
# Package Python per Algoritmi e Modelli di Nowcasting

Con il termine “nowcasting” facciamo riferimento all’insieme di tecniche finalizzate alla predizione delle condizioni meteorologiche all’istante di tempo attuale o comunque nell’immediato futuro (generalmente entro un massimo 5/10 minuti) circoscritte a una particolare zona di interesse. Questo concetto si affianca spesso a quello più noto di “forecasting”, che riguarda tuttavia previsioni di accuratezza inferiore, ma relative ad una finestra temporale più ampia, arrivando anche a stime di una settimana in avanti.

Sebbene ad un primo sguardo nowcasting e forecasting possano apparire molto simili tra loro, le finalità di questi strumenti presentano delle sostanziali differenze. Nel caso del nowcasting, infatti, più che fare previsioni su ciò che accadrà a livello meteorologico, il fine è quello di raccogliere precise statistiche relative a uno o più fenomeni di interesse in una zona circoscritta.

Il nowcasting è quindi un importante strumento statistico può fungere da supporto nelle analisi sul clima finalizzate a descrivere andamento, intensità e variabilità di fenomeni meteorologici, osservati a diversa scala temporale. Tuttavia, automatizzare il processo di nowcasting risulta essere un’operazione piuttosto complessa che, se affrontata tramite metodologia standard, richiede l’acquisto di strumentazione sofisticata e dai costi piuttosto elevati; per questo motivo, il nowcasting è spesso prerogativa delle stazioni meteo più attrezzate. L’idea è quindi quella di ridurre il più possibile il numero e la complessità dei sensori necessari al raccoglimento dei dati di nowcasting per rendere questa pratica maggiormente accessibile.

Il problema del nowcasting può essere efficacemente affrontato affidandosi a tecniche di Computer Vision e Deep Learning, limitando quindi la richiesta di sensori a semplici telecamere RGB. Questo tipo di approccio risulta particolarmente conveniente in termini di semplicità d’uso e di risorse impiegate, ma presenta una serie di problematiche di progettazione e implementazione che non sono facilmente affrontabili dai non esperti del settore. Per questo motivo nasce l’idea di `PyNowCast`, un package Python che permette di gestire algoritmi e modelli di nowcasting basati su DeepLearning in modo semplice e veloce, occupandosi in modo trasparente di tutti gli aspetti più complessi e macchinosi che caratterizzano questo tipo di tecnologia. Tramite PyCast, quindi, il nowcasting tramite Deep Learning sarà alla portata di tutti.

# 1. Quick Start

Si propone di seguito una guida rapida dall'utilizzo di PyNowCast con i principali passaggi da seguire per arrivare alla classificazione di una data immagine di input.

1. Clonare il repository PyNowCast ed installare i requisiti:
    - `git clone https://github.com/FabioLanzi/PyNowCast.git`
    - `cd PyNowCast; pip install -r requirements.txt`
2. Organizzare il dataset come descritto alla Sezione 2; supponiamo ad esempio di porre tale dataset nella directory `/nas/dataset/nowcast_ds`
3. Allenare il feature extractor come descritto nella Sezione 3 tramite l'apposito script `train_extractor.py`
    - esempio: `python train_extractor.py --exp_name='fx_example' --ds_root_path='/nas/dataset/nowcast_ds'`
4. Allenare il classificatore come descritto nella Sezione 4 tramite l'apposito script `train_classifier.py`
    - esempio: `python train_classifier.py --exp_name='nc_example' --ds_root_path='/nas/dataset/nowcast_ds' --pync_file_path='/nas/pync/example.pync'`
5. Utilizzare il classificatore precedentemente allenato su un'immagine a piacere utilizzando l'apposito comando `classify` dello script `pync.py`
    - esempio: `python pync.py --pync_file_path=/nas/pync/example.pync`

Nelle sezioni che seguono, saranno illustrati nel dettaglio tutti i passaggi qui accennati, accompagnati da opportune motivazioni relative alle scelte effettuate.

# 2. Dataset

Come prima cosa è necessario organizzare i dati sui quali si vuole allenare il modello di nowcasting secondo il semplice schema mostrato in Figura 1. Si avrà pertanto una directory principale, indicata con `<dataset_name>`, con un nome a piacere, che dovrà necessariamente contenere due sotto-directory denominate rispettivamente `train` e `test`, che a loro volta devono contenere le varie sotto-directory contenenti le immagini suddivise per classi.

