THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

ITI
Institut für Technik und Informatik

Projektarbeit

KI-basiertes Kolorieren von Comics

Auf Basis von Dual Decoder Colorization (DDColor)

Modul

Künstlich Neuronale Netze und Deep Learning

Swen Thalhäuser - Mat: 5262160
Lars Kammerer - Mat: 5234062

im Juli / 2025

Dozent: Prof. Dr. Frank Kammer

Aufteilung der Arbeit

Die Kapitel 1, 2 und 7 wurden von Swen Thalhäuser geschrieben.

Die Kapitel 3, 4, 5 und 6 wurden von Lars Kammerer geschrieben.

Contents

1 Einleitung

In diesem Paper wird die Anwendbarkeit eines Algorithmus zur Rekolorierung von Bildern auf einem neu zusammengestellten Datensatz untersucht. Als Datengrundlage dienen frei verfügbare Comics aus dem Internet. Grundlage der Untersuchung bildet das Paper *DDColor* aus dem Jahr 2023[1]. Ziel der Arbeit ist es zu evaluieren, ob der Rekolorierungsalgorithmus durch ausreichend Training in der Lage ist ansprechende Comiczeichnungen zu erzeugen.

Die Autoren des Originalpapers geben an, dass ihr Algorithmus zum Zeitpunkt der Veröffentlichung alle bestehenden Verfahren zur Bildrekolorierung hinsichtlich der Farbqualität übertrifft – und das ohne Rückgriff auf händisch erstellte Priors oder große vortrainierte Datensätze zur Farbzuteilung von Objekten.

2 Technische Grundlage: DDColor

DDColor basiert auf einem Variational Autoencoder, der eine besondere Architektur mit zwei parallel arbeitenden Decodern verwendet. Während der erste Decoder die Rekonstruktion des Bildes aus den vom Encoder erzeugten Feature-Maps übernimmt, ist der zweite Decoder speziell für die Farbgebung zuständig. Anschließend werden die von beiden Decodern generierten Ausgaben kombiniert, um die Farbräume A und B der Graustufenbilder zu erzeugen. Diese werden dann mit dem bereits vorhandenen Luminanzkanal zusammengeführt, um das vollständige Farbbild zu erstellen. Eine anschauliche Darstellung dieses Ablaufs findet sich in Grafik 1. Innerhalb des Kapitels Grundlagen werden alle relevanten Bauteile erklärt.

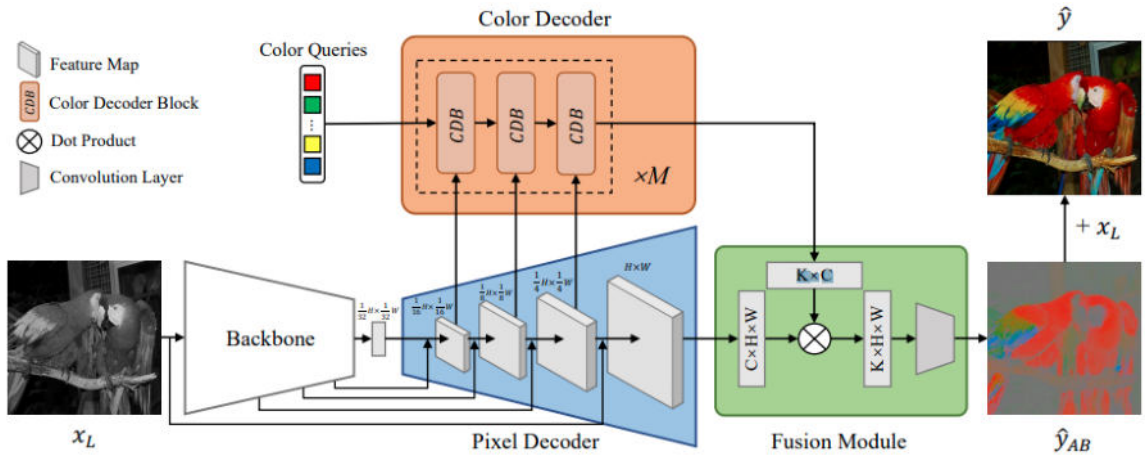


Figure 1: Architektur des DDColor Netzwerkes [1]

2.1 Farbraum CIELAB

Zu Beginn wird das Farbbild in ein Graustufenbild konvertiert. DDColor generiert anschließend auf Basis des vorhandenen Luminanzkanals (L) die beiden fehlenden Farbkanäle A und B im CIELAB-Farbraum. Dieser wurde 1976 von der *Commission Internationale de l'Éclairage* (CIE) definiert, um Farbdistanzen möglichst eng an die menschliche Wahrnehmung anzupassen.

- **L (Luminanz)**: Helligkeit von Schwarz (0) bis Weiß (100)
- **A**: Farbinformation zwischen Grün (-128) und Rot (+127)
- **B**: Farbinformation zwischen Blau (-128) und Gelb (+127)

2.2 Erklärung des Encoders

Der Encoder arbeitet innerhalb eines Backbone-Netzwerks, das relevante Merkmale aus dem Graustufenbild extrahiert. Als Backbone-Architektur kommt *ConvNeXt* zum Einsatz. Im Gegensatz zu klassischen Convolutional Neural Networks (CNNs) nutzt ConvNeXt unter anderem Depthwise Separable Convolutions. Dabei wird zunächst für jeden Kanal eine eigene 2D-Faltung (Depthwise Convolution) durchgeführt, gefolgt von einer kanalübergreifenden Faltung (Pointwise Convolution).

Die ConvNeXt-Architektur verwendet einen 7×7 Depthwise-Kernel, wodurch das Field of View vergrößert wird und das Modell eine größere "Weitsicht" erlangt – eine Idee, die aus Vision Transformers übernommen wurde. Außerdem ersetzt ConvNeXt die klassische Batch-Normalisierung durch LayerNorm und nutzt statt ReLU die aktivierungsfreundlichere Funktion Gaussian Error Linear Unit, kurz GELU. Zusätzlich kommt ein Inverted Bottleneck zum Einsatz, bei dem die Kanalanzahl vorübergehend erhöht wird, um die Lernfähigkeit für komplexe Merkmale zu verbessern.

Innerhalb der Bottleneck-Struktur werden für jede Eingabe (x_L) vier Feature-Maps mit unterschiedlichen Auflösungen erzeugt:

- Höhe und Breite geviertelt ($\frac{1}{4}$)
- Geachtelt ($\frac{1}{8}$)
- Gesechzehntelt ($\frac{1}{16}$)
- Durch 32 geteilt ($\frac{1}{32}$)

Die ersten drei Feature-Maps dienen über Shortcuts als zusätzliche Eingaben für den Pixel-Decoder. Die letzte Feature-Map bildet den primären Eingang für beide Decoder.

2.3 Erklärung des Pixeldecoders

Der Pixeldecoder besteht aus vier einzelnen Decoder-Schichten. Jede Schicht beginnt mit einem Convolution-Layer, der die Anzahl der Kanäle von C auf $C \times r^2$ erhöht, wobei r der Upsampling-Faktor ist. Dies ist notwendig, da sowohl die Höhe als auch die Breite des Bildes um den Faktor r vergrößert werden, wodurch sich die Gesamtzahl der Pixel um den Faktor r^2 erhöht.

Im Anschluss werden die erweiterten Kanäle durch eine PixelShuffle-Upsampling-Schicht räumlich umstrukturiert. Formal besitzt die Eingabe der PixelShuffle-Schicht die Form $(H, W, C \times r^2)$, wobei H und W die Höhe und Breite der Feature-Map vor dem Upsampling darstellen. Die PixelShuffle-Schicht ordnet die Kanäle so um, dass sie in ein räumliches Raster der Größe $r \times r$ aufgeteilt werden.

Für jedes Pixel (h, w) und Kanal c in der Ausgabemap mit der Form (rH, rW, C) werden die Werte aus den Kanälen

$$c \times r^2 + i \times r + j$$

der Eingabemap an die Position

$$(r \cdot h + i,\ r \cdot w + j,\ c)$$

der Ausgabemap verteilt, wobei $i, j \in \{0, \ldots, r-1\}$ die Position innerhalb des $r \times r$-Blocks angeben. Mathematisch lässt sich dies durch folgende Zuordnung beschreiben:

$$\mathbf{output}[r \cdot h + i,\ r \cdot w + j,\ c] = \mathbf{input}[h,\ w,\ c \times r^2 + i \times r + j].$$

Durch diese Umstrukturierung wird die räumliche Auflösung um den Faktor r in Höhe und Breite erhöht, während die Kanalanzahl von $C \times r^2$ auf C reduziert wird. Im Vergleich zu anderen Upsampling-Methoden wie Deconvolution oder Interpolation reduziert PixelShuffle

Artefakte wie Checkerboard-Effekte und ermöglicht eine effiziente Berechnung.

