



# 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:

Prof. Alessandro Dal Palù

Correlatore:

Prof. Ing. Renato Vacondio

Tesi di Laurea di:

Diego Calanzone (296787)

---

ANNO ACCADEMICO 2020-2021

# 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 dei 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 più a sud lungo il fiume. Il target di previsione è 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 misurazioni.

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

- Nel 2021, S.Gazza<sup>[17]</sup> 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 già 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):

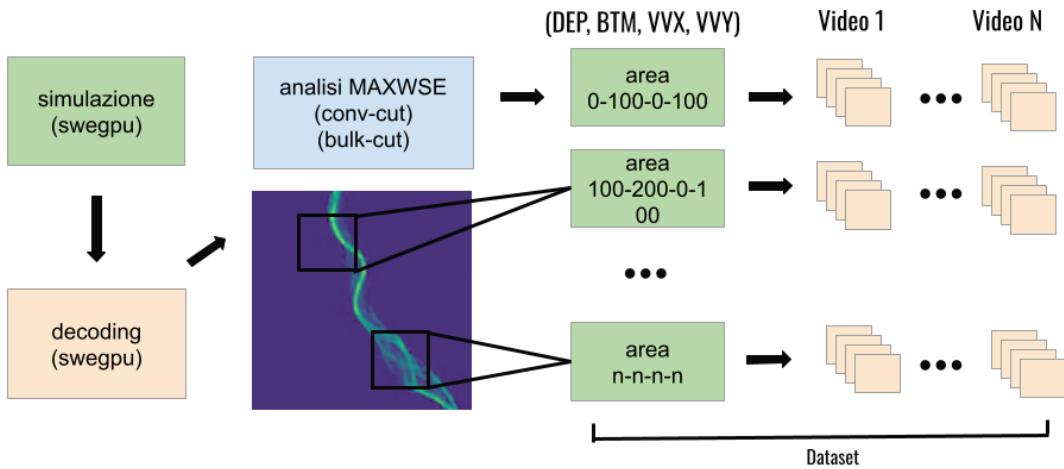


Figura 5.1: Schema di generazione del dataset: simulazione, decoding, filtraggio e sequenzializzazione.

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, cioè 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 è 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 è 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):

```
1 # iterato per DEP, BTM, VVX, VVY, MAXWSE
2 ./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:

1	4.71753e-01	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00
2	9.92509e-01	1.74219e-01	0.00000e+00	0.00000e+00	0.00000e+00
3	9.63321e-01	9.446960e-01	0.00000e+00	0.00000e+00	0.00000e+00
4	9.66145e-01	8.77755e-01	8.62921e-01	0.00000e+00	0.00000e+00
5	1.23323e+00	9.51391e-01	9.75382e-01	8.88735e-01	0.00000e+00
6	1.19594e+00	1.24599e+00	1.10181e+00	1.08521e+00	7.04663e-01
7	1.02829e+00	1.26774e+00	1.34795e+00	1.29785e+00	1.13589e+00
8	9.50625e-01	1.03486e+00	1.24184e+00	1.41947e+00	1.37174e+00
9	7.10541e-01	8.89352e-01	1.04285e+00	1.32930e+00	1.49813e+00
10	8.63403e-01	8.63929e-01	9.73405e-01	9.34986e-01	1.30047e+00
11	1.09732e+00	7.09958e-01	7.50827e-01	9.51506e-01	9.91025e-01
12	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*:  $6,784 \times 4,602 = 31,219,968$  celle

- *Torrente Baganza:*  $3,584 \times 2,816 = 10,092,544$  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. È 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 * ampiezza2
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'andamento della simulazione, l'allagamento. Spesso la lunghezza di questa sequenza è 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  $n - \text{lunghezza}$  frames.

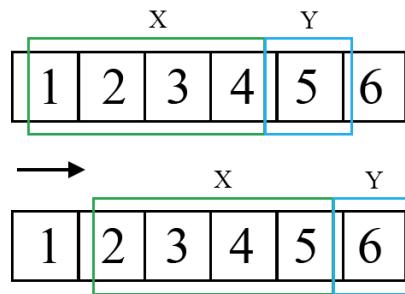


Figura 5.2: Sottosequenze con 4 frames in input e 1 frame target, scorrimento di 1 frame.

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* è la lista delle sequenze in input, identificate dall'id del frame di partenza; *Y* è un dizionario che associa le sequenze in input con le sequenze target, cioè 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):

