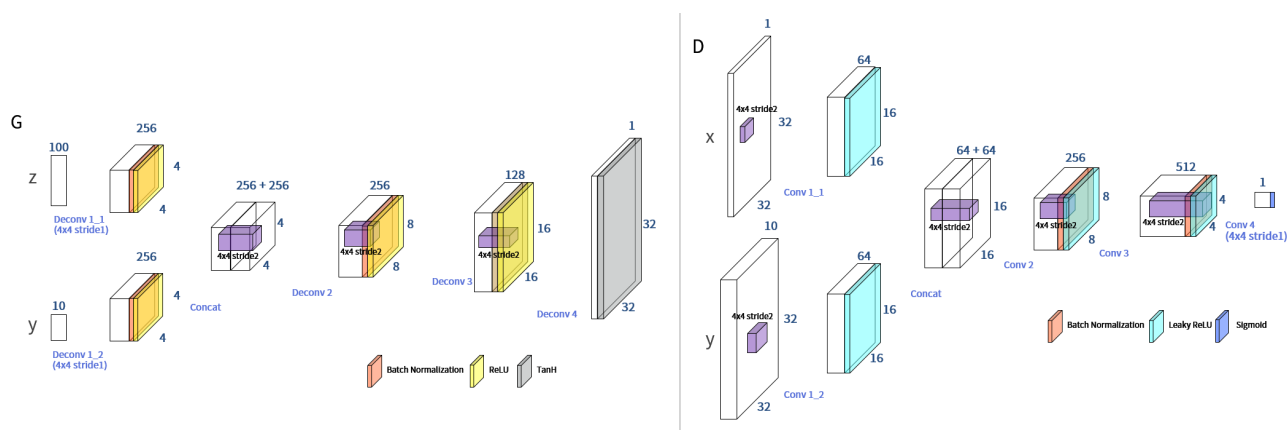


FIGURE 7 – Illustration du cDCGAN à implémenter.

Discriminateur. Notre discriminateur est similaire à celui donné précédemment. Toutefois, nous ajoutons le label y avant la classification en utilisant une concaténation. L'architecture à implémenter est illustrée sur la figure 7 et détaillée ci-après :

Branche 1 : Projection de l'image x

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

Branche 2 : Projection du label y

- Expansion de y en un tenseur de taille $10 \times 32 \times 32$
- Conv. (64 filters, kernel 4, stride 2, padding 1) + LeakyReLU ($\alpha = 0.2$)

Concaténation et classification

- Concatenation branches 1 et 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

Générateur. De la même façon, nous reprenons notre dernier générateur et ajoutons notre label y . Cette fois-ci, nous effectuons une première projection de notre vecteur de bruit z et de notre vecteur de label y avant de les concaténer. L'architecture est la suivante :

Branche 1 : Projection du vecteur z

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

Branche 2 : Projection du label y

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

Concaténation et génération de l'image

- Concaténation branches 1 et 2 (256 + 256 channels)
- Convolution Transpose (256 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + ReLU
- Convolution Transpose (128 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + ReLU
- Convolution Transpose (1 filters, kernel 4, stride 2, padding 1)
- Tanh

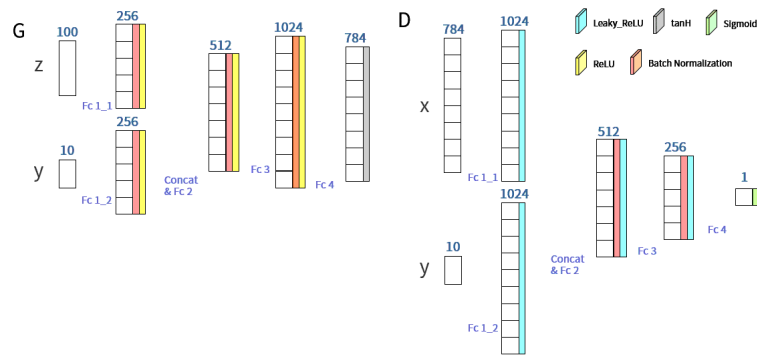


FIGURE 8 – Illustration du cGAN à implémenter.

