UNIVERSITÀ DI PARMA

Dipartimento di Scienze Matematiche, Fisiche e Informatiche

Corso di Laurea in Informatica

Predizione di allagamenti con Deep Learning su sequenze temporali

Floodings prediction through Deep Learning of temporal sequences

Relatore:

Chiar.mo Prof. Alessandro Dal Palù

Correlatore:

Prof. Ing. Renato Vacondio

Tesi di Laurea di:

Diego Calanzone - Matricola: 296787

ANNO ACCADEMICO 2020-2021

Ai miei fratelli.

"Sii sempre come il mare che infrangendosi contro gli scogli, trova sempre la forza di riprovarci."
Jim Morrison

Ringraziamenti

Desidero ringraziare tutte le persone che hanno rappresentato un supporto morale, motivazionale e didattico lungo questo percorso di studi. La scelta di questa tesi deriva dall'incontro tra un'opportunitá e la passione per l'intelligenza umana e artificiale, passione che ha esteso le mie ore di studio, le mie letture, i seminari e ha guidato le scelte lungo il mio percorso accademico.

Grazie alla disponibilitá del prof.Ali Gooya, dell'universitá di Leeds, ho potuto sviluppare i primi progetti di Deep Learning a fianco dei ricercatori del suo gruppo. L'interesse per la materia e il mondo della ricerca hanno fornito maggiore motivazione per i miei studi, oltre a una direzione per il futuro.

Ringrazio inoltre la comunitá dell'Italian Association of Machine Learning (IAML) per il materiale didattico, i seminari di ricerca e le incredibilmente utili discussioni che hanno contribuito alla soluzione proposta in questo studio.

Un ringraziamento finale é rivolto al prof. Alessandro Dal Palù e al team di ricerca HyLab per questa opportunità di tesi; grazie al contatto anticipato con la ricerca ho potuto cogliere le sfide di questo progetto come opportunitá di apprendimento ed esperienza, proponendo degli avanzamenti e puntando a mantenere vivo l'interesse per questa direzione.

Indice

Introduzione

Le reti neurali stanno acquistando una crescente popolarità della ricerca accademica e industriale. Le loro applicazioni coinvolgono un ampio spettro di settori: dall'analisi medica alla guida autonoma, dalle reti sociali online all'ingegneria e lo studio dei fenomeni fisici. Il Machine Learning ha rivisitato il concetto di "intelligenza artificiale" introducendo un nuovo paradigma di apprendimento basato sull'esperienza, come avviene per gli umani. Con il Deep Learning, gli algoritmi intelligenti possono individuare pattern processando una mole di dati elevata, raggiungendo un'alta capacità di astrazione e comprensione dei concetti.

Questo progetto di tesi propone l'applicazione di tecniche di Deep Learning in un ambito popolare della ricerca: la fluidodinamica computazionale (o Computational Fluid Dynamics, CFD). L'obiettivo consiste nello studiare l'efficacia delle reti neurali profonde ad apprendere le leggi della fisica di base dei fluidi, cercando un vantaggio in termini di efficienza rispetto agli attuali software di simulazione basati sull'analisi numerica. La rete neurale progettata per questa tesi ha lo scopo di prevedere gli stati futuri di una simulazione forniti quelli precedenti nel passato, sostituendo completamente un normale simulatore CFD. Per risolvere questo task, chiamato *"Video Prediction"*, si combinano piú tecniche di Deep Learning provenienti dalla ricerca attuale allo stato dell'arte; i modelli sviluppati e gli esperimenti sono quindi raccolti in una libreria dedicata a questo progetto di ricerca: deepSWE. Le prestazioni dei modelli e i suoi output sono quindi confrontati con quelli del simulatore CFD.

Capitolo 1

Intelligenza Artificiale, Machine Learning e Deep Learning

La definizione di *intelligenza artificiale* é derivabile da quella di *intelligenza*: l'insieme di facoltà mentali per la comprensione delle entità, dei fatti, delle loro relazioni e il raggiungimento di una comprensione razionale su di essi. Sono definiti "intelligenza artificiale" i sistemi sviluppati per completare tasks autonomamente con prestazioni comparabili ad un'intelligenza umana.

Il termine é oggi molto diffuso a causa della crescente importanza nel mercato informatico degli ultimi decenni, tuttavia la sua filosofia trova radici nel lavoro di matematici e filosofi dei millenni precedenti.

1.1 Replicare il pensiero umano

L'Intelligenza Artificiale é nata dall'intuizione che il ragionamento umano potesse essere formalizzato in una serie di passaggi e operazioni. Il tentativo dell'uomo di riprodurre l'intelligenza risale al lavoro dei filosofi dell'antica Grecia, India e Cina nel primo millennio a.c: Aristotele, con il Sillogismo, costruí un primitivo formalismo per manipolare e intuire espressioni logiche

(es: *se A é B e B é C, allora A é C*); con gli Elementi [1], Euclide formuló la geometria e al-Khwarizmi sviluppó l'algebra e e il concetto di "algoritmo" [2].

Nel XIII° secolo, Llull ipotizzó di poter effettuare calcoli tra espressioni logiche utilizzando dei componenti meccanici (analogamente alle future porte logiche). Ereditando questo concetto, Leibniz ipotizzó che il processo razionale potesse essere tradotto in operazioni di algebra e geometria attraverso un linguaggio rappresentativo (*Characteristica Universalis*). Leibniz, Hobbes e Descartes iniziarono a sviluppare la Physical Symbol System Hypothesis (PSSH): un sistema di pattern fisici (simboli) organizzabili strutture (espressioni) manipolabili per generarne nuove. Nel XX° secolo, con il sorgere dei primi calcolatori, Turing ideó una macchina in grado di manipolare simboli astratti, ispirando una nuova generazione di ricercatori allo studio delle *thinking machines*.

1.2 La nascita dell'IA

Il termine *"Intelligenza Artificiale"* viene coniato nel 1956 da McCarthy[3] durante il Dartmouth Workshop, riconosciuto come pietra miliare della disciplina. L'obiettivo della ricerca consisteva nella risoluzione di problemi complessi attraverso la formulazione di algoritmi intelligenti, quali ad esempio:

- *Reasoning and search*: risolvere un problema come se fosse un "labirinto" (o grafo) in cui trovare la strada migliore per raggiungere la soluzione. Questo approccio é alla base di algoritmi per risolvere giochi comuni (es: Tic Tac Toe, schacchi) o ad esempio per controllare le decisioni di un robot.

- *Semantic Networks*: con l'obiettivo di comprendere il linguaggio umano e creare un sistema in grado di dialogare, le reti semantiche consistono in "mappe" di parole collegate con archi che rappresentano relazioni

logiche. ELIZA [4] fu un prototipo di *chatterbot* in grado di rispondere in una conversazione in modo realistico.

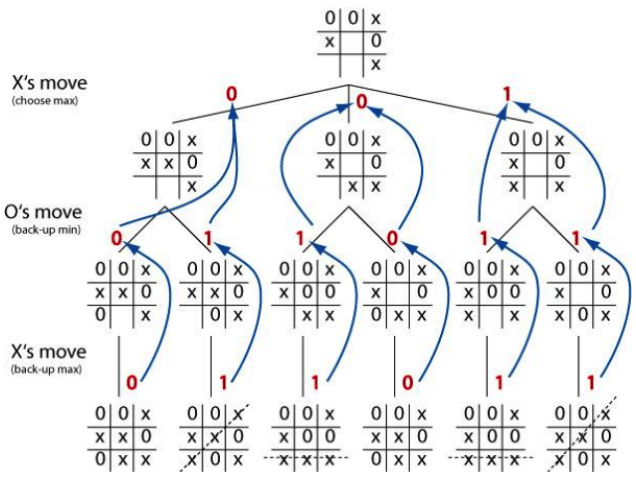


Figura 1.1: Albero delle mosse nel gioco TicTacToe e applicazione dell'algoritmo *MiniMax*

Questa famiglia di algoritmi si basa sulla premessa filosofica dell'intelligenza artificiale, per cui il pensiero umano puó essere formalizzato e ridotto alla computazione di simboli, prendendo il nome di **intelligenza artificiale simbolica**.

1.2.1 La difficoltà riscontrate con l'IA simbolica

Nel 1997 si svolse un confronto significativo tra l'intelligenza artificiale e quella umana: il computer "Deep Blue" di IBM sconfisse il campione modiale Garry Kasparov a scacchi. É necessario notare che l'algoritmo vincitore, basato su IA simbolica, non era tuttavia comparabile alla controparte umana. La difficoltà degli scacchi risiede principalmente nel ricordare le mosse, tenere conto di piú ipotetici scenari e anticipare l'avversario: principalmente operazioni che richiedono capacità di memorizzazione e calcolo rapido. DeepBlue fu progettato per risolvere problemi di questo tipo con semplicità: nel caso degli scacchi, le informazioni sul "mondo" da conoscere sono ridotte (una

schacchiera di 64 caselle con 32 pezzi su cui effettuare un numero limitato di mosse) e le regole del gioco possono essere introdotte come vincoli a priori dal programmatore. Paradossalmente, negli anni 90 i calcolatori erano in grado di risolvere problemi molto complessi per l'uomo, fallendo peró in task elementari alla portata di qualsiasi umano (*paradosso di Morovec*): riconoscere un oggetto da un'immagine, camminare o riconoscere un suono. Problemi di questo tipo nel mondo reale sono spesso difficili da formalizzare e vengono risolti attraverso la quantità di informazioni acquisite dall'esterno, l'intuito e l'esperienza.
Parallelamente all'IA basata sui formalismi e la logica nacque il **Machine Learning**, ossia l'apprendimento in maniera autonoma basato sull'esperienza e il riconoscimento di "pattern" dai dati.

1.3 Il Connessionismo e il Machine Learning

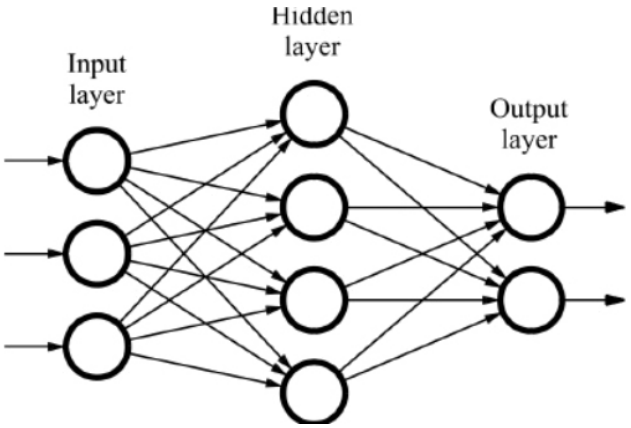


Figura 1.2: Semplice struttura di una rete neurale

Il *connessionismo*, in alternativa all'IA simbolica, nacque con l'ipotesi che l'intelligenza artificiale possa avere luogo simulando reti di neuroni. Nel 1943, Pitts e McCulloch svilupparono una rete artificiale per svolgere semplici funzioni logiche. Ispirandosi alla struttura biologica del neurone, nel 1958 Frank Rosenblatt ideó il *perceptron*: l'unità di computazione alla base delle moder-

ne reti neurali. Nonostante la critica [5] e la carente considerazione da parte dei ricercatori dell'epoca, l'interesse per i *perceptrons* riaffioró negli anni 80, quando Geoffrey Hinton e David Rumelhart idearono l'algoritmo di *backpropagation*: il meccanismo di apprendimento delle reti neurali moderne. Negli anni 90, le reti neurali riscontrarono successo in varie applicazioni di mercato quali il riconoscimento delle cifre scritte o delle parole da una traccia audio.

Il principio alla base delle reti neurali consiste nell'individuare una relazione tra i dati forniti in input e gli output: rispetto ad altri algoritmi nell'IA, le Neural Networks sono viste come una *black box*, di cui le regole interne sono difficilmente comprensibili. Di conseguenza, la scelta dei dati da fornire alla rete é cruciale per il successo dell'esperimento: le informazioni "selezionate" sono chiamate *features*. Per problemi di complessità crescente, o senza una ricca conoscenza a priori del problemi, risulta spesso difficile capire quali informazioni saranno importanti per ottenere i risultati sperati, di conseguenza le reti neurali sono progettate per imparare a identificare e sintetizzare le features (*representation learning*).

1.4 Il Deep Learning

Nel mondo reale sono varie e numerose le informazioni che percepiamo attraverso i sensi. Interagire con l'esterno, intraprendere azioni e comprendere gli eventi richiede una comprensione articolata e complessa. Per riconoscere un oggetto, ad esempio, bisogna tenere in considerazione la luminosità, l'angolazione, la forma, l'orientamento e tutti i dettagli distintivi che riusciamo a cogliere. Per un algoritmo, organizzare tutte queste informazioni da un semplice "flusso in ingresso di dati" é un problema che il solo representation learning non puó risolvere. É necessario scomporre il ragionamento in parti o *livelli di astrazione*, organizzando gerarchicamente le informazioni acquisite: il **Deep Learning** nasce proprio per svolgere questo compito.

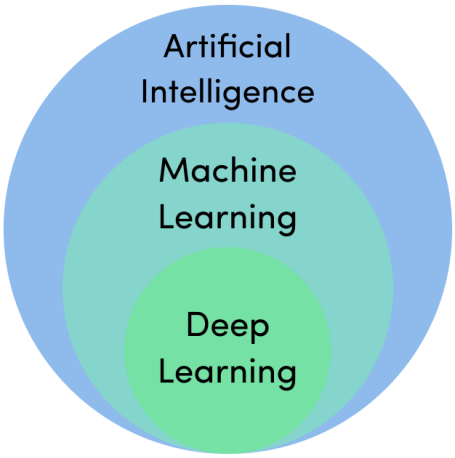


Figura 1.3: Gerarchia delle discipline: IA, Machine Learning e Deep Learning

Rispetto alle semplici reti neurali, una rete é definita *profonda* quando presenta numerosi strati di neuroni interni (si parla di **MLP**, o *Multi Layer Perceptron*). La profondità della rete assume una duplice importanza: dal punto di vista *computazionale*, piú strati permetteranno ai neuroni di specializzarsi su caratteristiche (features) differenti (es: uno strato per i bordi, uno per gli occhi, il successivo per il viso intero); dal punto di vista *concettuale*, la profondità della rete rappresenta la capacità di rappresentare la conoscenza su livelli di astrazione sempre piú complessi.
Solo nel XXI° secolo, il Deep Learning ha riscontrato una crescita esponenziale per campi di applicazione e interesse nella ricerca. Partendo dallo sviluppo di reti neurali per il riconoscimento di immagini (AlexNet [6]); piú recentemente acquistano popolarità architetture con miliardi di parametri, in grado di comprendere il linguaggio umano, scrivere articoli indistinguibili, comprendere e svolgere problemi di logica (i *Language Models* e le architetture Transformers-based come *GPT-3* [7] di OpenAI).

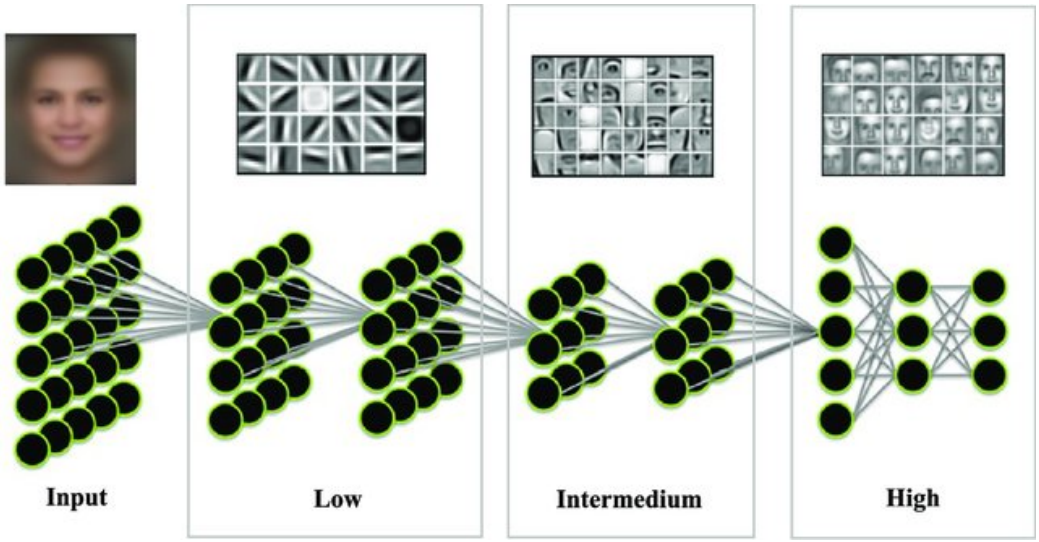


Figura 1.4: All'aumentare della profondità della rete, gli strati interni processano features piú astratte: contorni, oggetti, volti

1.5 Paradigmi di apprendimento, prevedere il futuro

Esistono molteplici paradigmi di apprendimento per le reti neurali. A seconda del problema da risolvere, la disponibilità dei dati varia e risulta sempre piú difficile indicare all'algoritmo l'obiettivo da raggiungere: nel mondo reale siamo esposti a informazioni oggettive e non strutturate a cui la nostra intelligenza attribuisce un significato. Prendendo come esempio i neonati, la loro comprensione del mondo si basa sulla pura osservazione. Crescendo, gli umani imparano a trarre conclusioni e previsioni interagendo con il mondo, osservando i risultati e traendo nuove ipotesi sulla base di un approccio *trial and error*. Come sostiene Yann Lecunn [8], questa conoscenza generalizzata del mondo é data per scontata per un umano, ma costituisce proprio "la materia oscura dell'intelligenza".
Le reti neurali possono essere addestrate secondo tre paradigmi:

- *Apprendimento supervisionato*: la rete neurale viene addestrata fornendo i dati input e gli output attesi. Ad esempio: per insegnare ad

una rete come riconoscere gli animali, forniamo sia le immagini che l'etichetta per ciascuna immagine.

- *Apprendimento non-supervisionato*: la rete neurale impara a distinguere i dati input attraverso le loro caratteristiche. Un esempio classico sono gli algoritmi di *clustering* per rappresentare gruppi di elementi con caratteristiche affini (es: persone con un hobby in comune in una società).

- *Apprendimento auto-supervisionato* (self-supervised): la rete neurale é in grado di acquisire una comprensione dei dati osservati al fine di prevedere i dati non osservati (prevedere i frame futuri in video o le parole mancanti in un dialogo).

É il *Self-Supervised Learning* (SSL) soggetto di ricerca e maggiore interesse da parte di Lecunn e molti altri ricercatori. Questo progetto di tesi si baserà su questo paradigma per imparare equazioni di fluidodinamica e prevedere il futuro da una sequenza di osservazioni passate.

Capitolo 2

Metodi computazionali per la simulazione dei fluidi

La simulazione del comportamento dei fluidi é oggetto di attiva ricerca e trova interesse nel mondo reale: la progettazione di un veicolo e lo studio della sua aerodinamica, o la previsione degli effetti di un'alluvione su un complesso di edifici. Sfruttando le risorse di calcolo offerte dai moderni computers, si ricorre alla **Computational Fluid Dynamics** (CFD): una branca della fluido dinamica che si affida all'analisi numerica e alle strutture dati per risolvere i problemi legati ai fluidi.
La base teorica di questi metodi é costituita dalle **equazioni di Navier-Stokes**: una famiglia di equazioni differenziali parziali per considerare le variazioni del fluido in densità e velocità nel tempo.

2.1 Metodi mesh-less e mesh-based

Considerando l'aria come fluido, si prendono come esempio le ali di un aereo durante la fase di volo: é necessario simulare la presenza dell'oggetto solido e le collisioni con le particelle d'aria. Per applicare un metodo computazionale, bisogna scomporre lo scenario in sezioni su cui effettuare i calcoli, ossia effettuare una *discretizzazione* suddividendo lo spazio in celle di una

griglia; per ogni cella, si applicano le equazioni di Navier-Stokes per simulare il comportamento del fluido con eventuali oggetti e quindi fornire lo stato al passo successivo nel futuro. Questo approccio é chiamato **mesh-based**: si introduce manualmente un oggetto tridimensionale in uno spazio e si analizza l'interazione con dei fluidi (come ad esempio nelle gallerie del vento per un aereo). Nonostante la semplicità del metodo, lo svantaggio consiste della dispendiosità di tempo e risorse per progettare qualsiasi oggetto tridimensionale considerato: nel mondo reale esistono innumerevoli forme e misure, pertanto risulta impossibile ricostruire manualmente la presenza di tutti i corpi nell'ambiente. Prendendo come esempio la simulazione delle onde del mare, bisogna inoltre considerare la variabilità della forma degli oggetti (le onde) e le loro trasformazioni (rotazione, traslazione): un metodo "statico" basato su griglia risulta inappropriato e inefficace.
In alternativa, si ricorre ai metodi **mesh-less**: il dominio continuo (il mondo reale) viene suddiviso in particelle senza informazioni topologiche (senza collegamenti tra di loro, reticolati o altre particolari strutture), che possono rappresentare un liquido, un solido o un gas. Ciascuna particella é in verità un'unità su cui effettuare computazione: l'algoritmo, per ogni cella, calcolerà il suo stato futuro in base alle interazioni con le particelle limitrofe.

2.2 Smoothed Particle Hydrodynamics

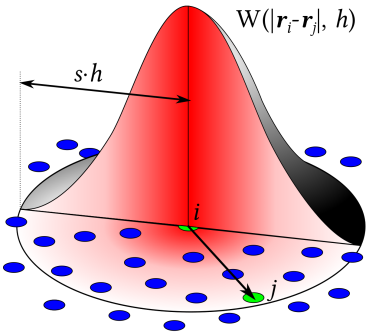


Figura 2.1: SPH: interpolazione locale di particelle con weighting function (kernel)

Piú concretamente, questo metodo prende il nome di **Smoothed Particle Hydrodynamics**, ed é definito *Lagrangiano*: le particelle posseggono le stesse proprietà del fluido, ossia posseggono velocità, densità e replicano le trasformazioni (traslazione, rotazione).
Alla base del metodo SPH é necessario risolvere le equazioni fondamentali:

$$\frac{\mathrm{d}\rho}{\mathrm{d}t} = \sum_b m_b \left(\mathbf{v}_a - \mathbf{v}_b\right) \cdot \nabla W_{ab} \tag{2.1}$$

Conservazione di massa

$$\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\sum_b m_b \left(\frac{p_b}{\rho_b^2} + \frac{p_a}{\rho_b^2}\right) \nabla W_{ab} \tag{2.2}$$

Conservazione del momento

$$\frac{\mathrm{d}e}{\mathrm{d}t} = \frac{1}{2}\sum_b m_b \left(\frac{p_b}{\rho_b^2} + \frac{p_a}{\rho_a^2}\right) \mathbf{v}_{ab} \cdot \nabla W_{ab} \tag{2.3}$$

Conservazione dell'energia

$$\frac{\mathrm{d}m_i}{\mathrm{d}t} = 0 \tag{2.4}$$

Condizione di massa costante

Dove ρ é la densità del fluido, v la velocità, t il tempo, e l'energia, ∇W_{ab} il gradiente del kernel applicato.
Come illustrato nella Figura 3.1, l'algoritmo calcola, per ogni particella, le equazioni in base ai vicini con un kernel "a campana" pesato in base alla distanza (*weighting function*), dato un raggio. Il problema principale risiede nel costo computazionale: in uno scenario semplice di 300.000 particelle, ciascuna con circa 250 vicini, simulare 1.5 secondi di simulazione richiede

circa 15 ore su un singolo processore da personal computer. L'obiettivo di questo progetto di tesi é quello di combinare l'**High Performance Computing (HPC)** con il **Deep Learning (DL)** per ovviare questa barriera temporale.

2.3 Il simulatore Parflood

2.3.1 Shallow water equations

Esistono scenari in un cui il problema puó essere formulato diversamente rispetto al metodo SPH: prendendo come esempio un fiume, l'estensione del fluido sull'asse orizzontale é ben maggiore rispetto all'asse verticale. In questo caso, si ricorre alla risoluzione delle **shallow water equations** (SWE), con un vantaggio in termini di efficienza. Si tratta di equazioni differenziali alla derivata parziale iperbolica, risolvibili con il metodo dei *volumi finiti*: si integrano le equazioni su un volume i cui estremi hanno valori finiti (condizioni di contorno); si suddivide il volume in *volumetti* di dimensione finita, quindi si scrivono le relazioni tra di essi da risolvere con l'ausilio di un calcolatore.

