國立勤益科技大學資訊工程系學生實務專題成果報告

※※※※※※※※※※※※※※※※※※※※※※※※※※
※　　　　　　　　　　　　　　　　　　　　　　　　※
※　「長門櫻-影像魅影」：多功能影像增強評分系統　※
※　　　　　　　　　　　　　　　　　　　　　　　　※
※※※※※※※※※※※※※※※※※※※※※※※※※※

執 行 期 間 ：113 學年度上學期至 113 學年度下學期

專題參與人員：陳右承

部別：四技日間部

組別：第 1 組

班別： 丙班　座號： 20　姓名： 陳右承
班別：　座號：　姓名：
班別：　座號：　姓名：

指導老師：陳明德 教授

中 華 民 國 114 年 5 月 22 日

目錄

圖目錄

「長門櫻-影像魅影」：多功能影像增強評分系統
Nagato-Sakura-Image-Charm

陳右承[1] 陳明德[6]
CHEN,YU-CHENG[1], CHEN,MING-DE[6]
國立勤益科技大學資訊工程系：
Department of Computer Science and Information Engineering：
National Chin-Yi University of Technology：

摘要：

「長門櫻-影像魅影」是一套整合影像增強與品質評估功能的桌面應用程式，致力於還原影像在壓縮、傳輸過程所產生的品質降低問題。系統核心採用 CNN 架構結合殘差密集區塊與注意力機制混合的模型，並針對不同場景需求開發場景特化模型。系統特色在於使用了區塊式處理架構與多種權重遮罩融合技術，有效解決大尺寸圖像處理時的記憶體限制與區塊邊界明顯問題。在影像評估方面，系統整合了傳統圖像指標（PSNR、SSIM）與輕量化 CNN 架構的評分模型，並提供差異熱力圖、色彩直方圖及邊緣比較等視覺化工具。系統支援多種運算環境，可透過自適應混合精度運算策略在不同硬體配置下維持高效處理；同時針對模型設計了基準測試功能，評估實際場景中的推理性能與資源佔用。本系統可廣泛應用於網路圖像優化、多媒體內容修復與影像品質評估等領域，提供全面且高效的影像處理解決方案。

Abstract：

"Nagato-Sakura-Image-Charm" is a comprehensive desktop application integrating image enhancement and quality assessment functions, designed to restore image quality degradation caused by compression and transmission processes. The system's core utilizes a CNN architecture combining Residual Dense Blocks and attention mechanisms, with specialized models. The system features a block-based processing architecture with various weight mask blending techniques, effectively addressing memory limitations when processing large-scale images while eliminating visible block boundaries. For image assessment, the system incorporates traditional metrics (PSNR, SSIM) alongside a lightweight CNN-based scoring model, and provides visualization tools including difference heat maps, color histograms, and edge comparison. The system supports various computing environments through adaptive mixed-precision computation strategies to maintain efficient processing across different hardware configurations. A sophisticated benchmarking function evaluates inference performance and resource utilization in real-world scenarios. This system can be widely applied to network image optimization, multimedia content restoration, and image quality assessment, offering a comprehensive and efficient solution for image processing challenges.

前言

隨著數位影像在現代社會的廣泛應用，圖像在網路傳輸和壓縮過程中不可避免地面臨品質降低的問題。影像失真、壓縮痕跡以及細節流失不僅影響視覺體驗，更可能導致關鍵資訊的喪失。另外由於我與其他影像愛好者常常進行圖片影片壓縮的流程，常會因為壓縮的設定感到苦惱，要怎麼判斷一張圖片一部影片的品質如何?現有的指標像 PSNR、SSIM 雖然判斷兩個同樣的影片能展現出兩個影片的差異，但如果我的影片內容不同呢? 又或著我只有一部影片呢? 這樣不就無法大概的判斷品質了?

為了解決這個問題，我開發了「長門櫻-影像魅影」(Nagato-Sakura-Image-Charm 簡稱 NS-IC)多功能影像增強評分系統，旨在為使用者提供一套全方位的影像處理與品質評估解決方案。此專案一開始源於「長門櫻計畫」中的「長門櫻影像評分」(Nagato-Sakura-Image-Quality-Classification 簡稱 NS-IQC) 從一開始單純的影像評分衍生到後來整合圖像處理核心(簡稱 NS-IQE)、GUI 介面以及「長門櫻影像分類」(Nagato-Sakura-Image-Classification 簡稱 NS-C) 組合成「長門櫻-影像魅影」桌面應用程式，能夠處理圖像強化、影片處理、品質評估等處理多樣化的圖形需求。

由於「長門櫻計畫」並不是這次專案的重點所以這邊就快速的介紹，「長門櫻計畫」的構想與目前主流單一模型構成 AGI 不同，我認為既然神經網路模型是由多個神經元所構成，那為何不把多個神經網路模型連接在一起，將現有神經路模型視為一個更大的神經元，就能形成出更大的模型呢? 抱著這個想法我開始進行研究將多個神經網路模型進行連接組合，形成類 AGI 助手的研究也就是「長門櫻計畫」。而本專案「長門櫻-影像魅影」就是由多個「長門櫻計畫」衍生出來的子模型或著可稱為專家模型所結合衍生出來的桌面應用程式。

專題簡介

一、製作目的

1. 解決數位媒體傳輸中的品質損失問題：

隨著網路應的普及，圖像及影像在上傳、傳輸和下載過程中常因壓縮而產生嚴重的品質降低。本系統旨在還原這些損失的細節與品質，帶來更好的觀看體驗。

2. 提供準確且易用的開源影像處理工具：

建立一個兼具專業處理能力與操作簡便性的應用程式，讓不管是 AI 愛好者、影像愛好者還是不具備相關知識的一般使用者也能輕鬆透過本程式提升圖像品質。

3. 打造兼容低端設備的處理架構：

開發能夠在較低端電腦設備上也能高效運行的處理系統，通過區塊處理與資源優化，使處理高解析度圖像和影片降低被顯存限制的窘境。(最低可支援 NVIDIA Geforce 600 系列)

4. 建立客觀與感知相結合的圖像評估體系：

圖像評估系統結合傳統指標與 AI 評分模型，能透過 AI 評分客觀且快速估算圖片品質的同時，也可以透過傳統指標以及視覺化指標比較圖片差異。

5. 促進影像處理技術的大眾化應用：

透過圖形化介面以及開放式架構，可降低影像處理技術的使用門檻，同時提供其他有興趣的開發者自行訓練新模型、加入擴展以及二次開發的可能性。

二、方法

為實現上述目標，我們採用了以下技術方法

1. 圖像處理模型架構 (NS-IQE)

我設計了一個深度學習架構，結合了卷積神經網絡(CNN)與注意力機制，針對特定場景進行復原優化。模型主體採用改良版殘差密集網絡架構，參考了 Real-ESRGAN 技術框架[1]，但針對不同類型的圖像失真進行了特定優化。

1.1 網絡主體結構

模型主幹由多層殘差密集連接區塊(RRDB)組成，具體架構如下：

- 初始特徵提取層使用 3×3 卷積核從 RGB 輸入圖像中提取 64 通道特徵
- 下採樣編碼器：通過兩個 4×4 卷積層(stride=2)將特徵圖逐步下採樣至原尺寸的 1/4，同時增加通道數至 256
- 連續 16 個 RRDB 單元進行深度特徵提取，每個 RRDB 包含三個密集連接層和一個殘差連接
- 整合自注意力機制[2]計算特徵間的全局關聯
- 通過兩次雙線性插值上採樣(每次 scale=2)恢復原始分辨率
- 最後重建層透過多層卷積操作將特徵轉換回 RGB 圖像空間

對於動畫 JEPG 壓縮還原模型（Kyouka《鏡花》系列），16 個 RRDB 區塊配合 64 基礎通道數可在效能與品質間取得良好平衡；但針對寫實 JEPG 壓縮還原模型（Kairitsu《界律》），在使用同種損失函數算法但不同權重下出現模型收斂不足現象，修改損失函數算法以及增加至 25K 樣本的訓練量才能達到理想效果。在訓練第 9 代《界律》模型時因未再出現進步，嘗試再添加樣本至 35K 後重啟訓練，模型仍沒有進一步提升 PSNR，我們推測達到模型飽和並終止實驗。

1.2 自注意力機制

為了處理大範圍的圖像結構恢復，我在 CNN 模型的基礎下實現了自注意力模塊：

```python
class AttentionBlock(nn.Module):
    def __init__(self, channels):
        super(AttentionBlock, self).__init__()
        self.query = nn.Conv2d(channels, channels // 8, kernel_size=1)
        self.key = nn.Conv2d(channels, channels // 8, kernel_size=1)
        self.value = nn.Conv2d(channels, channels, kernel_size=1)
        self.gamma = nn.Parameter(torch.zeros(1))
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        batch_size, channels, height, width = x.size()
        proj_query = self.query(x).view(batch_size, -1, height * width).permute(0, 2, 1)
        proj_key = self.key(x).view(batch_size, -1, height * width)
        energy = torch.bmm(proj_query, proj_key)
        attention = self.softmax(energy)
        proj_value = self.value(x).view(batch_size, -1, height * width)
        out = torch.bmm(proj_value, attention.permute(0, 2, 1))
        out = out.view(batch_size, channels, height, width)
        out = self.gamma * out + x
        return out
```

圖 1：注意力模塊

- 透過 Query、Key、Value 三路映射建立像素間的關聯矩陣
- 使用可學習的 gamma 參數控制注意力強度，實現自適應調節
- 在缺失細節區域建立長程依賴性，有效重建圖像全局結構

該機制允許模型在空間維度上建立長距離像素關聯，有利於恢復 JPEG 壓縮區塊邊界和修復馬賽克區域。實驗顯示，加入注意力機制後，對圖像邊緣和結構的保留能力提升了，尤其在高頻細節區域。

圖 2：q10 品質輸入圖

圖 3：開發早期版本模型輸出未引入注意力機制

圖 4：第六代 Kyouka-LQ 模型

1.3 區塊處理技術

為解決高解析度圖像處理時的顯存限制，我首先想到了 Stable Diffusion 中的 Tile Diffusion 並參考了其圖像分塊的設計，透過使用區塊處理系統大幅降低了對硬體的要求：

- 實現自適應分塊策略，將大型圖像分割為小區塊（預設 256×256 像素）
- 使用重疊策略（預設重疊 64 像素）避免區塊邊界處理不連續問題
- 採用多種權重遮罩方式（如高斯分佈、改進型高斯分佈等）進行區塊融合[3]
- 在區塊邊緣處使用動態泊松融合技術消除接縫痕跡

該方法使系統可在有限顯存條件下處理任意大小的圖像，讓低端設備處理高畫質圖片成為了可能同時維持了圖片的視覺質量。

2.差異化損失函數設計

根據不同類型的圖像失真，我們設計了專用損失函數：

2.1 動畫 JPEG 壓縮修復損失函數（用於 Kyouka 系列模型）

第 6 代 Kyouka《鏡花》算是較早期開發的模型，主要透過 Stable Diffusion 透過固定模型以及修改提示詞生成固定品質的圖片，在經由 DCT 壓縮成固定十種不同品質得 JPEG 圖片，由於那時候急切想提升圖像畫質進而使用相較激進的損失函數設定，優先追求模型的輸出的結果 PSNR，對於平衡缺乏考慮但由於動畫相較寫實圖片缺乏細節紋理，故這時還沒遇到甚麼問題。

```
total_loss = (mse_loss * 3.0 +
              0.3 * mse_down1 +
              0.1 * mse_down2 +
              0.8 * edge_loss +
              1.0 * high_freq_loss +
              0.01 * color_loss)
```

圖 5：第 6 代 Kyouka 及 Kyouka-LQ 的損失函數

第 7 代 Kyouka《鏡花》在開發時已經是較中後期，由於時間問題所以先開發其他功能的模型為最優先事項，故第 7 代 Kyouka《鏡花》只有針對中等畫質的 Kyouka-MQ，而處理更高畫質的 Kyouka-HQ 以及其他版本《鏡花》的則直接果斷終止訓練。第 7 代模型增加了更多損失函數的控制項能更細的調整損失函數的權重，大幅提升了模型的 PSNR 上限從第六代模型的 31 左右的 PSNR 提升到了 33 PSNR 左右

```
total_loss = (
    mse_loss * 1.0 +
    l1_loss * 1.2 +
    ssim_loss * 1.2 +
    sobel_loss * 1.5 +
    high_freq_loss * 1.2 +
    contrast_loss * 0.8 +
    gram_loss * 0.3 +
    fft_loss * 0.3 +
    region_loss * 0.4 +
    color_loss * 0.05
)
```

