UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

LICENCIATURA EN TECNOLOGÍAS PARA LA INFORMACIÓN EN CIENCIAS

ESCUELA NACIONAL DE ESTUDIOS SUPERIORES,
UNIDAD MORELIA

COLORACIÓN AUTOMÁTICA DE DIBUJOS USANDO REDES NEURONALES CONVOLUCIONALES

T E S I S

QUE PARA OBTENER EL TÍITULO DE

LICENCIADO EN TECNOLOGÍAS PARA LA INFORMACIÓN EN CIENCIAS

P R E S E N T A

CARLOS CORTÉS MÉNDEZ

DIRECTOR DE TESIS: DR. MIGUEL RAGGI PÉREZ

MORELIA, MICHOACAN **JUNIO, 2022**

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
ESCUELA NACIONAL DE ESTUDIOS SUPERIORES UNIDAD MORELIA
SECRETARÍA GENERAL
SERVICIOS ESCOLARES

MTRA. IVONNE RAMÍREZ WENCE
DIRECTORA
DIRECCIÓN GENERAL DE ADMINISTRACIÓN ESCOLAR
P R E S E N T E

Por medio de la presente me permito informar a usted que en la **sesión ordinaria 06** del **H. Consejo Técnico** de la Escuela Nacional de Estudios Superiores (ENES) Unidad Morelia celebrada el día **23 de marzo del 2022**, acordó poner a su consideración el siguiente jurado para la presentación del Trabajo Profesional del alumno **Carlos Cortés Méndez** adscrito a la Licenciatura en **Tecnologías para la Información en Ciencias** con número de cuenta **417012629,** quien presenta la tesis titulada: "Coloración automática de dibujos usando Redes Neuronales Convolucionales", bajo la dirección como **tutor** del Dr. Miguel Raggi Pérez.

El jurado queda integrado de la siguiente manera:

Presidente:	Dr. Sergio Rogelio Tinoco Martínez
Vocal:	Dra. Karina Mariela Figueroa Mora
Secretario:	Dr. Miguel Raggi Pérez
Suplente 1:	Dra. Marisol Flores Garrido
Suplente 2:	Dra. Adriana Menchaca Méndez

Sin otro particular, quedo de usted.

A t e n t a m e n t e
"POR MI RAZA HABLARA EL ESPIRITU"
Morelia, Michoacán a 08 de junio del 2022.

DRA. YUNUEN TAPIA TORRES
SECRETARIA GENERAL

Agradecimientos Institucionales

Me gustaría agradecer a la Universidad Nacional Autónoma de México, así como a la Escuela Nacional de Estudios Superiores Unidad Morelia por su compromiso con la educación y todo el apoyo que me brindaron durante mis estudios en sus aulas. Además, me gustaría también agradecer a la licenciatura en Tecnologías para la Información en Ciencias por todos las oportunidades y herramientas que me brindaron durante mis años de estudio. Asimismo, agradezco profundamente al laboratorio de análisis avanzado de datos y al proyecto UNAM-PAPIIT IN120220 por permitirme utilizar los servidores de cómputo necesarios, sin cuales realizar este trabajo hubiera sido imposible.

Agradecimientos Personales

Dedicada a mis padres por todo su apoyo constante e incansable.

Dedicada a mi asesor por todo su esfuerzo y tenacidad con este trabajo.

Dedicada a Frida por siempre inspirarme a ser mejor.

Dedicada a mis amigos por siempre hacerme sonreir.

Índice general

Índice de figuras

Resumen

En este trabajo se presenta el diseño y entrenamiento de una red convolucional que colorea automáticamente dibujos en *sketch* (boceto, bosquejo) (Figura 1.1). Los modelos presentados son relativamente pequeños dados los límites del equipo de cómputo del que se dispone, pero con estas ideas podrían entrenarse modelos más complejos. El principal aporte de este trabajo radica en que el entrenamiento se realiza utilizando espacios de colores diferentes a RGB, en los cuales se puede emplear directamente la saturación, una estrategia que ha sido poco explorada. Para esto se crearon dos espacios de colores, llamados H_xH_ySV y XYV, con los cuales se muestra experimentalmente que los modelos tienden a producir resultados más coloridos. Finalmente, el código necesario para limpiar los datos, así como entrenar los modelos y los pesos entrenados de estos están disponibles en GitHub `https://github.com/alcros33/SketchColorizaion`.

Abstract

This work addresses the design and training of a convolutional neural network that colorizes sketches (Figure 1.1). The presented models are relatively small given the tight schedules and limit of computing power available, however, these ideas could be used to train a more complex model. The main contribution of this work is the usage of alternative color spaces other than RGB on which changes to the saturation can be addressed directly. For this purpose, two color spaces were created, which we called: H_xH_ySV and XYV. We show experimentally that the usage of these spaces tends to produce more colorful results. All the code necesary to clean the data, train the model and also pre-trained weights is available at GitHub `https://github.com/alcros33/SketchColorizaion`.

Capítulo 1

Introducción

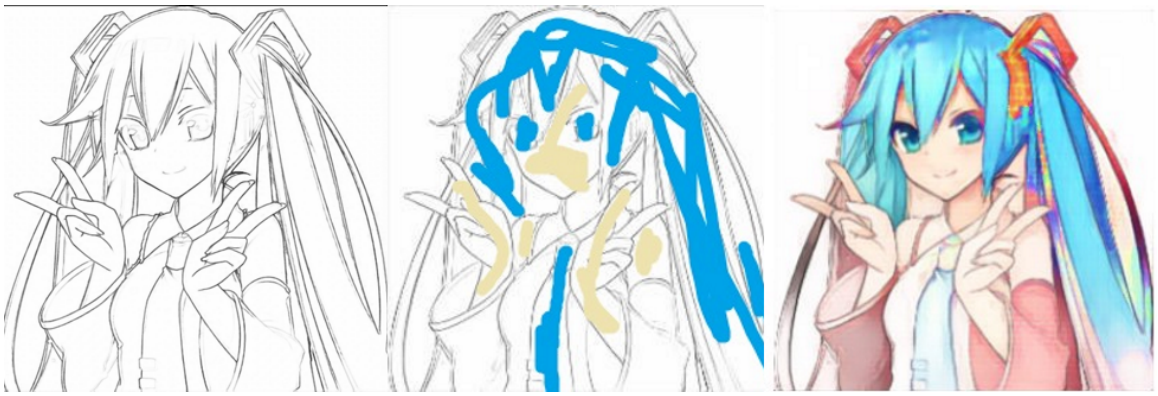


Figura 1.1: Ejemplo de resultado de coloración. De izquierda a derecha: entrada, referencia de color, resultado.

En el primer capítulo se introduce el trabajo, sus limitaciones y sus antecedentes. En el segundo y tercer capítulo se exponen la teoría y las técnicas necesarias que se consideraron para la construcción del modelo. En el cuarto capítulo se describe la estructura, funcionamiento y uso de las herramientas de software que se utilizaron para la implementación y entrenamiento del modelo. En el quinto capítulo se expone con detalles la composición del modelo propuesto, así como el razonamiento detrás de su construcción. En el sexto capítulo se describen aspectos técnicos y puntuales del proceso de entrenamiento del modelo, así como de los resultados obtenidos. En el séptimo capítulo se hace un análisis cualitativo de los resultados, sus problemas, y se proponen mejoras a estos como parte de un trabajo a futuro.

1.1. Antecedentes

Las redes neuronales convolucionales han producido muy buenos resultados en diversas áreas del procesamiento automático de imágenes (por ejemplo: clasificación, segmentación, detección, etc). Cuando se considera la tarea de transformar una imagen en otra, la red más importante es la UNet (Ronneberger et al., 2015) que se sigue tomando como base para la construcción de este tipo de modelos. El ejemplo más conocido de esta tarea es el modelo *Pix2Pix* que transforma imágenes de un conjunto a otro (Figura 1.2) (Isola et al., 2017). Sin embargo, gran parte del éxito que han tenido los modelos generadores de imágenes en la última década es gracias a la inclusión del entrenamiento por redes neuronales generativas adversarias (*generative adversarial networks*, GANs a partir de ahora), en el cual la función de pérdida se define en términos de otra red (Goodfellow et al., 2014).

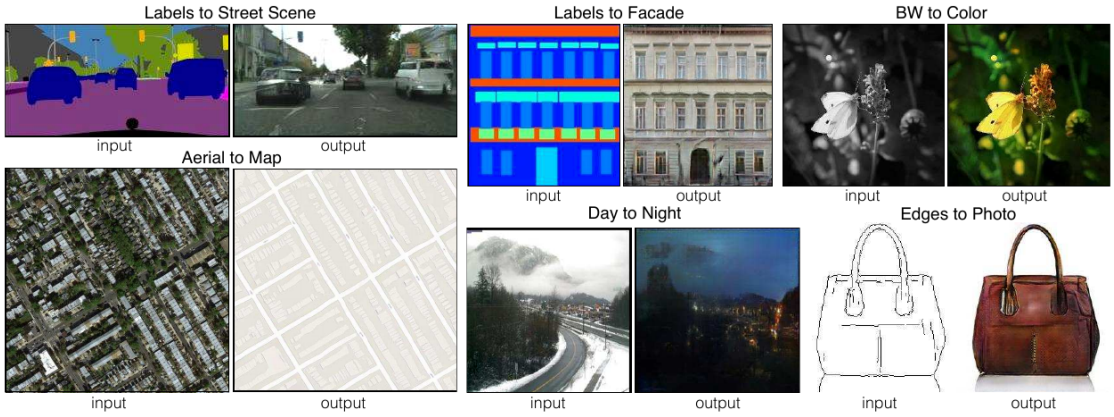


Figura 1.2: Ejemplo de distintas instancias entrenadas de Pix2Pix (Isola et al., 2017).

Un ejemplo más afín al problema que se trata en este trabajo es el de colorear fotografías en blanco y negro. De los múltiples modelos que se han desarrollado para colorear fotografías, el más exitoso ha sido la red *deoldify*, la cual produce resultados sorprendentemente realistas (Figura 1.3) (Antic, 2018). Parecería entonces que el problema de colorear dibujos debería de ser solucionado por deoldify también; sin embargo, al entrenar deoldify directamente en dibujos este produce resultados muy poco coloridos (Figura 1.4). Esto debido a que hay una diferencia fundamental entre los bocetos y las fotografías en blanco y negro: si en una fotografía se detecta una forma, figura o textura en particular, se puede reducir el conjunto de colores posibles para dicha área de la imagen a solo unos pocos (por ejemplo: plantas $\rightarrow$ tonos de verde, madera $\rightarrow$ tonos de café, etc.).

Por otro lado, colorear un boceto presenta varios problemas: primero no existen texturas que ayuden a determinar el color, además no existen sombras de gris que ayuden a identificar la iluminación, y finalmente, la tonalidad del color puede ser

independiente de la figura que lo contiene. Esta diferencia inherente entre los datos provoca que, aplicar directamente modelos de coloreado de fotografías, tienda a producir resultados con alto contenido de sepias (valores muy cercanos a la media).

Figura 1.3: Ejemplo de coloración de fotografías usando la red neuronal *deoldify* (Antic, 2018).

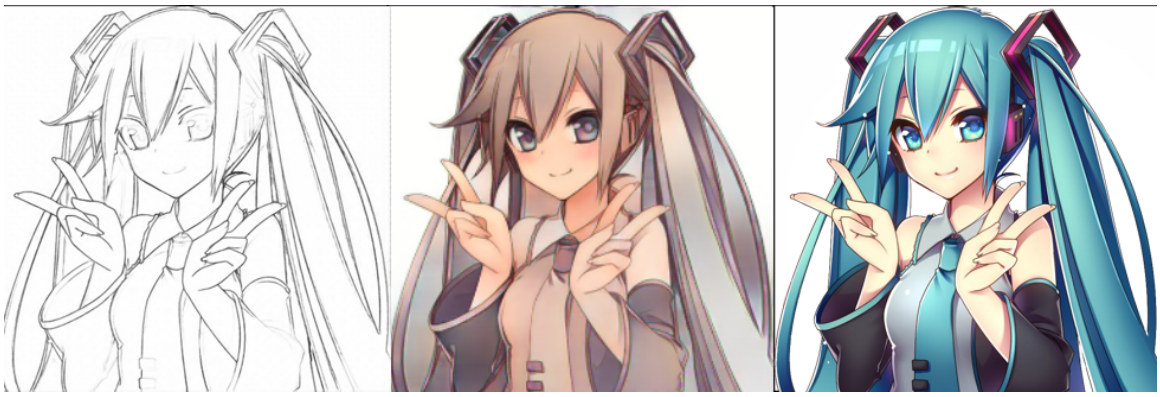


Figura 1.4: Resultado de coloración usando la red *deoldify* entrenada en el conjunto de imágenes *danbooru*. De izquierda a derecha: entrada, salida, imagen real.

En el caso del procesamiento de dibujos usando redes neuronales se puede observar un sesgo predominante a utilizar dibujos estilo anime. Esto es principalmente debido a la gran disponibilidad de conjuntos de dibujos con dicho estilo, siendo *danbooru* el más utilizado gracias a su tamaño, calidad y gratuidad (Anonymous, community, y Branwen, 2021). Entre estos trabajos podemos encontrar aquellos que aprovechan las etiquetas asociadas a los dibujos en el conjunto de datos para hacer multi-clasificación (Saito y Matsui, 2015), (Vallet y Sakamoto, 2015), (Baas, 2019) y que, según reportan,

tienen precisión considerable. Más relevantes para este trabajo son los modelos que generan dibujos como se puede observar en el trabajo de (Jin et al., 2017) en donde se generan personajes de anime aleatoriamente utilizando GANs para el entrenamiento.

Los modelos más afines a este trabajo son las propuestas de solución al problema de colorear bocetos. El primero que aparece cronológicamente es el de *style2paints* en 2017, en el cual se construye una red que recibe un dibujo ya coloreado, además del boceto que se pretende colorear, con la intención de que la red obtenga los colores de este y los aplique al boceto (según se describe en el artículo, una especie de *transferencia de estilo*) (L. Zhang et al., 2017). En la Figura 1.5 se observan algunos resultados reportados en el artículo y se puede observar que su calidad aún no es la de un resultado definitivo. Poco después, Hensman y Aisawa proponen un mecanismo distinto: se *sobreajusta* la red de manera controlada para que memorice los colores de un personaje en específico y, que de esta forma, sea capaz de colorear más instancias del personaje en particular. Tal acercamiento conlleva resultados mucho mejores; sin embargo, tiene la desventaja de que es necesario entrenarlo para cada instancia de personaje y estilo de dibujo que se desee colorear (un proceso que, según reportan, toma aproximadamente 15 minutos en una GPU NVIDIA GeForce GTX 1080Ti) (Hensman y Aizawa, 2017). Estos últimos detalles lo vuelven muy poco viable para emplearse en un ambiente de producción.

Figura 1.5: Tres ejemplos de coloración de *style2paints* (L. Zhang et al., 2017). Bocetos en blanco y negro: entradas; imágenes pequeñas a color: referencias; imágenes a color grandes: resultados.

El modelo con resultados más satisfactorios que pudimos encontrar en la literatura es el de (Ci et al., 2018), en el cual utilizan pistas de color para determinar el dibujo. En otras palabras, un usuario debe interactuar con una interfaz y colocar puntos de colores para cada sección del dibujo. Posteriormente, la red procede a utilizar estos colores para rellenar los espacios en blanco agregando sombras e iluminación. A pesar de que los resultados son sumamente buenos (Figura 1.6), no se considera como una solución definitiva al problema en cuestión, puesto que involucra un mayor esfuerzo por parte del usuario para realizar el coloreado.

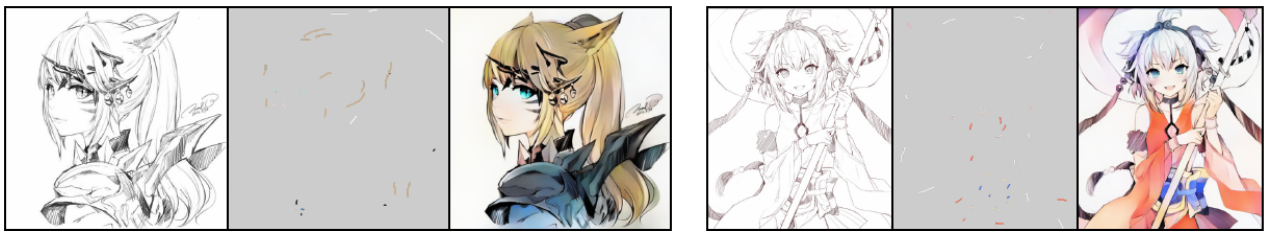


Figura 1.6: Dos ejemplos de coloración del modelo de (Ci et al., 2018). De izquierda a derecha: entrada, referencia de color y resultado.

1.2. Justificación

En algunos ámbitos artísticos (como la producción de comics) es común que se coloreen múltiples dibujos de los mismos personajes usando la misma paleta de colores. En particular, en el caso del *manga* (comics japoneses fuertemente inspirados por el trabajo de Osamu Tezuka), la publicación principal se realiza en blanco y negro, mientras que el coloreado se considera un objetivo secundario.

En el caso del manga, la mayoría de las obras más populares se publican semanalmente (en contraste con los comics americanos, que suelen publicarse mensualmente), por lo que los artistas suelen tener un horario de trabajo muy ajustado. Además, por motivos económicos y logísticos, los mangas se publican físicamente en papel periódico reciclado usando solo tinta negra (con unos pocos tonos grises para sombreado). Por estos motivos las editoriales solo publican mangas coloreados en ediciones especiales, para las cuales suelen requerir de artistas externos especializados en colorear.

Por otro lado, aprovechándose de la omniprescencia del internet en la era de información, varias revistas digitales han surgido en los últimos años (como *Webtoon* y *mangaplus*). En estas, los artistas pueden permitirse tiempos de publicación flexibles, así como el uso de herramientas digitales en todo su proceso creativo (boceto, coloreado, sombreado, etc.). Sin embargo, aún con estas herramientas a su disposición, colorear los paneles para las distintas páginas de los comics se mantiene como una tarea laboriosa.

Por todo lo anterior, proponer una solución de software utilizando métodos de inteligencia artificial para apoyar en el proceso de coloreado de múltiples dibujos parece pertinente. Para esto se decidió utilizar redes neuronales convolucionales debido a que experimentalmente han probado ser modelos muy exitosos para el procesamiento de imágenes.

Finalmente, aunque en este trabajo no se espera proveer una solución definitiva al problema consideramos que, al igual que trabajos previos, puede servir como una pequeña contribución en dirección a alcanzar una mejor solución.

1.3. Objetivo general

Crear una red neuronal que sea capaz de colorear dibujos en boceto de manera automática, permitiéndole al usuario proporcionar indicaciones sobre los colores utilizados.

1.4. Objetivos específicos

- Seleccionar o construir un conjunto de datos apropiado para entrenar el modelo.
- Diseñar la arquitectura de un modelo que, con mínima interacción del usuario, colore dibujos en boceto de modo que se avance hacia una mejor solución al problema.
- Implementar y entrenar el modelo usando un conjunto de dibujos coloreados.
- Utilizar el conocimiento adquirido en el entrenamiento para ajustar detalles finos en el modelo.
- Publicar en GitHub el código necesario para el entrenamiento y ejecución del modelo, así como los pesos del modelo entrenado .

