Image colorization

Salvatore Pappalardo
Ingegneria informatica
281621

Sofia Caterini
Ingegneria del cinema e dei mezzi di comunicazione
270222

Abstract

Abbiamo elaborato due modelli capaci di colorare in modo sufficientemente credibile delle immagini in bianco e nero. Tale sistema permette quindi di generare, a partire da un'immagine di input, una di output che ne è la corrispettiva nello spazio di colore RGB.
I due modelli sono: una Pix2Pix GAN modificata e una U-Net con perceptual loss. Abbiamo anche provato ad allenare la pix2pix integrando la U-Net in un setup NoGAN ma con scarsi risultati.
I due modelli sono comunque in grado di ottenere dei risultati soddisfacenti.

1. Introduzione

La colorazione di immagini in bianco e nero può sembrare un task semplice nell'immaginario umano. Un uomo, infatti, conosce dall'esperienza personale i colori che vengono assunti dagli oggetti: deve solo ricordare che il cielo è azzurro, il prato è verde; e per molti oggetti la mente è libera di assegnare loro i colori più verosimili e quindi plausibili. Ma l'assegnazione dei colori fatta in modo automatico da una rete neurale è un task ben più difficile che richiede l'apprendimento di feature ad alto livello dalle immagini. Le applicazioni in cui tale task è importante sono la colorazione di foto e/o video storici e task di assistenza artistica.

1.1. Lavori correlati

Per raggiungere l'obiettivo prefissato il primo passo è stato esaminare i precedenti lavori svolti a riguardo. Zhang et al. [1] e Larsson et al. [2], hanno implementato un autoencoder per la colorazione delle immagini. Lavori come quelli di [3] utilizzano invece delle GAN. Abbiamo cercato di trarre ispirazione da questi per raggiungere un modello finale ottimizzato.

2. Dataset

Qualsiasi dataset di immagini poteva essere utilizzato per allenare una rete di colorizzazione. La scelta, infine, è ricaduta su Imagenet. Volevamo utilizzare un dataset che potesse essere il più generico possibile, senza dover scegliere dei determinati synset. Per soddisfare questa necessità abbiamo usato una versione modificata di un utilissimo script trovato su Github[1] in modo da scaricare delle immagini da synset casuali, ma con la possibilità di scegliere il seme del generatore casuale di numpy.
Il dataset è stato, in prima istanza, scaricato su Google Drive, ma il caricamento sul notebook risultava troppo lento. La seconda opzione proposta è stata di dividerlo in bucket in modo da consentire un allenamento più veloce. Questo metodo, però, non consentiva un allenamento uniforme sul dataset, infatti si facevano 10 epoche per ogni bucket e alla fine di ciascuna decade si cambiava bucket.
La soluzione finale è stata quella di utilizzare una parte del dataset scaricato (circa 4k immagini) ridimensionate tutte allo stesso modo (256x256) senza l'impiego di buckets. Questo metodo si è rivelato essere il miglior compromesso tra prestazioni e allenamento, consentendo dei risultati buoni. Abbiamo suddiviso il dataset usando 4000 immagini per il training set, 450 per validation set e 1500 per test set.

3. Metodo

In prima battuta abbiamo creato un autoencoder che cercasse di ricreare un'immagine delle stesse dimensioni di quella dell'input ma a 3 canali. I risultati ottenuti non sono stati soddisfacenti, le reti non riuscivano a ricreare né le forme né i colori delle immagini del dataset e in uscita riuscivamo ad ottenere solo immagini bianche con macchie nere. Abbiamo quindi deciso di migliorare il nostro autoencoder dotandolo di skip connections. Tali connessioni sono in grado di portare le informazioni ad alta

