Università degli Studi di Bari · Aldo Moro

Dipartimento di Informatica

Corso di Laurea in Informatica

Next-Activity Prediction tramite Reti Convoluzionali Deformabili

Tesi di Laurea in
Algoritmi e Strutture Dati

Relatore:
Chiar.mo Prof.
Nicola Di Mauro

Laureando:
Antonio Matteo Carulli

LACAM

Anno Accademico 2018/2019

Questa pagina è stata intenzionalmente lasciata in bianco.

Indice

Sommario

In una realtà sempre più data-driven, i sistemi informativi aziendali raccolgono dati sulle attività interne di un'azienda, registrando eventi e processi che rappresentano modelli dell'azienda stessa. Il machine learning e, più di recente, il deep learning sono ormai soluzioni di default per l'estrazione di informazione dai dati. La disciplina del Process Mining applica queste tecniche ai processi di un'azienda per ottimizzarli, individuando colli di bottiglia e prevedendone l'andamento. I primi approcci sui dati sequenziali di un processo aziendale prevedevano l'uso di reti neurali ricorrenti LSTM e GRU, sviluppate specificatamente per trattare serie temporali. Alcune recenti implementazioni, invece, hanno proposto l'uso di reti convoluzionali, solitamente applicate a compiti di computer vision, per ottenere risultati altrettanto soddisfacenti ad un costo computazionale ridotto. In questo lavoro di tesi si è voluto mettere a confronto una architettura convoluzionale classica con un approccio convoluzionale deformabile, sviluppato per venire incontro ai limiti delle prime, mutuato dalla computer vision e adattato al compito di process mining della next-activity prediction.

Capitolo 1

Process Mining

1.1 Cenni Storici

La prima citazione di processo è da ricercare nelle teorie dell'economista scozzese Adam Smith, il quale, nel XVIII secolo, descrisse le operazioni necessarie alla produzione di una spilla in diciotto passi, mettendo enfasi sul fatto che gli stessi, in un periodo caratterizzato da beni prodotti a mano, erano spesso effettuati dallo stesso individuo [1]. In un suo articolo, Smith descriveva come tramite la divisione organizzata del lavoro, i lavoratori della fabbrica di spille potevano arrivare a produrre oltre duecento volte il numero di spille normalmente realizzabili. [2]. Sebbene le idee di Smith sulla divisione del lavoro vennero largamente impiegate, l'integrazione dei compiti in un processo funzionale venne considerata solo nel secolo successivo con Frederick Winslow Taylor, il quale nel suo Principles of Scientific Management si concentrava sulla standardizzazione dei processi (industriali) e una chiara divisione dei ruoli sia per i dirigenti che per i lavoratori [3].

Le prime definizioni di processo aziendale si hanno nel XX secolo con Davenport [4], che lo descrive come:

> "...un insieme strutturato di attività progettate per la realizzazione di uno specifico output per soddisfare le richieste di uno specifico cliente o mercato"

Definendo poi un processo aziendale come:

> "...la struttura secondo la quale un'azienda fa ciò che è necessario per realizzare un prodotto di valore per il cliente"

Definizione estesa da Rummler & Brache due anni dopo, con la distinzione in processi operativi, il cui output è un prodotto per il cliente, e processi di supporto, il cui output è invisibile al cliente ma essenziale ai dirigenti dell'azienda [5].

In accordo alla norma ISO 9001, i processi aziendali di aziende dotate di un sistema di gestione della qualità devono essere misurabili e monitorabili nel tempo mediante l'utilizzo di indicatori di prestazione chiave [6].

Definita la propria mission, un'azienda definisce i propri processi aziendali per raggiungere gli obiettivi prefissati e creare valore.

1.2 Processo Aziendale

In economia aziendale il processo aziendale (business process) è un insieme di attività interrelate, svolte all'interno dell'azienda nell'ambito della gestione operativa delle sue funzioni aziendali, che creano valore trasformando delle risorse (input del processo) in un prodotto finale (output del processo) a valore aggiunto, destinato ad un soggetto interno o esterno all'azienda (cliente) e teso al raggiungimento di un obiettivo aziendale.

Un processo aziendale quindi, descrive *come* un prodotto o servizio deve essere realizzato, scomponendo l'atto di produzione in una serie di attività ben definite. Possiamo quindi riformulare un processo come una sequenza di attività rappresentanti un'azione svolta da un agente, sia esso umano o macchina. Tipicamente, un'azione si manifesta come un'operazione su un oggetto fisico o informativo oppure come una decisione presa da un agente coinvolto nel processo.

La rappresentazione di un processo aziendale segue solitamente la rappresentazione a diagrammi di flusso dell'informatica in cui ogni azione è rappresentata come una figura geometrica collegata ad altre azioni o operazioni di controllo. Questo permette di avere una chiara definizione della sequenza di azioni da svolgere e di conseguenza anche delle loro dipendenze. Oltre alla rappresentazione, la stessa progettazione di un processo aziendale è spesso condivisa con i tipici approcci dell'informatica. In entrambe le discipline infatti, un problema viene affrontato scomponendo il risultato finale in operazioni elementari poi riordinate per ottenere le prestazioni migliori con il miglior utilizzo delle risorse disponibili avendo sempre a mente il risultato finale [2]. [7].

1.3 Process Mining

Il termine Process Mining si riferisce all'analisi di *processi aziendali* tramite l'estrazione di conoscenza da *log di eventi* ottenuti dai moderni sistemi informativi aziendali sfruttando tecniche predittive tipiche del data mining. La nascita della disciplina è legata in parte alla crescita della mole di dati sugli eventi aziendali registrata dai sistemi informativi e in parte dal bisogno di supportare e migliorare i processi aziendali in ambienti competitivi e in rapido cambiamento.

1.3.1 Event Log

Fondamento del Process Mining è l'analisi di un event log, ovvero un registro ordinato di eventi caratterizzati da una attività associata a un particolare caso e un riferimento temporale, oltre ad eventuali altri dati a supporto. Un processo completato viene chiamato *traccia*, possiamo quindi ridefinire un event log come un insieme di tracce in cui uno specifico evento compare al più una volta in tutto il registro.

Next-Activity Prediction

Uno dei problemi di maggiore attenzione, in quest'area di ricerca è lo sviluppo di modelli predittivi i quali, una volta appresi, possono essere sfruttati per predire tracce mancanti o l'evoluzione del processo in corso basandosi sui pattern estratti da un log completato. La motivazione alla base del problema deriva dalla supposizione che analizzando processi di successo già completati e mettendoli a confronto tra loro sia possibile generare o completare un processo in esecuzione di altrettanto successo. Data una traccia in esecuzione, ovvero un processo non completato, il problema della previsione dell' attività successiva consiste quindi nel prevedere la possibile attività seguente per la traccia corrente basandosi sulle informazioni estratte dal resto del log [8].

Capitolo 2

Apprendimento Automatico

2.1 Intelligenza Artificiale

Il desiderio di creare macchine pensanti ha sfiorato spesso la mente dell'uomo. Le storie di Efesto, Pigmalione e Dedalo sono solo alcuni dei primi esempi di leggendari inventori. Al giorno d'oggi, l'Intelligenza Artificiale è una delle aree di ricerca più attive e in costante evoluzione, con numerose applicazioni pratiche. Agli albori della disciplina, i tipi di problemi risolti dai ricercatori erano relativamente facili da trattare per un computer, problemi descrivibili tramite regole matematiche formali, ma presto affiorò la difficoltà per le macchine di portare a termine compiti di facile risoluzione per gli esseri umani ma difficili da descrivere formalmente, il genere di compiti che eseguiamo automaticamente e intuitivamente, come distinguere volti in un'immagine o distinguere parole in una conversazione vocale. Uno dei motivi per cui un computer ha difficoltà ad eseguire compiti che per noi sono intuitivi deriva dal fatto che questi stessi compiti sono il frutto di una vasta conoscenza sul mondo applicata all'immensa mole di dati percepita ogni giorno da ogni persona; per agire in maniera intelligente, un computer avrebbe quindi bisogno dello stesso livello di conoscenza di un essere umano. L'ottenimento di questa conoscenza è una delle principali difficoltà dell'Intelligenza Artificiale [9].

Knowledge Base Systems

I primi approcci si basavano sulla codifica della conoscenza su un dato problema in un linguaggio formale che permetteva a un computer di effettuare ragionamenti automatici tramite regole logiche inferenziali. Le difficoltà di questo approccio, chiamato **base di conoscenza**, riguardano la dipendenza da un team di esperti in grado di codificare efficacemente la propria conoscenza in regole formali adeguate [10].

2.2 Machine Learning

I limiti dei sistemi basati su basi di conoscenza hanno suggerito la necessità di costruire sistemi intelligenti in grado di acquisire conoscenza autonomamente. Individuando pattern all'interno dei dati, un computer è così in grado di approcciare problemi che richiedono conoscenza del mondo reale e prendere decisioni solitamente ritenute soggettive; algoritmi semplici come **naive Bayes**, infatti, possono distinguere con efficacia e-mail legittime da spam. La performance di questi algoritmi di machine learning dipende fortemente dal modo in cui i dati di input sono rappresentati. Lo stesso algoritmo che distingue email di spam da email legittime, ad esempio, non sarebbe in grado di ottenere le stesse informazioni estratte dal *testo* del messaggio da una foto dello stesso, sulla falsa riga di come è facile per gli esseri umani effettuare calcoli con il sistema di numerazione arabo piuttosto che con quello romano. Questo implica che per una performance ottimale è necessaria una rappresentazione ottimale delle *caratteristiche* (**feature**) del problema; le difficoltà maggiori in questo campo sono infatti la scelta delle feature e della loro struttura.

2.2.1 Representation Learning

A venire incontro a quest'ultimo problema è il **representation learning** che si basa sull'apprendimento automatico della struttura stessa oltre che di un'associazione tra essa e l'output desiderato, permettendo un rapido adattamento del sistema a nuovi compiti minimizzando l'intervento umano. Un esempio di questi sistemi sono gli *autoencoders*, ovvero sistemi composti da due funzioni: un encoder, che converte l'input in una nuova rappresentazione e un decoder che ritrasforma l'output precedente nella forma originale; gli autoencoders sono progettati per cercare di mantenere quante più informazioni possibile tra le due fasi aggiungendo proprietà desiderate alle nuove rappresenazioni.

Idealmente in questo tipo di sistemi si cerca di definire i *fattori di variazione* che definiscono le caratteristiche dell'entità di input e di decomporli per ottenere i legami fondamentali tra essi e scartare le informazioni superflue. Questo tipo di attività, tuttavia, può essere estremamente difficile in situazioni in cui la scelta della rappresentazione è tanto complessa quanto la risoluzione del compito per un umano, come ad esempio per il riconoscimento dell'accento da una registrazione.

2.2.2 Struttura di un Algoritmo di Machine Learning

Quasi tutti gli algoritmi di deep learning possono essere descritti dai seguenti quattro elementi:

- la specifica di un dataset,
- una funzione costo,
- una procedura di ottimizzazione,
- un modello

Dataset

Un compito di machine learning è generalmente descritto in termini di *come* il sistema dovrebbe elaborare un **esempio**, ovvero un insieme di **features** ottenute dalla misurazione di un oggetto o evento e formalmente definito come un vettore $\mathbf{x} \in \mathbb{R}^n$ dove ogni x_i del vettore è una feature, o caratteristica. Un insieme di esempi forma un **dataset**.

La forma del dataset cambia a seconda del tipo di algoritmo di apprendimento utilizzato le cui due grandi categorie principali sono: apprendimento supervisionato e apprendimento non supervisionato. Negli **algoritmi di apprendimento supervisionato** il dataset è composto da esempi ai quali sono associati un'etichetta o un valore obiettivo, mentre negli **algoritmi di apprendimento non supervisionato** il dataset contiene solo gli esempi. Gli approcci supervised si basano quindi sull'osservazione di diversi esempi da un vettore casuale $\mathbf{x}$ e da un valore o vettore di valori associati $\mathbf{y}$ e da queste osservazioni imparare a predire $\mathbf{y}$ da $\mathbf{x}$. Gli approcci unsupervised, invece, si basano sull'osservazione di esempi da un vettore casuale $\mathbf{x}$ del quale si cerca, implicitamente o esplicitamente, di derivare l'intera distribuzione di probabilità $p(\mathbf{x})$ o altre proprietà desiderate. Un modo comune di rappresentare un dataset è tramite una **design matrix** che consiste in una matrice le cui righe sono gli esempi e le colonne le feature che compongono ogni esempio; per esempio, se avessimo un dataset di 150 esempi formati da 4 features ciascuno, la design matrix risultante sarebbe la matrice $\boldsymbol{X} \in \mathbb{R}^{150x4}$.

In fase di valutazione, si è interessati a quanto l'algoritmo è performante su dati che non ha mai trattato, ovvero la sua capacità di **generalizzazione**, essendo questa una diretta indicazione di come si comporterà con dati reali. Pertanto, si misura la performance di un algoritmo su un **test set** separato da quello su cui è stato addestrato il sistema, andando a definire le matrici $\boldsymbol{X}^{(train)}$ e $\boldsymbol{X}^{(test)}$. Nella maggior parte degli algoritmi di machine learning sono presenti degli iperparametri, indicatori che controllano il comportamento dell'apprendimento i cui

valori non sono adattati dall'algoritmo in sè, a volte perchè troppo difficili da ottimizzare, altre perchè non sono il tipo di parametri adatti da apprendere sul training set. Pertanto, è necessario avere un **validation set** ignorato dall'algoritmo di apprendimento e utilizzato per adattare gli iper-parametri e stimare l'errore di generalizzazione. Non volendo influenzare la scelta di nessun parametro, compresi gli iper-parametri, dal test set, l'insieme di validazione viene costruito da quello di training, tipicamente usandone l'80% per il training e il 20% per la validazione.

