1

認識進階深度學習與 Keras

本書第 1 章將介紹三種不同的深度學習類神經網路，後續在書中都會用到。三種深度學習模型包含了 MLP、CNN 與 RNN，這些是本書所要談到的進階深度學習主題的基石，例如自動編碼器與 GAN。

本章將帶領你一同使用 Keras 函式庫來實作這些深度學習模型。我們會先說明 Keras 為什麼是一款絕佳的工具，接著，才是深入討論關於這三種深度學習模型的安裝與實作。

本章學習內容如下：

- 理解為什麼 Keras 函式庫是用於進階深度學習的絕佳方案
- 介紹 MLP、CNN 與 RNN—這些是多數進階深度學習模型的基石，本書後續都會用到
- 提供使用 Keras 與 TensorFlow 來實作 MLP、CNN 與 RNN 的各種範例
- 介紹各種重要的深度學習，包括最佳化、正規化、與損失函數

本章最後會用 Keras 來實作基礎的深度學習模型。下一章所談到的進階深度學習主題就是以本章的實作為基礎，例如深度網路、自動編碼器與 GAN。

支援建置運算圖式的模型、層再利用以及行為模式類似於 Python 函數的模型。同時，`Model` 與 `Layer` 類別也提供框架來實作不常見或實驗性深度學習模型與層。

安裝 Keras 與 TensorFlow

Keras 並非一個獨立的深度學習函式庫。如圖 *1.1.1*，它是建立在另一套深度學習函式庫（或稱為後端）之上，例如 Google 的 **TensorFlow**、MILA 的 **Theano** 或 Microsoft 的 **CNTK**。而 Keras 支援 Apache 的 **MXNet** 則是最近才完成的事情。我們會用 **Python 3 搭配 TensorFlow 後端**來完成本書的範例，這是因為 TensorFlow 實在太熱門而成為大家所常用的後端之一。

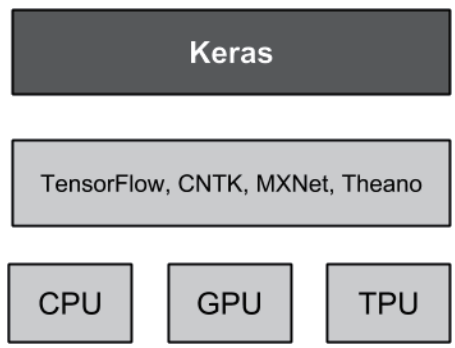


圖 1.1.1：Keras 是一個以其他深度學習模型為基礎的高階函式庫。
Keras 支援 CPU、GPU 與 TPU。

在 Linux 或 macOS 作業系統中，只要編輯 Keras 設定檔 `.keras/keras.json` 就可以切換到不同的後端。根據低階演算法在實作上的差異，同一套神經網路的速度會根據執行在不同的後端上而有所不同。

而看到硬體面，Keras 可在 CPU、GPU 與 Google 的 TPU 上執行。本書會用 CPU 與 NVIDIA GPU 來執行（GTX 1060 與 GTX 1080Ti）。

在進入本書其他部份之前，得先確保你把 Keras 與 TensorFlow 都安裝好了。安裝方式相當多元，例如以下語法是使用 `pip3`：

```
$ sudo pip3 install tensorflow
```

```
        plt.axis('off')

    plt.show()
    plt.savefig("mnist-samples.png")
    plt.close('all')
```

`mnist.load_data()` 方法在此非常方便，有了它就不再需要逐一載入所有 70,000 影像與標籤再存入陣列中了。請在命令列中執行 `python3 mnist-sampler-1.3.1.py`，會顯示訓練與測試資料集的標籤分配狀況：

```
Train labels:  {0: 5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6:
5918, 7: 6265, 8: 5851, 9: 5949}
Test labels:  {0: 980, 1: 1135, 2: 1032, 3: 1010, 4: 982, 5: 892, 6: 958, 7:
1028, 8: 974, 9: 1009}
```

程式稍後會隨機繪製 25 個數字並如圖 *1.3.1* 所示。

在討論多層感知器分類器模型之前，別忘了 MNIST 資料既然是 2D tensor，就應該根據輸入層類型來重新調整形狀。下圖說明如何將 3×3 灰階影像根據 MLP、CNN 與 RNN 輸入層來調整形狀：