Die kleinste (tiefste) Decoderschicht endet mit diesem Upsampling-Schritt. In den weiteren Schichten folgt nach dem Upsampling jeweils eine Convolution, die die hochaufgelösten Features mit den Shortcut-Verbindungen aus dem Encoder kombiniert. Diese Skip Connections stellen sicher, dass semantische Informationen aus dem Encoder erhalten bleiben und die Detailgenauigkeit verbessert wird. Anschließend folgt ein weiterer Convolution-Layer, der neue Feature-Maps lernt und für die nächste Decoderschicht vorbereitet.

Die Outputs aller Decoderschichten bilden eine Feature-Pyramide mit unterschiedlichen Auflösungen, die als Input für den Farbdecoder dienen. Diese Architektur adressiert Probleme wie Informationsverlust und Unschärfe, die bei einfachem Upscaling entstehen können, und verbessert somit die Qualität der Bildfarbgebung.

2.4 Erklärung des Color Decoders

Viele bestehende Verfahren zur automatischen Kolorierung von Bildern basieren auf externen Priors, um lebendige und realistische Farbergebnisse zu erzielen. Solche Priors umfassen beispielsweise vortrainierte generative Modelle (z. B. GANs), statistische Farbverteilungen aus Datensätzen oder manuell erstellte semantisch-farbige Zuordnungen. Diese Ansätze erfordern jedoch einen erheblichen Vorbereitungsaufwand und sind oft nur eingeschränkt auf andere Anwendungsszenarien übertragbar.

Um die Abhängigkeit von solchen manuell definierten Informationen zu reduzieren, schlagen die Autoren eine neuartige, query-basierte Farbkodierungskomponente (Color Decoder) vor. Diese basiert auf einer modifizierten Version eines Transformer-Decoders und verwendet adaptive Farbqueries, die lernfähig aus den visuellen Bildmerkmalen abgeleitet werden.

2.4.1 Aufbau des Color Decoders (CDB)

Der Color Decoder besteht aus mehreren aufeinanderfolgenden Blöcken, den sogenannten Color Decoder Blocks (CDBs). Jeder Block verarbeitet zwei Eingaben:

- Visuelle Bildmerkmale **(K und V)** $\mathcal{F}_l \in \mathbb{R}^{H_l \times W_l \times C}$ aus früheren Netzwerkstufen
- Farbqueries **(Q)**, die durch trainierbare Farb-Embeddings $\mathcal{Z}_0$ initialisiert werden

Die Farb-Embeddings $\mathcal{Z}_0$ sind eine Matrix der Form:

$$\mathcal{Z}_0 = [Z_0^1, Z_0^2, \ldots, Z_0^K] \in \mathbb{R}^{K \times C}$$

wobei

- K die Anzahl der Farbqueries ist
- C die Dimensionalität jedes Farbembeddings (z. B. 256) ist

Zu Beginn des Trainings sind alle Einträge in $\mathcal{Z}_0$ nullinitialisiert. Diese Embeddings werden anschließend durch Cross-Attention mit den Bildmerkmalen $\mathcal{F}_l$ semantisch angereichert:

$$\mathcal{Z}_l' = \text{softmax}(Q_l K_l^\top) V_l + \mathcal{Z}_{l-1} \tag{1}$$

Dabei ergeben sich:

$$Q_l = f_Q(\mathcal{Z}_{l-1}) \in \mathbb{R}^{K \times C}, \quad K_l = f_K(\mathcal{F}_l), \quad V_l = f_V(\mathcal{F}_l) \in \mathbb{R}^{H_l W_l \times C}$$

Die Funktionen f_Q, f_K und f_V sind lineare Projektionsschichten (Fully Connected Layers), die die Queries, Keys und Values für die Attention erzeugen.

2.4.2 Transformer-Verarbeitung

Nach der Cross-Attention folgt eine klassische Transformer-Verarbeitung der angereicherten Farbqueries:

$$\mathcal{Z}_l'' = \mathrm{MSA}(\mathrm{LN}(\mathcal{Z}_l')) + \mathcal{Z}_l' \tag{2}$$
$$\mathcal{Z}_l''' = \mathrm{MLP}(\mathrm{LN}(\mathcal{Z}_l'')) + \mathcal{Z}_l'' \tag{3}$$
$$\mathcal{Z}_l = \mathrm{LN}(\mathcal{Z}_l''') \tag{4}$$

wobei

- MSA (Multi-Head Self-Attention): Modelliert die Beziehungen zwischen Farbqueries
- MLP (Feedforward-Netzwerk): Führt eine nicht-lineare Transformation durch
- LN (Layer Normalization): Stabilisiert die Gradienten und verbessert das Training.

Wichtig ist, dass die Cross-Attention vor der Self-Attention erfolgt, da die Farbqueries zunächst semantisch leer sind und erst durch das Bildkontextwissen angereichert werden.

2.4.3 Multi-Skalen-Verarbeitung

Ein häufiges Problem früherer Kolorierungsmethoden besteht darin, dass sie Bildmerkmale nur auf einer festen Skala (z. B. $\frac{1}{16}$ der Auflösung) verarbeiten. Dadurch gehen feine semantische Details verloren, was zu Artefakten wie Color Bleeding führen kann. Der vorgeschlagene Color Decoder verwendet daher drei Auflösungsebenen:

$$\mathcal{F}_1 : \text{Feature Map mit Auflösung } \frac{1}{16}, \quad \mathcal{F}_2 : \frac{1}{8}, \quad \mathcal{F}_3 : \frac{1}{4}$$

Diese stammen aus einem vorgelagerten Pixel-Decoder und werden in den Farbdecoder eingespeist:

$$\mathcal{E}_c = \mathrm{ColorDecoder}(\mathcal{Z}_0, \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3) \tag{5}$$

Dabei ist

$$\mathcal{E}_c \in \mathbb{R}^{K \times C}$$

die finale farbsensitive Repräsentation, die sowohl semantisches als auch visuelles Wissen kodiert.

2.4.4 Organisation der Decoder-Blöcke

Die Decoder-Blöcke werden in Gruppen von jeweils drei CDBs organisiert. In jeder Gruppe wird jede Skala einmal sequenziell verarbeitet. Dieser Gruppenprozess wird M-mal wiederholt, was zu insgesamt

$$\text{Anzahl der CDBs} = 3M$$

Blöcken führt.

Um den Ablauf besser zu veranschaulichen, wird Grafik 2 herangezogen. Der Input Q stellt dabei die Farb-Queries dar, welche die zuvor beschriebenen Farb-Embeddings Z_0 enthalten. Die Inputs K und V sind visuelle Bildmerkmale, die aus den Shortcuts des Pixeldecoders stammen. Dabei steht K für den Key und V für den zugehörigen Value.

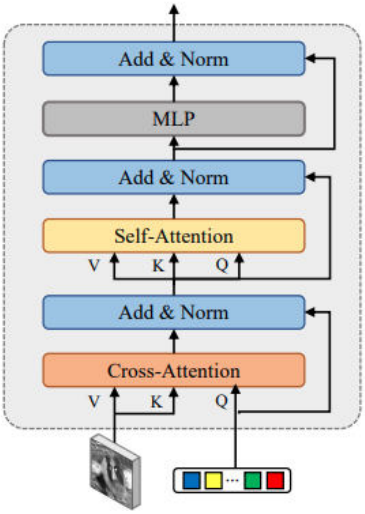


Figure 2: Aufbau des Color Decoder Blocks [1]

2.5 Erklärung des Fusionsmoduls

Das Fusionsmodul stellt eine kompakte und recheneffiziente Komponente dar, deren Ziel es ist, die lokalen Bildinformationen aus dem Pixel-Decoder mit den semantisch angereicherten Farbrepräsentationen des Color-Decoders zu einer konsistenten Farbvorhersage zu vereinen. Dieser Schritt bildet die letzte Stufe der Kolorierungspipeline und ist entscheidend für die Synthese qualitativ hochwertiger Farbbilder. Die zwei Eingaben des Fusionsmoduls sind:

- Das per-Pixel-Bildembedding
$$E_i \in \mathbb{R}^{C \times H \times W},$$
welches aus dem Pixel-Decoder stammt und lokale Strukturinformationen des Graustufenbildes enthält.

- Die semantisch-informierte Farbrepräsentation
$$E_c \in \mathbb{R}^{K \times C},$$
die aus dem Color-Decoder resultiert und globale Farbkonzepte über das Bild hinweg kodiert.

Zur Fusion dieser beiden Repräsentationen wird eine skalar-produkt-basierte Projektion berechnet, welche jedem Farbquery eine Aufmerksamkeit über das gesamte Bild zuweist:

$$\hat{\mathcal{F}} = E_c \cdot E_i \tag{6}$$

Dabei handelt es sich um ein skalares Produkt zwischen den Farbqueries (K) und den räumlich verteilten Bildfeatures ($H \times W$). Das resultierende Tensorfeld

$$\hat{\mathcal{F}} \in \mathbb{R}^{K \times H \times W}$$

beschreibt somit, wie stark jedes Farbquery auf einzelne Regionen des Bildes wirkt – vergleichbar mit einem „Farbaufmerksamkeitsprofil“.

