



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Tecniche di Image Captioning per applicazioni per i beni culturali

Corso di laurea in
Ingegneria Informatica

Andrea Leonardo

Matr. 6292833

Relatore

Prof. Marco Bertini

Data _____ / _____ / _____

L'unico modo per scoprire i limiti del possibile sta nell'andare un po' oltre e avventurarsi nell'impossibile.

Ogni tecnologia abbastanza progredita è indistinguibile dalla magia.

Michio Kaku

Se i quadri si potessero spiegare e tradurli in parole,
non ci sarebbe bisogno di dipingerli.

Paul Courbet

Indice

Abstract	i
Elenco delle Tabelle	iv
Elenco delle Figure	v
Elenco degli Estratti di Codice	vii
Abbreviazioni	ix
Capitolo 1 Introduzione	1
1.1 Descrivere un'opera d'arte	1
1.2 Obiettivo della tesi	1
Capitolo 2 Deep Learning	3
2.1 Le Reti Neurali	4
2.1.1 Anatomia di una Rete Neurale	4
2.1.2 Tipi di Reti Neurali	10
2.2 Addestramento di una NN	15
2.2.1 Apprendimento Supervisionato	15
2.2.2 Pre-elaborazione dei dati	16
2.2.3 Overfitting e Underfitting	17
Capitolo 3 Artwork Captioning	20
3.1 Image Captioning	21
3.1.1 Metodi di estrazione delle caratteristiche	21
3.1.2 Metodi di Attenzione	23
3.1.3 Metriche di Valutazione	26
3.2 Il Programma di "Artwork Captioning"	27
3.2.1 Descrizione del modello	27
3.2.2 Addestramento della rete	33
Capitolo 4 Mobile App	48

4.1	Sviluppo Applicazione con Flutter	49
4.1.1	Struttura dell'applicazione	50
4.1.2	Utilizzo dell'applicazione	55
4.2	Sistema Client-Server con Ngrok	57
Capitolo 5	Conclusioni	59
Bibliografia		68
Ringraziamenti		70

Elenco delle tabelle

2.1	Scelta dell'attivazione finale e della funzione obiettivo adatta al modello(2).	16
3.1	Risultati delle ultime 5 epoche di pre-allenamento della rete.	39
3.2	Risultati di 5 epoche di allenamento senza DA.	43

Elenco delle figure

2.1	Grafi di una NN semplice e una NN profonda.	6
2.2	Schema di interazione fra layers, funzione obiettivo e ottimizzatore (2)	7
2.3	Rete Neurale Feed-Forward	10
2.4	Funzionamento della convoluzione (2)	11
2.5	Esempio di Max-Pooling con blocchi 2x2	12
2.6	Un Residual Block di una ResNet (5)	12
2.7	Grafo di una RNN	13
2.8	Cella LSTM(6)	14
2.9	Cella GRU(6)	14
2.10	Classificatore di immagini di numeri da 0 a 9.	15
2.11	Esempi di data-augmentation.	18
2.12	Effetto della capacità di rete minore sul valore loss in fase di convalida(2).	19
3.1	Esempi di Image Captioning su fotografie.	21
3.2	Modello Encoder-Decoder	22
3.3	Esempio di Soft Attention su una foto di un segnale stradale(8)	24
3.4	Esempio di Hard Attention su una foto di un segnale stradale(8)	25
3.5	Rappresentazione Resnet101	29
3.6	Prima ricerca del lr sui dati di flickr30k.	39
3.7	Grafico dei valori di loss per le ultime 5 epoche di pre-allenamento della rete.	39
3.8	Esempio di Testing della rete preallenata su dati fotografici(1)	40
3.9	Esempio di Testing della rete preallenata su dati fotografici(2)	40
3.10	Esempio di Testing della rete preallenata sulla Monnalisa(1)	42
3.11	Esempio di Testing della rete preallenata sul David di Michelangelo(2)	42
3.12	Grafico dei valori di loss di 5 epoche di allenamento senza DA.	44
3.13	Grafico dei valori di loss per le successive 20 epoche di allenamento senza DA.	44

3.14 Esempio di Testing della rete allenata senza DA sulla Monnalisa(1)	45
3.15 Esempio di Testing della rete allenata senza DA sul David di Michelangelo(2)	45
3.16 Analisi dei valori di loss su 100 epocha senza DA.	46
3.17 Analisi dei valori di loss su 100 epocha con DA.	46
4.1 Diagramma di interazione tra frontend e backend.	49
4.2 Pagina principale dell'applicazione.	50
4.3 Fasi di inserimento dell'URL.	55
4.4 Schermate di scatto della foto con o senza inserimento dell'url.	56
5.1	62
5.2	62
5.3	62
5.4	63
5.5	63
5.6	63
5.7	64
5.8	64
5.9	64
5.10	65
5.11	65
5.12	65
5.13	66
5.14	66
5.15	67
5.16	67
5.17	67

Elenco degli Estratti di Codice

3.1	Inizializzazione del modello	27
3.2	Inizializzazione encoder	28
3.3	Definizione encoder	28
3.4	Funzioni di congelamento rete e fine tuning	30
3.5	Inizializzazione Decoder	30
3.6	Inizializzazione Visual Attention	31
3.7	Definizione Visual Attention	31
3.8	Definizione step di decoding	32
3.9	Utilizzo del Teacher Forcing	33
3.10	Struttura del Dataset nel file json	34
3.11	Split del dataset	35
3.12	Creazione del Vocabolario	35
3.13	Tokenizzazione delle captions	36
3.14	Definizione della classe ImageCaptionDataset	36
3.15	Creazione Dataloader	37
3.16	Inizializzazione del Learner	37
3.17	Metodo di ricerca del learning rate	38
3.18	Metodo training della rete	38
3.19	Switching dei layers di dropout	41
3.20	Applicazione della BeamSearch	41
3.21	Visualizzazione della caption e dell'attenzione	41
4.1	Definizione Widget principale	51
4.2	Classe Main	51
4.3	Gestione del body dell'HomePage	52
4.4	Inserimento dell'url tramite pulsante	52
4.5	Codice per scattare la fotografia	53

4.6	Dichiarazione del metodo <i>uploadImage</i> per fare richiesta all'url	53
4.7	Parsing del JSON e apertura del display del risultato	54
4.8	Dichiarazione del metodo di risposta alla richiesta post	57
4.9	Creazione del tunnel ngrok	58
5.1	Pulizia delle decrizioni di artpedia	60
5.2	Ricerca immagini con google_images_download	61

Abbreviazioni

- 0D** 0-Dimensionale.
- 1D** 1-Dimensionale.
- 2D** 2-Dimensionale.
- 3D** 3-Dimensionale.
- 4D** 4-Dimensionale.
- BLEU** Blilingual Evaluation Understudy.
- CNN** Convolutional Neural Network.
- DL** Deep Learning.
- GRU** Gated Recurrent Unit.
- IA** Intelligenza Artificiale.
- JSON** JavaScript Object Notation.
- LSTM** Long Short-Term Memory.
- ML** Machine Learning.
- MLP** Multilayer Perceptron.
- NLL** Negative Log Likelihood.
- NN** Neural Network.
- RELU** Rectified Linear Unit.
- ResNet** Residual Neural Network.
- RNN** Recurrent Neural Network.
- SGD** Stochastic Gradient Descent.
- WSGI** Web Server Gateway Interface.

Capitolo 1

Introduzione

La seguente introduzione fornisce una semplice spiegazione degli ambiti trattati nel progetto di questa tesi in modo da poter capire da subito quali siano i fondamenti di Intelligenza Artificiale (IA) alla base e lo scopo della stessa.

1.1 Descrivere un'opera d'arte

Descrivere un'immagine non è sicuramente una cosa semplice e il compito risulta ancora più arduo se il soggetto della descrizione è un'opera d'arte. Gli aspetti che caratterizzano un lavoro artistico sono molteplici, sia dal lato tecnico dell'opera (dimensione, autore, anno) sia da quello estetico, senza parlare poi dell'aspetto emotivo che può suscitare.

Quello che andremo a cercare tramite questa tesi è una valutazione spontanea dell'opera (1), ovvero una descrizione legata alle impressioni dell'osservatore, cercando di trarre le informazioni oggettive principali.

1.2 Obiettivo della tesi

Lo scopo della nostra tesi è dunque relativamente meno impegnativo rispetto ad una vera descrizione di un'immagine. Sarà però fondamentale distinguere una statua da un quadro, un uomo da una donna, un giovane da un anziano e sarebbe interessante

arrivare a descrivere nel modo più dettagliato possibile i gesti e gli abiti dei soggetti, le decorazioni dello sfondo e le emozioni dei volti.

Il progetto di questa tesi si baserà su una *Rete Neurale* di *Image-Captioning*, la quale avrà il compito di restituire una descrizione in output per un’immagine ricevuta come input. Questi concetti di IA verranno spiegati in breve nel prossimo capitolo e le descrizioni verranno accompagnate dalle scelte effettuate nella realizzazione del progetto.

In questa tesi verrà inoltre presentata una semplice applicazione mobile che permetta di testare la rete neurale su un telefono fornendole come input un’immagine scattata con la fotocamera, in modo da poter dare un’idea di come questo progetto possa essere utilizzato fisicamente, per esempio nella visita di un museo.

Capitolo 2

Deep Learning

Innanzitutto è necessario spiegare cosa sia il *Deep Learning* e come questa scienza ci permetta di raggiungere i nostri scopi.

Il *Deep Learning (DL)* è una sotto-branca del *Machine Learning (ML)*, il quale è un sottinsieme dell'IA che si occupa della creazione di modelli e algoritmi per l'apprendimento automatico.

Per poter fare Machine Learning(2) sono necessarie tre cose:

- Un input, rappresentato da punti di dati
- Esempi dell'output atteso
- Un modo per valutare l'esito del lavoro

Un modello di ML trasforma i suoi dati di input in output significativi, processo che viene "appreso" tramite l'esposizione a esempi noti di input e output. Ciò che rende *deep* il Deep Learning è la grande quantità di layers che compongono il modello, ovvero quante trasformazioni subiscono i dati di input per ottenere il risultato. Le composizioni di layer sovrapposti del DL sono note come *Reti Neurali*(Neural Network (NN)).

2.1 Le Reti Neurali

Nonostante nomi evocativi come "intelligenza artificiale", "apprendimento automatico" e "reti neurali", tali tecnologie hanno poco a che fare con il pensiero o l'intelligenza umana. Piuttosto, sono modi alternativi di programmare i computer, utilizzando grandi quantità di dati per addestrarli a svolgere un'attività(3). Vediamo dunque come sono composte e come funzionano concretamente le NN.

2.1.1 Anatomia di una Rete Neurale

Per l'addestramento di una NN sono necessari i seguenti elementi:

- i *layers*
- i dati di *input* e rispettivi *target*
- la *funzione obiettivo*
- l'*ottimizzatore*

Rappresentazione dei dati

Prima di descrivere nei dettagli il funzionamento di una rete neurale è fondamentale sapere come rappresentare i dati. Essi si rappresentano tramite array multidimensionali

noti come *Tensori*. Un Tensore è un contenitore di dati (quasi sempre numerici), che si potrebbe definire come una generalizzazione delle matrici. I tensori si distinguono in base al numero di *dimensioni* o *assi* e alla *forma*

Un tensore 0-Dimensionale (0D) è anche detto Scalare.

Un tensore 1D è anche detto Vettore.

Un tensore 2D è anche detto Matrice.

Esistono anche tensori di dimensione 3 o più elevata. Ad esempio un tensore 4D può rappresentare dataset di immagini espresse in numero di pixel e valori RGB. La forma di un vettore è una tupla di interi che descrive quante dimensioni ha il tensore lungo ciascun asse. Solitamente il primo asse indica l'*asse dei campioni*. L'insieme dei campioni utilizzati per l'allenamento della rete formano il *Dataset*. Solitamente però il modello di rete non elabora un intero dataset alla volta, ma esso viene suddiviso in lotti detti *batch*.

La valutazione di un modello deve prevedere la suddivisione dei dati disponibili in tre *set*: training, convalida e test. Il modello viene addestrato sui dati di training e valutato sul set di convalida per correggerne gli iperparametri (es. numero e dimensione dei layer). Viene infine messo alla prova sui dati di test.

I Layers

Il layer è la struttura dati di base di una rete neurale. Un *layer* è un modulo di elaborazione dei dati che prende come input uno o più tensori e produce in output uno o più tensori.

Nella maggior parte dei casi i layers hanno uno stato ed esso è rappresentato dai *pesi*, ovvero uno o più tensori appresi grazie alla discesa stocastica del gradiente. L'insieme dei pesi forma la conoscenza della rete.

Esistono diversi tipi di layers, adatti per forme di tensori e tipi di elaborazione dati differenti. Solitamente per tensori 2D si utilizzano layer densamente connessi o pienamente connessi. Per i dati sequenziali conservati in tensori 3D si utilizzano layer ricorrenti come i LSTM. I dati di immagini sotto forma di tensori 4D vengono invece elaborati tramite layer 2D.

Una Rete Neurale è un grafo aciclico e connesso di layer.

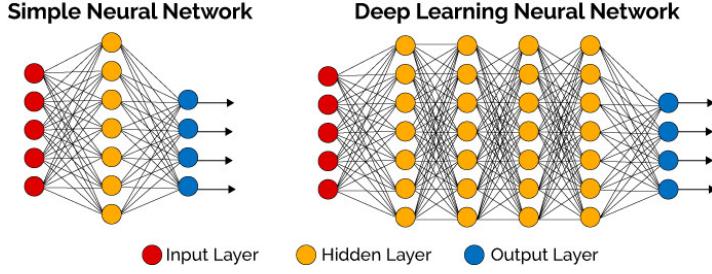


Figura 2.1: Grafi di una NN semplice e una NN profonda.

Un layer neurale può dunque essere visto come una funzione che modifica l'input tramite dei pesi e restituisce un valore di output che costituirà l'input del layer seguente. La funzione che viene effettivamente rappresentata dal layer è detta *funzione di attivazione*, e restituisce un output in base alla somma dei valori di input moltiplicati per i relativi pesi. Le funzioni di attivazione più comuni sono la *funzione di soglia*, la *funzione di Sigmund*, la *funzione rampa o rettificatrice* e la *funzione tangente iperbolica*.

Tra queste la più utilizzata per i layers intermedi è la funzione rettificatrice, che restituisce $\sum wX$ solo se questa somma risulta positiva, altrimenti restituisce valore nullo. I layer che utilizzano questa funzione di attivazione vengono detti Rectified Linear Unit (RELU). Dunque un layer può avere questa forma:

$$\text{output} = \text{relu}(\text{dot}(W, \text{input}) + b)$$

In questo caso i valori W e b rappresentano i pesi. Inizialmente ai pesi di una rete sono attribuiti valori casuali. Ovviamente questo non permetterà alla rete di predire subito in modo esatto gli output corretti (però in qualche modo è necessario istanziare una nuova rete). Successivamente gli input di allenamento vengono fatti girare nella rete riuniti in batch e i relativi valori di output vengono valutati da un funzione detta *funzione di loss* o *funzione obiettivo*. A seconda del valore di quest'ultima i pesi dei layers vengono modificati dall'*ottimizzatore*, con l'obiettivo di ridurre il valore di loss per i successivi input.

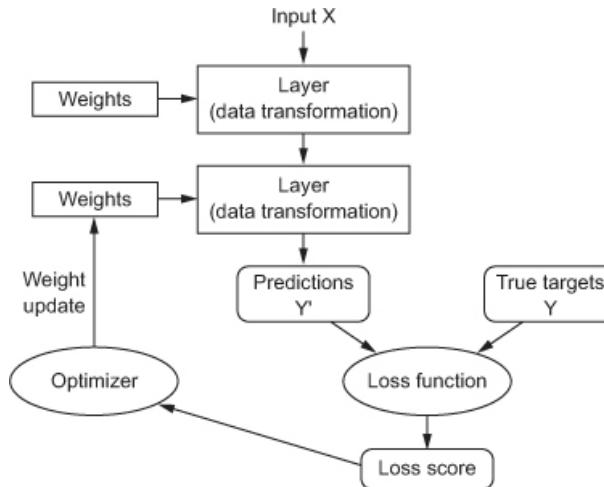


Figura 2.2: Schema di interazione fra layers, funzione obiettivo e ottimizzatore (2)

Questo procedimento è detto *addestramento (training)* della rete ed è ciò che costituisce l'*apprendimento* del DL.

La funzione obiettivo

La *funzione obiettivo* è la quantità da minimizzare durante l'addestramento. Essa rappresenta quanto siamo vicini al nostro obiettivo e il suo valore determina la variazione dei pesi dei layer allo scopo di abbassarne il valore.

Una rete neurale che presenta più valori di output può avere di conseguenza più funzioni obiettivo. Il valore di ognuna di esse viene però utilizzato per ricavarne un valore di loss unico (spesso attraverso un calcolo di media)

La funzione obiettivo serve a rappresentare il problema di apprendimento, dunque essa deve essere correlata adeguatamente al compito della rete per non avere risultati indesiderati dall'addestramento. La funzione obiettivo utilizzata nel progetto verrà spiegata successivamente, dopo aver steso le basi sull'image-captioning.

L'ottimizzatore

L'*ottimizzatore* determina il modo in cui la rete verrà aggiornata in base alla funzione obiettivo. Esso implementa una variante specifica della *Discesa Stocastica del Gradiente* (SGD). Il metodo della discesa del gradiente è il primo e più comune metodo di ottimizzazione. L'algoritmo si basa sull'aggiornamento iterativo di un parametro θ lungo la direzione opposta a quella del gradiente della funzione obiettivo $J(\theta)$.