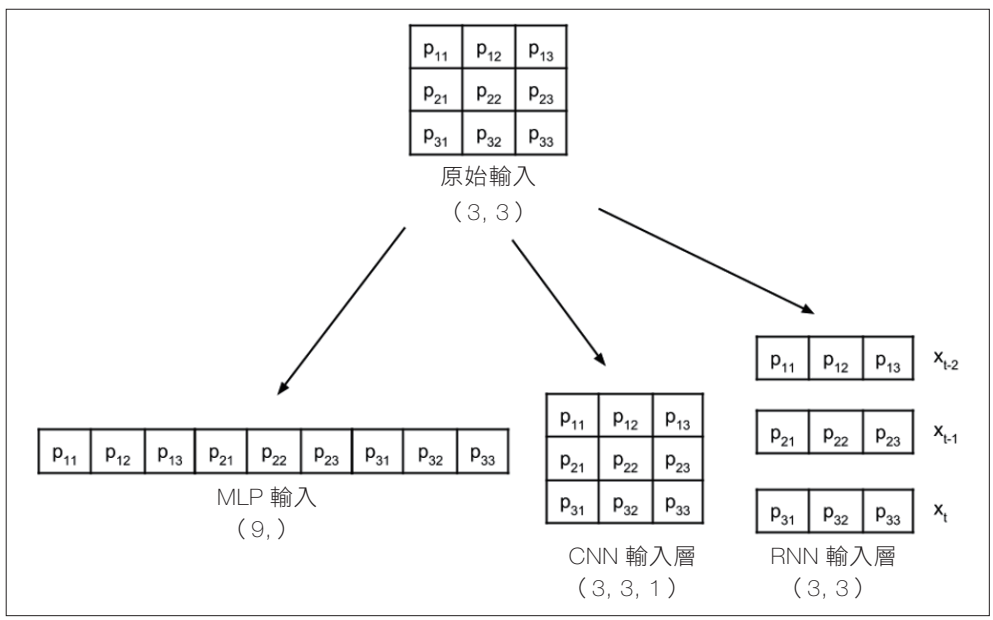
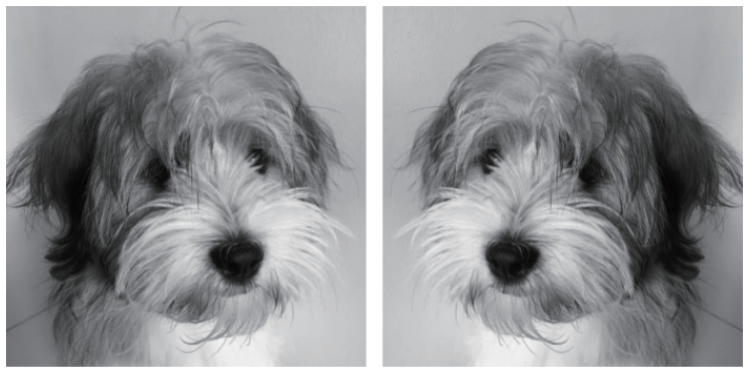


圖 1.3.2：類似於 MNIST 資料的輸入影像根據輸入層型態來調整形狀。
在此使用 3×3 灰階影像以便說明。

例如，一種簡單的資料增強做法就是翻轉狗狗的照片，如下圖（horizontal_flip=True）。如果這是張狗狗的照片，那在翻轉之後還是一隻狗。你還可以執行其他變形操作，縮放、旋轉、白化等等，但標籤依然不變：



原始影像　　翻轉後影像

圖 2.2.6：一種簡單的資料增強做法就是翻轉原始照片。

完整程式碼請由此取得：https://github.com/PacktPublishing/Advanced-Deep-Learning-with-Keras。

要精準重現原本論文成果的難度相當高，尤其是在最佳器與資料增強方面，因此，本書中的 Keras ResNet 實作與原本論文中模型，兩者在效能上還是有些許的不同。

ResNet v2

關於 ResNet 的第二篇論文 [4] 推出之後，上一段所介紹的模型就被稱為 ResNet v1。改良後的 ResNet 普遍稱為 ResNet v2。改良主要在於殘差區塊中各層的排列方式，如下圖。