[1] [1] jspkay/ImageNet-Dataset-Downloader (versione modificata)

frequenza negli ultimi layer, aspetto fondamentale per il nostro task, in quanto ogni immagine ricreata deve riuscire a contenere tutte le feature apprese nei livelli iniziali ovvero bordi e forme. Inizialmente abbiamo optato per ricreare un autoencoder con skip connections da zero, ma non abbiamo ottenuto dei buoni risultati quindi ci siamo soffermati sull'implementazione di uno specifico autoencoder con skip connections, la U-Net [4]. I nostri esperimenti sono proseguiti su due piani paralleli. Ovvero abbiamo contemporaneamente cercato di implementare due tipi di modelli: la U-Net e una Pix2Pix (U-Net come generatore e PatchGAN come discriminatore) [3]. La U-Net è stata implementata seguendo e modificando il modello [4], la Pix2Pix ricreando la rete del paper [3]. Entrambe riuscivano inizialmente a dare dei risultati visivi più soddisfacenti rispetto alle precedenti, in quanto riuscivano a ricreare perfettamente le forme dell'immagine in ingresso, ma non riuscivano a colorarle. Avevamo quindi in uscita un'immagine identica a quella in ingresso per quanto riguarda il contesto, ma non con i colori che auspicavamo. Per risolvere tale problema abbiamo ricercato una funzione di costo che migliorasse l'apprendimento dei colori e delle forme contemporaneamente. Il primo passo è stato scegliere una migliore funzione di costo. Inizialmente abbiamo usato l'errore medio assoluto (mae) per poi ricadare su una combinazione di mae e percpetual loss [5] implementata con una VGG16.
I nostri modelli finali sono risultati essere:

- Una U-Net con perceptual loss
- Una Pix2Pix con Perceptual Loss

L'idea iniziale era quella di unire le due reti utilizzando un allenamento NoGAN come descritto in [6].
Quindi pre-allenare sia il generatore (la nostra U-Net), che il discriminatore (ovvero la PatchGAN della Pix2Pix) e successivamente di inserirli nella GAN. Abbiamo inizialmente proceduto allenando entrambe le reti nella speranza di unirle successivamente. Tale sperimentazione, purtroppo, non è andata a buon fine. Ne descriviamo le motivazioni nella sezione Allenamento NoGAN.

3.1. U-Net

3.1.1 Funzioni di costo

Per scegliere la funzione di costo è stato innanzi tutto necessario capire che tipo di task fosse il nostro. La colorizzazione può infatti essere vista sia come un problema di classificazione probabilistica che come una regressione. Nel primo caso, ad ogni canale di ogni pixel bisognerebbe associare la probabilità che quella componente sia presente in quel pixel (i.e. la probabilità che un certo pixel sia rosso può essere espressa come il valore stesso di rosso in quel pixel). L'implementazione però era molto ostica, tanto da non provare neanche ad implementarla.
Allora, tra le tante funzioni, si è scelto di provare l'errore quadratico medio e l'errore assoluto medio. Dai vari esperimenti è emerso che l'errore assoluto medio aveva delle prestazioni migliori sul nostro dataset. Tuttavia, anche in questo caso, i risultati non erano dei migliori.

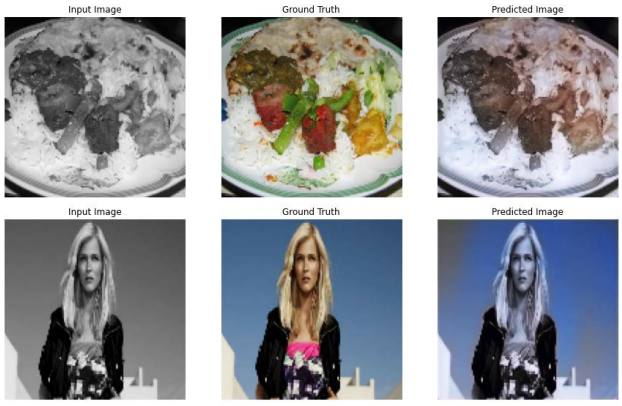

Figura 3.A - Unet allenata con l'errore assoluto medio dopo 100 epoche

Le immagini, seppure colorate con contrasti migliori e una più ampia gamma, non rispettavano le forme degli oggetti. Allora l'idea è stata quella di implementare la perceptual loss, utilizzando una VGG16 allenata con imagenet.