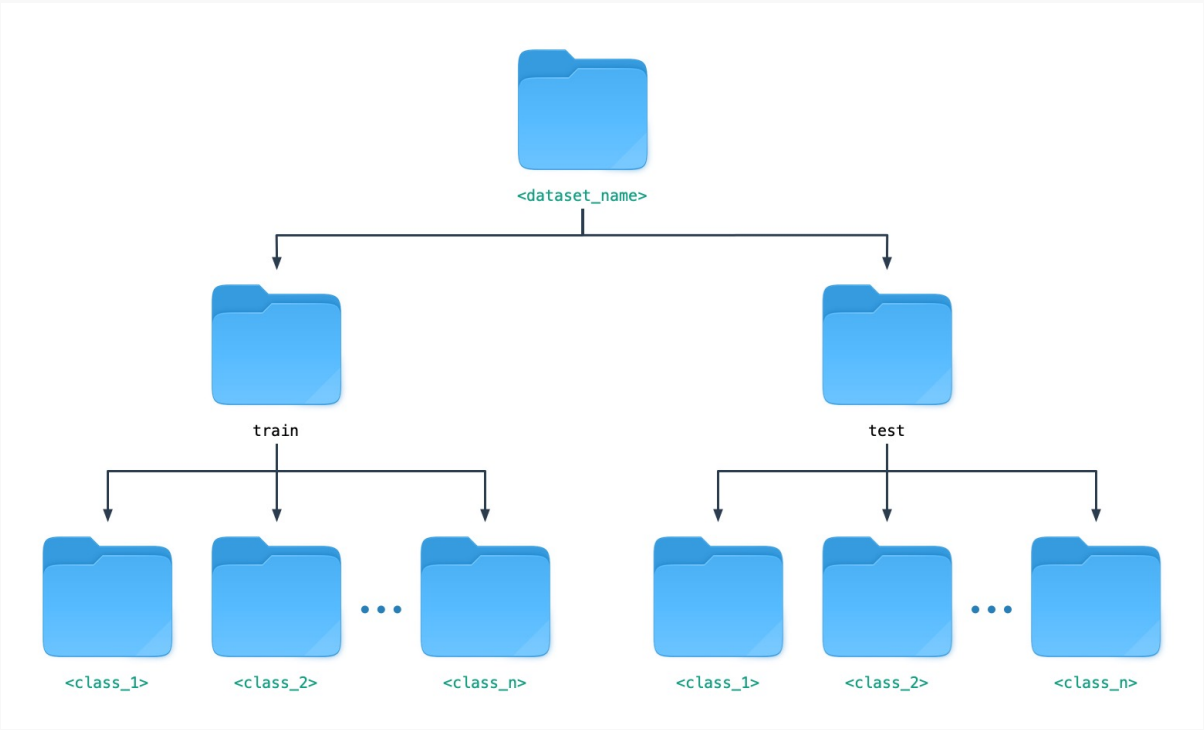


**Figura 1.** Struttura della directory contenente il dataset. I nomi indicati in verde in figura possono essere scelti a piacere.

*NOTA:* È fondamentale che tutte le immagini utilizzate per l'allenamento e la verifica del modello di nowcasting abbiano la stessa dimensione. Il package PyNowCast è stato infatti pensato per problemi di nowcasting a camera fissa, quindi si presuppone che tutte le immagini che compongono il training set e il test set provengano dalla stessa camera e presentino di conseguenza le medesime dimensioni.

## 2.1. Directory di Train e Test

La directory `train`, come il nome suggerisce, è quella preposta a ospitare le immagini di training, ovvero quelle sulle quali sarà allenato il modello di nowcasting; allo stesso modo, la directory `test` conterrà le immagini utilizzate per valutare le prestazioni del suddetto modello. Si noti che, per ottenere una valutazione corretta delle prestazioni del modello, gli insiemi composti dalle immagini di training e dalle immagini di test dovrebbero essere completamente disgiunti, quindi privi di immagini comuni.

Le sotto-directory `train` e `test` devono contenere a loro volta `n` sotto-directory, dove `n` (con `n` maggiore o uguale a 2) è il numero di classi scelte per lo specifico problema di nowcasting che si vuole approcciate con il presente framework. La directory di ogni classe deve contenere esclusivamente le immagini relative a quella specifica classe, affiancate al più da un file `sensors.json` contenente le informazioni relative ad eventuali dati aggiuntivi provenienti da appositi sensori (esempio: sensori di umidità, temperatura, pressione, ...).