ResNet v2 的主要修改在於：

- 運用 1×1 - 3×3 - 1×1 BN-ReLU-Conv2D 堆疊

- **解碼器**（**Decoder**）：解碼器會試著從潛在向量 $g(\mathbf{z})=\tilde{\boldsymbol{x}}$ 來回復輸入。雖然潛在向量的維度很低，但其大小還是足以讓解碼器回復輸入資料。

解碼器的目標是讓 $\tilde{\boldsymbol{x}}$ 愈接近 x 愈好。一般來說，編碼器與解碼器都是非線性函數。z 的維度是代表它可表示的顯著特徵數量。這個維度通常會比輸入維度來得小很多，這是為了效率考量，以及要將潛在編碼限制在只會去學習輸入分配中的最顯著屬性 [1]。

當潛在編碼的維度明顯高於 x 時，自動編碼器會傾向記住輸入。

$\mathcal{L}(\boldsymbol{x},\tilde{\boldsymbol{x}})$ 是一款合適的損失函數，可用於量測輸入 x 與輸出的差異，而後者就是回復後的輸入 $\tilde{\boldsymbol{x}}$。如以下方程式所示，**均方差**（**Mean Squared Error, MSE**）就是一款常用的損失函數：

$$\mathcal{L}(\boldsymbol{x},\tilde{\boldsymbol{x}}) = MSE = \frac{1}{m}\sum_{i=1}^{i=m}\left(x_i - \tilde{x}_i\right)^2 \quad \text{（方程式3.1.1）}$$

在本範例中，m 為輸出維度（以 MNIST 為例，m = 寬度 × 高度 × 通道 = $28\times28\times1 = 784$）。x_i 與 $\tilde{x}_i$ 則分別代表 x 與 $\tilde{x}$ 的元素。由於損失函數是用來量測輸入與輸出之間的差異，我們就能使用其他種類的損失函數，例如二元交叉熵或**結構相似性指標**（**structural similarity index, SSIM**）。

與其他種類的神經網路類似，自動編碼器會試著在訓練過程中讓這個誤差或損失函數愈小愈好。圖 *3.1.1* 就是一個自動編碼器。編碼器是一個把輸入 x，壓縮成一個低維度潛在向量 z 的函數。這個潛在向量代表輸入分配的各個重要特徵。解碼器會將潛在向量以 $\tilde{x}$ 的格式來回復原本的輸入。

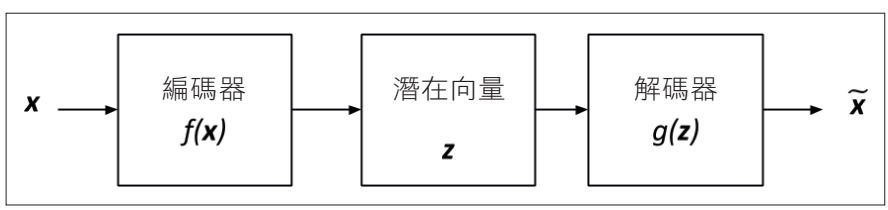


圖 3.1.1 自動編碼器的功能方塊圖。

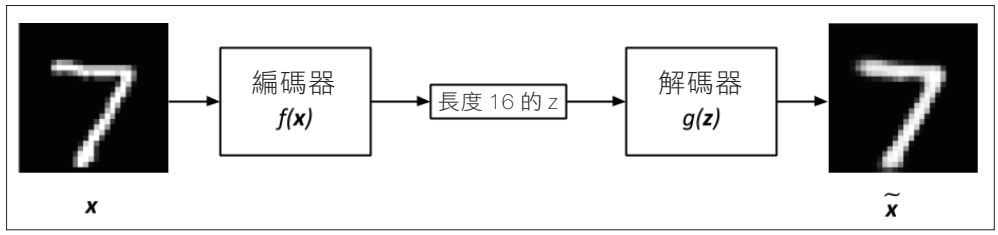


圖 3.1.2：用於 MNIST 數字輸入與輸出的自動編碼器，潛在向量的維度為 16。

對自動編碼器來說，x 可以是一個 MNIST 數字，維度等於 28×28×1 = 784。編碼器會把輸入轉換為一個低維度的 z，例如可以是長度 16 的潛在向量。解碼器會試著將 z 以 $\tilde{x}$ 的格式來回復輸入。就視覺上而言，每一個 MNIST 數字 x 看起來都會很類似於 $\tilde{x}$。圖 *3.1.2* 是這個自動編碼的過程，可以看到解碼後的數字 7，雖然沒有一模一樣但還算相當接近。

