CHAPITRE 4

Réseaux adverses génératifs

Objectifs du chapitre

Dans ce chapitre, vous allez :

- Découvrez la conception architecturale d'un réseau contradictoire génératif (GAN).
- Créez et entraînez un GAN convolutif profond (DCGAN) à partir de zéro à l'aide de Keras.
- Utilisez le DCGAN pour générer de nouvelles images.
- Comprendre certains des problèmes courants rencontrés lors de la formation d'un DCGAN.
- Découvrez comment l'architecture Wasserstein GAN (WGAN) résout ces problèmes.
- Comprendre les améliorations supplémentaires qui peuvent être apportées au WGAN, telles que incorporer un terme de pénalité de gradient (GP) dans la fonction de perte.
- Créez un WGAN-GP à partir de zéro en utilisant Keras.
- Utilisez le WGAN-GP pour générer des visages.
- Découvrez comment un GAN conditionnel (CGAN) vous donne la possibilité de conditionner sortie générée sur une étiquette donnée.
- Créez et entraînez un CGAN dans Keras et utilisez-le pour manipuler une image générée.

En 2014, Ian Goodfellow et coll. a présenté un article intitulé « Generative Adversarial Nets »1 lors de la conférence Neural Information Processing Systems (NeurIPS) à Montréal. L'introduction des réseaux antagonistes génératifs (ou GAN, comme on les appelle plus communément) est désormais considérée comme un tournant clé dans l'histoire de la modélisation générative, car les idées fondamentales présentées dans cet article ont donné naissance à certains des réseaux génératifs les plus réussis et les plus impressionnants. modèles jamais créés.

Ce chapitre présentera d'abord les fondements théoriques des GAN, puis nous verrons comment construire notre propre GAN à l'aide de Keras.

Introduction

Commençons par une courte histoire pour illustrer certains des concepts fondamentaux utilisés dans le processus de formation GAN.

Brickki Bricks et les forgerons

C'est votre premier jour dans votre nouvel emploi en tant que responsable du contrôle qualité chez Brickki, une entreprise spécialisée dans la production de blocs de construction de haute qualité de toutes formes et tailles (Figure 4-1).

Figure 4-1. La chaîne de production d'une entreprise fabriquant des briques de différentes formes et tailles (créée avec Midjourney)

Vous êtes immédiatement alerté d'un problème sur certains articles sortant de la chaîne de production. Un concurrent a commencé à fabriquer des copies contrefaites des briques Brickki et a trouvé le moyen de les mélanger aux sacs reçus par vos clients. Vous décidez de devenir un expert pour faire la différence entre les briques contrefaites et les briques réelles, afin de pouvoir intercepter les briques forgées sur la chaîne de production avant qu'elles ne soient remises aux clients. Au fil du temps, en écoutant les commentaires des clients, vous devenez progressivement plus apte à repérer les contrefaçons.

Les faussaires ne sont pas contents de cela : ils réagissent à vos capacités de détection améliorées en apportant quelques modifications à leur processus de contrefaçon, de sorte que désormais, la différence entre les vraies briques et les contrefaçons est encore plus difficile à repérer.

N'étant pas du genre à abandonner, vous vous recyclez pour identifier les contrefaçons les plus sophistiquées et essayez de garder une longueur d'avance sur les faussaires. Ce processus se poursuit, les faussaires mettant à jour de manière itérative leurs technologies de création de briques pendant que vous essayez de devenir de plus en plus compétent dans l'interception de leurs contrefaçons.

Chaque semaine qui passe, il devient de plus en plus difficile de faire la différence entre les vraies briques Brickki et celles créées par les forgerons. Il semble que ceci

> Un simple jeu du chat et de la souris suffit à entraîner une amélioration significative à la fois de la qualité de la contrefaçon et de la qualité de la détection.

L'histoire des briques Brickki et des forgerons décrit le processus de formation d'un réseau conflictuel génératif.

Un GAN est une bataille entre deux adversaires, le générateur et le discriminateur. Le générateur tente de convertir le bruit aléatoire en observations qui semblent avoir été échantillonnées à partir de l'ensemble de données d'origine, et le discriminateur tente de prédire si une observation provient de l'ensemble de données d'origine ou s'il s'agit d'une contrefaçon du générateur.
Des exemples d'entrées et de sorties des deux réseaux sont présentés dans la figure 4-2.

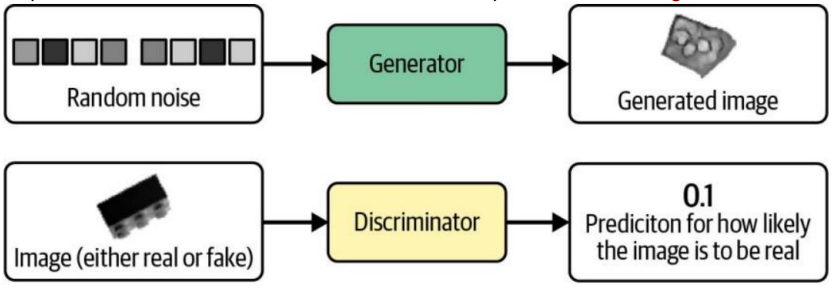

Figure 4-2. Entrées et sorties des deux réseaux dans un GAN

Au début du processus, le générateur produit des images bruitées et le discriminateur prédit de manière aléatoire. La clé des GAN réside dans la façon dont nous alternons la formation des deux réseaux, de sorte qu'à mesure que le générateur devient plus apte à tromper le discriminateur, celui-ci doit s'adapter afin de maintenir sa capacité à identifier correctement quelles observations sont fausses. Cela pousse le générateur à trouver de nouvelles façons de tromper le discriminateur, et ainsi le cycle continue.

GAN convolutif profond (DCGAN)

Pour voir cela en action, commençons à construire notre premier GAN à Keras, pour générer des images de briques.

Nous suivrons de près l'un des premiers articles majeurs sur les GAN, « Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks ».2 Dans cet article de 2015, les auteurs montrent comment construire un GAN à convolution profonde pour générer des images réalistes à partir d'une variété de ensembles de données. Ils introduisent également plusieurs changements qui améliorent considérablement la qualité des images générées.

Exécution du code pour cet exemple

Le code de cet exemple se trouve dans le notebook Jupyter situé à notebooks/04_gan/01_dcgan/dcgan.ipynb dans le référentiel de livres.

L'ensemble de données des briques

Tout d'abord, vous devrez télécharger les données d'entraînement. Nous utiliserons l' ensemble de données Images of LEGO Bricks qui est disponible via Kaggle. Il s'agit d'une collection rendue par ordinateur de 40 000 images photographiques de 50 briques de jouets différentes, prises sous plusieurs angles.

Quelques exemples d'images de produits Brickki sont présentés dans la figure 4-3.

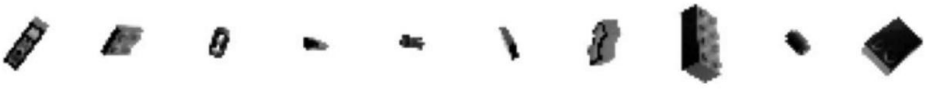

Figure 4-3. Exemples d'images de l'ensemble de données Bricks

Vous pouvez télécharger l'ensemble de données en exécutant le script de téléchargement de l'ensemble de données Kaggle dans le référentiel de livres, comme le montre l' exemple 4-1. Cela enregistrera les images et accompagnant les métadonnées localement dans le dossier /data . Exemple 4-1. Téléchargement du

Ensemble de données de briques

```
bash scripts/download_kaggle_data.sh joosthazelzet lego-brick-images
```

Nous utilisons la fonction Keras image_dataset_from_directory pour créer un ensemble de données TensorFlow pointé vers le répertoire où les images sont stockées, comme le montre l' exemple 4-2. Cela nous permet de lire des lots d'images en mémoire uniquement lorsque cela est nécessaire (par exemple, pendant la formation), afin que nous puissions travailler avec de grands ensembles de données sans nous soucier de devoir mettre l'intégralité de l'ensemble de données en mémoire. Il redimensionne également les images à 64 × 64, en interpolant entre les valeurs de pixels.

Exemple 4-2. Création d'un ensemble de données TensorFlow à partir de fichiers image dans un répertoire

```
train_data = utils.image_dataset_from_directory(
        "/app/data/lego-brick-images/dataset/",
        étiquettes=Aucun,
color_mode="grayscale",
image_size=(64, 64),
batch_size=128,
shuffle=True,                    graine = 42,

interpolation="bilinéaire", )
```

Les données originales sont mises à l'échelle dans la plage [0, 255] pour indiquer l'intensité des pixels. Lors de la formation des GAN, nous redimensionnons les données sur la plage [–1, 1] afin de pouvoir utiliser la fonction d'activation tanh sur la couche finale du générateur, qui a tendance à fournir des gradients plus forts que la fonction sigmoïde (Exemple 4-3). Exemple 4-3. Prétraitement du

Ensemble de données de briques

```
prétraitement def (img):                    img =
 (tf.cast(img, "float32") - 127,5) / 127,5
retourner img train = train_data.map (lambda
x : prétraiter(x))
```

Voyons maintenant comment nous construisons le discriminateur.

Le discriminateur

Le but du discriminateur est de prédire si une image est réelle ou fausse. Il s'agit d'un problème de classification d'images supervisé, nous pouvons donc utiliser une architecture similaire à celles avec lesquelles nous avons travaillé au chapitre 2 : couches convolutives empilées, avec un seul nœud de sortie.
L'architecture complète du discriminateur que nous allons construire est présentée dans le tableau 4-1.