Per allenare correttamente il modello di nowcasting, si consiglia di avere un training set bilanciato, quindi con un quantitativo di immagini simile per ogni classe; si consiglia inoltre di avere un numero di immagini maggiore di 1000 per ogni classe.

Per velocizzare il processo di inizializzazione del dataset, è possibile creare all'interno delle directory `train` e `test` un file di cache in formato JSON che prenderà il nome di `cache.json`. La creazione di tale file avviene in automatico quando si chiama il costruttore della classe `NowCastDS` con il parametro `create_cache=True` (*NOTA*: il valore di default è `False`).

```
training_set = NowCastDS(ds_root_path='/your/ds/root/path', mode='train',
create_cache=True)
test_set = NowCastDS(ds_root_path='/your/ds/root/path', mode='test',
create_cache=True)
```

## 2.2. Struttura dei File `sensor.json`

Considerando una classe con `K` immagini ed `m` valori provenienti dai sensori associati a ciascuna di esse, la struttura del relativo file opzionale `sensors.json` sarà la seguente:

```
{
    "img1_name": [x1_1, x1_2, ..., x1_m],
    "img2_name": [x2_1, x2_2, ..., x2_m],
    ...
    "imgK_name": [xK_1, xK_2, ..., xK_m]
}
```

Si tratta dunque di un file JSON in cui le chiavi sono i percorsi relativi alla directory contenente il file `sensors.json` stesso e i valori sono liste contenente gli `m` dati letti dai sensori per quella specifica immagine.

Si noti che, se si sceglie di inserire il file `sensor.json` all'interno di una directory relativa ad una classe, anche tutte le altre classi devono contenerlo. Qualora per alcune immagini non fossero disponibili uno o più valori relativi ad un sensore, sarà sufficiente inserire al loro posto il valore `null`. Se ad esempio per l'immagine `img1_name` non si disponesse del valore 2, all'interno del JSON si avrà:

- `"img1_name": [x1_1, null, ..., x1_m]`

## 2.3. Verifica della Correttezza della Struttura del Dataset

È possibile verificare la correttezza della struttura del proprio dataset utilizzando lo script `check_dataset_structure.py` tramite il seguente comando, in cui si indica con `<dataset_path>` il percorso assoluto alla directory principale del dataset:

- `python chech_dataset_structure.py <dataset_path>`

Lo script verificherà la presenza di errori strutturali e li comunicherà all'untento con un apposito messaggio auto-esplicativo. Saranno inoltre forniti avvertimenti su eventuali aspetti che non si ritengono ottimali per iniziare la procedura di traning; ad esempio potrebbe essere segnalata la presenza di un numero di immagini ritenuto insufficiente per una o più classi.

Gli errori saranno evidenziati con un pallino rosso e la dicitura "ERROR" e andranno necessariamente risolti prima di intraprendere la procedura di allenamento del modello di nowcasting.

Gli avvertimento saranno evidenziati con un pallino giallo e la dicitura "WARNING"; in questo casto non sarà necessario (sebbene caldamente consigliato) risolvere la problematica indicata prima di procedere con l'allenamento del modello.

## 2.4. Esempio

In Figura 1.2. si propone un esempio di struttura della directory `train` nel caso di un problema di nowcasting a due classi, in cui, partendo dall'immagine RGB e dai dati di un sensore di temperatura si vuole verificare la presenza o l'assenza di nebbia nell'immagine in ingresso. Le sotto-directory `fog` e `no_fog` contengono rispettivamente immagini con nebbia e immagini in cui la nebbia è assente. Le immagini mostrate in figura sono state acquisite presso l'Osservatorio di Modena.