由於編碼器與解碼器兩者皆為非線性函數，我們就能用神經網路來實作。例如在 MNIST 資料集，自動編碼器即可用 MLP 或 CNN 來實作。自動編碼器可透過向後傳播過程中將損失函數最小化來訓練。與其他神經網路一樣，對於損失函數的唯一要求是它必須可微分。

如果把輸入當作一個分配來處理，就能把編碼器轉成編碼器分配：$p(\boldsymbol{z}|\boldsymbol{x})$，而把解碼器視為一個解碼器分配：$p(\boldsymbol{x}|\boldsymbol{z})$。自動編碼器的損失函數可如下表示：

$$\mathcal{L} = -\log p(\boldsymbol{x}|\mathbf{z}) \qquad \text{（方程式 3.1.2）}$$

損失函數僅代表我們想要在指定潛在向量分配的前提下，盡量去回復原本的輸入分配。如果將解碼器輸出分配假設為高斯分配，則損失函數可簡化為 MSE：

$$\mathcal{L} = -\log p(\boldsymbol{x}|\mathbf{z}) = -\log \prod_{i=1}^{m} \mathcal{N}\left(x_i; \tilde{x}_i, \sigma^2\right) = -\sum_{i=1}^{m} \log \mathcal{N}\left(x_i; \tilde{x}_i, \sigma^2\right) \alpha \sum_{i=1}^{m} \left(x_i - \tilde{x}_i\right)^2 \qquad \text{（方程式3.1.3）}$$

生成器新的損失函數是最小化鑑別器對於假影像的正確預測，而假影像則是加上了 one-hot 標籤的條件。生成器會根據這個可以騙過鑑別器的 one-hot 向量來學習如何生成特定的 MNIST 數字。下圖是訓練生成器的過程：

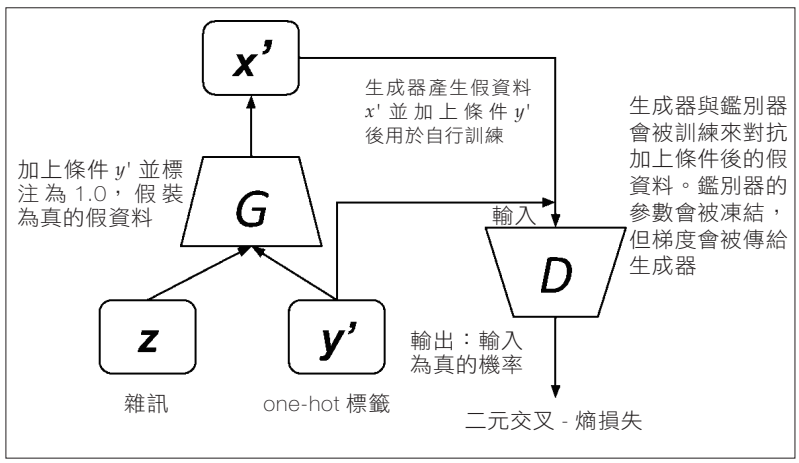


圖 4.3.3：透過對抗網路來訓練生成器的過程類似於訓練 GAN 生成器。
唯一的差別在於所生成的假影像會以 one-hot 標籤來加上條件。

以下範例以粗體強調了鑑別器模型中所做的修改。程式碼使用了一個 `Dense` 層來處理 one-hot 向量，並將其與影像輸入結合。`Model` 實例已針對影像與 one-hot 向量輸入做了修改。

範例 4.3.1，`cgan-mnist-4.3.1.py` 是這個 CGAN 鑑別器，粗體是 DCGAN 中的修改之處：

```
def build_discriminator(inputs, y_labels, image_size):
    """建置鑑別器模型

    經過Dense層之後，輸入會被組合起來
    堆疊 LeakyReLU-Conv2D 來鑑別真假影像
    與DCGAN論文不同，本網路如使用BN則不會收斂，所以在此不使用

    # 引數
        inputs (Layer)：鑑別器的輸入層(影像)
        y_labels (Layer)：用於對輸入加上條件的one-hot向量輸入層
        image_size：一邊的目標尺寸(假設影像為矩形)
```

