

# Making Pixel Art with Deep Unsupervised Learning

UNIMORE School in AI: Deep Learning, Vision and Language for Industry

---

## Introduzione

La Pixel art è una forma di arte digitale in cui le immagini vengono costruite a livello di pixel attraverso programmi di editing grafico. Ciò che identifica la pixel art è il suo stile visivo, dove i pixel sono da intendere come i “blocchi” discreti che costruiscono l’immagine.

Nell’immaginario comune viene ampiamente associata alla grafica a bassa risoluzione presente nei computer e nei videogame ad 8-bit e 16-bit. Al giorno d’oggi la necessità di utilizzare pixel art all’interno dei videogiochi è sparita grazie all’enorme miglioramento delle tecniche di computer graphics, ma nonostante ciò rimane ampiamente praticata da una community di pixel artist e imitata ancora nostalgicamente in alcuni moderni videogames.



Nonostante la creazione di pixel art a partire da una immagine di input possa sembrare un task banale da eseguire algoritmamente, i risultati sono quasi sempre artisticamente deludenti.

Lo scopo di questa relazione è quello di automatizzare la creazione di pixel art a partire da immagini esistenti in modo più artistico con l’ausilio di tecniche di deep learning inquadrata nell’ambito dello style-transfer, seguite da un post-processing eseguito con algoritmi di image processing in python.

---

# Descrizione del problema

## Caratteristiche della pixel art

Nel momento in cui si voglia trasformare una foto in una pixel art bisogna innanzitutto definire quali siano le caratteristiche che definiscono una pixel art.

Queste ultime possono essere sintetizzate in:

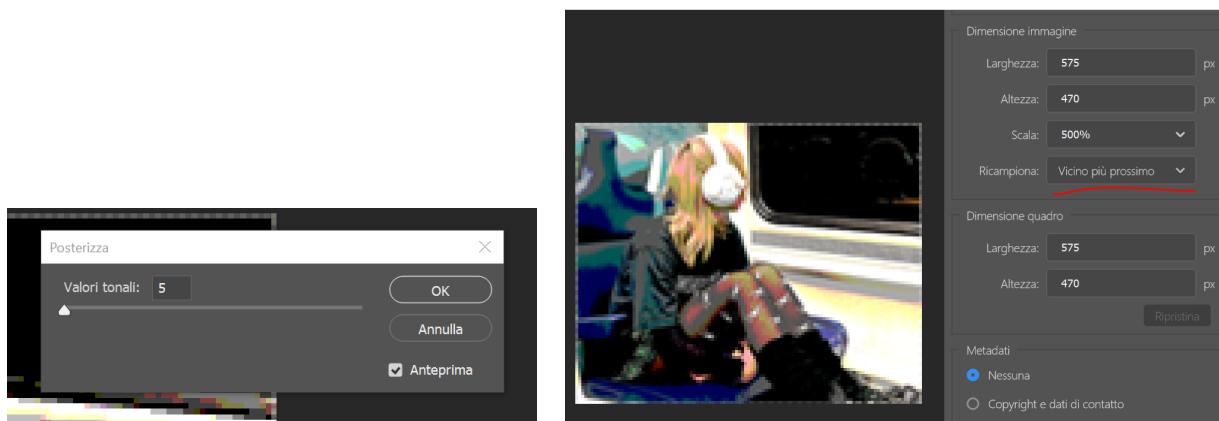
- **Una paletta di colori discreta e molto minimal**

La scelta dei colori è fondamentale. Devono essere il minor numero possibile ma che diano una rappresentazione fedele del contenuto. Inoltre devono essere distanti tra loro e in media più vividi.

- **I contorni devono essere nitidi e i riempimenti di colore continui**
- **L'immagine deve presentare una struttura a griglia**, che può essere più fine o più ampia, in cui ogni slot è quadrato ed è appunto il “pixel”.

## Approcci con interpolazione e posterizzazione

Un primo tentativo di creare una pixel art può essere quello di diminuire il numero di colori di una foto con posterizzazione, rimpicciolirla e ridimensionarla con interpolazione nearest neighbor.



