ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

APPRENDIMENTO CONTINUO PER IL RICONOSCIMENTO DI IMMAGINI

Relazione finale in
MACHINE LEARNING

Relatore
Prof. DAVIDE MALTONI

Presentata da
LORENZO GATTO

Correlatore
Dott. VINCENZO LOMONACO

Terza Sessione di Laurea
Anno Accademico 2016 – 2017

Ringraziamenti

Ringrazio innanzitutto il professor Davide Maltoni per avermi concesso di realizzare questa tesi.

Ringrazio il correlatore Vincenzo Lomonaco per l'energia usata per aiutarmi durante lo sviluppo del progetto e per aver risposto ai dubbi che sono emersi durante la sua realizzazione. Il suo aiuto è stato fondamentale.

Un ringraziamento speciale va alla mia famiglia ed agli amici che mi hanno supportato durante questi anni di Università. Loro mi sono stati vicini in tutti i cinque anni di studio.

L'ultimo ringraziamento lo vorrei dare all'Università, che ha messo a disposizione il server con l'hardware necessario a svolgere questo elaborato.

Indice

Introduzione

Negli ultimi anni il **deep learning** ha riscontrando molto interesse da parte della comunità scientifica, in gran parte grazie ai risultati ottenuti a partire dal 2012 nell'ambito di visione artificiale, del riconoscimento del parlato e della sintesi vocale. Malgrado la tecnologia di deep learning corrente non sia avanzata come l'intelligenza umana, le applicazioni sono svariate e il progresso nel settore è molto rapido.

I più importanti risultati nell'ambito di deep learning sono stati ottenuti addestrando modelli di machine learning su **dataset statici**, iterando più volte la procedura di addestramento su tutti i dati disponibili. Ciò si contrappone a come gli umani imparano, cioè vedendo i dati (immagini, suoni, ecc.) una sola volta ma riuscendo comunque a ricordare il passato con un elevato livello di precisione. Il modo in cui gli umani apprendono viene chiamato **apprendimento continuo** (o continuous learning).

Negli ultimi anni sono avvenute rivoluzioni in termini di accuratezza dei modelli in problemi come il riconoscimento di immagini, ma tali modelli memorizzano enormi quantità di dati a cui si accede ripetutamente durante la procedura di training.

Approcci banali che evitano di osservare un pattern del dataset ripetutamente soffrono di un problema chiamato **catastrophic forgetting**, per il quale si tende a dimenticare le caratteristiche dei pattern visti in passato, facendo sì che il modello riconosca solo pattern simili a quelli visti recentemente.

Varie soluzioni al problema sono state proposte, ma nessuna ottiene ancora performance simili a quelle ottenibili con l'approccio cumulativo, che esegue la procedura di addestramento su tutti i dati iterativamente. Risolvere il problema del continuous learning è molto importante, si pensi ad esempio alle automobili a guida autonoma. Dopo un incidente, si vuole che l'automobile impari come evitarlo senza dover ripetere l'addestramento su gigabyte o terabyte di dati, che potrebbe richiedere settimane di tempo, in modo tale che incidenti simili non si ripetano.

La tesi è suddivisa in quattro capitoli. Il primo capitolo introduce concetti fondamentali del machine learning e della visione artificiale. Il secondo capitolo si focalizza sulle reti neurali, introducendo alcune delle tipologie di

rete più popolari per risolvere il problema del riconoscimento di immagini. Il terzo capitolo introduce le tecniche principali proposte per risolvere il problema dell'apprendimento continuo, tra cui una tecnica denominata iCaRL. Il quarto capitolo mostra le performance di tali tecniche applicandole al dataset CORe50.

Il contributo dell'autore è stato quello di calcolare l'accuratezza della tecnica iCaRL su dataset CORe50 e confrontare la performance ottenuta con i risultati ottenuti da altri autori nello stesso dataset utilizzando altre tecniche.

Capitolo 1

Introduzione alla visione artificiale e machine learning

1.1 Machine learning

Il **machine learning** è un campo dell'informatica che si occupa di dare ai computer la possibilità di apprendere da esempi. Ciò si contrappone alla classica programmazione, in cui bisogna fornire un algoritmo esplicito per risolvere il problema in esame, in quanto è la macchina a figurarlo utilizzando apposite tecniche.

Il machine learning fa parte di una disciplina più ampia chiamata **intelligenza artificiale**, che comprende anche algoritmi che non imparano da soli, come quelli utilizzati nei videogiochi per far muovere i giocatori o nella parte di pianificazione del movimento di sistemi autonomi.

Il machine learning viene quindi utilizzato in quegli ambiti in cui gli ingegneri non sono in grado di scrivere un algoritmo soddisfacente per risolvere il problema in esame, a causa della sua complessità e della quantità di casi speciali da considerare.

Negli ultimi anni il **deep learning**, un approccio di machine learning che simula approssimativamente le reti di neuroni della corteccia celebrale umana, sta ricevendo molta attenzione, grazie alla performance ottenuta con i recenti progressi. Oggi il deep learning viene applicato in molti ambiti, tra cui la guida automatica, la traduzione di testo [Wu et al., 2016], i sistemi di raccomandazione [Covington et al., 2016], il riconoscimento del parlato [Chiu et al., 2017] e il rendering di immagini fotorealistiche [Chaitanya et al., 2017]. A volte gli algoritmi di deep learning non solo riescono a superare la performance ottenibile con altri approcci classici, ma riescono anche a **superare la performance umana**, come nel caso del il riconoscimento del parlato [Amodei et al., 2015],

riconoscimento di immagini [He et al., 2015a] e giochi da tavolo [Silver et al., 2017].

Studi recenti mostrano che nei prossimi 10 anni molti dei lavori attualmente svolti dal personale umano verranno automatizzati e ciò comporterà un drastico cambiamento nella società [Acemoglu and Restrepo, 2017]. La capacità di adattamento sarà quindi di primaria importanza.

Il progresso nel campo dell'intelligenza artificiale preoccupa però alcune persone, in quanto si potrebbero avere problemi nel **controllare le azioni compiute da sistemi intelligenti e autonomi**, i quali potrebbero assumere atteggiamenti ostili per l'uomo come parte della soluzione al problema che si vorrebbe risolvere. D'altro canto, il potenziale è elevato, in quanto i robot potrebbero lavorare al nostro posto e fornire soluzioni a problemi ancora irrisolti.

Ray Kurzweil, informatico, inventore e futurista americano, ipotizza nel suo libro "The Singularity Is Near" che la **singolarità**, cioè la presenza di macchine di intelligenza sovra-umana, avverrà intorno al 2045. Tale macchina ha il potenziale di creare macchine ancora più intelligenti, con radicali cambiamenti tecnologici e sociali.

1.1.1 Tipi di dati

I tipi di dati su cui lavorano i sistemi di machine learning possono essere numerici o categorici.

I tipi **numerici** possono essere continui (come i numeri reali) o discreti (ad esempio i numeri interi); in ogni caso tali valori possono essere ordinati. In ambito di computer vision i valori numerici sono molto utilizzati, dato che un'immagine viene solitamente rappresentata come un array tridimensionale di numeri, in cui le tre dimensioni sono altezza, larghezza e canale (RGB).

I tipi **categorici** sono invece associati a caratteristiche qualitative. Un esempio può essere la tipologia di macchina (SUV, berlina, ecc.) o il colore degli occhi. A volte può essere presente un ordinamento, ad esempio temperatura alta, media o bassa.

Dato che il lavoro di tesi è focalizzato sulla classificazione di immagini, gli input costituiranno i valori numerici associati ai pixel nei tre canali di colore (rosso, verde e blu).

1.1.2 Tipi di problemi

I problemi di machine learning possono essere raggruppati in vari modi, uno dei quali è in problemi **supervisionati e non-supervisionati**.

Nell'apprendimento **supervisionato** i dati di addestramento sono etichettati, cioè è presente un'etichetta che può essere una classe come nei problemi di classificazione (es. cane, gatto) o un numero come nei problemi di regressione (es. l'altezza di una persona). Lo scopo del problema è di predire l'etichetta su dati che l'algoritmo non ha visto in precedenza.

Nell'apprendimento **non-supervisionato** il training set non è etichettato. Tipici problemi di questo tipo sono il clustering dei dati o la riduzione di dimensionalità. Questo è un compito spesso più difficile, dove a volte la difficoltà sta proprio nel definire l'obbiettivo finale, cioè la funzione da minimizzare o massimizzare. È molto semplice però reperire dati di questo tipo.

Esiste anche una terza categoria, **semi-supervisionata**, in cui il training set è solo parzialmente etichettato e si può sfruttare la vastità di dati non etichettati, facilmente reperibili, per migliorare le prestazioni finali del modello addestrato. Attualmente questa tecnica non è molto utilizzata a causa della scarsa qualità degli algoritmi non-supervisionati in confronto alla contrapparte supervisionata, ma è un'ambito di ricerca con alto potenziale.

Come già detto un'altra classificazione dei modelli machine learning è tra **classificatori** e **regressori**. I **classificatori** hanno lo scopo di indovinare la classe di un sample, cioè un valore categorico. Solitamente, in fase di training, tale valore viene mappato in un array booleano tramite una tecnica chiamata one-hot encoding, in cui un solo valore dell'array, associato alla classe del sample, può essere 1 (tabella 1.1). In fase di testing o deployment i valori dell'array saranno compresi tra 0 e 1 in base alla confidenza di ogni classe. A volte più classi possono essere vere in contemporanea, in tal caso si imposteranno a 1 tutte le relative posizioni.

I **regressori** sono invece utilizzati per la predizione di valori continui. Il più noto algoritmo è la regressione lineare, che moltiplica ogni valori di ingresso per uno scalare trovato automaticamente dal software e somma al risultato un valore, chiamato bias, anch'esso trovato dall'algoritmo (figura 1.1). I valori ottimali del bias e degli altri scalari vengono trovati con una procedura chiamata **gradient descent**, spiegata nel dettaglio nel seguito del capitolo.

Un altro modo per classificare i modelli è in base a come avviene l'addestramento: in **batch**, in modo **incrementale** o in modo **naturale**.

Nell'addestramento in **batch** l'addestramento viene effettuato una sola volta su un determinato training set. Una volta che la rete è addestrata, essa viene utilizzata così com'è finchè non verrà sostituita con una nuova rete (dopo mesi o anni). La maggior parte dei sistemi di machine learning reali opera in questo modo.

L'addestramento **incrementale** avviene invece in sessioni successive, in cui sono visibili solo alcuni dati di training. Il sistema dovrebbe essere in grado di imparare a classificare pattern simili a quelli visti nelle ultime sessioni di

Sample	Cilindrata	Lunghezza	Categoria
1	3,300	4.3	SUV
2	2,600	4.4	Berlina
3	1,500	3.2	Utilitaria
4	1,800	4.3	Berlina

(a) Dati prima dell'encoding

Sample	Cilindrata	Lunghezza	SUV	Berlina	Utilitaria
1	3,300	4.3	1	0	0
2	2,600	4.4	0	1	0
3	1,500	3.2	0	0	1
4	1,800	4.3	0	1	0

(b) Dati con applicato one-hot encoding

Tabella 1.1: Esempio di one-hot encoding: il tipo di macchina, essendo categorico, viene mappato in un array con tre elementi

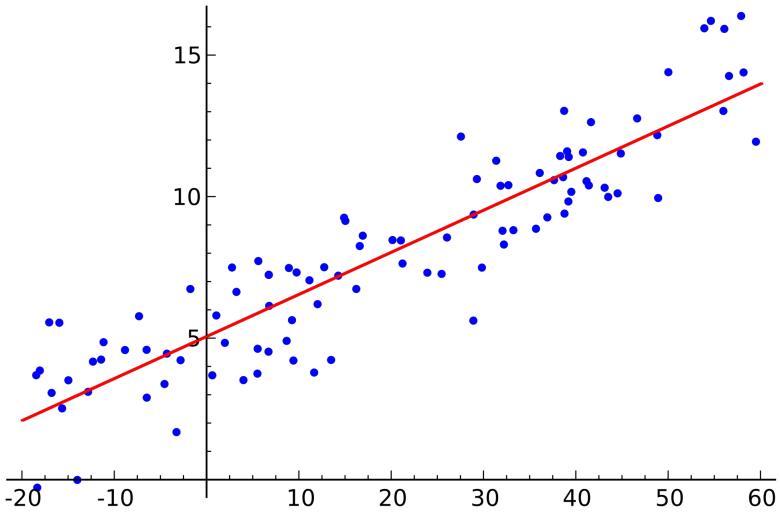

Figura 1.1: Regressione lineare con input a singola variabile. L'asse x indica l'input e l'asse y l'output. Il bias è 5. Fonte: wikipedia

addestramento, mantenendo una buona performance su pattern simili a quelli delle prime sessioni. Nella pratica questo tipo di addestramento soffre di un problema chiamato **catastrofic forgetting** [Goodfellow et al., 2013], in cui il sistema dimentica ciò che ha imparato in precedenza dopo il training su nuovi pattern. Particolare attenzione verrà posta a questo tipo di sistemi nella seconda parte della tesi.

L'addestramento **naturale** si riferisce al modo in cui imparano gli esseri umani, cioè imparando sempre cose nuove e soffrendo solo in minima parte di dimenticanze. L'approccio umano è principalmente non supervisionato, con una minima parte supervisionata (es. i nostri genitori ci han detto la "classe" degli oggetti che ci circondano). Per ottenere questo tipo di addestramento dovremmo probabilmente simulare in maniera più accurata il cervello umano, il cui meccanismo di funzionamento è putroppo ancora in maggior parte incompreso.

1.1.3 Training e valutazione prestazioni

Effettuare il **training** di un modello di machine learning vuol dire **trovare un insieme di parametri** Θ che permettono di far funzionare il modello con buona accuratezza sia sui dati di training che sui dati di test. Quanto un insieme di parametri funziona bene viene misurato tramite una funzione obiettivo $f(dataset, \Theta)$, che solitamente indica un errore da minimizzare.
$\Theta^* = argmin_\Theta f(dataset, \Theta)$ è il valore ottimo dei parametri Θ.

Non è corretto utilizzare i dati su cui verrà testato il modello durante l'addestramento, in quanto l'algoritmo potrebbe semplicemente memorizzare questi dati e farci sembrare che l'accuratezza sia del 100% in tali dati, mentre potrebbe funzionare male su dati mai visti. È per questo buona pratica utilizzare due dataset separati, chiamati **training set** e **test set**, allo scopo di addestrare e verificare la qualità del modello. Un modello di machine learning fa il suo addestramento sui pattern del training set, poi si verifica la sua efficacia provandolo sui pattern del test set. I modelli di machine learning cercano quindi di generalizzare informazioni prese dal training set al test set.

Tipici funzioni obbiettivo da minimizzare per problemi di categorizzazione sono le funzioni hinge loss e cross-entropy. Spiegheremo la funzione hinge loss in questo capitolo per introdurre le Support Vector Machines, quindi attualmente ci soffermiamo sulla cross-entropy.

La cross-entropy tra due distribuzioni discrete p e q misura quanto q differisce da p ed è definita da $H(p, q) = -\sum_v p(v) * \log q(v)$. Per utilizzarla come funzione di loss si pone p fisso come vettore *target* ottenuto con one-hot encoding e q pari al vettore di output del nostro modello. Se il nostro modello non restituisce direttamente valori di probabilità utilizzabili nella

formula sovrastante, si effettua una trasformazione con la funzione softmax $p_{ix} = e^{c_{ix}} / \sum_{z=1}^{n\_classes} e^{c_{zx}}$ dove p_{ix} è la probabilità stimata che il pattern x sia di classe i e c_{zx} è la confidenza della classe z per tale pattern.

Per effettuare il training è necessario impostare parametri che influenzano il training e la struttura del modello. Per differenziarli dai parametri che comporranno il modello finale, questi vengono chiamati **iperparametri**. Esempi di questo tipo di parametri sono il numero di strati in una rete neurale, il tipo di funzione di errore (loss function, la funzione obbiettivo) o il learning rate delle reti. Maggiori informazioni verranno date successivamente, ma al momento è giusto capire come scegliere questi parametri.

Se gli iperparametri fossero scelti in base alla prestazione sul training set, si otterrebbe un modello poco *interessante*, accurato sui dati del training set ma probabilmente non altrettanto accurato nei dati del test set. Un modello che memorizza tutti i dati del training set, come il classificatore Nearest-neighbour, sembrerebbe ideale, ma potrebbe comportarsi male in nuovi dati. Se invece si usasse il test set per scegliere tali parametri, il test set non sarebbe più considerabile un insieme di dati mai visto in precedenza, e le prestazioni ottenute su tale insieme di dati non stimerebbero più le prestazioni su dati mai visti prima. Per questo motivo dovrebbe venir sempre utilizzato un **validation set**, cioè un ulteriore insieme di dati etichettati usato esclusivamente per trovare i migliori iperparametri. L'addestramento continua ad essere fatto sul training set e si scelgono come iperparametri quelli che minimizzano la funzione di errore sul validation set. Solo alla fine viene provato il modello sul test set, su cui si dovrebbero avere performace simili a quelle ottenute sul validation set. La figura 1.2 mostra come partizionare i dati.