圖 6：第 7 代 Kyouka-MQ 的損失函數

mse_loss 降低使 PSNR 更穩定成長

l1_loss 強化銳利度

ssim_loss 保持圖片結構

sobel_loss 提高邊緣檢測權重，強化線條

high_freq_loss 提高高頻權重，增強線條細節

contrast_loss 提高對比度權重，使色塊更分明

gram_loss 降低紋理損失，減少真實感紋理生成

fft_loss 降低頻域損失

region_loss 提高區域損失，加強重點區域處理

color_loss 提高顏色損失，改善二次元鮮豔色彩

2.2 寫實 JPEG 壓縮修復損失函數（用於 Kairitsu 系列模型）

第 9 代 Kairitsu《界律》是我花最多時間反覆調整的模型，起初 Kairitsu 並非作為 JPEG 壓縮還原模型，而是 H.264 編碼還原模型但經過反覆實驗進行 H.264 圖片還原會有個致命問題，H.264 編碼基本分為平均位元率以及恆定品質編碼，兩者皆無法有效掌控變數，有的影片動態太多導致畫面會一直很糊有的因為動態少導致 CQ 即使很高但畫質仍然沒有降低多少，很難捕捉畫面特徵所以從第 8 代模型開始放棄該作法，從原先透過 FFMPEG 調用 NVENC 編碼器轉碼 10 種畫質影片後隔幀擷取的方法，改回類似 Kyouka 的作法先進行影片隔幀擷取再行 DCT 壓縮 10 種畫質。

```
total_loss = (
    loss_l1 * 1.5 +
    loss_mse * 0.05 +
    loss_ssim * 2.0 +
    loss_edge * 1.2 +
    loss_hf * 1.0 +
    loss_color * 0.1
)
```

圖 7：第 9 代 Kairitsu 的損失函數

2.3 動畫馬賽克還原復損失函數（用於 Ritsuka 系列模型）

第 7 代 Ritsuka《斷律》是繼《鏡花》模型完整開發之後開發的第二種模型，由於開發時間較晚因為時間問題所以只先試做了 2~4 格等寬動畫圖像的馬賽克還原模型，模型訓練的想法與 Kyouka 一樣但透過不同的訓練集以及調整損失函數來達到模型特化的效果。雖然我稱之為第 7 代模型，但其實與原版第 7 代模型有較大的差異(是由第 6 代模型改寫進行馬賽克特化所以也被稱為第七代)，對重建邊緣和結構進行了特化。

```
total_loss = (
    l1_pixel_loss * 1.5 +
    mse_pixel_loss * 0.2 +
    structure_loss * 2.5 +
    edge_loss * 3.0 +
    color_loss * 0.1
)
```

圖 8：第 7 代 Ritsuka 的損失函數

structure_loss 強調結構恢復，去除塊狀感

edge_loss 強制重建馬賽克邊緣

這些特定領域損失函數設計顯著提高了模型在各自場景下的表現，對於動漫壓縮圖像，重視保留線條及色塊均勻度；而對於馬賽克修復，則著重於邊界重建和整體結構恢復。

3.使用者友好的介面設計

為實現良好的使用體驗，我參考 Topaz AI 以及 Stable Diffusion 的操作方式與操作介面，透用 PyQt6 重新設計了一套全新友好的圖形化介面：

專案初期那時目的主要以 AI 評分為主要目的，所以設計非常簡陋，能顯示圖片及分數即可，那時專題名稱為「深度學習影像品質判斷」，Nagato-Sakura-Image-Quality-Classification 簡稱：NS-IQC)，那時就有圖片並排顯示以及分割顯示以及單獨顯示的功能，但是無法拖動圖片以及滑鼠滾輪放大縮小圖片，操作非常彆手。後來 NS-IQC 變成專案中的 extensions 改稱為 Nagato-Sakura-Image-Quality-Assessment，簡稱：NS-IQA)，在評估頁面作為 AI 評分。

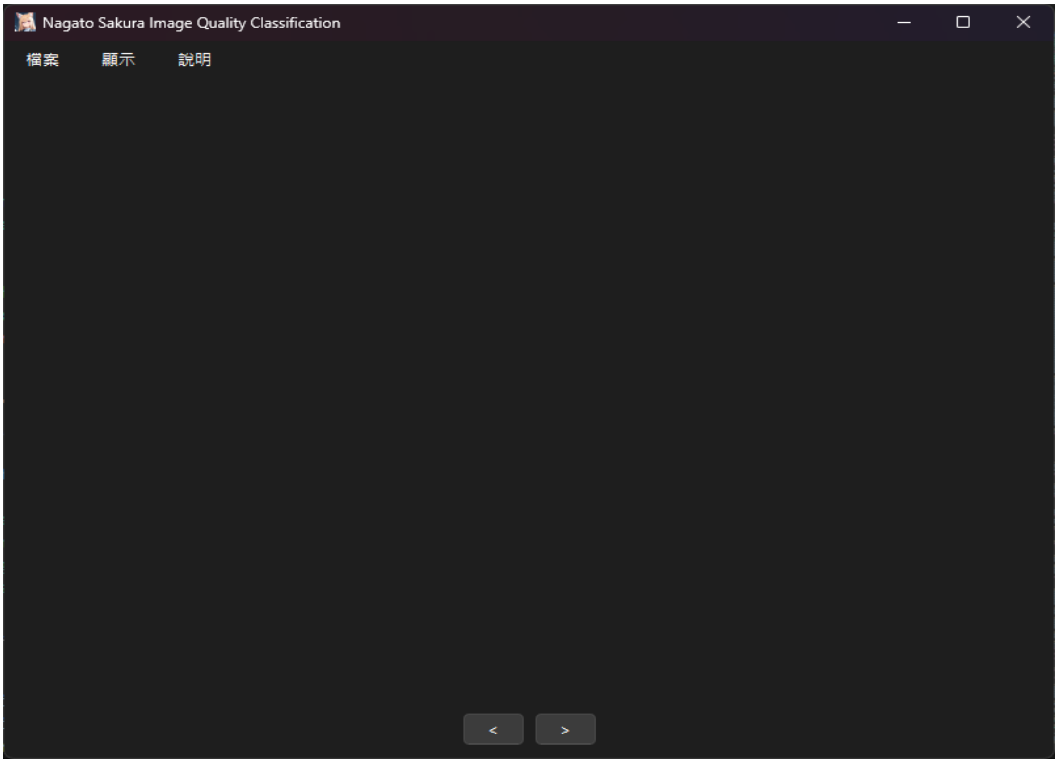

圖 9：專案初期的 GUI (NS-IQC)，那時主要目的還是以圖像評分為主

新版 GUI 大約在 3 月初自己內測的 0.0.1 Beta 版本就基本固定下來頂多進行些微調整維持到現在的 1.2.0 Beta 版本，最大的改動在 1.0.1 版本全面引入伸縮式參數選單，大幅提升預覽框的面積，早期版本雖然能使用滾輪放大但無法透過滑鼠拖動圖片，以及分割模式分割線會有 Bug 直到 3 月中完成 1.0.0 版本才修好所有預覽圖的問題，由於都忙著在改進程式碼功能 Github 拖到 4/19 才上傳 1.0.0 版本那時其實 1.0.2 版本都已經在完成了，1.1.0 都已經在開發中了，只是還在思考程式版本該怎麼編號以及思考 README.md 的編寫導致一直沒上傳，以下圖片是由 1.2.0 Beta 版本擷取。

參數面板，從 1.0.1 版本後調整為可伸縮式的參數面板，可以最大化預覽圖的顯示範圍。並且在 1.2.0 中將會添加模型推薦功能插件，透過分類模型來讀取圖片並進行分類推薦用戶該使用哪一種模型，現在參數面板能進行圖片保存的格式、模型的選擇、模型參數調整、圖片超分等功能。

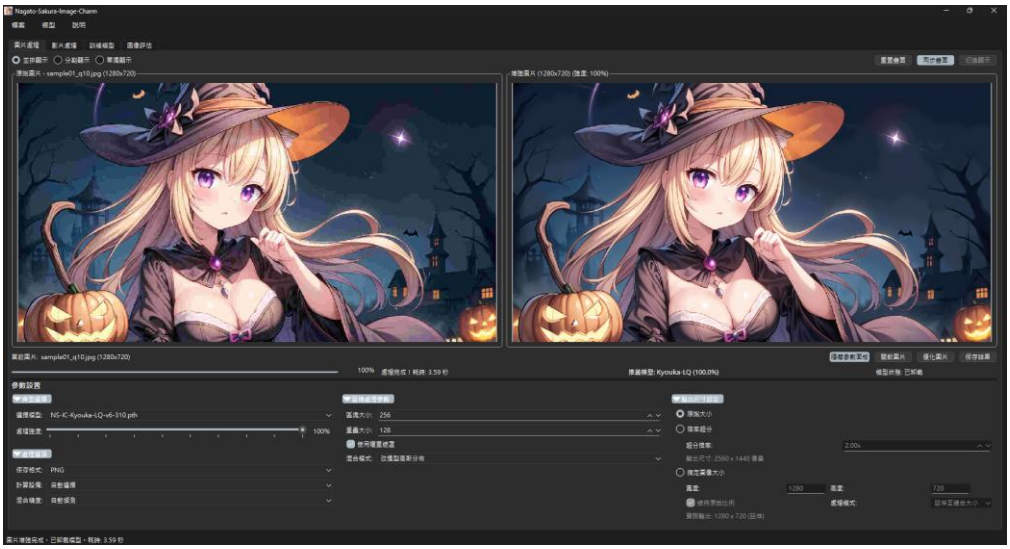

圖 10：圖片處理頁面 + 並排顯示模式 + 顯示參數面板

並排顯示模式，同時顯示完整的圖 A 以及完整的圖 B，並且可以透過右邊同步畫面按鈕來同步放大縮小移動圖 A 與圖 B 方便進行圖片同區域的比較，也可以關閉同步功能放大單一圖片進行查看。