2.3.2 Risoluzione delle equazioni

Le equazioni SWE derivano dalle equazioni di Navier-Stokes con l'assunzione di base che la velocità verticale del fluido sia di scala nettamente inferiore rispetto alla velocità orizzontale, pertanto integrando sulla prima é possibile derivare le equazioni SWE. Queste sono risolte dal simulatore **Parflood** in forma vettoriale:

$$U_t + F_x + G_y = S(U) \tag{2.5}$$

$$U = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, F_x = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}, G_y = \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}, S(U) = \begin{bmatrix} s1 \\ s2 \\ s3 \end{bmatrix} \tag{2.6}$$

Dove h é l'altezza del fluido, u la velocità lungo l'asse x, v la velocità lungo l'asse y e g l'accelerazione gravitazionale. Considerando la portata come prodotto tra l'altezza di fluido h e le componenti velocità u e v, é possibile gestire anche altezze di fluido molto ridotte, ovviando problemi di instabilità numerica sulle velocità.

2.3.3 Il software e il calcolo parallelo

Esistono vari software dedicati alla risoluzione delle SWE. Per questo progetto di tesi viene utilizzato il simulatore **Parflood**, sviluppato dal prof. Alessandro Dal Palú in collaborazione con il team HyLab dell'università degli studi di Parma.

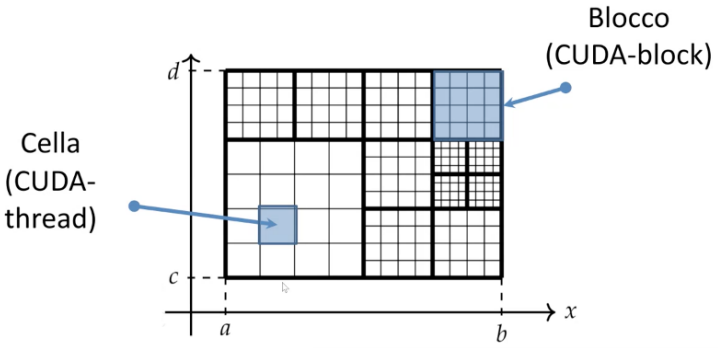


Figura 2.2: Griglia multi-risoluzione BUQG e parametri di parallelizzazione

L'input del simulatore consiste in "mappe" di rilevazioni di altezza di terreno e dei fluidi: la mappa é divisa in celle, ogni cella assume un valore per ogni grandezza (altezza del terreno, velocità orizzontale e verticale del fluido e altezza della colonna d'acqua); nel mondo reale, la struttura irregolare e complessa del terreno necessita di una migliore rappresentazione:

il simulatore Parflood é stato progettato per processare griglie con celle a risoluzione variabile, chiamate **Block Uniform Quad-tree Grid** (BUQG) [9] (Figura 2.2). La griglia é suddivisa in blocchi, ciascun blocco contiene lo stesso numero di celle, e ogni blocco si distingue dalla risoluzione delle proprie celle (uniformi tra loro); le equazioni SWE vengono calcolate per cella, per ciascuno step temporale; ogni cella é influenzata da quelle circostanti, i cui valori sono accessibili da aree di memoria contigue grazie all'architettura GPU. Ogni cella é gestita da un thread, ossia un'unità di comuputazione per il multiprocessore; piú thread possono sono raccolti in un blocco, all'interno del quale possono condividere una rapida memoria locale ad accesso condiviso.

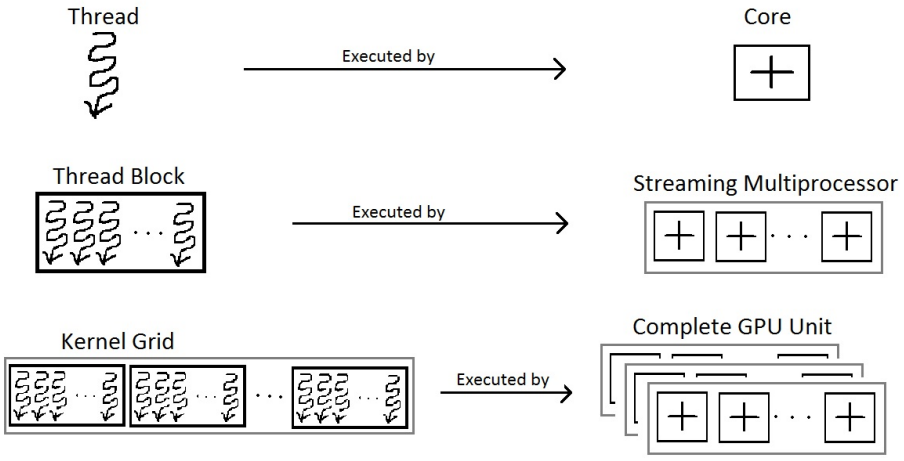


Figura 2.3: Gerarchia computazionale di una GPU Nvidia

2.3.4 Impostazione di una simulazione

Un fenomeno di alluvione simulato é composto da un terreno, un corpo liquido (il fluido) e una sorgente. Come nella realtà, l'osservazione consiste nel misurare dei valori a intervalli di tempo regolare, il simulatore parte da una scena iniziale e fornisce delle misurazioni a intevalli di tempo costanti. Le fasi seguite da Parflood per la generazione dei dati consistono in:

1. Acquisizione dei dati dell'ambiente e delle condizioni iniziali di simulazione al tempo t_0.

2. Per ogni passo temporale Δt: risoluzione delle equazioni SWE in modalità parallela e generazione delle matrici rappresentanti il nuovo stato al tempo: $t_0 + step * \Delta t$. Il formato delle matrici é binario: i dati sono leggibili attraverso software appositi.

3. Terminata la generazione dei file simulazione, per ogni passo temporale le matrici di misurazioni sono convertite in formato ASCII.

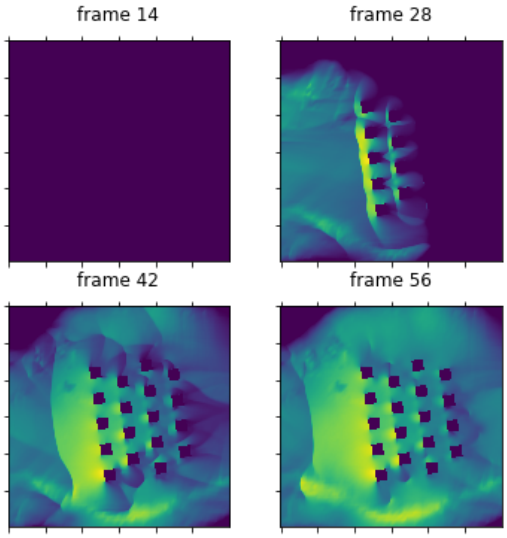


Figura 2.4: Variazione nel tempo delle altezze di fluido simulando l'inondazione di un bacino con ostacoli.

I parametri rilevanti forniti al simulatore a priori sono i seguenti:

- *START xxx*: tempo di inizio della simulazione (inizio alle ore xxx)
- *END xxx*: tempo in cui termina la simulazione (fine alle ore xxx)
- *DTOUTPUT xxx*: intervallo di tempo ogni quanto generare i frames della simulazione (in ore)

- *CR xxx*: numero di Courant utilizzato per la definizione del passo di calcolo

La frequenza di campionamento del simulatore $DTOUTPUT$ é stabilita in base al numero di frames desiderati per simulazione. Piú frames permettono di ricavare piú sequenze (o brevi "video") di inondazioni, tuttavia un campionamento eccessivamente frequente risulterà in frames troppo simili ai successivi. Il numero di frames di una simulazione é dato dalla semplice relazione:

$$frames = \frac{END - START}{DTOUTPUT} \tag{2.7}$$

Oltre ai parametri di simulazione, Parflood richiede una descrizione dell'ambiente. Questo é definito dalle matrici in input:

- *Batimetria (BTM)*: mappa del terreno, ossia una matrice le cui celle indicano l'elevazione del terreno in metri.
- *Scabrezza (MAN)*: rugosità della superficie. É anche definita *scabrezza relativa* quando confrontata al diametro di una condotta.
- *Condizoni iniziali (INH)*: matrice con elevazione del fluido e flag asciutto/bagnato per ciascuna cella.
- *Condizioni al contorno (BLN)*: per ogni cella é indicata la tipologia di contorno con un valore flag (onda in ingresso, muro, piana di deflusso). In caso di flussi in ingresso e in uscita, il file *BCC* specifica le misurazioni, segue un esempio:

tempo $[s]$	livello idrico $[m]$	portata $[m^3/s]$
0	130	50
18000	130	50
36000	200	500
54000	130	50
90000	130	50

- *Velocità iniziali (VHX, VHY)*: matrici distinte con ciascuna cella indicante la velocità del fluido e la direzione in metri/secondo.
- *Creazione della breccia (BRE)*: file contenente le coordinate geografiche della breccia rispetto alla mappa BTM, le misure della superficie della condotta, il tempo di inizio dell'afflusso e quindi il tempo di fine per il deflusso.
- *Griglia multi-risoluzione (PTS)*: questa matrice specifica le differenti risoluzioni per cella in una griglia, seguendo il protocollo *BUQG*.

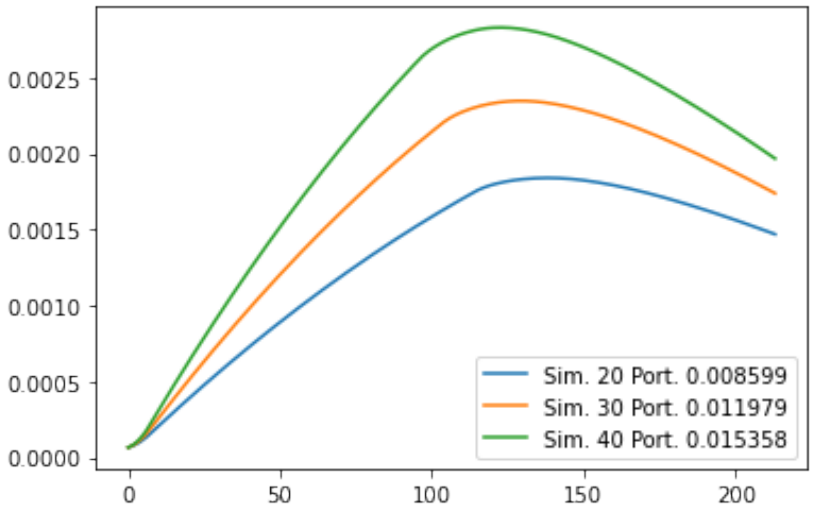


Figura 2.5: Andamento dell'altezza di fluido media (DEP) rispetto al timestep di simulazione. Sono confrontate tre simulazioni con portata media della breccia iniziale variabile (BCC).

2.3.5 Risultati della simulazione

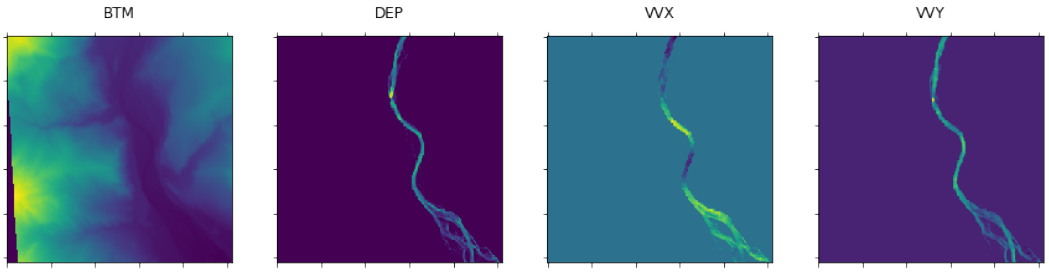


Figura 2.6: Illustrazione delle mappe fondamentali per descrivere un istante di simulazione t_i

Partendo dal tempo iniziale t_0 (START), Parflood calcola ogni Δt secondi (DTOUTPUT) le matrici rappresentanti lo stato della simulazione. Nel dettaglio, lo stato dell'ambiente simulato é descritto da quattro matrici:

- *Profondità (DEP)*: altezza delle colonne d'acqua per ciascuna cella.
- *Velocità X (VVX)*: velocità orizzontale del fluido per cella.
- *Velocità Y (VVY)*: velocità verticale del fluido per cella.
- *Batimetria (BTM)*: mappa del fondale, considerata costante lungo tutta la simulazione.
- *Altezze massime (MAXWSE)*: matrice contenente le massime altezze di fluido per ciascuna cella (Max Water Surface Elevation), calcolata al termine della simulazione

Una simulazione completa puó richiedere numerose ore, in base ai parametri e agli input del simulatore. Risulta inoltre oneroso, dal punto di vista computazionale, ottenere delle previsioni in tempo reale: una o piú GPU sono utilizzate per calcolare le equazioni SWE ad ogni istante temporale. L'obiettivo di questa tesi consiste nel proporre un algoritmo efficiente e accurato con l'ausilio del Deep Learning. Imparando dai dati del simulatore, il nostro modello predittivo punta a fornire previsioni dalle stesse matrici in input, sostituendo del tutto Parflood.

Capitolo 3

Fondamenti di reti neurali

3.1 Il neurone biologico

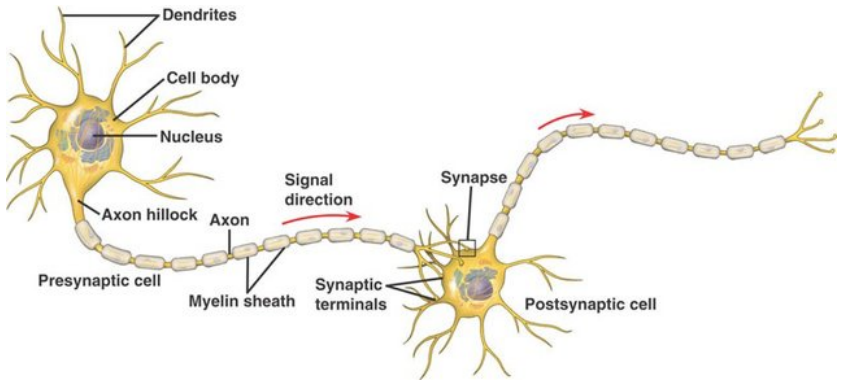


Figura 3.1: Struttura di un neurone e sinapsi

Una rete neurale nasce per simulare gli agglomerati di cellule nervose, i neuroni, presenti nel cervello umano. Un neurone é costituito da piú parti: i *dendriti*, ossia le diramazioni che ricevono i segnali dalle *sinapsi*; un nucleo, chiamato *soma*, in cui vengono processati e prodotti i segnali ricevuti e da trasmettere; l'*assone*, il condotto principale per la trasmissione di impulsi, protetto dalla *guaina mielinica*; infine il *terminale assonico*, ossia le diramazioni finali che dall'assone si collegano verso cellule muscolari, ghiandole o altri neuroni. Piú neuroni trasmettono impulsi di segnale "sparando" uno

dopo l'altro, questi segnali sono innescati da molecole chiamate *neurotrasmettitori*: vengono rilasciate nel terminale assonico (zona presinpatica) e sono assorbite dai dendriti del neurone successivo (zona postsinaptica). Entrati nel nucleo, i neurotrasmettitori possono avere un effetto *inibitorio* (il neurone non "spara" a sua volta, son presenti ioni di potassio) o *eccitatorio* (il neurone piú probabilmente manderún nuovo impulso, son presenti ioni di sodio). Esistono varie tipologie di neuroni nel cervello umano, agglomerati di queste cellule compongono le zone dell'encefalo, ciascuna specializzata a una funzione del corpo umano.

3.2 Il neurone artificiale

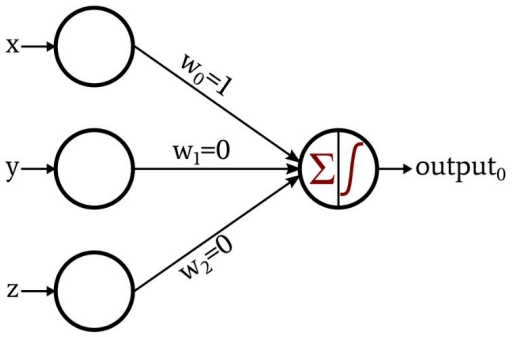


Figura 3.2: Struttura di un *Perceptron*

Una rappresentazione semplificata e virtuale del neurone biologico é il *perceptron* [10], a cui assomiglia per vari aspetti:

- Zona presinaptica: un perceptron riceve molteplici segnali x, y, z dai neuroni precedenti, ciascuno con un peso w_i.
- Nucleo: alla somma pesata dei segnali puó essere aggiunto un bias b, ossia un'informazione aggiuntiva influenzante. Il risultato é processato da una funzione di attivazione f, lineare o non lineare, il cui comportamento dipende dal tipo.

- Zona postsinaptica: nelle reti neurali classiche, il risultato della funzione di attivazione viene propagato verso *tutti* i neuroni allo strato successivo.

Presentato un input, la rete neurale effettua un *forward pass*, ossia le informazioni vengono processate strato per strato secondo le operazioni appena descritte, riassumibili con la formula:

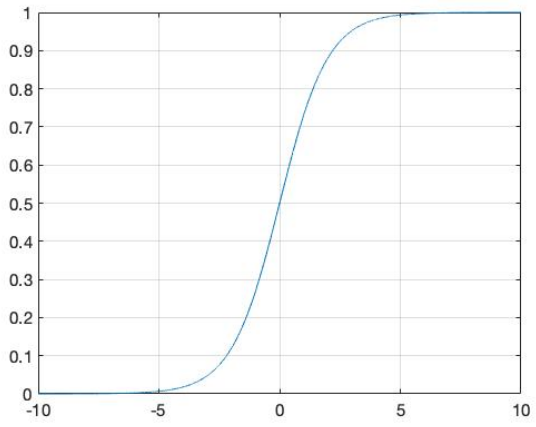
$$y = f(b + W^\top X) \quad , \quad W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad , \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{3.1}$$

3.2.1 Funzioni di attivazione

Alla somma pesata delle features in input viene applicata una funzione matematica che fornisca valori piú "regolari". Seguono le funzioni di attivazione piú diffuse e scelte per questo studio :

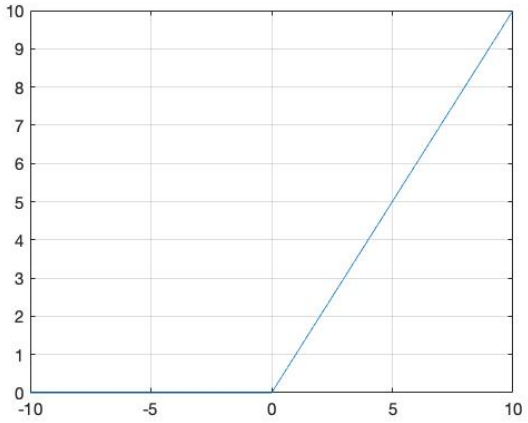
- *Funzione sigmoid*: gli input sempre piú distanti da zero sono "schiacciati" agli estremi dei valori dell'immagine: [0, 1].

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$



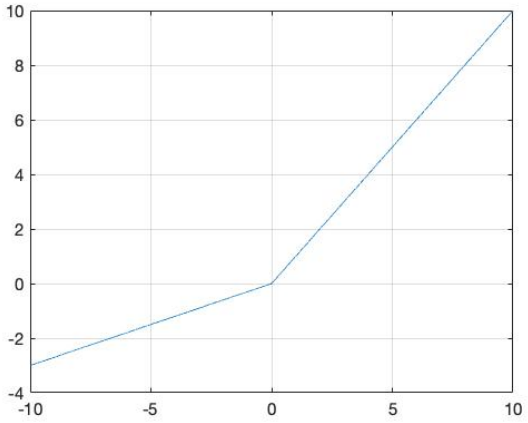
- *Funzione ReLU* (Rectified Linear Unit): gli input positivi sono lasciati invariati, quelli negativi vengono "appiattiti" a zero. L'immagine varia in: $[0, +\infty]$.

$$f(x) = max(0, x) \qquad (3.3)$$



- *Funzione LeakyReLU*: a differenza della ReLU, i valori negativi sono scalati con un coefficiente a, l'immagine quindi varia in: $[-\infty, +\infty]$.

$$f(x) = max(0, x) + a * min(0, x) \qquad (3.4)$$



Maggiore attenzione si é recentemente concentrata sulla funzione di attivazione *ReLU* e le sue varianti: é semplice da calcolare assieme alla sua derivata, quindi si ottengono vantaggi sul tempo di training e inferenza della rete; la sua natura permette inoltre di ovviare il problema dei *vanishing gradients*, riscontrato con altre funzioni tra cui la sigmoide.

3.2.2 Reti di neuroni artificiali a piú strati

Partendo dal perceptron, la singola unità di computazione, si possono unire piú celle in un *layer* (i neuroni dello stesso layer non sono connessi tra di loro) e connettendo questi ultimi si genera una *rete neurale* o **Multi-Layer Perceptron (MLP)**.

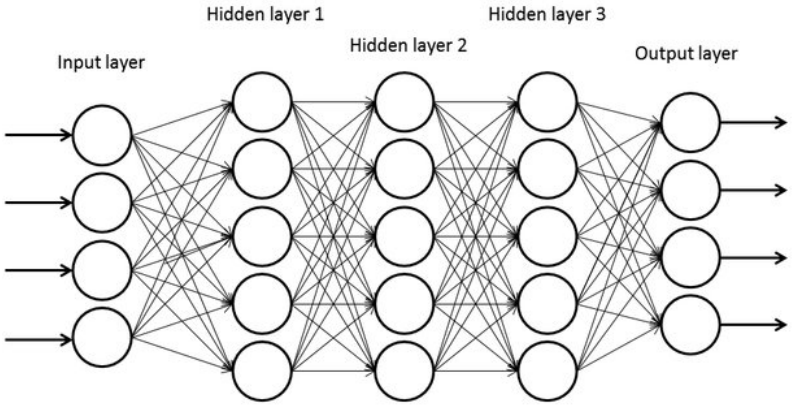


Figura 3.3: Deep Neural Network e tipologie di layer

Il primo strato della rete neurale (graficamente il piú a sinistra) é chiamato *input layer*: il numero di neuroni di questo strato corrisponde al numero di *features* per ogni esempio fornito in input (es: se la rete neurale analizza immagini 20x20, i neuroni sono 400, uno per pixel). L'ultimo strato della rete (graficamente il piú a destra) é l'*output layer*, il cui numero di neuroni rappresenta il numero di features predette (es: per classificare le immagini tra cane e gatto, i neuroni in output saranno 2). Gli strati interni sono chiamati *hidden layers*, all'aumentare della loro quantità si parla di **Deep Neural Network (DNN)**. I layer interni svolgono il ruolo fondamentale di estrarre le informazioni cruciali dagli input e processarle in base alla loro natura. Anticipando i prossimi capitoli, elenchiamo tipologie di layer popolari: layer *ricorrenti* (per input in sequenza temporale), layer *convoluzionali* (per le immagini) o layer *attenzione* (nuova alternativa ai due layer precedenti).

3.3 Addestrare una rete neurale

La rete neurale processa gli input forniti attraverso i singoli perceptrons e i pesi dei collegamenti tra di essi. Questi pesi, inizialmente casuali, costituiscono i parametri con i quali gli input saranno trasformati nelle previsioni: l'addestramento della rete neurale consiste nella correzione di questi al fine di minimizzare la differenza tra le previsioni $\hat{Y}_i$ e gli output attesi Y_i, misurata attraverso una *error function* (funzione di errore). La funzione di errore di base per questo progetto, oltre che la piú diffusa, é la *Mean Squared Error* (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 \tag{3.5}$$

Il set di dati utilizzato (o *dataset*) é composto da coppie (X_i, Y_i), dove X_i é il vettore di features e Y_i il vettore dei valori *target* (o risultati, o etichetta) per la previsione. Una rete neurale puó apprendere osservando anche piú esempi contemporaneamente: questo permette di dedurre dei pattern con maggior capacità di generalizzazione e maggior velocità; piú osservazioni sono quindi accodate in un *batch* (letteralmente "lotto" o "gruppo").
Per poter valutare la rete neurale e assestare la sua robustezza, si effettua il *train-test splitting*: il dataset é suddiviso in una porzione su cui addestrare il modello e in un'altra, mai vista in precedenza, su cui valutare le sue performance. Per ciascun gruppo di osservazioni:

1. Un batch é fornito in input alla rete neurale
2. La rete neurale genera il corrispondente batch di previsioni
3. Si calcola l'errore di previsione rispetto ai target Y e alle previsioni $\hat{Y}$ attraverso un'apposita *loss function*
4. Si aggiornano i pesi della rete neurale in base al risultato della loss function

Si allena quindi il modello su tutti i batch del dataset, costituendo un'iterazione di addestramento chiamata *epoch* (o "epoca"), piú iterazioni permetterano alla rete neurale di migliorare le capacità predittive.