A volte il validation set non viene fornito come parte del dataset e deve essere estratto casualmente tra i dati del training set. Nel caso in cui i dati di training siano pochi è consigliato utilizzare la cross-validation[1]. È opportuno garantire che validation set e test set abbiano distribuzioni molto simili dei dati, altrimenti si possono avere risultati inaspettati sulla prova finale del modello sul test set.

Nel caso in cui le prestazioni nel training set siano molto diverse dalla performance su validation o test set, si dice che il modello sta facendo **overfitting**, cioè sta memorizzando i dati di training invece di capirne i pattern (figura 1.3), a volte in maniera subdola su milioni di parametri che possono comporre un modello. Tecniche per ridurre questo problema possono essere quella di usare un modello più semplice o quella di aggiungere **regolarizzazione**, cioè far tendere i parametri del modello verso una soglia per cui non si pone il problema dell'overfitting (solitamente 0).

[1] https://www.openml.org/a/estimation-procedures/1

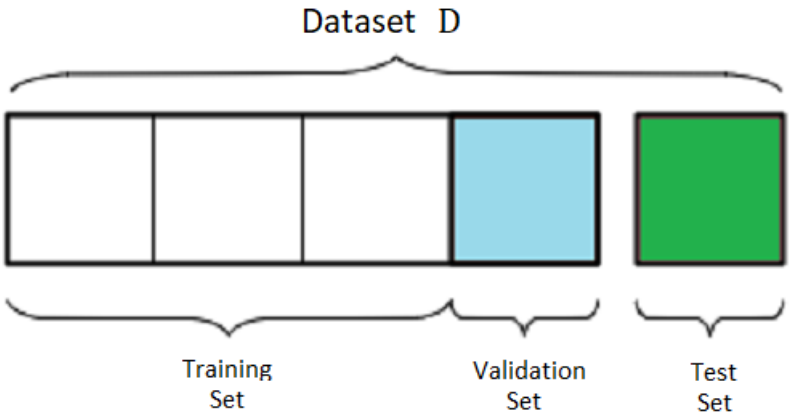

Figura 1.2: Come i dati sono partizionati in training, validation e test set

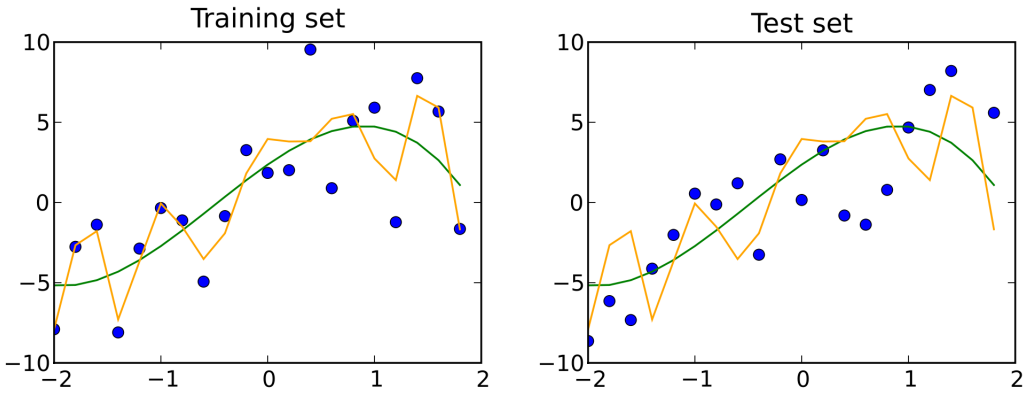

Figura 1.3: Overfitting: la linea gialla sta facendo overfitting, mentre la linea verde segue l'andamento dei dati. Fonte: wikipedia

1.2 Visione artificiale

La visione artificiale si occupa di comprendere il contenuto di immagini e video digitali. La quantità di dati visuali è aumentata a dismisura negli ultimi anni, grazie al grande numero di sensori presenti in smartphone, fotocamere, ecc.. È stato stimato che l'80% del traffico internet derivi da video, quindi estrarre informazioni da essi è una grande fonte di informazione. Il problema è che capire questo tipo di dato è semplice per un umano, la cui **metà del cervello si occupa di visione**, ma molto complesso per un computer; per tale motivo il traffico video può essere considerato la *materia oscura* del web. Negli ultimi anni il progresso nell'ambito di visione artificiale ha avuto un avanzamento senza precedenti, dovuto per la maggior parte al successo dei

modelli deep learning dal 2012 in poi [LeCun et al., 2015]. Le applicazioni su cui vengono applicati tali sistemi sono disparate, dalla diagnosi di tumori, alla guida automatica, la sorveglianza e la ricostruzione di immagini alterate.

1.2.1 Storia della visione artificiale

Lo studio della visione artificiale è iniziato negli anni '60 dalle università pioniere dell'intelligenza artificiale. Si pensò che la visione artificiale fosse un problema semplice, al punto che fu assegnato come progetto estivo nel gruppo di Artificial Intelligence dell'MIT. Dopo più di 50 anni, il problema della visione artificiale è stato solo parzialmente risolto. Negli anni successivi la maggior parte degli approcci funzionanti sono stati ottenuti grazie all'estrazione di feature tramite algoritmi ingegnerizzati da esperti in mesi di lavoro. Tra gli algoritmi più influenti è giusto citare l'estrazione di bordi [Canny, 1986], il rilevamento facciale [Paul Viola, 2001] e l'estrattore di punti di interesse SIFT [Lowe, 1999].

Dal 2006 al 2012 è stata svolta annualmente la competizione Pascal VOC [Everingham et al., 2015], il cui scopo è riconoscere immagini appartenenti a 20 classi.

Il più grande dataset che ha contribuito all'esplosione recente del deep learning è **ImageNet**, composto da oltre 14M di immagini di 22k categorie diverse, di cui un sottoinsieme di 1000 classi viene usato per l'ImageNet Challenge [Russakovsky et al., 2015]. Dalla figura 1.4 si può notare quanto il deep learning, grazie a cui è stata vinta la competizione dal 2012 [Krizhevsky et al., 2012] in poi, ha aiutato nel campo della visione artificiale. L'approccio utilizzato è basato su reti neurali convoluzionali, basate sull'approccio di LeCun utilizzato per la categorizzazione di cifre scritte a mano [LeCun et al., 1999] ma con maggiore profondità della rete e con modifiche che hanno reso possibile un training efficace.

1.3 Support vector machines

In questa sezione introduciamo le **Support Vector Machine** (SVM) per la classificazione di immagini. Esse saranno di grande aiuto per capire tecniche più complesse come reti neurali o reti neurali convoluzionali, l'attuale stato dell'arte per la visione artificiale.

Le Support Vector Machine sono composte da una **funzione di mapping lineare** e da una **funzione di costo** (o errore) hinge loss. La funzione di mapping trasforma l'input del modello in valori di confidenza di ciascuna classe. Tali valori di confidenza verranno poi utilizzati per calcolare l'errore commesso

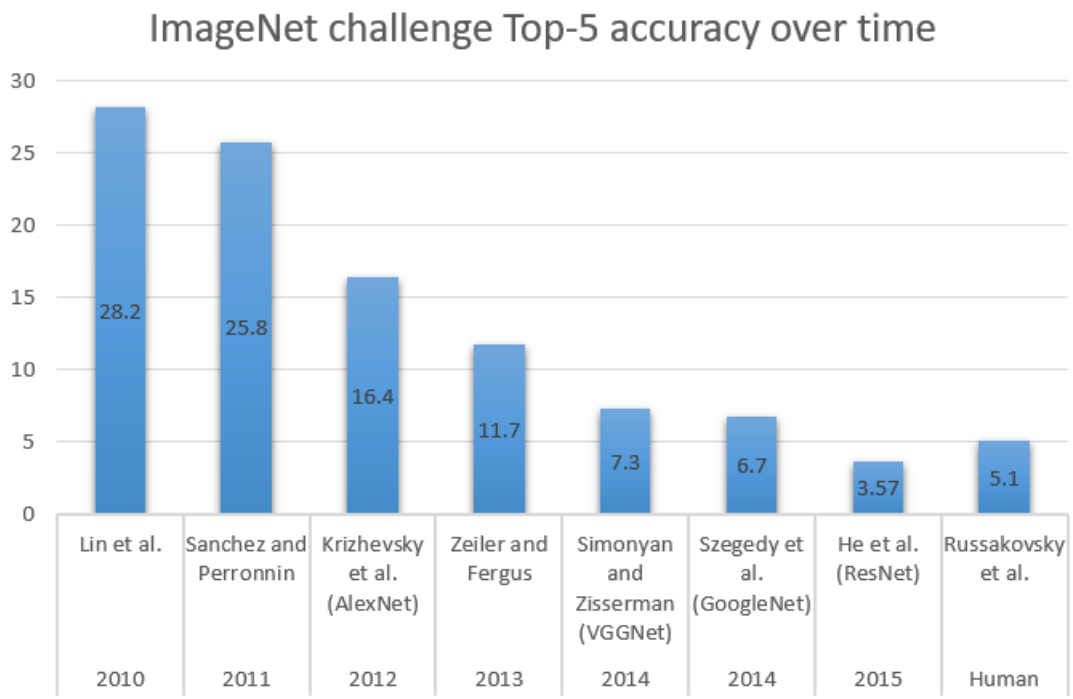

Figura 1.4: Accuratezza top-5 ottenuta negli anni nella competizione ImageNet.

dalla rete: in particolare si impone che la classe data come label nei dati abbia una confidenza maggiore delle altre con un certo livello di margine.

1.3.1 La funzione di mapping

Assumiamo che il training set sia composto da immagini $x_i \in \mathbb{R}^D, i \in 1..n, D \in \mathbb{N}$ dove n è il numero di immagini del training set, x_i è un sample e D il numero di features reali che compongono il sample. Indichiamo le label dei dati di training con $y_i \in \{1, ..K\}$. Per esempio, in ImageNet si ha $D = 256 \times 256 \times 3$ e $K = 1000$. Definiamo la funzione $f : R^D \to R^K$ che mappa i pixel in ingresso alle confidenze. La funzione per la SVM, essendo lineare, avrà la forma $f(x_i, W, b) = Wx_i + b$, dove W e b sono parametri. In tale funzione si assume che x_i sia "appiattito" in un singolo vettore di grandezza $D \times 1$, W sia di grandezza $K \times D$ e b di grandezza $K \times 1$. I parametri W sono chiamati pesi (weights) mentre i parametri b sono chiamati bias. Il compito successivo sarà quello di trovare valori di W e b in modo tale che le confidenza delle classi sia il più possibile compatibili con i valori del training set.

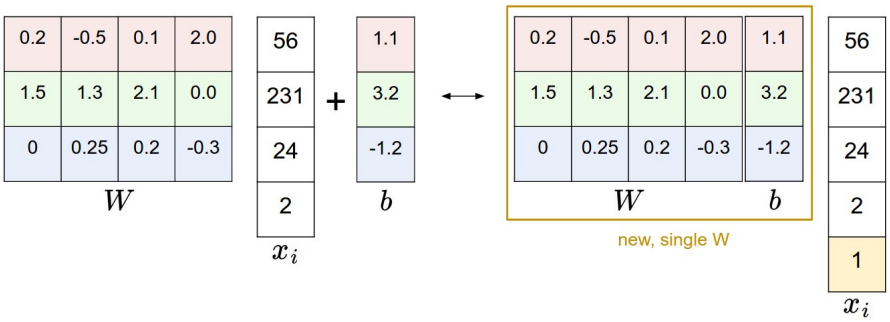

Figura 1.5: Trucco per rimuovere la variabile b.
Fonte: https://github.com/cs231n/cs231n.github.io

Prima di procedere è bene introdurre una tecnica per semplificare le formule, rappresentando W e b come una singola variabile. La funzione $f(x_\mathrm{i}, W, b) = Wx_\mathrm{i} + b$ può essere riscritta come $f(x_\mathrm{i}, W) = Wx_\mathrm{i}$ aggiungendo ad x_i una colonna con il valore 1. L'immagine 1.5 chiarisce la tecnica. È facile vedere che le due rappresentazioni sono del tutto equivalenti.

Il limite delle SVM è che il modello è lineare. Il più facile problema che una SVM non è in grado di risolvere è quello dello XOR. Si consideri la figura 1.6; è facile convincersi che non è possibile separare le due classi con una retta. Nel seguito della tesi considereremo soluzioni a questo problema, in particolare le reti neurali.

1.3.2 La funzione di errore

Nell'esempio in figura 1.7 possiamo vedere che, partendo da una SVM con pesi casuali, il gatto ha una confidenza bassa (-96.8) per la classe *cat*, mentre ha una confidenza molto alta (437.9) per la classe *dog*. Per il training e il testing del modello, è necessario avere funzioni che misurano quanto siamo insoddisfatti con i valori di confidenza ottenuti. Tali funzioni sono le funzioni di errore, tra cui le più utilizzate per i problemi di classificazione sono la hinge loss e la cross-entropy. Le support vector machine utilizzano la **hinge loss**, cioè

$$L_\mathrm{i} = \sum_{j \neq y_i} max(0, c_j - c_{y_i} + 1)$$

dove $c_j = f(x_i, W)_\mathrm{j}$ è la confidenza della j-esima classe per l'input x_i. Intuitivamente, si vuole che la confidenza di una classe sia inferiore a quella della vera classe con un certo margine, impostato ad 1 in questo caso. È dimostrabile che usare un altro valore per il margine non fa altro che scalare la soluzione trovata.

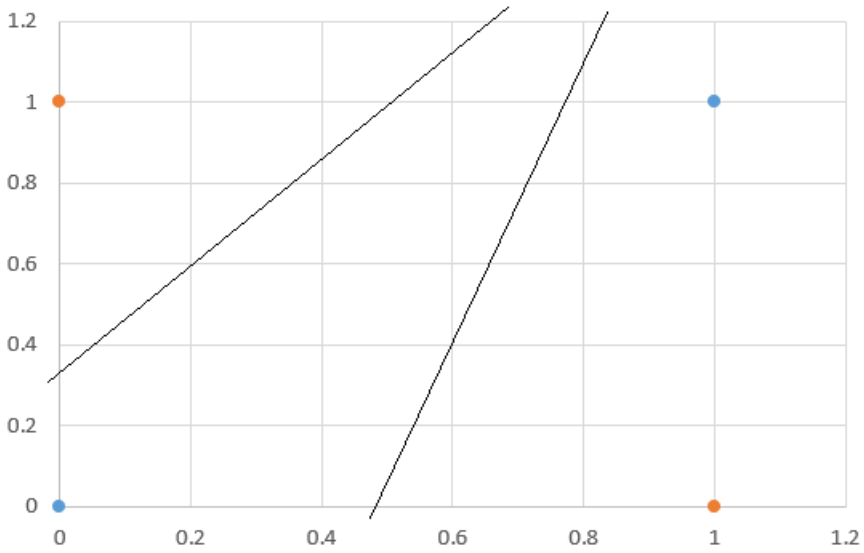

Figura 1.6: Il problema dello XOR: non si riescono a separare le due classi (arancione e blu) con una sola retta.

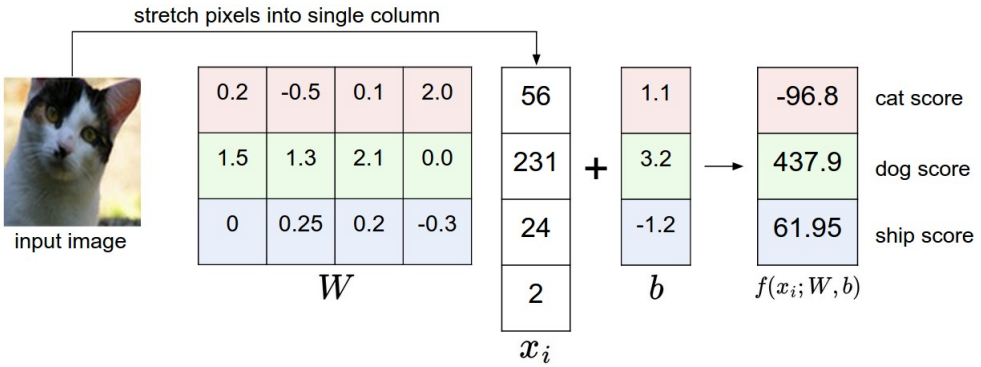

Figura 1.7: Esempio di SVM per classificazione di gatti, cani e navi.
Fonte: https://github.com/cs231n/cs231n.github.io

Al valore delle loss L_i bisogna aggiungere un altro fattore di **regolarizzazione $\mathbf{L_2}$** $R(W)$ pari a $R(W) = \sum_k \sum_z W_{\text{kz}}^2$. Tale fattore evita che i parametri della rete crescano indefinitamente e solitamente aumenta la performance sui dati di test a causa del minor overfitting in tali dati. La formula finale della loss è

$$L = \frac{1}{N} \sum_{i=0}^{n\_samples} L_i + \alpha R(W)$$

in cui il parametro α deve essere trovato attraverso la validazione.