L’aggiornamento viene sviluppato in modo da convergere gradualmente al valore ottimo della funzione obiettivo. Uno dei parametri del metodo è il learning rate η che determina l’ampiezza della variazione di θ in ciascuna iterazione e quindi influenza il numero di iterazioni necessarie a raggiungere il valore ottimo della funzione obiettivo.

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} J(\theta)$$

Il metodo è di semplice implementazione e nel caso in cui $J(\theta)$ sia una funzione (fortemente) convessa, la soluzione trovata sarà un punto di ottimo globale. Nel caso in cui la funzione obiettivo non sia convessa, la soluzione trovata potrebbe essere un minimo locale (o più in generale un punto stazionario).

Nel metodo appena descritto, il gradiente $\nabla_{\theta} J(\theta)$ è calcolato utilizzando, ad ogni iterazione, tutto il training set. Questo determina una elevata e ridondante complessità computazionale e non consente di effettuare gli update online di θ . Per ovviare a questo punto debole una soluzione migliore è quella del metodo della *discesa del gradiente stocastico*. L’idea è quella di calcolare il gradiente non più mediante tutto il training set ma mediante un singolo campione selezionato in modo casuale ad ogni iterazione.

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)})$$

L’utilizzo di un singolo campione determina una forte diminuzione della complessità computazionale che in ciascuna iterazione adesso è dovuta esclusivamente al numero D delle feature del training set. La stocasticità nella selezione dei campioni dal training set determina anche che la ricerca della soluzione possa non rimanere ”intrappolata” in un minimo locale di $J(\theta)$.

Nonostante il metodo SGD sia molto popolare ed ampiamente utilizzato, il processo di aggiornamento di θ risulta spesso molto lento. Fra i punti aperti infatti c’è una più opportuna regolazione del learning rate per velocizzare la convergenza e fare in modo che il processo di ricerca della soluzione non rimanga intrappolato in un minimo locale di $J(\theta)$.

Una delle idee che si è fatta strada è quella del “momento” che, per come è stato pensato, gioca il ruolo di una velocità v . L’idea è quella di calcolare, ad ogni iterazione, una media mobile esponenziale dei gradienti storici ed utilizzare questo valore come direzione da seguire nell’aggiornamento di θ . Il parametro $\beta \in [0,1)$ determina la velocità

di decadimento dei contributi dei gradienti storici.

$$v_t = \beta v_{t-1} - \eta \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)})$$

$$\theta_t + 1 = \theta_t + v_t$$

Risulta evidente quanto una opportuna regolazione del learning rate possa avere una forte influenza sugli effetti del metodo SGD. A tal riguardo sono stati proposti diversi meccanismi adattativi per una regolazione automatica del learning rate. Un importante miglioramento all'algoritmo della discesa del gradiente stocastico è dovuto al metodo AdaGrad che implementa una regolazione dinamica del learning rate basandosi sui valori storici del gradiente. La differenza tra AdaGrad e la classica discesa del gradiente risiede nel fatto che, durante il processo di aggiornamento di θ , il learning rate non sarà più costante ma sarà continuamente ricalcolato utilizzando i valori storici del gradiente accumulati fino alla corrente iterazione. Lo svantaggio di questo procedimento deriva dal fatto che, nei casi di training time molto lungo, l'elevato valore cumulativo dei gradienti farà tendere a zero il valore del learning rate e di conseguenza θ tenderà ad un valore stazionario non corretto.

Il problema della tendenza di θ alla non corretta stazionarietà è stato affrontato e risolto da Geoffrey Hinton con il *metodo RMSProp*. Questo algoritmo utilizza una finestra temporale dei gradienti storici e ne calcola, ad ogni iterazione, il momento cumulativo del secondo ordine (la varianza).

$$v_t = \beta v_{t-1} + (1-\beta)g_t^2$$

Il *metodo Adam* (Adaptive momentum estimation), ovvero quello utilizzato nel programma, introduce un ulteriore miglioramento a quanto visto in precedenza. Oltre a memorizzare la media mobile esponenziale dei quadrati dei gradienti delle precedenti iterazioni, viene memorizzata anche la media mobile esponenziale dei gradienti $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$. Il nome dell'algoritmo deriva dal fatto che, ad ogni iterazione, m_t e v_t corrispondono al momento primo (la media) ed al momento secondo (la varianza) del gradiente.

$$g_t = \nabla_{\theta_t} J(\theta_t)$$

$$\alpha_t = \alpha - \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$$

$$\theta_{t+1} = \theta_t - \alpha_t \frac{m_t}{\sqrt{v_t} + \epsilon}$$

2.1.2 Tipi di Reti Neurali

Sono utilizzate diverse strutture di reti neurali a seconda del metodo di apprendimento utilizzato e dello scopo dell'applicazione.

Reti Feed-Forward

Queste reti neurali artificiali possono condurre le informazioni in una sola direzione di elaborazione.

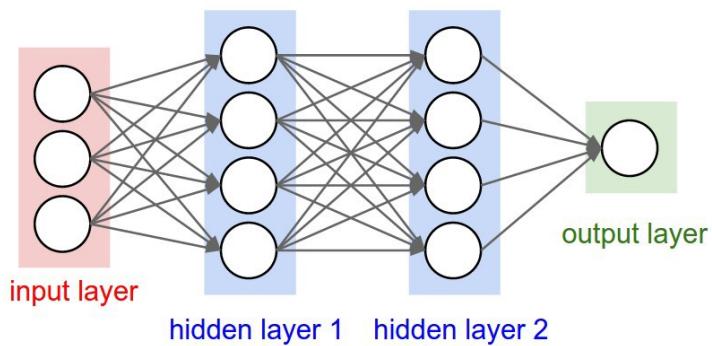


Figura 2.3: Rete Neurale Feed-Forward

Le reti neurali feedforward sono ideali per modellare le relazioni tra un insieme di predittore o variabili di input e una o più variabili di output. In altre parole, sono appropriati per qualsiasi problema di mappatura funzionale in cui vogliamo sapere come un numero di variabili di input influisce sulla variabile di output. Le reti neurali feedforward multistrato, chiamate anche percetroni multi-strato (MLP), sono in pratica il modello di rete neurale più ampiamente studiato e utilizzato.

Reti Convoluzionali

Una rete neurale convoluzionale (CNN) è un tipo di rete neurale artificiale feed-forward in cui il pattern di connettività tra i neuroni è ispirato da processi biologici. I layers si basano sulla operazione matematica di convoluzione(4):

la *convoluzione* di due funzioni reali $f(x)$ e $g(x) \in L_1(R)$, è la funzione $h(x)$ così definita: $h(t) = (f * g)(t) = \int f(t - x)g(x) dx$ ovvero l'integrale del prodotto tra le due funzioni, in cui una delle funzioni viene traslata dopo averne preso la simmetrica rispetto all'asse y.

Hanno diverse applicazioni nel riconoscimento di immagini e video, nei sistemi di raccomandazione, nell'elaborazione del linguaggio naturale e, recentemente, in bioinformatica. La differenza fondamentale tra un layer densamente connesso e un layer di convoluzione è questa: il primo apprende pattern globali mentre il secondo apprende pattern locali (nel caso di immagini i pattern si trovano all'interno di finestre 2D degli input). Questi ultimi possiedono 2 grandi proprietà:

- i pattern appresi sono *invarianti alla traslazione*, ovvero possono essere riscontrati ovunque e non necessariamente nella posizione dell'immagine dove son stati appresi.
- possono apprendere *gerarchie spaziali di pattern*, ovvero la grandezza e complessità dei pattern può aumentare durante la concatenazione dei layer in modo da avere un apprendimento più efficiente.

Le convoluzioni operano su tensori 3D, chiamati *mappe delle caratteristiche* (formati da due assi spaziali e un asse per la profondità) e si definiscono in base a due parametri: le *dimensioni delle patch estratte*, tipicamente 3x3 o 5x5, e la *profondità della mappa delle caratteristiche di output*, ovvero in numero di filtri calcolati.

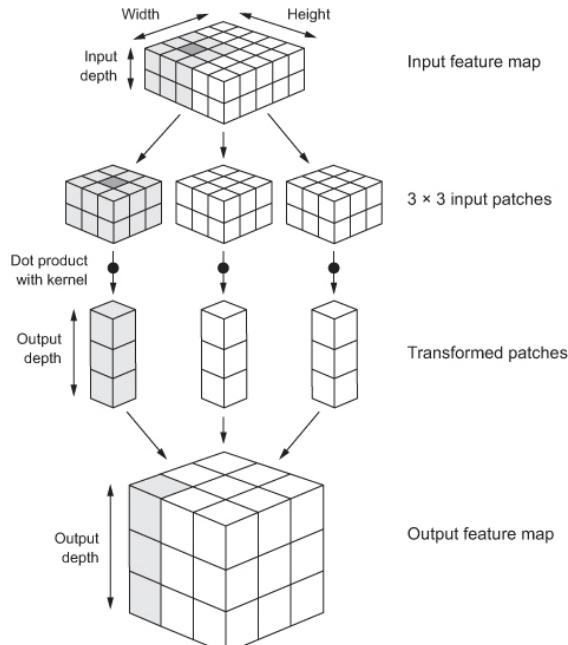


Figura 2.4: Funzionamento della convoluzione (2)

Tipicamente ogni layer convoluzionale viene fatto seguire da uno di *Max-Pooling*. Il Max-Pooling è un metodo per ridurre la dimensione di un'immagine, suddividendola in blocchi e tenendo solo quello col valore più alto. Così facendo si riduce il problema di overfitting riducendo il numero di coefficienti.

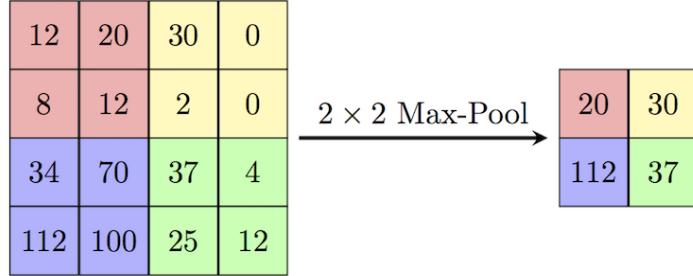


Figura 2.5: Esempio di Max-Pooling con blocchi 2x2

Reti Residuali

Una *Rete Residuale* è un tipo di CNN, la quale rende possibile l’addestramento fino a centinaia o addirittura migliaia di livelli ottenendo prestazioni convincenti. Secondo il teorema di approssimazione universale, data una capacità sufficiente, sappiamo che una rete feedforward con un singolo strato è sufficiente per rappresentare qualsiasi funzione. Tuttavia, il livello potrebbe essere enorme. Pertanto, c’è una tendenza comune nella comunità di ricerca secondo cui la nostra architettura di rete deve andare più in profondità.

Tuttavia, qualora il numero di livelli crescesse, il gradiente può diventare estremamente piccolo (vanishing gradient) a causa di un rapido aumento dell’accuratezza della rete, determinando un aggiornamento minimo dei pesi e causando un rallentamento del processo di training.

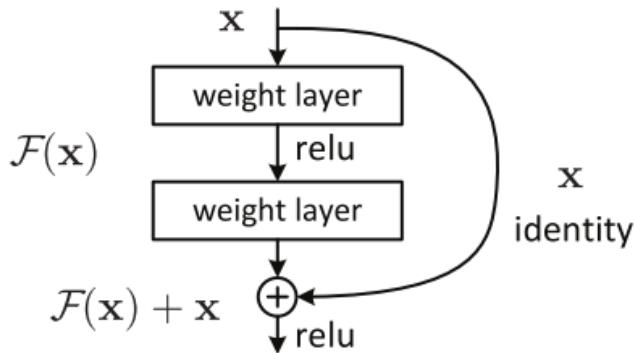


Figura 2.6: Un Residual Block di una ResNet (5)

L’idea centrale di una ResNet è l’introduzione di una cosiddetta ”connessione di collegamento di identità” che salta uno o più livelli. Invece di aspettare che il gradiente si

propaghi indietro (backpropagation) un livello alla volta, il percorso di skip connection gli consente di raggiungere i nodi iniziali efficacemente saltando quelli intermedi.

Come documentato dalla Microsoft(5) le ResNet a 50/101/152 strati sono più accurate di quella a 34 con margini considerevoli. Non osserviamo il problema del degrado del gradiente e quindi godiamo di significativi guadagni di precisione da una profondità considerevolmente aumentata. I vantaggi della profondità sono testimoniati da tutte le metriche di valutazione.

Reti Ricorrenti

Le *Reti Neurali Ricorrenti* (RNN), rispetto a quelle precedentemente descritte possiedono una memoria derivata dal fatto che il flusso di informazione scorre anche all'indietro.

In pratica una RNN è un tipo di rete neurale con un ciclo interno.

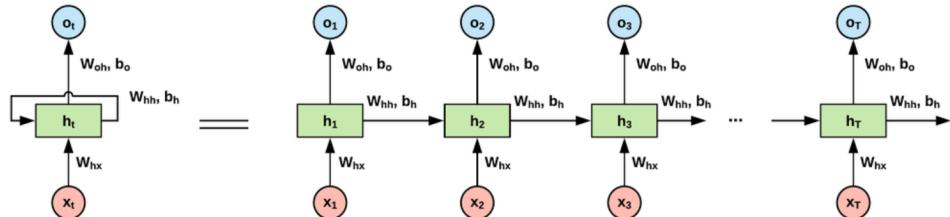


Figura 2.7: Grafo di una RNN

Una RNN elabora delle sequenze eseguendo iterazioni sugli elementi della sequenza e conservando uno stato contenente informazioni relative alle operazioni precedenti. Lo stato di una RNN subisce un *reset* solo fra l'elaborazione di due sequenze diverse. Dunque una sequenza costituisce un singolo punto di dati e la rete opera su di essa ciclicamente.

Due dei modelli di celle ricorrenti più popolari sono la cella Long Short-Term Memory (LSTM) e la cella Gated Recurrent Unit (GRU).

La cella *LSTM* mantiene uno stato della cella leggibile e scrivibile. Sono presenti 4 porte che regolano la lettura, la scrittura e l'output dei valori da e verso lo stato della cella, a seconda dei valori di input e dello stato della cella. Il primo gate determina ciò

che lo stato nascosto dimentica. Il gate successivo è responsabile della determinazione della parte in cui viene scritto lo stato della cella. La terza porta decide i contenuti che vengono scritti. Infine, l'ultima porta legge dallo stato della cella per produrre un'uscita.

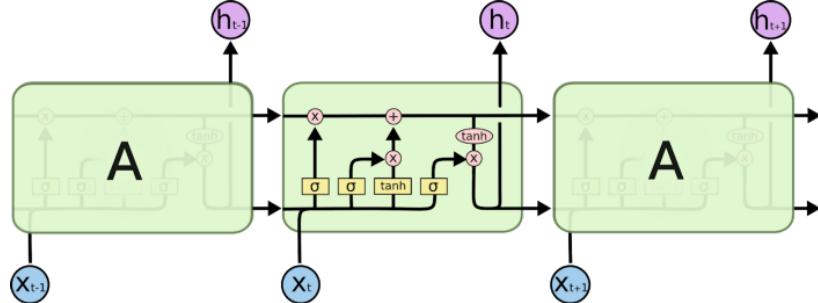


Figura 2.8: Cella LSTM(6)

La cella *GRU* è simile alla cella LSTM ma con alcune importanti differenze. Innanzitutto, non esiste uno stato nascosto. Lo stato della cella adotta la funzionalità dello stato nascosto dal design della cella LSTM. Successivamente, i processi di determinazione di ciò che lo stato della cellula dimentica e quale parte dello stato della cella è scritto vengono consolidati in un unico gate. Viene scritta solo la parte dello stato della cella che è stata cancellata. Infine, l'intero stato della cella viene fornito come output. Tutte queste modifiche insieme forniscono un design più semplice con meno parametri rispetto a LSTM.

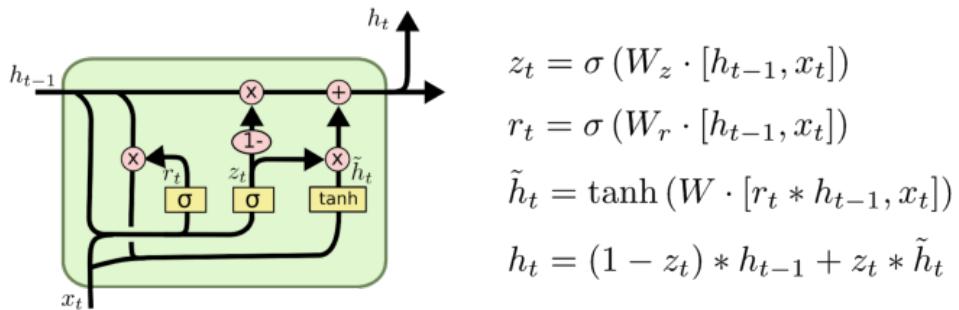


Figura 2.9: Cella GRU(6)

Secondo vari esperimenti(6)(7) nonostante abbia un minor numero di parametri, il modello GRU è stato in grado di ottenere prestazioni migliori (valori loss più bassi, training più rapido) rispetto al modello LSTM.

