UNIVERSITA' DEGLI STUDI "ROMA TRE"

DIPARTIMENTO DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

MACHINE LEARNING APPLICATO ALLA RIVELAZIONE DI TUMORI DELLA PELLE

Laureando

Piermarco Giustini

Matricola: 508002

Relatore

Prof. **Francesco Riganti Fulginei**

Anno Accademico 2019/ 2020

Indice

1 Introduzione

Le reti neurali offrono un insieme di strumenti molto potente che permette di risolvere problemi nell'ambito della classificazione, della regressione e del controllo non-lineare.

L'ispirazione per le reti neurali deriva dagli studi sui meccanismi di elaborazione dell'informazione nel sistema nervoso biologico, in particolare il cervello umano; Infatti, gran parte della ricerca sulle reti ha proprio lo scopo di capire più a fondo questi meccanismi.

Più in particolare ci si soffermerà su una parte dell'artificial intelligence, il deep learning, si presenterà il Deep learning per la visione computerizzata e il Deep learning per la generazione di immagini, grazie al framework opensource TensorFlow, e con l'utilizzo di una particolare libreria di esso Keras per la creazione di modelli.

L'obiettivo sarà quello di utilizzare questi particolari strumenti in un campo sensibile come quello della medicina per la prevenzione contro i tumori, infatti ci si soffermerà su come distinguere i melanomi della pelle da benigni e maligni.

Si utilizzeranno due particolari tipi di reti una Convolutional Neural Network, essa permetterà di distinguere le due classi di tumori tramite varie operazioni di convoluzioni sulle immagini, mentre la seconda rete utilizzata è una deep Convolutional Adversarial Network, questa invece permetterà di ricreare immagini di melanomi, mettendo in competizioni due reti una discriminate che si occuperà di distinguere le immagini vere da quelle false, e una generativa che invece si occuperà di crearle.

Un fattore importante anche sarà quello di utilizzare le due reti in modo sequenziale per sopperire alla scarsità di dati a disposizione utilizzando le reti generative, dunque andando a generare delle immagini di tumori benigni e maligni della pelle, per aiutare la fase di addestramento della Convolutional Neural Network.

2 Introduzione alle reti neurali

2.1 Approccio Biologico Deep Learning

Il cervello è composto da centinaia di miliardi di cellule nervose, i neuroni. Ogni volta che il cervello riceve uno stimolo sensoriale un gruppo di neuroni riceve un segnale, chiamato potenziale d'azione, se il potenziale d'azione è abbastanza forte viene propagato ai neuroni vicini tramite dei canali chiamati sinapsi. Questo processo si attiva a cascata, neuroni attivano altri neuroni, fino a quando il segnale si esaurisce.
I neuroni che si attivano insieme si legano insieme, questo è il concetto alla base dell'apprendimento, così i neuroni si uniscono in complessi reticoli che sono proprio le reti neurali.
Per l'appunto quest'ultime utilizzano lo stesso approccio per permettere alle macchine di apprendere dei dati cercando relazione tra di essi e raggiungendo livelli di astrazione sempre più profondi, o più semplicemente riuscendo a imparare ciò che un essere umano non sarebbe in grado di comprendere.
All'interno di una rete neurale artificiale i neuroni sono disposti su più strati:
uno strato di input che prende in ingresso i dati, ogni neurone di questa strada, rappresenta una proprietà del dataset (un insieme di dati), uno strato di output che fornisce il risultato della rete neurale uno più strati nascosti che si trovano tra lo strato di input e quello di output.
Il compito degli strati nascosti è di utilizzare la proprietà features o caratteristiche, che sono delle proprietà individuali e misurabili di un fenomeno osservato del dataset. Sono proprio queste caratteristiche che rendono una rete profonda, cioè quando contiene due più strati nascosti, in questi casi la rete utilizzerà le proprietà apprese in uno strato nascosto per apprendere ulteriori nuove proprietà ancora più significative nello strato successivo fino a utilizzarle per eseguire classificazioni, regressione e riconoscimento dei pattern[1].

[1] (https://blog.profession.ai/deep-learning-svelato-ecco-come-funzionano-le-reti-neurali-artificiali, s.d.)

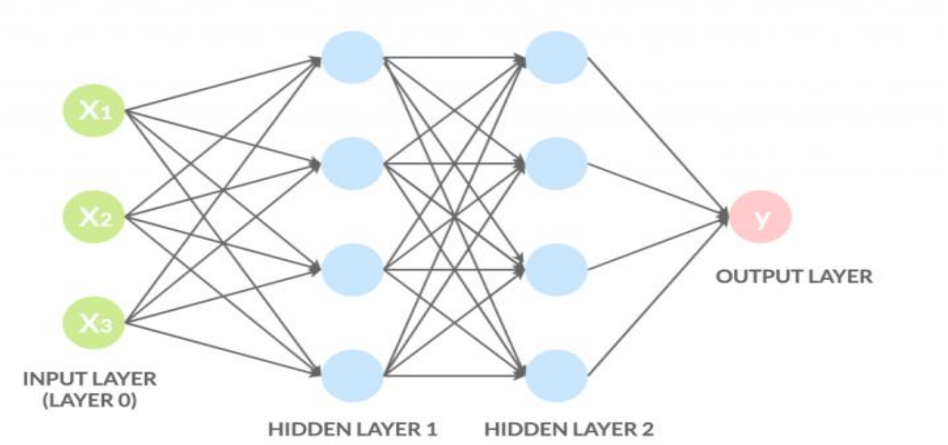


Figura 1 Anatomia di una rete neurale profonda con due o più neuroni

2.2 Tensori e interpretazione geometrica

È importante definire alcuni concetti che si incontreranno durante lo sviluppo delle reti in Python il primo concetto importante da analizzare è quello di tensore, che è definito da tre attributi chiave:

- Numero di assi, anche detto rango.
- Forma (shape) si tratta di una tupla di interi che descrive quante dimensioni ha il tensore lungo ciascun asse.
- Tipo di dati contenuti nel tensore.

Più in particolare si avrà a che fare con dei tensori 4D, questi tipi particolari di tensori permettono di rappresentare le immagini con più semplicità.
Essi hanno una forma definita come altezza e larghezza, numero di colori e numero di esempi (samples, height, width, color channel).

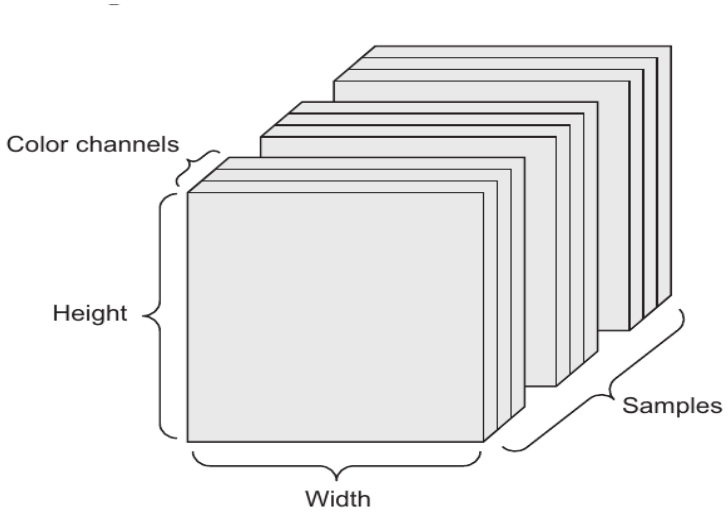


Figura 2 Tensore 4D per le immagini (Esempi, Larghezza, Altezza, Livelli di colore)

Un tipo di operazione sui tensori che è essenziale è il reshaping, eseguire il reshaping di un tensore significa ridisporre le sue righe e colonne per trovare la corrispondenza con una forma specifica, il nuovo tensore avrà lo stesso numero totale di coefficiente del tensore di partenza, però in una nuova forma.

Un altro concetto che si incontrerà spesso è quello di data batch, corrisponde all'asse dei campioni (chiamato anche la dimensione dei campioni), questo perché i modelli di deep learning non elaborano un intero dataset per volta piuttosto suddividono i dati in piccoli lotti detti batch.

Le reti neurali sono costituite interamente da catene di operazioni sui tensori e tutte queste operazioni non sono altro che trasformazioni geometriche dei dati di input.

Si consegue che si può interpretare una rete neurale come una trasformazione geometrica molto complessa di tensori in uno spazio multidimensionale, implementata tramite una lunga serie di semplici passi.

In 3D, ci si può referire a un piccolo esempio: si immaginano due fogli di carta colorata uno bianco e uno nero. Si pongono uno sopra l'altro, ora si stropicciano formando una pallina di carta. Questa pallina rappresenta i dati di input, e ciascun foglio di carta rappresenta una classe di dati in un problema di classificazione, il compito di una rete neurale è quello di trovare una trasformazione della pallina di carta che sia in grado di ridistenderla, in modo da rendere le due classi nuovamente ben separabili.

Con il Deep learning questa trasformazione sarebbe implementata con una serie di semplici trasformazioni dello spazio 3D[2].

2.3 Anatomia di una rete

Una volta definite le operazioni basilari sui tensori è possibile fare un approfondimento sull'anatomia di una rete di deep learning, passando a rassegna tutti gli elementi che la compongono, per comprenderne meglio la struttura e il funzionamento.

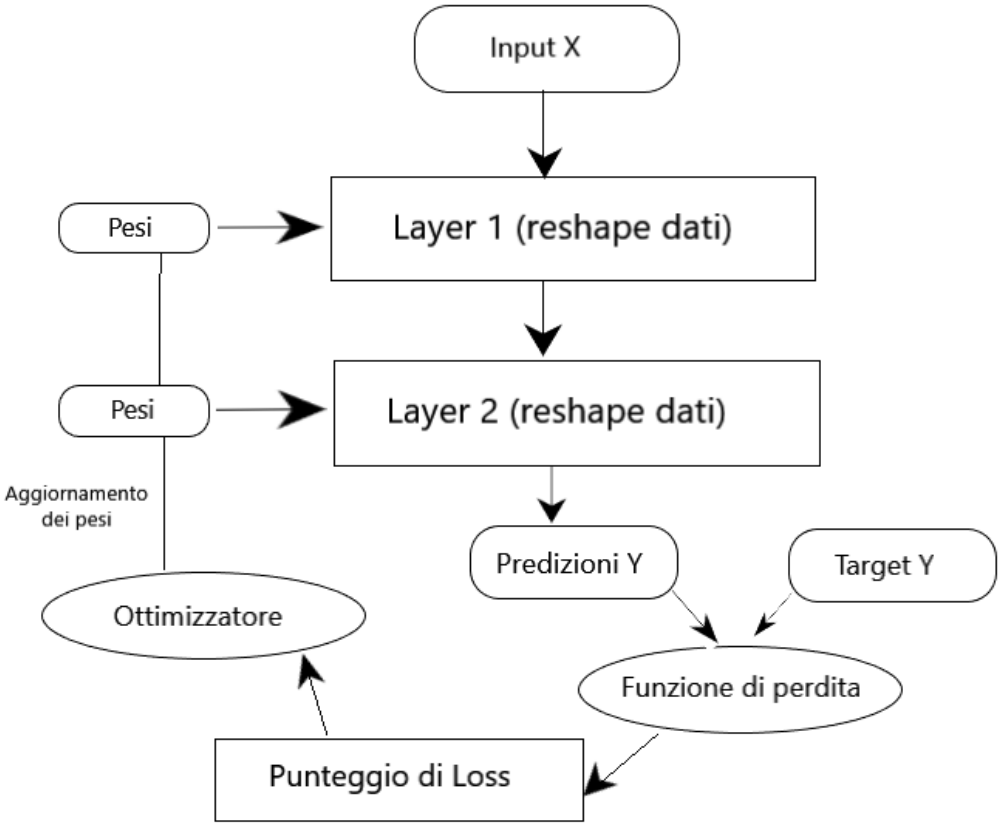


Figura 3 Modello semplificato di una rete neurale

[2] (Chollet, p. Tensori 3D Deep Learning con Python)

Il modello è composto da:

1. I layer, che per l'appunto combinati in una rete formano un modello.
2. Dati di input e i corrispondenti target.
3. La funzione obiettivo o di loss, cioè la funzione che dovrà essere minimizzata durante l'apprendimento, questa sarà la misura del successo della nostra rete.
4. L'ottimizzatore che determina il modo in cui la rete verrà aggiornata sulla base della funzione obiettivo, quest'ultimo sarà una variante specifica di discesa del gradiente.

In modo più analitico guardando la figura, si può dire che un modello o rete è composta da vari layer concatenati, che associano ai dati di ingresso (Input X) delle predizioni, dopo di che la funzione obiettivo (Funzione Loss) confronta queste predizioni (Predizioni Y) con i valori effettivi (Target Y), producendo un valore loss, che misura di quanto la predizione della rete manca il bersaglio, rappresentato dai valori effettivi o target, infine, l'ottimizzatore utilizza questo valore di loss generato per aggiornare i pesi prodotti dei vari layer.

Un layer è un modulo di elaborazione dei dati che prende come input uno o più tensori e ne produce in output, essi a loro volta sono uno o più tensori. Alcuni layer sono senza stato, ma nella maggior parte dei casi hanno uno stato dei pesi, che non sono altro che uno o più tensori appresi grazie alla discesa del gradiente, i quali, insieme formano la conoscenza della rete.

Layer differenti sono appropriati per tensori di formati differenti e per tipi di elaborazioni dati differenti (non posso utilizzare gli stessi layer per elaborazione di immagini e di testo).

La costruzione di modelli di deep learning in Keras viene svolta unendo insieme gruppi di layer compatibili a formare pipeline di trasformazione dei dati, si deduce da quello detto in precedenza che la compatibilità fra layer è un'importante proprietà.

Essa fa riferimento in particolare al fatto che ogni layer accetterà solo tensori di input di una certa forma e restituirà tensori di output di una forma differente.

2.4 Ottimizzazione basata sul gradiente

Come detto in precedenza ciascun layer neurale del nostro modello trasformerà all'interno della rete i suoi dati di input in tensori.
Ogni trasformazione creerà dei pesi o parametri addestrabili, questi pesi contengono le informazioni apprese dalla rete a causa della sua esposizione ai dati di addestramento che per comodità definirò come X.
Inizialmente, queste matrici di peso contengono piccoli valori casuali (anche se è possibile cambiare l'inizializzazione, ad esempio con una distribuzione gaussiana).
I valori inizialmente generati da questa rappresentazione saranno totalmente casuali, ma durante l'addestramento prenderanno di significato.
A questo punto occorre aggiustare gradualmente i pesi. Questo aggiustamento graduale, ovvero l'addestramento, il training, è sostanzialmente la fase di learning del deep learning, questo processo avviene in modo iterativo con quattro principali passi:

1. Il primo passo è senz' altro quello di estrarre un lotto o batch dai campioni di addestramento definiti come X, e i corrispondenti targets o predizioni Y.
2. Nel secondo passo si applica la rete a X (cioè un passo chiamato anche avanzamento) questo serve per ottenere le predizioni della rete che definiremo come Y_pred.
3. Nel terzo passo si calcola il valore di loss dalla rete sul lotto o batch dei dati, una misura della distanza fra Y e Y_pred.
4. Infine, come quarto e ultimo passo si aggiornano tutti i pesi della rete in modo che si riduca il valore loss sul lotto di batch.

Alla fine, si cerca di ottenere una rete che "manca" di poco il bersaglio (questo è il significato da considerare del termine di "loss") più la loss è bassa più ci si è avvicinati al bersaglio sui dati di addestramento, che offre quindi la minima distanza fra le predizioni Y_pred e le predizioni dei target Y.
L' obiettivo finale della rete, quindi, è quello di "minimizzare i risultati della funzione loss", tornando al passo quattro cioè quello di aggiornare i pesi della rete. Si sfrutta il

fatto che tutte le operazioni impiegate sono differenziabili, ciò significa che la funzione può essere derivata o è derivabile, l'operazione di derivazione ci consente data una funzione *f(x) = y*, che associa un numero reale x a un corrispondente valore reale in y, di analizzare delle piccole variazioni nello studio di una funzione, per una minima variazione definita come dx sull'asse delle ascisse, sarà associata una variazione dy sull'asse delle ordinate.

Ad esempio, se il risultato dell'operazione di derivazione è negativo allora vuol dire che a una piccola variazione di dx la funzione *f(x)*, cioè il valore di y è decrementato.

Quindi si deduce da questa operazione, che è essenziale per la ricerca del minimo valore di y.

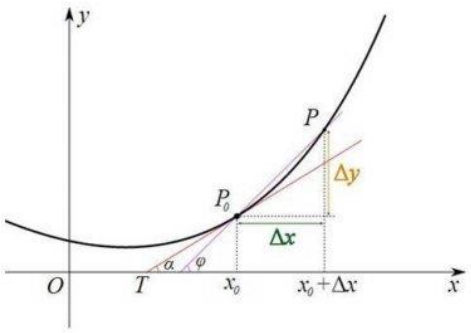


Figura 4 Operazione di derivazione per la ricerca del minimo

Un gradiente, quindi è la derivata di un'operazione su tensore, si tratta della generalizzazione del concetto di derivata alla funzione con input multidimensionale, ovvero alle funzioni che impiegano come ingresso dei tensori e non un semplice valore di x[3].

[3] (Chollet, p. Funzione Obiettivo Deep Learning con Python)

2.5 Ambiente di sviluppo

Prima di cominciare a mettere mano sul codice e sulla realizzazione del progetto vero e proprio, si decide cosa e quali strumenti si utilizzeranno per lavorare con le applicazioni di intelligenza artificiale, machine learning e deep learning, si è consapevoli però di quanta potenza di calcolo sia necessaria per implementare modelli sufficientemente robusti ed efficienti.

Per i primi test, infatti è per lo più sufficiente affidarsi al proprio personal computer, ma con l'aumentare della dimensione dei dataset l'esecuzione di algoritmi di addestramento complessi come quelli di deep learning diventa rapidamente proibitiva. Per risolvere questo problema, esistono numerosi servizi cloud che offrono potenza di calcolo, spesso a pagamento o altrimenti con varie limitazioni.

Uno di questi strumenti è Google Colab, una piattaforma che seppur con alcune limitazioni, permette di eseguire codice direttamente sul Cloud, sfruttando la potenza di calcolo fornita dalle risorse di Google.

Per sfruttare le funzionalità di tale piattaforma, tutto ciò di cui si ha bisogno è un account Google, mediante il quale si potrà effettuare il login.

Per eseguire codice, Google Colab sfrutta i Jupyter Notebook, questi non sono altro che documenti interattivi nei quali si può scrivere (e quindi eseguire) il codice.

Più precisamente, tali documenti permettono di suddividere il nostro codice in celle chiamate snippet, ognuna delle quali può contenere anche del testo informativo, eventualmente formattato in Markdown (indentazione simile a html).

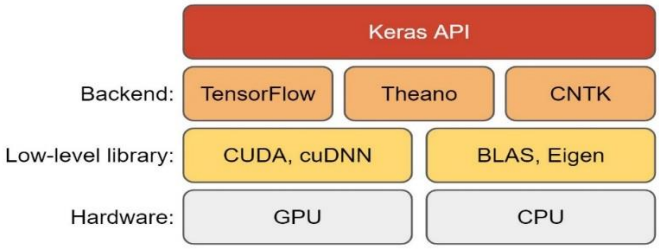
Tramite un unico documento, è possibile eseguire tutti gli step di un processo di analisi o processing, sia descriverne il comportamento in linguaggio naturale.

Il linguaggio di programmazione che si utilizzerà sul Jupiter Notebook sarà Phyton, il più comune in materia di deep learning.

Colab possiede vari tipi di configurazioni per permettere alla macchina di eseguire al meglio il codice. Cliccando sul menu Runtime, si potrà scegliere se abilitare il supporto all'uso della GPU, o non, comunque vengono forniti di base 12.5 Gb di ram e 64 Gb di disco, che sono facilmente upgradabili a 25 Gb di ram e 100 Gb di disco.

Altra scelta importante è quali librerie utilizzare per sviluppare la nostra rete, Keras è una libreria per modelli che fornisce gli elementi costitutivi di alto livello per lo sviluppo dei modelli di deep learning.
Non gestisce le operazioni di basso livello come la manipolazione dei tensori e la differenziazione, al contrario per farlo conta su una libreria per tensori specializzata e ottimizzata che funge da backend engine.
Quindi invece di scegliere un'unica libreria per tensori cui legare l'implementazione Keras gestisce il problema in modo modulare, questo significa che permette di differenziare i vari backend engine e decidere quello che ci risulta più famigliare, attualmente i tre principali sono:

- TensorFlow sviluppato dalla Google (quello utilizzato per il progetto)
- Theano
- Microsoft Cognitive Toolkit (CNTK).



Il codice completo sarà direttamente disponibile su Google Colab[4].

Figura 5 Livelli di sviluppo per deep learning

[4] https://colab.research.google.com/drive/1bMekZwTtThVOC5NS8M2jOXUskB8CE9V1?usp=sharing CNN
https://colab.research.google.com/drive/1tH_9mPaXjAPvXXJVfbNSzcU9S--PT_6P?usp=sharing DCGAN

3 Convolutional Neural Network

Una rete neurale convoluzionale o CNN è un modello di Deep Learning che può prendere un'immagine in input, riconoscere vari aspetti e oggetti nell'immagine ed essere in grado di differenziarli l'uno dall'altro.

La preelaborazione richiesta in un ConvNet è molto ridotta rispetto ad altri algoritmi di classificazione su immagini, e risulta più semplice sistemarle prima delle fasi di addestramento.

Nei metodi antecedenti alle reti convoluzionali i filtri erano progettati a mano, con una formazione difficile e spesso insufficiente, le ConvNets invece hanno la capacità di apprendere questi filtri, caratteristiche e pattern ricorrenti nell'immagine in modo automatico.

L'architettura di una ConvNet è analoga a quella del modello di connettività dei neuroni nel cervello umano, ed è stata ispirata dall'organizzazione della corteccia visiva, i singoli neuroni rispondono agli stimoli solo in una regione ristretta del campo visivo nota come campo ricettivo[5].