Capítulo 2

Aprendizaje automático

En años recientes, el constante crecimiento del poder de cómputo, así como el uso cotidiano de las computadoras en diversos ámbitos de nuestra vida han generado un enorme incremento en la producción y el almacenamiento de datos. Esta gran cantidad de datos disponibles han hecho posible el auge del nuevo enfoque en la computación basado en estadística: el *Machine Learning*, usualmente traducido como *Aprendizaje Automático*.

En el paradigma clásico de computación se considera como un objetivo buscar algoritmos, es decir, series de pasos que una computadora pueda ejecutar y que siempre lleguen a un resultado correcto. Sin embargo, existen problemas para los cuales aún no se conocen algoritmos generales que los resuelvan en tiempo razonable. Por otro lado, en el paradigma del aprendizaje automático se propone utilizar principios de probabilidad y de estadística para crear algoritmos que puedan llegar a cometer errores pero, a cambio, ofrezcan un tiempo de cómputo aceptable (Bramer, 2016).

Estos algoritmos estocásticos son comúnmente llamados *modelos* en el contexto del aprendizaje automático y consisten en una serie de operaciones paramétricas (es decir, operaciones que dependen de algunos valores internos llamados parámetros) que dictan su comportamiento y que se ajustan o "entrenan" usando la información obtenida de los datos, para que su rendimiento sea el deseado. Estos datos usualmente se representan como una tupla de atributos (que pueden ser categóricos o numéricos) y se deben dividir aleatoriamente en dos subconjuntos: uno grande (80 %-90 % de los elementos) llamado *conjunto de entrenamiento*, que servirá para ajustar los parámetros del modelo, y otro pequeño llamado *conjunto de pruebas o de validación* que servirá para medir el rendimiento del mismo (Bramer, 2016).

A continuación se describen brevemente las categorías en las cuales se divide el aprendizaje automático y se dan ejemplos de problemas que encajan en estas .

2.1. Aprendizaje supervisado

Cuando en los datos existe algún atributo importante que se quiere predecir, es decir que el modelo debe regresar una estimación de este atributo tomando como valor de entrada los demás, se dice que se está realizando aprendizaje supervisado, y al conjunto de datos se le llama *etiquetado* (Bramer, 2016). Ejemplos de tareas de aprendizaje supervisado incluyen:

- **Clasificación**. El modelo debe predecir un valor categórico al que pertenece la instancia de entrada. Algunos ejemplos son: clasificación y multi-clasificación de imágenes, detección de spam, diagnósticos médicos automáticos, etc.
- **Regresión**. El modelo debe predecir un valor númerico (usualmente continuo). Algunos ejemplos son: estimación de precios o ventas, estimar la edad de una persona, etc.
- **Eliminación de ruido**. El modelo recibe como entrada una señal con ruido y debe regresar la señal removiendo el ruido añadido.
- **Traducción automática**. El modelo recibe como entrada una cadena de texto en lenguaje natural en un idioma y debe proporcionar su traducción en otro idioma.
- **Segmentación de imágenes**. El modelo recibe como entrada una imagen y debe regresar una matriz de tamaño igual en la que cada valor representa la clase a la que pertenece ese pixel. Un ejemplo importante es la detección de tumores en radiografías.

El problema que compete a este trabajo cae dentro de la categoría de aprendizaje supervisado, ya que se espera que el modelo regrese el dibujo coloreado y se tiene un dibujo ya coloreado contra el cual comparar el resultado del modelo.

2.2. Aprendizaje no-supervisado

Cuando no se desea predecir un atributo en particular, sino que se desea encontrar patrones dentro de los datos, se está realizando aprendizaje no-supervisado (Bramer, 2016). Ejemplos de tareas de aprendizaje no-supervisado incluyen:

- ***Clustering* o Agrupamiento**. El modelo debe definir n grupos a los cuales van a pertenecer los datos de entrada según sus similitudes.
- **Síntesis**. El modelo debe ser capaz de generar nuevas instancias de datos que sean indistinguibles de las del conjunto de datos.

2.3. Aprendizaje por refuerzo

El aprendizaje por refuerzo se utiliza para entrenar *agentes* que actuen de cierta forma deseada en un ambiente. En general, este tipo de aprendizaje no depende de un conjunto de datos, sino que se retro-alimenta al modelo para cada acción que realice según su rendimiento para la tarea deseada. En otras palabras: cuando el modelo comete un error, se le "castiga"; mientras que, cuando exhibe la conducta deseada se le "premia"(Ravichandiran, 2018). Ejemplos de agentes entrenados usando aprendizaje por refuerzo incluyen:

- **Autos autónomos**. El modelo debe aprender a navegar por un ambiente sin chocar.
- **Jugadores autónomos**. El modelo deberá aprender a ganar en el juego para el cual se le entrenó.
- ***Bots* de chat**. El modelo deberá aprender a entablar una conversación y simular ser un humano.

2.4. Problemas del aprendizaje automático

El problema de entrenar un modelo de aprendizaje automático se puede equiparar al de ajustar una curva a una serie de puntos. Intuitivamente, en un *buen ajuste* del modelo, los valores que regresa la función obtenida son muy cercanos a los valores del conjunto de datos; sin embargo, no basta con cumplir esta propiedad para considerar que el modelo entrenado hace un buen trabajo, puesto que también deberemos requerir que sea capaz de predecir datos nuevos que no se encontraban en el conjunto de datos. Entonces, si el modelo se ajusta demasiado bien al conjunto de entrenamiento, podemos caer en el problema del ***sobreajuste***, en el cual el modelo predice muy bien los datos conocidos pero funciona muy mal en los desconocidos. Por otro lado, si utilizamos un modelo demasiado simple caeremos en el extremo opuesto, el cual es conocido como ***subajuste***, que es cuando el modelo es incapaz de predecir incluso el conjunto de entrenamiento. Entrenar un *buen modelo* de aprendizaje automático consiste en encontrar un punto medio entre el sobreajuste y el subajuste (Bramer, 2016).

Capítulo 3

Redes neuronales artificiales

Las redes neuronales artificiales son uno de los modelos más exitosos conocidos para realizar tareas de aprendizaje automático (tanto supervisado como no-supervisado). Las redes neuronales reciben su nombre debido a que en sus primeras iteraciones su propósito era el de simular un cerebro y, por lo tanto, estaban fuertemente inspiradas por los modelos conocidos de las neuronas biológicas y sus conexiones (Rosenblatt, 1958), principalmente en la regla Hebbiana del aprendizaje, la cual establece que durante el proceso del aprendizaje la conexión entre dos neuronas se refuerza si estas se activan al mismo tiempo (Hebb, 1961).

En los últimos años se han propuesto mejoras a los modelos de redes neuronales, muchas de las cuales no están basadas en modelos biológicos, por lo que en la actualidad ya no se interpretan como una simulación de un cerebro sino como un modelo de aprendizaje automático. En general, una red neuronal artificial cualquiera es una función diferenciable construida a partir de operaciones sencillas que a su vez son diferenciables, como transformaciones afines y funciones de activación no-lineales. Se pueden representar estas funciones como un grafo dirigido, en el cual cada nodo representa una operación y cada arista representa el resultado de aplicar la operación almacenada en el nodo y, a su vez, son los valores de entrada de los nodos a los que apuntan. A cada nodo de este digrafo se le llama comúnmente *capa*, mientras que a los valores numéricos propios de cada capa se les llama *pesos* o *parámetros* y a las aristas se les llama *activaciones*. En este contexto, a los parámetros que no se ajustan por el proceso de entrenamiento se les llama *hiper-parámetros* (Goodfellow, Bengio, y Courville, 2016).

Ahora bien, los pesos comúnmente se inicializan con ruido y se modifican poco a poco usando métodos de optimización iterativos para que el comportamiento de la red sea el deseado. A este proceso se le llama *entrenamiento*. Durante el entrenamiento se optimizan los pesos de las capas usando la derivada de una función de *error* que se construye específicamente para medir el rendimiento de la red, de modo que minimizar

esta función cause que el comportamiento de la red sea el deseado. Esta función toma el nombre de *pérdida* (o en inglés *loss*). Es común que la actualización de los pesos no se realice solo considerando el valor de la pérdida para un dato, sino que se considere el promedio de la pérdida en un pequeño subconjunto de datos que llamamos *lote* (*batch*), de modo que, en cada paso se considera un subconjunto diferente de datos. Además, en sistemas modernos, se puede procesar un lote de manera paralela en las tarjetas gráficas (Goodfellow et al., 2016).

En este capítulo se exponen las capas más importantes que usualmente sirven como bloques para construir redes neuronales artificiales, se explican brevemente las funciones de pérdida más comunes y se mencionan los principios más importantes de los métodos de optimización usados para el proceso del aprendizaje.

3.1. Capas importantes

3.1.1. Capas lineales

Las capas lineales son las primeras que fueron propuestas como parte del modelo original y también una de las fundamentales de los modelos actuales. Estas consisten en una transformación afín que recibe datos en forma de vector. En la siguiente ecuación se muestra una ejecución de una capa lineal, en la cual, la matriz W puede no ser cuadrada, de modo que la capa puede cambiar la dimensión del vector de entrada según el diseño del modelo (Minsky y Papert, 1969).

$$x_{i+1} = x_i W^T + b$$

3.1.2. Funciones de activación

Como la mayoría de las capas de una red neuronal son transformaciones afines y una composición de transformaciones afines se puede reducir a una sola transformación afín es necesario agregar capas que sean funciones no-lineales, justo después de cada capa lineal para que el espacio de funciones que consideramos sea más amplio que el espacio de transformaciones afines. Estas funciones van de $\mathbb{R} \to \mathbb{R}$ y usualmente no contienen parámetros entrenables (Minsky y Papert, 1969). Algunas de las más comunes son las siguientes:

- **Sigmoide**. Es la función de activación propuesta originalmente ya que mapea $\mathbb{R} \to (0, 1)$, evitando que la norma de estos crezca indefinidamente.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

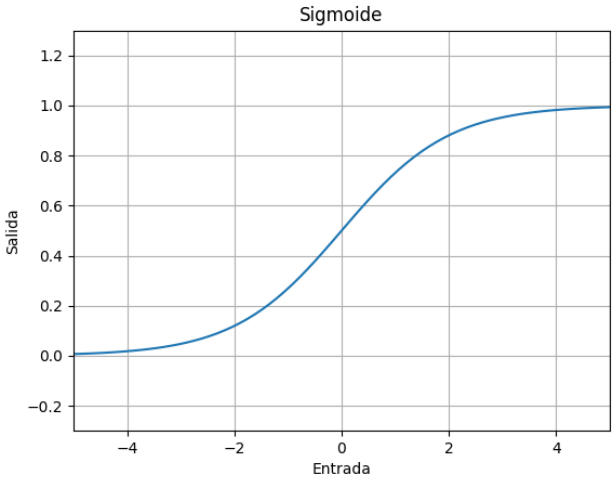


Figura 3.1: Gráfica de la función sigmoide.

- **Tangente hiperbólica**. Similar a la sigmoide, pero con distinto codominio. Mapea $\mathbb{R} \to (-1, 1)$.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

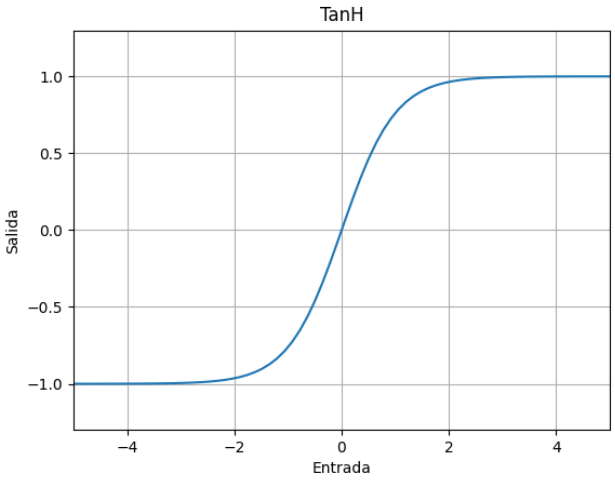


Figura 3.2: Gráfica de la función tangente hiperbólica.

- ***Rectified linear unit* (ReLU)**. La principal ventaja de esta función es que es sencilla y rápida de calcular. Como mostró Krizhevsky, esto permite hacer una mayor cantidad de experimentos en el mismo tiempo y en muchos casos puede acelerar la convergencia del modelo (Krizhevsky, Sutskever, y Hinton, 2012).

ReLU es la función más comúnmente utilizada en la práctica.

$$ReLU(x) = max(0, x)$$

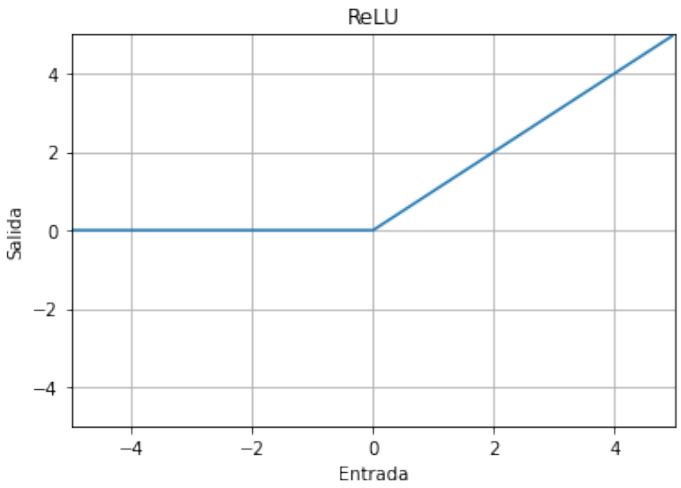


Figura 3.3: Gráfica de la función ReLU.

- **Leaky ReLU**. Similar a la ReLU pero considerando los números negativos. El parámetro α puede ser fijo durante el entrenamiento o ser un parámetro entrenable.

$$LeakyReLU(x) = \begin{cases} x & \text{Si } x > 0 \\ \alpha x & \text{en cualquier otro caso} \end{cases}$$

- ***Continuously Differentiable Exponential Linear Unit* (CELU)**. Similar a la Leaky Relu, pero con una curva suave en los negativos, además cerca del 0 se comporta como la identidad (Barron, 2017).

$$CELU(x) = max(0, x) + min(0, \alpha(e^{x/\alpha} - 1))$$

3.1.3. Normalización de lotes *BatchNorm*

Las redes muy profundas suelen presentar el siguiente problema: Sea A una matriz, x un vector de entrada y $f(x) = \text{ReLU}(Ax)$, entonces, es muy probable que $||f^n(x)|| \to \infty$ o bien $||f^n(x)|| \to 0$, donde. Dicho en palabras, multiplicar varias veces por una matriz suele producir resultados demasiado grandes o demasiado pequeños. A este problema se le llama *explosión* o *desvanecimiento* de los gradientes.

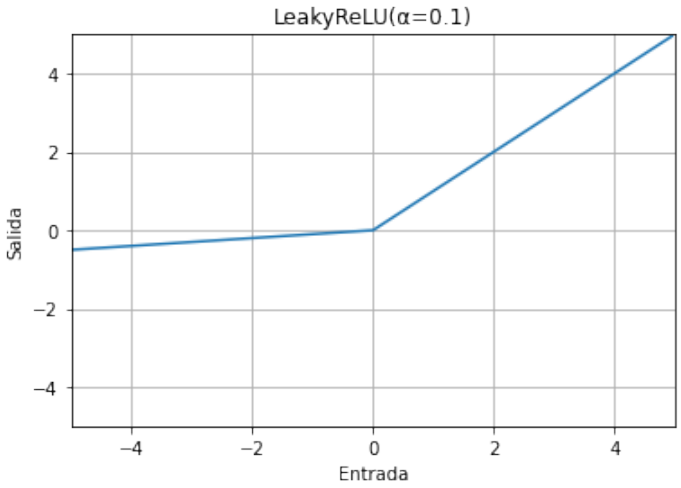


Figura 3.4: Gráfica de la función Leaky ReLU con $\alpha = 0.1$.

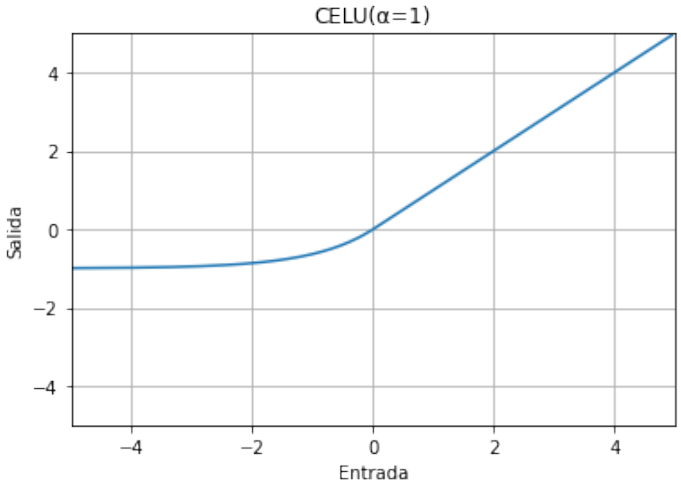


Figura 3.5: Gráfica de la función CELU con $\alpha = 1$.

Una solución bastante inmediata y sencilla de implementar es la repetida normalización de los resultados parciales de las capas, lo cual se conoce como *BatchNorm* puesto que se utilizan la media $\mu(x)$ y la desviación estándar $\sqrt{Var(x)}$ de cada lote para normalizar. Además, se añaden dos vectores con parámetros entrenables γ y β, que se inicializan en 1 y 0, respectivamente, para añadir un poco de flexibilidad a la red, así como un término ϵ el cual es un número pequeño que se añade con el objetivo de prevenir la división entre 0 (Ioffe y Szegedy, 2015).

$$BatchNorm(x) = \frac{x - \mu(x)}{\sqrt{Var(x) + \epsilon}}\gamma + \beta$$

3.1.4. Capas convolucionales

Las capas lineales son muy efectivas para aprender datos con representación vectorial, pero no funcionan muy bien con imágenes debido a que, a diferencia de los vectores, en estas la información contenida en cada pixel no depende de su posición absoluta en la imagen (por ejemplo, una foto de una persona sigue representando a una persona aunque se traslade la figura dentro de la imagen). Para intentar darle solución a este problema Fukushima sugiere usar capas convolucionales en lugar de lineales (Fukushima, 1980). En general, una convolución es una transformación que se aplica sobre una señal en la que se toma un promedio pesado deslizante de un rango de valores y así se genera una nueva señal. Las convoluciones tienen la facultad de obtener información de cada elemento de acuerdo a la información de sus vecinos más cercanos, lo cual es un comportamiento más deseable para las imágenes.

Además, en el contexto del procesamiento de imágenes se usan de manera casi exclusiva las convoluciones con pesos en dos dimensiones, por lo que en este trabajo la palabra *convolución* se usará únicamente para referirse a convoluciones en dos dimensiones. En estas, los pesos usados para el promedio conforman una matriz llamada *filtro* o *kernel* y es usualmente cuadrada, mientras que al número de posiciones que se recorre el filtro para generar cada número se le llama *stride* (es común usar `stride=1` para producir una imagen del mismo tamaño o `stride=2` para dividir a la mitad el ancho y alto de la imagen). Además, para que el resultado quedé del mismo tamaño se suele añadir un margen de 0s a la imagen antes de aplicar la convolución.