---

Il risultato, utilizzando Photoshop, è questo:



*Originale*



*Processata*

Come si può notare, i contorni non sono ben definiti e si presentano sfocati. I colori non sono uniformi e il criterio con cui vengono scelti fa sì che vi siano colori molto vicini tra loro che creano degli effetti di sfumatura.

Per far capire meglio la differenza inserisco un altro confronto, stavolta con la stessa immagine e la pixel art disegnata da me.



*Originale*



*Disegnata a mano*

---

## Approcci con Deep Learning

Sul web sono presenti vari articoli in cui si propongono soluzioni utilizzando tecniche di style-transfer unsupervised con reti neurali. Il training è unsupervised perché attualmente non esistono dataset che contengano coppie di immagini paired in questo contesto.

Ad esempio in [3] viene proposta un'architettura specifica per effettuare la trasformazione di un'immagine in una pixel art comprendente tre sottoreti GridNet, PixelNet e DepixelNet. Il dataset utilizzato consiste in un dominio A composto da immagini di cartoni animati su sfondo bianco e in un dominio B composto da immagini di pixel art su sfondo bianco. Il risultato è molto soddisfacente anche se in input sono inserite foto reali.

Un'altro dei famosi lavori sul neural style-transfer è [5] (Perceptual Losses for Real Time Style Transfer and Super Resolution). Questa architettura però presuppone che vengano utilizzate due foto, una da trasformare e una contenente lo stile che vi si vuole applicare, e la rete traina in tempo reale applicando lo stile scelto all'immagine. Il problema di questa tecnica è che c'è bisogno ogni volta di una singola immagine che rappresenti lo stile, mentre la mia idea era quella di estrarre feature da un intero dominio di immagini e utilizzare queste per la stilizzazione.

Una rete più generica e che è parsa abbastanza idonea ai miei scopi è la CycleGAN [1]. Si colloca nel contesto delle **generative adversarial network** e risolve il problema della image-to-image translation in modo unsupervised. Essa impara a tradurre un'immagine di un dominio X in un'immagine di un dominio Y in assenza di coppie di esempi. Può essere utilizzata in problemi di style transfer, object transfiguration, photo enhancement ed altri. La rete presenta due generatori, uno per il dominio A e uno per il dominio B, e allo stesso modo due discriminatori. Vi è una **Adversarial Loss**, che ha il compito di discriminare quanto un'immagine trasformata sia dissimile dal dominio target, e una **Cycle consistency loss** che misura quanto la trasformazione inversa ricostruisca fedelmente l'immagine originale.

Vi è inoltre un articolo [2] in cui è stato già effettuato il training per ottenere pixel art da immagini utilizzando la CycleGAN, seguendo la strada di utilizzare come dataset unpaired una classe con immagini di cartoni animati e una classe con immagini di pixel art entrambe

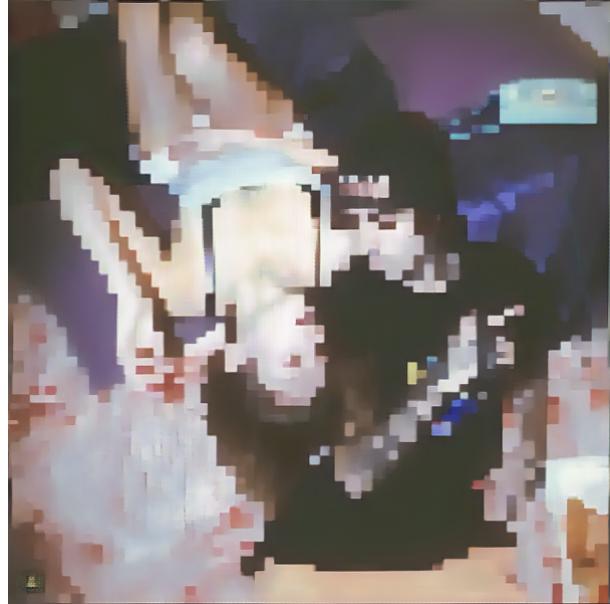
---

su sfondo bianco. E' anche stato reso disponibile il modello pre-trainato, quindi sono partita da lì e ho provato a trovare dei modi per effettuare un fine-tuning e un post processing dei risultati.

