

SimCLR-TS-CSI

Proponiamo un framework per il rilevamento delle anomalie su dati sequenziali, in particolare su serie temporali. Il modello viene allenato in modo self-supervised utilizzando il contrastive learning. La rete è costruita sulla base del paper (¹), in cui viene proposta un'implementazione del modello SimCLR adattata alle serie temporali (SimCLR-TS), e del paper (²), in cui viene proposta una variazione di SimCLR per migliorare la capacità della rete nel rilevamento delle anomalie. Il dataset utilizzato è TEP ed è lo stesso proposto nel paper (¹).

Struttura della rete

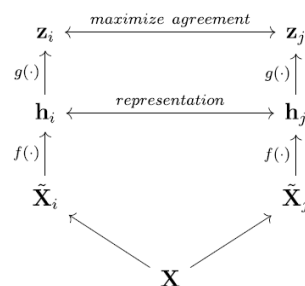


Fig.1 Contrastive Learning presentato in SimCLR

La rete è molto semplice, consiste di tre livelli consecutivi Conv1d - LeakyReLU - BatchNorm1d e di un livello lineare terminale per fare la classificazione. In tutti i livelli convolutivi il kernel ha dimensione 3, lo stride è 2 nei primi due mentre è 1 nell'ultimo. Il numero di filtri convolutivi è 64 nel primo livello, 128 nel secondo e 256 nel terzo. Nell'implementazione originale di SimCLR la $f(\cdot)$ è ResNet-50 in quanto lavora su dati di tipo immagine. Per adattare il modello alle serie temporali la $f(\cdot)$ viene sostituita con un encoder costituito da livelli Conv1d, adatti ad estrarre informazioni lungo il solo asse temporale da dati che non posseggono informazioni spaziali. Poiché non risultano evidenti vantaggi dall'uso della funzione non lineare $g(\cdot)$, in (¹) si cerca direttamente di minimizzare la distanza tra le rappresentazioni latenti h_i e h_j ottenute dall'encoder.

Viene utilizzata una contrastive loss basata su cosine similarity che segue la formula:

$$l_{i,j} = -\log\left(\frac{\exp(\text{sim}(h_i, h_j)/\tau)}{\sum_{k=1, k \neq i}^{2N} \exp(\text{sim}(h_i, h_k)/\tau)}\right)$$

La struttura della rete è la seguente:

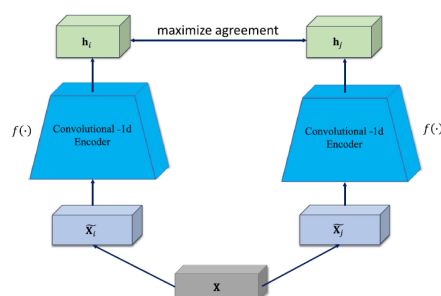


Fig.2 SimCLR-TS

Dataset

I dati provengono dal dataset TEP ³, una raccolta di dati temporali composti da 52 canali con anomalie annotate, suddivisi in training e test secondo quanto suggerito in (¹). Il dataset contiene 22 classi, la classe 0 corrisponde al comportamento corretto e le restanti classi da 1 a 21 corrispondono a diversi comportamenti anomali. Per le classi da 1 a 21 vengono forniti 480 campioni per canale come dati di training e 800 campioni come dati di test. Per la classe 0 vengono forniti 500 campioni di training e 960 campioni di test. In totale sono disponibili 10560 campioni per il training e 17760 campioni per il test. Entrambi i dataset sono costruiti in modo che le classi siano distribuite uniformemente e nessuna prevalga sulle altre.

Tutti dati vengono normalizzati prima dell'uso, usando la media e la varianza della porzione di training. Per creare i sample da dare in input alla rete, vengono create delle finestre di ampiezza $T=100$ contenenti T campioni consecutivi. Di conseguenza ogni segnale in input ha una dimensione $52 \times T$. Viene usato un approccio di tipo sliding window quindi, dato un segnale s , il segnale successivo inizia da $s + 1$. In tutto per ogni classe saranno disponibili $480-T$ segnali di training ($500-T$ per la classe 0) e $800-T$ segnali di test ($960-T$ per la classe 0).