In problemi di classificazione, una metrica comune è l'**accuracy** che consiste nella proporzione di esempi per i quali il modello ha generato l'output corretto. Analogamente si può misurare il **tasso d'errore** tramite la **loss 0-1** che restituisce 0 in caso di classificazione corretta e 1 altrimenti. Un'altra alternativa consiste invece nel calcolare l' **Errore Quadratico Medio**, o MSE, dall'inglese Mean Squared Error, del modello sul test set. Se $\hat{\mathbf{y}}^{(test)}$ è il vettore delle classificazioni sul test set, l'errore quadratico medio è dato da

$$MSE_{test} = \frac{1}{m} \sum_i (\hat{y}^{(test)} - y^{(test)})_i^2, \tag{2.1}$$

dove m è il numero di esempi nell'input. Si nota che l'errore si annulla quando $\hat{y}^{(test)} = y^{(test)}$ e

$$MSE_{test} = \frac{1}{m} \|\hat{y}^{(test)} - y^{(test)}\|_2^2, \tag{2.2}$$

cioè l'errore è lineare rispetto alla distanza Euclidea tra le predizioni e gli obiettivi.

Funzione Costo e Ottimizzazione

La funzione costo, o funzione loss o errore, rappresenta formalmente il modo di apprendere del sistema e varia in base al tipo di task da risolvere. Generalmente, la funzione costo è una funzione parametrica derivabile lineare o non lineare. Sviluppare un algoritmo di machine learning vuol dire quindi progettare un algoritmo che individui i valori ottimali dei parametri, o pesi (weights), $\boldsymbol{w}$ della funzione costo in maniera da ridurre l'errore di generalizzazione del modello tramite l'osservazione di un training set $(\boldsymbol{X}^{(train)}, \boldsymbol{y}^{(train)})$. Un modo per farlo è quello di minimizzare l'errore sul training set, ad esempio, ponendo la sua derivata a 0. Avendo a che fare con funzioni parametriche e tipicamente scalari che si basano su più input, è necessario utilizzare delle derivate parziali per ogni parametro della funzione; il **gradiente** di una funzione generalizza questo concetto. Data una funzione $f : \mathbb{R}^n \to \mathbb{R}$, il suo gradiente è $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ dove

$$\nabla_{\boldsymbol{x}} f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}. \tag{2.3}$$

Geometricamente, la derivata di una funzione rappresenta l'inclinazione della curva del grafico della funzione. In questo modo, è possibile sfruttare il gradiente, e quindi la derivata, di una funzione parametrica come guida nella ricerca dei parametri che la riducono il più velocemente possibile. Questo è conosciuto come il **metodo della Discesa del Gradiente** [11] (MDG), tramite il quale si individua, iterativamente, il nuovo punto

$$\mathbf{x'} = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}) \tag{2.4}$$

dove ϵ è detto *tasso d'apprendimento* o **learning rate**: uno scalare positivo che determina la grandezza dello step verso il nuovo punto. La complessità legata al calcolo dello step successivo tende facilmente ad essere $O(m)$ con m numero di esempi e per dataset molto grandi, questo può comportare tempi di calcolo estremamente lunghi, pertanto, considerando che il gradiente nel MDG non è altro che una aspettativa, esso è approssimabile su un sottoinsieme di esempi. Questa estensione del MGD, chiamata **Discesa Stocastica del Gradiente**, si basa sull'utilizzare, a ogni step del metodo, una **minibatch**, un piccolo sottoinsieme, di esempi $\mathbb{B} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m')}\}$ estratti uniformemente dall'insieme di training. La dimensione della minibatch, m' è genericamente scelta come valore molto basso, da uno a poche centinaia, e rimane fissa al crescere della dimensione m del set di training.

Modello

Oltre a ridurre l'errore sul training set, tuttavia, si è interessati anche a ridurre l'errore sul test set, o **errore di generalizzazione**; ma essendo i dati osservati dall'algoritmo esclusivamente quelli del dataset di addestramento, è necessario fare delle assunzioni sui dati. I dataset di training e test sono generati dalla stessa distribuzione di probabilità e da datasets comuni chiamata **processo di generazione dei dati**. Questo processo si basa sull'assunzione che gli esempi di ogni dataset siano **indipendenti** tra loro e che training e test set siano **identicamente distribuiti** dalla stessa **distribuzione di generazione dei dati** p_{data}, permettendo lo studio delle relazioni matematiche tra gli errori di training e di test. Tenendo in considerazione i parametri $\mathbf{w}$, si nota come sia necessario sceglierli dopo aver generato il dataset di training, basandosi sui suoi valori, e solo per ultimo generare il test set; se così non fosse infatti, si avrebbe un errore di training identico all'errore di generalizzazione. Così facendo l'errore di test che si aspetta di avere è *uguale o maggiore* di quello in fase di addestramento. Questo procedimento prende il nome di **cross validation**. I fattori che influenzano quanto bene un algoritmo di machine learning sarà in grado di performare, quindi, dipendono dalla sua capacità di minimizzare l'errore d'addestramento e la sua differenza con quello di test. Quando un modello non è in grado di ottenere un errore di training sufficientemente basso si ha **underfitting**, se il modello produce una differenza tra errore di training ed

errore di generalizzazione troppo alta, invece, si ha **overfitting**. È possibile controllare il *fitting* di un modello variando la sua **capacità**, ovvero la sua abilità di assumere la forma di una vasta gamma di funzioni. Una bassa capacità può portare all'underfitting, mentre un modello con un'alta capacità potrebbe memorizzare proprietà superflue dagli esempi che influiscono negativamente sull'errore in fase di test. Un modo di controllare la capacità di un modello è quello di scegliere il suo **spazio delle ipotesi**, ovvero l'insieme delle funzioni che l'algoritmo può scegliere come soluzioni. In generale, modelli con un'alta capacità riescono a risolvere compiti anche molto complessi, irrisolvibili da modelli con una bassa capacità, ma rischiano l'overfitting se applicati a problemi più semplici.

2.2.3 Regularization

Il teorema *No Free Lunch* per il machine learning [12] sostiene che, mediando su ogni possibile distribuzione di generazione di dati, qualsiasi algoritmo di classificazione ottiene lo stesso tasso d'errore di fronte a dati che non ha mai trattato. Ciò che implica questo teorema è che non può esistere un algoritmo universalmente migliore degli altri. Tuttavia, ciò si applica solo nel caso in cui si abbia a che fare con *ogni* possibile distribuzione di generazione dei dati; pertanto, l'approccio utilizzato è quello di fare di volta in volta delle assunzioni circa il tipo di distribuzioni che riguardano uno specifico compito di machine learning, introducendo delle opportune *preferenze* nell'algoritmo di apprendimento. Esistono diversi modi di esprimere queste preferenze, sia implicitamente che esplicitamente. Nel collettivo, l'applicazione di questi approcci è detta **regolarizzazione**, ovvero ogni modifica fatta ad un algoritmo atta a ridurre il suo errore di generalizzazione lasciando invariato il suo errore di training.

Una delle forme più comuni di regolarizzazione è quella dell' **early stopping**. Quando si addestrano modelli con una capacità di rappresentazione sufficiente a raggiungere l'overfitting si osserva spesso come l'errore di training tenda a diminuire costantemente nel tempo, al contrario dell'errore di validazione. L'approccio di early stopping si basa sull'assunzione che salvare i parametri ottenuti al momento in cui l'errore di validazione è più basso permetta di ottenere un modello migliore. Ogni volta che l'errore di validazione migliora, viene salvata una copia dei parametri associati se questa è maggiore delle precedenti. L'algoritmo termina quando non si registrano miglioramenti aggiuntivi nell'errore di validazione dopo un prefissato numero di iterazioni.

2.2.4 Principio di Massima Verosimiglianza

Statisticamente la funzione costo deve essere una buona stima della reale distribuzione di probabilità degli esempi nel dataset di training. Per scegliere una buona funzione costo si utilizza comunemente il **principio della massima verosimiglianza**.

Dato $\mathbb{X} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ un set di m esempi scelti indipendentemente dalla ditribuzione di generazione di dati sconosciuta $p_{data}(\mathbf{x})$. Sia $p_{model}(\mathbf{x}; \boldsymbol{\theta})$ una famiglia parametrica di distribuzioni di probabilità sullo stesso spazio identificato da θ tale che $p_{model}(\mathbf{x}; \boldsymbol{\theta})$ associ a ogni configurazione $\mathbf{x}$ un numero reale che stima la probabilità $p_{data}(\mathbf{x})$. La stima di massima verosimiglianza per $\boldsymbol{\theta}$ è

$$\boldsymbol{\theta}_{MV} = \arg\max_{\boldsymbol{\theta}} p_{model}(\mathbb{X}; \boldsymbol{\theta}), \tag{2.5}$$

$$= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{model}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \tag{2.6}$$

Si nota che il logaritmo della verosimiglianza lascia il suo argmax invariato ma trasforma il prodotto in una somma

$$\boldsymbol{\theta}_{MV} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} log\ p_{model}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) \tag{2.7}$$

Dividendo per m, lasciando invariato l'argmax, possiamo ottenere una versione della funzione costo come stima della distribuzione empirica $\hat{p}_{data}$ definita dai dati di training

$$\boldsymbol{\theta}_{MV} = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} log\ p_{model}(\mathbf{x}; \boldsymbol{\theta}) \tag{2.8}$$

Un modo di interpretare la stima di massima verosimiglianza è quello di vederla come il minimizzare la differenza tra la distribuzione empirica $\hat{p}_{data}$, definita sul training set, e quella del modello p_{model}, calcolando la differenza tra le due distribuzioni tramite la **divergenza di Kullback-Leibler** (KL) data da

$$D_{KL}(\hat{p}_{data}\ ||\ p_{model}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}}[log\ \hat{p}_{data}(\mathbf{x}) - log\ p_{model}(\mathbf{x})] \tag{2.9}$$

Il termine a sinistra è una funzione della sola distribuzione, non del modello, quindi quando quest'ultimo viene addestrato per minimizzare la divergenza KL, sarà sufficiente minimizzare

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}}\ [log\ p_{model}(\mathbf{x})] \tag{2.10}$$

che equivale a minimizzare l'**entropia incrociata** tra le due distribuzioni, dove per entropia incrociata si intende qualsiasi funzione di loss formata da una verosimiglianza logaritmica negativa tra la distribuzione empirica estratta dal training set e la distribuzione del modello. L'errore quadratico medio, ad esempio, è l'entropia incrociata tra la distribuzione empirica e un modello Gaussiano.

2.3 Deep Learning

Gli approcci di **deep learning** vengono incontro ai limiti del representation learning scomponendo la rappresentazione finale in una serie di altre rappresentazioni meno complesse apprese in modo progressivo dai dati di input. Uno degli esempi più caratteristici è quello del **multi-layer perceptron** (MLP).

2.3.1 Multi-Layer Perceptron

Un Multi-Layer Perceptron, o Feedforward Neural Network, è un tipo di rete neurale in cui si cerca di ottimizzare la funzione composta $f(\mathbf{x}, \theta)$ che approssima una funzione obiettivo $f^*(\mathbf{x})$ tramite l'apprendimento dei parametri θ di f.

Il termine *feedforward* deriva dal fatto che la funzione composta è tipicamente della forma $f^*(\mathbf{x}) = f^{(1)}(f^{(2)}(f^{(3)}(f^{(\cdots)}(\mathbf{x}))))$ dove ogni funzione intermedia $f^{(i)}$, detta *layer*, ha come input l'output del layer precedente senza essere parametro dei successivi, cioè senza avere connessioni di ritorno (feedback) nella catena di funzioni così creata. L'aggettivo *deep* in Deep Learning deriva dalla lunghezza della catena di funzioni che compongono la funzione da ottimizzare. Il modello assume quindi la forma di un grafo direzionato aciclico che rappresenta *come* i vari layer interagiscono. All'ultimo layer, detto di *output*, vengono associati gli esempi di addestramento, ad esempio etichette nel caso di un problema di classificazione, mentre il resto dei layer, detti nascosti (*hidden*), non sono direttamente influenzati dagli esempi.

2.3.2 Funzioni d'attivazione

La funzione che produce l'output di ogni hidden layer deve necessariamente essere non lineare; uno degli approcci più comuni è quello di applicare delle trasformazioni affini all'input controllate da parametri appresi seguite poi da una funzione non lineare fissa, detta di *attivazione* (activation function). La più diffusa tra le funzioni d'attivazione è la *Rectified Linear Unit*, o **ReLU**, definita come:

$$g(z) = max(0, z) \tag{2.11}$$

La non linearità di queste funzioni porta la funzione costo ad essere non convessa. Per funzioni convesse, il processo di ottimizzazione converge a partire da qualsiasi parametro iniziale, ma per funzioni non convesse il metodo della discesa stocastica del gradiente è sensibile a questi e non garantisce sempre la convergenza; per questo motivo, si preferisce inizializzare i pesi delle reti neurali a valori casuali molto bassi. La maggior parte delle volte, si utilizza l'entropia incrociata tra la distribuzione dei dati e quella del modello come funzione costo determinandone la forma in base al modo di rappresentare l'output. Supponendo che la rete neurale produca un

set di features nascoste $\mathbf{h} = f(\mathbf{x}, \boldsymbol{\theta})$ infatti, il ruolo dell'unità di output è quello di effettuare le ultime trasformazioni sulla base delle feature $\mathbf{h}$ per completare il compito che la rete deve svolgere.