3.1.2 Iperparametri e allenamento

Rispetto alla U-Net standard, la nostra rete ha alcune differenze. Tra tutte l'uso di attivazioni LeakyReLU, in modo da sopperire al problema della "dying ReLU". Inoltre, sono stati inseriti dei livelli di batch normalization, subito prima di un downsamplig e upsampling, per velocizzare il training. Infine, sono stati aggiunti dei livelli di droupout nei livelli più vicini all'input, per evitare overfitting. L'allenamento di questa U-Net è standard. Dopo 56 epoche la rete ha smesso di migliorare.

3.2. Pix2Pix

3.2.1 Funzioni di costo

La seconda rete è una Pix2Pix ovvero una GAN condizionata, composta quindi da generatore e discriminatore. Le funzioni di costo utilizzate sono la binary crossentropy, per quanto riguarda il discriminatore e, per il generatore, che come sappiamo ha il compito più difficile, la binary crossentropy per ingannare il discriminatore e la perceptual loss per generare immagini più simili a quelle del target.
I passi che ci hanno portato alla scelta di queste funzioni di costo sono analizzati in modo dettagliato nella sezione Esperimenti.

3.2.2 Iperparametri e allenamento

La costruzione della rete è quella standard della Pix2Pix. Sia generatore che discriminatore sono inizializzati tramite rumore gaussiano casuale con una media di 0.0 e una deviazione standard di 0.02, come riportato in [3].

I livelli più importanti sono quelli di convoluzione, la batch normalization e le attivazioni intermedie che sono LeakyReLU sempre per sopperire al problema che darebbero altrimenti le Relu.
L'allenamento di una GAN è un'operazione complessa in quanto, affinché la rete funzioni, c'è bisogno che si instauri un certo equilibrio tra la "bravura" del generatore a generare immagini false che sembrino vere e la "bravura" del discriminatore nel riconoscere le immagini false da quelle vere. Il compito del generatore è quello più difficile in quanto ha un task di creazione di immagini, rispetto a quello del discriminatore che deve solo compiere una classificazione. Per questo, come appreso a lezione, quando definiamo inizialmente il modello della nostra GAN assumiamo che il discriminatore sia inizialmente non allenabile.
Per addestrare il discriminatore, abbiamo bisogno di molte immagini reali e false. Abbiamo definito due funzioni, *generate real_samples* e *generate_fake_samples*. generate_real_samples prende dal dataset degli esempi casuali reali e il rispettivo target e vi assegna l'etichetta 1 (vera). D'altra parte, generate_fake_samples utilizza la nostra rete di generatori per creare immagini false in base al suo input. Etichettiamo il risultato atteso 0 per mostrare al discriminatore che queste immagini sono false. Adesso il processo di training delle due componenti della GAN prosegue alternato.
Abbiamo inserito uno pseudocodice del ciclo di training della nostra GAN:

```
1. For n_epochs:
2.     For n_steps do:
3.       Select a batch of real samples
4.       Generate a batch of fake samples
5.       Update the weights of discriminator
         for real samples
6.       Update the weights of discriminator
         for generated samples
7.       Update the weights of generator
         fixing the discriminator
```

La rete è stata così allenata per 50 epoche mappando in modo diverso i contributi delle loss tra discriminatore e generatore con un rapporto 1/100 (perché ricordiamo che è molto più facile il compito del discriminatore). Il training è stato effettuato con batch di grandezza 16 e con la versione Adam della SGD (che è dimostrato avere miglior efficienza in ogni tipo di architettura GAN) con learning rate 2e-4. La scelta di tale grandezza della batch è data dal fatto che è stato dimostrato empiricamente che una GAN si allena in modo migliore con una dimensione della batch minore di 64 e in particolare 8 o 16.