**Baseline:**



*Originale*



*Output CycleGAN*

Il risultato è visivamente carino, ma manca una discretizzazione netta dei colori e alcune regioni hanno delle sfumature troppo smooth.

**Approccio mio**

Per cercare di migliorare il risultato appena citato, ho scelto di effettuare un fine-tuning su CycleGAN utilizzando il modello già trainato reso disponibile sul sito [2]. Ho scelto di testare dei training con dataset diversi, e anche con una diversa configurazione di rete. In particolare, ho costruito un dataset di pixel art personalizzato poiché sul web non ne ho trovati molti.

Mi sono poi dedicata alla creazione di qualche algoritmo di post processing per regolarizzare l'output della rete.

---

## Dataset

Per la creazione di un dataset personalizzato ho scelto di utilizzare come dominio A delle foto estratte dal dataset fornito in <https://cocodataset.org/> e come dominio B delle immagini di pixel art raccolte da Instagram sotto l'hashtag #pixelart.

Il dominio A presenta circa 1000 foto di oggetti comuni, pre processate con lo script *coco\_preprocessing.py* per renderle quadrate e di dimensione 256 x 256 px.

Il dominio B consiste in circa 1000 immagini che inizialmente si presentavano così:



Ho quindi costruito uno script che tagliasse il contenuto superfluo dell'immagine e lasciasse solamente la parte di interesse, opportunamente ridimensionata. Questo procedimento è spiegato nel notebook *Dataset Extraction/Pre-Processing.ipynb*.

Il dataset completo è scaricabile al link

[https://drive.google.com/drive/folders/147jYnd\\_98wXHyLyxMAMS434hZJ4bts5C?usp=sharing](https://drive.google.com/drive/folders/147jYnd_98wXHyLyxMAMS434hZJ4bts5C?usp=sharing)

---

# Training

Il training dei modelli è stato effettuato attraverso il codice fornito in:

<https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix>

utilizzando lo script *train.py* con relativi parametri (come spiegato nel readme).

Gli output del processo e i log sono consultabili nella cartella *AISchools/Training*.

## Training 1: Resnet

Il primo training è stato effettuato utilizzando il mio dataset personalizzato e partendo dal modello pre-trained reso disponibile in [2] (*Modelli/baseline\_G\_A.pth*). La rete è la stessa utilizzata nell'articolo (*Resnet-9blocks*, che è di default)

Dato che sul web vengono forniti solamente i pth del generatore A, ho inserito una piccola modifica nell'inizializzazione dello state dict. In

*/AISchools/pytorch-CycleGAN-and-pix2pix/models/base\_model.py* ho creato la funzione

```
#####
# Carica uno state dict G_A da un path personalizzato

def load_network_init(self, path_name):

    for name in self.model_names:

        if(name == 'G_A'):

            if(isinstance(name, str)):

                load_filename = path_name

                load_path = os.path.join('./stateDict', load_filename)

                net = getattr(self, 'net' + name)

                if isinstance(net, torch.nn.DataParallel):

                    net = net.module

                print('Caricamento di G_A da: %s' % load_path)

                # if you are using PyTorch newer than 0.4 (e.g., built from

                # GitHub source), you can remove str() on self.device
```

```
state_dict = torch.load(load_path, map_location=str(self.device))

if hasattr(state_dict, '_metadata'):
    del state_dict._metadata

# patch InstanceNorm checkpoints prior to 0.4

for key in list(state_dict.keys()): # need to copy keys here because we mutate in
loop

    self.__patch_instance_norm_state_dict(state_dict, net, key.split('.'))

net.load_state_dict(state_dict)
```

e l'ho utilizzata in *train.py*

```
#####
model.load_network_init(opt.name_state_dict_init)
#####
```

Inserendo una opportuna opzione modificabile da linea di comando.

```
#### Mod

parser.add_argument('--name_state_dict_init', type = str, default = 'init.pth', help='nome del file di
inizializzazione presente nella cartella stateDict')

###
```

I parametri di training sono:

Dataset: coco2pixelart (TrainA: 1500 pic - TrainB: 1070 pic)

Model: CycleGAN

Generator: Resnet-9blocks

Epochs: 150

Norm: instance normalization

Batch size 16

Dropout : yes

Il training è stato effettuato su GPU Nvidia 3090 e ha impiegato circa 4 ore.