1.4 Backpropagation

Per addestrare la rete SVM vista in precedenza o una rete neurale a più strati che incontreremo nei capitoli successivi, occorre calcolare le derivate della funzione di errore rispetto ad ogni parametro della rete. Nel caso di una Support Vector Machine è abbastanza semplice ricavare formule esplicite per il calcolo delle derivate, ma la complessità di tali formule cresce con la complessità della rete, diventando un compito molto oneroso nel caso di reti multi-strato con centinaia di strati differenti.

La tecnica della **backpropagation** permette di effettuare un calcolo numerico dei gradienti in modo preciso ed efficiente, basandosi sulla formula della **chain rule**. La chain rule afferma che $\frac{df}{dx} = \frac{df}{dq} * \frac{dq}{dx}$. Nel nostro caso, f è la loss function e x verrà sostituito con ognuno dei parametri. Utilizzando come q una variabile per cui si è già calcolato il valore $\frac{df}{dq}$ e di cui si conosce una formula matematica per $\frac{dq}{dx}$, si procede per induzione per il calcolo di tutte le derivate. Si veda l'esempio in 1.8 per chiarezza.

Si inizia con la propagazione in avanti, calcolando il valore di output di ogni variabile. Poi inizia la propagazione all'indietro dei gradienti, il cui primo passo è impostare il valore di $\frac{df}{df}$ ad 1. Il valore di $\frac{df}{dh}$, impostando $q = f$ nella formula della chain rule, sarà $\frac{df}{df} * \frac{df}{dh}$. $\frac{df}{df}$ è il valore 1 già calcolato, mentre $\frac{df}{dh}$ è pari a $z = -4$, ottenuto derivando $f = z * h$ rispetto ad h. Per ottenere, ad esempio, $\frac{df}{dy}$, si pone $q = h$ ottenendo la formula $\frac{df}{dy} = \frac{df}{dh} * \frac{dh}{dy}$. $\frac{df}{dh}$ è stato già calcolato ed è uguale a -4, mentre $\frac{dh}{dy}$ è pari a 1, ottenuto derivando $h = x + y$ rispetto ad y. Il resto dei calcoli sono analoghi. Si nota che il calcolo delle derivate procede livello per livello, e basta saper calcolare le derivate di ogni livello in modo indipendente per poter procedere.

I moderni framework di deep learning permettono di creare reti computazionali utilizzando strati pre-disponibili e rendono trasparente il calcolo del gradiente, quindi non c'è bisogno di preoccuparsene per utilizzi standard. In più danno la possibilità di aggiungere strati personalizzati per progetti di ricer-

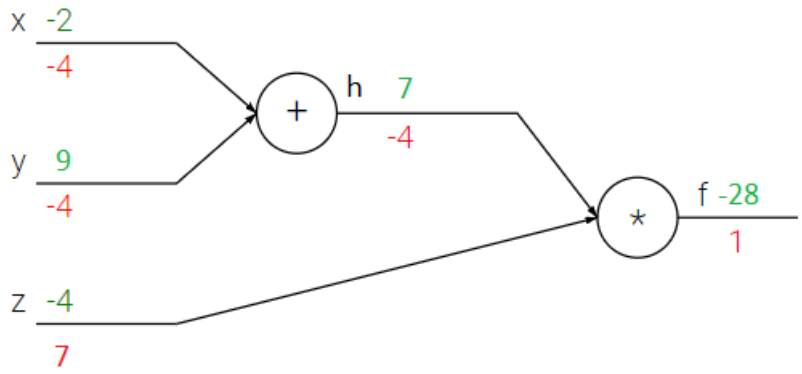

Figura 1.8: Backpropagation: i valori in verde sono i valori di ogni variabile di input o calcolata, mentre i valori in rosso sono i valori dei gradienti

ca, definendo le formule per la propagazione in avanti (calcolo delle variabili) e all'indietro (calcolo dei gradienti).

1.5 Gradient descent

Il **gradient descent** (discesa del gradiente) è un algoritmo di ottimizzazione molto utilizzato nell'ambito di machine learning. Esso permette di trovare un ottimo locale in un problema di ottimizzazione a più variabili. Nel caso di problemi convessi, l'ottimo locale coinciderà con l'ottimo globale. In una SVM, le variabili sono la matrice W e il vettore b e la funzione da minimizzare è la funzione di errore.

L'algoritmo gradient descent funziona muovendo ogni variabile in direzione opposta al suo gradiente.

```
for iter from 0 to number_iterations:
    weights_gradient = evaluate_gradient(loss_function, data, weights)
    weights += - learning_rate * weights_gradient
```

Il gradient descent non è utilizzato nella forma appena descritta in scenari con miglioni di campioni di training, perchè calcolare il gradiente richiede la visita di tutto il training set. Per questo motivo è stato proposto lo stochastic gradient descent, che approssima il gradiente ad ogni iterazione attraverso un sottoinsieme casuale del training set. La procedura diventa:

```
for iter from 0 to number_iterations:
    training_sample = sample(data, batch_size)
    weights_gradient = evaluate_gradient(loss_function, training_sample, weights)
    weights += - learning_rate * weights_gradient
```

La grandezza del batch size può variare da 1 ad alcune centinaia. È stato dimostrato teoricamente e empiricamente che questa procedura, chiamata **stochastic gradient descent** (SGC) permette una convergenza più veloce [Bottou, 2012]. Per una convergenza ancora migliore su problemi di ottimizzazione non-convessa, come i modelli di deep learning, si utilizza una tecnica chiamata **momentum**. La tecnica consiste nell'accumulare i gradienti in una variabile di media esponenziale invece di utilizzare solo i gradienti dell'ultimo batch. L'effetto è simile a quello che si ottiene lanciando una pallina dalla cima di una montagna: il movimento della pallina scende senza cambi di direzione repentini. L'eliminazione di movimenti a zig-zag permette una convergenza più veloce. La procedura aggiornata è:

```
for iter from 0 to number_iterations:
    training_sample = sample(data, batch_size)
    weights_gradient = evaluate_gradient(loss_function, training_sample, weights)
    cumulative_gradient = γ∗cumulative_gradient + (1−γ)∗weights_gradient
    weights += - learning_rate * cumulative_gradient
```

Valori di γ tipici sono intorno a 0.9. Questo è l'algoritmo di ottimizzazione utilizzato nelle recenti soluzioni vincitrici della competizione ImageNet. Negli ultimi anni sono state create varianti del Gradient Descent per cercare di velocizzare il training. Tra le varianti degne di nota ci sono Adagrad [Duchi et al., 2011], RMSProp[2] e la più recente e promettente Adam [Kingma and Ba, 2014].

C'è anche chi critica il gradient descent e la backpropagation per essere troppo discordanti dal funzionamento del cervello umano, ma al momento questa è la tecnica per il training delle reti neurali che offre i risultati migliori.

[2] `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

Capitolo 2

Reti neurali

Le **reti neurali** sono una tipologia di modelli più adatti a manipolare dati ad alta dimensionalità. Le reti neurali artificiali vengono utilizzate sia per cercare di capire le reti neurali biologiche che per risolvere problemi come quelli di visione artificiale, di sintesi vocale e di riconoscimento vocale, difficilmente risolvibili tramite altri approcci.

Tipiche strutture di queste reti sono composte da **strati di neuroni**, in cui ogni neurone di uno strato è connesso a tutti i neuroni dello strato precedente, senza formare cicli. Tale tipologia di reti viene chiamata feed-forward, differenziandosi dalle reti neurali ricorrenti che possono invece contenere cicli. La figura 2.1 mostra l'esempio di una rete feed-forward a 3 strati, di cui uno di input, uno di output e uno nascosto. Un modello di neurone moltiplica gli ingressi per dei parametri (da imparare in fase di training) e trasforma il valore ottenuto tramite una **funzione di attivazione** per ottenere l'output del neurone, come raffigurato in figura 2.2.

Una funzione di attivazione molto utilizzata è la funzione ReLu, cioè $f(x) = \max(0, x)$. Tale funzione è continua e derivabile (a parte nel punto x=0, dove si può porre la derivata pari a zero) e funziona molto bene sulle moderne architetture multi-strato.

Sebbene sia stato dimostrato che una rete a tre strati con un numero sufficiente di unità nascoste sia sufficiente a rappresentare qualunque funzione, è spesso preferibile utilizzare un numero superiore di strati nascosti di grandezza limitata, in quanto la rappresentazione a tre strati potrebbe avere un numero molto elevato di unità nascoste [Hastad, 1986].

Trovare il numero ideale di strati e il numero ideale di neuroni per strato è un'arte e solitamente si verifica l'efficacia di varie architetture tramite il validation set. Tipicamente si ha come risultato che un numero superiore di strati dà prestazioni superiori se ben regolarizzato, ma ciò succede a discapito dell'efficienza.

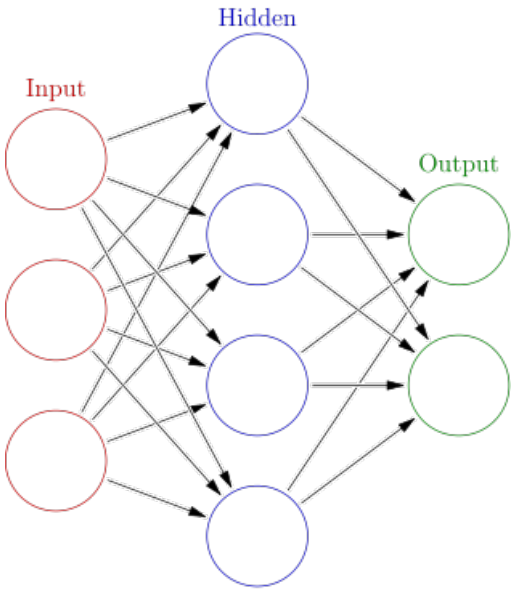

Figura 2.1: Rete neurale artificiale composta da 3 strati

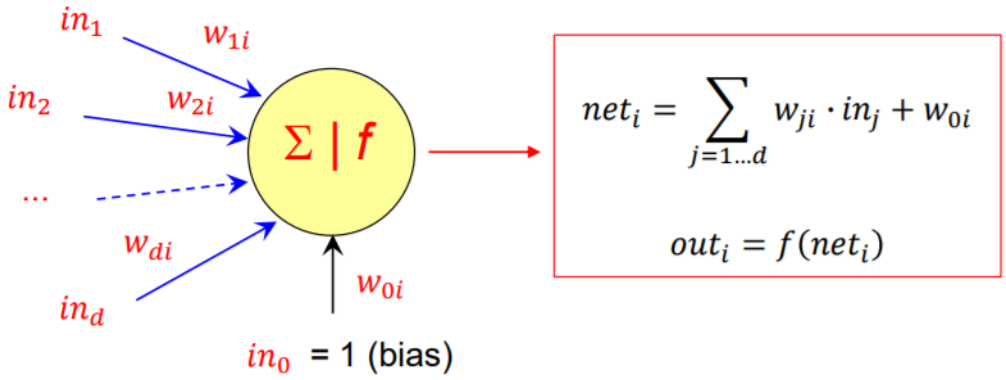

Figura 2.2: Modello di un neurone artificiale. Gli input vengono moltiplicati per dei parametri e sommati tra di loro. Il risultato viene poi trasformato con una funzione di attivazione f.
Fonte: http://bias.csr.unibo.it/maltoni/ml/

2.1 Funzioni di attivazione

Non è presente solo la funzione di attivazione ReLu. Anzi, essa ha guadagnato popolarità solo negli ultimi anni. I **percettroni**, così come vennero chiamati i neuroni da Rosenblatt nel 1956, utilizzano una funzione a scalino $f(x) = 1$ if $x > 0$ else 0. Tale funzione ha derivata pari a zero in tutti i punti (tranne che nel punto di $x = 0$ in cui la derivata non è definita), quindi non adatta all'utilizzo con gradient descent (i parametri della rete non cambierebbero mai). Si utilizzava una regola per l'addestramento chiamata **delta rule**, ma attualmente tale tecnica non viene più utilizzata in quanto non scalabile.

Una funzione di attivazione che invece fu utilizzata per anni è la **sigmoide**, nelle due varianti **logistica** $f(x) = 1/1 + e^{-x}$ e **iperbolica** $f(x) = \tanh x$. Entrambe le funzioni operano bene se utilizzate su reti a pochi strati (3 o 4), ma soffrono di un problema chiamato **vanishing gradient** su reti a più strati. Il problema consiste nel fatto che tali funzioni hanno derivata tendente a zero per valori di x molto diversi da zero, rallentando il training sugli strati iniziali di reti profonde, dopo che i gradienti degli strati superiori vengono moltiplicati per valori prossimi allo zero variate volte. Aumentare il learning rate sugli strati iniziali porta invece a instabilità sul training.

Dal 2012 in poi è stata molto usata la funzione **Rectified Linear Unit** (ReLu), cioè $f(x) = \max(0, x)$. Tale funzione si comporta molto bene su reti deep ed è stata utilizzata con successo anche su reti con centinaia di strati. Varianti come la Leaky Relu hanno derivata differente da 0 anche per input negativi, ma empiricamente non sono stati trovati vantaggi.

Si nota che la funzione di attivazione deve essere **non-lineare**. Si dimostra infatti che se essa è lineare, si può ottenere una rete equivalente eliminando lo strato dalla rete, perdendo ogni vantaggio del deep learning.

2.2 Inizializzazione dei parametri

L'inizializzazione dei parametri deve essere svolta correttamente per garantire una buona convergenza della rete con la tecnica del gradient descent (e varianti). L'inizializzazione non è molto importante nelle reti con funzione obiettivo convessa come le SVM, in quanto il training è semplice e tende sempre alla stessa soluzione, a prescindere dai valori iniziali. La funzione obiettivo di una rete a più strati diventa **non-convessa** a causa delle funzioni di attivazione non-lineari, e ciò rende più difficile la fase di training.

Solitamente si impostano i pesi con valori casuali di distribuzione gaussiana, mentre si impostano i bias a zero. La gaussiana viene impostata con media a 0 e deviazione standard $sqrt(2/(in + out))$ dove in e out sono il numero di neuroni di ingresso e in uscita a cui è collegato il neurone in esame. Questa

inizializzazione, chiamata **inizializzazione di Xavier**, permette di mantenere la deviazione standard dei gradienti pressoché invariata in ogni strato, facilitando il training. Per la derivazione matematica della formula si consulti il paper [Glorot and Bengio, 2010]

2.3 Reti neurali convoluzionali

Il funzionamento a strati delle reti neurali fully-connected permette il riuso di informazioni calcolate in precedenza. È stato dimostrato che anche il sistema visivo umano opera attraverso più strati computazionali [Kruger et al., 2013]. Il numero di connessioni su tali reti è però elevato e il gran numero di parametri fa sì che serve un numero molto elevato di immagini per ottenere una buona generalizzazione dei modelli.

Le **reti convoluzionali** riducono il numero di parametri tramite condivisione. Le reti neurali convoluzionali (CNN) esistevano già dal 1998 dove la rete di LeCun [LeCun et al., 1999] era lo stato dell'arte sul problema di riconoscimento di cifre scritte a mano. Da tale momento è aumentata la potenza di calcolo disponibile, la quantità e grandezza dei dataset (es. ImageNet) ed è stato risolto il problema del vanishing gradient tramite l'utilizzo della funzione ReLu. Ciò ha portato nel 2012 alla vittoria della competizione ImageNet alla rete AlexNet [Krizhevsky et al., 2012], non molto diversa dalla rete LeNet del 1998 di LeCun.

Le architetture moderne sono composte da strati predefiniti componibili per ottenere modelli con performance e accuratezze diverse. Trovare l'architettura perfetta è una questione di esperienza, per questo spiegheremo gli strati più utilizzati e forniremo esempi di reti vincitrici della competizione ImageNet.

2.3.1 Strato convoluzionale

Gli strati convoluzionali sono stati introdotti da LeCun [LeCun et al., 1999] appositamente per l'elaborazione di immagini. L'**elaborazione locale** e la **condivisione dei pesi** rendono il modello più semplice e più efficace. L'elaborazione locale fa sì che i neuroni processino nello stesso modo diverse parti di un'immagine, un comportamento desiderato: se è presente un gatto nella parte destra di un'immagine, dovrebbe essere riconosciuto e trattato allo stesso modo se fosse invece presente nella parte sinistra.

