

Deep Learning

Elisa Stabilini

II Semestre A.A. 22/23

Contents

1 Lezione 1	7
1.1 Knowledge-based	8
1.2 Learning Paradigm	8
1.3 Modelli e metriche	9
1.4 Tradeoffs da bilanciare	9
1.5 Classificazione della performance	10
1.6 Generalizzazione	10
1.7 Accuracy	11
2 Lezione 2	13
2.1 Minimi quadrati - Gauss	13
2.2 Metodi di derivata	13
2.2.1 Gradient descent	14
2.3 Fature scaling	14
2.4 Metodi misti	14
2.4.1 Stochastic Gradient Descent	15
2.4.2 Mini Batch Gradient descent	15
2.5 Schemi di modifica del gradiente	15
2.6 Evolutionary Algorithms - EA	16
3 Lezione 3	17
3.1 Neurone artificiale	17
3.2 Nomenclatura	18
3.3 Training	18
3.4 Backpropagation	18
3.5 Esempi	20
4 Lezione 4	23
4.1 Metodologia pratica	23
4.2 Considerazioni preliminari sui dati	24
4.2.1 Sbilanciamento dei dati	24
4.2.2 Post-analisi dati	25
4.3 Hyperparameter tune	25
4.3.1 Grid search	25
4.3.2 Random search	26
4.3.3 SMBO	26
4.4 Cross-validation	28

5 Lezione 5	29
5.1 Computational graphs	29
5.2 Recurrent Neural Networks	30
5.2.1 Model 1	32
5.2.2 Model 2	32
5.2.3 Model 3	32
5.2.4 Costruzione di RNNs	33
5.2.5 Allenamento di RNNs	33
5.3 Bidirectional RNNs	34
5.4 Bidirectional RNNs	34
5.5 Deep Recurrent Neural Networks	34
5.5.1 Recursive Neural Network	36
5.5.2 Problemi delle DRNN	36
5.6 Long Short Term Memory	37
6 Lezione 6	39
6.1 Immagini digitali	39
6.2 Computer Vision Problems	39
6.2.1 Classificazione	39
6.2.2 Riconoscimento di oggetti	40
6.2.3 Neural Style Transfer	40
6.3 Reti neurali convoluzionali - CNNs	40
6.3.1 Digital Image Transformation	41
6.3.2 Struttura delle CNN	42
6.3.3 Feature map	42
6.3.4 Rappresentazione dei colori	43
6.3.5 Pooling Layer	44
7 Lezione 7	47
7.1 Transfer Learning	47
7.2 Object localization and detection	49
7.2.1 Loalizzatore	49
7.2.2 YOLO	53
7.3 Identificazione	53
7.3.1 Mask R-CNN	54
8 Lezione 8	57
8.1 Unsupervised Learning	57
8.2 Generative model	57
8.2.1 Autoencoders	58
8.2.2 Generative Adversarial Networks	64
9 Lezione 9	67
9.1 Reinforcement learning	67
9.2 Markov Decision Process	68
9.2.1 MDP	68
10 Lezione 10	71
10.1 Nozioni di base	71
10.2 Circuiti quantistici	73
10.2.1 Misurazioni	75
10.2.2 Variational Quantum Circuits	75

10.2.3 Strategie di implementazione	76
10.3 Images	77
10.4 Sparse connectivity	77
10.5 Upspeed image localization	78
10.6 Kullback–Leibler divergence	79

Chapter 1

Lezione 1

Fino agli anni '80 il constraint maggiore nella pipeline era Dal 2010 ad oggi si entra nell'era del deep learning (si entra nell'era in cui le GPU sono in grado di fare calcoli grafici con precisioni float ecc.. quindi nascono grandi architetture in grado di fare calcoli complessi. Quindi si ha una transizione da un livello AI a un livello deep per cui è molto complesso. L'idea è quella di usare un sistema computazionale per risolvere una task in maniera efficiente. Capire quale modello usare per svolgere un determinato compito e quali software utilizzare per l'implementazione.

Dagli anni 50 (anni in cui sono stati fatti i primi test) agli anni 80 sono stati fatti degli studi in cui si introducono alcuni - pochi - modelli utili che però creano le basi per quello che si farà in futuro. Il più grande constraint in questa fase è la potenza di calcolo per cui si iniziano a sviluppare delle tecniche che consentono di parlare di intelligenza artificiale ma l'applicazione di queste tecniche risulta molto dispendiosa e poco efficiente. Dagli anni 80 le CPU diventano più potenti, si inizia a parlare quindi di Machine Learning (il più famoso è Deep Blue dell'IBM), in questa fase la tecnologia ci consente di fare un allenamento consistente. Dal 2010 ad oggi si sviluppa il cosiddetto deep learning, grazie allo sviluppo di GPU molto potenti che consentono di fare calcoli di algebra lineare molto velocemente. Si sviluppano quindi molti modelli. Quindi si ha una transizione da un mondo completamente AI a qualcosa di più specifico, il deep learning, dove deep indica il grado di complessità di questi algoritmi.

L'idea è quella di trasformare un sistema computazionale in qualcosa che esegue una task in modo intelligente (prima definizione che è stata data di AI).

In generale esistono almeno due tipologie di task in generale

1. task astratta e formale, questo tipo di task è abbastanza difficile per noi ma che per una macchina non è così complicato perché è in grado di simulare moltissime condizioni o pescare una partita simile nel passato non è difficile. Questo tipo di task è basato sulla *conoscenza*, per noi è complesso per la macchina no.
2. Task che sono *intuitive*, immediate per noi ma che per la macchina sono complesse, quello che pè difficile per noi per costruire macchine che svolgono questo lavoro è il difficile grado di formalizzazione. Noi abbiamo capacità di interpolare, astrarre il concetto e poi generalizzarlo a cose che non ho mai visto. Questo tipo di ragionamento una macchina non sa farlo, noi siamo in grado di interpolare, astrarre il concetto e poi applicarlo a oggetti o realtà mai viste, cosa che per la macchina è molto difficile. Quindi per risolvere

il secondo punto non posso usare una programmazione banale di un codice, devo quindi sviluppare una buona tecnica.

Artificial intelligence In passato si è provato ad estendere il primo approccio alla seconda categoria di task ma non ha funzionato molto bene, richiedeva molta supervisione umana e la qualità delle predizioni era abbastanza bassa. Mancava il concetto di rappresentazione dell'allenamento, del fatto che la macchina ha bisogno di imparare. Allora definiamo AI come il sistema computazionale che in maniera autonoma impara una task, è un sistema che deve fare una acquisizione della propria conoscenza.

Machine learning Nel '98 si è definito un programma computazionale che impara da un'esperienza e poi noi misuriamo la performance del modello di AI. Il loop da costruire in questo caso è quello di prendere dei dati, valutare la performance con una metrica di performance e poi iterare il procedimento

Deep learning La differenza tra machine learning e deep learning è che, se prima avevo dei dati, io programmatore dovevo fare la feature extraction per poi costruire e allenare un modello, oggi vista la complessità delle task e la potenza di calcolo a disposizione, si inserisce la feature extraction nel modello, generalmente questa fase rientra nel cosiddetto pre-processing.

1.1 Knowledge-based

non ha dato risultati quindi definisco AI come quel sistema computazionale che impara la task, è un sistema che deve fare la acquisizione della propria conoscenza.

Oggi si possono distinguere almeno tre ambiti di applicazione del deep learning: data mining, intuitive task, human interaction.

1.2 Learning Paradigm

Esistono diverse tipi di tecniche di learning:

- Supervised learning: ho dei dati rappresentati in uno spazio e con questi devo fare qualcosa ad esempio classificarli o fare una regressione. Quello che faccio è inviare il dataset nell'algoritmo, che dai dati fa una predizione che viene messa a confronto con l'output desiderato da un supervisor. È il modello di machine learning più comune e più semplice, può essere comunque in generale migliorato da tecniche di deep learning.
- Unsupervised learning: non ho informazioni sui dati che posso usare nel ciclo di supervisione che avevo prima. Mi chiedo quali sono le feature che posso estrarre da un sistema (posso farlo per fare PCA non lineare per comprimere le dimensioni dello spazio, density estimation, clustering). Sono tutte tecniche in cui non ho un target io ho dei dati e devo trovare un modo per modellarli, per trovare delle feature. Non ho un tipo di output fissato, quello che chiedo al modello è di scoprire cose sui dati, l'output naturalmente dipende dal tipo di informazione di cui ho bisogno.
- Reinforcement learning: in questo caso l'esempio più semplice è quello dei videogiochi. Ho dei dati che definiscono l'environment e ho un agente (che

è il mio modello). L'agente può fare delle cose nell'ambiente e a seconda di quello che fa ottiene un reward, se fa qualcosa che fa bene all'environment allora otterrà un reward positivo. Questi algoritmi sono utilizzati per la costruzione di modelli real time, per cui l'agente deve prendere decisioni istantanee sull'ambiente che lo circonda.

Nella realtà poi si usano diversi algoritmi (almeno 60 ne esistono) e la scelta di un algoritmo dipende dal problema che devo affrontare.

Pipeline Quello che è in comune a tutte le tecniche però è la **pipeline**: ho dei dati, questi dati entrano in un training-loop, chiede allo sviluppatore di definire una **funzione di costo** e un **ottimizzatore**. A questo punto comincia il training, a questo punto però devo capire se il modello funziona, se è in grado di generalizzare, non voglio che il modello funzioni solo sui dati che già conosce, devo quindi preoccuparmi della **cross validation**. Una volta verificata anche la cross-validation ottengo il modello finale.

1.3 Modelli e metriche

Partiamo dal primo gruppo che abbiamo definito quindi partiamo dai modelli e dalle funzioni di costo per capire cosa sono e come vengono utilizzati. Il modello è una funzione matematica che prende un input (vettoriale o no) e mi dà una predizione, quindi se ho fatto delle misure il mio obiettivo è quello di definire un modello che approssimi alla funzione y . Un modello di questo tipo è un modello **predittivo** (che può essere lineare o non lineare, questo non è importante).

Ad esempio posso considerare una **regressione lineare**: ho un input x , il mio obiettivo è costruire un modello \hat{y} che sia in grado di predire il risultato y della misura. Quindi moltiplico l'input per una matrice di pesi w e aggiunge un bias. Il problema è che prendo semplicemente una regressione lineare può essere che il mio modello non generalizzi bene.

Esistono anche **modelli non lineari** in cui la flessibilità dipende dalla scelta dell'architettura della rete neurale. Quindi già la scelta del modello è in realtà un problema, quale forma, quale shape scegliere, perché qui non ho un modello di fisica dietro che mi può aiutare nella scelta. Qui ho solo dei dati di cui non so nulla. Questo apre un problema ovvero quello del **tradeoff**, devo decidere tra massima accuratezza e minima interpretabilità o minima accuratezza e massima interpretabilità.

1.4 Tradeoffs da bilanciare

1. Il primo tradeoff da bilanciare è quello tra accuratezza ed interpretabilità (magari non so più dire esattamente come funziona il modello perché la complessità è aumentata di molto).
2. Altro tradeoff è quello tra capacità e flessibilità, quindi devo muovermi tra una situazione di underfitting e una situazione di overfitting. Nell'ultimo caso infatti l'architettura è addirittura troppo flessibile. Quando scelgo un modello quindi ricado sempre nel problema di bilanciare underfitting ed overfitting. Questo secondo caso naturalmente è inutile perché sicuramente tocca bene i dati ma poi non è in grado di fare predizioni

1.5 Classificazione della performance

Si usano degli estimatori statistici che misurano la distanza tra la predizione del modello ed i dati, spesso si usano **cost/loss/error function**. Se anziché fare regressione però faccio ad esempio classificazione ho bisogno di altri tipi di funzione come ad esempio l'**accuracy**. L'idea è sempre quello di definire una funzione matematica che sia adatta alla valutazione della performance dell'algoritmo sul problema che voglio affrontare.

Un esempio di funzione costo che posso utilizzare è il **mean squared error** (MSE), per diversi valori dei parametri io voglio minimizzare questa loss. Naturalmente il modello può essere complicato però l'obiettivo resta sempre questo.

Quando i dati hanno degli errori e delle correlazioni, un'altra funzione che si utilizza è il χ^2 .

Se faccio classificazione il problema cambia, ad esempio se faccio una classificazione binaria quello che voglio fare è massimizzare una categoria piuttosto che un'altra quindi si utilizza la **cross-entropy**.

Un'altra ancora è la **log-likelihood**, questa viene utilizzata spesso nei casi di unsupervised learning: ad esempio voglio la densità che mi genera una determinata distribuzione.

1.6 Generalizzazione

Abbiamo visto i problemi di accuracy, underfitting e overfitting, abbiamo visto anche la cost function quindi adesso si può passare al problema della generalizzazione di un modello. Il mio obiettivo è quello di calcolare due loss function una per il training set e una per il test set, questo mi consente di capire se sto facendo il lavoro corretto o no.

Il grafico che viene mostrato presenta a 0 un modello molto rigido, più mi sposto verso destra più il modello è flessibile. In teoria io voorei minimizzare la funzione costo. Se considero un modello troppo sensibile il modello è come se "imparasse a memoria gli stessi dati", l'errore sui dati di test però dopo aver raggiunto un minimo comincia ad aumentare. Questo gap prende il nome di **generalization gap**. Dal grafico si vede che sono stata in grado di trovare l'optimum capacity ma in corrispondenza di questo punto i due errori non assumono lo stesso valore. Questo fenomeno prende il nome di *bias-variance tradeoff*.

Il conflitto tra training e test error è detto bias variance tradeoff quando raggiungo il minimo di questo tradeoff le capacità di predizione sono al top.

ϵ è l'errore, infatti ho sicuramente sul mio dato un errore di perturbazione però il suo valore di aspettazione è nullo, faccio questo tipo di assunzione perché normalmente quello dei dati è un rumore bianco

$$\begin{aligned} E[\epsilon] &= 0 \\ E[\epsilon^2] &\rightarrow \text{Var}(\epsilon)\sigma^2 \\ y_0 &= \hat{y}(x_0) + \epsilon \end{aligned}$$

Quindi posso calcolare, date queste tre premesse il tradeoff che scrivo di seguito

$$\begin{aligned}
 E[(y_0 - \hat{y}(x_0))^2] &= E[(y(x_0) + \epsilon - \hat{y}(x_0))^2] \\
 &= E[(y(x_0) - \hat{y}(x_0))^2] + E[\epsilon^2] - 2E[(y(x_0) - \hat{y}(x_0))\epsilon] \\
 &= E[(y(x_0) - \hat{y}(x_0))^2] + \sigma^2 - 0^1 \\
 &= E[(y(x_0) + E[\hat{y}(x_0)] - E[\hat{y}(x_0)])^2] - E[(\hat{y}(x_0) - \hat{y}(x_0))^2] + \sigma^2 \\
 &= E[(y(x_0) - E[\hat{y}(x_0)])^2] + E[(E[\hat{y}(x_0)] - \hat{y}(x_0))^2] + \\
 &\quad + 2E[(y(x_0) - E[\hat{y}(x_0)])(E[\hat{y}(x_0)] - \hat{y}(x_0))] + \sigma^{22} \\
 &= E[y^2(x_0)] - E[\hat{y}(x_0)]^2 - 2y(x_0)E[\hat{y}(x_0)] + \\
 &\quad + E[\hat{y}(x_0)]^2 + E[y^2(x_0)] - 2E[\hat{y}(x_0)]E[\hat{y}(x_0)] - 2E[y(x_0)]^2 + \sigma^2 \\
 &= (E[y(x_0)] - y(x_0))^2 + (E[y^2(x_0)] - (E[y(x_0)])^2) + \sigma^2 \\
 &= (Bias[\hat{y}(x_0)])^2 + Var[\hat{y}(x_0)] + \sigma^2
 \end{aligned}$$

Un bias elevato indica underfitting, la varianza invece è misurata tra il modello e se stesso, se questa quantità è troppo alta quello che sta succedendo è che il Idealmente io vorrei evitare sia di imparare la varianza che di avere una situazione di underfitting che di overfitting. In sostanza la varianza sta misurando l'oscillazione del modello attorno a se stesso. Un forte underfitting è dominato da un alto bias, se mi trovo in una situazione in cui la varianza è elevata sono in una situazione di overfitting per cui il modello impara a memoria i dati e non sono più in grado di generalizzare.

Dal punto di vista pratico non possiamo effettuare questi calcoli per valutare il modello perché non siamo in grado di determinare i valori di questi parametri per il modello.

Penso aggiungere alla loss function dei fattori correttivi, dei pesi, se i pesi sono troppo grandi la loss function avrà un valore troppo elevato quindi la soluzione non è ideale, se faccio questo sto indirettamente cercando un modello semplice che abbia pochi pesi e con valori vicini a zero. Quello che devo fare è moltiplicare il weight-decay per un parametro reale λ . Quello che sto facendo è spostare il problema da uno spazio a un altro, prima avevo il problema della flessibilità, ora invece ho il problema inverso ovvero quello di tunare il parametro λ in maniera adeguata.

Se λ è troppo grande la penalità applicata alla loss function è troppo elevata e il modello non impara dai dati (underfitting), se la λ invece è troppo piccola rischio di arrivare ad una situazione di overfitting. Il corretto tuning della λ non mi consente di evitare l'overfitting ma mi dà delle tecniche numeriche per aggirarla.

Tutti questi parametri da tunare che non sono fissati sono i cosiddetti **hyper-parameter**. Una cosa che è importante ricordare è che spesso non è sufficiente fare uno splitting training e test ma è utile avere una suddivisione del mio dataset in tre parti, la parte aggiunta è quella del **validation set**. L'idea è che se ho un iperparametro come ad esempio la λ studio il valore migliore della λ sul validation set, non posso studiarlo sul test set perché altrimenti rischio di propagare l'errore, il bias del mio modello anziché ridurlo. Quindi idealmente voglio lavorare con **tre** dataset diversi.

1.7 Accuracy

Non sempre la loss function è la formula fondamentale per capire un modello, ad esempio nel caso di una classificazione si cerca di plottare una sorta di matrice, la

matrice che voglio deve essere sostanzialmente una matrice diagonale (con valori molto più alti sulla diagonale e più bassi fuori diagonale). L'utilizzo di questa matrice mi consente di affrontare e risolvere il problema dei falsi positivi e dei falsi negativi nella classificazione.

Posso, anziché utilizzare la matrice, calcolare la accuracy, definita come segue:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \quad (1.1)$$

D'altra parte confrontando quello che vedo sulla matrice con l'accuracy si vede che questa non descrive benissimo la situazione, posso quindi calcolare altri stimatori come ad esempio la **precisione** o la **recall**

$$Prec = \frac{TP}{TP + FP} \quad (1.2)$$

$$Rec = \frac{TP}{TP + FN} \quad (1.3)$$

(1.4)

Lo stimatore più utilizzato è la F1 definita come segue:

$$F1 = 2 \cdot \frac{Prec \cdot Rec}{Prec + Rec} \quad (1.5)$$

Oppure uso le **receiver operation characteristic curve** ROC curve, in questo modo posso confrontare i classificatori e verificare se sono abbastanza stabili, idealmente, quello che vorrei è che tutte le curve dei classificatori si comportassero nello stesso modo, sicuramente li vorrei tutti sul semipiano superiore. Posso anche fare l'integrale della ROC e in questo caso ottengo la AUC ovvero la **area under the ROC curve** Matrice di confusione in cui vorrei che la matrice sia sostanzialmente una matrice diagonale. Il problema quando guardo questa matrice è che si capisce più o meno la situazione però posso calcolare ad esempio l'accuracy. L'accuracy però non mi descrive bene tutta l'informazione che ho nella matrice, per questo posso ad esempio calcolare anche la precisione e la recall (che mi ricorda che i negativi ad esempio non vanno bene).

È importante trovare delle metriche che descrivano bene questa situazione, una delle metriche più utilizzate è la F1, oppure se voglio essere ancora più precisa posso usare delle curve ROC. Se ho diversi classificatori posso controllare che i modelli (i classificatori) siano effettivamente stabile. L'AUC si prende sempre quella con la AUC più grande, è un modo generalmente robusto.

Chapter 2

Lezione 2

Argomento della lezione sono i modelli di apprendimento ovvero le tecniche per l'apprendimento dei parametri. Se si guarda l'intero workflow dell'algoritmo la sezione su cui ci concentriamo oggi è l'ottimizzazione, l'ottimizzatore infatti regola l'apprendimento dei parametri.

L'obiettivo è quello di minimizzare la funzione costo, devo trovare il minimo di questa funzione, devo trovare il set di parametri ideale che minimizza la funzione costo. Naturalmente quando sono in più dimensioni, la forma matematica della funzione costo può essere complessa quindi gli algoritmi devono essere molto robusti. Esistono famiglie di funzioni:

- Minimi quadrati (equazioni normali)
- Metodi basati sulla derivata (ovvero gradiente), sono quelli più utilizzati per il deep learning.
- Algoritmi genetici: sono algoritmi stocastici che non hanno bisogno di conoscere la derivata

Anche la funzione di ottimizzazione fa parte dei parametri che vanno ottimizzati.

2.1 Minimi quadrati - Gauss

Le equazioni normali, quelli legati alla gaussiana sono estremamente efficienti (eg carico la matrice sulla GPU) d'altra parte però se ho molti parametri risulta difficile risolvere l'equazione ai minimi quadrati, rischio di non avere abbastanza memoria per poter fare la minimizzazione. In questo caso il calcolo è un calcolo esatto, il problema però è che fare le inversioni delle matrici necessarie a risolvere le equazioni l'operazione risulta assolutamente poco efficiente (soprattutto se si considera l'elevato grado di complessità delle reti neurali attuali). Normalmente per modelli troppo grandi non si utilizza questo metodo ma si preferiscono i metodi che fanno utilizzo della derivata.

2.2 Metodi di derivata

Queste tecniche sono chiamate di *ottimizzazione al primo ordine* naturalmente il gradiente è interessante ma non è esaustivo, con il gradiente infatti ho informazioni circa gli estremi ma idealmente dovrei calcolare una derivata seconda per

distinguere tra massimi e minimi. Metodi al secondo ordine sono detti metodi **hessiani** tuttavia anche i metodi hessiani non sono la soluzione migliore perchè presentano un elevato grado di complessità nella risoluzione dell'equazione e ci si riporta ai problemi del metodo precedente.

2.2.1 Gradient descent

Si parte dalla loss function e seleziono una soluzione iniziale per i parametri, a quel punto valuto nuovamente la funzione costo in w , a questo punto ripeto l'operazione fino a che non raggiungo idealmente un minimo del gradiente. Normalmente la legge di evoluzione per w_i è la seguente:

$$w_i := w_i - \eta \frac{\partial}{\partial w_i} j(w) \quad (2.1)$$

dove η è un numero reale positivo, il segno - mi serve per muovermi dalla parte opposta rispetto alla direzione in cui cresce la funzione. Il parametro η è chiamata learning rate, anche in questo caso si tratta di un iperparametro che devo tunare, infatti se è troppo grossa potrei rimbalzare da una parte all'altra del minimo senza avvicinarmi (quindi quello che succede è che la mia funzione costo oscilla molto), se invece scelgo un learning rate troppo piccolo quello che succede è che l'allenamento è troppo lento, quindi impiego troppi step per raggiungere il minimo della mia funzione costo.

L'altra cosa che può capitare è che se uso un valore completamente fuori scala allora la funzione costo non converge ma diverge, in sostanza mi allontano dal minimo.

2.3 Feture scaling

Se la distribuzione risulta in qualche modo compressa in una direzione, quello che succede è che con la definizione data per l'evoluzione dei pesi l'algoritmo fa fatica a tenere sotto controllo gli spostamenti nella direzione "compressa". Quello che si può fare è fare una standardizzazione sui dati, prima di partire con l'allenamento, anzichè iniziare l'ottimizzazione su questi dati cerco di lavorare su una "copia" dei dati che contenga dati tutto sommato uniformi e con una distribuzione sostanzialmente omogenea (non voglio avere numeri troppo grossi e numeri troppo piccoli).