Tableau 4-1. Résumé du modèle du discriminateur

Calque (type)	Forme de sortie	Paramètre #
Couche d'entrée	(Aucun, 64, 64, 1)	0
Conv2D	(Aucun, 32, 32, 64)	1 024
FuiteReLU	(Aucun, 32, 32, 64)	0
Abandonner	(Aucun, 32, 32, 64)	0
Conv2D	(Aucun, 16, 16, 128)	131 072
Normalisation par lots	(Aucun, 16, 16, 128)	512
FuiteReLU	(Aucun, 16, 16, 128)	0
Abandonner	(Aucun, 16, 16, 128)	0
Conv2D	(Aucun, 8, 8, 256)	524,288
Normalisation par lots	(Aucun, 8, 8, 256)	1 024

FuiteReLU	(Aucun, 8, 8, 256)	0
Abandonner	(Aucun, 8, 8, 256)	0
Conv2D	(Aucun, 4, 4, 512)	2 097 152
Normalisation par lots	(Aucun, 4, 4, 512)	2 048

Calque (type)	Forme de sortie	Paramètre #
FuiteReLU	(Aucun, 4, 4, 512)	0
Abandonner	(Aucun, 4, 4, 512)	0
Conv2D	(Aucun, 1, 1, 1)	8 192
Aplatir	(Aucun, 1)	0

Paramètres totaux 2 765 312

Paramètres entraînables 2 763 520

Paramètres non entraînables 1 792

Le code Keras pour construire le discriminateur est fourni dans l'exemple 4-4. Exemple 4-4. Le discriminateur

```
discriminator_input = layer.Input(shape=(64, 64, 1)) x = layer.Conv2D(64, kernel_size=4, ❶
strides=2, padding="same", use_bias = False)(
                                        entrée_discriminateur
) ❷
x =
couches.LeakyReLU(0.2)(x)
x =
couches.Dropout(0.3)(x) x
= couches.Conv2D(
        128, kernel_size=4, strides=2, padding="same", use_bias = False
)(X)
x = couches.BatchNormalization (élan = 0,9) (x)
x =
couches.LeakyReLU(0.2)(x)
x =
couches.Dropout(0.3)(x) x
= couches.Conv2D(
        256, kernel_size=4, strides=2, padding="same", use_bias = False
)(X)
x = couches.BatchNormalization (élan = 0,9) (x)
```

```
x =
couches.LeakyReLU(0.2)(x)
x =
couches.Dropout(0.3)(x) x
= couches.Conv2D(
        512, kernel_size=4, strides=2, padding="same", use_bias = False
)(X)
x = couches.BatchNormalization (élan = 0,9) (x)
x =
couches.LeakyReLU(0.2)(x)
x =
couches.Dropout(0.3)(x) x
= couches.Conv2D(
        1,
kernel_size=4,
strides=1,
padding="valid", use_bias
= False, activation =
'sigmoïde'
)(X)

discriminator_output = layer.Flatten()(x) discriminator =                    ❸
models.Model(discriminator_input,
```

❶ discriminator_output) Définissez❹ la couche d'entrée du discriminateur (la

❷ image).

Empilez les couches Conv2D les unes sur les autres, avec les couches BatchNormalization, LeakyReLU et Dropout prises en sandwich entre les deux.

❸ Aplatissez la dernière couche convolutive : à ce stade, la forme du tenseur est 1 × 1 × 1, il n'est donc pas nécessaire d'avoir une couche Dense finale .

❹ Le modèle Keras qui définit le discriminateur : un modèle qui prend une image d'entrée et génère un nombre unique compris entre 0 et 1.

Remarquez comment nous utilisons une foulée de 2 dans certaines couches Conv2D pour réduire la forme spatiale du tenseur lors de son passage à travers le réseau (64 dans l'image d'origine, puis 32, 16, 8, 4 et enfin 1), tandis que augmenter le nombre de canaux (1 dans l'image d'entrée en niveaux de gris, puis 64, 128, 256 et enfin 512), avant de se réduire à un seul

prédiction.

Nous utilisons une activation sigmoïde sur la couche Conv2D finale pour générer un nombre compris entre 0 et 1.

Le générateur

Construisons maintenant le générateur. L'entrée du générateur sera un vecteur tiré d'une distribution normale standard multivariée. Le résultat est une image de la même taille qu'une image dans les données d'entraînement d'origine.

Cette description peut vous rappeler le décodeur dans un auto-encodeur variationnel. En fait, le générateur d'un GAN remplit exactement le même objectif que le décodeur d'un VAE : convertir un vecteur dans l'espace latent en image. Le concept de mappage d'un espace latent vers le domaine d'origine est très courant dans la modélisation générative, car il nous donne la possibilité de manipuler des vecteurs dans l'espace latent pour modifier les caractéristiques de haut niveau des images dans le domaine d'origine.

L'architecture du générateur que nous allons construire est présentée dans le tableau 4-2.

Tableau 4-2. Résumé du modèle du générateur

Calque (type)	Forme de sortie	Paramètre #
Couche d'entrée	(Aucun, 100)	0
Remodeler	(Aucun, 1, 1, 100)	0
Conv2DTranspose	(Aucun, 4, 4, 512)	4 819 200
Normalisation par lots	(Aucun, 4, 4, 512)	2 048
Calque (type)	**Forme de sortie**	**Paramètre #**
CV	(Aucun, 4, 4, 512)	0
Conv2DTranspose	(Aucun, 8, 8, 256)	2 097 152
Normalisation par lots	(Aucun, 8, 8, 256)	1 024
CV	(Aucun, 8, 8, 256)	0
Conv2DTranspose	(Aucun, 16, 16, 128)	524 288
Normalisation par lots	(Aucun, 16, 16, 128)	512

CV	(Aucun, 16, 16, 128)	0
Conv2DTranspose	(Aucun, 32, 32, 64)	131 072
Normalisation par lots	(Aucun, 32, 32, 64)	256
CV	(Aucun, 32, 32, 64)	0
Conv2DTranspose	(Aucun, 64, 64, 1)	1 024

Paramètres totaux 3 576 576

Paramètres entraînables 3 574 656 Paramètres non entraînables 1 920

Le code de construction du générateur est donné dans l'exemple 4-5. Exemple 4-5. Le

Générateur

```
generate_input = layer.Input(shape=(100,)) x = ❶
layer.Reshape((1, 1, 100))(generator_input) x = layer.Conv2DTranspose( ❷

        512, kernel_size=4, strides=1, padding="valid", use_bias = False
)(x) x ❸
= couches.BatchNormalization(momentum=0.9)(x)
x = couches.LeakyReLU(0.2)(x) x =
calques.Conv2DTranspose(
        256, kernel_size=4, strides=2, padding="same", use_bias = False
)(X)
x = couches.BatchNormalization(momentum=0.9)(x)
x = couches.LeakyReLU(0.2)(x) x =
calques.Conv2DTranspose(
        128, kernel_size=4, strides=2, padding="same", use_bias = False
)(X)
x = couches.BatchNormalization(momentum=0.9)(x)
x = couches.LeakyReLU(0.2)(x) x =
calques.Conv2DTranspose(
        64, kernel_size=4, strides=2, padding="same", use_bias = False
)(X)
x = couches.BatchNormalization(momentum=0.9)(x)
x = couches.LeakyReLU(0.2)(x)
générateur_output = couches.Conv2DTranspose(
        1,
```

```
        kernel_size=4, strides=2,
padding="same",
use_bias = False, activation
= 'tanh'

)(x) générateur = models.Model(generator_input,
générateur_sortie)
```

❶ Définissez la couche d'entrée du générateur : un vecteur de longueur 100.

❷ Nous utilisons un calque Reshape pour donner un tenseur 1 × 1 × 100, afin que nous puissions commencer à appliquer des opérations de transposition convolutive.

❸ Nous transmettons cela à travers quatre couches Conv2DTranspose , avec Couches BatchNormalization et LeakyReLU prises en sandwich entre les deux.

❹ La couche finale Conv2DTranspose utilise une fonction d'activation tanh pour transformer la sortie dans la plage [–1, 1], afin de correspondre au domaine d'image d'origine.

❺ Le modèle Keras qui définit le générateur, un modèle qui accepte un vecteur de longueur 100 et génère un tenseur de forme [64, 64, 1].

Remarquez comment nous utilisons une foulée de 2 dans certaines couches Conv2DTranspose pour augmenter la forme spatiale du tenseur lorsqu'il traverse le réseau (1 dans le vecteur d'origine, puis 4, 8, 16, 32 et enfin 64), tandis que diminuer le nombre de canaux (512 puis 256, 128, 64 et enfin 1 pour correspondre à la sortie en niveaux de gris).

Suréchantillonnage et Conv2DTranspose

Une alternative à l'utilisation des couches Conv2DTranspose consiste à utiliser à la place une couche UpSampling2D suivie d'une couche Conv2D normale avec la foulée 1, comme indiqué dans l'exemple 4-6.

Exemple 4-6. Exemple de suréchantillonnage

```
x = couches.UpSampling2D(taille = 2)(x) x =
layer.Conv2D(256, kernel_size=4, strides=1, padding="same")(x)
```

La couche UpSampling2D répète simplement chaque ligne et colonne de son entrée afin de doubler la taille. La couche Conv2D avec la foulée 1 effectue ensuite l'opération de convolution. C'est une idée similaire à la transposition convolutive, mais au lieu de combler les espaces entre les pixels avec des zéros, le suréchantillonnage répète simplement les valeurs de pixels existantes.

Il a été démontré que la méthode Conv2DTranspose peut conduire à des artefacts ou à de petits motifs en damier dans l'image de sortie (voir Figure 4-4) qui gâchent la qualité de la sortie. Cependant, ils sont toujours utilisés dans bon nombre des GAN les plus impressionnants du monde.

littérature et se sont révélés être un outil puissant dans le domaine du praticien de l'apprentissage profond toolbox.

Figure 4-4. Artifacts when using convolutional transpose layers (source: Odena et al., 2016)[3]

Both of these methods—`UpSampling2D` + `Conv2D` and `Conv2DTranspose`—are acceptable ways to transform back to the original image domain. It really is a case of testing both methods in your own problem setting and seeing which produces better results.