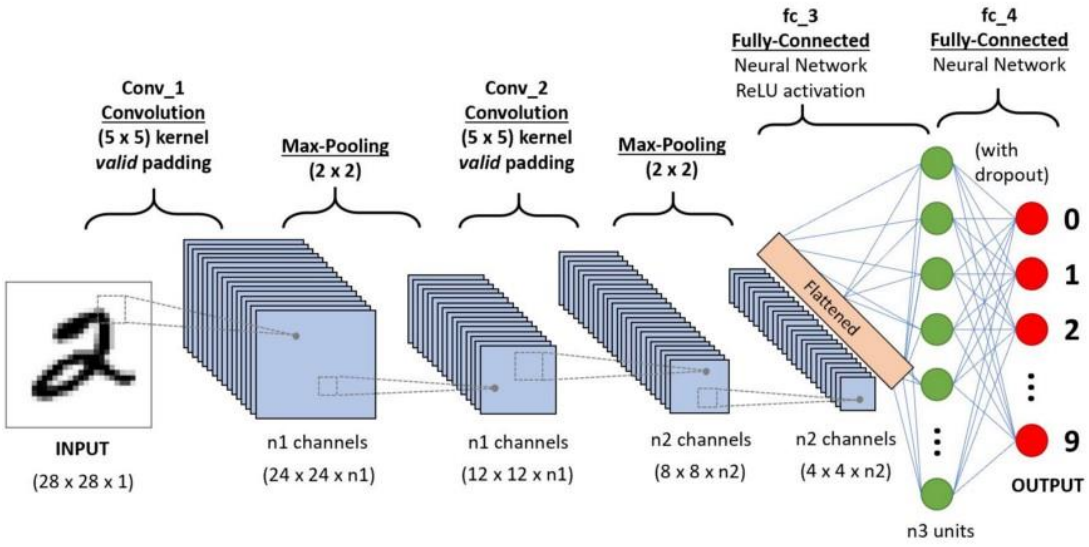


Figura 6 Esempio di modello di Convolutional Neural Network applicato al dataset MINST

5 (https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, s.d.)

3.1 Dati e immagini

Per i dati si utilizza un dataset di immagini preso da una competizione su Kaggle (una famosa piattaforma di machine learning, più in generale per tutta la scienza dei dati), più di preciso nel ISIC Archive[6], questo dataset comprende 3212 immagini di melanomi alla pelle di due tipologie maligni e benigni divisi in due cartelle una di train che servirà per allenare la nostra rete e una di test per la validazione.
Il primo passo è stato quello di scaricare in locale le immagini e poi fare l'upload di quest'ultime su Google Drive per poterle visualizzare e utilizzare.

```
drive.mount('/content/drive', force_remount = True)
```

Questo snippet di codice inserito all'interno dei nostri fogli Colab ci consentirà di accedere a Google Drive tramite un processo semplice di autorizzazione. Una volta entrati su Google Drive si possono definire i vari percorsi nel quale sono salvate le immagini.

```
folder_benign_train =
'/content/drive/MyDrive/Dati/train/benign'
folder_malignant_train =
'/content/drive/MyDrive/Dati/train/malignant'
```

Nel codice sottostante è definita una lambda function, un tipo di funzione anonima che ha la sua espressione in una sola riga, essa convertirà le immagini aperte tramite la libreria Image, in array numpy[7].
Essa verrà utilizzata per caricare una per una le immagini tramite un ciclo for.
Per esempio, una volta terminato avremo X_benign che risulterà essere un array numpy di shape (1419, 224, 224, 3) cioè un tensore 4D, dove il primo valore indica il

6 (https://www.isic-archive.com/#!/topWithHeader/onlyHeaderTop/gallery?filter=%5B%5D)I dati utilizzati non sono stati utilizzati per scopo monetario sono ai fini della ricerca

7 (https://numpy.org/doc/stable/reference/routines.html)

numero di samples, il secondo e il terzo corrispondo alla larghezza e all'altezza, e il quarto al numero dei canali di colore dell'immagine.
Questa funzione è solo per l'immagine dei tumori benigni nella cartella train ma lo stesso procedimento sarà ripetuto altre tre volte con i tumori maligni della cartella train, e quelli benigni e maligni della cartella test.

```
read = lambda imname: np.asarray(Image.open(imname).convert("R
GB"))
ims_benign = [read(os.path.join(folder_benign_train,
filename)) for filename in os.listdir(folder_benign_train)]
X_benign = np.array(ims_benign, dtype='uint8')
```

Genero degli array numpy formati solo da 0 che sono i tumori benigni e 1 che sono i tumori maligni[8].

```
y_benign = np.zeros(X_benign.shape[0])
y_malignant = np.ones(X_malignant.shape[0])
```

Si concatenano i tumori benigni e maligni, sull'asse 0 cioè sull'asse x.
Così si avranno 2 dataset con le immagini X_train e X_test che saranno dei tensori 4D e y_train e y_test che contengono le lables cioè tensori a 1D che avranno la stessa grandezza del numero di samples dei tensori 4D.

```
X_train = np.concatenate((X_benign, X_malignant), axis = 0)
y_train = np.concatenate((y_benign, y_malignant), axis = 0)
```

Ora grazie alla libreria matplotlib si possono vedere le immagini del dataset per farsi una migliore idea su che cosa stiamo lavorando.

[8] (https://docs.python.org/3/library/os.html)

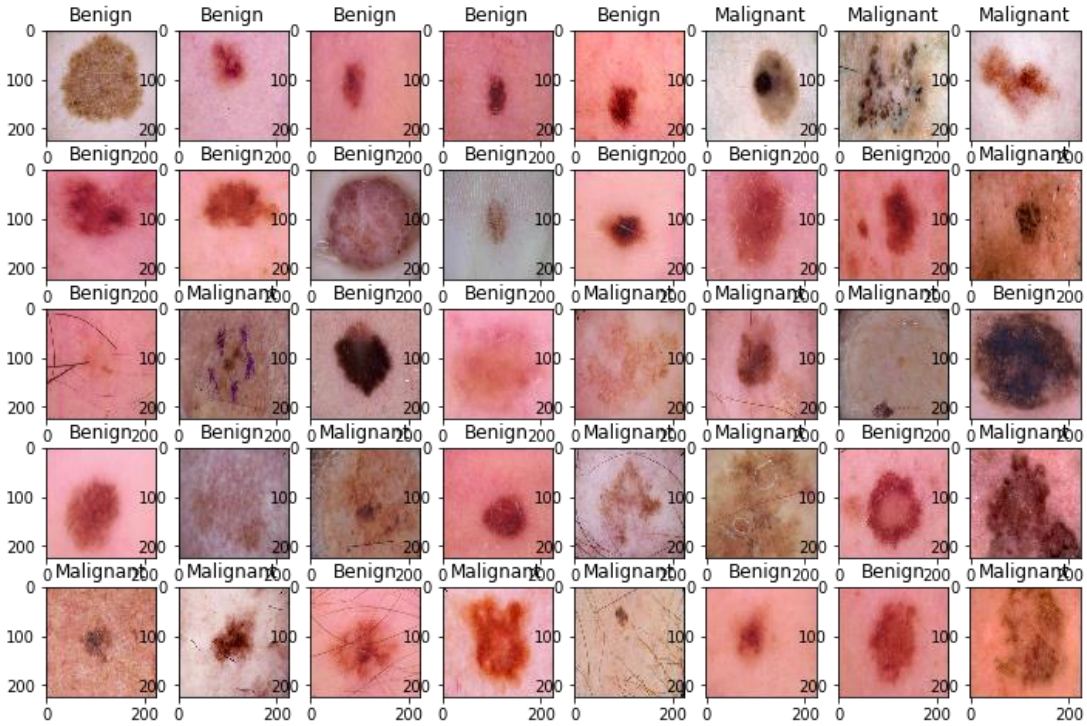


Figura 7 Immagini con matplotlib di melanomi

3.2 Modello

3.2.1 Normalizzazione

Una volta presa dimestichezza con i dati si passa alla creazione del modello di deep learning per la visione computerizzata delle immagini.

Per poter utilizzare il modello in modo appropriato, bisogna effettuare un ridimensionamento dei dati, una tecnica che trasforma i dati grezzi in un formato comprensibile da dare in pasto alla nostra rete.

I dati grezzi (dati del mondo reale) sono quasi sempre incompleti e non possono essere inviati in questo modo ad un modello di deep learning, ciò causerebbe alcuni errori, che renderebbero il modello non adatto ad eseguire una buona previsione.

Per questo motivo si effettua un lavoro detto di scaling o normalizzazione si aggiungere/sottrarre una costante e poi moltiplica/divide per un'altra costante, in modo che le caratteristiche possano trovarsi tra determinati valori minimi e massimi.

Normalmente, l'intervallo specificato è tra [0,1], come avviene per la standardizzazione. Come si può vedere nel codice sottostante ho utilizzato il formato float32 e poi ho standardizzato per avere valori compresi tra 1 e 0.

```
X_train.astype('float32')
X_test.astype('float32')
X_train = (X_train - 127.5) / 127.5
X_test = (X_test - 127.5) /127.5
```

A questo punto si deve creare il modello, ma la prima domanda è quello di definire che tipo di problema si stia affrontando.
L' obbiettivo è quello di saper distinguere un tumore benigno da uno maligno, quindi ci ritroviamo nella categoria di un problema di classificazione binario. Per prima cosa quindi ci dobbiamo servire di un'ulteriore passaggio per sistemare i nostri dati da dare in pasto alla rete, modifichiamo le lables tramite la funzione di Keras to_categorical(), quest' ultima converte il nostro array numpy fatto da integers compresi tra 0 e 1 in una matrice di classi binarie.

```
y_train = tf.keras.utils.to_categorical(y_train, num_classes=2)
y_test =tf.keras.utils.to_categorical(y_test, num_classes=2)
```

3.2.2 Layer del modello (Conv2D, MaxPool2D, Dropout)

Si costruisce il modello per la rete, con la funzione sequential di Keras si possono concatenare molto rapidamente tutti i vari layer, aggiungendoli uno ad uno con la funzione add, sempre di Keras.
Si utilizzano principalmente 3 tipi di layer che sono Conv2D, MaxPool2D e Dropout. Conv2D è un'operazione di convoluzione, tramite di essa la rete impara vari pattern locali e non un insieme di informazioni. L'operazione di convoluzione si adopera su tensori 4D per l'appunto, come le immagini, chiamate anche mappe delle caratteristiche di input, esse presentano i due classici assi spaziali altezza e larghezza,

e anche uno sulla profondità che è pari a 3, perché le immagini sono informato RGB (red, green, blue)[9].

Una volta svolta l'operazione verrà restituita una mappa delle caratteristiche di output, con la stessa dimensione del tensore precedente.

I parametri chiave dell'operazione di convoluzione in ordine sono:

- `[Filters]` La profondità del filtro, cioè il numero di filtri calcolati dalla convoluzione.
- `[Kernel Size]` La dimensione del filtro estratto, l'immagine viene divisa in tante finestre. Con 3x3 la finestra avrà la dimensione di 9 pixel totali.
- `[Padding]` Consiste nell'aggiungere un numero appropriato di righe e colonne in modo che sia possibile centrare tutte le finestre, senza avere uno scarto disallineato. Con Same si applica un Padding tale da avere un output con la stessa larghezza e la stessa altezza dell'input.
- `[Input shape]` Rappresenta la dimensione della mappa delle caratteristiche di input. Quindi come la dimensione delle nostre immagini `(224, 224, 3)`
- `[Activation]` La funzione di attivazione. Con ReLU si indica, un'operazione non lineare per evitare che i layer del modello apprendano solo trasformazioni lineari[10].

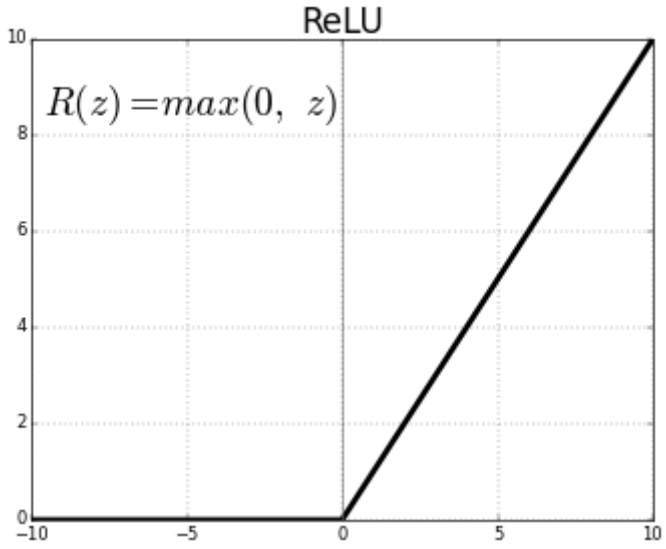


Figura 8 Funzione di attivazione ReLU

[9] (https://keras.io/api/layers/convolution_layers/convolution2d/)

10 (https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6)

- `[Kernel initializer]` La distribuzione iniziale dei pesi.

MaxPool2D consiste nell' estrarre finestre dalle mappe delle caratteristiche di input di produrre in output il valore massimo di ogni canale. Cioè esso ha il compito di sub campionare le mappe delle caratteristiche.
Ad esempio, il primo passaggio di Maxpooling con pool_size = `(2,2)` dimezza le dimensioni delle caratteristiche di input da `(224, 224, 64)` a `(112, 112, 64)` da notare non altera la profondità della convoluzione che rimane sempre di 64 filtri.
DropOut invece, modifica non la funzione di costo della rete, ma la rete stessa, questo layer prevede di impostare un certo numero di dati in input a 0 (questo aiuta la rete a non incappare in un problema di overfitting)[11].
Per esempio, con il valore `0,3` verranno scartati, cioè messi a `0` il `30%` dei valori della mappa delle caratteristiche di input.

Ecco come si comportano i filtri delle ConvNet:

```
model.add(Conv2D(Filters, kernel_size, padding, input_shape,
activation, kernel_initializer))
model.add(MaxPool2D(pool_size)))
model.add(Dropout(0.3))
```

[11] (https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D)

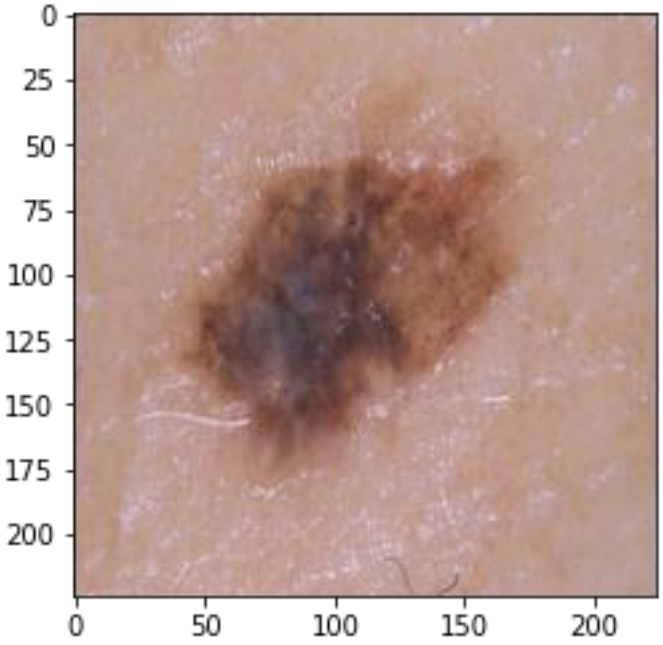


Figura 9 Immagine di neo senza alcun tipo di layer

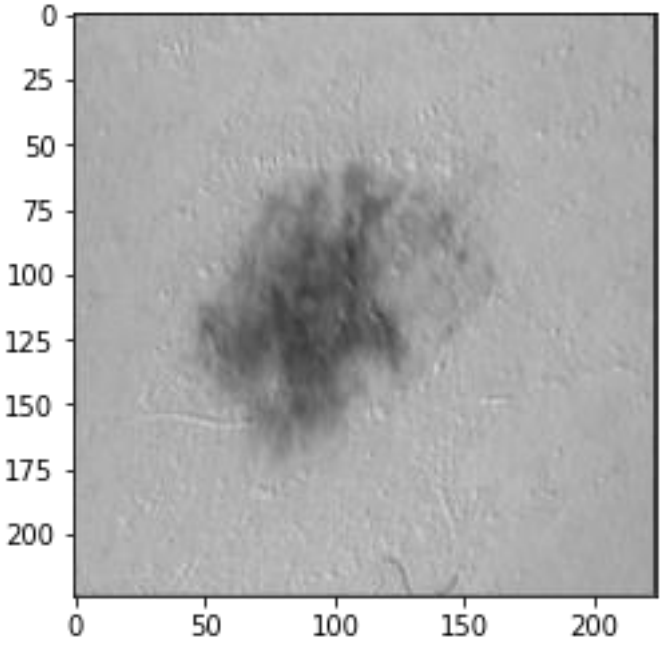


Figura 10 Immagine del 3 layer di attivazione di CONV2D

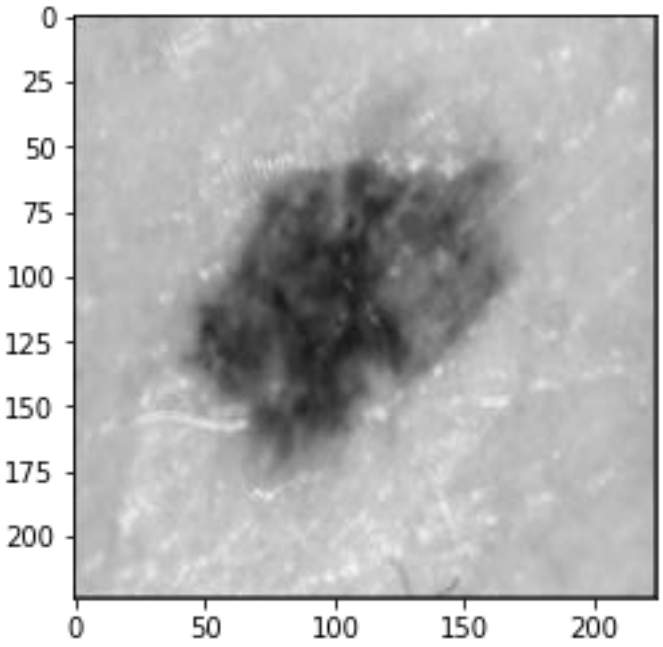


Figura 11 Immagine del 7 layer di attivazione di CONV2D

Diamo uno sguardo più in generale a cosa succede in ogni layer definito dal nostro modello, in questo modo, è possibile fasi un'idea generale di come stia lavorando:

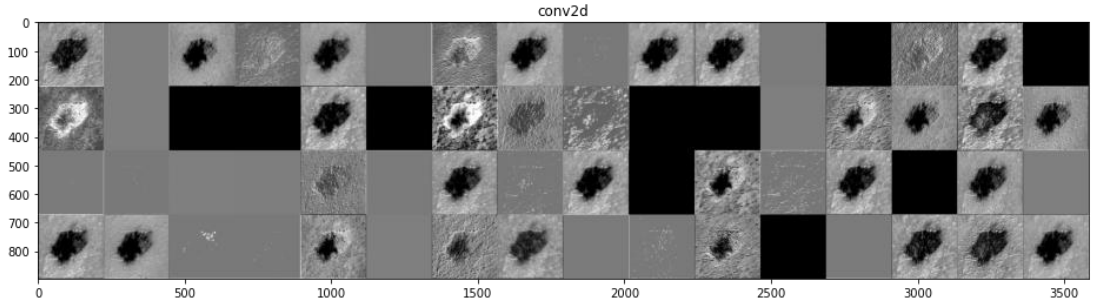


Figura 12 Primo strato conv2D

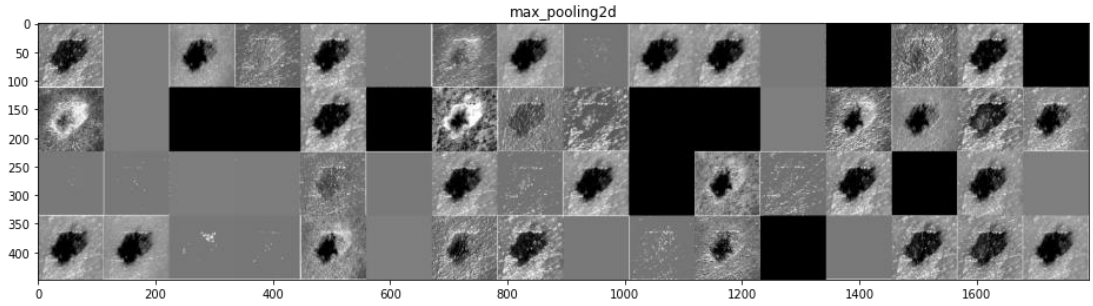


Figura 13 Primo strato MaxPooling2D

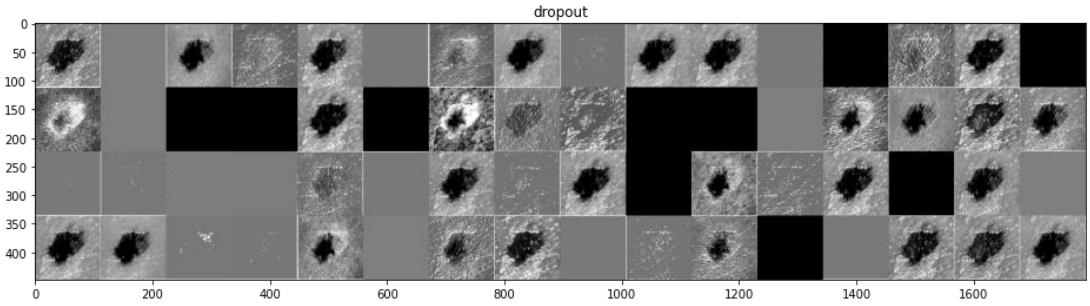


Figura 14 Primo strato DropOut

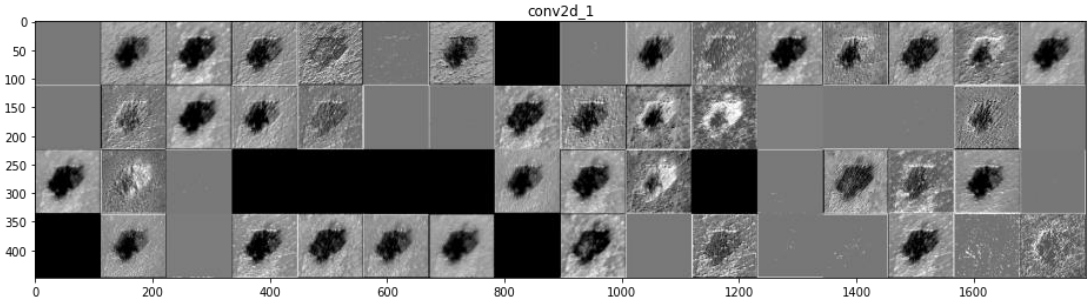


Figura 15 Secondo strato conv2D

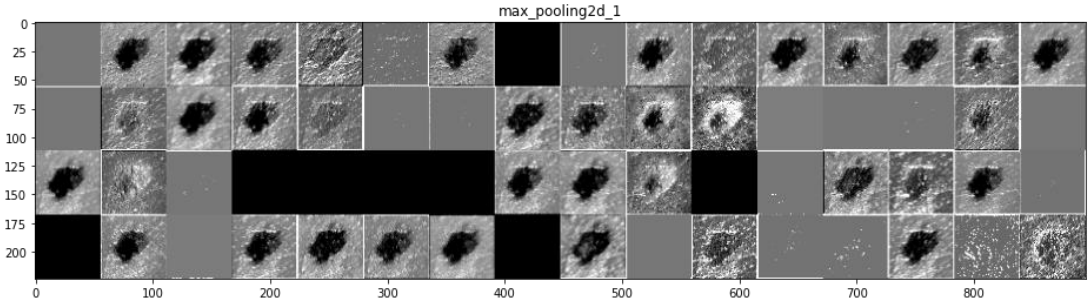


Figura 16 Secondo strato MaxPooling2D

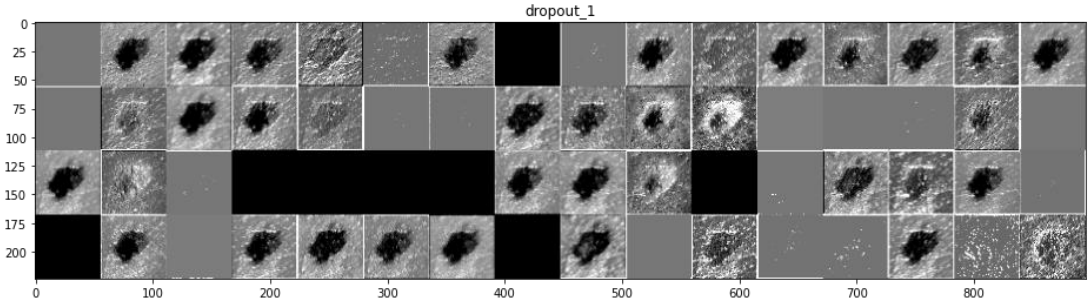


Figura 17 Secondo strato DropOut

3.2.3 Attivazione e Compilazione

Una volta definite le operazioni principali per la ConvNet si appiattisce il tensore 3D in un tensore 1D per poter poi darlo in pasto a un layer densamente connesso.
Il primo dense layer ha un tot di unità nascoste o anche neuroni del layer, più unità nascose introduciamo, più la rete avrà libertà di apprendere nuovi parametri.
Per l'ultimo layer Dense come attivazione si utilizza la funzione Sigmoid e non più Relu.
Il motivo principale per cui usiamo la funzione Sigmoid è perché esiste tra (0 e 1), pertanto è particolarmente comoda per i modelli in cui si devono prevedere le probabilità come output, poiché la probabilità di qualsiasi cosa esiste solo tra 0 e 1.
Inoltre, è importante inserire soltanto due unità queste corrispondo alle due classi che dobbiamo andare a distinguere[12].