```

1  [
2   (
3     'area-0-0-0-0' ,
4     [
5       ...
6       "id-7" ,
7       "id-8" ,
8       ...
9     ] ,
10    [
11      ...
12      "id-7": [14, 15, 16, 16] ,
13      "id-8": [15, 16, 17, 18]
14      ...
15    ]
16  )
17 ]

```

### 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 è 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  $4 \times 768 \times 768 = 2,359,296$  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  $120 \times 240 \times 3 = 86,400$  matrici; il peso complessivo di un dataset è di  $2,359,296 \times 86,400 = 203,843,174,400$  bytes  $\approx \mathbf{203.4 \text{ GB}}$ . Generando sotto-sequenze temporali con l'algoritmo a finestra di scorrimento, ogni frame può ricorrere in memoria più volte e il peso del dataset aumenta oltre le risorse disponibili. Quando vengono richieste sequenze di frame sovrapposte, è 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 più 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 è 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 ( $\rightarrow$  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 ( $\rightarrow$  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 ( $\rightarrow$  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 può accumularsi alcuna quantità di fluido, pertanto possono essere escluse. Questi valori vengono sostituiti con zeri e non cambieranno mai valore ( $\rightarrow$  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 ( $\rightarrow$  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 ( $\rightarrow$  appendice A.3, riga 153-174).
7. Si scartano le sequenze in cui non c'è una sostanziale differenza tra i frames, quindi non avviene alcun evento interessante. La "dinamicità"  $\delta$  di una sequenza é stimata con la formula:

$$\delta_i = 1 - \text{accuracy} = 1 - \frac{TP + TN}{TP + TN + FP + FN} \quad (5.1)$$

$$\forall i \in [m + 1, \dots, m + k]$$

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 ( $\rightarrow$  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) ( $\rightarrow$  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 è richiesta da PyTorch Lightning per utilizzare anche i loop di training default della classe *pytorchlightning.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 è **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. È così possibile creare più ambienti per contesti diversi e trasferire con facilità e modularità l'intero spazio di sviluppo dal personal computer al cluster HPC. **Anaconda** è una distribuzione per Python ed R con il compito di semplificare la gestione delle librerie, questa è 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:

```
1 conda create -n swe-deeplearning python=3.6
2 source activate swe-deeplearning
3 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 costruito sopra PyTorch puro, permettendo di separare l'aspetto ingegneristico da quello di ricerca con modelli modulari e facilmente modificabili.

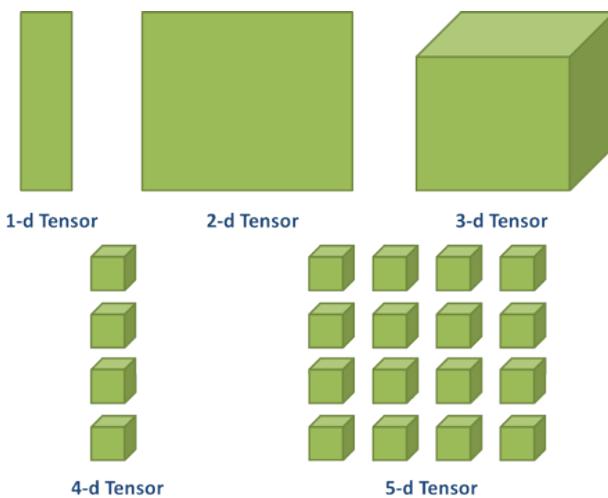


Figura 5.3: Visualizzazione di un tensor con dimensioni variabili.

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 tensori utilizzando 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*:

$$h' = (f \circ g)' = (f' \circ g) \cdot g' \quad (5.2)$$

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**.

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}} \quad (5.3)$$

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

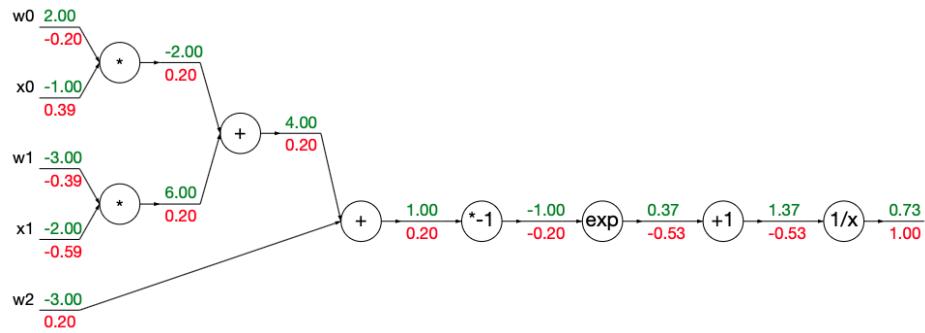


Figura 5.4: Grafo computazionale della funzione sopra indicata, in verde i risultati del forward pass, in rosso i gradienti (backward pass).

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.

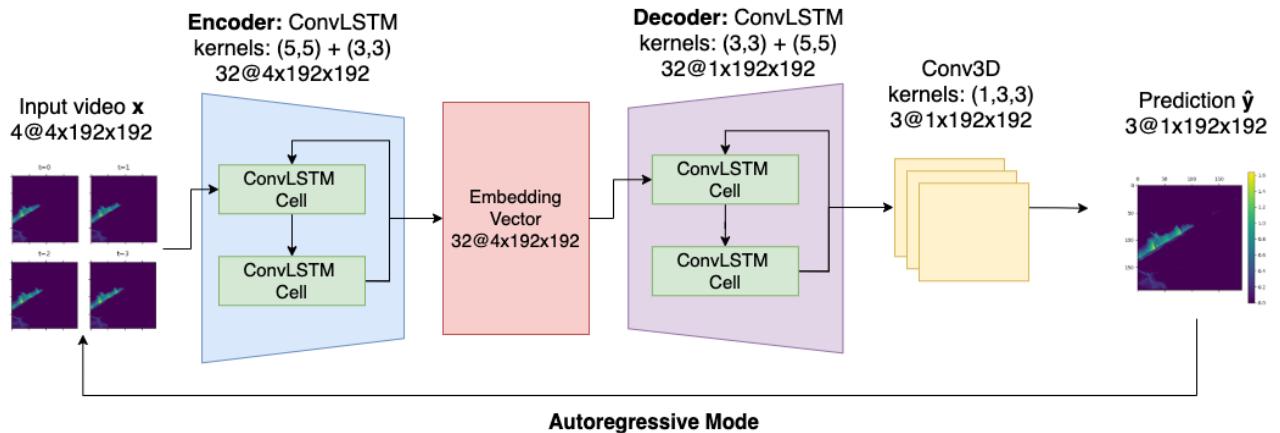


Figura 5.5: Architettura ConvLSTM auto-encoder a 32 filtri ed esempio della generazione del next-frame.