3.3.1 Algoritmi di discesa del gradiente e Backpropagation

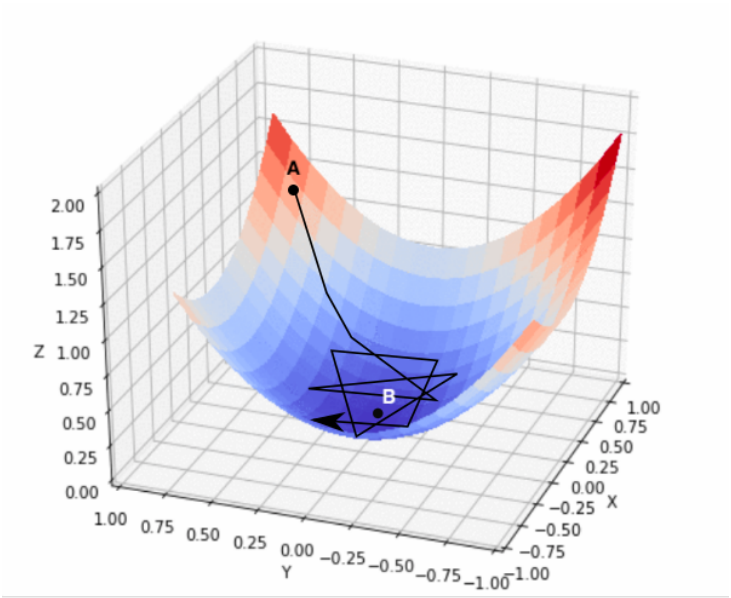


Figura 3.4: Visualizzazione di una error function convessa e discesa del gradiente

Alla base del meccanismo di correzione dei pesi della rete neurale vi é un algoritmo di *backpropagation*: i pesi della rete vengono aggiornati attraverso la propagazione di "aggiustamenti" dall' ultimo layer verso i primi di input, contrariamente rispetto al *forward pass*.

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma \nabla F(\mathbf{w}_n) \tag{3.6}$$

Le correzioni dei pesi sono calcolate in base al valore del peso attuale w_n, un learning rate γ (parametro di training) e ∇F il vettore di derivate parziali della *error function*:

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix} \tag{3.7}$$

La funzione errore F é infatti una funzione convessa, minimizzare l'errore consiste nel trovare il punto di minimo (graficamente un avvallamento, Figura 5.2). Il gradiente di F rappresenta la direzione di massima pendenza da un certo punto $F(a)$, pertanto si considera $-\nabla F$ per raggiungere il *minimum*‘, per tal motivo questa tecnica é chiamata **gradient descent** ("discesa del gradiente"). Il learning rate γ indica "l'ampiezza del passo di convergenza": se troppo elevato puó causare divergenza, nel caso opposto il training richiederà tempi piú lunghi.

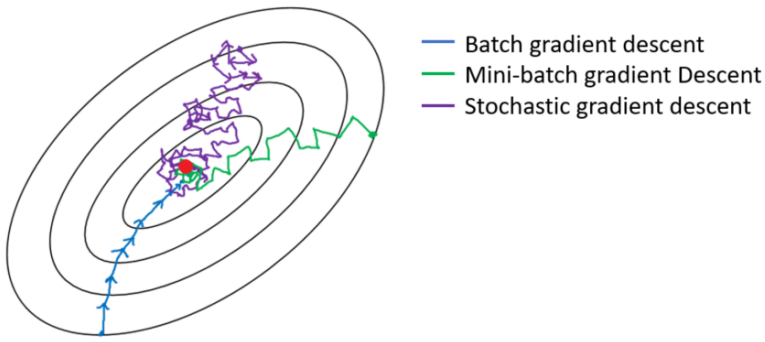


Figura 3.5: Confronto della convergenza degli algoritmi di gradient descent

In base al momento in cui aggiornare i pesi della rete neurale, esistono tre principali algoritmi di gradient descent (Figura 4.12):

- *Batch gradient descent*: l'errore viene calcolato per ogni osservazione nel dataset, i pesi sono tuttavia aggiornati dopo che tutto il training set é stato processato. Questo metodo rende la convergenza stabile ma non ottimale.

- *Stochastic gradient descent (SGD)*: i pesi vengono aggiornati una volta calcolato l'errore per ogni osservazione. Non é necessario quindi tenere in memoria una grossa mole di dati, tuttavia l'errore tenderà ad oscillare spesso a causa della frequenza di aggiornamento dei gradienti.

- *Mini-batch gradient descent*: tecnica ottimale, l'aggiornamento dei pesi avviene dopo aver calcolato l'errore per un insieme di osservazioni (mini batch).

3.3.2 Problemi di diffusione del gradiente

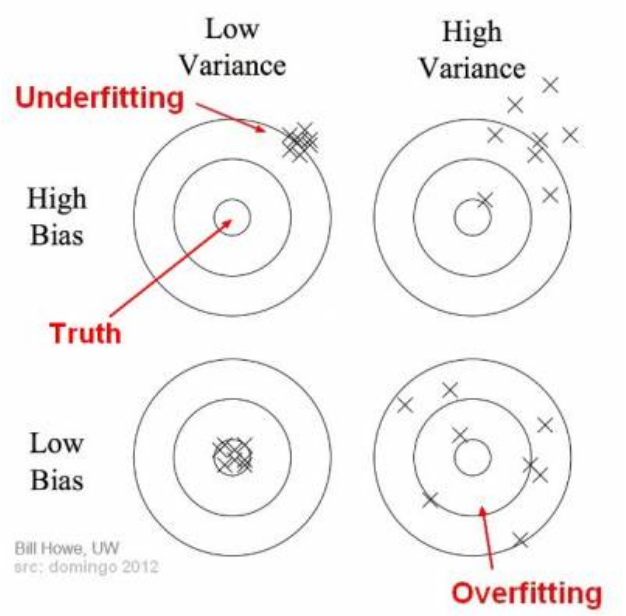
Esistono dei casi estremi in cui l'aggiornamento dei pesi con i gradienti puó fallire. Un primo problema sorto é quello dei **vanishing gradients**: il vettore gradiente ∇F contiene valori sempre piú piccoli propagandosi verso i layer piú interni, i cui pesi quindi subiscono un aggiornamento minimo se non ignorabile. Le ragioni possono sono molteplici:

- Rete neurale troppo profonda: un numero elevato di layers puó causare la "frammentazione" dei gradienti. Una soluzione é l'utilizzo di layer differenti, quali i layer residuali [11].

- Funzione di attivazione: le funzioni *sigmoid* e *tangente iperbolica* possono causare questo problema in quanto "schiacciano" i valori agli estremi. La funzione $ReLU$ e le sue varianti sono utilizzate per ovviare questo problema.

Al contrario, i gradienti possono raggiungere valori talmente alti da "esplodere", o non essere piú rappresentabili (**exploding gradients**). Questo puó essere causato da numeri in input troppo elevati o dall'accumulazione di gradienti nelle *reti neurali ricorrenti* (RNNs). Per ovviare questo problema, é possibile regolarizzare i pesi della rete neurale, applicare una normalizzazione sui dati in input (*batch normalization*) o limitare superiormente il valori dei gradienti (*gradient clipping*).

3.3.3 Overfitting e regolarizzazione



Un problema comune e presente anche in questo studio é l'**overfitting**: la rete neurale manca di generalizzazione e impara a prevedere molto bene solo rispetto ai samples del training set. L'overfitting é descritto generalmente da un bias basso e da una varianza alta: con **bias** si intende la divergenza tra le previsioni e i target; con **varianza** si intende la variabilitá delle previsioni in base agli input, piccoli cambiamenti nei dati possono portare a previsioni molto diverse se la varianza é alta. L'overfitting deriva anche dalla scelta del modello: un modello eccessivamente flessibile cerca pattern piú complessi, imparando anche da informazioni non utili introdotte nei dati (rumore). Cercando di mantenere il modello semplice, si forza la rete neurale ad apprendere i pattern principali al fine di minimizzare la funzione errore; questa tecnica é chiamata **regolarizzazione.** Regolarizzare un modello consiste nell'aggiungere alla loss function il "costo" dei pesi della rete neurale, in questo modo si punta a minimizzarli e quindi a non apprendere una rappresentazione troppo complessa degli inputs.

Capitolo 4

Reti neurali profonde e analisi dello spazio-tempo

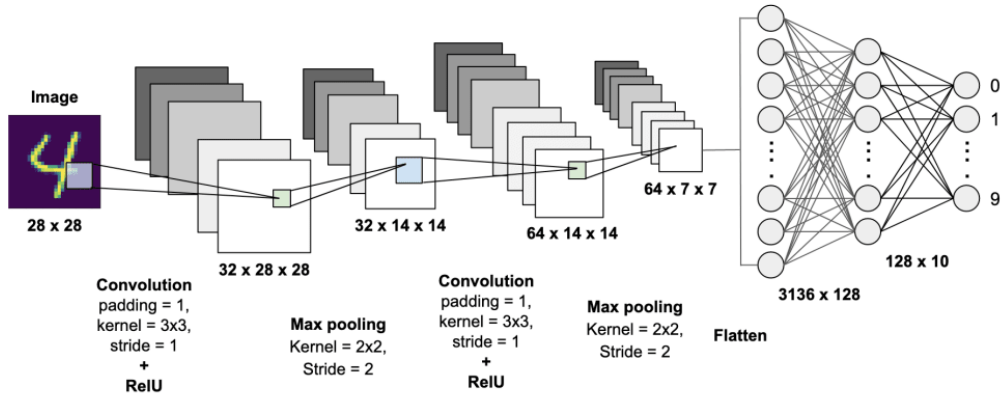


Figura 4.1: Architettura di una Convolutional Neural Network per la classificazione dei caratteri trascritti (MNIST dataset)

Le reti neurali condividono con altri algoritmi la problematica del **curse of dimensionality**: per problemi sempre piú complessi, la dimensionalitá dei dati e il costo per processarli crescono, ma non é garantito che l'informazione utile aumenti a sua volta, presentando rumore e ridondanza. Quando il numero di features é troppo elevato, diventa quindi necessario individuare le caratteristiche piú importanti e generiche per poter trarre un senso dal da-

taset e comprendere le relazioni al suo interno. Nell'ambito della **computer vision**, algoritmi applicati alle immagini e ai video, le **deep convolutional neural networks** (Deep CNNs) si occupano di questo problema.

4.1 Features spaziali: layer convoluzionali e iperparametri

Una CNN é composta da layer **convoluzionali**: analogamente ai neuroni biologici nella corteccia visuale, questi layer analizzano progressivamente porzioni dell'immagine di dimensioni pari al *receptive field* (campo recettivo) attraverso "filtri" chiamati **kernel**:

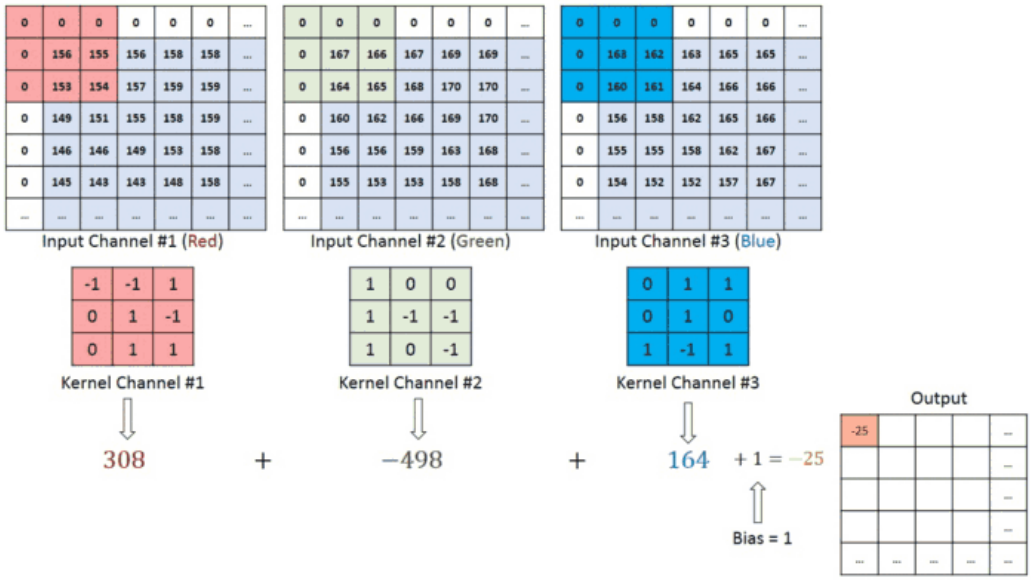


Figura 4.2: Applicazione di un kernel in un layer convoluzionale 2D su immagini RGB

Il kernel é una matrice di dimensione fissa (solitamente 3x3, 5x5 o 7x7, numeri dispari per avere sempre elementi centrali) e profondità pari al numero di canali dell'immagine. Un kernel é costituito da pesi modificabili in fase di

training (*learnable*), l'operazione di *convoluzione* consiste nell'applicare il kernel dall'angolo top-left dell'immagine:

1. Prodotto scalare tra la matrice kernel K e la porzione di immagine I a cui si sovrappone il kernel.

2. Sommatoria dei prodotti per ottenere una cella $O_{u,v}$ della matrice output O:

$$O_{u,v} = \sum_{i=1}^{n} \sum_{j=1}^{n} F_{i,j} I_{i,j} \tag{4.1}$$

3. Se l'immagine possiede piú canali (es: RGB: red, green, blue), le operazioni *1* e *2* sono applicare per ogni canale, quindi i singoli valori sono sommati.

4. Il kernel scorre verso la prossima porzione di immagine con un passo, o **stride**, predefinito, quindi si ripete l'intera procedura.

Completata un'operazione di convoluzione, l'immagine in input al layer é trasformata in una matrice di dimensioni pari o ridotte (estrazione delle features) e un numero di canali spesso maggiore; questo permette ai livelli successivi di dedicare piú kernel a delle potenziali caratteristiche dell'immagine da analizzare.

4.1.1 Dimensione e padding

Per ottenere la dimensione desiderata della matrice output, é possibile espandere le dimensioni della matrice input aggiungendo celle nulle ai bordi (**padding**) in due modi:

- *Same padding*: considerando ad esempio una matrice 5x5x1, si estendono larghezza e altezza a 7x7x1 con padding uguale a 1. Applicando un kernel 3x3x1, il risultato é una matrice 5x5x1, esattamente come quella in input.

- *Valid padding*: se non si applica alcun padding, la matrice assume dimensione pari o multiplo rispetto al kernel.

La *dimensione del kernel*, lo *stride* e il *padding* sono definiti **iperparametri** di una CNN. É possibile aggiustarli stimando la dimensione della matrice risultato attraverso la formula:

$$O_{size} = \frac{w - k + 2p}{s} + 1 \tag{4.2}$$

Dove w é la larghezza dell'immagine (width), k é la dimensione del kernel, p il padding scelto ($2p$ per i due bordi) ed s lo stride.

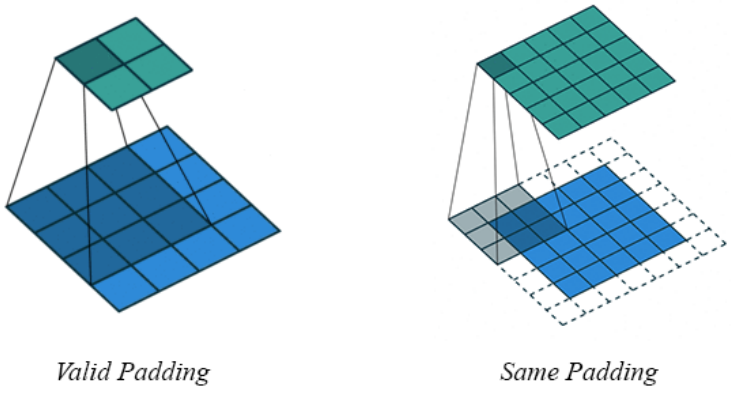


Figura 4.3: Visualizzazione e confronto delle tecniche di padding

4.1.2 L'operazione di pooling

Al fine di ridurre la dimensionalità dell'immagine in input, un layer convoluzionale puó essere seguito da un layer di pooling. Similarmente alla convoluzione, si effettuano operazioni scorrendo un filtro (o kernel) sulla matrice in input. Esistono due tipologie:

- *Average Pooling*: restituisce la media dei valori delle celle nel kernel.

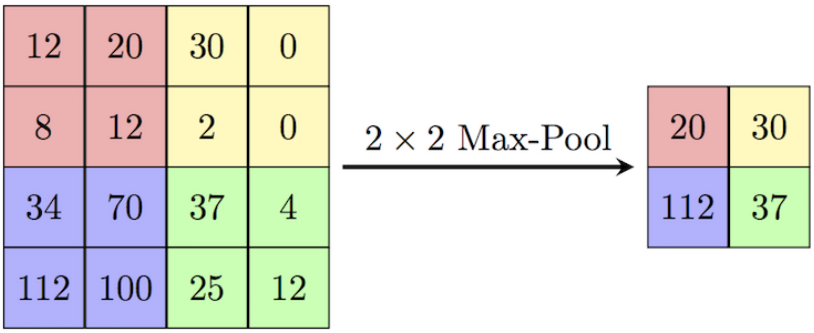


Figura 4.4: Illustrazione dell'operazione di max pooling con kernel size 2x2

- *Max Pooling*: restituisce il valore massimo tra le celle nel kernel.

Considerando una kernel size 2x2, la dimensione della matrice in input sarà quindi ridotta per due fattori. La scelta di ridurre la dimensionalità dell'input con queste operazioni é il frutto di un compromesso tra il costo computazionale e la qualità delle previsioni: semplificare l'immagine con valori medi/massimi locali puó comportare la perdita di informazioni minime ma importanti. Nel caso di questo studio: nella fotografia aerea di un fiume anche poche celle, che rappresentano un edificio o un corpo solido, influenzeranno la direzione del fluido, la velocità e la profondità.

4.2 Features temporali: Video Generation e modelli seq2seq

Definiti gli elementi per estrarre features nello spazio, é necessario discutere l'aspetto temporale: in un video possono esistere relazioni tra pixel di un fotogramma e quelli di un fotogramma successivo o precedente nel passato (es: analizzando il rimbalzo di una palla, la sua traiettoria dipende dalla posizione di provenienza). Un video consiste in una sequenza di frames, il modello dovrà quindi essere in grado di leggere array di immagini per generare il futuro: le reti neurali che si occupano di questo problema si chiamano modelli **seq2seq** (sequence-to-sequence). Nel caso del processing di testi in

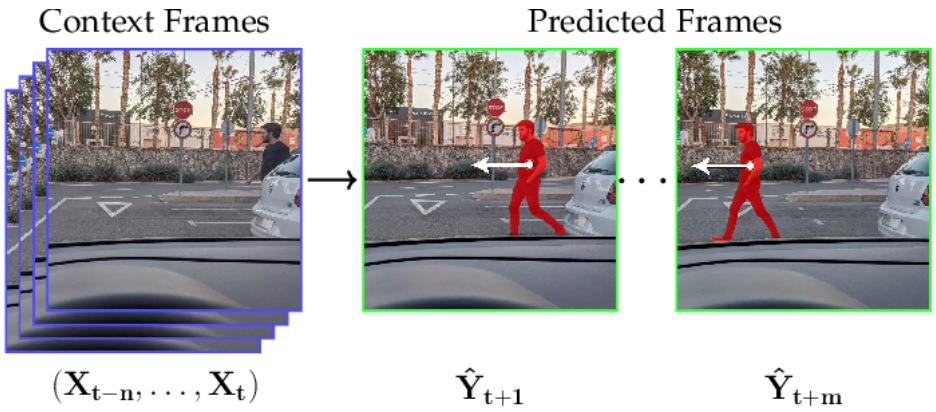


Figura 4.5: Input, target e previsioni in un modello sequence-to-sequence per la previsione dei video.

una lingua umana (NLP, Natural Language Processing), i modelli seq2seq sono utilizzati per la traduzione delle lingue o per creare agenti in grado di rispondere a delle domande (chatbots). Un modello seq2seq é suddiviso in tre parti principali:

1. *Encoder*: codifica degli input in una rappresentazione compressa
2. *Encoder embedding vector*: codifica la sequenza intera di rappresentazioni compresse in un unico vettore rappresentativo
3. *Decoder*: decodifica il vettore embedding in una sequenza output

In un modello seq2seq per il linguaggio, ogni parola é codificata da un numero o un vettore (word embedding), la sequenza di parole é "compressa" dalla rete neurale in un vettore rappresentativo per poi essere decodificata in una sequenza output. Applichiamo analogamente lo stesso principio ai video, con la differenza che ciascun frame (un'immagine) necessita di essere trasformato in una rappresentazione (features estratte) con un encoder apposito, ad esempio una rete convoluzionale; per i motivi appena indicati si introduce una tipologia di layer in grado di codificare sia features spaziali (pixel delle immagini) che temporali (la sequenza di frames): i layer **ConvLSTM** [12].

4.2.1 Convoluzione ricorrente: ConvLSTM

In generale, per ricordare il cambiamento degli input nel tempo esistono i **le reti neurali ricorrenti**: un neurone (cella) non ha solo un input x_t e un output h_t, ma anche uno stato interno c_t che tiene conto dei valori forniti in precedenza. Tra le varie tipologie di layer ricorrenti si distinguono quelli con celle **Long Short Term Memory** (LSTM), in grado di gestire sequenze molto lunghe.

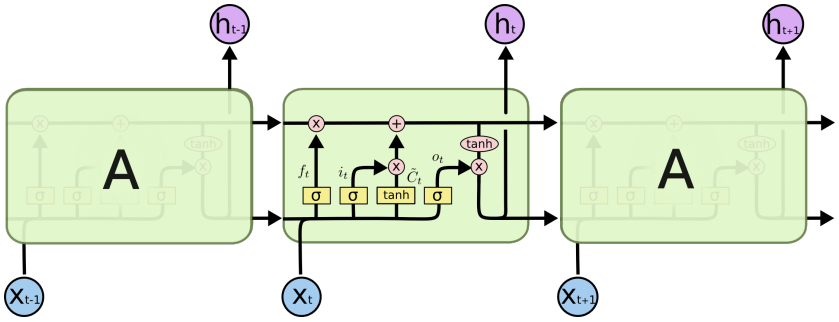


Figura 4.6: Illustrazione di una cella ricorrente LSTM "srotolata" nel tempo.

Per prevedere i frames in un video si ricorre a una variante di celle LSTM: al posto di singoli numeri, il layer ConvLSTM riceve matrici 2D (le immagini), effettua una convoluzione 2D e processa la sequenza di immagini nel tempo come un classico layer LSTM. Si denotano con "$*$" l'operazione convoluzione, "$\circ$" il prodotto Hadamard e σ la funzione attivazione *sigmoide*. Inoltre, W_* sono kernel (matrici con pesi apprendibili) e b_* sono dei *bias*.

1. **Forget gate layer**: f_t consiste nella quantità di informazioni da memorizzare rispetto all'input X_t attuale concatenato con l'output H_{t-1} della cella precedente moltiplicato per il suo stato interno C_{t-1}.

$$f_t = \sigma\left(W_{xf} * X_t + W_{hf} * H_{t-1} + W_{cf} \circ C_{t-1} + b_f\right) \tag{4.3}$$

2. **Input gate layer**: da x_t e h_{t-1} si deriva il nuovo stato interno C_t (equazione 4.5) con la funzione attivazione *tanh* (hyperbolic tangent).

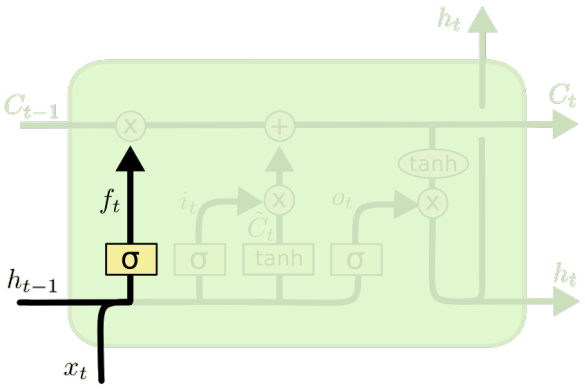


Figura 4.7: Illustrazione del forget gate layer nella cella LSTM.