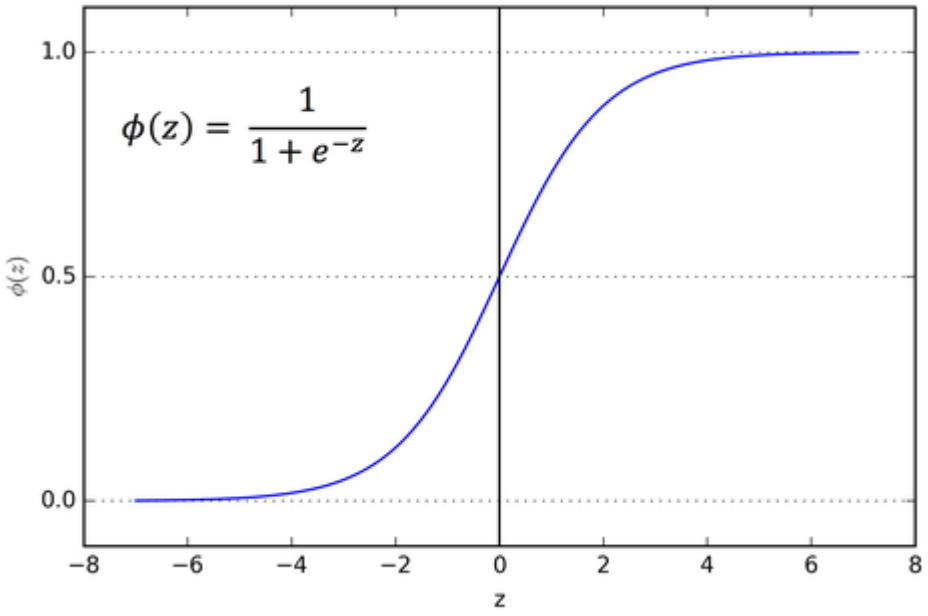


Figura 18 Funzione di attivazione Sigmoid

```
model.add(Flatten())
model.add(Dense(units, activation, kernel_initializer))
```

[12] (https://stats.stackexchange.com/questions/233658/softmax-vs-sigmoid-function-in-logistic-classifier)

```
model.add(Dense(2, activation='sigmoid'))
```

Infine, si sceglie una funzione obiettivo e un ottimizzatore, poiché si tratta di un problema come detto in precedenza di classificazione binaria, per calcolare il valore di distanza (loss) si utilizza la binary_crossentropy, o anche entropia incrociata che misura la distanza fra le distribuzioni di proprietà.
Mentre come ottimizzatore si può scegliere tra la funzione Rsmprop o Adam entrambe adatte per problemi di classificazione binaria.
La metrica invece sarà basata su "accuracy".

```
model.compile(optimizer , loss , metrics)
```

3.3 Convalida all'approccio k-fold

Per valutare la rete e aggiustarne i parametri per migliorarne le prestazioni i risultati, in termini di valori di accuratezza e loss, si deve subire un processo di convalida, ma poiché i dati risultano essere pochi il set di convalida potrebbe restituire punteggi molto differenti a seconda dei dati di convalida utilizzati.
Per evitare questo problema, la migliore pratica è di utilizzare la convalida incrociata o K-fold.
Essa consiste nel suddividere i dati disponibili in K partizioni identiche, e in modo iterativo addestrare K-1 partizioni, viene lasciata una singola partizione K per la convalida, così tante volte per quanto sono le partizioni K create.
Il punteggio di convalida finale deriva dalla media dei K punteggi di convalida ottenuti[13].

[13] (https://stackoverflow.com/questions/64626869/k-fold-crossvalidation-on-images)

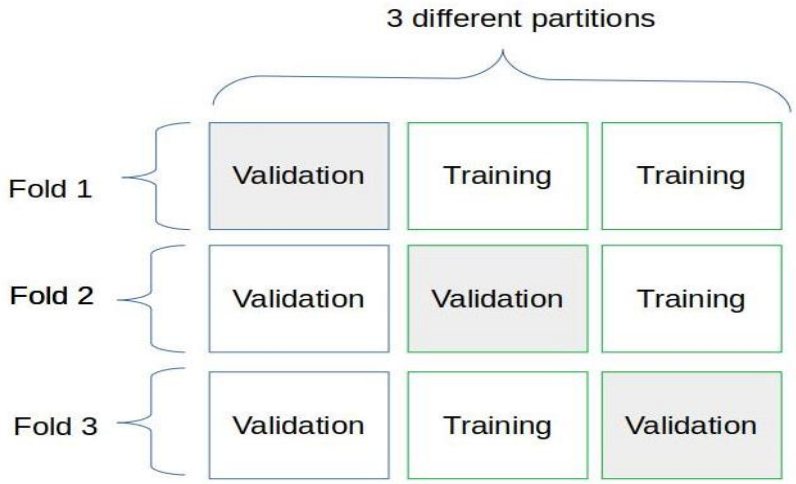


Figura 19 Struttura di una suddivisione di K-fold con K= 3

```
for train, test in kfold.split(inputs, targets):

    model.fit(inputs[train], targets[train], epochs,
    batch_size, verbose=0)
    scores = model.evaluate(inputs[test], targets[test],
    verbose=0)
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])
    fold_no = fold_no + 1
```

3.4 Addestramento

Una volta definito il modello, il metodo di convalida della rete si iniziano i primi cicli di addestramento, variando i parametri della rete creata per trovare la migliore soluzione al problema che stiamo andando ad affrontare.

Come sistema di giudizio si utilizzeranno i grafici, e i punteggi di convalida della K-fold.

I grafici si basano su accuracy che risulterà essere la precisione della rete, mentre la loss è la penalità per una previsione sbagliata più il valore di perdita risulta essere basso migliore saranno le capacità di prevedere della rete.

In altre parole, la loss è un numero che indica la cattiva previsione del modello su un singolo esempio. Se la previsione del modello è perfetta equivale a 0 altrimenti sarà maggiore.
Inoltre, sono presenti anche i valori di validazione per K-fold con 3 split con il punteggio accuracy medio, accanto una parentesi che indica la varianza massima tra le tre prove (+-1 indica la differenza media tra le varie convalide che in questo caso è di un punto), e per ultimo il punteggio di loss medio.
La funzione per allenare la rete si presenta così:

```
history = model.fit(X_train, y_train, validation_split,
epochs, batch_size, verbose,
callbacks=[learning_rate_reduction])
plot(history)
```

3.4.1 Scelta Ottimizzatore

La prima scelta e una delle più importanti, ricade su quale funzione di ottimizzazione impiegare per la rete, per quanto riguarda il learning rate delle CNN si utilizza un valore pari `0.00001` che è anche il più consigliato ed utilizzato.
Mentre per la scelta dell'ottimizzatore si può sia utilizzare Adam o RMSprop:

```
optimizer = Adam(lr=lr)
optimizer = RMSprop(lr = lr)
```

entrambi si adattano bene per il tipo di problema che andiamo ad affrontare, per decidere quale utilizzare si svolgono delle prove:

- Adam è il primo ottimizzatore utilizzato come si può vedere dal grafico si comporta particolarmente bene, non raggiungere il valore dell'`80%` di accuracy nella fase di traning e la validazione si stabilizza intorno al `70%` di accuracy.

Per la funzione di loss, si comporta altrettanto bene aggirandosi nella fase di traning nell'intorno `0.6`, mentre sui dati di validazione si aggira nell'intorno di `0.51`.

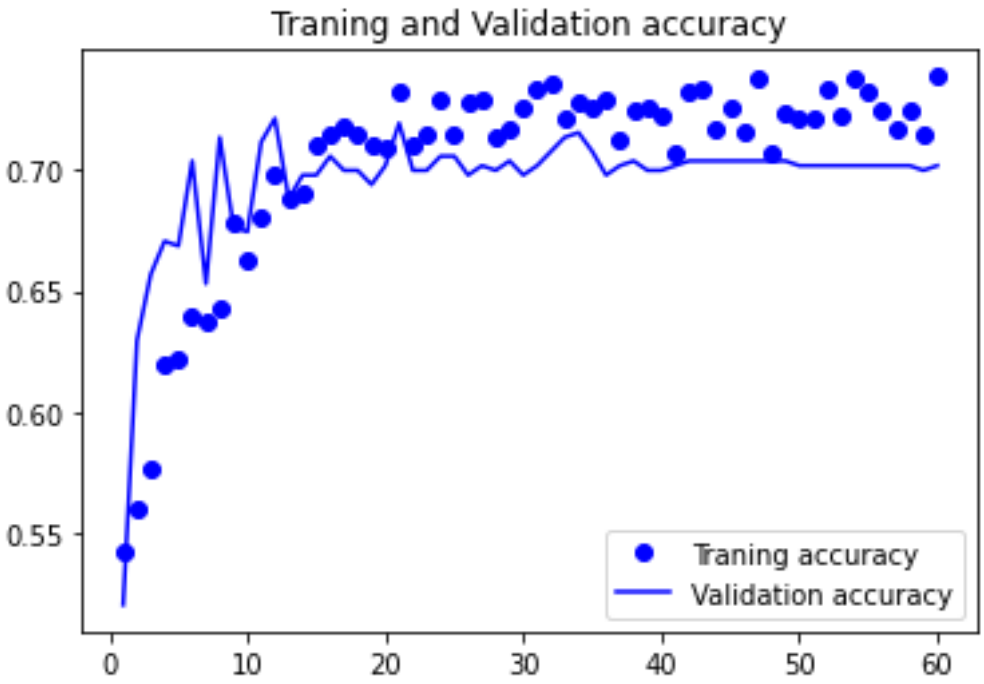


Figura 20 Grafico accuracy con ottimizzatore Adam

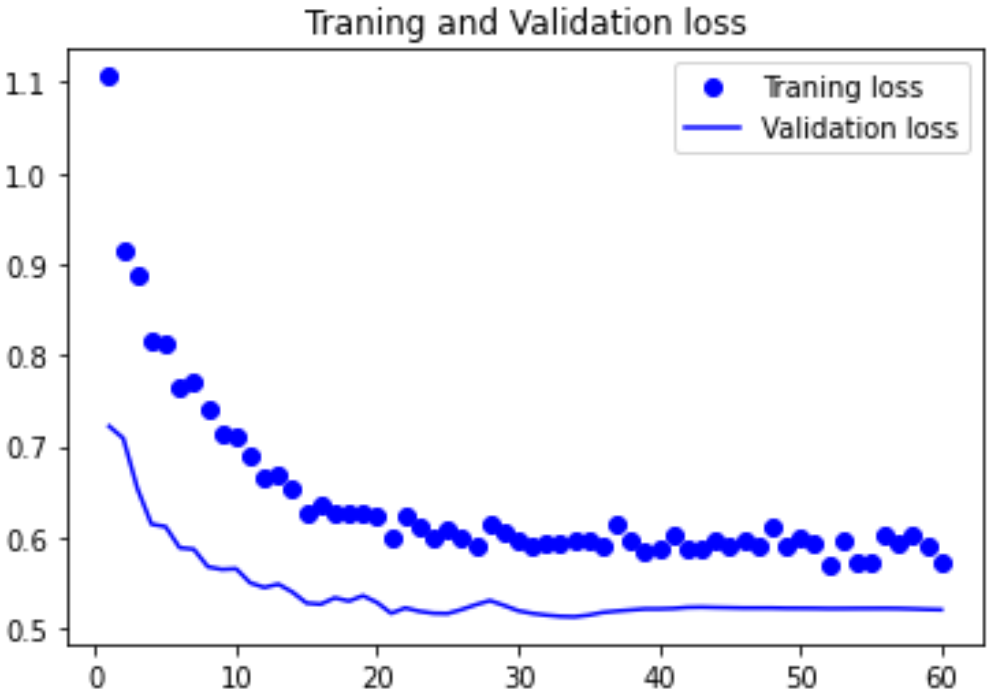


Figura 21 Grafico loss con ottimizzatore Adam

```
Media degli scores nella validazione K-Fold:
> Accuracy: 71.01 (+- 1.26)
> Loss: 0.52
```

- Rsmprop è il secondo ottimizzatore utilizzato come si può vedere dal grafico si comporta bene ma non quanto Adam non raggiungere il valore dell'`75%` di accuracy nella fase di traning e la validazione si stabilizza intorno al `68%`.
  Per la funzione di loss si comporta bene aggirandosi nella fase di traning nell'intorno `0.55`, mentre nella validation nell'intorno `0.52`.

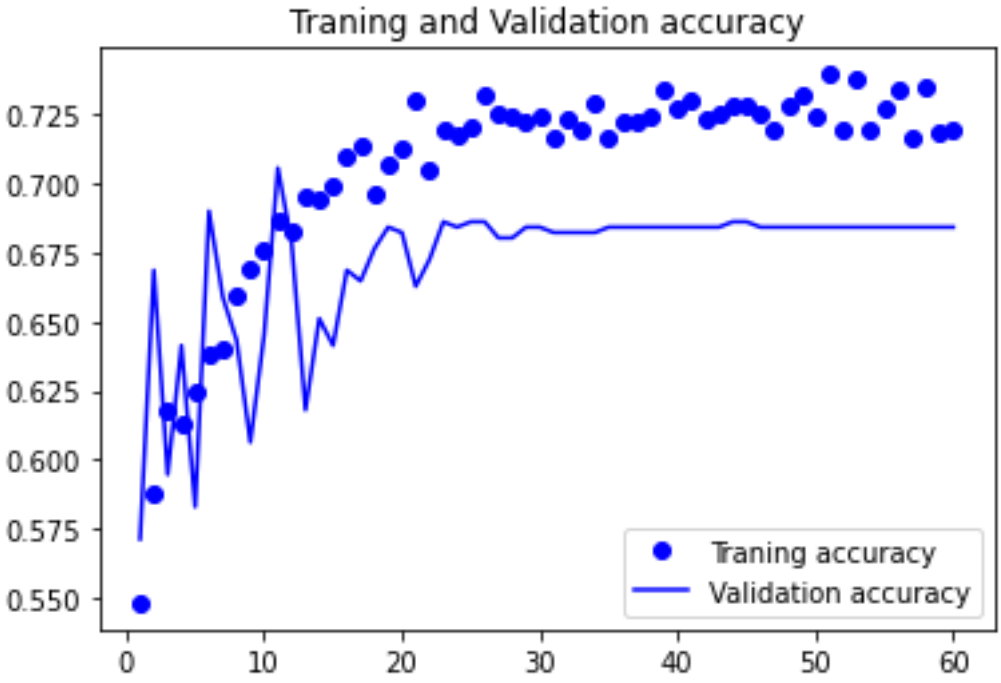


Figura 22 Grafico accuracy con ottimizzatore Rsmprop

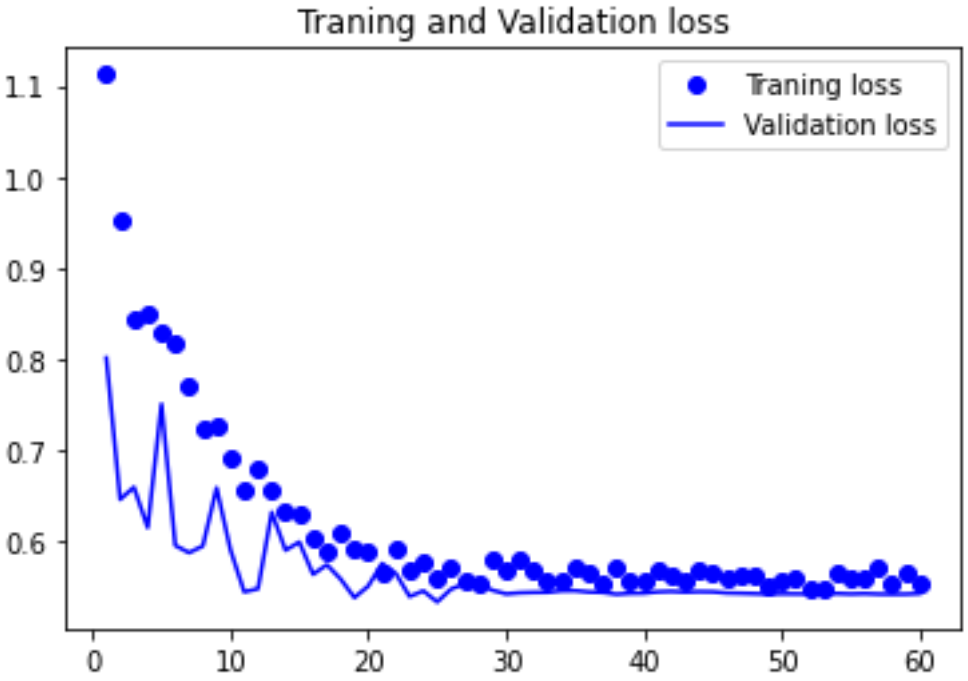


Figura 23 Grafico loss con ottimizzatore Rsmprop

`Media degli scores nella validazione K-Fold:`

```
> Accuracy: 70.00 (+- 1.03)
> Loss: 0.52
```

Da entrambi i grafici si può vedere come le reti anche utilizzando ottimizzatori differenti soffrano di un problema di overfitting. Nel capito successivo si vede cosa è il problema dell'overfitting e underfitting e come risolverlo.
Si può notare inoltre come le reti non riescano dopo un tot di epoche a migliorare le proprie prestazioni stallandosi a un valore fisso, questo perché aggiungere epoche non aiuterà la rete ad aggiornare i pesi in modo migliore anzi favorirà il problema di un sovrastimamento sui dati di train, quini la rete imparerà lentamente a distinguere in modo migliore le immagini sul dataset di allenamento ma non quello sul dataset di validazione.

3.4.2 Il problema di Overfitting e Underfitting

Durante la progettazione, un'idea che può venire in mette è quella, di aumentare il numero di neuroni, questo consentirebbe alla rete di conoscere molte più caratteristiche e si potrebbe tradurre in un aumento esponenziali delle prestazioni della rete.
In parte questo sembra essere vero, aumentando il numero di neuroni i grafici sono molto migliorati e più performarti in termini di accuracy score e loss score raggiungendo valori molto più elevati rispetto a quelli precedenti.
Vedendo già dai grafici sottostanti si può notare come durante la fase di training con un impiego di un numero elevato di neuroni per lo strato dense (sull'ordine dei `1024`) si raggiunge `82%` nella traning accuracy mentre la validation accuracy si stabilizza sotto il `75%`. Lo stesso concetto vale per i valori di loss.
Questo altro dislivello tra i valori viene chiamato overfitting, cosa significa, in statistica e in informatica, si parla di sovrastimamento, quando un modello statistico molto complesso si adatta troppo bene ai dati osservati (l'immagine dei tumori) perché ha un numero eccessivo di parametri rispetto al numero di osservazioni.

Un modello assurdo e sbagliato può adattarsi perfettamente se è abbastanza complesso rispetto alla quantità di dati disponibili.