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 concentrerà su quest’ultima modalità 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_conv_lstm	ConvLSTMCell	(5,5)	32.1 K
1	encoder_2_conv_lstm	ConvLSTMCell	(3,3)	18.5 K
2	decoder_1_conv_lstm	ConvLSTMCell	(3,3)	18.5 K
3	decoder_2_conv_lstm	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_conv_lstm	ConvLSTMCell	(5,5)	115 K
1	encoder_2_conv_lstm	ConvLSTMCell	(3,3)	73.9 K
2	decoder_1_conv_lstm	ConvLSTMCell	(3,3)	73.9 K
3	decoder_2_conv_lstm	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 è stato effettuato in più 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 (dinamicità):** 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):

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

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

$$\begin{aligned} sum \left( \left( \begin{bmatrix} 3.2 & 0 & 0 \\ 6.2 & 4.5 & 0 \\ 2.3 & 0 & 1.1 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 1.8 \\ 0 & 0 & 0 \\ 0.3 & 0 & 9.1 \end{bmatrix} \right)^2 \right) &= sum \left( \begin{bmatrix} 3.2 & 0 & -1.8 \\ 6.2 & 4.5 & 0 \\ 2 & 0 & -8 \end{bmatrix}^2 \right) = 140.17 \\ sum \left( \left( \begin{bmatrix} 3.2 & 0 & 0 \\ 6.2 & 4.5 & 0 \\ 2.3 & 0 & 1.1 \end{bmatrix} - \begin{bmatrix} 3.2 & 0 & 0.5 \\ 6.2 & 0 & 0 \\ 2.3 & 0 & 1.1 \end{bmatrix} \right)^2 \right) &= sum \left( \begin{bmatrix} 0 & 0 & -0.5 \\ 0 & 4.5 & 0 \\ 0 & 0 & 0 \end{bmatrix}^2 \right) = 20.50 \end{aligned}$$

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

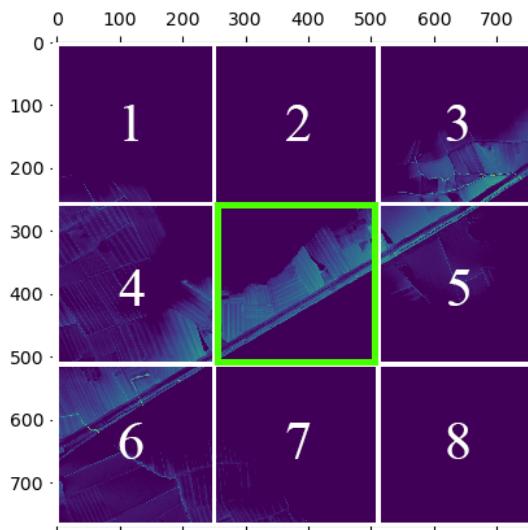


Figura 5.6: Fotogramma 768x768 scomposto in 3x3 tiles, al centro il tile target con le condizioni al contorno dettate dai tiles vicini (numerati).

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)**:

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

Dove  $y_i$  é il target,  $\beta_0$  il bias e  $\beta_j$  i parametri della rete neurale;  $\lambda$  é un coefficiente di regolarizzazione, per  $\lambda \rightarrow \infty$  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 quantità 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  $\pm 5\text{cm}$  ( $P=\text{bagnate}$ ,  $N=\text{asciutte}$ ).

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (6.1)$$

- **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.

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

- **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.