Trasformazioni (Data Augmentations)

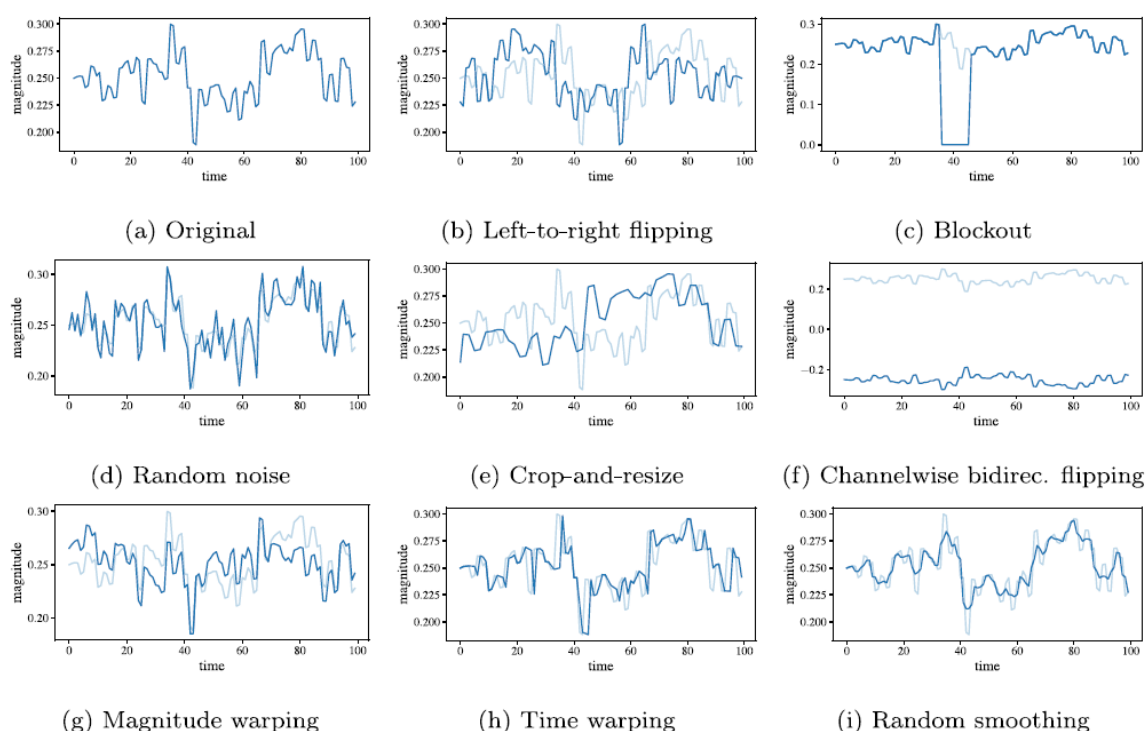


Fig.3 Trasformazioni proposte per SimCLR-TS. Noi abbiamo selezionato [b, c, d, e, g]. A queste si aggiunge la trasformazione 'permute channels' che non viene mostrata perché ha senso solo visualizzando tutti i canali.

Sono state selezionate sei trasformazioni dal paper di (¹) in particolare le tre che forniscono l'accuratezza migliore e le tre peggiori. In questo modo costruiamo due set di trasformazioni che chiameremo Soft Augmentations (che contiene le trasformazioni con accuratezza migliore) e Hard Augmentations (contenente le altre).

Soft Augmentations = { Left to Right, Crop Resize , Random Noise }

1. Left to Right : Il segnale viene capovolto lungo l'asse verticale tramite una moltiplicazione per matrice antidiagonale.
2. Crop Resize : Dal segnale originale si ottiene una copia dilatata nel tempo di un fattore 2, passando da un intervallo T ad uno 2T. I dati intermedi sono generati per interpolazione. Successivamente si campiona un istante nella prima metà del segnale generato (con periodo 2T) e si produce il segnale finale considerando i T istanti successivi. Nota : Con questa procedura avevamo delle prestazioni molto basse e il training diventava troppo lungo. Per un singolo batch di 64 finestre si impiegavano 6 minuti e qualche secondo. Abbiamo cambiato il modo di generare il segnale aumentato scambiando i passi di campionamento e generazione dei seguenti T istanti. Così facendo otteniamo lo stesso risultato ma con una riduzione del 45% del tempo di esecuzione dato che i cicli da eseguire per generare il segnale sono stati dimezzati. Il tempo di esecuzione di un batch è passato da 6 minuti a 3min 15 sec circa.
3. Random Noise : Applica il rumore bianco al segnale aggiungendo (o sottraendo) la deviazione standard del segnale moltiplicato per una variabile campionata da una distribuzione uniforme nel range (-1, 1).