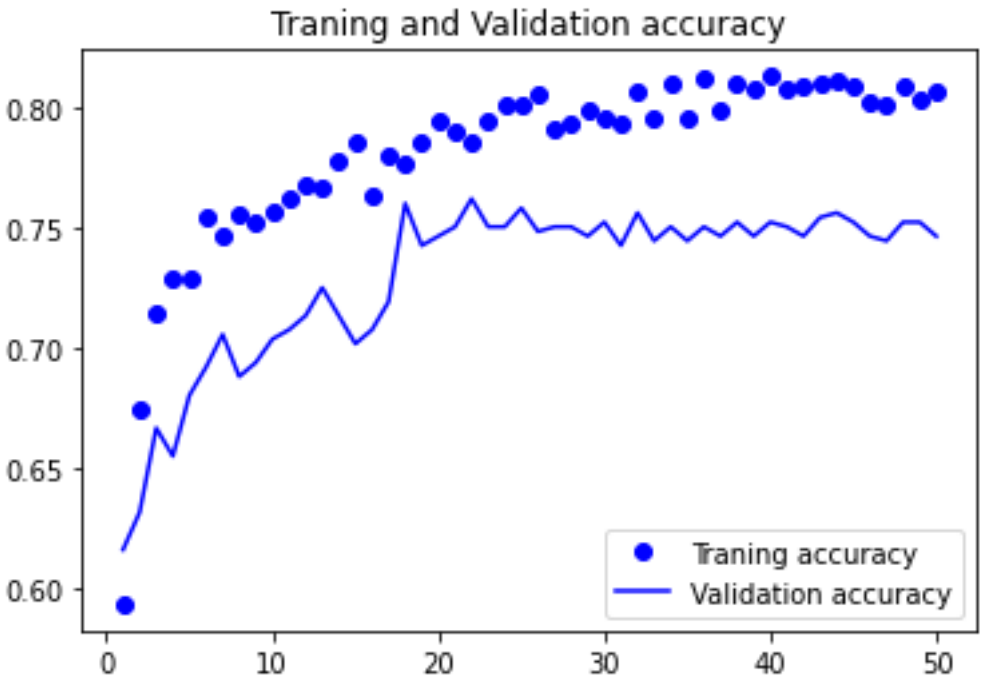


Figura 24 Grafico accuracy con 1024 neuroni layer dense

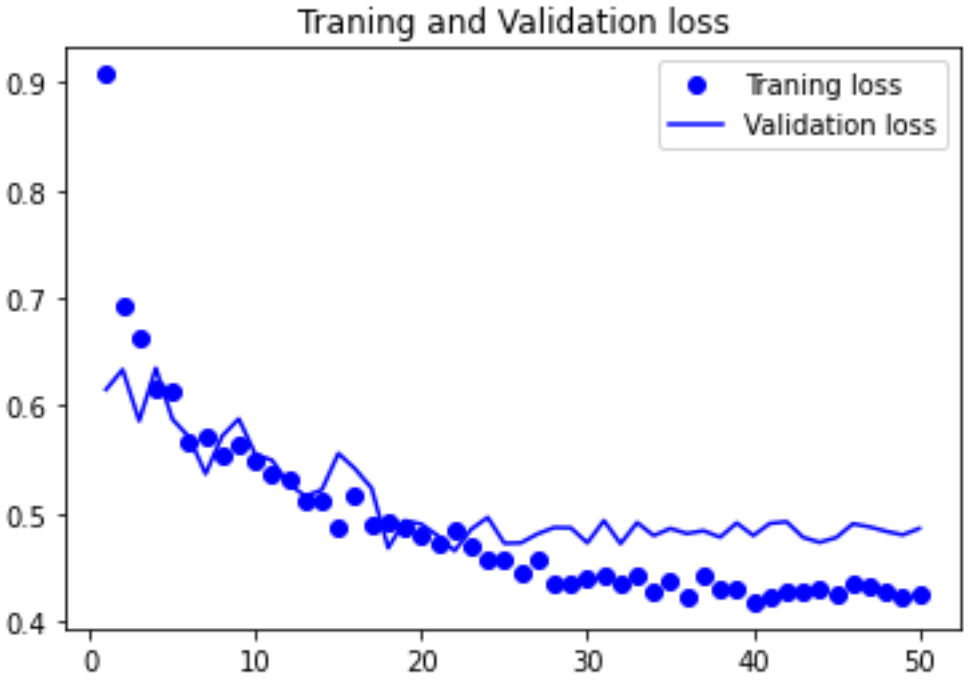


Figura 25 Grafico loss con 1024 neuroni layer dense

```
Media degli scores nella validazione K-Fold:
> Accuracy: 74.326 (+- 0.934)
> Loss: 0.502
```

Un'altra cosa che si nota è il fatto che pur aumentando il numero di neuroni per quanto si incappa in un problema di overfitting la rete in fase di convalida K-fold ha fatto degli score migliori rispetto ai precedenti, questo se pur velato indica un problema di underfitting, ci sono pochi parametri nel modello e un'elevata discrepanza nella classificazione, il processo di apprendimento probabilmente, è troppo semplice.
Per ovviare al problema si diminuiscono i neuroni e si aumenta il modello con uno stack di Conv2D e Maxpooling2D.
Per finire modifichiamo la nostra rete con tutte le costatazioni che sono state fatte:

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 224, 224, 64)	1792
max_pooling2d_12 (MaxPooling	(None, 112, 112, 64)	0
dropout_12 (Dropout)	(None, 112, 112, 64)	0
conv2d_13 (Conv2D)	(None, 112, 112, 64)	36928
max_pooling2d_13 (MaxPooling	(None, 56, 56, 64)	0
dropout_13 (Dropout)	(None, 56, 56, 64)	0
conv2d_14 (Conv2D)	(None, 56, 56, 64)	36928
max_pooling2d_14 (MaxPooling	(None, 28, 28, 64)	0
dropout_14 (Dropout)	(None, 28, 28, 64)	0
flatten_6 (Flatten)	(None, 50176)	0
dense_12 (Dense)	(None, 512)	25690624
dense_13 (Dense)	(None, 2)	1026

Total params: 25,767,298

Trainable params: 25,767,298
Non-trainable params: 0

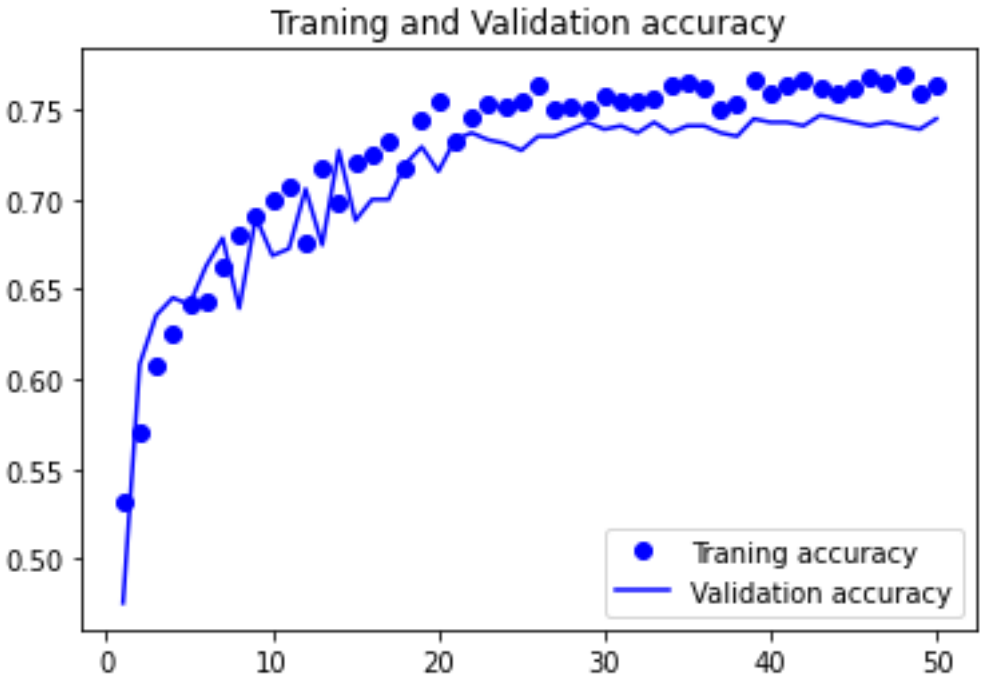


Figura 26 Grafico modello finale accuracy con Adam

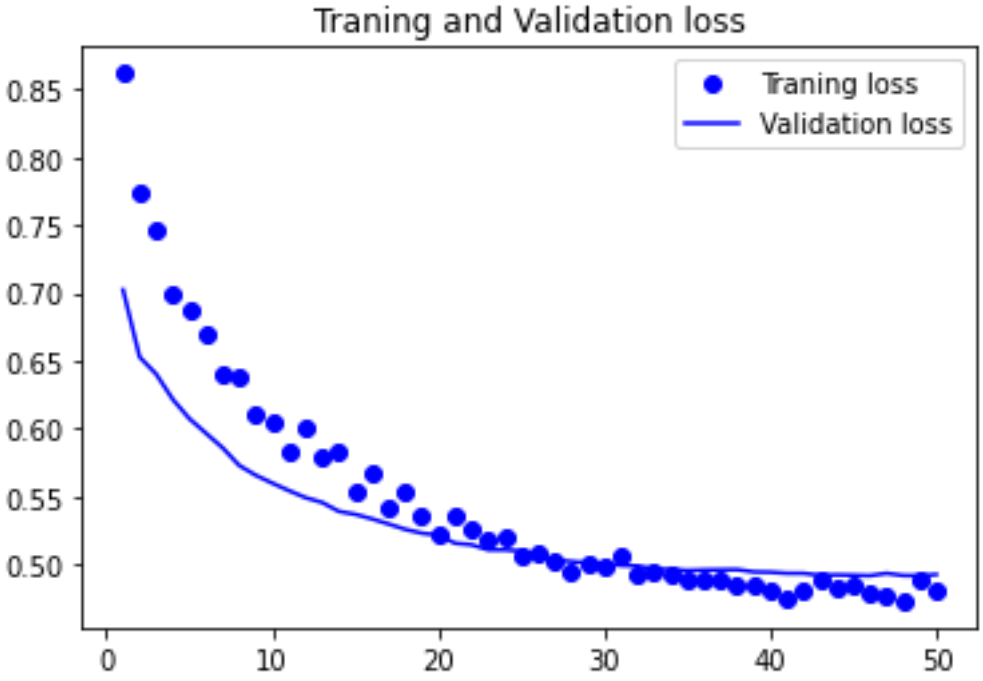


Figura 27 Grafico modello finale loss con Adam

```
> Fold 1 - Loss: 0.498 - Accuracy: 74.883%
> Fold 2 - Loss: 0.478 - Accuracy: 77.497%
> Fold 3 - Loss: 0.489 - Accuracy: 75.504%
Media degli scores per tutte le folds:
> Accuracy: 75.961 (+- 1.115)
> Loss: 0.488
```

Il modello ora creato risulta molto bilanciato sia in termini di coerenza di valori in fase di allenamento e validazione, la discrepanza tra i valori di train e validazione è quasi annullata. Per il grafico di loss si soffre ancora un po' di overfitting ma la validazione incrociata K-fold conferma che non è una controtendenza disruttiva della rete[14].

4 Generative adversarial Network

La Rete Generativa Avversaria[15] o anche Generative adversarial network (GAN), è un metodo che si basa sull'addestramento in maniera competitiva di due reti:

- la rete generativa che si occuperà di generare un determinato tipo di dato che può essere un'immagine un testo.
- la rete discriminante che si occuperà di riconoscere se quel determinato tipo di dato creato è vero o falso.

[14] (https://towardsdatascience.com/dont-overfit-how-to-prevent-overfitting-in-your-deep-learning-models-63274e552323)
15 (https://atcold.github.io/pytorch-Deep-Learning/it/week09/09-3/)

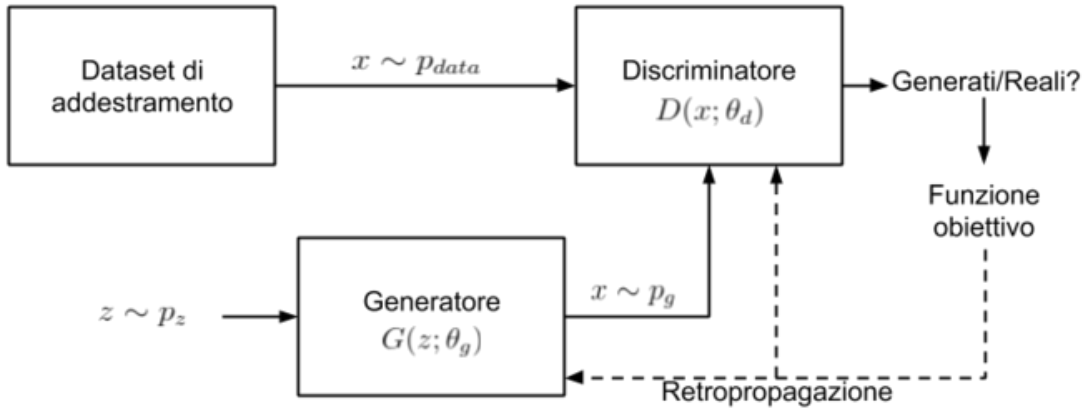


Figura 28 Semplice modello di una generale rete GAN

4.1 DCGAN

Nel caso specifico ci si occuperà di una particolare tipologia di GAN per la generazione di immagini chiamata Deep Convolutional Generative Adversarial Network o anche DCGAN.

Essa sarà sempre costituita da un Generatore G e un discriminatore D.

Il comportamento è analogo, il generatore crea un'immagine il discriminatore giudica se l'immagine generata è vera o falsa e su questo risultato vengono aggiornati i pesi delle due reti, un'immagine inoltre, viene classificata come falsificata non solo quando si discosta troppo dall'immagine di base, ma anche quando è perfetta, cioè se il generatore dovesse semplicemente prendere un valore medio da tutti i dati e crearne uno nuovo, la creazione meccanica sarebbe evidente.

Quindi il discriminatore filtra anche i dati che non hanno un effetto naturale. Entrambe le reti cercano di svolgere il proprio ruolo in competizione, quando il discriminatore rileva il set di dati falso, restituisce i dati.

In questo caso la rete generativa non è ancora abbastanza efficace e deve quindi continuare ad apprendere. Allo stesso tempo, però, anche il discriminatore ha imparato, poiché le due reti neurali si addestrano a vicenda, ci stiamo confrontando con un sistema di deep learning antagonista.

Il generatore cerca di creare insiemi dei dati che sembrino così reali, da essere classificati come tali dal discriminatore.

Il discriminatore, invece, cerca di analizzare e comprendere gli esempi reali in modo così preciso, che i dati falsificati non abbiano alcuna possibilità di essere identificati come reali.

Un' altro elemento importante quando si fa riferimento alle reti DCGAN è quello di riuscire a creare una competizione equilibrata tra le due reti.

Se una delle due reti è di molto superiore all'altra nella fase di apprendimento, il sistema collassa, si può incappare spesso in questo genere di problemi e non è semplice correggerli, ad esempio se il generatore risulterà nettamente superiore al discriminatore, quest'ultimo classificherà tutti i dati falsificati come autentici, se invece fosse il discriminatore a essere nettamente superiore, classificherà tutti i dati del generatore come falsi, in questo caso le reti arriveranno a uno stallo e non riusciranno più a addestrarsi.

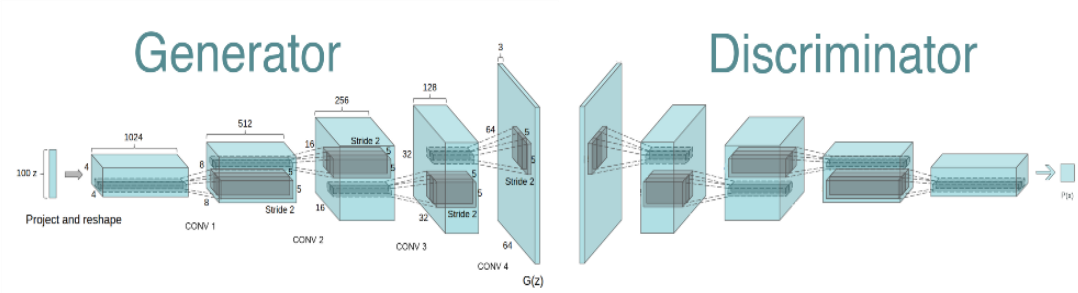


Figura 29 Modello di una DCGAN a sx Generatore a dx Discriminatore

Eviterò di inserire di nuovo tutti i passaggi sulla normalizzazione dei dati visto che risultano essere dei passaggi analoghi a quelli utilizzati per le Convolutional Neural Network, in quanto ci si trova a che fare sempre con le immagini di melanomi benigni e maligni[16].

[16] (https://www.tensorflow.org/tutorials/generative/dcgan)

4.2 Modello Generatore

La prima cosa che si fa è quella di creare il generatore, questo modello avrà il compito di generare le immagini, sulla base dei pesi aggiornati durante l'addestramento.
La funzione sequential di Keras ci permetterà di mettere in concatenazione in modo semplice tutti i layer grazie alla funzione add.
Le modalità con la quali si tende a costruire un generatore nella DCGAN, sono particolari e hanno delle regole non scritte che però devono essere eseguite per avere un buon bilanciamento della rete, cioè la rete deve iniziare con un livello Dense che prende un seme (rumore casuale) come input, quindi sovra campiona più volte fino a raggiungere la dimensione dell'immagine desiderata.
Il sovra campionamento sarà fatto dal due layer LeakyReLU e un layer di Conv2DTraspose, per capire meglio come funziona basta vedere l'immagine [29], partendo dal layer dense, eseguendo vari reshaping si torna alla forma dell'immagine desiderata.
Il blocco di layer utilizzati è:

- `[LeakyReLU]` è un layer di attivazione, ha lo stesso scopo di ReLU inserito all'interno di activation, la differenza sostanziale che LeakyReLU permette la prevenzione della morte della rete. La differenza sostanziale si trova nel fatto che l'output di ReLU è x e per tutti le x > = 0, mentre è 0 per tutti gli altri x. In generale, questo funziona molto bene in molte reti neurali, ma poiché questo rende il modello un pochino più inefficiente, il processo di addestramento tende ad essere influenzato solo dalle caratteristiche nel set di dati che effettivamente contribuiscono al potere decisionale del modello.

  Contrariamente, gli output di LeakyReLU sono piccoli e diversi da zero per tutti le x < 0. In questo modo, si evita la morte della rete neurale.

  Funziona molto bene nelle reti antagoniste perché evita un arresto forzato nel caso che una delle due reti lavori meglio dell'altra, cioè "vinca la competizione"[17].

[17] (https://www.machinecurve.com/index.php/2019/11/12/using-leaky-relu-with-keras/)

- `[Conv2DTranspose]` Strato di convoluzione trasposto è anche definito come operazione di de-convoluzione, infatti si comporta in modo opposto all'operazione di convoluzione vista nella CNN (si parte da un layer dense per andare verso le stesse dimensioni dell'immagine).

  La necessità di utilizzare le convoluzioni trasposte nasce dal desiderio di utilizzare una trasformazione che vada nella direzione opposta di una normale convoluzione.

  Cioè da qualcosa che ha la forma dell'output di una di una convoluzione, a qualcosa che ha la forma del suo input (dimensione dell'immagine (224, 224, 3)), pur mantenendo un modello di connettività compatibile guarda fig[29][18].
- `[BatchNormalization]` Serve per normalizzare i dati in input.

  La normalizzazione batch applica una trasformazione che mantiene l'output medio vicino a `0` e la deviazione standard dell'output vicino a `1`[19].
- `[Reshape]` Layer che permette di fare un reshape del tensore prima di passarlo di nuovo a un layer di convoluzione. Per verificare la correttezza del reshaping si inserisce anche una riga di codice con assert, questa keyword permette di sollevare un errore in caso di un cattivo reshaping altrimenti non succede nulla (comoda durante la realizzazione del modello).
- `[Dense]` Si comporta allo stesso modo delle CNN, la cosa interessante è il fatto che in input_shape, invece di inserire la nostra immagine inseriamo lo shape del rumore casuale inizialmente generato dal generatore.

Il blocco iniziale si presenterà in questo modo, i blocchi successivi non presenteranno il layer dense:

```
model.add(layers.Dense(units, use_bias, input_shape))
model.add(layers.BatchNormalization())
model.add(layers.LeakyReLU())
model.add(layers.Reshape(reshape))
assert model.output_shape == (None, (reshape))
```

[18] (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose)

[19] (https://keras.io/api/layers/normalization_layers/batch_normalization/)

4.3 Attivazione e Generazione immagine

A differenza dei layer intermedi l'ultimo layer di attivazione non sarà più con LeakyReLU, ma presenterà l'attivazione 'Than'[20].

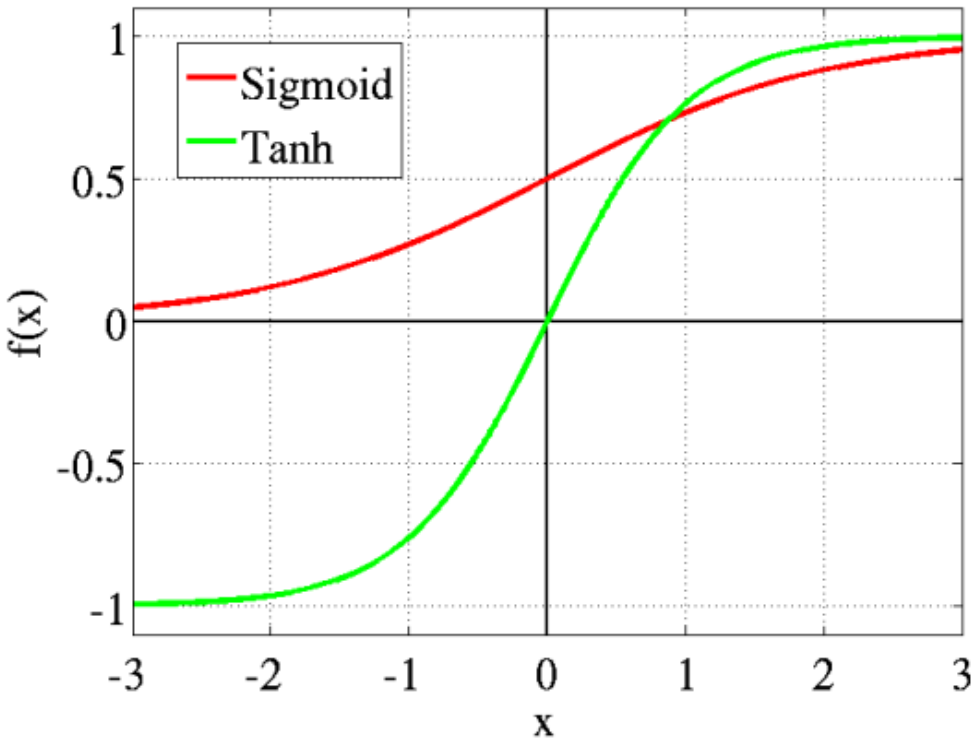


Figura 30 Funzione di attivazione Than vs Sigmoid

Tanh funziona come la funzione Sigmoid, l'intervallo su cui è definita va da `(-1 a 1)`, è letteralmente la funzione tangente in trigonometria, viene utilizzata principalmente per la classificazione tra due classi.

```
model.add(layers.Conv2DTranspose(filter, kernel_size, strides, padding, use_bias, activation))
```

20 (https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6)

Una vota completato il modello si testa il funzionamento del generatore generando un'immagine con rumore completamente casuale, in una distribuzione normale di valori da `1` a `100`.

```
generator = make_generator_model_tensorflow()
noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)
```

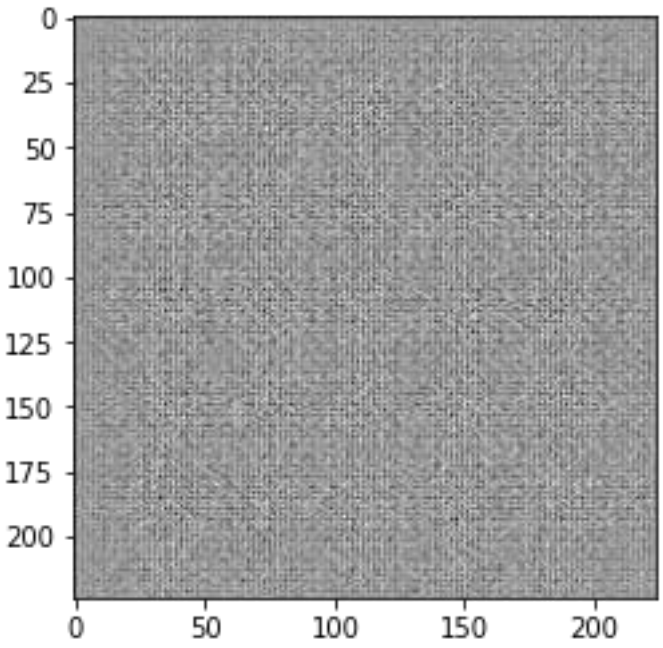


Figura 31 Immagine generata dal rumore causale iniziale

4.4 Discriminatore

Il discriminatore della DCGAN non è altro che una Convolutional Neural Network, con la differenza che invece di utilizzare un'attivazione ReLU si utilizza di nuovo, LeakyReLU.
I vari blocchi del modello che utilizzo sono:

```
model.add(layers.Conv2D(64, (5, 5), strides=(2, 2),
padding='same', input_shape=[224,224,3],
kernel_initializer=weight_initializer))
model.add(layers.LeakyReLU())
model.add(layers.Dropout(0.5))
```

4.5 Perdita Ottimizzatori

Si utilizza l'entropia incrociata binaria (Binary Cross Entropy, BCE) fra l'obiettivo e l'output, che vogliamo generare per entrambi i modelli, più in particolare:

- Per il discriminatore, si quantifica la capacità di quest'ultimo di distinguere le immagini reali da quelle false. Si confrontano le previsioni del discriminatore su immagini reali con un array di 1 e le previsioni del discriminatore su immagini false (generate) con un array di 0, poi si fa la somma dei valori di loss generati dagli output reali e quelli falsi.

```
real_loss = cross_entropy(tf.ones_like(real_output),
real_output)
fake_loss = cross_entropy(tf.zeros_like(fake_output),
fake_output)
total_loss = real_loss + fake_loss
```

- Mentre la perdita del generatore quantifica la capacità di ingannare il discriminatore, se il generatore funziona bene, il discriminatore classificherà le immagini false come reali, si confronteranno le decisioni dei discriminatori sulle immagini generate con un array di 1, il metodo di loss per il generatore è definito come la 2 riga dello snippet di codice precedente.