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

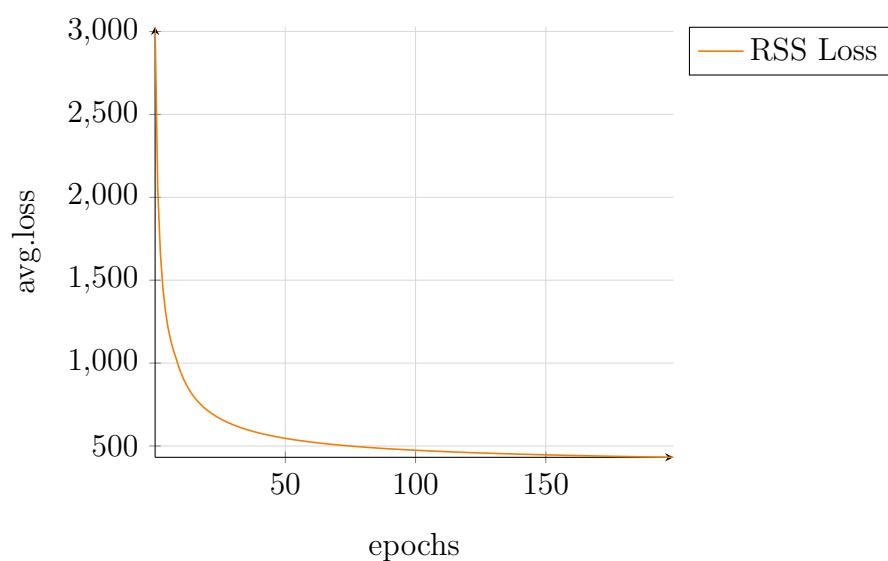
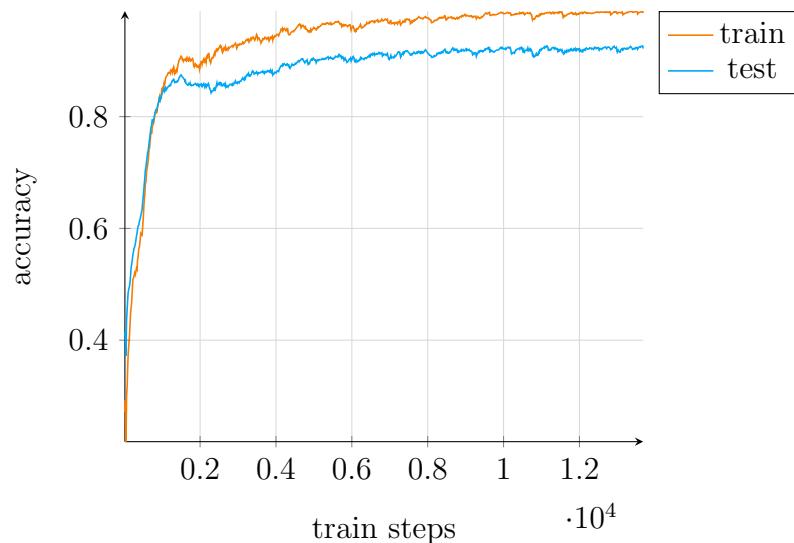
Le misure di accuracy nei grafici seguenti sono sottoposte a *smoothing* utilizzando una **exponential weighthed function** ricorsiva:

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

Questo permette una visualizzazione più 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**.



