PART 2 TensorFlow

4. Workshop 2 : CIFAR10 資料集 - GAN & CNN 影像處理進階

- < CIFAR10 > : Intro to Generative Adversarial Networks (GAN)
- < CIFAR10 > : Classifying Images with CNNs)

< CIFAR10 > : Intro to Generative Adversarial Networks (GAN)

> **[Reference] :**
>
> - FRANÇOIS CHOLLET, **Deep Learning with Python**, Chapter 8, Section 5, Manning, 2018. (https://tanthiamhuat.files.wordpress.com/2018/03/deeplearningwithpython.pdf (https://tanthiamhuat.files.wordpress.com/2018/03/deeplearningwithpython.pdf))
>
> - Tom Hope, Yehezkel S. Resheff, Itay Lieder, "**Learning TensorFlow - A Guide to building Deep Learning Systems**", `Chapters 5` , O'Reilly (2017) (pdf) https://goo.gl/iEmehh (https://goo.gl/iEmehh)
>   - `[ Code ]` : https://github.com/gigwegbe/Learning-TensorFlow (https://github.com/gigwegbe/Learning-TensorFlow)
>
> - **CIFAR-10 and CIFAR-100 datasets**, https://www.cs.toronto.edu/~kriz/cifar.html (https://www.cs.toronto.edu/~kriz/cifar.html)

In [1]:

```python
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[1]:

```
'2.0.8'
```

[GAN] :

-- a forger network and an expert network, each being trained to best the other.

As such, a GAN is made of two parts:

- `Generator network` — Takes as input a random vector (a random point in the latent space), and decodes it into a synthetic image

- `Discriminator network (or adversary)` — Takes as input an image (real or synthetic), and predicts whether the image came from the training set or was created by the generator network.

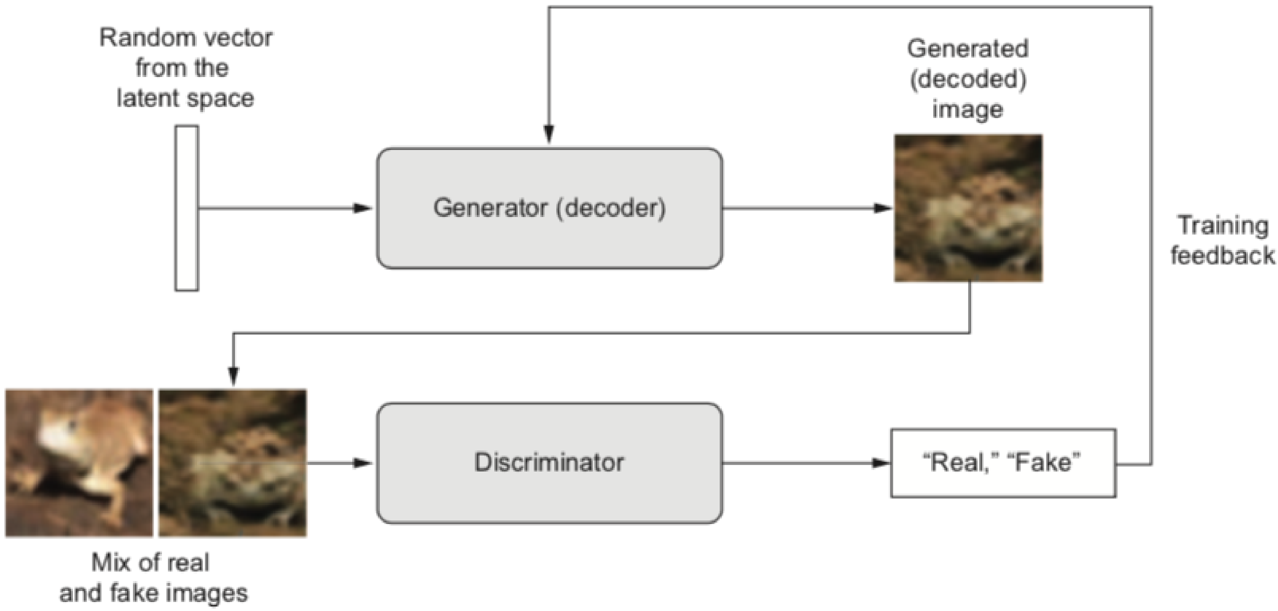


Figure 8.15 A generator transforms random latent vectors into images, and a discriminator seeks to tell real images from generated ones. The generator is trained to fool the discriminator.

A schematic GAN implementation

In what follows, we explain how to implement a GAN in Keras, in its barest form -- since GANs are quite advanced, diving deeply into the technical details would be out of scope for us. Our specific implementation will be a deep convolutional GAN, or DCGAN: a GAN where the generator and discriminator are deep convnets. In particular, it leverages a `Conv2DTranspose` layer for image upsampling in the generator.

We will **train our GAN on images from CIFAR10, a dataset of 50,000 32x32 RGB images belong to 10 classes (5,000 images per class).** To make things even easier, we will only use images belonging to the class "frog".

Schematically, our GAN looks like this:

- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(32, 32, 3)`.
- A `discriminator` network maps images of shape (32, 32, 3) to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, as we would train any regular image classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that, at every step, we move the weights of the generator in a direction that will make the discriminator more likely to classify as "real" the images decoded by the generator. I.e. we train the generator to fool the discriminator.

A schematic GAN implementation

In what follows, we explain how to implement a GAN in Keras, in its barest form -- since GANs are quite advanced, diving deeply into the technical details would be out of scope for us. Our specific implementation will be a deep convolutional GAN, or DCGAN: a GAN where the generator and discriminator are deep convnets. In particular, it leverages a `Conv2DTranspose` layer for image upsampling in the generator.

We will train our GAN on images from CIFAR10, a dataset of 50,000 32x32 RGB images belong to 10 classes (5,000 images per class). To make things even easier, we will only use images belonging to the class "frog".

Schematically, our GAN looks like this:

- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(32, 32, 3)`.
- A `discriminator` network maps images of shape (32, 32, 3) to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, as we would train any regular image classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that, at every step, we move the weights of the generator in a direction that will make the discriminator more likely to classify as "real" the images decoded by the generator. I.e. we train the generator to fool the discriminator.

The generator

First, we develop a `generator` model, which turns a vector (from the latent space -- during training it will sampled at random) into a candidate image. One of the many issues that commonly arise with GANs is that the generator gets stuck with generated images that look like noise. A possible solution is to use dropout on both the discriminator and generator.

In [1]:

```python
import keras
from keras import layers
import numpy as np

latent_dim = 32
height = 32
width = 32
channels = 3

generator_input = keras.Input(shape=(latent_dim,))

# First, transform the input into a 16x16 128-channels feature map
x = layers.Dense(128 * 16 * 16)(generator_input)
x = layers.LeakyReLU()(x)
x = layers.Reshape((16, 16, 128))(x)

# Then, add a convolution layer
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

# Upsample to 32x32
x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU()(x)

# Few more conv layers
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

# Produce a 32x32 1-channel feature map
x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x)
generator = keras.models.Model(generator_input, x)
generator.summary()
```

```
Using TensorFlow backend.

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 32768)             1081344
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32768)             0
_________________________________________________________________
reshape_1 (Reshape)          (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 256)       819456
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 32, 32, 256)       1048832
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 256)       1638656
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 32, 32, 256)       0
```

```
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 256)       1638656
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 32, 32, 3)         37635
=================================================================
Total params: 6,264,579
Trainable params: 6,264,579
Non-trainable params: 0
_________________________________________________________________
```

The discriminator

Then, we develop a `discriminator` model, that takes as input a candidate image (real or synthetic) and classifies it into one of two classes, either "generated image" or "real image that comes from the training set".

In [2]:

```
discriminator_input = layers.Input(shape=(height, width, channels))
x = layers.Conv2D(128, 3)(discriminator_input)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Flatten()(x)