Formation de la DCGAN

Comme nous l'avons vu, les architectures du générateur et du discriminateur dans un DCGAN sont très simples et pas si différentes des modèles VAE que nous avons examinés au chapitre 3.
La clé pour comprendre les GAN réside dans la compréhension du processus de formation du générateur et du discriminateur.

Nous pouvons former le discriminateur en créant un ensemble de formation dans lequel certaines images sont de véritables observations de l'ensemble de formation et d'autres sont de fausses sorties du générateur. Nous traitons ensuite cela comme un problème d'apprentissage supervisé, où les étiquettes sont 1 pour les images réelles et 0 pour les fausses images, avec une entropie croisée binaire comme perte.

fonction.

Comment devons-nous former le générateur ? Nous devons trouver un moyen de noter chaque image générée afin qu'elle puisse être optimisée pour obtenir des images avec un score élevé. Heureusement, nous avons un discriminateur qui fait exactement cela ! Nous pouvons générer un lot d'images et les transmettre via le discriminateur pour obtenir un score pour chaque image. La fonction de perte pour le générateur est alors simplement l'entropie croisée binaire entre ces probabilités et un vecteur de uns, car nous voulons entraîner le générateur à produire des images que le discriminateur pense être réelles.

Il est essentiel d'alterner la formation de ces deux réseaux, en veillant à ne mettre à jour les poids d'un seul réseau à la fois. Par exemple, lors du processus de formation du générateur, seuls les poids du générateur sont mis à jour. Si nous permettions aux poids du discriminateur de changer également, le discriminateur s'ajusterait simplement de manière à avoir plus de chances de prédire que les images générées sont réelles, ce qui n'est pas le résultat souhaité. Nous voulons que les images générées soient prédites proches de 1 (réel) parce que le générateur est fort, et non parce que le discriminateur est faible.

Un diagramme du processus de formation du discriminateur et du générateur est présenté à la figure 4-5.

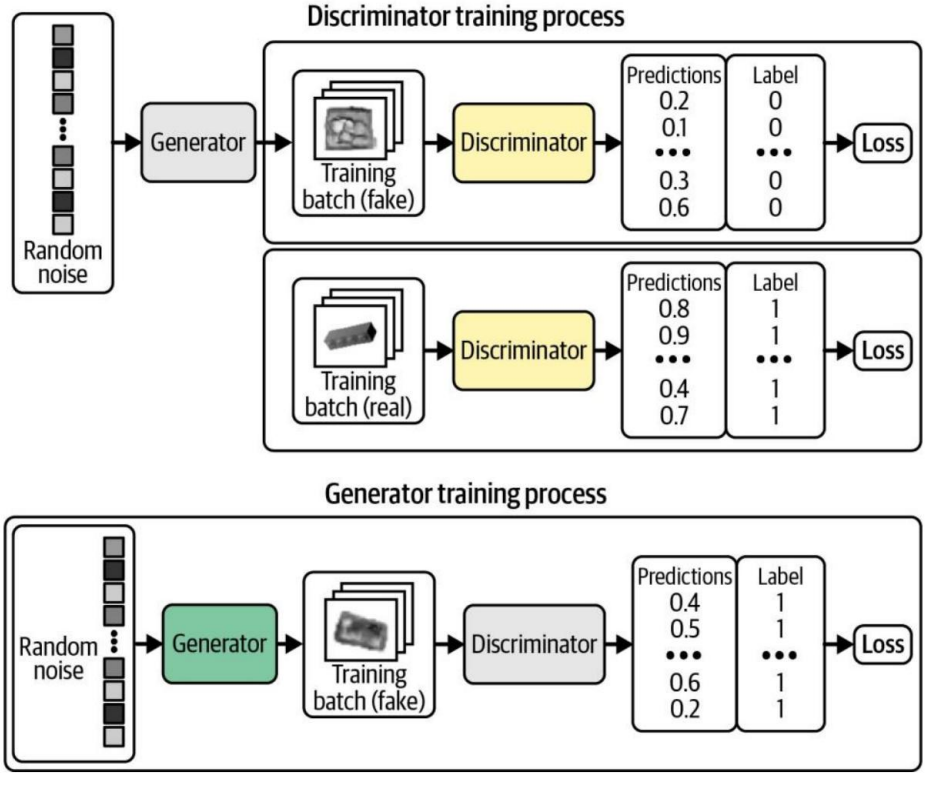

Figure 4-5. Entraînement du DCGAN : les cases grises indiquent que les poids sont gelés pendant l'entraînement

Keras nous offre la possibilité de créer une fonction train_step personnalisée pour implémenter cette logique. L'exemple 4-7 montre la classe de modèle DCGAN complète .

Exemple 4-7. Compilation du DCGAN

```
class DCGAN(models.Model) : discriminateur,                    def __init__(soi,
générateur, latent_dim) : super(DCGAN, self).__init__() self.discriminator
= discriminateur self.generator = générateur latent_dim

                                                                self.latent_dim =


    def compile(self, d_optimizer, g_optimizer) : super(DCGAN,
self).compile()
self.loss_fn = pertes.BinaryCrossentropy() self.d_optimizer =          ❶
            d_optimizer self.g_optimizer = g_optimizer
```

```
                self.d_loss_metric = metrics.Mean(name="d_loss")
self.g_loss_metric = metrics.Mean(name="g_loss")

        @propriété                  def metrics (self): return
[self.d_loss_metric, self.g_loss_metric]

        def train_step(self, real_images) :
                taille_lot =
tf.shape(real_images)[0]
random_latent_vectors = tf.random.normal( shape=(batch_size,
self.latent_dim)
)  ❷

                avec tf.GradientTape() comme gen_tape, tf.GradientTape() comme
disc_tape : généré_images = self.generator ( random_latent_vectors, formation = True

                        ❸
                    ) real_predictions = self.discriminator(real_images,
formation = Vrai)                ❹
                fake_predictions =
self.discriminator
( généré_images, formation = True
)  ❺

                real_labels = tf.ones_like(real_predictions) real_noisy_labels = real_labels
+ 0.1 * tf.random.uniform(
                        tf.shape (real_predictions)
                )
                fake_labels = tf.zeros_like (fake_predictions)
fake_noisy_labels = fake_labels - 0.1 * tf.random.uniform(

tf.shape(fake_predictions) )


                d_real_loss = self.loss_fn(real_noisy_labels,
real_predictions)                        d_fake_loss =
self.loss_fn(fake_noisy_labels, fake_predictions)

d_loss = (d_real_loss + d_fake_loss) / 2.0                ❻                g_loss =

self.loss_fn (real_labels, fake_predictions)                ❼

gradients_of_discriminator = disc_tape.gradient(
                d_loss, self.discriminator.trainable_variables
            )
            gradients_of_generator =
gen_tape.gradient ( g_loss, self.generator.trainable_variables
                                                                )
```

```
                self.d_optimizer.apply_gradients(
                    zip(gradients_of_discriminator,
discriminateur.trainable_variables)
                ❽
                ) self.g_optimizer.apply_gradients(
                    zip(gradients_of_generator,
générateur.trainable_variables)                                    )

                self.d_loss_metric.update_state(d_loss)
self.g_loss_metric.update_state(g_loss)
renvoie {m.name : m.result() pour m dans
auto.metrics}

dcgan = DCGAN(
    discriminateur=discriminateur, générateur=générateur, latent_dim=100 )


dcgan.compile(
    d_optimizer=optimizers.Adam(
        taux d'apprentissage = 0,0002, bêta_1 = 0,5, bêta_2 = 0,999
    ),
    g_optimizer=optimiseurs.Adam(
        taux d'apprentissage = 0,0002, bêta_1 = 0,5, bêta_2 = 0,999
    ),
) dcgan.fit(train,
époques = 300)
```

❶ La fonction de perte pour le générateur et le discriminateur est BinaryCrossentropy.

❷ Pour former le réseau, échantillonnez d'abord un lot de vecteurs à partir d'un standard multivarié distribution normale.

❸ Ensuite, transmettez-les via le générateur pour produire un lot d'images générées.

❹ Demandez maintenant au discriminateur de prédire la réalité du lot d'images réelles…

❺ …et le lot d'images générées.

❻ La perte du discriminateur est l'entropie croisée binaire moyenne sur les images réelles (avec l'étiquette 1) et les fausses images (avec l'étiquette 0).

❼ La perte du générateur est l'entropie croisée binaire entre les prédictions du discriminateur pour les images générées et une étiquette de 1.

❽ Mettez à jour les poids du discriminateur et du générateur séparément.

Le discriminateur et le générateur se battent constamment pour la domination, ce qui peut rendre le processus de formation DCGAN instable. Idéalement, le processus de formation trouvera un équilibre permettant au générateur d'apprendre des informations significatives du discriminateur et la qualité des images commencera à s'améliorer. Après suffisamment de temps, le discriminateur a tendance à finir par dominer, comme le montre la figure 4-6, mais cela ne pose peut-être pas de problème car le générateur a peut-être déjà appris à produire des images de qualité suffisamment élevée à ce stade.