3.3. Metriche di valutazione

Entrambe le reti sono state valutate con gli stessi criteri.
Per scegliere le metriche di valutazione abbiamo analizzato alcuni lavori correlati per rendere i nostri risultati facilmente confrontabili. Risulta evidente che per il nostro task, ovvero la colorizzazione delle immagini, la metrica effettivamente più rilevante sarebbe la valutazione visiva umana, ma tale metrica non si presta ad una valutazione oggettiva ma qualitativa e per l'analisi dei risultati è necessario attenersi a valutazioni quantitative. Per questo, come [2], abbiamo utilizzato la RMSE, Root Mean Square Error e la PSNR, Peak Signal to Noise Ratio. Abbiamo inoltre implementato una metrica particolare che descrivesse l'accuratezza pixel per pixel nella generazione di un'immagine a colori rispetto a quella in input in bianco e nero, col nome Custom Accuracy. L'implementazione di tale metrica verrà analizzata in dettaglio nella sezione Custom Accuracy degli Esperimenti.

4. Esperimenti

4.1. Implementazione della funzione di costo

Per allenare al meglio la U-Net volevamo che la rete fosse in grado di assegnare diversi colori alle diverse caratteristiche dell'immagine. A tale scopo abbiamo pensato di implementare una funzione di costo in grado di scegliere i colori sulla base delle caratteristiche dell'immagine, ovvero abbiamo implementato una perceptual loss combinata all'errore assoluto medio.
Abbiamo scoperto che keras tratta ogni pixel dell'immagine come un signolo output, producendo cioè una loss multidimensionale. Inizialmente, non essendo a conoscenza del dettaglio, abbiamo implementato una loss completamente scalare, ma la rete "ha imbrogliato".

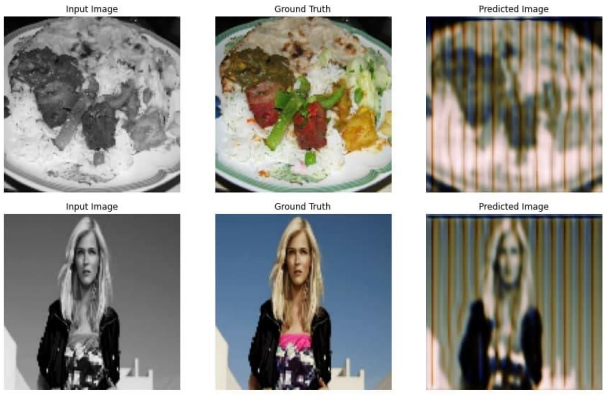

Figura 4.A - Risultati con una loss scalare

Utilizzando una loss scalare, infatti, sono andate perse le informazioni spaziali. La conseguenza è stata che la rete ha creato delle linee verticali (forme, per minimizzare la

perceptual loss) colorate (colori, per minimizzare l'errore assoluto medio).
Per risolvere questo problema è stato sufficiente utilizzare una loss che fosse un tensore (come la mae) e sommare a questo il termine della perceptual loss.

4.2. Perceptual loss

Con l'implementazione descritta i risultati ottenuti sono migliorati e pare che la rete sia in grado di fare un'associazione tra forma e colore. Tra tutti gli esempi, il più lampante è il seguente.

Figura 4.B - la perceptual loss associa forme e colori

Si vede nell'esempio che l'allenamento con la perceptual loss potrebbe aver associato al cane un punteggio simile a quello di una pianta, colorandolo di verde. Questa interpretazione è suggerita dalla forma del cane che potrebbero ricordare dei fili di erba.