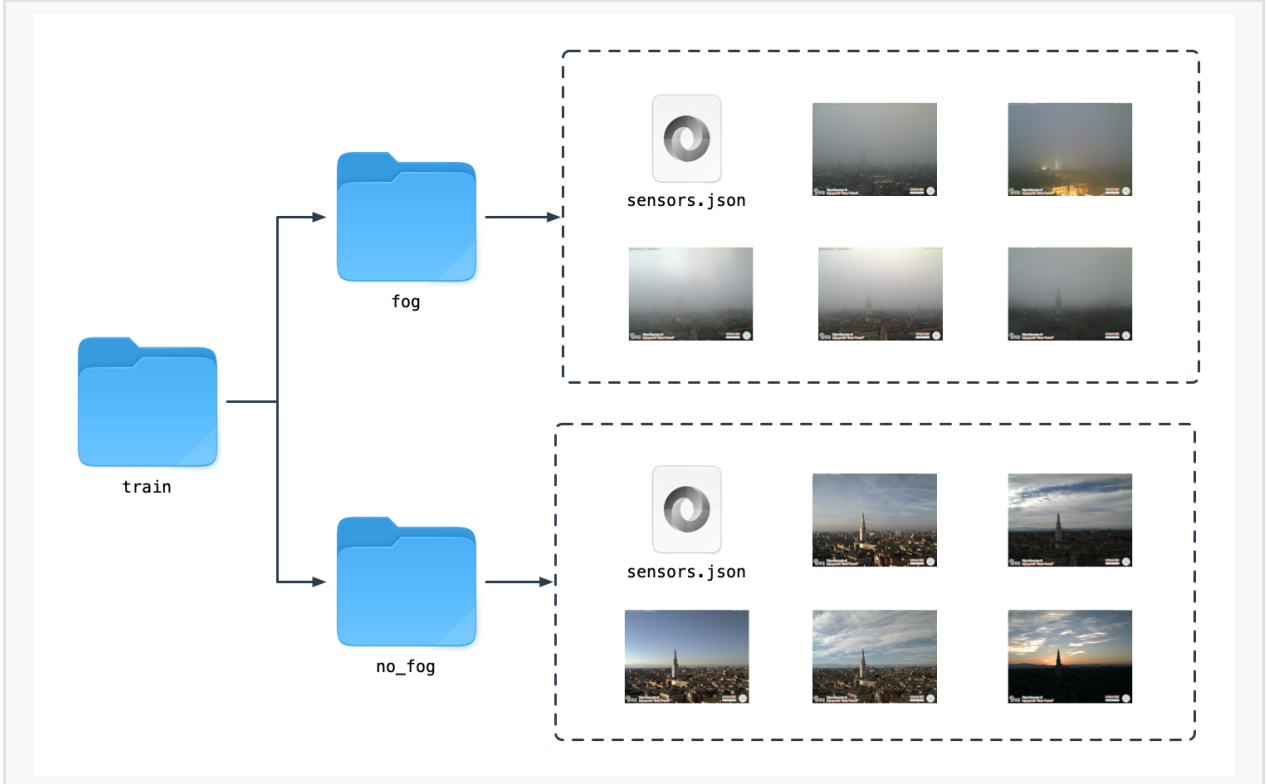


**Figura 2.** Esempio di struttura della directory `train` nel caso di un problema di nowcasting a due classi (classe `fog` e classe `no_fog`).

Un esempio completo che mostra la struttura di un dataset valido, seppur contenente un numero esiguo di immagini, è contenuto all'interno di questo stesso repository:

- `PyNowCast/dataset/example_ds`

*NOTA*: il dataset `example_ds` ha solo uno scopo esemplificativo e non può essere utilizzato per allenare un modello di nowcasting a causa del numero ridotto di immagini presenti.

## 3. Feature Extractor

Il feature extractor è un componente essenziale della maggior parte dei modelli di classificazione basati su reti neurali; il suo compito è, come suggerisce il nome stesso, quello di estrarre una serie di caratteristiche che "riassumano" i tratti salienti dell'oggetto passato in ingresso, che nel nostro caso è un'immagine RGB proveniente da una camera fissa.

Nell'ambito della Computer Vision, esistono una serie di feature extractor ([1], [2], [3], [4]) standard che vengono utilizzati per un vasto numero di teask, in quanto risultano molto flessibili e in grado di fornire feature di alto livello che possono venire incontro alle esigenze di problemi anche molto diversi tra loro.