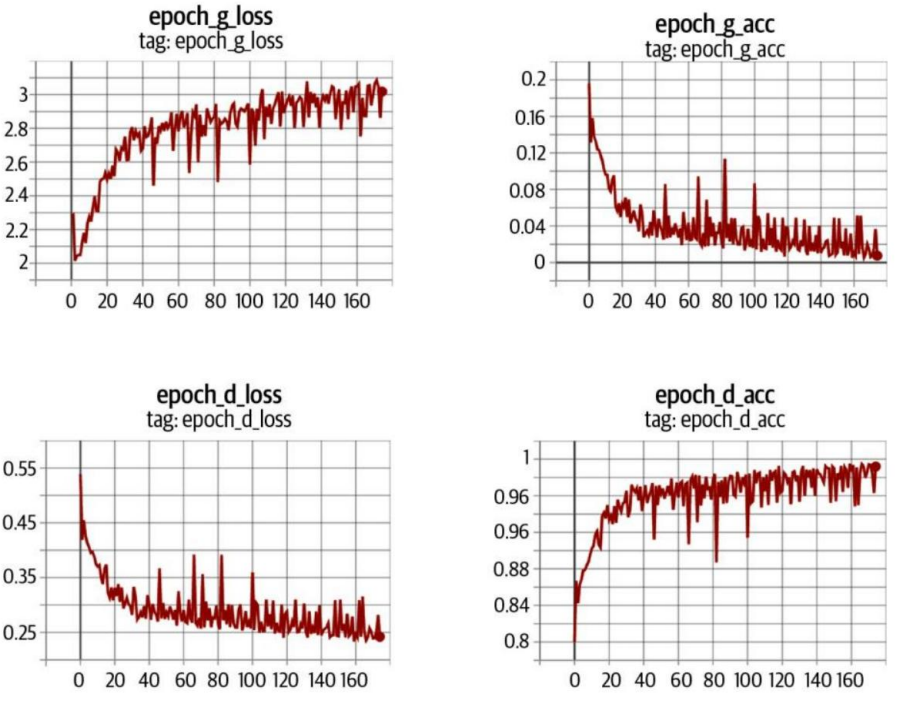

Figure 4-6. Perte et précision du discriminateur et du générateur pendant l'entraînement

Ajouter du bruit aux étiquettes

Une astuce utile lors de la formation des GAN consiste à ajouter une petite quantité de bruit aléatoire aux étiquettes de formation. Cela contribue à améliorer la stabilité du processus de formation et à affiner les images générées. Ce lissage des étiquettes agit comme un moyen d'apprivoiser le discriminateur, de sorte qu'il se voit confier une tâche plus difficile et ne domine pas le générateur.

Analyse de la DCGAN

En observant les images produites par le générateur à des époques spécifiques au cours de la formation (Figure 4-7), il est clair que le générateur devient de plus en plus apte à produire des images qui auraient pu être tirées de l'ensemble de formation.

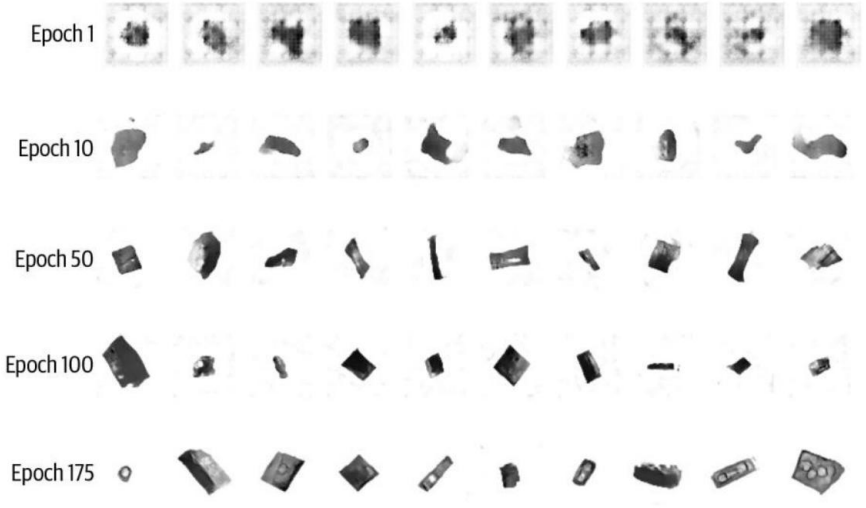

Figure 4-7. Sortie du générateur à des époques spécifiques pendant l'entraînement

Il est quelque peu miraculeux qu'un réseau neuronal soit capable de convertir un bruit aléatoire en quelque chose de significatif. Il convient de rappeler que nous n'avons fourni au modèle aucune fonctionnalité supplémentaire au-delà des pixels bruts, il doit donc élaborer des concepts de haut niveau tels que la façon de dessiner des ombres, des cuboïdes et des cercles entièrement par lui-même. Une autre exigence d'un modèle génératif réussi est qu'il ne reproduise pas uniquement les images de l'ensemble d'apprentissage. Pour tester cela, nous pouvons trouver l'image de l'ensemble d'entraînement qui es le plus proche d'un exemple généré particulier. Une bonne mesure de la distance est la distance L1, définie comme :

```
def compare_images(img1, img2) : return
np.mean(np.abs(img1 -
img2))
```

La figure 4-8 montre les observations les plus proches dans l'ensemble d'apprentissage pour une sélection d'images générées. Nous pouvons voir que même s'il existe un certain degré de similitude entre les images générées et l'ensemble d'entraînement, elles ne sont pas identiques. Cela montre que le générateur a compris ces fonctionnalités de haut niveau et peut générer des exemples distincts de ceux qu'il a déjà vus.

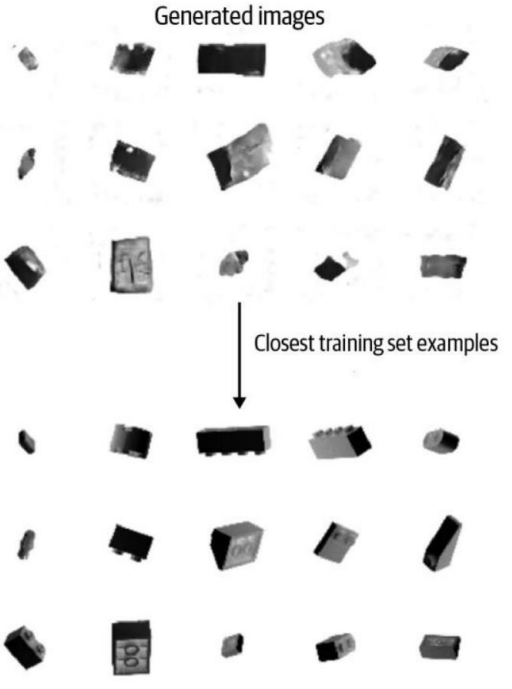

Figure 4-8. Correspondances les plus proches des images générées à partir de l'ensemble d'entraînement

Formation GAN : trucs et astuces

Si les GAN constituent une avancée majeure pour la modélisation générative, ils sont également notoirement difficiles à former. Nous explorerons certains des problèmes et défis les plus courants rencontrés lors de la formation des GAN dans cette section, ainsi que des solutions potentielles. Dans la section suivante, nous examinerons certains ajustements plus fondamentaux au cadre GAN que nous pouvons apporter pour remédier à bon nombre de ces problèmes.

Le discriminateur domine le générateur

Si le discriminateur devient trop fort, le signal de la fonction de perte devient trop faible pour entraîner des améliorations significatives dans le générateur. Dans le pire des cas, le discriminateur apprend parfaitement à séparer les images réelles des fausses images et les gradients disparaissent complètement, ce qui ne nécessite aucune formation, comme le montre la figure 4-9.

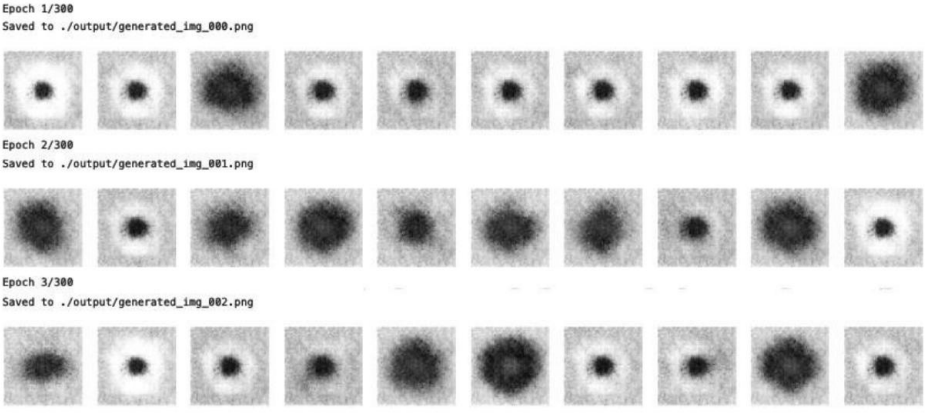

Figure 4-9. Exemple de sortie lorsque le discriminateur domine le générateur

Si vous constatez que la fonction de perte de votre discriminateur s'effondre, vous devez trouver des moyens d'affaiblir le discriminateur. Essayez les suggestions suivantes :

- Augmentez le paramètre de taux des couches Dropout dans le discriminateur à diminuer la quantité d'informations qui circulent à travers le réseau.
- Réduire le taux d'apprentissage du discriminateur.
- Réduire le nombre de filtres convolutifs dans le discriminateur.
- Ajoutez du bruit aux étiquettes lors de la formation du discriminateur.
- Retournez les étiquettes de certaines images au hasard lors de la formation du discriminateur.

Le générateur domine le discriminateur

Si le discriminateur n'est pas assez puissant, le générateur trouvera des moyens de tromper facilement le discriminateur avec un petit échantillon d'images presque identiques. C'est ce qu'on appelle l'effondrement des modes.

Par exemple, supposons que nous devions entraîner le générateur sur plusieurs lots sans mettre à jour le discriminateur entre les deux. Le générateur serait enclin à trouver une seule observation (également appelée mode) qui trompe toujours le discriminateur et commencerait à mapper chaque point de l'espace d'entrée latent sur cette image. De plus, les gradients de la fonction de perte s'effondreraient jusqu'à près de 0, elle ne serait donc pas en mesure de s'en remettre.
État.

Même si nous essayions ensuite de recycler le discriminateur pour l'empêcher d'être trompé par ce seul point, le générateur trouverait simplement un autre mode qui trompe le discriminateur, puisqu'il est déjà devenu insensible à son entrée et n'est donc pas incité à diversifier sa sortie.

L'effet de l'effondrement des modes est visible sur la figure 4-10.