2.2 Addestramento di una NN

Avendo dato una rapida occhiata alle reti neurali, lo scheletro del sistema, vediamo ora come queste vengano utilizzate e quali sono gli aspetti principali di cui tener conto per svolgere un corretto addestramento. Innanzitutto è necessario esplicare il tipo di apprendimento che svolgerà la nostra rete neurale.

Nel ML esistono principalmente 4 tipi di apprendimento:

- Apprendimento supervisionato
- Apprendimento non supervisionato
- Apprendimento auto-supervisionato
- Apprendimento per rinforzo

Per il progetto di questa tesi ci interessa l'*apprendimento con supervisione*.

2.2.1 Apprendimento Supervisionato

L'*Apprendimento Supervisionato* consiste nell'imparare a mappare i dati di input su determinati target noti, in base ad un insieme di esempi. L'insieme di esempi consistuisce il dataset di training. Le due attività più comuni di apprendimento con supervisione sono:

- *Classificazione*: Il classificatore elabora i dati in input e restituisce in output una classe ossia una categoria. Se la scelta è tra sole due categorie la classificazione è detta *binaria*.
- *Regressione*: Il regressore elabora i dati in input e restituisce in output un numero reale.

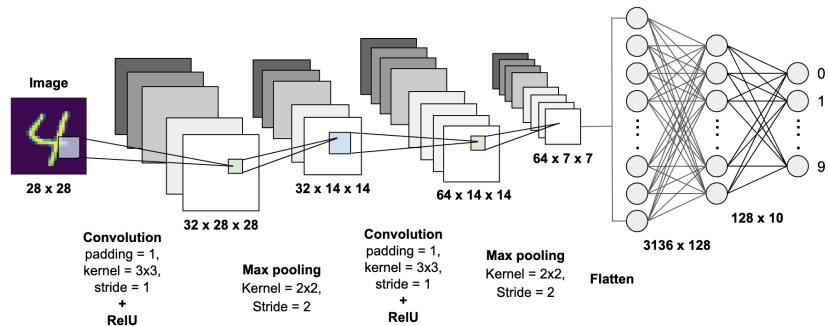


Figura 2.10: Classificatore di immagini di numeri da 0 a 9.

In base al tipo di attività che la rete deve svolgere è necessario scegliere l'*attivazione finale* e la *funzione obiettivo* adeguate. Nella seguente tabella sono indicate le scelte più comuni per i principali problemi di classificazione e regressione.

Tabella 2.1: Scelta dell'attivazione finale e della funzione obiettivo adatta al modello(2).

Tipo di problema	Attivazione finale	Funzione obiettivo
Classificazione binaria	sigmoid	binary_crossentropy
Multiclasse, mono-etichetta	softmax	categorical_crossentropy
Multiclasse, multietichetta	sigmoid	binary_crossentropy
Regressione a valori arbitrari	/	mse
Regressione a valori $\in [0, 1]$	sigmoid	mse o binary_crossentropy

In questa tesi elaboreremo un problema un po' più complesso, ovvero la *Generazione di sequenze*: data un'immagine cercheremo di predire un titolo che la descriva. La generazione di sequenze talvolta può essere riformulata come una serie di problemi di classificazione (come predire ripetutamente una parola o un token in una sequenza). Prima di vedere però come funziona effettivamente un generatore di sequenze vediamo quali altri fattori è necessario conoscere per l'addestramento di una rete neurale.

2.2.2 Pre-elaborazione dei dati

Come detto in precedenza, i dati che entrano in una rete neurale si presentano sotto forma di particolari tensori. I dati che abbiamo nella realtà però, non sono in questa forma. Basti pensare ad un'immagine formato jpg o png che vogliamo utilizzare in un dataset di classificazione. Inoltre i dati dovrebbero presentarsi al modello in modo da cercare di facilitarne il compito.

La pre-elaborazione dei dati consiste dunque nel prendere dei dati grezzi e renderli adatti per una NN. Questa avviene mediante queste operazioni:

- **Vettorizzazione**: consiste nella trasformazione dei dati in tensori.
- **Normalizzazione**: si rendono i valori più possibilmente piccoli (tipicamente tra 0 e 1) e omogenei.
- **Gestione valori mancanti**: si creano dati con valori mancanti per allenare la rete a questa eventualità nei dati di test.

2.2.3 Overfitting e Underfitting

Vediamo infine come gestire la rete durante l’addestramento. Una NN per lavorare bene, dovrebbe gestire al meglio la tensione tra *ottimizzazione* e *generalizzazione*. L’ottimizzazione è il processo di aggiustamento del modello in modo che abbia le migliori prestazioni possibili nell’apprendimento dei dati di addestramento. La generalizzazione consiste invece nella capacità del modello di comportarsi correttamente su dati mai visti prima. L’obiettivo è quello di avere una buona generalizzazione, sulla quale però non abbiamo alcun controllo. In base al rapporto tra questi la rete può trovarsi in stati differenti.

Underfit

Un rete neurale si trova in uno stato di *underfit* quando al minor valore di loss sui dati di addestramento corrisponde un minore valore di loss sui dati di test. Questa situazione è tipica delle fasi di inizio del training, quando la rete non ha ancora incorporato nel modello tutti i pattern rilevanti presenti nei dati di addestramento. Questa situazione è positiva poichè, finché la generalizzazione continua a migliorare, sarà semplicemente sufficiente continuare ad allenare la rete su nuove epoche (tempo di allenamento di tutti i dati di training).

Overfit

Un rete neurale si trova in uno stato di *overfit* quando la generalizzazione smette di migliorare e le metriche di convalida si appiattiscono iniziando a degradare. Questo è dato dal fatto che la rete inizia a seguire i pattern specifici dei dati di addestramento, i quali risultano fuorvianti o irrilevanti per l’interpretazione di nuovi dati. Questa situazione non è affatto piacevole, poichè un overfitting troppo grande può portare la rete in stati irreversibili, costringendone un riaddestramento dall’inizio. L’unico modo per combattere l’overfitting è cercare di prevenirlo attraverso dei processi di *regolarizzazione*.

Il primo, se non anche il più efficace, metodo per combattere l’overfitting è *aumentare i dati di addestramento*. Questo è rigorosamente logico, poichè sottponendo la

rete a un maggior numero di esempi essa potrà riconoscere uno stesso pattern in più rappresentanti aumentandone dunque la generalizzazione.

Questo però non è sempre possibile, specialmente quando è necessario creare un dataset da zero. Esiste però un processo, utile specialmente in caso di datasets di piccole dimensioni, noto come *data-augmentation*.

La data-augmentation consiste nel generare più dati di addestramento a partire da campioni di addestramento esistenti, aumentando i campioni tramite una serie di trasformazioni casuali. Nel caso di immagini è possibile generare delle immagini differenti, ma dall'aspetto credibile.



Figura 2.11: Esempi di data-augmentation.

Come si può vedere nella figura 2.12, si possono creare immagini molto simili, ma effettivamente diverse in fatto di valori tensoriali. In questo caso sono state utilizzate due delle principali tecniche di data-augmentation, ovvero la *rotazione* e il *flip orizzontale*. Esistono molti altri tipi di trasformazioni che modificano la forma come il *cropping*, ovvero la selezione di solo una porzione di immagine. Vengono spesso utilizzate anche trasformazioni che modificano i colori e la luminosità dell'immagine, ma ho ritenuto fossero poco adatte per il tipo di addestramento che doveva subire la rete. L'effetto della data-augmentation è quello di non riproporre mai alla rete una stessa identica immagine. Questo però non garantisce l'assenza di overfit, poiché si parla di immagini diverse, ma pur sempre simili.

Ci sono dunque altre tecniche per evitare overfitting, come per esempio la *riduzione delle dimensioni della rete*. Questa tecnica prevede una diminuzione di parametri da aggiornare e dunque un'difficoltà minore di training. Bisogna comunque fare attenzione a impiegare un modello che abbia un numero sufficiente di parametri da non generare underfit e non essere quindi in grado di apprendere i pattern desiderati.

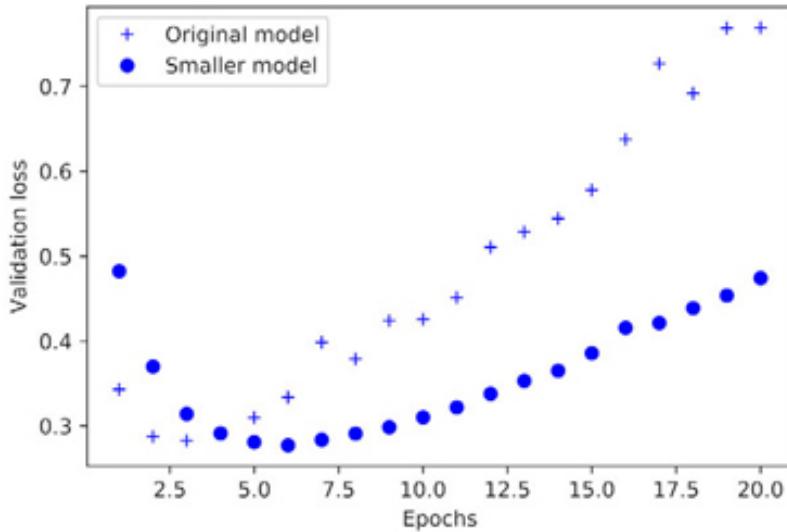


Figura 2.12: Effetto della capacità di rete minore sul valore loss in fase di convalida(2).

La gestione della capacità di memorizzazione è dunque un problema non risolvibile a priori, ma solitamente attraverso la prova di reti di dimensione differente. Per ridurre ulteriormente il rischio di overfitting è utile affiancare ai precedenti metodi la *regolarizzazione dei pesi*, ovvero l'aggiunta nella funzione obiettivo di un costo relativo alla presenza di grossi pesi. In pratica questo permette alla rete di distribuire in modo più uniforme l'influenza dei vari pesi rispetto al valore finale di loss.

L'ultimo meccanismo di regolarizzazione che è doveroso citare è l'aggiunta del *dropout*. Questa tecnica si applica singolarmente al layer e consiste nell'azzeramento di un numero casuale di caratteristiche di output durante la fase di addestramento. Solitamente il tasso di azzeramento si trova tra 20% e 50%. In fase di test l'output viene ridotto di scala in base al tasso di dropout. Il concetto di base è che aggiungendo del rumore nei valori di output si cerca di evitare la prematura memorizzazione di pattern specifici indesiderati, i quali sono principio di overfitting.

Capitolo 3

Artwork Captioning

Vediamo ora il programma di image captioning sviluppato, come è stata addestrata la rete e quali risultati sono stati ottenuti.

Per prima cosa diamo un’occhiata a come funziona una rete di Image Captioning, in modo da comprendere al meglio il codice e le scelte che verranno spiegate in seguito.

3.1 Image Captioning

L’*Image Captioning* è il processo di generazione della descrizione testuale di un’immagine. Utilizza sia l’elaborazione del linguaggio naturale che la visione artificiale per generare le descrizioni.

A young boy is playing basketball. 	Two dogs play in the grass. 	A dog swims in the water. 	A little girl in a pink shirt is swinging. 
A group of people walking down a street. 	A group of women dressed in formal attire. 	Two children play in the water. 	A dog jumps over a hurdle. 

Figura 3.1: Esempi di Image Captioning su fotografie.

3.1.1 Metodi di estrazione delle caratteristiche

I modelli di Image Captioning possono essere suddivisi in due categorie principali: un metodo basato su un modello di linguaggio di probabilità statistica per generare caratteristiche artigianali e un modello di rete neurale basato su un modello di linguaggio codificatore-decodificatore per estrarre caratteristiche profonde.

Modello di Linguaggio statistico

Questo metodo è un sistema basato sulla stima della massima verosimiglianza, che apprende direttamente il rivelatore visivo e il modello del linguaggio dal set di dati di descrizione dell’immagine. Le parole vengono rilevate applicando una rete neurale convoluzionale (CNN) all’area dell’immagine. La struttura della frase viene quindi ad-

destrata direttamente dalla didascalia per ridurre al minimo le ipotesi a priori sulla struttura della frase. Infine, passa a un problema di ottimizzazione per cercare la frase più probabile.

Modello di Rete Neurale Encoder-Decoder

Il metodo di generazione della descrizione dell’immagine basato sul modello encoder-decoder viene proposto con l’applicazione della RNN.

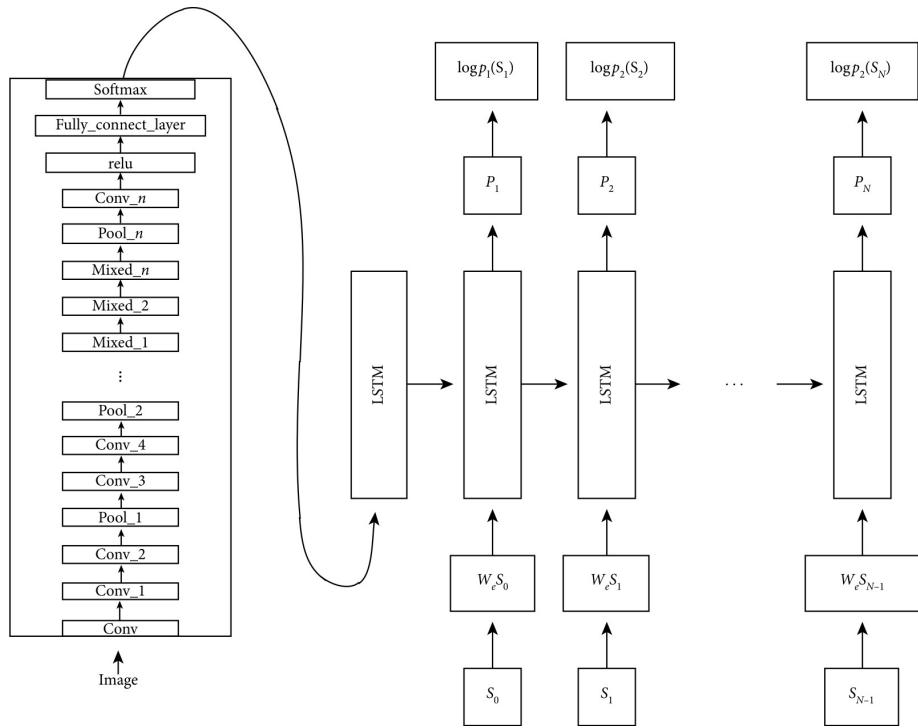


Figura 3.2: Modello Encoder-Decoder

Nel modello, il *codificatore* è una CNN e le caratteristiche dell’ultimo strato o strato convoluzionale completamente connesso vengono estratte come caratteristiche dell’immagine. Il *decodificatore* è una rete neurale ricorrente, utilizzata principalmente per la generazione della descrizione delle immagini.

L’output dell’ultimo layer nascosto dell’Encoder viene dato al primo passo temporale del Decoder. Viene impostato $x_1 = \text{START}$ e viene assegnata l’etichetta desiderata $y_1 = \text{prima parola della sequenza}$. Analogamente, impostiamo $x_2 = \text{vettore di parole della prima parola}$ e ci aspettiamo che la rete preveda la seconda parola. Infine, nell’ul-

timo passaggio, viene assegnato a $x_T = \text{ultima parola}$, l'etichetta di destinazione $y_T = \text{END}$.

Durante l'allenamento, l'input corretto viene fornito al decoder in ogni fase temporale, anche se il decoder ha commesso un errore in precedenza. Durante il test, invece, l'output del decodificatore all'istante t viene retroazionato e diventa l'ingresso del decodificatore all'istante $t + 1$.

3.1.2 Metodi di Attenzione

Il meccanismo di attenzione, derivante dallo studio della visione umana, è un'abilità cognitiva complessa che gli esseri umani hanno nella neurologia cognitiva. Quando le persone ricevono informazioni, possono ignorare consapevolmente alcune delle informazioni mentre si focalizzano su altre informazioni. Questa capacità di auto-selezione è chiamata *attenzione*. Questo meccanismo è stato inizialmente proposto per essere applicato alla classificazione delle immagini. E' risultato particolarmente rilevante anche nell'elaborazione del linguaggio naturale, quando le persone leggono testi lunghi, l'attenzione umana è focalizzata spesso su parole chiave. Nei modelli di NN, la realizzazione del meccanismo di attenzione alla rete neurale di avere la capacità di concentrarsi sul suo sottoinsieme di input (o caratteristiche), per selezionare input o caratteristiche specifici. La parte principale del meccanismo di attenzione è costituita dai seguenti due aspetti: la decisione deve prestare attenzione a una determinata parte dell'input e l'assegnazione di limitate risorse di elaborazione delle informazioni alla parte importante. Al momento, le formule di calcolo del meccanismo di attenzione tradizionale sono mostrate nelle seguenti equazioni.

$$a_t = align(m_t, m_s) = \frac{\exp(f(m_t, m_s))}{\sum_s \exp(m_t, m_s')}$$

$$f(m_t, m_s) = \begin{cases} m_t^T m_s & dot \\ m_t^T W_a m_s & general \\ W_a[m_t; m_s] & concat \\ v_a^T \tanh(W_a m_t, U_a m_s) & perception \end{cases}$$

L'idea progettuale è quella di collegare il modulo di destinazione m_t con il modulo di origine m_s tramite una funzione e infine normalizzare per ottenere la distribuzione di

probabilità. In questa sezione discutiamo due meccanismi alternativi per il metodo dell’attenzione: attenzione stocastica e attenzione deterministica.