Il passo successivo è quello di definire gli ottimizzatori delle due reti, si utilizza la funzione Adam con un learning rate di (0.0002) in alternativa si può usare anche 1e-4 che è suggerito da molti paper nel campo delle reti generative.

```
optimizer = tf.keras.optimizers.Adam(0.0002)
```

4.6 Addestramento

Il ciclo di addestramento inizia con il generatore che riceve un rumore casuale come input, questo rumore casuale iniziale viene utilizzato come seme per produrre una prima immagine.

A questo punto il discriminatore classifica immagini reali (tratte dal set di addestramento) e poi le immagini false (prodotte dal generatore).

Le varie loss vengono calcolate per ciascun modello e i gradienti vengono utilizzati per aggiornare i pesi del generatore e il discriminatore, tutta questa iterazione viene fatta per ogni epoca di allenamento.

Nei paragrafi a venire sarà mostrato un grafico, esso rappresenterà i valori di loss epoca per epoca durante la fase di addestramento tra il Discriminatore (giallo) e il Generatore (blu), a seguire saranno presenti due valori che indicano la media dei punteggi di loss dei due modelli.

Inoltre, saranno visibili due immagini:

- Una in bianco e nero, queste saranno le immagini generete durante l'ultima epoca di addestramento.
- E un set di foto invece generato salvando il modello e i suoi pesi, dopo di che le immagini vengono create e trasformate in formato RGB, e mostrate tramite la libreria matpoltlib.

Quello che si introdurrà nei prossimi paragrafi sono i vari passaggi con la quale si raggiunge un certo livello di dettaglio nella creazione di immagini di melanomi che siano appetibili all'occhio umano.

Il modo di procedere per allenare le reti GAN non è semplice e nemmeno analitico, molte prove che sono state svolte non sono state inserite in quanto non portavano significativamente ad alcun risultato, o in alcuni casi rompevano quell'equilibrio che si era riusciti a raggiungere negli allenamenti precedenti.

Quello che farò è paragrafo dopo paragrafo elencare le varie modiche che ho apportato alla rete di partenza per migliorarne le prestazioni, mostrando sempre i grafici e le

immagini generate, per avere un riscontro visivo di tutto ciò che succede durante un allenamento di una DCGAN.

4.6.1 Layer dei Modelli

La prima prova è stata fatta utilizzando due reti simmetriche cioè con lo stesso numero di layer convoluzionali e deconvoluzionali per il generatore e il discriminatore:

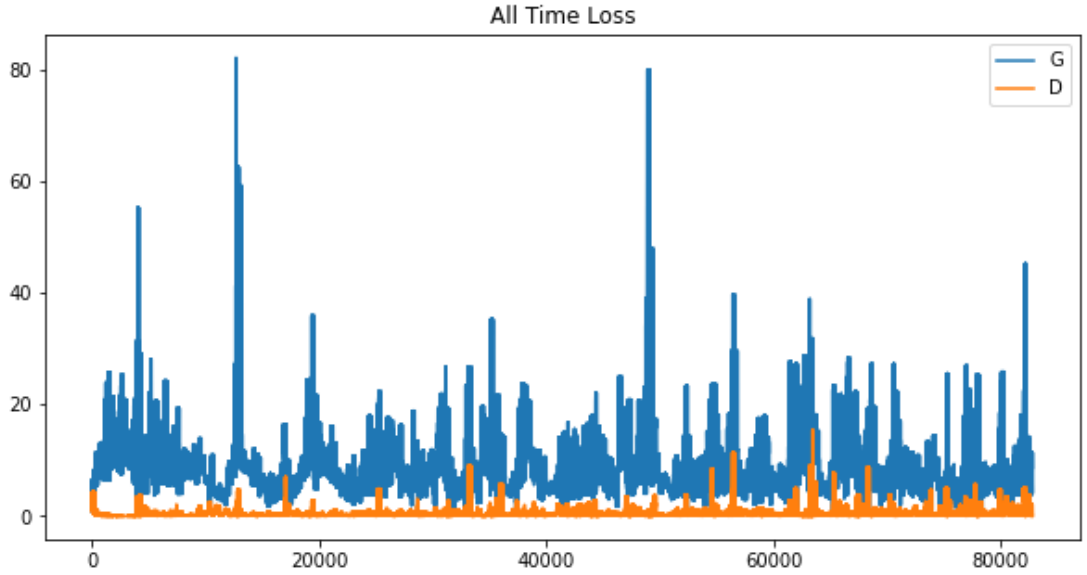


Figura 32 Grafico Disc e Loss con stesso numero Layer convoluzionali e deconvoluzionali

```
Gen_loss mean:  19.66
Disc_loss mean:  0.18
```

Come si evince dal grafico, la DCGAN non si sta allenando bene, infatti, il discriminatore risulta troppo più bravo nel distinguere le immagini generate, il discriminatore ha troppi parametri dal quale imparare, o le distribuzioni fornite dalla discesa del gradiente non aiutano la rete generatrice a imparare nulla di buono.
Un altro fattore è l'instabilità della rete ci sono alcune epoche in cui i picchi raggiungo Gen_loss pari ad 80.

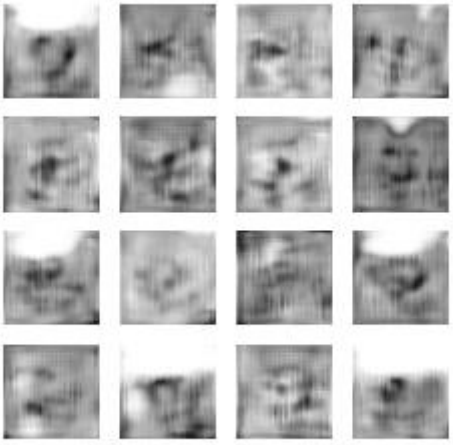


Figura 33 Immagini generate all'ultima epoca di addestramento con stessi layer convoluzionali e deconvoluzionali

Queste sono le foto dei tumori alla nostra ultima epoca di addestramento si nota abbastanza bene come la DCGAN non si stia allenando bene, infatti ci sono molte striature nelle foto, questo è dovuto al fatto che il generatore non sa bene cosa generare a posto di quegli spazi.

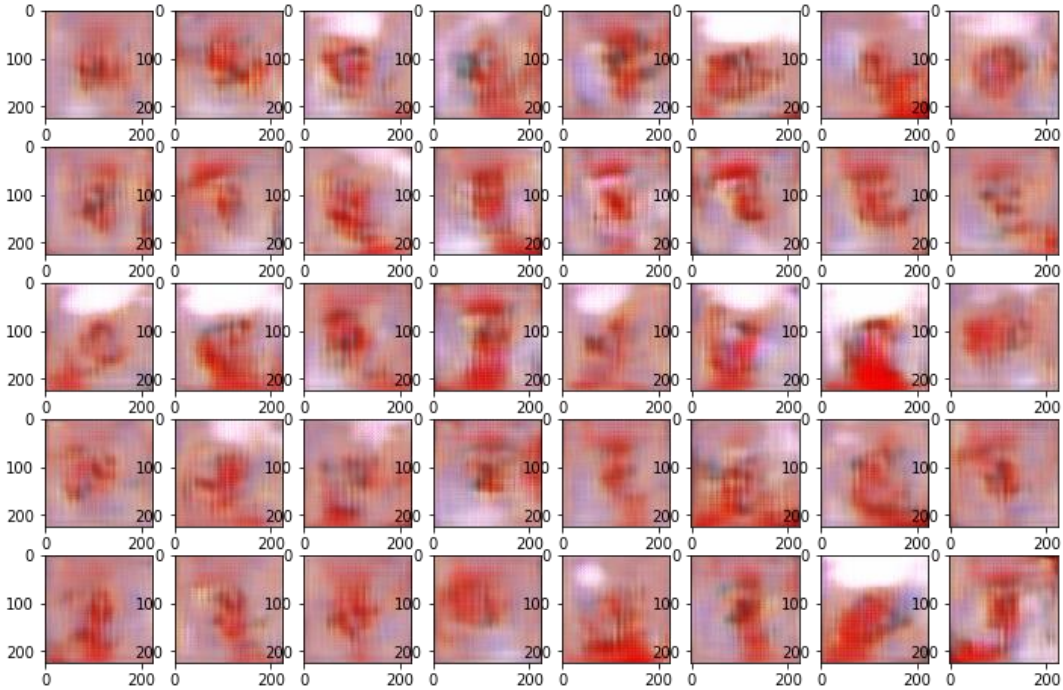


Figura 34 Visualizzazione immagini con stessi layer convoluzionali e deconvoluzionali

Come seconda prova l'obiettivo era quello di sistemare l'instabilità della rete e il fatto che il discriminatore era estremamente più potente del generatore, quindi, si prova a togliere uno stack di layer convoluzionale al discriminatore (Per stack si intende un blocco formato da Conv2D, Maxpooling, Dropout), così da renderlo "meno potente" nel distinguere le immagini reali da quelle false:

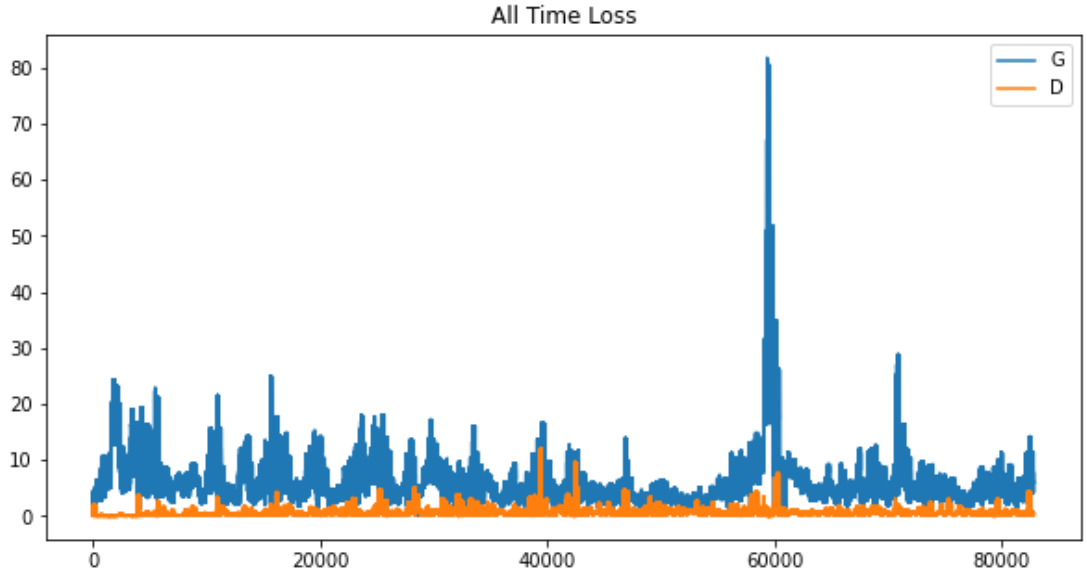


Figura 35 Grafico Disc e Loss togliendo uno stack di layer al discriminatore

```
Gen_loss mean:  7.94
Disc_loss mean:  0.55
```

In questo caso la competizione sembra essere un pochino più bilanciata ma comunque c'è ancora un grande divario tra i valori di loss, questo sta ad indicare che non c'è ancora un buon bilanciamento delle due reti, la competizione è ancora impari, e inoltre per quanto si migliori il valore medio di loss del generatore, ci sono ancora degli altissimi picchi, sempre sintomo di un'instabilità della rete.

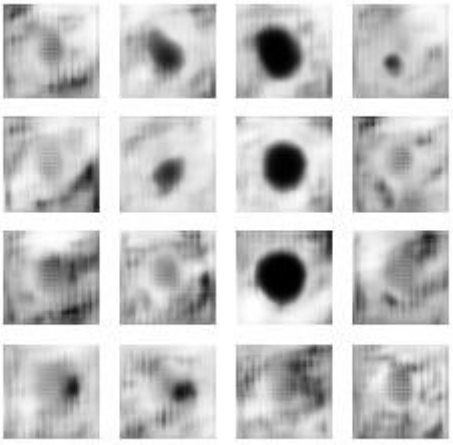


Figura 36 Immagini generate all'ultima epoca di addestramento togliendo uno stack al discriminatore

In questo caso il generatore sembra ad aver imparato a distinguere la forma e a ricrearla.

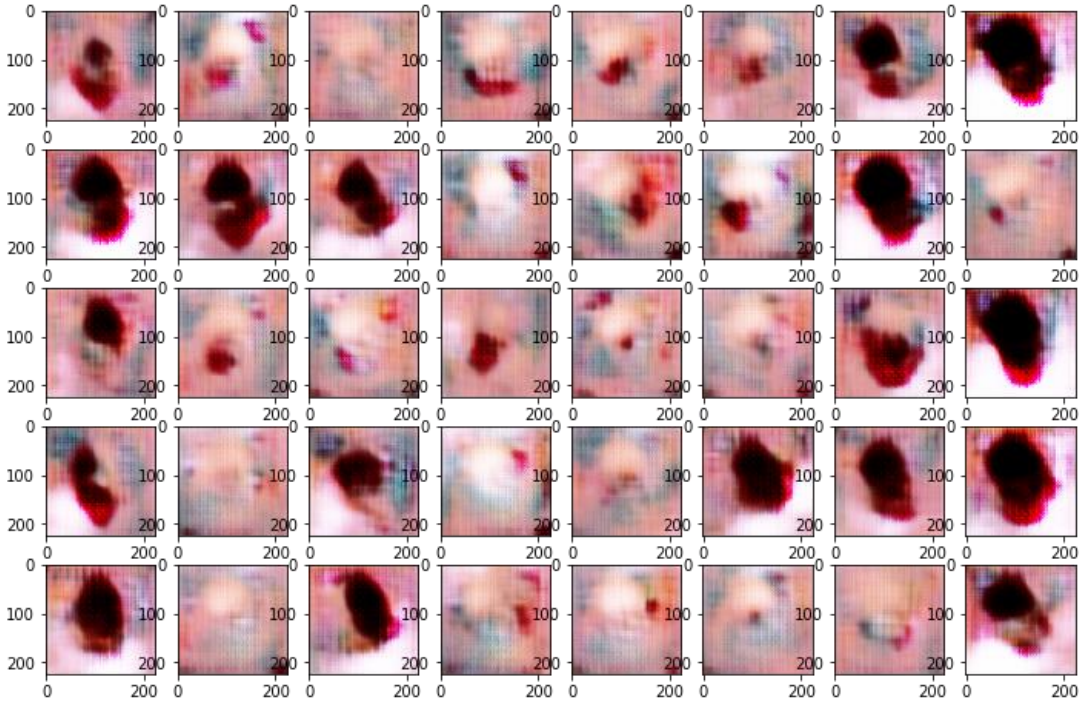


Figura 37 Visualizzazione immagini togliendo uno stack di layer al discriminatore

Come terza prova ho deciso non soltanto di modificare i layer della rete discriminatoria ma andare a diminuire entrambi i layer delle due reti, quindi sia quelli del generatore

che il discriminatore, l'obiettivo era quello soprattutto di migliorare la stabilità della rete:

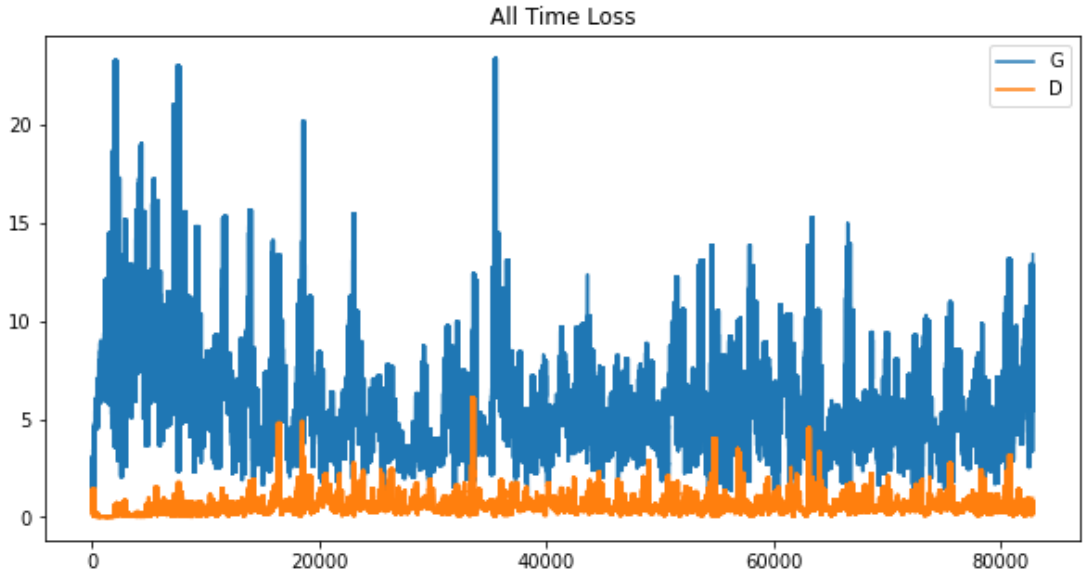


Figura 38 Grafico Disc e Loss abbassando il numero totale di layer della rete

```
Gen_loss mean:  6.16
Disc_loss mean:  0.63
```

Non sempre nel deep learning dare molti parametri alla rete dà la qualifica di migliore, abbassando i parametri totali dalla quale le due reti debbono imparare, l'allenamento seppur una fase iniziale molto instabile è migliore.

Si nota come la rete ora non subisca più picchi di loss per il generatore esorbitanti, la differenza media tra il generatore il discriminatore inizia lentamente ad avvicinarsi e a rimanere stabile fino alla fine dell'allenamento.

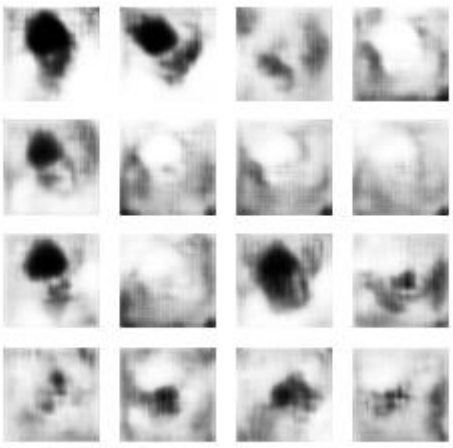


Figura 39 Immagini generate all'ultima epoca di allenamento abbassando il numero totale di layer della rete

Le immagini generate in questo caso sembrano aver imparato molti più pattern da distinguere rispetto a quelle create in precedenza.

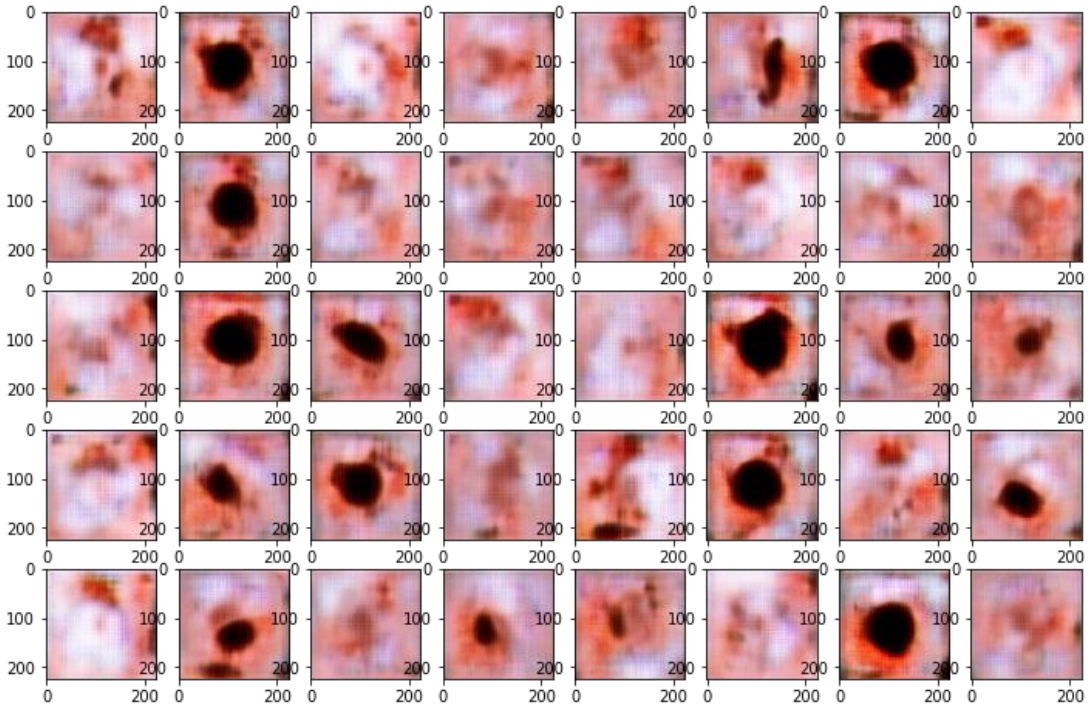


Figura 40 Visualizzazione immagini abbassando il numero totale di layer della rete

4.6.2 Inizializzazione dei pesi

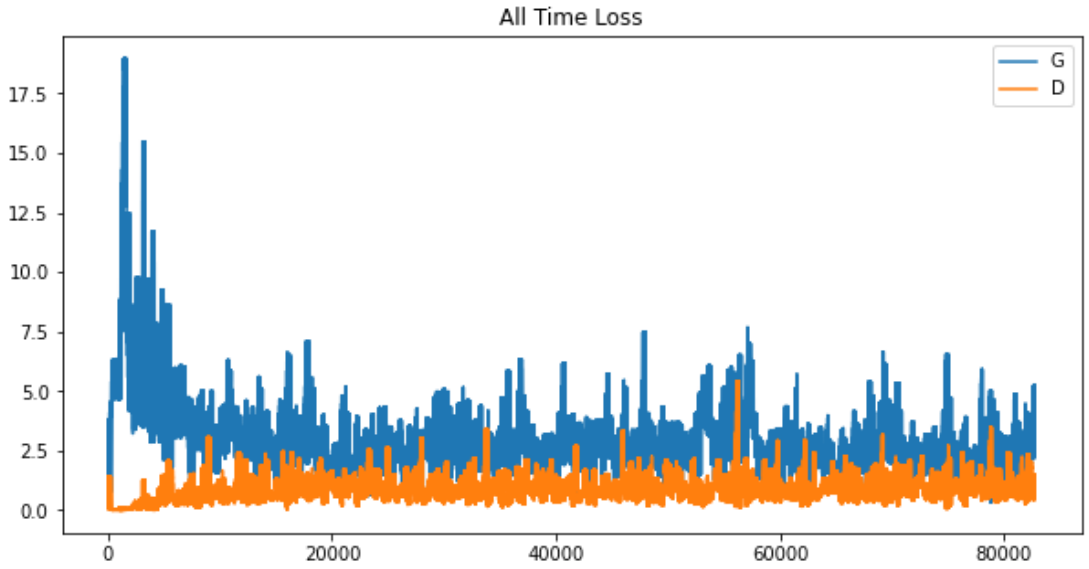
La rete generata in precedenza si contraddistinguono molto per la loro instabilità, le oscillazioni tra un'epoca e l'altra sono molto nette esistono vari metodi per evitare questi problemi uno di questi è l'inizializzazione del peso.