2.3.3 Unità lineari

Un'unità basata su trasformazioni affini senza nonlinearità viene comunemente chiamata unità lineare. Date le features $\boldsymbol{h}$ un layer di unità lineari produce il vettore $\hat{\boldsymbol{y}} = \boldsymbol{W}^\intercal \boldsymbol{h} + \boldsymbol{b}$. Tipicamente, questo tipo di layer sono usati per produrre la media di una distribuzione Gaussiana condizionale

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}) \tag{2.12}$$

per la quale massimizzare la verosimiglianza logaritmica equivale a minimizzare l'errore quadratico medio. Gli algoritmi di ottimizzazione basati sul gradiente non incontrano difficoltà con questo tipo di layer in quanto non saturano, cioè non incontrano mai asintoti orizzontali.

2.3.4 Funzione Softmax

Volendo rappresentare una distribuzione di probabilità su una variabile discreta con n possibili valori, come nel caso di un problema di classificazione a più classi, si può utilizzare la funzione **softmax**. Innanzitutto, si vuole generare un vettore di probabilità $\hat{\boldsymbol{y}}$ con $\hat{y}_i = P(y = i|\mathbf{x})$. Per ottenere una distribuzione valida, ogni $\hat{y}_i$ deve essere compreso tra 0 e 1 e la somma dell'intero vettore deve essere 1. Per assicurarsi di avere sempre valori che generano un buon gradiente, applichiamo un layer lineare prima della funzione d'attivazione. L'unità lineare predice le probabilità logaritmiche non normalizzate

$$\boldsymbol{z} = \boldsymbol{W}^\intercal \boldsymbol{h} + \boldsymbol{b} \tag{2.13}$$

dove $z_i = log\ \hat{P}(y = i \mid \boldsymbol{x})$. La funzione softmax esegue l'esponenziale e normalizza i valori per ottenere la $\hat{\boldsymbol{y}}$ desiderata. Formalmente

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{2.14}$$

La presenza dell'esponenziale permette alla funzione softmax di interagire bene con il metodo della massima verosimiglianza logaritmica. In questo caso, si vuole massimizzare $\log\ P(y = i|\boldsymbol{x}) = \log \text{softmax}(\boldsymbol{z})_i$, quindi è naturale esprimere la

softmax in termini dell'esponenziale dato che questo si semplifica con il logaritmo della massima verosimiglianza

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j) \tag{2.15}$$

Si nota come nel primo termine dell'equazione 2.15, l'input z_i ha sempre un contributo diretto nella funzione costo, ma ciò non intacca il processo d'apprendimento in quanto la variabile $\boldsymbol{z}$, detta **logit**, non può saturare e nemmeno nel caso in cui il secondo termine, che porta l'intero vettore $\boldsymbol{z}$ a diminuire, è molto piccolo. Analizzando ulteriormente il secondo termine $\log \sum_j \exp(z_j)$ si fa notare come esso può essere approssimato a $\max_j z_j$ dato che $\exp z_k$ è ignorabile per ogni z_k molto minore di $\max_j z_j$. Questo ci fa intuire che la funzione costo basata sulla verosimiglianza logaritmica negativa tende a penalizzare fortemente la predizione sbagliata incontrata più spesso. Se la risposta corretta fornisce già il contributo maggiore alla softmax, allora il primo termine z_i e il secondo termine $-\log \sum_j \exp(z_j)u \approx -\max_j z_j = -z_i$ tendono ad annullarsi, generando un esempio che apporterà un basso contributo al costo totale di addestramento che sarà invece dominato dagli altri esempi non ancora correttamente classificati.

2.3.5 Forward e Back-Propagation

Quando si usa una feedforward neural network per elaborare un input $\boldsymbol{x}$ e produrre un output $\hat{\boldsymbol{y}}$, il flusso di informazioni si sposta da $\boldsymbol{x}$ alle unità nascoste di ogni layer e infine produce $\hat{\boldsymbol{y}}$. Questo processo è chiamato propagazione in avanti, o **forward propagation** e durante l'addestramento può proseguire fino alla produzione di un costo scalare $J(\boldsymbol{\theta})$. La valutazione del gradiente sulla base di questo costo può essere computazionalmente pesante, l'algoritmo di **back-propagation** offre una soluzione semplice e leggera.

Regola della Catena

La regola della catena è una regola di derivazione che permette di calcolare la derivata della funzione composta di due funzioni derivabili. L'algoritmo di back-propagation sfrutta la regola della catena in uno specifico ordine di operazioni che lo rende altamente efficiente.

Dati $x \in \mathbb{R}$, $f : \mathbb{R} \to \mathbb{R}$ e $g : \mathbb{R} \to \mathbb{R}$, supponiamo $y = g(x)$ e $z = f(g(x))$. Per la regola della catena

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.16}$$

Generalizzando oltre il caso scalare, supponiamo di avere $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n$ e $f : \mathbb{R}^n \to \mathbb{R}$. Se $\boldsymbol{y} = g(\boldsymbol{x})$ e $\boldsymbol{z} = f(\boldsymbol{y})$, allora

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} \tag{2.17}$$

O analogamente, in notazione vettoriale

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \right)^{\intercal} \nabla_{\boldsymbol{y}} z \tag{2.18}$$

dove $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ è la matrice Jacobiana[1] $n \times m$ di g. Si nota quindi che il gradiente di una variabile $\boldsymbol{x}$ è dato dal prodotto di una matrice jacobiana $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ e un gradiente $\nabla_{\boldsymbol{y}} z$. L'algoritmo di back-propagation consiste nell'esecuzione di questo prodotto jacobiana-gradiente per ogni operazione nel grafo di elaborazione [12].

2.3.6 Algoritmi di ottimizzazione

Metodo del momento

Il metodo della discesa stocastica del gradiente (DSG), seppur molto popolare, tende ad essere lento in alcune occasioni. Il metodo del momento [13] viene incontro proprio a questo problema velocizzando l'apprendimento. L'algoritmo del momento memorizza una media mobile dei gradienti passati che decade esponenzialmente continuando a muoversi nella direzione dei gradienti. Ispirato alla fisica, il metodo del momento interpreta l'opposto del gradiente come una forza che sposta una particella nello spazio dei parametri seguendo le leggi di Newton. L'algoritmo introduce la variabile $\boldsymbol{v}$ rappresentante la velocità della particella che, assunta di massa unitaria, permette di interpretare la stessa velocità come il momento[2] della particella. Un iperparametro $\alpha \in [0, 1)$ determina quanto velocemente il contributo dei gradienti passati decade esponenzialmente. Formalmente la regola di aggiornamento è data da

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right) \tag{2.19}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v} \tag{2.20}$$

dove la velocità $\boldsymbol{v}$ accumula i gradienti $\nabla_{\boldsymbol{\theta}}(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}))$. Si nota come maggiore è α rispetto a ϵ, maggiore è il contributo dei gradienti passati rispetto ai nuovi alla direzione corrente. Lo pseudocodice della DSG con momento è rappresentato dall'algoritmo 1.

[1] La matrice jacobiana di una funzione è la matrice i cui elementi sono le derivate parziali prime della funzione.

[2] In fisica, il momento è dato da massa per velocità

Algorithm 1 Discesa stocastica del gradiente (DSG) con momento

Require: Tasso d'apprendimento ϵ, parametro del momento α
Require: Parametro iniziale $\boldsymbol{\theta}$, velocità iniziale $\boldsymbol{v}$
  while criterio di stop non rispettato **do**
    Seleziona una minibatch di m esempi dal training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ con i corrispondenti obiettivi $\boldsymbol{y}^{(i)}$
    Calcola la stima del gradiente: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} (\sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}))$
    Aggiorna la velocità: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Applica il passo d'aggiornamento: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

Si nota come, in precedenza, la lunghezza del passo fosse semplicemente la norma del gradiente moltiplicata per il tasso d'apprendimento, mentre ora aumenta in base alla grandezza e all'allineamento di una *sequenza* di gradienti.

Adaptive Learning Rate

Il tasso d'apprendimento (learning rate) di una rete neurale è facilmente considerabile l'iperparametro più difficile da impostare dato l'importante contributo che dà alla performance del modello. Le funzioni costo sono spesso molto sensibili ad alcune direzioni nello spazio dei parametri e insensibili ad altre. Se ipotizziamo che le direzioni di "sensitività" siano in qualche maniera allineate con gli assi dello spazio, avrebbe senso pensare di usare un tasso d'apprendimento separato per ogni parametro e adattarlo automaticamente durante la fase di apprendimento. I primi approcci, come l'algoritmo delta-bar-delta, si basavano sulla derivata parziale della loss, che richiedeva un'ottimizzazione sull'intero insieme di esempi (full batch optimazion); diversi algoritmi incrementali (mini batch-based) sono poi stati sviluppati più recentemente.

Attualmente, non è stata provata la superiorità di un metodo di ottimizzazione sull'altro, ma la famiglia di algoritmi basata sull'adattamento del tasso di apprendimento ha mostrato una maggiore robustezza rispetto agli altri [14].

Adam

Uno di questi algoritmi è **Adam** [15], che sta per Adaptive Moments. In Adam, infatti, il momento è incorporato direttamente come una stima del momento del primo ordine[3] del gradiente, includendo una correzione del bias delle stime dei mo-

[3]Il momento del primo ordine, o momento statico, è una proprietà geometrica di un oggetto rappresentante la distribuzione della massa o della forma dell'area in cui è definito, in relazione a un certo asse.

Algorithm 2 Algoritmo Adam

Require: Dimensione dello step ϵ

Require: Fattori di decadimento esponenziale per la stima dei momenti p_1 e p_2 in $[0, 1)]$

Require: Piccola costante δ usata per stabilizzazione numerica

Require: Parametri iniziali $\boldsymbol{\theta}$

Inizializza le variabili del primo e secondo momento $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$

Inizializza il passo temporale $t = 0$

while criterio di stop non rispettato **do**

Seleziona una minibatch di m esempi dal training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ con i corrispondenti obiettivi $\boldsymbol{y}^{(i)}$

Calcola il gradiente: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} (\sum_i L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}))$

$t \leftarrow t + 1$

Aggiorna la stima del primo momento: $\boldsymbol{s} \leftarrow p_1 \boldsymbol{s} + (1 - p_1)\boldsymbol{g}$

Aggiorna la stima del secondo momento: $\boldsymbol{r} \leftarrow p_2 \boldsymbol{r} + (1 - p_2)\boldsymbol{g} \odot \boldsymbol{g}$

Correggi il bias del primo momento: $\hat{\boldsymbol{s}} = \frac{\boldsymbol{s}}{1 - \boldsymbol{p}_1^t}$

Correggi il bias del secondo momento:$\hat{\boldsymbol{r}} = \frac{\boldsymbol{r}}{1 - \boldsymbol{p}_2^t}$

Calcola l'aggiornamento dei parametri: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$

Applica l'aggiornamento: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

menti del primo e del secondo[4] ordine per compensare l'inizializzazione all'origine (algoritmo 2).

[4]Il momento del secondo ordine, o momento di inerzia, è una proprietà geometrica di un corpo che ne misura l'inerzia al variare della sua velocità angolare

2.4 Convolutional Neural Networks

Una rete neurale *convoluzionale* è un tipo specializzato di rete neurale per elaborare dati di tipo griglia come immagini, interpretate come matrici (griglie 2-D) di pixel, e serie temporali, interpretate come vettori (griglie 1-D) registrando valori ad intervalli fissi. [16].

2.4.1 L'operazione di Convoluzione

Il nome deriva dall'operazione matematica su cui si basano; una **convoluzione** è un'operazione su due funzioni reali che nella sua forma più generale, applicata a serie temporali, assume la forma:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.21}$$

Dove t è un intero rappresentante un indice temporale; x e w sono funzioni definite sullo spazio di t, chiamate rispettivamente **input** e **kernel**, mentre $*$ indica l'operazione di convoluzione; l'output è generalmente detto **feature map**.

Nel machine learning, l'input assume tipicamente la forma di un array multidimensionale di dati e il kernel quella di un array multidimensionale di parametri adattati durante l'apprendimento. Questi array multidimensionali sono detti **tensori**. Dovendo memorizzarli esplicitamente in memoria, si considerano le due funzioni come sempre nulle tranne che nello spazio dei punti dei quali salviamo i valori. In pratica, questo vuol dire che possiamo considerare la sommatoria infinita come una sommatoria sul numero (finito) di elementi negli array. Ad esempio, nel caso di un'immagine bidimensionale I associata a un kernel bidimensionale K, si avrà

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.22}$$

L'operazione di convoluzione è commutativa, quindi possiamo riscrivere l'equazione precedente come

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.23}$$

La proprietà commutativa sorge dalla **riflessione** del kernel rispetto all'input, ottenendo all'aumentare di m un aumento dell'indice in input ma una diminuzione nel kernel. L'equazione 2.23 è solitamente più semplice da implementare nelle librerie di machine learning in quanto comporta una minore variazione dei valori che m e n possono assumere, la funzione più diffusa tra le varie librerie tuttavia è

la **cross-correlation**, una convoluzione senza la riflessione del kernel (equazione 2.24), formalmente

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{2.24}$$

Una convoluzione discreta può essere vista come un prodotto per una matrice molto sparsa, ovvero con la maggior parte degli elementi uguali a 0, composta da diversi elementi uguali tra loro. In una dimensione, una convoluzione discreta uniforme assume la forma di una matrice di Toeplix, per la quale ogni riga è formata dagli elementi della riga precedente spostati di una posizione. In due dimensioni assume la forma di una matrice circolante a due blocchi. Il motivo della sparsità della matrice deriva dal fatto che il kernel è generalmente molto più piccolo dell'input. Ogni algoritmo di una rete neurale che utilizza matrici che non richiedono una struttura particolare è generalmente compatibile con l'operazione di convoluzione senza ulteriori modifiche; esse inoltre danno la possibilità di lavorare con input di dimensioni variabili e godono di alcune importanti proprietà descritte nelle successive sezioni.