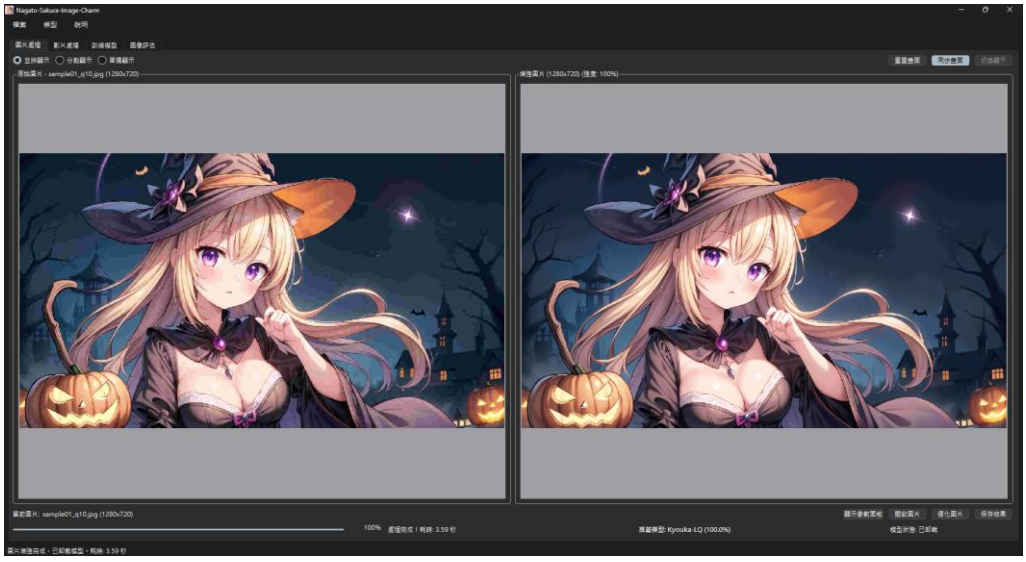

圖 11：圖片處理頁面 + 並排顯示模式 + 隱藏參數面板

分割模式，預設左邊顯示半張圖 A 右邊顯示半張圖 B，中間顯示分割線可以自由左右拖動分割線，達到對比圖片的效果。

圖 12：圖片處理頁面 + 分割顯示模式 + 隱藏參數面板

單獨顯示，一次顯示一張圖片，透過瞬間切換顯示圖片達到視覺的衝擊，感受兩張圖片的差異性。

圖 13：圖片處理頁面 + 單獨顯示模式 + 隱藏參數面板

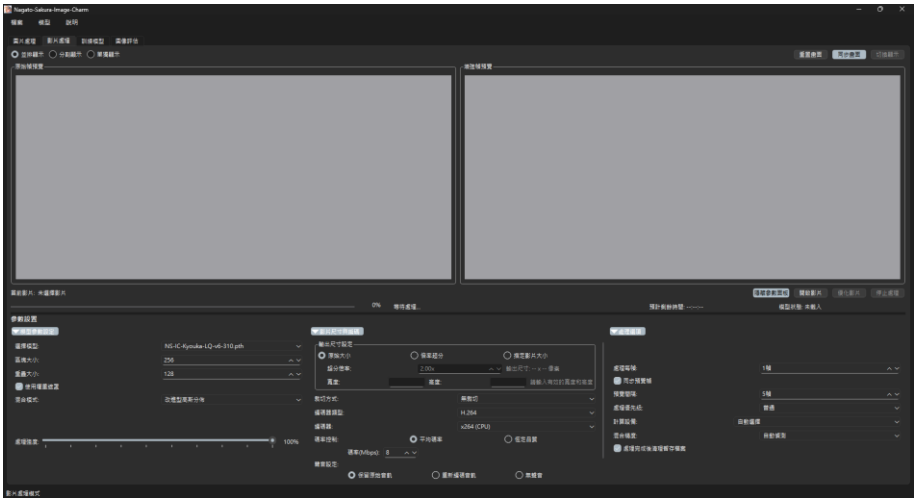

圖 14：影片處理頁面 + 並排顯示模式 + 顯示參數面板

圖 15：訓練頁面 + DCT 壓縮資料處理頁

圖 16：訓練頁面 + 第 5 代模型訓練頁

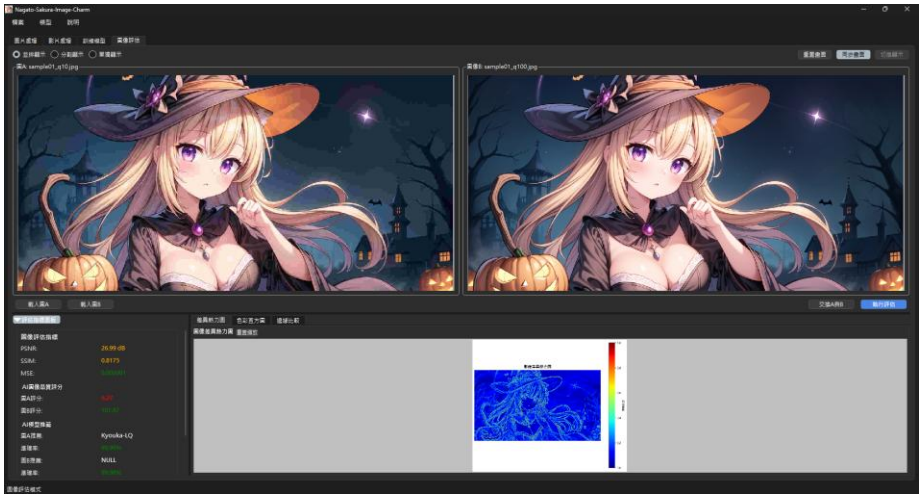

圖 17：評估頁面

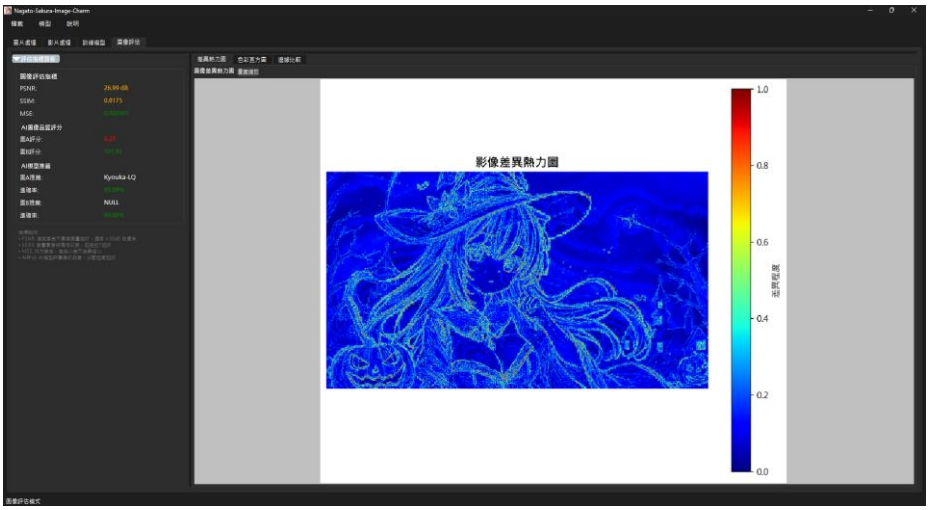

圖 18：評估頁面 + 差異熱力圖

圖 19：評估頁面 + 色彩直方圖

圖 20：評估頁面 + 邊緣比較圖

圖 21：下載頁面 + 官方模型頁 + 圖像預覽

圖 22：下載頁面 + 官方模型頁 + 清單預覽

圖 23：下載頁面 + 自訂 url

圖 24：基準測試 + 測試設定頁面

圖 25：基準測試 + 測試結果頁面

圖 26：基準測試 + 測試設定頁面 GTX-1660Ti Max Q 跑分結果

圖 27：基準測試 + 測試設定頁面 RTX 3070 跑分結果

三、結果

本專題已成功實現以下成果：

1. 性能與效果

- 動畫還原：Kyouka 系列模型在動漫圖像上達到 31.4dB PSNR，已經能良好還原劣質照片，同時在主觀視覺質量上表現優異，尤其在線條清晰度和色彩均勻性方面。
- 馬賽克還原：Ritsuka-HQ 模型在測試集上達到 31.4B PSNR，能夠成功還原 2~4 格狀區塊效應並保持邊緣銳利度。
- 寫實還原：Kairitsu 泛用模型在寫實圖像上達到驚人的 38.5dB PSNR，已經能良好還原寫實的劣質照片。
- 處理效率：透過區塊處理技術，可以在最小 512MB 顯存的顯示卡或是處理器以 128×128 分塊大小處理圖片，並且在高端顯卡上可手動及自動開啟混合精度，並透過最大 512×512 分塊大小(顯存占用約 2.5~3GB)更快的處理圖片。

2.實用性驗證

- 用戶測試：將程式提供給我的幾位朋友，多數任為系統介面直觀易用，不太需要專業知識即可操作。
- 場景適應性：可以根據圖片類型處理不同類型的圖片，即使沒有符合需求的模型也可以透過內建的第 5 代模型訓練器以及 Github 提供的模型訓練器倉庫針對場景訓練適合自己場景的模型。
- 即插即用：內建模型下載器不需要離開 UI 就能下載到模型，並支持外部模型導入，以及模型管理功能，操作體驗良好。

3.系統穩定性

- 通過區塊處理以及顯存管理優化，有效的避免顯存溢出問題並且有良好的利用系統資源。
- 透過檢測設備自動決定是否開啟混合精度，確保不同設備下皆能以最佳狀態以及穩定運行。

製作理論探討

一、圖像處理模型設計

1.殘差密集網絡

殘差密集區塊是本系統的核心，其設計理念源自於兩個關鍵概念：殘差學習與密集連接。 殘差學習解決了深層網絡中的梯度消失問題，使模型能夠有效學習高頻細節；而密集連接則最大化特徵重用，增強網絡的表達能力。具體實現如下：

```python
class RRDB(nn.Module):
    """殘差密集區塊，用於提取特徵並保留細節"""
    def __init__(self, in_channels, growth_channels=32):
        super(RRDB, self).__init__()
        self.dense1 = nn.Sequential(
            nn.Conv2d(in_channels, growth_channels, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.dense2 = nn.Sequential(
            nn.Conv2d(in_channels + growth_channels, growth_channels, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.dense3 = nn.Sequential(
            nn.Conv2d(in_channels + 2 * growth_channels, growth_channels, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.final_conv = nn.Conv2d(in_channels + 3 * growth_channels, in_channels, kernel_size=3, padding=1)
        self.lrelu = nn.LeakyReLU(0.2, inplace=True)

    def forward(self, x):
        residual = x
        out1 = self.dense1(x)
        out = torch.cat([x, out1], dim=1)
        out2 = self.dense2(out)
        out = torch.cat([out, out2], dim=1)
        out3 = self.dense3(out)
        out = torch.cat([out, out3], dim=1)
        out = self.final_conv(out)
        return self.lrelu(residual + out * 0.2)
```

圖 28：殘差密集連接區塊(RRDB)實現，src/models/NS_ImageEnhancer.py

- 殘差學習：不學習整張圖，只學習"差異部分"。就像修照片時，不重畫整張圖，只修補需要改進的部分。
- 密集連接：每位工人都能看到所有前任工人的工作成果。比如第三位工人能同時看到原始圖像、第一位和第二位工人的處理結果，這樣能做出更好的決定。

每個 RRDB 區塊像是接力賽跑，一個處理完把成果傳給下一個。

我還設置了"鼓勵因子" β=0.2，讓每次修改不會太激進，這樣整體更穩定。

整個 RRDB 的運作可以用下面的數學式表示：

$F_{D(x)} = x + \beta * H_{D(x)}$

這表示最後的輸出是「原始輸入 + 一小部分的改進結果」。

中間的 H_D(x) 是一連串密集連接的運算，會把多層的資訊整合起來：

$H_{D(x)} = Conv([x, F_{1(x)}, F_{2(x)}, ..., F_{n(x)}])$

每一層 F_n 都會用卷積（也就是一種圖像處理操作）來轉換資料，而且會看到前面所有層的結果。這樣，特徵就可以被「重複使用」，而不是每一層都從零開始。

在 RRDB 中，每個密集連接層都能直接訪問前面所有層的特徵圖，形成了豐富的特徵組合。密集連接的主要優勢是實現了特徵重用和深層監督，這對於圖像增強特別是對細節和紋理的恢復非常重要。同時，殘差連接允許網絡關注輸入與輸出之間的差異，這對於圖像恢復任務尤其有效[4]。

2.自注意力機制

在處理圖片時，有時候光看局部還不夠，因為有些圖像特徵需要結合整體結構來理解，自注意力模塊計算過程實現如下：

```python
class AttentionBlock(nn.Module):
    def __init__(self, channels):
        super(AttentionBlock, self).__init__()
        self.query = nn.Conv2d(channels, channels // 8, kernel_size=1)
        self.key = nn.Conv2d(channels, channels // 8, kernel_size=1)
        self.value = nn.Conv2d(channels, channels, kernel_size=1)
        self.gamma = nn.Parameter(torch.zeros(1))
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        batch_size, channels, height, width = x.size()
        proj_query = self.query(x).view(batch_size, -1, height * width).permute(0, 2, 1)
        proj_key = self.key(x).view(batch_size, -1, height * width)
        energy = torch.bmm(proj_query, proj_key)
        attention = self.softmax(energy)
        proj_value = self.value(x).view(batch_size, -1, height * width)
        out = torch.bmm(proj_value, attention.permute(0, 2, 1))
        out = out.view(batch_size, channels, height, width)
        out = self.gamma * out + x
        return out
```