Deterministic “Soft” Attention

Il termine “Soft” si riferisce alla distribuzione di probabilità della distribuzione dell’attenzione. Per ogni parola nella frase di input S, la probabilità è data secondo il vettore di contesto $Z_t(8)$. Infine, la somma ponderata di tutte le regioni viene calcolata per ottenere la distribuzione di probabilità.

$$E_{p(s_t|a)}[\hat{z}_t] = \sum_{i=1}^L \alpha_{t,i} a_i$$

La funzione obiettivo può essere scritta come segue:

$$L_d = -\log(P(y|x)) + \lambda \sum_i^L (1 - \sum_t^C \alpha_{ti})^2$$

L’attenzione deterministica è parametrizzata e quindi può essere incorporata e modellata per la formazione diretta. Il gradiente può essere ritrasferito attraverso il modulo del meccanismo di attenzione ad altre parti del modello.

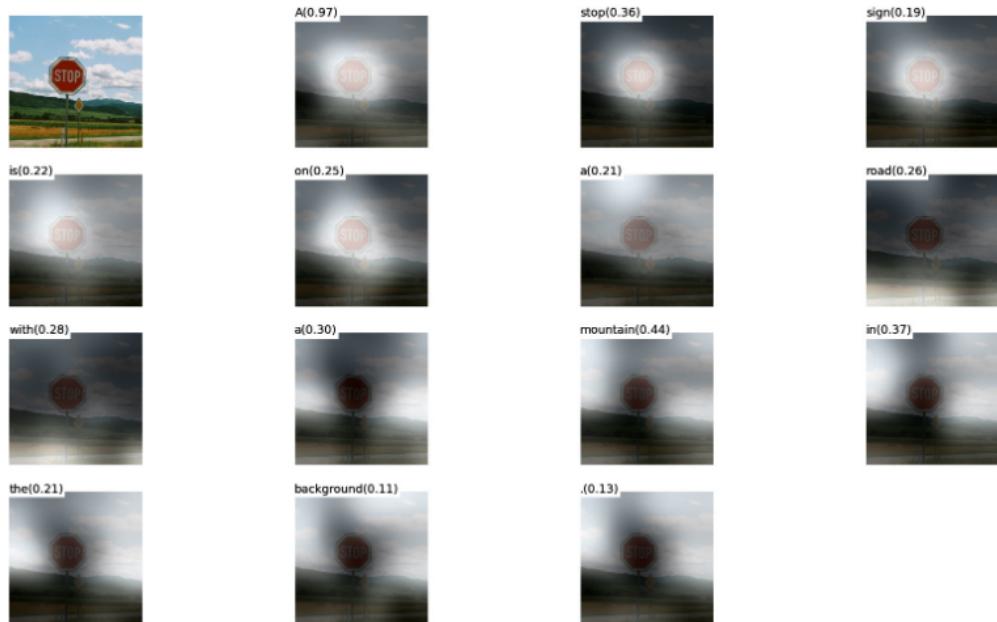


Figura 3.3: Esempio di Soft Attention su una foto di un segnale stradale(8)

Stochastic “Hard” Attention

L’*attenzione stocastica* si concentra solo su una posizione ed è un processo di selezione casuale di una posizione unica. Campiona lo stato nascosto dell’input in base alla probabilità, piuttosto che lo stato nascosto dell’intero codificatore.

Il vettore di contesto $Z_t(8)$ viene calcolato come segue:

$$p(s_{t,i} = 1 | s_{j < t}, a) = \alpha_{t,i}$$

$$\hat{z}_t = \sum_i s_{t,i} a_i$$

dove $s_{t,i}$ indica se selezionare l’i-esima posizione nelle mappe delle caratteristiche L, se selezionata, impostata a 1, altrimenti -1. Per ottenere la retropropagazione del gradiente, è necessario il campionamento Monte Carlo per stimare il gradiente del modulo. Uno svantaggio dell’attenzione forte è che le informazioni vengono selezionate in base al metodo di campionamento massimo o campionamento casuale. Pertanto, la relazione funzionale tra la funzione di loss e la distribuzione dell’attenzione non è realizzabile e non è possibile utilizzare l’addestramento all’algoritmo di backpropagation.

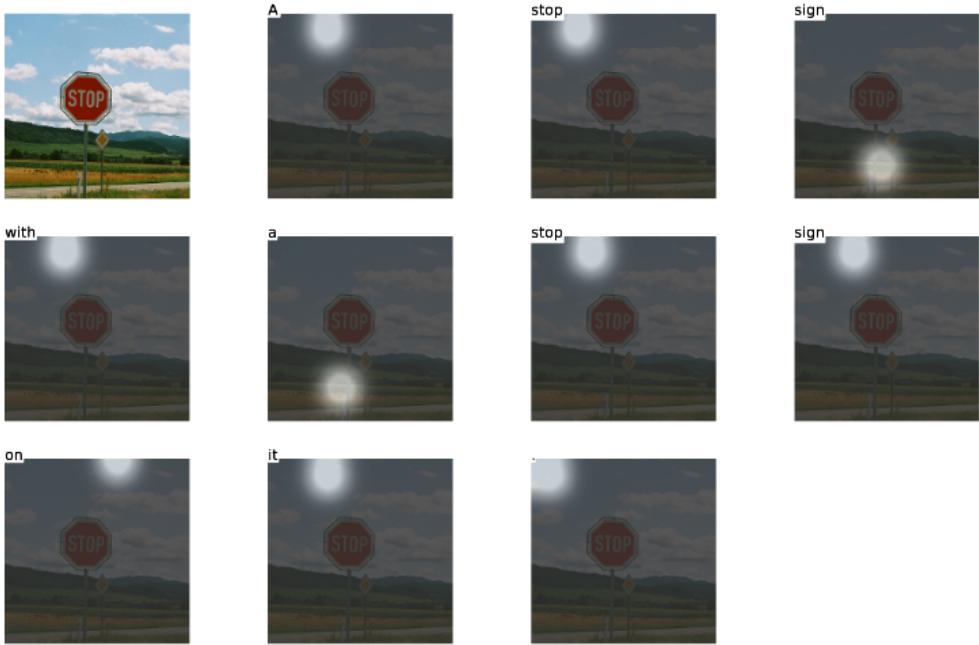


Figura 3.4: Esempio di Hard Attention su una foto di un segnale stradale(8)

3.1.3 Metriche di Valutazione

Nella valutazione dei risultati della generazione di frasi, BLEU (9), METEOR (10), ROUGE (11), CIDEr (12) e SPICE (13) sono generalmente usati come indici di valutazione. La metrica utilizzata per questo progetto è la Blilingual Evaluation Understudy (BLEU) e ne spiego ora brevemente il funzionamento.

La sua idea di base è che più una dichiarazione di traduzione automatica è vicina a una dichiarazione di traduzione professionale umana, migliori saranno le prestazioni. In questa attività viene osservato quante immagini sono equivalenti alle frasi associate.

Il vantaggio di BLEU è che la granularità che considera è un n-gram piuttosto che una parola, considerando informazioni di corrispondenza più lunghe. Lo svantaggio della metrica BLEU è che indipendentemente dal tipo di n-gram abbinato, esso verrà trattato allo stesso modo. Ad esempio, l'importanza dei verbi dovrebbe essere intuitivamente maggiore dell'articolo.

3.2 Il Programma di "Artwork Captioning"

Dopo avere illustrato tutti i funzionamenti alla base del nostro programma, è arrivato ora il momento di vederne l'implementazione pratica. Sarà di seguito mostrato il funzionamento del programma con estratti di codice per comprenderne meglio il significato. La programmazione è stata svolta in linguaggio Python tramite Pytorch. PyTorch è una libreria di apprendimento automatico open source basata sulla libreria Torch, utilizzata per applicazioni come la visione artificiale e l'elaborazione del linguaggio naturale, sviluppata principalmente dal laboratorio di ricerca AI di Facebook.

3.2.1 Descrizione del modello

Il modello implementato, ripreso dal lavoro di Fabio M. Graetz(14), è una rete neurale Encoder-Decoder, rappresentata nel programma dalla classe *ImageCaptionGenerator*:

```

1 class ImageCaptionGenerator(nn.Module):
2     def __init__(self, device, filter_width, num_filters, vocab_size,
3                  emb_sz, out_seqlen, n_layers=3, prob_teach_forcing=1, p_drop=0.3):
4         super().__init__()
5         self.n_layers, self.out_seqlen = n_layers, out_seqlen
6         self.filter_width = filter_width
7         self.num_filters = num_filters
8         self.device = device
9         ...

```

Listing 3.1: Inizializzazione del modello

In seguito sono inizializzati gli elementi principali della rete:

- l'encoder
- il decoder
- la visual attention
- il teacher forcing

Analizziamoli in ordine.

Encoder

L'encoder è così inizializzato nella classe del modello:

```
1 self.encoder = Encoder(device, emb_sz, n_layers, filter_width,
2 num_filters)
```

Listing 3.2: Inizializzazione encoder

Come si può vedere si richiama il costruttore di una classe precedentemente definita come *Encoder*:

```
1 class Encoder(nn.Module):
2     def __init__(self, device, dec_hidden_state_size, dec_layers,
3                  filter_width, num_filters):
4         super().__init__()
5         # Visual Encoder
6         self.device = device
7         self.base_network = nn.Sequential(*list(models.resnet101(
8             pretrained=True).children())[:-2])
9         self.freeze_base_network()
10        self.concatPool = AdaptiveConcatPool2d(sz=1)
11        self.adaptivePool = nn.AdaptiveAvgPool2d((filter_width,
12                                              filter_width))
13        self.filter_width = filter_width
14
15        self.output_layers = nn.ModuleList([
16            fc_layer(2*num_filters, dec_hidden_state_size) for _ in
17            range(dec_layers)
18        ])
```

Listing 3.3: Definizione encoder

L'encoder è implementato dunque come una CNN, più specificamente una ResNet. È formato da una resnet101 preaddestrata, come si può vedere dal parametro *pretrained* = *true*. Ogni layer della ResNet è composto da diversi blocchi. Un'operazione qui si riferisce a una convoluzione, una normalizzazione batch e un'attivazione ReLU a un ingresso, tranne l'ultima operazione di un blocco, che non ha ReLU.

Nell'implementazione di PyTorch si distinguono tra i blocchi che includono 2 operazioni detti Basic Blocks e i blocchi che includono 3 operazioni detti Bottleneck Blocks.

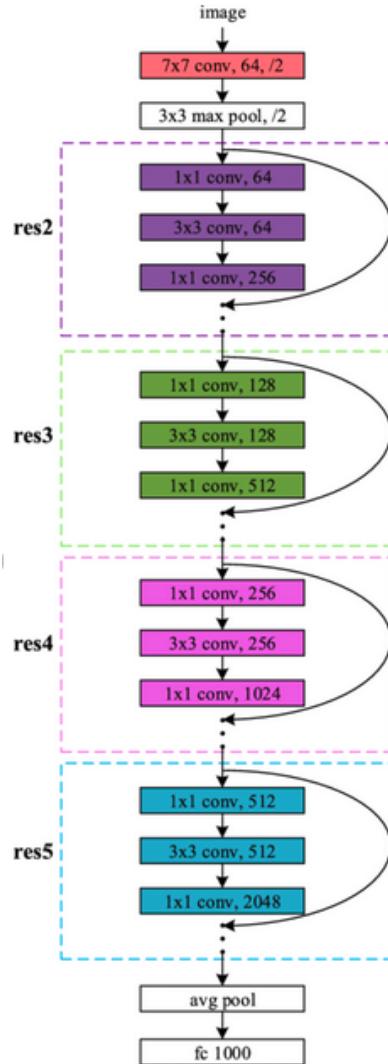


Figura 3.5: Rappresentazione Resnet101

La rete residuale, composta da 101 layers, contiene più di 43 milioni di parametri, anche se un numero molto inferiore rispetto alla VGG16, rete convoluzionale spesso utilizzata, che ne conta circa 138 milioni. Per velocizzare il processo di apprendimento è stata dunque utilizzata una versione preaddestrata su *Imagenet*. ImageNet è un'ampia base di dati di immagini, realizzata per l'utilizzo, in ambito di visione artificiale, nel campo del riconoscimento di oggetti.

Seguono la Resnet dei livelli di Pooling per ridurre drasticamente le dimensioni dei dati, e dei livelli completamente connessi. I *Fully Connected Layers* guardano l'output del livello precedente, che rappresenta le mappe di attivazione di caratteristiche di alto livello, e determina quali di queste sono maggiormente correlate a una particolare classe.

Nella stessa classe sono definite 2 funzioni utili per l'addestramento:

```

1 def freeze_base_network(self):
2     for layer in self.base_network:
3         requires_grad(layer, False)
4
5 def fine_tune(self, from_block=-1):
6     for layer in self.base_network[from_block:]:
7         requires_grad(layer, True)

```

Listing 3.4: Funzioni di congelamento rete e fine tuning

Il metodo *freeze_base_network* comporta il "congelamento" dei layers della rete di base, ovvero della Resnet101, in modo da non apportare variazione nei pesi della rete già allenata ed evitare un peggioramento delle prestazioni dell'encoder.

Il metodo *fine_tune* si occupa invece di un'operazione nota per l'addestramento di reti preaddestrate, ovvero il Fine Tuning.

Esso consiste invece in uno "scongelamento" di alcuni layers della rete precedentemente congelata, tipicamente i blocchi finali. Questa tecnica regola leggermente le rappresentazioni più astratte del modello riutilizzato, per renderle più rilevanti per il problema in questione. Può rendersi particolarmente utile quando l'addestramento sembra non poter migliorare cedendo all'overfitting.

Decoder

In seguito è inizializzato il Decoder, ovvero la RNN che si occupa della generazione della caption relativa all'immagine.

```

1 self.emb = nn.Embedding(vocab_size, emb_sz)
2 self.rnn_dec = nn.GRU(num_filters + emb_sz, emb_sz, num_layers=n_layers
3 , dropout=0 if n_layers == 1 else p_drop) # square to enable
4 weight tying
5 self.out_drop = nn.Dropout(p_drop)
6 self.out = nn.Linear(emb_sz, vocab_size)
7 self.out.weight.data = self.emb.weight.data
8 self.f_b = nn.Linear(emb_sz, num_filters)

```

Listing 3.5: Inizializzazione Decoder

Come si può vedere dal codice, il decoder è formato da una RNN composta da celle GRU. La rete ricorrente è seguita da un layer di dropout, per ridurre il rischio di overfitting, e da un layer lineare densamente connesso che applica la trasformazione lineare $y = xW^T + b$. Il decoder viene attivato dopo l'encoder e la visual attention nella funzione *forward* dell'ImageCaptionGenerator.

Visual Attention

Il metodo di attenzione viene inizializzato così nella classe della rete:

```
1 self.att = VisualAttention(num_filters, emb_sz, 500)
```

Listing 3.6: Inizializzazione Visual Attention

Anche in questo caso come per l'encoder si tratta del costruttore di una classe separata, ovvero della classe *VisualAttention* così definita:

```
1 class VisualAttention(nn.Module):
2     def __init__(self, num_filters, dec_dim, att_dim):
3         super().__init__()
4         self.attend_annot_vec = nn.Linear(num_filters, att_dim)
5         self.attend_dec_hidden= nn.Linear(dec_dim, att_dim)
6         self.f_att = nn.Linear(att_dim, 1)
7
8     def forward(...):
9         ...
```

Listing 3.7: Definizione Visual Attention

Rappresenta una rete di attenzione "soft" o deterministica, e presenta la successione di layers descritta da Xu et al.(8).

Teacher Forcing

L'ultimo elemento da citare per comprendere il funzionamento del modello è il *Teacher Forcing*. Questa è una strategia per l'addestramento di reti neurali ricorrenti che utilizzano l'output del modello da una fase temporale precedente come input. Il Teacher Forcing consiste nell'utilizzo dell'output effettivo o previsto dal set di dati di addestra-

mento nella fase temporale corrente $y(t)$ come input nella fase temporale successiva $X(t+1)$, piuttosto che l'output generato dalla rete.

Facciamo un esempio per rendere il processo più chiaro. Ipotizziamo che al tempo t la RNN abbia costruito la seguente frase:

A portrait of a woman

La rete avrebbe ora trovato la parola *woman*, ma se la frase ideale utilizzata nei dati di training fosse,

A portrait of a nobleman in a dark background

allora il teacher forcing "costringerebbe" la rete a utilizzare come input della RNN al tempo $t+1$ la parola *nobleman*. Questa tecnica aiuta la rete ad apprendere più rapidamente i pattern, velocizzandone la convergenza.

Descritta l'anatomia del modello, possiamo vedere ora come esso si comporta durante la fase di training. Quando chiamato, il modello eseguirà il metodo *forward*. Innanzitutto viene chiamato il metodo *encode*, che richiama a sua volta il metodo *forward* dell'encoder in modo da ottenere i risultati della CNN. Successivamente all'interno di un ciclo for viene girata la RNN tramite il metodo *decode_step*.

```

1 def decode_step(self, dec_inp, h, annotation_vecs):
2     context_vec, alpha = self.att(annotation_vecs, h[-1])
3     beta = torch.sigmoid(self.f_b(h[-1]))
4     context_vec = beta * context_vec
5     emb_inp = self.emb(dec_inp).unsqueeze(0)
6
7     output, h = self.rnn_dec(torch.cat([emb_inp, context_vec.
8         unsqueeze(0)], dim=2), h)
9
10    output = self.out(self.out_drop(output[0]))
11
12    return F.log_softmax(output, dim=1), h, alpha

```

Listing 3.8: Definizione step di decoding

In questa funzione viene eseguita la visual attention, la rete ricorrente e infine il layer di dropout. Essa restituisce come primo output un tensore di probabilità delle parole che potrebbero corrispondere a quella fase di captioning. Eseguito lo step di decoding la

funzione sceglie la parola dell'input successivo, tenendo conto del teacher forcing tramite una probabilità di esecuzione:

```

1 if (dec_inp == 1).all() or (y is not None and i >= len(y)):
2     break
3 elif y is not None and (self.prob_teach_forcing > 0) and (random.random()
4     () < self.prob_teach_forcing):
5     dec_inp = y[i].to(self.device)
6 else:
7     dec_inp = dec_output.data.max(1)[1]

```

Listing 3.9: Utilizzo del Teacher Forcing

Si può vedere infatti che se un numero random risulta minore della probabilità di teacher forcing impostata, allora verrà scelta la parola dell'output corretto preso dai dati di training, altrimenti verrà scelta la parola con probabilità più alta nei dati di output della RNN.