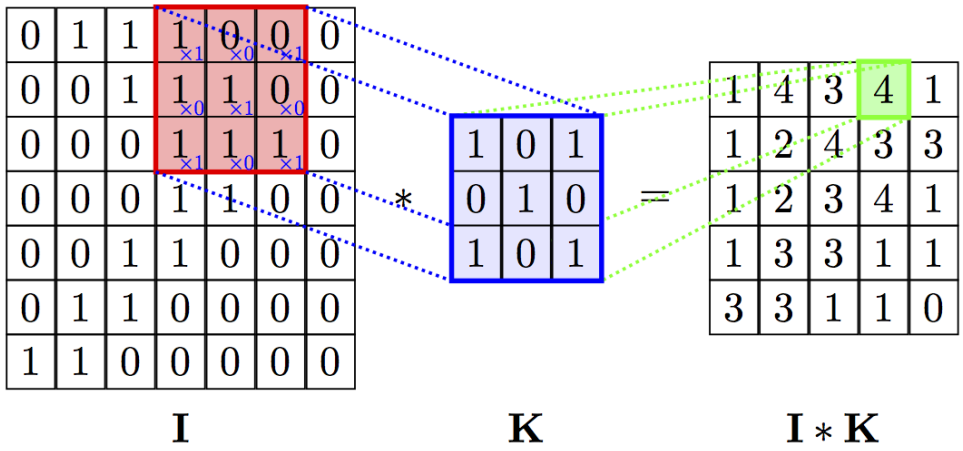


Figura 3.6: Ejemplo de una convolución 2D con `stride=1` , `kernel_size=(3,3)` y sin margen.

Dado que las imágenes son tensores de tamaño (C, H, W), donde C es el número de canales, las capas convolucionales se construyen con $C_{in} \times C_{out}$ filtros, donde C_{in} es el número de canales de la imagen de entrada y C_{out} los de la imagen de salida.

Cada uno de los canales de la imagen de salida será generado por C_{in} filtros que actuarán simultáneamente sobre todos los canales de la imagen de entrada. Las capas convolucionales son también transformaciones afines y los pesos que la red deberá aprender en estas son, efectivamente, los $C_{in} \times C_{out}$ filtros. Para construir una red convolucional se reduce el tamaño de la imagen usando algunas capas con `stride=2`, mientras que se aumenta el número de canales hasta que se obtiene una imagen muy pequeña pero muy profunda. Entonces, esta se transforma en vector promediando sobre ancho y alto para, finalmente, utilizar capas lineales para obtener la salida.

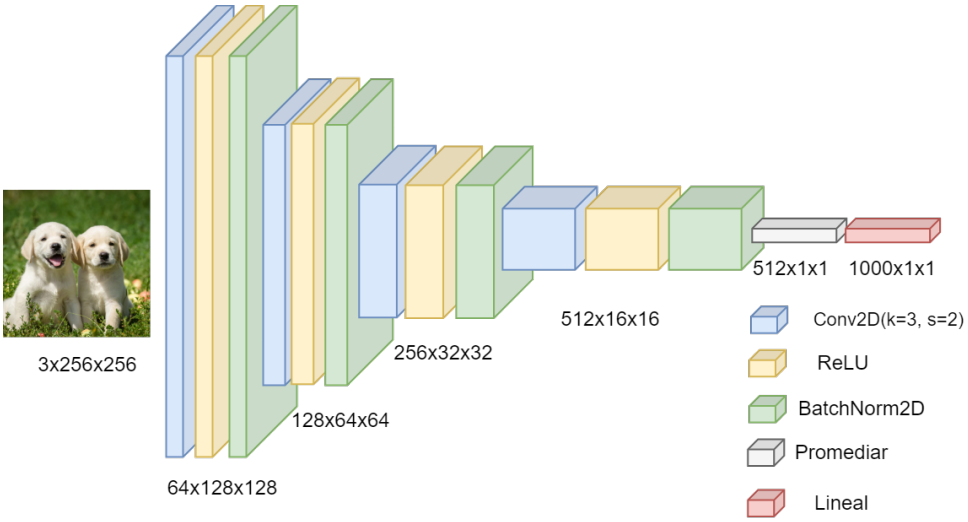


Figura 3.7: Ejemplo de una red convolucional que recibe como entrada una imagen de tamaño $3 \times 256 \times 256$ y la clasifica entre 1000 categorías.

3.1.5. *PixelShuffle*

Las capas convolucionales son muy efectivas para procesar imágenes; sin embargo, estas solo pueden preservar o disminuir el tamaño de las imágenes, por lo que es necesario incluir otro tipo de capas para construir redes generadoras de imágenes. Las capas de PixelShuffle fueron hechas con esta función en mente y consisten en una convolución con `kernel_size=(1,1)` de varios filtros que aumenta el número de canales en la imagen, para después juntar los canales en una nueva imagen de mayor tamaño, alternando entre estos como se muestra en la Figura 3.8 (Shi et al., 2016). Los pesos de esta convolución comúnmente se inicializan de acuerdo al método ICNR de Aitken (Aitken et al., 2017).

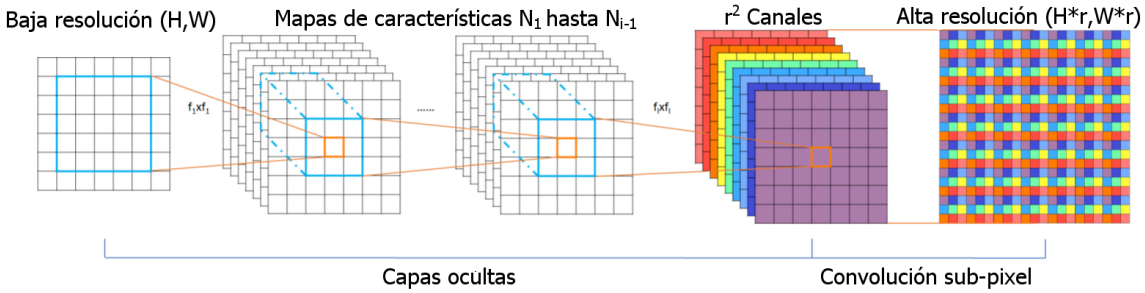


Figura 3.8: Ejemplo de una ejecución de PixelShuffle que aumenta el ancho y alto por un factor de r (Shi et al., 2016).

3.1.6. *MaxPool*

Las capas de MaxPool sirven para reducir el tamaño de las imágenes y funcionan de manera similar a una convolución, pero utilizando un máximo deslizante en lugar de un promedio deslizante. Esto es, se reduce el ancho y alto de la imagen en un factor k, construyendo una nueva imagen que contendrá el valor máximo de todas las secciones cuadradas de pixeles de tamaño $k \times k$ (Zhou y Chellappa, 1988).

3.1.7. *Dropout*

Las capas de Dropout vuelven 0 algunos elementos del tensor con cierta probabilidad p, únicamente durante el entrenamiento. Además, multiplica todos los valores por $\frac{1}{1-p}$ para que el valor de la norma sea el mismo. Esto se hace con el objetivo de crear una dificultad artificial para obligar a que el modelo generalice y que pueda predecir correctamente aún en condiciones desfavorables (Hinton, Srivastava, Krizhevsky, Sutskever, y Salakhutdinov, 2012).

3.1.8. Bloques residuales

Para poder construir redes aún más profundas sin caer en el desvanecimiento de los gradientes, se propusieron los bloques residuales (*ResBlock*, a partir de ahora) los cuales son un bloque de capas con dos caminos (uno con convoluciones y otro con la identidad) que se suman. Dicho de otra manera; sea x la entrada, B el bloque de convoluciones y Act la función de activación. El ResBlock devuelve $Act(x + B(x))$. Se les llama *residuales* porque el principio detrás de ellos es que el bloque de convoluciones ya no debe aprender a predecir la salida óptima, sino el residuo entre la entrada y la salida óptima. Los resultados experimentales muestran que esto mejora el rendimiento de las redes (He, Zhang, Ren, y Sun, 2015a).

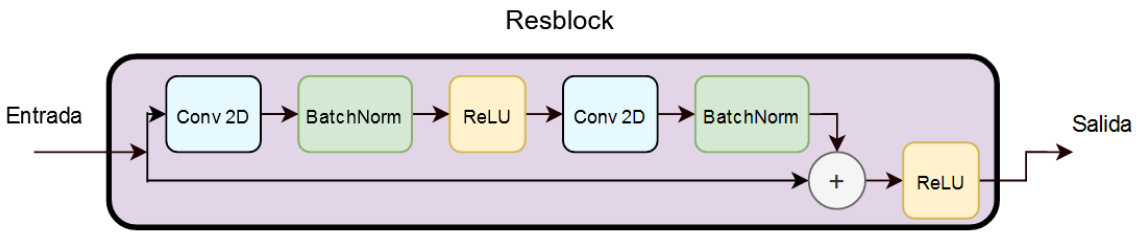


Figura 3.9: Ejemplo de un ResBlock como fue propuesto originalmente.

3.2. Funciones de pérdida

La función de pérdida debe ser capaz de medir el rendimiento del modelo comparando los valores que predice contra las salidas deseadas (tomadas de los datos de entrenamiento). Usualmente son funciones de distancia (Goodfellow et al., 2016). A continuación se enlistan las funciones de pérdida pertinentes a este trabajo, en las cuales $\hat{y}$ se referirá a la salida del modelo, mientras que y se referirá a la salida esperada (tomada del conjunto de datos).

- **Distancia manhatan o L1**.

$$L1(y, \hat{y}) = \text{mean}(|y - \hat{y}|)$$

- **Error cuadrado promedio o MSE**. Penaliza los errores grandes más que los pequeños, es básicamente la distancia euclidiana (L2) omitiendo la raíz cuadrada.

$$MSE(y, \hat{y}) = \text{mean}((y - \hat{y})^2)$$

- **L1 Suave**. Una combinación entre L2 para números pequeños y L1 para números grandes, β es un parámetro, usualmente $\beta = 1$ (Girshick, 2015).

$$SmoothL1(y, \hat{y}) = \text{mean}\left(\begin{cases} \frac{(y-\hat{y})^2}{2\beta} & \text{Si } |y - \hat{y}| < \beta \\ |y - \hat{y}| - \frac{\beta}{2} & \text{en cualquier otro caso} \end{cases}\right)$$

- **Entropía cruzada binaria (*Binary cross entropy*, BCE)**. Se usa para tareas de clasificación binaria, cuando el valor que regrese la red debe estar entre $(0, 1)$. En la Figura 3.10 se observa que el valor de la pérdida aumenta exponencialmente cuando la salida de la red es muy distinto al esperado. Es pertinente resaltar que en el caso de que la salida de la red sea exactamente 0 o exactamente 1, uno de los términos se vuelve indeterminado; para resolver este problema, la mayoría de las implementaciones de esta función truncan el

resultado de $\log(0)$ a -100.

$$BCE(y, \hat{y}) = -\text{mean}(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

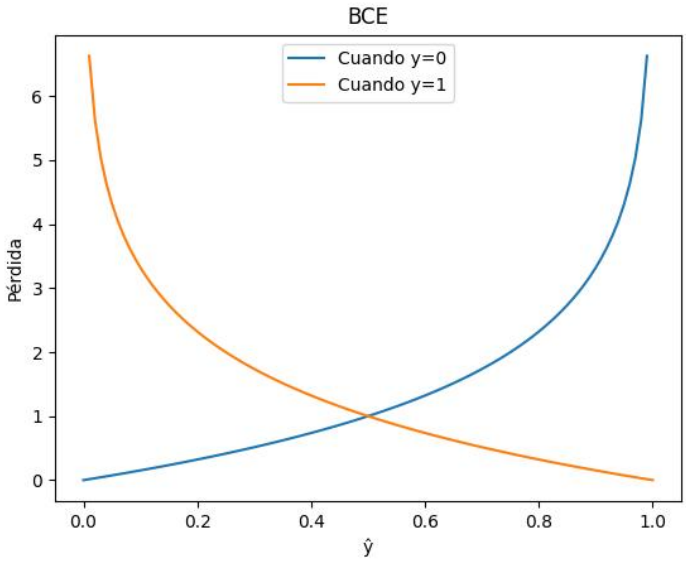


Figura 3.10: Gráfica de la función BCE.

- **Pérdida Perceptual**. La idea es usar una red convolucional previamente entrenada (denotada con ϕ) para codificar las imágenes de entrada y después considerar un promedio ponderado de la distancia (usualmente MSE) entre las activaciones de las capas convolucionales. En la ecuación siguiente $\phi_i(x)$ representa la activación de la capa i de la red al recibir x como entrada y w_i es el peso que se le asigna a esa capa. De acuerdo a (Gatys, Ecker, y Bethge, 2015) esta función de pérdida simula muy bien la distancia de dos imágenes percibida por un humano, a pesar de que la distancia pixel por pixel sea muy grande.

$$PerceptualLoss(y, \hat{y}) = \sum_i w_i \cdot MSE(\phi_i(\hat{y}), \phi_i(y))$$

- **Pérdida de Estilo**. Similar a la pérdida perceptual, pero esta mide la diferencia entre el *estilo* de dos imágenes. Se propone que el estilo se puede representar como la matriz de covarianza de las activaciones, la cual se construye con el producto externo de la forma vectorizada de las activaciones. En la siguiente ecuación, ψ_i es el resultado de reordenar los números de $\phi_i(x)$, un tensor de tamaño (C, H, W) y convertirlo en una matriz de tamaño $(C, H \times W)$. Al resultado del producto matricial $\psi_i \psi_i^T$ se le llama *matriz de Gram* (Gatys et al.,

2015).

$$StyleLoss(y, \hat{y}) = \sum_i \frac{MSE(\psi_i \psi_i^T, \hat{\psi_i} \hat{\psi_i^T})}{C \times H \times W}$$

Cabe resaltar que se puede definir una función de pérdida tomando una combinación lineal de otras funciones de pérdida y de hecho es una práctica muy común.

3.3. Entrenamiento

Como se mencionó anteriormente, el proceso de entrenamiento consiste de un método iterativo en el que se actualizan los pesos de acuerdo a los gradientes de la función de pérdida. El método por el cual se calculan estos gradientes se llama *retro-propagación*, ya que consiste en calcular el gradiente de una operación considerando el gradiente de las siguientes operaciones utilizando la regla de la cadena (Rumelhart, Hinton, Williams, et al., 1988).

3.3.1. Métodos de optimización

Los métodos de optimización usados en redes neuronales están basados en el descenso por la gradiente, atribuido originalmente a (Cauchy et al., 1847). Este método se basa en la observación de que cualquier función multivariada que sea diferenciable disminuye en cada punto en la dirección contraria a su gradiente. Por este motivo, al moverse de manera iterativa en esa dirección se debería converger a un mínimo local.

En la siguiente ecuación se muestra una iteración del método de descenso por gradiente, en la cual w_t representa un tensor de pesos de la red en la iteración t, L la función de pérdida, α un hiper-parámetro llamado *tasa de aprendizaje*, y es la salida esperada de la red y $\hat{y}$ es la salida real de la red.

$$w_{t+1} = w_t - \alpha \nabla L(\hat{y}, y)$$

El valor de la tasa de aprendizaje es de mucha importancia, por ejemplo, con uno muy bajo, el tiempo de entrenamiento se puede incrementar considerablemente y el entrenamiento tiende a estancarse en mínimos locales no favorables. Por otro lado, con uno muy alto el entrenamiento podría divergir.

En la práctica suele ser muy difícil conocer el valor exacto del gradiente de la función de error, puesto que depende de todo el conjunto de datos, por lo que comúnmente se aproxima promediando el error en un pequeño subconjunto de datos (cada lote).

La construcción de nuevos optimizadores para entrenar redes neuronales es objeto de una línea muy amplia de investigación y se considera que una discusión teórica

más profunda acerca de la eficiencia de optimizadores está más allá del alcance de este trabajo. Algunos de los más utilizados son: Adagrad (Duchi, Hazan, y Singer, 2011), Adadelta (Zeiler, 2012), Adam (Kingma y Ba, 2014), Lookahead (M. R. Zhang, Lucas, Hinton, y Ba, 2019), Ranger (Wright, 2019), etcétera. En este trabajo se decidió utilizar Ranger por su velocidad de cálculo sin sacrificio de eficacia.

3.4. Métricas

Una métrica es una función que compara de alguna forma las salidas esperadas del modelo con la salidas actuales y proveen una manera de medir el rendimiento del modelo y comprarlo con otros. Estas usualmente no se utilizan como funciones de pérdida, ya que en su mayoría no son continuas o hacen que la convergencia sea más difícil (Goodfellow et al., 2016). En esta sección se presentan las más relevantes para el presente trabajo.

- **Accuracy**. (Traducida comúnmente como *exactitud*). Dado un problema de clasificación, mide en qué proporción el modelo regresa la etiqueta correcta. El mínimo valor posible es 0 y el máximo es 1.

$$Accuracy(y, \hat{y}) = \begin{cases} 1 & \text{si } y = \hat{y} \\ 0 & \text{en otro caso} \end{cases}$$

- **Distancia de Fréchet usando Inception**. (Abreviada FID, a partir de ahora) Se usa para medir el rendimiento de los generadores de imágenes. FID consiste en emplear la red convolucional InceptionV3 (Szegedy et al., 2015) ya entrenada en *ImageNet* para procesar tanto las imágenes generadas como las verdaderas imágenes y obtener las activaciones producidas por alguna de las últimas capas para, posteriormente, calcular la media y la matriz de covarianza sobre el eje de las instancias. La FID finalmente se calcula con la siguiente fórmula, en la cual Y, μ, Σ representan: el conjunto de las imágenes, la media de las activaciones producidas por InceptionV3 y la matriz de covarianza de estas para las imágenes reales. Así mismo, y $\hat{Y}, \hat{\mu}, \hat{\Sigma}$ representan sus contrapartes para las imágenes generadas. Entre menor sea el valor de la FID, más parecias serán las imágenes generadas a las reales y mejor será el modelo (Heusel, Ramsauer, Unterthiner, Nessler, y Hochreiter, 2017).

$$FID(Y, \hat{Y}) = ||\mu - \hat{\mu}||^2 + \text{tr}(\Sigma + \hat{\Sigma} - 2\sqrt{\Sigma\hat{\Sigma}})$$

3.5. Datos para el entrenamiento

Uno de los principales factores por el cual las redes neuronales son tan exitosas en los últimos años es debido al importante incremento en los datos disponibles para el entrenamiento de los modelos, particularmente en el caso de las imágenes. Uno de los conjuntos de datos más relevantes en el campo de visión computacional es *ImageNet*, el cual consta de mas de 14 millones de imágenes etiquetadas en 1000 categorías (Deng et al., 2009) y de hecho se considera el estándar de-facto para entrenar y medir el rendimiento de modelos de visión computacional, el cual se reporta en forma de la exactitud que tienen los modelos al clasificar las imágenes de este conjunto.

3.5.1. Pre-procesamiento

Dado que las redes neuronales reciben como entrada tensores, es necesario transformar los datos crudos en tensores para poderlos utilizar. En el caso de las imágenes, no es necesario un gran proceso para esto: Podemos pensar una imagen de tamaño $H \times W$ como una matriz de dimensiones $(C \times H \times W)$, en donde C es el número de canales (3 para las imágenes a color y 1 para las imágenes en blanco y negro), en la cual se almacenan los valores de los pixeles que componen la imagen. Finalmente, se normalizan usando la media y la desviación estándar de cada canal, calculados en todo el conjunto de datos.