圖 1：注意力模塊

自注意力機制讓模型能夠關注圖像中最重要的區域，並在這些區域間建立長距離依賴關係。這個機制通過計算所有空間位置之間的關係來實現，使得特徵表示更加全局化[5]。

打個比方：

Q（Query）：代表正在尋找什麼特徵？

K（Key）：每個位置提供什麼特徵？

V（Value）：每個位置的實際內容是什麼？

自注意力機制會根據 Q 和 K 的匹配程度，來決定從 V 中讀取多少資訊。

在我的實現中，自注意力模塊首先通過 1×1 卷積計算 query、key 和 value 映射，然後計算 query 和 key 的點積來獲得注意力矩陣。這個注意力矩陣用於加權 value，最終通過可學習的參數 gamma 控制注意力特徵與輸入特徵的融合比例。

3.區塊處理與無縫拼接技術

為了處理高分辨率圖像並減少顯存消耗，我採用了區塊處理技術，將大圖像分割成小區塊進行處理，然後使用無縫拼接技術將處理後的區塊重新組合。實現如下：

```
def create_weight_mask(size, device, mode='改進型高斯分佈'):
    x = torch.linspace(-1, 1, size)
    y = torch.linspace(-1, 1, size)
    X, Y = torch.meshgrid(x, y, indexing='ij')
    if mode == '高斯分佈':
        sigma = 0.5
        weight = torch.exp(-(X**2 + Y**2) / (2 * sigma**2))
    elif mode == '改進型高斯分佈':
        sigma_center = 0.7
        sigma_edge = 0.3
        dist = torch.sqrt(X**2 + Y**2)
        sigma = torch.ones_like(dist) * sigma_center
        edge_mask = (dist > 0.5)
        sigma[edge_mask] = sigma_edge
        weight = torch.exp(-(X**2 + Y**2) / (2 * sigma**2))
        weight = torch.clamp(weight, min=0.1)
    elif mode == '線性分佈':
        dist = torch.sqrt(X**2 + Y**2)
        weight = 1.0 - dist
        weight = torch.clamp(weight, min=0.1)
    elif mode == '餘弦分佈':
        dist = torch.sqrt(X**2 + Y**2).clamp(max=1)
        weight = torch.cos(dist * torch.pi/2)
        weight = torch.clamp(weight, min=0.1)
    elif mode == '泊松分佈':
        dist = torch.sqrt(X**2 + Y**2)
        weight = torch.exp(-dist * 3)
        weight = torch.clamp(weight, min=0.1)
    weight_tensor = weight.view(1, 1, size, size).to(dtype=torch.float32)
    return weight_tensor.to(device)
```

圖 29：權重遮罩生成函數實現，src/processing/NS_PatchProcessor.py

區塊處理中的關鍵挑戰是消除區塊邊界處的接縫效應。為解決這個問題，我實現了多種混合模式，包括高斯分佈、改進型高斯分佈、線性分佈、餘弦分佈和泊松分佈。其中，改進型高斯分佈通過不同的 σ 值處理中心區域和邊緣區域，確保平滑過渡[6]。

4. 損失函數設計 (以第 9 代 Kairitsu 寫實圖像模型示範)

4.1 JPEGRestorationLoss 的數學構成：

$$L_{total} = \lambda_1 \cdot L_{L1} + \lambda_2 \cdot L_{MSE} + \lambda_3 \cdot L_{SSIM} + \lambda_4 \cdot L_{edge} + \lambda_5 \cdot L_{hf} + \lambda_6 \cdot L_{color} + \lambda_{adv} \cdot L_{adv}$$

其中各分量為：

- $L_{L1} = \|G(x) - y\|_1$ (絕對誤差)
- $L_{MSE} = \|G(x) - y\|_2^2$ (均方誤差)
- $L_{SSIM} = 1 - SSIM(G(x), y)$ (結構相似性)
- $L_{edge} = \|\nabla G(x) - \nabla y\|_1$ (邊緣梯度差異)
- $L_{hf} = \|H(G(x)) - H(y)\|_1$ (高頻成分差異)
- $L_{color} = \|H_c(G(x)) - H_c(y)\|_1$ (色彩直方圖差異)
- $L_{adv} = -\log(D(G(x)))$ (對抗損失)

4.2 初始訓練階段分析 (Epochs 1-240)

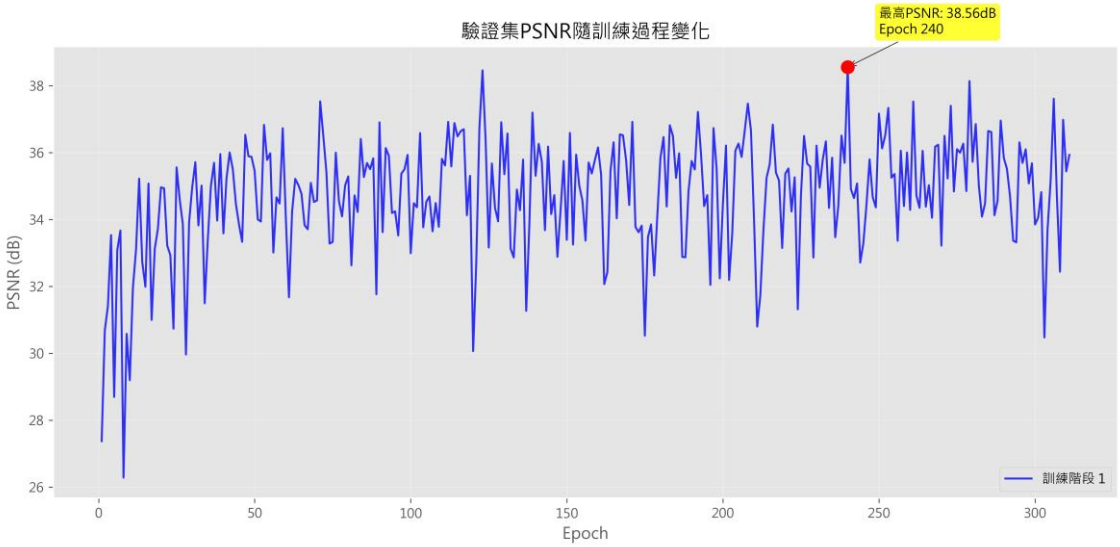

圖 30：Kairitsu 訓練過程中的 PSNR 變化

從 PSNR 進程圖可以看出，Kairitsu 模型在訓練約 240 個 Epochs 時達到最佳 PSNR 值約 38.5 dB，之後略有波動但維持在較高水平。這表明模型在寫實圖像還原上取得了出色的效果，並且驗證了損失函數設計的有效性。

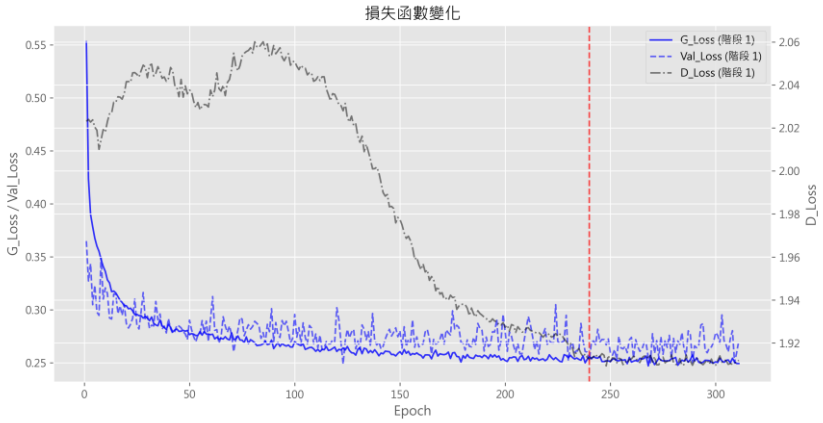

圖 31：Kairitsu 訓練中的損失函數變化

從損失函數變化圖可以觀察到，生成器損失(G_Loss)從初始的約 0.55 快速下降到 0.3 以下，並在後期穩定在 0.25 左右。驗證損失(Val_Loss)也呈現類似趨勢，證明模型泛化能力良好。鑑別器損失(D_Loss)則始終保持在 2.0-2.05 區間，表明 GAN 對抗訓練達到了良好的平衡狀態。

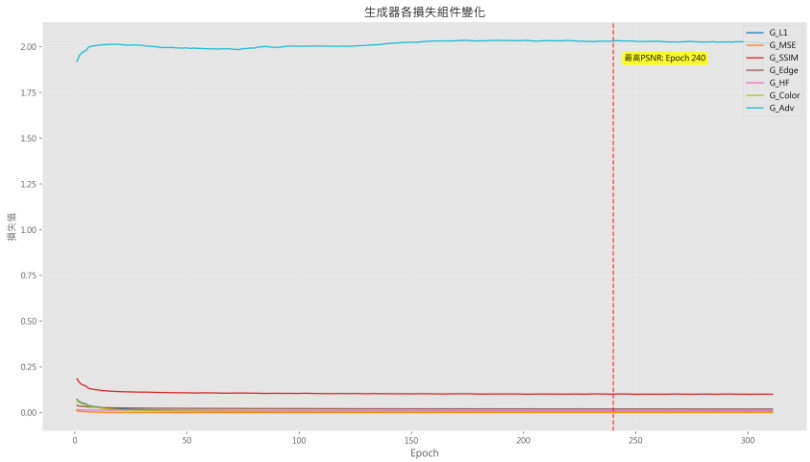

圖 32：生成器各損失分量的貢獻

生成器各損失分量圖揭示了更為細緻的訓練動態：

- SSIM 損失(G_SSIM)在訓練初期佔比最大(~0.18)，後期穩定在 0.10 左右，表明結構相似性一直是模型優化的重要目標
- L1 損失(G_L1)從 0.072 降至 0.008-0.01，下降幅度最大，反映了模型對整體內容還原的進步
- 邊緣損失(G_Edge)和高頻損失(G_HF)在訓練中期後保持相對穩定(分別約 0.02 和 0.01)，說明模型對細節和紋理的處理能力達到平衡

- MSE 損失(G_MSE)和色彩損失(G_Color)在整個訓練過程中持續下降，最終分別穩定在約 0.0002 和 0.004-0.005，表明模型在像素精確度和色彩還原方面持續改進

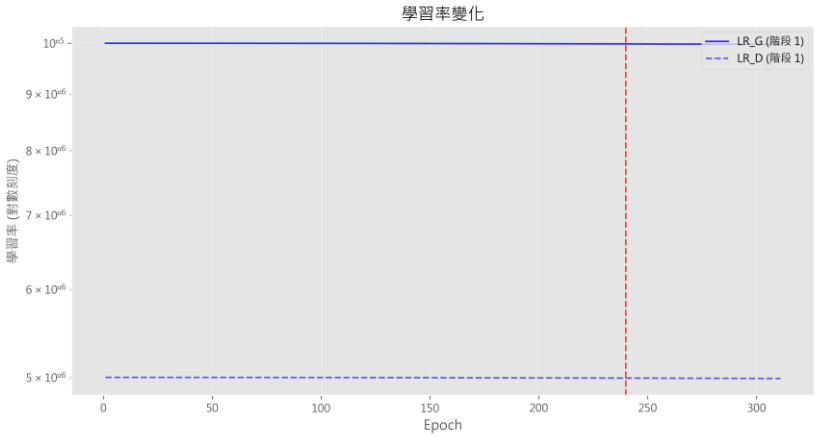

圖 33：學習率變化趨勢

學習率變化圖顯示了精心設計的學習率調度策略：生成器學習率(LR_G)從初始的 1e-5 緩慢降至約 9.98e-6，而判別器學習率(LR_D)從 5e-6 降至約 4.99e-6。這種緩慢衰減的策略有效防止了模型訓練後期的震盪，使其能夠在接近最優解時進行精細調整。