---

## Training 2: Unet

Siccome il risultato del primo training non mi è parso molto soddisfacente, ho deciso di effettuare una prova con un'altra configurazione di rete. Nell'articolo [4] viene mostrato che effettuare il training con una rete Unet come generatore dia dei buoni risultati. Nel repository della CycleGAN scelto per questo progetto è presente proprio un'opzione per utilizzare la Unet al posto della Resnet.

Nel file *base\_options.py* in *pytorch-CycleGAN-and-pix2pix/options* è presente l'opzione di training:

```
parser.add_argument('--netG', type=str, default='resnet_9blocks',
help='specify generator architecture [resnet_9blocks | resnet_6blocks | unet_256 | unet_128]')
```

Ho scelto come parametro *unet\_256*.

Ho inoltre cercato di replicare la configurazione presente nel paper [4]. La funzione di adversarial loss e identity loss sono presenti in codice. Utilizzo come adversarial la LSGAN (che è di default):

```
parser.add_argument('--gan_mode', type=str, default='lsgan', help='the type of GAN objective. [vanilla| lsgan | wgangp]. vanilla GAN loss is the cross-entropy objective used in the original GAN paper.')
```

L'unica differenza sarà solamente la mancanza della *topology-aware loss* che viene aggiunta insieme alla cycle-consistency loss di base.

Il dataset utilizzato in questa prova è lo stesso che viene utilizzato nel paper, ovvero composto da un gruppo di immagini di cartoni animati su sfondo bianco e un gruppo di pixel art su sfondo bianco. [8]

I parametri di training sono:

Dataset: Cartoon2pixel (TrainA: 1000 pic - TrainB: 1000 pic)

Model: CycleGAN

Generator: Unet-256

---

Epochs: 150

Norm: instance normalization

Batch size 16

Dropout : yes

Il training è stato effettuato su GPU Nvidia 3090 e ha impiegato circa 2 ore.

## Algoritmi di Post Processing

Al risultato di output della rete neurale applico degli algoritmi che discretizzano il numero di colori (palette) e la grandezza dei pixel per ottenere un risultato visivamente più standard.

### Palette Extraction

Spiegazione dettagliata in [Palette\\_extraction.ipynb](#).

Viene utilizzato l'algoritmo *k-means* messo a disposizione da OpenCV per effettuare una clusterizzazione dei colori e diminuirne così notevolmente la quantità.

La scelta del numero di cluster da inserire nella funzione viene effettuata portando l'immagine nel Color Space HSV e conteggiando attraverso l'istogramma delle tonalità (Hue) il numero di colori indipendenti presenti.

Viene poi creata una funzione (*palette\_extraction*) che serve ad esportare i colori ottenuti con i passaggi precedenti. Questa funzione viene usata nel modulo *PaletteGen.py*.

### Downsampling

Spiegazione dettagliata in [Downsampling.ipynb](#).

Implementazione di un algoritmo che estrae dall'immagine una palette di colori (tramite *PaletteGen.py*), discretizza l'immagine in slot quadrati (pixel) ed associa ad ognuno di essi il colore della palette che più vi si avvicina.

Il codice sviluppato qui viene utilizzato nel modulo *Downsampling.py*

---

## Video Processing

Con il codice sviluppato in precedenza ho realizzato anche un piccolo script che elaborasse dei video: *video\_processing.py*. Esso utilizza la rete del primo training tramite la funzione `resnet_process` presente in *prova\_gen\_resnet.py*, la funzione `palette_extractor` presente in *PaletteGen.py* e la funzione `pixxelate` presente in *Downsampling.py*.

Il rendering è un bel po' lento, ma un risultato di prova è il file *video\_out.avi*.

## Risultati

Il testing dei risultati complessivi può essere visualizzato nel notebook *Main - Making Pixel Art with Deep Unsupervised Learning.ipynb*.

Alternativamente, nel repository sono presenti due script (*prova\_gen\_unet.py* e *prova\_gen\_resnet.py*) che possono essere eseguiti in locale.

I modelli si trovano nella cartella *AISchools/Modelli*.