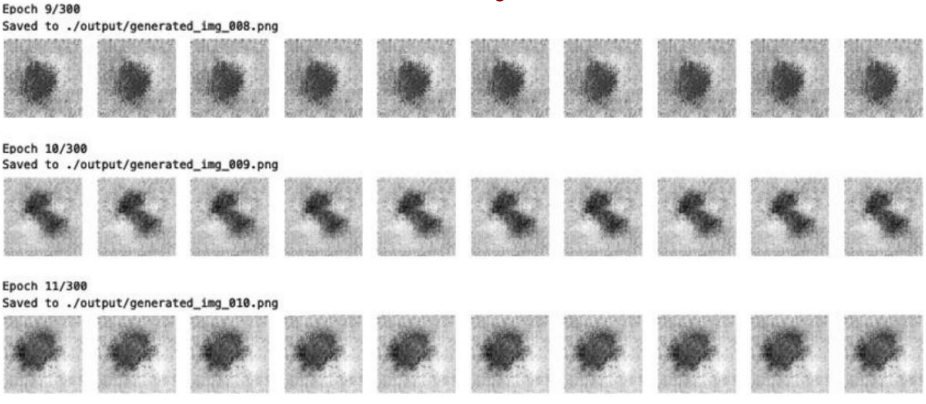

Figure 4-10. Exemple d'effondrement de mode lorsque le générateur domine le discriminateur

Si vous constatez que votre générateur souffre d'un effondrement de mode, vous pouvez essayer de renforcer le discriminateur en utilisant les suggestions opposées à celles répertoriées dans la section précédente. Vous pouvez également essayer de réduire le taux d'apprentissage des deux réseaux et d'augmenter la taille des lots.

Perte non informative

Puisque le modèle d'apprentissage profond est compilé pour minimiser la fonction de perte, il serait naturel de penser que plus la fonction de perte du générateur est petite, meilleure est la qualité des images produites. Cependant, étant donné que le générateur n'est évalué qu'en fonction du discriminateur actuel et que celui-ci s'améliore constamment, nous ne pouvons pas comparer la fonction de perte évaluée à différents moments du processus de formation. En effet, sur la figure 4-6, la fonction de perte du générateur augmente effectivement avec le temps, même si la qualité des images s'améliore nettement. Ce manque de corrélation entre la perte du générateur et la qualité de l'image rend parfois la formation GAN difficile à surveiller.

Hyperparamètres

Comme nous l'avons vu, même avec de simples GAN, il existe un grand nombre d'hyperparamètres à régler. Outre l'architecture globale du discriminateur et du générateur, il convient de prendre en compte les paramètres qui régissent la normalisation des lots, l'abandon, le taux d'apprentissage, les couches d'activation, les filtres convolutifs, la taille du noyau, la progression, la taille du lot et la taille de l'espace latent. Les GAN sont très sensibles à de très légers changements dans tous ces paramètres, et trouver un ensemble de paramètres qui fonctionne est souvent un cas d'essais et d'erreurs éclairés, plutôt que de suivre un ensemble de lignes directrices établies.

C'est pourquoi il est important de comprendre le fonctionnement interne du GAN et de savoir comment interpréter la fonction de perte, afin de pouvoir identifier des ajustements judicieux des hyperparamètres susceptibles d'améliorer la stabilité du modèle.

Relever les défis du GAN

Ces dernières années, plusieurs avancées clés ont considérablement amélioré la stabilité globale des modèles GAN et réduit la probabilité de certains des problèmes énumérés précédemment, tels que l'effondrement des modes.

Dans le reste de ce chapitre, nous examinerons le Wasserstein GAN avec Gradient
Penalty (WGAN-GP), qui apporte plusieurs ajustements clés au cadre GAN que nous avons exploré jusqu'à présent pour améliorer la stabilité et la qualité de la génération d'images.
processus.

Wasserstein GAN avec pénalité de dégradé (WGAN-GP)

Dans cette section, nous allons construire un WGAN-GP pour générer des visages à partir de l'ensemble de données CelebA que nous avons utilisé au chapitre 3.

Exécution du code pour cet exemple

Le code de cet exemple se trouve dans le notebook Jupyter situé à l' adresse notebooks/04_gan/02_wgan_gp/wgan_gp.ipynb dans le référentiel de livres.

Le code a été adapté de l'excellent tutoriel WGAN-GP créé par Aakash Kumar Nain, disponible sur le site de Keras.

Le Wasserstein GAN (WGAN), présenté dans un article de 2017 d'Arjovsky et al.4, a été l'une des premières grandes étapes vers la stabilisation de la formation GAN. Avec quelques changements, les auteurs ont pu montrer comment former des GAN qui possèdent les deux propriétés suivantes (citées dans l'article) :

- Une métrique de perte significative qui est en corrélation avec la convergence du générateur et qualité de l'échantillon
- Amélioration de la stabilité du processus d'optimisation

Plus précisément, l'article présente la fonction de perte de Wasserstein pour le discriminateur et le générateur. L'utilisation de cette fonction de perte au lieu de l'entropie croisée binaire entraîne une convergence plus stable du GAN.

Dans cette section, nous définirons la fonction de perte de Wasserstein, puis verrons quels autres changements nous devons apporter à l'architecture du modèle et au processus de formation pour intégrer notre nouvelle fonction de perte.

Vous pouvez trouver la classe de modèle complète dans le bloc-notes Jupyter situé au chapitre 05/wgangp/faces/train.ipynb dans le référentiel de livres.

Perte de Wasserstein

Rappelons d'abord la définition de la perte d'entropie croisée binaire, la fonction que nous utilisons actuellement pour entraîner le discriminateur et le générateur du GAN (équation 4-1).

Équation 4-1. Perte d'entropie croisée binaire

$$-\frac{1}{n}\sum_{i=1}^{n} y_i \log(p_i) + (1 - y_i)\log(1 - p_i)$$

Pour entraîner le discriminateur GAN D, nous calculons la perte en comparant les prédictions pour les images réelles $p_i = D(x_i)$ à la réponse $y_i = 1$ et les prédictions pour les images générées $p_i = D(G(z_i))$ à la réponse $y_i = 0$. Par conséquent, pour le discriminateur GAN, minimiser la fonction de perte peut être écrit comme indiqué dans l'équation 4-2. Équation 4-2. Minimisation des pertes du discriminateur GAN

$$\min_D -\left(\mathbb{E}_{x \sim p_X}[\log D(x)] + \mathbb{E}_{z \sim p_Z}[\log(1 - D(G(z)))]\right)$$

Pour entraîner le générateur GAN G, nous calculons la perte en comparant les prédictions pour les images générées $p_i = D(G(z_i))$ à la réponse $y_i = 1$. Par conséquent, pour le générateur GAN, minimiser la fonction de perte peut être écrit comme indiqué dans l'équation 4-3. Équation 4-3. Minimisation des pertes du générateur GAN

$$\min_G -\left(\mathbb{E}_{z \sim p_Z}[\log(D(G(z)))]\right)$$

Comparons maintenant cela à la fonction de perte de Wasserstein.

Premièrement, la perte de Wasserstein nécessite que nous utilisions $y_i = 1$ et $y_i = -1$ comme étiquettes, plutôt que

1 et 0. Nous supprimons également l'activation sigmoïde de la couche finale du discriminateur, de sorte que les prédictions pi ne soient plus contraintes de tomber dans la plage [0, 1] mais peuvent désormais être n'importe quel nombre dans la plage (−∞, ∞). Pour cette raison, le discriminateur d'un WGAN est généralement appelé un critique qui produit un score plutôt qu'une probabilité.

La fonction de perte de Wasserstein est définie comme suit :

$$-\frac{1}{n}\sum_{i=1}^{n} y_i p_i$$

Pour entraîner le critique WGAN D, nous calculons la perte en comparant les prédictions pour les images réelles pi = D xi à la réponse yi = 1 et les prédictions pour les images générées pi = DG zi à la réponse yi = −1. Par conséquent, pour le critique du WGAN, minimiser la fonction de perte peut s'écrire comme suit :

$$\min_D -\left(\mathbb{E}_{x \sim p_X}[D(x)] - \mathbb{E}_{z \sim p_Z}[D(G(z))] \right)$$

En d'autres termes, le critique du WGAN tente de maximiser la différence entre ses prédictions pour les images réelles et les images générées.

Pour entraîner le générateur WGAN, nous calculons la perte en comparant les prédictions des images générées pi = DG zi à la réponse yi = 1. Par conséquent, pour le générateur WGAN, la minimisation de la fonction de perte peut s'écrire comme suit :

$$\min_G -\left(\mathbb{E}_{z \sim p_Z}[D(G(z))] \right)$$

En d'autres termes, le générateur WGAN essaie de produire des images qui reçoivent la note la plus élevée possible par le critique (c'est-à-dire que le critique est trompé en lui faisant croire qu'elles sont réelles).

La contrainte Lipschitzienne

Cela peut vous surprendre que nous permettions désormais au critique de produire n'importe quel nombre dans la plage (−∞, ∞), plutôt que d'appliquer une fonction sigmoïde pour limiter la sortie à la plage habituelle [0, 1]. La perte de Wasserstein peut donc être très importante, ce qui est troublant : il faut généralement éviter les grands nombres dans les réseaux de neurones !

En fait, les auteurs de l'article du WGAN montrent que pour que la fonction de perte de Wasserstein fonctionne, nous devons également imposer une contrainte supplémentaire au critique. Concrètement, c'est

exige que la critique soit une fonction continue 1-Lipschitz. Examinons cela pour comprendre ce que cela signifie plus en détail.

La critique est une fonction D qui convertit une image en prédiction. Nous disons que cette fonction est 1-Lipschitz si elle satisfait l'inégalité suivante pour deux images d'entrée quelconques, x1 et x2 :

$$\left| D(x_1) - D(x_2) \right| \leq 1$$

Ici, $|x1 - x2|$ est la différence absolue moyenne au niveau des pixels entre deux images et $|D(x1) - D(x2)|$ est la différence absolue entre les prédictions critiques. Essentiellement, nous exigeons une limite à la vitesse à laquelle les prédictions du critique peuvent changer entre deux images (c'est-à-dire que la valeur absolue du gradient doit être au maximum de 1 partout). Nous pouvons voir cela appliqué à une fonction 1D continue de Lipschitz dans la figure 4-11 : à aucun moment la ligne n'entre dans le cône, quel que soit l'endroit où vous placez le cône sur la ligne. En d'autres termes, il existe une limite à la vitesse à laquelle la ligne peut monter ou descendre à tout moment.