Questo stato é moltiplicato per l'input gate layer i_t (equazione 4.4) per determinare la quantità di informazioni in input da ricordare; i_t é infine sommato allo stato della cella precedente C_{t-1}.

$$i_t = \sigma\left(W_{xi} * X_t + W_{hi} * H_{t-1} + W_{ci} \circ C_{t-1} + b_i\right) \tag{4.4}$$

$$C_t = f_t \circ C_{t-1} + i_t \circ \tanh\left(W_{xc} * X_t + W_{hc} * H_{t-1} + b_c\right) \tag{4.5}$$

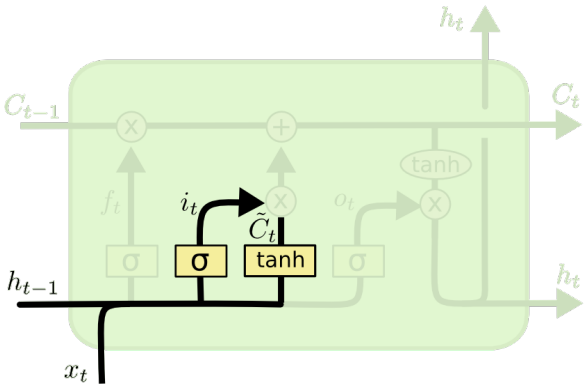


Figura 4.8: Illustrazione dell'input gate layer nella cella LSTM.

3. **Output gate**: con la funzione sigmoide σ si filtrano le informazioni da far passare da x_t e h_{t-1} ottenendo o_t, questo si moltiplica per la

funzione tangente iperbolica ("spinge" i valori in un intervallo tra -1 e 1) dello stato della cella attuale C_t, ottenendo H_t.

$$o_t = \sigma\left(W_{xo} * X_t + W_{ho} * H_{t-1} + W_{co} \circ C_t + b_o\right) \tag{4.6}$$

$$H_t = o_t \circ \tanh\left(C_t\right) \tag{4.7}$$

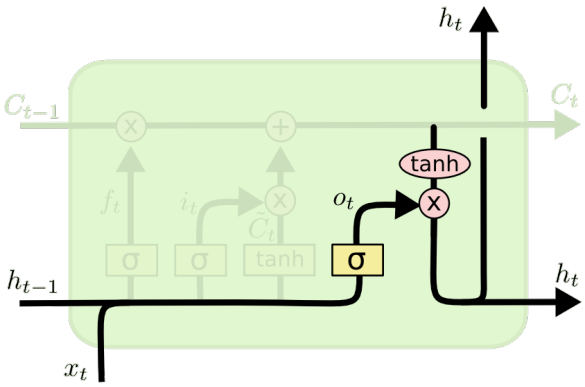


Figura 4.9: Illustrazione dell'output gate layer nella cella LSTM.

Gli output di un layer t-esimo ConvLSTM sono quindi la matrice H_t e lo stato interno C_t; si noti che rispetto ad una cella semplice LSTM, si effettuano convoluzioni 2D con i kernel W_* sugli input e gli output, che sono immagini.

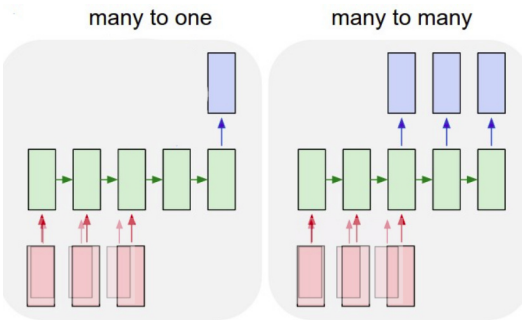


Figura 4.10: Confronto tra due tecniche di previsione delle sequenze con celle LSTM.

É possibile ottenere output di due tipi:

- *many-to-one*: da una sequenza di n frames in input si genera il prossimo frame $n+1$. Per prevedere piú frames si concatena la previsione alla sequenza iniziale e si fornisce nuovamente al modello (**modalità autoregressiva**).

- *many-to-many*: si aumenta il numero di celle LSTM con k nuovi elementi, quindi ciascuno genererà il frame $n+i$ della sequenza futura.

In questa tesi ci si concentrerà sulla prima modalità, al fine di irrobustire la rete neurale a prevedere il futuro basandosi sulle sue stesse previsioni, quindi in modalità autoregressiva.

4.2.2 Layer convoluzionali 3D

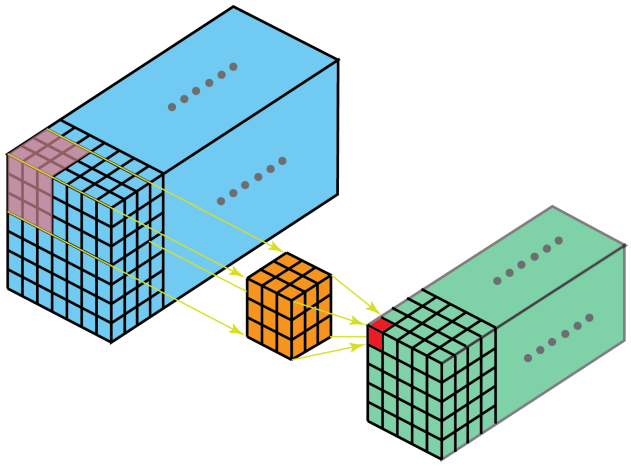


Figura 4.11: Convoluzione 3D: applicazione di un kernel 3x3x3 su una sequenza di immagini

Analogamente alla convoluzione 2D, esiste l'operatore convoluzione 3D per processare sequenze di immagini. In questa tesi, utilizziamo i layer Conv3D per convertire il vettore *embedding* (ossia il vettore che codifica la frequenza di frames in input) nelle previsioni finali. Il vettore embedding é fornito nella forma *batch, frame, canali interni, altezza, larghezza*, dove il numero

di canali interni corrisponde alla dimensionalità dello spazio latente (numero di kernels negli hidden layers). Il kernel ha tre dimensioni: si effettua quindi una convoluzione anche in profondità, nella sequenza dei frames. Nella soluzione proposta, il layer Conv3D trasforma il vettore embedding nella sequenza di frames predetta con il numero di canali richiesto (dep, vvx, vvy).

Questa nuova tipologia di layer é implementata in architetture allo stato dell'arte per la segmentazione delle immagini (ossia identificazione e contornatura delle entità in un'immagine),la classificazione delle azioni illustrate in un video o la previsione dei frames futuri. Per quest'ultimo task, la convoluzione 3D costituisce il layer di base ad esempio per l'architettura *CrevNet* [13].

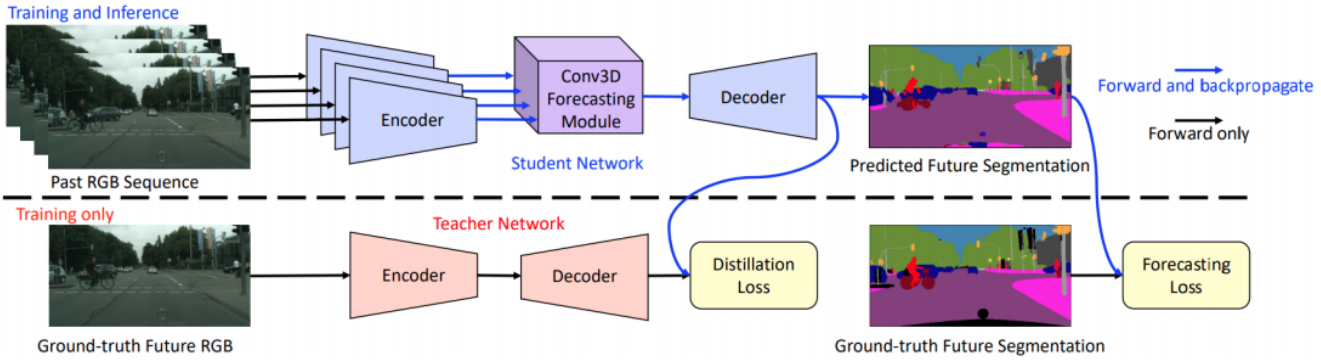


Figura 4.12: Architettura proposta da Chiu et.al [12], composta da 2 auto-encoders (student e teacher network) con blocchi conv3D.

Capitolo 5

Esperimenti e implementazione della soluzione

5.1 Stato dell'arte e formulazione della soluzione

Varie architetture di reti neurali sono state proposte in passato per apprendere le equazioni SWE, principalmente composte da layer feed forward e convoluzionali. Con l'obiettivo di prevedere i livelli d'acqua nel futuro, sono state proposte diverse formulazioni del problema differenziati dalla dimensionalità del dati:

- Nel 2018, M.Varesi [14] propone una rete neurale feed forward a 5 strati: sono fornite in input 10 misurazioni di altezza d'acqua a monte e si cerca di prevedere il livello a valle, in un punto piú a sud lungo il fiume. Il target di previsione é uno scalare e le condizioni ambientali non sono parametri forniti al modello.
- Nel 2020, Kabir et.al[15] propongono una rete convoluzionale 2D per analizzare la serie temporale di un vettore di misurazioni. In questo caso, si utilizzano layer convoluzionali 2D lungo la serie di misurazio-

ni, anziché estrarre features da un'immagine. Questo approccio viene introdotto da Kiranyaz et.al[16] per la regressione di serie storiche.

- Nel 2021, S.Gazza[17] implementa una rete convoluzionale 2D per prevedere i livelli di fluido in matrici 2D. I layer convoluzionali sono utilizzati come feature extractors della matrice all'istante precedente, quindi la rete apprende le trasformazioni per ottenere l'istante successivo. Anche in questo caso, le condizioni ambientali non sono parametrizzate.

Questo progetto di tesi propone avanzamenti nella ricerca su due fronti: dal punto di vista della dimensionalità dei dati, l'allagamento del terreno é rappresentato da fotogrammi (matrici), quindi l'obiettivo é generare il futuro del video (**Video Generation**); circa la varietà dei dati, si tengono in considerazione il terreno, le velocità delle correnti e gli ostacoli. In questo modo, il modello é in grado di generalizzare il comportamento del fluido indipendentemente dalle condizioni ambientali, inoltre é possibile prevedere fattori quali la velocitá delle correnti, sostituendo completamente un simulatore CFD. Il vantaggio atteso consiste principalmente nel tempo di computazione delle previsioni (obiettivo: real-time) anche con condizioni ambientali completamente nuove. Gli avanzamenti discussi in questa tesi sono accompagnati dallo sviluppo di un framework: **deepSWE**. Basata su PyTorch, questa libreria offre gli strumenti di Deep Learning per progettare e addestrare reti neurali sui dati dei simulatori CFD, anche con modelli giá esistenti. Sono inoltre pubblicati i dataset, i modelli pre-addestrati e gli esperimenti condotti nel corso di questo studio.

5.2 Simulazione e generazione del dataset

Il simulatore Parflood permette di generare sequenze di matrici di misurazione per ogni "istante" di un'inondazione. Per modellare il problema come task di previsione di un video, sono state definite delle fasi di generazione e processing dei dati (Figura 5.6):

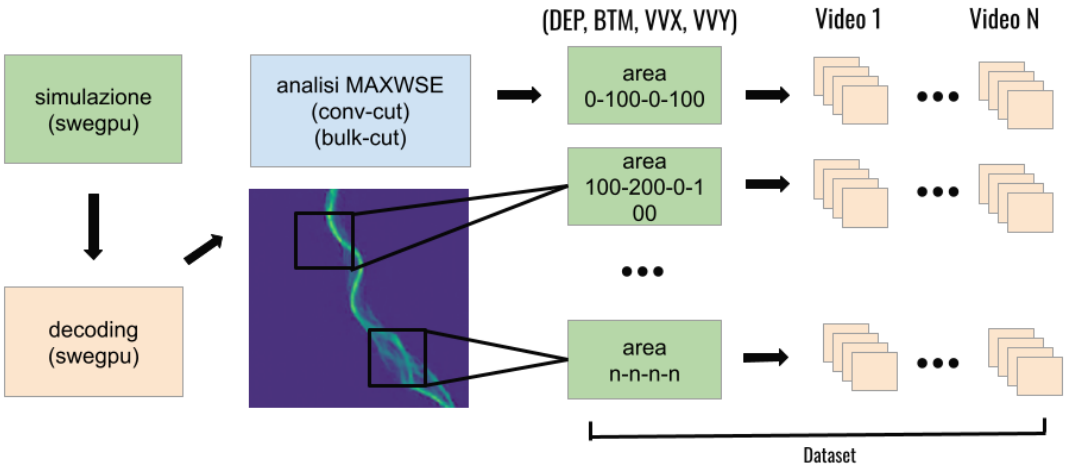


Figura 5.1: Schema di generazione del dataset: simulazione, decoding, filtraggio e sequenzializzazione.

1. *Simulazione*: esecuzione del simulatore Parflood e raccolta delle misurazioni "grezze": BTM, DEP, VVX, VVY, MAXWSE.

2. *Decoding*: conversione dei file di misurazione da binario ad ASCII, le misurazioni sono rappresentate con valori *float32*.

3. *Analisi e sequenzializzazione*: dall'intera regione si selezionano le zone che contengono una quantità sufficiente di fluido da analizzare, quindi sono estratte delle sotto-aree di dimensione fissa (768x768, cioé una griglia 3x3 di celle 256x256).

5.2.1 Simulazione e specifiche hardware

Il simulatore parallelo Parflood consiste in un software eseguibile dipendente dalle librerie NVIDIA per il controllo delle schede grafiche, **CUDA Toolkit** (versione utilizzata in questa tesi: **11.0**). Le simulazioni sono eseguite su una rete di calcolatori (cluster) ad alte prestazioni (High Performance Computing, HPC) fornito dall'università degli studi di Parma. Il cluster é composto dalle risorse per il calcolo, schede grafiche (GPU) e multiprocessori (CPU), e

computer adibiti alla gestione degli utenti e degli esperimenti da eseguire (o *jobs*). Per quest'ultimo compito é predisposto un server **SLURM**, un software di gestione del carico di lavoro e delle risorse (workload manager) per Linux. Ai fini degli esperimenti di questa tesi, sono stati prefissati i requisiti di minimi per le risorse hardware da richiedere a SLURM:

Task	RAM	CPU	Cores	GPU
Training	32-256 GB	XEON E5-2683v4	16	2xNvidia A100
Simulazione	32 GB	XEON E5-2683v4	16	1xNvidia P100

Preparati i file necessari alla simulazione (specificati nel paragrafo 2.3.4), si genera uno script *bash* da eseguire sul server HPC, contenente le specifiche dei requisiti per SLURM:

```
1 #SBATCH --job-name=SWE_full_sim
2 #SBATCH --output=%x.o%j
3 #SBATCH --error=%x.e%j
4 #SBATCH --nodes=1
5 #SBATCH --tasks-per-node=1
6 #SBATCH --gres=gpu:1
7 #SBATCH --partition=gpu
8 #SBATCH --mem=16G
9 #SBATCH --time=24:00:00
```

Successivamente, lo script contiene il comando per avviare la simulazione con **swegpu** (eseguibile per Parflood), definendo una gpu da utilizzare e la risoluzione standard (senza multi-risoluzione):

```
1 ./swegpu $input -order=1 -gpu=0 -multi=hi > "$folder/output_$(
      date '+%Y-%m-%d').txt"
```

5.2.2 Decoding e conversione

Eseguita la simulazione, vengono generati i file con le matrici BTM, DEP, VVX, VVY, MAXWSE. Segue quindi il comando di decodifica per ciascun tipo di file (selezionando tutti i frames, risoluzione originale e output non binario):

```
# iterato per DEP, BTM, VVX, VVY, MAXWSE
./swegpu -decode $folder/simulation/"${filename}0000.*" $folder/
    decoded/decoded -all -frames 0 $last -binary=0 -res=
    $resolution
```

I file generati contengono matrici 768x768 valori in formato *float32*, come visibile in questa sezione 12x5:

```
4.71753e-01 0.00000e+00 0.00000e+00 0.00000e+00 0.00000e+00
9.92509e-01 1.74219e-01 0.00000e+00 0.00000e+00 0.00000e+00
9.63321e-01 9.46960e-01 0.00000e+00 0.00000e+00 0.00000e+00
9.66145e-01 8.77755e-01 8.62921e-01 0.00000e+00 0.00000e+00
1.23323e+00 9.51391e-01 9.75382e-01 8.88735e-01 0.00000e+00
1.19594e+00 1.24599e+00 1.10181e+00 1.08521e+00 7.04663e-01
1.02829e+00 1.26774e+00 1.34795e+00 1.29785e+00 1.13589e+00
9.50625e-01 1.03486e+00 1.24184e+00 1.41947e+00 1.37174e+00
7.10541e-01 8.89352e-01 1.04285e+00 1.32930e+00 1.49813e+00
8.63403e-01 8.63929e-01 9.73405e-01 9.34986e-01 1.30047e+00
1.09732e+00 7.09958e-01 7.50827e-01 9.51506e-01 9.91025e-01
1.12370e+00 1.07250e+00 8.73722e-01 9.12303e-01 1.02149e+00
```

5.2.3 Analisi ed estrazione dai files

In base allo scenario di inondazione, le matrici processate dal simulatore hanno una dimensione variabile. Scegliendo di mantenere la risoluzione nativa in questa tesi, per ciascun terreno il numero di celle varia:

- *Torrente Arda*: 6,784 x 4,602 = 31,219,968 celle

- *Torrente Baganza*: 3,584 x 2,816 = 10,092,544 celle

Una tale quantità di dati richiede un costo computazionale e temporale elevato. Inoltre, non tutte le celle contengono informazioni interessanti: la mappa comprende montagne e terreni senza la presenza di fluido. É stato quindi ideato un semplice algoritmo (chiamato "conv-cut") per selezionare le porzioni di terreno con un minimo volume d'acqua (soglia minima: **0.1 metri**):

regione ← lettura matrice MAXWSE
ampiezza ← 768
soglia ← 0.1
celle minime ← 0.1 * ampiezza^2
stride ← 35
for y ← range(0, altezza regione - ampiezza, stride):
    for x ← range(0, larghezza regione - ampiezza, stride):
        area ← porzione di regione (x : x + ampiezza, y : y + ampiezza)
        area valida ← celle dell'area > soglia
        if num_celle(area valida) > celle minime:
            aree valide.aggiungi((x, y))

Lo script python **conv-cut** passa quindi le coordinate (xmin,xmax,ymin,ymax) delle porzioni valide allo script **bulk-cut**, il quale legge tutti i file decodificati e per ciascuno effettua il cropping con il seguente comando bash:

```
1 for file in *.ESTENSIONE; do cat $file | sed -n $ymin','$ymax'p'
      | cut -d" " -f $xmin-$xmax >> $destdir/$file; done
```

Per ogni terreno le matrici di misurazione sono quindi organizzate nel seguente modo:

- baganza/
  area 1/
  region.BTM

```
    decoded-0000.DEP
    decoded-0000.VVX
    decoded-0000.VVY
    ...
area 2/
    ...
area n/
    ...
```

5.3 Preparazione dei dati

Terminate le fasi di simulazione e deconding, i dati sono raccolti in cartelle in base all'area geografica. Per generare tensors con sequenze di frames sono stati creati due moduli python per la gestione dei dati:

- **DataPartitions**: partiziona i frames logicamente in serie temporali.
- **DataGenerator**: estrae le matrici dai file, genera i tensors e applica delle operazioni di pre-processing.

5.3.1 Organizzazione delle sequenze temporali

Prendendo come riferimento la classe di un dataset in *Tensorflow - Keras*, si crea un modulo per etichettare logicamente i dati di training (features) e i dati da prevedere (target). Applicando il self-supervised learning, i dati utilizzati sono gli stessi: per prevedere un video, i frames sono sia l'input del modello che l'obiettivo. Il simulatore di fluidodinamica genera un'unica sequenza di n frames, ossia un unico "video" che illustra l'andamendo della simulazione, l'allagamento. Spesso la lunghezza di questa sequenza é eccessiva per essere processata in una sola volta, ad esempio: in 24 ore di simulazione, con una frequenza di campionamento di 1 frame ogni 30 secondi, si generano **2880** fotogrammi. Il modulo **DataPartitions** gestisce le

sequenze temporali suddividendo l'array di fotogrammi principale in sotto sequenze con una "**finestra di scorrimento**". Si fissa una lunghezza di 5 frames e si prelevano rispettivamente 4 frame in input e 1 frame target; ci si sposta di un frame in avanti e si genera la successiva sequenza della stessa lunghezza, quindi si ripete il tutto fino a $n - lunghezza$ frames.

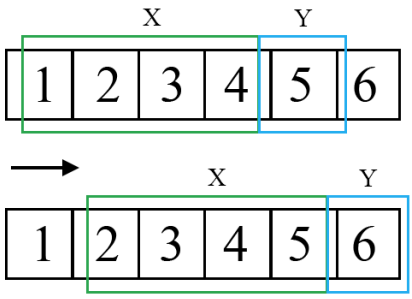


Figura 5.2: Sottosequenze con 4 frames in input e 1 frame target, scorrimento di 1 frame.

I parametri richiesti dalla classe DataPartitions sono quindi:

- *past frames*: numero di fotogrammi in input (passato)
- *future frames*: numero di fotogrammi target da prevedere
- *root*: percorso della cartella contenente tutte le aree e i frames
- *partial*: porzione del dataset da considerare (da 0 a 1) per lavorare su un sottoinsieme di dati

La classe genera delle strutture dati con i seguenti metodi:

- *get_areas()*: restituisce la lista dei nomi delle aree che contengono frames, nella forma: *area-xmin-xmax-ymin-ymax*
- *get_partitions()*: genera una lista di n-uple, ciascuna per area, della forma *(area, X, Y)*, dove X é la lista delle sequenze in input, identificate dall'id del frame di partenza; Y é un dizionario che associa le sequenze in input con le sequenze target, cioé liste di id frames target

Con *partizione* intendiamo una n-upla generata dalla funzione *get_partitions()* che contiene la lista delle sequenze X,Y per ciascuna area geografica; esempio: l'area 0-0-0-0 contiene 40 frames (0-39), si consideri l'ottava sequenza che ha quindi i frames passati 7-13 (6) e i frames futuri 14-16 (4):

```
[
    (
      'area-0-0-0-0',
      [
        ...
        "id-7",
        "id-8",
        ...
      ],
      [
        ...
        "id-7": [14, 15, 16, 16],
        "id-8": [15, 16, 17, 18]
        ...
      ]
    )
]
```

5.3.2 Lettura delle sequenze temporali e ottimizzazione dell'I/O

Il DataGenerator ha il compito di processare le *partizioni* generate e caricare i fotogrammi dal disco alla RAM. Sono state sviluppate due modalità di caricamento: *pre-loading*, tutto il dataset é memorizzato in ram; *"on the fly"*, in fase di training si carica un batch alla volta in memoria, sostituito poi dal batch successivo. Il caricamento dei dati *on the fly* ha permesso di addestrare la rete neurale su un dataset molto esteso, ovviando i limiti di memoria volatile: una matrice float32 da 768x768 pesa 4 x 768 x 768 = 2,359,296 Bytes $\approx$ 2.3 MB; una simulazione contiene in media 120 aree geografiche,

ciascuna con circa 240 fotogrammi a 3 canali (dep, vvx, vvy), quindi 120 x 240 x 3 = 86,400 matrici; il peso complessivo di un dataset é di 2,359,296 x 86,400 = 203,843,174,400 bytes ≈ **203.4 GB**. Generando sotto-sequenze temporali con l'algoritmo a finestra di scorrimento, ogni frame puó ricorrere in memoria piú volte e il peso del dataset aumenta oltre le risorse disponibili. Quando vengono richieste sequenze di frame sovrapposte, é possibile ottimizzare gli accessi con buffer di "cache" e l'impiego dell'algoritmo di **seconda chance**: alla lettura, vengono caricati in memoria i frames, quindi quelli piú frequenti hanno maggiore probabilità di permanere in cache e possono essere letti da RAM anziché dal disco rigido.

5.3.3 Pre-processing dei dati e formattazione

Il nucleo del modulo DataGenerator é la funzione di generazione dei tensors *get_data()*, in cui si esegue la pipeline di pre-processing:

1. Il modulo **DataPartitions** legge le cartelle di misurazioni e genera la lista di aree e sequenze temporal (→ appendice **A.1**, riga **26-56**).

2. Per ogni area si legge la sua mappa BTM. Per avere valori che partano da zero, si effettua uno shift sottraendo il valore della cella minima (→ appendice **A.3**, riga **53-70**).