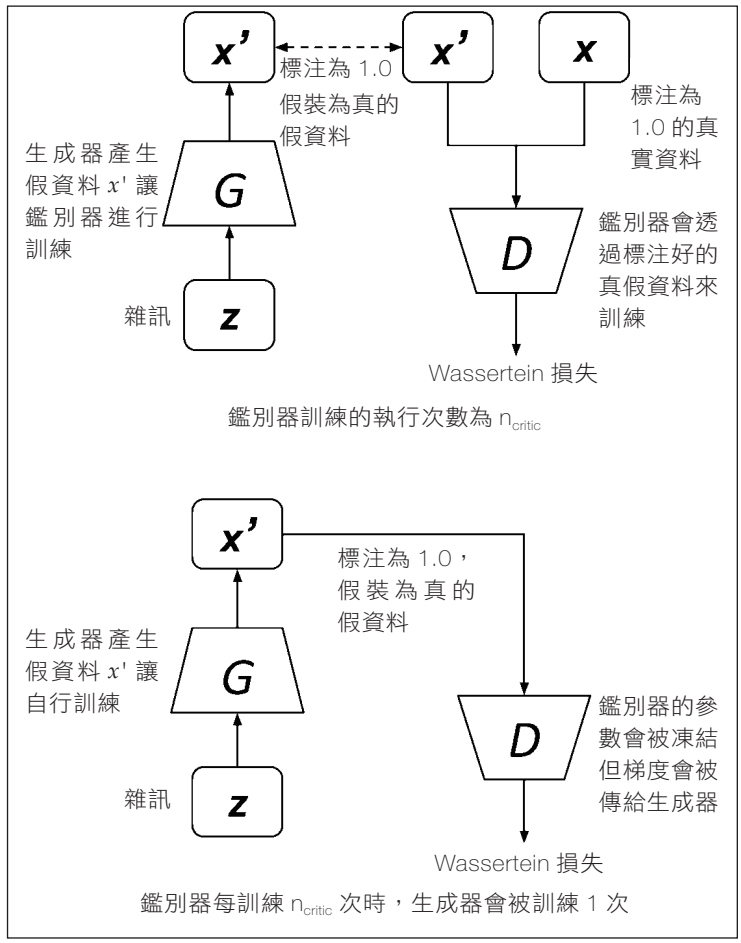


圖 5.1.3 上半：訓練 WGAN 鑑別器需要來自生成器的假資料，與來自真實分配的真實資料。下半：訓練 WGAN 生成器需要生成器的假資料並將其假裝為真實。

類似於 GAN，WGAN 會選擇性地訓練鑑別器與生成器（透過對抗）。不過，在 WGAN 中，在生成器訓練 1 次遞迴（程式碼 9 到 11 行）之前，鑑別器（也稱為 critic) 會先訓練 n_{critic} 次遞迴（程式碼 2 到 8 行）。這與 GAN 的不同之處在於，GAN 中鑑別器與生成器的訓練遞迴次數是相等的。訓練鑑別器代表去學習鑑別器的各個數（權重與偏差值）。這需要抽樣一小批真資料（程式碼第 3 行）與抽樣一小批假資料（程式碼第 4 行），並在把抽樣資料送入鑑別器網路之後計算鑑別器參

下表是 InfoGAN 與一般 GAN 的損失函數比較。InfoGAN 的損失函數與一般 GAN 的不同之處在於多了：$-\lambda I\left(c;\mathcal{G}\left(z,c\right)\right)$，$\lambda$ 為一數值較小的常數。將 InfoGAN 的損失函數最小化相當於一般 GAN 的損失最小化，以及共通資訊最大化 $I\left(c;\mathcal{G}\left(z,c\right)\right)$。