2.6 Erzeugung des finalen Bildes

Um die generierten Merkmale in konkrete Farbinformationen zu transformieren, wird eine kanalfokussierte Faltung mittels einer 1×1-Convolution angewendet:

$$\hat{y}_{AB} = \mathrm{Conv}_{1\times1}(\hat{\mathcal{F}}), \quad \hat{y}_{AB} \in \mathbb{R}^{2\times H\times W} \tag{7}$$

Diese Operation reduziert die Dimensionalität von K auf 2 und liefert so direkt die AB-Kanäle des Bildes im CIELAB-Farbraum. Die Faltung wirkt dabei ausschließlich kanalübergreifend und erhält die räumliche Auflösung vollständig bei. Abschließend wird das finale kolorierte Bild

$$\hat{y} \in \mathbb{R}^{3\times H\times W}$$

durch Konkatenation der vorhergesagten Farbkomponenten $\hat{y}_{AB}$ mit dem ursprünglichen Luminanzkanal

$$x_L \in \mathbb{R}^{1\times H\times W}$$

erzeugt:

$$\hat{y} = \mathrm{Concat}(x_L, \hat{y}_{AB})$$

Damit liegt das Endergebnis im vollständigen CIELAB-Farbraum vor und kann anschließend für Visualisierungs- oder Weiterverarbeitungszwecke in den RGB-Farbraum konvertiert werden.

2.7 Loss-Funktionen

Für das Training des vorgeschlagenen Farbkolorierungsmodells wird ein zusammengesetzter Zielverlust verwendet, der verschiedene Aspekte der Bildqualität berücksichtigt. Insgesamt kommen fünf komplementäre Verlustfunktionen zum Einsatz, um sowohl die strukturelle Genauigkeit, die semantische Konsistenz als auch die visuelle Attraktivität der generierten Farbbilder zu maximieren:

1. **Pixelbasierter Loss $\mathcal{L}_{pix}$**
   Dieser auf der L1-Norm basierende Verlust misst die absolute Differenz zwischen dem generierten Bild $\hat{y}$ und dem Ground-Truth y auf Pixelebene:

   $$\mathcal{L}_{\mathrm{pix}} = \|\hat{y} - y\|_1 \tag{1}$$

   Er dient als direkte Supervision und stellt sicher, dass die generierten Bilder möglichst nah an den echten Farbbildern liegen.

2. **Perzeptueller Loss $\mathcal{L}_{per}$**
   Zur Erfassung der semantischen Bildähnlichkeit werden Feature-Repräsentationen aus einem vortrainierten VGG16-Netzwerk extrahiert. Der Verlust vergleicht diese Repräsentationen zwischen dem generierten Bild $\hat{y}$ und dem echten Bild y:

$$\mathcal{L}_{\text{per}} = \sum_{l \in \mathcal{L}} \lambda_l \cdot \|\phi_l(\hat{y}) - \phi_l(y)\|_2^2 \tag{2}$$

   Hierbei bezeichnet $\phi_l(\cdot)$ die Aktivierung der l-ten Schicht. Dieser Verlust sorgt dafür, dass die kolorierten Bilder nicht nur pixelweise, sondern auch inhaltlich konsistent sind.

3. **Adversarial Loss für den Generator $\mathcal{L}_{adv}$**
   Ein PatchGAN-Discriminator wird eingesetzt, um zwischen echten und generierten Bildern zu unterscheiden. Der Generator wird dabei so trainiert, dass er Bilder erzeugt, die möglichst realistisch wirken:

$$\mathcal{L}_{adv} = -\mathbb{E}_{\hat{y}}[\log D(\hat{y})]$$

   Diese Komponente trägt wesentlich zur visuellen Natürlichkeit der Ergebnisse bei.

4. **Discriminator Loss $\mathcal{L}_D$**
   Der Diskriminator selbst wird mit einem klassischen Binary Cross Entropy Loss trainiert, um reale Bilder als echt und generierte Bilder als künstlich zu klassifizieren:

$$\mathcal{L}_{\text{adv}} = -\log\left(D(\hat{y})\right) \tag{3}$$

   Dadurch wird der Diskriminator darin bestärkt, authentische Farbverteilungen von generierten zu unterscheiden – was wiederum die Lernkurve des Generators beeinflusst.

5. **Farbintensitätsloss $\mathcal{L}_{col}$**
   Ein neu eingeführter Verlustterm, der die Farbintensität der generierten Bilder maximieren soll. Der sogenannte Colorfulness Score wird basierend auf der Standardabweichung und dem Mittelwert der Farbpixelverteilung berechnet:

$$\mathcal{L}_{\text{col}} = 1 - \frac{\sigma_{\text{rgyb}}(\hat{y}) + 0.3 \cdot \mu_{\text{rgyb}}(\hat{y})}{100} \tag{4}$$

   Diese Komponente fördert lebendige und visuell ansprechende Resultate, ohne die semantische Konsistenz zu beeinträchtigen.

Die finale Verlustfunktion des Generators setzt sich als gewichtete Summe der einzelnen Komponenten zusammen:

$$\mathcal{L}_\theta = \lambda_{pix} \cdot \mathcal{L}_{pix} + \lambda_{per} \cdot \mathcal{L}_{per} + \lambda_{adv} \cdot \mathcal{L}_{adv} + \lambda_{col} \cdot \mathcal{L}_{col}$$

Die Hyperparameter $\lambda_{pix}, \lambda_{per}, \lambda_{adv}, \lambda_{col}$ dienen der Gewichtung der jeweiligen Verlustanteile und werden experimentell bestimmt.

3 Aufsetzen

Im Folgenden werden alle nötigen Schritte erläutert, um das Projekt *"DDColor"* inkl. Trainings-Pipeline aufzusetzen. An dieser Stelle sei auf *"https://github.com/piddnad/DDColor"* verwiesen. Die dort angegebene "READ.ME" Datei gibt eine Grundlage. Die Autoren geben an, dass eine Python-Version 3.7 oder höher benötigt wird. Dies stimmt nur teils. Die angegebenen Vorausgesetzten Pakete bzw. Module erzeugen Versionskonflikte bei nahezu allen Python Versionen. Durch ausprobieren ließ sich die Python-Version 3.10.18 als die unproblematischste Version identifizieren.

Es ergibt sich also folgender Aufbau, der sich sowohl unter Windows als auch Linux bewährt hat.

```
1    conda create -n ddcolor python=3.10.18
2    conda activate ddcolor
3    conda install -c conda-forge dlib
```

Listing 1: conda dlib work around

Darauffolgend können alle im Projekt vorgegebenen Befehle ausgeführt werden, ohne dabei Versionskonflikte zu erzeugen. Leider ist bisher keine funktionierende Möglichkeit bekannt das Paket "dlib" ohne diesen Umweg zu installieren und zu nutzen. Für die Ausführung der Trainings-Pipeline ist diese jedoch unverzichtbar. Das Paket "dlib" setzt CMake voraus, welches aus bisher nicht bekannten Gründen sowohl unter Linux als auch Windows nicht korrekt erkannt wird. Selbst die von den Autoren genutzte Python-Version 3.9 funktioniert nicht.

4 Datensatz & Vorbereitung

Im Folgenden wird die Organisation, Nutzung und Aufbereitung der für das Training benötigten Daten beschrieben. Dazu ist zu verdeutlichen, welche Daten genutzt werden und warum. Anschließend wird erläutert, wie die Datensätze für ein Training aufbereitet werden und welche Überlegungen zu einzelnen Entscheidungen geführt haben. Dabei wird sich lediglich auf die beschriebenen Datensätze bezogen, weitere Datensätze, die ggf. bei der Suche auftraten werden nicht berücksichtigt.

Um die Idee, Zeichnungen mithilfe künstlicher Intelligenz farblich auszufüllen, umsetzen zu können, werden zunächst möglichst viele Bilder benötigt, die sowohl farbig vorliegen als auch leicht in eine zeichnerische, farblose Form überführt werden können. Dazu eignen sich eindeutig gemalte Bilder in Farbe. Comic-Zeichnung bzw. Comics im Allgemeinen bilden diese Voraussetzungen vollständig ab. Entsprechende Datensätze stehen online ausreichend zur Verfügung. Zwei geeignete Datensätze sind:

Comic-9K:[2] Folgend auch *"Comic-Modern"*, ist ein Datensatz mit einer Größe von 190 GB und umfasst modern gezeichnete Comic-Heftseiten. Sie sind **vorwiegend** in ihrer digitalen Ursprungsform verfügbar und entsprechend klar und deutlich. Es ist zu beachten, dass einige dieser Bilder eine Seitengröße von 2000 Pixel überschreiten können. Des Weiteren handelt es sich um Seiten von Comicheften, entsprechend sind auch teils Titelbilder bzw. Titelblätter mit eingearbeitet, sie sind jedoch angesichts der Menge an Daten zu vernachlässigen.