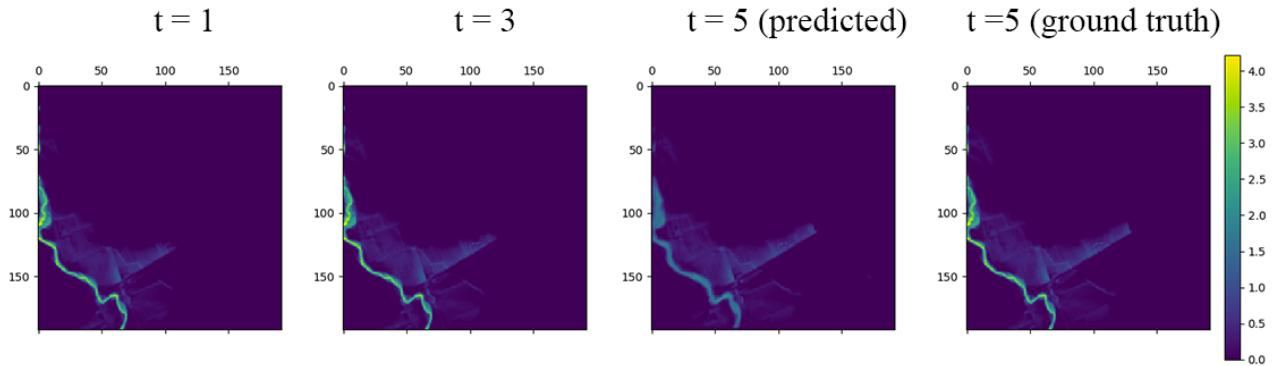
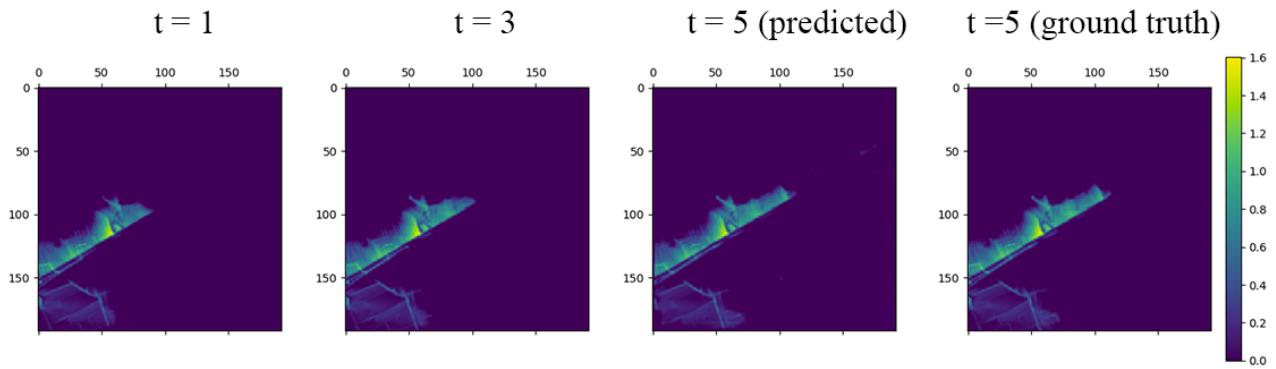
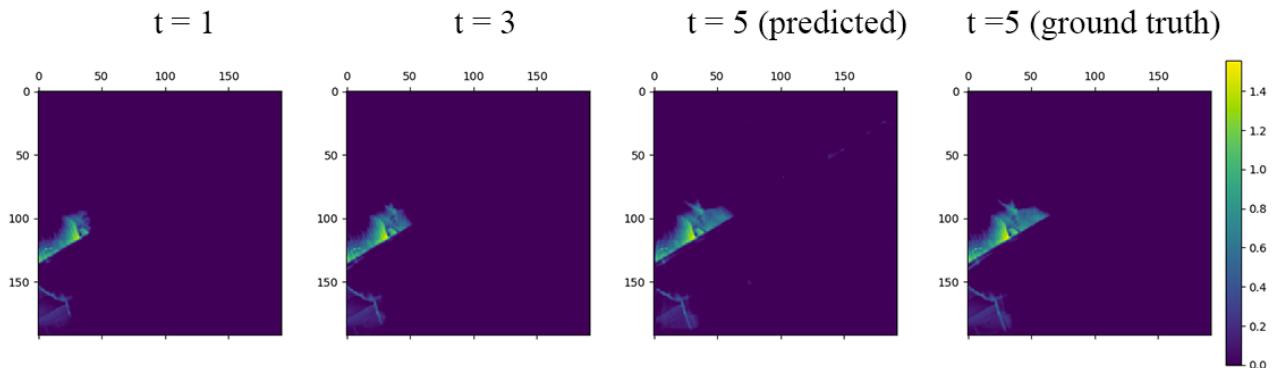
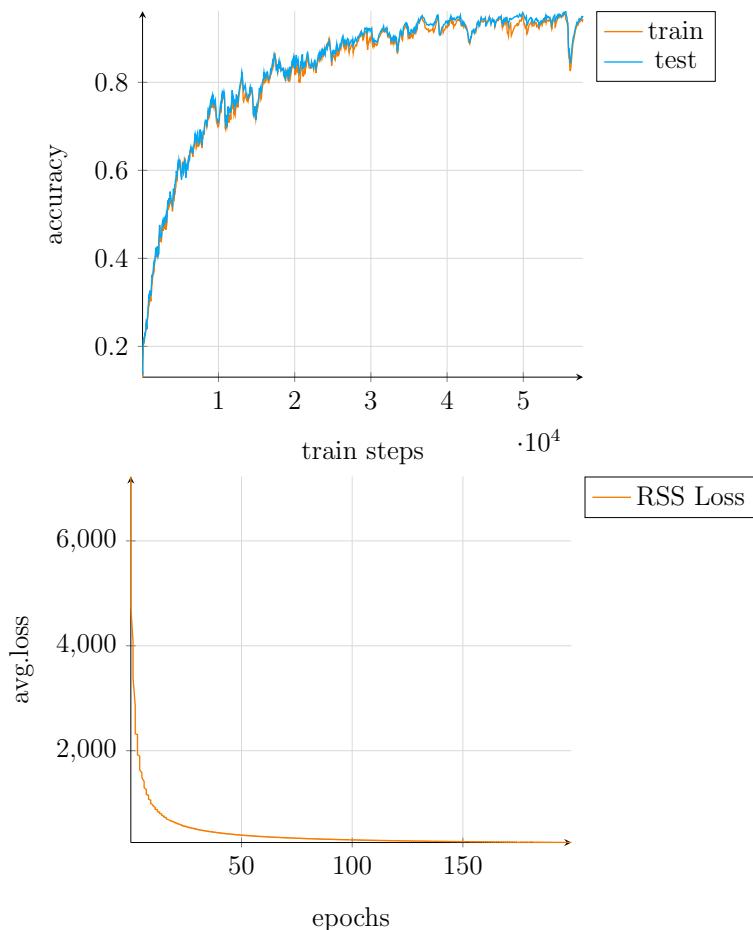
Figura 6.1: Confronto next-frame, zona **arda\_1**.Figura 6.2: Confronto next-frame, zona **arda\_2**.Figura 6.3: Confronto next-frame, zona **arda\_3**.

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**.