3. Per ogni frame nella sequenza si leggono le matrici DEP, VVX e VVY effettuando un lookup della cache indicata sopra, leggendo i valori dal buffer quando possibile (→ appendice **A.3**, riga **97-126**).

4. Nella matrice **DEP** sono presenti valori estremi pari a $1.7 * 10^{38}$, sono utilizzati dal simulatore per indicare celle in cui non puó accumularsi alcuna quantità di fluido, pertanto possono essere escluse. Questi valori vengono sostituiti con zeri e non cambieranno mai valore (→ appendice **A.3**, riga **147-150**).

5. Si considerano matrici da 768x768 pixels. Le matrici DEP, VVX e VVY sono concatenate per costituire i canali di ogni frame, formando

un tensor di shape (768, 768, 3). Se il downsampling 4x é abilitato, si applica un filtro gaussiano di kernel (3,3) e un doppio downsampling 2x, passando a immagini da 192x192 (→ appendice **A.3**, riga **59-63, 120-124**).

6. Si genera il tensor della sequenza i-esima, x_i, composto da 4 frames concatenati del passato e di forma (4, 768, 768, 4). Rispettivamente, il tensor con la sequenza target y_i ha la medesima forma se si prevendono 4 frames nel futuro, oppure (1, 768, 768, 3) per il frame successivo (→ appendice **A.3**, riga **153-174**).

7. Si scartano le sequenze in cui non c'é una sostanziale differenza tra i frames, quindi non avviene alcun evento interessante. La "dinamicità" δ di una sequenza é stimata con la formula:

$$\delta_i = 1 - \text{accuracy} = 1 - \frac{TP + TN}{TP + TN + FP + FN} \qquad \forall i \in [m+1, ..., m+k] \tag{5.1}$$

Ossia l'inverso dell'accuratezza tra ogni frame i della sequenza target e il primo frame in input, si valutano le celle bagnate e si ignorano quindi quelle asciutte del terreno (→ appendice **A.2**, riga **18-53**).

8. Il modulo **SWEDataset** incapsula le classi *DataPartitions* e *DataGenerator* per rendere la sequenza di batches iterabile. Dopo aver caricato le sequenze con *get_data()*, il tensor é sottoposto a *reshape* per spostare la dimensione dei canali e ottenere il formato *channel-first*, richiesto da PyTorch come standard per i layer convoluzionali. La forma finale del tensor é quindi: (4, 3, 768, 768) (→ appendice **A.4**, riga **89-91**).

9. Le sequenze sono organizzate in batch di 4 osservazioni, concatenando quindi 4 sequenze per volta si ottiene un tensor batch di forma: (4, 4, 3, 768, 768). La stessa procedura é applicata ai tensor target y_i, mantenendo l'ordine con le osservazioni x_i, appaiate per gli stessi indici.

10. Il modulo **SWEDataModule**, infine, incapsula l'oggetto di classe *SWEDataset* per suddividere i dati in più datasets: training, test e validation. Questa formattazione tra moduli é richiesta da PyTorch Lightning per utilizzare anche i loop di training default della classe $pytorch_lightning.Trainer$(→ *appendice* ***A.4****, riga* ***1-51****).*

5.4 Ambiente di sviluppo per il Deep Learning

Il linguaggio di programmazione utilizzato in questo progetto di tesi é **Python 3.6**. Data la numerosità delle librerie per il calcolo scientifico, si utilizzano degli ambienti virtuali che incapsulano tutte le dipendenze per il progetto. É cosí possibile creare piú ambienti per contesti diversi e trasferire con facilità e modularità l'intero spazio di sviluppo dal personal computer al cluster HPC. **Anaconda** é una distribuzione per Python ed R con il compito di semplificare la gestione delle librerie, questa é installata sul clusterHPC in versione "lightweight": **miniconda**. Elenchiamo le librerie fondamentali per questo progetto:

- **PyTorch 1.7.1**: deep learning e machine learning framework per lo sviluppo della rete neurale
- **PyTorch Lightning 1.4.2**: versione ottimizzata di pytorch per il supporto multi-gpu/tpu e tecniche di training avanzate
- **Numpy 1.20.2**: framework per la manipolazione di tensors e operazioni di pre-processing dei dati
- **SciKit-Learn 0.24**: libreria di algoritmi di machine learning e strumenti di data pre-processing

La creazione di un ambiente anaconda avviene con pochi semplici comandi, specificando la versione di python. I pacchetti possono poi essere installati

al suo interno:

```
conda create -n swe-deeplearning python=3.6
source activate swe-deeplearning
conda install pytorch torchvision pytorch-lightning -c pytorch
```

5.4.1 Introduzione a PyTorch

PyTorch é un *framework* di machine learning e deep learning sviluppato dal team di ricerca in AI di Facebook (FAIR) nel 2016. La libreria offre principalmente strumenti per la manipolazione degli **tensors**, la costruzione di reti neurali e il calcolo differenziale automatizzato. PyTorch si propone come framework alternativo ad altre libreria per il deep learning: Tensorflow (sviluppata da Google) o Theano; PyTorch Lightning nasce come framework ad alto livello construito sopra PyTorch puro, permettendo di separare l'aspetto ingegneristico da quello di ricerca con modelli modulari e facilmente modificabili.

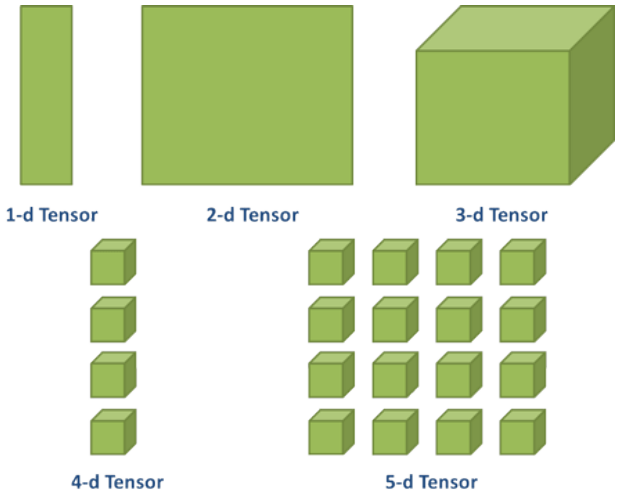


Figura 5.3: Visualizzazione di un tensor con dimensioni variabili.

Con *tensor* si intende un array n-dimensionale rappresentato da un oggetto python costituito da: una forma (*shape*, cardinalità), delle funzioni sui

dati (mean, sum, min, max), un tipo (float32, float64) e i bytes di informazione. Gli input della rete neurale sono dei tensors: un'immagine é un tensor di forma *(canale, altezza, larghezza)*, un video é un tensor di forma *(timestep, canale, altezza, larghezza)*. A differenza di Numpy, PyTorch permette di effettuare operazioni sui tensor utilizzato GPU e TPU, parallelizzando le operazioni tra matrici e ottenendo un netto guadagno sui tempi di computazione.

5.4.2 Backpropagation e reti neurali con PyTorch

Al fine di correggere i pesi della rete neurale, si calcolano i gradienti della funzione errore, quindi anche i gradienti dell'output della rete stessa. La rete neurale puó essere vista come una complessa funzione composta la cui derivata é calcolata applicando la *chain-rule*:

$$h' = (f \circ g)' = (f' \circ g) \cdot g' \tag{5.2}$$

Per ottimizzazione, é possibile calcolare i gradienti "locali" delle funzioni che formano quella composta e quindi applicare la chain-rule. Per tenere traccia delle operazioni effettuate sui tensori lungo la rete neurale, si costruisce un **grafo computazionale**.

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \tag{5.3}$$

Funzione sigmoid applicata a una semplice funzione a 2 variabili (x1, x2).

A differenza di altri framework di deep learning, PyTorch costruisce un grafo computazionale *dinamicamente*, ossia ogni volta che la rete viene attraversata. Questo permette maggiore flessibilità: la rete puó svolgere operazioni differenti ad ogni iterazione, i gradienti sono calcolati sul momento. La versatilità di PyTorch e la semplicità del codice rendono questo framework popolare nella community di ricerca: é possibile trovare le implementazioni

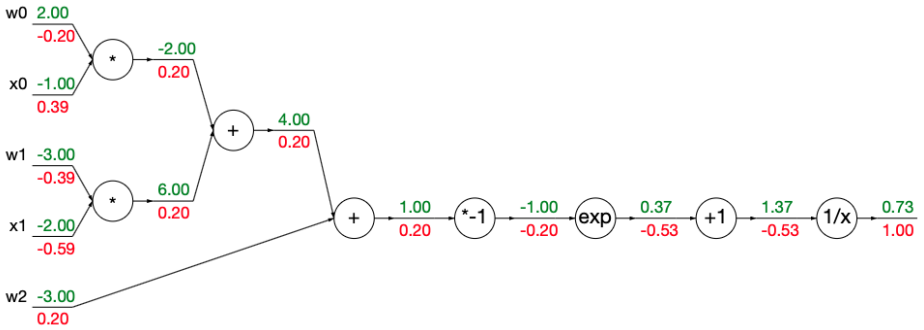


Figura 5.4: Grafo computazionale della funzione sopra indicata, in verde i risultati del forward pass, in rosso i gradienti (backward pass).

di numerose architetture di reti neurali con cui poter sperimentare, per tal motivo PyTorch é stato scelto per questo progetto di tesi.

Il package **torch.autograd** si occupa della costruzione del grafo computazionale e del calcolo delle derivate in modo automatico; con il modulo **torch.nn**, si definiscono i layer per comporre la rete neurale con una classe derivata da *nn.Module*; infine, il package **torch.optim** permette di selezionare un algoritmo di discesa del gradiente per il training della rete neurale (si sceglie **Adam** per questo progetto). Si puó osservare l'implementazione di tutti questi moduli nello script di training →appendice **B.3**.

5.5 Implementazione della rete neurale

La rete neurale progettata per questo studio é definibile un **auto-encoder**:

1. **Encoder**: i frame X_i della sequenza sono iterativamente forniti a un layer ConvLSTM di due celle. La dimensionalità dei frames (numero di canali) é aumentata da (DEP, VVX, VVY, BTM) $4 \rightarrow 16$ per analizzare sufficienti features. L'output di questo layer rappresenta la sequenza compressa (embedding vector).

2. **Decoder**: l'embedding vector é l'input di un ulteriore layer ConvLSTM che iterativamente genera i frames; la sequenza é processata da un layer Conv3D per ridurre il numero di canali in output da 16 → 3 (DEP, VVX, VVY).

L'implementazione PyTorch della cella ConvLSTM rispetta la descrizione del paragrafo 4.2.1 ed é consultabile in appendice →**B.1**. Un layer ConvLSTM é composto da 2 celle ConvLSTM in successione: l'output della prima cella é fornito alla seconda, quindi il loro stato interno cambia nel tempo quando iterativamente si fornisce un frame per volta al layer. Allo stesso modo, l'embedding vector é fornito frame per frame al layer ConvLSTM decoder.

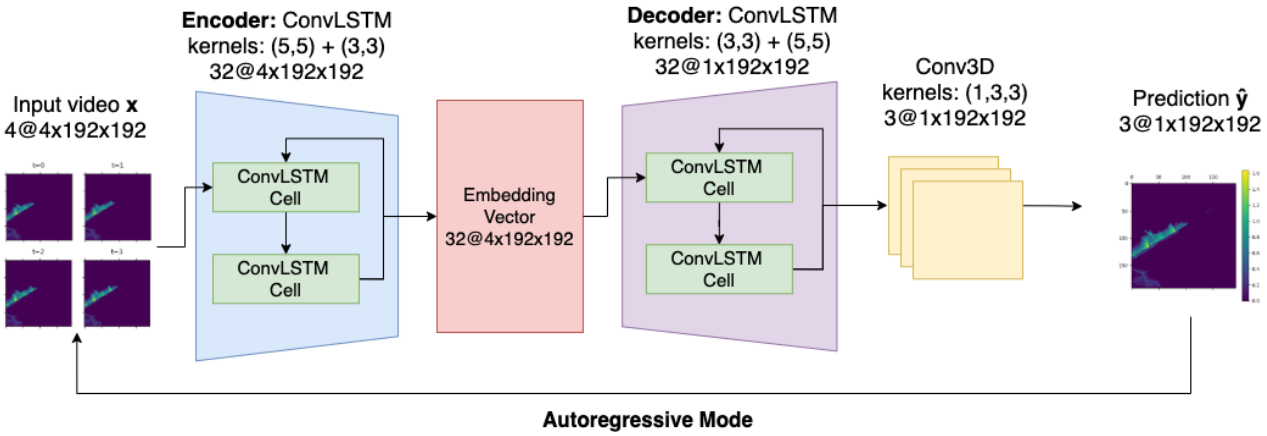


Figura 5.5: Architettura ConvLSTM auto-encoder a 32 filtri ed esempio della generazione del next-frame.

L'implementazione PyTorch dell'auto-encoder ConvLSTM é consultabile in appendice →**B.2**. La prima cella ConvLSTM utilizza kernel 5x5 rispetto alla seconda (kernel 3x3), questo permette alla rete neurale di separare le features su due livelli di astrazione: prima in un raggio piú esteso e poi maggiormente nel dettaglio. Il numero di kernel per cella é un iperparametro, in questo studio si testano reti con 32 filtri per cella.
Il blocco decoder del modello puó generare piú di un frame futuro, alternativamente si concatena il next-frame alla sequenza in input (**modalitá**

regressiva). Ci si concentrerá su quest'ultima modalitá per verificare come la rete neurale performa sulle sue stesse previsioni e come l'errore di regressione si accumula.
Si denomina la rete neurale di questo studio *"deepSWE"*.

Tabella 5.1: Allocazione e numero di parametri deepSWE 16 kernels

	Name	Type	Kernel Size	Params
0	encoder_1_convlstm	ConvLSTMCell	(5,5)	32.1 K
1	encoder_2_convlstm	ConvLSTMCell	(3,3)	18.5 K
2	decoder_1_convlstm	ConvLSTMCell	(3,3)	18.5 K
3	decoder_2_convlstm	ConvLSTMCell	(5,5)	51.3 K
4	decoder_CNN	Conv3D	(1, 3, 3)	435
	Total trainable params			120 K

Tabella 5.2: Allocazione e numero di parametri deepSWE 32 kernels

	Name	Type	Kernel Size	Params
0	encoder_1_convlstm	ConvLSTMCell	(5,5)	115 K
1	encoder_2_convlstm	ConvLSTMCell	(3,3)	73.9 K
2	decoder_1_convlstm	ConvLSTMCell	(3,3)	73.9 K
3	decoder_2_convlstm	ConvLSTMCell	(5,5)	204 K
4	decoder_CNN	Conv3D	(1, 3, 3)	867
	Total trainable params			468 K

5.5.1 Training e metriche

L'addestramento della rete neurale é stato effettuato in piú sessioni su volumi di dati differenti. Sono state testate due versioni del modello: una rete a 16 kernel per layer convoluzionale, semplificata, ed una a 32 filtri. Abbiamo poi individuato i seguenti parametri come ottimali:

- **Batch Size**: 4
- **Numero di kernels**: 32
- **Regioni**: Arda, Baganza
- **Learning rate**: 1e-4
- **Epochs**: 200
- **Porzione test set**: 10%
- **% Variazione sequenza (dinamicitá)**: 20%
- **Frequenza di campionamento dataset**: 1 frame / 3 minuti
- **Downsampling**: 4x con gaussian blur, kernel (3,3)

La funzione errore trattata in questo studio é una variante della funzione Mean Squared Error. Si desidera penalizzare la rete neurale in base all'errore cumulativo di tutte le celle e la loro posizione, quindi si considera la **Residual Sum of Squares** (RSS):

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y_i})^2 \tag{5.4}$$

Si visualizza un esempio di calcolo RSS con errori per valore e per posizione:

$$sum\left(\left(\begin{bmatrix} 3.2 & 0 & 0 \\ 6.2 & 4.5 & 0 \\ 2.3 & 0 & 1.1 \end{bmatrix} - \begin{bmatrix} \mathbf{0} & 0 & \mathbf{1.8} \\ \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{0.3} & 0 & \mathbf{9.1} \end{bmatrix}\right)^2\right) = sum\left(\begin{bmatrix} 3.2 & 0 & -1.8 \\ 6.2 & 4.5 & 0 \\ 2 & 0 & -8 \end{bmatrix}^2\right) = \mathbf{140.17}$$

$$sum\left(\left(\begin{bmatrix} 3.2 & 0 & 0 \\ 6.2 & 4.5 & 0 \\ 2.3 & 0 & 1.1 \end{bmatrix} - \begin{bmatrix} 3.2 & 0 & \mathbf{0.5} \\ 6.2 & \mathbf{0} & 0 \\ 2.3 & 0 & 1.1 \end{bmatrix}\right)^2\right) = sum\left(\begin{bmatrix} 0 & 0 & -0.5 \\ 0 & 4.5 & 0 \\ 0 & 0 & 0 \end{bmatrix}^2\right) = \mathbf{20.50}$$

La funzione errore viene calcolata su una porzione della matrice predetta: una matrice 768x768 é in realtá una griglia di 3x3 tiles da 256x256; il tile centrale é il target della previsione, gli 8 tile "vicini" forniscono le condizioni al contorno, cioé le correnti entranti e le loro velocitá. Questo permette di prevedere una sequenza di k frame futuri in base al numero di tiles circostanti che la rete neurale osserva.

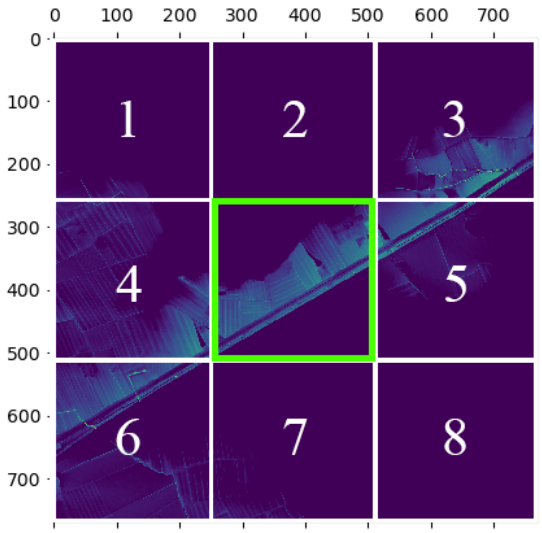


Figura 5.6: Fotogramma 768x768 scomposto in 3x3 tiles, al centro il tile target con le condizioni al contorno dettate dai tiles vicini (numerati).

L'addestramento del modello é avviato da uno script python eseguito in un job SLURM, il training loop é suddiviso nelle seguenti fasi:

1. Caricamento degli iperparametri dagli argomenti a linea di comando, preparazione delle cartelle per il training (appendice → **B.3**, riga **39-45**)

2. Preparazione del modulo dataset, che include generazione delle partizioni, lettura da disco e operazioni di pre-processing (appendice → **B.3**, riga **50-68**)

3. Inizializzazione della rete neurale con gli iperparametri forniti, wrapping in caso di multi-gpu e definizione dell'ottimizzatore per l'algoritmo di gradient descent. Si applica la regolarizzazione della rete neurale impostando un *weight_decay*, questo permette di ridurre la varianza del modello ed evitare overfitting (appendice → **B.3**, riga **71-93**)

4. Training loop (appendice → **B.3**): per ogni epoca si itera il dataset un batch per volta, quindi per ciascuno si calcola l'errore e si propagano i gradienti (riga **122-132**). Ad ogni epoca vengono salvati i pesi della rete neurale e salvati il tempo di inferenza e l'accuratezza del modello (riga **128-171**).

Per evitare l'overfitting (→ 3.3.3) si applica la formula di regolarizzazione **L2 Norm (Ridge Regression)**:

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 = \mathrm{RSS} + \lambda\sum_{j=1}^{p}\beta_j^2 \tag{5.5}$$

Dove y_i é il target, β_0 il bias e β_j i parametri della rete neurale; λ é un coefficiente di regolarizzazione, per $\lambda \to \infty$ l'impatto dei pesi nella funzione errore cresce.
La sua implementazione in PyTorch consiste nell'aggiunta del parametro *weight_decay* nell'ottimizzatore Adam (→ appendice B.3, riga **88**).

Capitolo 6

Risultati e valutazione

La rete neurale é stata addestrata in piú sessioni su regioni differenti. In alcuni casi si é scelta una frazione del dataset per valutare le performance del modello anche con una quantitá limitata di informazioni. L'implementazione dei test é consultabile in appendice → **B.4**.

# Test	Regione	#Sequences	# Frames	Rete	RAM usage
1	Arda	276	1380	deepSWE 32	57.74 GB
2	Arda + Baganza	1172	5860	deepSWE 32	182.46 GB
3	Arda + Baganza	1656	13248	deepSWE 32	307.62 GB

6.0.1 Metriche di valutazione

Al fine di valutare il modello sono state adottate delle metriche basate su tre fattori: accuratezza, somiglianza strutturale, errore di regressione.

- **Accuracy**: rapporto tra le celle correttamente predette e il totale. Si considera un'approssimazione di ±5cm (P=bagnate, N=asciutte).

$$\text{Accuracy } = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.1}$$

- **Mean Squared Error**: introdotta nel paragrafo → 3.3.

- **Structural Similarity**: indice di similarità strutturale delle immagini, c_1 e c_2 sono costanti per stabilizzare il rapporto.

$$\mathrm{SSIM}(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{\left(\mu_x^2 + \mu_y^2 + c_1\right)\left(\sigma_x^2 + \sigma_y^2 + c_2\right)} \tag{6.2}$$

- **Mean Absolute Error**: a differenza di MSE gli errori non sono enfatizzati da un esponente, anche in questo caso si punta al minimo valore positivo.

$$\mathrm{MAE(x, y)} = \frac{\sum_{i=1}^{n} |x_i - y_i|}{n} \tag{6.3}$$

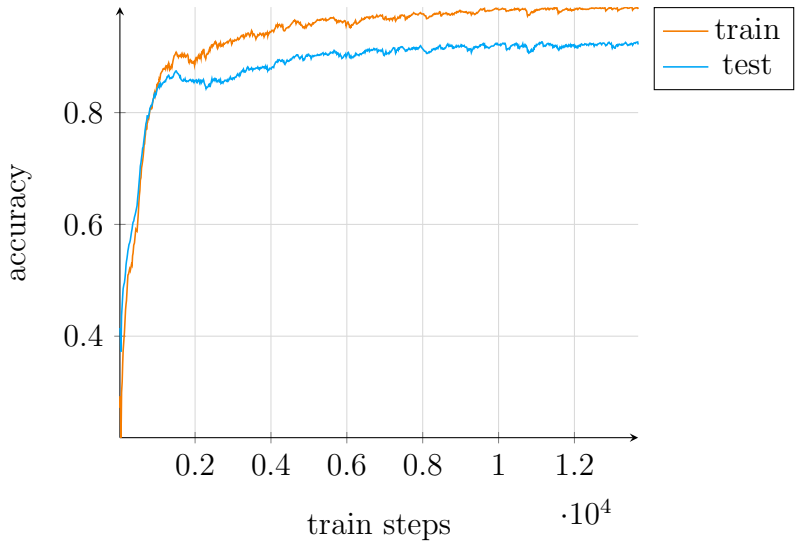
Le misure di accuracy nei grafici seguenti sono sottoposte a *smoothing* utilizzando una **exponential weigthed function** ricorsiva:

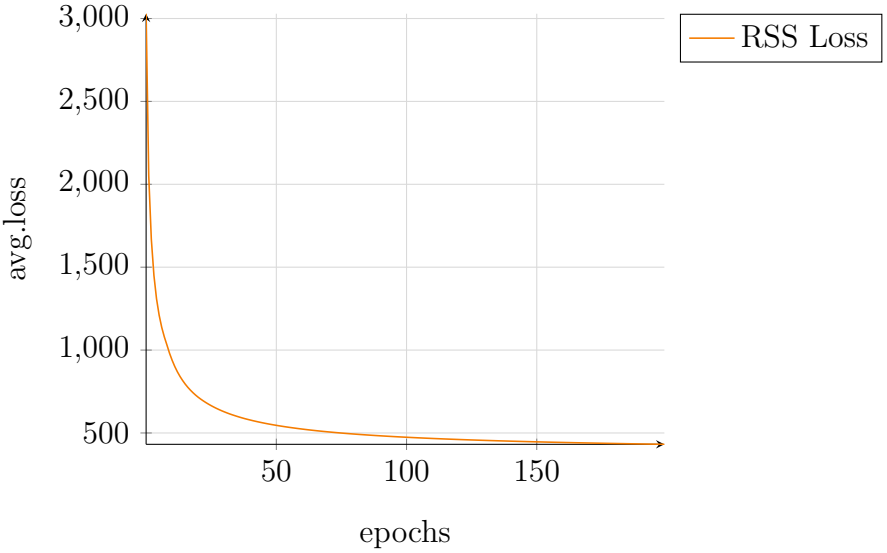
$$\begin{aligned} y_0 &= x_0 \\ y_t &= (1 - \alpha)y_{t-1} + \alpha x_t \end{aligned} \tag{6.4}$$

Questo permette una visualizzazione piú chiara del trend di apprendimento, con minore influenza dei valori estremi. Si utilizza un fattore $\alpha = 0.9$.