Por otra parte, existe otro paso del pre-procesamiento de las imágenes que, aunque en teoría es opcional, es comúnmente utilizado y se considera un estándar. Este proceso consiste en realizar transformaciones aleatorias a las imágenes (como rotaciones, cambio de iluminación, reflejo, etc.) con el objetivo de *aumentar* artificialmente el número de datos. De ahí que este proceso recibe el nombre de *data augmentation*, traducido como *aumento de datos* de ahora en adelante. Cabe resaltar que, así como el resto de los componentes de la red, el aumento de datos tiene varios hiper-parámetros que deben de ser ajustados manualmente, ya que su uso descuidado puede resultar perjudicial para el modelo (Krizhevsky et al., 2012).

3.6. Modelos importantes

En esta sección se exponen brevemente algunos modelos relevantes para este trabajo, es común encontrar en la literatura varias versiones de cada modelo. Usualmente se construyen primero pequeños bloques con algunas capas y después se varía el número de bloques. Los modelos más grandes usualmente tienen mejor rendimiento, a cambio de utilizar más recursos computacionales.

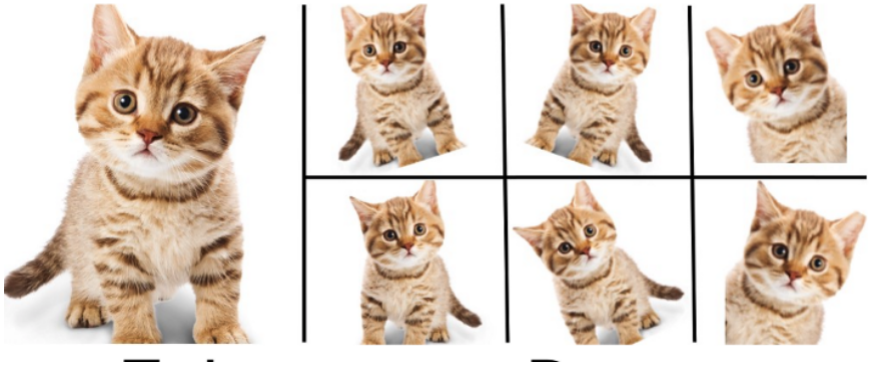


Figura 3.11: Ejemplo de aumento de datos usando rotación, zoom y reflejo.

3.6.1. VGGNet

La red convolucional llamada *VGGNet* (Figura 3.12) fue creada por el Visual Geometry Group de la universidad de Oxford en 2014 para realizar clasificación de imágenes en el conjunto de datos Imagenet (Deng et al., 2009) (Simonyan y Zisserman, 2014). Esta red destacó al momento de su creación, aunque en la actualidad se utiliza por su simpleza en aplicaciones como la pérdida perceptual descrita anteriormente (Johnson, Alahi, y Fei-Fei, 2016), así como para la tarea de extracción de características en general.

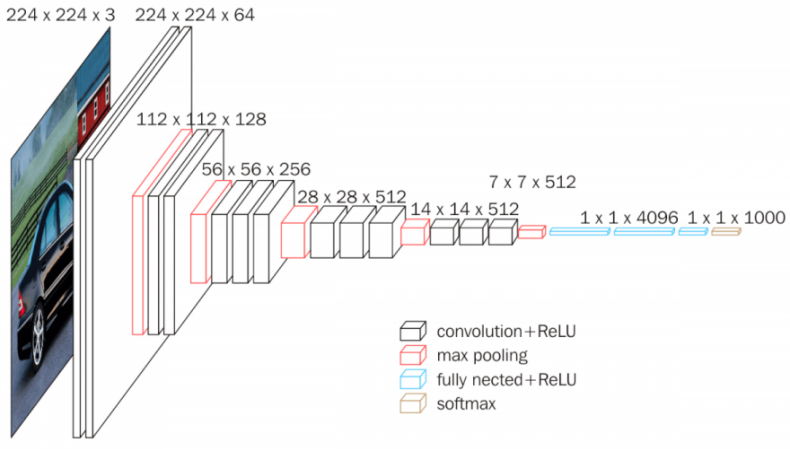


Figura 3.12: Diagrama de la red VGG.

3.6.2. U-Net

La red U-Net fue propuesta por (Ronneberger et al., 2015) para realizar segmentación en imágenes biomédicas. Esta red no contiene capas lineales, por lo tanto, puede aceptar imágenes de cualquier tamaño. La red (Figura 3.13) se caracteriza por

la estructura que asemeja una U, en la cual la parte izquierda tiene convoluciones que reducen el tamaño de la imagen y aumentan el número de canales, mientras que la parte derecha (además de las convoluciones) disminuye el número de canales y aumenta el tamaño de la imagen. La característica más importante de esta red consiste de las conexiones residuales que unen a la parte izquierda con la derecha y que tienen el objetivo de permitir que la red utilice información de más alto nivel (como la imagen de entrada) en las últimas capas. Aunque actualmente se implmenta con algunas mejoras recientes (como reemplazar maxpool por convoluciones de stride 2, reemplazar up-conv por pixelshuffle, etc.), las arquitecturas basadas en UNet son el estándar para realizar tareas de traducción de imagen a imagen

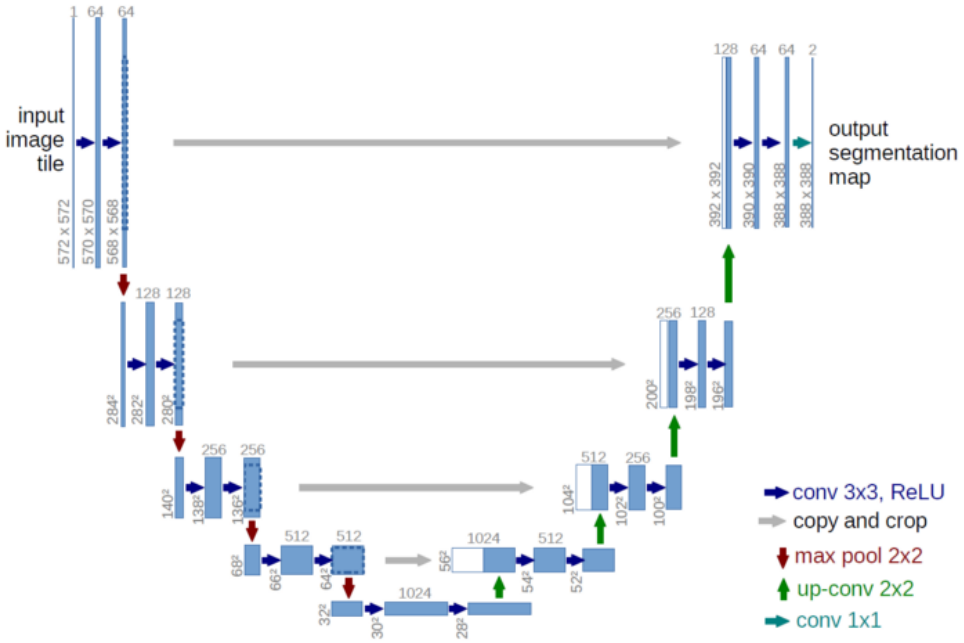


Figura 3.13: Diagrama de la red UNet (Ronneberger et al., 2015).

3.6.3. GAN

Los modelos más exitosos para la generación de imágenes son las redes generadoras adversarias (GANs, a partir de ahora). El principio detrás de las GANs es, que además de la red generadora de imágenes, se entrena a la par una red clasificadora (llamada discriminador) para distinguir entre las imágenes reales y las generadas. Además, considera como parte de la función de pérdida del generador qué tanto fue capaz de "engañar" al discriminador. Si se alterna apropiadamente el entrenamiento entre estas dos redes, se consigue un generador de imágenes que produce resultados que

sean indistinguibles de aquéllos en el conjunto de entrenamiento (Goodfellow et al., 2014).

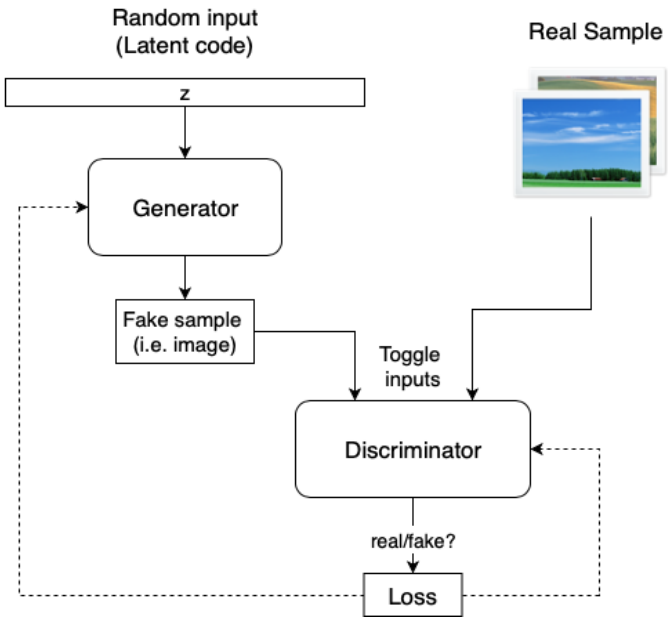


Figura 3.14: Diagrama de una GAN (Horev, 2018).

(Antic, 2018) propuso una leve mejora para entrenar GANs y fue precisamente la que utilizó para entrenar deoldify. Esta técnica recibe el nombre de NoGAN y consiste en partir el entrenamiento en 3 pasos distintos:

1. Entrenar exclusivamente el generador utilizando una función de pérdida convencional hasta convergencia.

2. Utilizar el generador para crear un conjunto de datos que contenga tanto imágenes reales como falsas y entrenar exclusivamente el discriminador con este conjunto de datos hasta convergencia.

3. Entrenar juntos al generador y al discriminador (usando la técnica usual GAN) durante un tiempo reducido.

Según Antic, utilizar NoGAN además de mejorar los resultados, acelera enormemente el entrenamiento.

A pesar de los contundentes resultados que pueden generar los modelos basados en GAN, es bien sabido que, entrenar GANs puede llegar a ser complicado. El entrenamiento de modelos GAN carece de estabilidad y presenta varios problemas (desvanecimiento, colapso, etc.) que no tienen soluciones que funcionen bien para todos los casos (Goodfellow et al., 2014) (Isola et al., 2017) (Ci et al., 2018). Por lo tanto,

existe una gran cantidad de hiper-parámetros que deben ajustarse para el entrenamiento y que en múltiples casos sólo pueden ser determinados experimentalmente o incluso requerir varias ejecuciones por cuestiones aleatorias.

Capítulo 4

Herramientas a utilizar

Como se mencionó en la sección anterior, las redes neuronales se componen de varias capas y durante el entrenamiento los pesos en estas deben de ser optimizados usando métodos basados en descenso por gradiente, por lo que es necesario conocer la derivada de la función de pérdida la cual se calcula usando el método de retro-propagación. Una red profunda conlleva una expresión matemática bastante compleja, por lo que calcular su derivada no es una tarea trivial. En respuesta a este problema se crearon distintos *frameworks* para entrenar redes neuronales, los cuales no solo contienen la implementación de las capas más populares, sino también implementación de la retro-propagación del error, necesaria para realizar el entrenamiento. Estos frameworks son extremadamente eficientes ya que tienen muy bien paralelizadas las operaciones con tensores, además no requieren hardware especializado sino que pueden ser ejecutados en GPUs de computadoras normales, por lo que en este trabajo se decidió utilizar *PyTorch*.

4.1. Biblioteca PyTorch

Pytorch es una biblioteca de python desarrollada por *Meta* (antes *Facebook*) para el desarrollo y entrenamiento de redes neuronales artificiales. Pytorch tiene varios componentes, los cuales están implementados en C y C++ utilizando bibliotecas como CUDA y OpenMP para asegurar el rendimiento necesario para realizar el proceso de entrenamiento (que es computacionalmente intenso). Los componentes de PyTorch se exponen al usuario a través de una API orientada a objetos de alto nivel escrita en python. En esta sección se repasarán brevemente los componentes principales de PyTorch (Paszke et al., 2019) (The PyTorch team, 2019).

4.1.1. Tensores en PyTorch

Como se planteó en el capítulo anterior, las redes neuronales se construyen a partir de las operaciones de matrices de n dimensiones (*tensores* de ahora en adelante), las cuales se deben de realizar de manera paralela para que el entrenamiento pueda ser viable computacionalmente. Estas operaciones se encuentran implementadas en PyTorch en la clase `torch.Tensor`. Los tensores de PyTorch se pueden manipular a través de una API similar a la de la biblioteca *numpy*, de manera que las operaciones se pueden realizar usando una notación intuitiva, como se muestra en el siguiente ejemplo donde A y B son tensores del mismo tamaño.

```
A + B # suma elemento por elemento
A * B # producto elemento por elemento
A * 10 # producto escalar
A @ B # producto matricial
A[0,0,0] # accesar el elemento en la posicion 0,0,0
```

Los tensores de PyTorch pueden guardarse tanto en la memoria RAM como en la GPU y transferirse entre una y otra con una sola línea de código: `A.cpu()` o `A.cuda()`, respectivamente; de modo que las operaciones entre dos tensores que estén guardados en la GPU se realizan de manera paralela ejecutando un kernel de CUDA.

4.1.2. El módulo nn

En este módulo se encuentran implementadas la mayoría de las capas más comunes para construir redes neuronales, las cuales se instancian con hiper-parámetros y luego pueden ser llamadas como funciones. Además, PyTorch permite crear nuevas capas que implementen nuevas operaciones (y combinaciones de capas existentes) creando una clase que herede de `torch.nn.Module` y que implemente el método `.forward(x)`. La clase `nn.Module` detecta (recursivamente) a todos sus miembros que también sean de tipo `nn.Module`, con el fin de detectar todos sus parámetros y de esta manera poder llevar a cabo el cálculo de los gradientes. Al trabajar con PyTorch se recomienda crear clases para los módulos que componen el modelo.

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
```

```
        return F.relu(self.conv2(x))

# o alternativamente usando nn.Sequential
model = nn.Sequential(
    nn.Conv2d(1, 20, 5),
    nn.ReLU(),
    nn.Conv2d(20, 20, 5),
    nn.ReLU())
```

Fragmento de código 4.1: Ejemplo de un modelo de PyTorch con dos capas convolucionales.

4.1.3. El sistema autograd

Como se mencionó anteriormente, una parte esencial en cualquier sistema para entrenar redes neuronales es el de calcular gradientes. En el caso de PyTorch es el sistema llamado *autograd*, el cual consiste en crear un grafo dirigido acíclico que se guarda en la memoria como miembro de la clase tensor y contiene un registro de todas las operaciones que fueron realizadas para *construir* el tensor. El grafo describe varias composiciones de funciones y, por lo tanto, la derivada puede ser calculada usando la regla de la cadena. Tal paradigma permite realizar experimentos con facilidad, incluyendo la creación de nuevos tipos de capas y otras operaciones que pueden ser añadidas de manera dinámica en tiempo de ejecución. Para realizar el cálculo de los gradientes basta con llamar el método `.backward()` en el último resultado de los cálculos de la función que se desea utilizar (la función de pérdida en el caso de las redes neuronales); sin embargo, sólo se puede llamar `.backward()` en un tensor de dimensión 1 para que este pueda ser interpretado como el resultado de la función de pérdida.

4.1.4. Optimizadores

En el módulo `optim` están implementados los métodos de optimización basados en gradiente más comunmente utilizados. Estos se abstraen en clases denominadas *optimizadores*, las cuales se construyen con los parámetros de un modelo. Una vez construidos pueden ser utilizados para realizar iteraciones de optimización utilizando los gradientes guardados en los parámetros de dicho modelo.

```
# Definir el optimizador como Adam, el cual tendra la tarea de
    optimizar los parametros del modelo
optimizer = optim.Adam(model.parameters(), lr=1e-3)
# Hacer que el gradiente de todos los parametros sea igual a a 0
    para limpiarlos
optimizer.zero_grad()
```

```
# Obtener la salida del modelo dada una entrada input
output = model(input)
# Calcular la funcion de perdida usando la salida output y la salida
    esperada target
loss = loss_fn(output, target)
# Calcular el gradiente de todos los parametros involucrados en el
   calculo de loss. Los gradientes quedaran almacenados en cada
   parametro
loss.backward()
# Actualizar los pesos usando el metodo Adam y el gradiente de cada
   parametro.
optimizer.step()
```

Fragmento de código 4.2: Ejemplo de una iteración de optimización.

4.1.5. Biblioteca torchvision

`torchvision` es un módulo de ayuda para visión computacional, en el cual se encuentran implementadas algunas de las arquitecturas más comunes de redes convolucionales, de las cuales se pueden descargar sus pesos ya entrenados en el conjunto de imágenes *ImageNet*. Estos pueden ser importados al código en una sola línea (ejemplo: `m = models.resnet18(pretrained=True)`).

A su vez, en este módulo se incluyen enlaces a los repositorios de algunos conjuntos de imágenes que se usan para el entrenamiento y prueba de modelos: como lo son COCO, CIFAR y MNIST; entre otros.

El módulo también contiene implementaciones de varias transformaciones aleatorias usadas para aumentar los datos, las cuales se definen de manera similar a los módulos descritos anteriormente y pueden ser utilizadas para transformar aleatoriamente imágenes de acuerdo a las especificaciones.

4.2. Biblioteca fast.ai

La biblioteca `fast.ai` del lenguaje python es desarrollada por el grupo de investigación con el mismo nombre, con el objetivo de facilitar el desarrollo y entrenamiento de modelos de redes neuronales. fast.ai está construido encima de PyTorch, sin embargo, su API no se propone como una extensión de PyTorch, sino como un nuevo marco de trabajo diseñado con el objetivo de agilizar el desarrollo, de modo que se requiera la menor cantidad de código posible para crear y entrenar modelos con rendimiento cercano al estado del arte. Esto último lo logran habilitando por defecto varias mejoras que de manera experimental han determinado que funcionan muy bien en la mayoría de los casos (Howard y Gugger, 2020).

4.2.1. Visión con fast.ai

El módulo de visión computacional de fast.ai tiene implementadas utilidades para el procesamiento necesario de tareas complicadas relacionadas con imágenes (como segmentación y detección de objetos), así como las transformaciones más usadas para aumentar los datos, las cuales están disponibles en la función `fv.aug_transforms()`. Además, muchas de estas transformaciones se ejecutan en la GPU, por lo que son más rápidas que sus equivalentes en `torchvision`.

4.2.2. Pre-procesamiento de datos

El módulo de `data` de fast.ai permite abstraer la creación de un `Dataloader`, un objeto iterable en el cual se define el pre-procesamiento de los datos en términos de los procesos y su orden. En el fragmento de código 4.3 se muestra la definición de un `Dataloader` cuya función es cargar parejas de imágenes y sus etiquetas correspondientes para un conjunto de imágenes. Las imágenes están en la carpeta `data` y están separadas en 2 carpetas `train` y `valid`, para marcar los conjuntos de entrenamiento y validación, además están subdividas en carpetas de acuerdo a sus etiquetas (Ej. `data/train/carro/012.jpg`, donde `carro` es la etiqueta para clasificar). Cada imagen se reescala las imágenes a 256x256 pixeles y se añade un aumento aleatorio de datos con una máxima rotación de 30 grados en ambas direcciones y un factor de deformación de 0.1. Finalmente, se asigna un tamaño de lote de 32.