3.2.2 Addestramento della rete

Vediamo adesso come la rete è stata addestrata e quali tecniche sono state utilizzate. Sono stati preparati 2 datasets, uno per un pre-addestramento della rete e uno per il training effettivo sulle opere d'arte. Diamo dunque subito un'occhiata a come questi siano stati creati e elaborati.

Preparazione dei Datasets

Innanzitutto è stato utilizzato un dataset fotografico per allenare la rete su concetti base come la sintassi delle frasi o il riconoscimento di oggetti semplici. Il dataset utilizzato è *Flickr30k*. Flickr30k è uno dei set di dati di image-captioning più famoso e utilizzato, contenente, come si può intuire dal nome, oltre trentamila immagini, ognuna associata a 5 descrizioni. Più specificamente esso è diviso in 29000 immagini per il training, 1014 immagini di validazione e 1000 immagini di test. Per l'allenamento concentrato invece sulle opere d'arte è stato creato manualmente un dataset separato che conta di 150 immagini, ognuna associata a due descrizioni.

L'elaborazione dei dati è stata fatta attraverso un file json, dalla struttura simile a quello di flickr30k, in modo da poter utilizzare le medesime funzioni per estrarre le descrizioni e i nomi delle immagini per entrambi i dataset. JavaScript Object Notation (JSON)(15) è un semplice formato per lo scambio di dati. Per le persone è facile da leggere e scrivere, mentre per le macchine risulta facile da generare e analizzarne la sintassi. Si basa su un sottoinsieme del Linguaggio di Programmazione JavaScript. Esso si basa principalmente sull'organizzazione dei dati tramite strutture universali quali gli array e le coppie nome:valore.

Il dataset artististico si trova dunque nel file ArtDataset.json così strutturato:

```

1  {
2      "images": [
3          {"sentids": [0, 1],
4              "imgid": 0,
5              "sentences": [
6                  {"raw": "First caption",
7                      "imgid": 0,
8                      "sentid": 0},
9                  {"raw": "Second caption",
10                     "imgid": 0,
11                     "sentid": 1}],
12                     "split": "train", "filename": "0.jpg"},
13                     {"sentids": [2, 3],
14                     ...
15                     }
16                     ...
17     ]
18 }
```

Listing 3.10: Struttura del Dataset nel file json

Il file JSON presenta dunque un array di immagini rappresentate principalmente da:

- un "imageid"
- un array di "sentences" che rappresentano le descrizioni
- un valore di "split" che si occupa della suddivisione tra training, validation e test
- un "filename"

Compresa la struttura dei file json, possiamo ora vedere l'elaborazione dei dati fino alla generazione dei dataloader che verranno utilizzati dalla rete. Prima di tutto vengono

caricati i file, vengono suddivise le immagini in tre vettori in base al loro valore di split e vengono create delle tuple composte dal filename e da un vettore di captions in modo da poter gestire meglio i dati:

```

1 dataset_json = json.load((PATH/'dataset_flickr30k.json').open())
2
3 train = [d for d in dataset_json["images"] if d["split"] == "train"]
4 ...
5
6 train_fns_caps = [(d["filename"], [c["raw"] for c in d["sentences"]])
7                     for d in train]
8 ...

```

Listing 3.11: Split del dataset

Le descrizioni appena memorizzate vanno però trasformate in vettori di numeri, dove ogni numero corrisponde ad una parola. Per questo è necessario un *Vocabolario*, che mappi ogni parola con un valore numerico. Per praticità vengono scelte solo le parole che si ripetono almeno 2 volte in modo da escludere vocaboli rari e inusuali, i quali non solo amplierebbero il numero di parametri della rete, ma risulterebbero difficili da imparare e da ritrovare negli esempi di testing. Le descrizioni delle immagini di training vengono perciò suddivise in singole parole o *tokens* tramite un *Tokenizer* e mappate in una variabile *Vocab* che verrà opportunamente salvata in un file .pickle in modo da poterla riutilizzare in futuro.

```

1 tokenizer = Tokenizer(n_cpus=1)
2 train_captions_tokenized = tokenizer.process_all(
3     list(itertools.chain.from_iterable(list(zip(*train_fns_caps))[1]))
4 )
5
6 vocab = Vocab.create(train_captions_tokenized, max_vocab=50000,
7     min_freq=2)
8 pickle.dump(vocab, open(PATH/"vocab.pkl", 'wb'))

```

Listing 3.12: Creazione del Vocabolario

A questo punto tramite il metodo *build_data* ogni tupla viene elaborata in modo da generarne una nuova composta dal nome del file e da un vettore di captions, stavolta tradotte numericamente tramite il metodo *numericalize_tokens*. Per ogni immagine un ciclo for analizza tutte le captions relative, spartendole tramite il tokenizer e traducen-

dole tramite la funzione precedente. Le nuove tuple vengono salvate anch'esse in dei file pickle per un utilizzo futuro. In questo caso l'operazione è molto consigliata poiché per datasets molto grandi questa operazione può richiedere tempi elevati.

```
1 %time build_data(train_fns_caps, PATH, "train")
2 train_data = pickle.load((PATH/"valid.pkl").open('rb'))
```

Listing 3.13: Tokenizzazione delle captions

L'elaborazione dei dati iniziali è quasi al termine. Rimane solamente creare dei *Dataloader* da passare alla rete neurale. Un Dataloader rappresenta un iteratore Python su un dataset, che presenta comode funzionalità come il batching automatico e la modifica dell'ordine di loading dei dati. L'argomento più importante da passare a un dataloader è il dataset. Esso è reappresentato dal programma dalla classe *ImageCaptionDataset*:

```
1 class ImageCaptionDataset(Dataset):
2     def __init__(self, data, transform=None):
3         self.filenames = data[0]
4         self.captions = data[1]
5         self.transform = transform
6         self.mult_captions_per_image = not isinstance(train_data
7 [1][0][0], int)
8
9     def __len__(self):
10        return len(self.filenames)
11
12    def __getitem__(self, idx):
13        ...
14        return (image, caption)
```

Listing 3.14: Definizione della classe ImageCaptionDataset

Questa classe estende la classe di pytorch *Dataset* la quale presenta due metodi principali: uno per ricevere la grandezza del dataset e uno che rende un valore del dataset a seconda di un indice. Se sono state passate delle trasformazioni tramite la variabile *transform* allora il metodo *__getitem__* restituirà l'immagine trasformata e una casuale caption ad essa relativa. Tramite questo metodo è facilmente applicabile dunque una data-augmentation "infinita", poichè ogni immagine prima di essere analizzata dalla rete verrà trasformata casualmente. Questo permette, almeno teoricamente, di non dare mai in pasto alla rete due immagini identiche, prevenendo l'overfitting.

Al Dataloader vengono inoltre passati un *Sampler*, il *batch_size* e la funzione *collate_fn*. Un Sampler è una classe utilizzata per specificare la sequenza di indici utilizzati nel caricamento dei dati. Rappresentano dunque oggetti iterabili sugli indici dei dataset. Il *SortSampler* scorre i dati di testo in ordine di lunghezza ed è spesso utilizzato per i dataset di validazione. Il *SortishSampler* invece passa attraverso i dati di testo in ordine di lunghezza con un po' di casualità, ed è un campionatore spesso utilizzato per i dati di training. Il batch_size rappresenta chiaramente la grandezza dei batches, mentre la funzione collate_fn raccoglie i campioni, aggiunge il padding e se necessario, inverte l'ordine dei tokens.

```

1 ImageCaptionDataset(train_data, transform=valid_tfms)
2 trn_sampler = SortishSampler(train_data[1], key=lambda x: len(
3     train_data[1][x]), bs=bs)
4 trn_dl = DataLoader(dataset=train_dataset, batch_size=bs, sampler=
5     trn_sampler, collate_fn=imgcap_collate_func)

```

Listing 3.15: Creazione Dataloader

Inizializzati i Dataloader non resta altro che riunirli in un DataBunch e passarli a un Learner (classe che racchiude tutti gli elementi necessari al training di una NN), insieme al modello di rete, alla funzione di loss, all'ottimizzatore e alle metriche di valutazione:

```

1 dataBunch = DataBunch(train_dl=trn_dl, valid_dl=val_dl, device=gpu,
2     path=PATH, collate_fn=imgcap_collate_func)
3 learn = Learner(data=dataBunch, model=to_device(imgCapGen, gpu),
4     opt_func=opt_fn, loss_func=ImageCaptionLoss, callback_fns=[
5         ShowGraph], metrics=[BleuMetric()])

```

Listing 3.16: Inizializzazione del Learner

Come funzione di loss viene utilizzata una Negative Log Likelihood (NLL) per confrontare la caption predetta e opportunamente ridimensionata con quella corretta. Come ottimizzatore viene utilizzato il Metodo Adams e come metrica la BLEU.

Fase di Pre-Training

Le fasi di training sono composte da due funzioni in ripetizione. La prima è *find-appropriate_lr* e quale si occupa di trovare il punto di massima decrescita nella funzione

dei valori di loss individuata da *lr_find* (metodo del learner di pytorch). La ricerca del learning rate è importante per poter decidere l'impatto che si ha ogni volta sulla modifica dei pesi della rete. E' giusto cominciare con learning rate più alti, per poi abbassarsi quando si pensa di avere raggiunto l'ottimo.

```
1 lr = find_appropriate_lr(learn, plot=True)
```

Listing 3.17: Metodo di ricerca del learning rate

La seconda *fit_one_cycle* è invece la funzione di allenamento vera e propria che passa i dati attraverso la rete e calcola la funzione di loss riportando i risultati in una tabella che mostra:

- numero di epoca
- valore di loss per i dati di training
- valore di loss per i dati di validazione
- valore della metrica
- tempo impiegato

Bisogna stare attenti a controllare che i valori di loss di training non scendano troppo rispetto a quelli di validazione, poichè questo significa spesso che la rete si sta adattando troppo ai dati di allenamento, smettendo di generalizzare. Per questo vengono fatti dei salvataggi del learner ogni volta che esso porta risultati migliori. In modo da poter tornare indietro in caso ci si trovi inaspettatamente in una situazione di overfitting dalla quale solitamente è difficile uscire.

```
1 learn.fit_one_cycle(num_epoche, lr, callbacks=[  
2     SaveModelCallback(learn, monitor='bleu_metric'),  
3     CSVLogger(learn, filename='train_img_cap_gen')  
4 ])
```

Listing 3.18: Metodo training della rete

Per iniziare sono stati inseriti nel databunch i dati fotografici di flickr30k e sono stati allenati per una ventina di epoche, poichè i risultati mostrati sono stati subito convincenti e adatti per il pre-addestramento che la nostra rete doveva subire. Vediamo dunque come è avvenuto questo allenamento.

per prima cosa cerchiamo ogni volta il lr appropriato aiutandoci con il metodo precedentemente descritto:

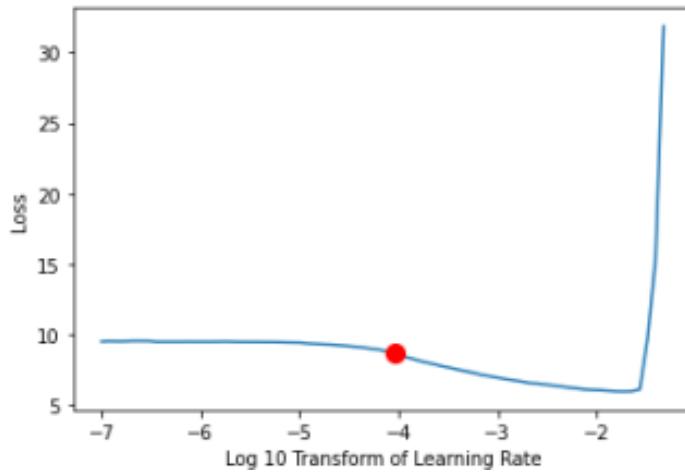


Figura 3.6: Prima ricerca del lr sui dati di flickr30k.

Dopodichè alleniamo la rete con serie di 5 epoche fino a raggiungere dei valori di loss abbastanza bassi, monitorando i casi di overfitting:

Tabella 3.1: Risultati delle ultime 5 epoche di pre-allenamento della rete.

epoch	train_loss	valid_loss	bleu_metric	time
0	1.731756	1.412989	0.054964	09:22
1	1.543743	1.526991	0.062434	09:14
2	1.434996	1.245779	0.071799	09:15
3	1.414266	1.326153	0.077977	09:15
4	1.378954	1.326028	0.079181	09:18

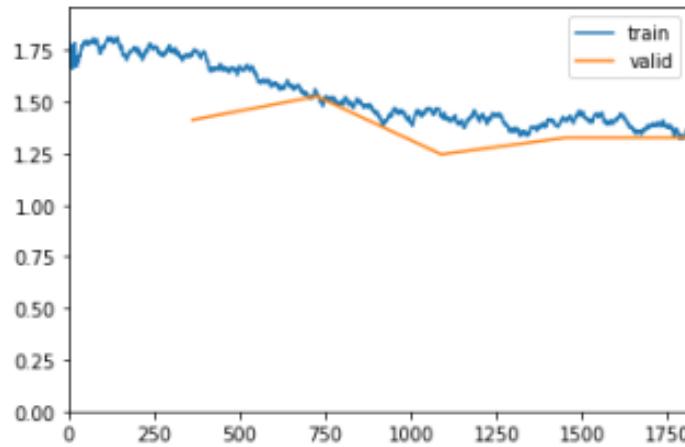


Figura 3.7: Grafico dei valori di loss per le ultime 5 epoche di pre-allenamento della rete.

La figura 3.7 mostra i valori di loss dopo 20 epochi di training senza data augmentation. I risultati dopo questo breve allenamento appaiono già ottimi per i nostri obiettivi come si può vedere nelle immagini seguenti. Questo è dato soprattutto dal fatto che il riconoscimento di elementi fotografici è facilitato dall'utilizzo della ResNet preaddestrata.

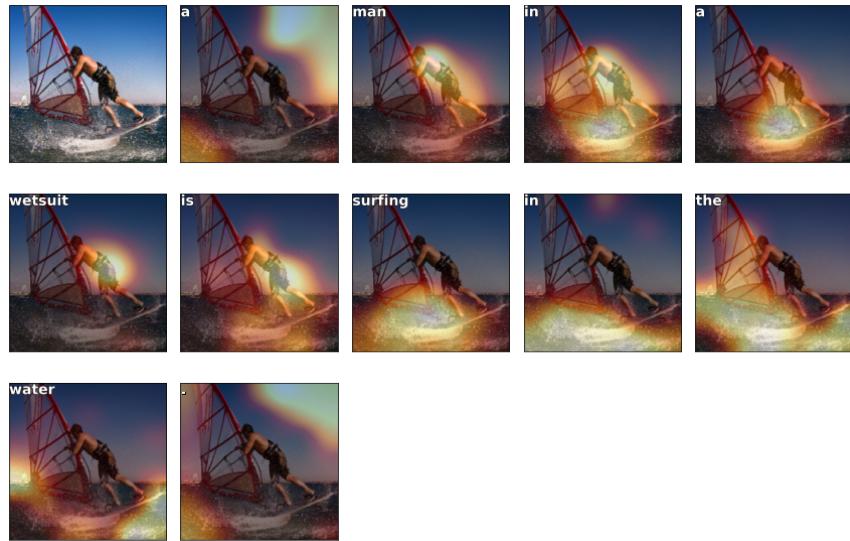


Figura 3.8: Esempio di Testing della rete preallenata su dati fotografici(1)



Figura 3.9: Esempio di Testing della rete preallenata su dati fotografici(2)

Fase di Testing

Prima di passare all'addestramento sul secondo dataset descriviamo come avviene il test della rete, per verificarne i risultati effettivi. Innanzitutto è opportuno fare lo switch dei

layers di dropout per adattarli alla fase di testing. Questo è possibile tramite il metodo *eval*:

```
1 learn.model.eval()
```

Listing 3.19: Switching dei layers di dropout

A questo punto viene passata un’immagine opportunamente vettorizzata e dimensionata a una *Beam Search*, un algoritmo di ricerca basato su euristiche che esplora un grafo espandendo il nodo più promettente in un insieme limitato di nodi. In pratica viene mandata in input l’immagine nella rete e vengono prese in considerazioni le k parole più probabili risultate da ogni passaggio della rnn, fino a che non viene raggiunta una lunghezza massima delle frasi o tutte e k quelle considerate non sono terminate naturalmente. Viene infine scelta la più corretta fra esse, o le n frasi più corrette a seconda del parametro *num_results*.

```
1 beam_width = k
2 beam_search = BeamSearch(art_learn.model.encode, art_learn.model.
                           decode_step, beam_width)
3 results = beam_search(image)
```