Alla base di uno strato convoluzionale c'è l'operazione di **convoluzione**, cioè l'applicazione di filtri digitali. Un filtro bidimensionale viene fatto scorrere nell'immagine eseguendo il prodotto scalare tra la porzione dell'input coperta e il filtro (entrambi trattati come vettori). Si veda l'immagine 2.3. Spesso si

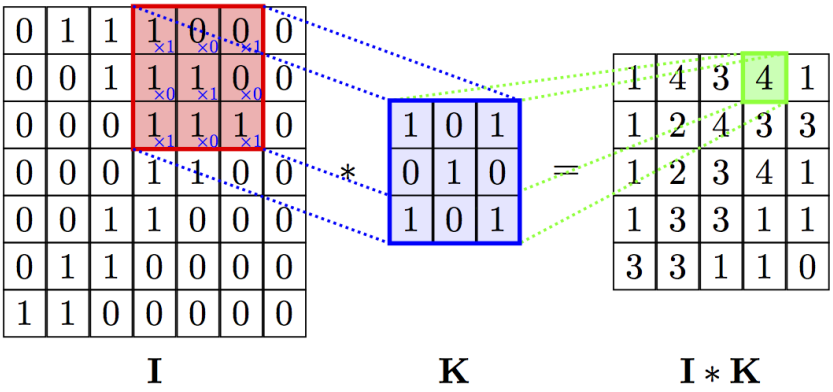

Figura 2.3: Convoluzione bidimensionale tra immagine I e filtro K
Fonte: https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution

vuole preservare il numero di neuroni in orizzontale e in verticale; per questo si aggiunge un padding, cioè un bordo intorno all'immagine iniziale, solitamente inizializzato a 0 o posto pari al neurone più vicino. Ad esempio, per filtri 3×3, basta aggiungere un padding di larghezza 1 in ogni lato per mantenere le stesse dimensioni dell'input. Per filtri 5×5, bisognerebbe aggiungere un padding di 2. I filtri sono di solito quadrati e di grandezza dispari, anche se nulla vieta di provare alternative.

Un lettore attento avrà notato che nelle immagini reali l'input non è bidimensionale ma tridimensionale (la terza dimensione è data dai colori). Per questo viene utilizzata una convoluzione tridimensionale: il filtro e l'immagine di input diventano tridimensionali. La terza dimensione del filtro eguaglia la terza dimensione dell'input e la convoluzione avviene in modo analogo, iterando nell'input facendo la convoluzione su tensori[1] tridimensionali di stessa grandezza. Un'animazione del processo è presente al link `http://cs231n.github.io/convolutional-networks/` (sezione Convolution Demo).

Il risultato di una convoluzione tridimensionale è comunque bidimensionale, ma solitamente non viene utilizzato un solo filtro ma una serie di filtri, e il risultato ottenuto dai vari filtri viene posto in un tensore tridimensionale di output la cui profondita è pari al numero di filtri. Ogni matrice che si ottiene dall'applicazione di un filtro viene chiamata **feature map**.

Tale tensore può essere poi l'input ad un'altra convoluzione, spesso dopo uno strato ReLu. I filtri della seconda convoluzione avranno profondità pari alla profondità di tale tensore, cioè pari al numero di filtri al passo precedente.

[1] Un tensore è un array multi-dimensionale

2.3.2 Strato di attivazione

Lo scopo di uno strato di attivazione è di aggiungere non-linearità alla rete, in quanto senza di essa la complessità delle funzioni di mapping esprimibili dalla rete sarebbe notevolmente ridotta. Uno strato di attivazione funziona prendendo in input un tensore e fornendo in output un tensore di pari dimensioni ottenuto applicando una determinata funzione di attivazione ad ogni elemento del tensore di input.

La funzione di attivazione più utilizzata è la funzione Rectifiled Linear Unit (ReLu) che restituisce zero per input negativi mentre preserva gli input positivi, cioè $f(x) = \max(0, x)$. Lo strato ReLu è sostituibile con altre funzioni di attivazione come sigmoidi o Leaky-Relu, introdotte in precedenza per le reti fully-connected. Al momento la funzione ReLu è quella preferita.

2.3.3 Strato di pooling

Uno strato di pooling ha lo scopo di ridurre la dimensionalità dei tensori per velocizzare l'utilizzo della rete sia in fase di training che di testing. Ciò permette di utilizzare le risorse computazionali per ottenere maggiori profondità piuttosto che avere reti "larghe", che danno pochi benefici.

Il pooling può essere **max-pooling** o **avg-pooling**. Il pooling agisce su matrici, quindi nel caso di tensori tridimensionali agisce su matrici ottenute iterando sulla dimensione dei colori o feature-map.

La forma più comune di pooling è il max-pooling 2×2, che dimezza le due dimensioni prendendo il massimo calcolato su regioni 2×2, spostandosi di due posizioni ad ogni iterazione (figura 2.4). La dimensione della finestra e la quantità dello spostamento sono due iperparametri diversi. È possibile aggiungere dei padding nei bordi anche se viene fatto meno frequentemente che per gli strati convoluzionali. L'avg-pooling opera in modo analogo, calcolando la media invece che il massimo dei valori.

Un'alternativa al pooling consiste nell'evitare di calcolare tutti i valori di output durante le convoluzioni. Ad esempio si può scartare un valore su due di output, in orizzontale e in verticale, evitandone la computazione. L'iperparametro che regola tale comportamento è chiamato **stride**. Un valore di stride pari a 1 equivale a uno strato convoluzionale classico (che calcola tutti i valori di output); impostando la stride a 2 si calcola un valore su 2 e così via.

2.3.4 Strato fully connected

Classiche architetture di reti convoluzionali hanno strati convoluzionali all'inizio della rete per terminare con strati fully-connected per il calcolo dei valori di confidenza delle classi. L'output dell'ultimo strato convoluzionale

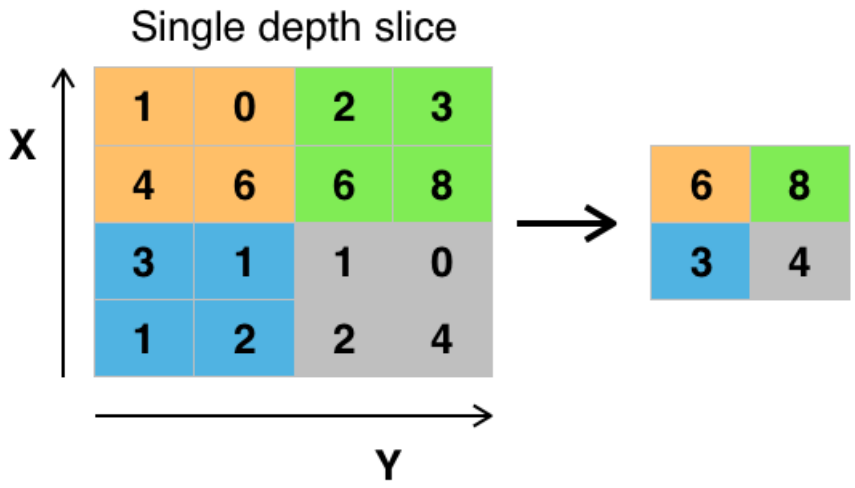

Figura 2.4: Max-pooling 2×2 senza padding
Fonte: https://en.wikipedia.org/wiki/Convolutional_neural_network

deve venir "appiattito", cioè trasformato da tridimensionale a monodimensionale, per essere utilizzato come input di uno strato FC. È comune utilizzare la funzione di attivazione ReLu per un buon training e il dropout per garantire una buona generalizzazione della rete tra uno strato FC e l'altro.

Le architetture più moderne tendono a non utilizzare molti strati fully connected, a causa dell'alto numero di parametri, che può superare quello presente in tutti gli strati convoluzionali presenti. L'architettura ResNet [He et al., 2015b] ad esempio usa un average-pooling globale (media calcolata sulle intere feature-map) applicato al risultato dell'ultima convoluzione seguito da un singolo strato fully-connected da cui si ottengono le confidenze delle classi (o valori reali se il problema è di regressione).

2.3.5 Dropout

Il **dropout** [Srivastava et al., 2014] è una forma di normalizzazione molto efficiente e semplice da implementare. Esso non sostituisce ma coopera con le altre forme di normalizzazione come la regolarizzazione L_2. Il dropout mappa un tensore di input in un tensore di output di pari dimensioni. Durante il training, un neurone (una cella del tensore) viene mantenuto attivo con probabilità p e impostato a 0 altrimenti (figura 2.5). Durante il testing, lo strato di dropout non imposta a 0 alcun valore ma moltiplica i valori di ingresso per p, simulando l'effetto del dropout.

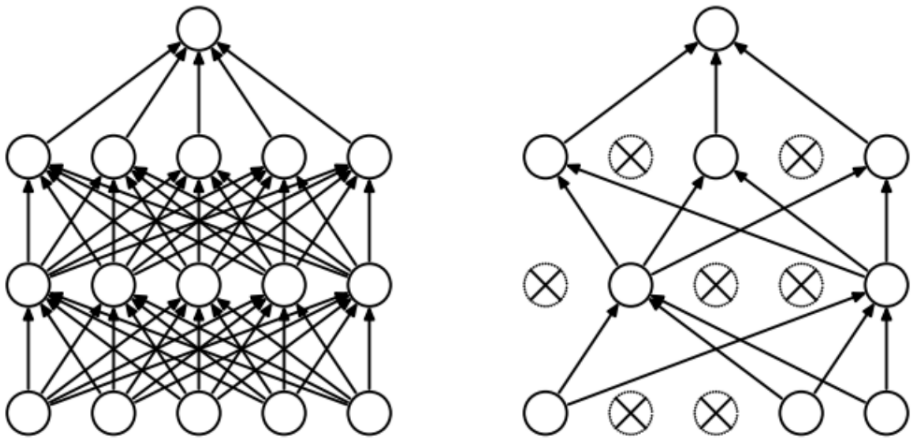

Figura 2.5: Effetto del dropout in una rete
Fonte: [Srivastava et al., 2014]

Per correttezza si dovrebbe ripetere la procedura di selezione durante il testing, ma ciò aggiungerebbe stocasticità nella rete e diventerebbe appropriato ripetere il testing in varie esecuzioni considerando la media dei risultati come predizione. Il paper [Srivastava et al., 2014] mostra come l'approssimazione con una sola esecuzione si comporti in modo simile all'esecuzione multipla con dropout.

Una variante del dropout, del tutto equivalente, consiste nel moltiplicare i valori di output per $1/p$ durante il training per evitare di compiere alcuna operazione durante il testing. Ciò è vantaggioso perché velocizza la fase di testing, che potrebbe avvenire su hardware meno performance come quello a disposizione negli smartphone. Un valore molto utilizzato per l'iperparametro p è 0.5.

Empiricamente è stato dimostrato che il dropout aumenta la generalizzazione delle reti neurali ottenendo un effetto simile a quello ottenibile con un *ensemble* di reti, cioè effettuando il training di più reti su partizioni del training set indipendentementi e fornendo la media (o la moda) degli output ottenuti dalle varie reti come risultato durante la fase di testing.

2.3.6 Local response normalization

La **Local Response Normalization** è stata introdotta in [Krizhevsky et al., 2012] in cui viene spiegato come essa aumenti leggermente la generalizzazione della rete Alexnet.

Più recentemente viene preferito utilizzare dropout e/o batch normalization [Ioffe and Szegedy, 2015] come normalizzazione, in quanto si è visto che offrono migliori performance.

La Local Response Normalization è specifica per problemi di computer vision. Denotando con $a^i_{x,y}$ l'attività di un neurone della feature-map i in posizione (x,y), la risposta normalizzata $b^i_{x,y}$ è data dalla formula

$$b^i_{x,y} = a^i_{x,y}/(k + \alpha * \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a^j_{x,y})^2)^\beta$$

L'ordine delle feature map è arbitrario e determinato prima di iniziare il training. Le costanti k, n, α e β sono iperparametri, trovati con validazione.

2.4 Architetture classiche

Creare un'architettura di rete per un problema di computer vision che abbia buone prestazioni non è facile, perché bisogna decidere quanti strati devono essere presenti nella rete e quali, collegandoli nel giusto ordine, in modo tale che il training sia sufficientemente veloce e l'accuratezza superiore ad una soglia. Solitamente si vuole che la rete entri nella memoria della GPU in modo da evitare swapping da memoria centrale o da disco durante il suo utilizzo.

Solitamente si fanno **tentativi** più o meno casuali per trovare l'architettura migliore **verificandone l'efficacia su validation set**, ma è comunque buona norma partire da un'architettura classica che solitamente è un buon punto di partenza a prescindere dal particolare dataset.

Spiegheremo alcune architetture che hanno vinto il challenge ImageNet, in particolare **AlexNet** [Krizhevsky et al., 2012], **VGG** [Simonyan and Zisserman, 2014] e **ResNet** [He et al., 2015b]. In applicazioni reali solitamente si prende un modello già addestrato su ImageNet e si esegue il training dell'ultimo strato o si continua il training dell'intera rete (fine-tuning). Questo perchè i pesi imparati su ImageNet sono abbastanza generali da essere utili per tutti i problemi che coinvolgono immagini, anche se non di categorizzazione.

2.4.1 AlexNet

L'architettura AlexNet [Krizhevsky et al., 2012] è l'architettura vincitrice della competizione ImageNet nel 2012. Tale architettura è molto simile all'architettura di LeCun del 1998 [LeCun et al., 1999] a cui è ispirata, con il cambio della funzione di attivazione da sigmoide a ReLu. Questo "dettaglio"

ha mitigato il problema del vanishing gradient al punto da permettere l'addestramento di questa rete e delle reti vincitrici della competizione negli anni successivi.

Gli strati che compongono l'architettura AlexNet sono

- Input: le immagini del dataset ImageNet sono scalate in modo tale che il lato più breve sia di 256 pixel. Poi la parte centrale viene ritagliata per avere un'immagine 256×256. Ad ogni pixel è stato sottratta la media per canale per normalizzare i dati. Il training viene fatto su ritagli casuali di grandezza 224×224.
- 11×11 conv, 96 - 4: strato convoluzionale con 96 filtri 11×11 e stride 4
- normalization: Local response normalization con parametri $k = 2$, $n = 5$, $\alpha = 10^{-4}$ e $\beta = 0.75$.
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- 5×5 conv, 256 - 1: strato convoluzionale con 256 filtri 5×5 e stride 1
- normalization: Local response normalization con parametri $k = 2$, $n = 5$, $\alpha = 10^{-4}$ e $\beta = 0.75$.
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- 3×3 conv, 384 - 1: strato convoluzionale con 384 filtri 3×3 e stride 1
- 3×3 conv, 384 - 1: strato convoluzionale con 384 filtri 3×3 e stride 1
- 3×3 conv, 256 - 1: strato convoluzionale con 256 filtri 3×3 e stride 1
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- FC 4096: strato fully-connected con 4096 unità di output
- FC 4096: strato fully-connected con 4096 unità di output
- FC 1000: strato fully-connected con 1000 unità di output, contentente la confidenza di ogni classe

Gli strati convoluzionali e FC sono seguiti da strati ReLu (tranne l'ultimo FC) e il dropout con probabilità 0.5 è applicato tra gli strati FC per migliorare la generalizzazione del modello. Il training è stato effettuato su due GPU utilizzando cuda-convnet (i framework di deep-learning sono stati rilaciati successivamente). La procedura di training utilizzata è stata **stochastic gradient descent con momentum**, con learning rate inizializzato a 0.01 e

abbassato a mano quando la funzione di loss non si abbassava più. Il momentum è stato impostato al classico valore di 0.9, il weight decay a 0.005 e il la grandezza di un batch è di 128 immagini. Per tutti i dettagli su come hanno fatto il training su due GPU si consiglia di leggere il paper [Krizhevsky et al., 2012].

Nei dati di test, hanno ottenuto un errore top-1 del 37.5% e top-5 del 17%, superando ampiamente i risultati ottenibili senza deep learning.

2.4.2 VGG

La rete VGG [Simonyan and Zisserman, 2014] ha vinto la competizione ImageNet 2014. Rispetto ad AlexNet, cresce di profondità (versioni a 16 o 19 strati contro gli 8 strati di AlexNet). Gli autori di VGG hanno notato che due strati con filtri 3×3 si comportano in modo simile ad un singolo strato con filtri 5×5, avendo però meno parametri. Per questo hanno deciso di usare solo filtri 3×3 nella rete. Di seguito la configurazione di rete con 16 strati. La rete alternativa con 19 strati è molto simile, e si rimanda al paper per la descrizione.