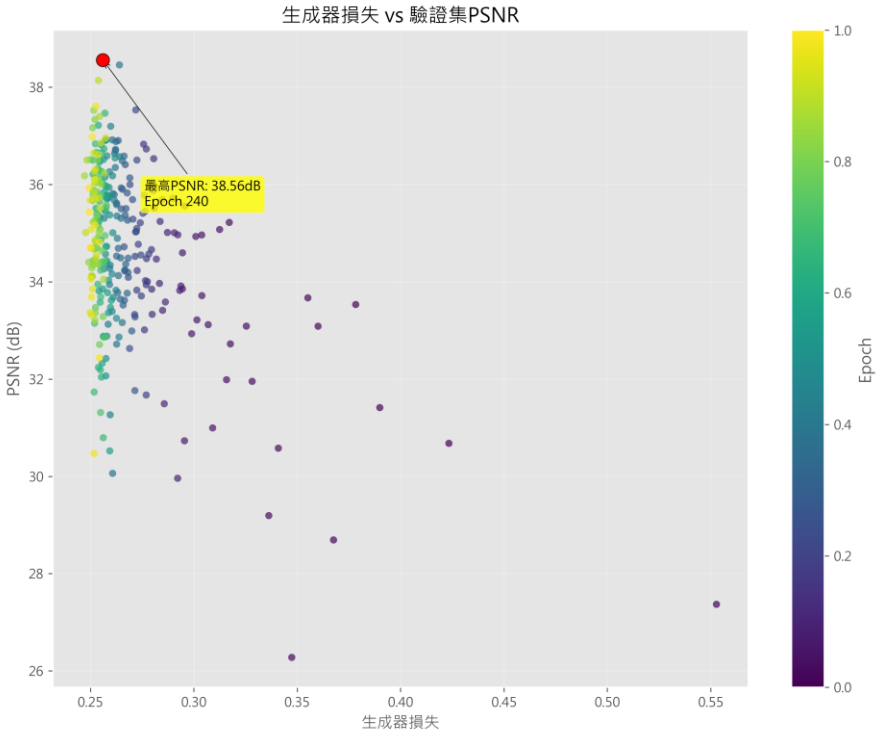

圖 34：生成器損失與 PSNR 關係

生成器損失與 PSNR 散點圖揭示了一個重要發現：訓練過程中最高 PSNR 值(38.5 dB 左右)出現時，生成器損失約為 0.25-0.26，並非訓練中的最低損失值。這印證了在圖像還原任務中，多目標優化的複雜性和必要性。我的多組件損失函數設計能夠在不同視覺質量指標間取得平衡，產生更符合人類視覺感知的高質量圖像，而不僅僅追求單一數值指標的最優。

4.3重啟訓練階段分析 (Epochs 241-323)

為了突破第 240 個 Epochs 時最佳 PSNR 值 38.5 dB，我將訓練集資料筆數繼續往上疊加從約 25K 張 HD 圖片往上堆到約 33K 張圖，進行了參數調整並重啟訓練：

--resume ./models/NS-IC-JPEG-Restoration-v9.1_best_psnr.pth.tar --num_epochs 1000 --min_lr 1e-9 --augment --learning_rate 5e-6

- 將學習率從之前的 1e-5 減至 5e-6
- 設置更低的最小學習率 1e-9，允許更細緻的參數調整
- 保留數據增強策略，確保模型泛化能力

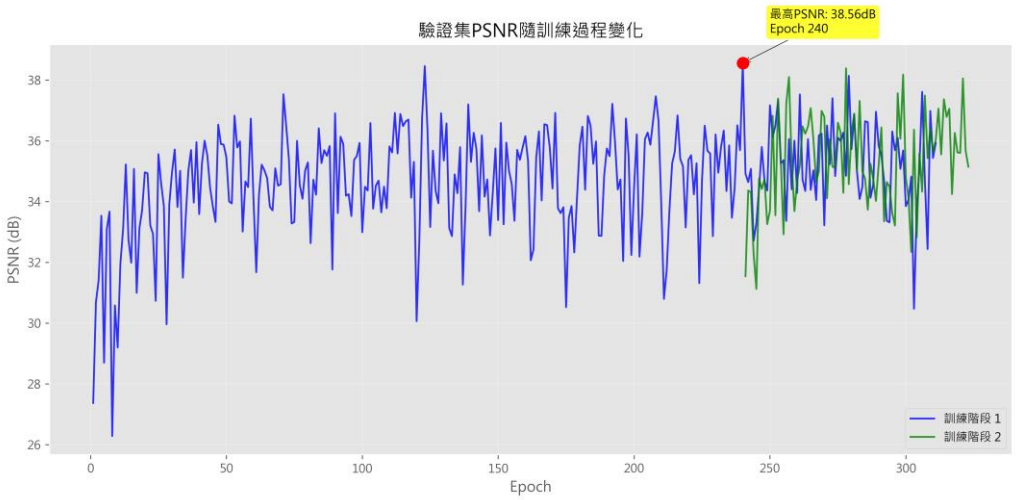

圖 35：重啟訓練後的 PSNR 變化

重啟訓練後的 PSNR 表現呈現出波動趨勢。在第 321 個 Epoch 時達到了重啟階段的最高值 38.05 dB，但仍未超過初始訓練階段的 38.56 dB。這表明降低學習率並重啟訓練並沒有幫助模型突破原有的性能上限，模型很可能已經達到了在當前架構和訓練數據條件下的最佳表現。

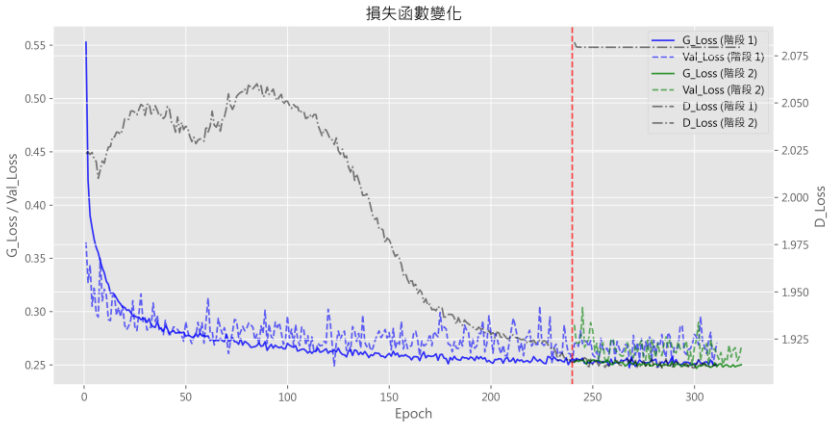

圖 36：重啟訓練後的損失函數變化

重啟訓練後，生成器損失(G_Loss)保持在 0.245-0.255 之間小幅波動，相比初始訓練階段更加穩定。判別器損失(D_Loss)則固定在 2.079 左右，顯示判別器已達到穩定狀態。這種穩定的損失曲線進一步證實了模型已接近收斂。

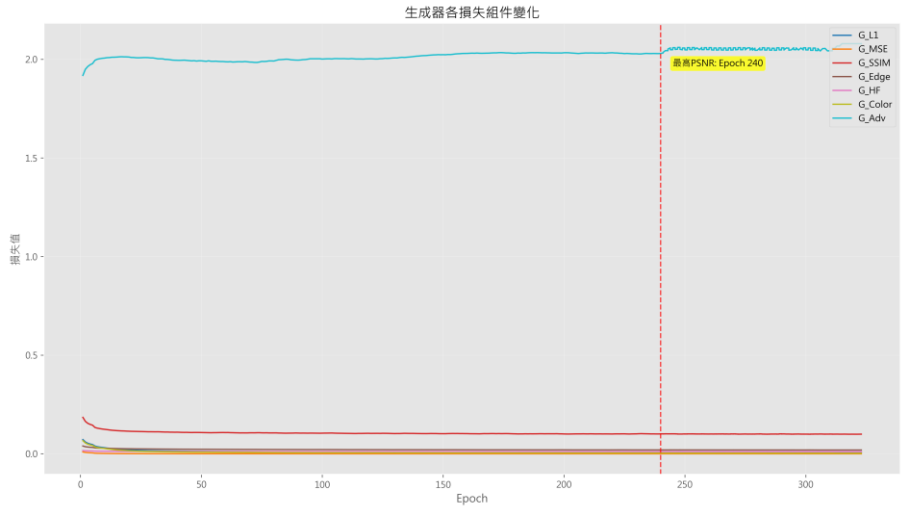

圖 37：重啟訓練後各損失分量變化

各損失分量在重啟訓練期間變化不大：

- G_L1 穩定在 0.008 左右
- G_SSIM 保持在 0.10 左右
- G_Edge 和 G_HF 分別維持在 0.019 和 0.011 附近
- G_MSE 和 G_Color 繼續保持在較低水平

這種穩定的分量表現說明模型對各類特徵的重建能力已經達到平衡，難以通過簡單的參數調整獲得顯著突破。

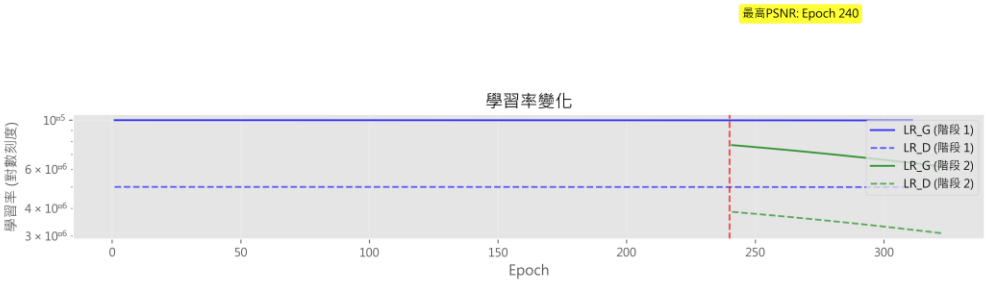

圖 38：重啟訓練後的學習率變化

學習率曲線顯示，重啟訓練時使用了更低的起始學習率(5e-6)，到第 323 個 Epoch 時進一步降至約 6.17e-6。這種保守的學習策略確實提高了訓練穩定性，但也限制了模型探索新的參數空間的能力。

4.4 訓練實驗總結

通過初始訓練和重啟訓練的對比分析，我們獲得了以下重要發現：

- 性能上限: Kairitsu 模型在約 240 個 Epoch 時達到了 38.56 dB 的 PSNR 峰值，之後即使調整學習率並重啟訓練，也未能突破這一上限。
- 參數調整的局限性: 僅調整學習率等超參數對於已接近收斂的模型幫助有限。要顯著提升性能，可能需要修改模型架構、擴充訓練數據或重新設計損失函數。
- 多目標優化的平衡: 在 JPEG 還原任務中，最高 PSNR 並不對應最低的總損失，這反映了不同損失分量之間的複雜交互關係。我們的多組件損失函數設計成功地在各種視覺質量指標間取得了平衡。
- 學習率策略: 緩慢降低的學習率策略對於模型的穩定訓練至關重要，但過低的學習率也可能限制模型突破性能瓶頸的能力。

這些發現不僅對模型的開發具有指導意義，也為其他圖像復原任務提供了寶貴的經驗。未來工作中，我將探索更先進的模型架構和損失函數設計，以突破當前的性能限制。

二、圖像品質評估模型設計 (圖像評分模型 NS-IQA)

1. 深度可分離卷積架構

為了在保持評分準確性的同時優化資源使用，我放棄了原本 NS-IQC 的 Transformer 架構，重新設計了一個輕量化的 CNN 模型，採用深度可分離卷積代替標準卷積，大幅降低參數量和計算需求：

```python
class DepthwiseSeparableConv(nn.Module):
    """深度可分離卷積層，將標準卷積分解為逐深度卷積和逐點卷積"""
    def __init__(self, in_channels: int, out_channels: int, kernel_size: int, stride: int, padding: int):
        super().__init__()
        self.depthwise = nn.Conv2d(in_channels, in_channels, kernel_size=kernel_size,
                                   stride=stride, padding=padding, groups=in_channels, bias=False)
        self.pointwise = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.depthwise(x)
        x = self.pointwise(x)
        return x
```

圖 39：深度可分離卷積實現，src/processing/NS_ImageQualityScorer.py

深度可分離卷積將標準卷積操作分解為兩個步驟：

1.逐深度卷積：對每個輸入通道分別應用單獨的卷積核，保持通道數不變

2.逐點卷積：使用 1×1 卷積進行通道間的信息融合和維度調整

逐深度卷積和逐點卷積。這種分解可以顯著降低計算複雜度，從 $O(MNK^2)$降低到 $O(K^2M + NM)$，其中 M 是輸入通道，N 是輸出通道，K 是卷積核尺寸[7]。這種技術使模型體積大幅縮小，但保持了不錯的性能水平。