# One dropout layer - important trick!
x = layers.Dropout(0.4)(x)

# Classification layer
x = layers.Dense(1, activation='sigmoid')(x)

discriminator = keras.models.Model(discriminator_input, x)
discriminator.summary()

# To stabilize training, we use learning rate decay
# and gradient clipping (by value) in the optimizer.
discriminator_optimizer = keras.optimizers.RMSprop(lr=0.0008, clipvalue=1.0, decay=1e-
discriminator.compile(optimizer=discriminator_optimizer, loss='binary_crossentropy')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_2 (InputLayer)         (None, 32, 32, 3)         0         
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 30, 30, 128)       3584      
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 30, 30, 128)       0         
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 14, 14, 128)       262272    
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 14, 14, 128)       0         
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 6, 6, 128)         262272    
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 6, 6, 128)         0         
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 2, 2, 128)         262272    
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 2, 2, 128)         0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513       
=================================================================
Total params: 790,913
Trainable params: 790,913
Non-trainable params: 0
_________________________________________________________________
```

The adversarial network

- **Finally, we setup the GAN, which chains the generator and the discriminator.**
- This is the model that, when trained, will move the generator in a direction that improves its ability to fool the discriminator.
- This model turns latent space points into a classification decision, "fake" or "real", and it is meant to be trained with labels that are always "these are real images". So training `gan` will updates the weights of `generator` in a way that makes `discriminator` more likely to predict "real" when looking at fake images.
- Very importantly, we set the discriminator to be frozen during training (non-trainable): its weights will not be updated when training `gan` .
- If the discriminator weights could be updated during this process, then we would be training the discriminator to always predict "real", which is not what we want!

In [3]:

```
# Set discriminator weights to non-trainable
# (will only apply to the `gan` model)
discriminator.trainable = False

gan_input = keras.Input(shape=(latent_dim,))
gan_output = discriminator(generator(gan_input))
gan = keras.models.Model(gan_input, gan_output)

gan_optimizer = keras.optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
gan.compile(optimizer=gan_optimizer, loss='binary_crossentropy')
```

How to train your DCGAN

Now we can start training. To recapitulate, this is schematically what the training loop looks like:

```
for each epoch:
    * Draw random points in the latent space (random noise).
    * Generate images with `generator` using this random noise.
    * Mix the generated images with real ones.
    * Train `discriminator` using these mixed images, with corresponding targets,
 either "real" (for the real images) or "fake" (for the generated images).
    * Draw new random points in the latent space.
    * Train `gan` using these random vectors, with targets that all say "these are
real images". This will update the weights of the generator (only, since discrimin
ator is frozen inside `gan`) to move them towards getting the discriminator to pre
dict "these are real images" for generated images, i.e. this trains the generator
 to fool the discriminator.
```

Let's implement it:

In [4]:

```
import os
from keras.preprocessing import image

# Load CIFAR10 data
(x_train, y_train), (_, _) = keras.datasets.cifar10.load_data()

# Select frog images (class 6)
x_train = x_train[y_train.flatten() == 6]

# Normalize data
x_train = x_train.reshape(
    (x_train.shape[0],) + (height, width, channels)).astype('float32') / 255.

iterations = 10000
batch_size = 20
save_dir = './'

# Start training loop
start = 0
for step in range(iterations):
    # Sample random points in the latent space
    random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))

    # Decode them to fake images
    generated_images = generator.predict(random_latent_vectors)

    # Combine them with real images
    stop = start + batch_size
    real_images = x_train[start: stop]
    combined_images = np.concatenate([generated_images, real_images])

    # Assemble labels discriminating real from fake images
    labels = np.concatenate([np.ones((batch_size, 1)),
                             np.zeros((batch_size, 1))])
    # Add random noise to the labels - important trick!
    labels += 0.05 * np.random.random(labels.shape)