網路	損失函數	方程式
GAN	$\mathcal{L}^{(D)}=-\mathbb{E}_{x\sim p_{data}}\log\mathcal{D}\left(\boldsymbol{x}\right)-\mathbb{E}_{z}\log\left(1-\mathcal{D}\left(\mathcal{G}\left(\mathbf{z}\right)\right)\right)$ $\mathcal{L}^{(G)}=-\mathbb{E}_{z}\log\mathcal{D}\left(\mathcal{G}\left(\mathbf{z}\right)\right)$	4.1.1 4.1.5
InfoGAN	$\mathcal{L}^{(D)}=-\mathbb{E}_{x\sim p_{data}}\log\mathcal{D}\left(\boldsymbol{x}\right)-\mathbb{E}_{z,c}\log\left(1-\mathcal{D}\left(\mathcal{G}\left(z,c\right)\right)\right)-\lambda I\left(c;\mathcal{G}\left(z,c\right)\right)$ $\mathcal{L}^{(G)}=-\mathbb{E}_{z,c}\log\mathcal{D}\left(\mathcal{G}\left(z,c\right)\right)-\lambda I\left(c;\mathcal{G}\left(z,c\right)\right)$ 如為連續編碼，InfoGAN建議使用 $\lambda<1$。本範例設定 $\lambda=0.5$。而如果是離散編碼，InfoGAN建議使用 $\lambda=1$。	6.1.1 6.1.2

表 6.1.1：GAN 與 InfoGAN 的損失函數比較

以 MNIST 資料集來說，InfoGAN 可以學會抽離語意後的離散與連續編碼，藉此做到修改生成器輸出屬性。類似於 CGAN 與 ACGAN，型態為長度 10 之 one-hot 標籤的離散編碼就可以用來指定所要生成的數字。不過，我們還可以加入兩個連續編碼，一個用來控制書寫風格的角度，另一個則用來控制筆畫粗細。下圖是 InfoGAN 的 MNIST 數字編碼。我們保留了長度較短的未抽離語意編碼來代表其他所有的屬性：

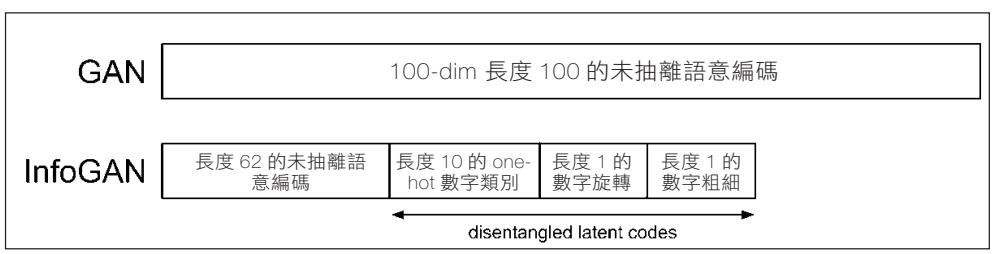


圖 6.1.3：GAN 與 InfoGAN 之編碼比較，用於 MNIST 資料集。

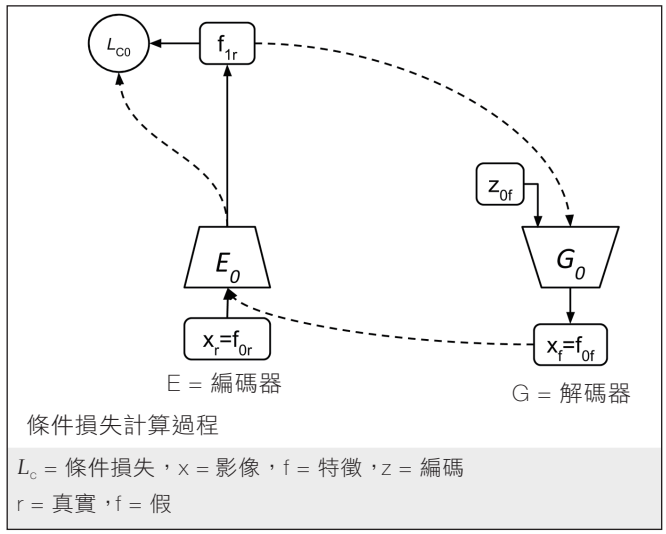


圖 6.2.4：圖 6.2.3 的簡易版，僅顯示計算 $\mathcal{L}_0^{(G)_{cond}}$ 過程中所需的網路元素。

然而，條件損失函數卻帶來了新的問題。生成器忽略了輸入雜訊編碼，而只單純考量 f_{i+1}。**方程式** *6.2.4* 中的 Entropy 損失函數 $\mathcal{L}_0^{(G)_{cond}}$ 是用於確保生成器不會忽略雜訊編碼 z_i。Q- 網路可由生成器的輸出來回復原本的雜訊編碼。回復後的雜訊與輸入雜訊兩者之差同樣是藉由 *L2* 或 MSE 來衡量。下圖是計算 $\mathcal{L}_0^{(G)_{ent}}$ 過程中所用到的網路元素：