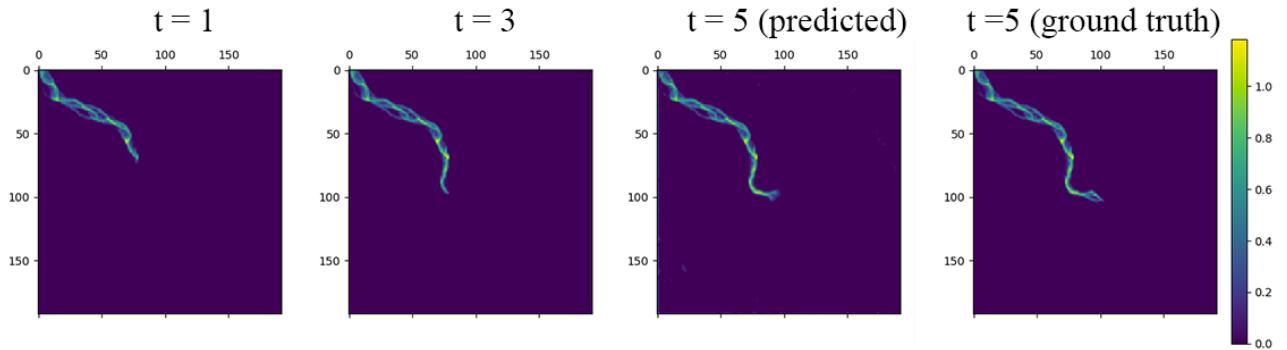
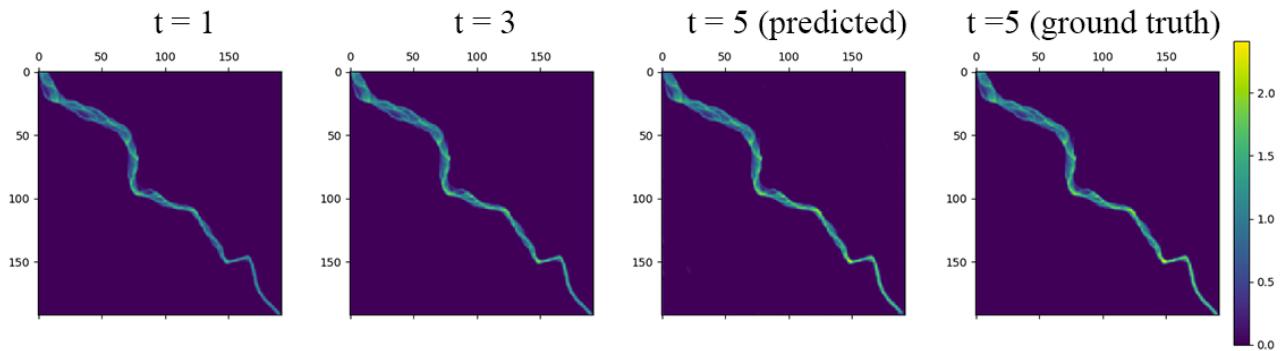
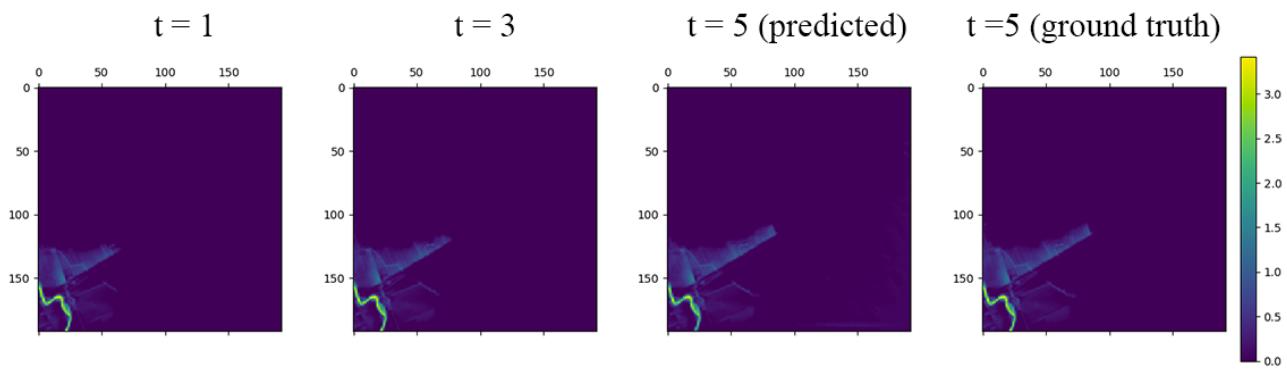
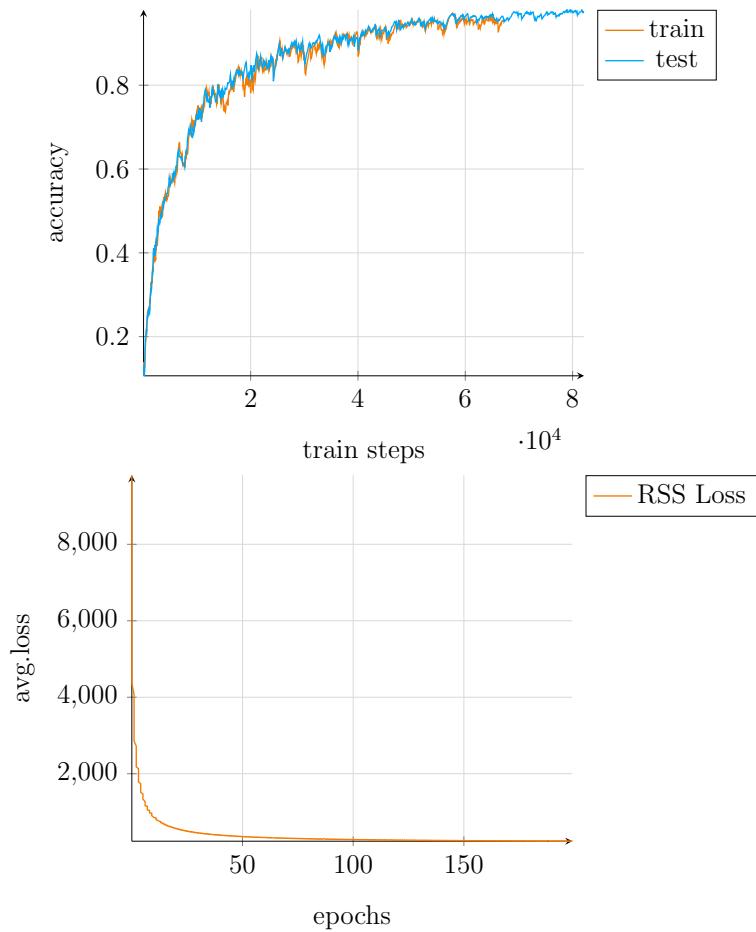
Figura 6.4: Confronto next-frame, zona **Baganza**.Figura 6.5: Confronto next-frame, zona **Baganza**.Figura 6.6: Confronto next-frame, zona **Arda**.