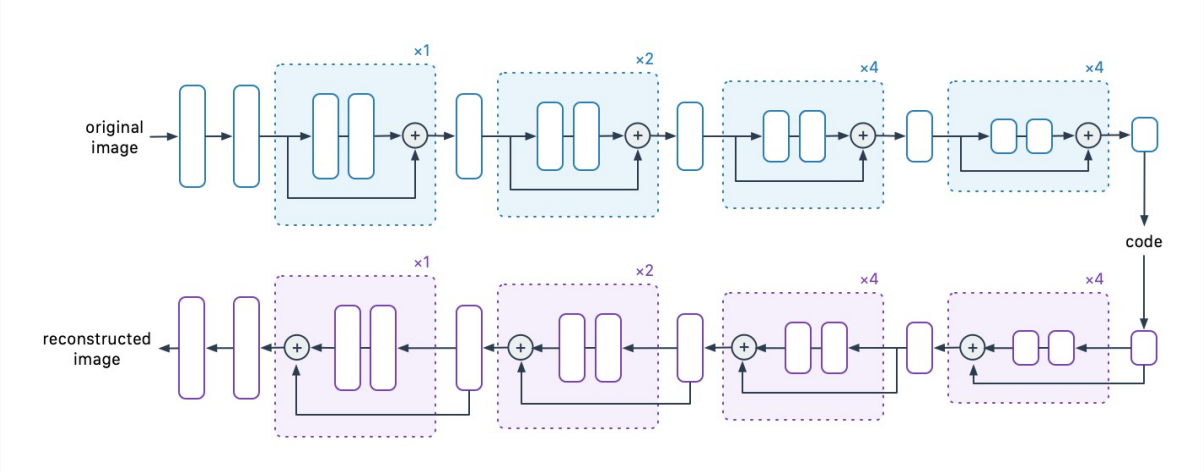


**Figura 2.** Schema a blocchi dell'autoencoder utilizzato durante la fase di allenamento del feature extractor; il feature extractor vero e proprio è rappresentato dalla sola parte di encoding (ramo "blu" in figura).

Nel nostro caso specifico, tuttavia, risulta più opportuno affidarsi ad un feature extractor maggiormente mirato e incentrato sulle nostre esigenze particolari. Il problema che *PyNowCast* va ad affrontare infatti è molto vincolato e i vincoli che lo contraddistinguono, se ben sfruttati, possono semplificarlo enormemente. In questo senso, per il nostro package abbiamo deciso di orientarci verso un feature extractor personalizzato basato su un autoencoder, che sia in grado di essere allenato efficacemente in modo non supervisionato.

## 3.1. Autoencoder

Per adottare questa soluzione ci siamo basati su una semplice osservazione: poiché ci troviamo a trattare immagini provenienti da una camera fissa che riprende una certa porzione di paesaggio, gli elementi che variano tra un'immagine e l'altra sono essenzilamente la condizione metereologica e le condizioni di illuminazione. Fortunatamente ciò che varia sono esattamente le feature che servono ad un classificatore che si occupa di nowcasting.

In tal senso, l'uso di un autoencoder composto da un encoder e un decoder speculari risulta particolarmente appropriato. Allenando un autoencoder di questo tipo a ricostruire semplicmente le immagini di input all'uscita dell'encoder si avrà un "codice" che rappresneta per l'appunto un riassunto delle immagini di input. Trovando la giusta dimensione di tale codice, l'encoder sarà forzato a rimuovere tutte le informazioni ridondati, che nel nostro caso sono appunto le caratteristiche che non variano tra un'immagine e l'altra; al contempo dovrà preservare gli elementi mutevoli delle medesime (meteo e condizioni di illuminazione).

L'autoencoder utilizzato è mostrato in Figura 3.

## 3.2. Procedura di Allenamento

Per avviare la procedura di allenamento del feature extractor è possibile utilizzare lo script `train_extractor.py` con le seguenti opzioni:

- `--exp_name`: etichetta assegnata alla corrente procedura di allenamento del feature extractor; per differenziare le etichette assegnate al feature extractor da quelle assegnate al modelo utilizzato per la classificazione, si suggerisce di utilizzare il prefizzo `fx_`.
- `--ds_root_path`: percorso della directory principale contenete il proprio dataset.
- `--device`: tramite questa opzione è possibile selezionare il device su cui sarà effettuata la procedura di allenamento del feature extractor; i possibili valori sono i seguenti: `'cuda'` o `'cuda:<numero specifica gpu>'` per un allenamento su GPU (consigliato) o `'cpu'` per un allenamento su CPU (sconsigliato). Se non specificato, il valore di default è `'cuda'`