Figure 3: Beispielhafte Abbildung einiger Bilder in Comic-9K. Bilder dienen nur der Veranschaulichung und sind entsprechend zugeschnitten. Enthält Bilder aus: [2]

Comic-Panels:[3] Folgend auch *"Comic-Old"* genannt, ist ein Datensatz, der Abbilder von alten gezeichneten Comics enthält und 64 GB groß ist. Es sind Panels (Ausschnitte) entsprechender Comicseiten. Die Panels stehen nicht in ihrem digitalen Original zur Verfügung und haben entsprechende Eigenschaften eines Fotos. Da es sich um augenscheinlich gezeichnete ältere Comics handelt, ist auf den meisten Panels ein leichter Gelbstich zu erkennen. Die Bilder überschreiten eine Größe von 800 Pixeln nicht.

Figure 4: Beispielhafte Abbildung einiger Bilder in Comic-Panels. Bilder dienen nur der Veranschaulichung und sind entsprechend zugeschnitten. Enthält Bilder aus: [3]

Anmerkung: Es existiert ebenfalls eine nicht in Panels aufgeteilte Version von *"Comic-Panels"*. Diese wird jedoch der Einfachheit halber nicht genutzt, da ggf. einige Bilder im Zuge der Aufbereitung zurechtgeschnitten werden und ein Großteil der Arbeit beim Panels bereits gemacht wurde.

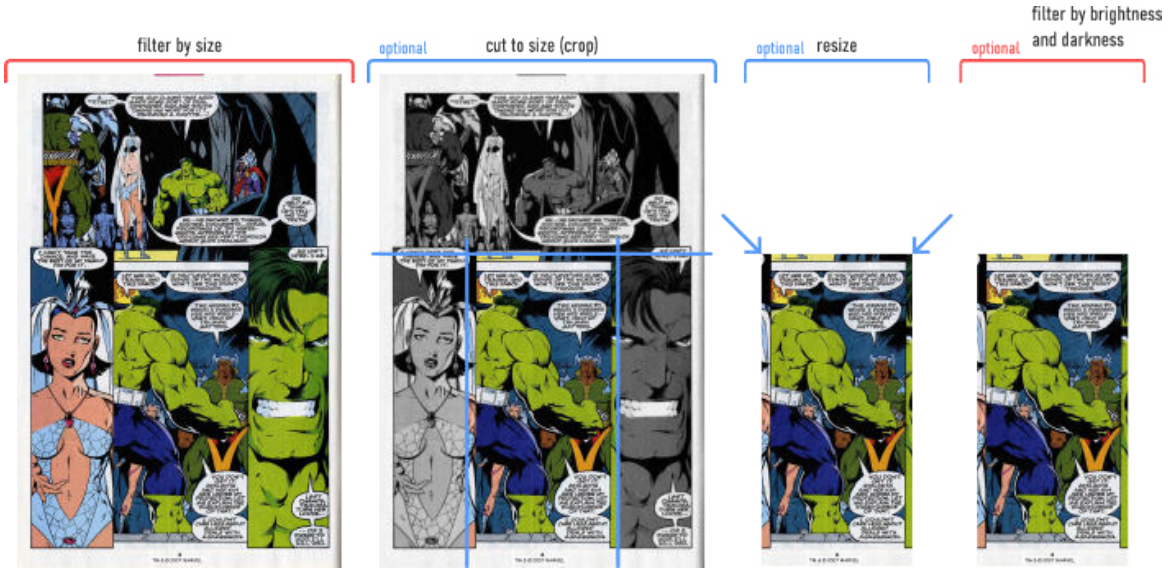


Figure 5: Visualisierung der Datensatz-Aufbereitung. Enthält Bilder aus: [2]

filter by size: Filter von Bildern anhand einer maximalen und minimalen Pixel-Grenze.

cut to size (crop): Darunter ist das Runterschneiden auf einen Pixelwert zu verstehen. Es ist variierbar, wie horizontal oder vertikal geschnitten werden soll. Dies ist unter Anderem relevant, da sich z.B. beim Datensatz *"Comic-Old"* viele unbrauchbare Sprechblasen im oberen Teil des Bildes befinden. Das obere Segment ist daher häufig unbrauchbar und kann weggeschnitten werden.

resize: Darunter versteht sich das Verkleinern des Bildes ohne Schnitt. Zu beachten ist die Verarbeitung des Bildes im Trainingsprozess. Ist ein Bild zu stark komprimiert, kann es hier zu Problemen kommen, da Konturen nicht nachgezogen werden können. Siehe Kapitel: 6.

filter by brightness and darkness: Filtert Bilder mit zu hohem Weiß oder Schwarzanteil nach Prozent heraus. Ein zu hoher Weißanteil deutet z.B. auf zu viel Text hin, während ein zu hoher Schwarzanteil Probleme beim Training verursachen kann. Schwarze Flächen sind im allg. ungünstig, da sie auch im Training schwarz bleiben und daher wenige Farbinformationen enthalten.

5 Implementierung

In diesem Kapitel werden alle Anpassungen an dem im Paper verlinkten Projekt beschrieben. Dazu wird zunächst erläutert, an welchem Punkt eine Änderung vorgenommen wird und welches Resultat daraus entstanden ist. Alle Modifikationen beschränken sich auf den Punkt des Projektes, an dem sich *"ground truth (gt)"* und *"low quality (lq)"* trennen. Eine Implementierung des Projektes ist hier zu finden: *https://github.com/Larisio/ddcolor*

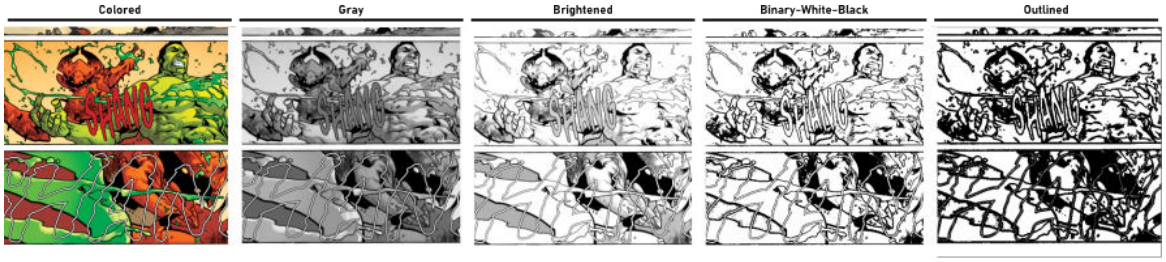


Figure 6: Visualisierung des Aufbereitungsprozesses von Farbzeichnung in Zeichnung. Enthält Bilder aus: [2]

Um eine farbige Zeichnung in eine ungefärbte Zeichnung zu transformieren, bedarf es nur wenige angleichende Schritte. Zunächst muss das Bild nach dem Laden vom RGB-Farbraum in den LAB-Farbraum umgewandelt werden. Der wesentliche Unterschied zwischen diesen beiden Farbabbildungen ist die Komplexität für das Umwandeln in ein Schwarz-Weiß-Bild. In RGB wird jeder Pixel mit den Grundfarben Rot, Grün und Blau codiert. In LAB ist es Helligkeit (L) und eine Abbildung a*b, die den Farbton vorgeben. Entsprechend können a*b durch Nullwerte ersetzt werden, was übrig bleibt sind dann Grautöne bzw. Helligkeit.

Im zweiten Schritt wird das Bild aufgehellt. In dieser Implementierung kann für jedes Training ein entsprechender Aufhellungsfaktor gesetzt werden. Dies ist gerade bei dunklen Bildern wichtig, um Konturen einer Zeichnung deutlicher von Flächen trennen zu können.

In den letzten beiden Schritten wird das Bild in eine Binär-Form gebracht. Es werden alle Pixel, die einen bestimmten, durch Einstellungen variierbaren, Hellichkeitswert überschreiten, weiß und der Rest wird Schwarz gefärbt. Um Flächen besser abzuschließen, werden anschließend Konturen nachgefahren. Dieser Schritt passiert zuletzt, da ein Nachziehen der Konturen auf einem Schwarz-Weiß-Bild häufig in schwarzen Flächen mündet und den Trainingsdatensatz unvorhersehbar beeinflussen kann. Er ist optional.

Das durch diese Schritte entstandene Bild wird ab diesem Punkt als *"lq"* genutzt. Alle beschriebenen Schritte sind einfach mit dem Python-Paket OpenCV2 umzusetzen.

6 Training

Das folgende Kapitel beschreibt die Erkenntnis aus mehreren Trainings-Durchläufen, die zu wenigen primären Ergebnissen geführt haben. Anhand der Ergebnisse werden angewandte Variationen verdeutlicht und entsprechende Erkenntnisse gewonnen.