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

Zona	Accuracy $\pm 5\text{cm}$	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**.



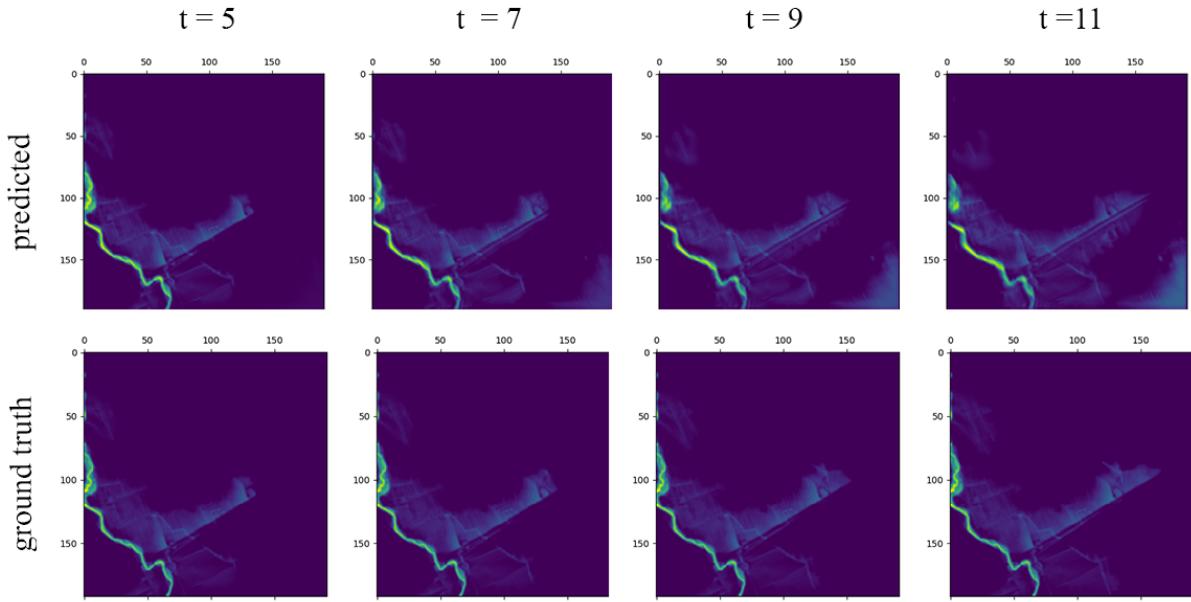


Figura 6.7: Confronto 4 frames futuri (auto-regressive), torrente **Arda**.

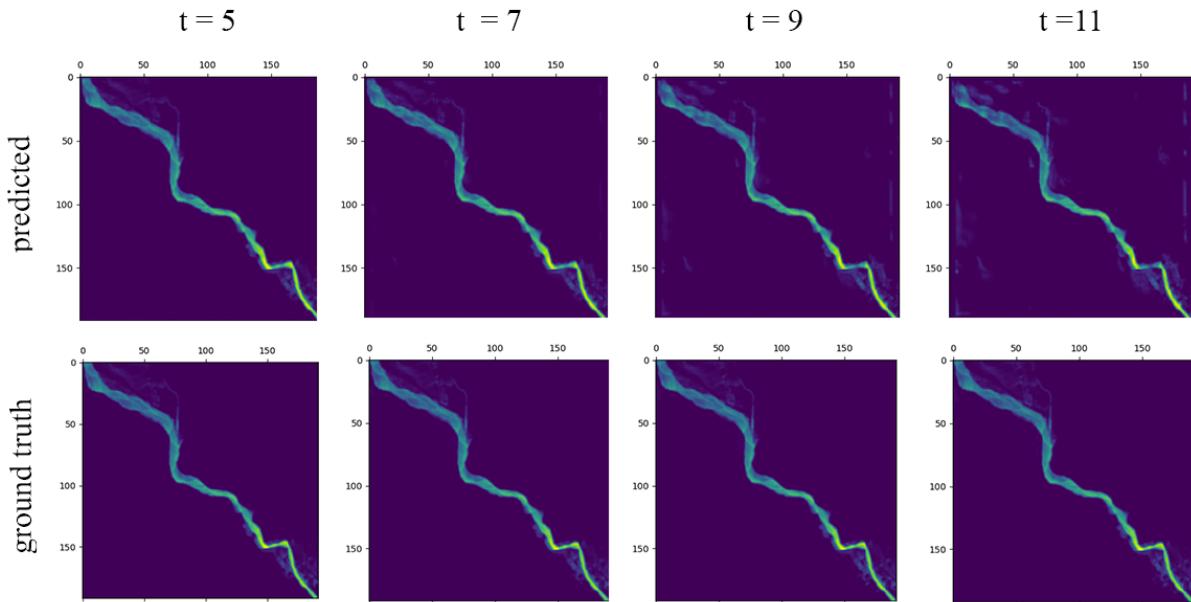


Figura 6.8: Confronto 4 frames futuri (auto-regressive), torrente **Baganza**.