Hard Augmentations : { Blockout , Magnitude Warping, Permute Channels }

1. Blockout : Azzerà una porzione del segnale a partire da un campione casuale.
2. Magnitude Warping : Somma una funzione sinusoidale al segnale con possibilità di cambiare frequenza e modulo.
3. Permute Channels : Permuta i canali del segnale.

Applicazione delle trasformazioni

Il primo degli obiettivi è quello di capire se la combinazione di trasformazioni (e tutti i possibili ordinamenti di applicazione) danno un risultato superiore in termini di accuratezza rispetto a quelli riscontrati nel paper (¹).

Poiché le configurazioni da testare sono molte, è necessario settare nel modo corretto il file di configurazione "*config_augmenter.json*" nella cartella `/config`.

Si ricorda che data una TS, l'ordine delle trasformazioni (se tutti i flag di questo file sono true) consiste nell'applicare prima le hard augmentations e successivamente le soft augmentations. Questi due blocchi possono essere attivati o disattivati in modo indipendente tramite due flag, come spiegato di seguito.

Il flag `is_hard_augm = false` evita l'applicazione di una hard augmentation. Il flag `is_hard_augm = true` attiva l'applicazione di una hard augmentation. Viene selezionata una dalla lista `hard_augm_list` e vengono provate tutte, singolarmente, per ogni training.

Il flag `is_multiple_augm = true` attiva le combinazioni di varie soft augmentations generandole a partire dalla lista `soft_augm_list`. Se invece è impostato a false saranno applicate le soft augmentation singolarmente, per ogni training.

Esempi di combinazioni :

- **Default** : `is_hard_augm = is_multiple_augm = False` → Se entrambi sono a false, l'algoritmo applicherà le singole soft augmentations , una per volta per ogni training. In questo modo per ogni soft augmentation della lista alleniamo un modello e otteniamo l'accuratezza.
- **Singola Hard + Singola Soft** : `is_hard_augm = True and is_multiple_augm = False` → Come per il caso precedente ma si aggiunge un pezzo ovvero si applica una delle hard augmentation e a seguire una delle soft augmentation. Con questa configurazione si generano `#HardAugm * #SoftAugm` modelli.

- **Singola Hard + Soft Multiple :**

is_hard_augm = is_multiple_augm = True → Siamo nel caso più complesso dove per ogni hard augmentation vengono applicate tutte le possibili disposizioni $D_{n,k}$ delle soft augmentation con $k = 2$ fino a n . Otteniamo $\#HardAugm * D_{n,k}$ modelli.

In questo modo esploriamo tutte le possibili applicazioni delle augmentation in uno schema in cui l'ordine tra le hard e le soft viene sempre mantenuto. Inoltre analizziamo l'effetto a monte delle hard augmentation.

Il file "*augmenter.py*" contiene il codice per applicare le trasformazioni secondo la configurazione corrente.

Baseline

Nel paper (¹) i risultati vengono confrontati con quelli ottenuti da un modello di baseline allenato in modo supervisionato. Per rendere paragonabili i risultati, il modello di baseline ha la stessa architettura del modello contrastive (3 livelli Conv1d - LeakyReLU - BatchNorm1d) e viene allenato nella classificazione delle anomalie utilizzando una Cross Entropy Loss. Il training procede per 300 epoche.

```
SimCLR TS(
  (encoder): Sequential(
    (0): EncoderLayer(
      (encoder_layer): Sequential(
        (0): Conv1d(52, 64, kernel_size=(3,), stride=(2,))
        (1): ReLU()
        (2): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): EncoderLayer(
      (encoder_layer): Sequential(
        (0): Conv1d(64, 128, kernel_size=(3,), stride=(2,))
        (1): ReLU()
        (2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (2): EncoderLayer(
      (encoder_layer): Sequential(
        (0): Conv1d(128, 256, kernel_size=(3,), stride=(1,))
        (1): ReLU()
        (2): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
  (cls_linear): Sequential(
    (0): Linear(in_features=5632, out_features=22, bias=True)
  )
)
```

Fig.4 Strutture della rete.