Esempio di feature scaling è la normalizzazione ovvero trasformo secondo la legge

$$x_i = \frac{x_i - \mu_{x_i}}{\sigma_{x_i}} \quad (2.2)$$

2.4 Metodi misti

Il problema è che se scrivo la derivata parziale della loss function ho la derivata di un oggetto come una MSE, se il dataset è grande è un oggetto difficile da maneggiare: ho bisogno di fare una somma mostruosa, devo sommare su tutti gli n parametri, quindi dovrei caricare in memoria tutti i dati, quello che si può fare. Proprio per questo motivo anche i metodi di gradient descent *puri* non vengono utilizzati spesso con dataset grandi. Per evitare di dover sommare su tutti i parametri combino all'utilizzo di tecniche al primo ordine l'utilizzo di tecniche stocastiche

Batch Gradient Descent : si tratta di un full bath, quando il mio dataset è sufficientemente piccolo quindi sono in grado di immagazzinare in memoria tutti i dati allora utilizzo il metodo di gradient descent puro. Questo sarebbe l'ideale ma molto spesso non è possibile.

2.4.1 Stochastic Gradient Descent

Per prima cosa faccio uno shuffle dei dati dei training e pescò a caso un unico punto e cerco di minimizzare il gradiente in quella data direzione. Naturalmente non è ideale perchè rischio di andare in direzioni dello spazio dei parametri che non mi interessano e di conseguenza perdo tempo. Tuttavia, il SGD proprio per questo motivo mi consente di esplorare meglio lo spazio dei parametri, si conseguenza mentre un gradiente puro può rimanere "incastrato" in un minimo locale con un SGD posso uscire

2.4.2 Mini Batch Gradient descent

Una via di mezzo tra il full gradient descent (puro) e il gradiente stocastico è il mini batch gradient descent, in sostanza definisco una taglia per il mio batch, il mio sottoinsieme dei dati che voglio utilizzare e calcolo su questi dati la loss function. Scelgo la dimensione del batch, naturalmente di dimensione inferiore al dataset In questo modo sono in grado di unire gli aspetti positivi di entrambi i metodi ed esplorare lo spazio dei parametri senza però perdere troppo dal punto di vista dell'efficienza dell'ottimizzazione.

Sui software che usiamo si usano sempre 32 (o multipli di 32) oggetti per formare un minibatch (uso multipli di 32 perché così copro tutta la struttura di calcolo della GPU e ottengo una struttura di calcolo molto efficiente).

2.5 Schemi di modifica del gradiente

Se la morfologia della funzione è troppo complessa magari anche con i minibatch è troppo difficile trovare il minimo quello che posso fare è modificare lo schema di evoluzione dei pesi.

Momento Se la funzione è troppo complessa, trovare il minimo della funzione costo significa trovare un minimo che sia *sufficientemente buono* per avere una funzione costo accettabili. Una delle tecniche che si usa per migliorare in qualche modo il metodo del gradiente prevede l'aggiunta di un *momento* alla regola di evoluzione per i pesi aggiungo un termine che sia proporzionale al valore del peso stesso (motivo per cui prende il nome di momento). In questo modo il gradiente si muoverà verso il minimo, però lo forzo ad andare un pochino oltre, questo mi consente (se ad esempio mi trovo in un punto di sella) di spostarmi e raggiungere eventualmente un punto di minimo.

RMSPropagation Non ha molto senso fare un algoritmo che scende in modo identico per ogni parametro. Quello che fa questo schema è normalizzare il learning rate dinamico per cui l'algoritmo anzichè scendere in modo identico per ogni parametro mi consente di spostarmi in modo diverso per parametri diversi.

Tutti gli altri metodi come ad esempio AdaGrad Adam AdaMax in sostanza modificano leggermente l'ottimizzatore per migliorare la prestazione della rete neurale rispetto a quello che devo fare.

Metodo di Newton : è un metodo hessiano, quindi del secondo ordine. Consiste in sostanza nello sviluppare con Taylor la funzione costo fino al secondo ordine. Anche in questo caso il problema è il calcolo dell'hessiana (che diventa non banale soprattutto quando ho un modello non lineare)

Metodo quasi-newton : Naturalmente il metodo hessiano visto prima non è il metodo più veloce, si è cercato quindi in qualche modo di migliorare i metodi hessiani, alcuni metodi quasi hessiani sono BFGS, L-BFGS, DFP. In sostanza sono metodi che cercano di usare l'hessiana senza però tutti i problemi relativi al salvataggio dei dati e al doversi portare dietro strutture complesse.

2.6 Evolutionary Algorithms - EA

Sono i metodi di ottimizzazione basati su una similitudine con l'evoluzione genetica. In pratica si costruisce una procedura per cui i parametri di un modello vengono inizializzati e dopo di che si fa una selezione. Gli individui iniziali si riproducono, si applicano delle mutazioni, avvengono dei cross-over, dopo di che si selezionano gli individui più preformanti. Viene utilizzato nei casi in cui il gradiente non è analitico o è complesso da calcolare o anche provando con schemi di gradiente diversi il risultato non è un risultato ottimale.

Chapter 3

Lezione 3

La lezione verte sui modelli non lineari. Abbiamo sempre parlato del training loop, oggi parleremo dei modelli moderni (sviluppati negli ultimi 30 anni). Quando si parla di modelli di apprendimento non lineari si parla di Artificial Neural Networks (ANNs). Se il problema è caratterizzato da un numero elevato di features è pressoché impossibile utilizzare modelli lineari. Per questo motivo si utilizzano i modelli non lineari che in generale rappresentano meglio lo spazio delle possibilità e se la rete è sufficientemente flessibile con un modello non lineare posso anche rappresentare una funzione lineare.

Storicamente il primo modello di neurone artificiale nasce nel 1943, nel 1974 nasce l'algoritmo di back-propagation che consente di allenare le reti neurali. Da qui in avanti iniziano ad essere sviluppato moltissimi modelli di reti neurali. Dopo gli anni 2000, grazie alle GPU ci si sposta da un modello tutto sommato semplice a modelli che sono più complessi (si inizia a utilizzare il termine di deep learning). Quindi per prima cosa si parte dallo sviluppo dell'idea teorica, dopo di che si ha il salto dovuto allo sviluppo di un algoritmo, l'ultimo salto è quello caratterizzato dall'aumento della capacità computazionale.

3.1 Neurone artificiale

Immagino di avere degli input, a ciascuno dei quali è associato un peso w , eventualmente viene fornito come argomento anche un bias e quello di cui ho bisogno ora è una funzione che unisca in qualche modo i componenti (generalmente si fa una sorta di sommatoria) e a questo punto prendo una funzione non lineare. La somma di tutti gli input per i pesi viene quindi passata per argomento alla mia funzione non lineare detta **funzione di attivazione**.

Un esempio di funzione non lineare è la sigmoide definita come:

$$g(x) = \frac{1}{1 + e^{-x}} \quad (3.1)$$

oppure anche una tangente iperbolica.

Quando vedo un modello di neurone ogni input è un nodo, a ciascun nodo è associato un peso, questi input pesati vengono combinati e dopo di che l'input (divenuto unidimensionale) viene modulato dalla funzione di attivazione. Quello che devo capire è qual'è la modulazione - non lineare - che mi consenta una propagazione corretta dell'input tale da avere un output desiderato.

Il motivo principale per cui lavoro con la non-linearità è che mi serve un modello non lineare, voglio lavorare in uno spazio con più funzioni rispetto a uno spazio

di modelli lineari, dopo di che se utilizzo una rete neurale abbastanza profonda e abbastanza flessibile è anche possibile approssimare una funzione lineare con una serie di funzioni di attivazione non-lineari.

Una rete neurale è un oggetto in cui collego diversi neuroni. Ad esempio posso considerare una rete neurale di 3 layer, il primo layer è l'**input layer**, dopo di che c'è un **hidden layer**, così chiamato perché lavora "nascosto" non posso accedere ai suoi output. L'ultimo layer prende quindi il nome di output layer.

Quando definisco la rete devo definire non solo i nodi ma anche la matrice o il tensore dei pesi, deve essere definito tra ciascuna coppia di layer. In questo modello i miei parametri da ottimizzare sono le matrici dei pesi, li devo ottimizzare per poter avere l'output migliore possibile.

3.2 Nomenclatura

Feed Forward Neural Network: FNN, rete in cui tutti i collegamenti avvengono da sinistra verso destra, non ci sono loop, di conseguenza l'informazione può scorrere unicamente in una direzione.

Multilayer Perceptron: MLP, ho implementato una rete neurale utilizzando come unità fondamentali dei percetroni

Deep Neural Networks: DNN, indica tutte le reti neurali con più di un hidden layer. In generale questo termine fa riferimento ad un modello con un grande numero di parametri

3.3 Training

Un punto importante è l'allenamento della rete neurale, se considero il modello di ottimizzazione più semplice, in teoria devo utilizzare un metodo di gradient descent. Il problema è che se ho un modello con molti parametri il valore di aggiornamento dei pesi deve essere calcolato un numero di volte pari al numero di parametri. Questo passaggio diventa incredibilmente inefficiente. Infatti l'equazione per il gradient descent sarà del tipo

$$w_{ij}^{(l)} := w_{ij}^{(l)} - \eta \frac{\partial}{\partial w_{ij}^{(l)}} J(w) \quad (3.2)$$

dove l'indice l indica il layer a cui appartiene il nodo e ij sono gli indici del nodo di partenza e di arrivo. Naturalmente il problema è che se ho un modello con molti parametri devo calcolare il gradiente moltissime volte per ogni interazione del gradient descent. Se non c'è un algoritmo che mi permetta di calcolare l'algoritmo in modo efficiente (soprattutto perchè negli anni 70 non avevo tutta la potenza di calcolo necessaria per fare questo calcolo).

3.4 Backpropagation

L'algoritmo che facilita il calcolo di questo gradiente è l'algoritmo di back-propagation che può essere implementato in qualunque algoritmo di ottimizzazione che faccia uso del gradiente, è in grado di ridurre il numero elevato di computazioni che andrebbero effettuate grazie alla chain rule .

1. il primo step consiste nel calcolo di *tutte le activation function per tutti i nodi di tutti i layers* $a_i^{(l)}$, quindi parto dall'input a sinistra e arrivo fino all'output. Questo passaggio è chiamato di **forward propagation**
2. idea è quella di calcolare quanto la mia rete sbaglia rispetto al target, ai dati. Posso quindi associare a ciascun nodo un *errore di predizione*. Indico con $\delta_j^{(l)}$ l'errore di predizione associato al nodo j del layer l . Questa fase prende il nome di **backward propagation**.
3. Devo tornare indietro layer per layer cercando di accumulare gli errori e cercando di capire dove il modello sbaglia e capire quale layer accumula più errore. A questo punto mi chiedo se in qualche modo posso cercare un collegamento tra il gradiente della loss function e queste informazioni che ho calcolato
4. Se riesco a fare questa cosa, ottenuto il gradiente posso, usare il metodo di gradient descent aggiornando di volta in volta il peso con il gradiente che ho appena calcolato

Esempio Considero un MLP. Per prima cosa devo calcolare tutte le funzioni di attivazione dei nodi presenti nel modello. A questo punto devo calcolare l'errore commesso da ogni nodo. Dopo di che è stato dimostrato che il gradiente della loss function può essere scritto come segue

$$\frac{\partial}{\partial w_{ij}^{(l)}} J(w) = \frac{\partial J}{\partial z_i^{(l+1)}} a_j^{(l)} := \delta_i^{(l+1)} a_j^{(l)} \quad (3.3)$$

per $l = 1, \dots, L - 1$. Si noti che nell'ultimo passaggio sto usando il fatto che la derivata della loss function rispetto a z è esattamente l'errore che sto commettendo, per quello ho sostituito con la notazione per l'errore.

A questo punto si può costruire una relazione ricorsiva per calcolare l'errore di un layer più a sinistra tramite l'errore del layer più a destra. Questo è molto interessante perché posso ricavare gli errori per tutti i nodi di tutti i layer da destra a sinistra.¹. A questo punto posso fare una unica chiamata alla rete e poi in un'unica chiamata calcolare il gradiente di cui ho bisogno.² In questo modo la funzione di aggiornamento dei pesi risulta molto più semplice da utilizzare. In questo modo il gradient descent può essere applicato.

Quando si fa backpropagation ci sono comunque alcuni accorgimenti da tenere a mente:

- Non posso partire da una rete neurale in cui tutti i pesi sono nulli, scegliere un'inizializzazione troppo vicina a zero non è ideale perché altrimenti è particolarmente difficile per il mio modello modificare i parametri (ogni piccola variazione è immediatamente annullata dall'avere pesi nulli).
- Random: è una tecnica che prevede l'assegnazione di un peso a ciascun nodo in maniera casuale, questa tecnica però rischia di rompere la simmetria del modello quindi risulta non ideale.
- glorot/xavier: in entrambi si inizializza ciascun peso con un valore estratto da una distribuzione normale (quindi cerco in qualche modo di ridurre la casualità del problema)

¹Per la formula esatta vedere le slide

²Per un esempio dell'applicazione dell'algoritmo guardare le slide

- he: l'obiettivo in questo caso è evitare che la funzione di attivazione possa sturare, in sostanza è poco efficiente inizializzare gli algoritmi troppo lontano dal dominio di non linearità della funzione di attivazione. Questo è particolarmente importante quando si usano **dati standardizzati** perchè in questo modo sono in un certo senso "garantiti" del rimanere all'interno della regione di non linearità.

Quindi in questo senso l'inizializzazione dei pesi viene fatta in modo random ma secondo una distribuzione normale a media nulla e una piccola varianza, possibilmente anche cercando di fare in modo che l'input della funzione di attivazione di un nodo rimanga nella regione di non linearità della funzione stessa.³

3.5 Esempi

MLP - Multi layer perceptron: Ad esempio per costruire un classificatore posso utilizzare un percettrone multilayer con nell'ultimo layer una funzione di attivazione soft-max in maniera tale che l'output dei nodi dell'ultimo layer sommi a uno.

Recurrent Network: sono reti in cui sono presenti cicli tra i diversi nodi. Servono a creare una unità di memoria per cui in ciascun nodo ci sono anche eventuali componenti in più che devo sommare che sono definite nel layer stesso (molto utile per *serie temporali*)

Recursive Neural Network: Usate prevalentemente quando si lavora con *natural language processing* consentono di "spacchettare dei nodi" e poi riunirli in un secondo momento

Long short-term memory: Sono un'altra variazione di reti neurali ricorrenti, in questo caso sono composte da un "nucleo" della rete che è in grado di trattare i dati di input di output ed eventualmente ricordare informazioni

CNN - Convolutional Neural Network: Di per se non fanno nulla di predittivo ma sono molto utili per fare un filtraggio di tutti i dati di input (utilizzata spesso per il riconoscimento di immagini). La cosa particolarmente utile è la creazione di correlazione tra pixel distanti (in questo modo ad esempio posso mantenere invarianza per rotazione traslazione). Questa rete lavora semplicemente con la matrice di correlazione quindi è in grado di riconoscere un'immagine anche se ruotata/parziale ecc. Alla fine però, per effettuare il riconoscimento, al termine della rete ho una architettura tipo MLP

GAN - Generative adversarial network in questo modello l'idea è che ho dei dati - veri - e voglio creare dei dati *falsi* che però siano più verosimili possibile. Il compito della rete è quello di costruire una rete che registri noise ecc. dai miei dati. Per valutarla si mettono a confronto i dati in una rete neurale *discriminator* che cerca di capire se una immagine è un dataset in generale è vero o falso. Quindi in sostanza ho due reti neurali che lavorano in competizione (adversarial) l'una contro

³In pratica voglio che il prodotto input peso (ovvero l'argomento della funzione di attivazione) non esca dalla regione di non linearità di quest'ultima

l'altra. Ad esempio posso usare una GAN per imparare la distribuzione di probabilità sottostante a dei dati e poi migliorare il mio modello grazie alla disponibilità eventualmente di dati "fittizi".

Pattern recognition algorithms: Anche questi vengono utilizzati per lo più per il trattamento delle immagini digitali. In questo contesto esistono vari modelli: **semantic segmentation**, **classification + localization** (fallisce nel momento in cui ho più oggetti nell'immagine). **Object detection**, **instance segmentation** (mi fornisce anche l'informazione su quali sono esattamente i pixel che corrispondono ad un determinato oggetto). Queste tecniche vengono utilizzate spesso con diversi tipi di CNNs.

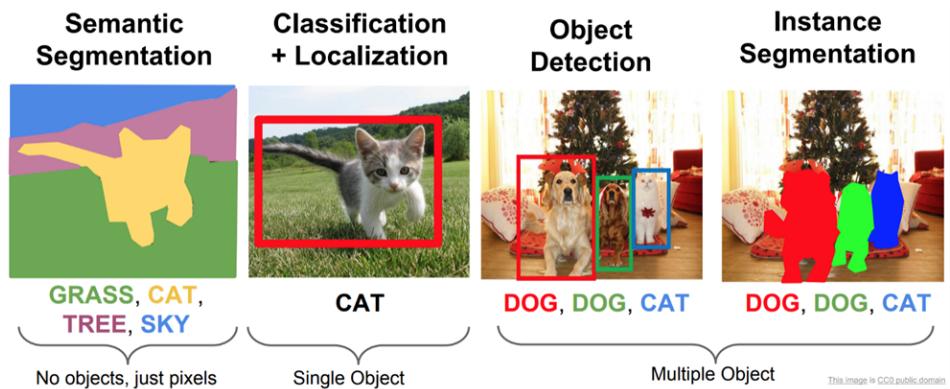


Figure 3.1: Esempi di utilizzo dei diversi algoritmi di pattern recognition

Chapter 4

Lezione 4

Model selection and hyper-parameter tuning

Vedremo le tecniche di training validation e vedremo alla fine come si fa a scegliere il best model.

Se si ripercorrono le lezioni precedenti si possono catalogare tutti i parametri e le variabili definite: capacità, modello, (numero di layer, nodi per layer, funzioni di attivazione se lavoro con reti neurali non lineare) parametri per la loss function e early stopping, metriche di performance (se voglio gradient descent devo decidere anche il learning rate e lo schema di variazione). Se faccio lo split training e validation si osserva che anche la scelta della frazione di dati è un problema. Inoltre ho tutte le proprietà che riguardano i dati e come lavorare nei casi in cui ho dati sbilanciati.

Ho tutta una serie di parametri che diventano *problema dell'ottimizzazione*, si parla di **hyperparameter** proprio perchè sono ulteriori rispetto al problema della modellizzazione dei dati. Quindi in sostanza sto facendo due ottimizzazioni: una più locale per risolvere il problema che mi pongono i dati e una seconda di minimizzazione degli altri parametri che mi restituiscono un modello *pttimo*.

Dato che ho molti parametri, tutti da testare e con uno spazio delle possibilità complesso la task è molto difficile e soprattutto *time consuming*.

4.1 Metodologia pratica

Non si tratta solo di conoscere i modelli e applicarli o programmarli ma imparare come muoversi. In teoria quando partiamo dal design del processo di allenamento questo deve tenere sotto controllo almeno quattro aspetti:

1. Se sto facendo un fit devo per prima cosa ricercare quali sono i modelli migliori, più utilizza.ti. Devo cercare di capire dove la comunità è arrivata e valutare, testare le performance di questo modello, tempi di training, quali sono i parametri e i valori migliori per questi parametri.
2. Definizione degli obiettivi: dove voglio arrivare? Ho un dataset di un certo tipo e ho bisogno di migliorare la performance in una certa direzione tenendo conto delle limitazioni anche tecniche che ho. è importante fissare degli obiettivi che siano realistici.
3. Proporre delle metriche per misurare la capacità del mio modello.

4. Proporre degli aggiornamenti incrementali in modo iterativo e soprattutto sistematico.

Ci sono scelte che devo fare per evitare di rimanere bloccati su una soluzione che è ottimale solamente per il problema specifico che sto affrontando e non per il caso generale.

Early stopping: Questo genere di algoritmi blocca il processo di learning una volta che all'aumentare delle iterazioni la cost function rimane pressochè costante. D'altra parte esistono algoritmi che mi consentono anche di proseguire per qualche iterazione in più per poter verificare di essere in un minimo "sufficientemente buono"

Neural Network Dropout: Metodo leggermente più sofisticato, consiste nel definire una quantità di dropout, per cui parto da una rete neurale con tutti i nodi e le connessioni attive, nell'esempio sulle slide si tratta di una FNN, in cui però durante il training, delle connessioni vengono *disattivate* (in maniera stocastica o secondo una regola specifica). Al termine dell'allenamento avrò una rete neurale con una architettura diversa da quella di partenza, per cui alcuni nodi non saranno più collegati, e soprattutto con un numero inferiore di parametri rispetto alla rete neurale di partenza. Quindi parto da un modello molto grosso e molto flessibile epr arrivare ad un modello più piccolo ma più efficiente che sicuramente non presenterà alcuni dei problemi della rete precedente come ad esempio Il problema, anche qui, quando definisco un layer di dropout devo definire una percentuale, una frequenza di dropout.

4.2 Considerazioni preliminari sui dati

Primo consiglio, quando si riceve un dataset è quello di cercare di capire i dati. Mi chiedo quindi come prima cosa se esistono delle correlazioni, per farlo posso ad esempio cercare di plottare la matrice di correlazione (questo posso farlo quando ho molti parametri). Primo ragionamento che possa fare è **ho trovato molte variabili correlate quindi posso eliminare alcune di queste variabili**. Questo è utile perché con meno dati, con meno ridondanza e un modello più piccolo posso arrivare anche ad un modello migliore. Per prima cosa quindi devo capire bene con che cosa ho a che fare.

4.2.1 Sbilanciamento dei dati

Se ho dei dati sbilanciati il rischio è che il modello non abbia una capacità di predizione omogenea perchè in alcune regioni dello spazio delle variabili non ho abbastanza statistica. Quello che posso fare per risolvere questo problema è, ad esempio, **ripesare le classi**. Questo processo consiste nel fornire più volte i dati appartenenti alle zone dello spazio per cui ho meno informazione come input del mio modello. Oppure posso fare **oversampling**, ovvero prendere i dati e cercare di modificarli per "ricrearne di nuovi". L'altra opportunità ancora, se ho abbastanza dati è quella di fare **data augmentation**.

In questa fase i problemi sono tanti per cui è particolarmente importante valutare bene la performance del modello e soprattutto pesare adeguatamente i dati provenienti dalle regioni in cui ho poca statistica.

4.2.2 Post-analisi dati

Gli altri step sono tutti step in cui la casistica si divide in successo e insuccesso.

Una volta capito un po' i dati si può procedere **verificando** la performance del modello sui dati di training. Se il primo fit non funziona allora posso aumentare la capacità del modello (tanto prima o poi dovrei arrivare in overlearning, altrimenti semplicemente la loss function migliora).

Se invece la performance sul dataset di training è accettabile allora posso allora provo a testare il modello sui dati di *validazione*. Se il fine tuning del modello fatto precedentemente (in sostanza il tuning dei parametri fatto durante il trainig) fallisce, nonstante io abbia provato schemi diversi e via dicendo allora significa che **ho qualche problema con i dati**, ad esempio può esserci del rumore oppure ci sono *inconsistenze nel dataset*.

Se possibile quello che si può fare è anche provare a chiedere più dati, se non è possibile quello che posso fare è ridurre ancora la dimensione del modello e rifare il finetuning

Se non sono riuscita comunque ad ottenere nulla di buono allora devo per prima cosa cercare di capire di quanti altri dati ho bisogno. Tuttavia in moltissimi casi il tutto si ferma qui perchè non posso acquisire nuovi dati (il costo di acquisire nuovi dati è troppo alto).

A questo punto l'unica altra cosa che posso fare è cambiare algoritmo, ricominciare da capo con un nuovo modello. Per capire in generale di quanti dati ho bisogno a seconda del modello che voglio usare posso fare un plot del tipo 4.1.

Io non so a priori se i dati che mi hanno dato contengono tutte le possibili informazioni per generalizzare il modello che sto utilizzando. Se sto trattando un problema nuovo devo generare grafici di questo tipo per capire se sto facendo un lavoro consistente oppure no, perchè altrimenti non saprei dire in che regione dei dati mi trovo.