```
import fastai.data.all as fd
import fastai.vision.all as fv
tfms = fv.aug_transforms(max_rotate=30.0,max_warp=0.1)
dblock = fd.DataBlock(blocks    = (fv.ImageBlock, fv.CategoryBlock),
                      get_items = fv.get_image_files,
                      get_y     = fd.parent_label,
                      splitter  = fd.GrandparentSplitter(),
                      item_tfms = fv.Resize((256, 256)),
                      batch_tfms= tfms)
data = dblock.dataloaders("data/", bs=32)

# Ahora se puede utilizar asi
# data[0] es el conjunto de entrenamiento
# data[1] es del validacion
for x,y in data[0]:
    optimizer.zero_grad()
    result = model(x)
    loss = loss_fn(result, y)
    loss.backward()
```

```
20      optimizer.step()
```

Fragmento de código 4.3: Ejemplo de un dataloader definido usando fast.ai.

4.2.3. Learners

Otra ventaja que viene con utilizar fast.ai es el uso de los *learners* (aprendices), los cuales son objetos que abstraen y juntan datos, modelo, función de pérdida, optimizador y métricas; de modo que el entrenamiento se puede realizar llamando al método `.fit()` sobre el learner construido, el cual ejecuta el ciclo de entrenamiento. El comportamiento del ciclo de entrenamiento se puede modificar a través de un sistema de `Callbacks` (funciones de llamada posterior) que pueden ser creadas por el usuario o importadas de la propia biblioteca de fast.ai. Algunas Callbacks son súmamente útiles, por ejemplo: `OneCycleScheduler`, que varía sistemáticamente la tasa de aprendizaje de acuerdo a (Smith y Topin, 2017); `SaveModelCallback`, que guarda el modelo cada vez que mejore; o `CSVLogger`, que escribe métricas del modelo a un archivo CSV cada época.

Capítulo 5

Diseño del experimento

En concordancia con la teoría expuesta anteriormente, se propone construir y entrenar un modelo de redes convolucionales para resolver el problema de la coloración de dibujos. El modelo deberá recibir como entrada una pareja de datos: el dibujo sin color y algún tensor que contenga información de los colores para que se pueda controlar el comportamiento del coloreado a través de este. En la Figura 5.1 se esquematiza de manera muy general la estructura del modelo en el cual, durante el proceso de entrenamiento, tanto los bocetos como las referencias de color serán sintetizados a partir del dibujo coloreado.

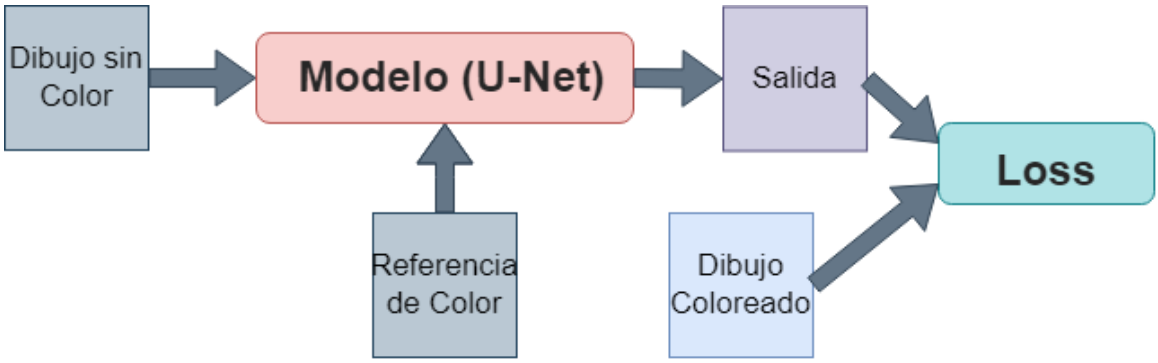


Figura 5.1: Diagrama del modelo propuesto.

5.1. Datos

Se decidió utilizar un conjunto de datos llamado *Danbooru2020*, el cual es una compilación de las imágenes almacenadas en el sitio web *Danbooru* hasta el 2020. *Danbooru* es un repositorio de dibujos estilo anime gratuito mantenido por su comunidad, en el que se encuentran almacenadas mas de tres millones de imágenes, las

cuales están muy bien etiquetadas en diversas y explicativas etiquetas, como se puede apreciar en la Figura 5.2 (Anonymous et al., 2021). Una vez descargadas las imágenes del conjunto de datos (junto con sus metadatos) se procedió a filtrar las imágenes de acuerdo a las siguientes reglas:

- Excluir todas las imágenes que no estuvieran etiquetadas como `safe`, para evitar incluir contenido sensible o explícito.
- Excluir las imágenes si tienen las etiquetas `simple_background` o `white_background`, para evitar imágenes demasiado simples que aporten poca información al modelo.
- Excluir las imágenes si contienen alguna de las siguientes etiquetas: `sketch`, `grayscale` o `photo`. Esto con la finalidad de evitar imágenes sin color.
- Con el fin de obtener imágenes coloridas: Excluir las imágenes si no contienen alguna etiqueta relacionada con el color de cabello (Ej. `blue_hair`, `black_hair`, etc.).
- Con el fin de obtener imágenes coloridas: Excluir las imágenes si no contienen alguna etiqueta relacionada con el color de los ojos (Ej. `blue_eyes`, `red_eyes`, etc.).
- Excluir las imágenes muy anchas o muy altas para no distorsionarlas al momento de reescalarlas. La razón entre su altura y su anchura debe estar en el intervalo $[\frac{3}{4}, \frac{4}{3}]$.

De esta forma, al final del proceso de filtrado, se obtuvieron 250 mil imágenes de dibujos coloridos, apropiados para el entrenamiento de la red. Además, se agregaron alrededor de 50 mil capturas de pantalla (tomadas de los animes disponibles en el catálogo del sitio web *Crunchyroll* (Crunchyroll, 2020), muestreados aleatoriamente), a las cuales se les realizó un proceso de limpieza de datos, en parte manual y en parte ayudado por redes neuronales. El proceso consistió en entrenar modelos con una proporción cada vez mayor de los datos, que a su vez ayudaban a limpiar las partes nuevas del total de datos que se incluían iterativamente, con lo cual se pudo entrenar el modelo sucesivamente. En cada paso, se revisaban manualmente los resultados y se corregían errores.

5.1.1. Sintetización de bocetos

Para entrenar la red se necesita un conjunto de datos que contenga parejas de bocetos con sus respectivas versiones coloreadas; sin embargo, dicho conjunto de datos no estaba disponible durante el desarrollo de este trabajo. En cambio, se decidió

Figura 5.2: Ejemplo de una imagen del conjunto de datos y algunas de sus etiquetas.

generar los bocetos a partir de las imágenes coloreadas (color → boceto), de modo que el objetivo de la red fuera recuperar la imagen original a partir del boceto (boceto → color). Para este proceso se encontraron diversos métodos, algunos de los cuales se exponen brevemente a continuación; seguidos de imágenes de muestra de sus resultados.

- **XDOG** o Diferencia Extendida de Gaussianos: consiste en considerar la diferencia entre dos imágenes producidas a partir de dos desenfoques gaussianos, ligeramente desfasados, para resaltar las secciones de la imagen en las que los colores cambian bruscamente (Winnemöller, Kyprianidis, y Olsen, 2012).
- **SketchKeras**: es un modelo de redes neuronales basado en UNet que transforma dibujos en bocetos (Illyasviel, 2017).
- **SketchSimplification**: de manera similar a SketchKeras es una red convolucional, pero pone énfasis en obtener bocetos más limpios, a diferencia de SketchKeras que intenta conseguir resultados más *realistas* (Simo-Serra, Iizuka, y Ishikawa, 2018).
- **Anime2Sketch**: también es un modelo de redes convolucionales, aunque representa un punto intermedio entre los dos anteriores (Xiang et al., 2022).

Después de observar múltiples resultados de los distintos métodos, se decidió utilizar un promedio pixel por pixel entre los resultados de SketchKeras y los de Ani-

Figura 5.3: Resultado de los distintos tipos de bocetos generados. De izquierda a derecha: Original, SketchKeras, SketchSimplification, XDOG y Anime2Sketch.

me2Sketch para generar los bocetos de las imágenes en el conjunto de datos. Para esto se descargaron las implementaciones en PyTorch y pesos de modelos entrenados de Anime2Sketch (Xiang, Liu, Yang, Zhu, y Shen, 2021) y de SketchKeras (higumax, 2020).

Figura 5.4: Resultado del promedio entre SketchKeras y Anime2Sketch.

5.1.2. Espacios de colores

Durante el desarrollo del modelo se observó una tendencia de los modelos a producir imágenes poco coloridas (revolviéndose alrededor de tonos de sepia). Esto podría deberse a que utilizando el espacio de colores RGB es complicado definir qué significa *colorido*. Por ejemplo, en RGB el color azul es exactamente lo contrario al amarillo (su distancia en este espacio de color es máxima); sin embargo, para un modelo que coloree sería preferible que pintara el azul como amarillo, en lugar de sepia. Dicho de otra forma, es preferible que este modelo cree resultados *coloridos*, aunque contengan colores incorrectos. Para lograr este propópsito, una alternativa de espacio de colores podría ser HSV, el cual consta de:

- *Hue* que representa la tonalidad del color (medido como un ángulo del círculo cromático).
- *Saturation* que es la saturación del color (que está fuertemente correlacionada con lo que interpretamos como “colorido").
- *Value* que es la iluminación del color.

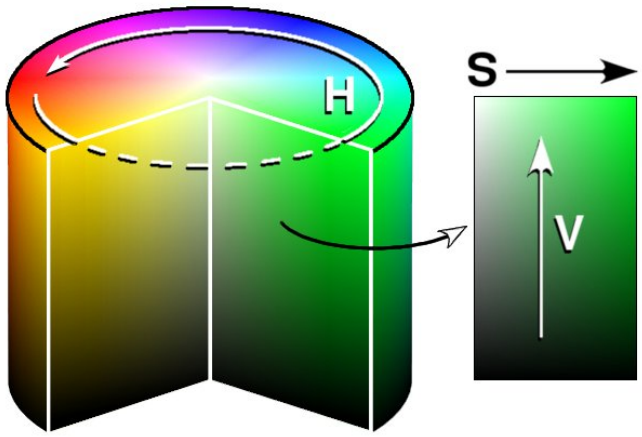


Figura 5.5: Representación del espacio de colores HSV.

En este espacio de colores es más factible expresar matemáticamente lo que *poco colorido* significa, utilizando el componente de saturación; sin embargo, el problema que surge inmediatamente es que el componente hue es un ángulo y, por lo tanto, definir una distancia en este espacio es algo ambiguo. Por ejemplo: un hue (normalizado) de 0 es exactamente lo mismo que un hue de 1, mientras que ambos son lo más alejado posible de un hue de 0.5. Para superar estas limitaciones se proponen como parte de este trabajo dos espacios de colores derivados del HSV: el espacio XYV de 3 componentes, así como el H_xH_ySV de 4 componentes. Ambos espacios se construyen a partir del HSV y se describen a continuación.

El espacio XYV consiste en representar las coordenadas polares descritas por HS (ángulo y radio) en coordenadas cartesianas. Esto es, se calcula el coseno y seno del ángulo H y se multiplican ambos por el radio S para generar las componentes XY, mientras que la componente v se copia directamente.

```
def hsv2xyv(hsv):
    h = hsv[:,0:1]*tau-pi
    s = hsv[:,1:2]
    return torch.cat((s*torch.cos(h), s*torch.sin(h), hsv[:,2:]),dim
   =1)
```

```
5
6 def xyv2hsv(xyv):
7     hue = (torch.atan2(xyv[:,1], xyv[:,0]) + pi)/tau
8     sat = torch.sqrt(xyv[:,0]*xyv[:,0], xyv[:,1]*xyv[:,1])
9     return torch.stack((hue, sat,xyv[:,2]),dim=1).clamp(0,1)
```

Fragmento de código 5.1: Implementación de la conversión de HSV a XYV y su inverso.

Por otra parte, para el espacio H_xH_ySV las componentes H_xH_y también se obtienen calculando coseno y seno, pero no se multiplican por el valor de S; por lo que estas componentes siempre estarán en el círculo unitario. Además, ambas componentes SV se copian tal cual. Se encontró experimentalmente que hacer que el modelo genere las imágenes en H_xH_ySV producía resultados más *coloridos* .