Purtroppo i risultati che otteniamo da questo training sono molto lontani da quelli attesi. In (¹) si raggiunge un'accuratezza di circa 48% con la sola baseline, mentre noi otteniamo valori prossimi alla classificazione random (tra 0,5% e 0,7%). Come sanity check abbiamo cercato di mandare la rete in overfitting fornendo pochissimi dati di training (al massimo 20 batch da 64 sample) e testando sugli stessi dati. Ci saremmo aspettati un'accuratezza molto elevata, prossima al 100%, invece otteniamo valori che variano tra 30% e 50% in base a differenti configurazioni di learning rate e weight decay. Durante il training la loss scende correttamente con il giusto andamento ma evidentemente la rete non riesce ad estrarre correttamente le features. Abbiamo tentato diverse configurazioni degli iperparametri ma il risultato non cambia, pensiamo che possa esserci un grave problema di fondo ma non sappiamo come muoverci.

AGGIORNAMENTO 18/3:

Prima di effettuare il test, il modello viene sempre impostato in modalità 'evaluation' tramite l'istruzione `model.eval()`. Quest'ultima agisce su vari livelli della rete, in particolare su BatchNorm1d. Abbiamo notato che rimuovendola l'accuratezza cambia drasticamente e finalmente il modello è in grado di andare in overfitting sul dataset di training (facendo lo stesso esperimento citato sopra). Anche l'accuratezza sul dataset di test sale parecchio, attorno al 22%,

ma resta lontana dall'obiettivo del 48%. Quindi la media e la varianza calcolate da BatchNorm durante il training non rispecchiano i dati durante il test.

AGGIORNAMENTO 24/3:

Inizialmente ci siamo dimenticati di inizializzare i pesi della rete. Poi abbiamo aggiunto una funzione di inizializzazione che imposta i pesi dei vari livelli così:

- livelli Conv1d : inizializzazione di kaiming uniforme, inizializzazione costante a 0 per i bias
- livelli BatchNorm1d : inizializzazione costante a 1 per i pesi, a 0 per i bias
- livelli Linear : inizializzazione di kaiming normale, inizializzazione costante a 0 per i bias

Per cercare di raggiungere i risultati del paper abbiamo lanciato una random search su learning rate e weight decay dell'ottimizzatore (sono gli unici due parametri su cui gli autori sostengono di aver fatto tuning) ma i risultati variano tra 10% e 22%.





Adattamento di CSI al modello



SimCLR-TS (¹) non mira al rilevamento delle anomalie e per questo scarta le trasformazioni che deteriorano l'accuratezza del classificatore. Ispirati dal lavoro di (²), pensiamo che le trasformazioni dannose possano avere un effetto benefico se applicate in un contesto di anomaly detection in quanto dovrebbero aiutare il modello a distinguere meglio tra in-distribution e out-distribution.

Il numero di trasformazioni e l'ordine in cui vengono applicate influenza molto il risultato quindi il nostro primo obiettivo è stabilire quale sia la combinazione potenzialmente migliore. A tale scopo il modello viene allenato con tutte le possibili combinazioni di trasformazioni, per ognuna si valuta l'accuratezza ottenuta e si salva quella che ha generato il risultato migliore (come già detto nel paragrafo delle trasformazioni).

Una volta individuata la giusta configurazione è opportuno fare un tuning della rete per trovare i valori corretti per gli iperparametri (learning rate, weight decay, ...).

Il training dell'encoder procede per 400 epoche e costituisce una prima fase di pre-training. Nella fase successiva l'encoder allenato viene "congelato" e si procede con l'allenamento supervisionato del classificatore finale. Questo training procede solamente per 4 epoche.

1. Johannes Pöppelbaum, Gavneet Singh Chadha, Andreas Schwung,
Contrastive learning based self-supervised time-series analysis,
<https://doi.org/10.1016/j.asoc.2021.108397>     

2. Jihoon Tack, Sangwoo Mo, Jongheon Jeong, Jinwoo Shin,
CSI: Novelty Detection via Contrastive Learning on Distributionally Shifted Instances,
<https://doi.org/10.48550/arXiv.2007.08176>  

3. <https://paperswithcode.com/dataset/tep> 