2.4.2 Interazioni Sparse

Tradizionalmente i layer di una rete neurale eseguono prodotti per una matrice di parametri con un parametro diverso per descrivere l'interazione di ogni unità di output con ogni untià di input. Quindi ogni unità di output interagisce con ogni unità di input. Una rete convoluzionale, invece, contiene tipicamente **interazioni sparse** tra le unità grazie alle dimensioni inferiori del kernel rispetto all'input. In questa maniera si riescono a ridurre i requisiti di memoria del modello riuscendo ad aumentarne l'efficienza statistica; elaborando un'immagine, ad esempio, le piccole dimensioni del kernel permettono di individuare dettagli piccoli ma rilevanti come i bordi delle figure utilizzando decine o centinaia di pixel rispetto ai milioni dell'intera figura. I benefici si espandono però anche al numero di operazioni necessarie per calcolare l'output. Se si hanno m input e n output, il prodotto tra matrici richiederebbe $m \times n$ parametri risultando in un tempo d'esecuzione pari a $O(m \times n)$, ma usando un kernel che limita il numero di connessioni a k, con $k \ll m$, l'approccio a interazioni sparse richiede solo $k \times n$ parametri e un tempo d'esecuzione pari a $O(k \times n)$. Le unità nei layer più profondi tendono a interagire *indirettamente* con una porzione maggiore dell'input permettendo alla rete di descrivere efficacemente interazioni complesse, tra molte variabili, ricostruendole da semplici blocchi composti solo da interazioni sparse.

2.4.3 Condivisione dei Parametri

In una rete neurale tradizionale, ogni parametro nella matrice dei pesi è usato esattamente una volta, per calcolare l'output, quando viene moltiplicato per un elemento dell'input e mai più ripreso. In una rete convoluzionale invece si usa sempre lo stesso insieme di parametri identificati dal kernel ad ogni posizione dell'input. Il costo d'esecuzione rimane invariato (è sempre $O(k \times n)$), ma ciò riduce ulteriormente lo spazio di memoria richiesto dal modello a k parametri, che, si ricorda, essendo k molto minore di m ed essendo solitamente $m \approx n$, è praticamente insignificante in confronto a $m \times n$. Queste due proprietà mostrano la maggiore efficienza dell'operazione di convoluzione rispetto al prodotto denso di matrici in termini di requisiti di memoria ed efficienza statistica.

2.4.4 Rappresentazioni Equivarianti

Per le convoluzioni, l'uso di parametri comuni rende il layer **equivariante** alla traslazione. Una funzione si dice equivariante se a una modifica dell'input equivale una modifica identica all'ouput. Formalmente, una funzione $f(x)$ è equivariante rispetto alla funzione g se vale $f(g(x)) = g(f(x))$. Nel caso delle convoluzioni, se si sceglie g come una funzione qualsiasi che trasla l'input, allora la funzione di convoluzione è equivariante rispetto a g. Questo risulta particolarmente vantaggioso quando si ha una funzione di un gruppo ristretto di punti adiacenti che estrae informazioni utili per più locazioni di input; ad esempio, l'individuazione dei bordi di un'immagine nel primo layer, dato che appaiono piò o meno ovunque nell'immagine, motiva la condivisione di quei parametri su tutta l'immagine.

2.4.5 Pooling

Un tipico layer convoluzionale consiste di tre fasi. Nella prima, vengono effettuate una serie di convoluzioni che producono un insieme di attivazioni lineari. Nella seconda fase, detta **fase di identificazione**, si applica un'attivazione non lineare a ogni attivazione precedente. Infine, nell'ultima fase, si utilizza una funzione detta di **pooling** per modificare ulteriormente l'output. In particolare, una funzione di pooling sostituisce l'output del layer a una certa locazione con un'approssimazione statistica delle locazioni vicine. Ad esempio, la funzione **max pooling** ritorna l'output maggiore in un intorno rettangolare. In generale, l'operazione di pooling aiuta a rendere l'output invariante a piccole traslazioni dell'input, questo torna utile in tutti i casi in cui si è interessati alla *presenza* di determinate caratteristiche piuttosto che alla loro esatta posizione. Dato che l'operazione di pooling effettua una approssimazione su un insieme di punti vicini tra loro, è possibile impiegare meno unità di pooling che unità di identificazione, ritornando un'approssimazio-

ne di regioni di pooling distanti k punti tra loro piuttosto che regioni adiacenti, aumentando ulteriormente l'efficienza computazionale in quanto il layer successivo elaborerà di fatto circa k input in meno. In molti casi, questa stessa proprietà diventa essenziale per gestire input di diversa dimensione, ad esempio ritornando un numero fisso di approssimazioni a prescindere dalle dimensioni dell'input.

2.4.6 Varianti della funzione di convoluzione

Nel contesto di una rete neurale, per convoluzione si intende solitamente un'operazione che consiste nell'applicazione di più convoluzioni in parallelo. Ciò deriva dal fatto che una convoluzione con un singolo kernel è in grado di estrarre solo un tipo di feature, mentre normalmente si è interessati ad estrarre più feature ad ogni layer della rete. Inoltre, l'input ha tipicamente la forma di una griglia di osservazioni vettoriali piuttosto che una griglia di valori reali e l'input di un layer è dato dall'output del precedente. Pertanto, le implementazioni utilizzano la cosiddetta batch mode descrivendo un'immagine, ad esempio, in termini di un tensore 4-D nel quale un asse rappresenta la dimensione di batch, due assi per le coordinate spaziali e un asse come indice nei vari canali (come l'intensità di blu, rosso e verde per ogni pixel). Di conseguenza le operazioni lineari su cui si basano le convoluzioni non sono commutative, anche riflettendo il kernel, se non quando ogni operazione ha lo stesso numero di canali di input e output.

Supponendo di avere un tensore 4-D $\boldsymbol{K}$ rappresentante il kernel dove $K_{i,j,k,l}$ descrive la forza della connessione tra un'unità del canale di input i e un'unità nel canale di output j con uno spostamento di k righe e l colonne tra le due unità. Considerando un input $\boldsymbol{V}$ di osservazioni, dove $V_{i,j,k}$ descrive il valore dell'unità di input nel canale i, alla riga j e colonna k, e un output $\boldsymbol{Z}$ della stessa forma di $\boldsymbol{V}$. Se $\boldsymbol{Z}$ è prodotto tramite la convoluzione di $\boldsymbol{K}$ su $\boldsymbol{V}$, senza riflettere $\boldsymbol{K}$ e ignorando la dimensione di batch per chiarezza di notazione, formalmente

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n} \tag{2.25}$$

dove la sommatoria su l, m e n viene effettuata su tutti i valori per cui le operazioni di indicizzazione sul tensore sono valide. Si potrebbe anche voler saltare alcune posizioni del kernel per ridurre il costo computazionale, seppur al costo di un'estrazione di caratteristiche meno rifinita. Se si vogliono scegliere elementi ogni s punti in ogni direzione nell'output, possiamo definire la funzione di convoluzione c *sottocampionata* come

$$Z_{i,j,k} = c(\boldsymbol{K}, \boldsymbol{V}, s)_{i,j,k} \sum_{l,m,n} [V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n}] \tag{2.26}$$

dove s è detta **stride** della convoluzione sottocampionata.

2.4.7 Padding

Una caratteristica essenziale di ogni implementazione di una rete convoluzionale è la capacità di 'imbottire' con degli zeri (dall'inglese to zero pad), l'input $\boldsymbol{V}$ per ingrandirlo in modo da evitare che la dimensione dell'output di ogni layer diminuisca di un pixel in meno della larghezza del kernel a ogni iterazione. L'operazione di padding permette di controllare la larghezza del kernel e la dimensione dell'output indipendentemente, evitando così di dover scegliere tra un rapido rimpicciolimento dell'estensione spaziale della rete e kernel più piccoli, entrambe soluzioni che limiterebbero il potere espressivo della rete. I casi più interessanti di zero-padding sono i seguenti.

Valid Convolution Padding

Un caso estremo è quello di non usare nessun padding, permettendo al kernel di visitare solo locazioni nell'input dove è contenuto per intero. Così facendo, oltre ad avere un output la cui dimensione diminuisce a ogni layer, come notato prima, tutti i punti dell'output sono una funzione dello stesso numero di punti dell'input, regolarizzandone il comportamento. Si fa notare come la diminuzione della dimensione dell'output è tanto maggiore quanto è grande il kernel, limitando il numero di layer convoluzionali inseribili nella rete.

Same Convolution Padding

Un altro caso è quello in cui si inseriscono tanti zeri quanti sono necessari per ottenere un output delle stesse dimensioni dell'input. In questa maniera, la rete non ha un limite di layer convoluzionali che può includere se non quello dato dai limiti dell'hardware usato dato che l'operazione di convoluzione non limita le possibilità architetturali per il layer successivo. Si nota tuttavia, che i punti vicini ai bordi hanno un'influenza minore sull'output dei punti al centro della griglia, in quanto selezionati dal kernel meno volte.

Full Convolution Padding

Per evitare la situazione precedente in cui i punti sui bordi non danno lo stesso contributo all'output, si potrebbe usare una forma padding tale che siano inseriti tanti zeri quanto necessario a fare in modo che ogni punto dell'input sia visitato esattamente k volte in ogni direzione, risultando in una griglia di output di larghezza $m + k - 1$. Tramite questo approccio però i punti sul bordo della griglia di output sono il risultato di una funzione di meno punti dell'input in confronto ai punti più centrali, rendendo più complicato l'apprendimento di un kernel dal comportamento ideale per ogni posizione nella feature map.

2.5 Recurrent Neural Networks

Una rete neurale **ricorrente** è un tipo di rete neurale specializzata nell'elaborazione di dati sequenziali basando il proprio funzionamento sulla condivisione dei parametri permettendo l'estensione del modello a sequenze di forma diversa mantenendo un certo livello di generalizzazione [12].

Il tipo di condivisione dei parametri è analogo a una rete convoluzionale applicata a serie temporali 1-D, ma con un effetto più profondo che nelle CNN. In una rete convoluzionale, infatti, l'output di una convoluzione è una sequenza in cui ogni elemento è una funzione di un numero ristretto degli elementi vicini corrispondenti nell'input, in base alla larghezza del kernel usato. In una rete ricorrente, invece, ogni elemento dell'output è una funzione degli elementi dell'output precedente ed è ottenuto con la stessa funzione d'aggiornamento usata per gli output precedenti. In questo modo, i parametri vengono condivisi in un grafo d'elaborazione molto profondo.

2.5.1 Grafi d'Elaborazione Ricorrenti

Dato un insieme di operazioni ricorrente, o ricorsivo, è possibile 'sgrovigliarlo'[5] in una sequenza di operazioni che vanno a formare un grafo di elaborazione con una struttura ripetitiva. Partiamo dalla forma classica di un sistema dinamico:

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}) \tag{2.27}$$

dove $\boldsymbol{s}^{(t)}$ indica lo stato del sistema. L'equazione 2.27 è ricorrente in quanto lo stato del sistema al tempo t è definito in termini dello stato del sistema al tempo $t-1$. Per un numero di passi temporali τ è possibile sgrovigliare il grafo applicando la definizione $\tau - 1$ volte. Ad esempio, srotolando l'equazione 2.27 per $\tau = 3$ passi temporali, avremmo

$$\boldsymbol{s}^{(3)} = f(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}) \tag{2.28}$$

$$= f(f(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta}) \tag{2.29}$$

Sgrovigliando l'equazione applicando ripetutamente la definizione in questo modo ci ha portati a un'espressione che non include ricorrenza e può essere rappresentata con un classico grafo di elaborazione aciclico direzionato (sezione 2.3.1).

Consideriamo ora un sistema dinamico guidato da un segnale esterno $\boldsymbol{x}^{(t)}$, formalmente

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \tag{2.30}$$

[5] Dall'inglese 'to unfold'

dove, si fa notare, lo stato del sistema incorpora informazioni sull'intera sequenza passata. Associando lo stato alle hidden units di una rete, possiamo riscrivere l'equazione 2.30 usando la variabile $\boldsymbol{h}$ per indicare lo stato, come

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \tag{2.31}$$

che è la forma tipica delle hidden units di una rete ricorrente. Possiamo rappresentare la ricorrenza sgrovigliata dopo t passi con una funzione $g^{(t)}$

$$\begin{aligned} \boldsymbol{h}^{(t)} &= g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \cdots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) && (2.32) \\ &= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) && (2.33) \end{aligned}$$

La funzione $g^{(t)}$ prende l'intera sequenza passata $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \cdots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ in input e ritorna lo stato corrente, ma la struttura ricorrente permette di fattorizzare $g^{(t)}$ come l'applicazione ripetuta di una funzione f, introducendo due importanti proprietà:

1. Il modello appreso ha sempre la stessa dimensione di input, a prescindere dalla lunghezza della sequenza, in quanto specificato in termini della transizione da uno stato all'altro.

2. È possibile usare la *stessa* funzione f con gli stessi parametri ad ogni passo temporale.

Rendendo possibile l'apprendimento di un singolo modello f operante su tutti i passi temporali e lunghezze di sequenza piuttosto che apprendere un modello diverso $g^{(t)}$ per ogni possibile time step. L'apprendimento di un singolo modello condiviso permette la generalizzazione su sequenze di lunghezza diversa da quelle viste nel training set e permette al modello di essere stimato con molti esempi d'addestramento in meno di quelli che sarebbero stati necessari senza la condivisione dei parametri.