```
1 def hsv2hhsv(hsv):
2     h = hsv[:, 0:1]*tau-pi
3     return torch.cat((torch.cos(h), torch.sin(h), hsv[:,1:]), dim=1)
4
5 def hhsv2hsv(hhsv):
6     hue = (torch.atan2(hhsv[:,1],hhsv[:,0]) + pi)/tau
7     sv = hhsv[:,2:]
8     return torch.cat((hue[:,None],sv),dim=1).clamp(0,1)
```

Fragmento de código 5.2: Implementación de la conversión de HSV a H_xH_ySV y su inverso.

Ambos espacios de colores poseen una ventaja importante sobre el otro y es la razón por la cual en este trabajo se emplean ambos espacios. Por un lado, el espacio H_xH_ySV tiene la ventaja de que la tonalidad del color es distinta de la saturación y esto permite que el modelo tenga la libertad de predecir estos por separado, sin "temor" a equivocarse. Pero, por otro lado, en este espacio es muy sencillo distinguir las imágenes reales de las generadas (una función necesaria para utilizar GANs o pérdida perceptual), ya que las imágenes reales tendrán coordenadas H_xH_y en el círculo unitario, mientras que las generadas no. Es por esta razón que decidimos utilizar ambos espacios, usando el H_xH_ySV para la salida de la red UNet y el XYV para la entrada del discriminador y la red de pérdida perceptual.

5.2. Modelo

Para el modelo se propone usar una arquitectura basada en UNet a la cual se le agregarán otros dos componentes: el *Codificador Z* y el *Decodificador Z*. Los propósitos y detalles de cada componente del modelo se discuten a continuación. Posteriormente, se describe el diseño del discriminador necesario para entrenar utilizando la técnica de NoGAN. Finalmente, cabe destacar que para el desarrollo de los experimentos se construyeron modelos relativamente pequeños, para agilizar el proceso de entre-

namiento. Sin embargo, se plantea la posibilidad de aumentar sistemáticamente la complejidad de estos para mejorar la calidad de los resultados en un futuro trabajo.

5.2.1. UNet

Para construir la UNet todas las convoluciones que se utilizaron se construyeron con `kernel_size=3` y `stride=1`, a menos que se indique lo contrario. Mientras que para los ResBlocks se utilizó la función CELU con $\alpha = 1$ como activación (Barron, 2017) y se agrega un BatchNorm inmediatamente después de esta. En la Figura 5.6 se muestra un diagrama de las capas de un Resblock que convierte de `Ci` canales a `Co` canales y potencialmente disminuye el tamaño de la imagen con `stride=s`.

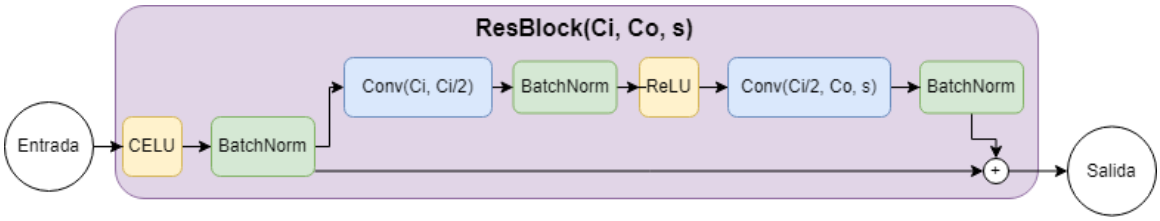


Figura 5.6: Diagrama de los bloques residuales utilizados en el modelo.

La UNet recibirá como entrada una imagen del boceto de un solo canal, mientras que la salida será una imagen de las mismas dimensiones pero de 4 canales (H_xH_ySV). Además, la red internamente está compuesta de 2 bloques codificadores y 3 bloques decodificadores, así como 2 conexiones de salto entre estos que concatenan el resultado de un bloque al resultado de otro. Los bloques se componen de convoluciones, bloques residuales ligeramente modificados y PixelShuffles que se configuraron como se muestra en la Figura 5.7, en la cual entre paréntesis está el número de canales (y tener una suma de estos indica que se concatenaron dos resultados). Los colores de cada capa representan lo siguiente:

- ResBlocks
- PixelShuffle

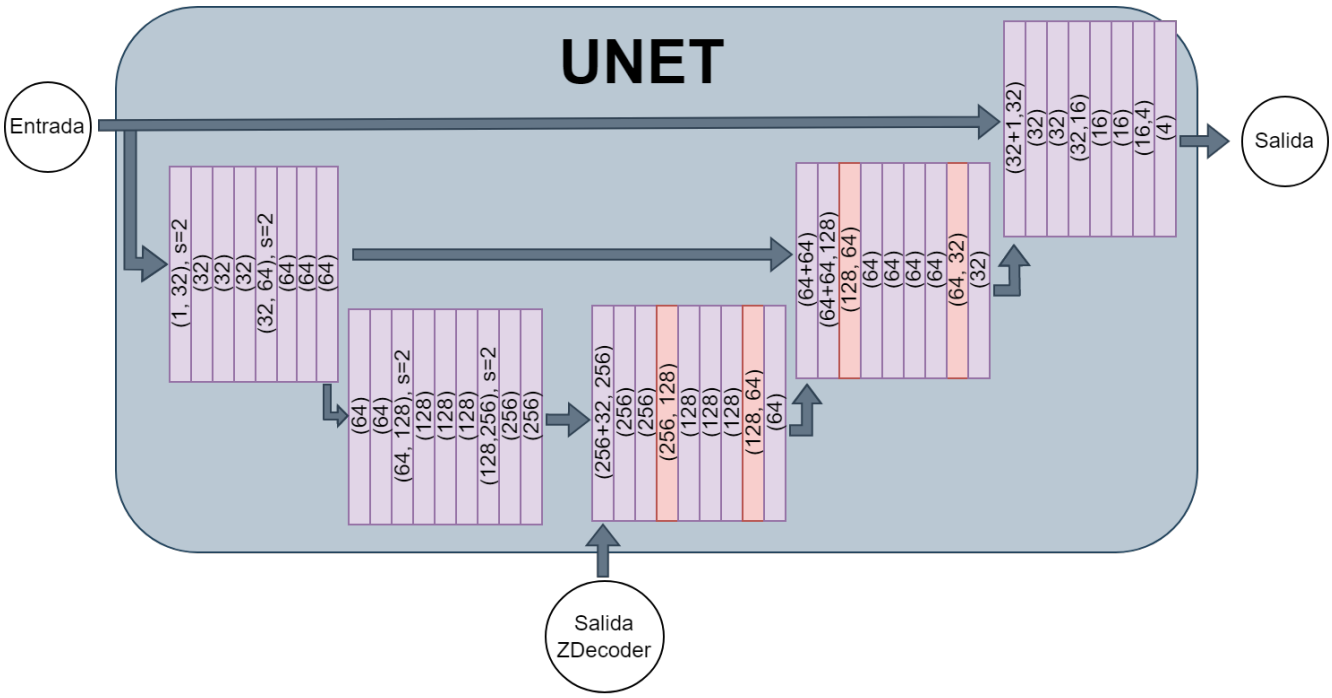


Figura 5.7: Diagrama detallado de la UNet propuesta. Los parámetros en paréntesis son (C_i, C_o) (en el caso que $C_i = C_o$, el segundo número se omite).

5.2.2. *ZEncoder*

Al tensor que contiene la información relevante al color se le llamó Z y consecuentemente a la parte del modelo que está encargada de codificar la referencia de color y producir este vector se le llamó *ZEncoder*. El ZEncoder consiste primero en interpolar la imagen de referencia a un tamaño pequeño (se escogió experimentalmente 96×96) de modo que se pierdan los rasgos finos y se preserven los colores generales de la zona. Luego agregar una especie de dropout (marcado como *HueDropout* en el diagrama) que con cierta probabilidad ($p = 0.25$) vuelve blanco (`(1,1,1)` en RGB) algunos pixeles de la imagen, únicamente durante el entrenamiento. Dado que no se tiene un conjunto de datos óptimo para entrenar el modelo (que contiene: boceto real, referencia real y coloreado real), sino que se deben utilizar referencias y bocetos sintetizados, el objetivo del HueDropout es agregar dificultad al modelo para que sea capaz de colorear aunque las referencias no sean muy buenas. Posteriormente, la capa marcada como PositionalInfo está basada en las utilizadas para los modelos de tipo *transformer* (Vaswani et al., 2017) y agrega información sobre la posición de los pixeles para evitar que esta se pierda en las convoluciones (que están diseñadas para ignorarla). Finalmente, se codifica la imagen resultante usando una red convolucional (Figura 5.8) compuesta de ResBlocks, capas lineales y terminando en TanH para

acotar los valores del vector entre $(-1, 1)$, produciendo un tensor de tamaño lote $\times 64$.

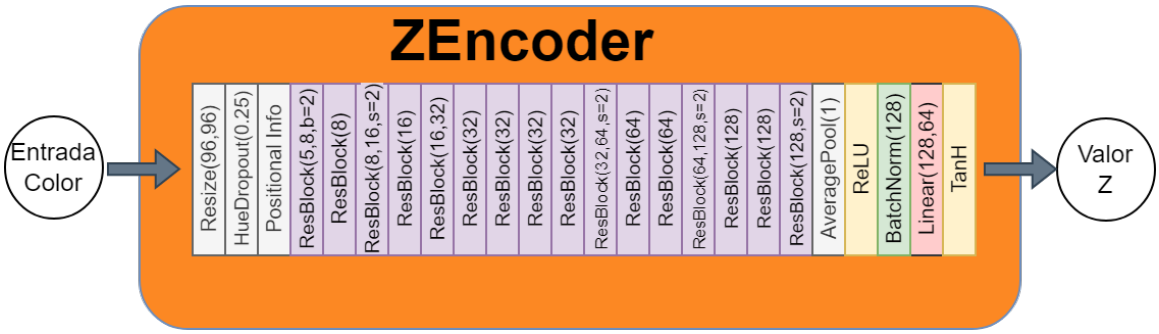


Figura 5.8: Diagrama detallado del ZEncoder.

5.2.3. *ZDecoder*

El *ZDecoder* es una red convolucional (Figura 5.9) que tiene el propósito de decodificar el vector producido por el ZEncoder y convertirla en un tensor de tamaño lote $\times$ 32 $\times$ 8 $\times$ 8. Este tensor se concatena con el segundo bloque codificador de la UNet (interpolando el tamaño de ser necesario) y, además, los primeros 4 canales de este resultado se propagan directamente hacia la función de pérdida (como se describe en la siguiente sección). Se espera que el vector Z producido por un ZEncoder entrenado contenga información relevante del color, para que la UNet sea capaz de colorear usando una referencia de color. Además se considera la posibilidad de que un usuario manualmente modifique este vector para manipular el resultado.

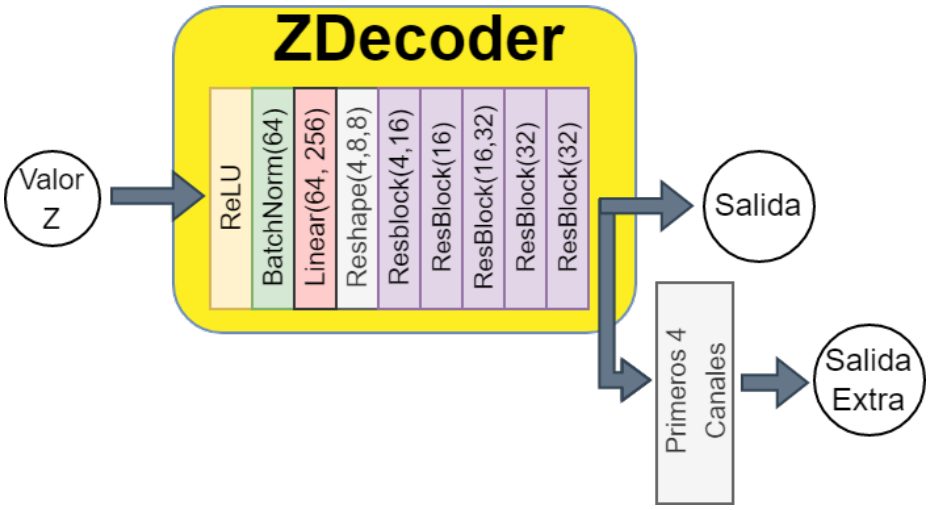


Figura 5.9: Diagrama detallado del ZDecoder.

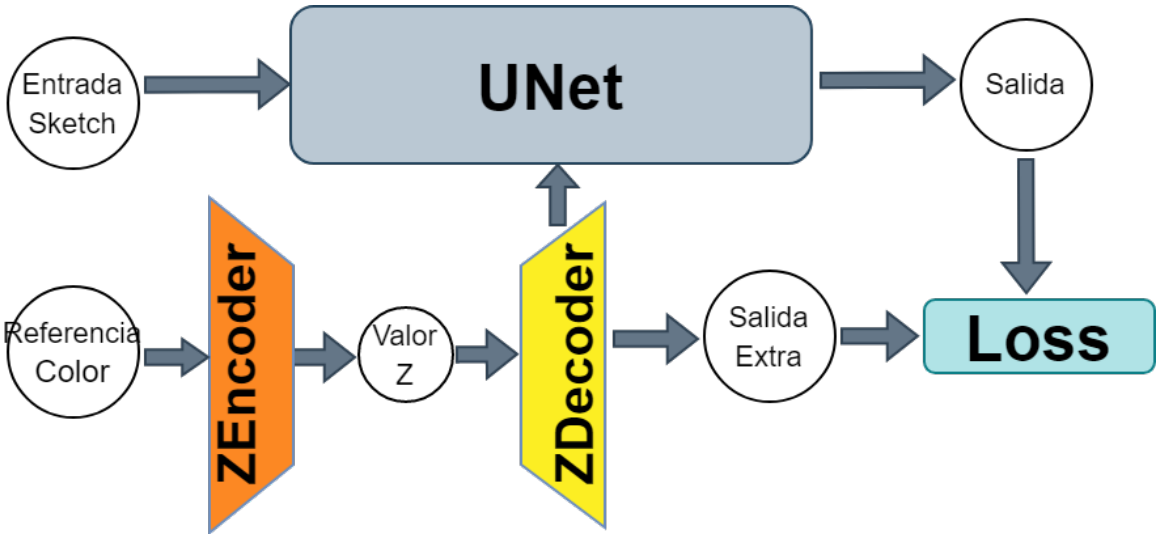


Figura 5.10: Diagrama general del modelo propuesto.

5.3. Función de pérdida

Para la función de pérdida (ecuación 5.1) se consideró inicialmente la suma de dos distancias MSE. En las siguientes ecuaciones:

- x es el sketch de entrada.
- y es la imagen original en color convertida a $\mathrm{H}_x\mathrm{H}_y\mathrm{SV}$.
- $\hat{y}$ es la salida de la red en $\mathrm{H}_x\mathrm{H}_y\mathrm{SV}$.
- $\hat{y_e}$ es la salida extra de la red, también en $\mathrm{H}_x\mathrm{H}_y\mathrm{SV}$.
- y_d es la imagen original en color (también convertida a $\mathrm{H}_x\mathrm{H}_y\mathrm{SV}$), interpolada para ser de las mismas dimensiones que $\hat{y_e}$.

Debido a las conexiones de salto que tiene, la red podría tratar de utilizar únicamente la información que obtiene de la última conexión de salto (ignorando las capas intermedias y por lo tanto la referencia de color). Por este motivo se incluyó una *salida extra* que trata de reconstruir de manera burda la imagen original utilizando sólo la información en las capas intermedias. De esta manera se tratará de forzar a la red a utilizar las capas intermedias.

$$L(x, y) = MSE(\hat{y}, y) + MSE(\hat{y_e}, y_d) \tag{5.1}$$

```
def loss(x,y):
    yp, ye = model(x,y)
    yd = F.interpolate(y, size=ye.shape[2:])
```

```
    return F.mse_loss(yp, y) + F.mse_loss(ye, yd)
```

Fragmento de código 5.3: Implementación de la función de pérdida.

5.3.1. Pérdida perceptual

Después de entrenar la red hasta la convergencia utilizando la función de pérdida anterior, se propone utilizar la pérdida perceptual. Para esto es necesaria una red VGG entrenada; sin embargo, esta red entrenada en el espacio de colores utilizado (XYV) no está disponible para descargarse, por lo que esta debió entrenarse para el trabajo. Se modificó levemente la arquitectura de VGG con el objetivo de acelerar el proceso del entrenamiento y el modelo está descrito por el diagrama 5.11.

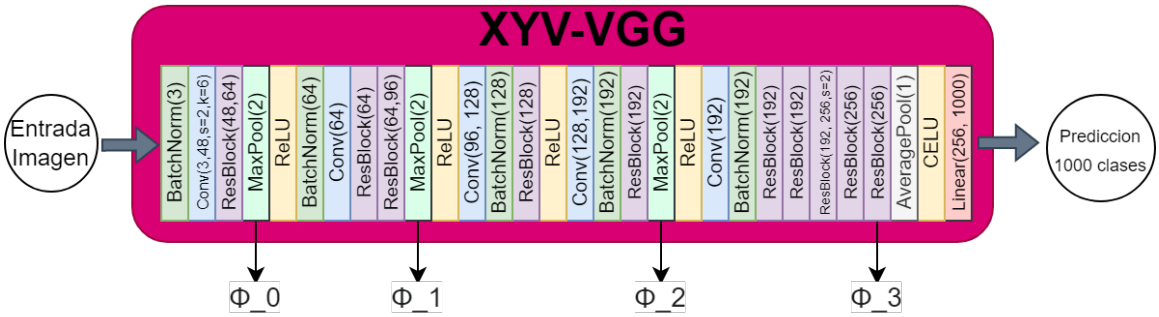


Figura 5.11: Diagrama de la red VGG utilizada para la pérdida perceptual.

Los tensores que se consideraron para la pérdida perceptual están marcados con ϕ_0 hasta ϕ_3 y representan las salidas de las 3 capas MaxPool de la red, así como la salida de la última capa convolucional. Para la pérdida perceptual se considera primero un promedio pesado de las distancias (distancia L1 suave) de estos tensores, así como un promedio pesado de la distancia L1 suave de las matrices de Gram (descritas en la pérdida de estilo). Los pesos de estos promedios están marcados como w_0 hasta w_3 para los tensores y w_4 hasta w_7 para las matrices de Gram. Al final, estos dos se suman, multiplicando la parte del estilo por 200 para que ambos queden en la misma escala. Para los experimentos se utilizaron los pesos $w = \{0.04, 0.12, 0.18, 0.6, 0.04, 0.12, 0.14, 0.05\}$ que se determinaron tomando como base los utilizados en (Gatys et al., 2015) y ajustándolos experimentalmente.

```
def gram(fi):
    bs ,c, h, w = fi.shape
    fi = fi.view(bs, c, h*w)
    return (fi @ fi.transpose(1,2))/(c*h*w)

def p_loss(yp ,y, ws):
    fi0_3 = vgg(hhsv2xyv(y))
```

```
    fip0_3 = vgg(hhsv2xyv(yp))
    content = 0
    for fi,fip,w in zip(fi0_3, fip0_3, ws):
        content += w*F.smooth_l1_loss(fi, fip) )
    style = 0
    for fi,fip,w in zip(fi0_3, fip0_3, ws[4:]):
        style += w*F.smooth_l1_loss(gram(fi), gram(fip))
    return content + 200*style
```

Fragmento de código 5.4: Implementación de la pérdida perceptual.

Una vez implementada la pérdida perceptual se añade otro término a la función de pérdida, así como pesos a los términos en esta de acuerdo a la siguiente ecuación (5.2), en la cual PLoss representa el valor de la pérdida perceptual calculada de acuerdo a lo anterior.

$$L(x, y) = 4MSE(\hat{y}, y) + 2MSE(\hat{y}_e, y_d) + \text{PLoss}(\hat{y}, y) \tag{5.2}$$

```
def loss(x,y):
    yp, ye = model(x,y)
    yd = F.interpolate(y, size=ye.shape[2:])
    return 4*F.mse_loss(yp,y) + 2*F.mse_loss(ye,yd) + p_loss(yp,y)
```

Fragmento de código 5.5: Implementación de la función de pérdida con pérdida perceptual.

5.4. NoGAN

Una vez entrenado el generador, se deberá proceder con el entrenamiento del discriminador para, finalmente, hacerlos competir de acuerdo a lo que indican las ideas de NoGAN. El discriminador (Figura 5.12) se construirá como una red convolucional relativamente pequeña, para que tenga dimensiones similares al generador, y se entrenará para recibir imágenes en XYV y usando como función de pérdida BCE.

Figura 5.12: Diagrama del discriminador propuesto.

Durante el entrenamiento conjunto, la función de pérdida del generador considerará otro término (ecuación 5.3). En esta función de pérdida μ_d representa el promedio de lo que el discriminador predice de la salida $\hat{y}$ para el lote. Dado que el discriminador se entrenará utilizando la función sigmoide, una salida de la red negativa significará que el discriminador cree que es falso; por lo tanto, el generador buscará minimizar el negativo de ese número. Por otra parte, λ es un peso que se le da a esa parte de la pérdida y se determinó experimentalmente en $\lambda = 0.003$.

$$L_G(x, y) = 4MSE(\hat{y}, y) + 2MSE(\hat{y_e}, y_d) + PLoss(\hat{y}, y) - \lambda\mu_d \tag{5.3}$$