- Input le immagini del dataset ImageNet sono scalate a più risoluzioni (256×256 e 512×512) per aumentare gli input. La rete riceve però ritagli casuali di grandezza 224×224.
- 3×3 conv, 64 - 2: strato convoluzionale con 64 filtri 3×3 e stride 2
- 3×3 conv, 64 - 2: strato convoluzionale con 64 filtri 3×3 e stride 2
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- 3×3 conv, 128 - 2: strato convoluzionale con 128 filtri 3×3 e stride 2
- 3×3 conv, 128 - 2: strato convoluzionale con 128 filtri 3×3 e stride 2
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- 3×3 conv, 256 - 2: strato convoluzionale con 256 filtri 3×3 e stride 2
- 3×3 conv, 256 - 2: strato convoluzionale con 256 filtri 3×3 e stride 2
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- 3×3 conv, 512 - 2: strato convoluzionale con 512 filtri 3×3 e stride 2
- 3×3 conv, 512 - 2: strato convoluzionale con 512 filtri 3×3 e stride 2
- 3×3 pool - 2: max-pooling fatto con finestra 3x3 con stride 2

- FC 4096: strato fully-connected con 4096 unità di output
- FC 4096: strato fully-connected con 4096 unità di output
- FC 1000: strato fully-connected con 1000 unità di output, contentente la confidenza di ogni classe

Anche per VGG si è usato SGD con momentum a 0.9. Gli strati di normalizzazione sono stati rimossi, in quanto gli autori hanno notato un rallentamento del training senza un incremento di performance. Anche la rete VGG utilizza weight decay a 0.005 e dropout a 0.5 tra gli strati FC.

2.4.3 ResNet

Le reti residuali (ResNet) sono un'innovazione nel campo delle reti neurali convoluzionali vincitrice della competizione ImageNet nel 2015 [He et al., 2015b], con un errore top-5 del 3,75%. Gli autori di ResNet hanno empiricamente dimostrato che aumentare il numero di strati in una rete neurale oltre una certa soglia diminuiva l'accuratezza delle reti... ma ciò è controsenso perchè aggiungendo uno strato ad una rete basterebbe imparare un mapping di identità per mantenere le prestazioni della rete più piccola.

Per questo gli autori hanno proposto lo **strato residuale**, nel quale si svolge una convoluzione classica ma si aggiunge l'input al risultato. Se l'input e l'output sono di grandezze diverse, l'input viene trasformato con un'altra convoluzione a filtri 1×1 prima di essere sommato all'output per portarlo ad avere lo stesso numero di feature-map. La grandezza di una feature-map è invece preservata tramite padding. Un beneficio di questa tecnica è che la regolarizzazione L_2, che fa tendere i pesi verso zero, non fa più dimenticare ciò che è stato imparato in precedenza, ma semplicemente lo preserva.

Gli autori sono riusciti con successo a far convergere il training su una rete con 152 strati, ma è possibile utilizzare più strati se si hanno a disposizione le risorse computazionali necessarie.

2.5 Framework per il deep learning

I framework per il deep learning permettono di creare reti con estrema flessibilità riducendo estremamente la complessità del codice rispetto all'utilizzo di Numpy (per CPU), CUDA (per GPU) o simili librerie di computazione vettorizzata. Ciò perchè essi **automatizzano il calcolo del gradiente**, permettono l'esecuzione sia su CPU che su GPU con un semplice cambio di configurazione e a volte permettono il calcolo distribuito.

L'esecuzione su GPU viene solitamente fatta utilizzando librerie CUDA molto ottimizzate, che permettono di avere uno speed-up rispetto all'esecuzione su CPU che può superare il 50x. Per questo bisogna attualmente utilizzare GPU Nvidia, le uniche a supportare CUDA. Un'alternativa a CUDA è OpenCL, supportato su pressocchè tutte le GPU, ma è più lento e meno utilizzato, anche se ciò potrebbe cambiare nei prossimi anni.

I framework per deep learning più popolari sono Tensorflow, Caffe, Pytorch e Theano. Essi differiscono dallo scopo per cui sono stati creati e per le caratteristiche che offrono.

Tensorflow [Abadi et al., 2016] è stato sviluppato e reso open-source da Google per facilitare la ricerca e il deployment di modelli deep learning. Esso funziona attraverso la definizione di **grafi statici**, che sono successivamente riutilizzabili più volte nella fase di computazione. I grafi su cui viene effettuato il training sono serializzabili su disco e riutilizzabili senza avere a disposizione il codice che li ha generati. I nodi dei grafi su Tensorflow possono essere di tre tipi:

- **Placeholder**: sono il fermaposto per gli input, ad esempio le immagini e le label per il training. Valori effettivi vanno forniti ogni volta che si esegue una computazione sul grafo.

- **Variable**: variabili mantenute in memoria che contengono i parametri della rete, come i pesi e i bias. Solitamente si effettua il training cambiando il valore delle variabili attraverso gradienti compiuti in automatico dal framework. Dopo il training si serializza il grafo su disco per riutilizzare i pesi in un'applicazione reale. Nel caso di continuous learning, si continuerà con il training.

- **Operation**: è un operazione, come moltiplicazioni tra matrici, convoluzione o l'assegnamento di nuovi valori alle variabili, necessario in fase di training.

Utilizzare Tensorflow può essere a volte complicato, e per questo sono stati creati framework che funzionano ad un livello di astrazione più alto che permettono di semplificare il codice per usi comuni. Tra questi ci sono Keras, Sonnet, tf-slim e molti altri. Keras, che supporta anche Theano come backend, è attualmente la scelta più popolare.

Caffe [Jia et al., 2014] è un framework creato da UC Berkley e spesso permette di fare training e fine-tuning di reti neurali senza scrivere codice, almeno per gli utilizzi più classici quali il training e il fine-tuning di reti neurali convoluzionali. Funziona attraverso file di configurazione *prototxt* per la dichiarazione del grafo e degli iperparametri, permettendo l'esecuzione sia su CPU che su GPU. Lo scopo principale di Caffe è di produzione, anche se può

essere utilizzato anche per scopi di ricerca. I file di configurazione hanno il problema di essere molto lunghi per reti profonde, che possono avere oltre 100 strati, e questo può spingere a generare i file di configurazione con degli script, mentre su altri framework si userebbe un ciclo for nel codice.

PyTorch [Paszke et al., 2017], successore di Torch, è un framework orientato alla ricerca che si basa sull'idea di **grafo dinamico**. Ciò lo rende estremamente flessibile e semplice da usare, soprattutto se il grafo contiene flussi di controllo o per reti ricorrenti. Il codice è molto simile a quello ottenibile con Numpy, con la differenza che è eseguibile su GPU e permette il calcolo automatico del gradiente.

Capitolo 3

L'apprendimento continuo

3.1 Introduzione all'apprendimento continuo

Le competizioni come ImageNet o Pascal, che hanno aiutato lo sviluppo di nuovi algoritmi per la visione artificiale, sono state realizzate per modellare **ambienti statici**, cioè in cui si fa il training della rete e il testing in due momenti distinti, senza aggiornare il modello dopo la fase di training.

Sappiamo però che gli esseri umani riescono ad assumere nuove conoscenze continuamente, aggiungendole a quelle precedenti, con dimenticanze limitate. Ciò vorrebbe dire, per i problemi di classificazione, aggiungere **nuove classi** o far vedere al modello **nuovi esempi** per classi già conosciute, riuscendo successivamente a riconoscere pattern simili a quelli visti recentemente o in passato.

Semplici approcci, come quello di continuare il training di un modello di machine learning su nuovi campioni (aggiungendo una classe nell'ultimo strato se necessario), non funzionano nella pratica a causa di un problema denominato **catastrophic forgetting** [Goodfellow et al., 2013], che fa tendere i modelli a dimenticare le caratteristiche dei pattern visti in passato favorendo gli ultimi visti. L'approccio all'apprendimento continuo che continua l'addestramento su nuovi dati senza opportuni accorgimenti per evitare dimenticanze viene chiamato **naïve**.

Varie tecniche per mitigare questo problema sono state proposte negli ultimi anni. Tra loro le più performanti sono iCaRL, Learning without Forgetting [Li and Hoiem, 2016] e Elastic weight consolidation [Kirkpatrick et al., 2016].

Prima di presentare queste tecniche è bene presentare **tre scenari** di continuous learning che si possono presentare in problemi di riconoscimento di immagini. In questi scenari i dati sono presentati in **batch** successivi, presentanti diverse caratteristiche in base al tipo di problema.

- **New Instances (NI)**: dopo il training su un certo insieme di classi, i nuovi campioni appartengono ad una delle classi viste in precedenza. Il modello deve quindi consolidare la conoscenza acquisita dai nuovi pattern (si pensi, ad esempio, ad un nuovo modello di macchina o una vista da un'angulatura particolare) senza dimenticare ciò che ha imparato in precedenza (deve cioè continuare a riconoscere i modelli di macchine visti in precedenza).

- **New Classes (NC)**: i campioni appartenenti a classi diverse vengono presentati in momenti diversi. Il modello deve mantenere le performance sulle vecchie classi riuscendo però ad imparare a riconoscere le nuove.

- **New Istances and Classes (NIC)**: i nuovi pattern possono appartenere ad una nuova classe o ad una classe già vista in precedenza. Il modello deve poter imparare nuove classi consolidando la conoscenza per classi già conosciute. Questo è il caso più generale e difficile da risolvere.

Un semplice approccio per trattare i tre casi precedenti ottenendo una buona accuratezza, chiamato **cumulativo**, è quello di memorizzare tutti i dati su disco e rifare il training della rete su tutti i dati periodicamente. Infatti, questo è il modo in cui funzionano la maggior parte dei sistemi moderni. Questa soluzione è però a volte proibitiva a causa della mole di dati da memorizzare e del tempo richiesto per reiterare la procedura di training su tutti i campioni. Per questo il problema dell'apprendimento continuo sta ricevendo speciale attenzione nel mondo della ricerca. Introduciamo ora le principali strategie proposte dalla ricerca per mitigare il problema del catastrophic forgetting in reti neurali.

3.2 iCaRL

Incremental Classifier and Representation Learning (iCaRL) [Rebuffi et al., 2016] è una tecnica per la classificazione di pattern specifica per il caso New Classes (NC). La tecnica è stata progettata e testata per la classificazione di immagini, ma è abbastanza generale da poter essere applicata ad altri tipi di dato.

iCaRL impara un classificatore e una rappresentazione dei dati allo stesso tempo, distinguendolo da altri approcci che hanno prestazioni comparabili ma che si basano su una rappresentazione fissa dei dati, come CWR [Lomonaco and Maltoni, 2017].

I progettatori di iCaRL hanno definito tre proprietà per un buon algoritmo incrementale:

- deve essere addestrabile su stream di dati in cui esempi di classi diverse appaiono in momenti diversi
- deve offrire una buona accuratezza nel rilevare le classi viste fino ad ora (non soffrire di catastrophic forgetting)
- essere veloce nell'imparare nuove classi e non occupare troppa memoria, per evitare l'ovvia soluzione di memorizzare tutti i dati e ripetere l'addestramento dall'inizio

Sulla base di queste proprietà hanno sviluppato il metodo iCaRL che funziona grazie ai seguenti tre principi:

- categorizzazione in base alla media-degli-esemplari per classe
- selezione degli esemplari in modo da creare una lista prioritizzata (i primi esemplari sono i più importanti)
- imparare la rappresentazione dei pattern usando la distillazione della conoscenza e consolidazione degli esemplari

Gli **esemplari** sono delle immagini selezionate dinamicamente tra le immagini del training set. iCaRL garantisce che il numero totale delle immagini memorizzate in memoria non superi una soglia K fissa. È riportato l'algoritmo utilizzato per la classificazione.

CLASSIFY

input x //immagine da classificare
require $P = (P_1, ..., P_t)$ //classi degli esemplari
require $\varphi : X \to \mathbb{R}^D$ //da immagine a rappresentazione come vettore di feature

for y = 1,. . . ,t **do**
    $\mu_y \leftarrow \frac{1}{|P_y|} \sum_{p \in P_y} \varphi(p)$ //media degli esemplari per classe
end for
$y^* \leftarrow \text{argmin}_{y=1,...,t} \left\| \varphi(x) - \mu_y \right\|$ //prototipo più vicino

output class label y^*

Viene calcolata la rappresentazione dell'immagine (l'output del penultimo strato di una rete neurale) e viene trovata la classe con la rappresentazione più vicina. La rappresentazione di una classe è data dalla media delle rappresentazioni degli esemplari selezionati per tale classe.

iCaRL processa batch di dati contenenti nuove classi utilizzando l'algoritmo

IncrementalTrain

input $X^s, \ldots, X^t$
input K
require Θ
require $P = (P_1, \ldots, P_{s-1})$

$\Theta \leftarrow UpdateRepresentation(X^s, \ldots, X^t; P, \Theta)$
$m \leftarrow K/t$
for $y = 1,\ldots,s-1$ **do**
    $P_y \leftarrow ReduceExemplarSet(P_y, m)$
end for
for $y = s,\ldots,t$ **do**
    $P_y \leftarrow ConstructExemplarSet(X_y, m, \Theta)$
end for

$P \leftarrow (P_1, \ldots, P_t)$

UpdateRepresentation

input $X^s, \ldots, X^t$
require $P = (P_1, \ldots, P_{s-1})$
require Θ

$D \leftarrow \bigcup_{y=s,\ldots,t}\{(x,y) : x \in X^y\} \cup \bigcup_{y=1,\ldots,s-1}\{(x,y) : x \in P^y\}$

for $y = 1,\ldots,s-1$ **do**
    $q_i^y \leftarrow g_y(x_i) forall(x_i, \cdot) \in D$
end for

esegui l'addestramento della rete con la funzione loss
$l(\Theta) = -\sum_{(x_i,y_i)\in D}[\sum(\delta_{y=y_i} \log g_y(x_i) + \delta_{y\neq y_i} \log(1 - g_y(x_i)) + \sum_{y=1}^{s-1}(q_i^y \log g_y(x_i) + (1 - q_i^y) \log(1 - g_y(x_i))]$
che consiste dei termini di classificazione e distillazione

ReduceExemplarSet

input m
input $P = (p_1, \ldots, p_{|P|})$

$P \leftarrow (p_1, \ldots, p_m)$

output exemplar set P

ConstructExemplarSet

input image set $X = x_1, ..., x_n$ of class y
input m //numero di esemplari
require current feature function $\varphi : X \to \mathbb{R}^D$

$\mu \leftarrow \frac{1}{n} \sum_{x \in X} \varphi(x)$
for k = 1,. . . ,m **do**
$p_k \leftarrow argmin_{x \in X} \left\| \mu - \frac{1}{k} [\varphi(x) + \sum_{j=1}^{k-1} \varphi(p_j)] \right\|$
end for
$P \leftarrow (p_1, \ldots, p_m)$

output exemplar set P

iCaRL sfrutta una rete CNN usandola come **estrattore di feature** seguito da uno strato con tanti nodi quante sono le classi osservate fino ad ora. È quindi il penultimo strato a contenere la rappresentazione dei pattern. Anche se si potrebbe utilizzare l'ultimo strato per la classificazione, gli autori hanno notato che tale strato soffre il forgetting molto più del penultimo, che impara le nuove classi ma riesce a tenere buone performance sulle vecchie classi grazie alla distillation loss e alla classificazione per media degli esemplari. La media degli esemplari di ogni classe cambia con il cambio delle rappresentazioni garantendo sempre una buona classificazione.

Tutti i vettori usati come rappresentazione di un'immagine sono **L_2-normalizzati** e i risultati di ogni operazione sui vettori di feature, come le medie, sono rinormalizzati prima di essere utilizzati per la classificazione, anche se ciò non è stato esplicitato nello pseudocodice per semplicità di notazione.

iCarl **aggiorna la rappresentazione** dei dati in questo modo: ottiene insiemi di campioni $X_s, \ldots, X_t$ di classi $s..t$ che vengono passati alla procedura IncrementalTrain. Tale procedura chiama a sua volta la procedura UpdateRepresentation che per prima cosa memorizza gli output (le probabilità date dalle sigmoidi) delle vecchie classi per i nuovi dati e per gli esemplari memorizzati in memoria. Dopodiché effettua il training minimizzando l'errore di classificazione (cross-entropy) per le nuove classi e cercando al contempo di mantenere le probabilità precedentemente memorizzate per le vecchie classi. L'ultima parte della loss è chiamata distillation loss e venne introdotta in [Hinton et al., 2015] per cercare di imitare il comportamento di una rete neurale complessa in una rete neurale più piccola (e quindi più performante). iCaRL prende l'idea per imitare il comportamento della rete pre-addestramento nella rete post-addestramento cercando di mantenere le classificazioni ottenute nelle vecchie classi.