2.5.2 Reti Neurali Ricorrenti

Consideriamo ora una rete ricorrente che associa a una sequenza di input di $\boldsymbol{x}$ valori una corrispondente sequenza di output di $\boldsymbol{o}$ valori. Supponendo di voler ottenere un output discreto, assumendo $\boldsymbol{o}$ come delle probabilità logaritmiche non normalizzate (sezione 2.3.4), una funzione costo L misura quanto $\boldsymbol{o}$ dista dal corrispondente valore obiettivo $\boldsymbol{y}$ calcolando il vettore delle probabilità normalizzate $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$. Le connessioni dall'input alle hidden units della rete sono parametrizzate dalla matrice dei pesi $\boldsymbol{U}$, quelle tra hidden units dalla matrice $\boldsymbol{W}$ e

quella dalle hidden units all'unità di output da $\boldsymbol{V}$. L'operazione di forward propagation ha inizio con la specifica dello stato iniziale $\boldsymbol{h}^{(0)}$ e poi, per ogni time step t da $t = 1$ a $t = \tau$, si applicano le seguenti equazioni di aggiornamento:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \tag{2.34}$$

$$\boldsymbol{h}^{(t)} = \text{relu}(\boldsymbol{a}^{(t)}) \tag{2.35}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \tag{2.36}$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{o}^{(t)}) \tag{2.37}$$

$$\tag{2.38}$$

dove i parametri sono i vettori di bias $\boldsymbol{b}$ e $\boldsymbol{c}$ insieme a le matrici dei pesi $\boldsymbol{W}$, $\boldsymbol{V}$ e $\boldsymbol{U}$ descritte in precedenza.

Questo è un esempio di una rete ricorrente che associa, a una sequenza di input, una sequenza di output della stessa lunghezza. La loss totale per una data sequenza di $\boldsymbol{x}$ valori accoppiata a una sequenza di $\boldsymbol{y}$ valori sarebbe quindi semplicemente data dalla somma delle loss a ogni time step. Ad esempio, se $L^{(t)}$ è la verosimiglianza logaritmica negativa di $y^{(t)}$ dati $\{\boldsymbol{x}^{(t)}, \cdots, \boldsymbol{x}^{(t)}\}$, allora

$$L\left(\{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \cdots, \boldsymbol{y}^{(\tau)}\}\right) \tag{2.39}$$

$$= \sum_t L^{(t)} \tag{2.40}$$

$$= -\sum_t log\ p_{model}\left(y^{(t)} | \{\boldsymbol{x}^{(t)}, \cdots, \boldsymbol{x}^{(\tau)}\}\right) \tag{2.41}$$

dove $p_{model}\left(y^{(t)} | \{\boldsymbol{x}^{(t)}, \cdots, \boldsymbol{x}^{(\tau)}\}\right)$ è ottenuto osservando l'esempio per $y^{(t)}$ dal vettore di output $\hat{\boldsymbol{y}}^{(t)}$ del modello. Il calcolo del gradiente per questa funzione di loss rispetto ai parametri è un'operazione costosa. La complessità temporale è di $O(\tau)$ e non può essere ridotta tramite parallelizzazione perchè il grafo risultante è per natura sequenziale: ogni time step è calcolabile solo dopo aver calcolato il precedente. Gli stati calcolati nel forward pass devono essere memorizzati fino al loro riutilizzo durante il backward pass, quindi il costo in memoria è sempre $O(\tau)$. L'operazione di back-propagation applicata al grafo sgrovigliato con costo $O(\tau)$ è detta **back-propagation temporale** (back-propagation through time), o BPT.

2.5.3 Problema della Scomparsa del Gradiente

Quando il grafo di elaborazione è molto profondo, come nel caso di reti feedforward con molti layer o, molto più comunemente, reti ricorrenti che costruiscono grafi profondi da ripetute applicazioni della stessa operazione a ogni time step di una sequenza temporale, gli algoritmi di ottimizzazione vanno incontro ad un serio

problema. Supponiamo di avere un grafo contenente una connessione che consiste nel moltiplicare ripetutamente la matrice $\boldsymbol{W}$. Dopo t passi, l'operazione equivale a moltiplicare per $\boldsymbol{W}^t$. Supposto che $\boldsymbol{W}$ sia fattorizzabile per decomposizione spettrale[6] in $\boldsymbol{W} = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}$ si nota che

$$\boldsymbol{W}^t = (\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1})^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1} \tag{2.42}$$

Ogni autovalore λ_i che non è prossimo a 1 in valore assoluto tenderà ad esplodere, se $|\lambda_i| > 1$, o a scomparire, se $|\lambda_i| < 1$. I problemi dell'**esplosione e scomparsa del gradiente** derivano dal fatto che i gradienti in questi grafi dipendono da $\text{diag}(\boldsymbol{\lambda})^t$. Gradienti che scompaiono rendono difficile capire in che direzione spostare i parametri in modo da migliorare la funzione costo, mentre gradienti che esplodono rendono l'apprendimento instabile.

In una rete ricorrente, anche supponendo di avere parametri che rendono la rete stabile, si arriva ad avere pesi in lunghe interazioni esponenzialmente minori di quelli in interazioni più brevi. La ricorrenza, infatti, consistendo nella composizione della stessa funzione a ogni time step, può essere interpretata come una moltiplicazione di matrici, ad esempio

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}^\intercal\boldsymbol{h}^{(t-1)} \tag{2.43}$$

si può intendere come una semplice rete ricorrente senza un'attivazione non lineare e un input $\boldsymbol{x}$. Si nota come la ripetizione del prodotto di $\boldsymbol{W}$ sia molto simile al metodo della potenze usato per trovare l'autovalore massimo di una matrice e il corrispondente autovettore, semplificabile in

$$\boldsymbol{h}^{(t)} = (\boldsymbol{W}^t)^\intercal\boldsymbol{h}^{(0)} \tag{2.44}$$

e se $\boldsymbol{W}$ ammette una decomposizione spettrale della forma

$$\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\intercal \tag{2.45}$$

con $\boldsymbol{Q}$ ortogonale, la ricorrenza è ulteriormente semplificabile in

$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^\intercal\boldsymbol{\Lambda}^t\boldsymbol{Q}\boldsymbol{h}^{(0)} \tag{2.46}$$

Gli autovettori, come appare nell'equazione 2.46, sono elevati alla potenza di t. Questo vuol dire che gli autovettori maggiori di 1 (analogamente minori di 1) in valore assoluto tenderanno ad esplodere (analogamente svanire) e ogni componente dello stato iniziale $\boldsymbol{h}^{(0)}$ che non è allineato con l'autovettore massimo sarà eventualmente scartato.

[6]Sia $\boldsymbol{A}$ una matrice quadrata $n \times n$ con n autovettori[7]linearmente indipendenti q_i, con $i = \{1, \cdots, n\}$, allora $\boldsymbol{A}$ è fattorizzabile in $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$ dove $\boldsymbol{Q}$ è la matrice quadrata $n \times n$ la cui i-esima colonna è l'autovettore q_i di $\boldsymbol{A}$ e $\boldsymbol{\Lambda}$ è la matrice diagonale i cui elementi diagonali sono i corrispondenti autovalori $\Lambda_{ii} = \lambda_i$

[7]L'autovettore di una funzione è un vettore non nullo la cui immagine è il vettore stesso moltiplicato per un numero, detto autovalore.

2.5.4 Gated RNNs

I modelli ricorrenti che hanno riportato i risultati migliori in diverse applicazioni pratiche appartengono alla categoria delle cosiddette **gated RNNs** basate sull'idea di creare delle connessioni, i cui pesi possono mutare dinamicamente a ogni passo temporale, con derivate immuni all'esplosione o alla scomparsa del gradiente.

Long Short-Term Memory

Le reti **Long Short-Term Memory** [17], o LSTM, si basano sull'introduzione di cicli interni per generare connessioni nelle quali il gradiente può propagarsi per molti passi temporali. In aggiunta, i pesi di queste connessioni, anzichè essere predeterminati, sono appresi tramite unità nascoste apposite, dette **unit gates**, che modificano dinamicamente la scala temporale di integrazione. Una rete LSTM ha quindi delle celle LSTM con gli stessi input e output di una classica unità ricorrente, ma un numero maggiore di parametri e un sistema di gated units che controllano il flusso dell'informazione. La componente più importante è l'unità stato $s_i^{(t)}$ dotata di un ciclo interno lineare il cui peso, o la costante temporale associata, è controllato da una unità **forget gate** $f_i^{(t)}$ (per il time step t e la cella i) che imposta il peso a un valore compreso tra 0 e 1 tramite un'unità sigmoide come

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \tag{2.47}$$

dove $\boldsymbol{x}^{(t)}$ è l'attuale vettore di input, $\boldsymbol{h}^{(t)}$ è il vettore dell'hidden layer attuale, contenente l'output di tutte le celle LSTM, e $\boldsymbol{b}^f$, $\boldsymbol{U}^f$ e $\boldsymbol{W}^f$ sono rispettivamente bias, pesi di input e pesi ricorrenti per le celle forget gates. Lo stato interno della cella LSTM è quindi aggiornato, con un peso condizionale del ciclo interno $f_i^{(t)}$, nel seguente modo

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i(t)\sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right) \tag{2.48}$$

dove $\boldsymbol{b}$, $\boldsymbol{U}$ e $\boldsymbol{W}$ sono i bias, pesi di input e pesi ricorrenti interni alla cella LSTM. L'unità **external input gate** $g_i(t)$ è calcolata in maniera simile, ma con i suoi parametri separati

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right) \tag{2.49}$$

L'output $h_i^{(t)}$ della cella LSTM può essere disattivato con un **output gate** $q_i^{(t)}$, sfruttando sempre un'unità sigmoide, ottenendo

$$h_i^{(t)} = \tanh(s_i^{(t)}) q_i^{(t)} \tag{2.50}$$

$$q_i^{(t)} = \sigma\left(b_i^q + \sum_j U_{i,j}^q x_j^{(t)} + \sum_j W_{i,j}^q h_j^{(t-1)}\right) \tag{2.51}$$

con i parametri $\boldsymbol{b}^q$, $\boldsymbol{U}^q$ e $\boldsymbol{W}^q$ a indicare i suoi bias, pesi di input e pesi ricorrenti.

Gated Recurrent Units

Un altro tipo di gated RNNs è quello delle reti ricorrenti con unità dette **Gated Recurrent Units** [18] che, a differenza delle LSTM, sfruttano una singola gated unit per controllare simultaneamente il forgetting factor e la decisione di aggiornare l'unità stato. L'equazione di aggiornamento è la seguente

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)}\right) \tag{2.52}$$

dove $\boldsymbol{u}$ è l'unità d'aggiornamento (update gate) e $\boldsymbol{r}$ l'unità di reset (reset gate), definite come

$$u_i^{(t)} = \sigma\left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)}\right) \tag{2.53}$$

e

$$r_i^{(t)} = \sigma\left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)}\right) \tag{2.54}$$

Le unità di aggiornamento e reset possono 'ignorare' individualmente parti del vettore stato. Le unità di aggiornamento possono copiarlo (a un estremo del sigmoide) o ignorarlo completamente (all'altro estremo) sostituendolo con il nuovo valore dello stato obiettivo. Le unità di reset invece decidono quali parti dello stato verrano usate per calcolare il successivo, introducendo un effetto nonlineare aggiuntivo alla relazione tra gli stati passati e quelli futuri [9].

2.6 Reti Neurali e Process Mining

Negli ultimi anni, la diffusione nell'uso di reti neurali artificiali in campi diversi, dal riconoscimento di immagini all'elaborazione del linguaggio naturale, ha causato un interesse crescente nell'applicazione di tecniche di deep learning mirate all'analisi di log di eventi per ottenere dettagli accurati sull'evoluzione di un processo aziendale.

Le prime applicazioni di reti neurali al problema della predizione dell'attività successiva riguardavano l'uso di reti LSTM. Uno studio recente confronta l'utilizzo di diversi tipi di Recurrent Neural Networks, da generiche RNN a LSTM e reti GRU, provando la loro superiorità rispetto a metodi tradizionali [19].

Una valida alternativa alle RNNs per la gestione di dati sequenziali, come è il caso per i record di un event log, è data dall'uso di reti neurali convoluzionali. Infatti, le stesse peculiarità che permettono loro di eccellere nel campo della computer vision, l'estrazione di features locali e la modularità di rappresentazione, possono essere efficaci anche nella gestione di sequenze temporali, trattando il tempo come dimensione spaziale. Tali CNN 1D si rendono competitive rispetto alle RNN ad un minore costo computazionale.