6.1 Parameter

Durch die ausgearbeitete Trainings-Pipeline ist es möglich, alle relevanten Trainingsparameter nach belieben einzustellen. Die meist genutzten sind:

Name	**Beschreibung**
gt_size	Größte des *"gt"* Bildes in Pixel.
color_enhance, color_enhance_factor	Boolean, ob die Farben nachbearbeitet werden sollen und wenn ja, zu welchem Faktor.
threshold, threshold_replace	Schwellwert, ab wann ein Pixel mit dem Farbwert "threshold_replace" ersetzt werden soll.
outline_thickness	Dicke der Konturen in Pixel.
brightness_factor	Um welchen Faktor ein Bild aufgehellt werden soll.
total_iter	Anzahl Iterationen
milestones	Meilensteine der Multistep Lernrate
lr	Lernrate

Table 1: Auflistung der wichtigsten Trainingsparameter. **Fett** markiert für diese Implementierung hinzugefügt.

Nach wenigen kurzen Durchläufen zeigt sich ein Muster. Farben werden nur in sehr vielen Iterationen aufgebaut. Dies deckt sich mit den Angaben der Autoren. Ebenfalls zeichnet sich ab, dass eine größere Datenmenge und ein höherer Pixelwert für *"gt_size"* (512) langsamer eine im Ansatz erfolgreiche Färbung erreicht.

Zu beobachten ist dies in den drei primären Trainingsdurchläufen:

- *"train_ddcolor_l_run_modern_512_20k_long"*
- *"train_ddcolor_l_run_modern_256_8k_long"*
- *"train_ddcolor_l_run_old_256_10k_long"*

Die beiden ersten Trainingsdurchläufe bauen auf der gleichen Datengrundlage (Comic-Modern) auf. Dem Namen entsprechend mit 20.000 und 8.000 Bildern. Beide Datensätze sind nach den gleichen Vorgaben ausgewählt und auf 512 Pixel zugeschnitten und gedrückt. Der zweite Durchlauf nutzt als *"gt_size"* einen Pixelwert von 256 anstatt 512 Pixel. Beide Trainings liefen sonst unter gleichen Parametern 120.000 Iterationen durch.

Weiterführend sind folgende Begriffserklärungen relevant:

fid (Fréchet Inception Distance) misst den Unterschied zwischen der Verteilung von echten und generierten Bildern in einem Merkmalsraum, der von einem vortrainierten neuronalen Netzwerk erzeugt wird.

cf (Colorfulness) ist eine numerische Metrik, die angibt, wie intensiv und vielfältig die Farben in einem Bild sind.

Der erste Trainingslauf (links) hat in seiner Gesamtheit bessere Werte für *"fid"* und *"cf"*, jedoch werden die besten Ergebnisse wesentlich später erzielt. Zusätzlich ist zu bedenken, dass eine größere Eingabe erheblich mehr Rechenaufwand bedeutet. Der "Kosten-Nutzen-Faktor" zahlt sich, angesichts dieser Ergebnisse, nicht aus.

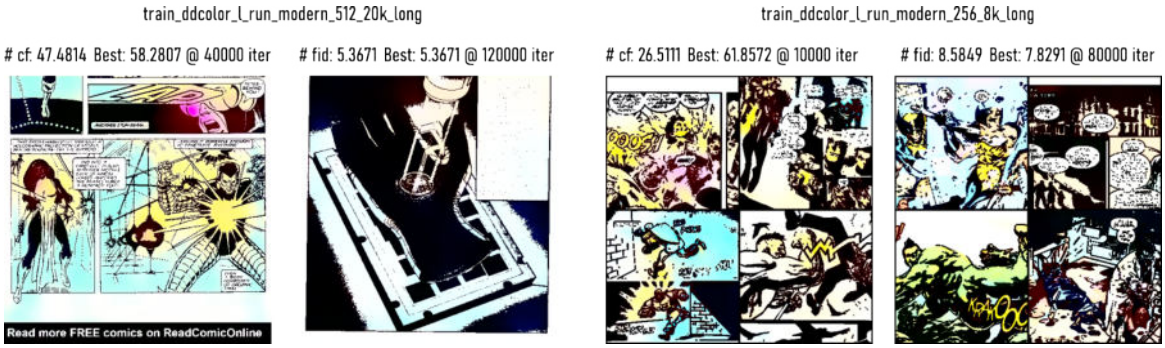


Figure 7: Vergleich der Trainingsläufe *"train_ddcolor_l_run_modern_512_20k_long"* gegen *"train_ddcolor_l_run_modern_256_8k_long"*. **fid** (lower is better), **cf** (higher is better).

Anmerkung: Der erste Wert ist beim Abschluss des Trainings erzeugt, während *"Best"* die entsprechend beste Leistung bei bestimmter Iteration ist. Die jeweiligen Bilder sind das Ergebnis von *"Best"*. Dies wird fortlaufend bei ähnlichen Inhalten genau so gehandelt.

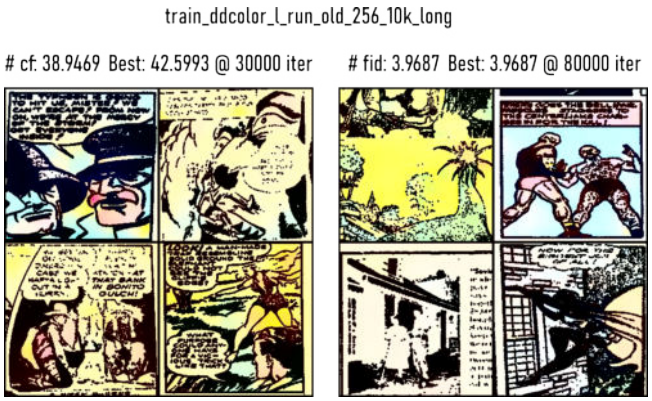


Figure 8: Darstellung der Ergebnisse des Trainings-Durchlauf *"train_ddcolor_l_run_old_256_10k_long"*

Anmerkung: Im Anhang befindet sich eine graphische Darstellung des Pixel-Loss, der ebenfalls verdeutlicht, warum Durchläufe mit 512 Pixel Eingaben ungeeignet sind.

Mit einer Eingabegröße von 256 Pixel, 80.000 Iterationen und 10.000 Bildern als Datengrundlage generiert aus dem Datensatz *"Comic-Old"* werden die zu diesem Punkt besten Werte erreicht. Der Wert *"fid"* ist besser als bei beiden vorangegangenen Läufen. Wobei der Wert für *"cf"* zwischen den beiden Vorgängern liegt.

Aus mehreren kurzen Durchläufen zeigt sich auch: Flächen aus ausschließlich weißen Pixel sorgen für ein schlechteres Training. Der Datensatz *"Comic-Old"* weist einen altertümlichen gelben Schimmer auf, der für bessere Werte zu sorgen scheint. Dieser Effekt kann künstlich durch den Parameter "threshold_replace" unterstützt werden. Ein Wert von ca. 190 erzeugt eine leichte graue Schattierung, die diesen Effekt stützt. Durch diese Erkenntnisse ergibt sich der voraussichtlich finale Trainingsversuch.

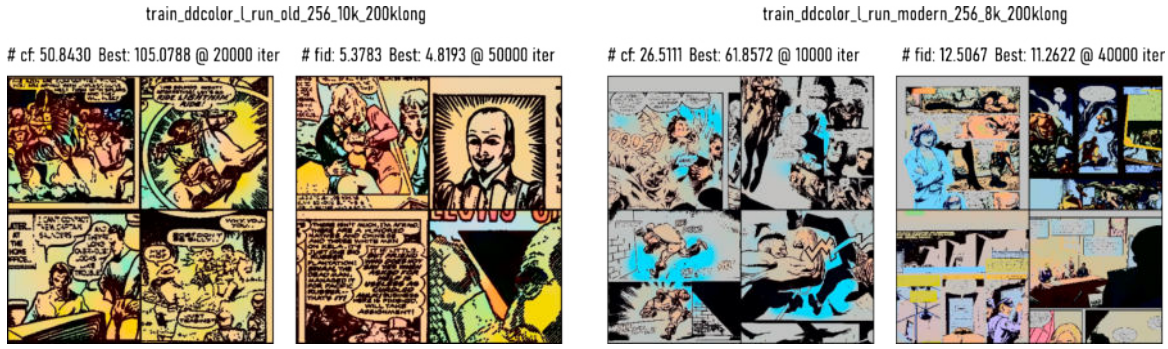


Figure 9: Vergleich der Trainingsläufe *"train_ddcolor_l_run_old_256_10k_200klong"* gegen *"train_ddcolor_l_run_modern_256_8k_200klong"*. **fid** (lower is better), **cf** (higher is better).