Listing 3.20: Applicazione della BeamSearch

Il risultato del metodo è dunque un tensore di valori rappresentanti delle parole del nostro vocabolario. Bisogna dunque tradurli e mostrare la frase sia testualmente sia tramite le aree di concentrazione della soft attention nei vari step della RNN. Questo viene fatto dalla funzione *visualize_attention*, la quale prende i valori di attenzione provenienti dai risultati della beam search e stampa un’immagine per ogni parola della frase facendo notare su quale parte dell’immagine si sia concentrata la rete per dare quel responso.

```
1 print(vocabulary.textify(results[0]))
2 visualize_attention(im, results[0], results[1], denorm, vocabulary,
                      att_size=7, sz=sz, thresh=0.02)
```

Listing 3.21: Visualizzazione della caption e dell’attenzione

Come si può vedere dalla figura 3.8 visualizzare graficamente l’attenzione può essere molto utile per capire come la rete interpreti i pattern, dove si focalizzi principalmente e anche se mostra qualche principio di overfitting. In questo caso i risultati sono buoni e

si può notare da come le parole *man*, *surfing* e *water* siano effettivamente associate a zone dell'immagine pertinenti e ben delineate. Tutti questi aspetti positivi potrebbero indurci a pensare che dunque l'addestramento sia terminato e che la rete possa dare risultati gratificanti anche per immagini di ambito culturale. Così purtroppo non è e il motivo principale è che i pattern che spesso vengono analizzati dalle reti convoluzionali sono molto diversi da quelli presenti in foto di quadri o statue, specialmente per la frequente mancanza di realismo o elementi animati. Inoltre va considerato che nemmeno la costruzione sintattica delle descrizioni è simile a quelle di nostro interesse, come anche i vocaboli utilizzati. Questo è possibile vederlo dagli esempi seguenti:

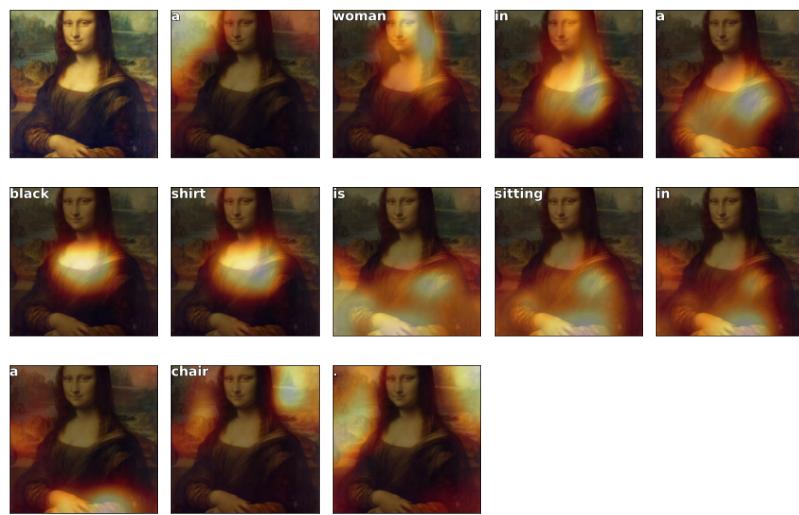


Figura 3.10: Esempio di Testing della rete preallenata sulla Monnalisa(1)

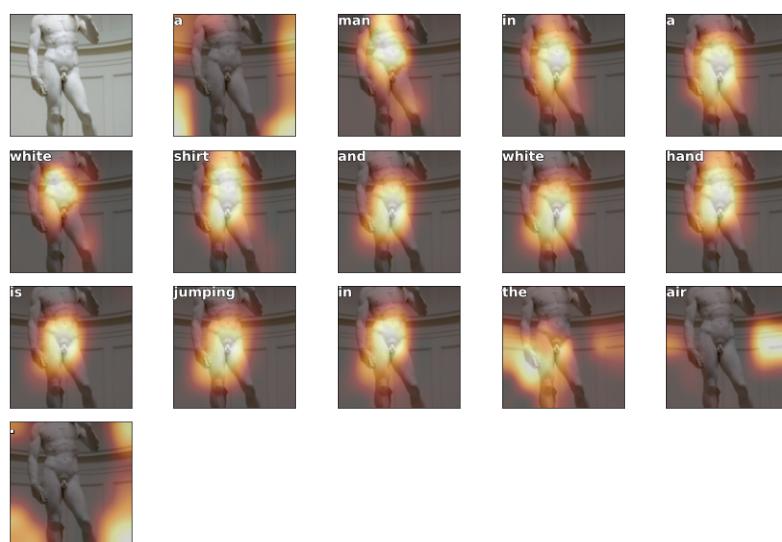


Figura 3.11: Esempio di Testing della rete preallenata sul David di Michelangelo(2)

Come si può vedere dalle figure 3.11 e 3.12 alcuni pattern vengono effettivamente riconosciuti come la presenza di un uomo e di una donna o il riconoscimento di determinati colori. Mancano però tutte le identificazioni di tipo culturale, come la distinzione tra quadro e statua, la presenza di nudità o la descrizione del soggetto. E' per questo che ora la rete verrà allenata una seconda volta su un dataset differente, il quale presenta 150 esempi di statue e quadri.

Seconda fase di Training

Per la seconda fase di training è stato utilizzato un dataset costruito personalmente. Le dimensioni di esso non sono però minimamente soddisfacenti per un corretto allenamento di una rete. Fortunatamente esistono delle tecniche per aiutare la rete ad apprendere nuovi pattern anche da datasets molto piccoli. Queste tecniche sono principalmente la *data augmentation* e i *layers di dropout*.

Per vedere come la data augmentation possa effettivamente contribuire ad un migliore allenamento della rete, essa verrà addestrata in primis senza l'utilizzo di tale tecnica e diversamente in seguito. La nostra fortuna è che con un dataset così piccolo l'allenamento della rete è molto veloce, anche in caso di un elevato numero di epoche. Dunque è possibile effettuare molti test per stabilire quando la rete vada in overfit e quali siano le migliori tecniche di data augmentation per il nostro scopo.

Tabella 3.2: Risultati di 5 epoche di allenamento senza DA.

epoch	train_loss	valid_loss	bleu_metric	time
0	2.265495	2.984629	0.146926	00:03
1	2.213262	2.915236	0.153221	00:03
2	2.167023	2.759534	0.163358	00:03
3	2.143535	2.858844	0.164633	00:03
4	2.093411	2.706269	0.154933	00:03

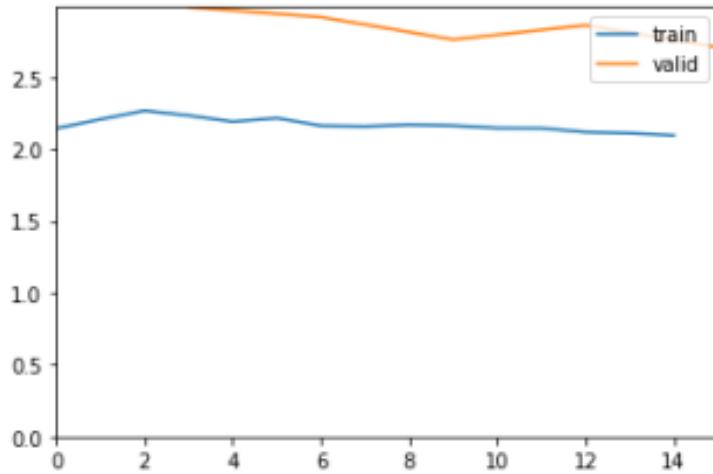


Figura 3.12: Grafico dei valori di loss di 5 epochi di allenamento senza DA.

La tabella 3.2 mostra che già dopo poche epochi (circa 20) i valori di loss per la validazione risultano già superiori di quelli di training, anche se comunque in decrescita. La metrica però mostra ottimi risultati principalmente per la decisione personale di allenare la rete con frasi che iniziassero con pattern predefiniti come "a portrait of..." oppure "a marble statue of...", i quali vengono appresi velocemente dal modello. Nelle successive 20 epochi però la situazione peggiora e si inizia a perdere la decrescita nei valori di loss di validazione, chiaro sintomo di perdita di generalizzazione e dunque di overfitting.

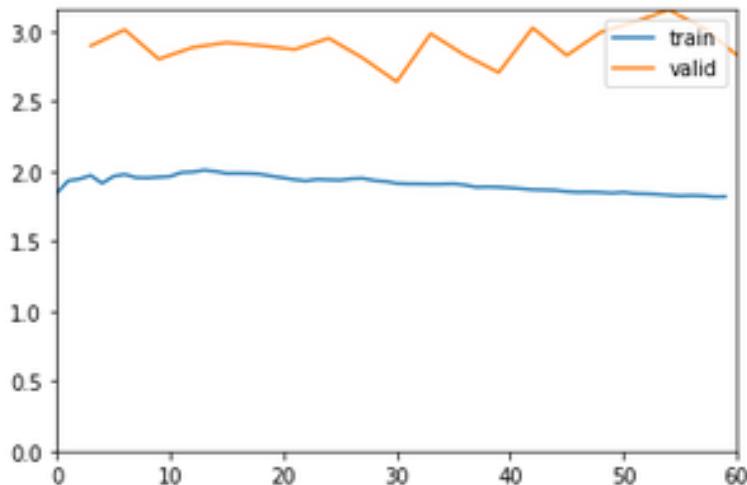


Figura 3.13: Grafico dei valori di loss per le successive 20 epochi di allenamento senza DA.

La prematura comparsa di sintomi di overfitting è dato principalmente dall'utilizzo di un piccolo dataset senza implementarlo con una DA. Nonostante questo dalla fase di

testing possiamo notare che ci sono dei netti miglioramento nelle interpretazioni culturali. Facciamo dunque un paragone con le medesime immagini utilizzate in precedenza:



Figura 3.14: Esempio di Testing della rete allenata senza DA sulla Monnalisa(1)

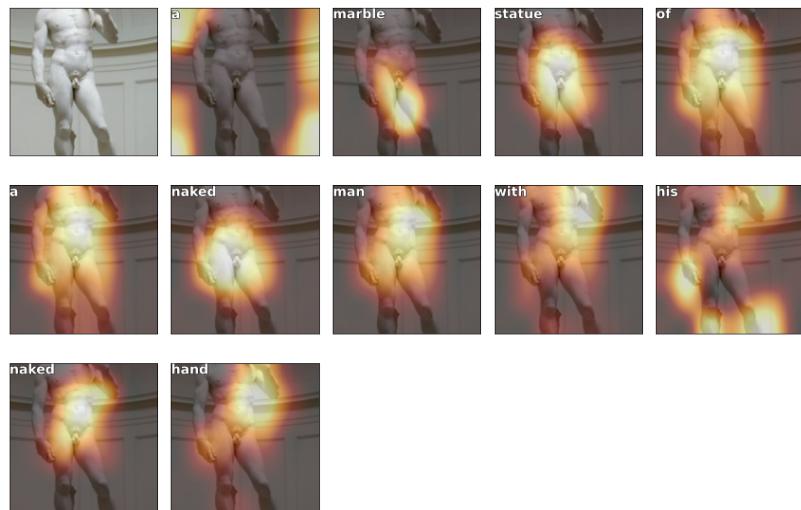


Figura 3.15: Esempio di Testing della rete allenata senza DA sul David di Michelangelo(2)

Le captions generate risultano dunque migliori, e questo è già un buon risultato. Si notano però delle cose da migliorare. Innanzitutto la struttura logica della frase non è corretta nel secondo esempio e inoltre le aree di attenzione denotate dalla rete non sono molto precise, spesso quasi casuali. Questo indica chiaramente che la rete ha imparato dei pattern specifici e li riporta per "fiducia", ma questo comportamento può spesso risultare scorretto e andrebbe sistemato riducendo l'overfitting. Facciamo dunque un nuovo allenamento, adoperando adesso una data augmentation intensiva, in modo da ampliare il più possibile in nostro dataset.

Da un'indagine iniziale su 100 epochhe, si può notare come la rete vada in overfit dopo circa un quinto delle stesse, anche con l'ausilio della DA::

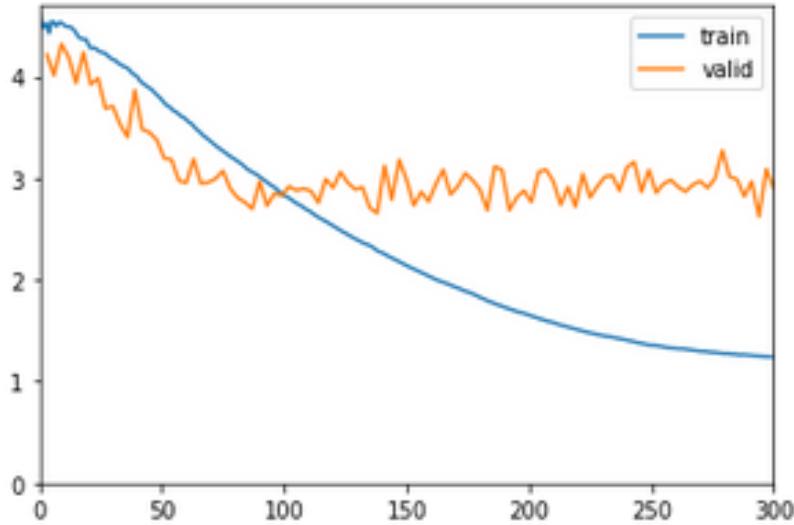


Figura 3.16: Analisi dei valori di loss su 100 epochhe senza DA.

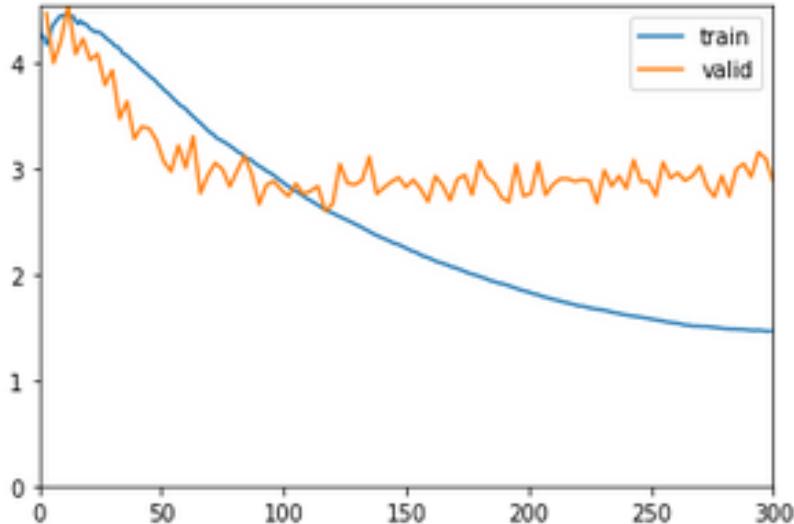


Figura 3.17: Analisi dei valori di loss su 100 epochhe con DA.

Quindi è evidente che un'aggressiva DA, composta principalmente da operazioni di cropping, flip e rotazioni, non basti a rendere la rete meno incline all'overfitting, anche se la curva dei valori di loss risulta meno rumorosa e la decrescita più duratura. La rete è stata comunque allenata con l'alternazione di learning rates alti e bassi e con l'implementazione del fine tuning. Ma per ottenere risultati migliori è opportuno accrescere maggiormente il dataset, sia per i dati di training, dai quali la rete può apprendere

meglio i pattern, sia per i dati di valutazioni, i quali potrebbero mostrare pattern anche completamente differenti da quelli a cui il modello è abituato.

Capitolo 4

Mobile App

Sviluppo di applicazione mobile che utilizzi la rete neurale precedentemente descritta su foto scattate da telefono.

In questo capitolo verrà descritta la realizzazione di un'applicazione per telefono. Lo scopo era utilizzare come backend il codice Python senza dovere fare trascrizioni in altri linguaggi e farlo interagire con un frontend in Dart rappresentato la parte "materiale" dell'applicazione. L'interazione avviene tramite un sistema client-server mediante Ngrok. Lo schema seguente mostra come è strutturato il progetto:

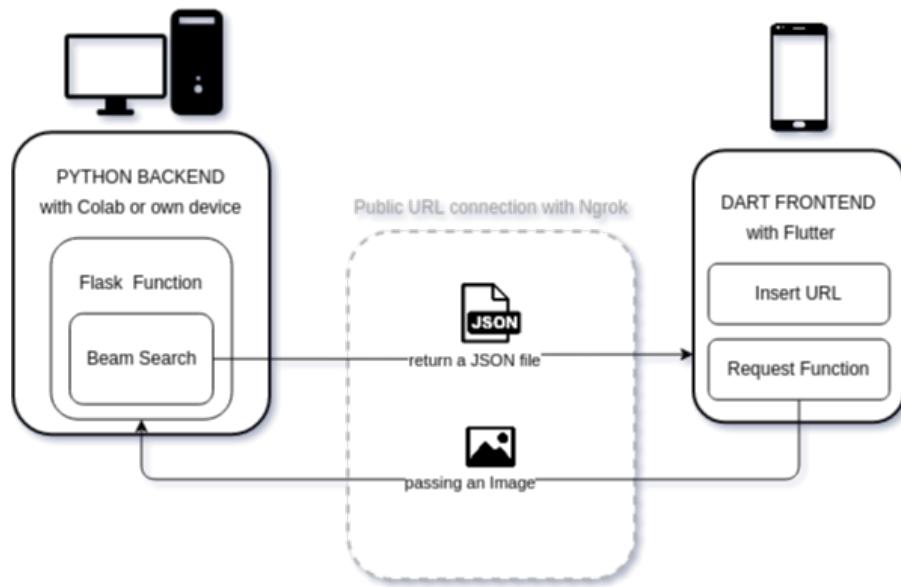


Figura 4.1: Diagramma di interazione tra frontend e backend.