Dopo aver effettuato il fine-tuning della rete bisogna **aggiornare gli insiemi di esemplari** per le varie classi. Per prima cosa si riducono le grandezze degli insiemi di esemplari per le vecchie classi, scegliendo semplicemente i primi m esemplari, con $m = K/t$ (t è il numero totale di classi viste finora). Poi si scelgono m esemplari per le nuove classi, in modo tale che la media dei primi k esemplari sia sempre il più simile possibile alla media di tutte le rappresentazioni delle immagini delle nuove classi. Questa operazione è costosa in quanto si visita ogni immagine per la scelta di ogni esemplare, ma permette di rendere efficace la procedura di riduzione di esemplari.

Gli autori di iCaRL hanno dimostrato empiricamente che iCaRL riesce con successo a classificare sia pattern di classi viste recentemente che pattern di classi viste in passato. Usare una rappresentazione fissa dei dati tende a preferire le classi con cui è stata creata la rappresentazione, mentre il fine-tuning della rete senza opportune strategie per evitare il forgetting tende fortemente a classificare i pattern come facente parte delle ultime classi viste.

Usando una configurazione di rete fissa (AlexNet, VGG o qualsiasi altra), la quantità di memoria utilizzata da iCaRL è lineare rispetto al numero di classi osservate fino a questo momento. La memoria aggiuntiva richiesta da una nuova classe è infatti quella richiesta per collegare i neuroni del penultimo strato della rete a un nuovo neurone di classificazione (usato solo per la classificazione in fase di training, come spiegato in precedenza). Per utilizzare iCaRL con successo bisogna in più memorizzare almeno un esemplare per classe.

3.3 Learning without Forgetting

Il metodo **Learning without Forgetting** (LwF) [Li and Hoiem, 2016] permette di fare il fine-tuning di una rete CNN senza memorizzare alcun pattern passato ma in modo più efficace rispetto al metodo naïve. Come vedremo nella parte sperimentale della tesi, questo trade-off causa una perdita di accuratezza rispetto al metodo iCaRL.

Il metodo LwF usa i nuovi dati sia per il training delle nuove classi che come pattern senza label (unsupervised) per le vecchie classi, per le quali LwF cerca di **mantenere la vecchia predizione** presente dopo lo strato softmax o sigmoide (se la classificazione è multiclasse), cioè cerca di aggiornare la rete mantenendo inalterate le probabilità osservate per le vecchie classi nei nuovi campioni. Questo comportamento è analogo a ciò che viene svolto in [Hinton et al., 2015] e a ciò che avviene con la tecnica iCaRL, con la differenza che iCaRL usa un insieme di esemplari per rinforzare la memoria mentre LwF usa solamente nuovi campioni.

Questa caratteristica permette a LwF di alleviare il forgetting rispetto al metodo naïve e in modo inaspettato a migliorare la performance sulle nuove classi grazie al miglior estrattore di feature che riesce ad addestrare.

Formalmente si ha una CNN con parametri condivisi Θ_s (tutti gli strati tranne l'ultimo), parametri specifici per i vecchi task Θo (l'ultimo strato prima del softmax) e parametri per la classificazione del nuovo task Θ_n. L'algoritmo per il training LwF è:

LwFTraining

input Θ_s //parametri condivisi
input Θ_o //parametri vecchi task
input X_n, Y_n //dati di training per la nuova classe

$Y_o \leftarrow \text{CNN}(X_n, \Theta_s, \Theta_o)$ //output dei vecchi task
$\Theta_n \leftarrow RandomInit(|\Theta_n|)$ //inizializzazione nuovi parametri

define $\hat{Y}_o$ as $\text{CNN}(X_n, \hat{\Theta}_s, \hat{\Theta}_o)$ //output vecchio task
define $\hat{Y}*_n$ as $\text{CNN}(X_n, \hat{\Theta}_s, \hat{\Theta}_n)$ //output nuovo task

$O_s^*, O_o^*, O_n^* \leftarrow \text{argmin}_{\hat{\Theta}_s, \hat{\Theta}_o, \hat{\Theta}_n}(\lambda_o L_{old}(Y_o, \hat{Y}_o) + L_{new}(Y_n, \hat{Y}_n) + R(\hat{\Theta}_s, \hat{\Theta}_o, \hat{\Theta}_n))$

output O_s^*, O_o^*, O_n^*

Prima di tutto si memorizzano nella variabile Y_o gli output ottenuti dalla rete CNN nelle nuove immagini, sotto forma di probabilità per classe. Poi si aggiungono nuovi nodi nell'ultimo strato della rete CNN con opportuna inizializzazione casuale dei parametri per accomodare le nuove classi (non necessario se non sono presenti nuove classi). Si effettua poi il training utilizzando metodi classici come lo stochastic gradient descent. Si possono ottenere performance leggermente superiori facendo prima il training di Θ_n bloccando Θ_s e Θ_o per poi fare il training congiunto. Ciò permette di cambiare Θ_s in minor misura garantendo una performance migliore della rete sui vecchi task.

La loss sulle nuove classi è semplicemente la cross entropy

$$L_{new}(y_n, \hat{y}_n) = -y_n \log(\hat{y}_n)$$

dove y_n è il vettore one-hot delle classi (ground truth) e $\hat{y}_n$ è il vettore ottenuto dalla rete dallo strato softmax per le nuove classi.

Per le vecchie classi si vuole invece mantenere la vecchia predizione.

$$L_{old}(y_o, \hat{y}_o) = H(y'_o, \hat{y}'_o) = -\sum_{i=1}^{l} y'_o(i) \log(\hat{y}'_o(i))$$

dove l è il numero delle vecchie classi e $y'_o(i)$ e $\hat{y}'_o(i)$ sono le versioni modificate delle probabilità $y_o(i)$ e $\hat{y}_o(i)$ ricavate tramite formule

$$y'_o(i) = \frac{y_o(i)^{1/T}}{\sum_j y_o(j)^{1/T}}$$

e

$$\hat{y}'_o(i) = \frac{\hat{y}_o(i)^{1/T}}{\sum_j \hat{y}_o(j)^{1/T}}$$

Questa modifica, proposta da Hilton [Hinton et al., 2015], migliora leggermente la performance e viene consigliato il valore $T = 2$.

La regolarizzazione utilizzata è solitamente L_2 con parametro weight decay ottenuto con validazione. Il parametro λ_o invece decide quanto favorire la performance sulle vecchie classi rispetto alle nuove.

Nei nostri esperimenti, presentati nel prossimo capitolo, notiamo che LwF riesce a mitigare il forgetting rispetto ad un fine-tuning triviale, ma non riesce comunque a raggiungere la performance iCaRL a causa della diversa distribuzione di pattern di classi diverse che rende meno efficace la tecnica della distillation.

3.4 Elastic weight consolidation

Elastic weight consolidation [Kirkpatrick et al., 2016] è una tecnica sviluppata da DeepMind per limitare il catastrophic forgetting. Essa è particolarmente interessante perchè fornisce un criterio formale per **rallentare selettivamente il training** sui pesi che maggiormente partecipano alla classificazione dei vecchi task.

La tecnica è ispirata al modo in cui si pensa che i mammiferi riescano a mantenere le informazioni in memoria. Quando un topo acquisisce una nuova abilità, alcune delle sinapsi eccitate vengono rafforzate, comportando l'incremento di volume di alcune spine dentritiche dei neuroni. Queste spine dentritiche riescono quindi a persistere dopo aver imparato a svolgere altri compiti successivamente, mantenendo le capacità acquisite per mesi [Yang et al., 2009]. Quando le spine vengono eliminate selettivamente, le abilità vengono perse. C'è quindi evidenza che le abilità nei mammiferi siano acquisite e preservate rendendo meno elastiche una porzione delle sinapsi che contribuiscono allo svolgimento del compito.

La tecnica Ewc vede una rete neurale come una sequenza di strati con proiezioni lineari seguiti da funzioni di attivazioni non-lineari. La procedura di learning consiste nell'imparare parametri Θ che massimizzano la performance su un determinato task. Viene mostrato in [Hecht-Nielsen, 1989] e [Sussmann,

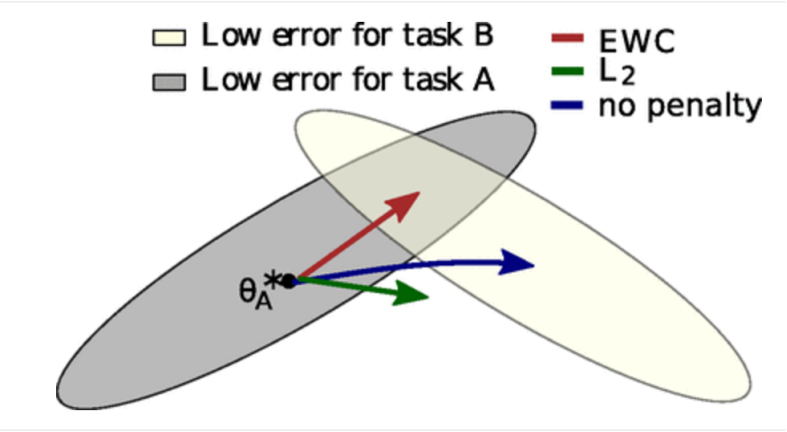

Figura 3.1: Elastic weight consolidation confrontato con penalità L_2 o senza penalità
Fonte: [Kirkpatrick et al., 2016]

1992] che molte configurazioni di Θ risultano nella stessa performance della rete. Ewc sfrutta questa proprietà cercando una soluzione per il task B, Θ^*_B vicina alla soluzione precedentemente trovata per il task A, Θ^*_A (figura 3.1). Mentre impara il task B, Ewc cerca di mantenere la performance sul task A facendo tendere i parametri verso Θ^*_A, con una penalità quadratica. Ogni parametro ha una sua costante che indica quanto il parametro sia importante per mantenere buona performance sul task A. Sul paper viene mostrato come usare la stessa costante per ogni parametro riduca sostanzialmente la performance della tecnica.

Matematicamente quando si fa il training sul task B si minimizza la formula

$$L(\Theta) = L_B(\Theta) + \sum_i \frac{\lambda}{2} F_i(\Theta_i - \Theta^*_{A,i})^2$$

dove $L_B(\Theta)$ è la loss per il task B preso singolarmente, λ indica l'importanza della performance sul vecchio task rispetto ai nuovi task e F_i è la diagonale della matrice di Fisher, utilizzata per dare maggior importanza a preservare alcuni parametri.

Si può continuare con il training in nuovi task aggiungendo componenti prior alla loss.

La tecnica è stata utilizzata con successo sul dataset MNIST e sul reinforcement learning su giochi Atari, mostrando come i pesi degli strati più bassi vengano condivisi come accade con il training cumulativo.

3.5 CopyWeights with Re-init

La tecnica **CopyWeights with Re-init** (CWR) è stata presentata in [Lomonaco and Maltoni, 2017] come baseline per verificare il funzionamento di effettive tecniche per mitigare il catastrophic forgetting. CWR non fa propriamente parte di tale categoria perchè non effettua il fine-tuning della rete.

La tecnica parte da una rete pre-inizializzata (ad esempio su ImageNet) e scarta gli strati fully-connected finali. I neuroni dell'ultimo strati, che può essere ReLu, pool, ecc., vengono collegati ad un numero di neuroni pari al numero di classi nel problema, in maniera fully-connected. Durante il training si mantengono due insiemi di pesi separati per l'ultimo strato, **cw** e **tw** di pari dimensioni. **cw** contiene i pesi consolidati mentre **tw** i pesi temporanei, utilizzati per il training. cw è inizialmente inizializzato a 0.

All'arrivo di un batch, si inizializza **tw** con pesi casuali, utilizzando l'inizializzazione più appropriata, e si svolge il training. I pesi delle classi corrispondenti alle classi del batch vengono successivamente copiati in **cw**, che funziona come memoria a lungo termine, in modo analogo alla corteccia del nostro cervello [R Preston and Eichenbaum, 2013]. È bene notare che **tw** è sempre completamente reinizializzato per il corretto funzionamento della tecnica. Se non lo si reinizializzasse, ogni nuova classe cercherebbe di "sovrastare" le classi imparate in precedenza in quanto i nuovi pattern fanno parte solamente delle nuove classi.

Malgrado la sua semplicità, vedremo nel prossimo capitolo che questa tecnica è molto competitiva oltre che efficiente.

3.6 Il dataset CORe50

Il dataset **CORe50** [Lomonaco and Maltoni, 2017] è stato creato perché, malgrado il problema del catastrophic forgetting sia di primaria importanza, pochi dataset sono presenti per valutare le prestazioni delle varie tecniche. Tra le alternative a CORe50 più degne di nota possiamo menzionare iCubWorld-Transf[1] e YouTube-8M [Abu-El-Haija et al., 2016]. iCubWorld-Transf è molto simile a CORe50 in quanto è stato sviluppato per l'utilizzo in ambiente di robotica. CORe50 presenta però un numero di sessioni più elevato e di lunghezza maggiore, rendendolo leggermente più complesso anche grazie degli sfondi più complessi. YouTube-8M presenta invece un numero molto elevato di video registrati in ambienti naturali ed eterogenei. La luce, la distanza e le pose non

[1] https://robotology.github.io/iCubWorld/

Figura 3.2: I 50 oggetti presenti nel dataset CORe50. Ogni colonna corrisponde ad una categoria.
Fonte: https://vlomonaco.github.io/core50/

sono controllate e gli autori di CORe50 lo hanno per questo ritenuto troppo complesso per le attuali tecniche di apprendimento continuo.

Il dataset CORe50 è stato sviluppato appositamente per il riconoscimento di oggetti e **supporta i tre scenari NC, NI e NIC**.

I normali dataset come ImageNet o Pascal potrebbero essere suddivisi in batch con una logica dipendente dalla tipologia di learning che si vuole testare (NC, NI, NIC) ed utilizzati per il testing di tecniche di apprendimento continuo, ma gli oggetti di una classe sono eterogenei, mentre nel dataset CORe50 l'autore ha ripreso gli stessi oggetti in sessioni diverse, le quali variano per sfondo, illuminazione, posa, occlusione, ecc.. Il fatto che gli stessi oggetti sono ripresi in sessioni temporali coerenti, cioè in cui gli oggetti si muovono lentamente davanti a una telecamera, è stato ritenuto cruciale per testare tecniche di apprendimento incrementale.

Ciò permette al dataset di simulare in modo migliore un ambiente reale, in cui si visualizzano ripetutamente gli stessi oggetti, rendendolo un migliore dataset per alcune applicazioni come la robotica.

CORe50 è una collezione di **50 oggetti domestici appartenenti a 10 categorie**: adattatori, telefoni cellulari, forbici, lampadine, lattine, bicchieri, palline, pennarelli, tazze e telecomandi (figura 3.2). La classificazione può essere fatta a livello di oggetto (50 oggetti) o a livello di categoria (10 oggetti). Il primo modo, quello di default, è più difficile in quanto diversi oggetti possono essere molto simili e difficilmente distinguibili a causa di prospettive e occlusione. Il dataset presenta immagini raccolte in **11 sessioni**, 8 indoor e 3 outdoor, con diverso sfondo e illuminazione. Per ogni sessione e per ogni

oggetto è registrata una sequenza video di circa 300 frames RGB-D 128x128, che corrispondono a circa 15 secondi. Il fatto che i frame sono RGB-D significa che è presente anche informazione sulla profondità di ogni pixel, anche se tale informazione è leggermente rumorosa e richiederebbe del preprocessing per utilizzarla (non utilizzeremo tale informazione nei nostri test).

Gli oggetti vengono ruotati da una mano, che provoca l'**occlusione parziale degli oggetti**. A volte l'oggetto può andare leggermente fuori dal frame, ma ciò è un comportamento voluto in quanto simula ciò che succede nel mondo reale.

Tra le 11 sessioni, 8 sono state scelte per i batch di training e 3 sono state scelte dagli autori per il testing, cercando di bilanciare la complessità dei due insiemi.

Capitolo 4

Sperimentazione di iCaRL su dataset CORe50

Il **continuos learning** di dati multi-dimensionali è uno dei problemi di ricerca più interessanti. A causa del catastrophic forgetting che si ha utilizzando la strategia di addestramento naïve, quasi tutti i sistemi di machine learning reali funzionano ripetendo l'addestramento su tutti i dati iterativamente, cioè ripetendo la procedura di addestramento su tutti i campioni disponibili finché non si è soddisfatti dell'accuratezza ottenuta, verificata su un insieme di campioni di test separato. Tale approccio viene chiamato cumulativo. Quando iterare la procedura di addestramento su tutti i campioni non è possibile a causa della vastità dei dati, l'utilizzo della strategia naïve provoca la perdita di gran parte dell'accuratezza che si può ottenere con l'approccio cumulativo.