Der Trainingslauf *"train_ddcolor_l_run_old_256_10k_200klong"* zeigt bessere Resultate und dies nicht nur visuell. Unter Betrachtung der im Anhang verfügbaren Grafiken ist zu erkennen, dass gerade der *"real score"* auf den modernen Daten Probleme bereitet, während er sich auf dem Datensatz *"Comic-Old"* stabilisiert. Im *"fake score"* sieht der Graf für *"Comic-Old"* leicht besser aus, da er eher Richtung 0 tendiert. Der GAN-Loss ist instabil für beide Verläufe, wobei *"train_ddcolor_l_run_old_256_10k_200klong"* leicht Richtung niedrigere Werte tendiert.

Figure 10: Ergebnis nach dem Training auf Validierungsbild.
Training: *"train_ddcolor_l_run_old_256_10k_200klong"*

Unter Betrachtung eines Bildes aus dem Validierungssatz ist zu erkennen, dass durchaus Farben auf Strukturen des Bildes trainiert wurden. Kleidung und Gesichter sind im Rahmen ihrer Struktur gefärbt. Ebenfalls ist ein klarer Gelbstich zu erkennen, welcher jedoch aufgrund des Trainingsdatensatzes zu erwarten war.

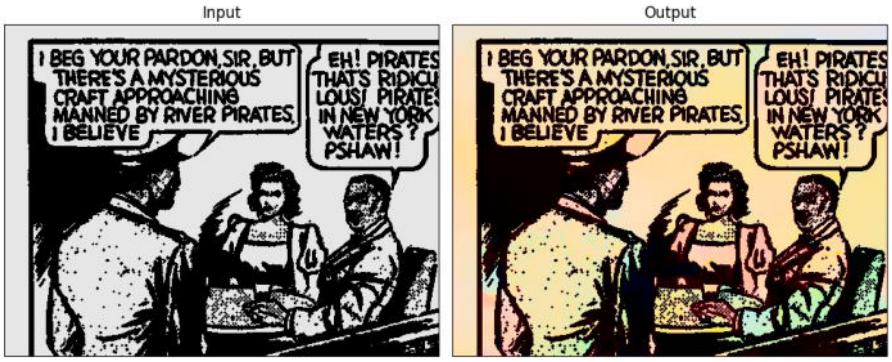


Figure 11: Ergebnis nach dem Training auf ein Bild außerhalb des Trainings- oder Validierungsdatensatzes. Training: *"train_ddcolor_l_run_old_256_10k_200klong"*

Die Abbildung 11 zeigt, dass Bilder, die außerhalb der Trainings- oder Validierungsdaten liegen, zwar eingefärbt wurden, aber es an korrekter Farbgebung mangelt. Kleidung, Gesichter und teils Gegenstände werden im Ansatz gefärbt, jedoch wesentlich weniger. Objekte und Flächen ohne wirklich zugewiesene Farbe werden schlichtweg gelb.

7 Fazit

Abschließend lässt sich feststellen, dass die KI die Erwartungen an die Kolorierung von Comics nur bedingt erfüllen konnte. Es zeigt sich, dass der teils stark stilisierte und von der Realität abweichende Zeichenstil von Comics – insbesondere im Vergleich zu realistischen Bilddaten wie denen aus ImageNet – dazu führt, dass die KI Schwierigkeiten bei der Objekterkennung hat. Dies wirkt sich negativ auf die Qualität der Kolorierung aus.

Auffällig war, dass ältere Comics tendenziell besser koloriert wurden, jedoch häufig einen Gelbstich aufwiesen. Dieser ist aber auch in den Originaldaten vorhanden. Bei neueren Comics, etwa aus dem Superheldengenre, zeigte die KI hingegen Schwierigkeiten, spezifische Merkmale der Figuren zu erkennen und entsprechend differenziert zu kolorieren.

Ein weiterer Aspekt ist die gestiegene Komplexität des Zeichenstils in neueren Comics. Die Vielzahl an Objekten stellt eine Herausforderung für die KI dar, da diese häufig mehrere einzelne Karikaturen als ein einheitliches Bild mit ähnlichem Stil interpretiert, obwohl sie inhaltlich nicht miteinander verbunden sind.

Der im Kapitel 6 gewählte Ansatz zeigt grundlegend positive Eigenschaften und könnte mit mehr Interationen sowie einer gewissen weiteren Optimierung und genaueren Spezifizierung auf einen bestimmten Zeichenstil der Comics größere Erfolge erzielen. Dies sollte über ein genaueres auswählen des Datensatzes gut umsetzbar sein.