    # Train the discriminator
    d_loss = discriminator.train_on_batch(combined_images, labels)

    # sample random points in the latent space
    random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))

    # Assemble labels that say "all real images"
    misleading_targets = np.zeros((batch_size, 1))

    # Train the generator (via the gan model,
    # where the discriminator weights are frozen)
    a_loss = gan.train_on_batch(random_latent_vectors, misleading_targets)

    start += batch_size
    if start > len(x_train) - batch_size:
      start = 0

    # Occasionally save / plot
    if step % 100 == 0:
        # Save model weights
        gan.save_weights('gan.h5')

```

```
        # Print metrics
        print('discriminator loss at step %s: %s' % (step, d_loss))
        print('adversarial loss at step %s: %s' % (step, a_loss))

        # Save one generated image
        img = image.array_to_img(generated_images[0] * 255., scale=False)
        img.save(os.path.join(save_dir, 'generated_frog' + str(step) + '.png'))

        # Save one real image, for comparison
        img = image.array_to_img(real_images[0] * 255., scale=False)
        img.save(os.path.join(save_dir, 'real_frog' + str(step) + '.png'))
```

```
discriminator loss at step 0: 0.685675
adversarial loss at step 0: 0.667591
discriminator loss at step 100: 0.756201
adversarial loss at step 100: 0.820905
discriminator loss at step 200: 0.699047
adversarial loss at step 200: 0.776581
discriminator loss at step 300: 0.684602
adversarial loss at step 300: 0.513813
discriminator loss at step 400: 0.707092
adversarial loss at step 400: 0.716778
discriminator loss at step 500: 0.686278
adversarial loss at step 500: 0.741214
discriminator loss at step 600: 0.692786
adversarial loss at step 600: 0.745891
discriminator loss at step 700: 0.69771
adversarial loss at step 700: 0.781026
discriminator loss at step 800: 0.69236
adversarial loss at step 800: 0.748769
discriminator loss at step 900: 0.663193
```

Let's display a few of our fake images:

In [5]:

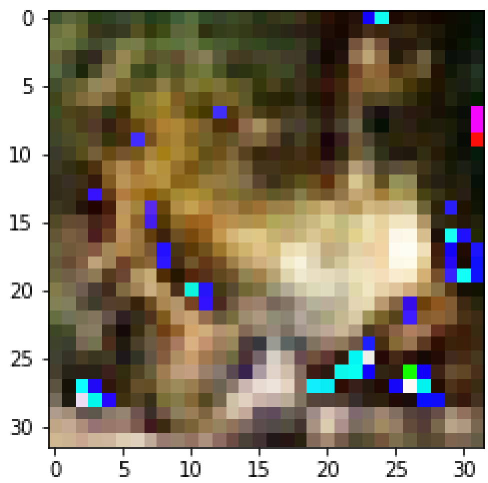
```
import matplotlib.pyplot as plt

# Sample random points in the latent space
random_latent_vectors = np.random.normal(size=(10, latent_dim))

# Decode them to fake images
generated_images = generator.predict(random_latent_vectors)

for i in range(generated_images.shape[0]):
    img = image.array_to_img(generated_images[i] * 255., scale=False)
    plt.figure()
    plt.imshow(img)

plt.show()
```



Froggy with some pixellated artifacts.

Figure 8.18 Play the discriminator: in each row, two images were dreamed up by the GAN, and one image comes from the training set. Can you tell them apart? (Answers: the real images in each column are middle, top, bottom, middle.)

< CIFAR10 > : Classifying Images with CNNs by TensorFlow

[Reference] :

- Tom Hope, Yehezkel S. Resheff, Itay Lieder, "**Learning TensorFlow - A Guide to building Deep Learning Systems**", `Chapters 4` , O'Reilly (2017) (pdf) https://goo.gl/iEmehh (https://goo.gl/iEmehh)
  - `[ Code ]` : https://github.com/gigwegbe/Learning-TensorFlow (https://github.com/gigwegbe/Learning-TensorFlow)
- **CIFAR-10 and CIFAR-100 datasets**, https://www.cs.toronto.edu/~kriz/cifar.html (https://www.cs.toronto.edu/~kriz/cifar.html)

In [ ]:

```

```