6.0.2 Risultati

- **Test 1**: previsione **next-frame**. Il training della rete per 1 epoch dura ≈**557s**; il tempo medio di inferenza per batch é di **0.0764s**; il tempo medio per prelevare un batch (4 sequenze) da disco é di **0.07s**.





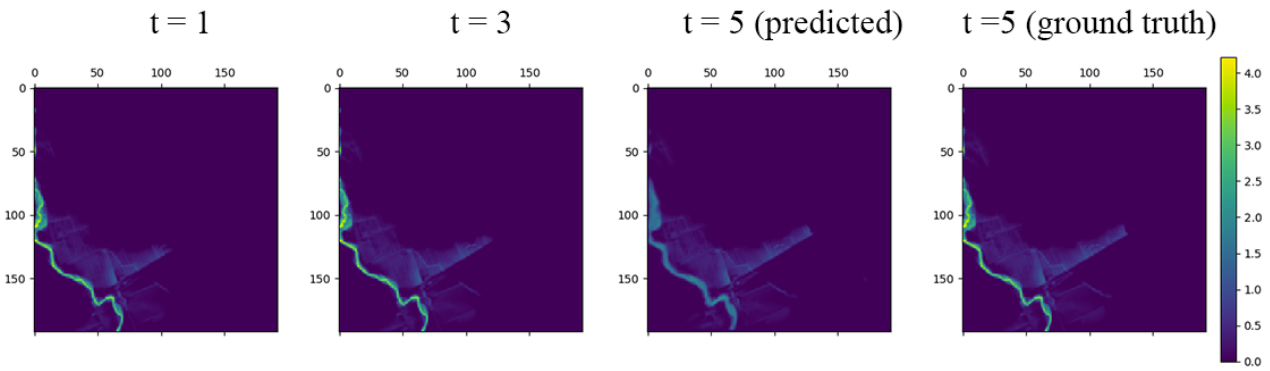


Figura 6.1: Confronto next-frame, zona **arda_1**.

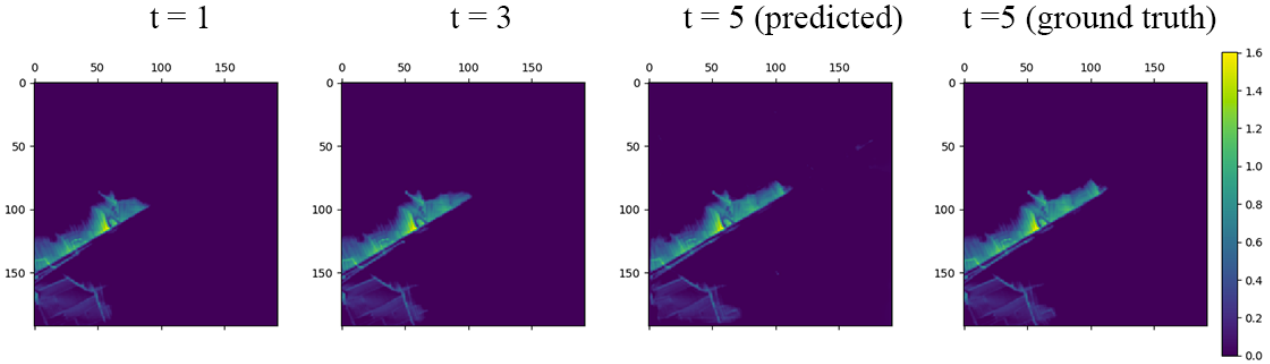


Figura 6.2: Confronto next-frame, zona **arda_2**.

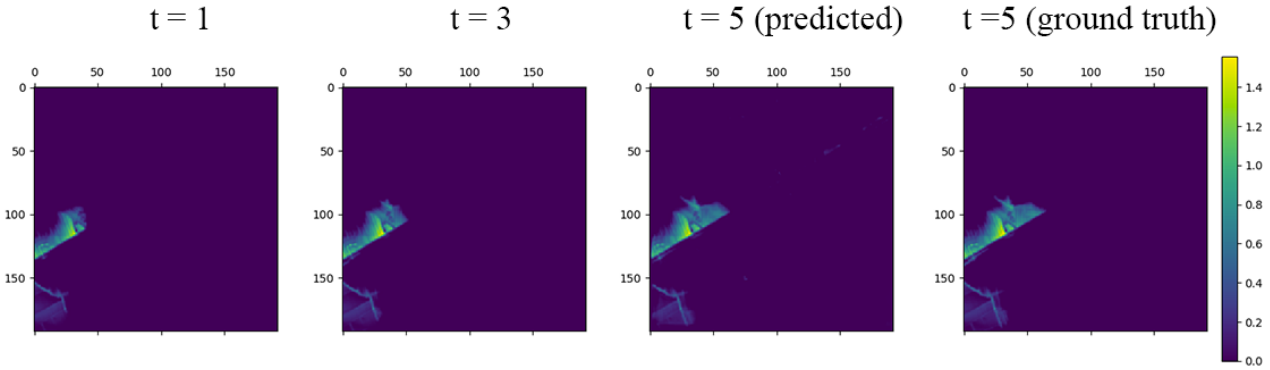
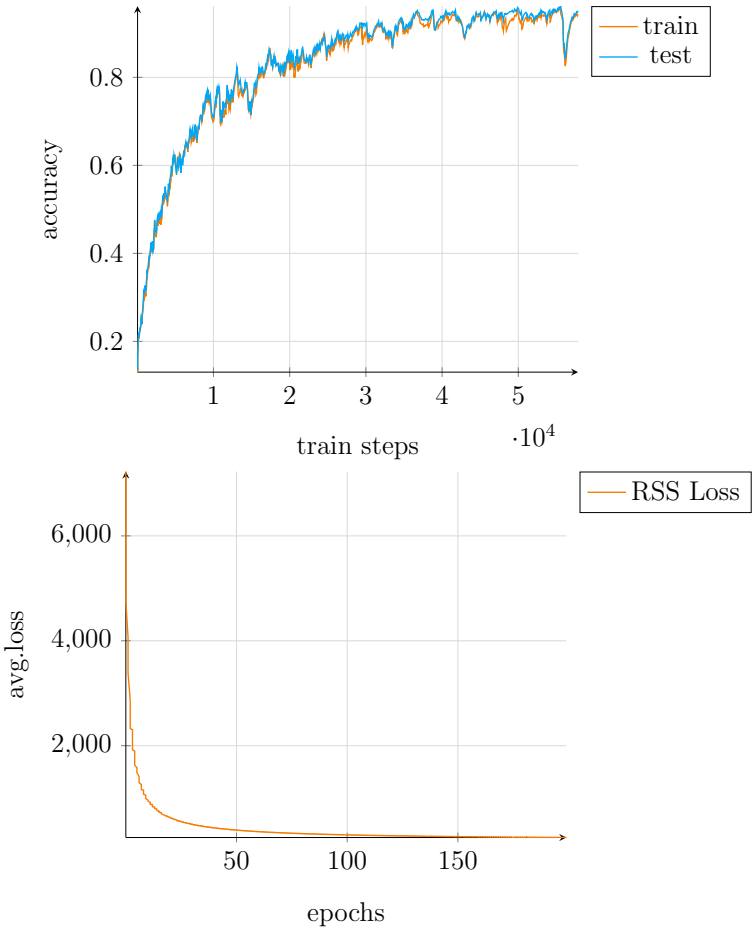


Figura 6.3: Confronto next-frame, zona **arda_3**.

Tabella 6.1: Punteggi su test set di deepSWE 32, previsione next-frame (Arda)

Zona	Accuracy ±5cm	SSIM	MSE (L2 Loss)	MAE (L1 Loss)
arda_1	96.36%	93.56%	0.0045	0.0134
arda_2	97.54%	89.32%	0.0050	0.0197
arda_3	95.04%	90.01%	0.0056	0.0184

- **Test 2**: previsione **next-frame**. Il training della rete per 1 epoch dura intorno ai **424s**; il tempo medio di inferenza per batch é di **0.0351s**; il tempo medio per prelevare un batch (4 sequenze) da disco é di **0.04s**.



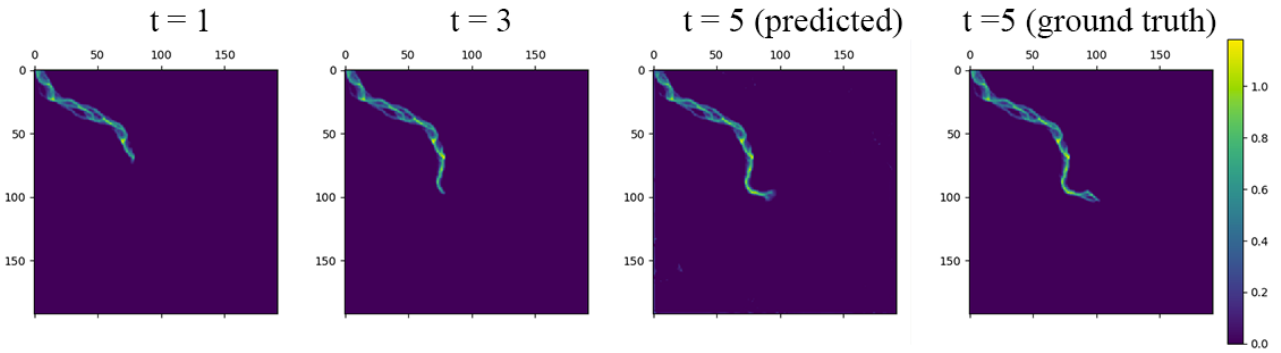


Figura 6.4: Confronto next-frame, zona **Baganza**.

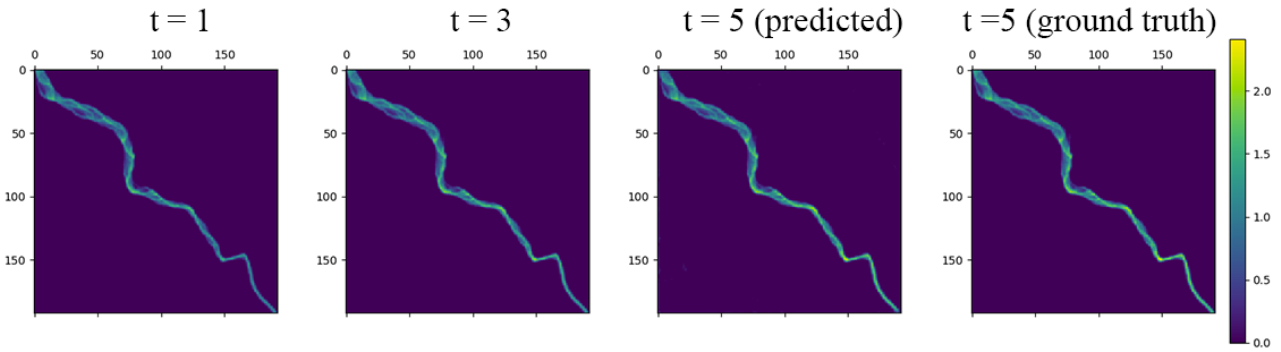


Figura 6.5: Confronto next-frame, zona **Baganza**.

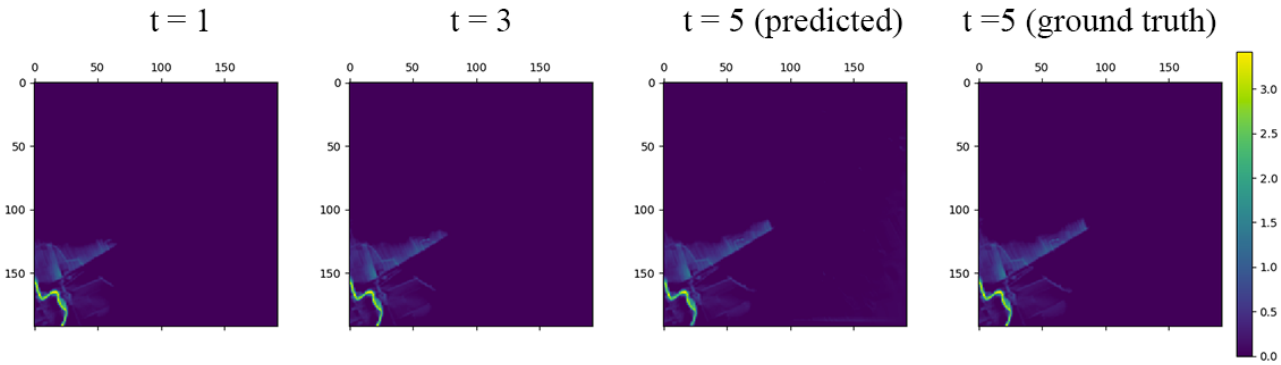
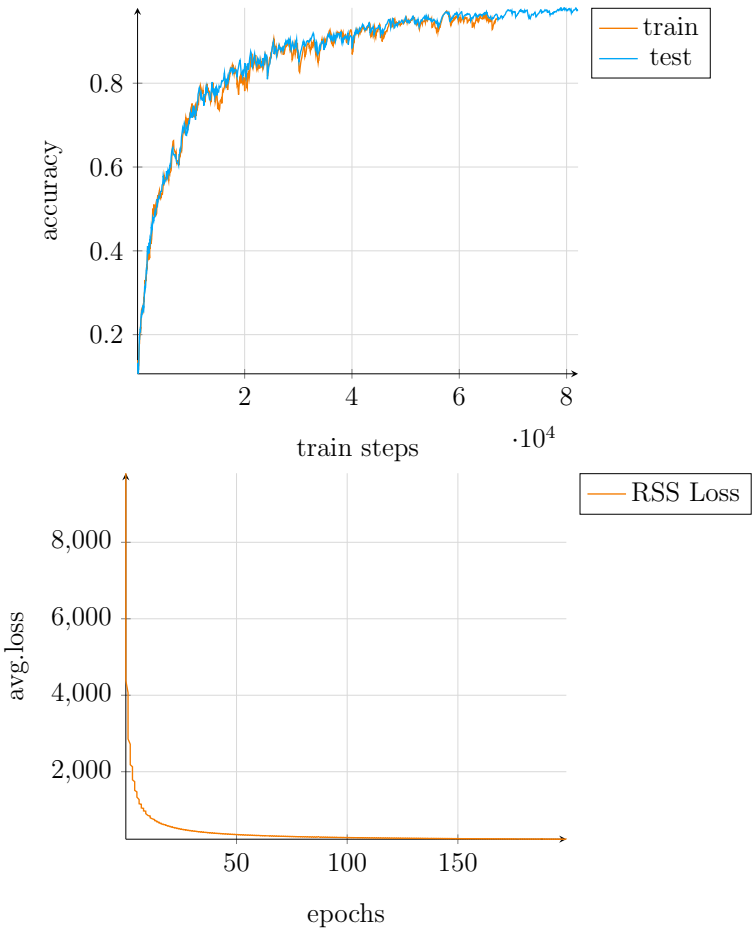


Figura 6.6: Confronto next-frame, zona **Arda**.

Tabella 6.2: Punteggi su test set di deepSWE 32, previsione next-frame (Arda + Baganza)

Zona	Accuracy ±5cm	SSIM	MSE (L2 Loss)	MAE (L1 Loss)
Arda	98.65%	94.56%	0.0098	0.0029
Baganza	98.37%	97.65%	0.0052	0.0008

- **Test 3**: previsione sequenza **next 4 frames**. Il training della rete per 1 epoch dura intorno ai **391s**; il tempo medio di inferenza per batch é di **0.058s**; il tempo medio per prelevare un batch (4 sequenze) da disco é di **0.04s**.



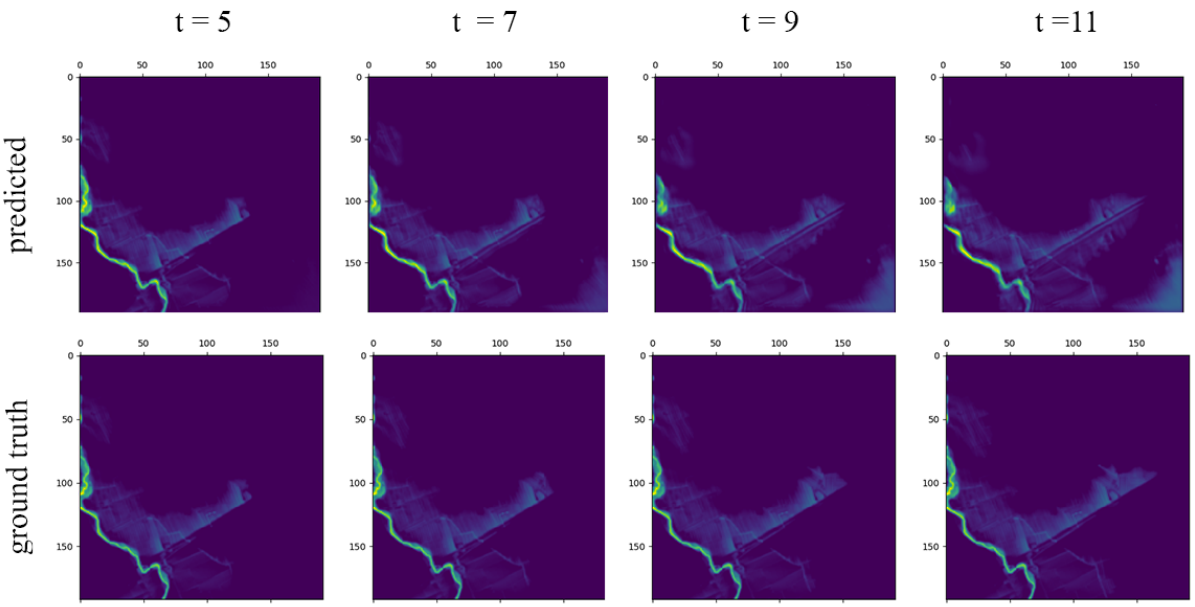


Figura 6.7: Confronto 4 frames futuri (auto-regressive), torrente **Arda**.

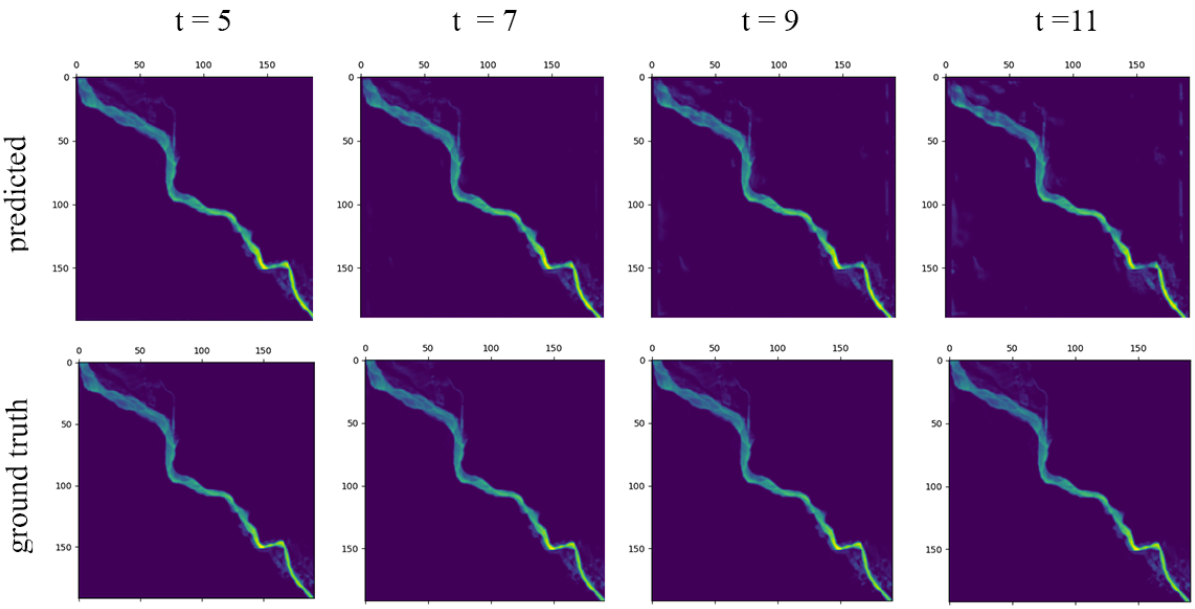


Figura 6.8: Confronto 4 frames futuri (auto-regressive), torrente **Baganza**.

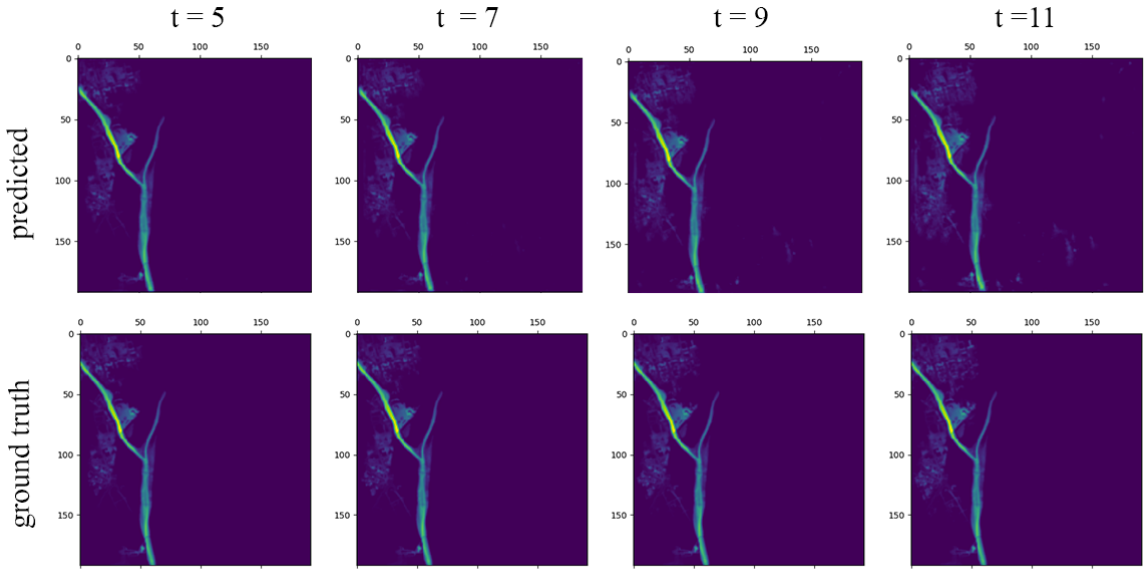


Figura 6.9: Confronto 4 frames futuri (auto-regressive), torrente **Baganza**.

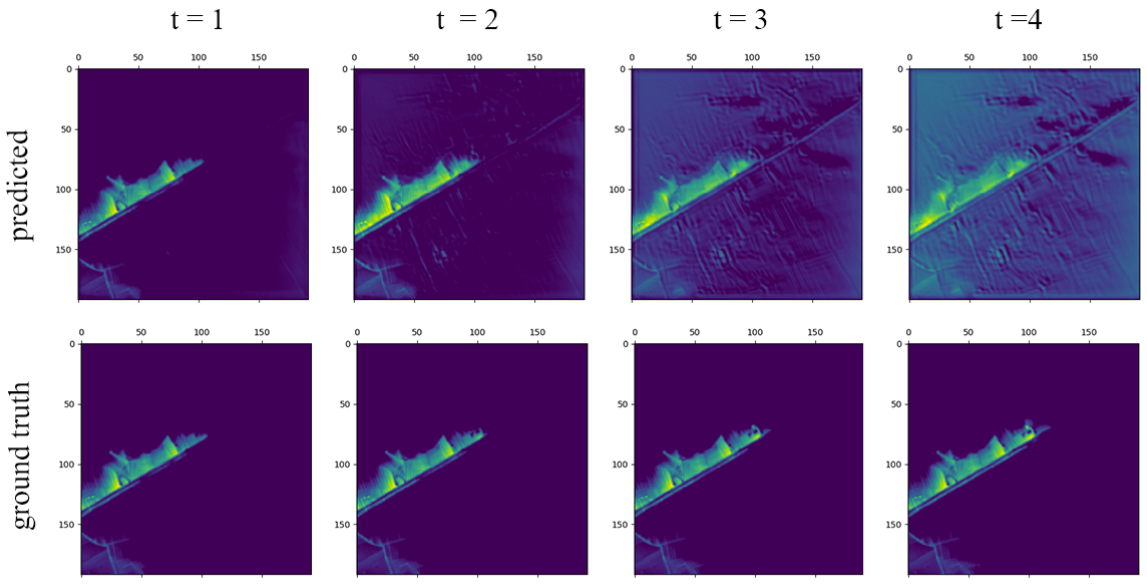


Figura 6.10: Confronto 4 frames futuri (seq2seq), downsampling 4x, torrente **Baganza**.