List of Figures

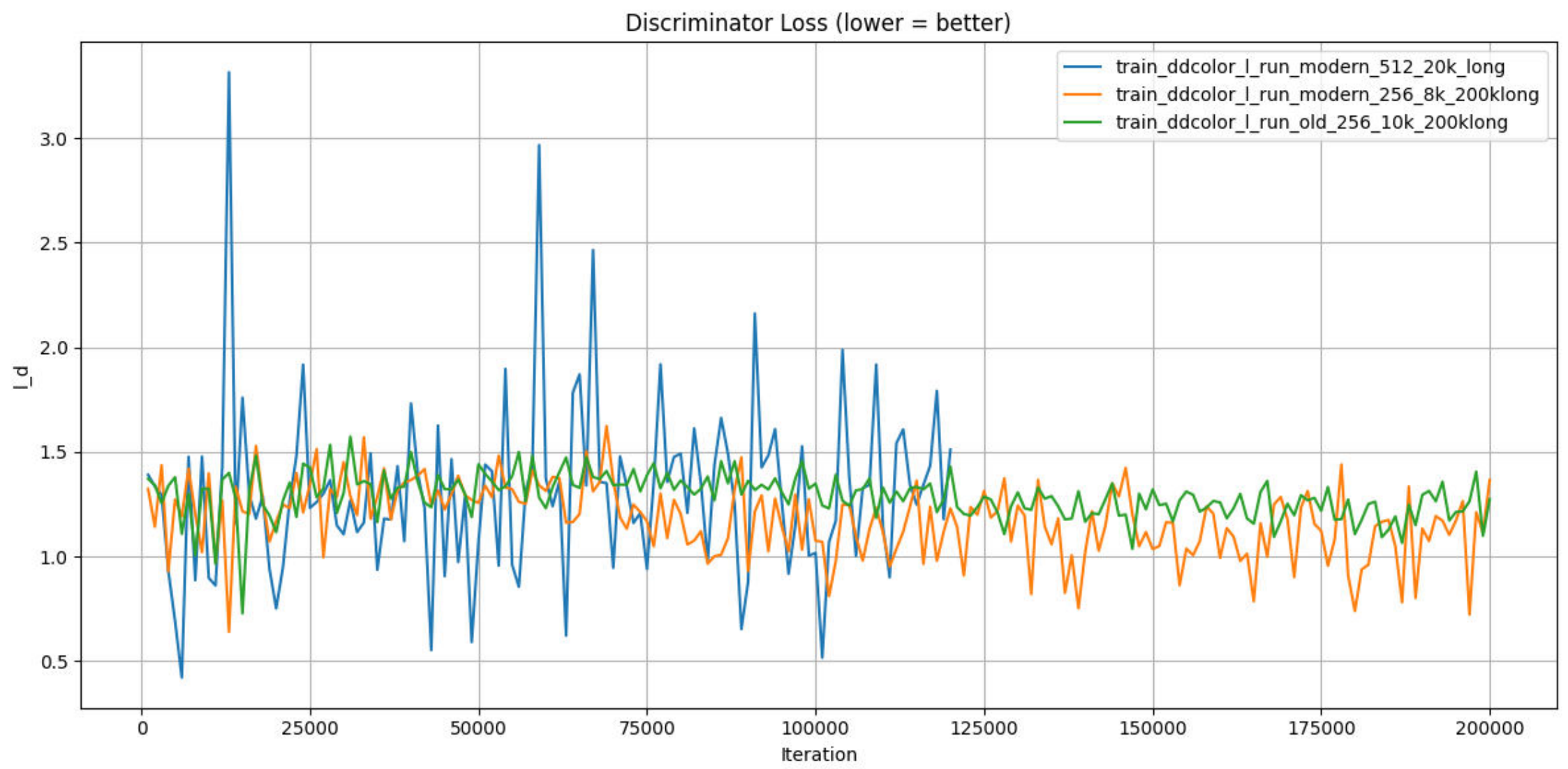
References

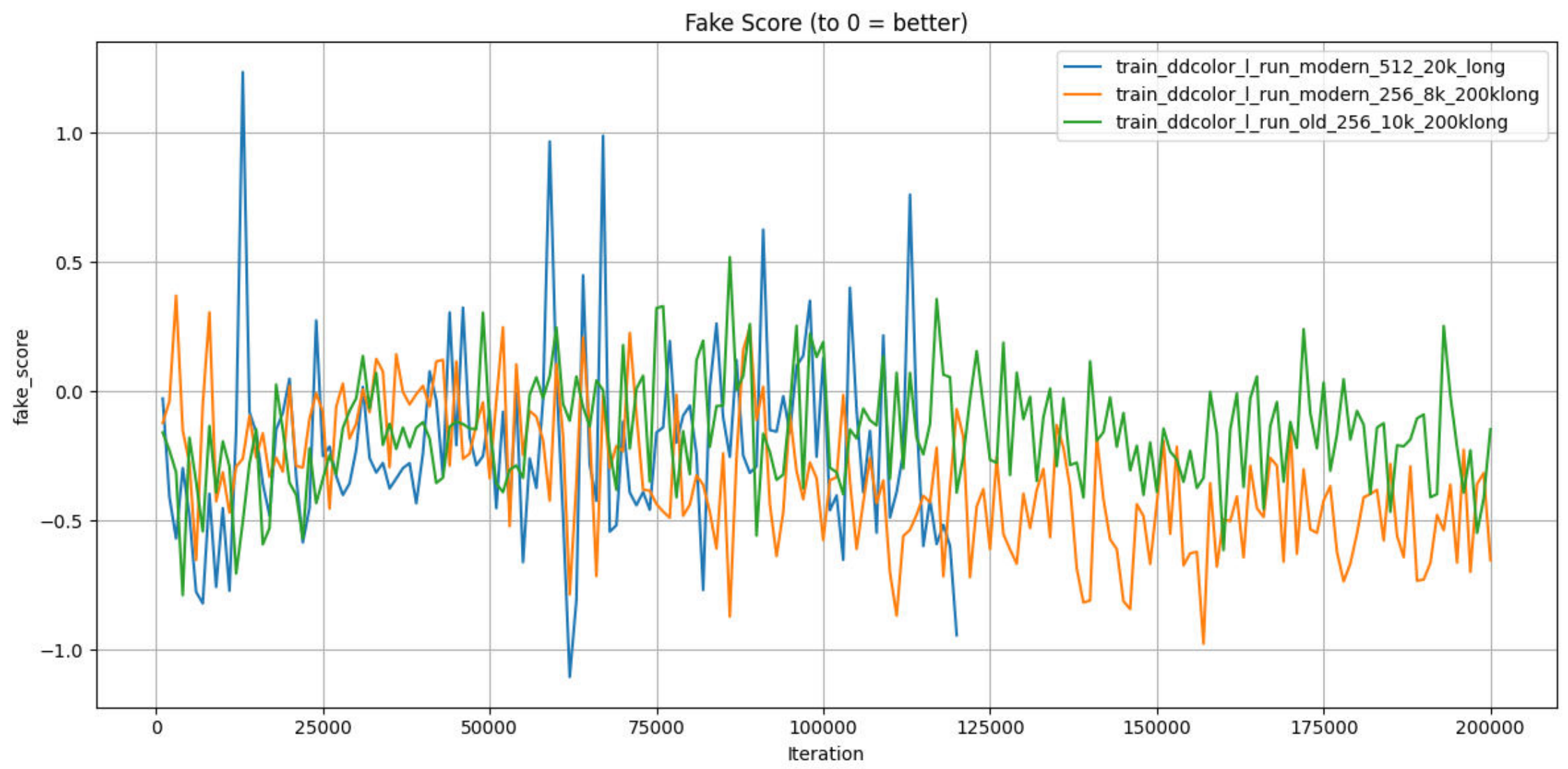
[1] X. Kang, T. Yang, W. Ouyang, P. Ren, L. Li, and X. Xie, "Ddcolor: Towards photo-realistic image colorization via dual decoders," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 328–338.

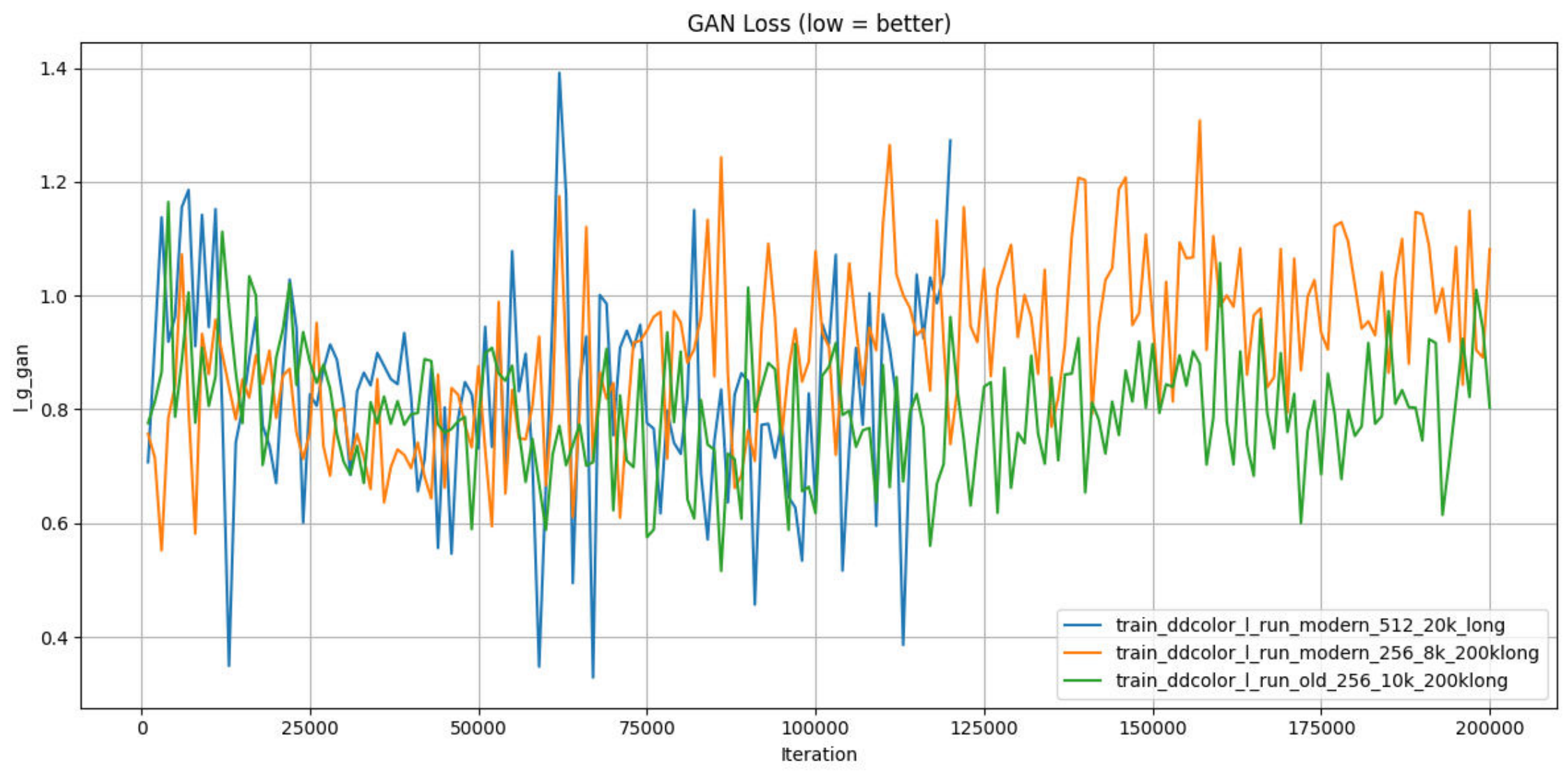
[2] VITA-MLLM, "Comic-9k." [Online]. Available: https://huggingface.co/datasets/VITA-MLLM/Comic-9K