Si propone di seguito un esempio di chiamata:

- `python train_extractor.py --exp_name='fx_try1' --ds_root_path='/nas/dataset/nowcast_ds'`

Per gli utenti più esperti, è possibile modificare il file `conf.py` per personalizzare i parametri del training; salvo casi molto particolari, tuttavia, si suggerisce di utilizzare i parametri di default. Per completezza, si riporta la porzione del file di configurazione relativa all'allenamento del feature extractor:

```
# feature extractor settings
FX_LR = 0.0001  # learning rate used to trane the feature extractor
FX_N_WORKERS = 4  # worker(s) number of the dataloader
FX_BATCH_SIZE = 8  # batch size used to trane the feature extractor
FX_MAX_EPOCHS = 256  # maximum training duration (# epochs)
FX_PATIENCE = 16 # stop training if no improvement is seen for a
'FX_PATIENCE' number of epochs
```

## 4. Classificatore

Il cuore di PyNowCast è il modello utilizzato per la classificazione che si compone di due elementi fondamentali: il feature extractor di cui si è discusso nella Sezione 3 e una rete completamente connessa a 3 livelli che svolge il ruolo di classificatore vero e proprio. La struttura semplificata dell'intero modello è mostrata in Figura 4 (presupponendo un problema di classificazione a 3 classi).

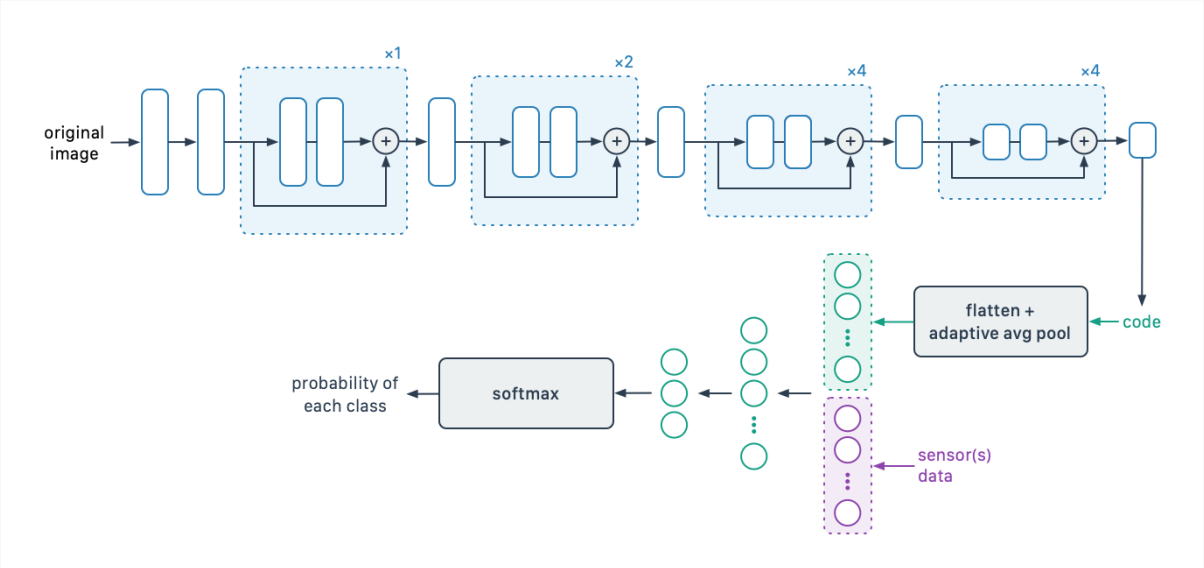


**Figura 4.** Struttura semplificata del modello di classificazione (feature extractor + rete completamente connessa a 3 livelli). Nel caso in figura, si considera un problema di classificazione a 3 classi.

Data un'immagine di input il feature extractor ne estrae una rappresentazione compatta (indicata con il termine "code" in Figura 4). Tale rappresentazione viene opportunamente ridimensionata e "srotolata" ottenendo un array di valori che rappresenta l'ingresso delle rate completamente connessa preposta alla classificazione. Se disponibili, anche i dati dei sensori relativi all'immagine di input sono passati in ingresso alla rete completamente connessa, affiancandosi quindi alle feature visuali. L'output finale del modello è rappresentato da $N$ valori compresi tra 0 e 1 che rappresentano le probabilità associate a ciascuna delle $N$ classi del problema specifico che si sta affrontando.