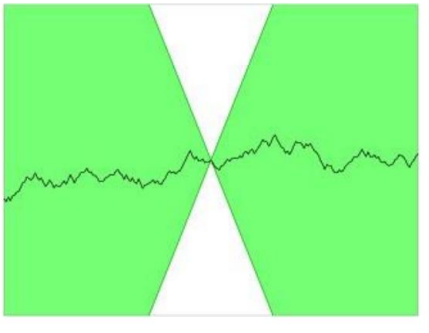

Figure 4-11. Une fonction continue Lipschitzienne (source : Wikipédia)

Pour ceux qui souhaitent approfondir la justification mathématique expliquant pourquoi la perte de Wasserstein ne fonctionne que lorsque cette contrainte est appliquée, Jonathan Hui propose une excellente explication.

Appliquer la contrainte Lipschitz

Dans l'article original du WGAN, les auteurs montrent comment il est possible d'appliquer la contrainte Lipschitzienne en limitant le poids de la critique dans une petite fourchette, [–0,01, 0,01], après chaque lot d'entraînement.

L'une des critiques de cette approche est que la capacité d'apprentissage du critique est fortement diminuée, puisque l'on réduit ses poids. En fait, même dans le WGAN original

Dans l'article, les auteurs écrivent : « La réduction du poids est clairement un moyen terrible d'appliquer une contrainte Lipschitz. » Une critique sévère est essentielle au succès d'un WGAN, car sans gradients précis, le générateur ne peut pas apprendre à adapter ses poids pour produire de meilleurs échantillons.

Par conséquent, d'autres chercheurs ont recherché des moyens alternatifs pour appliquer la contrainte de Lipschitz et améliorer la capacité du WGAN à apprendre des fonctionnalités complexes. L'une de ces méthodes est le Wasserstein GAN avec gradient pénalité.

Dans l'article présentant cette variante5, les auteurs montrent comment la contrainte de Lipschitz peut être appliquée directement en incluant un terme de pénalité de gradient dans la fonction de perte pour le critique qui pénalise le modèle si la norme de gradient s'écarte de 1. Il en résulte un effet bien plus important. processus de formation stable.

Dans la section suivante, nous verrons comment intégrer ce terme supplémentaire dans la fonction de perte pour notre critique.

La perte de pénalité graduelle

La figure 4-12 est un diagramme du processus de formation du critique d'un WGAN-GP. Si nous comparons cela au processus original de formation du discriminateur de la figure 4-5, nous pouvons voir que l'ajout clé est la perte de pénalité de gradient incluse dans la fonction de perte globale, aux côtés de la perte de Wasserstein des images réelles et fausses.

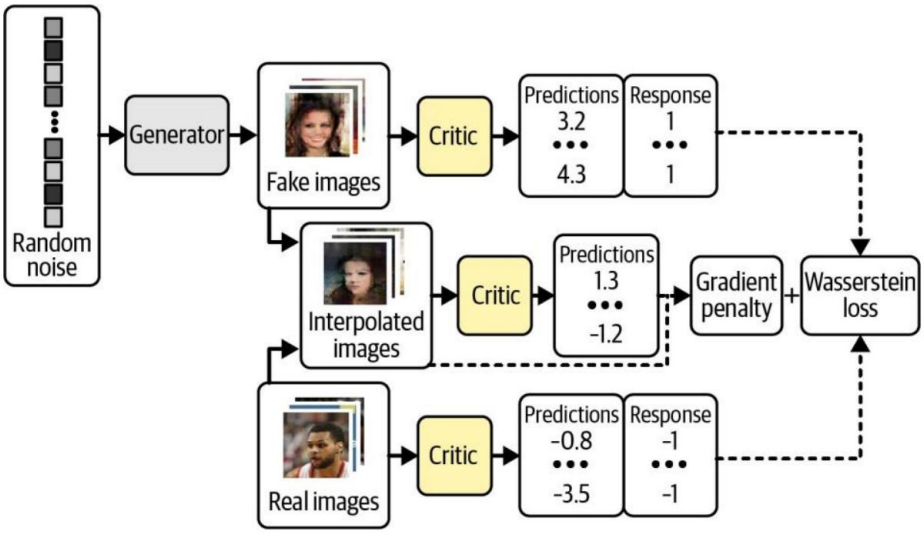

Figure 4-12. Le processus de formation des critiques WGAN-GP

La perte de pénalité de gradient mesure la différence au carré entre la norme du gradient des prédictions par rapport aux images d'entrée et 1. Le modèle va

être naturellement enclin à trouver des poids qui garantissent que le terme de pénalité de gradient est minimisé, encourageant ainsi le modèle à se conformer à la contrainte de Lipschitz.

Il est impossible de calculer ce gradient partout pendant le processus de formation, c'est pourquoi le WGAN-GP évalue le gradient en seulement quelques points. Pour garantir un mélange équilibré, nous utilisons un ensemble d'images interpolées situées à des points choisis au hasard.
le long de lignes reliant le lot d'images réelles au lot de fausses images par paires, comme le montre la figure 4-13.

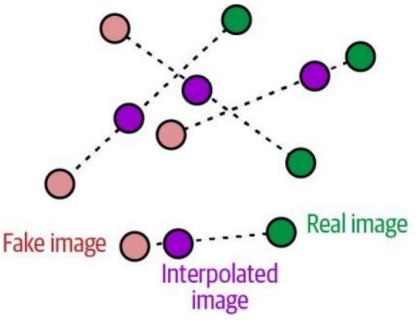

Figure 4-13. Interpolation entre les images

Dans l'exemple 4-8, nous montrons comment la pénalité de gradient est calculée dans le code. Exemple

4-8. La fonction de perte de pénalité de gradient

```
def gradient_penalty (self, batch_size, real_images, fake_images) : alpha =

tf.random.normal([batch_size, 1, 1, 1], 0.0, 1.0) diff = fake_images - real_images ❶
interpolées =
images_réelles + alpha * diff ❷

        avec tf.GradientTape() comme gp_tape : gp_tape.watch
(interpolé)
                pred = self.critic(interpolé, ❸ formation=Vrai)

        grads = gp_tape.gradient(pred, [interpolated]) ❹
[0] norme =
tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2, 3]))
❺ gp = tf.reduce_mean((norme - 1.0) ** 2) ❻ retour
```

❶ Chaque image du lot reçoit un nombre aléatoire, entre 0 et 1, stocké comme vecteur alpha.

❷ Un ensemble d'images interpolées est calculé.

❸ Le critique est invité à noter chacune de ces images interpolées.

❹ Le gradient des prédictions est calculé par rapport aux images d'entrée.

❺ La norme L2 de ce vecteur est calculée.

❻ La fonction renvoie la distance quadratique moyenne entre la norme L2 et 1.

Formation du WGAN-GP

L'un des principaux avantages de l'utilisation de la fonction de perte de Wasserstein est que nous n'avons plus à nous soucier de l'équilibre entre la formation du critique et celle du générateur. En fait, lors de l'utilisation de la fonction de perte de Wasserstein, le critique doit être entraîné à la convergence avant de mettre à jour le générateur. assurez-vous que les dégradés pour la mise à jour du générateur sont exacts. Cela contraste avec un GAN standard, où il est important de ne pas laisser le discriminateur devenir trop fort.

Par conséquent, avec les GAN de Wasserstein, nous pouvons simplement former le critique plusieurs fois entre les mises à jour du générateur, pour nous assurer qu'il est proche de la convergence. Un ratio typique utilisé est de trois à cinq mises à jour critiques par mise à jour du générateur.

Nous avons maintenant introduit les deux concepts clés derrière le WGAN-GP : la perte de Wasserstein et le terme de pénalité de gradient inclus dans la fonction de perte critique. L'étape de formation du modèle WGAN qui intègre toutes ces idées est présentée dans l'exemple 4-9.

Exemple 4-9. Formation du WGAN-GP

```
def train_step (self, real_images): batch_size =

tf.shape(real_images)[0]

        pour i dans la plage (3) :          ❶
random_latent_vectors =

tf.random.normal( shape=(batch_size, self.latent_dim) )

                avec tf.GradientTape() comme bande : fake_images
= self.generator(
```

```
                    random_latent_vectors, formation = True
                )
                fake_predictions = self.critic(fake_images, formation =
Vrai)                    real_predictions = self.critic(real_images, formation
= Vrai)

                c_wass_loss = tf.reduce_mean(fake_predictions) -
tf.reduce_mean(                                    real_predictions
                ) ❷
                ) c_gp = self.gradient_penalty(
                    batch_size, real_images, fake_images
                ) ❸
                c_loss = c_wass_loss + c_gp                * self.gp_weight  ❹

c_gradient = bande.gradient(c_loss,

self.critic.trainable_variables)

self.c_optimizer.apply_gradients(

                zip(c_gradient,
self.critic.trainable_variables)                                    ) ❺

        random_latent_vectors = tf.random.normal( shape=(batch_size, self.latent_dim)
avec                                                                )
tf.GradientTape() en tant que bande : fake_images =
self.generator (random_latent_vectors, training = True) fake_predictions = self.critic
(fake_images, training = True) g_loss = -tf.reduce_mean (fake_predictions)
                                                                    ❻


        gen_gradient = bande.gradient(g_loss,
self.generator.trainable_variables)
        self.g_optimizer.apply_gradients(
            zip (gen_gradient,
self.generator.trainable_variables)                                 ) ❼

        self.c_loss_metric.update_state(c_loss)
self.c_wass_loss_metric.update_state(c_wass_loss)
        self.c_gp_metric.update_state(c_gp)

self.g_loss_metric.update_state(g_loss)

renvoie {m.name : m.result() pour m dans
```

❶ self.metrics} Effectuer trois mises à jour critiques.