Ciò succede perché all'arrivo di un nuovo batch di dati, le caratteristiche dei pattern imparate nei precedenti batch tendono ad essere dimenticate effettuando il training sui nuovi pattern. Questo fenomeno è chiamato **catastrophic forgetting** e lo scopo di questo capitolo è di confrontare le varie tecniche che sono state ideate negli ultimi anni per mitigare questo problema.

Le applicazioni reali che possono essere migliorate con un apprendimento continuo efficace sono disparate; esempi sono:

- Rilevamento spam: applicazioni come gmail o hotmail ricevono un enorme stream di dati da classificare come spam o non-spam, e ri-effettuare il training su tutti i dati è pressoché impossibile.

- Robotica: i robot devono ricordare gli oggetti visti in precedenza, malgrado la loro memoria limitata che rende impossibile memorizzare tutti i frame visti in passato

- Pubblicità: i sistemi di raccomandazione come quelli utilizzati da Adsense potrebbero ottenere uno stream di dati (es. click) ricordando accuratamente le preferenze degli utenti.

Il lavoro sperimentale della tesi è la valutazione della strategia iCaRL su dataset CORe50. Le altre strategie - Learning without Forgetting, Elastic Weight Consolidation, naïve e CWR - sono state testate da Vincenzo Lomonaco e Davide Maltoni[1] sullo stesso dataset e ciò ci permetterà di svolgere un confronto completo delle strategie.

I motivi per cui è stato scelto di utilizzare il dateset CORe50 sono che il dataset è stato specificatamente progettato per problemi di Continuos object recognition, con esecuzioni standard che permettono di condividere e confrontare facilmente le prestazioni ottenute da altri autori. È stato ritenuto importante il fatto che il dataset fornisca immagini temporalmente coerenti degli stessi oggetti, le quali rendono il dataset più appropriato per l'apprendimento continuo.

Anche se il dataset supporta i tre scenari di continuos learning NI, NC e NIC, il nostro lavoro si concentra sul caso NC in quanto è l'unico supportato dalla strategia iCaRL.

I test sono stati svolti reti CaffeNet e VGG modificate per funzionare con input 128×128, come spiegate nelle successive sezioni. Ogni valore di accuratezza riportato è calcolato su tutti i dati di test presenti nel dataset come suggerito nel paper di introduzione al dataset [Lomonaco and Maltoni, 2017], senza escludere le immagini di classi non ancora incontrate nei batch di training. Ciò permette una visualizzazione più semplice delle curve per il confronto delle varie tecniche.

Il codice per ripetere gli esperimenti è disponibile online su github[2].

4.1 Implementazione delle reti e delle strategie

La strategia iCaRL è stata implementata originariamente dagli autori della strategia utilizzando il framework Tensorflow [Rebuffi et al., 2016]. Il codice fornito dagli autori funziona effettuando il training di una rete ResNet con 18 strati inizializzata casualmente ed utilizzando come dataset ImageNet, adattato per l'addestramento nel scenario NC (New Classes). I nostri esperimenti si sono invece svolti in maniera diversa, partendo da reti pre-addestrate su ImageNet (reti pre-addestrate sono facilmente reperibili in rete) ed effettuando il

[1] https://vlomonaco.github.io/core50/

[2] https://github.com/lorenzogatto/iCaRL

fine-tuning di tali reti per il dataset CORe50. Gli strati finali fully-connected sono re-inizializzati e addestrati da zero in quanto le loro dimensioni sono state ridotte rispetto alle reti originali, mentre i pesi degli strati convoluzionali sono compatibili in dimensione e pre-inizializzati.

La pre-inizializzazione in un dataset di grandi dimensioni permette di raggiungere livelli di accuratezza più elevati rispetto ad un addestramento da zero, grazie alla generalità dei filtri imparabili su ImageNet.

Le altre strategie (cumulativa, naïve, ecc.) sono state implementate da Vincenzo Lomonaco e Davide Maltoni utilizzando il framework Caffe. Dato che lo scopo della tesi è stato di testare le stesse architetture di rete sulle diverse strategie, è stato opportuno verificare che le reti, implementate in framework differenti, fossero effettivamente equivalenti. Per trasformare le implementazioni per il framework Caffe in implementazioni per il framework Tensorflow è stato utilizzato il tool caffe-tensorflow[3]. Piccoli accorgimenti sono stati fatti nel risultato ottenuto dal tool in quanto esso non supporta livelli di padding custom, come un padding di 2 su uno strato convoluzionale con filtri 3×3, in quanto Tensorflow supporta di default 0 padding o il padding *SAME* che mantiene le stesse dimensioni dell'input (1 in questo caso). Tensorflow fornisce comunque la possibilità di aggiungere del padding aggiungendo uno strato apposito che è stato utilizzato per rendere le implementazioni per Caffe e Tensorflow equivalenti.

Le grandezze dei tensori in output da ogni strato sono state verificate per le due reti implementate e non sono state notate differenze tra i due framework. Abbiamo inoltre provato le reti con entrambi i framework nello scenario cumulativo e i livelli di accuratezza ottenuti sono stati molto simili. È possibile che ci siano delle piccole differenze, ad esempio, nel modo in cui i due framework scelgono come effettuare il pooling con stride 2 e finestre 3×3 quando la grandezza dell'input non è tale che le dimensioni modulo 2 sia 1 (in tal caso si sarebbe solo un modo sensato di fare tale pooling). Malgrado ci possano essere delle lievi differenze tra i due framework che non abbiamo esaminato, pensiamo che tali differenze non siano tali da poter influenzare in maniera consistente i risultati ottenuti.

4.2 Configurazione del server

La configurazione del server utilizzata non è importante per poter replicare i risultati della tesi, basati sull'accuratezza delle predizioni. Usare una GPU di ultima generazione è invece importante per poter accelerare la computazione che richiedebbe giorni o settimane anche su una CPU moderna.

[3] https://github.com/ethereon/caffe-tensorflow

La configurazione del server utilizzata per eseguire i test è:

- GPU Nvidia GeForce GTX Titan con 12GB RAM
- CPU Intel Xeon CPU E5-1650 v3 @ 3.50GHz 6 core, 12 threads
- RAM 32 GB di RAM
- Hard disk hard disk meccanico da 3 TB Seagate ST3000DM001-1ER1
- Sistema operativo Linux Ubuntu 14.04

4.3 Rete Mid-CaffeNet

La rete CaffeNet è nata per un errore commesso durante l'implementazione di AlexNet per il framework Caffe, per il quale gli strati di pooling e di normalizzazione sono stati invertiti rispetto alla rete di riferimento. Le due reti hanno pressoché le stesse prestazioni computazionali e presentano livelli di accuratezza molto simili. Per completezza e per mostrare le modifiche da noi apportate per far funzionare la rete con input 128×128 e con 50 classi, riportiamo tutti gli strati della rete.

Gli strati che compongono l'architettura CaffeNet da noi utilizzata, che chiamiamo Mid-CaffeNet, sono

- Input: le immagini del dataset CORe50 non vengono scalate o ritagliate come consono per le il dataset ImageNet. Viene sottratta la media per canale per normalizzare i dati, rispettivamente di 123, 117 e 104 per i canali Red, Green e Blue. L'ingresso avrà dimensione $batch\_size \times 128 \times 128 \times 3$.
- 11×11 conv, 96 - 4: strato convoluzionale con 96 filtri 11×11 e stride 4
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- normalization: Local response normalization[4] con parametri $local\_size = 5$, $\alpha = 10^{-4}$ e $\beta = 0.75$
- 5×5 conv, 256 - 1: strato convoluzionale con 256 filtri 5×5 e stride 1
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- normalization: Local response normalization con parametri $local\_size = 5$, $\alpha = 10^{-4}$ e $\beta = 0.75$

[4]Come definita in `http://caffe.berkeleyvision.org/tutorial/layers/lrn.html`

Strategia	**Accuratezza media**	**Std. dev**	**Immagini memorizzate**
Cumulativa	64,65%	1,04%	O(#images)
iCaRL	43,62%	0,66%	O(#classi)
CWR	42,32%	1,09%	O(1)
LwF*	27,60%	1,70%	O(1)
EWC*	26,22%	1,18%	O(1)
Naïve	10,75%	0,84%	O(1)

Tabella 4.1: Risultati aggregati di Mid-CaffeNet in varie strategie
*risultati non pubblicati[5]

- 3×3 conv, 384 - 1: strato convoluzionale con 384 filtri 3×3 e stride 1
- 3×3 conv, 384 - 1: strato convoluzionale con 384 filtri 3x3 e stride 1
- 3×3 conv, 256 - 1: strato convoluzionale con 256 filtri 3×3 e stride 1
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- FC 2048: strato fully-connected con 2048 unità di output
- FC 2048: strato fully-connected con 2048 unità di output
- FC 50: strato fully-connected con 50 unità di output, contentente la confidenza di ogni classe

Si noti come gli ultimi strati siano stati ridotti di grandezza rispetto alla rete CaffeNet standard e che continuano ad essere presenti strati di dropout tra gli strati FC con parametro keep_prob=0.5.

La rete è stata testata sullo scenario NC su dataset CORe50 con le strategie iCaRL, CWR, LwF, naïve e EWC. Le 10 esecuzioni sono standard e fornite con il dataset.

I risultati ottenuti da tutte le strategie sono riportati in tabella 4.1 e graficamente in figura 4.1. La stragegia cumulativa, cioè quella che memorizza tutti i dati e che all'arrivo di pattern di nuove classi ripete il training su tutti i dati, è ovviamente la più accurata. Essa però ha un impatto sulla memoria e sul tempo di training molto più elevato delle altre strategie.

La strategia iCaRL si comporta piuttosto bene: riesce a battere le strategie che non memorizzano pattern, anche se di poco. L'impatto sulla memoria e sul tempo di training sono accettabili per utilizzare la strategia in sistemi reali. L'impatto sulla memoria è stato riportato come proporzionale al numero di classi perchè bisogna memorizzare almeno un immagine per classe. Non siamo

[5]https://vlomonaco.github.io/core50/leaderboard

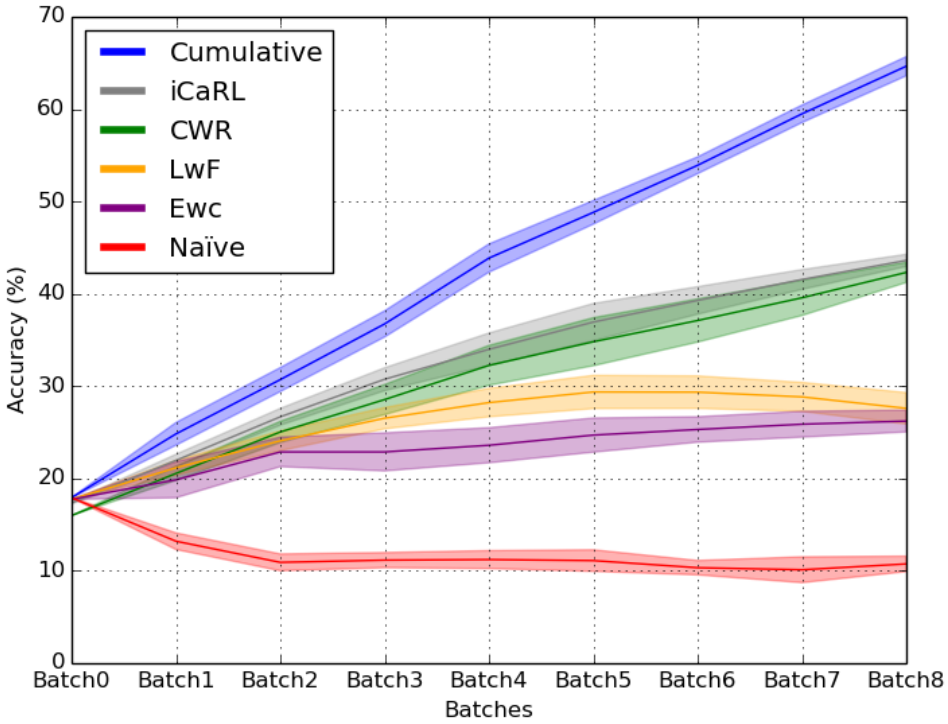

Figura 4.1: Confronto dell'accuratezza ottenuta dalle varie strategie con rete Mid-CaffeNet dopo il training su ciascuno dei batch. L'area intorno una linea indica il minimo e massimo valore di accuratezza ottenuti tra le 10 run.

consapevoli di studi che illustrano quante immagini sia necessario memorizzare per ogni classe per un numero di classi asintotico (tendente all'infinito). Dai nostri esperimenti possiamo consigliare l'utilizzo di almeno 10 pattern per classe (tabella 4.2). La tabella 4.3 mostra l'accuratezza ottenuta nei dati di test dopo ogni batch di dati per la strategia iCaRL, per ognuna delle 10 esecuzioni predefinite.

La strategia CWR, malgrado la sua semplicità, ha una performance che la rende molto interessante. Questa strategia garantisce un'accuratezza competitiva con la strategia iCaRL, malgrado il minor impatto sulla memoria e sul tempo di esecuzione (la selezione degli esemplari in iCaRL è un'operazione onerosa). Per problemi pratici in cui la strategia cumulativa non sia possibile a causa delle sue scarse prestazioni, suggeriamo di provare l'utilizzo di iCaRL e della strategia CWR, dopo aver verificato l'eventuale presenza di nuove strategie più efficaci.

Le strategie LwF e Ewc, malgrado la loro fondazione teorica, non sono

Batch	**10**	**25**	**50**	**100**
Batch0	17.45%	17.38%	17.39%	17.39%
Batch1	20.51%	21.62%	22.28%	22.63%
Batch2	23.62%	25.29%	26.32%	26.93%
Batch3	27.23%	29.28%	30.97%	31.17%
Batch4	29.60%	32.47%	34.08%	34.75%
Batch5	31.28%	34.35%	36.20%	36.87%
Batch6	33.02%	36.60%	38.73%	39.78%
Batch7	34.93%	38.71%	41.30%	42.72%
Batch8	37.09%	40.68%	43.55%	45.50%

Tabella 4.2: Accuratezza media calcolata sulle prime tre run calcolata variando la quantità di immagini per classe memorizzate

RunID	**Batch0**	**Batch1**	**Batch2**	**Batch3**	**Batch4**	**Batch5**	**Batch6**	**Batch7**	**Batch8**
0	17.41%	22.37%	26.17%	31.53%	32.41%	34.62%	37.58%	40.92%	43.94%
1	17.46%	22.36%	26.40%	29.49%	33.11%	35.21%	38.53%	41.61%	43.00%
2	17.29%	22.10%	26.39%	31.90%	36.71%	38.77%	40.08%	41.37%	43.71%
3	17.37%	21.54%	24.66%	29.38%	33.82%	36.97%	40.63%	44.29%	45.08%
4	17.21%	23.03%	27.33%	29.84%	33.12%	37.25%	38.91%	41.02%	43.29%
5	17.17%	20.83%	26.78%	31.17%	35.66%	38.87%	40.71%	41.77%	44.22%
6	17.29%	22.52%	28.43%	31.23%	33.33%	36.31%	38.32%	41.33%	43.92%
7	17.56%	22.21%	26.95%	33.15%	36.28%	40.08%	41.41%	42.24%	42.81%
8	17.52%	20.98%	27.13%	30.77%	34.71%	38.07%	40.15%	40.89%	43.19%
9	17.31%	21.62%	26.56%	29.02%	30.73%	33.28%	36.48%	40.03%	43.03%
avg	**17.36%**	**21.96%**	**26.68%**	**30.75%**	**33.99%**	**36.94%**	**39.28%**	**41.55%**	**43.62%**
dev.std	**0.12%**	**0.66%**	**0.91%**	**1.24%**	**1.76%**	**2.01%**	**1.49%**	**1.07%**	**0.66%**

Tabella 4.3: Accuratezza dettagliata ottenuta dopo ogni batch delle run con strategia iCaRL su rete Mid-CaffeNet. L'accuratezza riportata è cumulativa, cioè è sempre calcolata su tutte le immagini che compongono il test set.

invece riuscite ad ottenere risultati soddisfacenti nei nostri esperimenti: riescono ad alleviare il forgetting, ma non in maniera sufficiente. Si nota come la strategia naïve non sia adatta all'apprendimento continuo.