2. 全局平均池化策略

在評分模型中，我放棄了傳統的多層全連接層分類器，而採用全局平均池化(Global Average Pooling)來減少參數量並提高模型的泛化能力：

```python
class NagatoSakuraImageQualityClassificationCNN(nn.Module):
    """長門櫻輕量化CNN模型，使用深度可分離卷積、批量歸一化和全局平均池化減少參數量。"""
    def __init__(self, dropout_rate: float = 0.5):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            DepthwiseSeparableConv(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            DepthwiseSeparableConv(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            DepthwiseSeparableConv(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            DepthwiseSeparableConv(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            DepthwiseSeparableConv(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            DepthwiseSeparableConv(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
        )
        # 全局平均池化
        self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout_rate),
            nn.Linear(512, 1)
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.global_avg_pool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

圖 40：圖像品質評估模型的完整架構，src/processing/NS_ImageQualityScorer.py

全局平均池化將每個特徵圖壓縮到單一值，代表該特徵在整個圖像中的平均激活程度。這不僅大幅減少了參數量，還能有效避免過擬合，並保持空間信息的完整性[8]。這種策略適合於圖像品質評估，因為它允許模型關注整體圖像的統計特徵，而非局部細節。

三、圖像分類模型架構 (推薦模型模型 NS-C)

1. EfficientNet 遷移學習策略

我採用 EfficientNet-B0 作為圖像分類模型的基礎架構，並通過遷移學習進行針對性調整：

```
num_classes = len(self.class_names)
self.model = models.efficientnet_b0(weights=None)
self.model.classifier[1] = nn.Linear(self.model.classifier[1].in_features, num_classes)
try:
    self.model.load_state_dict(torch.load(model_path, map_location=self.device, weights_only=True))
    logger.info("使用 weights_only=True 成功載入模型")
except TypeError:
    self.model.load_state_dict(torch.load(model_path, map_location=self.device))
    logger.info("使用兼容模式載入模型")
```

圖 41：EfficientNet 遷移學習實現，src/processing/NS_ImageClassification.py

EfficientNet 系列模型使用複合縮放法(Compound Scaling)，通過平衡網絡深度、寬度和圖像分辨率，實現了優異的精度和效率權衡[9]。通過將預訓練模型的最後分類層替換為適應我設定特定類別數量的新層，利用遷移學習保留底層特徵提取能力，同時針對特定任務進行優化。

2. 多編碼容錯標籤處理

在實際部署中，為了處理不同環境下可能出現的編碼問題，因為之前為了運行智譜 AI 和清華大學 KEG 實驗室聯合發佈的 GLM-4-0414 模型，導致我需要將 Windows 設定強制 UTF-8 編碼讓我發現了類別編碼錯誤的問題，為了避免再次出現編碼問題，我實現了多編碼嘗試的標籤讀取機制：

```
def load_model(self, model_path: Optional[str] = None, labels_path: Optional[str] = None) -> bool:
    """載入分類模型和標籤文件"""
    try:
        base_dir = os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
        default_models_dir = os.path.join(base_dir, "extensions", "Nagato-Sakura-Image-Classification", "models")
        if labels_path is None:
            labels_path = os.path.join(default_models_dir, "labels.txt")
        if not os.path.exists(labels_path):
            logger.error(f"找不到標籤文件: {labels_path}")
            return False
        encodings = ['utf-8', 'big5', 'gbk', 'cp950', 'latin-1']
        for encoding in encodings:
            try:
                with open(labels_path, 'r', encoding=encoding) as f:
                    self.class_names = [line.strip() for line in f.readlines()]
                logger.info(f"成功載入標籤文件: {labels_path} (使用 {encoding} 編碼)")
                break
            except UnicodeDecodeError:
                continue
```

圖 42：多編碼標籤處理，src/processing/NS_ImageClassification.py

這種設計充分考慮了跨平台部署時可能遇到的編碼差異問題，尤其是處理包含中文等多字節字符的標籤時。通過按優先順序嘗試多種常見編碼，系統可以最大限度地確保標籤文件的正確讀取，提高模型的適應性和穩定性。

四、混合精度計算技術

為了提高計算效率和減少顯存使用，我實現了自動適應的混合精度計算：

```python
def _should_use_amp(self, device):
    """自動檢測是否應使用混合精度計算"""
    if device.type != 'cuda':
        return False
    gpu_name = torch.cuda.get_device_name(device)
    logging.info(f"正在檢測GPU '{gpu_name}' 是否適合使用混合精度...")
    try:
        cuda_version = torch.version.cuda
        if cuda_version:
            cuda_major = int(cuda_version.split('.')[0])
            if cuda_major < 10:
                logging.info("CUDA版本低於10.0，禁用混合精度計算")
                return False
    except Exception as e:
        logging.warning(f"無法獲取CUDA版本信息: {e}")
    excluded_gpus = ['1650', '1660', 'MX', 'P4', 'P40', 'K80', 'M4']
    for model in excluded_gpus:
        if model in gpu_name:
            logging.info(f"檢測到GPU型號 {model} 在排除列表中，禁用混合精度")
            return False
    amp_supported_gpus = ['RTX', 'A100', 'A10', 'V100', 'T4', '30', '40', 'TITAN V']
    for model in amp_supported_gpus:
        if model in gpu_name:
            logging.info(f"檢測到GPU型號 {model} 支持混合精度計算")
            return True
    cc_match = re.search(r'compute capability: (\d+)\.(\d+)', gpu_name.lower())
    if cc_match:
        major = int(cc_match.group(1))
        minor = int(cc_match.group(2))
        compute_capability = float(f"{major}.{minor}")
        if compute_capability >= 7.0:
            logging.info(f"GPU計算能力 {compute_capability} >= 7.0，啟用混合精度")
            return True
    try:
        test_tensor = torch.randn(1, 3, 64, 64, device=device, dtype=torch.float32)
        with torch.no_grad():
            try:
                with torch.amp.autocast(device_type='cuda'):
                    _ = self.model(test_tensor)
                logging.info("混合精度測試成功，啟用混合精度計算")
                return True
            except Exception as e:
                logging.warning(f"混合精度測試失敗: {e}")
                return False
    except Exception as e:
        logging.warning(f"混合精度功能測試出錯: {e}")
    logging.info("無法確定GPU是否支持混合精度，為安全起見禁用")
```

圖 43：自動混合精度檢測函數，src/threads/NS_EnhancerThread.py

混合精度計算透過在適合的操作中使用較低精度(FP16)而在必要操作中保持較高精度(FP32)，可以顯著提高計算效率[10]。系統會自動檢測 GPU 型號和計算能力，只在支援的硬體上啟用此功能，確保兼容性和穩定性。

五、流程圖

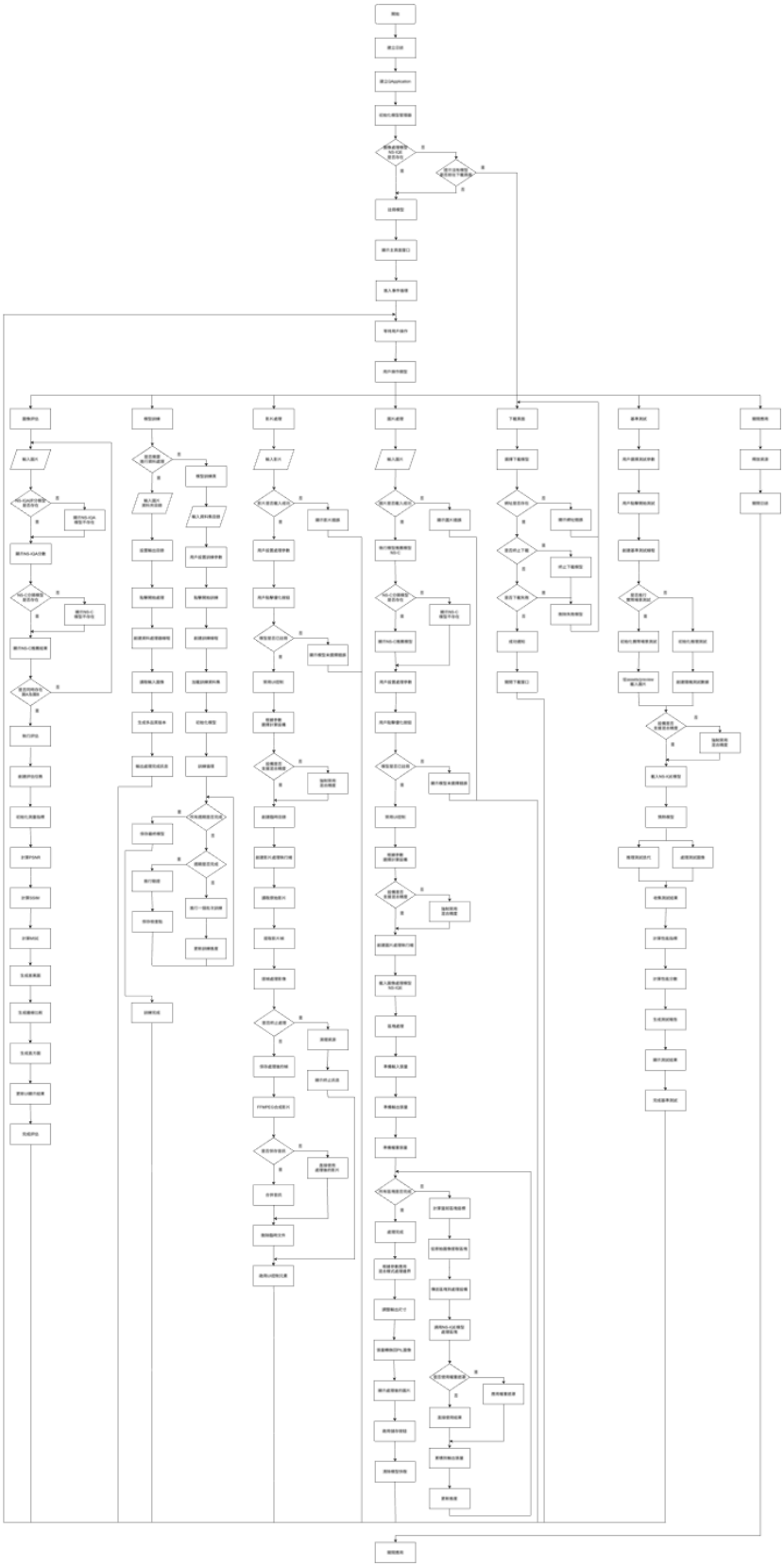

圖 44：完整流程圖

軟硬體分析

一、系統架構設計

本系統採用模組化設計，主要由以下幾個關鍵部分組成：

- 核心模型模組: 包含神經網路定義和推理功能
- 圖像處理模組: 處理圖像分塊、增強和重組
- 任務執行模組: 多線程處理和進度報告
- 使用者介面模組: 提供友好的交互控制

1.模組化設計與資料流

系統的模組化設計允許各部分獨立開發和測試，同時通過明確的接口進行交互。

主要資料流向如下：

輸入圖像 → 分塊處理 → 模型推理 → 區塊融合 → 輸出結果

2.多線程優化

為了提高界面響應性並充分利用系統資源，我們實現了多線程處理：

```python
class EnhancerThread(QThread):
    progress_signal = pyqtSignal(int, int)
    finished_signal = pyqtSignal(Image.Image, float)

    def __init__(self, model, image_path, device, block_size=256, overlap=64,
                 use_weight_mask=True, blending_mode='改進型高斯分佈', strength=1.0,
                 upscale_factor=1, target_width=0, target_height=0,
                 maintain_aspect_ratio=False, resize_mode="延伸至適合大小",
                 use_amp=None):
        super().__init__()
        self.model = model
        self.image_path = image_path
        self.device = device
        self.block_size = block_size
        self.overlap = overlap
        self.use_weight_mask = use_weight_mask
        self.blending_mode = blending_mode
        self.strength = strength
        self.upscale_factor = upscale_factor
        self.target_width = target_width
        self.target_height = target_height
        self.maintain_aspect_ratio = maintain_aspect_ratio
        self.resize_mode = resize_mode
        self.weight_mask_cache = {}
        try:
            model_device = next(model.parameters()).device
            if model_device != device:
                logging.debug(f"將模型從 {model_device} 移動到 {device}")
                self.model = model.to(device)
                if torch.cuda.is_available():
                    torch.cuda.empty_cache()
        except Exception as e:
            logging.error(f"檢查模型設備時出錯: {e}")
            self.model = model.to(device)
        if use_amp is None:
            self.use_amp = self._should_use_amp(device)
        else:
            self.use_amp = use_amp
        if device.type == 'cuda':
            gpu_name = torch.cuda.get_device_name(device)
            logging.info(f"使用GPU: {gpu_name}")
            logging.info(f"CUDA版本: {torch.version.cuda}")
            logging.info(f"混合精度計算: {'啟用' if self.use_amp else '禁用'}")
        else:
            logging.info(f"使用CPU作為計算設備")
```

圖 45：增強處理線程實現 1，src/threads/NS_EnhancerThread.py