Il modello ottenuto dalla u-net non è presente poiché github non permette il caricamento di file troppo grandi. E' possibile scaricarlo al link

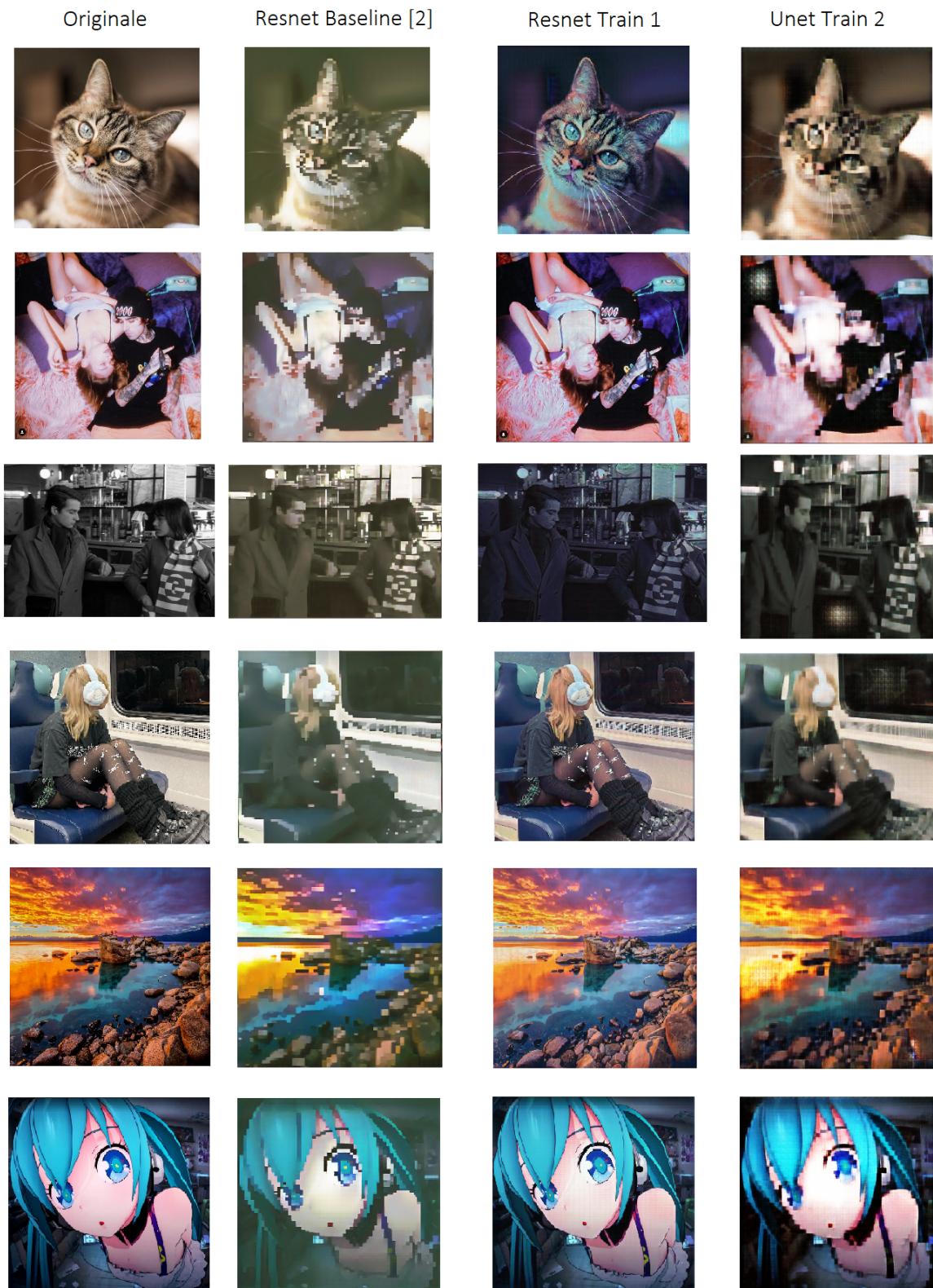
[https://drive.google.com/file/d/1UdTWPK\\_phTg-Ge3WsDjGkaLdRVHbjFrb/view?usp=sharing](https://drive.google.com/file/d/1UdTWPK_phTg-Ge3WsDjGkaLdRVHbjFrb/view?usp=sharing)