4.3. Custom Accuracy

Al fine di valutare oggettivamente il modello abbiamo anche implementato una custom accuracy. Questa metrica calcola la percentuale di pixel predetti correttamente dalla rete rispetto a tutti i pixel predetti. Nell'implementazione abbiamo definito anche dei "vincoli di uguaglianza". È infatti ragionevole dire che due pixel con valori RGB molto simili siano "uguali" nel contesto di un'intera immagine. Per inferire quindi l'uguaglianza di due pixel si valutano le norme della differenza tra il predetto e il ground truth.

$$\left\|P_{truth} - P_{pred}\right\|_{\infty} < \varepsilon_{\infty} \cap \left\|P_{truth} - P_{pred}\right\|_{2} < \varepsilon_{2}$$
Equazione 1 - Viconli d'uguaglianza

Nel momento in cui la norma infinito della differenza è minore di ε_{∞} e la norma 2 della differenza è minore di ε_2 i due pixel sono considerati uguali. Nel nostro modello abbiamo assunto

$$\varepsilon_{\infty} = 0.14, \qquad \varepsilon_2 = 0.17$$
Equazione 2 - Valori scelti

Tali valori sono stati scelti sulla base di alcune prove sperimentali.

4.4. Allenamento NoGAN

Il motivo per cui abbiamo deciso di concentrarci su due modelli separati era utilizzarli entrambi per sfruttare questo metodo di allenamento. L'allenamento NoGAN [6] è un particolare tipo di training che permette di far ottenere alla rete una stabilità migliore rispetto ad una GAN allenata da zero. L'idea è quella di allenare discriminatore e generatore con dei metodi tradizionali per poi inserirli in un'architettura di tipo GAN (nel nostro caso la pix2pix).
Purtoppo l'applicazione di questo metodo non ha avuto i benefici sperati. Il discriminatore, infatti, doveva essere allenato sui prodotti del generatore (la U-Net).

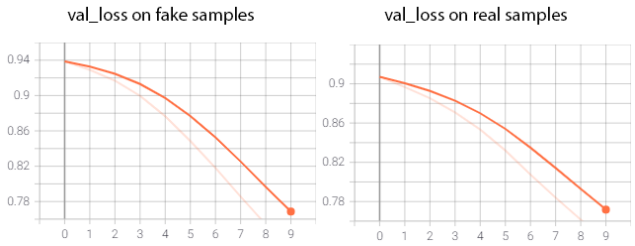

Figura 4.C - Loss del discriminatore

L'andamento in figura 4.C è stato ottenuto utilizzando il discriminatore della pix2pix con un ottimizzatore Adam e learning rate di 1e-6. Inoltre sono stati aggiunti dei regolarizzatori L2 ai livelli di convoluzione, per consentire un migliore allenamento sulle features. Probabilmente avremmo dovuto continuare l'allenamento per qualche altra epoca, infatti l'allenamento della GAN non ha prodotto risultati soddisfacenti, probabilmente perché il discriminatore "non era abbastanza bravo" da consentire dei miglioramenti. Infatti la loss del generatore è risalita.

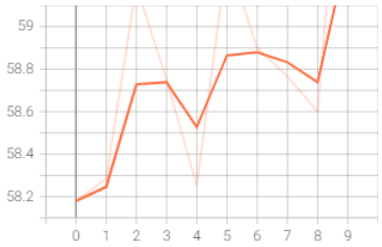

Figura 4.D - Andamento della loss del generatore durante l'addestramento NoGAN.