Tabella 6.3: Punteggi su test set in modalità auto-regressiva (1) e seq2seq (2).

Zona	Accuracy ±5cm	SSIM	MSE (L2 Loss)	MAE (L1 Loss)
Arda[1]	59.59%	81.74%	0.0657	0.0422
Baganza[1]	61.72%	86.85%	0.0302	0.0144
Arda[2]	44.87%	56.93%	0.1072	0.0370

Questa sessione di training é mirata al prevedere piú di un frame futuro. In modalitá **auto-regressiva** il modello genera previsioni piú fedeli alla realtà, nonostante inizino a divergere proseguendo nel tempo (accumulazione dell'errore di regressione). Nel caso del torrente Arda (Figura 6.7): il modello prevede nuovi livelli di fluido in zone asciutte e lontane dal corso d'acqua (t=11), inoltre i livelli giá esistenti tendono a diminuire dove nella realtà si mantengono costanti (t=9).
In modalità **sequence2sequence** l'output della rete non é riutilizzato in input, ma la sequenza é generata dalle celle ConvLSTM del blocco decoder. In questo caso, l'accuratezza delle previsioni é piú bassa: i frame piú distanti dal presente sono sempre meno nitidi e accurati (Figura 6.10), sorgono i dettagli (indesiderati) del terreno.
Il primo metodo risulta quindi maggiormente accurato: la rete neurale si basa solo sul prossimo frame futuro per proseguire la serie temporale, quindi minore errore di regressione é accumulato.

Capitolo 7

Conclusioni

7.1 Progressi e risultati

Con questo studio sono state affrontate varie limitazioni legate ai modelli proposti da S. Gazza (2021) e Kabir et.al (2020):

- Abbiamo strutturato l'input come immagini a piú canali, introducendo le matrici batimetria e velocitá. Abbiamo quindi parametrizzato le condizioni del terreno, permettendo al modello di prevedere allagamenti in piú regioni; i test effettuati dimostrano una buona capacitá di generalizzazione nelle previsioni.

- Abbiamo esteso il dominio a una griglia di 9 tiles, dove il centro é la regione target. Utilizzando le sezioni circostanti come condizioni al contorno, abbiamo addestrato la rete a proseguire l'allagamento nel futuro con piú informazioni di contesto dal passato. Il numero di tiles circostanti detta la finestra temporale per cui possiamo prevedere dei frames futuri, aumentando le condizioni al contorno si puó puntare a una sequenza piú estesa.

- Una simulazione del torrente Arda con Parflood impiega **939.9 s** ($\approx$16 minuti), mentre il training della rete neurale puó richiedere fino a **24 ore** (tempo offline). Tuttavia, il rapporto é diverso sulla generazione

dei frames (tempo online): la rete neurale impiega in media **≈14 ms** per generare il frame successivo; Parflood impiega in media **≈148 ms**, i calcoli effettuati nella rete neurale sono **x10.6** volte piú rapidi. Il costo computazionale puó essere ridotto ottimizzando l'architettura neurale (MobileNet[18], per le reti convoluzionali), quindi sarebbe possibile implementare il modello su dispositivi portatili come centraline installate sul posto.

- Abbiamo esteso il dataset su piú torrenti, ottenendo **165** aree geografiche 768x768, **40,425** sequenze video, quindi **202,125** raw frames (non filtrati). Maggiore varietá nelle osservazioni ha permesso alla rete neurale di generalizzare il comportamento degli allagamenti, con previsioni piú robuste indipendentemente da una specifica area geografica (rischio presente nel modello di Gazza[17]).

- L'introduzione delle convoluzioni ricorrenti (ConvLSTM) ha permesso di apprendere le relazioni tra pixel nello spazio e nel tempo, con stati interni nei layer sensibili alla sequenza di input. Questa tipologia di rete neurale é la base comune a modelli allo stato dell'arte per video prediction, quali CrevNet[13] e PhyDNet[19]. La nostra architettura raggiunge un'accuratezza media del **≈97%** per il next-frame e **≈60%** per una sequenza nel futuro di 8 frames.

7.2 Limitazioni e sviluppi futuri

- La dimensione del batch e il numero di filtri sono stati ridotti per via del limite delle risorse computazionali. Con maggiore capacitá di calcolo (cloud GPU, cloud TPUs) sarebbe possibile esplorare architetture molto piú complesse per sequenze temporali estese. Il modello proposto in questo studio ha infatti un numero di parametri e una struttura molto piú semplici rispetto allo stato dell'arte [13][19].

- Date le dimensioni dei datasets, il downsampling 4x ha permesso il training con le risorse presenti, al costo di una risoluzione immagine più bassa. Ulteriori studi possono essere dedicati all'ottimizzazione del dataset rimuovendo la ridondanza: le sequenze temporali possono essere generate "su richiesta" al posto di essere memorizzate, riducendo la ripetizione di molti frames.

- L'auto-encoder utilizzato genera le previsioni in modo deterministico. Un problema comune tra questi modelli é la generazione di previsioni "sfocate", poiché la rete approssima i possibili futuri in un unico frame. Il problema sta nella rappresentazione compressa degli input (spazio latente), spesso irregolare ed eterogenea, per cui nuovi samples generati da questo spazio latente possono contenere o essere rumore. Questo rumore si accumula in modalitá auto-regressiva, quindi sempre piú avanti nel futuro le previsioni divergono dalla realtá. I modelli probabilistici risolvono questo problema gestendo la distribuzione dello spazio latente[20].

- La lunghezza delle sequenze temporali in questo studio é molto ridotta, estendere la finestra di previsione del modello é un obiettivo futuro.

Si considererà necessario complicare l'architettura della rete neurale e con molta probabilità passare a un nuovo tipo di feature extractor per la componente temporale. Le reti neurali basate su "attenzione"[21] stanno acquistando una crescente importanza nella ricerca attuale, é opportuno citare una nuova architettura per la video prediction basata su modelli probabilistici e moduli d'attenzione: Video-GPT[22].

- Secondo la ricerca attuale, le reti neurali per apprendere da equazioni differenziali parziali (PDE) come le SWE includono leggi esplicite della fisica; questa casistica prende il nome di **physics-informed machine learning**. Architetture neurali come SINDy auto-encoder[23] includono nella funzione errore metriche per valutare i gradienti degli output (anche intermedi) della rete rispetto ai gradienti delle leggi reali introdotte. Crescente interesse é rivolto verso questi metodi, li riteniamo pertanto cruciali da considerare per gli sviluppi futuri di questo studio.

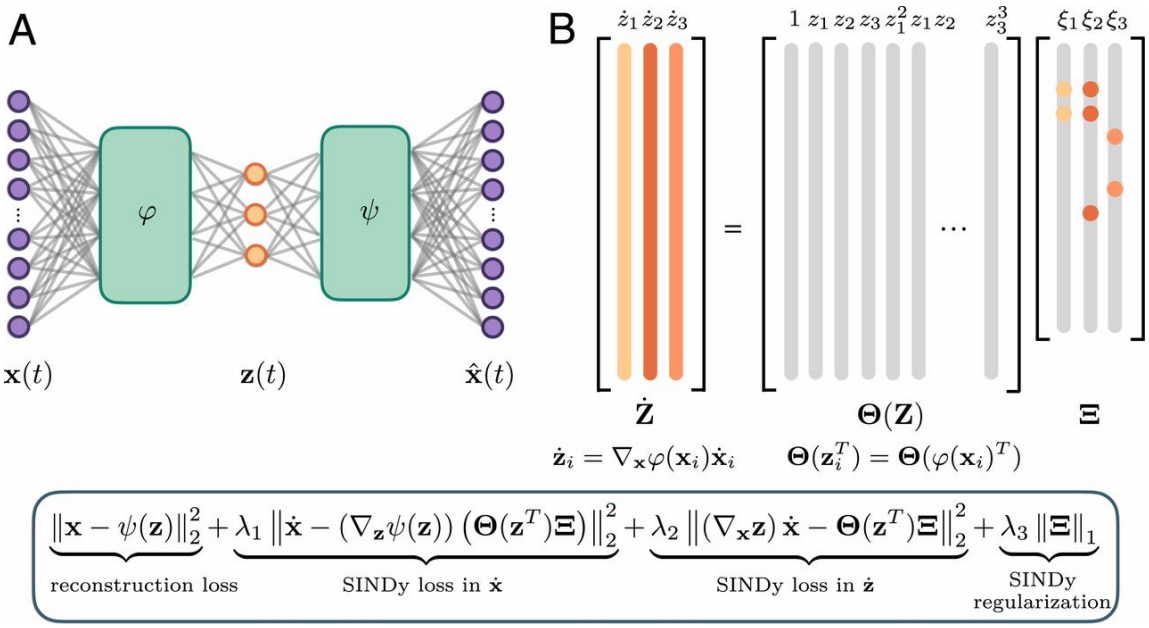


Figura 7.1: SINDy auto-encoder e funzione errore composta (con regolarizzazione).

Appendice A

Implementazione dei moduli data processing

A.1 Il modulo DataPartitions

```
1  class DataPartitions():
2      def __init__(self, past_frames, future_frames, root,
           partial=None, shuffle=True):
3
4          self.root = root
5          self.past_frames = past_frames
6          self.future_frames = future_frames
7          self.dataset_partitions = []
8          self.areas = []
9          self.partial = partial
10         self.shuffle = shuffle
11         self.create_partitions()
12
13     def get_areas(self):
14         ''' Returns list of areas (source folder name)
15         '''
16         return self.areas
17
18     def get_partitions(self):
```

```
19          ''' Returns dataset partitions
20          '''
21          if self.shuffle:
22              numpy.random.shuffle(self.dataset_partitions)
23          return self.dataset_partitions
24
25      def create_partitions(self):
26          ''' Creates the logical partitions (input-label)
27          '''
28          self.areas = os.listdir(self.root)
29          self.areas = [x for x in sorted(self.areas) if x.
                startswith("mini-") and os.path.isdir(self.root +
                x)]
30          numpy.random.shuffle(self.areas)
31
32          if self.partial is not None:
33              self.areas = self.areas[:int(len(self.areas) *
                    self.partial)]
34
35          if len(self.areas) <= 0:
36              raise Exception("Nessuna cartella area valida
                    trovata.")
37
38          # for each area
39          for area in self.areas:
40
41              n_frames = len([x for x in os.listdir(self.root +
                     area) if x.endswith(".DEP")])
42              # number of sequences
43              size = n_frames - self.past_frames - self.
                    future_frames
44
45              partition_raw = []
46              labels = dict()
47
48              for i in range(size):
49                  partition_raw.append("id-{}".format(i))
50                  labels["id-{}".format(i)] = list(
```

```
51                      range(i + (self.past_frames), i + (self.
                            past_frames) + (self.future_frames)))
52
53              if self.shuffle:
54                  numpy.random.shuffle(partition_raw)
55              # folder_name, x_frame_id, y_frames_id
56              self.dataset_partitions.append((area,
                    partition_raw, labels))
```

A.2 Il modulo Preprocessing

```
1  import numpy as np
2  import math
3
4  import torch as th
5  import pytorch_ssim
6  from torch.autograd import Variable
7
8
9  class Preprocessing():
10     def __init__(self):
11         pass
12
13     def filter(self, tensor):
14         tensor[np.isnan(tensor)] = 0
15         tensor[tensor > 10e5] = 0
16         return tensor
17
18     def eval_datapoint_diff(self, X, Y, threshold,
           sensibility=0.1, grid_size=3):
19         ''' Returns false/true if the given sequence of
               frames is "sufficiently dynamic"
20         '''
21         center = int(X.shape[1] / grid_size)
22         start = center
23         end = 2 * center
```

```
24
25          # compares the % difference between last and first
                frame
26          first_frame = self.filter(X[0, start:end, start:end,
                0])
27
28          # Difference between first frame and any in the
                output
29          deltas = []
30
31          for frame in Y:
32              current_frame = self.filter(frame[start:end,
                    start:end, 0])
33
34              wet_cells = (current_frame * first_frame) #
                    conjunction
35              wet_cells = np.array(wet_cells > 0).astype(int) #
                     total wet cells
36
37              # totale celle bagnate del fiume nella previsione
38              diff = np.abs((current_frame - first_frame))
39              similar_cells = (diff < sensibility).astype(int)
40              similar_cells = similar_cells*wet_cells #
                    conjuction
41
42              similarity = (np.sum(similar_cells)/np.sum(
                    wet_cells))
43
44              if np.isnan(similarity):
45                  similarity = 1
46
47              distance = 1 - similarity
48
49              # Only one has to satisfy the condition
50              if distance >= threshold:
51                  return True
52
53          return False
```

```
54
55      def eval_datapoint(self, X, Y, threshold=1e-1):
56          ''' Returns false/true if the given sequence of
                frames is "sufficiently dynamic"
57              threshold = [0,1]
58          '''
59          return self.eval_datapoint_diff(X, Y, threshold)
```

A.3 Il modulo DataGenerator

```
1  class DataGenerator():
2      def __init__(self, root, dataset_partitions, past_frames,
           future_frames, input_dim, output_dim, dynamicity,
3                  filtering=True, buffer_memory=1e2,
                       buffer_size=1e3, batch_size=16, caching=
                       True, downsampling=False):
4          '''
5              Data Generator
6              Inputs:
7
8                  - Path containing folders of frames
9                  - List of the names of these folders
10                 - Partitions: [(ids_x(x, 10), ids_y(x, 4))]
11         '''
12
13         self.input_dim = input_dim
14         self.output_dim = output_dim
15         self.dataset_partitions = dataset_partitions
16         self.past_frames = past_frames
17         self.future_frames = future_frames
18         self.caching = caching
19         self.batch_size = batch_size
20         self.blurry_filter_size = (3, 3)
21         self.downsampling = downsampling
22         self.root = root
23         self.buffer = []
```

```
24          self.buffer_size = buffer_size
25          self.buffer_memory = buffer_memory
26          self.buffer_hit_ratio = 0
27          self.preprocessing = Preprocessing() # filtering
                algorihtm
28          self.dynamicity = dynamicity
29          self.filtering = filtering
30
31      def get_data(self):
32          'Generates batches of datapoints'
33          X, Y = self.__data_generation()  # seq, t, h, w, c
34          return X, Y
35
36      def __data_generation(self):
37          'Generates the raw sequence of datapoints (filtered)'
38
39          X = None
40          Y = None
41
42          print("[x] {} areas found".format(len(self.
                dataset_partitions)))
43
44          # For each area
45          for area_index, area in enumerate(self.
                dataset_partitions):
46              # For each sequence
47              loaded = 0
48
49              print("Area {} ".format(area_index), end="",
                    flush=True)
50              for i, sequence in enumerate(area[1]):
51
52                  # --- BTM
53                  btm_filenames = [x for x in os.listdir(self.
                        root + self.dataset_partitions[area_index
                        ][0]) if
54                                   x.endswith(".BTM")]
55                  if len(btm_filenames) == 0:
```

```
56                      raise Exception("No BTM map found for the
                            area {}".format(self.
                            dataset_partitions[area_index][0]))
57                  btm = iter_loadtxt(self.root + self.
                        dataset_partitions[area_index][0] + "/" +
                        btm_filenames[0], delimiter=" ")
58
59                  # --- Preprocessing
60                  if self.downsampling:
61                      btm = cv.GaussianBlur(btm, self.
                            blurry_filter_size, 0)
62                      btm = cv.pyrDown(btm)
63
64                  # riduzione valori il sottraendo minimo
65                  min_btm = numpymin(btm)
66                  btm = btm - min_btm
67
68                  # duplicating as channel for each frame
69                  btm.resize(btm.shape[0], btm.shape[1], 1)
70                  btm_x = numpytile(btm, (self.past_frames, 1,
                        1, 1))
71
72                  deps = None
73                  vvx_s = None
74                  vvy_s = None
75
76                  framestart = int(sequence.replace("id-", ""))
77
78                  for k in range(framestart, framestart + self.
                        past_frames + self.future_frames):
79
80                      gid = "{}-{}-{}".format(area_index,
                            sequence, k) # global id for the
                            buffer cache dictionary
81
82                      extensions = ["DEP", "VVX", "VVY"]
83                      matrices = []
84
```

```
85                      # gets datapoint filename and checks for
                            files
86                      dep_filenames = [x for x in os.listdir(
                            self.root + self.dataset_partitions[
                            area_index][0]) if
87                                           x.endswith(".DEP")]
88
89                      if len(dep_filenames) == 0:
90                          raise Exception(
91                              "No DEP maps found for the area
                                    {}".format(self.
                                    dataset_partitions[area_index
                                    ][0]))
92
93                      # asserting that all maps are named with
                            the same prefix
94                      dep_filename = dep_filenames[0].split("."
                            )[0][:-4]
95
96                      # 1 frame -> 3 matrices (3 extensions)
97                      for i, ext in enumerate(extensions):
98                          accesses += 1
99                          global_id = "{}-{}".format(i, gid)  #
                                indice linearizzato globale
100
101                         # ----- Cache
102                         if self.caching:
103                             cache_frame = self.buffer_lookup(
104                                 global_id
105                             )
106                             if cache_frame is False:
107                                 frame = iter_loadtxt(
108                                     self.root + self.
                                            dataset_partitions[
                                            area_index][0] + "
                                            /{}{:04d}.{}".format(
                                            dep_filename,k, ext),
                                            delimiter=" ")
```

```
109                                 self.buffer_push(global_id,
                                        frame)
110                             else:
111                                 frame = cache_frame
112                                 hits += 1
113
114                         # ----- No cache
115                         else:
116                             frame = iter_loadtxt(
117                                 self.root + self.
                                        dataset_partitions[
                                        area_index][0] + "/{}{:04d
                                        }.{}".format(dep_filename,
                                        k, ext), delimiter=" ")
118
119                         # --- On-spot Gaussian Blurring
120                         if self.downsampling:
121                             frame = cv.GaussianBlur(frame,
                                    self.blurry_filter_size, 0)
122                             frame = cv.pyrDown(frame)
123
124                         matrices.append(frame)
125
126                     frame, vvx, vvy = matrices
127
128                     # ---
129
130                     if deps is None:
131                         deps = numpyarray([frame])
132                     else:
133                         deps = numpyconcatenate((deps,
                                numpyarray([frame])))
134
135                     if vvx_s is None:
136                         vvx_s = numpyarray([vvx])
137                     else:
138                         vvx_s = numpyconcatenate((vvx_s,
                                numpyarray([vvx])))
```

```
139
140                     if vvy_s is None:
141                         vvy_s = numpyarray([vvy])
142                     else:
143                         vvy_s = numpyconcatenate((vvy_s,
                               numpyarray([vvy])))
144
145                 # ---------
146
147                 deps[deps > 1.7e38] = 0
148                 vvx_s[vvx_s > 1.7e38] = 0
149                 vvy_s[vvy_s > 1.7e38] = 0
150                 btm_x[btm_x > 1.7e38] = 0
151
152                 # --- X
153                 x_dep = deps[:self.past_frames]
154                 x_dep.resize((x_dep.shape[0], x_dep.shape[1],
                        x_dep.shape[2], 1))
155
156                 x_vx = vvx_s[:self.past_frames]
157                 x_vx.resize((x_vx.shape[0], x_vx.shape[1],
                        x_vx.shape[2], 1))
158
159                 x_vy = vvy_s[:self.past_frames]
160                 x_vy.resize((x_vy.shape[0], x_vy.shape[1],
                        x_vy.shape[2], 1))
161
162                 x = numpyconcatenate((x_dep, x_vx, x_vy,
                        btm_x), axis=3)
163
164                 # --- Y
165                 y_dep = deps[self.past_frames:]
166                 y_dep.resize((y_dep.shape[0], y_dep.shape[1],
                        y_dep.shape[2], 1))
167
168                 y_vx = vvx_s[self.past_frames:]
169                 y_vx.resize((y_vx.shape[0], y_vx.shape[1],
                        y_vx.shape[2], 1))
```

```
170
171                 y_vy = vvy_s[self.past_frames:]
172                 y_vy.resize((y_vy.shape[0], y_vy.shape[1],
                        y_vy.shape[2], 1))
173
174                 y = numpyconcatenate((y_dep, y_vx, y_vy),
                        axis=3)
175
176                 # Applying filtering on the sequence if it's
                        dynamic enough
177                 if self.filtering:
178                     score, valid = self.preprocessing.
                            eval_datapoint(x[:, :, :, :3], y, self
                            .dynamicity)
179                 else:
180                     valid = True
181
182                 if valid:
183                     loaded += 1
184
185                     if X is None:
186                         X = numpyexpand_dims(x, 0)
187                     else:
188                         X = numpyconcatenate((X,
                                numpyexpand_dims(x, 0)))
189
190                     if Y is None:
191                         Y = numpyexpand_dims(y, 0)
192                     else:
193                         Y = numpyconcatenate((Y,
                                numpyexpand_dims(y, 0)))
194
195                     print("x ", end="", flush=True)
196                 else:
197                     print("- ", end="", flush=True)
198
199             print("\n[{}%] {} valid sequences loaded".format(
                    round((area_index + 1) / len(self.
```

```
                dataset_partitions) * 100), loaded))
200
201         return X, Y
202
203     def buffer_lookup(self, k):
204         ''' Get sequence (datapoint) from cache given the
                start frame global id '''
205
206         if self.caching:
207             for i, x in enumerate(self.buffer):
208                 # Returns found record
209                 if x["global_id"] == k:
210                     self.buffer[i]["fresh"] += 1
211                     return x["value"]
212                 # Set any read record to 0 (second chance)
213                 elif self.buffer[i]["fresh"] != 0:
214                     self.buffer[i]["fresh"] -= 1
215
216         return False
217
218     def buffer_push(self, k, x):
219         ''' Add sequence (datapoint) to cache with start
                frame global id '''
220         if self.caching:
221             # Makes space
222             if len(self.buffer) >= self.buffer_size:
223                 for i, j in enumerate(self.buffer):
224                     if j["fresh"] == 0:
225                         del self.buffer[i]
226             # Push
227             self.buffer.append({'fresh': self.buffer_memory,
                    'global_id': k, 'value': x})
```