Nella repo utilizzata per la CycleGAN è presente uno script di test (*test.py*). Io ho deciso di non utilizzarlo ma di testare un sample di immagini manualmente, poiché non ho ritenuto necessario il confronto delle funzioni di loss ai fini della valutazione del risultato (anche perché risultavano molto oscillatorie). Essendo un task che dà in output un risultato "artistico" la valutazione delle performance andrebbe fatta con dei sondaggi su un campione umano.

Ho testato i risultati su varie immagini di input, e ho inserito sia la versione di output raw che la versione post-processata. I risultati sono consultabili nella cartella *AISchools/Risultati*.

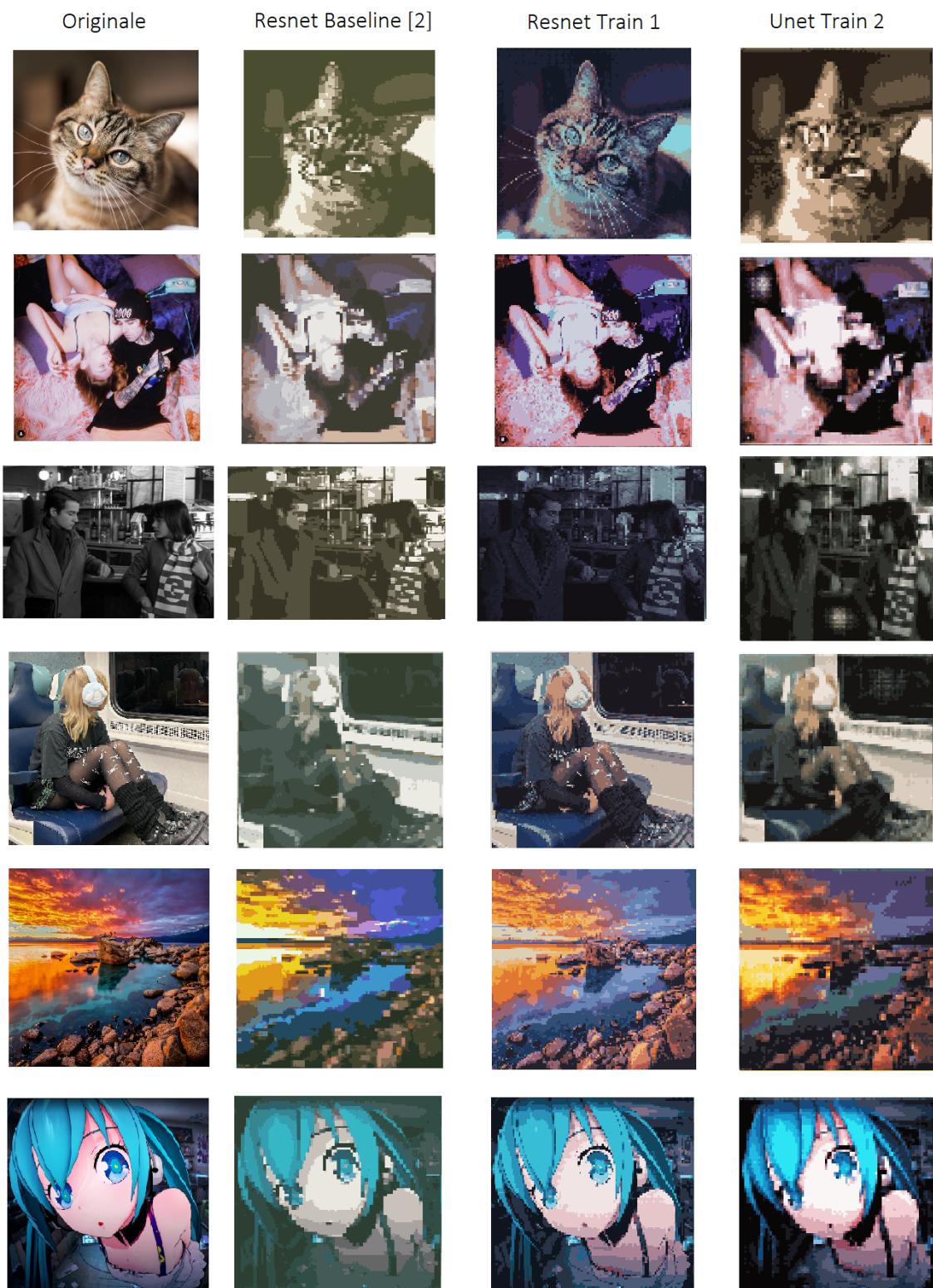
---

Risultati reti neurali:



---

Risultati reti neurali + post processing:



---

## Conclusioni

Per quanto riguarda il primo training effettuato, il risultato non è simile ad una pixel art. Il motivo sta probabilmente nel fatto che i due domini del mio dataset sono visivamente e semanticamente molto diversi, quindi la rete non è riuscita a cogliere un pattern preciso di trasformazione. Situazione diversa è se si utilizza il dataset cartoni animati -> pixel art con sfondo bianco, dato che la somiglianza tra i due domini è molto più marcata. Nonostante lo scopo principale non sia stato raggiunto nel primo training, almeno la rete ha imparato a rendere i colori un po' più stilosi.

Nel secondo training invece i risultati in termini di pixeling sono stati raggiunti, anche se visivamente il risultato è diverso da quello di partenza. Nel modello [2] da cui sono partite le immagini di output risultano in qualche modo desaturate, cosa che non accade utilizzando la u-net. Uno svantaggio di questo modello è però il fatto che richiede una dimensione di input prefissata (in questo caso 256 x 256), mentre con il generatore resnet non vi sono limiti di dimensione.

Applicando infine gli algoritmi di post-processing si ottengono risultati diversi per ogni approccio utilizzato.