Pratique

Suivez les instructions dans le notebook fourni (cf consignes du TME précédent pour utiliser les notebooks). Vous devrez notamment définir dans le code :

- Les architectures des deux réseaux de neurones
- Le *prior* $P(\mathbf{z})$ que l'on utilisera. Ici on choisit un *prior* gaussien $\mathcal{N}(0, I)$ de dimension n_z (un hyperparamètre que l'on pourra faire varier, par défaut 100)
- La loss de "classification" du discriminateur, une *binary cross-entropy*
- Les optimiseurs pour chaque réseau, on utilisera *Adam*
- Des parties du code d'apprentissage

Avec les hyperparamètres fournis par défaut dans le notebook, votre cDCGAN devrait générer des chiffres conditionnés relativement corrects après quelques centaines de batches.

Questions

10. Commentez vos expériences avec le DCGAN conditionnel
11. Pourrait-on enlever le vecteur \mathbf{y} des entrées du discriminateur (c'est-à-dire avoir $cD(\mathbf{x})$ et non $cD(\mathbf{x}, \mathbf{y})$) ?

2.3 Architectures cGAN pour MNIST

De même que pour l'architecture Deep Convolutional GAN conditionnel (cDCGAN), nous allons cette fois implémenter l'architecture GAN conditionnel simple. Cette dernière est basée sur des couches "fully connected" au lieu de couches de convolution.

Discriminateur. Notre discriminateur est similaire à celui donné précédemment. L'architecture à implémenter est illustrée sur la figure 8 et détaillée ci-après :

Branche 1 : Projection de l'image x

- Linear (1024 neurones) + LeakyReLU ($\alpha = 0.2$)

Branche 2 : Projection du label y

- Linear (1024 neurones) + LeakyReLU ($\alpha = 0.2$)

Concaténation et classification

- Concatenation branches 1 et 2 (1024 + 1024)
- Linear (1024 neurones, no bias) + BatchNorm + LeakyReLU ($\alpha = 0.2$)
- Linear (512 neurones, no bias) + BatchNorm + LeakyReLU ($\alpha = 0.2$)
- Linear (1 neurone)
- Sigmoid activation

Générateur. De la même façon, nous reprenons notre dernier générateur et ajoutons notre label y . Cette fois-ci, nous effectuons une première projection de notre vecteur de bruit z et de notre vecteur de label y avant de les concaténer. L'architecture est la suivante :

Branche 1 : Projection du vecteur z

- Linear (256 neurones, no bias) + Batch Norm + ReLU

Branche 2 : Projection du label y

- Linear (256 neurones, no bias) + Batch Norm + ReLU

Concaténation et génération de l'image

- Concatenation branches 1 et 2 (256 + 256)
- Linear (512 neurones, no bias) + Batch Norm + ReLU
- Linear (1024 neurones, no bias) + Batch Norm + ReLU
- Linear (1024 neurones)
- Tanh

Pratique

Suivez les instructions dans le notebook fourni (cf consignes du TME précédent pour utiliser les notebooks). Vous devrez notamment définir dans le code :

- Les architectures des deux réseaux de neurones
- Le *prior* $P(\mathbf{z})$ que l'on utilisera. Ici on choisit un *prior* gaussien $\mathcal{N}(0, I)$ de dimension n_z (un hyperparamètre que l'on pourra faire varier, par défaut 100)
- La loss de "classification" du discriminateur, une *binary cross-entropy*
- Les optimiseurs pour chaque réseau, on utilisera *Adam*
- Des parties du code d'apprentissage

Questions

12. Commentez vos expériences avec le GAN conditionnel, reportez la meilleure génération de chiffres conditionnés
13. Il est relativement plus difficile de générer des chiffres conditionnés avec un cGAN qu'avec un cDCGAN, pourquoi ?

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.

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.