```python
def run(self):
    try:
        enhanced_image = self.enhance_image_by_blocks()
        self.finished_signal.emit(enhanced_image, self.elapsed_time)
    except Exception as e:
        logging.error(f"處理圖片時出錯: {str(e)}")
        try:
            image = Image.open(self.image_path).convert("RGB")
            self.elapsed_time = 0
            self.finished_signal.emit(image, self.elapsed_time)
        except Exception as open_err:
            logging.error(f"無法開啟原圖: {str(open_err)}")
            error_img = Image.new("RGB", (400, 300), color=(50, 50, 50))
            self.finished_signal.emit(error_img, 0)
```

圖 46：增強處理線程實現 2，src/threads/NS_EnhancerThread.py

主要處理邏輯在後台線程中執行，通過信號機制與主界面通信，包括進度更新和結果傳遞。這種設計確保了即使在處理大圖像時，用戶界面也能保持響應。

3.記憶體管理與優化

處理高分辨率圖像時，記憶體管理至關重要。我實現了以下優化：

- 分塊處理: 將大圖像分割成可管理的小區塊，避免一次加載整個圖像到顯存
- 緩存控制: 實現權重掩碼緩存，避免重複計算
- 主動釋放: 在關鍵步驟後釋放不必要的張量和緩存

```python
def clear_cache(self):
    """清空模型快取，釋放記憶體和顯存"""
    try:
        logger.info("開始清理模型快取和釋放顯存...")
        for model_path, model in self.model_cache.items():
            try:
                logger.debug(f"將模型從顯存移至 CPU: {os.path.basename(model_path)}")
                model.cpu()
            except Exception as e:
                logger.error(f"將模型移至 CPU 時出錯: {str(e)}")
        self.model_cache.clear()
        self.current_model = None
        self.current_model_path = None
        for path in self.model_statuses:
            if self.model_statuses[path] == "active":
                self.model_statuses[path] = "registered" if path == self.registered_model_path else "available"
        import gc
        gc.collect()
        if torch.cuda.is_available():
            try:
                before_free = torch.cuda.memory_reserved() / 1024**2
                logger.info(f"CUDA 緩存清理前顯存保留: {before_free:.2f} MB")
            except Exception as e:
                logger.warning(f"無法獲取釋放前顯存資訊: {str(e)}")
            torch.cuda.empty_cache()
            try:
                after_free = torch.cuda.memory_reserved() / 1024**2
                logger.info(f"CUDA 緩存清理後顯存保留: {after_free:.2f} MB")
                logger.info(f"釋放了約 {before_free - after_free:.2f} MB 顯存")
            except Exception as e:
                logger.warning(f"無法獲取釋放後顯存資訊: {str(e)}")
        gc.collect()
        logger.info("模型快取清理完成")
        return True
    except Exception as e:
        logger.error(f"清空模型快取時出錯: {str(e)}")
        return False
```

圖 47：主動釋放記憶體，src/utils/NS_ModelManager.py

二、硬體需求分析

1.最低硬體需求

- CPU: 支援 AVX 指令集的處理器
- RAM: 2GB 以上系統記憶體
- GPU: *選用*支援 CUDA 的 NVIDIA 顯卡（至少 512MB 顯存）
- 儲存: 50MB 用於基本安裝(包含 NS-IQA 及 NS-C)，額外空間用於模型和圖像

2.推薦硬體配置

- CPU: 4 核心或更高的處理器
- RAM: 8GB 以上系統記憶體
- GPU: 3GB 以上支援 CUDA 的 NVIDIA 顯卡，最好支援 AMP (RTX 30 系)
- 儲存: 1GB 以上

3.GPU 架構對性能的影響

- NVIDIA Turing 架構開始完全支援混合精度計算 (不包含 GTX 16 系)
- RTX 系列顯卡可通過混合精度獲得顯著性能提升
- 顯存大小影響可處理的最大圖像尺寸

4.硬體自適應功能

系統能夠自動檢測並適應可用硬體，由於內容過長這邊放程式調用的方式，詳細可去 NS_DeviceInfo.py 查看 class SystemInfo 中 def collect_device_info() 的實現：

```
def get_system_info():
    """獲取系統和設備資訊"""
    return SystemInfo.collect_device_info()
```

圖 48：系統信息收集，src/utils/NS_DeviceInfo.py

測試結果

一、圖像質量評估

1. 客觀評估指標

我們採用多種指標評估圖像質量，包括：

- 峰值信噪比(PSNR): 衡量像素精確度
- 結構相似性指數(SSIM): 衡量結構保存程度
- 均方誤差(MSE): 衡量整體像素差異

```python
def evaluate_images(self, img1: Union[Image.Image, np.ndarray], img2: Union[Image.Image, np.ndarray], advanced: bool = False) -> Dict[str, Any]:
    if img1 is None or img2 is None:
        return {"error": "需要兩張有效的圖像進行比較"}
    results: Dict[str, Any] = {}
    try:
        results["psnr"] = self.calculate_psnr(img1, img2)
        results["ssim"] = self.calculate_ssim(img1, img2)
        results["mse"] = self.calculate_mse(img1, img2)
        results["difference_map"] = self.generate_difference_map(img1, img2)
        results["histogram_img1"] = self.get_image_histograms(img1)
        results["histogram_img2"] = self.get_image_histograms(img2)
        if advanced:
            results["hsv_histogram_img1"] = self.get_hsv_histograms(img1)
            results["hsv_histogram_img2"] = self.get_hsv_histograms(img2)
            results["edge_img1"] = self.detect_edges(img1)
            results["edge_img2"] = self.detect_edges(img2)
            edge_comparison_img, edge_similarity = self.compare_edges(img1, img2)
            results["edge_comparison"] = edge_comparison_img
            results["edge_similarity"] = edge_similarity
            hash_similarities = self.calculate_image_hash_similarity(img1, img2)
            results.update(hash_similarities)
            color_stats1 = self.color_analysis(img1)
            color_stats2 = self.color_analysis(img2)
            results["color_stats_img1"] = {f"{k}_img1": v for k, v in color_stats1.items()}
            results["color_stats_img2"] = {f"{k}_img2": v for k, v in color_stats2.items()}
    except Exception as e:
        print(f"圖像評估過程中發生錯誤: {e}")
        results["error"] = f"評估過程中發生錯誤: {str(e)}"
    results = {k: v for k, v in results.items() if v is not None}
    return results
```

圖 49：圖像評估函數，src/processing/NS_ImageEvaluator.py

2. 視覺評估工具

為了幫助用戶直觀比較處理前後的圖像質量，我結合多種視覺化評估工具：

- 分割視圖: 在同一畫面中顯示處理前後的圖像，通過滑動分割線比較差異
- 差異熱力圖: 生成視覺化的差異圖，突顯處理前後變化最大的區域
- 邊緣比較視圖: 專門比較圖像邊緣細節的變化

(具體效果可參考前方圖 18~20，以及 src/processing/NS_ImageEvaluator.py)。

二、性能測試

1. 基準測試模組

我實現了完整的基準測試模組，評估模型在不同硬體上的性能：