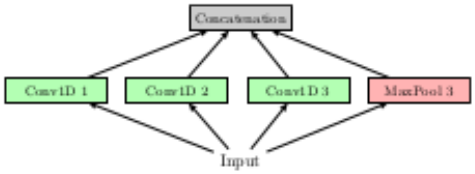
Capitolo 3

Stato dell'arte

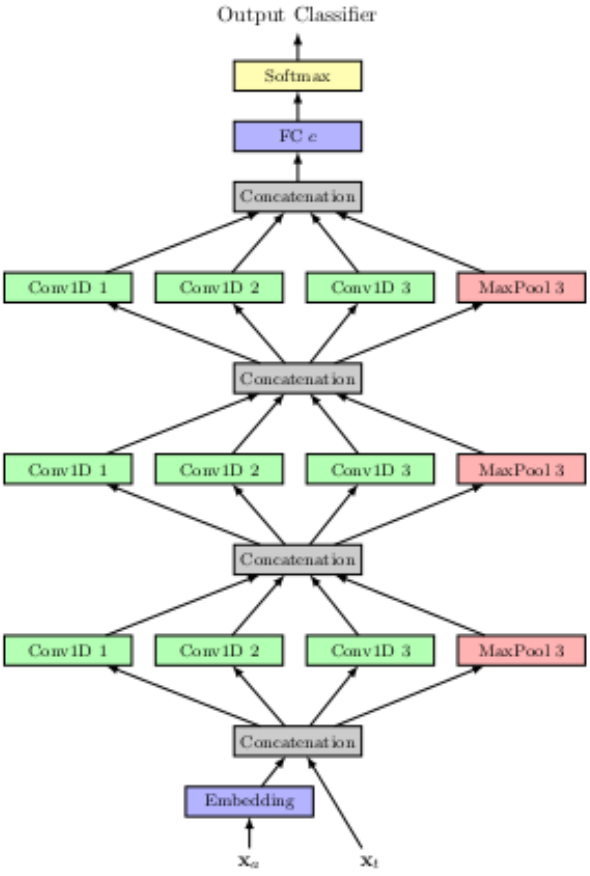
3.1 Inception Networks

In [20] gli autori propongono un modulo di Inception come blocco di base per l'architettura GoogLeNet composto da un layer convoluzionale dotato di più filtri di diversa lunghezza che operano contemporaneamente sull'input. L'idea alla base dell'Inception module si basa sul ruolo della dimensione del kernel in una CNN; kernel di grandi dimensioni sono in grado di individuare feature globali, mentre kernel più piccoli individuano feature locali più efficacemente. Tuttavia, le parti salienti di immagini o sequenze possono essere di dimensioni estremamente variabili, rendendo la scelta del numero di filtri un problema non semplice da gestire.

L'architettura proposta prevede la concatenazione di più convoluzioni a filtri di dimensione crescente (nella versione naive, rispettivamente tre layer da uno, due e tre filtri più un layer di MaxPooling), allargando la rete con convoluzioni parallele, come mostrato nella figura seguente.



(a) Naïve inception module



(b) The overall architecture.

In [21], questa architettura viene applicata al problema della Next Activity Prediction su dataset di Process Mining, mettendo a confronto l'approccio convoluzionale con metodi random e LSTM provandone la superiorità sia in termini di accuratezza che di tempo.

3.2 Deformable Convolutions

3.2.1 Introduzione

In [22], viene proposta una soluzione al problema della gestione delle variazioni geometriche nelle immagini, uno dei problemi chiave della computer vision per il riconoscimento visivo. Gli approcci classici prevedono tecniche di data augmentation sull'input, ad esempio applicando trasformazioni affini sugli esempi di training, o l'applicazione manuale di algoritmi e features invarianti alle trasformazioni sviluppati in maniera diversa da caso a caso. Queste soluzioni classiche hanno tuttavia un paio di aspetti negativi. Nel primo caso le trasformazioni geometriche sono assunte costanti e conosciute in modo da migliorare i dati e progettare algoritmi ad hoc, ma ciò impedisce di generalizzare su compiti con trasformazioni diverse non propriamente modellate. Nel secondo caso, invece, la progettazione di features e algoritmi invarianti alle trasformazioni può essere molto difficile, nei casi in cui queste trasformazioni, anche se conosciute, sono molto complesse. In breve, una rete convoluzionale è per sua natura limitata nella rappresentazione di grandi e sconosciute trasformazioni. Ciò è dovuto alle strutture geometriche fisse usate dalle CNN: un'unità convoluzionale campiona la feature map di input a posizioni fisse, un layer di pooling riduce la risoluzione spaziale a un tasso predefinito, ecc. Sono assenti meccanismi interni per la gestione di trasformazioni geometriche causando diversi problemi, come ad esempio la dimensione costante del campo recettivo di tutte le unità d'attivazione in un layer, soluzione non auspicabile in convoluzioni di alto livello che codificano il significato di determinate posizioni spaziali. Diverse posizioni, infatti, potrebbero corrispondere agli stessi oggetti a scala diversa o deformati; in un task di semantic segmentation per riconoscere *quali* oggetti si trovano all'interno di un'immagine, sarebbe desiderabile essere in grado di adattare dinamicamente diverse scale e le dimensioni del campo ricettivo.

3.2.2 Convoluzione Deformabile

La convoluzione 2-D consiste in due passaggi: campionamento su una tassellatura regolare R della feature map di input e somma dei valori campionati pesati da $\boldsymbol{w}$. La tassellatura R, che ha la forma di una griglia, definisce il campo ricettivo della rete. Ad esempio

$$R = \{(-1,-1),(-1,0),\ldots,(0,1),(1,1)\} \tag{3.1}$$

rappresenta un kernel 3×3.

Nel dettaglio l'operazione di convoluzione è la seguente, per ogni p_0 nell'output y pesato da $\boldsymbol{w}$:

$$\boldsymbol{y}(\boldsymbol{p}_0) = \sum_{\boldsymbol{p}_n \in R} \boldsymbol{w}(\boldsymbol{p}_n) \cdot \boldsymbol{x}(\boldsymbol{p}_0 + \boldsymbol{p}_n + \Delta \boldsymbol{p}_n) \tag{3.2}$$

Dove $\boldsymbol{R}$ indica la tassellatura regolare di campionamento, aumentata con gli offset $\{\Delta p_n \mid n = 1, \cdots, N\}$ con $N = |R|$. Il campionamento avviene quindi sulle posizioni irregolari $p_n + \Delta p_n$, ma, essendo queste tipicamente frazionali, l'equazione 3.2 è implementata tramite interpolazione bilineare, diventando:

$$\boldsymbol{x}(\boldsymbol{p}) = \sum_{\boldsymbol{q}} G(\boldsymbol{q}, \boldsymbol{p}) \cdot \boldsymbol{x}(\boldsymbol{q}) \tag{3.3}$$

dove $\boldsymbol{p} = \boldsymbol{p}_0 + \boldsymbol{p}_n + \Delta \boldsymbol{p}_n$ indica un punto arbitrario frazionale, $\mathbf{q}$ enumera le posizioni in $\boldsymbol{x}$ e $G(\cdot, \cdot)$ è il kernel dell'interpolazione bilineare definito come

$$G(\boldsymbol{q}, \boldsymbol{p}) = g(q_x, p_x) \cdot g(q_y, p_y) \tag{3.4}$$

dove $g(a, b) = max(0, 1 - \mid a - b \mid)$, che si dimostra essere un'operazione di rapida esecuzione essendo non nulla solo per poche $\boldsymbol{q}$. [22]

3.2.3 Reti Deformabili

I moduli deformabili hanno lo stesso input e output delle loro controparti tradizionali, quindi non necessitano di modifiche per essere aggiunte ad un modello già esistente. Gli offset sono calcolati durante l'apprendimento tramite una apposito layer convoluzionale. L'apprendimento poi procede per retropropagazione tramite l'operazione di interpolazione bilineare delle equazioni 3.3 e 3.4. In particolare, l'operazione di retropropagazione calcola il gradiente rispetto all'offset $\Delta \boldsymbol{p}_n$ tramite

$$\frac{\partial \boldsymbol{y}(\boldsymbol{p}_0)}{\partial \Delta \boldsymbol{p}_n} = \sum_{\boldsymbol{p}_n \in R} \boldsymbol{w}(\boldsymbol{p}_n) \cdot \frac{\partial \boldsymbol{x}(\boldsymbol{p}_0 + \boldsymbol{p}_n + \Delta \boldsymbol{p}_n)}{\partial \Delta \boldsymbol{p}_n} \tag{3.5}$$

$$= \sum_{\boldsymbol{p}_n \in R} \left[\boldsymbol{w}(\boldsymbol{p}_n) \cdot \sum_{\boldsymbol{q}} \frac{\partial G(\boldsymbol{q}, \boldsymbol{p}_0 + \boldsymbol{p}_n + \Delta \boldsymbol{p}_n)}{\partial \Delta \boldsymbol{p}_n} \boldsymbol{x}(\boldsymbol{q}) \right] \tag{3.6}$$

dove $\frac{\partial G(\boldsymbol{q}, \boldsymbol{p}_0 + \boldsymbol{p}_n + \Delta \boldsymbol{p}_n)}{\partial \Delta \boldsymbol{p}_n} \boldsymbol{x}(\boldsymbol{q})$ deriva dall'equazione 3.4 e, essendo gli offset $\Delta \boldsymbol{p}_n$ bidimensionali, $\partial \Delta \boldsymbol{p}_n$ viene usata per indicare $\partial \Delta \boldsymbol{p}_n^x$ e $\partial \Delta \boldsymbol{p}_n^y$, per semplicità.

Capitolo 4

Soluzione adottata

In questo lavoro di ricerca si è cercato di portare avanti il lavoro iniziato da [21] testando le performance di nuovi modelli convoluzionali rispetto ai classici approcci ricorrenti su dati 1-D e nello specifico del campo del process mining sviluppando una rete deformabile come in [22] adattata alla gestione di dati sequenziali e una rete convoluzionale il cui kernel è modificato da una maschera binaria costante. Mostreremo infine i risultati confrontandoli con una rete convoluzionale standard.

4.1 Next-Activity Prediction

4.1.1 Introduzione

Introduciamo alcuni concetti base di process mining per descrivere meglio il tipo di compito che le reti devono completare. In generale ogni tecnica di process mining punta ad estrarre informazioni utili da log di eventi di un processo aziendale, dove ogni evento fa riferimento a un'attività di un particolare caso, o traccia di processo.

Siano $\mathcal{A}$ l'insieme delle attività, $\mathcal{C}$ l'insieme dei casi in termini dei loro identificatori e sia $\mathcal{D}_i$ l'insieme degli attributi con $\forall 1 \leq i \leq m$. Un **evento** è una tupla $e = (a, c, t, d_1, \ldots, d_m)$ dove $a \in \mathcal{A}, c \in \mathcal{C}, t$ è il timestamp e $d_i \in \mathcal{D}_i$ sono eventuali altri attributi che non verranno considerati nel continuo della descrizione, come in [21]. Dato un evento $e = (a, c, t, d_1, \ldots, d_m)$ è quindi possibile definire le funzioni $\pi_{\mathcal{A}}(e) = a, \pi_{\mathcal{C}}(e) = c, \pi_{\mathcal{T}}(e) = t$ e $\pi_{\mathcal{D}_i}(e) = d$.

Una **traccia** è una sequenza finita di eventi $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$, con $e \in \mathcal{E}$ e $n = |\sigma|$, tale che $\pi_{\mathcal{T}}(e_i) \leq \pi_{\mathcal{T}}(e_{i+1})$ e $\pi_{\mathcal{C}}(e_i) \leq \pi_{\mathcal{C}}(e_{i+1}), \forall 1 \leq i \leq n-1$. Data la traccia $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$, un prefisso σ^k di σ, con $k = |\sigma^k| \leq |\sigma|$, è la traccia $\sigma^k = \langle e_1, e_2, \ldots, e_k \rangle$. In particolare, una traccia rappresenta un processo completo, iniziato e completato, mentre un prefisso rappresenta un processo in esecuzione

(traccia corrente). Un **event log**, infine, è un insieme di tracce tali per cui ogni evento appare al più una volta in tutto il log.

4.1.2 Formulazione del compito

Il task di next-activity prediction consiste nella predizione dell'attività di un processo in esecuzione. Dato un log di eventi $\mathcal{L} = \{\sigma_i\}_{i=1}^{N}$ di N tracce, si costruisce un dataset $\mathcal{S} = \{(s_i, a_i)\}_{i=1}^{M}$, con $M > N$, dove s_i è una sequenza di eventi corrispondente al prefisso σ_j^k di una traccia $\sigma_j \in \mathcal{L}$, con $k \in \mathbb{N}^+$, e a_i è l'attività del $(k+1)$-esimo evento della traccia σ_j. Formalmente, sia $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$ una traccia e $\Pi_{\mathcal{A}}(\sigma, k) = \pi_{\mathcal{A}}(e_k)$ la funzione che ritorna l'attività del k-esimo evento di σ, allora

$$\mathcal{S} = \{(s_i, a_i)\}_{i=1}^{M} = \bigcup_{j=1}^{N} \left\{ (\sigma_j^{k_j}, \Pi_{\mathcal{A}}(\sigma_j, k_j + 1)) \right\}_{k_j=1}^{|\sigma_j|} \tag{4.1}$$

In particolare, $\mathcal{S}$ è il dataset di tutti i prefissi di tutte le tracce in $\mathcal{L}$ etichettate con l'attività successiva associata nella traccia corrispondente.

4.1.3 Rappresentazione delle Feature

Dato che $\mathcal{S}$ consiste di prefissi ed attività, è necessario convertire gli elementi di $\mathcal{S}$ in rappresentazioni numeriche $\boldsymbol{x} \in \mathbb{R}^l$, con $l \in \mathbb{N}^+$. Per ogni coppia $(s, a) = (\langle e_1, e_2, \ldots, e_k \rangle, a) \in \mathcal{S}$, costruiamo la corrispondente rappresentazione numerica $((\boldsymbol{x}_{act}, \boldsymbol{x}_t), y)$. Sia $f : \mathcal{A} \to [1, 2, \ldots, \mathcal{C}]$ una funzione che assegna un valore numerico ad ogni attività, con $\mathcal{C} = |\mathcal{A}|$ è la cardinalità del vocabolario delle attività, ad esempio il numero delle classi. Allora $\boldsymbol{x}_{act} = (x_1, x_2, \ldots, x_k)$ è il vettore per il quale $x_i = f(\pi_A(e_i))$ $\forall 1 \leq i \leq k$, mentre $\boldsymbol{x}_t = (x_1, x_2, \ldots, x_k)$ è il vettore delle rappresentazioni numeriche delle informazioni temporali delle attività. Analogamente $y = f(a)$.