4.3.1 Parametri utilizzati su strategia iCaRL

Sono stati mantenuti parametri differenti per il primo batch e gli altri batch. I parametri utilizzati sono:

- lr_1: learning rate per il primo batch di immagini pari a 0.08
- lr_o: learning rate per gli altri batch pari a 0.04
- stored_images: immagini memorizzate per classe pari a 50 (tranne quando specificato diversamente) per un totale di 2500 immagini

- epoche: sono state utilizzate 5 epoche per l'addestramento in ogni batch
- momentum: è stato utilizzato lo stochastic gradient descent con momentum posto a 0.9
- weight_decay: il weight decay (la costante della regolarizzazione L_2) è stato posto a 0.0005

4.4 Rete Mid-VGG

La rete Mid-VGG è stata ottenuta dalla rete VGG-CNN-M introdotta in [Chatfield et al., 2014] a cui sono state apportate modifiche per farla funzionare su input 128×128. Dato che la rete ha subito alcune modifiche la riportiamo per completezza:

- Input: le immagini del dataset CORe50 non vengono scalate o ritagliate come consono per le il dataset ImageNet. Viene sottratta la media per canale per normalizzare i dati, rispettivamente di 123, 117 e 104 per i canali Red, Green e Blue. L'ingresso avrà dimensione $batch\_size \times 128 \times 128 \times 3$.
- 7×7 conv, 96 - 2: strato convoluzionale con 96 filtri 7×7 e stride 2
- normalization: Local response normalization[6] con parametri $local\_size = 5$, $\alpha = 5 * 10^{-4}$, $\beta = 0.75$ e $k = 2$.
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- 5×5 conv, 256 - 2: strato convoluzionale con 256 filtri 5×5 e stride 2 (padding di 1 invece che 2)
- normalization: Local response normalization con parametri $local\_size = 5$, $\alpha = 5 * 10^{-4}$, $\beta = 0.75$ e $k = 2$.
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2
- 3×3 conv, 512 - 1: strato convoluzionale con 512 filtri 3×3 e stride 1
- 3×3 conv, 512 - 1: strato convoluzionale con 512 filtri 3×3 e stride 1
- 3×3 conv, 512 - 1: strato convoluzionale con 512 filtri 3×3 e stride 1
- 3×3 pool - 2: max-pooling fatto con finestra 3×3 con stride 2

[6] Come definita in `http://caffe.berkeleyvision.org/tutorial/layers/lrn.html`

Strategia	**Accuratezza media**	**Std. dev**	**Immagini memorizzate**
Cumulativa	70.41%	0.53%	O(#images)
iCaRL	49.01%	0.63%	O(#classi)
CWR	39.14%	0.22%	O(1)
Naïve	9.34%	0.82%	O(1)

Tabella 4.4: Risultati aggregati di Mid-VGG-M in varie strategie[7]

- FC 2048: strato fully-connected con 2048 unità di output
- FC 2048: strato fully-connected con 2048 unità di output
- FC 50: strato fully-connected con 50 unità di output, contentente la confidenza di ogni classe

Si noti come gli ultimi strati siano stati ridotti di grandezza rispetto alla rete VGG standard e che continuano ad essere presenti strati di dropout tra gli strati FC con parametro keep_prob=0.5.

La rete è stata testata sullo scenario NC su dataset CORe50 solo con le strategie iCaRL, CWR e naïve, in quanto risultati con strategie LwF e Ewc non sono stati resi disponibili al momento della stesura della tesi. I risultati ottenuti dalle strategie testate sono riportati in tabella 4.4 ed in figura 4.2 e sono simili a quelli ottenuti con rete Mid-CaffeNet. La tabella 4.5 mostra l'accuratezza ottenuta dal metodo iCaRL dopo ogni batch di dati per 10 esecuzioni. Le 10 esecuzioni sono standard e fornite con il dataset.

La strategia iCaRL si comporta piuttosto bene. Riesce a battere le strategie che non memorizzano pattern, anche se di poco. Anche con la rete Mid-VGG abbiamo analizzato come il cambio del numero di immagini per classe influisca sull'accuratezza (tabella 4.6) con risultati analoghi a quelli trovati per la rete Mid-CaffeNet: ci vogliono almeno 10 immagini per classe, ed è poco utile memorizzarne più di 50.

La strategia CWR, malgrado la sua semplicità e che non faccia fine-tuning della rete, ha una performance che la rende molto competitiva, grazie al minor impatto sulla memoria e al minor tempo di esecuzione (la selezione degli esemplari in iCaRL è un'operazione onerosa), ed anche con questa rete la strategia naïve non riesce ad ottenere livelli di accuratezza soddisfacenti (9.34% contro 70.41% ottenibili con la strategia cumulativa).

[7] https://vlomonaco.github.io/core50/leaderboard

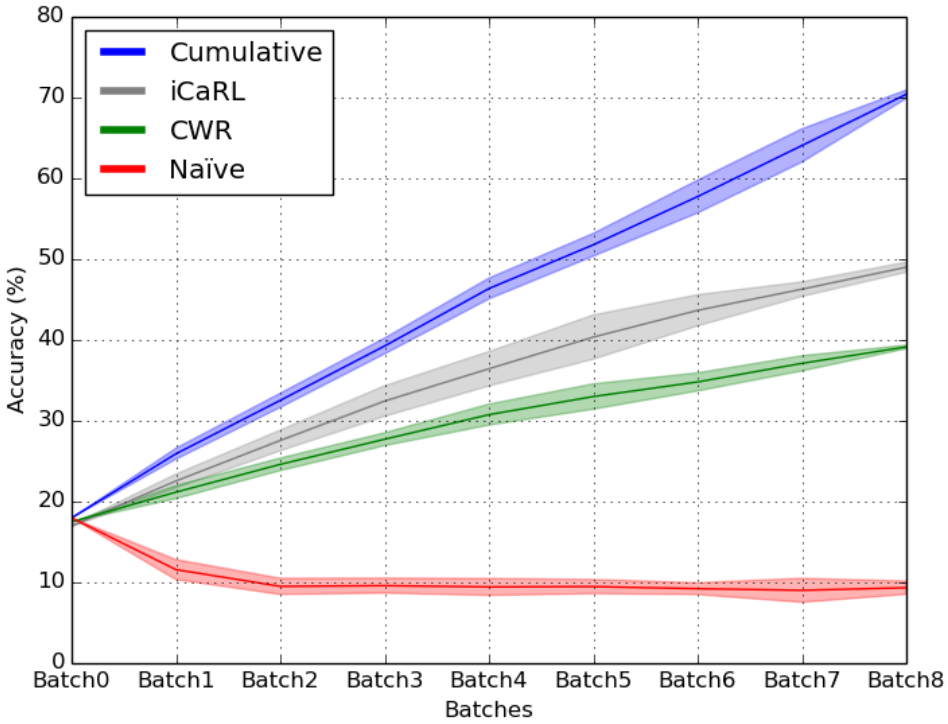

Figura 4.2: Confronto dell'accuratezza ottenuta dalle varie strategie con rete Mid-VGG dopo il training su ciascuno dei batch. L'area intorno una linea indica il minimo e massimo valore di accuratezza ottenuti tra le 10 run.

4.4.1 Parametri utilizzati su strategia iCaRL

Sono stati mantenuti parametri differenti per il primo batch e gli altri batch. I parametri utilizzati sono riportati con una breve spiegazione:

- lr_1: learning rate per il primo batch di immagini pari a 0.02
- lr_o: learning rate per gli altri batch pari a 0.05
- stored_images: immagini memorizzate per classe pari a 50 (tranne quando specificato diversamente) per un totale di 2500 immagini
- epoche: sono state utilizzate 5 epoche per l'addestramento in ogni batch
- momentum: è stato utilizzato lo stochastic gradient descent con momentum posto a 0.9

RunID	Batch0	Batch1	Batch2	Batch3	Batch4	Batch5	Batch6	Batch7	Batch8
0	17.02%	23.36%	27.92%	33.39%	35.54%	38.01%	41.34%	46.46%	48.74%
1	17.08%	22.82%	25.82%	29.27%	34.17%	37.29%	42.40%	47.09%	49.46%
2	16.87%	23.02%	28.17%	34.02%	39.46%	41.40%	44.15%	45.92%	49.47%
3	16.84%	21.50%	24.93%	30.66%	34.78%	38.47%	44.14%	47.44%	48.62%
4	17.13%	23.40%	28.20%	31.55%	35.88%	41.01%	42.09%	43.97%	48.30%
5	16.86%	21.65%	28.20%	34.01%	38.01%	42.80%	46.63%	46.97%	49.63%
6	17.09%	23.20%	29.19%	33.79%	36.38%	41.35%	44.06%	46.00%	49.86%
7	17.12%	23.57%	28.75%	34.81%	38.85%	43.86%	45.58%	46.34%	48.29%
8	16.93%	21.60%	27.96%	33.23%	38.65%	43.92%	45.81%	46.33%	48.12%
9	16.85%	21.41%	26.56%	29.64%	32.68%	35.53%	40.52%	46.34%	49.65%
avg	**16.98%**	**22.55%**	**27.57%**	**32.44%**	**36.44%**	**40.36%**	**43.67%**	**46.29%**	**49.01%**
dev.std	**0.11%**	**0.85%**	**1.28%**	**1.89%**	**2.13%**	**2.74%**	**1.93%**	**0.90%**	**0.63%**

Tabella 4.5: Accuratezza dettagliata ottenuta dopo ogni batch delle run con strategia iCaRL su rete Mid-VGG. L'accuratezza riportata è cumulativa, cioè è sempre calcolata su tutte le immagini che compongono il test set.

Batch	10	25	50	100
Batch0	16.93%	17.13%	16.99%	17.06%
Batch1	20.34%	22.29%	23.06%	23.32%
Batch2	23.71%	25.94%	27.30%	28.11%
Batch3	27.43%	30.45%	32.23%	33.36%
Batch4	30.56%	34.11%	36.39%	37.94%
Batch5	32.93%	36.96%	38.90%	40.29%
Batch6	36.14%	39.11%	42.63%	44.12%
Batch7	38.63%	42.73%	46.49%	47.69%
Batch8	41.36%	45.52%	49.22%	50.93%

Tabella 4.6: Accuratezza media su prime tre run calcolata variando la quantità di immagini per classe memorizzate

- weight_decay: il weight decay (la costante della regolarizzazione L_2) è stato posto a 0.0005

Conclusioni

Nell'ultimo capitolo sono state messe a confronto le prestazione di sei strategie sul problema dell'apprendimento continuo. È stato messo in luce come le strategie proposte dalla ricerca negli ultimi anni abbiano performance notevolmente inferiori rispetto alla strategia **cumulativa**, che riesegue il training su tutti i dati quando vengono presentati nuovi campioni.

Alcune delle tecniche presentate non memorizzano campioni passati, come Elastic weight consolidation e Learning without Forgetting. Tali tecniche hanno ottenuto performance superiori alla strategia naïve, che esegue l'addestramento solo su nuovi campioni senza alcun accorgimento, ma non si sono rivelate competitive con la tecnica CopyWeights With Re-init che non effettua il fine-tuning della rete neurale pre-inizializzata sul dataset ImageNet.

Quest'ultima tecnica è stata infatti inaspettatamente molto performante ed ha una performance molto simile a quella ottenibile con iCaRL, avendo però minor impatto sulla memoria e sul tempo di training. **iCaRL** memorizza infatti campioni selezionati tra le immagini viste in precedenza per poter rinfrescare la memoria del modello e mantenere la capacità di individuare le classi viste meno recentemente.

I risultati ottenuti dimostrano come lo stato attuale delle tecniche per l'apprendimento continuo sia ancora nella sua infanzia e possiamo aspettarci notevoli incrementi prestazionali nei prossimi anni.

Sviluppi Futuri

La performance molto elevata ottenuta con la tecnica CopyWeights With Re-init in confronto a tecniche che fanno fine-tuning della rete meriterebbe approfondimenti per capire la motivazione dietro ai risultati ottenuti. Approcci che migliorano lo stato dell'arte potrebbero infatti ispirarsi a tale tecnica per migliorarla.

Un'idea per migliorare iCaRL può essere quella di memorizzare i campioni passati a risoluzione più bassa per riuscire a memorizzare più immagini nello stesso quantitativo di memoria, verificando se ciò comporti un aumento pre-

stazionale rispetto alla versione di iCaRL standard utilizzando una quantità fissa di memoria.

Un altro modo per migliorare le prestazioni dei sistemi di apprendimento continuo può avvenire capendo in maniera più approfondita in che modo il cervello umano sia in grado di riconoscere pattern anche a distanza di anni dall'ultima volta che sono stati osservati. La strategia Ewc è un primo passo, ma l'utilizzo della back-propagation fa risultare la tecnica non biologicamente plausibile.

Esperti del settore come Geoffrey Hinton (Università di Toronto) pensano che bisogna liberarsi della backpropagation e del gradient descent perché troppo dissimili al funzionamento del cervello, ma ancora non sono state proposte soluzioni alternative che offrano performance comparabile a quelle ottenibili con le attuali tecniche.

Bibliografia

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *CoRR*, abs/1609.08675, 2016. URL `http://arxiv.org/abs/1609.08675`.

Daron Acemoglu and Pascual Restrepo. Robots and jobs: Evidence from us labor markets. 2017.

Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR*, abs/1512.02595, 2015. URL `http://arxiv.org/abs/1512.02595`.

Leon Bottou. *Stochastic Gradient Descent Tricks*, volume 7700, page 430–445. Springer, January 2012. URL `https://www.microsoft.com/en-us/research/publication/stochastic-gradient-tricks/`.

John Canny. A computational approach to edge detection. *IEEE*, PAMI-8 Issue: 6:679–698, 1986.

Chakravarty R. Alla Chaitanya, Anton Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *SIGGRAPH 2017*, 2017.

Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *CoRR*, abs/1405.3531, 2014. URL `http://arxiv.org/abs/1405.3531`.

C.-C. Chiu, T. N. Sainath, Y. Wu, R. Prabhavalkar, P. Nguyen, Z. Chen, A. Kannan, R. J. Weiss, K. Rao, K. Gonina, N. Jaitly, B. Li, J. Chorowski, and M. Bacchiani. State-of-the-art Speech Recognition With Sequence-to-Sequence Models. *ArXiv e-prints*, December 2017.

Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR 12(Jul)*, pages 2121–2159, 2011.

Mark Everingham, S. M. Ali Eslami, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111 (1):98–136, Jan 2015. ISSN 1573-1405. doi: 10.1007/s11263-014-0733-5. URL `https://doi.org/10.1007/s11263-014-0733-5`.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL `http://proceedings.mlr.press/v9/glorot10a.html`.

I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio. An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks. *ArXiv e-prints*, December 2013.

J Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 6–20, New York, NY, USA, 1986. ACM. ISBN 0-89791-193-8. doi: 10.1145/12130.12132. URL `http://doi.acm.org/10.1145/12130.12132`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015a. URL `http://arxiv.org/abs/1502.01852`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015b. URL `http://arxiv.org/abs/1512.03385`.

R. Hecht-Nielsen. Theory of the backpropagation neural network. In *International 1989 Joint Conference on Neural Networks*, pages 593–605 vol.1, 1989. doi: 10.1109/IJCNN.1989.118638.

G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. *ArXiv e-prints*, March 2015.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014. URL `http://arxiv.org/abs/1408.5093`.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`.

James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *CoRR*, abs/1612.00796, 2016. URL `http://arxiv.org/abs/1612.00796`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

N. Kruger, P. Janssen, S. Kalkan, M. Lappe, A. Leonardis, J. Piater, A. J. Rodriguez-Sanchez, and L. Wiskott. Deep hierarchies in the primate visual cortex: What can we learn for computer vision? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1847–1871, Aug 2013. ISSN 0162-8828. doi: 10.1109/TPAMI.2012.272.

Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*, pages 319–, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66722-9. URL `http://dl.acm.org/citation.cfm?id=646469.691875`.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015.

Zhizhong Li and Derek Hoiem. Learning without forgetting. *CoRR*, abs/1606.09282, 2016. URL `http://arxiv.org/abs/1606.09282`.

Vincenzo Lomonaco and Davide Maltoni. Core50: a new dataset and benchmark for continuous object recognition. *CoRR*, abs/1705.03550, 2017. URL `http://arxiv.org/abs/1705.03550`.

David G. Lowe. Distinctive image features from scale-invariant keypoints. 1999.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Michael Jones Paul Viola. Rapid object detection using a boosted cascade of simple features. 2001.

Alison R Preston and Howard Eichenbaum. Interplay of hippocampus and prefrontal cortex in memory. 23:R764–73, 09 2013.

Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, and Christoph H. Lampert. icarl: Incremental classifier and representation learning. *CoRR*, abs/1611.07725, 2016. URL `http://arxiv.org/abs/1611.07725`.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL `http://arxiv.org/abs/1409.1556`.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

Héctor J. Sussmann. Uniqueness of the weights for minimal feedforward nets with a given input-output map. *Neural Networks*, 5(4):589 – 593, 1992. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(05)80037-1. URL `http://www.sciencedirect.com/science/article/pii/S0893608005800371`.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL `http://arxiv.org/abs/1609.08144`.

Guang Yang, Feng Pan, and Wen-Biao Gan. Stably maintained dendritic spines are associated with lifelong memories. *Nature*, 462:920 EP –, Nov 2009. URL `http://dx.doi.org/10.1038/nature08577`.