```
def loss(x, y, lam=3e-3):
    yp, ye = model(x,y)
    yd = F.interpolate(y[:,:3], size=ye.shape[2:])
    md = discriminator(hhsv2xyv(yp)).mean()
    loss = 4*F.mse_loss(yp, y) + 2*F.mse_loss(ye, yd)
    loss += p_loss(yp, y) - lam*md
    return loss
```

Fragmento de código 5.6: Implementación de la función de pérdida considerando la salida del discriminador.

Capítulo 6

Resultados del experimento

Los experimentos fueron realizados en el servidor de cómputo *DeepZero* ubicado en el Laboratorio de Análisis Avanzado de Datos de la ENES Morelia, el cual cuenta con las siguientes características:

- **Procesador:** AMD Ryzen Threadripper 2990WX con 32 nucleos físicos y 64 lógicos.
- **RAM:** 120 GB de Memoria DDR4
- **GPU:** 2 tarjetas NVIDIA GeForce RTX 2080 Ti con 11GB de VRAM.
- **Sistema operativo:** GNU/Linux Ubuntu 21.04

Además, se emplearon las siguientes versiones del software necesario:

- `CUDA 11.4`
- `Python 3.8.8`
- `Pytorch 1.7.1`
- `torchvision 0.8.2`
- `Fastai 2.3.0`

6.1. Entrenamiento

Se implementaron los modelos descritos anteriormente utilizando para cada capa su implementación de PyTorch. Se inicializaron los pesos con ruido normal de Kaiming (He, Zhang, Ren, y Sun, 2015b); con excepción de las capas PixelShuffle que ya están

implementadas en fast.ai y se inicializan de manera diferente por diseño. Además, se entrenó la red basada en VGG para clasificar imágenes en el conjunto de ImageNet, el cual pasamos por un filtro caricaturizante para ayudar con el rendimiento de esta red al predecir dibujos. Este entrenamiento se ejecutó por 20 épocas tomando un aproximado de 14 horas.

Una vez implementado el modelo UNET, se contabilizaron sus parámetros y el resultado fue de 7.9 millones (en contraste con deoldify, que tiene alrededor de 70 millones). Más aún, durante el entrenamiento de este se utilizó el data augmentation implementado en fast.ai, utilizando como hiper-parámetros: rotación máxima = 30°, deformación máxima = 0.2 e iluminación máxima = 0.1 puesto que se determinó que perturbaciones de mayor tamaño interferirían con la integridad de las imágenes y provocarían que las partes importantes de estas se alteren demasiado. Posteriormente se construyó un Learner de fast.ai conteniendo el modelo y los datos y se configuró el optimizador como ranger, usando la implementación de fast.ai del mismo, agregándole la *callback* `GradientClip`, que trunca el valor de los gradientes cuando son muy grandes, para regularizar el modelo. Finalmente, el modelo se entrenó utilizando la política `one_cycle`, para variar sistemáticamente la tasa de aprendizaje, y se ejecutó por 10 épocas con solo pérdida MSE y una tasa de aprendizaje máxima de 3×10^{-3} para, posteriormente, entrenarlo por otras 3 épocas incluyendo la pérdida perceptual.

El entrenamiento tomó aproximadamente 50 horas usando un tamaño de lote de 10 y un tamaño de imagen de 256×256 pixeles. Los resultados en este punto del entrenamiento se presentan en la Figura 6.1, para los cuales el tiempo de inferencia se midió en 41 milisegundos para una sola imagen y 81 milisegundos para un lote.

Figura 6.1: Ejemplos de los resultados de coloración. De izquierda a derecha: entrada, referencia de color y resultado.

6.2. Entrenamiento NoGAN

Para continuar con el entrenamiento NoGAN, se entrenó el discriminador para diferenciar entre las imágenes generadas y las reales, mientras que se dejaba fijo el generador. El discriminador tiene 3.2 millones de parámetros y se entrenó con 50 % de imágenes reales y 50 % de imágenes falsas, utilizando los mismos parámetros de data augmentation, tamaño de lote y tamaño de imagen. De igual forma, se escogió ranger como el optimizador y se ajustó utilizando el método `fit_one_cycle` de los learners de fast.ai, con máxima tasa de aprendizaje de 3×10^{-3}. Después de 10 épocas de entrenamiento, tomando aproximadamente 30 horas, el modelo registró un valor de la función de pérdida del orden de 10^{-7} y una precisión de 0.8 (ambos valores calculados sobre el conjunto de validación).

Con ambos modelos entrenados se procedió a ponerlos a competir por el método GAN, alternando el entrenamiento después de cada n_b lotes o hasta que el valor de la función de pérdida fuera suficientemente pequeña (menor a 10^{-4}). Sin embargo, después de varios experimentos para variar los hiper-parámetros del entrenamiento (ej. n_b, λ, la tasa de aprendizaje, etc.) se encontró que el generador siempre divergía y sus resultados empeoraban consistentemente con respecto a su versión pre-GAN. Estos resultados pueden observarse en la Figura 6.2.

Figura 6.2: Ejemplos de imágenes producidas por la red después del entrenamiento con GANs.

6.3. Puntuación FID

En virtud de los resultados obtenidos con GANs, se seleccionó el modelo pre-GAN como modelo de coloreado final y se usó la implementación de (Seitzer, 2020) para calcular la puntuación FID. El valor FID del generador se obtuvo utilizando los valores por defecto de esta implementación. Estos son: un tamaño del lote de 50 y obteniendo las activaciones del tercer bloque (cuando son de tamaño 2048) de modo que se compararon las imágenes generadas con las reales, proceso que demoró aproximadamente 10 minutos. La distancia se estimó en **30.03** para el modelo pre-GAN; lo cual es ligeramente mejor que la distancia de **57.06** reportada en (Ci et al., 2018). Sin embargo, se debe remarcar que esto no significa que el modelo de este trabajo es superior al del artículo, ya que esta puntuación no se obtuvo en igualdad de condiciones. Esto debido a que el conjunto de imágenes utilizadas en dicho artículo no se encuentra disponible. Además, la puntuación FID no es una métrica definitiva sobre la calidad del modelo.

Capítulo 7

Conclusiones

Como se hizo notar en el capítulo anterior, el entrenamiento se considera en su mayoría como exitoso y, como consecuencia, el modelo fue capaz de generar dibujos coloreados con una calidad razonable, aunque aún queda espacio para mejorar. En particular, el modelo parece ser capaz de detectar los ojos, las caras y el cabello apropiadamente y marcar bien la separación entre ellos, aplicando los colores correspondientes. Sin embargo, el problema no se considerará como resuelto, debido a que los resultados presentan varios problemas al colorear bocetos (en contraste con colorear blanco y negro), los cuales se exponen en esta sección.

7.1. Problemas y limitaciones en el modelo

Como se puede observar en la Figura 6.1, los resultados tienen una calidad aceptable; sin embargo, suelen diferenciarse por la carencia de detalles finos como el sombreado o la iluminación. Esto se debe al diseño del modelo, en el cual se utilizaron imágenes bastante pequeñas tanto como entrada (256×256 pixeles) como referencia de color (96×96 pixeles), con el objetivo de que el modelo fuera capaz de colorear con una idea burda que asociara colores con posiciones; pero esta a su vez, no permite preservar los detalles finos. Aún así, este es un problema que se debe reconocer para la búsqueda de una mejor solución.

Otra limitante es que el modelo fue entrenado para colorear el boceto utilizando como referencia de color a la propia imagen real. En tal sentido, para que el modelo pueda colorear nuevos dibujos, el usuario debe proveer grandes parches de color de manera burda alrededor de las áreas deseadas (Figura 7.1), de manera similar a las pistas de color utilizadas por (Ci et al., 2018).Tal situación implica que aún es necesaria una gran interacción del usuario para realizar una buena coloración. Por otro lado, tratar de utilizar otro dibujo distinto como referencia, provoca una coloración

incorrecta en el sentido de que los colores quedan fijos en el área en la que se encontraban en el dibujo de referencia original, sin respetar el contexto del boceto que se desea colorear. Un ejemplo de este último fenómeno se observa en la Figura 7.2, en la cual, a pesar de que el modelo respeta las figuras y los bordes (e incluso identifica correctamente la cara) coloca los colores en la misma región de la imagen en la que se encontraban originalmente.

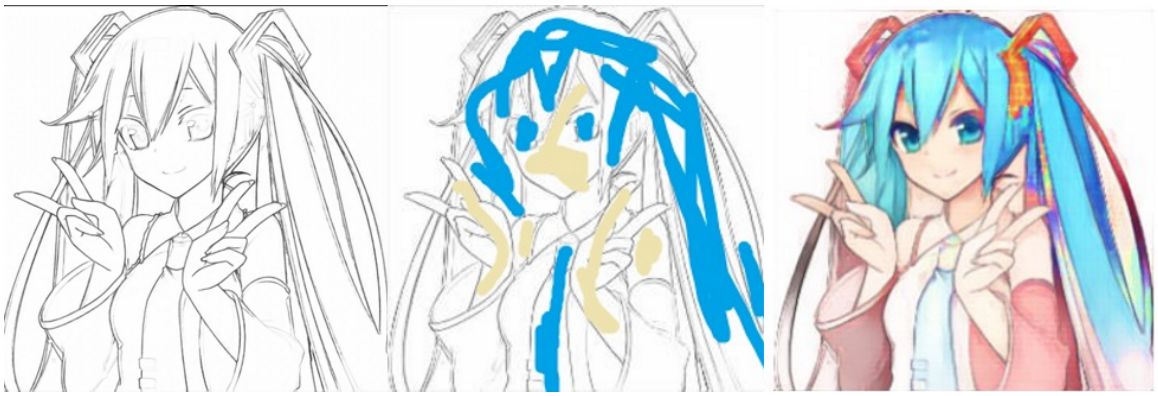


Figura 7.1: Ejemplo de resultado de coloración. De izquierda a derecha: entrada, referencia de color y resultado.

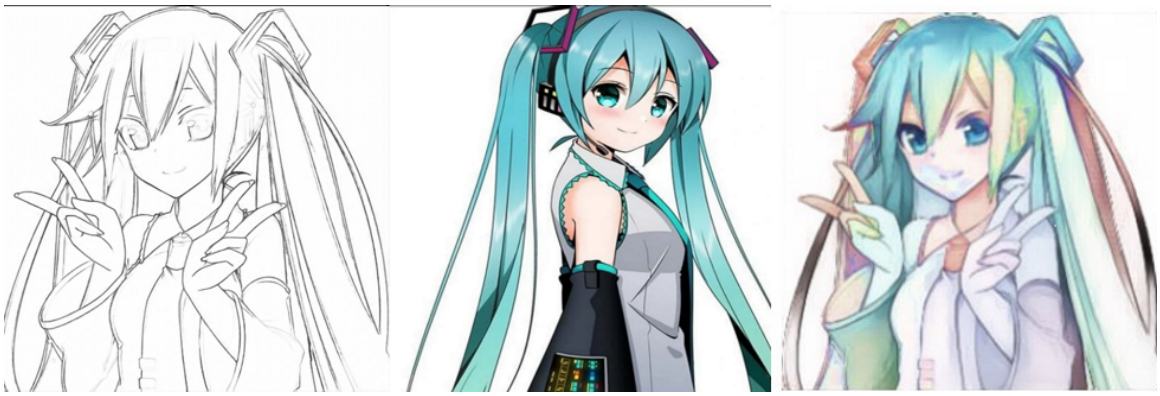


Figura 7.2: Ejemplo de resultado de coloración. De izquierda a derecha: entrada, referencia de color, resultado.

Un inconveniente más que se identificó proviene del conjunto de datos, dado que la mayoría de los personajes de anime tienen la piel de colores claros, el modelo está sesgado y tiende a pintar la piel de esos colores, incluso ignorando cuando las pistas de color le indican colores más oscuros (Figura 7.3).

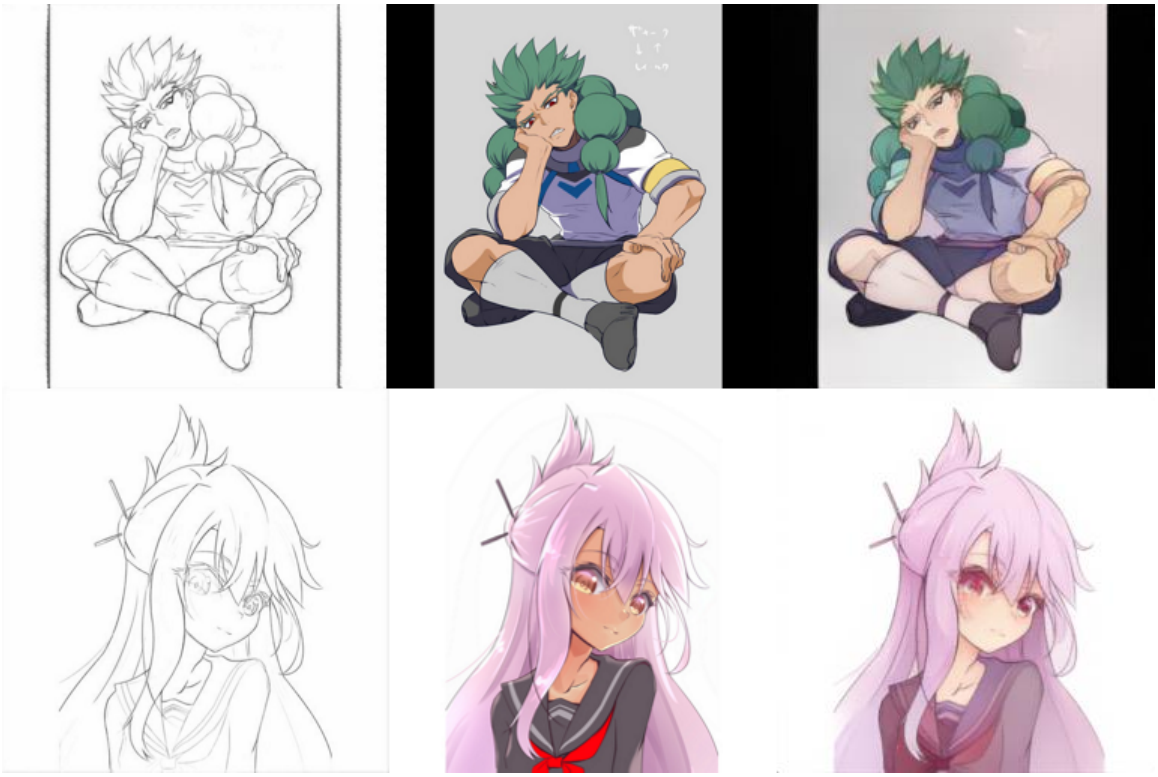


Figura 7.3: Ejemplo de sesgo en el tono de piel. De izquierda a derecha: entrada, referencia de color y resultado.

7.2. Trabajo a futuro

En este trabajo se muestra que el problema de colorear dibujos puede ser más complicado que el de colorear fotografías y que no es para nada trivial; sin embargo, se considera que los modelos propuestos podrían ser considerados como base para futuras investigaciones que proporcionen una mejor solución a este problema.

En primer lugar, los espacios de colores alternativos propuestos (XYV y H_xH_ySV) probaron ser útiles, pero su planteamiento aún queda abierto a modificaciones que puedan agregar información útil o eliminar información redundante. En ese sentido, un trabajo que compare el rendimiento de estos espacios contra el estándar RGB en distintos modelos de redes convolucionales podría ser pertinente.

De igual forma, se considera la posibilidad de agregar sistemáticamente más capas y más parámetros a ambos generador y discriminador, hasta llegar al punto de equilibrio entre rendimiento y complejidad. Además se sugiere la inclusión de nuevas capas, funciones de activación y optimizadores que puedan ser generadas en el futuro, lo cual podría elevar el rendimiento del modelo sin aumentar mucho su complejidad.