4.3 Hyperparameter tune

L'obiettivo da un punto di vista meccanico è quello di ottenere la migliore performance possibile per il test-set e possibilmente anche per il training set (mi aspetto che questi valori siano vicini se il mio modello funziona bene). Alcuni esempi di miglioramento del modello consistono nell'aumentare il numero di layers o nodi per layers, il learning rate, il weight decay e il dropout rate. Da un punto di vista pratico quello che sto facendo nella fase di tuning degli iperparametri è affrontare un **problema di ottimizzazione**.

4.3.1 Grid search

Prendo i parametri, faccio una griglia, per ogni parametro inserisco un massimo e un minimo per ciascun parametro e dopo di che eseguo il fitting per ogni coppia di valori, quindi in sostanza per ogni iperspazio. Sicuramente dal punto di vista computazionale non è la soluzione più efficiente ma mi può aiutare a capire se un modello è stabile no. Nella pratica scelgo di allenare il mio modello n volte e per ogni fit stampo il valore dell'errore per poi indagare e cercare di capire i valori ottimali dei parametri

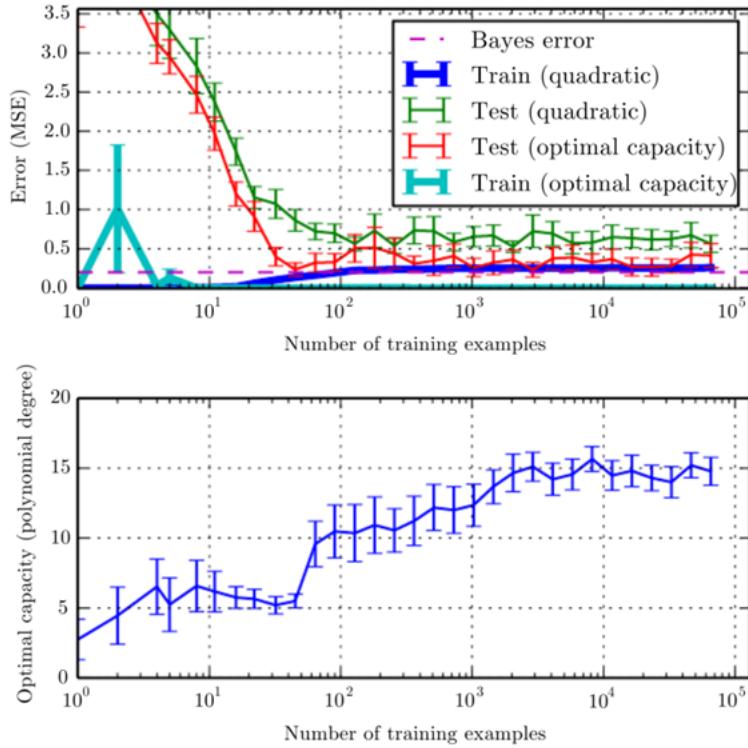


Figure 4.1: In questo grafico ho plottato l’andamento della loss-function sui diversi set di dati sulla base del modello che sto utilizzando. Nei primi casi sto utilizzando un modello quadratico per fitare dei dati estratti da un polinomio di gradi 15, naturalmente in questo caso quello che si osserva è che all’aumentare del numero di dati nemmeno il training funziona perchè evidentemente i dati contengono più informazione di quella che in questo momento sono in grado di rappresentare. In questo caso, conoscendo anche con esattezza il modello sono in grado anche di valutare l’errore bayesiano

4.3.2 Random search

Anzichè fare la griglia chiedo a un generatore di numeri casuali di darmi delle coppie di parametri quello che si osserva è che il random search, aggiungendo un po’ di stocasticità può aiutarmi a trovare quello che mi interessa. Ad esempio nella figura si osserva che utilizzando un algoritmo con una parte stocastica sono in grado di coprire parzialmente anche il picco 4.2 Il vantaggio di queste due tecniche è che **non esiste una correlazione tra un punto e il successivo** quindi posso eseguire dei fit in parallelo.

4.3.3 SMBO

Questo gruppo di tecniche prende il nome di Sequential Model-Based Optimization. L’idea è quella di non solo provare, ma man mano che proviamo anche fare un *guess* dell’andamento della loss function, quindi l’algoritmo fa dei tentativi e poi cerca di capire il minimo (eg. TPE Tree-Structured Parzen Estimator). Il vantaggio è che in questo caso riduco il numero di step necessari a migliorare gli iperparametri. L’idea è quella di stimare la mia funzione loss ad un altro algoritmo

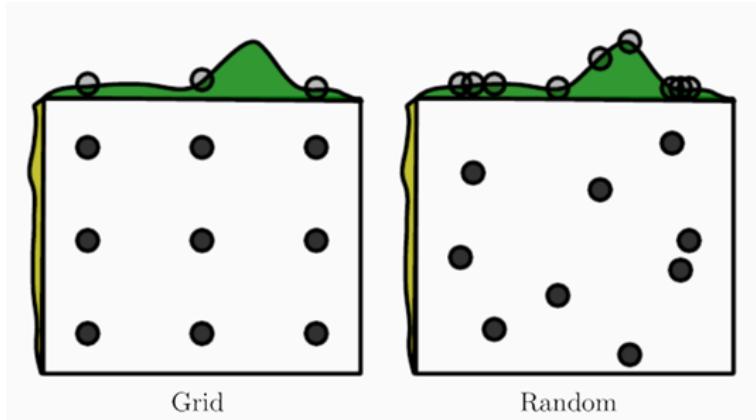


Figure 4.2: Confronto dei metodi grid e random search nell'esplorazione dello spazio degli iperparametri

statistico di tipo bayesiano che sia poi in grado di trovare i minimi.

L'idea è che l'SMBO fa una minimizzazione della funzione f che però è una funzione molto complicata nel senso di molto pesante dal punto di vista computazionale (eg. la loss function deve fare tutta la minimizzazione ma deve fare anche l'ottimizzazione degli iperparametri). Allora posso creare una nuova funzione surrogata, che sia più facile da maneggiare che **riesce a proporre dei nuovi punti da esplorare** perciò mano a mano che procedo in modo iterativo con l'algoritmo quello che succede è che non solo esplorerò meglio lo spazio e conoscerò meglio la *landscape* della mia funzione ma migliorerò anche la mia funzione surrogata \bar{f} .

L'algoritmo parte da una loss function, un numero di tentativi e una inizializzazione della funzione surrogata. Inizializzo una history dei miei tentativi e ad ogni iterazione dell'algoritmo cerco il minimo migliore per una funzione L che valuta la mia funzione surrogata (questo è il grosso dell'algoritmo che deve valutare che cosa sia questa funzione L). Dopo di che, una volta ottenuta la funzione surrogata posso valutare la loss nel punto fissato. Questa valutazione mi consente a questo punto di costruire un nuovo modello iterando il procedimento.

TPE L'SMBO cerca di fare un surrogare la mia funzione con un modello probabilistico, in sostanza è come se cercassi di capire la forma di una densità di probabilità e una volta che ho questa distribuzione posso fare il sampling e di conseguenza avere dei nuovi tentativi di massimizzare questa probabilità (come in un normale sampling probabilistico). Il vantaggio è che in questi stessi punti la mia loss function mi dà i migliori risultati, i miei iperparametri sono meglio ottimizzati. Il punto di valutazione successivo è calcolato con il seguente *improvement criterion*:

$$EI_{y^*}(\mathbf{x}) = \int_{-\infty}^{+\infty} \max(y^* - y, 0)p(y|\mathbf{x})dy \quad (4.1)$$

Questa quantità misura quanto la loss function mi aspetto che sia inferiore rispetto ad un certo valore di soglia y^* scelto in maniera tale che $p(y < y^*) = \gamma$ dove γ è un parametro dell'algoritmo. A questo punto posso applicare il teorema di Bayes per calcolare $p(y|\mathbf{x})$:

$$p(\mathbf{x}|y) = l(\mathbf{x}) \text{ se } y < y^* \quad (4.2)$$

$$p(\mathbf{x}|y) = g(\mathbf{x}) \text{ se } y \geq y^* \quad (4.3)$$

Dove $l(\mathbf{x})$ e $g(\mathbf{x})$ sono le distribuzioni di probabilità stimate usando gli \mathbf{x}_i di prova tali che $f(\mathbf{x}_i)$ è rispettivamente più piccola o più grande (eventualmente uguale a) di y^* .¹

Quindi EI per l'algoritmo TPE ha una soluzione chiusa

$$EI_{y^*}(\mathbf{x}) = \int_{-\infty}^{+\infty} \max(y^* - y, 0) \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} dy \propto \left(\gamma + \frac{g(x)}{l(\mathbf{x})(1 - \gamma)} \right)^{-1} \quad (4.4)$$

4.4 Cross-validation

A questo punto, una volta vista l'ottimizzazione degli iperparametri si possono unire le varie componenti del processo di learning fino alla cross-validation. Considero nuovamente la suddivisione del dataset, la questione ora però è che se la validation è unica può essere che io mi fermi in un punto che non è ideale, quindi devo combinare le fasi di training e validation in maniera tale da ottenere il risultato migliore possibile e allo stesso tempo fare fine parameter tuning.

Uno dei meccanismi più utilizzati per aggirare questo problema è la cross validation per cui prendo i dati e li suddivido in più partizioni e di volta in volta vario la partizione "a rotazione" in modo tale da non lavorare sempre con lo stesso data-set. Dopo aver fatto diverse rotazioni per calcolare il risultato finale che passo all'algoritmo per l'hyperparametrization utilizzo delle metriche. Esistono diverse tecniche per la corss validation: ci sono quelle di tipo *leave-one-out* o *leave-p-out* che sono tecniche esaustive.

Un esempio sono le k -fold corss validation che sono tecniche *non-exhaustive*, è una delle tecniche più utilizzate anche se un po' più rozza rispetto alle tecniche esaustive (che invece sono molto più lunghe e molto più pesanti).

Parto da un dataste completo, per prima cosa definisco quante partizioni voglio k (indica proprio questo numero), definisco quindi la dimensione del mio test set, dopo di che, le altre $k - 1$ partizioni sono utilizzate come dati di training. A questo punto svolgo tutto il processo di allenamento della rete neurale, dopo di che ripeto il processo facendo variare la partizione che utilizzo come dataset di testing. A questo punto ho k risultati su cui calcolo la media, questo valore medio della loss-function

In questo modo ho risolto due problemi: il primo è quello del bias dei dati che utilizzo come training e come validation (e soprattutto del quanti dati uso per ciascuna delle due parti) inoltre passo all'algoritmo per l'iper-ottimizzazione il valore medio della loss function che è sicuramente da un punto di vista statistico è più significativa di un singolo tentativo.

¹è come se stessi facendo un importance sampling

Chapter 5

Lezione 5

L'argomento di questa lezione sono le *recurrent neural networks* o RNNs. Questo tipo di reti neurali richiede la conoscenza di argomenti di matematica e geometria quali ad esempio i grafi.

5.1 Computational graphs

Per trattare questo primo argomento supponiamo di avere una loss function che dipende da tre parametri

$$J(a, b, c) = 3(a + bc) + 2(b + a) \quad (5.1)$$

Se volessi rappresentare la J da un punto di vista grafico dovrei disegnare qualcosa del tipo 5.1, dove parto dagli input e poi calcolo in diversi momenti diverse parti dell'operazione, per questo motivo utilizzo diversi layers.

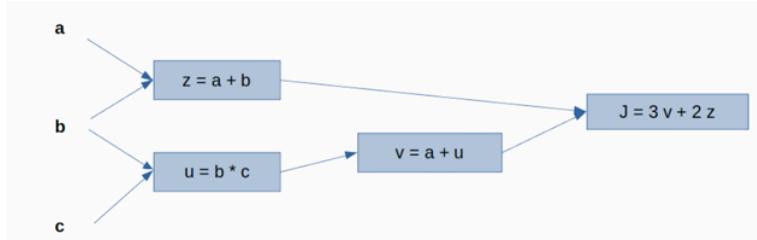


Figure 5.1: Grafo per operazioni concatenate

In questo modo la costruzione di z e u può essere fatta in parallelo. TF quando riceve una operazione di questo tipo la prima cosa che fa è lo splitting, l'individuazione dei nodi e poi la ricerca di una parallelizzazione possibile. Questo grafico è un primo sempio di **grafo**.

Da un punto di vista tecnico un grafo è una sorta di *catena di eventi*, è una **struttura che raggruppa operazioni** (non per forza operazioni algebriche), il secondo punto è che, considerata la **connettività** tra i diversi nodi posso costruire una **catena di eventi** in cui non tutti gli eventi dipendono dagli altri. Un grafo è definito da un insieme di **nodi** o vertici, e di **archi**. Un arco è detto **diretto** se gli archi sono orienati, o equivalentemente se è possibile creare un loop nel grafo.

Nota: Anche il MLP è un grafo, in questo caso però non sono presenti cicli, si parla pertanto di DAG (Acyclic Directed Graph). I diagrammi su cui si concentra

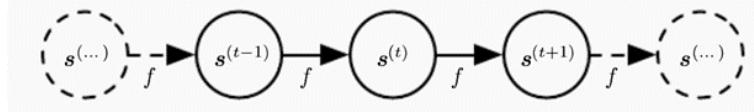


Figure 5.2: Diagramma per una serie temporale

questa lezione sono i grafi direzionali, ovvero all'interno dei quali posso definire una ricorrenza. Un altro esempio di grafi che è utile tenere a mente per il seguito sono quelli che rappresentano un sistema concatenato del tipo 5.2. Questo sistema è sostanzialmente una serie temporale, come indica la dipendenza da t , dove il nodo tratteggiato rappresenta tutti gli istanti precedenti al primo rappresentato, analoga considerazione vale per l'ultimo nodo. I nodi quindi, rappresentano stati di un sistema in istanti diversi, le funzioni f che collegano gli stati dovranno processare lo stato dell'istante precedente in quello successivo.

Da un punto di vista formale, matematico, per descrive il grafo rappresentato in 5.2 si può utilizzare l'equazione seguente:

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \mathbf{w}) \quad (5.2)$$

dove \mathbf{w} rappresenta gli eventuali parametri da cui dipende la funzione f . Questa relazione contiene una ricorsione descritta in modo compatto. Questa relazione è una relazione **folded**, compressa, mi rappresenta una chain rule che in realtà può essere molto estesa (come si vede dalla rappresentazione grafica del grafo). Per rappresentare in maniera esplicita il grafo avrei dovuto utilizzare una relazione **unfolded**.

5.2 Recurrent Neural Networks

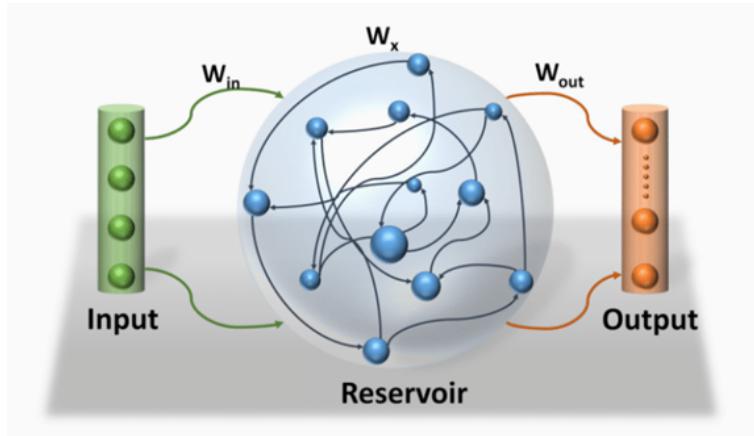


Figure 5.3: Echo state network

Negli ultimi 40 anni, c'è sempre stato l'obiettivo di costruire una **Echo State Network**, è un tipo di architettura in cui ho degli input pesati da una matrice dei

pesi e poi una architettura strana, una matrice multidimensionale chiamata **reservoir**, che costituisce sostanzialmente una riserva di pesi. All'interno di questa architettura, come quella rappresentata in figura 5.3, i pesi sono collegati in maniera stocastica. Un po' come un cervello multidimensionale in cui non ho delle regole ben precise che collegano i diversi nodi al suo interno.

Il beneficio principale di questo modello sarebbe l'aumentare ancora di più il comportamento non lineare di questa architettura, a questo punto il flusso dell'informazione, la flessibilità di questo modello in cui nulla è definito, sarebbe di fatto il massimo della potenza e della capacità di calcolo che possiamo attualmente immaginare. Naturalmente il problema principale di questa architettura riguarda l'allenamento, ad ora non si è mai riusciti ad allenare correttamente una architettura di questo tipo.

Tuttavia da questo modello sono nate diverse altre architetture, ad esempio le macchine di Boltzmann sono nate a partire da questo modello.

Supponiamo di avere una sequenza di punti $\mathbf{x}^{(t)}$

$$(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}) \quad (5.3)$$

sono una serie di numeri che sono correlati (ad esempio una serie temporale di dati correlati e, tramite la dipendenza dal tempo casualmente collegati eg. predizione per serie temporali come stock market, scrittura o parola in automatico, scrittura di un testo ...).

La domanda quindi è come fare a scrivere un modello di Deep Learning che possa trattare questo tipo di problemi; MLP infatti da questo punto di vista è abbastanza limitato perché quello che succede è che si perdono tutte le eventuali correlazioni temporali presenti.

La prima volta che si è riusciti ad allenare ed utilizzare una architettura RNN di questo tipo fu nel 1986. Per costruire questa architettura si parte dal vettore posizioni, dalla misura effettuata a diversi istanti e poi si applica la f ad un valore h da definire. Questo processo può essere rappresentato graficamente come in 5.4. Dove nella rappresentazione folded ho la \mathbf{x} che entra e tramite f mi restituisce

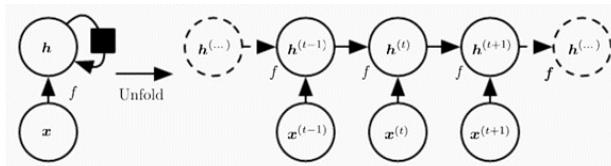


Figure 5.4: Rappresentazione RNN per serire temporale - modello grafo più semplice

un nuovo stato \mathbf{h} , dopo di che ho un box esterno che mi dice che su \mathbf{h} esiste una relazione di ricorrenza. Se rappresento lo stesso grafico in modalità unfolded ottengo invece il grafico di destra, per cui ad ogni istante t passo la \mathbf{x} ad \mathbf{h} tramite f :

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \mathbf{w}) \quad (5.4)$$

Si noti che questo è un esempio di RNN *senza output*. Da un punto di vista pratico di costruzione di reti neurali, ci sono diversi design che si sono dimostrati utili e che vedremo nel seguito.

5.2.1 Model 1

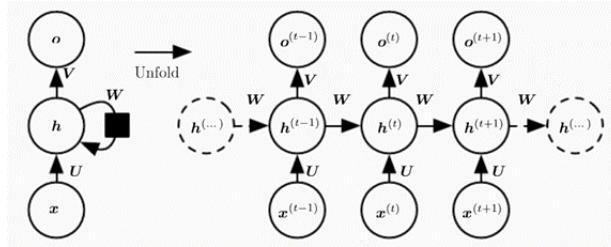


Figure 5.5: Rappresentazione modello 1

In questo caso la RNN produce un output ad ogni tempo t e ho una sorta di connessione tra tutti i layer che sono *hidden*, nascosti. Quindi il design è fatto in maniera tale che l'input \mathbf{x} viene modulato da una matrice di pesi \mathbf{U} , vengono generati gli hidden state che comunicano con tutti i loro precedenti (agli istanti precedenti) mediante una matrice di pesi \mathbf{W} che pesa gli \mathbf{h} tra un layer e un altro e poi ho una terza matrice di pesi \mathbf{V} prima che modula gli \mathbf{h} fino ad ottenere l'output.

La cosa importante da tenere presente di questo modello è che esiste una correlazione e ho una propagazione dell'informazione da tempi precedenti a tempi successivi. Questo modello è il più usato perché mi consente di avere una predizione istante per istante del mio input perché a ciascun passaggio la rete mi produce un output.

5.2.2 Model 2

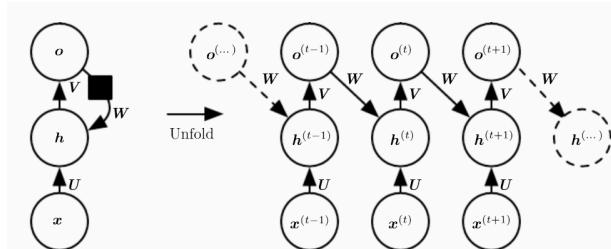


Figure 5.6: Rappresentazione modello 2

In questo caso l'input \mathbf{x} genera con una matrice di pesi \mathbf{U} l'hidden variable \mathbf{h} che però *non è più in connessione con se stessa* ma, tramite la matrice \mathbf{V} viene generato l'output. Con questo output si crea il loop, l'output viene iniettato, tramite la \mathbf{W} per generare \mathbf{h} . In questo caso la ricorrenza non è più su un singolo elemento ma è su due elementi.

Questo modello è interessante perché mi consente di correlare l'Informazione nello spazio esterno delle osservabili, non nello spazio degli stati hidden.

5.2.3 Model 3

Questa RRN ha connessioni ricorsive tra le diverse hidden units ma non generiamo degli output per ogni tempo t , aspettiamo un tempo finale in cui tutta

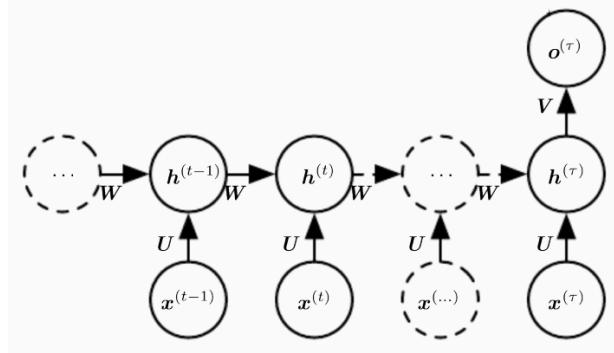


Figure 5.7: Rappresentazione modello 3

l'informazione viene processata e poi viene fornito un unico output finale. Quindi ho ad ogni step un input \mathbf{x} che viene modulato un tensore \mathbf{U} a dare la mia \mathbf{h} . A sua volta \mathbf{h} viene modulata con il tensore \mathbf{W} e combinata con il nuovo input, fino all'ultimo step in cui questa combinazione, modulata tramite il tensore \mathbf{V} viene restituita come output. Questo tipo di architettura viene utilizzato ad esempio per costruire dei classificatori.

5.2.4 Costruzione di RNNs

Se consideriamo ad esempio il primo design che abbiamo visto, di fatto applichiamo sempre la medesima procedura: per prima cosa calcolo una sorta di activation function come $\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}$, il parametro \mathbf{b} è un bias che posso eventualmente inserire per migliorare l'apprendimento della rete neurale, il risultato di questa operazione viene passato come argomento ad una funzione non lineare (generalmente si utilizza una tangente iperbolica) per ricavare la $\mathbf{h}^{(t)}$. A questo punto la $\mathbf{o}^{(t)}$ sarà data da un coefficiente c + il tensore dei pesi per l'output moltiplicato per l'hidden precedente. Quindi $\mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$.

Questa è la formula che poi devo propagare su tutti i layers. In questo modo ho definito la **forward pass** cioè la costruzione del meccanismo che fa andare avanti l'apprendimento della rete neurale.

5.2.5 Allenamento di RNNs

Una volta capito come costruire la rete neurale il problema che si pone è quello dell'allenamento. In generale si può scrivere un algoritmo di back-propagation anche per le RNNs, infatti i gradienti vengono calcolati utilizzando metodi detti **BPTT** ovvero *back-propagation thorough time*.

Suppongo di avere una loss function che a differenza di quelle precedenti deve accogliere al suo interno una serie temporale in cui ciascun elemento della serie è un elemento multidimensionale. A questo punto quello che sto facendo in sostanza è **definire la loss function totale come somma delle singole loss function**¹.