4.2 Deformable ConvNet 1D

Dovendo gestire sequenze temporali anzichè immagini, è stato necessario modificare l'equazione (3.4) in modo da eseguire un'operazione di interpolazione lineare adeguata; verranno anche definiti in seguito i dettagli dell'implementazione.

Tassellatura regolare e offset

La tassellatura regolare che definisce il kernel dell'operazione di convoluzione deformata è definita, per dati sequenziali, come un vettore del tipo

$$R = (-k, \ldots, -1, 0, 1, \ldots, +k) \tag{4.2}$$

con $k = \text{floor}(ks/2)$ e ks dimensione del kernel. Per un kernel di dimensione 3 e strides=1, si avrà quindi

$$R = (-1, 0, 1) \tag{4.3}$$

I valori dell'offset sono invece ottenuti tramite un layer convoluzionale tradizionale, calcolato con il doppio dei filtri della feature map di input e same padding, sottratto all'input stesso. Il risultato dell'operazione è poi sommato alla griglia regolare $\mathcal{R}$ per ottenere gli spostamenti degli indici di input che verranno calcolati tramite l'equazione

$$G(\mathbf{q}, \mathbf{p}) = g(q, p) \tag{4.4}$$

con g() invariata da $g(a, b) = max(0, 1- \mid a - b \mid)$.
L'equazione completa è analoga all'equazione 3.3 con il solo adattamento della funzione G con quella dell'equazione precedente. Il risultato della convoluzione deformata è un tensore della stessa dimensione della feature map di input, quindi, da questo punto di vista, la convoluzione deformabile si comporta come una convoluzione standard con same padding.

4.2.1 Implementazione

Il layer deformabile è stato implementato in Keras[1], una libreria open-source di deep learning basata sul backend TensorFlow[2], come una sottoclasse di Layer, la superclasse che definisce un layer generico nella libreria. La classe padre è formata da tre metodi principali che ne definiscono il comportamento. Il primo, **__init__()**, ha il compito di inizializzare gli attributi della classe avendo come parametri di input il numero di filtri, la dimensione del kernel ed eventuali altri parametri come la funzione d'attivazione o di regolarizzazione, il secondo, **build()**, definisce e inizializza i pesi che verranno utilizzati nel layer e ha come parametro la dimensione

[1] https://keras.io
[2] https://www.tensorflow.org/

Algorithm 3 Funzione __init__() di DeformableConv1D

Require: Numero di filtri di output **f**, Dimensione del kernel **ks**

Inizializza gli attributi dei filtri e della dimensione del kernel:

filters $\leftarrow$ **f**

kernel_size $\leftarrow$ **ks**

Inizializza l'attributo della tassellatura regolare R

R $\leftarrow$ **regularGrid(kernel_size)**

Algorithm 4 Funzione regularGrid

Require: Dimensione del kernel **ks**

Crea un array di interi della stessa dimensione del kernel inizializzato a zero:

$\boldsymbol{R} \leftarrow \text{zeros}(\boldsymbol{ks}, \text{int32})$

Inizializza la variabile degli elementi della tassellatura:

$\boldsymbol{e} \leftarrow -\text{floor}(\boldsymbol{ks}/2)$

for i in range($\boldsymbol{ks}$) **do**

Aggiorna l'elemento corrente della tassellatura:

$\boldsymbol{R}[i] \leftarrow \boldsymbol{e}$

Ottieni l'elemento successivo:

$\boldsymbol{e} \leftarrow \boldsymbol{e} + 1$

end for

return R

dell'input e il terzo, **call()**, contiene la logica del layer avendo come parametro il tensore di input.

La sottoclasse DeformableConv1D implementa i tre metodi per override. Il nuovo metodo __init__() inizializza gli attributi dei filtri e della dimensione del kernel e definisce l'attributo della tassellatura regolare **R** tramite il nostro metodo regularGrid(). Il metodo build(), invece, crea il tensore dei pesi **W** della stessa dimensione del kernel. Infine, il metodo call() genera il tensore degli offset dall'output di Conv1D, a sua volta sottoclasse di Layer e classe standard di Keras per layer convoluzionali 1D, con padding 'same' e attivazione 'relu', per poi passarlo come parametro insieme al tensore di input al nostro metodo linearInterpolation() che calcola i valori offsettati che vengono poi pesati dai parametri definiti durante la fase di build() ottenendo l'output finale del layer. Lo pseudocodice dei metodi è riportato negli algoritmi 3, 5 e 6, mentre il codice completo originale è visibile nell'appendice A.1.

Algorithm 5 Funzione build() di DeformableConv1D

Require: Dimensione dell'input **input_shape**,
Inizializza la variabile della forma del tensore dei pesi:
W_shape ← shape(kernel_size, 1)
Dichiara il tensore dei pesi da apprendere e lo aggiunge alla lista dei pesi associati al layer corrente:
$\boldsymbol{W}$ = add_weight($\boldsymbol{W}, \ldots, trainable = True$)

Algorithm 6 Funzione call() di DeformableConv1D

Require: Tensore di input $\boldsymbol{x}$,
Definisci l'operazione di convoluzione 1D standard per il calcolo degli offset con 'same' padding:
offconv←**Conv1D**() dove Conv1D è il metodo nativo di Keras di un layer convoluzionale 1D.
Inizializza la variabile degli offset:
offset←**offconv**($\boldsymbol{x}$)
Calcola il tensore delle locazioni interpolate:
$\boldsymbol{y}$ ← linearInterpolation($\boldsymbol{x}$, **offset**)
Pesa il tensore risultante con i pesi creati nel metodo build:
$\boldsymbol{y} \leftarrow \sum_i \boldsymbol{W_i} * \boldsymbol{y}$
return $\boldsymbol{y}$

4.3 Masked ConvNet

Si è voluto implementare contemporaneamente, un layer convoluzionale standard dal kernel costante. In particolare viene applicata una maschera binaria casuale, generata in fase di pretraining. L'effetto della maschera è quello di generare un tensore sparso che deforma il kernel che ignora conseguentemente alcuni valori del campo ricettivo che rappresenta. La maschera, così come il kernel stesso, non è considerata come un parametro da apprendere, ma rimane costante una volta definita per un certo layer riducendo il numero di parametri totali appresi dalla rete.

4.3.1 Implementazione

Il layer a kernel fisso MaskedConv1D è implementato come sottoclasse di Conv1D della quale implementa per override solo due metodi. Il metodo __init__() chiama il costruttore della superclasse specificando di non apprendere i pesi del layer corrente, mentre il metodo call() genera la maschera binaria e la applica al kernel generato dal metodo build(), che viene sempre chiamato subito prima dell'esecu-

Algorithm 7 Funzione linearInterpolation() di DeformableConv1D

Require: Tensore di input $\boldsymbol{x}$, Tensore degli offset **offset**
  Definisci il tensore delle locazioni di input $\boldsymbol{Q}$
  Calcola il tensore delle distanze **off** $\leftarrow$ offset $- x$
    offset$\leftarrow$**offconv**$(\boldsymbol{x})$
  Calcola il tensore delle locazioni di input aumentate dall'offset:
    $\boldsymbol{P} \leftarrow \boldsymbol{Q} - \boldsymbol{x}$
  Esegui l'operazione di interpolazione:
    $\boldsymbol{y} = g(\boldsymbol{Q}, \boldsymbol{P} + \boldsymbol{R})$
  return $\boldsymbol{y}$

Algorithm 8 Funzione g() di DeformableConv1D

Require: Tensore $\boldsymbol{q}$, Tensore $\boldsymbol{p}$
  Esegui l'operazione di interpolazione lineare $G \leftarrow \max(0, 1 - |q - p|)$
  return G

zione del corpo del metodo call(). Lo pseudocodice del metodo call() è definito nell'algoritmo 9 mentre il codice completo è presente nell'appendice A.2.

4.4 Dettagli Architetturali

Dato il dataset di training $\mathcal{D} = \{(x^i_{act}, x^i_t), y^i\}_{i=1^M}$, l'input delle architetture è dato dalla concatenazione del risultato di un *embedding* layer, che associa ogni parola di x_{act} in un vettore di dimensione fissa in $\mathbb{R}^d$ con $d = \text{ceil}(C/2)$, con l'input temporale x_t di valori in $\mathbb{R}^{k \times d}$ per una rappresentazione finale di valori in $\mathbb{R}^{k \times d+1}$.

Si susseguono poi una serie di layer convoluzionali e di max pooling il cui ultimo layer è input di un layer Dense il cui output è usato per calcolare le probabilità finali tramite una attivazione softmax (sezione 2.3.4). Come non linearità degli hidden layer si è scelto di usare una attivazione ReLU (sezione 2.3.2).

Per testare le performance dei nuovi modelli, si sono valutate tre reti per ogni approccio (standard, deformabile e costante) differenti solo nel numero di moduli convoluzionali in sequenza tra loro. In particolare sono state implementate reti da uno, due e tre moduli successivi, in cui ogni layer, ove possibile, aveva un kernel (o una griglia regolare per le convoluzioni deformabili) di lunghezza 3, seguito da un layer di max pooling.

Algorithm 9 Funzione call() di MaskedConv1D

Require: Tensore di input $\boldsymbol{x}$

Genera la maschera binaria delle stesse dimensioni del kernel:

mask$\leftarrow$ rand$(2, \text{size} = (\text{kernel_size}, 1, 1))$

Dove rand è una funzione che genera valori interi casuali e il cui primo parametro definisce il range di valori da generare, partendo da zero, mentre il secondo ne definisce la dimensionalità.

Aggiorna il kernel applicandovi la maschera binaria:

kernel $\leftarrow$ kernel $*$ mask

Chiama il metodo della superclasse per completare la convoluzione:

super.call()

4.5 Software e Hardware utilizzati

La performance dei modelli è stata testata sui seguenti cinque diversi dataset di process mining, come in [21], e per ognuno di essi, si è effettuata una 3-fold cross validation (sezione 2.2.2). Le metriche utilizzate sono il punteggio Brier e la accuracy. Il punteggio Brier può essere interpretato come una misura della calibrazione di un insieme di predizioni, calcolando l'errore quadratico medio della previsione per un certo elemento dell'input con il valore reale, ripetendo per ogni elemento dell'input. Il punteggio di Brier, quindi, è tanto migliore quanto più è **basso**. I log adottati, le cui statistiche sono riportate nella tabella 4.1, sono i seguenti:

- Receipt phase[3]: contiene i record dell'esecuzione della fase di ricevuta del processo di applicazione del permesso di costruzione in un anonimo comune olandese.

- helpdesk [4]: contiene gli eventi di un processo di biglietteria del reparto helpdesk di una software agency italiana.

- sepsis[5]: contiene eventi di casi di sepsi registrati dal sistema ERP di un ospedale.

- bpi12[6]: descrive il processo di un'applicazione di prestiti. Preprocessato come in [23].

[3]https://doi.org/10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6
[4]https://doi.org/10.17632/39bp3vv62t.1
[5]https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460
[6]https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

Tabella 4.1: Statistiche dei dataset

	classi	casi	sequenze
Receipt phase	26	1434	7143
helpdesk	8	3804	9906
sepsis	16	1050	14164
nasa	47	2566	71072
bpi12	22	13087	151419

- nasa[7]: contiene eventi al livello di chiamate dei metodi della classe NASA Crew Exploration Vehicle descritti da una esecuzione di una esaustiva suite di test d'unità.

L'addestramento delle reti sfrutta la libreria di hyperparameter optimization "hyperopt" [8] per la scelta della combinazione migliore degli iperparametri della rete utilizzando il 20% del training set come validation set scegliendo la configurazione di parametri con il valore di loss in validazione migliore. Come in [21], l'algoritmo di ottimizzazione per la scelta degli iperparametri è un Parzen Estimator ad albero. L'ottimizzazione dei pesi, si basa sull'algoritmo Adam (sezione 2.3.6), mentre si è optato per una regolarizzazione tramite early stopping (sezione 2.2.3), fermando la fase di training dopo aver smesso di osservare miglioramenti nella loss per 20 epoche consecutive su un limite di 200 epoche massime.

Si è scelto di implementare i layer definiti precedentemente in Python3 tramite la libreria Keras[9] usando il backend Tensorflow 2.0[10]. L'addestramento delle reti è avvenuto sulla piattaforma online fornita da Google Colab in contemporanea a una macchina locale per lo sviluppo. I tempi d'esecuzione sono riportati nelle tabelle 4.2, 4.3 e 4.4.

[7]https://doi.org/10.4121/uuid:60383406-ffcd-441f-aa5e-4ec763426b76
[8]https://github.com/hyperopt/hyperopt
[9]https://keras.io
[10]https://www.tensorflow.org

Tabella 4.2: Numero di parametri e tempo d'apprendimento in secondi dalla media delle tre fold su tutti i log per le architetture a un solo modulo.

	# parametri			tempo(s)		
	Std	Def	Mask	Std	Def	Mask
receipt	1716	1975	1268	27.8	43.2	65.4
helpdesk	821	256	341	11.6	15.8	23.4
sepsis	1456	675	688	20.6	55.2	31.4
nasa	5015	5810	2711	131.9	239.2	96.2
bpi12	2186	1453	1034	196.1	376.5	205.9

Tabella 4.3: Numero di parametri e tempo d'apprendimento in secondi dalla media delle tre fold su tutti i log per le architetture a due moduli.