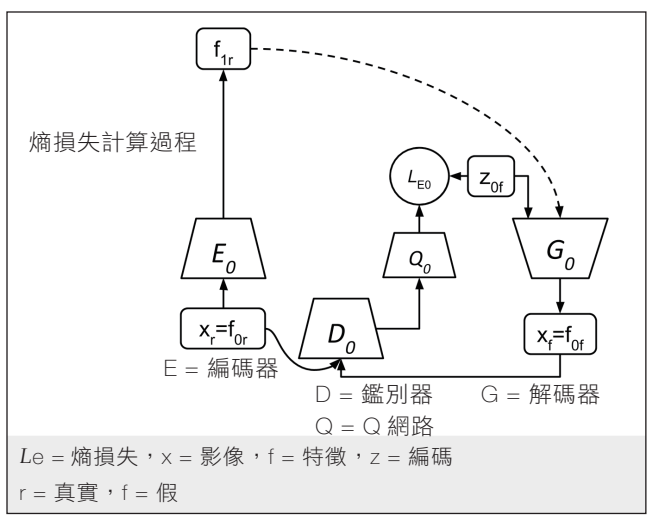


圖 6.2.5：圖 6.2.3 的簡易版，僅顯示計算 $\mathcal{L}_0^{(G)_{cond}}$ 過程中所需的網路元素

最後的損失函數與常見的 GAN 損失函數相當類似，是由一個鑑別器損失 $\mathcal{L}_i^{(D)}$ 與生成器損失（透過對抗）$\mathcal{L}_i^{(G)_{adv}}$ 所組成的。下圖是計算 GAN 損失過程中所用到的網路元素：

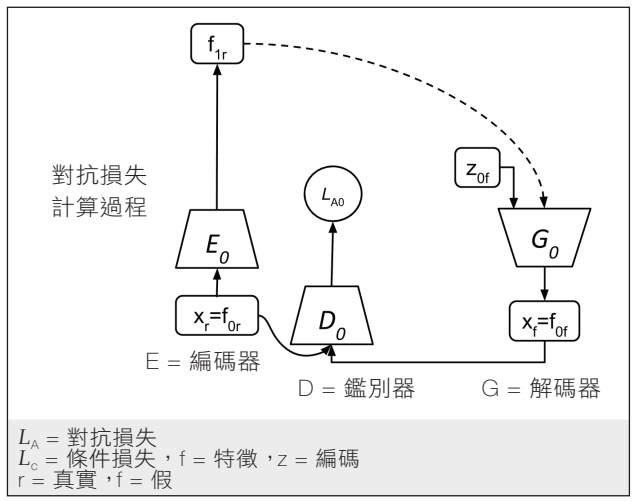


圖 6.2.6：圖 6.2.3 的簡易版，只顯示了計算 $\mathcal{L}_i^{(D)}$ 與 $\mathcal{L}_i^{(G)_{adv}}$ 過程中所需的網路元素。

由**方程式** *6.2.5* 可知，三個生成器損失函數的加權後加總就是最終的生成器損失函數。在後續的 Keras 範例中，除了熵損失的權重設為 10.0 之外，其餘所有權重值都設為 1.0。從**方程式** *6.2.1* 到**方程式** *6.2.5*，i 代表編碼器與 GAN 的群組編號或層級。原本的論文會先獨立訓練網路，接著再一起訓練。在獨立訓練過程中，則是先訓練編碼器。最後在聯合訓練時會同時用到真實與假的資料。