¹Per i dettagli dell'algoritmo e del calcolo del gradiente vedere le slide della lezione oppure le immagini 10.10 e 10.11 dell'appendice

5.3 Bidirectional RNNs

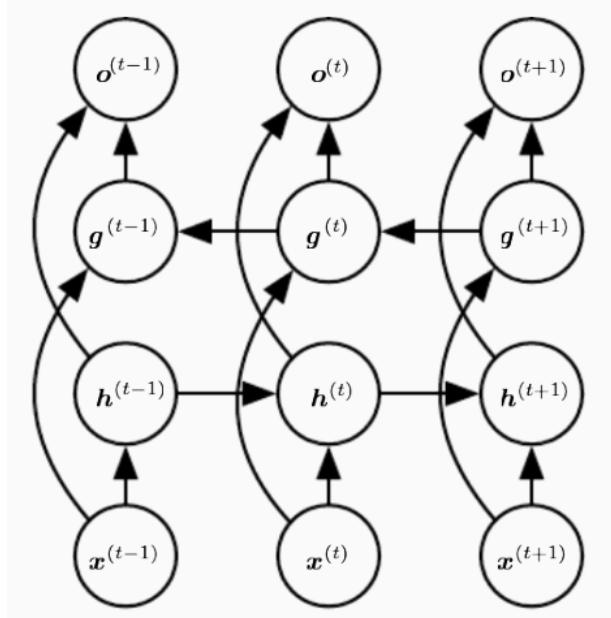


Figure 5.8: Esempio bidirectional RNN

Abbiamo visto che in generale è possibile utilizzare metodi di minimizzazione della loss function che fanno uso del gradiente anche per reti neurali di tipo RNN, tuttavia, non tutti i tipi di architetture supportano questo algoritmo. Ad esempio vediamo ora un'architettura del tipo bidirectional RNN in cui la propagazione dell'informazione dell'hidden state non va solo in una direzione ma può andare sia avanti che indietro nel tempo.

5.4 Bidirectional RNNs

Questa architettura così com'è non ha alcuna applicazione pratica, è stata utilizzata però per costruire architetture encoder-decoder, per cui l'architettura della rete viene separata in due parti (5.9): la parte superiore *encoder* e la parte inferiore *decoder*. La logica è che io vorrei avere un modello che sia in grado di comprimere i dati in uno spazio più piccolo (*encoder*) e dopo averci lavorato riportarli nello spazio originale (*decoder*). Questo modello viene utilizzato ad esempio per traduzioni tra lingue diverse, per cui è conveniente fare una compressione in uno spazio più piccolo per poi decomprimere l'informazione e tornare nello spazio originario.

5.5 Deep Recurrent Neural Networks

Finora abbiamo sempre parlato di modelli di reti neurali, non di *deep* neural network. Per trasformare una RNN in Deep-RNN dovrà aggiungere unità di hidden units, per cui, ad esempio nell'immagine 5.10 aggiungo un ulteriore strato di hidden units, z . Oppure posso ad esempio saltare alcune connessioni ed avere più

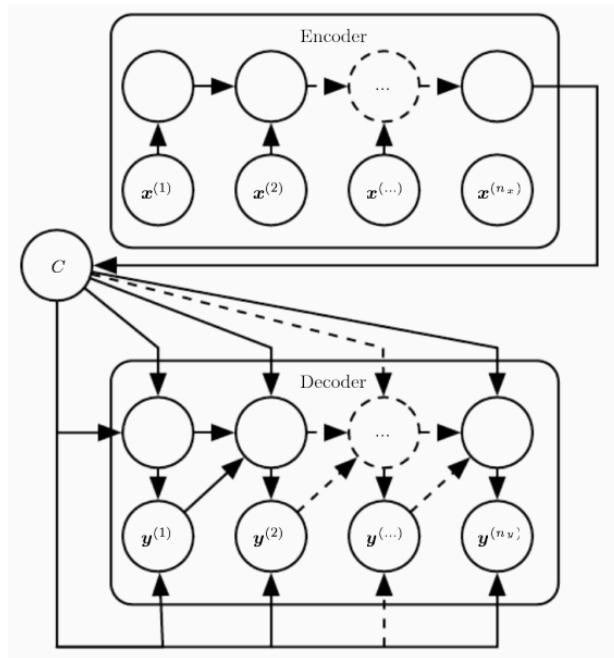


Figure 5.9: Esempio bidirectional RNN

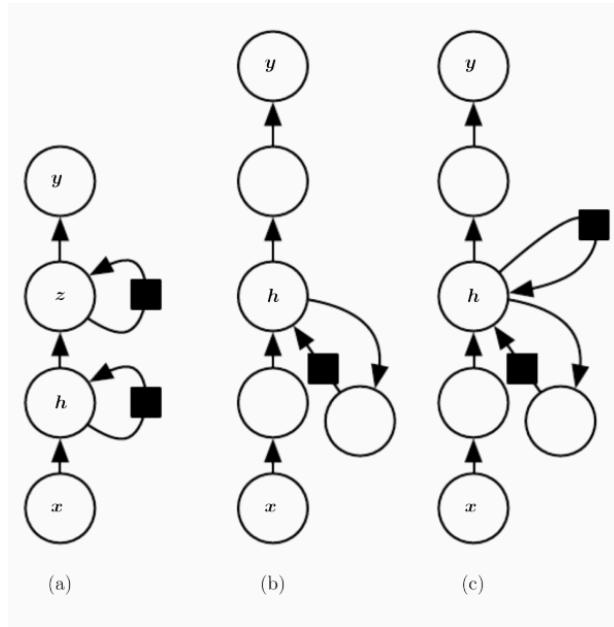


Figure 5.10: Esempio bidirectional RNN

layer ricorsivi. Da un punto di vista pratico quindi quello che si deve fare è semplicemente aumentare il numero dei parametri, dei layers, e dei nodi.

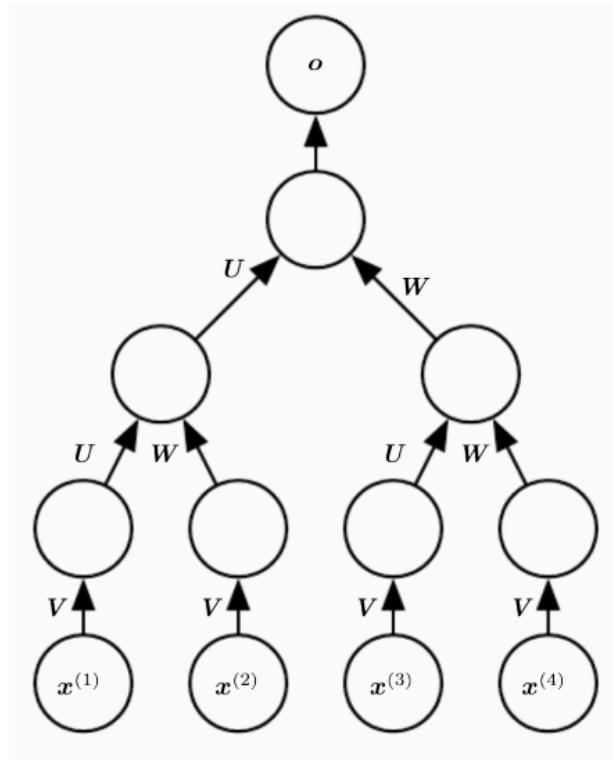


Figure 5.11: Esempio bidirectional RNN

5.5.1 Recursive Neural Network

Tra i diversi modelli di RNNs, una delle famiglie più importanti è quella delle *recursive neural network* o **RvNN**. Questo tipo di reti neurali può essere usato per un tipo specifico di predizioni, ha determinate caratteristiche dal punto di vista dell'architettura, prende come input una sequenza correlata però è una rete più semplice rispetto ad RNN. Questo tipo di rete ha avuto una grande applicazione nell'ambito del Natural Language Processing. Un esempio di architettura è rappresentato in 5.11. In fisica modelli di questo tipo vengono utilizzati molto raramente.

5.5.2 Problemi delle DRNN

Uno dei principali limiti all'utilizzo estensivo di RNNs è l'allenamento: se ho degli hidden states $h^{(t)}$ sicuramente il gradiente è la somma su tutti i tempi e su tutti gli hidden units, il problema è che facendo il calcolo effettivamente si ottiene una certa quantità ma le funzioni di attivazione che sto utilizzando sono funzioni che hanno una derivata molto limitata. Se calcolo il prodotto tra numeri che sono di fatto definiti tra 0 e 1, in generale con numeri vicino a zero ottengo un numero numericamente molto piccolo. Il gradiente di fatto quindi è proporzionale al gradiente dell'activation function rispetto agli hidden layer che però, finché utilizzo funzioni di attivazione non lineari, tendo a 0. Quindi in sostanza i miei parametri sono controllati da un numero che non è in grado di darmi informazioni su una direzione di movimento utile. Questo tipo di comportamento prende il nome di *vanishing gradient*

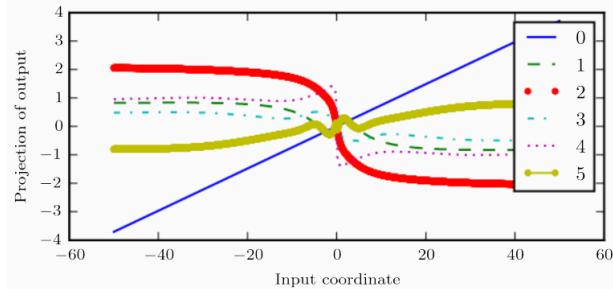


Figure 5.12: Esempio output RNN

Variazione dell'output in funzione dell'input rispetto alla funzione di attivazione che utilizzo, in blu è rappresentato il risultato dell'applicazione di una funzione di attivazione lineare, in verde è rappresentata l'applicazione di una funzione *tanh*, in rosso l'applicazione successiva di due tangenti iperboliche ecc..

Quindi il mio modello risultata essere praticamente privo di informazione, quindi si ferma molto prima di quanto in realtà avrei bisogno per una corretta definizione dei parametri, si ferma quasi sicuramente in un minimo locale.

Quindi le RNNs (soprattutto le DRNNs, in cui l'input passa più volte all'interno dell'activation function) generano naturalmente gradienti che sono piccoli ed avere un risultato non ideale, quindi si ha una sorta di perdita di informazione, di fatto le RNNs si dimenticano facilmente dell'informazione di un tempo precedente.

Quindi ho architetture che funzionano bene solo se l'autocorrelazione o la correlazione temporale della sequenza è piccola. Per questo motivo nascono le cosiddette *short-term memory*.

5.6 Long Short Term Memory

Le LSTM sono architetture che sfruttano di fatto le proprietà delle RNNs. si tratta di una rete ricorsiva che fa delle operazioni un po' diverse rispetto a quelle viste finora. Per parlare di questo tipo di modello si introduce il concetto di **porte** che regolano il flusso d'informazione.

La linea sopraficcia indica la direzione in cui si propaga l'informazione., questa costituisce una sorta di componente che mantiene la memoria del sistema mentre sulla parte inferiore ho la x di input che viene passata a diverse activation function e che viene modulata da h , dopo di che ci sono diverse operazioni che cercano di modulare il risultato dell'activation.

Partendo dalla riga superiore, l'idea è quella di avere una cellula che si occupa della memoria, quindi che sia in grado di portare avanti nel tempo l'informazione delle hidden unit e dell'input e dell'output (anche lui viene passato a questo blocco dopo essere stato modulato Questo box è una sorta di summary di tutte le informazioni che vengono iniettate ed elaborate dalla rete. In particolare, l'equazione che regola l'informazione che viene conservata e che viene persa nel tempo è la seguente

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \quad (5.5)$$

dove la f_t mi dice quanta parte di informazione dell'istante precedente viene dimenticata dalla rete, quindi si tratta sostanzialmente di una porta di tipo forget; mentre la i_t gioca il ruolo del peso che posso applicare all'input iniziale, decide

quanta informazione di input viene conservata, quindi gioca il ruolo di una porta di input, decide quanta informazione di input voglio ricordarmi.

Il forget gate (parte a sinistra nell'immagine) ha possibilità di decisione su cosa fare con l'informazione ed è definita come prodotto tra il sigmoide per la somma di una serie di termini, ciascuno pesato per una propria matrice dei pesi. Per cui è definito come

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \quad (5.6)$$

Spostandosi sul blocco centrale ci si concentra invece sulla nuova informazione che viene memorizzata, per cui:

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \quad (5.7)$$

$$\tilde{C}_t = \sigma_g(W_c x_t + U_c h_{t-1} + b_c) \quad (5.8)$$

In questo caso quindi ho due funzioni di attivazioni diverse: la sigmoide che decide quali valori aggiornare e come aggiornarli, mentre la tangente iperbolica sta generando nuovi candidati per la memoria futura. Conoscendo questi componenti in sostanza posso costruire l'equazione che mi consente di evolvere il sistema da C_{t-1} a C_t .

Infine, l'**output gate** fa un'operazione di post-processing dello stato in cui si trova la cella per cui si avrà

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \quad (5.9)$$

e

$$h_t = o_t \sigma_h(C_t) \quad (5.10)$$

Le LSTM funzionano molto bene nell'ambito delle serie temporali, oltre alle LSTM, qualche anno fa è nata anche la GRU che ha un'architettura più semplice, sostituisce i blocchi dell'LSTM con due blocchi ovvero un reset gate e un update gate.

Chapter 6

Lezione 6

6.1 Immagini digitali

Un’immagine è un artefatto che rappresenta una nostra percezione della realtà. Ci sono diversi modi per ottenere un’immagine, ci sono immagini che dipendono da due coordinate e volumi (rappresentati mediante tensori anziché matrici) che sono tridimensionali.

Le immagini possono essere catturate tramite metodi naturali o tramite strumenti ottici (tutto ciò che riguarda telescopi, lenti microscopi ecc.). Quello che ci interessa è la *digitalizzazione* dell’immagine. Un esempio può essere una rappresentazione mediante matrice in scala di grigi. Quando parliamo di immagini digitali possiamo fare cose diverse, tra cui

- salvataggio e condivisione di immagini
- applicazione ed implementazione di algoritmi per processare le immagini

Quando definiamo un’immagine da un punto di vista digitale si possono fare diverse scelte che dipenderanno fondamentalmente da due parametri che sono **risoluzione** e **densità**. Ad esempio, se si osserva la figura 6.1, entrambe le immagini presentano la stessa risoluzione, tuttavia con densità diverse. Infatti, per **risoluzione** si intende la misura che i miei pixels occupano, la forma totale, lo spazio che potrei occupare, però a destra si vede meglio il cerchio perché ho una densità di pixels per inch più alta, una PPI più alta e questo fa sì che la qualità dell’immagine di destra sia migliore.

6.2 Computer Vision Problems

6.2.1 Classificazione

L’input è un set di immagini che devono essere classificate, le immagini di input devono rispettare una serie di caratteristiche per cui devono possedere stessa densità e risoluzione. Una volta fornito l’input l’informazione dell’immagine deve essere propagata in una rete neurale perché poi l’immagine possa essere classificata. Questo tipo di compito è generalmente svolto dalle reti neurali convoluzionali.

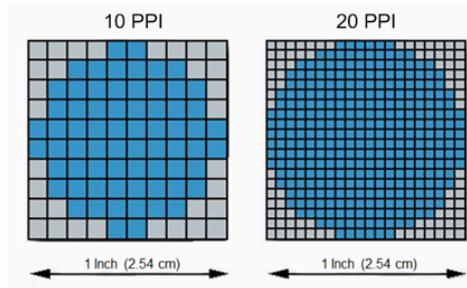


Figure 6.1: Le due immagini rappresentano entrambe un cerchio. Entrambe le immagini hanno la stessa risoluzione, tuttavia sono caratterizzate da una densità (o PPI - Pixels Per Inch) diverse

6.2.2 Riconoscimento di oggetti

Nel problema del riconoscimento degli oggetti la task viene "complicata" rispetto alla semplice classificazione, in questo caso infatti ci si deve occupare dell'identificazione di un oggetto all'interno di una immagine che ha una dimensione variabile. Per cui va identificata la presenza dell'oggetto, la sua collocazione all'interno dell'immagine (insieme a tutti gli eventuali altri elementi presenti) e dopo di che questi oggetti devono essere classificati. L'algoritmo quindi deve essere in grado di riconoscere gli oggetti all'interno di un ambiente complesso e definirne i *bounding box* per capire dove si trovano. Esempio dell'output che ci aspettiamo è rappresentato in figura 6.2

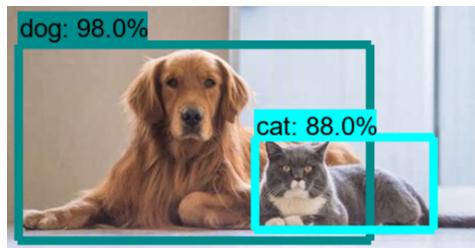


Figure 6.2: Esempio di output di image recognition

6.2.3 Neural Style Transfer

Esistono modelli che riescono a fare delle procedure di post-processing e generare delle immagini 6.3, per cui le reti neurali hanno imparato ad applicare dei filtri sulla base di autori diversi. L'idea è quella di allenare un modello generativo che sia in grado di capire le caratteristiche della pittura di un determinato autore e costruire delle mappe che poi possano essere applicate ad immagini reali.

6.3 Reti neurali convoluzionali - CNNs

Con un MLP siamo stati in grado di creare un classificatore, in questo caso ha funzionato tutto perché le immagini di MNIST hanno solamente 28 x 28 pixels. Il problema si pone quando si considerano task reali per cui posso avere, ad esempio

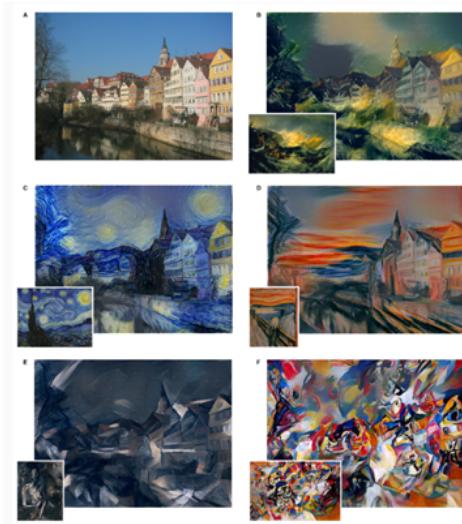


Figure 6.3: Esempi di utilizzo dei diversi algoritmi di pattern recognition

immagini da $1000 \times 1000 \times 3^1$ ho 3 milioni di input. Per costruire un MLP dovrei, solo per la lettura costruire una matrice di $1000 \times 3M$, quindi con 3 miliardi di entries. Naturalmente un numero così elevato di entries mi satura, solo per la lettura i parametri che ho a disposizione.

Il fatto di avere tanti parametri mi pone un problema non solo per quanto riguarda la potenza computazionale e la capacità di calcolo ma anche problemi di sovra-rappresentazione del modello. **Per fissare 3 miliardi di parametri avrei bisogno di un dataset di training enorme**, il che naturalmente rende la task ancora più complicata.

6.3.1 Digital Image Transformation

Per risolvere questo problema si utilizzano tecniche di **trasformazione delle immagini digitali**. Quando si hanno delle immagini si possono applicare diversi tipi di trasformazione:

- trasformazioni affini e filtri
- pattern recognition: obiettivo è quello di sviluppare operazioni logiche che mi portino come risultato finale ad una classificazione

Trasformazioni affini e filtri

Filtraggio Le operazioni di filtraggio possono essere effettuate utilizzando **prodotti di convoluzione**, per cui io definisco un kernel di convoluzione ω e lo applico ai valori di pixels (x e y) della mia immagine.². Per applicare il kernel devo lavorare su tutto il dominio dell'immagine (sommo su tutti i pixels).

$$o(x, y) = \omega * f(x, y) = \sum_{dx=-a}^a \sum_{dy=-b}^b \omega(dx, dy) f(x + dx, y + dy) \quad (6.1)$$

¹Le ultime 3 dimensioni sono quelle riservate al colore che viene espresso in rgb

² ω è in sostanza una matrice

dove $o(x, y)$ rappresenta l'immagine filtrata e $f(x, y)$ rappresenta l'immagine originale.

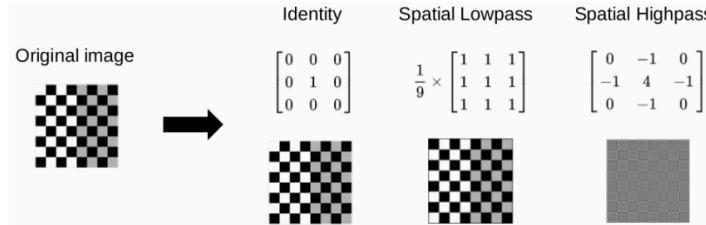


Figure 6.4: Nella b sto saturando il valore dei bordi, quindi ho i bordi più spessi. Nell'immagine (c) invece ho applicato un filtro per diminuire l'intensità e invertire i colori. In questo modo sono passata da una scala sostanzialmente bicromatica ad una scala di grigi.

Trasformazioni Affini Le trasformazioni sono rappresentate mediante una matrice, l'immagine modificata è data semplicemente dal prodotto tra la matrice di trasformazione e le coordinate dei singoli pixels dell'immagine originale

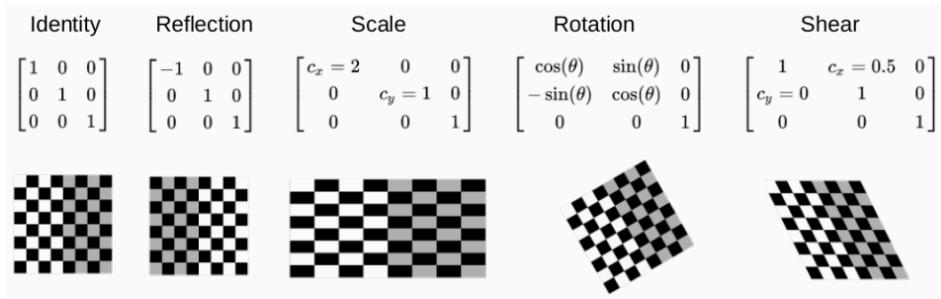


Figure 6.5: Esempi di trasformazioni affini

6.3.2 Struttura delle CNN

Una rete CNN è composta da due layer principali ovvero un layer di convoluzione e un layer di pooling. Applicati questi due layer, il risultato dell'applicazione di queste trasformazioni viene mappato in un layer di flattening che poi mi restituisce l'output finale. Le caratteristiche fondamentali di una CNN sono

- utilizzo di convoluzione per ridurre il numero di parametri da determinare
- evitare problemi dati dalla rotazione o dallo scaling dell'immagine

6.3.3 Feature map

Per prima cosa bisogna definire un convoluzione che mi consenta di ridurre il numero dei parametri, l'idea è che anziché associare informazione a ciascun pixel, cerco di comprimere l'informazione di un blocco di pixels.

Un operatore di convoluzione è definito come segue:

$$s(t) = (x * \omega)(t) = \int x(a)\omega(t-a)da, \quad (6.2)$$

dove $\omega(a)$ è una funzione peso, una sorta di funzione di densità di probabilità, $s(t)$ prende il nome di feature map ed è ottenuto come integrale dell'input con il kernel. La cosa importante in questo passaggio è che l'output ha una dimensione che dipende dalla dimensione del kernel (proprietà del prodotto tra matrici); quindi sono riuscita a ridurre la dimensione dell'immagine senza però perdere informazione perché ogni elemento dell'output contiene tutta l'informazione che è stata passata come input.

NB: Quando faccio ottimizzazione, quello che ottimizzo e modifco, sono i valori delle entries del kernel. Quindi la domanda che mi pongo è "Quali parametri, per una data shape di kernel, mi consente di costruire un buon classificatore?"