## Codice

I notebook sono stati sviluppati su Google Colab. Link:

<https://drive.google.com/drive/folders/13h47yOhOigjhw3Q5CEcUPJ8uX-URthvB?usp=sharing>

- Color\_shifting.ipynb
- Palette\_extraction.ipynb
- Downsampling.ipynb
- Dataset Extraction/Pre-Processing.ipynb
- Main - Making Pixel Art with Deep Unsupervised Learning.ipynb

---

Codice python:

- PaletteGen.py: Estrae la palette di colori da un'immagine con k-means
- Downsampling.py: Rende l'immagine pixelata utilizzando una palette di colori
- dataset\_extraction\_pre\_processing.py: Utilizzato per pre-processare le immagini per il dataset di pixel art
- coco\_preprocessing.py: Utilizzato per pre-processare il dataset COCO
- prova\_gen\_unet.py: Processing di un'immagine con unet trainata in training 2.
- prova\_gen\_resnet.py: Processing di un'immagine con resnet trainata in training 1.
- video\_processing.py: Generazione di video con rete neurale e algoritmi.

---

## Bibliografia

[1] CycleGAN: Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, Zhu, Park, Isola, Efros, ICCV 2017. Code:  
<https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix>

[2]

<https://inikolaeva.medium.com/make-pixel-art-in-seconds-with-machine-learning-e1b1974ba572>

[3] Deep Unsupervised Pixelization, Chu, 2018

[4] A Pixel image generation algorithm based on CycleGAN, Kuang, Huang, Xu, 2021

[5] Perceptual Losses for Real Time Style Transfer and Super Resolution, Johnson, Alahi, Fei Fei

[7] Dataset: MS COCO: <https://cocodataset.org/>, Tagged Anime Illustrations:

<https://www.kaggle.com/datasets/mylesoneill/tagged-anime-illustrations?resource=download>

[8] Dataset: Cartoon to PixelArt

<https://drive.google.com/file/d/1qDXB5g0Cb0VwlSXwnfeiehPHuTgxWhdG/view>

[9] Dataset mio: coco2pixelart

[https://drive.google.com/drive/folders/147jYnd\\_98wXHyLyxMAMS434hZl4bts5C?usp=sharing](https://drive.google.com/drive/folders/147jYnd_98wXHyLyxMAMS434hZl4bts5C?usp=sharing)