❷ Calculez la perte de Wasserstein pour le critique – la différence entre la prédiction moyenne des fausses images et des images réelles. Calculez le terme de pénalité de

❸ gradient (voir exemple 4-8).

❹

❺

La fonction de perte critique est une somme pondérée de la perte de Wasserstein et de la pénalité de gradient.

Mettre à jour les poids du critique.

Calculez la perte de Wasserstein pour le générateur.

Mettez à jour les poids du générateur.

Normalisation par lots dans un WGAN-GP

Une dernière considération à noter avant de former un WGANGP est que la normalisation par lots ne doit pas être utilisée dans le critique. C'est car la normalisation par lots crée une corrélation entre les images du même lot, ce qui rend la perte de pénalité de gradient moins efficace.
Des expériences ont montré que les WGAN-GP peuvent toujours produire d'excellents résultats même sans normalisation par lots chez le critique.

Nous avons maintenant couvert toutes les différences clés entre un GAN standard et un WGAN-GP. Récapituler:

- Un WGAN-GP utilise la perte de Wasserstein.
- Le WGAN-GP est formé en utilisant des étiquettes de 1 pour réel et –1 pour faux.
- Il n'y a pas d'activation sigmoïde dans la couche finale du critique.
- Inclure un terme de pénalité de gradient dans la fonction de perte pour le critique.
- Former le critique plusieurs fois pour chaque mise à jour du générateur.
- Il n'y a pas de couches de normalisation par lots dans le critique.

Analyse du WGAN-GP

Jetons un coup d'œil à quelques exemples de sorties du générateur, après 25 époques d'entraînement (Figure 4-14).

Figure 4-14. Exemples de visages WGAN-GP

Le modèle a appris les attributs importants de haut niveau d'un visage et il n'y a aucun signe d'effondrement du mode.

Nous pouvons également voir comment les fonctions de perte du modèle évoluent au fil du temps (Figure 4-15) : les fonctions de perte du critique et du générateur sont très stables et convergentes.

Si nous comparons la sortie WGAN-GP à la sortie VAE du chapitre précédent, nous pouvons voir que les images GAN sont généralement plus nettes, en particulier la définition entre les cheveux et l'arrière-plan. Cela est vrai en général ; Les VAE ont tendance à produire des images plus douces qui brouillent les limites des couleurs, tandis que les GAN sont connus pour produire des images plus nettes et mieux définies.

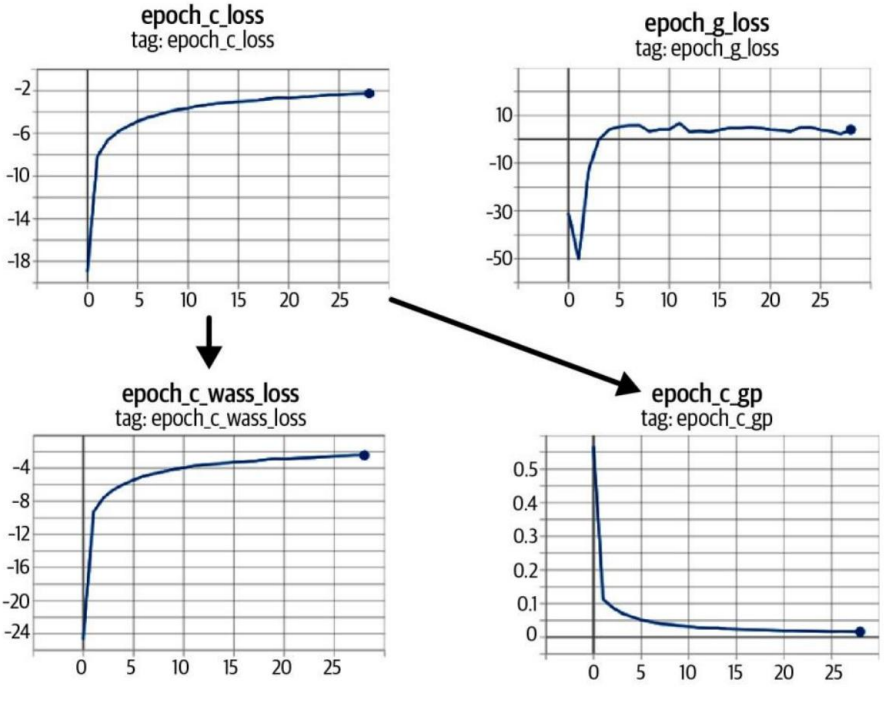

Figure 4-15. Courbes de perte WGAN-GP : la perte critique (epoch_c_loss) se décompose en perte de Wasserstein (epoch_c_wass) et perte de pénalité de gradient (epoch_c_gp)

Il est également vrai que les GAN sont généralement plus difficiles à former que les VAE et mettent plus de temps à atteindre une qualité satisfaisante. Cependant, de nombreux modèles génératifs de pointe sont aujourd'hui basés sur le GAN, car les récompenses de la formation de GAN à grande échelle sur des GPU sur une période plus longue sont significatives.

GAN conditionnel (CGAN)

Jusqu'à présent dans ce chapitre, nous avons construit des GAN capables de générer des images réalistes à partir d'un ensemble de formation donné. Cependant, nous n'avons pas été en mesure de contrôler le type d'image que nous souhaitons générer, par exemple un visage masculin ou féminin, ou une grande ou petite brique. Nous pouvons échantillonner un point aléatoire dans l'espace latent, mais nous n'avons pas la capacité de comprendre facilement quel type d'image sera produit compte tenu du choix de la variable latente.

Dans la dernière partie de ce chapitre, nous porterons notre attention sur la construction d'un GAN dont nous sommes capables de contrôler la sortie – un GAN dit conditionnel. Cette idée, introduite pour la première fois dans « Conditional Generative Adversarial Nets » par Mirza et Osindero en 20146, est une extension relativement simple de l'architecture GAN.

Exécution du code pour cet exemple

Le code de cet exemple se trouve dans le notebook Jupyter situé à notebooks/04_gan/03_cgan/cgan.ipynb dans le référentiel de livres.

Le code a été adapté de l'excellent tutoriel CGAN créé par Sayak Paul, disponible sur le site de Keras.

Architecture CGAN

Dans cet exemple, nous conditionnerons notre CGAN sur l' attribut cheveux blonds de l'ensemble de données faces. Autrement dit, nous pourrons préciser explicitement si nous voulons ou non générer une image avec des cheveux blonds. Cette étiquette est fournie dans le cadre de l'ensemble de données CelebA. L'architecture CGAN de haut niveau est illustrée à la Figure 4-16.

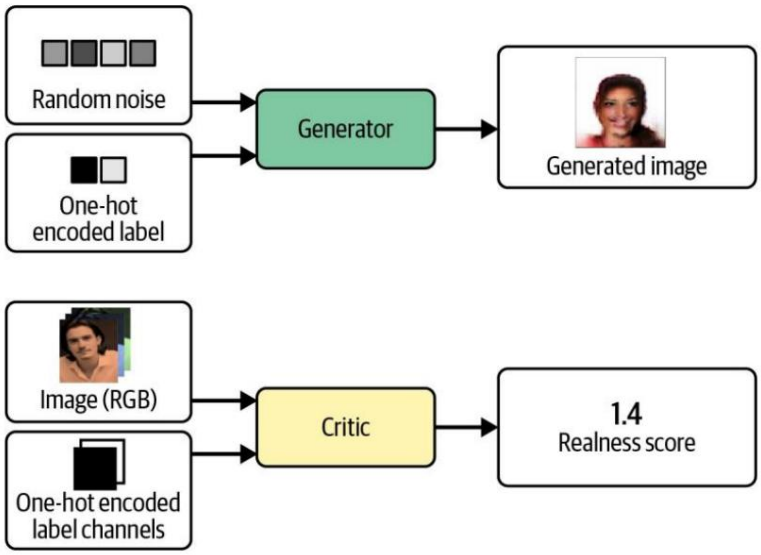

Figure 4-16. Entrées et sorties du générateur et du critique dans un CGAN

La principale différence entre un GAN standard et un CGAN est que dans un CGAN, nous transmettons des informations supplémentaires au générateur et au critique concernant l'étiquette. Dans le générateur, ceci est simplement ajouté à l'échantillon d'espace latent sous la forme d'un vecteur codé à chaud. Dans la critique, nous ajoutons les informations d'étiquette sous forme de canaux supplémentaires à l'image RVB. Nous faisons cela en répétant le vecteur codé à chaud pour remplir la même forme que les images d'entrée.

Les CGAN fonctionnent parce que le critique a désormais accès à des informations supplémentaires concernant le contenu de l'image, le générateur doit donc s'assurer que sa sortie est conforme à l'étiquette fournie, afin de continuer à tromper le critique. Si le générateur produisait parfaitement

images que

En désaccord avec l'étiquette de l'image, le critique serait en mesure de dire qu'elles étaient fausses simplement parce que les images et les étiquettes ne correspondaient pas.

Dans notre exemple, notre étiquette codée à chaud aura une longueur de 2, car il existe deux classes (Blonde et Not Blond). Cependant, vous pouvez avoir autant d'étiquettes que vous le souhaitez. Par exemple, vous pouvez entraîner un CGAN sur l'ensemble de données Fashion-MNIST pour générer l'un des 10 articles de mode différents, en incorporant un vecteur d'étiquette codé à chaud de longueur 10 dans le fichier. entrée du générateur et 10 canaux d'étiquettes codés à chaud supplémentaires dans l'entrée du critique.

Le seul changement que nous devons apporter à l'architecture est de concaténer les informations d'étiquette aux entrées existantes du générateur et du critique, comme le montre l' exemple 4-10.