Aspetti negativi della convoluzione

Padding Il problema principale della convoluzione è che in generale i bordi non sono ben collegati tra di loro, rischio di perdere informazione importante che può essere presente lungo i bordi. Infatti ho tanta informazione sull'elemento centrale e il modo in cui questo è collegato agli altri (gli elementi centrali vengono utilizzati più volte durante l'operazione di convoluzione mentre i bordi vengono chiamati molto meno spesso).

Per risolvere questo probelma vengono utilizzate tecniche dette di **padding**, per cui ai bordi vengono aggiunti dei pixel fittizi con peso 0, in questo modo aumento virtualmente la dimensione dell'immagine. Si noti però che, con l'utilizzo del padding inserisco un ulteriore parametro da ottimizzare.

Stride Quando si utilizza una CNN è anche possibile definire uno stride, ovvero quanti step devo fare prima di applicare la convoluzione (ad esempio posso fare salti di 2 o 3 pixels). naturalmente in questo caso la dimensione dell'immagine finale dovrà tenere conto dello stride (se salto dei pixels l'immagine di output sarà più piccola). Tenedo conto sia dello stride s che del padding p la dimensione dell'immagine che esce dalla convoluzione può essere calcolata come:

$$\left(\frac{n+2p-k}{s} + 1\right) \times \left(\frac{n+2p-k}{s} + 1\right), \quad (6.3)$$

dove n è la dimensione dell'input, k è la dimensione del kernel di convoluzione, p il padding e s lo step dello stride.

Vantaggi delle CNN

Partendo dal presupposto che abbiamo un kernel più piccolo dell'immagine è che abbiamo delle matrici sparse (per un approfondimento si veda la sezione 10.4 della appendice), poi abbiamo dei parametri che vengono messi in comune, quindi creo una sorta di rete di correlazione tra i diversi pixels vicini ed è una rappresentazione **equivariante**, indipendentemente da dove parto il risultato finale sarà sempre in qualche modo legato alla matrice di partenza.

6.3.4 Rappresentazione dei colori

Utilizzando immagini colorate (che posso rappresentare in rgb), quello che succede è che devo per forza definire un kernel di convoluzione per ogni canale di colore, di conseguenza mi trovo a lavorare con un kernel di convoluzione 3D. Nonostante questo l'applicazione del kernel è comunque in grado di riportarmi ad una matrice

bidimensionale (come nei casi precedenti). Quello che verrà costruito quindi sarà una struttura analoga a quella rappresentata in 6.6, dai prodotti di convoluzione con i due filtri ottengo delle matrici flat che poi posso combinare. Lavorando con `tf` si ha la possibilità infatti di definire anche il numero di filtri che voglio applicare, da ciascun filtro estrarro l'informazione che voglio, dopo di che posso ri-unificare questa informazione e proseguire con l'algoritmo.

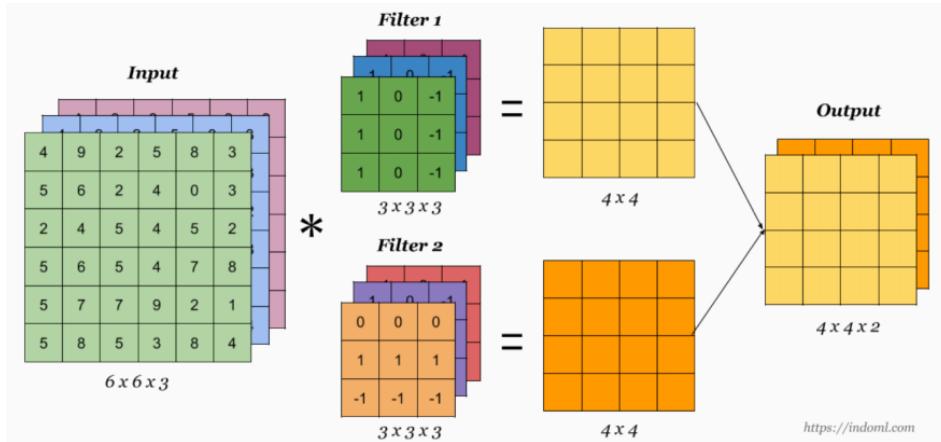


Figure 6.6: Esempio di rete convoluzionale in cui sono applicati due filtri e in cui ho un kernel di evoluzione per ogni canale di colore

Quindi, completando quanto visto prima posso, per ottenere un output, applicare una activation function non lineare. Quindi utilizzo diversi filtri, una convoluzione per ciascun canale, unisco i risultati, utilizzo una funzione di attivazione non lineare fino all'output (vedi Fig. 6.7). L'aspetto importante è che abbiamo diminuito di molto il numero di parametri da determinare, infatti gli unici parametri da determinare sono quelli del kernel.

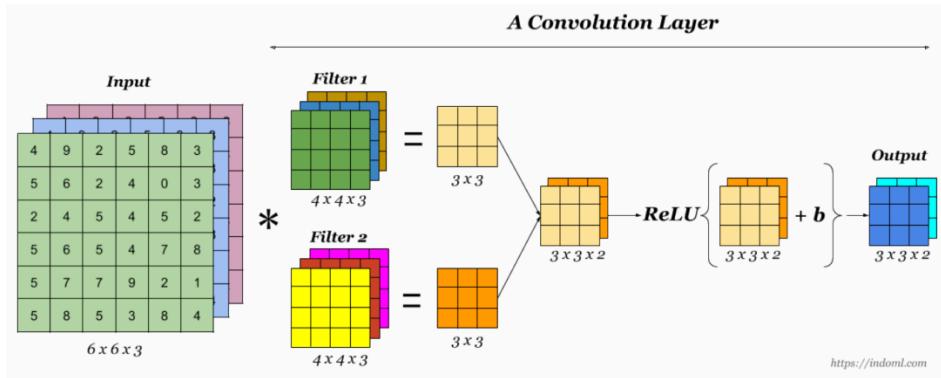


Figure 6.7: Esempio di rete convoluzionale in cui sono applicati due filtri e in cui ho un kernel di evoluzione per ogni canale di colore

6.3.5 Pooling Layer

Per le CNN esistono due tecniche di pooling che sono le più utilizzate, ovvero **max pooling** e **average pooling**. Nel caso del max pooling, una volta definito il set-

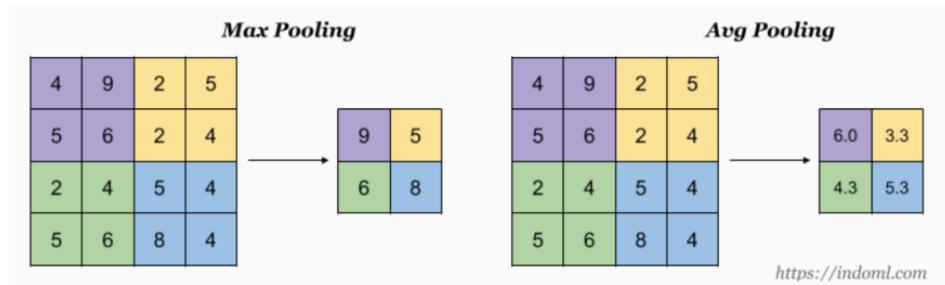


Figure 6.8: Esempio di applicazione delle tecniche di pooling

tore su cui voglio fare pooling, ne setto il massimo, in questo modo non sto solo riducendo la dimensione dello spazio su cui sto lavorando ma sto anche facendo una semplificazione, infatti sto decidendo di eliminare tutta l'informazione che non è numericamente rilevante.

Nel caso dell'average pooling, quello che succede è che, anziché restituire il massimo per ciascun settore, mi faccio restituire il valore medio per ciascun settore. Un esempio di queste tecniche si ha in figura 6.8. Il vantaggio delle tecniche di pooling è che pur perdendo informazione, mi consente di conservare l'informazione fondamentale.

Il vantaggio dell'utilizzo del pooling consiste nel fatto che se fornisco come input la matrice originale o la trasposta o la ruotata, *comunque* i numeri che ottengo saranno gli stessi anche se collocati in posti diversi. Questo mi dà proprio la garanzia che il modello che applicherò dopo i layer di convoluzione abbia input simili, numericamente da una matrice all'altra come input non mi cambia molto. Quando dopo la convoluzione applico ad esempio un MLP che devo allenare questo riceverà degli input simili.

Al termine di tutte le operazioni descritte sopra, si applica un layer di flattening che mi restituisce un **feature vector**, è una sorta di riassunto dell'informazione dell'immagine.

Chapter 7

Lezione 7

Le CNN servono non solo per fare classificazione ma anche per task più complicate ad esempio posso fare una frammentazione semantica, identificazione dell'oggetto e localizzazione dell'oggetto ma anche cosa più complicate come localizzazione e identificazione di diversi oggetti e anche la s

Per affrontare problemi di questo tipo è utile definire alcuni concetti:

Bounding Box: è il rettangolo che delimita il perimetro dell'oggetto in termini di valori massimi; generalmente il bounding box viene rappresentato tramite i valori della coordinata sinistra in alto e della coordinata destra in basso.

Segmentare: in questo contesto significa attivare i pixel quindi disegnare una sorta di maschera che mi consenta di coprire l'oggetto che mi interessa classificare.

7.1 Transfer Learning

Il problema a questo punto è che lavorando con delle immagini si lavora in uno spazio molto ampio. Fin qui il problema è stato affrontato in maniera molto standard per cui abbiamo un singolo compito da eseguire (una sola task), un dataset che è molto ampio e un *problema isolato*, nel senso che, una volta che cambio dataset e cambio anche il singolo problema. Finora il dataset è sempre stato trattato come una cosa caratteristica del problema, non ci si è mai chiesti se l'informazione ottenuta da un dataset potesse essere propagata ad un altro dataset.

In un certo senso quello che vorremo fare è un **retraining**, salvare delle caratteristiche che ho imparato e trasmetterle ad un nuovo dataset. Implementare quest'idea all'interno del codice pone in realtà alcune difficoltà. Supponiamo di avere due dataset, di cui però il secondo molto più piccolo del primo, questo succede spesso in applicazioni di ambito sanitario. Quello che ci si chiede è se esiste una tecnica per trasferire l'informazione che ho estratto da un dataset più esteso ad un dataset più piccolo su cui devo affrontare il medesimo problema. L'altra domanda che ci si pone è se questa possibilità possa ridurre il tempo necessario per il training della rete (se so già come gestire l'immagine potrei utilizzare meno epoche).

Proprio in questo contesto nasce il **transfer learning** ho un dataset1 molto grosso da cui estraggo l'informazione che poi propago al dataset2, molto più piccolo ma su cui sono interessata a lavorare (vedi immagine 7.1). La tecnica prende il nome di transfer learning perché consiste in un **trasferimento della tecnica**

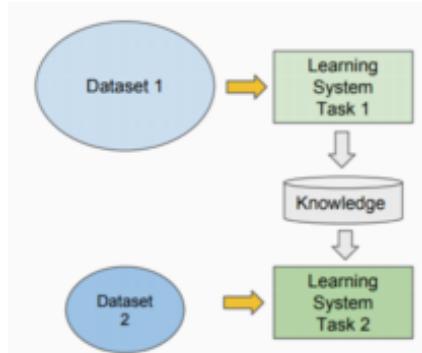


Figure 7.1: Workflow dell'applicazione di un algoritmo di transfer learning

di apprendimento. Il problema dell'avere pochi dati può essere ad esempio un raggiungimento molto veloce dell'overlearning e soprattutto la qualità del risultato della rete.

Quello che potrei fare ad esempio è una procedura di questo tipo: prendere un dataset che conosco molto bene, costruisco un ottimo classificatore che alleno e dopo di che trasferire il modello allenato su un altro problema (diciamo sulla classificazione di un altro dataset).

Esempio Se si tratta di un allenamento normale, prendo l'input e alleno il modello per il numero di epoche necessario. Quindi il training va fatto su tutta l'architettura, a questo punto un'opzione è quella di dire, una volta terminato l'allenamento, fisso tutti i parametri escluso l'ultimo layer, quello di predizione. Quindi quello che faccio è un "ri-allenamento" solo dell'ultimo layer, di questi pochi parametri.

L'idea del transfer learning è questa, tuttavia anziché "ri.allenare" solamente l'ultimo layer cioè quello della classificazione vera e propria, divido l'architettura della rete in due parti, per la prima parte della rete i parametri vengono fissati, a questa parte attacco una nuova architettura in cui tutti i pesi possono variare e vengono allenati sul dataset più piccolo.

Questo viene fatto perché variando solamente l'ultimo layer la variabilità dei parametri è molto limitata, quindi è difficile che l'architettura possa essere applicata in maniera efficiente anche ad un nuovo dataset.

Nota: questo è un approccio che ha senso se si ha a che fare con due dataset che hanno alcune caratteristiche:

- medesimo input (tipologia di input)
- uno dei due dataset è molto più ampi dell'altro
- ci sono effettivamente delle feature interessanti anche per il dataset più piccolo e che possono essere estratte dal primo dataset

Un'idea interessante a questo punto può essere quella di prendere le architetture più preformanti e utilizzarle come filtro, sopo l'applicazione del filtro posso allenare gli ultimi layer che mi consento di aggiustare il modello per la task specifica. Un esempio di applicazione di quest'idea è rappresentato nell'immagine 7.2.

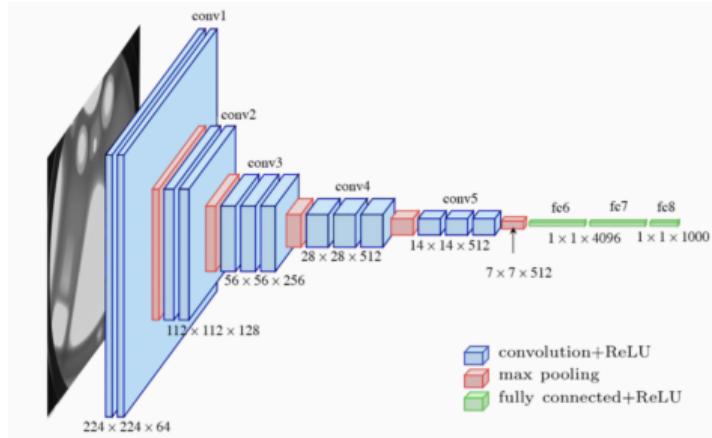


Figure 7.2: Rappresentazione dell'applicazione di un modello di transfer learning per il riconoscimento e la classificazione di immagini. In questo caso specifico, un'architettura costituita prevalentemente da CNNs viene utilizzata come "filtro" su un dataset simile. Come si può osservare dall'immagine i parametri da ri-allenare sono solamente una piccola parte del totale dei parametri del modello.

7.2 Object localization and detection

I problemi di localizzazione e identificazione "discendono" in un certo senso dai problemi di classificazione. In un problema di classificazione devo semplicemente capire di che oggetto si tratta. Il problema di localizzazione semplicemente aggiunge uno step al problema di classificazione per cui devo capire in quale zona dell'immagine si trova l'oggetto. Infine, nei problemi di identificazione si hanno immagini in cui sono presenti in generale più oggetti che vanno distinti, identificati e localizzati.

7.2.1 Loalizzatore

Per costruire un classificatore mi basta creare una CNN seguita da un MLP con una attivazione di tipo softmax. Per fare localizzazione, non solo vado a guardare le categorie finali ma inserisco, ma devo inserire una nuova parte che risolva la questione della localizzazione, per farlo inserisco altre coordinate, prima di tutto (b_x, b_y) che rappresentano il centro del bounding box. Assieme a queste due coordinate avrò anche le coordinate b_w , la *width* del bounding box e la sua altezza b_h *height*. Quindi ho introdotto quattro variabili in più, l'output dell'MLP non è più solamente quello dato dal softmax, ma anche queste nuove quattro variabili. Quello che vorrei fare è un classificatore insieme ad un regressore, questo è abbastanza particolare perché significa che **la predizione del modello sarà costruita ad hoc**. L'output sarà un vettore n-dimensionale, in cui

- il primo elemento è un booleano p_c . Se $p_c = 1$ allora ho trovato l'oggetto all'interno dell'immagine, altrimenti non ho trovato nulla. Questo può essere sensato perché all'interno del dataset non è detto che tutte le immagini contengano l'oggetto che voglio trovare.
- subito dopo trovo le quattro coordinate che fanno riferimento al bounding box b_x, b_y, b_w, b_h

- per ultime attacco le variabili corrispondenti alla categoria $c_1, c_2, \dots c_{n-5}$

In questo modo definisco l'output in maniera amolto semplice, infatti **posso utilizzare qualsiasi metrica vista finora per l'allenamento**

Sliding window

Se si passa ad esempi più complicati, eg. face-detection, il problema si traduce nel raccogliere tutte le coordinate dei punti che devo identificare e poi produrre una matrice o un tensore che le contenga.

Quindi per fare object detection, il modo più semplice di procedere è il seguente: partendo da un dataset la prima cosa che posso fare è allenare un classificatore di CNN che mi dica se un certo oggetto è presente o meno all'interno dell'immagine (creo un classificatore in sostanza). Il secondo step, per poter fare localizzazione, è quello di applicare questo classificatore, il modello che ho allenato precedentemente e poi farlo scivolare sull'immagine, in sostanza applico il classificatore ad una serie di sotto-immagini.

Il problema è che, se ogni volta che devo localizzare un oggetto in un'immagine devo riapplicare un tot di volte il classificatore avrò un tempo di esecuzione altissimo. Quindi, sicuramente è una tecnica funzionante ma non particolarmente efficiente

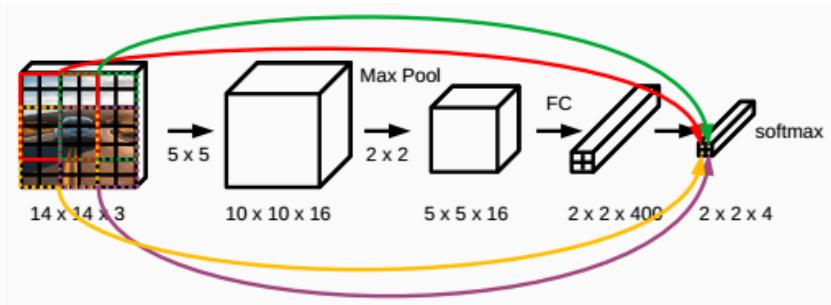


Figure 7.3: Esempio di architettura per un algoritmo a sliding window con un output tensoriale

L'idea, per migliorare la tecnica precedente è quella di prendere la CNN che ho usato in precedenza e sostituire il layer flat con un tensore (eg. da n nodi flat passo a un tensore $1 \times 1 \times n$) per ottenere qualcosa di più flessibile. Fatto questo modifco nuovamente l'architettura in cui, anziché utilizzare un tensore come quello precedente utilizzo un tensore $2 \times 2 \times n$, in cui cerco di mappare per ogni riga e colonna di quell'MLP "multiplo" una posizione della sliding window (per un esempio vedere 7.3). Quindi l'obiettivo è quello di avere un tensore finale che sia compatibile con la parte precedente dell'architettura e che sia tale che ogni spostamento della sliding window venga fatto in parallelo e ogni blocco contiene l'informazione ottenuta da una posizione della sliding window. Quindi sono in grado di avere un unico layer softmax con tutte le sliding window contemporaneamente.

L'idea è quella di provare ad avere subito come output della rete il bounding box, quindi avere un oggetto multidimensionale che tenga conto della griglia che ho utilizzato per definire lo scivolamento della finestra. Questa tecnica è abbastanza veloce se si ha, ad esempio una GPU.

Evaluating object localization Questo tipo di calcolo mi dà una griglia con tutti i diversi risultati e le diverse probabilità di localizzazione di un oggetto

rispetto ad un altro per le diverse immagini. Quello che può succedere è che si ottengano diversi bounding box che sono compatibili (eg. probabilità simile per i due bounding box di contenere l'oggetto). Quello che si fa in questi casi è misurare la distanza di overlap dei bounding box calcolando la **Intersect over the Union** (IoU), che si calcola come rapporto tra l'area dell'intersezione dei box e area dell'unione. Si considerano come **oggetti le previsioni con $\text{IoU} \geq 0.5$** . Alla fine prendo il bounding box che ha la probabilità più grande dopo aver superato il filtraggio della IoU.

Sono tecniche semplici che possono essere implementate, tecniche buone per problemi che possono essere risolti con tempi qualunque, non real-time.

Anchor boxes

Le tecniche viste finora possono avere diversi problemi, un primo problema si pone se devo fare la classificazione di due oggetti che hanno dimensioni diverse, soprattutto se so che all'interno del mio dataset sono diverse le immagini con questa patologia. Dal punto di vista dell'architettura, la soluzione migliore a questo tipo di problema è quella di definire degli **anchor box** specifici per ciascun oggetto, per cui definisco la forma del target output, aggiungo all'interno nel mio target una seconda tipologia di anchor box che lavorerà con coordinate differenti. Queste tecniche di scansione tengono presente la possibilità di osservare uno o più oggetti a patto che siano isolati, non con un background, se si deve lavorare con più oggetti allora vanno definiti gli anchor box.

Accelerating object detection

Negli ultimi anni sono state introdotte diverse tecniche per evitare di dover fare sliding diversi o IoU e in generale velocizzare il processo di localizzazione dell'immagine. L'idea in generale è quella di evitare di utilizzare sliding windows (per migliorare la performance) e studiare invece i metodi che mi diano una **proposta di regione da osservare**. Ci sono diverse tecniche che compiono questa operazione, molte di queste tecniche non sono di machine learning ma utilizzano diverse proprietà delle immagini, come ad esempio l'intensità di un pixels.