Esso è piuttosto importante per addestrare reti generative che risultino stabili, in primo luogo, i pesi del modello devono essere centrati sugli intorni dello zero con un leggero aumento della deviazione di 0,02 punti percentuale (2%), questo stabilizza sia Discriminatore che Generatore durante l'allenamento e impedisce ai gradienti del modello di svanire o esplodere, durante il processo di allenamento, l'inizializzazione dei pesi viene implementata per ogni layer Conv2DTranspose per il generatore, e Conv2D per il discriminatore[21].

Per implementarla basterà utilizzare la funzione di Keras, dove mean sta per la media e stddev per la deviazione standard:

weight_initializer = tf.keras.initializers.TruncatedNormal(mean=0.0, stddev=0.02)

Il processo di addestramento varia così:



[21] (https://keras.io/api/layers/initializers/)

Figura 41 Grafico Disc e Loss con inizializzazione dei pesi

```
Gen_loss mean:  5.88
Disc_loss mean:  0.74
```

La rete è migliorata, a parte le prime epoche dove ci sono forte oscillazioni, non supera mai i `7.5` punti loss per il generatore e 1 per il discriminatore.
Il fattore di una competizione equilibrata però non è ancora presente, e soprattutto per quanto si è riusciti a rendere l'allenamento più stabile non sembra essere ancora sufficiente.

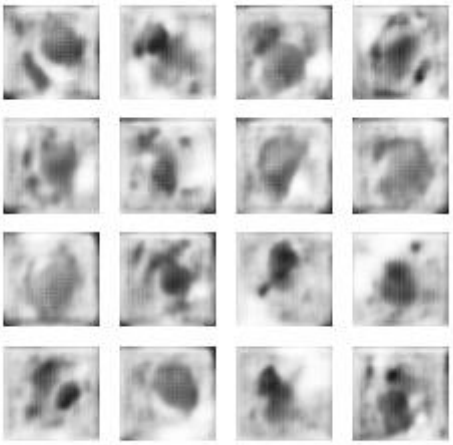


Figura 42 Immagini generate all'ultima epoca di allenamento con inizializzazione dei pesi

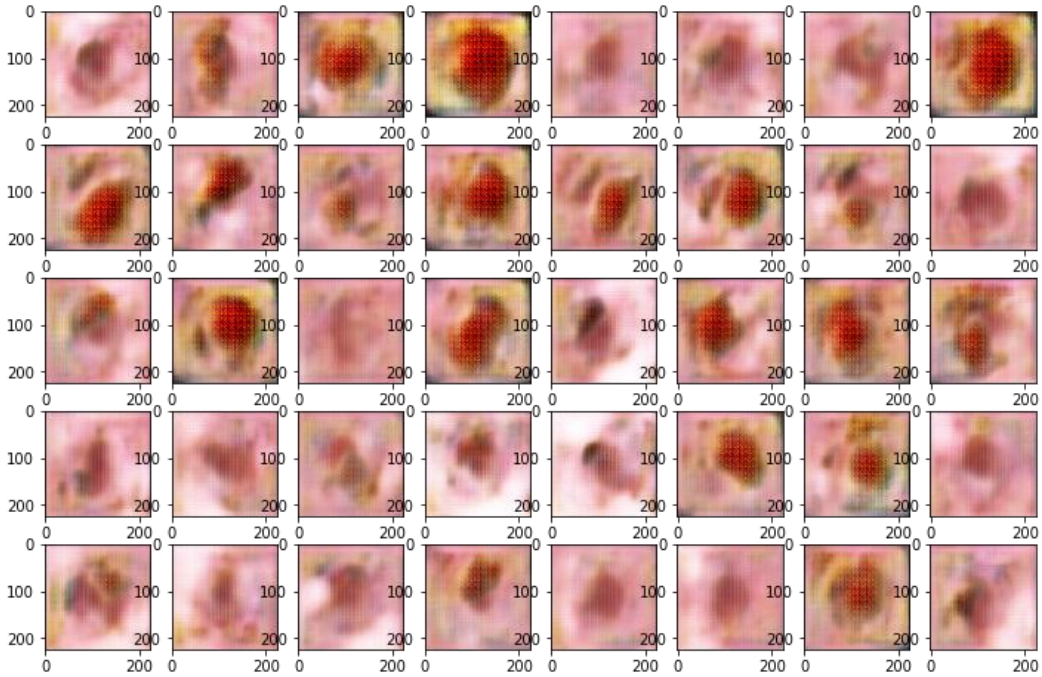


Figura 43 Visualizzazione immagini con inizializzazione dei pesi

4.6.3 Label Smoothing e Rumore di Istanza

Un altro metodo di regolarizzazione che può essere applicato durante l'allenamento è chiamato Label Smoothing.

Ciò che fa è che essenzialmente è impedire al discriminatore di essere troppo sicuro di sé o poco sicuro nelle sue previsioni, se il discriminatore diventa troppo certo che ci sia un melanoma ad esempio benigno della pelle, in un'immagine specifica, il generatore può sfruttare questo fatto e iniziare continuamente a generare solo immagini di quel determinato tipo e a sua volta smettere di migliorare.

Questo tipo di comportamento si può evitare, impostando le etichette delle immagini vere e generate in modo che siano comprese nell'intervallo tra 0 e 0.3 [0, 0.3] per le classi negative e 0.7 e 1 [0.7, 1] per quelle positive, ciò impedirà alle probabilità complessive di avvicinarsi di molto alle due soglie.

Un altro problema che si nota dalle immagini è quanto le distribuzioni con le quali vengono generate sono particolarmente strane, con queste striature e forme ovali, non ben definite, inoltre le reti sono molti instabili avendo svariati picchi.

Per stabilizzare questo tipo di comportamento si usa una tecnica chiamata rumore di istanza.

Questa "trucco" serve per stabilizzare le reti, esso consiste nell'inserire del rumore casuale tra le lables, aggiungendo una piccola quantità di errore (intorno al 0.05) quindi relativamente piccolo.

Si tende ad evitare che una volta trovata una distribuzione vera, il discriminatore sia estremamente incline all'overfitting, e quindi quasi sempre ad essere ottimale, e che fornisca dei valori del gradiente non utili al generatore per creare delle immagini che riescano ad ingannarlo, questo come nel nostro caso è denotato da una forte instabilità della rete[22].

Aggiungendo questi due accorgimenti vediamo come varia l'andamento della nostra rete per 400 epoche:

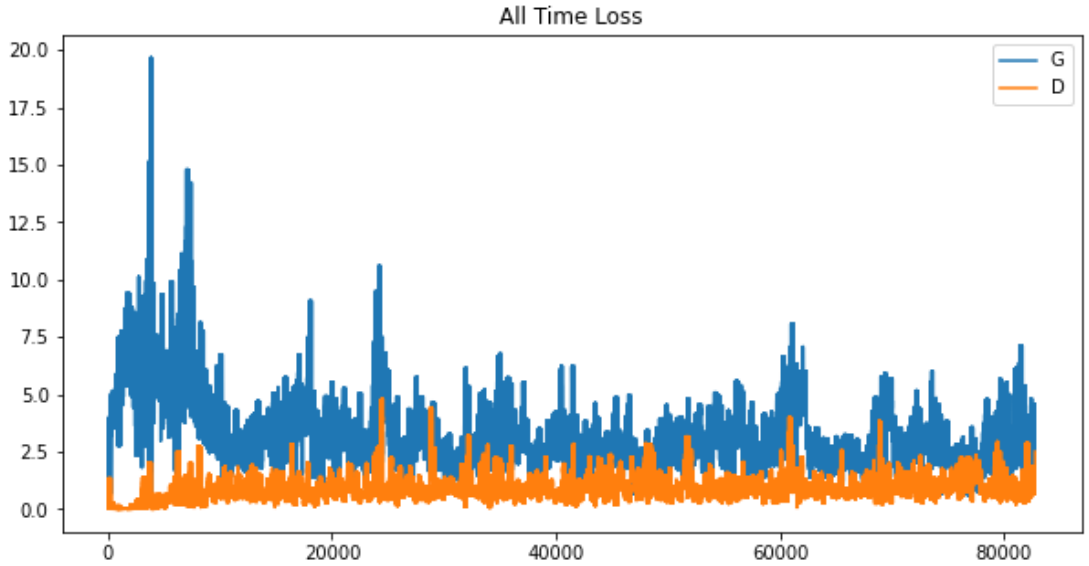


Figura 44 Grafico Disc e Loss con Lables Smoothing e noise

```
Gen_loss mean:  3.15
Disc_loss mean:  0.82
```

I valori medi ma soprattutto quello del generatore, si sono abbassati, la cosa che si nota guardando i grafici precedenti è come, regolarizzando i parametri subiscano vari

[22] (https://www.inference.vc/instance-noise-a-trick-for-stabilising-gan-training/)

miglioramenti sia in termini di Disc_loss: aumentando sempre di più (aumentare in questo caso la loss non è un fattore negativo in quanto il discriminatore inizia a far fatica nel distinguere le immagini reali da quelle false, il generatore sta compiendo un buon lavoro), e la Gen_loss inizia a diminuire.
Anche le immagini che vengono generate risultano molto meno rumorose e instabili già a colpo d'occhio, l'ultimo passo è quello di poter allenare la rete per più epoche evitando che diverga.

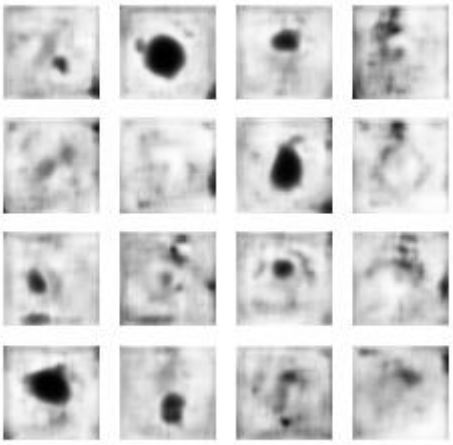


Figura 45 Immagini generate ultima epoca con Lables Smoothing e noise

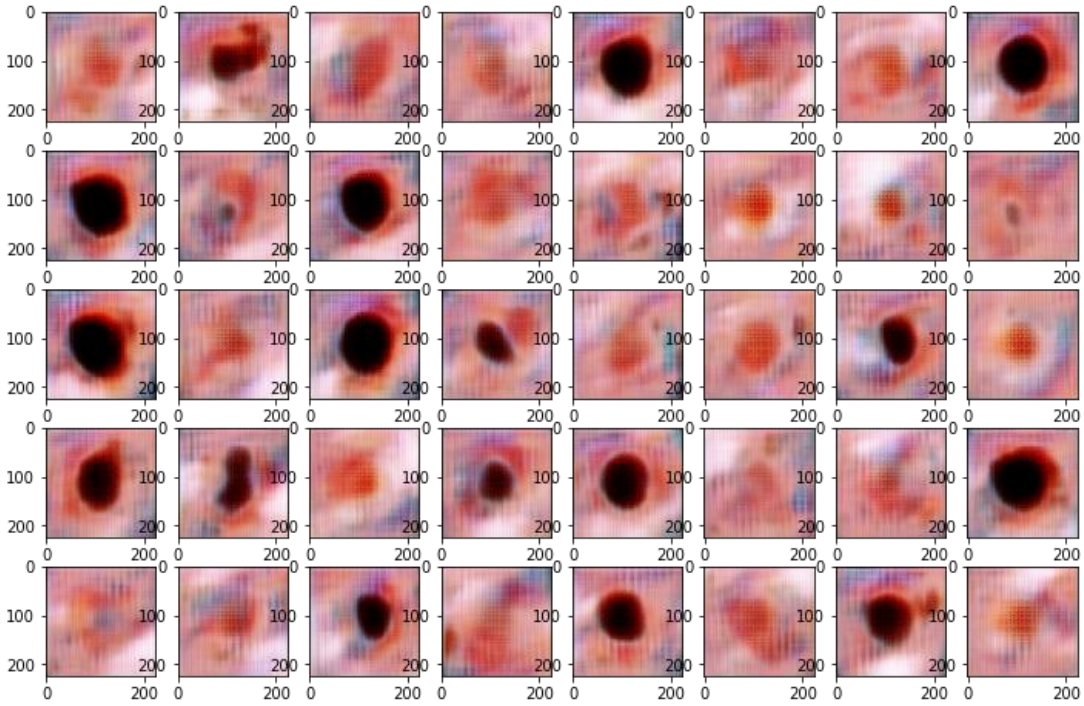


Figura 46 Visualizzazione immagini con Lables Smoothing e noise

4.6.4 Learning Rate Decay

Per evitare che il modello possa imparare dei pattern sempre differenti o che impari troppo bene, dovuto al numero di epoche elevato, si aggiunge una funzione che permette a ogni numero prefissato di epoche di diminuire il tasso di apprendimento della rete, quindi per lunghi allenamenti (superiori alle 500 epoche) si evita di incappare in questo problema.

```
new_lr = tf.compat.v1.train.cosine_decay(new_lr,
min(global_step, decay_steps), decay_steps)
```

Questa funzione applica un tasso di decadimento del coseno a una velocità di apprendimento iniziale fornita, per l'ottimizzatore Adam si è utilizzato un learning rate iniziale pari a 0.0002, che è un valore standard per le reti generative.
Per farla funzionare correttamente, la funzione richiede anche un valore global_step, questo serve per calcolare il tasso di apprendimento decaduto è semplicemente un

contatore delle epoche di addestramento finora fatte. Mentre decay_steps è un valore che definiamo noi, di solito viene scelto sulla base del numero delle epoche per la quale si allena la rete.

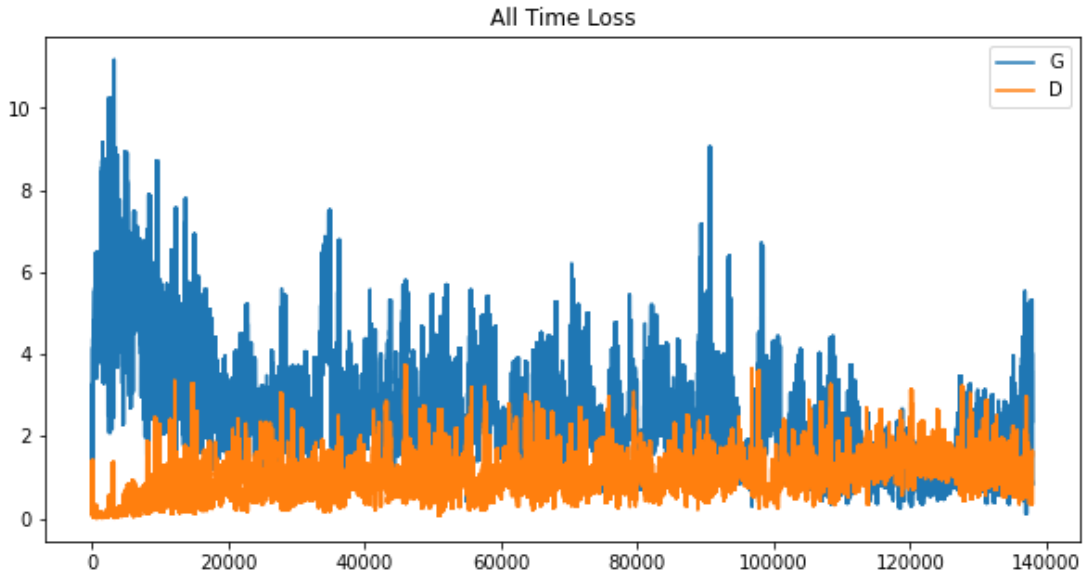
Un ciclo di if inserito nella funzione di train si occuperà ogni numero prestabilito di epoche di aggiornare il learning rate della rete, in new_lr verrà sovrascritto il nuovo learning rate mentre quello presente nella funzione è il learning rate iniziale.

La funzione di minimo si occuperà di decidere di quanto si deve abbassare il learning rate, evitando che la rete con il passare delle epoche annulli completamente il suo tasso di apprendimento, in questo decay_steps funziona da deterrente per questo problema, fungendo da lower bound, inoltre decay_steps si utilizza per identificare il numero di passaggi di decadimento[23].

Il passo successivo una volta calcolato il nuovo learning rate è quello di ridefinire l'ottimizzatore, facendo così avremo un learning rate che decrementa lentamente nel tempo:

```
tf.compat.v1.train.AdamOptimizer(new_lr)
```

Vediamo come varia l'apprendimento:



[23] (https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/cosine_decay)

Figura 47 Grafico Disc e Loss con learning rate reduction

```
Gen_loss mean:  1.96
Disc_loss mean:  0.90
```

A questo punto con la possibilità di aumentare le epoche evitando il problema di creare immagini clone, si può notare come la due reti entrino in competizione.
In questo caso il generatore si sta comportando particolarmente bene e riesce nelle fasi finali ad ingannare il discriminatore.
Questo fattore risulta anche evidente dai valori medi che migliorano sensibilmente per il discriminatore mentre di molto per il generatore.

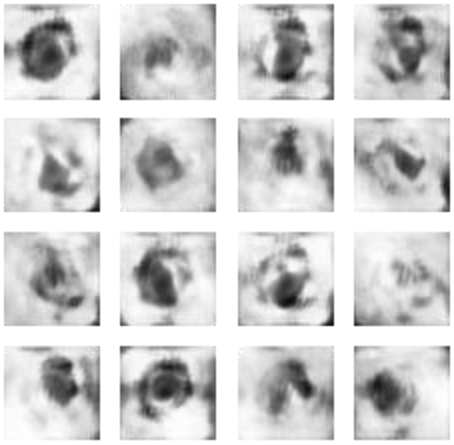


Figura 48 Immagini generate all'ultima epoca con learning rate decay

Per vedere meglio se il generatore riesce ad ingannare il discriminatore, ho fatto un sotto grafico di `25` epoche nell'intorno dell'epoca `500`:

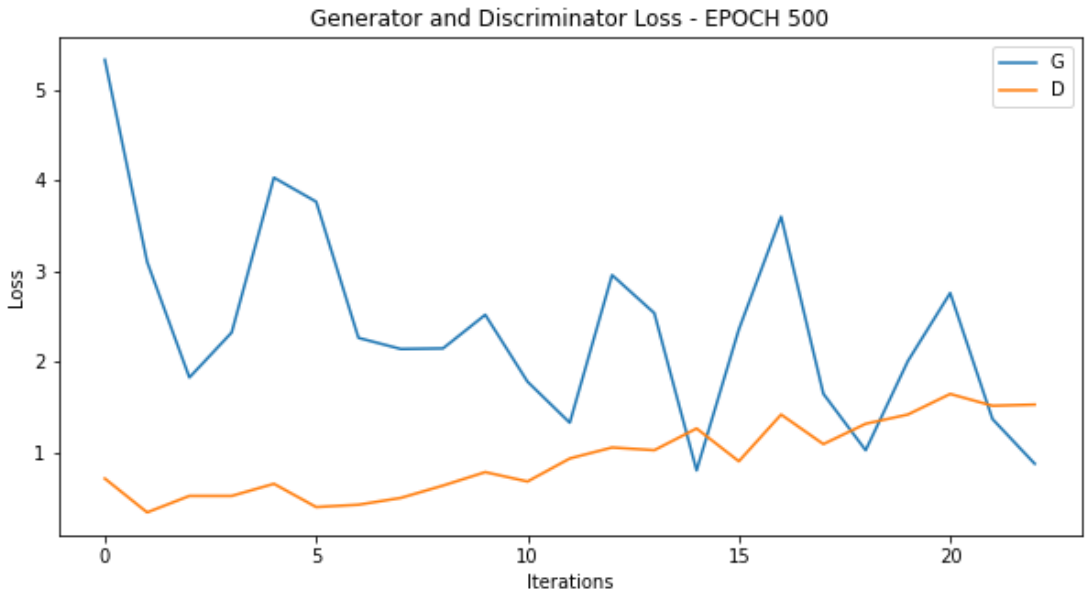


Figura 49 Grafico intermedio per la visualizzazione dei valori di loss delle reti

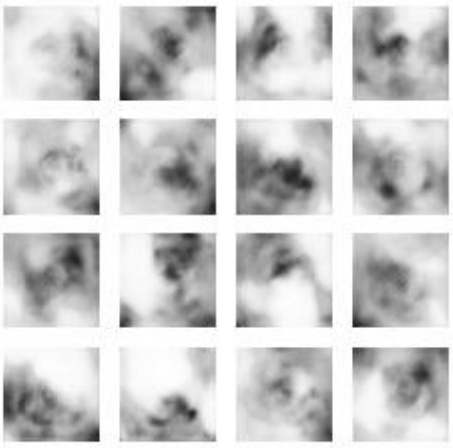


Figura 50 Immagini generate ultima epoca con learning rate decay

Le immagini sottostanti risultano essere più piacevoli e sembrano essere meno affette da strane distribuzioni:

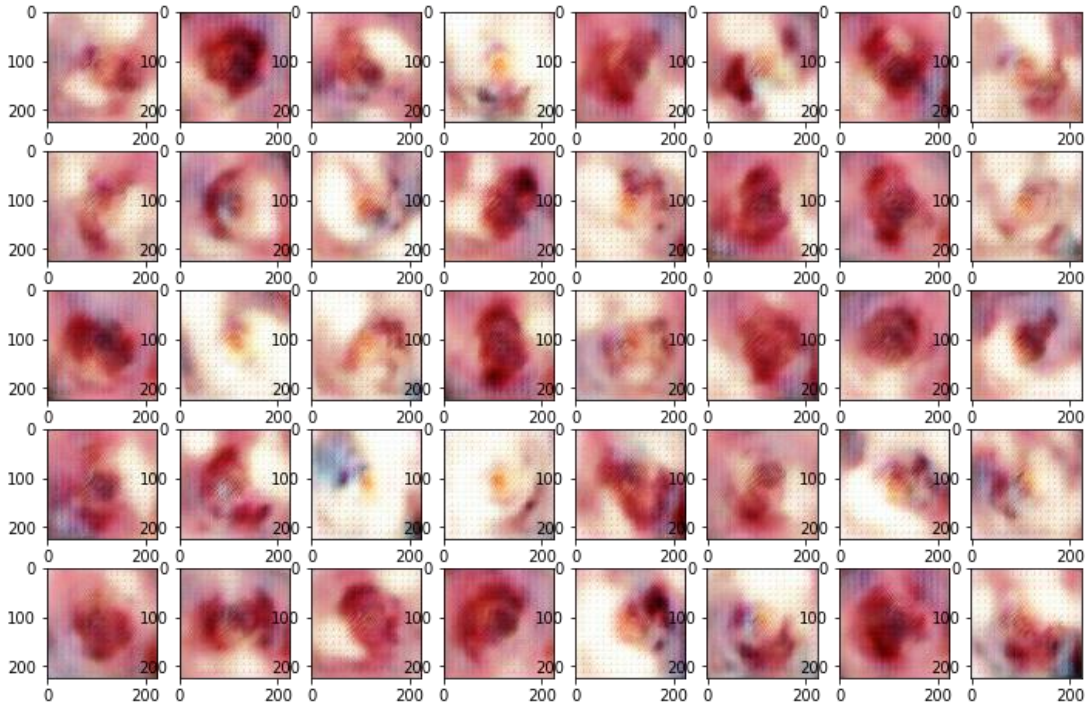


Figura 51 Visualizzazione immagini con learning rate decay

Con l'inserimento del learning rate decay ci possiamo concentrare su allungare i tempi di allenamento per la DCGAN senza rischiare di farla convergere, cioè che una delle due funzioni di loss vada rapidamente verso valori di limite superiore allo 0, nei grafici sottostanti si vede come si allenano diverse reti per svariate epoche.