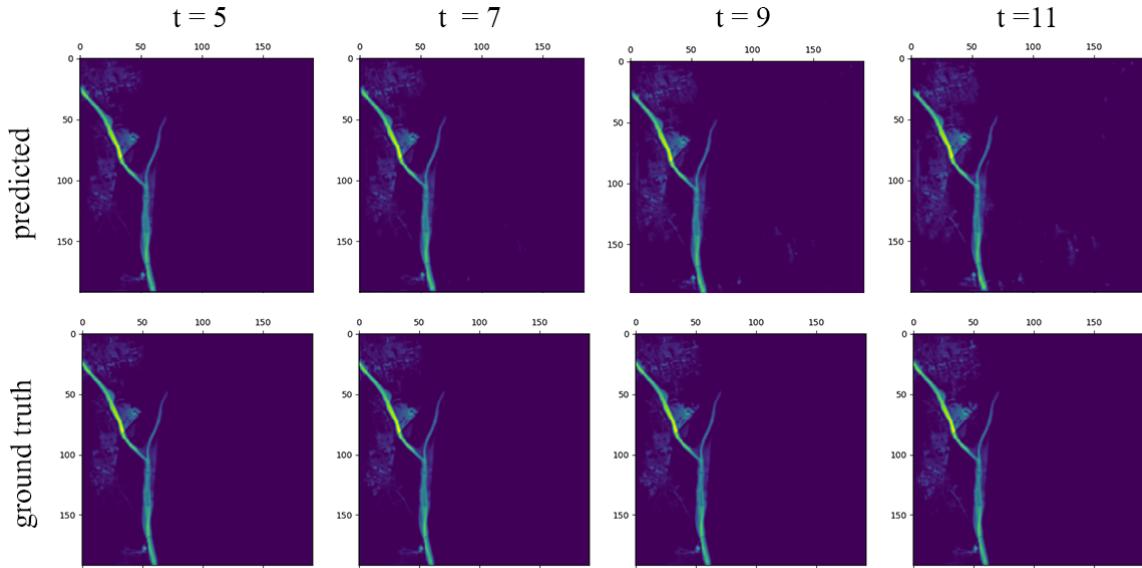


Figura 6.9: Confronto 4 frames futuri (auto-regressive), torrente **Baganza**.

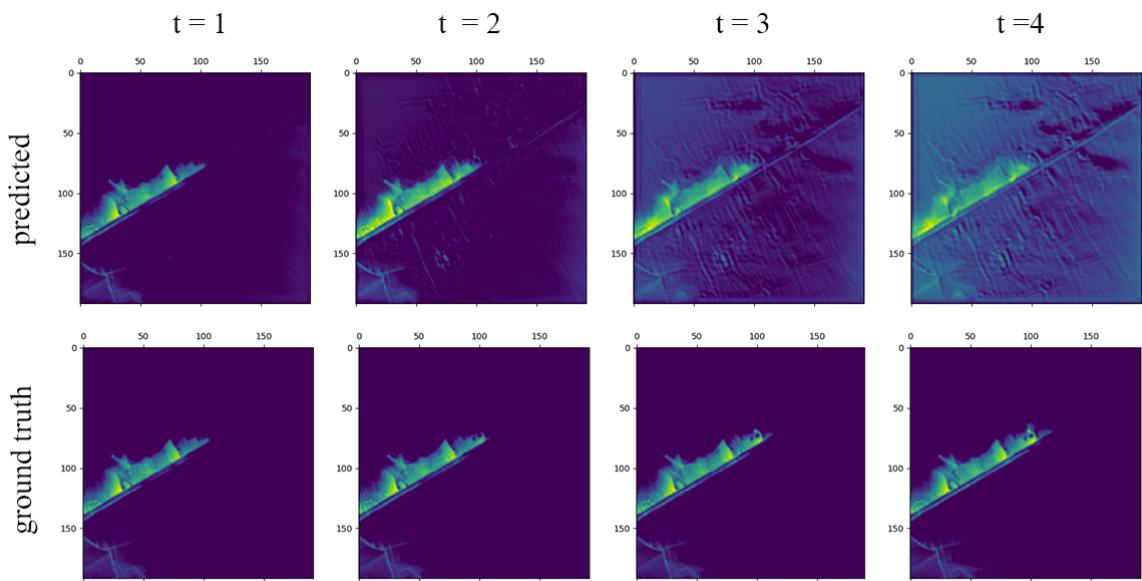


Figura 6.10: Confronto 4 frames futuri (seq2seq), downsampling 4x, torrente **Baganza**.

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

Zona	Accuracy $\pm 5\text{cm}$	SSIM	MSE (L2 Loss)	MAE (L1 Loss)
Arda <sup>1</sup>	59.59%	81.74%	0.0657	0.0422
Baganza <sup>1</sup>	61.72%	86.85%	0.0302	0.0144
Arda <sup>2</sup>	44.87%	56.93%	0.1072	0.0370

Questa sessione di training é mirata al prevedere piú di un frame futuro. In modalità **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 già 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.

# 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.