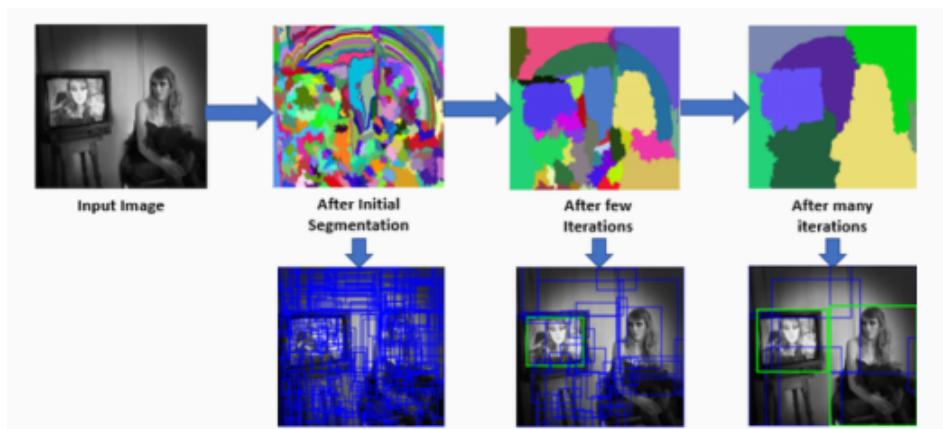


Figure 7.4: Esempio di applicazione di un algoritmo di selective search ad una immagine

Selective search L'idea è quella di partire da una ricerca stocastica che identifica zone che hanno ad esempio la stessa tonalità o la stessa intensità che facilitino l'identificazione dell'oggetto. Dopo di che si fanno alcune iterazioni, si definiscono delle soglie per poter raggruppare determinate regioni (generalmente si utilizzano algoritmi greedy per combinare le regioni) e si ottengono poi delle predizioni per i bounding box. Queste sono tecniche un po' complicate, non sono classificatori quindi non mi darà immediatamente gli oggetti ma mi restituisce sicuramente dei candidati interessanti

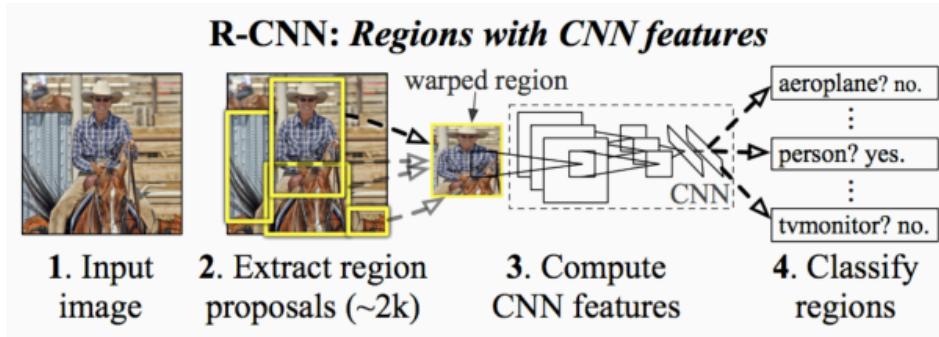


Figure 7.5: Workflow di una architettura di tipo R-CNN

Queste tecniche vengono utilizzate spesso nelle R-CNNs ovvero *Regions with CNN features*, in questi modelli quello che si fa è partire da un algoritmo che mi suggerisce delle aree dove guardare, dopo di che si taglano le regioni di interesse e a queste si applica il classificatore. Questo è stato il primo modello, pubblicato nel 2013, in cui si suggerisce di togliere una sliding window. Il problema di questo modello è che rimane comunque lenta, pur avendo eliminato la sliding window e l'identificazione dei pixels. In sostanza diminuisco il numero di regioni che devo classificare ma impiego molto tempo a proporre una regione. Il funzionamento dell'algoritmo è rappresentato in figura 7.6

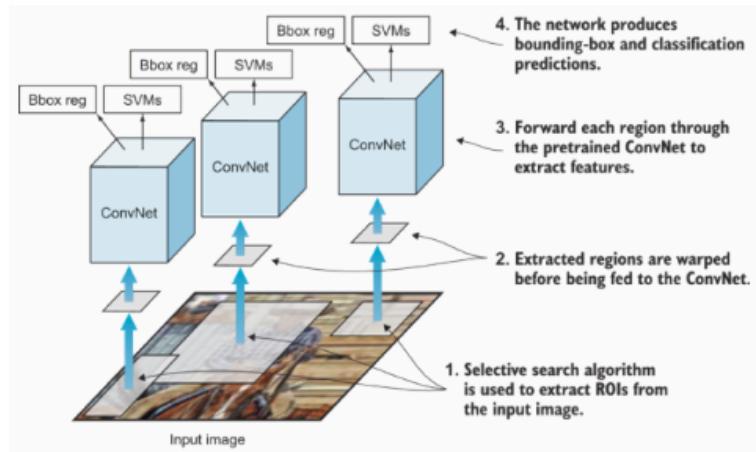


Figure 7.6: Rappresentazione del funzionamento dell'algoritmo di RCNN

Nel 2015 gli stessi autori hanno fatto un update del modello e hanno proposto il Fast R-CNN, in questo caso non si usa più un selective search ma si sfrutta il transfer learning. Per cui prendo una CNN che dovrebbe estrarre dall'immagine

le regioni (le coordinate delle regioni), la CNN osserva l'immagine, mi restituisce le coordinate delle regioni di interesse e su queste faccio la classificazione 7.7. In sostanza quello che faccio è invertire il processo per cui, il grosso del calcolo viene fatto una volta sola.

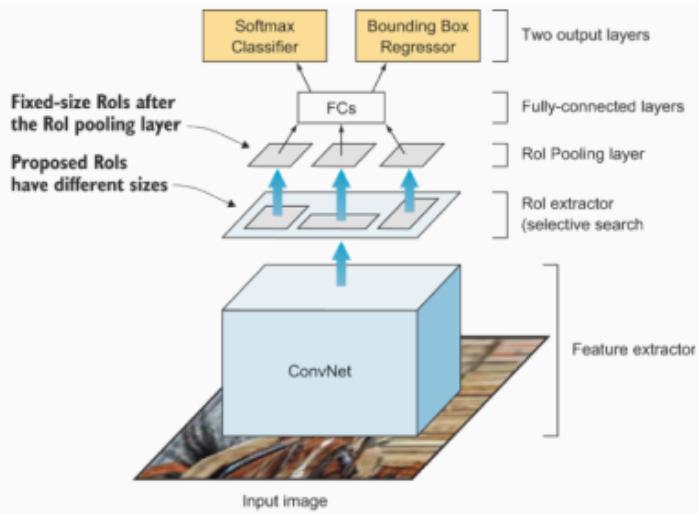


Figure 7.7: Workflow dell'applicazione di un algoritmo di transfer learning per la localizzazione di oggetti

L'unico problema che resta è che ho comunque un modello molto pesante per la proposta della regione. Sempre in questo contesto è stata proposta la **Faster CNN**, per cui si utilizza una **Region Proposal Network** che in pratica applica una maschera alla CNN e quindi mi dà già le regioni in cui ci sono degli oggetti. L'altra differenza è che la Region Proposal Network è che mi dà sì i bounding box degli oggetti proposti, ma allo stesso tempo mi dà anche la maschera con i feature-maps generati dal classificatore (immagine 7.8).

7.2.2 YOLO

Le reti basate sulle tecniche YOLO, **You Only Look Once**, in sostanza creano uno splitting della griglia e ad ogni bounding box associano una probabilità e un offset associato all'oggetto che cerco. Alla fine creo una sorta di mappa, delle classi di probabilità, delle regioni da cui poi ricavo le mappe finali. Effettuo in sostanza una sorta di segmentazione, come si può vedere nella figura 7.9

7.3 Identificazione

Il problema dell'identificazione è legato alla **segmentazione semantica**. Il problema è simile a quelli visti finora, in sostanza ho la mia immagine e voglio poter dire ciascun pixel a quale figura appartiene. Un modo brute-force per portare a termine questo compito è quello di utilizzare diverse reti convoluzionali e poi un softmax massiccio (7.10), con tutti i pixels per poi ottenere i risultati. Questo si può fare ma il problema è che si finisce molto velocemente in overlearning perché l'oggetto su cui applico il softmax è delle dimensioni dell'immagine.

La prima cosa che è stata fatta per cercare di evitare di lavorare con modelli così grandi è una sorta di auto-encode per cui l'immagine è passata a una rete neurale

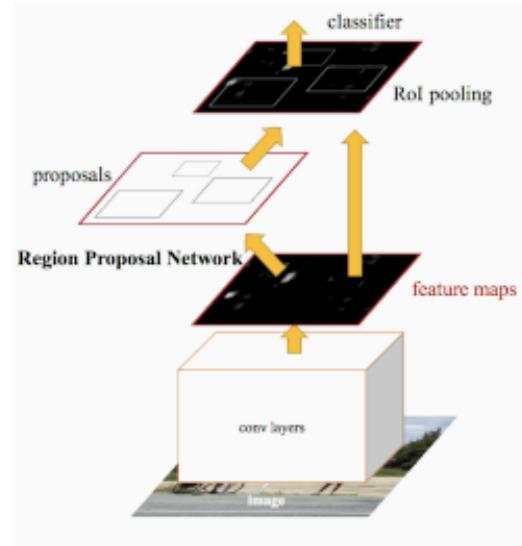


Figure 7.8: Workflow dell'applicazione di un algoritmo che utilizza Region Proposal Networks

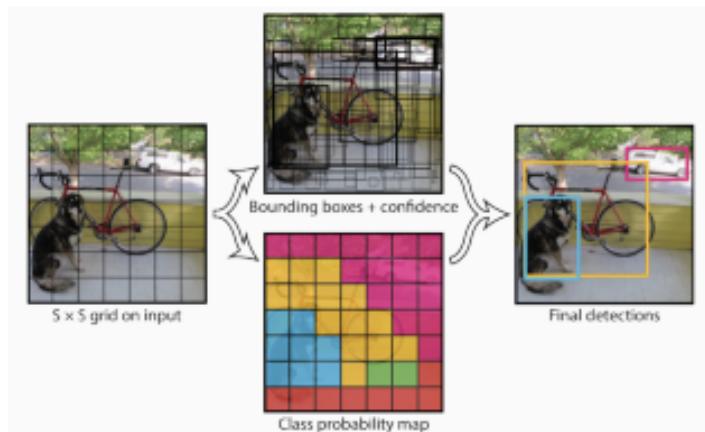


Figure 7.9: Esempio di funzionamento dell'algoritmo *You Only Look Once*

che comprime l'informazione (questo passaggio è noto come **down-sampling**), fatto questo si fa una operazione di **up-sampling** che riporta l'immagine dallo spazio latente più piccolo allo spazio originale. Il problema di queste architetture è che nonostante siano più veloci queste architetture rimangono comunque molto pesanti (vedi 7.11).

7.3.1 Mask R-CNN

In questa architettura si prende il modello di faster CNN che si occupa dell'estrazione e della classificazione delle regioni, dopo di che si prende un secondo modello di CNN che, all'interno delle regioni di interesse identificate dal modello precedente, determina le diverse categorie di pixels. In questo modo creo una sorta di maschera per ciascuna regione che è stata selezionata in precedenza (vedi figura 7.12).

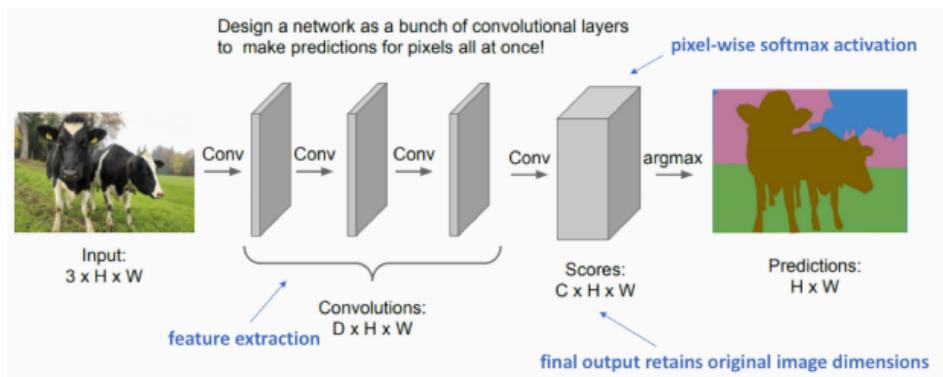


Figure 7.10: Workflow dell'applicazione di un algoritmo di transfer learning

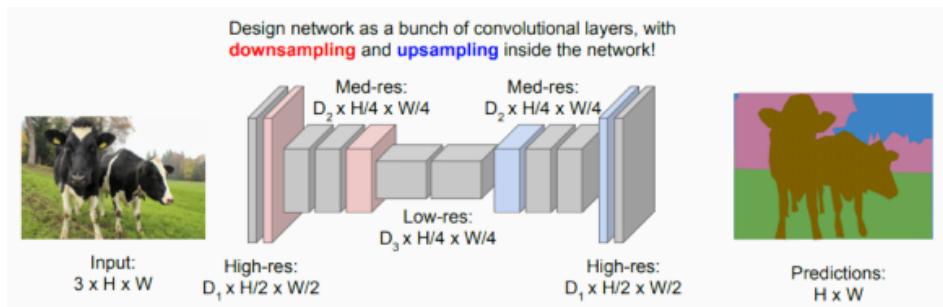


Figure 7.11: Workflow dell'applicazione di un algoritmo di transfer learning

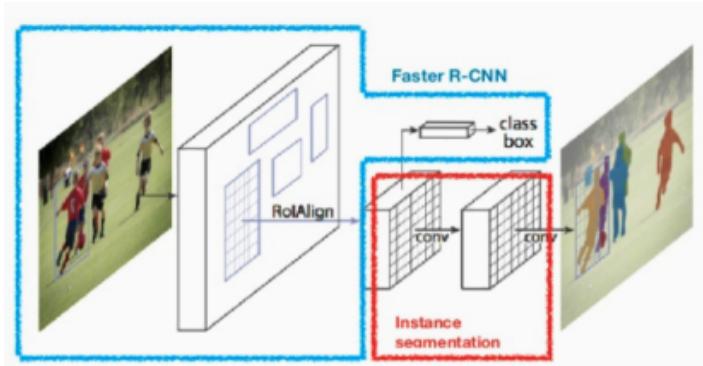


Figure 7.12: Workflow dell'applicazione di un algoritmo di transfer learning

Chapter 8

Lezione 8

8.1 Unsupervised Learning

In precedenza sono già stati nominati i modelli di unsupervised learning, ora cerchiamo di studiarli e trovare le principali differenze con i modelli di supervised learning. In un modello di supervised learning il procedimento è sempre identico: ho un dataset contenente il dato ed un tipo di etichetta associata al dato. Dopo di che si entra nel main loop dell'algoritmo che fa la parte di apprendimento supervisionato, la loss-function confronta le previsioni del modello con le informazioni del dataset. Implementato l'algoritmo si arriva quindi ad un output e si conclude il ciclo.

Nel caso di unsupervised learning, già il dataset di input cambia perché contiene unicamente il dato, senza informazioni aggiuntive o "etichette" di alcun tipo. Con questa tipologia di tecniche il problema che ci si trova spesso ad affrontare è quello di fare "stime" (eg. cercare di fare una approssimazione o stima di una densità di probabilità). In questo caso, il compito dell'allenamento è quello di determinare delle proprietà e caratteristiche in più del modello, caratteristiche che non sono in grado di ricavare analiticamente.

8.2 Generative model

Un modello generativo è un modello di rete neurale in cui, dato un training-set voglio capire qual'è la funzione di probabilità che rappresenta quell'insieme. In questo contesto si possono distinguere due tipologie di probabilità:

1. $p_{data}(x)$: è la probabilità empirica o teorica che caratterizza il dataset
2. $P_{model}(x)$: la distribuzione di probabilità che caratterizza il modello che ho costruito con la mia rete neurale

Quello che può succedere ad esempio è che avere un dataset e sapere che i dati sono stati costruiti a partire da una probabilità nota $P_{data}(x)$, questa probabilità controlla in sostanza il comportamento dell'immagine. In generale purtroppo al $P_{data}(x)$ è quasi impossibile da calcolare, quello che posso fare è trovare degli strumenti per approssimarla quindi ho una funzione multidimensionale che genera le immagini secondo una certa distribuzione di probabilità. Il problema è che questa $p_{data}(x)$ è difficile, a volte impossibile da ricavare. Quello che posso fare quindi è trovare una $p_{model}(x)$ che possa approssimare la $p_{data}(x)$. Il compito è

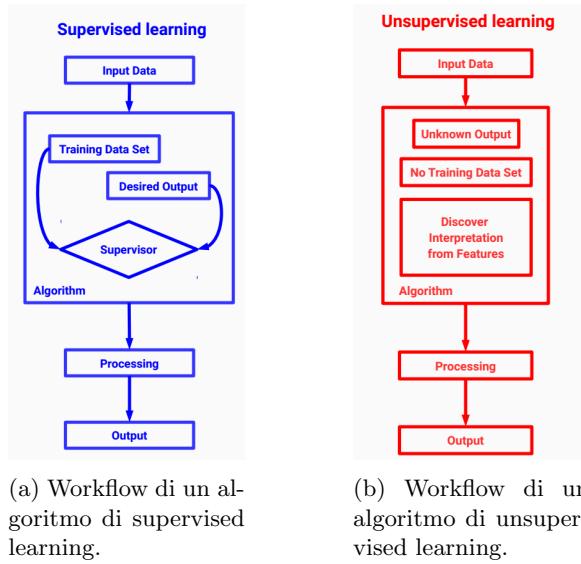


Figure 8.1: Confronto degli algoritmi di supervised e unsupervised learning

quello di trovare una tecnica che mi consenta di modellizzare analiticamente il $p_{data}(x)$ prior che mi consenta di stimare il comportamento dei dati originali.

Quindi l'obiettivo è quello di estrarre dal dataset una density estimation, cercare di stabilire una tecnica per imparare il $p_{model}(x)$ a partire dal $p_{data}(x)$. Questo problema ha suscitato moltissimo interesse negli ultimi anni, quindi esistono moltissimi modelli diversi che cercano di raggiungere questo obiettivo.

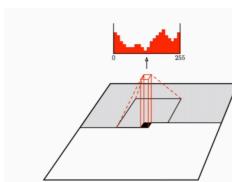
Una delle applicazioni più interessanti di queste tecniche è la "data augmentation", questo è particolarmente utile quando si ha a che fare con dataset piccoli, in questo modo una volta trovata una modellizzazione per la probabilità $p_{data}(x)$, si può campionare la $p_{model}(x)$ per avere più dati su cui fare il training.

8.2.1 Autoencoders

Trattamento probabilistico delle immagini

Gli autoencoders sono i primi modelli che sono stati elaborati. Come prima cosa bisogna definire alcuni concetti del trattamento probabilistico delle immagini.

Dato un campione di immagini bidimensionali, con valori diversi di pixels, costruire la probabilità per questo dataset è complicato.



PixelCNN, van der Oord et al. 2016

Figure 8.2: Esempi di utilizzo dei diversi algoritmi di pattern recognition

Dall'insieme di immagini posso ricavare un istogramma con l'indice del pixel sulle x , indice del pixel che va da 0 a 255, e sul valore di ogni bin il valore di

ogni pixel di ogni immagine; in pratica con questa operazione mappo ogni immagine in una distribuzione di probabilità (chiaramente dopo aver normalizzato gli istogrammi), questo processo è parzialmente rappresentato nella figura 8.2.

Ad ogni dataset si può associare un istogramma probabilistico. Ci sono diversi modi poi per fittare questa distribuzione con dei modelli.

La prima famiglia di modelli è quella della multigaussiana: sono modelli facili da trattare, sono detti trattabili perché definisco come densità della underlying distribution, il prodotto delle diverse dimensioni (diversi elementi) che costituiscono il dataset. Quindi diciamo che faccio il prodotto di tutti i pixels tenendo in conto anche gli altri pixels. Cerco di costruirmi una sorta di composizione bayesiana della mia distribuzione (composizione che naturalmente propagherà al suo interno anche le varie correlazioni presenti all'interno delle immagini).

Questo modello è trattabile perché la $p(x)$ è complicata ma è comunque una forma funzionale che posso descrivere senza problemi. La task a questo punto diventa quella di massimizzare la likelihood del training dataset, in pratica ricevo i vari istogrammi e voglio trovare la likelihood massima una volta definito un modello probabilistico.

Questo tipo di approccio normalmente fallisce, o in generale non è efficiente, soprattutto quando si ha a che fare con molte dimensioni.

Per affrontare il problema in maniera diversa, come prima cosa si elimina la discretizzazione, si utilizza una probabilità $p(x)$ che sia un po' più flessibile:

$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz, \quad (8.1)$$

dove θ rappresenta i parametri del modello, z lo spazio delle variabili latenti. In questo modo ho una densità di probabilità che sicuramente è parametrica (devo poter tunare la forma funzionale), quello che succede è che **sposto la discretizzazione nello spazio delle variabili latenti** z , una sorta di spazio in cui identifico e trattiengo solo le informazioni che non sono ridondanti. Il problema è che purtroppo questa equazione non è trattabile, perché la $p_{\theta}(x|z)dz$ può essere una qualunque funzione. Quindi in ogni caso non ho risolto il problema, l'ho solo spostato all'interno dell'integrale.

Tutti i metodi che sono stati definiti a partire da questo integrale, definiscono una densità di probabilità che **non può essere ottimizzata**, però ci sono delle tecniche che mi consentono, usando la distribuzione, di ottenere risultati approssimati.

D'ora in avanti si lavorerà in spazi in continui in cui però l'oggetto resta sempre intrattabile.

Auto-Encoders

Date queste definizioni, il modello dell'AE (*auto-encoder*) è molto semplice. Supponiamo di avere un dataset di dati non annotati, costruiamo un modello che sia in grado di estrarre i diversi valori dei diversi pixels e che poi, in qualche modo, **comprime l'informazione in uno spazio latente**, spazio in cui vive la variabile z , di dimensione minore della x . L'idea è appunto quella di fare una compressione, fare un encoder, si noti che **il modello che compie questa operazione di compressione può essere di qualunque tipo**: può essere lineare, non lineare, una NN, una DNN, una CNN. Il punto fondamentale è che, se riesco a fare questa operazione in cui **estraggo solo le feature importanti**, ho una macchina che è

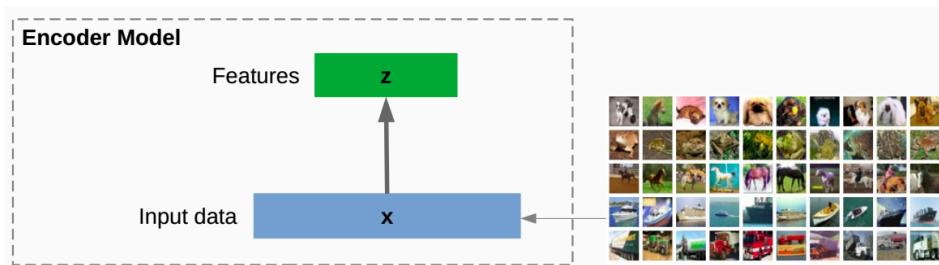


Figure 8.3: Esempi di utilizzo dei diversi algoritmi di pattern recognition

in grado di eliminare tutta l'informazione ridondante o inutile, eliminare tutto il rumore dal dataset.

Io posso costruire il dataset di input, definisco un modello di encoding che comprime l'informazione in uno spazio latente. Dopo di che, costruisco un secondo modello, chiamato **decoder** che farà l'operazione inversa e mi confronta l'input con le immagini ricostruite dal decoder (vedi figura 8.4).

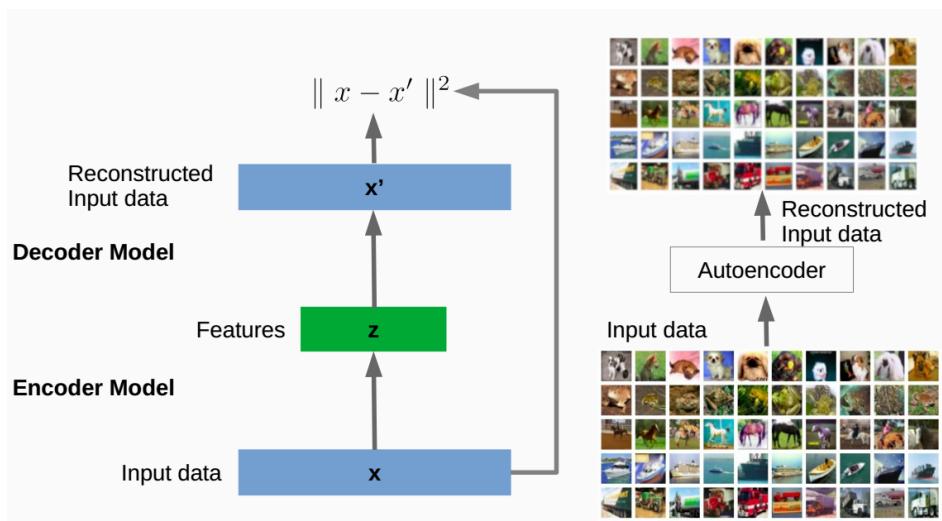


Figure 8.4: Esempio di autoencoder

Un esempio di modello di autoencoder combinato con una rete neurale è rappresentato in 8.5. Per allenare questo tipo di architettura subito dopo la compressione in spazio latente attacco una D-CNN(De-Convolutional Neural Network)

Un tipo di applicazione è quella del filtraggio per cui si prendono le immagini, le si fanno passare all'interno dell'encoder model, in questo modo le immagini vengono proiettate nello spazio latente; a questo punto si utilizza il vettore z dello spazio latente come feature vector su cui poi viene fatta la classificazione 8.6. La prima applicazione del modello di autoencoder infatti, è stata quella di un classificatore, questo procedimento è quello che poi ha ispirato la costruzione delle NN e la loro applicazione oggi.