Spieghiamo dunque ora come è stata strutturata l'app e in seguito come interagisca con il backend in python visto precedentemente, analizzando nel dettaglio i blocchi dello schema in figura. Il frontend sarà dunque costituito da files dart che descrivano mediante dei *widget* la struttura dell'applicazione e le funzioni di cui essa dispone.

4.1 Sviluppo Applicazione con Flutter

L'applicazione è stata sviluppata su AndroidStudio tramite l'utilizzo di *Flutter*. Flutter è un framework open source sviluppato da Google per la produzione di interfacce native Android e iOS. Esso è infatti scritto in *Dart*, un linguaggio di programmazione sviluppato anch'esso da Google per la creazione di app su qualunque piattaforma con lo scopo di sostituire Javascript.

L'applicazione deve essere in grado di adempiere ai seguenti compiti:

- Inserire l'URL per il collegamento al backend
- Scattare una fotografia (mostrando a video la fotocamera)
- Inviare la foto al backend
- Mostrare la foto scattata e la caption calcolata dalla rete

4.1.1 Struttura dell'applicazione

L'applicazione è strutturata in una pagina singola, che sarà dunque immediatamente presente all'apertura. Essa presenta una semplice Navbar con il nome dell'applicazione, un body centrale dove sarà mostrato l'output della fotocamera, un pulsante per inserire l'url e un pulsante per scattare la foto.

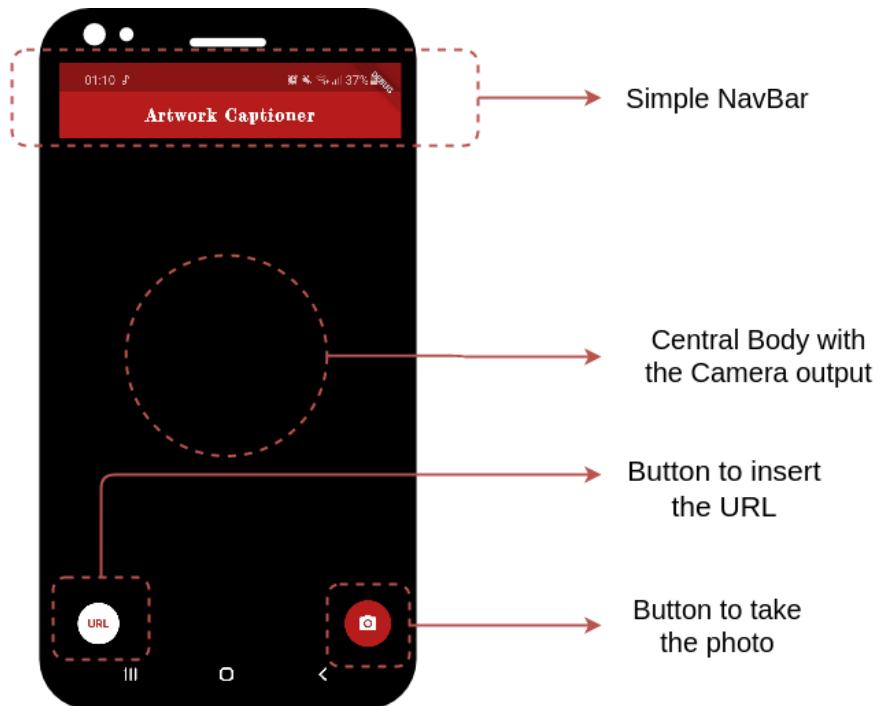


Figura 4.2: Pagina principale dell'applicazione.

Per creare questa semplice pagina è necessario utilizzare uno StatelessWidget chiamato *MyApp*, all'interno del quale sia però presente uno StatefulWidget, che ne descrive l'home page. La caratteristica *stateful* specifica il fatto che questo widget può cambiare nel tempo, modificando i suoi parametri.

```

1 class MyApp extends StatelessWidget {
2   final CameraDescription camera;
3
4   MyApp({this.camera});
5
6   @override
7   Widget build(BuildContext context) {
8     return MaterialApp(
9       title: 'Flutter Demo',
10      home: MyHomePage(title: 'Flutter Home Page', camera: camera),
11    );
12 }

```

Listing 4.1: Definizione Widget principale

L'unico parametro che viene passato attraverso le classi è *camera*, il quale rappresenta la fotocamera del telefono da utilizzare, e andrà passata nella classe main al costruttore di *MyApp* in questo modo:

```

1 void main() async {
2   WidgetsFlutterBinding.ensureInitialized();
3   final cameras = await availableCameras();
4   final firstCamera = cameras.first;
5
6   runApp(MyApp(camera: firstCamera));
7 }

```

Listing 4.2: Classe Main

La classe *MyHomePage* rappresenta la struttura effettiva della pagina ed essendo uno stateful widget la classe *_MyHomePageState* è allegata ad essa per descriverne lo stato e come esso varia. Il widget *build* ritorna una Scaffold costituita da tre elementi principali:

- Un' *AppBar* che rappresenta la navbar con il titolo
- Un *body* dove si vedrà l'output della fotocamera
- Due pulsanti per l'inserimento dell'url e lo scatto

La classe presenta inoltre varie funzioni, che servono principalmente per l'interazione e la modifica della pagina stessa. La costruzione dell'navbar è molto semplice, trattandosi solo di un testo, al quale vengono applicati degli stili. Il body ha un comportamento dinamico. Se la fotocamera è pronta all'uso e la variabile *_CameraOn* è vera, allora

mostrerà l'output della fotocamera, altrimenti caricherà una pagina di loading. La scelta dell'inserimento di questa variabile di decisione deriva dal fatto che una volta scattata la foto, l'applicazione dovrà attendere la risposta dalla rete neurale. Continuare a mostrare il video della fotocamera risulta fastidioso e non dà l'idea di un corretto funzionamento. In questo modo l'utente può rendersi conto di aver scattato la foto e aver messo in moto la rete neurale proprio da questa pagina di caricamento.

```
1 if (snapshot.connectionState == ConnectionState.done && _CameraOn) {
2     return CameraPreview(_controller);
3 } else {
4     return Center(
5         child: Container(
6             child: Padding(
7                 padding: EdgeInsets.fromLTRB(50, 0, 0, 50),
8                 child: Image.asset(
9                     "images/pencil.gif",
10                    height: 100,
11                    width: 100,
12                ),
13            )));
14 }
```

Listing 4.3: Gestione del body dell'HomePage

Vediamo ora gli elementi più funzionali della pagina, ovvero i due pulsanti. Il primo, quello mostrato a sinistra in figura 4.2, ha il compito di inserire l'Url a cui poi verrà fatta la richiesta. Il suo metodo *onPressed*, apre una finestra di dialogo dove è possibile inserire del testo (tramite tastiera richiamata dal pulsante nascosto della barra di scrittura) e farne il "submit" in modo da registrare il valore nella variabile *url* della classe.

```
1 RawMaterialButton(
2     onPressed: () {
3         TextEditingController customController = TextEditingController();
4         showDialog(
5             context: context,
6             builder: (context) {
7                 ...
8             }).then((value) => setState(() {url = value;}));
9 }
```

Listing 4.4: Inserimento dell'url tramite pulsante

Per modificare il valore della variabile di una classe è necessario usare il metodo *setState*, come si può vedere dal codice sopra.

Il secondo pulsante, quello mostrato a destra in figura 4.2, è l'elemento più importante poichè ha lo scopo di scattare la foto, inviarla al backend e mostrare il risultato all'utente. Per prima cosa il metodo *onPressed* di questo pulsante, dopo essersi assicurato che la camera sia disponibile, crea un percorso per il salvataggio della foto scattata, la quale per semplicità viene nominata con la data dello scatto. A quel punto scatta la foto e la salva nel percorso creato in precedenza tramite il metodo *takePicture*.

```
1 FloatingActionButton(  
2     child: Icon(Icons.camera_alt),  
3     backgroundColor: Colors.red[900],  
4     onPressed: () async {  
5         try {  
6             await _initializeControllerFuture;  
7             final path = join(  
8                 await getTemporaryDirectory().path,  
9                 '${DateTime.now()}.png',  
10            );  
11            await _controller.takePicture(path);  
12            setState(() { imgPath = path; });  
13            ...  
14        } catch (e) {  
15            print(e);  
16        }  
17    },  
18 )
```

Listing 4.5: Codice per scattare la fotografia

Dopodichè viene controllato se è stato inserito l'url tramite il metodo *isURL*. In caso viene impostato *_CameraOn* a False, in modo da poter renderizzare la schermata di loading al posto della fotocamera. Fatto questo viene chiamato il metodo *uploadImage*:

```
1 Future<http.Response> uploadImage(filename) async {  
2     var request = http.MultipartRequest('POST', Uri.parse(url));  
3     request.files.add(await http.MultipartFile.fromPath('image',  
4         filename));  
5     var streamedRes = await request.send();  
6     var res = http.Response.fromStream(streamedRes);  
7     return res;  
8 }
```

Listing 4.6: Dichiarazione del metodo *uploadImage* per fare richiesta all'url

Con il suddetto metodo inviamo all'url una richiesta passandogli l'immagine appena creata e salviamo la risposta in una variabile res, che ci aspettiamo contenga un file json da decodificare (come mostrato in figura 4.1). La richiesta viene gestita dal backend, come vedremo successivamente, mentre dal lato frontend, una volta ricevuta la risposta, ne controlliamo lo *statusCode*.

```
1 if (resp.statusCode == 200) {  
2     final Map parsed = json.decode(resp.body);  
3     setState(() {  
4         imgPath = path;  
5         BC.update(parsed["caption"] as String, "5");  
6         Navigator.push(  
7             context,  
8             MaterialPageRoute(  
9                 builder: (context) =>  
10                    DisplayPictureScreen(imagePath: path),  
11            ),  
12        );  
13    });  
14 }
```

Listing 4.7: Parsing del JSON e apertura del display del risultato

Se esso risulta corretto (ovvero =200) possiamo parsare i dati, fare l'update dell'oggetto *BeamCaption* della classe con i dati parsati e aprire una schermata di risultato per l'utente rappresentata dalla classe *DisplayPictureScreen*. Questa nuova classe è un widget costituito da un'appbar, un body raffigurante la foto scattata e finestra soprastante (oggetto della classe *GridElement*) che include il testo della caption e la profondità della BeamSearch. Se torniamo indietro alla pagina principale è necessario che il body si aggiorni e torni a mostrare l'output della fotocamera, perciò si aggiorna lo stato dell'homepage riportando il valore di *_CameraOn* a True.

4.1.2 Utilizzo dell'applicazione

Per utilizzare l'applicazione bisogna seguire un iter specifico, anche se intuitibile, costituito dalle seguenti operazioni:

- Inserimento dell'URL
- Scatto della fotografia
- Visualizzazione risultati e ritorno alla pagina principale

Inserimento dell'URL

Per inserire l'url occore premere il pulsante in basso a sinistra con la scritta "URL", in seguito inserire nell'apposita barra l'indirizzo generato da Ngrok e premere il pulsante "submit" in modo che l'applicazione lo memorizzi:



Figura 4.3: Fasi di inserimento dell'URL.

L'applicazione non verificherà immediatamente se il testo inserito corrisponda ad un url effettivo o se sia quello corretto. Se il testo immesso non corrispondesse ad un url non sarebbe possibile scattare una fotografia e apparirebbe un messaggio di errore. Lo stesso messaggio apparirebbe se non fosse stato minimamente eseguito questo passo di inserimento, poiché non vi è un server url di default, ma esso va inserito ogni volta che si apre l'applicazione.

Se il testo inserito corrispondesse ad un url, ma non fosse effettivamente quello creato da noi con il nostro script python (per esempio inserendo <https://www.google.com/>), una volta scattata la fotografia, si verificherebbe un'attesa infinita di una risposta, che probabilmente non arriverà mai.

Scatto della fotografia

Intuitivamente per scattare la foto da inviare alla rete è sufficiente premere il pulsante in basso a destra in figura 4.2. L'unico accorgimento che va fatto, come detto in precedenza, è la necessità di effettuare in precedenza l'inserimento corretto dell'url, in modo da non avere il messaggio di errore.

Se la procedura di inserimento è stata fatta correttamente allora si vedrà la sostituzione dell'output della fotocamera con una schermata di loading:

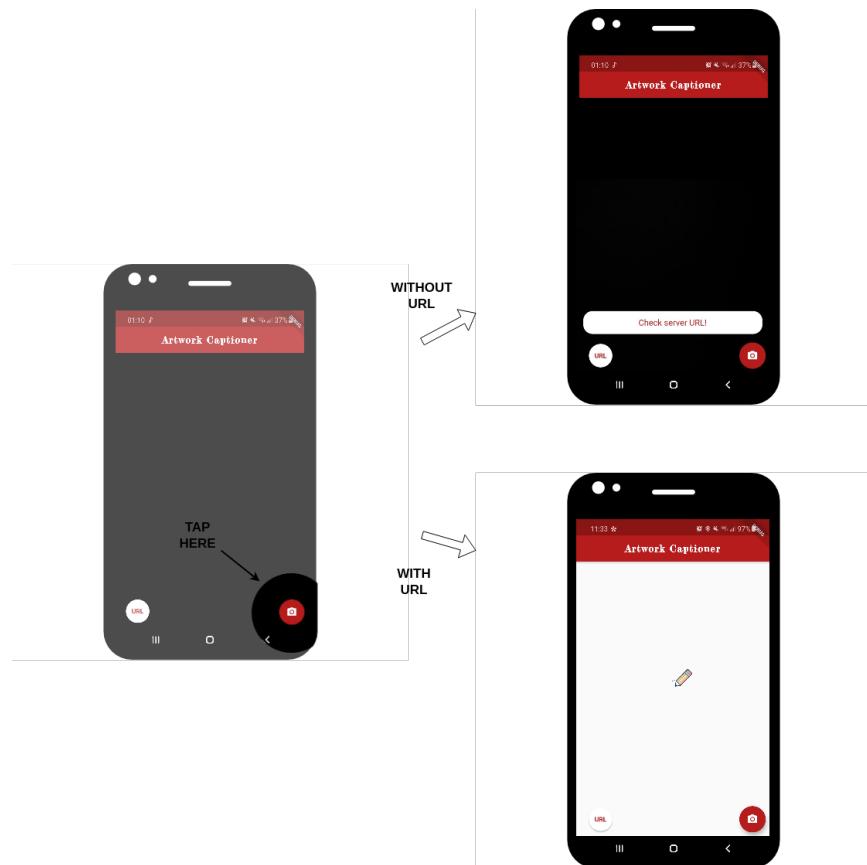


Figura 4.4: Schermate di scatto della foto con o senza inserimento dell'url.

Se appare la schermata di loading (immagine in basso in figura 4.4) dunque la foto è stata scattata ed inviata alla rete neurale e si sta effettuando il processo di inferenza. Una volta concluso apparirà la schermata di risultato rappresentante la foto e la caption generata dalla rete. Per tornare indietro e scattare una nuova foto basta premere il pulsante in alto a sinistra della schermata e apparirà nuovamente il rendering della fotocamera per un nuovo scatto.

4.2 Sistema Client-Server con Ngrok

Vista la costruzione dell'app e il suo utilizzo non ci resta che vedere come funziona, nel lato backend, la comunicazione col frontend. Abbiamo anticipato il fatto che comunichino tramite un url pubblico e una funzione che deve restituire un file json data in input un'immagine.

Come abbiamo visto nella dichiarazione del metodo *uploadImage*, una volta scattata la foto, viene fatta una richiesta HTTP POST al server indicato dall'url precedentemente registrato. A questa richiesta viene allegata un immagine, celata agli utenti poichè la richiesta è di tipo "post", che serve da input per la rete che si trova nel nostro codice python. E' dunque necessario generare un server che possa rispondere ad una richiesta del genere tramite un output intelligente che andrà elaborato dalla nostra applicazione. Per questo utilizzeremo un'applicazione *Flask*, che fungerà come nostra WSGI. Scriviamo dunque la funzione di nostro interesse e, tramite il decoratore *route*, decidiamo il tipo di url e di richiesta che faranno da trigger alla nostra funzione:

```
 1 from flask import *
 2 app = Flask(__name__)
 3
 4 @app.route('/', methods=['GET', 'POST'])
 5 def upload_file():
 6     if request.method == 'POST':
 7         ...
 8         if file and allowed_file(file.filename):
 9             filename = secure_filename(file.filename)
10             path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
11             file.save(path)
12             beam_search = BeamSearch(learn.model.encode, learn.model.
13             decode_step, beam_width)
14             image = valid_tfms(Image.open(path).convert('RGB'))
15             results = beam_search(image)
16             dictionary = {
17                 "caption": vocabulary.textify(results[0]),
18                 "beam_width": beam_width
19             }
20             return jsonify(dictionary)
21             return jsonify({"data": "False"})
```

Listing 4.8: Dichiarazione del metodo di risposta alla richiesta post

Come si può vedere se la richiesta è corretta, l'immagine passata per parametro viene salvata e utilizzata dal learner per generare una caption. Quest'ultima viene wrappata in un file json, di facile interpretazione per entrambi i linguaggi comunicanti, e restituita come output al client che ha fatto richiesta. L'unica cosa che resta da implementare è la generazione di un indirizzo web che consenta di fare richiesta. Ngrok è un software che permette di creare un indirizzo web che punta al proprio localhost. Avremo a disposizione un'url di questo tipo: `http://seriesofnumberandletters.ngrok.io`

```
1 public_url = ngrok.connect(port).public_url
2 print(" * ngrok tunnel \"{}\" -> \"http://127.0.0.1:{}\"".format(
3     public_url, port))
4 app.config["BASE_URL"] = public_url
```

Listing 4.9: Creazione del tunnel ngrok

Tramite *Ngrok* è possibile dunque generare un url temporaneo, utile per condividere l'applicazione web direttamente dal proprio computer, specialmente in fase di testing.