A.4 I moduli wrapper del Dataset

```
1 class SWEDataModule(pl.LightningDataModule):
```

```
    def __init__(self, root, past_frames, future_frames,
        downsampling=False, caching=False, dynamicity=100,
        shuffle=True, image_size=768, batch_size=4, workers=4,
         filtering=True, test_size=0.1, val_size=0.1, partial=
        None):
        super(SWEDataModule, self).__init__()

        self.test_size = test_size
        self.val_size = val_size
        self.partial = partial
        self.root = root
        self.past_frames = past_frames
        self.future_frames = future_frames
        self.filtering = filtering
        self.batch_size = batch_size
        self.workers = workers
        self.image_size = image_size
        self.shuffle = shuffle
        self.dynamicity = dynamicity
        self.caching = caching
        self.downsampling = downsampling

    def prepare_data(self):
        dataset = SWEDataset(
            root=self.root,
            past_frames=self.past_frames,
            future_frames=self.future_frames,
            partial=self.partial,
            filtering=self.filtering,
            image_size=self.image_size,
            shuffle=self.shuffle,
            dynamicity=self.dynamicity,
            caching=self.caching,
            downsampling=self.downsampling
        )

        test_len = int(len(dataset) * self.test_size)
        val_len = int(len(dataset) * self.val_size)
```

```
36          train_len = len(dataset) - test_len - val_len
37
38          datasets = random_split(dataset, [train_len, test_len
                , val_len])
39
40          self.train_loader = DataLoader(datasets[0],
                batch_size=self.batch_size, num_workers=self.
                workers)
41          self.test_loader = DataLoader(datasets[1], batch_size
                =self.batch_size, num_workers=self.workers)
42          self.val_loader = DataLoader(datasets[2], batch_size=
                self.batch_size, num_workers=self.workers)
43
44      def train_dataloader(self):
45          return self.train_loader
46
47      def val_dataloader(self):
48          return self.val_loader
49
50      def test_dataloader(self):
51          return self.test_loader
52
53
54  class SWEDataset(Dataset):
55      def __init__(self, past_frames, future_frames, root,
            dynamicity, shuffle=True, filtering=True, caching=
            False, numpy_file=None, image_size=256, batch_size=4,
56                       buffer_memory=100, buffer_size=1000, partial
                            =None, downsampling=False):
57          ''' Initiates the dataloading process '''
58
59          if root is None and numpy_file is None:
60              raise Exception("Please specify either a root
                    path or a numpy file: -r /path -npy /dataset.
                    py")
61
62          self.filtering = filtering
63          self.batch_size = batch_size
```

```
64
65          self.partitions = DataPartitions(
66              past_frames=past_frames,
67              future_frames=future_frames,
68              root=root,
69              partial=partial,
70              shuffle=shuffle
71          )
72
73          self.dataset = DataGenerator(
74              root=root,
75              dataset_partitions=self.partitions.get_partitions
                    (),
76              past_frames=self.partitions.past_frames,
77              future_frames=self.partitions.future_frames,
78              input_dim=(self.partitions.past_frames,
                    image_size, image_size, 4),
79              output_dim=(self.partitions.future_frames,
                    image_size, image_size, 3),
80              batch_size=batch_size,
81              buffer_size=buffer_size,
82              buffer_memory=buffer_memory,
83              downsampling=downsampling,
84              filtering=self.filtering,
85              dynamicity=dynamicity,
86              caching=caching
87          )
88
89          self.X, self.Y = self.dataset.get_data()
90          self.X = self.X.transpose(0, 1, 4, 2, 3)
91          self.Y = self.Y.transpose(0, 1, 4, 2, 3)
92
93      def __len__(self):
94          return self.X.shape[0]
95
96      def __getitem__(self, idx):
97          return (self.X[idx], self.Y[idx])
```

Appendice B

Implementazione della rete neurale in PyTorch

B.1 Il modulo ConvLSTM

```
1  import torch.nn as nn
2  import torch
3  import pytorch_lightning as pl
4
5  # Crediti per il riferimento: https://github.com/ndrplz/
       ConvLSTM_pytorch
6
7  class ConvLSTMCell(pl.LightningModule):
8
9      def __init__(self, input_dim, hidden_dim, kernel_size,
           bias):
10
11         super(ConvLSTMCell, self).__init__()
12
13         self.input_dim = input_dim
14         self.hidden_dim = hidden_dim
15
16         self.kernel_size = kernel_size
```

```
17          self.padding = kernel_size[0] // 2, kernel_size[1] //
                2
18          self.bias = bias
19
20          self.conv = nn.Conv2d(in_channels=self.input_dim +
                self.hidden_dim,
21                                out_channels=4 * self.
                                    hidden_dim,
22                                kernel_size=self.kernel_size,
23                                padding=self.padding,
24                                bias=self.bias)
25
26      def forward(self, input_tensor, cur_state):
27          h_cur, c_cur = cur_state
28
29          combined = torch.cat([input_tensor, h_cur], dim=1)  #
                concatenate along channel axis
30
31          combined_conv = self.conv(combined)
32          cc_i, cc_f, cc_o, cc_g = torch.split(combined_conv,
                self.hidden_dim, dim=1)
33          # LSTM gates
34          i = torch.sigmoid(cc_i)
35          f = torch.sigmoid(cc_f)
36          o = torch.sigmoid(cc_o)
37          g = torch.tanh(cc_g)
38
39          c_next = f * c_cur + i * g
40          h_next = o * torch.tanh(c_next)
41
42          return h_next, c_next
43
44      def init_hidden(self, batch_size, image_size): #
            inizializzazione dei tensor
45          height, width = image_size
46          return (torch.zeros(batch_size, self.hidden_dim,
                height, width, device=self.conv.weight.device),
```

```
47                 torch.zeros(batch_size, self.hidden_dim,
                       height, width, device=self.conv.weight.
                       device))
```

B.2 Il modulo auto-encoder seq2seq

```
1  import torch
2  import torch.nn as nn
3
4  from models.ae.ConvLSTMCell import ConvLSTMCell
5
6  class EncoderDecoderConvLSTM(nn.Module):
7      def __init__(self, nf, in_chan, out_chan=1):
8          super(EncoderDecoderConvLSTM, self).__init__()
9
10         self.encoder_1_convlstm = ConvLSTMCell(input_dim=
               in_chan,
11                                                hidden_dim=nf,
12                                                kernel_size
                                                      =(5, 5),
13                                                bias=True)
14
15         self.encoder_2_convlstm = ConvLSTMCell(input_dim=nf,
16                                                hidden_dim=nf,
17                                                kernel_size
                                                      =(3, 3),
18                                                bias=True)
19
20         self.decoder_1_convlstm = ConvLSTMCell(input_dim=nf,
               # nf + 1
21                                                hidden_dim=nf,
22                                                kernel_size
                                                      =(3, 3),
23                                                bias=True)
24
25         self.decoder_2_convlstm = ConvLSTMCell(input_dim=nf,
```

```
                                                hidden_dim=nf,
                                                kernel_size
                                                    =(5, 5),
                                                bias=True)

        self.decoder_CNN = nn.Conv3d(in_channels=nf,
                                     out_channels=out_chan,
                                     kernel_size=(1, 3, 3),
                                     padding=(0, 1, 1))


    def autoencoder(self, x, seq_len, future_step, h_t, c_t,
        h_t2, c_t2, h_t3, c_t3, h_t4, c_t4):

        outputs = []

        # encoder
        for t in range(seq_len):
            h_t, c_t = self.encoder_1_convlstm(input_tensor=x
                [:, t, :, :], cur_state=[h_t, c_t])
            h_t2, c_t2 = self.encoder_2_convlstm(input_tensor
                =h_t, cur_state=[h_t2, c_t2])
        # encoder_vector
        encoder_vector = h_t2

        # decoder
        for t in range(future_step):
            h_t3, c_t3 = self.decoder_1_convlstm(input_tensor
                =encoder_vector, cur_state=[h_t3, c_t3])
            h_t4, c_t4 = self.decoder_2_convlstm(input_tensor
                =h_t3, cur_state=[h_t4, c_t4])
            encoder_vector = h_t4
            outputs += [h_t4]  # predictions

        outputs = torch.stack(outputs, 1)
        outputs = outputs.permute(0, 2, 1, 3, 4)
        outputs = self.decoder_CNN(outputs)
        outputs = torch.nn.ReLU()(outputs)
```

```
58
59          return outputs
60
61      def forward(self, x, future_seq=0, hidden_state=None):
62          """
63          input_tensor:
64              5-D Tensor of shape (b, t, c, h, w)        #
                    batch, time, channel, height, width
65          """
66          # find size of different input dimensions
67          b, seq_len, _, h, w = x.size()
68
69          # initialize hidden states
70          h_t, c_t = self.encoder_1_convlstm.init_hidden(
                batch_size=b, image_size=(h, w))
71          h_t2, c_t2 = self.encoder_2_convlstm.init_hidden(
                batch_size=b, image_size=(h, w))
72          h_t3, c_t3 = self.decoder_1_convlstm.init_hidden(
                batch_size=b, image_size=(h, w))
73          h_t4, c_t4 = self.decoder_2_convlstm.init_hidden(
                batch_size=b, image_size=(h, w))
74
75          # autoencoder forward
76          outputs = self.autoencoder(x, seq_len, future_seq,
                h_t, c_t, h_t2, c_t2, h_t3, c_t3, h_t4, c_t4)
77
78          return outputs
```

B.3 Lo script di training

```
1 from utils.data_lightning import preloading
2 from utils.data_lightning import otf
3 import numpy as np
4 from tqdm import tqdm
5 import matplotlib.pyplot as plt
6 import matplotlib as mat
```

```
7  import argparse
8  import torch as th
9  import torch.nn as nn
10 import os
11 from datetime import datetime
12 import matplotlib as mpl
13 import torch.optim as optim
14 import time
15
16 from models.ae import seq2seq_ConvLSTM
17
18 mat.use("Agg") # headless mode
19
20 # ---- Metriche
21 def accuracy(prediction, target, threshold = 1e-2):
22     # totale celle bagnate del fiume nel target
23     total = (target * prediction).cpu().detach().numpy()
24     total = np.array(total > 0).astype(int) # TP + TN + FP +
           FN
25
26     # totale celle bagnate del fiume nella previsione
27     diff = np.abs((target - prediction).cpu().detach().numpy
           ())
28     correct_cells = (diff < threshold).astype(int)
29     correct_cells = correct_cells*total # TP + TN
30
31     accuracy = np.sum(correct_cells)/np.sum(total)
32     return accuracy
33
34 def rss_loss(input, target):
35     return th.sum((target - input) ** 2)
36
37 parser = argparse.ArgumentParser(description='Trains a given
       model on a given dataset')
38
39 (...) argomenti arg_parse
40
41 args = parser.parse_args()
```

```
42
43 # ---- Setting up the run
44
45 (...) creazione cartella della sessione di training
46
47 plotsize = 15
48
49 # Classe Wrapper di pytorch per implementare gli oggetti
       DataLoader e DataPartitions
50 dataset = preloading.SWEDataModule(
51     root=args.root,
52     test_size=args.test_size,
53     val_size=args.val_size,
54     past_frames=args.past_frames,
55     future_frames=args.future_frames,
56     partial=args.partial,
57     filtering=args.filtering,
58     batch_size=args.batch_size,
59     workers=args.workers,
60     image_size=args.image_size,
61     shuffle=False,
62     dynamicity=100,
63     caching=args.caching,
64     downsampling=args.downsampling)
65
66 dataset.prepare_data()
67 batches = len(dataset.train_dataloader())
68 sequences = batches * args.batch_size
69
70 # ---- Model
71 net = seq2seq_ConvLSTM.EncoderDecoderConvLSTM(nf=args.filters
       , in_chan=args.in_channels, out_chan=args.out_channels)
72
73 # Parallelism
74 if th.cuda.is_available():
75     dev = "cuda:0"
76 else:
77     dev = "cpu"
```

```
78  device = th.device(dev)
79
80  # Wrapping del modello in caso di multi-gpu
81  if th.cuda.device_count() > 1:
82    print("[!] Yay! Using ", th.cuda.device_count(), "GPUs!")
83    net = nn.DataParallel(net)
84
85  net = net.to(device)
86
87  # ---- Training loop
88  optimizer = optim.Adam(net.parameters(), lr=args.
        learning_rate, weight_decay=1e-5) # L2, Ridge Regression
89  losses = []
90  avg_losses = []
91  errors = []
92  test_errors = []
93  epochs = args.epochs
94
95  for epoch in range(epochs):  # loop over the dataset multiple
        times
96
97      print("---- Epoch {}".format(epoch))
98      epoch_start = time.time()
99      training_times = []
100     query_times = []
101     query_start = time.time()
102
103     # itera il dataset
104     iter_dataset = tqdm(dataset.train_dataloader())
105     for batch in iter_dataset:
106         query_end = time.time()
107         query_times.append(query_end-query_start)
108
109         x, y = batch
110
111         optimizer.zero_grad()
112
```

```
113         if args.otf: # otf batches have an extra useless
                dimension
114             x = x[:,0]
115             y = y[:,0]
116
117         x = x.float().to(device)
118         y = y.float().to(device)
119
120         # ---- Predicting
121         start = time.time()
122         outputs = net(x, args.future_frames)  # 0 for layer
                index, 0 for h index
123
124         # concentrazione sul tile centrale solamente
125         center = outputs.shape[3]//3
126
127         # ---- Batch Loss
128         loss = rss_loss(outputs[:, :args.out_channels, 0,
                center:2*center, center:2*center], y[:, 0, :args.
                out_channels, center:2*center, center:2*center])
129         acc = accuracy(outputs[:, :args.out_channels, 0,
                center:2*center, center:2*center], y[:, 0, :args.
                out_channels, center:2*center, center:2*center],
                threshold=args.accuracy_threshold)
130
131         loss.backward()
132         optimizer.step()
133
134         end = time.time()
135         training_times.append(end - start)
136
137         losses.append(loss.item())
138         query_start = time.time()
139
140         # ----- Calcolo metriche sul test set (running test
                accuracy)
141         if args.test_size > 0:
142             size = len(dataset.datasets[1])
```

```
143             random_test_batch = dataset.datasets[1][random.
                    randint(0, size - 1)]
144             x_test, y_test = batch
145             x_test = x_test.float().to(device)
146             y_test = y_test.float().to(device)
147             test_outputs = net(x_test, 1)
148             test_acc = accuracy(test_outputs[:, :3, 0, center
                    :2*center, center:2*center], y_test[:, 0, :3,
                    center:2*center, center:2*center], threshold=
                    args.accuracy_threshold)
149
150             writer.add_scalars('accuracy', {'train': acc,
151                                             'test': test_acc,
152                                             }, epoch * len(
                                                    dataset.
                                                    train_dataloader
                                                    ()) + i)
153         else:
154             test_acc = 0
155             writer.add_scalar('train_accuracy',
156                               acc,
157                               epoch * len(dataset.
                                      train_dataloader()) + i)
158
159         # Average loss ogni 3 steps
160         if i % 3:
161             writer.add_scalar('avg training loss',
162                               np.mean(losses),
163                               epoch)
164
165         iter_dataset.set_postfix(
166             loss=np.mean(losses),
167             train_acc=acc,
168             test_acc=test_acc,
169             fwd_time=np.mean(training_times),
170             query_time=np.mean(query_times)
171         )
172
```

```
173     epoch_end = time.time()
174     print("\navg.loss {:.2f}\ttook {:.2f} s\tavg. inference
            time {:.2f} s\tavg.query time/batch {:.2f} s"
175             .format(epoch, np.mean(losses), epoch_end-
                    epoch_start, np.mean(training_times), np.mean(
                    query_times)))
176     avg_losses.append(np.mean(losses))
177
178     # checkpoint weights
179     th.save(net.state_dict(), weights_path)
180
181 end = time.time()
182 print(end - start)
183
184 print('[!] Finished Training, storing final weights...')
185
186 # Loss plot
187 mpl.rcParams['text.color'] = 'k'
188
189 plt.title("average loss")
190 plt.plot(range(len(avg_losses)), avg_losses)
191 plt.savefig("runs/" + foldername + "/avg_loss.png")
192 plt.clf()
193
194 print("Avg.training time: {}".format(numpy.mean(
        training_times)))
```

B.4 Lo script di test

```
1 # %%
2 import os
3 from datetime import datetime
4 from utils.data_lightning.otf import SWEDataset
5 import numpy as np
6
7 import matplotlib.pyplot as plt
```

```
8  import matplotlib as mat
9
10 import torch as th
11 from torch.autograd import Variable
12
13 from models.experiments.nfnets import seq2seq_NFLSTM
14 from models.ae import seq2seq_ConvLSTM
15
16 import argparse
17 import pytorch_ssim
18 import time
19
20 mat.use("Agg") # headless mode
21
22 def accuracy(prediction, target, threshold = 1e-2):
23
24     total = (target * prediction).cpu().detach().numpy()
25     total = np.array(total > 0).astype(int) # TP + TN + FP +
           FN
26
27     diff = np.abs((target - prediction).cpu().detach().numpy
           ())
28     correct_cells = (diff < threshold).astype(int)
29     correct_cells = correct_cells*total # TP + TN
30
31     accuracy = np.sum(correct_cells)/np.sum(total)
32     return accuracy
33
34 def str2bool(v):
35     if isinstance(v, bool):
36         return v
37     if v.lower() in ('yes', 'true', 't', 'y', '1'):
38         return True
39     elif v.lower() in ('no', 'false', 'f', 'n', '0'):
40         return False
41     else:
42         raise argparse.ArgumentTypeError('Boolean value
               expected.')
```

```
43
44 # --------------------------------
45
46 (...) preprocessing argomenti dello script
47
48 # -------------- Setting up the run
49
50 (...) preparazione directories
51
52 # -------------- Data definition
53 if th.cuda.is_available():
54     dev = "cuda:0"
55 else:
56     dev = "cpu"
57 device = th.device(dev)
58
59 plotsize = 15
60
61 # -------------- Model
62 net = seq2seq_ConvLSTM.EncoderDecoderConvLSTM(nf=args.filters
       , in_chan=args.in_channels, out_chan=args.out_channels)
63
64 if args.multigpu:
65     net = nn.DataParallel(net)
66
67 net.load_state_dict(
68     th.load(args.weights_path, map_location=device)
69 )
70
71 net = net.to(device)
72 net.eval() # evaluation mode
73
74 # ------------------------------
75
76 inference_times = []
77
78 ssim = pytorch_ssim.SSIM()
79 l1 = th.nn.L1Loss()
```

```
80  l2 = th.nn.MSELoss()
81
82  ssim_score = 0
83  l1_score = 0
84  l2_score = 0
85  acc_score = 0
86
87  dataset = SWEDataset(
88      root=args.root,
89      past_frames=args.past_frames,
90      future_frames=args.future_frames,
91      partial=args.partial,
92      dynamicity=args.dynamicity,
93      downsampling=args.downsampling,
94      blur_radius=args.blur_radius
95  )
96
97  # Selezione del primo batch valido (che soddisfi la soglia di
        dinamicitÃ¡)
98  i = np.random.randint(len(dataset))        # random batch
99  datapoint = dataset[i]
100
101 while datapoint is None:
102     i = np.random.randint(len(dataset))        # random batch
103     datapoint = dataset[i]
104
105 print("[!] Datapoint loaded!")
106
107 # b, t, c, h, w
108 x_in, y = datapoint
109
110 # ------------- Plotting
111 test_dir = "runs/" + foldername + "/"
112
113 for i, frame in enumerate(x_in[0]):
114     plt.matshow(frame[0].cpu().detach().numpy())
115     plt.savefig(test_dir + "/{}.png".format(i))
116     plt.clf()
```

```
117
118 # Previsione per n steps futuri
119 for s in range(args.future_frames):
120
121     start = time.time()
122     # 1, t, c, h, w
123     outputs = net(x_in, 1)
124     end = time.time()
125     inference_times.append(end - start)
126
127     # Plot delle matrici
128     plt.matshow(outputs[0,0,0].cpu().detach().numpy())
129     plt.savefig(test_dir + "/pred_{}.png".format(s))
130     plt.clf()
131
132     plt.matshow(y[0,s,0].cpu().detach().numpy())
133     plt.savefig(test_dir + "/true_{}.png".format(s))
134     plt.clf()
135
136     # Calcolo delle metriche di valutazione
137     img1 = Variable(outputs[0,0,0].unsqueeze(0).unsqueeze(0),
            requires_grad=False)
138     img2 = Variable(y[0,s,0].unsqueeze(0).unsqueeze(0),
           requires_grad=True)
139
140     acc_score += accuracy(img1, img2, threshold=1e-1)
141     ssim_score += ssim(img1, img2)
142     l1_score += l1(img1, img2)
143     l2_score += l2(img1, img2)
144
145     # Builds a new sequence with its own prediction
146     tmp = th.empty(x_in.shape)
147     tmp[0, :(args.past_frames-1), :, :, :] = x_in[0, 1:, :,
           :, :] # get last n-1 frames
148
149     # output + btm
150     new_frame = th.cat((
151         outputs[:,:,0,:,:],
```

```
152             x_in[:, 0, 3, :, :].unsqueeze(0)
153         ), dim=1)
154 
155         tmp[0, -1, :, :, :] = new_frame
156         x_in = tmp
157 
158         print(". ", end="", flush=True)
159 
160 acc_score = acc_score/args.future_frames
161 ssim_score = ssim_score/args.future_frames
162 l1_score = l1_score/args.future_frames
163 l2_score = l2_score/args.future_frames
164 
165 # Logging
166 stats = "Accuracy: {}\nSSIM: {}\nL1: {}\nMSE:{}\nAvg.
        Inference Time: {}".format(acc_score, ssim_score, l1_score
        , l2_score, np.mean(inference_times))
167 print(stats, flush=True)
168 
169 text_file = open("runs/" + foldername + "/avg_score.txt", "w"
        )
170 n = text_file.write(stats)
171 text_file.close()
```

Bibliografia

[1] Euclide. *Elementi*. 300 a.c. ca.

[2] Muhammad ibn Musa al Khwarizmi. *The Compendious Book on Calculation by Completion and Balancing*. 820 d.c.

[3] Nathaniel Rochester Claude E. Shannon John McCarthy, Marvin L. Minsky. A proposal for the dartmouth summer research project on artificial intelligence. 1955.

[4] Joseph Weizenbaum. Elizaâa computer program for the study of natural language communication between man and machine. 1966.

[5] Seymour Papert Marvin Minsky. A review of "perceptrons: An introduction to computational geometry". *The M.I.T. Press, Cambridge, Mass*, 1969.

[6] Geoffrey E. Hinton Alex Krizhevsky, Ilya Sutskever. Imagenet classification with deep convolutional neural networks. 2012.

[7] Nick Ryder Melanie Subbiah Jared Kaplan Prafulla Dhariwal Arvind Neelakantan Pranav Shyam Girish Sastry Amanda Askell Sandhini Agarwal Ariel Herbert-Voss Gretchen Krueger Tom Henighan Rewon Child Aditya Ramesh Daniel M. Ziegler Jeffrey Wu Clemens Winter Christopher Hesse Mark Chen Eric Sigler Mateusz Litwin Scott Gray Benjamin Chess Jack Clark Christopher Berner Sam McCandlish Alec Radford Ilya Sutskever Dario Amodei Tom B. Brown, Benjamin Mann. Language models are few-shot learners. *OpenAI*, 2020.

[8] Ishan Misra Yann LeCun. Self-supervised learning: The dark matter of intelligence. 2021.

[9] Renato Vacondio Massimiliano Turchetto, Alessandro Dal Palù. A general design for a scalable mpi-gpu multi-resolution 2d numerical solver. 2019.

[10] F. ROSENBLATT. The perceptron: A probabilistic model for information storage and organization in the brain. 1958.

[11] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Deep residual learning for image recognition. 2015.

[12] Hao Wang Dit-yan Yeung Wai-kin Wong Wang-chun Woo Xingjian Shi, Zhourong Chen. Convolutional lstm network: A machine learning approach for precipitation nowcasting. 2015.

[13] Steve Easterbrook Sanja Fidler Wei Yu, Yichao Lu. Crevnet: Conditionally reversible video prediction. 2019.

[14] Marco Varesi. Apprendimento e predizione di dati fluviali tramite reti neurali. *Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università di Parma*, 2018.

[15] Xilin Xia Qiuhua Liang Jeffrey Neal G. Pender Syed Rezwan Kabir, S. Patidar. A deep convolutional neural network model for rapid prediction of fluvial flood inundation. 2020.

[16] Osama Abdeljaber Turker Ince Moncef Gabbouj Daniel J.Inman Serkan Kiranyaz, Onur Avci. A deep convolutional neural network model for rapid prediction of fluvial flood inundation. 2020.

[17] Simone Gazza. Reti neurali per la predizione del livello di allagamento nelle alluvioni. *Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università di Parma*, 2021.

[18] Bo Chen Dmitry Kalenichenko Weijun Wang Tobias Weyand-Marco Andreetto Hartwig Adam Andrew G. Howard, Menglong Zhu. Mobilenets: Efficient convolutional neural networks for mobile vision applications. 2017.

[19] Nicolas Thome Vincent Le Guen. Disentangling physical dynamics from unknown factors for unsupervised video prediction. 2020.

[20] Max Welling Diederik P. Kingma. An introduction to variational autoencoders. 2019.

[21] Niki Parmar Jakob Uszkoreit Llion Jones Aidan N. Gomez-Lukasz Kaiser Illia Polosukhin Ashish Vaswani, Noam Shazeer. Attention is all you need. 2017.

[22] Pieter Abbeel Aravind Srinivas Wilson Yan, Yunzhi Zhang. Videogpt: Video generation using vq-vae and transformers. 2021.

[23] J. Nathan Kutz Steven L. Brunton Kathleen Champion, Bethany Lusch. Data-driven discovery of coordinates and governing equations. 2019.

[24] Juan Carlos Niebles Hsu-kuang Chiu, Ehsan Adeli. Segmenting the future. 2019.