使用 Keras 實作 StackedGAN 的生成器與鑑別器需要一些修改，好加入輔助點來存取各種中間特徵。**圖** *6.2.7* 是在 Keras 中來顯示這個生成器模型。**範例** *6.2.2* 說明了如何建置這兩個生成器（`gen0` 與 `gen1`）的函式，對應到上述的 $Generator_0$ 與 $Generator1$。`gen1` 生成器是由三個 `Dense` 層所組成，並使用標籤與雜訊編碼 z_{1f} 作為輸入，第三層會產生假的特徵 f_{1f}。`gen0` 與其他先前介紹過的 GAN 生成器類似，且在 `gan.py` 中可使用生成器建置器來產生實例：

```
# gen0: feature1 + z0 to feature0 (image)
gen0 = gan.generator(feature1, image_size, codes=z0)
```

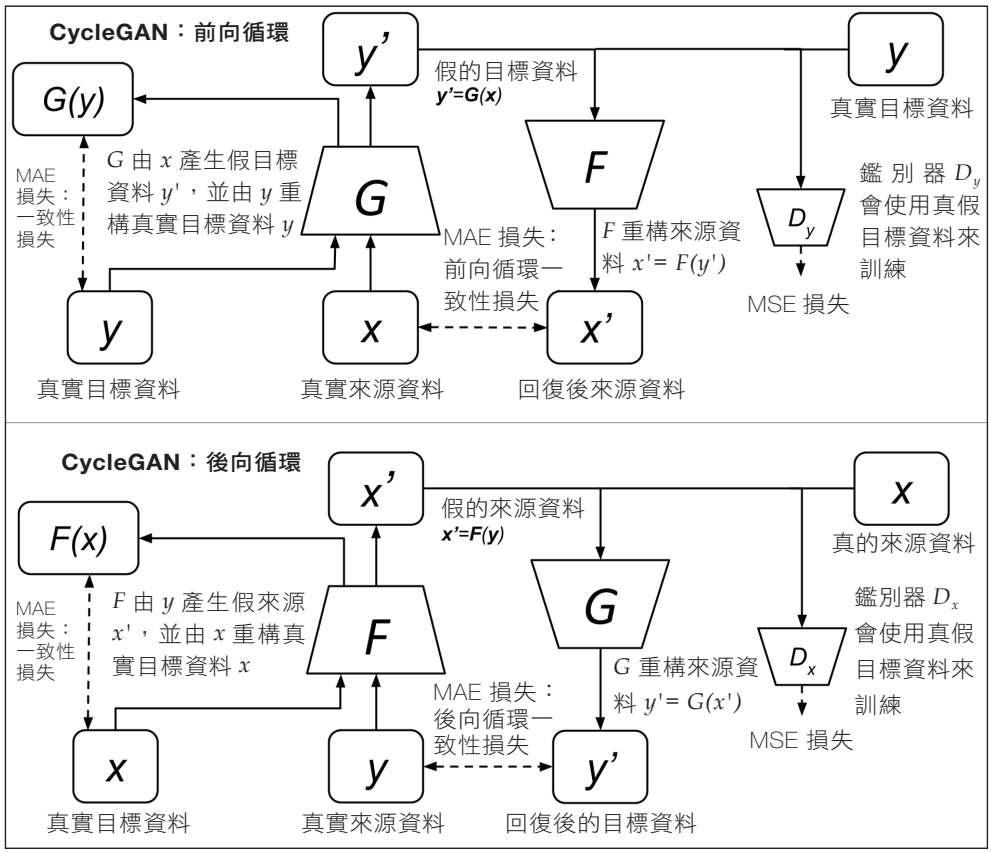


圖 7.1.5：具備身分損失的 CycleGAN 模型，如上圖左側。

在風格轉換問題中，有可能無法成功把顏色從來源影像轉換到假的目標影像上。問題描述如圖 *7.1.4*。為了解決這個問題，CycleGAN 就導入了前向與後向循環的身分損失函數：

$$\mathcal{L}_{identity} = \mathbb{E}_{x\sim p_{data}(x)}\left[\left\|F(x)-x\right\|_1\right] + \mathbb{E}_{y\sim p_{data}(y)}\left[\left\|G(y)-y\right\|_1\right] \quad \text{（方程式 7.1.14）}$$

CycleGAN 的總損失可改寫為：

$$\mathcal{L} = \lambda_1\mathcal{L}_{GAN} + \lambda_2\mathcal{L}_{cyc} + \lambda_3\mathcal{L}_{identity} \quad \text{（方程式 7.1.15）}$$