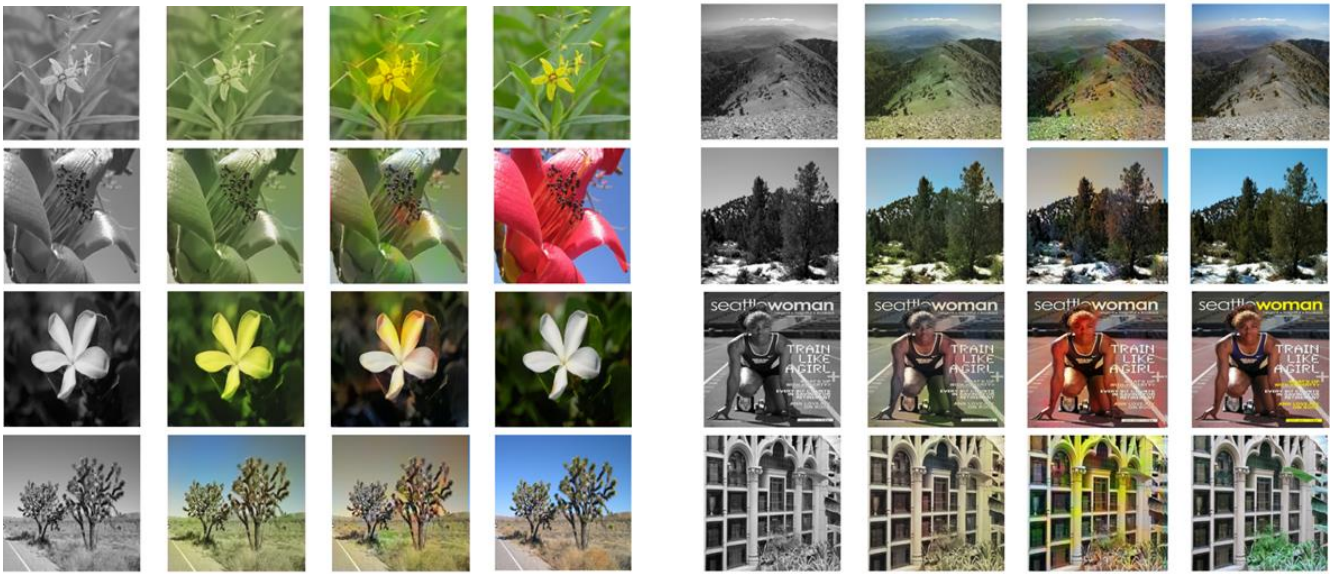

Figura 4.A - Esempi di risultati

5. Risultati e conclusioni

I risultati ottenuti sono soddisfacenti. Alcuni delle immagini sono state colorate davvero bene, mentre altre meno.
Il miglior modo per valutare i risultati è ovviamente osservarli visivamente, come si può vedere nella figura 5.A. Per quanto riguarda i criteri di valutazione oggettivi, nella tabella sottostante abbiamo riassunto i risultati ottenuti dai nostri modelli, utilizzando il nostro testset.

	CustomAccuracy	***RMSE***	***PSNR***
U-Net	0.265	0.206	--150.4
Pix2Pix	0.380	0.216	--153.5

Purtuppo non è stato possibile fare un confronto oggettivo con altri metodi come [2] in quanto sarebbe stato necessario scaricare i dataset utilizzati da questi articoli, ma i limiti di Google Colab e Google Drive non ic hanno permesso di perseguire il nostro obiettivo.

Possiamo comunque ritenerci soddisfatti dei risultati ottenuti.

6. Riferimenti

[1] Richard Zhang, Phillip Isola, Alexei A. Efros, « Colorful Image Colorization.,» *University of California, Berkeley. 1603.08511v5,* 2016.

[2] Gustav Larsson, Micheal Maire, Gregory Shakhnarovich., «Learning Representation for Automatic Colorization.,» *University of Chicago. 1603.06668v3,* 2017.

[3] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros, «Image-to-Image Traslation with Conditional Adversarial Networks.,» *Berkley AI Research (BAIR) Laboratory, 1611.07004v3,* p. 2018.

[4] Olaf Ronneberger, Philipp Fischer, Thomas Brox, «U-Net: Convolutional Networks for Biomedical Image Segmentation,» *arXiv:1505.04597,* 2015.

[5] Justin Johnson, Alexandre Alahi, Li Fei-Fei, «Perceptual Losses for Real-Time Style Transfer and Super-Resolution,» *Department of Computer Science, Standford University 1603.08155v1,* 2016.

[6] github/jantic, «DeOldify,» *https://github.com/jantic/DeOldify#what-is-nogan.*