I valori di epoche sottostanti non risultano reali in quanto per i limiti imposti da Google su l'utilizzo dei fogli Jupiter di Google Colab, non è possibile allenare più di otto ore consecutive una rete su una GPU.

Per questo motivo si utilizza un sistema di checkpoint, si salvano all'interno di una cartella predefinita i vari dati, l'indice dei checkpoint salvati, e il checkpoint stesso.

In questo lavoro ci viene in aiuto la funzione checkpoint di TensorFlow, che permette una gestione intuitiva e rapida del problema, mentre la funzione manager è utile in fase di gestione del numero di checkpoint salvati, il problema che si riscontra non gestendo durante la fase di l'allenamento il numero di checkpoint salvati, è la sovra scrizione in modo scorretto dei nuovi salvataggi a discapito di quelli vecchi[24].

[24] (https://www.tensorflow.org/guide/checkpoint)

```
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint
(generator_optimizer=generator_optimizer,
 discriminator_optimizer=discriminator_optimizer,
 generator=generator,
 discriminator=discriminator)
manager = tf.train.CheckpointManager(checkpoint,
chekpointdir, max_to_keep=1)
```

Questo evita una volta terminata la possibilità di utilizzare le GPU di Google di ricominciare da capo l'addestramento.

Inoltre, si automatizza questo processo inserendo un if all'inizio della fase di train che in caso di detection di salvataggi, riinizia l'allenamento da quel punto.

Questo permette di allenare la rete per molte epoche e misurare i miglioramiti sia a livello visivo che in termini di loss.

Il primo grafico sono 1200 epoche e il secondo sono 1800 epoche:

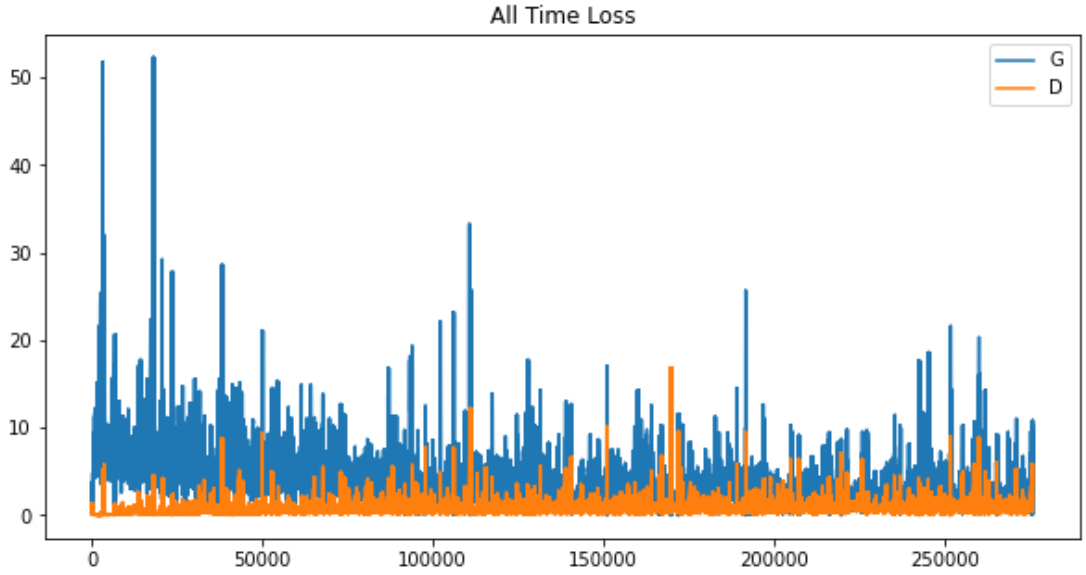


Figura 52 Grafico Disc e Loss allenamento per 1200 epoche

```
Gen_loss mean:  1.90
Disc_loss mean:  0.9
```

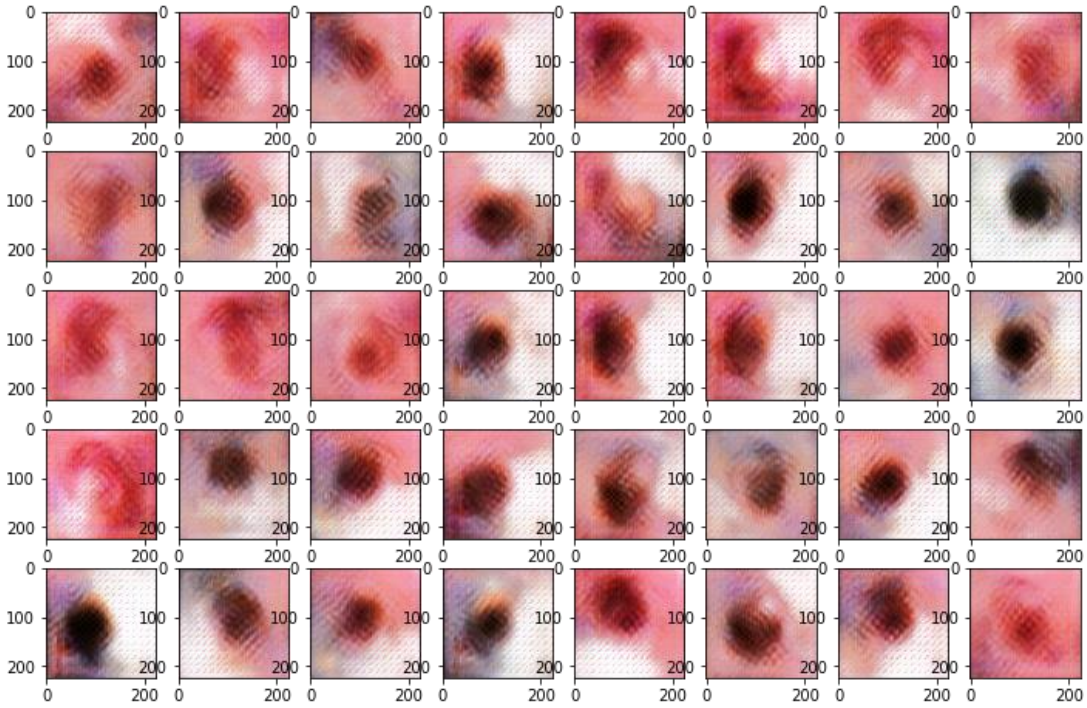


Figura 53 Visualizzazioni immagini generate per 1200 epoche

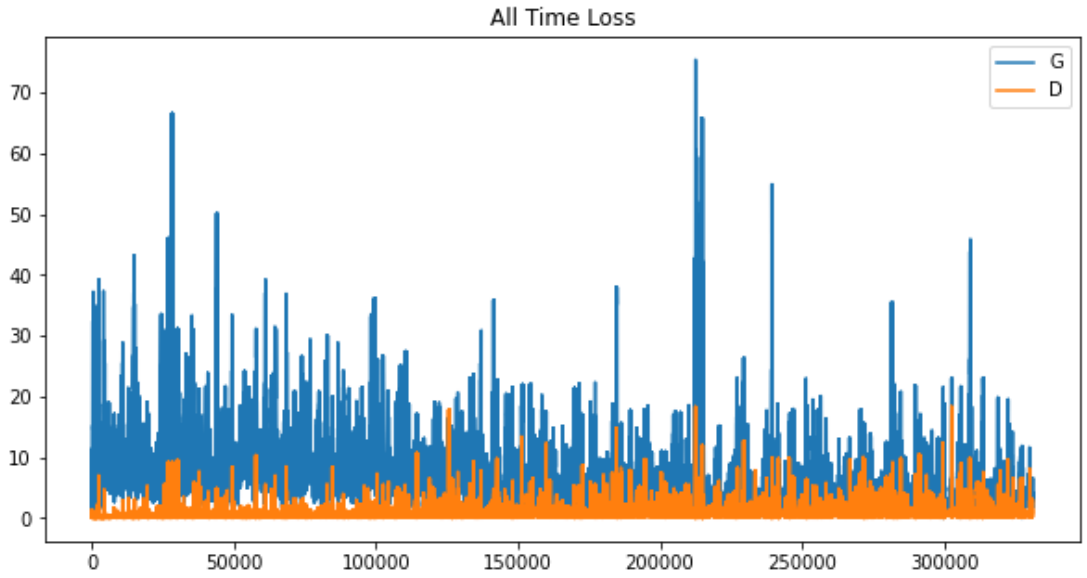


Figura 54 Grafico Disc e Loss per 1800 epoche

```
Gen_loss mean:  1.89
Disc_loss mean:  0.97
```

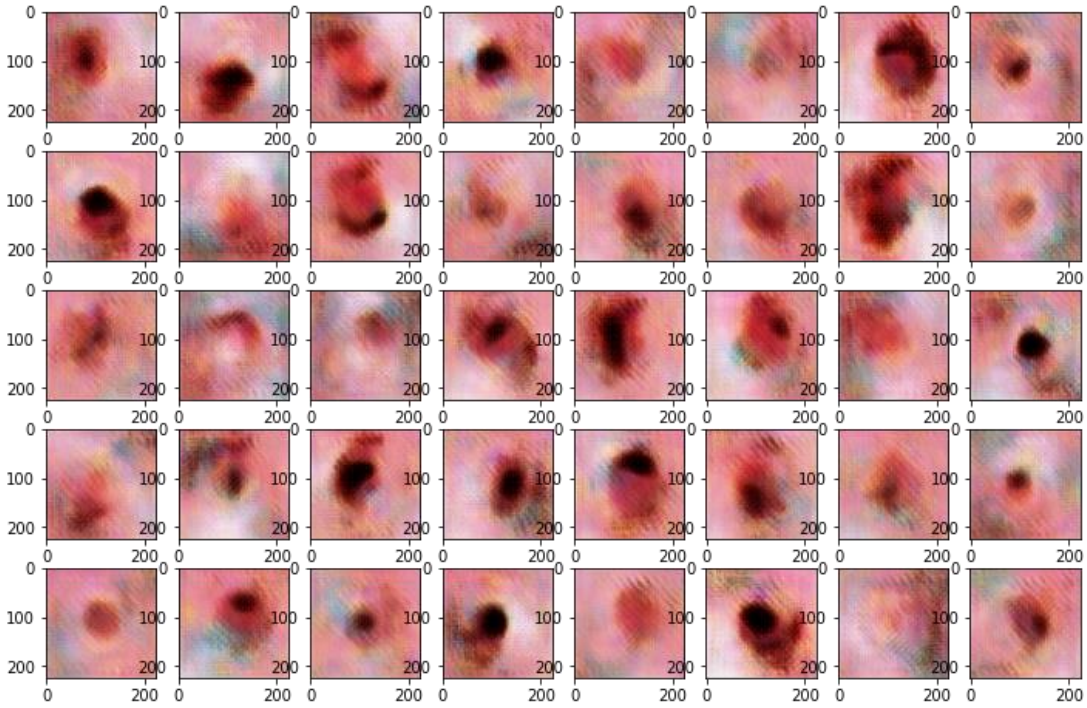


Figura 55 Visualizzazione immagini generate alla 1800 epoca

Con l'aumentare delle epoche si nota come le zone di indecisione della rete vadano scomparendo e le immagini iniziano sempre di più ad assomigliare a dei melanomi della pelle, anche i valori medi del generatore e del discriminatore si stabilizzano.

5 Allenamento con immagini generate

L' importanza dei dati nel campo dell'intelligenza artificiale è vitale, spesso si tende a sottovalutare molto la loro funzione, questo è dovuto al fatto che si tende a soffermarsi più sull'analizzare la parte algoritmica e puramente matematica tralasciando il fatto che anche un ottimo algoritmo, o in questo caso modello di deep learning, senza un adeguato numero di campioni e dati non è comunque in grado di lavorare in modo corretto.

Spesso si sente dire la frase che i dati sono diventati il nuovo petrolio, in parte questa affermazione per quanto semplice, è vera.

Ad esempio, le grandi aziende utilizzano le loro posizioni predominanti di mercato per raccogliere dati e utilizzarli poi per fidelizzare i propri clienti o magari scegliere a chi

inviare un determinato prodotto e a chi no, tramite le informazioni che noi rilasciamo in rete sono in grado di “predire” i nostri gusti e interessi.
A volte però queste immense moli di dati non sono facilmente reperibili, o come si è utilizzato in questa tesi servono dei processi più complicati per reperirli, per distinguere un tumore benigno o maligno della pelle, serve accertarsene, e quindi eseguire una biopsia su di esso per avere una risposta certa e per sapere a quale delle due categorie esso appartiene.
Già leggendo queste righe si capisce la criticità nel reperire questi immagini, quindi quello che si cerca di fare è aiutare ad allargare il numero di dati disponibili così da permettere anche su piccoli, dataset una migliore precisione nelle predizioni.
Per sopperire alla mancanza di dati utilizzerò i dati generati artificialmente in precedenza, e in più la tecnica della data agumentation.

5.1 Data Agumentation

La data agumentation è una tecnica che può essere usata per espandere in modo artificiale il numero del dataset di addestramento di partenza, creando e modificando versioni delle immagini.
Come detto sopra allenare le reti di deep learning avendo a disposizione più dati, può dar vita a modelli più accurati, in caso anche di scarsità dei dati di partenza, inoltre la creazione di nuove immagini, può aumentare l’abilità del modello allenato di generalizzare ad imparare su nuove immagini.
L'intento è quello di espandere il set di dati di training con nuovi esempi plausibili.
Ad esempio, un capovolgimento orizzontale di un'immagine di un neo può avere senso, perché la foto potrebbe essere stata scattata da sinistra o da destra, come un capovolgimento verticale della foto, mentre potrebbe essere sbagliato un aumento o una diminuzione nei livelli di esposizione alla luce dell’immagine.
Pertanto, è chiaro che la scelta delle tecniche specifiche di aumento dei dati utilizzate per un set di dati di training deve essere fatta con attenzione, la rete neurale convoluzionale o la CNN, possono apprendere funzionalità nuove rispetto al posizionamento dell'immagine.

L'aumento dei dati viene applicato solo al set di dati di training e non al set di dati di convalida o test, questo potrebbe alterare i risultati su futuri set di convalida dei dati[25].
Ora si dà uno sguardo a come implementare l'aumento dei dati, tramite la funzione offerta da Keras ImageDataGenerator, è possibile modificare i valori della funzione per cambiare la modalità con la quale i dati vanno aumentati.
Questa funzione è estremamente potente per quanto però essa sia anche distruttiva, si vede nell'esempio sottostante il perché.
Prendendo l'immagine del neo in fig[56] si applica la funzione ImageDataGenerator, con uno shift laterale:

```
datagen = ImageDataGenerator(width_shift_range=[-300,300])
```

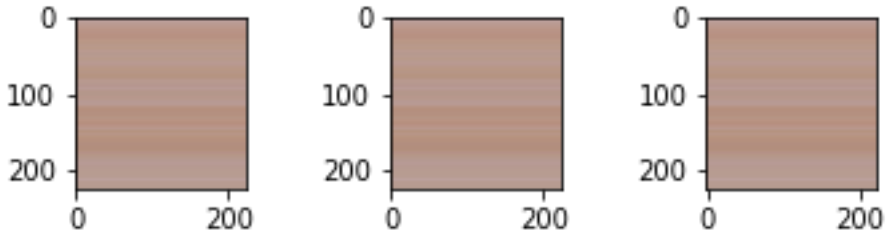


Figura 56 Utilizzo contro producente data agumentation

Invece che un miglioramento otteniamo un significativo peggioramento della qualità dell'immagine. Per questo motivo prima di passare alla fase di allenamento utilizzando la data agumentetion è indispensabile calibrare tutti i vari parametri[26].

```
datagen = ImageDataGenerator(width_shift_range=[-15,15])
```

[25] (https://keras.io/api/preprocessing/image/)
26(https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/)

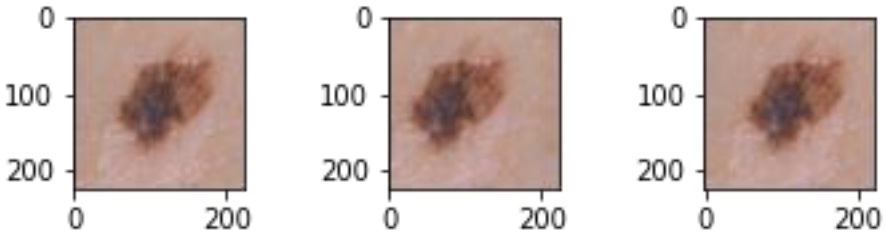


Figura 57 Shift laterale data agumentation

In questo caso le immagini sono decentrate ma la parte significativa è ancora visibile. Ripetiamo questo processo per i parametri più significativi che si ha intenzione di inserire nella funzione per evitare 'infelici' alterazioni delle immagini.

```
datagen = ImageDataGenerator(height_shift_range=[-15,15])
```

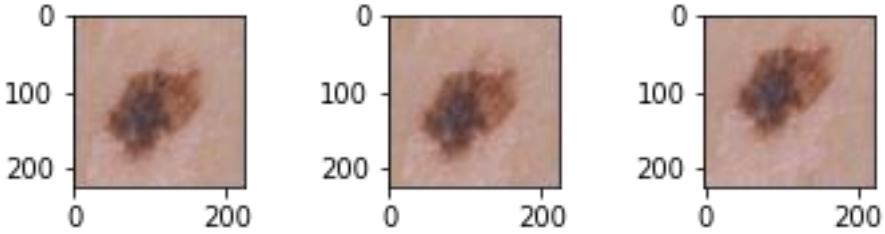


Figura 58 Shift in altezza data agumentation

```
datagen = ImageDataGenerator(horizontal_flip=True, vertical_fl
ip=True)
```

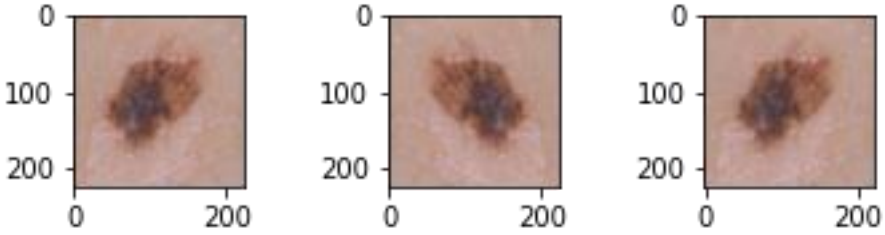


Figura 59 Flip orizzontale dell'immagine data agumentation

Horizontal flip e Vertical flip applicano solo un ribaltamento dell'immagine.

```
datagen = ImageDataGenerator(zoom_range= [0.7,1.0])
```

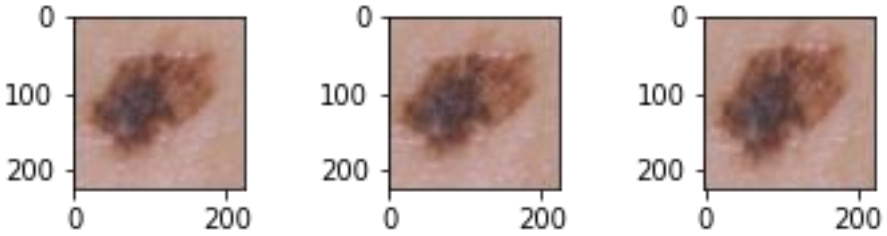


Figura 60 Zoom dell'immagine data agumentation

Applica uno zoom che è compreso tra l'intervallo `0.7` e `1`.