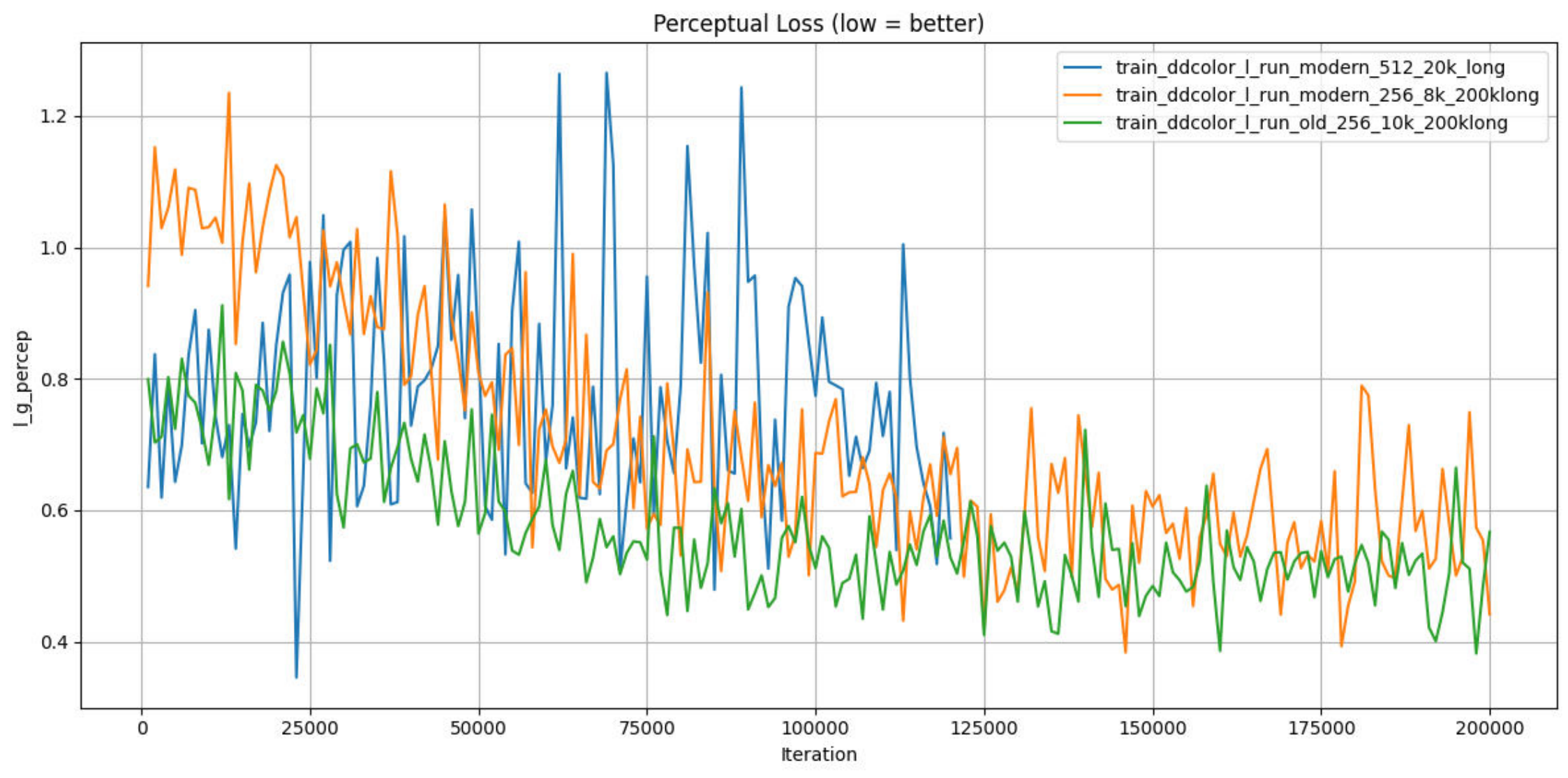
[3] M. Iyyer, V. Manjunatha, A. Guha, Y. Vyas, J. Boyd-Graber, H. Daumé III, and L. Davis, "The amazing mysteries of the gutter: Drawing inferences between panels in comic book narratives," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.


Color Loss (low = better)
l_g_color
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0
25000
50000
75000
100000
125000
150000
175000
200000
Iteration
train_ddcolor_l_run_modern_512_20k_long
train_ddcolor_l_run_modern_256_8k_200klong
train_ddcolor_l_run_old_256_10k_200klong

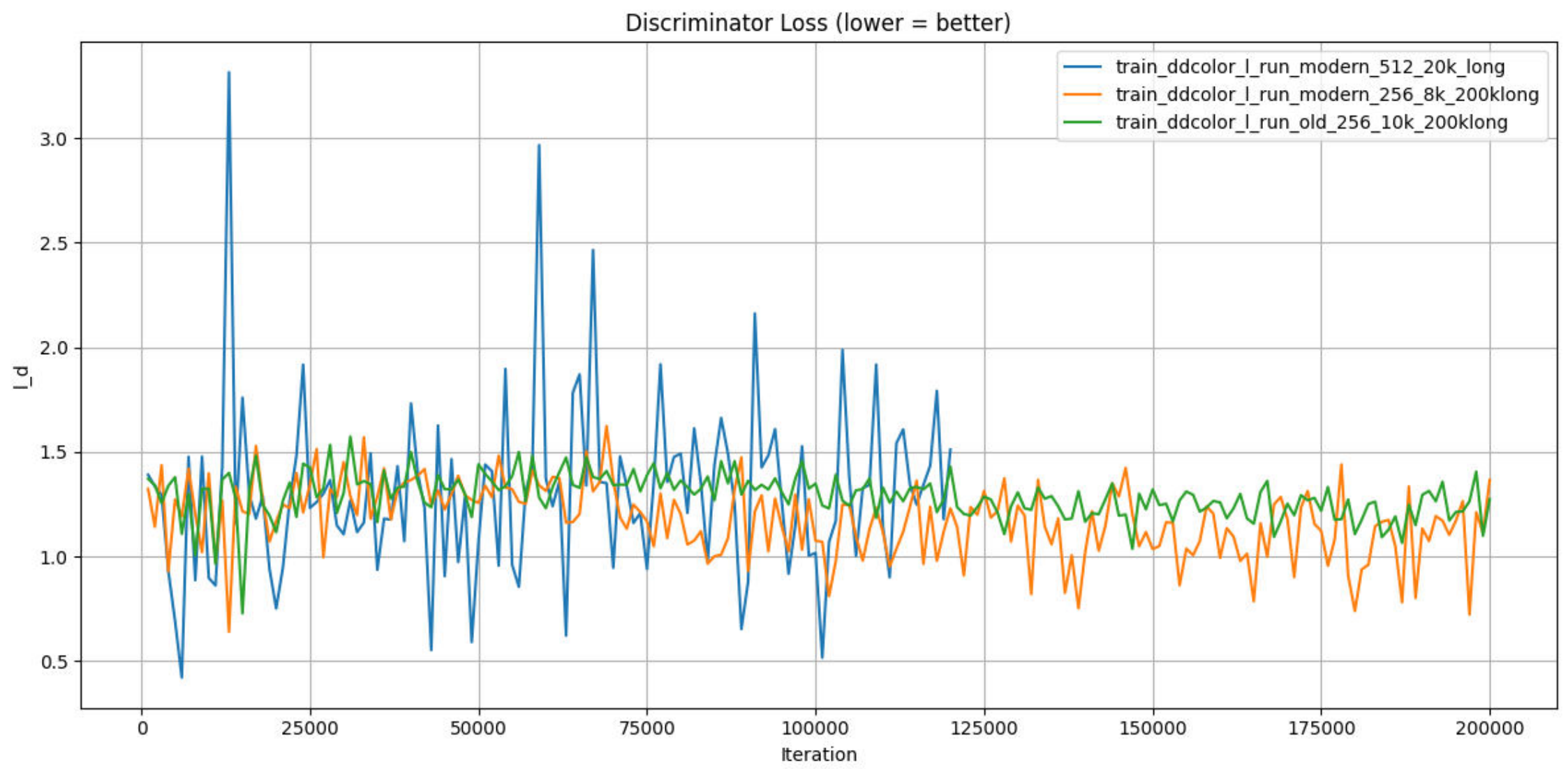

Discriminator Loss (lower = better)
train_ddcolor_l_run_modern_512_20k_long
train_ddcolor_l_run_modern_256_8k_200klong
train_ddcolor_l_run_old_256_10k_200klong
l_d
3.0
2.5
2.0
1.5
1.0
0.5
0
25000
50000
75000
100000
125000
150000
175000
200000
Iteration


Fake Score (to 0 = better)
train_ddcolor_l_run_modern_512_20k_long
train_ddcolor_l_run_modern_256_8k_200klong
train_ddcolor_l_run_old_256_10k_200klong
fake_score
1.0
0.5
0.0
−0.5
−1.0
0
25000
50000
75000
100000
125000
150000
175000
200000
Iteration


GAN Loss (low = better)
l_g_gan
Iteration
1.4
1.2
1.0
0.8
0.6
0.4
0
25000
50000
75000
100000
125000
150000
175000
200000
train_ddcolor_l_run_modern_512_20k_long
train_ddcolor_l_run_modern_256_8k_200klong
train_ddcolor_l_run_old_256_10k_200klong


Perceptual Loss (low = better)
train_ddcolor_l_run_modern_512_20k_long
train_ddcolor_l_run_modern_256_8k_200klong
train_ddcolor_l_run_old_256_10k_200klong
l_g_percep
1.2
1.0
0.8
0.6
0.4
0
25000
50000
75000
100000
125000
150000
175000
200000
Iteration


Pixel Loss (low = better)
train_ddcolor_l_run_modern_512_20k_long
train_ddcolor_l_run_modern_256_8k_200klong
train_ddcolor_l_run_old_256_10k_200klong
l_g_pix
2.5
2.0
1.5
1.0
0.5
0
25000
50000
75000
100000
125000
150000
175000
200000
Iteration


Discriminator Loss (lower = better)
train_ddcolor_l_run_modern_512_20k_long
train_ddcolor_l_run_modern_256_8k_200klong
train_ddcolor_l_run_old_256_10k_200klong
l_d
3.0
2.5
2.0
1.5
1.0
0.5
0
25000
50000
75000
100000
125000
150000
175000
200000
Iteration