```python
def run_real_usage_test(self, model, device, width, height, num_images, block_size, overlap,
                        use_amp=None, progress_callback=None, step_callback=None):
    """長門櫻會用真實場景為主人測試模型性能，請耐心等待"""
    if step_callback:
        step_callback("長門櫻正在為主人初始化實際場景測試...")
    model.eval()
    self.stop_flag = False
    self.is_cpu_test = (device.type == 'cpu')
    warmup_count = 1
    gc.collect()
    if device.type == 'cuda':
        torch.cuda.empty_cache()
        log_gpu_memory(device, "實際場景測試開始前")
    if step_callback:
        step_callback("長門櫻正在為主人準備 assets/preview 中的測試圖片...")
    total_images_needed = num_images + warmup_count
    image_files = self._get_test_images(total_images_needed)
    images = []
    if not image_files:
        if progress_callback:
            progress_callback(0, 100)
        return {"error": "長門櫻無法從 assets/preview 目錄中找到測試用圖片，請主人確認目錄是否存在並包含圖片"}
    for img_path in image_files:
        try:
            img = Image.open(img_path).convert("RGB")
            img_array = np.array(img)
            img_array = np.clip(img_array, 0, 255)
            img = Image.fromarray(img_array.astype(np.uint8))
            img = img.resize((1280, 640), Image.Resampling.LANCZOS)
            images.append(img)
        except Exception as e:
            logger.error(f"長門櫻加載圖片 {img_path} 時遇到了困難: {str(e)}")
            if progress_callback:
                progress_callback(0, 100)
            return {"error": f"加載測試圖片失敗了，長門櫻很抱歉: {str(e)}"}
    if not images:
        if progress_callback:
            progress_callback(0, 100)
        return {"error": "長門櫻無法處理任何測試圖片，請主人檢查圖片格式是否正確"}
    if len(images) < total_images_needed:
        logger.warning(f"長門櫻只找到 {len(images)} 張圖片，少於需要的 {total_images_needed} 張")
        while len(images) < total_images_needed:
            images.append(images[len(images) % len(image_files)])
    if use_amp is None and device.type == 'cuda':
        use_amp = self._should_use_amp(device)
    else:
```

圖 50：實際場景測試函數，src/processing/NS_Benchmark.py

2. 不同設備性能比較

表 1：不同設備使用標準精度處理圖像的性能比較表

處理設備	平均處理時間	標準精度	性能分數
AMD Ryzen 7 4800HS	23.83/s	0.03MP/s	564
AMD Ryzen 5 5600	27.70/s	0.03MP/s	485
Intel Core i9 13900K	11.42/s	0.07MP/s	1178
NVIDIA GTX 1660Ti Max Q	2.52/s	0.31MP/s	4853
NVIDIA RTX 3070	1.00/s	0.78MP/s	12176

處理設備	平均處理時間	混合精度	性能分數
NVIDIA RTX 3070	0.61/s	1.26MP/s	19762

表 2：不同設備啟用混合精度的性能表

測試結果顯示，NVIDIA RTX 3070 開啟混合精度相比未開啟後性能大幅度提升，使用 256×256 區塊大小並設置重疊 128 大小時，處理 1280×640 的圖片從原本的 1 秒縮短至 0.61 秒，縮短了足足 39%的處理時間，性能分數也從 12176 提升至 19762，提升幅度約為 62.3%。

3. 區塊大小與性能關係測試

我測試了 RTX 3070 在處理 1280×640 圖片不同區塊大小對性能的影響：

表 3：處理區塊大小比較表

區塊大小	重疊大小	處理時間
128×128	16	0.66/s
128×128	32	0.92/s
128×128	64	1.49/s
256×256	16	0.32/s
256×256	32	0.32/s
256×256	64	0.48/s
256×256	128	0.61/s
512×512	16	0.41/s
512×512	32	0.41/s
512×512	64	0.41/s
512×512	128	0.54/s
512×512	256	0.54/s

結果表明，區塊大小根據圖片大小設置恰當能減少分塊數量加速計算，減少重疊可有效加快計算速度，但有可能會降低處理質量。

4. 模型適配性測試

我針對同一動畫圖像，測試了專用模型與泛用模型的適配性：

表 4：專用模模型與泛用模型效果比較表

圖像類型	PSNR	SSIM	AI 評分 (NS-IQA-K)
原始圖	N/A	N/A	101.50
q10 畫質輸入圖	26.99	0.8175	6.19
Kyouka	30.55	0.9177	82.88
Kyouka-LQ	31.04	0.9264	89.25

結果證明在處理專用場景，即使是訓練結果 PSNR 更低的 Kyouka-LQ(310)還是成功擊敗了泛用型的的 Kyouka(314)模型，專用模型對於特定場景可以消耗更少的時間訓練、更低的精確度就能達成超越泛用模型的效果，這也驗證了我選擇將模型連接組合的正確性，透過專門的分類模型進行圖片的分類，推薦出最佳的專用模型處理特定場景能達到更好的效果，而非訓練一個泛用模型，除非場景過於複雜或著沒有適合的專用模型。

5. 模型處理專用場景測試

前面提過除了處理動畫 JPEG 壓縮的 Kyouka《鏡花》模型以外，針對動畫馬賽克還原以及寫實畫面 JPEG 壓縮還原我分別有 Ritsuka《斷律》以及 Kairitsu《界律》處理相關的場景，以下次試圖皆並非包含在訓練集中。

5.1Ritsuka《斷律》的測試

表 5：Ritsuka-HQ 處理不同馬賽克大小圖片比較表

圖像類型	PSNR	SSIM
原始圖	N/A	N/A
Q70 畫質輸入圖 (PIXEL4)	22.58	0.7319
Q80 畫質輸入圖 (PIXEL3)	23.07	0.7755
Q90 畫質輸入圖 (PIXEL2)	27.24	0.8959
Ritsuka-HQ (Q70)	31.26	0.9571
Ritsuka-HQ (Q80)	31.63	0.9634
Ritsuka-HQ (Q90)	36.77	0.9870

結果表明 Ritsuka《斷律》模型對於動畫馬賽克還原具有良好的處理能力，能有效將向素等寬馬賽克圖片還原回原始圖片，並且對於 2~4 格的馬賽克圖片都能有效還原細節，以下提供樣片參考。

圖 51：Q70 畫質輸入圖 (PIXEL4)

圖 52：Ritsuka-HQ 處理後 Q70

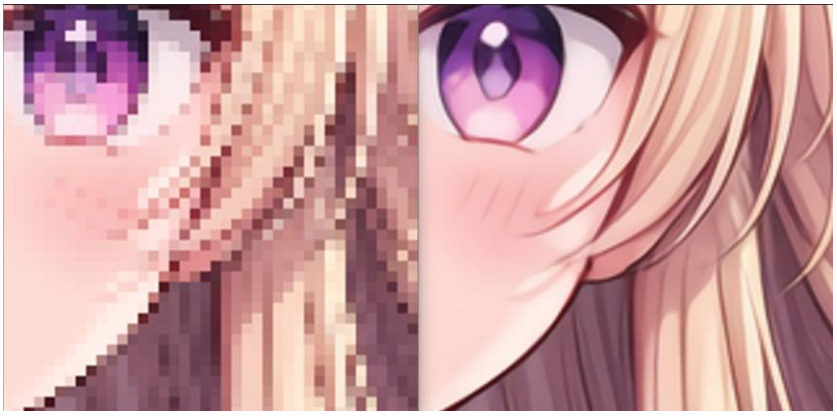

圖 53：Ritsuka-HQ Q70 處理細節對比 1：左處理前，右處理後

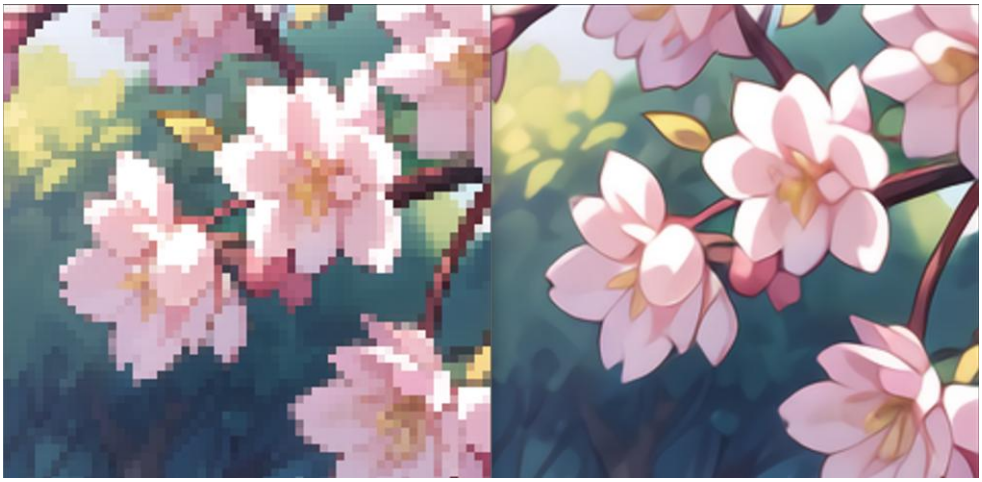

圖 54：Ritsuka-HQ Q70 處理細節對比 2：左處理前，右處理後

5.2 Kairitsu《界律》測試大一入學照 (由 NOKIA 3 拍攝，模擬原畫質不佳又被壓縮的情況)

為了節省測試時間所以畫質以 20 為間隔快速測試

表 6：Kairitsu 處理不同寫實 JPEG 壓縮圖片比較表

圖像類型	PSNR	SSIM
原始圖	N/A	N/A
Q10 畫質輸入圖	26.54	0.7403
Q30 畫質輸入圖	30.65	0.8686
Q50 畫質輸入圖	32.46	0.9078
Q70 畫質輸入圖	34.63	0.9405
Q90 畫質輸入圖	34.96	0.9446
Kairitsu (Q10)	27.33	0.7649
Kairitsu (Q30)	30.38	0.8627
Kairitsu (Q50)	31.42	0.8925
Kairitsu (Q70)	31.93	0.9034
Kairitsu (Q90)	32.12	0.9090

結果表明 Kairitsu《斷律》泛用型模型對於低畫質圖片擁有較好的處理能力，但對於中高畫質效果不如預期，根據上方製作理論探討章節中，圖像處理模型設計中的第 4 節損失函數設計中表明了目前模型架構深度不足再學習到更多細節處理的部份之前就達到飽和，一種是限制畫質範圍，訓練專用模型讓模型更加針對畫質學習，另一種就是加深模型深度能學習更多細節，但我沒時間再對模型進行修改了，因為光是訓練及調適模型就至少需要 2 個禮拜以上甚至是 1 個多月，而且這是在沒調整模型架構的前提下，以下提供樣片參考。

圖 55：Kairitsu Q10 畫質輸入圖

圖 56：Kairitsu 處理後 Q10

圖 57：Kairitsu Q10 處理細節對比 1：左處理前，右處理後

圖 58：Kairitsu Q10 處理細節對比 2：左處理前，右處理後

結論

通過本專題的研究與實現，我成功設計並開發了一個高效、靈活的圖像品質增強系統。

1. 模型架構: 基於殘差密集網路(RRDB)和自注意力機制的模型架構在圖像細節恢復方面表現優異。
2. 優化技術: 分塊處理和混合精度計算等優化技術大幅降低了顯存需求，使低端設備也能夠處理高解析度圖像，同時提高了處理速度。在 RTX 3070 上，混合精度計算將處理時間縮短了 39%，性能大幅提升。
3. 多樣化損失函數: 針對不同任務設計的多樣化損失函數組合，顯著提高了對特定場景圖像的處理效果。實驗證明，多目標優化策略能夠平衡各種視覺質量指標，產生更符合人類視覺感知的高質量圖像。
4. 用戶體驗: 直觀的用戶界面和豐富的視覺化工具使專業和非專業用戶都能有效地使用系統，並能快速地評估和處理圖像。分割視圖、差異熱力圖等工具提供了直觀的質量比較方式。
5. 硬體適應性: 系統的自適應特性使其能夠在各種硬體配置上高效運行，從入門級設備到高端設備都能獲得良好的使用體驗。自動檢測 GPU 能力並優化處理策略的功能大大提高了系統的通用性。

總體而言，本專題實現了高效、高質量的圖像增強系統，不僅在視覺質量上達到了顯著的提升，還通過多種優化技術提高了計算效率，使系統在實際應用中具有很高的實用性和可擴展性。

建議

基於本專題的實施經驗和測試結果，我提出以下建議以供未來改進程式參考：

1.改善模型之間的連接，由於開發時間緊迫導致模型之間連接度不夠高，很多地方都需要人工調整，理想狀況是當輸入 A 圖片先經由推薦模型選擇處理圖片的模型，假設先取前 2 高分的圖片處理模型，接著維持現有的圖片處理流程分塊處理圖片並拼接回完整圖片，並將圖片暫存到 Temp，接著透過評分模型對圖像進行評分，並保留分數較高的圖片輸出，就能完美全自動最佳化圖片，當然理想狀態是連結的每一個模型都能正確發揮作用，並保留像現在一樣半自動的處理方式(避免模型誤差無法手動指定處裡模型。

2.提高模型的深度，目前模型根據我長時間訓練的結果發現，基本上目前的第 9 代模型基本上在 PSNR 約為 38.5 dB 之後就難以再成長，不管是調整訓練策學習策略、損失函數還是添加更多訓練集都無法更進一步提升模型，推測模型達到訓練飽和可能需要進行架構調整或著提高 RRDB 或模型深度才有辦法在更進一步，又或著是使用外部的損失函數例如:VGG 來改進模型的訓練效果，但由於開發的時間限制(以目前的架構 16 RRDB, 64 features 訓練一個 25K 資料集模型用 RTX 3070 來算大概需要整整 5 天還不含測試參數浪費的時間，如果再修改訓練時間會成指數成長來不及弄)以及為了兼容舊版本的模型所以放棄。

3.訓練更多樣的模型，和上面的理由一樣沒有足夠的時間，不然我想要嘗試訓練專門針對 Youtube 壓縮影片進行模型特化，首先準備原片上傳 youtube 後透過 youtube dlp 等下載器下載，在透過影片隔幀擷取並打上畫質標記，讓下載的影片圖片還原成上傳前的影片圖片獲取特徵，以達到針對場景還原的效果，又或者是圖片添加噪點訓練降噪模型，或調整圖片的對比度著作色彩還原模型等。另外我在開發時曾想過「偽光線追蹤模型」，透過輸入遊戲光線追蹤開啟以及關閉的圖片，讓關閉光線追蹤的圖片還原成開啟光線追蹤的圖片達到偽光追的效果，但是因為素材不足，且認為模型還原能力可能不夠好放棄。

4.影片處理增加第三維度(時間)，目前程式處理影片的方式說到底就只是進行逐幀擷取然後經由圖片處裡模型處裡再透過 FFMPEG 還原，缺乏時間線這個概念，可能會導致幀與幀之間缺乏上下文的關係使效果不理想，可以更改模型根據訓練輸入影片的幀數與畫質來讓模型更好的處理連續影片。

5.結合擴散模型，我認為除了可以結合 Transformer 模型獲取圖片上下文、時間線以及全域特徵以外，還可以結合擴散模型達到生成的效果，以更好的還原圖片缺失的訊息，又或著可稱之為讓 AI 腦補原來圖片的效果。

參考文獻

[1] Xintao Wang, Liangbin Xie, Chao Dong, and Ying Shan. Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic Data. In arXiv:2107.10833 [eess.IV], 2021.

[2] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-Attention Generative Adversarial Networks. In arXiv:1805.08318 [stat.ML], 2018.

[3] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep Learning Face Attributes in the Wild. arXiv:1411.7766 [cs.CV], 2014.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV], 2015.

[5] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-local Neural Networks. arXiv:1711.07971 [cs.CV], 2017.

[6] Chen Chen, Qifeng Chen, Jia Xu, and Vladlen Koltun. Learning to See in the Dark. arXiv:1805.01934 [cs.CV], 2018.

[7] Sebastian Bosse, Dominique Maniry, Klaus-Robert Müller, Thomas Wiegand, and Wojciech Samek. Deep Neural Networks for No-Reference and Full-Reference Image Quality Assessment. arXiv:1612.01697 [cs.CV], 2016.

[8] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network. arXiv:1312.4400 [cs.NE], 2013.

[9] Mingxing Tan, and Quoc V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv:1905.11946 [cs.LG], 2019.

[10] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. arXiv:1710.03740 [cs.AI], 2017.