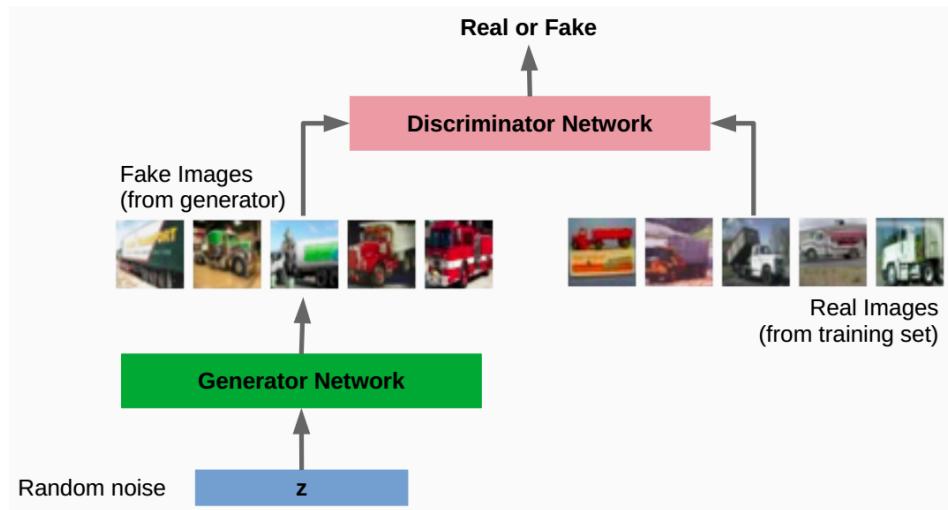


Figure 8.5: Esempio di autoencoder

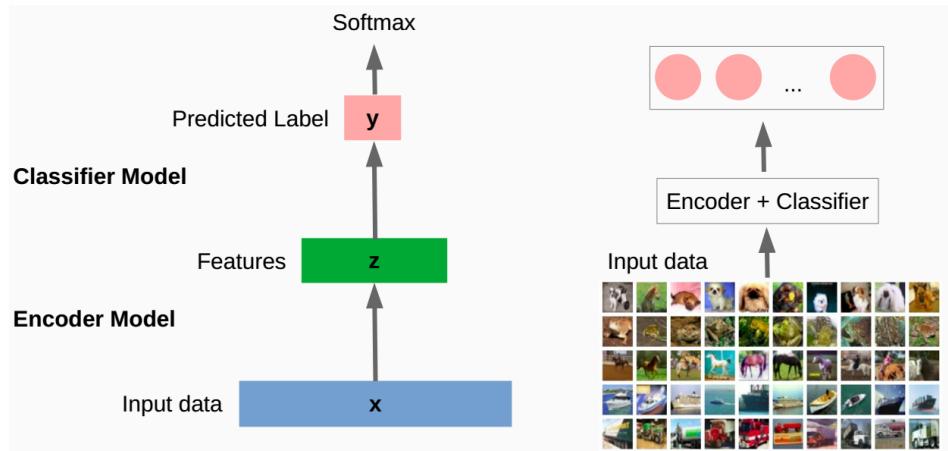


Figure 8.6: Esempio di applicazione di un auto-encoder per realizzare un classificatore

Variational Auto-Encoders

Il modello di encoder che abbiamo visto finora però, non ha realmente imparato la distribuzione sottostante, mi consente di ridurre di molto la dimensionalità del problema, ma non risolve il problema dell'imparare l'underlying density, non posso utilizzarlo per generare campioni verosimili dall'immagine di partenza.

L'auto-encoder mi consente di fare solo compressione e de-compressione. Lo spazio latente è interessante perché mi permette di togliere ridondanza e semplificare il problema, infatti posso controllare i dati di input tramite poche dimensioni. Per utilizzare un modello simile allo scopo di generare un nuovo dataset devo inserire un elemento di stocasticità.

Consideriamo nuovamente il modello del decoder 8.7

Consideriamo un dataset x e cerchiamo di mi concentro sul decoder, questo perchè è la parte che devo verificare per ottenere immagini nuove, immagini modificate. Quello che devo fare è ottenere $p_{\theta^*}(z)$ che farà il sampling delle variabili

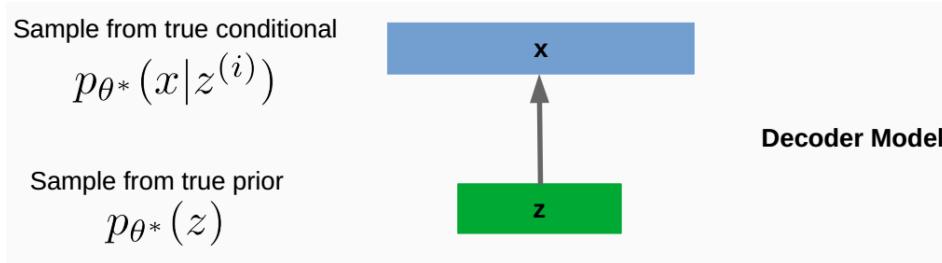


Figure 8.7: Workflow del decoder per la generazione di dati

nello spazio latente. Facendo il sampling del prior, posso fare una il calcolo una nuova probabilità $p_{\theta^*}(x|z^{(i)})$.

Quindi modellizzo una funzione che si occupi dell'estrazione della z e per ogni estrazione calcolo la probabilità condizionata tenendo conto della posizione del pixel. Il problema in questo caso è che sto lavorando con una funzione parametrica quindi devo scegliere che forma dare alla p_{θ^*} . L'obiettivo quindi è **stimare i parametri migliori θ^* per questo modello generativo**, infatti capire i parametri significa avere chiaro com'è fatto lo spazio latente.

Naturalmente posso scegliere di usare una p_{θ^*} qualunque, quindi posso usare una formula chiusa ma posso usare anche una rete neurale o un modello non lineare. Dal punto di vista pratico è risultato molto efficiente l'utilizzo di due funzioni: una distribuzione normale (o in generale parametrico fisso) per il prior, ovvero per la $p(z)$ e una rete neurale per il campionamento condizionato di $p(x|y)$.

Partendo sempre dalla 8.1, che è intrattabile, quello che voglio fare è massimizzare la log-likelihood in modo da avvicinare il modello parametrico ai dati. In questo caso ho bisogno comunque calcolare $p_{\theta}(x|z)$, dato che questo però è impossibile, posso usare il teorema di Bayes per campionare la probabilità inversa (*posterior*)

$$p_{\theta}(z|x) = \frac{p_{\theta}(x|z)p_{\theta}(z)}{p_{\theta}(x)}. \quad (8.2)$$

D'altra parte $p_{\theta}(x)$ è ancora intrattabile, in questo caso però posso introdurre un modello non lineare (encoder) che sia in grado di mappare la $p_{\theta}(z|x)$; questa probabilità è mappabile perché dipende dall'input.

In sostanza quindi ho solo due parti che possono essere adeguatamente modellizzate: $p_{\theta}(z|x)$ e la $p_{\theta}(z)$.

Probabilistic generation of data Supponendo di partire dall'encoder, anziché fare un semplice encoding introduco una rete neurale $q_{\phi}(z|x)$ che dipende dai parametri ϕ e che mi genera una z in modo condizionale rispetto al dato input x . Quello che chiedo alla rete neurale non è di compattarmi i dati ma di restituirmi due numeri: una media $\mu_{z|x}$ e una matrice di covarianza $\Sigma_{z|x}$; quindi voglio una media e una matrice di covarianza per z osservata x .

Dopo di che, scelgo una forma funzionale per la $p(z)$, quello che si fa generalmente è utilizzare una distribuzione normale che prende come input quello generato dall'encoder ovvero $\mathcal{N}(\mu_{z|x}, \Sigma_{z|x})$, questa funzione sarà responsabile per il campionamento della $p(z)$.

In sostanza quindi la rete neurale è responsabile del generare una media e una covarianza che siano plausibili per i miei dati di input, fatto questo però quello

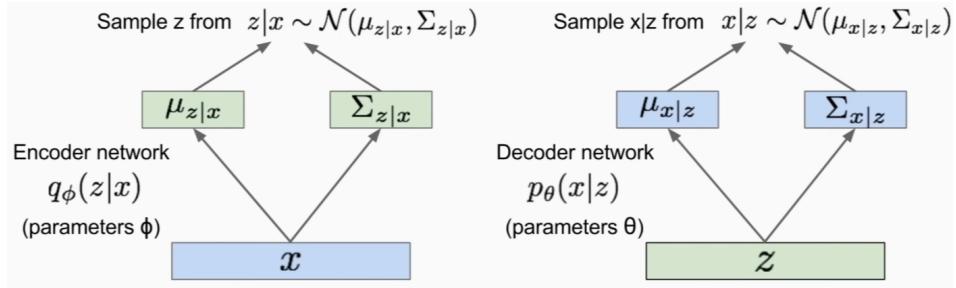


Figure 8.8: Workflow del processo di generazione di dati tramite encoder con rete neurale e decoder con rete neurale

che faccio è dire "mi discosto di poco da questa predizione" voglio qualcosa che sia distribuito più o meno in maniera simile, come dimostra il fatto che io stia campionando una distribuzione Gaussiana.

Nel decoder, quello che si farà è prendere le z ricavate allo step precedente (tramite l'encoder), questa saranno gli input per una rete neurale che imparerà la $p_\theta(x|z)$ e mi estrarrà una media $\mu_{x|z}$ e una matrice di covarianza $\Sigma_{x|z}$ in modo tale da poter costruire una nuova distribuzione normale $\mathcal{N}(\mu_{x|z}, \Sigma_{x|z})$ e generare quindi tutte le coppie $x|z$ per costruire le immagini.

A questo punto, se questo meccanismo funziona bene devo costruire l'architettura finale, ovvero l'autoencoder variazionale completo. Prima di questo ultimo passaggio, consideriamo nuovamente la likelihood nominata prima. Quello che voglio fare è massimizzare il valore di aspettazione dell'integrale e ottengo quanto riportato in 8.9

$$\begin{aligned}
 \log p_\theta(x^{(i)}) &= \mathbb{E}_{z \sim q_\phi(z|x^{(i)})} [\log p_\theta(x^{(i)})] \\
 &= \mathbb{E}_z \left[\log \frac{p_\theta(x^{(i)}|z)p_\theta(z)}{p_\theta(z|x^{(i)})} \right] \\
 &= \mathbb{E}_z \left[\log p_\theta(x^{(i)}|z) \right] - \mathbb{E}_z \left[\log \frac{q_\phi(z|x^{(i)})}{p_\theta(z)} \right] + \mathbb{E}_z \left[\log \frac{q_\phi(z|x^{(i)})}{p_\theta(z|x^{(i)})} \right] \\
 &= \underbrace{\mathbb{E}_z \left[\log p_\theta(x^{(i)}|z) \right]}_{J(x^{(i)}, \theta, \phi)} - D_{KL} \left(q_\phi(z|x^{(i)}) || p_\theta(z) \right) + D_{KL} \left(q_\phi(z|x^{(i)}) || p_\theta(z|x^{(i)}) \right)
 \end{aligned}$$

Figure 8.9: Espressione e valutazione della log-likelihood

Sempre nella riga finale della 8.9 si evidenziano due proprietà importanti dell'espressione: nel primo termine ho la differenza tra il logaritmo di p_θ e Kullback–Leibler divergenza 10.6 tra il modello di encoding e il modello dello spazio latente (queste due quantità sono quantità calcolabili), è una quantità positiva e la parte di destra è trattabile e calcolabile perché ho ancora una rete neurale e la funzione di encoding. Il problema è la probabilità più a destra dell'equazione che rimane comunque non trattabile.

In ogni caso però non ci serve avere un valore numerico o una soluzione analitica, la cosa importante è che la quantità è positiva e l'unica cosa che mi interessa è capire se la J possa essere effettivamente massimizzata (magari imparando qualche dipendenza parametrica durante l'allenamento). Quindi se ho una sorta di **variational lower bound** (ELBO), ed effettivamente questo ELBO lo abbiamo, allora

sappiamo che

$$\log_{p_\theta}(x^{(i)}) \geq J(x^{(i)}, \theta, \phi). \quad (8.3)$$

Quindi posso ricavare θ e ϕ cercando durante il training di massimizzare J :

$$\theta^*, \phi^* = \arg \max_{\theta, \phi} \sum_{i=1}^N J(x^{(i)}, \theta, \phi). \quad (8.4)$$

Quindi la struttura completa del variational autoencoders sarà quella rappresentata in 8.10

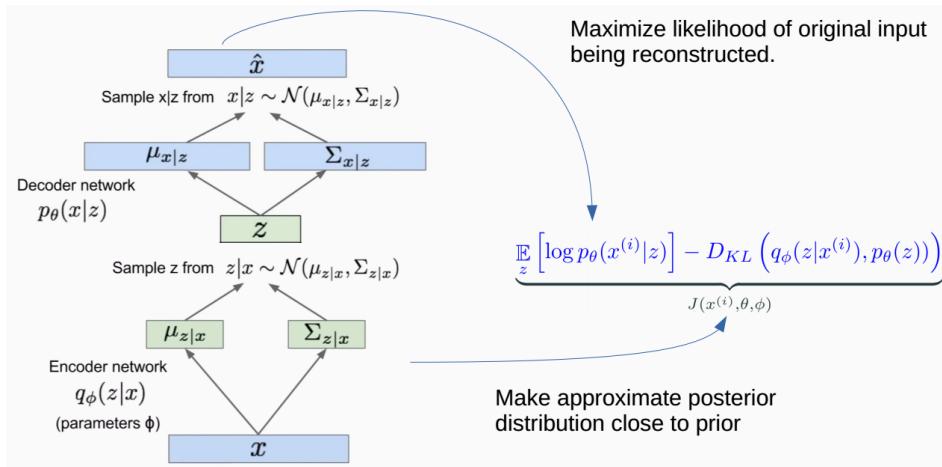


Figure 8.10: Archittettura completa del variational autoencoders

Il problema di questo modello è che **utilizzando molte gaussiane perdo la nitidezza dell'immagine**, quindi le immagini hanno un po' un effetto di blur. Questo è il problema fondamentale delle VAE, sono in grado di generare immagini verosimili ma non nitide, sono quindi modelli semplici, in alcuni casi utili, ma assolutamente non completi.

Per questo motivo attualmente si è passati ad un paradigma generativo diverso.

8.2.2 Generative Adversarial Networks

Visti i problemi delle VAE, all'incirca sette anni fa un gruppo di ricercatori ha deciso di provare ad utilizzare una nuova strategia, il cambiamento principale è stato quello di mettere da parte tutto ciò che riguarda distribuzioni di probabilità da campionare e spazio latente.

Struttura e funzionamento delle GAN

Anche in questo caso si parte con un dataset, l'obiettivo è quello di modellizzare. Avendo già utilizzato un encoder, quello che possiamo fare è rimuovere la parte di sampling, che sostanzialmente è quella che dà problemi; quindi quello che facciamo è passare un rumore (rumore qualunque: gaussiano, uniforme ecc.) e lo passo come input alla rete neurale che devo poi restituirmi un'immagine verosimile. Il workflow di questo tipo di algoritmo generatore è rappresentato in figura 8.11

L'implementazione di questo modello, nella sua versione originale, è nota come *two-player game*. Si hanno due reti neurali: un *generatore* che prende come argomento un random noise e che genera un'immagine verosimile ma finta. L'idea

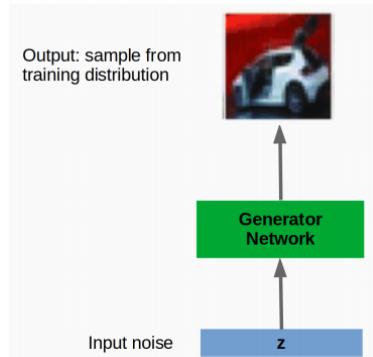


Figure 8.11: Workflow di una rete generatrice

principale è quella di utilizzare quest'immagine come una sorta di estensione del dataset iniziale, a questo punto posso chiedere ad una seconda rete neurale di classificare le immagini (vedi immagine 8.12) in due categorie "reale" o "finta".

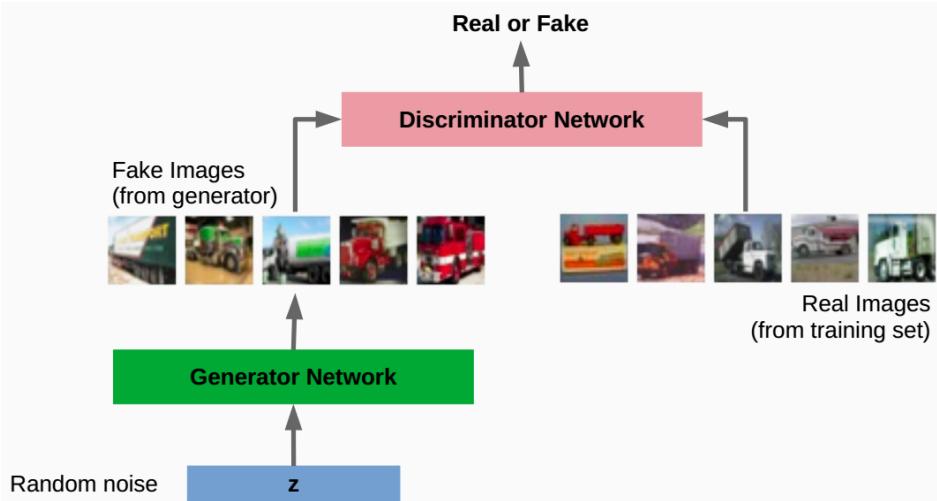


Figure 8.12: Workflow di una GAN

Se sono in grado di bilanciare adeguatamente l'allenamento delle due reti, allora arriverò ad un risultato soddisfacente se il discriminatore sarà "ingannato" dal generatore

Loss function

Con questo paradigma, dal punto di vista della loss function cambia tutto, quello che si fa è cercare di calcolare una maximum likelihood per il generatore e per il discriminatore, in sostanza le due reti vengono allenate contemporaneamente utilizzando una funzione minmax 8.5

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}[\log D_{\theta_d}(x)] + \mathbb{E}[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))] \right], \quad (8.5)$$

dove, i parametri del generatore θ_g in modo che $D_{\theta_d}(G_{\theta_g}(z)) \rightarrow 1$ e quindi il discriminatore pensa che $G_{\theta_g}(z)$ sia un'immagine reale. Il discriminatore invece,

ottimizza θ_d in modo tale da fare la seguente classificazione: se $D_{\theta_d}(x) \rightarrow 1$ l'immagine è reale, se $D_{\theta_d}(G_{\theta_g}(z)) \rightarrow 0$ allora l'immagine è fake. Quindi da una parte ho il discriminatore che vuole massimizzare la somma dei due termini e dall'altra il generatore che cerca di minimizzarla. Il problema è che non posso fare massimizzazione e minimizzazione insieme, quello che si può fare è procedere per step durante i quali una volta tengo fissi i parametri del generatore la volta seguente tengo fissi i parametri del discriminatore.

Quindi posso identificare i seguenti due step:

1. applicare un algoritmo di gradient ascent al discriminatore per massimizzare

$$\max_{\theta_d} \left[\mathbb{E}[\log D_{\theta_d}(x)] + \mathbb{E}[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))] \right] \quad (8.6)$$

2. applicare un algoritmo di gradient descent al generatore per minimizzare

$$\min_{\theta_g} \left[\mathbb{E}[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))] \right], \quad (8.7)$$

generalmente questo ultimo passaggio viene sostituito da una massimizzazione tramite gradient ascent di

$$\max_{\theta_g} \left[\mathbb{E}[\log(D_{\theta_d}(G_{\theta_g}(z)))] \right]. \quad (8.8)$$

Per un modello di questo tipo, quello che segnala un apprendimento "buono" da parte della rete neurale è la **convergenza delle loss function**, se la loss function del generatore o del discriminatore continuano ad oscillare significa che non ho ancora allenato la rete a sufficienza. È importante osservare che **il tempo di convergenza per una GAN è di molto superiore** al tempo di convergenza tipico di una rete standard, con una GAN tempi di convergenza tipici sono dell'ordine delle centinaia di migliaia di epoche.

Chapter 9

Lezione 9

In questa lezione ci occupiamo del terzo paradigma del machine learning, ovvero il reinforcement learning, la tecnica più recente tra le diverse tecniche di machine learning.

9.1 Reinforcement learning

Quando si parla di reinforcement learning la situazione è più complessa, si definiscono diverse componenti che interagiscono fra di loro e devono in modo continuativo e sistematico un modello matematico che prenda decisioni sostanzialmente in tempo reale. In questo contesto sono definiti il modello, l'ambiente di gioco, e le operazioni matematiche che consentono di costruire una architettura di calcolo.



Figure 9.1: Workflow di un algoritmo di reinforcement learning

Il reinforcement learning è un modello di apprendimento in cui si hanno due elementi fondamentali:

1. **agente:** l'agente è il modello matematico che dovrà prendere le decisioni, compirà l'azione a_t al tempo t che modifica l'ambiente
2. **ambiente** (o environment): è il contesto in cui si muove l'agente, una volta modificato dall'agente l'ambiente restituirà la modifica sotto forma di un nuovo stato s_t

Inoltre l'azione dell'agente sull'environment viene valutata tramite un **reward** r_t , il reward non è come una loss function ma è più come lo score di un videogame (non è detto che mi serva avere uno score altissimo, posso comunque raggiungere il mio obiettivo, il problema è che magari non raggiungerò l'obiettivo nel migliore dei modi).

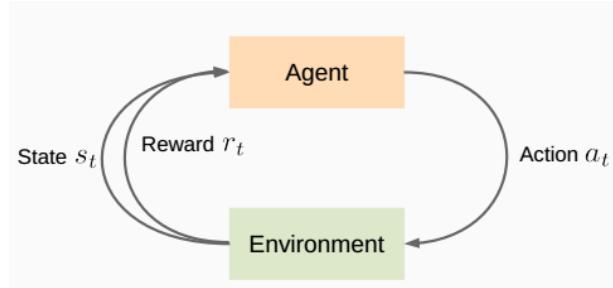


Figure 9.2: Esempio di ciclo iterativo per reinforcement learning

9.2 Markov Decision Process

Il ragionamento di episodi e di necessità di tenere memoria del passato, non sbagliare e dover portare avanti l'azione in maniera sensata mi porta a definire un **processo di decisione correlato**. Per definire quanto detto in maniera matematica si utilizzando i Markov Decision Process.

Per prima cosa so definisce una **Markov property**, ovvero caratteriziamo il problema specifico tramite un vettore *world* 9.1:

$$\text{World} = (S, A, R, P, \gamma), \quad (9.1)$$

dove S rappresenta i possibili stati, A un insieme di azioni possibili, R la distribuzione del reward che viene assegnato, P la probabilità di transizione che mi permette di capire com'è il sistema evolverà da uno stato iniziale a quello successivo, γ è il discount factor (minore di uno) che in sostanza mi elimina casi patologici, in sostanza mi permette di dare un peso piccolo a determinate azioni perché avvenute in condizioni impreviste.

9.2.1 MDP

Al tempo t_0 il sistema è inizializzato ad uno stato s_0 , dopo di che l'algoritmo procede tramite una serie di step:

1. si seleziona l'azione (aspetto importante è capire come proporre l'azione)
2. a questo punto, l'ambiente campiona il reward $r_t \sim P(\cdot | s_t, a_t)$
3. sempre l'ambiente campiona lo stato successivo $s_{t+1} \sim P(\cdot | s_t, a_t)$
4. il reward r_t e lo stato successivo s_{t+1} vengono trasmessi all'agente

La difficoltà principale è quella di determinare una policy π che stabilisca quale sia l'azione più efficiente per massimizzare il reward. L'obiettivo è quello di trovare una policy, una routine π^* che massimizzi il fattore

$$\sum_{t \geq 0} \gamma^t r_t \quad (9.2)$$

Non si tratta di trovare un unico valore finale ma una somma, quindi tutta la storia di quello che ha fatto l'agente viene immagazzinata all'interno di questa espressione, in questo caso utilizzando questa espressione per assegnare uno score alla policy quello che succede è che effettivamente l'agente impara dalla catena di eventi, quindi dopo diversi allenamenti certe mosse verranno evitate perché si sa che nel lungo termine porteranno ad un reward molto basso.

Optimal policy

L'obiettivo è quello di trovare una policy π^* che massimizzi la somma dei reward e che sia anche in grado di gestire la casualità del problema individuando un pattern corretto. Formalmente questo può essere espresso come segue:

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi \right], \quad (9.3)$$

con $s_0 \sim p(s_0)$, $a_t \sim \pi(\cdot | s_t)$, $s_{t+1} \sim p(\cdot | s_t, a_t)$. Quindi, voglio una policy π^* che massimizzi il massimo del valore di aspettazione della somma definita in precedenza. Per ogni episodio, per ogni iterazione posso cercare di massimizzare la somma ma poi devo cercare di identificare la policy che mi porta ad un set di azioni ideali.