### 4.1. Procedura di Allenamento

Per avviare la procedura di allenamento del feature extractor è possibile utilizzare lo script `train_classifier.py` con le seguenti opzioni:

- `--exp_name`: etichetta assegnata alla corrente procedura di allenamento del classificatore; per differenziare le etichette assegnate al classificatore da quelle assegnate al feature extractor, si suggerisce di utilizzare il prefizzo `nc_`.
- NOTA: se è stato precedentemente allenato un feature extractor con un dato `exp_name`, è possibile utilizzare la medesima etichetta anche per il classificatore (a meno del prefisso) per caricare automaticamente i pesi del feature extractor velocizzando notevolmente la procedura di training del classificatore.
- `--ds_root_path`: percorso della directory principale contenete il proprio dataset.
- `--pync_file_path`: percorso del file `.pync` contenente i risultati della procedura di allenamento del classificatore; se non specificato, tale file sarà salvato nell'attuale working directory e il suo nome sarà quello specificato con l'opzione `--exp_name` (più l'estensione `.pync`).
- `--device`: tramite questa opzione è possibile selezionare il device su cui sarà effettuata la procedura di allenamento del classificatore; i possibili valori sono i seguenti: `'cuda'` o `'cuda:<numero specifica gpu>'` per un allenamento su GPU (consigliato) o `'cpu'` per un allenamento su CPU (sconsigliato). Se non specificato, il valore di default è `'cuda'`

Si propone di seguito un esempio di chiamata:

- `python train_classifier.py --exp_name='nc_try1' --ds_root_path='/nas/dataset/nowcast_ds'`

Per gli utenti più esperti, è possibile modificare il file `conf.py` per personalizzare i parametri del training; salvo casi molto particolari, tuttavia, si suggerisce di utilizzare i parametri di default. Per completezza, si riporta la porzione del file di configurazione relativa all'allenamento del feature extractor:

```
# nowcasting classifier settings
NC_LR = 0.0001  # learning rate used to trane the nowcasting classifier
NC_N_WORKERS = 4  # worker(s) number of the dataloader
NC_BATCH_SIZE = 8  # batch size used to trane the nowcasting classifier
NC_MAX_EPOCHS = 256  # maximum training duration (# epochs)
NC_PATIENCE = 16 # stop training if no improvement is seen for a
‘NC_PATIENCE’ number of epochs
```

### 4.2. File `.pync`

I risultati della procedura di allenamento vengono salvati in un apposito file con estensione `.pync` il cui path può essere impostato dall'untente tramite l'opzione `--pync_file_path`; tale file contiene i pesi del classificatore e alcune informazioni utili in fase di evaluation del modello, come il nome delle classi considerate e la dimensione dell'array dei valori dei sensori.

Dato un file `.pync` è possibile consultare le informazioni in esso contenute utilizzando il comando `show-info` contenuto nello script `pync.py` nel modo che seghe:

- `python pync.py show-info --pync-file-path=/your/pync/file.pync`

Si mostra di seguito un esempio di output del comando di cui sopra:

```
▶ Showing info of '/your/pync/file.pync'
├── model weights (size): 10536 bytes
├── sensor data len: 3
└── classes:
    ├──[0] sun
    ├──[1] rain
    ├──[2] fog
    └──[3] snow
```

## 5. Classificazione di un'Immagine

Una volta ottenuto un file `.pync` in seguito all'allenamento di un classificatore, applicarlo ad un'immagine di test risulta molto semplice: è infatti sufficiente utilizzare il comando `classify` contenuto in `pync.py` specificando il path dell'immagine da classificare e il path del file `.pync` che si vuole utilizzare. Si propone di seguito un esempio di chiamata del comando.

- `python pync.py classify --img_path=/your/input/image.jpg --pync_file_path=your/pync/file.pync`

In output si ottengono quindi le probabilità associate a ciascuna delle classi contemplate, con un'apposita indicazione della classa a probabilità più alta. Si veda l'esempio seguente.

```
▶ Classifying image '/your/input/image.jpg'
├──[0]──[sun]:  0.10 %
├──[1]──[rain]: 84.11 % ‹●
├──[2]──[fog]: 14.17 %
└──[3]──[snow]:  1.62 %
```