Exemple 4-10. Couches d'entrée dans le CGAN

```
critique_input = layer.Input(shape=(64, 64, 3)) label_input =                    ❶
calques.Input(shape=(64, 64, 2))
x = layer.Concatenate(axis = -1)([critic_input, label_input])

... generateur_input = layer.Input(shape=(32,)) label_input = layer.Input(shape=(2,))  ❷

x = layer.Concatenate(axis = -1)([generator_input, label_input])
x = calques.Reshape((1,1, 34))(x) ...
```

❶ Les canaux d'image et les canaux d'étiquettes sont transmis séparément au critique et concaténés.

❷ Le vecteur latent et les classes d'étiquettes sont transmis séparément au générateur et concaténés avant d'être remodelés.

Former le CGAN

Nous devons également apporter quelques modifications au train_step du CGAN pour correspondre aux nouveaux formats d'entrée du générateur et critique, comme le montre l' exemple 4-11. Exemple 4-11. Le train_step du CGAN

```
def train_step (self, données): one_hot_labels          images_réelles,
= données                                                  ❶

    image_one_hot_labels = one_hot_labels[:, Aucun, Aucun, :] image_one_hot_labels = tf.repeat(  ❷

        image_one_hot_labels, répétitions = 64, axe = 1
    )
    image_one_hot_labels = tf.repeat(
        image_one_hot_labels, répétitions = 64, axe = 2                    )
batch_size = tf.shape(real_images)[0]

    pour i dans la plage (self.critic_steps) :
random_latent_vectors = tf.random.normal( shape=(batch_size,
self.latent_dim)                                                  )

        avec tf.GradientTape() comme bande : self.generator(          fausses_images =

                [random_latent_vectors, one_hot_labels], formation = True
            )  ❸

            fake_predictions = self.critic(
                [fake_images, image_one_hot_labels], formation = True
                ❹
            ) real_predictions = self.critic(
                [real_images, image_one_hot_labels], formation = True
            )
```

```
c_wass_loss = tf.reduce_mean(fake_predictions) - tf.reduce_mean(
                real_predictions
            )
            c_gp = self.gradient_penalty(
                batch_size, real_images, fake_images, image_one_hot_labels
                ❺
            ) c_loss = c_wass_loss + c_gp          * self.gp_weight

        c_gradient = tape.gradient(c_loss, self.critic.trainable_variables)
        self.c_optimizer.apply_gradients(
            zip(c_gradient, self.critic.trainable_variables)                    )

    random_latent_vectors = tf.random.normal( shape=(batch_size,
self.latent_dim)                                                  )
```

```
        avec tf.GradientTape() comme bande : self.generator(                fausses_images =
                    [random_latent_vectors, one_hot_labels], formation = True
                ❻
            ) fake_predictions = self.critic(
                    [fake_images, image_one_hot_labels], formation = True
            )
            g_loss = -tf.reduce_mean (fake_predictions)                                gen_gradient =
tape.gradient(g_loss, self.generator.trainable_variables)
```



```
        self.g_optimizer.apply_gradients(
            zip (gen_gradient, self.generator.trainable_variables)                    )
```

❶ Les images et les étiquettes sont décompressées à partir des données d'entrée.

❷ Les vecteurs codés à chaud sont étendus en images codées à chaud qui ont la même taille spatiale que les images d'entrée (64 × 64).

❸ Le générateur est maintenant alimenté par une liste de deux entrées : les vecteurs latents aléatoires et les vecteurs d'étiquettes codés à chaud.

❹ Le critique est désormais alimenté par une liste de deux entrées : les images fausses/réelles et les canaux d'étiquettes codés à chaud.

❺ La fonction de pénalité de gradient nécessite également que les canaux d'étiquettes codés à chaud soient transmis lors de l'utilisation du critique.

❻ Les modifications apportées à l'étape de formation critique s'appliquent également à l'étape de formation du générateur.

Analyse du CGAN

Nous pouvons contrôler la sortie CGAN en passant une étiquette particulière codée à chaud dans l'entrée du générateur.
Par exemple, pour générer un visage avec des cheveux non blonds, on passe le vecteur [1, 0]. Pour générer un visage aux cheveux blonds, on passe le vecteur
[0, 1].

La sortie du CGAN est visible dans la figure 4-17. Ici, nous conservons les mêmes vecteurs latents aléatoires dans tous les exemples et modifions uniquement le vecteur d'étiquette conditionnel. Il est clair que le CGAN a appris à utiliser le vecteur d'étiquette pour contrôler uniquement l'attribut couleur des cheveux des images. Il est impressionnant que le reste de l'image change à peine : c'est la preuve que les GAN sont capables d'organiser les points dans l'espace latent de telle manière que les caractéristiques individuelles puissent être découplées les unes des autres.

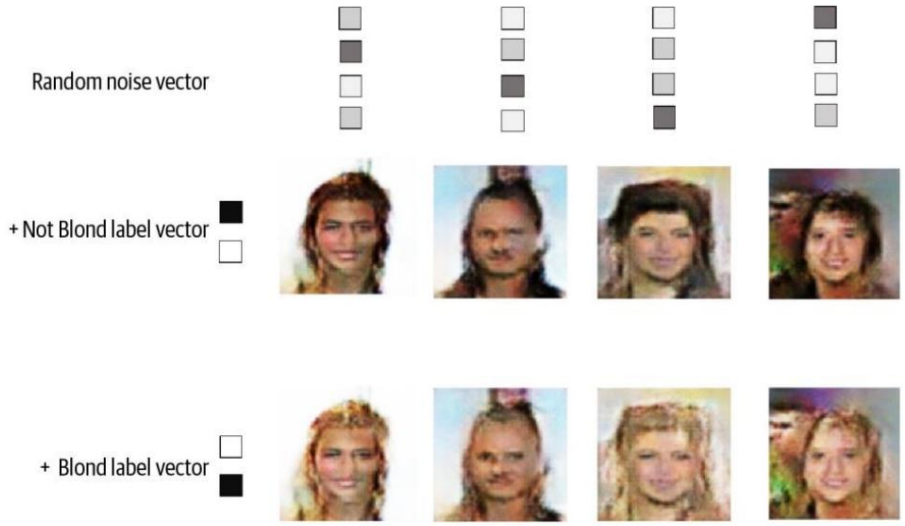

Figure 4-17. Sortie du CGAN lorsque les vecteurs Blond et Not Blond sont ajoutés à l'échantillon latent

Si des étiquettes sont disponibles pour votre ensemble de données, c'est généralement une bonne idée de les inclure en entrée de votre GAN même si vous n'avez pas nécessairement besoin de conditionner la sortie générée sur l'étiquette, car elles ont tendance à améliorer la qualité des images générées. Vous pouvez considérer les étiquettes comme une simple extension hautement informative de l'entrée de pixels.

Résumé

Dans ce chapitre, nous avons exploré trois modèles de réseaux contradictoires génératifs (GAN) : le GAN à convolution profonde (DCGAN), le GAN Wasserstein plus sophistiqué avec pénalité de gradient (WGAN-GP) et le GAN conditionnel (CGAN).

Tous les GAN sont caractérisés par une architecture générateur contre discriminateur (ou critique), le discriminateur essayant de « faire la différence » entre les images réelles et fausses et le générateur visant à tromper le discriminateur. En équilibrant la manière dont ces deux adversaires sont entraînés, le générateur GAN peut progressivement apprendre à produire des observations similaires à celles de l'ensemble d'entraînement.

Nous avons d'abord vu comment entraîner un DCGAN à générer des images de briques jouets. Il a pu apprendre à représenter de manière réaliste des objets 3D sous forme d'images, y compris des représentations précises de l'ombre, de la forme et de la texture. Nous avons également exploré les différentes manières dont la formation GAN peut échouer, notamment l'effondrement des modes et la disparition des gradients.

Nous avons ensuite exploré comment la fonction de perte de Wasserstein a résolu bon nombre de ces problèmes et rendu la formation GAN plus prévisible et plus fiable. Le WGAN-GP place l'exigence de 1-Lipschitz au cœur du processus de formation en incluant un terme dans la fonction de perte pour tirer la norme de gradient vers 1.

Nous avons appliqué le WGAN-GP au problème de la génération de visages et avons vu comment, en choisissant simplement des points dans une distribution normale standard, nous pouvons générer de nouveaux visages. Ce processus d'échantillonnage est très similaire à un VAE, bien que les visages produits par un GAN soient assez différents : souvent plus nets, avec une plus grande distinction entre les différentes parties de l'image.

Enfin, nous avons construit un CGAN qui nous a permis de contrôler le type d'image générée. Cela fonctionne en transmettant l'étiquette en entrée au critique et au générateur, donnant ainsi au réseau les informations supplémentaires dont il a besoin pour conditionner la sortie générée sur une étiquette donnée.

Dans l'ensemble, nous avons vu à quel point le framework GAN est extrêmement flexible et capable de s'adapter à de nombreux domaines problématiques intéressants. En particulier, les GAN ont permis des progrès significatifs dans le domaine de la génération d'images avec de nombreuses extensions intéressantes du cadre sous-jacent, comme nous le verrons au chapitre 10.

Dans le chapitre suivant, nous explorerons une autre famille de modèles génératifs idéale pour la modélisation

données séquentielles : modèles autorégressifs. Les références

1. Ian J. Goodfellow et al., « Generative Adversarial Nets », 10 juin 2014, https://arxiv.org/abs/1406.2661

2. Alec Radford et al., « Apprentissage de représentation non supervisé avec des réseaux contradictoires génératifs à convolution profonde », 7 janvier 2016, https://arxiv.org/abs/1511.06434 .

3. Augustus Odena et al., « Déconvolution et artefacts en damier », 17 octobre 2016, https://distill.pub/2016/deconv-checkerboard.

4. Martin Arjovsky et al., « Wasserstein GAN », 26 janvier 2017, https://arxiv.org/abs/1701.07875 .

5. Ishaan Gulrajani et al., « Formation améliorée des GAN de Wasserstein », 31 mars 2017, https://arxiv.org/abs/1704.00028.

6. Mehdi Mirza et Simon Osindero, «Conditional Generative Adversarial Nets», 6 novembre 2014, https://arxiv.org/abs/1411.1784.