Una volta scelti i parametri e i valori si inseriscono tutti nella funzione finale di data agumentation e si passa alla fase di allenamento.
Per allenare la rete si utilizza sempre la funzione fit(), ma a posto di inserire il set di dati e lables si utilizza la funzione flow, quest'ultima ci permette di generare in modo automatico dei batch di dati aumentati.
Si addestra il modello per circa 60 epoche, una volta finito il ciclo di addestramento si passa alla prova del modello sui dati di convalida, non è stato possibile effettuare una convalida di K-fold in quanto la ram necessaria era limitata rispetto al carico effettivamente richiesto.
Per questo viene utilizzata la funzione evaluetion sul modello, passando i dati di convalida della rete, essa restituisce i punteggi di Loss e di Accuracy.

```
scores = model_with_generated1.evaluate(X_test, y_test)
acc_per_fold = (scores[1] * 100)
loss_per_fold = (scores[0])
```

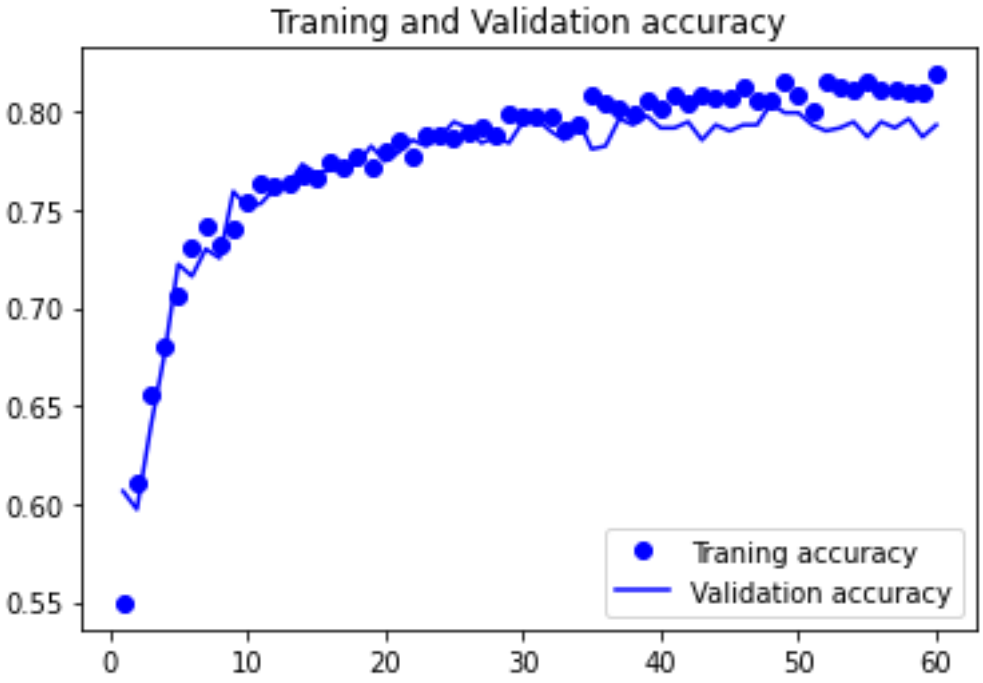


Figura 61 Grafico di accuracy utilizzando solo la data agumentation sui dati reali

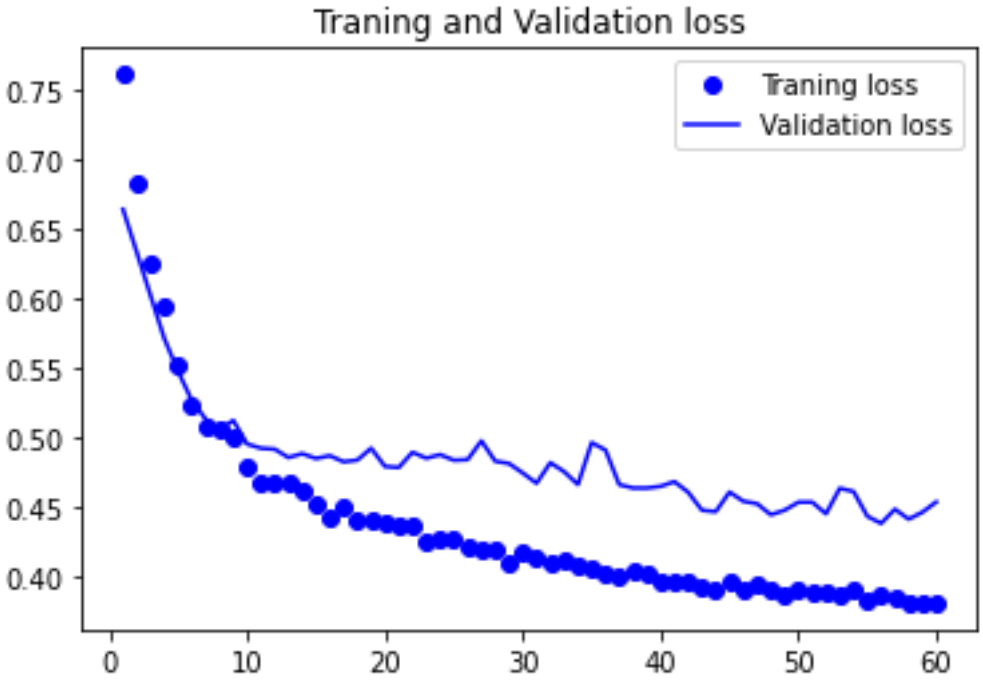


Figura 62 Grafico di loss utilizzando la data agumentation solo sui dati reali

```
> Loss: 0.47424243807792664 - Accuracy: 79.32098507881165%
```

Si nota che utilizzando la data agumentation si riscontrano dei benefici, in termini di miglioramento sugli score della rete in quanto si è abbassato il valore di loss da 0.49 a 0.47, mente in accuracy è dove si nota un netto miglioramento passando da 76% in convalida K-fold a un 79%.
Utilizzando la data agumentation però la cosa che risalta all'occhio è la sofferenza della rete in fase di allenamento di overfitting, in questo caso non è stato possibile sistemare questo comportamento, la tecnica di aumentare le immagini "generandole di nuove", crea inevitabilmente dei problemi, le immagini generate come in fig[60] sono sempre le stesse e anche variando la mappa delle caratteristiche di input comunque la rete impara a riconoscere dei pattern ben specifici rispetto ad altri, specializzandosi sempre di più nel distinguere le immagini nel set di addestramento.
Questo ci potrebbe far desistere nell'utilizzarla, però d'altro canto anche soffrendo di overfitting, in fase di validazione i punteggi risultano migliorati, sia in accuracy e loss.

5.2 Immagini generate artificialmente DCGAN

A questo punto si utilizzano i modelli salvati per generare le immagini dei tumori, e aggiungerli al nostro set di dati di partenza, per aumentare le features dalla quale la nostra rete possa apprendere nuove caratteristiche significative.
Per prima cosa vengono generate e salvate le immagini grazie ai modelli creati della rete generativa.
Il primo problema che si riscontra è quante immagini generate inserire e in quale rapporto, all'interno del dataset di partenza.
Per rispondere a questo problema non c'è un esatto valore ma l'unica cosa che si può fare sono dei tentativi, con varie dimensioni da affiancare al set di partenza e vedere un eventuale peggioramento o miglioramento.
Per prima cosa si ricarica il modello utilizzato con tutti i suoi pesi, si generano le immagini e si salvano in una cartella di Google Drive. Nella seconda parte si passa di nuovo alla fase di normalizzazione e creazione delle lables, non approfondirò i vari passaggi anche perché sono già stati ampiamenti spiegati nel paragrafo delle Convolutional Neural Network.

La prima prova che si svolge è utilizzando in proporzione ¼ di dati generati rispetto a quelli reali:

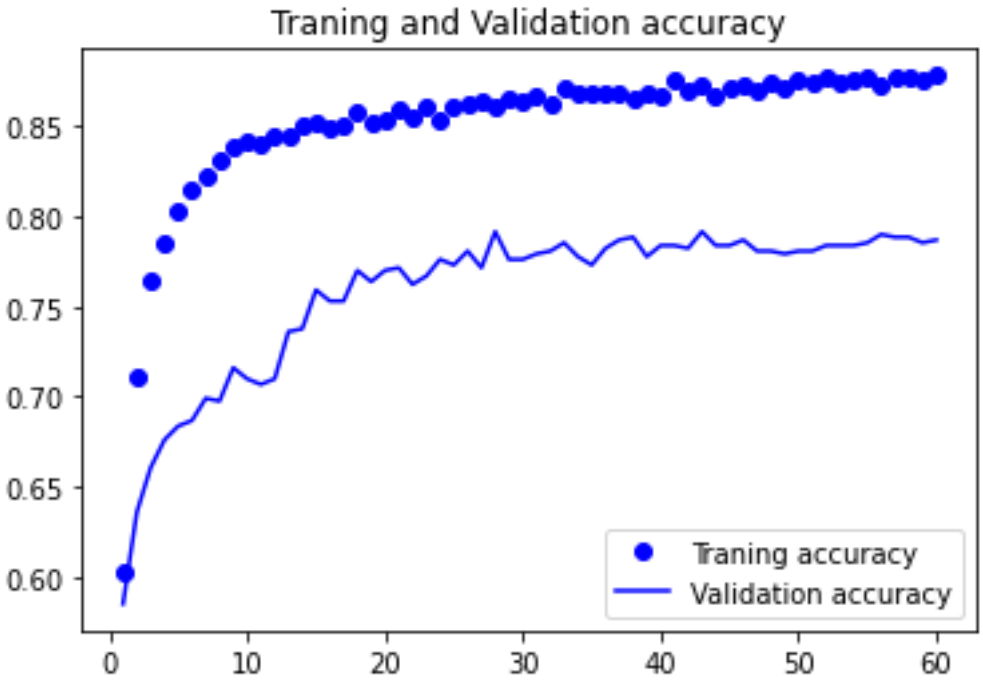


Figura 63 Grafico di accuracy utilizzando i dati reali + 1/4 di dati generati

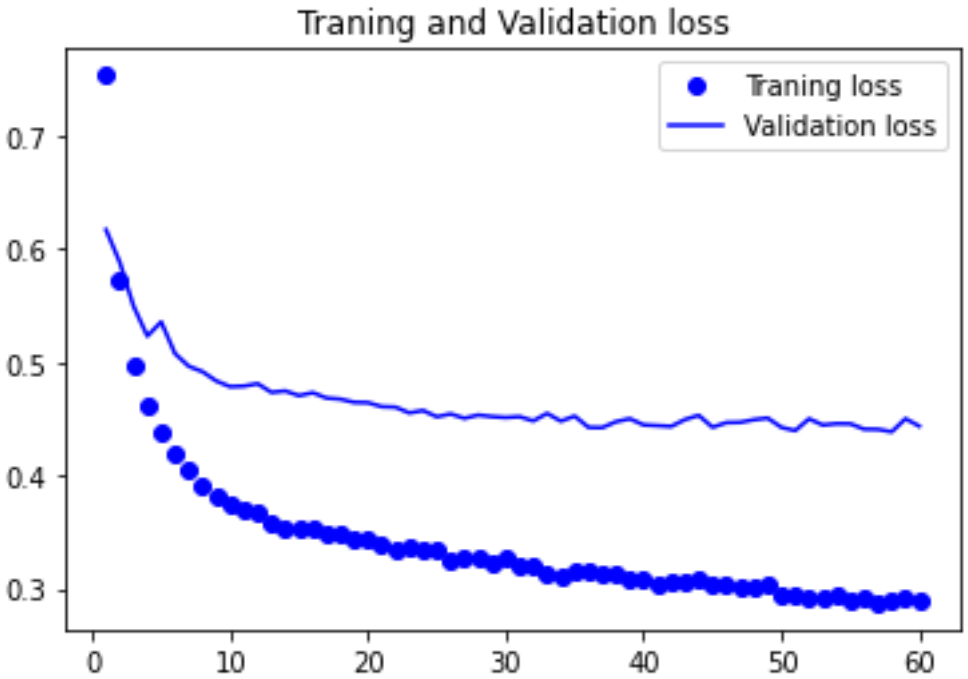


Figura 64 Grafico di loss utilizzando i dati reali + 1/4 di dati generati

```
> Loss: 0.43378378987312317 - Accuracy: 78.70370149612427%
```

Si vede come già detto per la data agumentation la rete soffra di overfitting, si ha un leggero peggioramento dei valori di accuracy ma miglioramento sui valori di loss.
La seconda prova è stata svolta aggiungendo lo stesso numero di dati generati rispetto a quelli veri, avendo così uno shape di `(5128, 224, 224, 3)`:

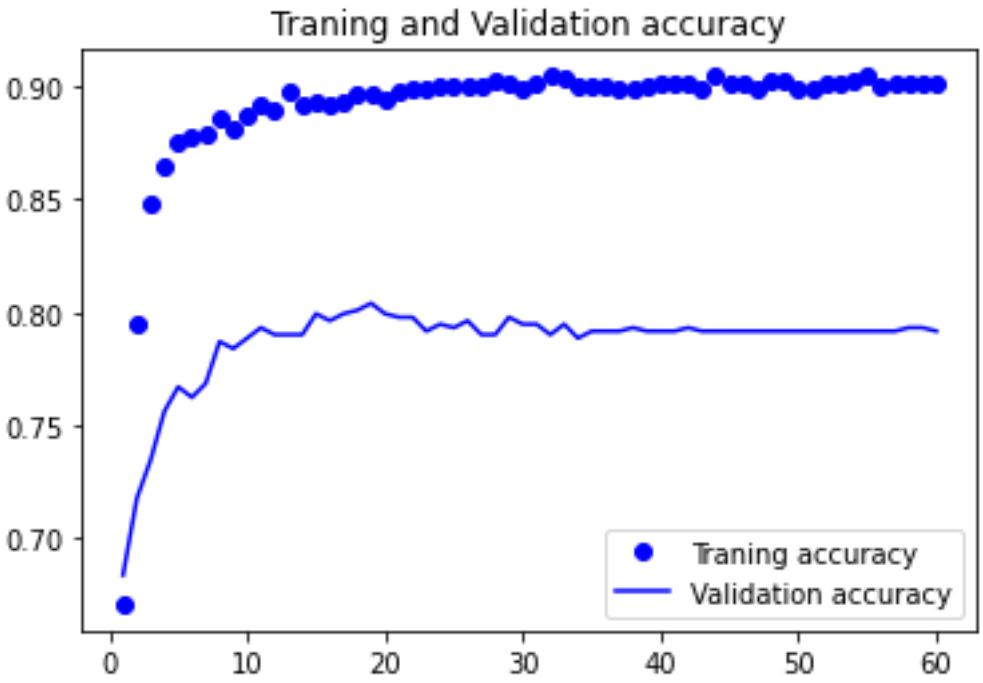


Figura 65 Grafico accuracy utilizzando stesso numero di reali e generati

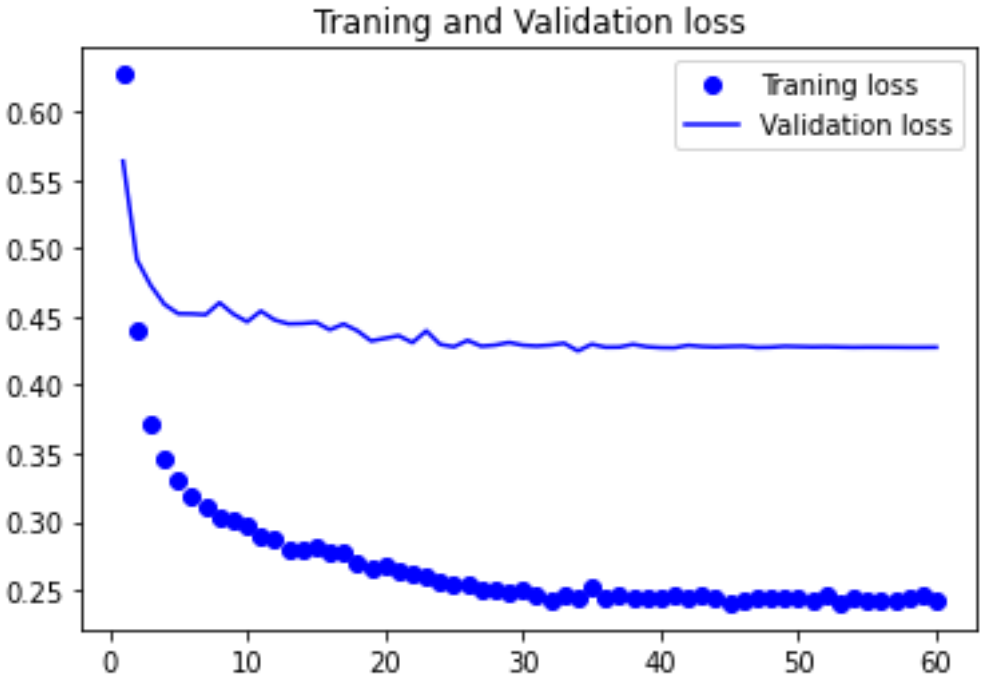


Figura 66 Grafico di loss utilizzando stesso numero di reali e generati

```
> Loss: 0.4174817705154419 - Accuracy: 80.16666865348816%
```

La rete anche con le accortezze di decrementare il learning rate continua a soffrire di overfitting ma in compenso, ha migliorato di nuovo i suoi valori in fasi di convalida.
Continuando a variare i parametri delle percentuali delle immagini generate artificialmente non si raggiungono i risultati sperati, i valori di loss e accuracy in fase di validazione rimangono sempre negli intorni di `0.41` e `79%`, senza aumentare in modo visibili le prestazioni.
Il problema più grave invece è l'overfitting che risulta completamente fuori controllo, raggiungendo rapidamente il `98%` di accuracy e il `0.10` di Loss in fase di training, senza migliorare gli score della rete in fase di evaluetion.

5.3 Accuracy vs Loss

La perdita è definita come la differenza tra il valore previsto dal modello e il valore reale, essa denota la probabilità prevista dal modello che il campione appartenga a una determinata classe.

Mentre la precisione è la metrica per misurare le prestazioni del modello, è definito come numero di previsioni corrette diviso il numero totale di previsioni, la maggior parte delle volte si osserva che la precisione aumenta con la diminuzione della perdita. Questo potrebbe non essere sempre vero come nell'esempio fornito sopra, perché l'accuratezza e la perdita misurano due cose diverse.
In conclusione, la domanda che ci si deve porre è quale modello prediligere, se uno con alta accuracy e alta loss, o uno con bassa loss e bassa accuracy, non c'è una risposta alla domanda in quanto essa dipende dai campi nella quale andremo ad utilizzare la rete neurale[27]. Darò un parere nelle conclusioni.

6 Conclusione

In conclusione, essendo una persona autocritica la domanda che mi è sorta spontanea mentre provavo questo approccio, è se valeva veramente la pena?
La risposta alla domanda è, dipende.
Il tempo utilizzato per lo sviluppo delle due reti e varie prove è molto alto. Facendo un piccolo schema temporale di come sono andate le cose; In un mese e mezzo ho finito di scrivere il codice base per entrambi i modelli, e per più di due mesi ho svolto le prove e ho cercato di migliorare le prestazioni aggiustando il tiro prova dopo prova, variando parametri consultando documentazioni e scavando sempre più a fondo alla radice dei problemi che insorgono (escludendo poi i vari bag durante la stesura del codice).
Inoltre, anche utilizzando i sistemi hardware di Google si incontrano varie limitazioni sul tempo di utilizzo delle GPU e quindi rende il processo più lungo del dovuto soprattutto per allenare le reti generative che richiedo all'incirca 20 ore totali all'una.
Questo ovviamente mi fa pensare che un processo così complicato come usare una rete generativa per migliorare le prestazioni dell'altra non è uno degli approcci più semplici sia in termini di sforzo mentale che di tempo, e non lo consiglierei a chi vuole per

[27] (https://kharshit.github.io/blog/2018/12/07/loss-vs-accuracy)

esempio distinguere volti o fare una semplice classificazione binaria o multi-classe di immagini.

Però essendo il concetto di particolare criticità, ogni miglioramento della rete risulta essere importante, e cercare di spremere a fondo la scarsità dei dati a disposizione può essere una valida idea sul quale concentrarsi.

Utilizzare la complessità delle reti generative in concomitanza di ambiti di ricerca per aumentare i dati sul quale studiare e fare prove, è una quindi una valida alternativa.

La paura più grande alla quale sono andato incontro durante l'utilizzo di questo approccio è di non avere alcun tipo di risultato, per quanto le immagini di melanomi generate riescano ad ingannare un discriminatore che è la stessa tipologia di rete convoluzionale utilizzata per la detection finale, il pensiero è che le immagini generate non apportino alcun valore alla mappa delle caratteristiche di input, non migliorando e nemmeno peggiorando di conseguenza alcun tipo di parametro della rete.

I risultati però alla fine ci sono e portano comunque un sostanziale decremento del valore di loss, e a un leggero incremento dell'accuracy della rete a discapito però di un aumento dell'overfitting, che non è stato possibile correggere in alcun modo.

Avendo probabilmente un hardware dedicato e più tempo per lavorare, avrei provato altri modelli di reti generative magari provando una CycleGAN o Pix2Pix, per vedere come si sarebbero comportate e quali benefici avrebbero portato alla rete CNN di partenza in termini di prestazioni, e se le reti generative possono cambiare il modo di fruizione e utilizzo dei dati a disposizione, dandogliene un valore aggiunto.

Immagini

Link Immagini:

https://gluon.mxnet.io/_images/dcgan.png

https://miro.medium.com/max/446/1*oePAhrm74RNnNEolprmTaQ.png

https://miro.medium.com/max/875/1*uAeANQIOQPqWZnnuH-VEyw.jpeg

https://www.google.com/url?sa=i&url=https%3A%2F%2Flibrary.weschool.com%2Flezione%2Fnotazione-delle-derivate-6698.html&psig=AOvVaw1xrVnnKGVSSPyBLR0tBNF1&ust=1614938175812000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCLDpiOGvlu8CFQAAAAAdAAAAABAD

https://i2.wp.com/blog.profession.ai/wp-content/uploads/2018/07/Schermata-2018-07-25-alle-18.04.10.png?resize=1024%2C592&ssl=1

https://www.kaggle.com/amitalexander/scaler-vector-tensor-basics-for-neural-network

https://www.digikey.com/maker-media/6c3d4f5f-98e0-4104-ad8c-fb0b47000109

Riferimenti

Chollet, F. (s.d.). *Deep learning con Phyton.* Apogeo.

http://www.andreaminini.com/ai/machine-learning/differenza-tra-overfitting-e-underfitting. (s.d.).

https://atcold.github.io/pytorch-Deep-Learning/it/week09/09-3/. (s.d.).

https://blog.profession.ai/deep-learning-svelato-ecco-come-funzionano-le-reti-neurali-artificiali. (s.d.). Tratto da ProfessionAI.

https://docs.python.org/3/library/os.html. (s.d.).

https://keras.io/api/layers/convolution_layers/convolution2d/. (s.d.).

https://keras.io/api/layers/initializers/. (s.d.).

https://keras.io/api/layers/normalization_layers/batch_normalization/. (s.d.).

https://keras.io/api/preprocessing/image/. (s.d.).

https://kharshit.github.io/blog/2018/12/07/loss-vs-accuracy. (s.d.).

https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/. (s.d.).

https://numpy.org/doc/stable/reference/routines.html. (s.d.).

https://stackoverflow.com/questions/64626869/k-fold-crossvalidation-on-images. (s.d.).

https://stats.stackexchange.com/questions/233658/softmax-vs-sigmoid-function-in-logistic-classifier. (s.d.).

https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. (s.d.).

https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. (s.d.).

https://towardsdatascience.com/dont-overfit-how-to-prevent-overfitting-in-your-deep-learning-models-63274e552323. (s.d.).

https://www.inference.vc/instance-noise-a-trick-for-stabilising-gan-training/. (s.d.).

https://www.isic-archive.com/#!/topWithHeader/onlyHeaderTop/gallery?filter=%5B%5D. (s.d.).

https://www.machinecurve.com/index.php/2019/11/12/using-leaky-relu-with-keras/. (s.d.).

https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/cosine_decay. (s.d.).

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose. (s.d.).

https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D. (s.d.).

https://www.tensorflow.org/guide/checkpoint. (s.d.).

https://www.tensorflow.org/tutorials/generative/dcgan. (s.d.).