Quello che si fa normalmente è convertire questa richiesta, che normalmente è un po' difficile da gestire (può essere una *if* condition, un decision tree, una condizione binaria, una funzione matematica), quindi per cercare di il problema in un modo più continuo si utilizza una funzione valore. Suppongo di avere una serie di stati, azioni e reward e definire una funzione valore che mi permetta di determinare la policy in funzione dello stato osservato, quindi non cerco più una policy tra tutte quelle che ho a disposizione ma mi concentro su quelle proposte.

Si definisce quindi la funzione valore per uno stato s che soddisfa la policy π come

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right] \quad (9.4)$$

, questa funzione $V^\pi(s)$ misura la bontà dello stato s . Lavorando solo su questa funzione però non ho controllo sull'azione perché questa resta bloccata nel parametro π , l'azione non è una variabile di questa funzione.

In sostanza passo la value function alla Q-value function dove l'argomento è la coppia stato-azione quindi è questa coppia che viene valutata.

$$Q^\pi(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]. \quad (9.5)$$

In questo modo ho una metrica che mi consente di valutare quanto buona è la coppia azione-stato.

Bellman equation A questo punto in teoria potrei decidere di calcolare l'intera matrice di Q-value per poi trovare la policy più efficace. Questo approccio, naturalmente inefficiente è quello che si utilizzava prima dell'introduzione delle tecniche di deep learning, questo approccio però mi permette anche una scarsissima flessibilità.

Per ovviare a questo problema quindi, è stata fatta una leggera modifica alle equazioni viste in precedenza cerchiamo di identificare un valore Q^* che ottimizzi

il Q-value semplicemente massimizzando la policy in cui il valore di aspettazione della somma è alta.

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]. \quad (9.6)$$

La cosa interessante è che esiste un'equazione, dimostrata da Bellman che specifica questa dipendenza: esiste la possibilità di scrivere una Q-value function ottimizzata che sia data dal valore di aspettazione della formula 9.7

$$Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]. \quad (9.7)$$

Quello che vogliamo è una Q function ottimizzata che prenda come argomenti la coppia stato azione, soprattutto voglio usare il membro di destra che rappresenta l'azione futura per creare una correlazione tra i diversi step temporali, quindi cerco di usare questo termine per massimizzare il reward possibile per i tempi futuri. In pratica vorrei trovare un sottoinsieme della coppia stato-azione che mi consenta di ricavare in modo indiretto la policy π utilizzando però come dato sempre le misure fatte.

Questo collegamento tra Q^* a un tempo t e all'istante successivo è ciò che mi è ciò che ha permesso l'utilizzo di un algoritmo iterativo:

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i^*(s', a') | s, a \right], \quad (9.8)$$

se ho un numero sufficiente di step temporali ($i \rightarrow \infty$), allora Q_i converge a Q^* .

Tramite questo approccio sono riuscita a implementare un algoritmo iterativo, a questo punto quello che devo fare è cercare di rendere il tutto un po' più flessibile, potrei quindi pensare di **parametrizzare la Q con una rete neurale**, quindi:

$$Q(s, a; \theta) \approx Q^*(s, a), \quad (9.9)$$

dove θ rappresenta i parametri del modello neurale. Se il modello è una DNN allora si parla di **Deep Q-Learning**.

Se ho già definito la Bellman equation semplicemente quello che faccio è inserire la Bellman equation all'interno di una loss function

$$L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} \left[(y_i - Q(s, a; \theta_i))^2 \right]. \quad (9.10)$$

Quindi semplicemente sto usando un modello non lineare per determinare la condizione che soddisfi la mia Bellman equation

$$y_i = \mathbb{E}_{s' \sim \epsilon} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right] \quad (9.11)$$

Chapter 10

Lezione 10

In questo momento siamo in una fase in cui ci si sta spostando (o per lo meno ci ci si sta provando) da un paradigma computazionale classico (CPU, GPU) verso nuovi paradigmi. Questo spostamento è determinato dalla necessità di avere a disposizione hardware più potenti per eseguire calcoli più velocemente (si parla per questo motivo di hardware accelerators). Ad esempio in questo momento dell'ambito HEP si sta compiendo la transizione da CPU a GPU.

I passi successivi sono quelli che includono l'utilizzo di schede FPGA/ASIC e di chip quantistici (vedi 10.1).



Figure 10.1: Rappresentazione dell'evoluzione dell'hardware

10.1 Nozioni di base

La ricerca in ambito quantistico è suddivisa in quattro aree di ricerca:

1. quantum communication: studia la possibilità di trasmettere informazioni tramite fotoni (si occupa ad esempio di crittografia quantistica, si possono trattare sia fotoni ottici - come nel caso della fibra ottica - che di fotoni non ottici, si studia la possibilità di conservare la correlazione tra stati ecc.),
2. quantum computing: più limitato ai chip, quindi al fare un esperimento computazionale che ha un comportamento quantistico
3. quantum simulation: si utilizzano hardware classici per testare un comportamento quantistico (possibilità di eseguire un codice su un chip quantistico)
4. sensing & methodology: area di ricerca in cui si usano i sistemi quantistici per misurare una qualche quantità (può essere utilizzato per realizzare un detector).

Nel 2019 Google è stata in grado di costruire un chip a 53 qubit e ha dimostrato che questo chip è in grado di eseguire compiti con tempi di calcolo molto migliori. Questo ha sostanzialmente aperto la porta alla quantum supremacy, per cui in sostanza è stato determinato un set di operazioni che sono molto più efficienti su questo chip piuttosto che su un chip classico.

L'obiettivo ora è capire per che cosa possono essere utilizzati efficientemente questi tipi di chip (un po' come è successo in passato per le GPU).

Qubit Nello spazio 2D di Hilbert dove definiamo una base computazionale

$$|0\rangle = (0, 1)^T \text{ e } |1\rangle = (1, 0)^T \quad (10.1)$$

Chiamiamo qubit un qualunque stato che sia una sovrapposizione di questi due stati:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \rightarrow , \quad (10.2)$$

dove $\alpha, \beta \in \mathbb{C}$ e $|\alpha|^2 + |\beta|^2 = 1$. Questo qubit può essere graficamente rappresentato sulla sfera di Bloch 10.2 Osservando l'immagine si capisce come ci siano

Figure 10.2: Rappresentazione di un qubit sulla sfera di Bloch

alcuni vantaggi rispetto al mondo classico: infatti **un'unità di qubit ha infinite possibilità**. Quindi se si considera uno spazio a 2 qubit, questo vive in uno spazio 4-dimensionale, quindi un generico stato a 2 qubit è definito come:

$$|\psi_2\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle, \text{ con } \sum_{i,j=0}^1 |\alpha_{ij}|^2 = 1. \quad (10.3)$$

Quindi nel caso di un generico sistema ad n qubit, lo stato generico sarà:

$$|\psi_n\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle, \text{ dove } \sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1. \quad (10.4)$$

Fino a qui il tutto è matematicamente banale, il problema è che finora sto scalando scalando in modo esponenziale in termini di quantità di memoria necessaria per poter utilizzare il sistema, infatti dato che ogni qubit è rappresentato da 2 numeri complessi, quindi da 4 numeri, se assumo una precisione doppia, la quantità di memoria a disposizione su un computer normale raggiunge al massimo circa 25 qubit. In generale non si possono superare, anche con i computer migliori una rappresentazione a 37 - 38 qubit, quindi la possibilità di potersi sganciare da un hardware classico consente anche di eliminare il problema della memoria per la conservazione dell'informazione.

Tecniche per l'implementazione di un qubit

1. Superconducting qubits: si ha una sorta di circuito elettrico con resistenza e capacità, questo sistema viene portato ad una temperatura dell'ordine del mK. A questa temperatura, questo sistema ha un comportamento simile a quello di un oscillatore armonico perfetto. Ad oggi questa è la tecnica più utilizzata perché più economica rispetto a quella del trapped ion. Per lavorare con i qubit si realizza una guida che inietta dei fotoni nel circuito che, a seconda della frequenza che sto iniettando, determina il comportamento del circuito tramite la risonanza.

2. Trapped ions: si prende l'atomo e si cerca di lavorare con livelli di energia specifici degli elettroni che vengono intrappolati tramite LASER a frequenze specifiche. Il vantaggio rispetto al caso precedente è che non devo stampare un chip ma tutto è realizzato in termini ottici.
3. Silicon quantum dots
4. Topological qubits
5. Diamond vacancies

Attualmente uno dei problemi principali è che l'hardware è super-noisy, il rumore di fondo alle misurazioni è ancora molto alto. Per questo motivo questo periodo viene chiamato anche NISQ (Noisy Intermediate-Scale Quantum era).

Quello che si può fare in attesa che la nostra precisione migliori è sicuramente studiare nuovi algoritmi che sfruttino le potenzialità della computazione quantistica ed adattare i problemi e le strategie che oggi applichiamo agli hardware classici all'hardware quantistico.

In particolare ci sono tre gruppi di algoritmi che possono essere adattati: i circuiti digitali basati sulle porte unitarie, gli algoritmi di annealing (sistemi di quantum computing analogici - per cui si mappa la hamiltoniana del sistema nel chip e si chiede ad esempio di fare un'evoluzione temporale o una minimizzazione ecc.) e infine i circuiti variazionali quindi cercare di usare l'architettura del circuito come architettura di rete neurale.

10.2 Circuiti quantistici

Dato uno stato iniziale $|\psi\rangle$ se voglio passare ad uno stato successivo devo applicare delle porte (o operatori unitari), eventualmente in sequenza, come si vede ad esempio in figura 10.3. Esistono porte che sono indipendenti, che lavorano in maniera

$$|\psi'\rangle = U_2 U_1 |\psi\rangle \rightarrow |\psi\rangle - \boxed{U_1} - \boxed{U_2} - |\psi'\rangle$$

Figure 10.3: Esempio di evoluzione di uno stato tramite operatori unitari (porte)

indipendente per ogni qubit ma anche porte che utilizzano uno o più qubit. Queste sono generalmente più interessanti perché sono quelle che consentono di creare e sfruttare correlazione, entanglement ecc. In figura 10.4 è riportato un elenco delle porte logiche più diffuse.

Per simulare questi sistemi per prima cosa definisco la ψ come un vettore nello spazio complesso. Il risultato sarà il prodotto 10.5 :

$$\psi'(\sigma) = \sum_{\sigma'} G(\sigma, \sigma') \psi(\sigma_1, \dots, \sigma'_{i1}, \dots, \sigma'_{iN_{targets}}, \dots, \sigma_N), \quad (10.5)$$

dove la somma corre sui qubits su cui la porta agisce e $G(\sigma, \sigma')$ è una matrice di porta che agisce sul vettore. È evidente che in questo le operazioni sono semplici, l'unico vincolo è la quantità di memoria necessaria per conservare queste informazioni.

Hadamard gate Per via della sua definizione la porta di Hadamard H crea una sovrapposizione di stati

Operator	Gate(s)	Matrix
Pauli-X (X)		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Figure 10.4: Esempi di quantum gates

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

(a) Matrice unitaria per l'Hadamard gate.

$$|0\rangle \rightarrow \boxed{H} \rightarrow \frac{|0\rangle + |1\rangle}{\sqrt{2}} \equiv |+\rangle$$

$$|1\rangle \rightarrow \boxed{H} \rightarrow \frac{|0\rangle - |1\rangle}{\sqrt{2}} \equiv |-\rangle$$

(b) Esempio di applicazione della porta Hadamard a due stati.

Figure 10.5: Rappresentazione matriciale e applicazione di un'Hadamrd gate

Rotazione La rotazione è definita come esponenziale di un angolo che moltiplica una matrice di Pauli. La cosa interessante di queste porte è che sono parametriche (dipendono dall'angolo θ) e soprattutto per queste porte esiste una dimostrazione matematica del fatto che se si moltiplicano tre porte di rotazione con angoli diversi si è in grado di coprire tutte le possibili unitarie esistenti

$$U = R_Z(\theta_1)R_Y(\theta_2)R_Z(\theta_3), \quad (10.6)$$

Questo è importante perché in ML si parla sempre di funzioni non lineari e queste sono effettivamente delle operazioni non lineari e parametriche, quindi in generale se costruisco delle porte non lineari otterrò in generale un comportamento non lineare.

CNOT La porta CNOT agisce come segue. La CNOT è anche un esempio di porta di controllo dove ho un primo qubit e poi un qubit target (vedi figura 10.7). In questo modo riesco a manipolare la mia architettura e creare stati che non esistono classicamente ma ho un sistema in cui due stati classi in qualche modo coesistono.

Quindi è interessante capire se la presenza di questi stati che non esistono classicamente può darmi un qualche vantaggio.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

(a) Matrice CNOT

$|0\rangle \xrightarrow{H} \frac{|0\rangle + |1\rangle}{\sqrt{2}} \equiv |+\rangle$

 $|1\rangle \xrightarrow{H} \frac{|0\rangle - |1\rangle}{\sqrt{2}} \equiv |-\rangle$

(b) Esempio di applicazione della porta CNOT

Figure 10.6: Rappresentazione matriciale e applicazione di una CNOT gate

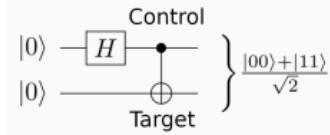


Figure 10.7: Esempi di control gate ottenuta con CNOT

10.2.1 Misurazioni

Dato che siamo nella NISQ tutte queste quantità non vanno solo simulate ma anche misurate, non ho un risultato analitico, quindi devo applicare una porta di misura che andrà a contare quante volte ho ottenuto lo stato 0 e quante volte lo stato 1 (se poi si fa un istogramma si ottiene una distribuzione 50% e 50%). L'aspetto importante è che in questo caso gli errori sui circuiti quantistici sono molto più importanti (sto facendo misurazioni statistiche quindi l'errore ha un'influenza molto maggiore).

10.2.2 Variational Quantum Circuits

Come si è visto in precedenza, i tre paradigmi del machine learning sono quelli del supervised learning, unsupervised learning e reinforcement learning. Tutte queste tecniche sono basate sull'idea di allenare dei modelli non lineari, quello che cci si chiede è se si può creare un circuito variazionale (come un variational autoencoding) che ci consenta di arrivare ad un risultato.

Il vantaggio pratico in questo caso sarebbe quello di **ridurre il numero di parametri** perché l'utilizzo di quantum chips mi consente la rappresentazione di stati classicamente non accessibili e poia anche un vantaggio energetico (i quantum chip dovrebbero consumare molto meno, un singolo qubit dopo la fase iniziale di raffreddamento rimane stabile).

La domanda è se dal punto di vista teorico si possa dimostrare la possibilità e la fattibilità di lavorare in questo spazio.

Teorema di Solovay-Kitaev Siamo in uno spazio di Hilbert e ci si chiedere se esiste una trasformazione unitaria che mi porti dalla stellina al punto verde. Dato che ho una trasformazione unitaria posso decomporla in n trasformazioni unitarie con l'obiettivo di trovare il path ottimale. Dato che la traiettoria α è una *possibile soluzione* ma non è detto che sia *la soluzione* posso cercare di variare un pochino questo percorso per trovare la soluzione ottimale.

Sia $\{U_i\}$ un insieme denso di trasformazioni unitarie e sia V una soluzione ottimale allora $|U_k \dots U_2 U_1 - V| < \delta$ dove il numero finito di porte è $k \sim \mathcal{O}\left(\log \frac{1}{\delta}\right)$

dove $c < 4$. L'approssimazione è efficiente e utilizza un numero di porte finite, quindi in generale è una soluzione fattibile, non solo teorica.

10.2.3 Strategie di implementazione

Approccio semi-classico La prima strategia è una tipologia di calcolo ibrido in cui i qubit al tempo iniziale sono inizializzati a 0, dopo di che si applica una porta unitaria parametrica che dipende da un angolo θ ed eventuali altre variabili x . Dopo di che lascio eseguire il circuito (alla fine attacco la porta di misure) e poi classicamente, con il risultato, costruisco la loss function e la passo ad un ottimizzatore classico.

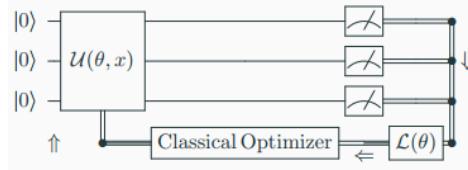


Figure 10.8: Esempio di training loop di un circuito quantistico

Data re-uploading Il problema è che devo poter inserire gli input, in questo senso ho un sistema vincolato nello spazio degli stati iniziali. Quindi devo trovare un modo di mappare il mio input negli *stati* iniziali. Finora l'allenamento avveniva su una rete FFN in cui i dati vengono elaborati da sinistra a destra, ora in un quantum computer si cambia paradigma. L'input non viene più immesso all'inizio ma viene introdotto come una serie di parametri che vengono immessi nelle porte, insieme ai parametri devo immettere anche i dati.

Preso un certo numero di qubit ho una fase di preprocessing in cui i dati ven-

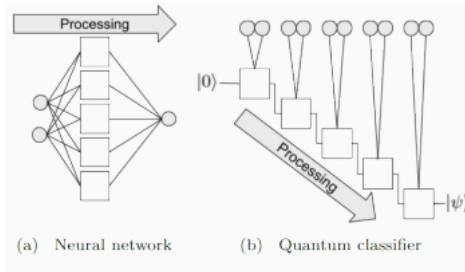


Figure 10.9: Esempio di applicazione del principio di data re-uploading

gono codificati sotto forma di stati quantistici, dopo di che si definisce il circuito quantistico parametrizzato che viene ottimizzato classificamente, vengono raccolte le misure e si minimizza la loss. Questo processo viene iterato fino a che non si ottiene un risultato soddisfacente.

Appendix

10.3 Images

Thus:

$$\nabla_{\mathbf{h}^{(t)}} J = \left(\frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right)^T (\nabla_{\mathbf{h}^{(t+1)}} J) + \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} \right)^T (\nabla_{\mathbf{o}^{(t)}} J)$$

Using design 1, where $\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$ and $\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)})$:

$$\nabla_{\mathbf{h}^{(t)}} J = \mathbf{W}^T \text{diag} \left(1 - (\mathbf{h}^{(t+1)})^2 \right) (\nabla_{\mathbf{h}^{(t+1)}} J) + \mathbf{V}^T (\nabla_{\mathbf{o}^{(t)}} J)$$

Figure 10.10: Slide 1 calcolo gradiente per reti neurali recurrent

Thus:

$$\nabla_{\mathbf{h}^{(t)}} J = \left(\frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right)^T (\nabla_{\mathbf{h}^{(t+1)}} J) + \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} \right)^T (\nabla_{\mathbf{o}^{(t)}} J)$$

Using design 1, where $\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$ and $\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)})$:

$$\nabla_{\mathbf{h}^{(t)}} J = \mathbf{W}^T \text{diag} \left(1 - (\mathbf{h}^{(t+1)})^2 \right) (\nabla_{\mathbf{h}^{(t+1)}} J) + \mathbf{V}^T (\nabla_{\mathbf{o}^{(t)}} J)$$

Figure 10.11: Slide 2 calcolo gradiente per reti neurali recurrent

10.4 Sparse connectivity

Mentre in un MLP si ha una correlazione totale perchè ciascun nodo è collegato a tutti i nodi del layer successivo e viceversa, ogni nodo riceve informazione da tutti i nodi del layer precedente; quando si ha *sparse connectivity* l'informazione sia in input che in output non viaggia da tutti i nodi verso tutti i nodi.

Si potrebbe inizialmente pensare che ci sia una perdita di informazione tra layers

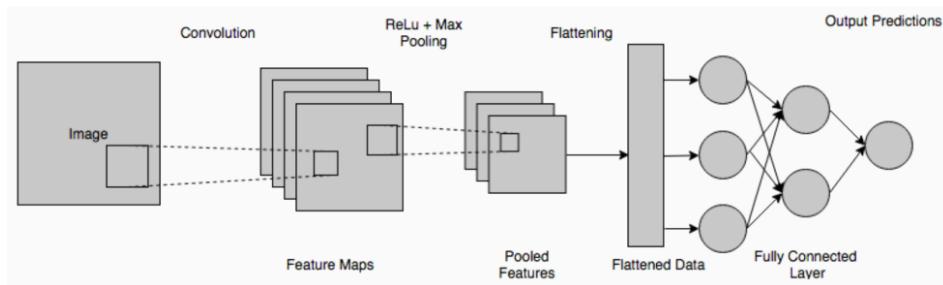


Figure 10.12: Esempio di una CNN completa

successivi, in realtà se si costruisce una DNN si vede facilmente che g_3 riceve informazione da ogni input. Questo è effettivamente utile perché mi consente di diminuire il numero di parametri che devo ottimizzare.

In questo senso una deep CNN mi consente una sorta di equilibrio per cui ho la conservazione dell'informazione tipica di un MLP ma senza tutti i parametri che dovrebbero avere per un MLP (denso).

10.5 Upspeed image localization

Nella presente sezione sono riportati dei grafici che mostrano un confronto tra le velocità di training e di test per diversi algoritmi di localizzazione di oggetti che sono stati illustrati in precedenza. La descrizione

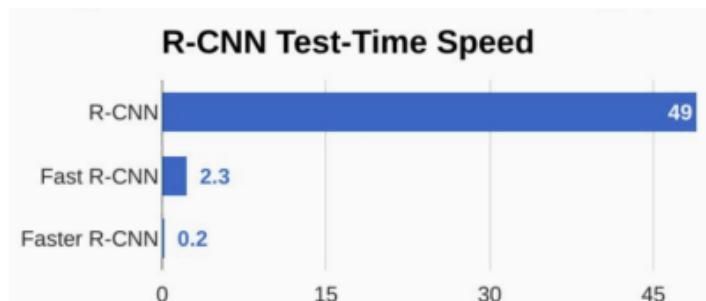


Figure 10.13: Workflow dell'applicazione di un algoritmo di transfer learning

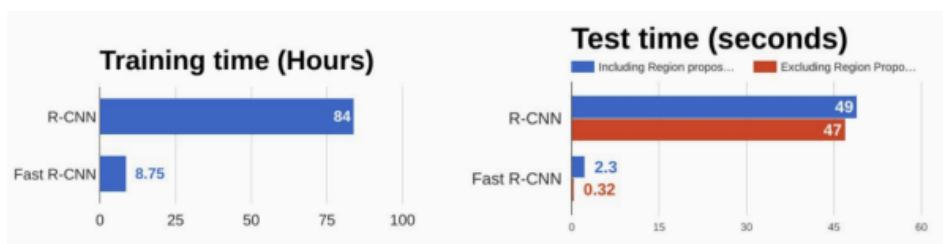


Figure 10.14: Workflow dell'applicazione di un algoritmo di transfer learning

10.6 Kullback–Leibler divergence

In teoria della probabilità e in teoria dell'informazione, la divergenza di Kullback–Leibler (anche detta divergenza di informazione, entropia relativa, o KLIC) è una misura non simmetrica della differenza tra due distribuzioni di probabilità P e Q . Specificamente, la divergenza di Kullback–Leibler di Q da P , indicata con $D_{KL}(P|Q)$ è la misura dell'informazione persa quando Q è usata per approssimare P : KL misura il numero atteso di bit extra richiesti per la Codifica di Huffman di campioni P quando si utilizza un codice basato su Q , piuttosto che utilizzare un codice basato su P . Tipicamente P rappresenta la "vera" distribuzione di dati, osservazioni, o una distribuzione teorica calcolata con precisione. La misura Q tipicamente rappresenta una teoria, modello, descrizione, o approssimazione di P .

Per due distribuzioni discrete P e Q la divergenza KL di Q da P è definita come:

$$D_{KL}(P|Q) = \sum_i P(i) \log_2 \left(\frac{P(i)}{Q(i)} \right). \quad (10.7)$$

Una proprietà importante della divergenza KL è che è sempre positiva.