	# parametri			tempo(s)		
	Std	Def	Mask	Std	Def	Mask
receipt	5716	3182	1300	13.9	55.2	41.3
helpdesk	3925	419	373	16.9	21.7	49.1
sepsis	4560	1078	720	70.9	82.6	39.8
nasa	8119	9317	2743	118.5	336.7	119.0
bpi12	5290	2344	1066	222.5	205.2	272.2

4.6 Conclusioni

4.6.1 Risultati a confronto

Le tabelle 4.2, 4.3 e 4.4 mostrano come il numero di parametri delle soluzioni implementate sia quasi sempre minore di quello delle convoluzioni standard, indicando le minori dimensioni dei modelli proposti. In termini di metriche, visibili nelle tabelle 4.5, 4.6 e 4.7 l'approccio deformabile ha mostrato, su una media dei tre test effettuati, una accuracy e un brier score marginalmente migliori concorrendo a validare l'ipotesi di partenza che le variazioni nei valori temporali di un processo possano essere paragonabili alle variazioni geometriche in un'immagine e affrontate con gli stessi mezzi nonchè l'importanza del ruolo del campo ricettivo di una rete convoluzionale.

Tabella 4.4: Numero di parametri e tempo d'apprendimento in secondi dalla media delle tre fold su tutti i log per le architetture a tre moduli.

	# parametri			tempo(s)		
	Std	Def	Mask	Std	Def	Mask
receipt	8820	4389	1332	99.6	49.7	140.2
helpdesk	7029	582	405	41.3	26.4	101.1
sepsis	7664	1481	752	52.6	139.9	94.0
nasa	11223	12824	2775	229.6	219.2	133.7
bpi12	8394	3235	1098	188.2	217.7	420.9

4.6.2 Sviluppi Futuri

Sono da indagare i contributi di ogni punto nel campo ricettivo deformato e da testare diverse forme di inizializzazione dei pesi che possono influenzare differentemente la struttura del campionamento seguendo le intuizioni di alcuni recenti studi [24] sulle caratteristiche del campo ricettivo delle reti convoluzionali, che già preannunciavano la validità teorica di un approccio deformabile, ed ampliando l'area di ricerca nelle reti convoluzionali in contesti diversi dalla computer vision.

4.6.3 Riepilogo

In questo lavoro di tesi, si sono proposte due soluzioni convoluzionali per la soluzione della predizione dell'attività successiva di una traccia in esecuzione, uno dei più complessi compiti di process mining. A confronto con una rete convoluzionale tradizionale, l'approccio deformabile si è mostrato più performante su diversi dataset tratti da log reali, provando la validità del lavoro effettuato.

Tabella 4.5: Risultati delle tre convoluzioni testate nelle architetture a singolo modulo. I risultati sono stati calcolati effettuando la media delle tre iterazioni del 3-fold cross validation e la relativa deviazione standard.

Modulo Singolo			
dataset	**metodo**	**brier score**	**accuracy**
helpdesk	standard	0.044 ± 0.001	0.775 ± 0.007
helpdesk	deformable	0.043 ± 0.001	0.774 ± 0.005
helpdesk	masked	0.044 ± 0.001	0.768 ± 0.006
receipts	standard	0.011 ± 0.000	0.813 ± 0.003
receipts	deformable	0.009 ± 0.000	0.860 ± 0.001
receipts	masked	0.010 ± 0.000	0.829 ± 0.004
sepsis	standard	0.030 ± 0.001	0.627 ± 0.018
sepsis	deformable	0.031 ± 0.000	0.633 ± 0.005
sepsis	masked	0.034 ± 0.001	0.552 ± 0.007
nasa	standard	0.003 ± 0.000	0.891 ± 0.002
nasa	deformable	0.003 ± 0.000	0.891 ± 0.002
nasa	masked	0.005 ± 0.000	0.821 ± 0.010
bpi12	standard	0.014 ± 0.000	0.774 ± 0.000
bpi12	deformable	0.014 ± 0.000	0.768 ± 0.004
bpi12	masked	0.017 ± 0.000	0.729 ± 0.004
media	standard	0.0204	0.776
media	deformable	0.0200	0.785
media	masked	0.0220	0.740

Tabella 4.6: Risultati delle tre convoluzioni testate nelle architetture a due moduli. I risultati sono stati calcolati effettuando la media delle tre iterazioni del 3-fold cross validation e la relativa deviazione standard.

Modulo Doppio			
dataset	**metodo**	**brier score**	**accuracy**
helpdesk	standard	0.044 ± 0.001	0.774 ± 0.004
helpdesk	deformable	0.043 ± 0.001	0.774 ± 0.006
helpdesk	masked	0.044 ± 0.001	0.769 ± 0.005
receipts	standard	0.010 ± 0.000	0.838 ± 0.002
receipts	deformable	0.009 ± 0.000	0.863 ± 0.005
receipts	masked	0.010 ± 0.000	0.829 ± 0.002
sepsis	standard	0.031 ± 0.000	0.628 ± 0.009
sepsis	deformable	0.032 ± 0.000	0.615 ± 0.007
sepsis	masked	0.035 ± 0.002	0.571 ± 0.030
nasa	standard	0.003 ± 0.000	0.887 ± 0.003
nasa	deformable	0.003 ± 0.000	0.893 ± 0.002
nasa	masked	0.003 ± 0.000	0.865 ± 0.011
bpi12	standard	0.014 ± 0.000	0.773 ± 0.001
bpi12	deformable	0.014 ± 0.000	0.773 ± 0.001
bpi12	masked	0.015 ± 0.000	0.755 ± 0.006
media	standard	0.0204	0.780
media	deformable	0.0202	0.777
media	masked	0.0214	0.765

Tabella 4.7: Risultati delle tre convoluzioni testate nelle architetture a tre moduli. I risultati sono stati calcolati effettuando la media delle tre iterazioni del 3-fold cross validation e la relativa deviazione standard.

Modulo Triplo			
dataset	**metodo**	**brier score**	**accuracy**
helpdesk	standard	0.044 ± 0.001	0.773 ± 0.005
helpdesk	deformable	0.043 ± 0.000	0.773 ± 0.004
helpdesk	masked	0.045 ± 0.001	0.765 ± 0.011
receipts	standard	0.010 ± 0.000	0.841 ± 0.002
receipts	deformable	0.009 ± 0.000	0.860 ± 0.006
receipts	masked	0.011 ± 0.000	0.822 ± 0.007
sepsis	standard	0.030 ± 0.000	0.639 ± 0.001
sepsis	deformable	0.031 ± 0.001	0.632 ± 0.005
sepsis	masked	0.036 ± 0.001	0.554 ± 0.006
nasa	standard	0.003 ± 0.000	0.890 ± 0.004
nasa	deformable	0.003 ± 0.000	0.892 ± 0.003
nasa	masked	0.004 ± 0.001	0.831 ± 0.040
bpi12	standard	0.014 ± 0.000	0.774 ± 0.001
bpi12	deformable	0.014 ± 0.000	0.774 ± 0.000
bpi12	masked	0.016 ± 0.000	0.745 ± 0.002
media	standard	0.0202	0.783
media	deformable	0.0204	0.779
media	masked	0.0220	0.751

Appendice A

Codice

A.1 DeformableConv1D

```
import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.layers import Layer, Conv1D, Dense
from tensorflow.python.keras.utils import conv_utils
from tensorflow.python.framework import tensor_shape
from tensorflow.python.ops import nn_ops
import numpy as np

# Deformable 1D Convolution
class DeformableConv1D(Layer):
    def __init__(self, filters, kernel_size=3):
        super(DeformableConv1D, self).__init__()
        self.filters = filters
        self.kernel_size = kernel_size
        self.R = tf.constant(self.regularGrid(self.kernel_size),
                                               tf.float32)
    def build(self, input_shape):
        W_shape = (self.kernel_size, 1)
        self.W = self.add_weight(
            name='W',
            shape=W_shape,
            trainable=True,
            dtype=self.dtype)
        super(DeformableConv1D, self).build(input_shape)
```

```
def call(self, x):
    offconv = Conv1D(x.shape[-1]*2, self.kernel_size,
                     padding='same', activation='relu',
                     trainable=True)
    offset = offconv(x)
    y = self.linearInterpolation(x, offset)
    y = tf.reduce_sum(self.W * y, [0])
    y = tf.reshape(y, [-1, x.shape[1], x.shape[2]])
    return y

"""
   Regular grid
   kernel_size: integer
"""
def regularGrid(self, kernel_size):
    R = np.zeros(kernel_size, dtype='int32')
    j = -(np.floor(kernel_size/2))
    for i in range(R.shape[0]):
        R[i] = j
        j += 1
    return R

"""
    linear interpolation
    x: (b, ts, c)
    offset: (b, ts, c)
"""
def linearInterpolation(self, x, offset):
    # input locations
    Q = tf.where(tf.equal(K.flatten(x), K.flatten(x)))
    Q = tf.cast(Q, tf.float32)

    offset = offset - x
    offset = K.flatten(offset)

    # offset locations
    P = Q + offset

    # regulard grid sampling
    # unstack necessary to bypass colab memory cap
    ylist = []
```

```
        for pn in tf.unstack(self.R):
          G = self.g(Q, P+pn)
          ylist.append(G * K.flatten(x))

        return tf.stack(ylist)

    """
        linear interpolation kernel
        q: input location
        p: offset location
    """
    def g(self, q, p):
        g = tf.subtract(tf.squeeze(q), tf.squeeze(p))
        g = tf.abs(g)
        g = tf.subtract(1.0, g)
        return tf.maximum(0.0, g)
```

A.2 MaskedConv1D

```
import tensorflow as tf
from tensorflow.keras.layers import Conv1D
from numpy import random.randint as rand

# Masked 1D Convolution
class MaskedConv1D(Conv1D):
    def __init__(self, filters, kernel_size, **kwargs):
        super(MaskedConv1D, self).__init__(filters,
                                           kernel_size,
                                           trainable=False,
                                           **kwargs)

    def call(self, x):
        # random boolean mask
        mask = rand(2, size=(self.kernel_size[0],1, 1))
        self.kernel = self.kernel * mask
        return super(MaskedConv1D, self).call(x)
```

Bibliografia

[1] A. Smith, *An Inquiry Into the Nature and Causes of the Wealth of Nations*. Goldsmiths'-Kress library of economic literature, Printed at the University Press for T. Nelson and P. Brown, 1827.

[2] M. v. Rosing and A.-W. Scheer, *The Complete Business Process Handbook, Volume 1 - Body of Knowledge from Process Modeling to BPM*, vol. 1. Morgan Kaufmann, 03 2015.

[3] F. W. Taylor, *The Principles of Scientific Management*. Harper & Brothers, 1911.

[4] T. Davenport, *Process Innovation: Reengineering Work Through Information Technology*. Harvard Business Review Press, 1993.

[5] G. Rummler and A. Brache, *Improving performance: how to manage the white space on the organization chart*. Management Series, Jossey-Bass Publishers, 1990.

[6] ISO Central Secretary, "Quality management systems — requirements," Standard ISO TR 9001:2015, International Organization for Standardization, 2015.

[7] V. Shankararaman, *Business Enterprise, Process, and Technology Management: Models and Applications: Models and Applications*. Business Science Reference, 2012.

[8] W. van der Aalst, A. Adriansyah, A. K. A. de Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. van den Brand, R. Brandtjen, J. Buijs, A. Burattin, J. Carmona, M. Castellanos, J. Claes, J. Cook, N. Costantini, F. Curbera, E. Damiani, M. de Leoni, P. Delias, B. F. van Dongen, M. Dumas, S. Dustdar, D. Fahland, D. R. Ferreira, W. Gaaloul, F. van Geffen, S. Goel, C. Günther, A. Guzzo, P. Harmon, A. ter Hofstede, J. Hoogland, J. E. Ingvaldsen, K. Kato, R. Kuhn, A. Kumar, M. La Rosa, F. Maggi, D. Malerba, R. S. Mans, A. Manuel, M. McCreesh, P. Mello, J. Mendling, M. Montali,

H. R. Motahari-Nezhad, M. zur Muehlen, J. Munoz-Gama, L. Pontieri, J. Ribeiro, A. Rozinat, H. Seguel Pérez, R. Seguel Pérez, M. Sepúlveda, J. Sinur, P. Soffer, M. Song, A. Sperduti, G. Stilo, C. Stoel, K. Swenson, M. Talamo, W. Tan, C. Turner, J. Vanthienen, G. Varvaressos, E. Verbeek, M. Verdonk, R. Vigo, J. Wang, B. Weber, M. Weidlich, T. Weijters, L. Wen, M. Westergaard, and M. Wynn, "Process mining manifesto," in *Business Process Management Workshops* (F. Daniel, K. Barkaoui, and S. Dustdar, eds.), (Berlin, Heidelberg), pp. 169–194, Springer Berlin Heidelberg, 2012.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[10] D. B. Lenat and R. V. Guha, *Building Large Knowledge-Based Systems; Representation and Inference in the Cyc Project*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[11] A. Cauchy, "Méthode générale pour la résolution des systemes d'équations simultanées," 1847.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[13] B. Polyak, "Some methods of speeding up the convergence of iteration methods," *Ussr Computational Mathematics and Mathematical Physics*, vol. 4, pp. 1–17, 12 1964.

[14] T. Schaul, I. Antonoglou, and D. Silver, "Unit tests for stochastic optimization," 2013.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[16] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[18] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[19] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, *Object Recognition with Gradient-Based Learning*, pp. 319–345. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999.

[20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[21] N. Di Mauro, A. Appice, and T. M. A. Basile, "Activity prediction of business process instances with inception cnn models," in *AI*IA 2019 – Advances in Artificial Intelligence* (M. Alviano, G. Greco, and F. Scarcello, eds.), (Cham), pp. 348–361, Springer International Publishing, 2019.

[22] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," *CoRR*, vol. abs/1703.06211, 2017.

[23] N. Tax, I. Verenich, M. La Rosa, and M. Dumas, "Predictive business process monitoring with lstm neural networks," *Lecture Notes in Computer Science*, p. 477–492, 2017.

[24] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," 2017.