Por otro lado, una parte importante del diseño del modelo es precisamente que, aunque este se construya usualmente a partir de una imagen, es el valor Z al final

el responsable de codificar el color del dibujo, por lo que en teoría los valores en este tensor de tamaño 64 podrían ser modificados sistemáticamente para alcanzar un mayor control sobre la coloración que realiza el modelo. Sin embargo, una exploración del espacio latente de este valor Z y su efecto en el resultado de la coloración se consideró como fuera del alcance de este trabajo.

Finalmente, se reconoce que la disponibilidad de conjuntos de datos apropiados para entrenar modelos de coloración es una limitante, por lo que se considera la posibilidad, para un futuro trabajo, de construir un conjunto de datos más numeroso, limpio y diverso en contenido. Un conjunto como este, deberá contener bocetos reales e incluso varias coloraciones para un mismo boceto. Sin embargo, la construcción de un conjunto de datos así, involucraría un gran esfuerzo por parte de una cantidad considerable de personas.

Referencias

Aitken, A., Ledig, C., Theis, L., Caballero, J., Wang, Z., y Shi, W. (2017). Checkerboard artifact free sub-pixel convolution: A note on sub-pixel convolution, resize convolution and convolution resize. *arXiv preprint arXiv:1707.02937*.

Anonymous, community, D., y Branwen, G. (2021, January). *Danbooru2020: A large-scale crowdsourced and tagged anime illustration dataset* [dataset]. `https://www.gwern.net/Danbooru2020`. Descargado de `https://www.gwern.net/Danbooru2020` (Recuperado: 2021-03-20)

Antic, J. (2018). *Deoldify.* `https://deoldify.ai`. (Recuperado el 25 de Junio del 2020)

Baas, M. (2019, July). *Danbooru2018 pretrained resnet models for pytorch* [pretrained model]. `https://rf5.github.io`. Descargado de `https://rf5.github.io/2019/07/08/danbuuro-pretrained.html` (Recuperado 10 de Noviembre de 2020)

Barron, J. T. (2017). Continuously differentiable exponential linear units. *CoRR*, *abs/1704.07483*. Descargado de `http://arxiv.org/abs/1704.07483`

Bramer, M. (2016). *Principles of data mining 3rd ed.* Springer-Verlag London.

Cauchy, A., y cols. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, *25*(1847), 536–538.

Ci, Y., Ma, X., Wang, Z., Li, H., y Luo, Z. (2018). User-guided deep anime line art colorization with conditional adversarial networks. En *Proceedings of the 26th acm international conference on multimedia* (pp. 1536–1544).

Crunchyroll. (2020). *Crunchyroll.* `https://www.crunchyroll.com/es`. (Recuperado 10 de Noviembre de 2020)

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., y Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. En *Cvpr09.*

Duchi, J., Hazan, E., y Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, *12*(7).

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, *36*(4), 193–202.

Gatys, L. A., Ecker, A. S., y Bethge, M. (2015). A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*.

Girshick, R. (2015, December). Fast r-cnn. En *Proceedings of the ieee international conference on computer vision (iccv).*

Goodfellow, I., Bengio, Y., y Courville, A. (2016). *Deep learning.* MIT Press. (`http://www.deeplearningbook.org`)

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial nets. En *Advances in neural information processing systems* (pp. 2672–2680).

He, K., Zhang, X., Ren, S., y Sun, J. (2015a). Deep residual learning for image recognition. *CoRR*, *abs/1512.03385*. Descargado de `http://arxiv.org/abs/1512.03385`

He, K., Zhang, X., Ren, S., y Sun, J. (2015b). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, *abs/1502.01852*. Descargado de `http://arxiv.org/abs/1502.01852`

Hebb, D. O. (1961). *The organization of behavior*. na.

Hensman, P., y Aizawa, K. (2017). Cgan-based manga colorization using a single training image. *CoRR*, *abs/1706.06918*. Descargado de `http://arxiv.org/abs/1706.06918`

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., y Hochreiter, S. (2017). Gans trained by a two time-scale update rule converge to a local nash equilibrium. En *Advances in neural information processing systems* (pp. 6626–6637).

higumax. (2020). *My implementation of sketchkeras in pytorch.* `https://github.com/higumax/sketchKeras-pytorch`. GitHub.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., y Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, *abs/1207.0580*. Descargado de `http://arxiv.org/abs/1207.0580`

Horev, R. (2018, Dec). *Style-based gans – generating and tuning realistic artificial faces.* `https://www.lyrn.ai/2018/12/26/a-style-based-generator-architecture-for-generative-adversarial-networks/`. (Recuperado el 23 de Marzo del 2019)

Howard, J., y Gugger, S. (2020). fastai: A layered API for deep learning. *CoRR*, *abs/2002.04688*. Descargado de `https://arxiv.org/abs/2002.04688`

Illyasviel. (2017). *Sketch keras.* `https://github.com/lllyasviel/sketchKeras`. (Recuperado el 25 de Junio del 2020)

Ioffe, S., y Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, *abs/1502.03167*. Descargado de `http://arxiv.org/abs/1502.03167`

Isola, P., Zhu, J.-Y., Zhou, T., y Efros, A. A. (2017, July). Image-to-image translation with conditional adversarial networks. En *Proceedings of the ieee conference on computer vision and pattern recognition (cvpr).*

Jin, Y., Zhang, J., Li, M., Tian, Y., Zhu, H., y Fang, Z. (2017). Towards the automatic anime characters creation with generative adversarial networks. *arXiv preprint arXiv:1708.05509*.

Johnson, J., Alahi, A., y Fei-Fei, L. (2016). Perceptual losses for real-time style transfer and super-resolution. En *European conference on computer vision* (pp. 694–711).

Kingma, D. P., y Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A., Sutskever, I., y Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. En *Advances in neural information processing systems* (pp. 1097–1105).

Minsky, M., y Papert, S. A. (1969). *Perceptrons: An introduction to computational geometry*. MIT press.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. En H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, y R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. Descargado de `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`

Ravichandiran, S. (2018). *Hands-on reinforcement learning with python*. Packt.

Ronneberger, O., Fischer, P., y Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. En *International conference on medical image computing and computer-assisted intervention* (pp. 234–241).

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., y cols. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, *5*(3), 1.

Saito, M., y Matsui, Y. (2015). Illustration2vec: a semantic vector representation of illustrations. En *Siggraph asia 2015 technical briefs* (pp. 1–4).

Seitzer, M. (2020, August). *pytorch-fid: FID Score for PyTorch.* `https://github.com/mseitzer/pytorch-fid`. (Version 0.2.1)

Shi, W., Caballero, J., Huszár, F., Totz, J., Aitken, A. P., Bishop, R., ... Wang, Z. (2016). Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. En *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1874–1883).

Simonyan, K., y Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Simo-Serra, E., Iizuka, S., y Ishikawa, H. (2018). Mastering sketching: Adversarial augmentation for structured prediction. *ACM Transactions on Graphics (TOG)*, *37*(1), 1–13.

Smith, L. N., y Topin, N. (2017). Super-convergence: Very fast training of residual

networks using large learning rates. *CoRR*, *abs/1708.07120*. Descargado de http://arxiv.org/abs/1708.07120

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... Rabinovich, A. (2015). Going deeper with convolutions. En *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1–9).

The PyTorch team. (2019). *Pytorch documentation.* https://pytorch.org/docs/stable/index.html.

Vallet, A., y Sakamoto, H. (2015). A multi-label convolutional neural network for automatic image annotation. *Journal of Information Processing*, *23*(6), 767-775. doi: 10.2197/ipsjjip.23.767

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. *CoRR*, *abs/1706.03762*. Descargado de http://arxiv.org/abs/1706.03762

Winnemöller, H., Kyprianidis, J. E., y Olsen, S. C. (2012). Xdog: an extended difference-of-gaussians compendium including advanced image stylization. *Computers & Graphics*, *36*(6), 740–753.

Wright, L. (2019). *Ranger - a synergistic optimizer.* https://github.com/lessw2020/Ranger-Deep-Learning-Optimizer. GitHub.

Xiang, X., Liu, D., Yang, X., Zhu, Y., y Shen, X. (2021). *Anime2sketch: A sketch extractor for anime arts with deep networks.* https://github.com/Mukosame/Anime2Sketch. GitHub.

Xiang, X., Liu, D., Yang, X., Zhu, Y., Shen, X., y Allebach, J. P. (2022). Adversarial open domain adaptation for sketch-to-photo synthesis. En *Proceedings of the ieee/cvf winter conference on applications of computer vision.*

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, *abs/1212.5701*. Descargado de http://arxiv.org/abs/1212.5701

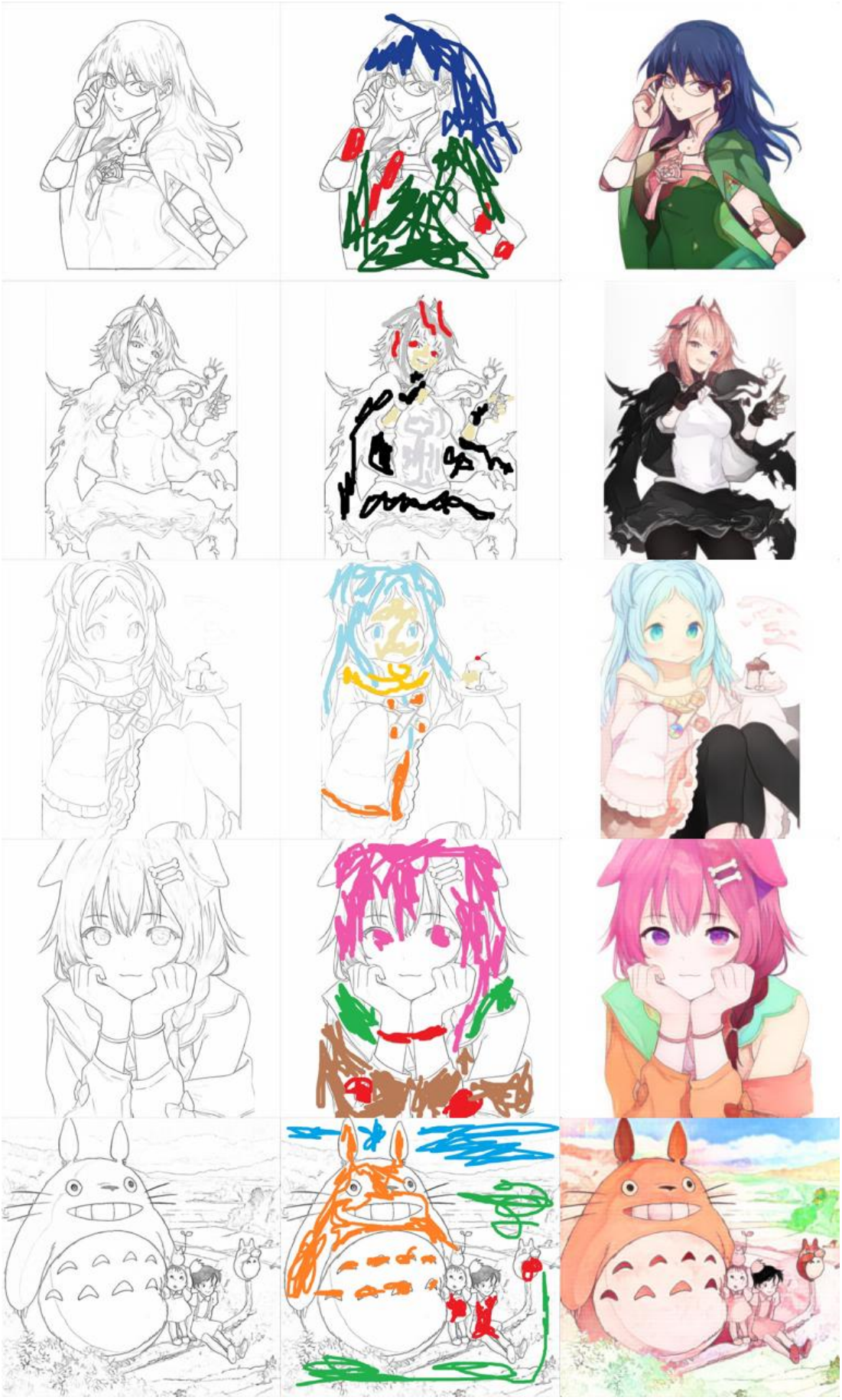
Zhang, L., Ji, Y., y Lin, X. (2017). Style transfer for anime sketches with enhanced residual u-net and auxiliary classifier GAN. *CoRR*, *abs/1706.03319*. Descargado de http://arxiv.org/abs/1706.03319

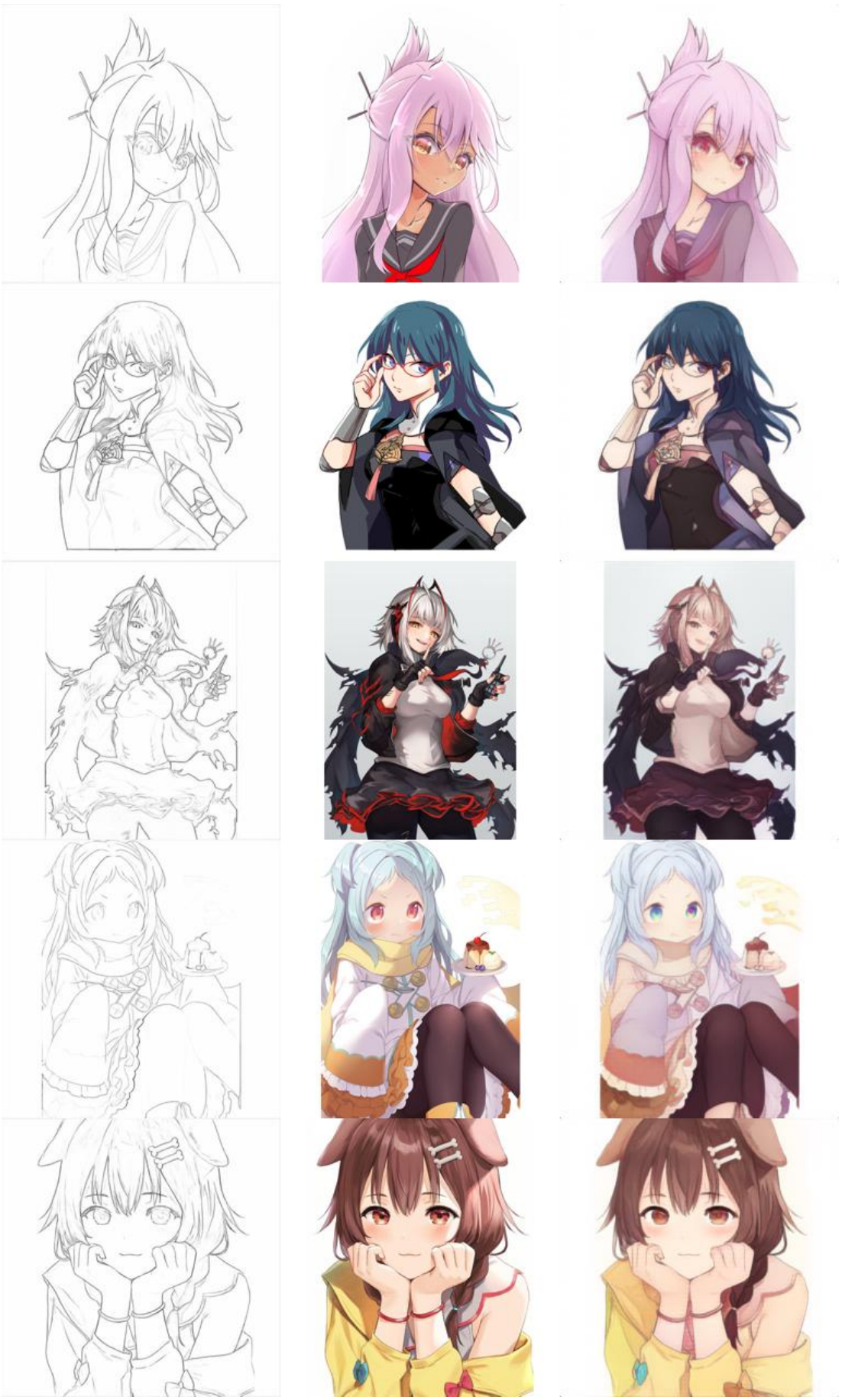
Zhang, M. R., Lucas, J., Hinton, G. E., y Ba, J. (2019). Lookahead optimizer: k steps forward, 1 step back. *CoRR*, *abs/1907.08610*. Descargado de http://arxiv.org/abs/1907.08610

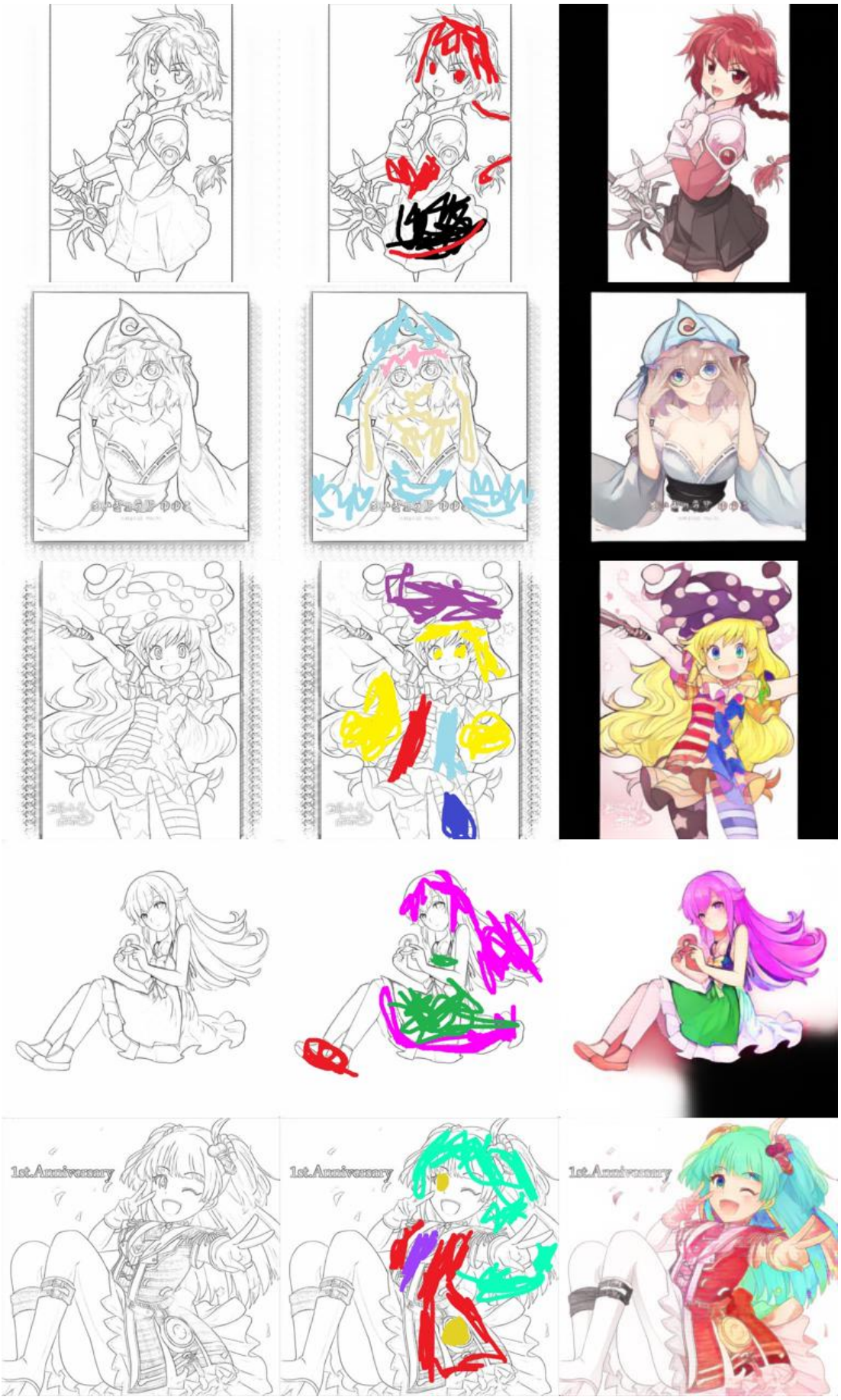
Zhou, Y.-T., y Chellappa, R. (1988). Computation of optical flow using a neural network. En *Icnn* (pp. 71–78).

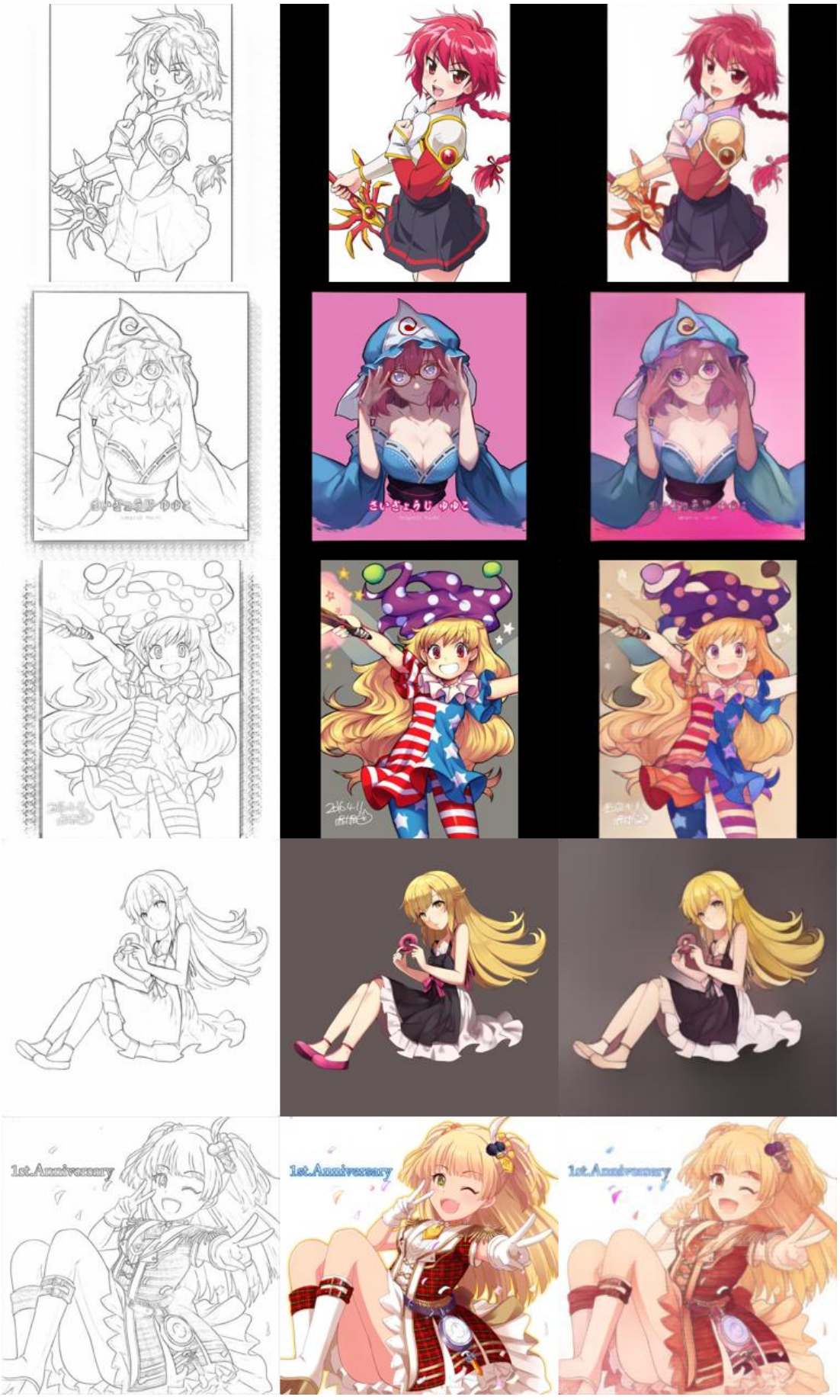
Apéndice A

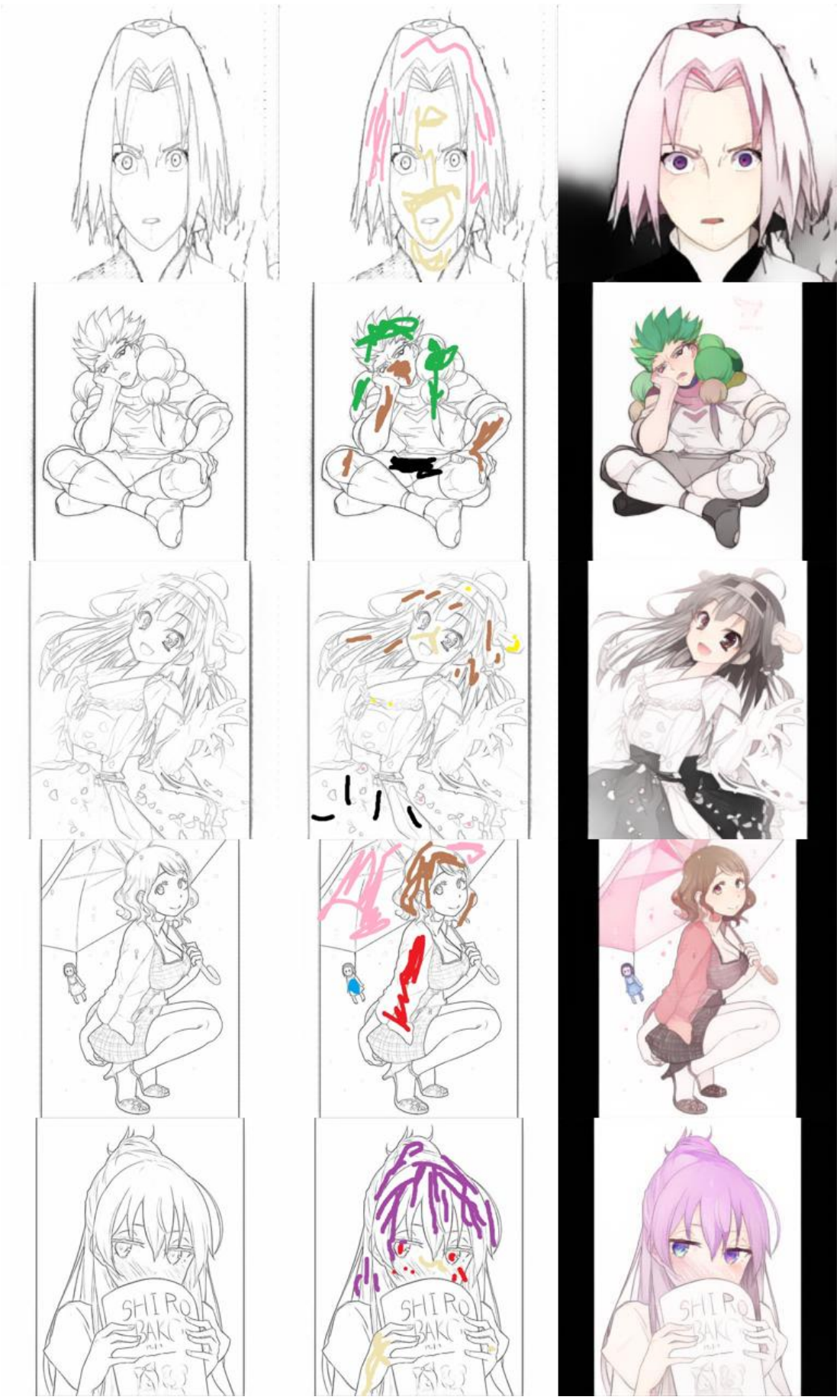
Muestras de coloración generadas por el modelo






1st.Anniversary
1st.Anniversary
1st.Anniversary


1st.Anniversary
1st.Anniversary
1st.Anniversary


SHIRO
SHIRO