Capitolo 5

Conclusioni

I risultati ottenuti da questa implementazione della rete neurale sono, tutto sommato, soddisfacenti, anche se ancora molto approssimativi. Questa conclusione spinge a trovare metodi per il miglioramento della rete e ci aiuta inoltre a capire quali siano le principali correzioni da attuare per realizzare un prodotto più efficiente. La rete riesce nella maggior parte dei casi a distinguere di che tipo di opera si tratta, e questo è un bene, vista la precisa metodologia di descrizioni che è stata utilizzata nella creazione del dataset. Ciò dimostra che nell'ambito dell'image captioning una struttura comune delle caption di training può aiutare molto la rete per la costruzione della frase.

Questo accorgimento non basta però a garantire un pieno successo, poichè una delle caratteristiche delle opere d'arte è proprio la varietà del loro contenuto, spesso molto difficile da rappresentare testualmente utilizzando il numero minore di parole possibili. Per far in modo che la rete sia in grado di imparare un numero relativamente alto di pattern diversi, è necessario che essa venga sottoposta ad un numero elevato di esempi per ciascun di questi pattern. La data augmentation in questo caso può aiutare, ma non con un dataset piccolo come quello utilizzato in questo progetto. La soluzione migliore è dunque quella di ampliare il dataset, raggiungendo almeno qualche migliaio di esempi di training. Questo non risultava particolarmente agile da compiere manualmente, dunque ho provato ad agire su un dataset di opere culturali esistente, noto come *artpedia*, il quale è utilizzato per reti di question answering.

Ovviamente l'utilizzo di questo dataset presupponeva l'uso delle descrizioni culturali presenti all'interno dello stesso, le quali non essendo state scritte personalmente non

godevano del vantaggio descritto precedentemente. Poichè poteva comunque riusltare come una buona chance di implementazione, ho tentato di migliorare il contenuto delle caption contenute nel dataset attraverso delle modifiche:

- Rimozione della punteggiatura
- Trasformazione in carattere minuscolo delle parole "Virgin" e "Christ", in modo da non escluderle nel passaggio seguente
- Eliminazione di tutte le captions che contengano parole Maiuscole, le quali spesso indicano nomi propri di persone o luoghi
- Eliminazione di tutte le captions che contengano date

```
1 for d in artpedia_train_fns_caps:
2     print("-----")
3     s = 0
4     while s < len(d[1]):
5         exclude = set(string.punctuation)
6         d[1][s] = ''.join(ch for ch in d[1][s] if ch not in exclude)
7         d[1][s] = d[1][s].replace('Virgin', 'virgin')
8         d[1][s] = d[1][s].replace('Christ', 'christ')
9         found = False
10        for char in range (1, len(d[1][s])):
11            if d[1][s][char].isupper() or d[1][s][char].isdigit():
12                found = True
13            if (found and len(d[1]) > 1):
14                d[1].pop(s)
15            s-=1
16        s+=1
```

Listing 5.1: Pulizia delle descrizioni di artpedia

Questa soluzione ha portato ha una riduzione drastica dei termini utilizzata, anche se molte immagine sono state costrette a possedere termini inutili poichè disposte di una singola descrizione. In ogni caso il dataset presentava anche il problema della reperibilità delle immagini. Esso infatti non disponeva delle immagini concret, ma solo di un indirizzo url associato, il quale per circa il 90% delle immagini risultava deprecato. Dunque, per risolvere questo problema è stato necessario utilizzare un downloader automatico di google immagini noto come *google-images-download*:

```

1 from google_images_download import google_images_download
2 from termcolor import colored
3
4 for i in range (0, len(titles)):
5     print(colored(str(i), 'red'))
6     response = google_images_download.googleimagesdownload()
7     arguments = {"keywords":titles[i],"limit":1,"print_urls":True,
8                  "output_directory":'path',
9                  "format":"jpg", "size": "medium"}
10    paths = response.download(arguments)
11    for path in pathlib.Path('path'+titles[i]).iterdir():
12        if path.is_file():
13            old_name = path.stem
14            old_extension = path.suffix
15            directory = path.parent
16            new_name = str(i)+'.jpg'
17            path.rename(pathlib.Path(directory, new_name))
18            os.replace('path'+ titles[i]+'/'+str(i)+'.jpg', 'path'+str(i) +
19            ".jpg")
20            os.rmdir('path'+titles[i])

```

Listing 5.2: Ricerca immagini con google.images.download

Una volta elaborato correttamente il dataset, sono stati utilizzati i dati di training, ma sfortunatamente con scarsi esiti. Ciò è il risultato di una eterogeneità elevata nelle descrizioni delle immagini, che non aiuta la rete a comprenderne correttamente i patterns. Si può dunque dedurre dai test effettuati che sono necessari un numero di dati proporzionali alla quantità di patterns che la rete deve individuare, e inoltre le descrizioni devono cercare di rappresentare una struttura comune in modo da aiutare la rete nella costruzione della frase. In seguito verranno riportate delle immagini rappresentanti i risultati ottenuti dalla rete in fase di preallenamento e in seguito all’allenamento dedicato.

Pretrained examples:



Figura 5.1



Figura 5.2

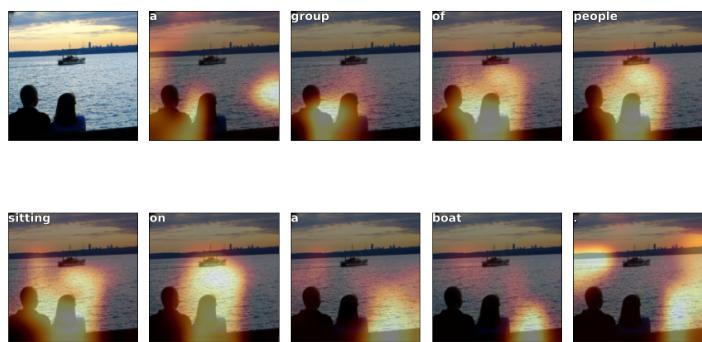


Figura 5.3

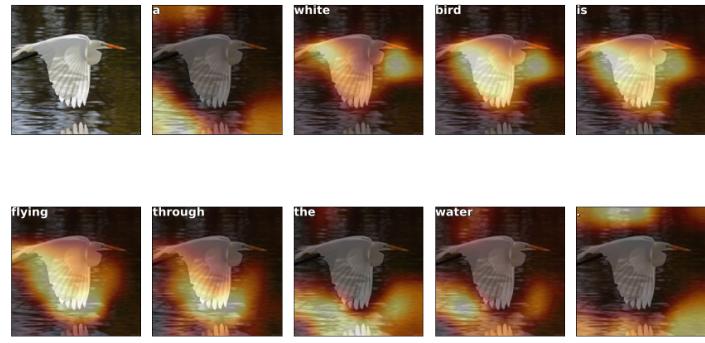


Figura 5.4



Figura 5.5

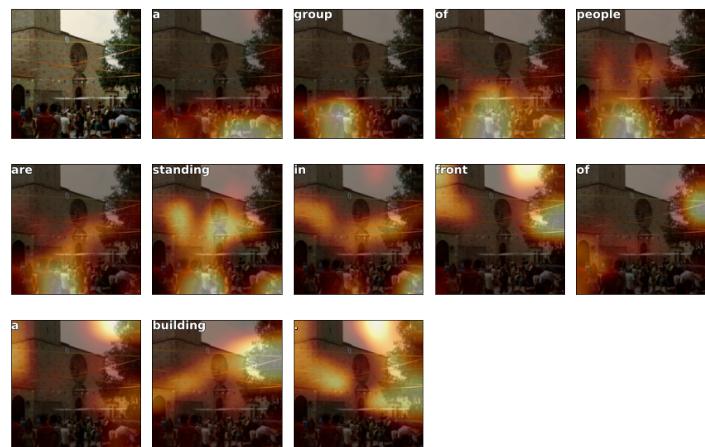


Figura 5.6

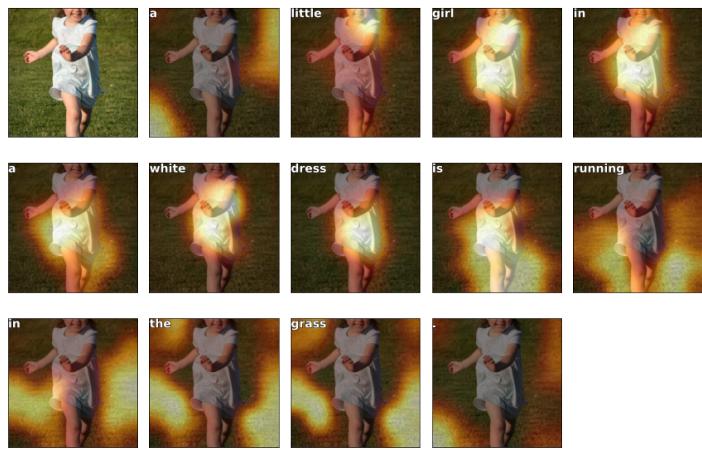


Figura 5.7



Figura 5.8

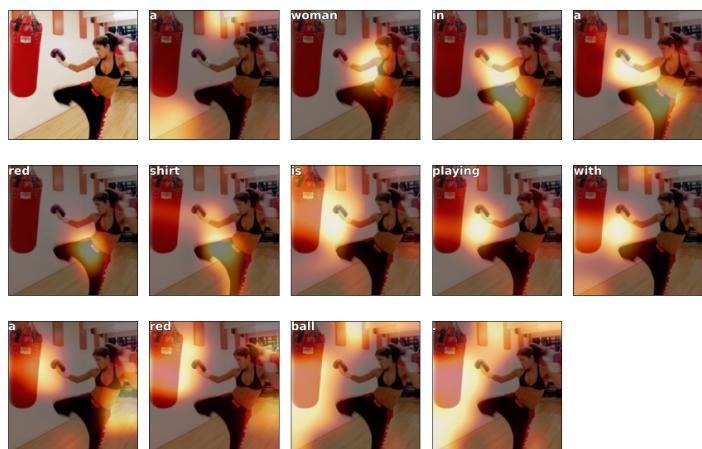


Figura 5.9

Cultural example after the second session of training:



Figura 5.10



Figura 5.11

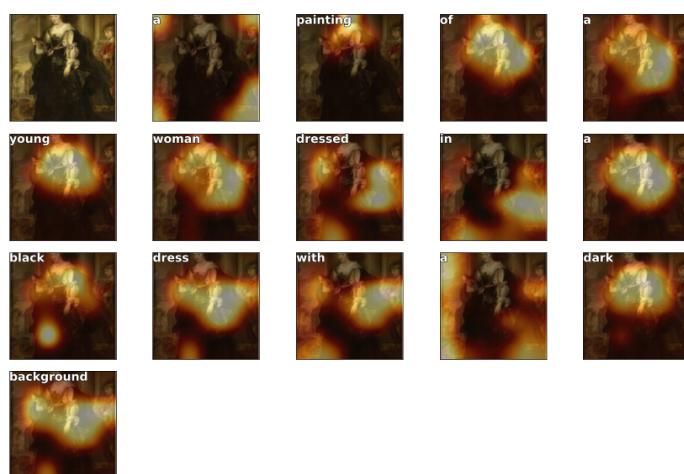


Figura 5.12



Figura 5.13

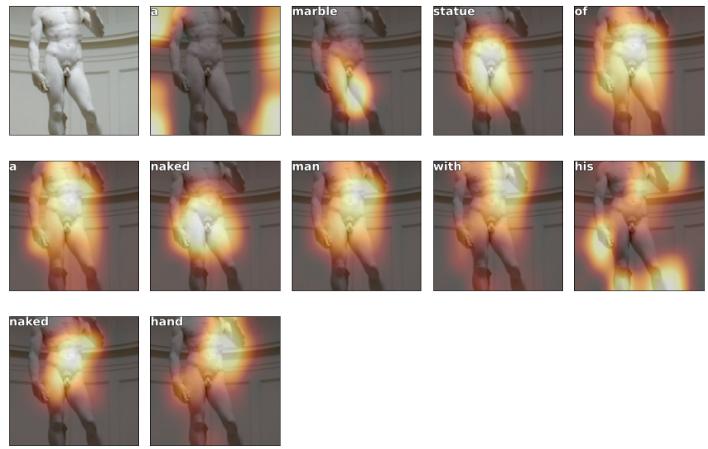


Figura 5.14

Mobile App screenshot examples:

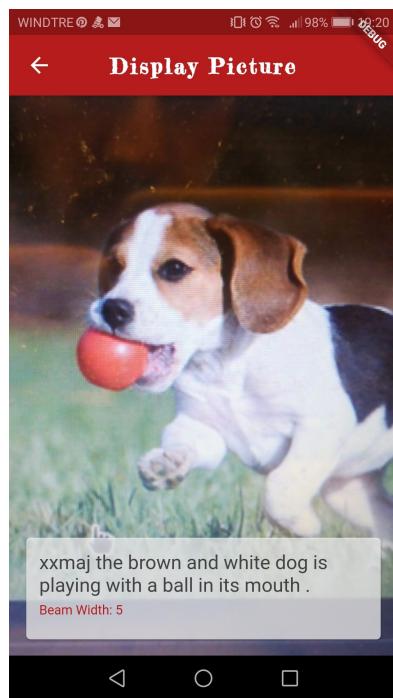


Figura 5.15

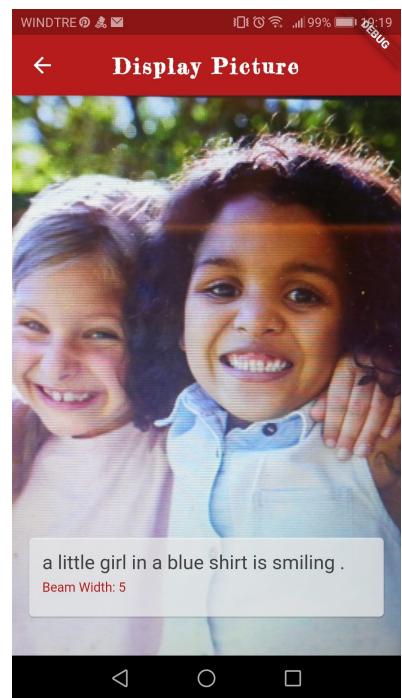


Figura 5.16



Figura 5.17

Bibliografia

- [1] G. Dorfles, A. Pinotti, M. Ragazzi, C. Dalla Costa. *Le Arti.* ©ATLAS, 2014
- [2] Francois Chollet. *Deep Learning con Python.* ©APOGEO 2020
- [3] Daniel Huttenlocher.
<https://techcrunch.com/2017/08/01/what-you-should-know-about-ai/>
- [4] Enciclopedia Treccani
https://www.treccani.it/enciclopedia/elenco-opere/Encyclopedia_della_Matematica
- [5] Kaiming, HeXiangyu, ZhangShaoqing, RenJian Sun. *Deep Residual Learning for Image Recognition.* Microsoft Research 2015
- [6] Eric Muccino.
<https://medium.com/mindboard/lstm-vs-gru-experimental-comparison-955820c21e8b>
- [7] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.* Université de Montréal 2014
- [8] Kelvin Xu, Jimmy Lei Ba, Ryan Kiros et al. *Show, Attend and Tell: Neural Image CaptionGeneration with Visual Attention* in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2048–2057, Boston, MA, USA, June 2015.
- [9] P. Kishore, S. Roukos, T. Ward, and W.-J. Zhu, *BLEU: a method for automatic evaluation of machine translation* in Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, Philadelphia, PA, USA, July 2002.

- [10] S. Banerjee and L. Alon *METEOR: an automatic metric for MT evaluation with improved correlation with human judgments* in Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation And/or Summarization, pp. 65–72, Ann Arbor, MI, USA, June 2005.
- [11] C.-Y. Lin *ROUGE: a package for automatic evaluation of summaries* in Proceedings of the Text Summarization Branches Out, Workshop on Text Summarization Branches Out, Barcelona, Spain, July 2004.
- [12] R. Vedantam, C. Lawrence Zitnick, and D. Parikh, *Cider: consensus-based image description evaluation* in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4566–4575, Boston, MA, USA, June 2015.
- [13] P. Anderson, B. Fernando, M. Johnson, and S. Gould *Spice: semantic propositional image caption evaluation* in Computer Vision—ECCV 2016, vol. 11, no. 4, pp. 382–398, Springer, Cham, Switzerland, 2016.
- [14] Fabio M. Graetz.
<https://github.com/f91/Neural-Image-Caption-Generation-Tutorial>
- [15] <https://www.json.org/json-it.html>

Ringraziamenti

