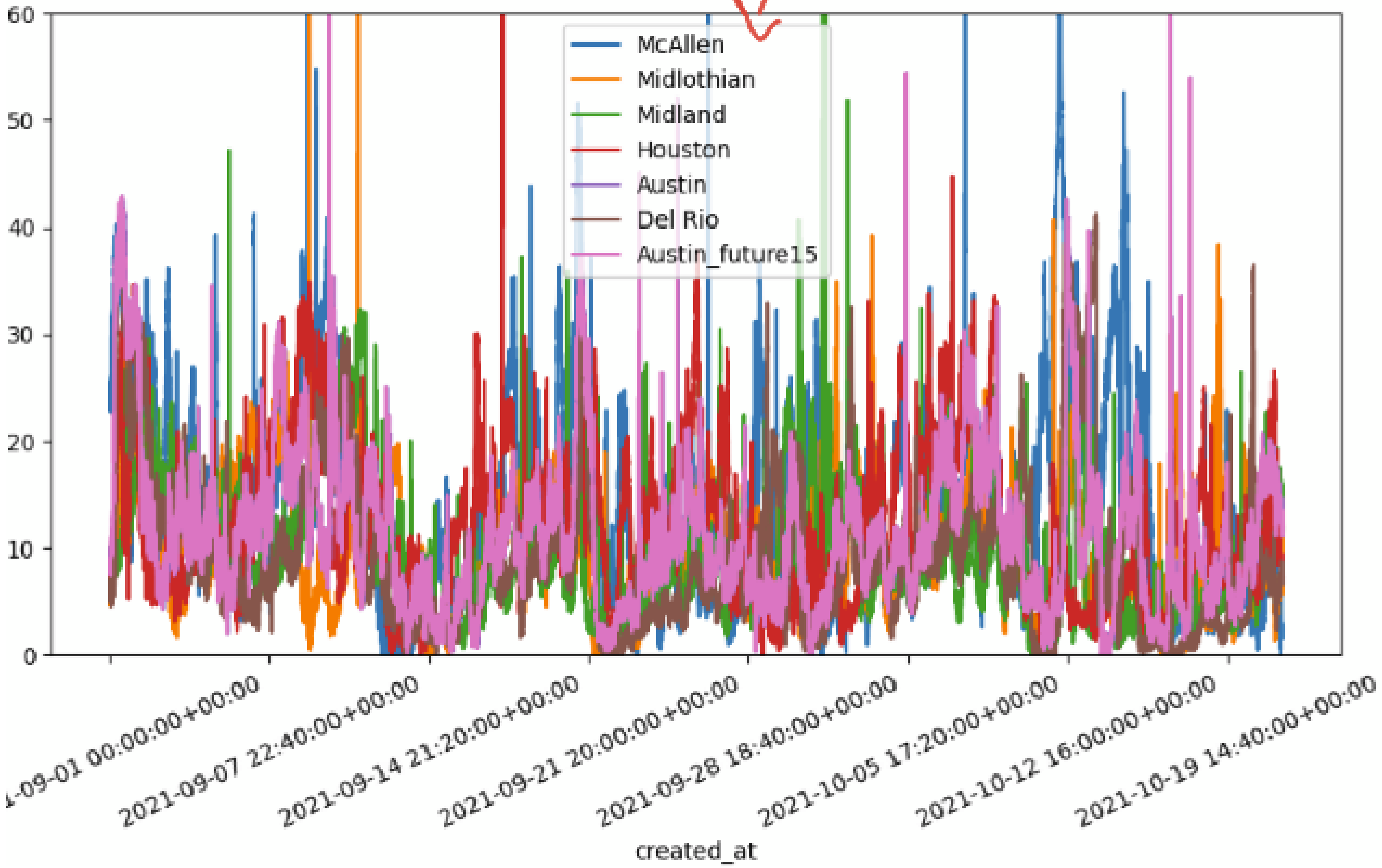
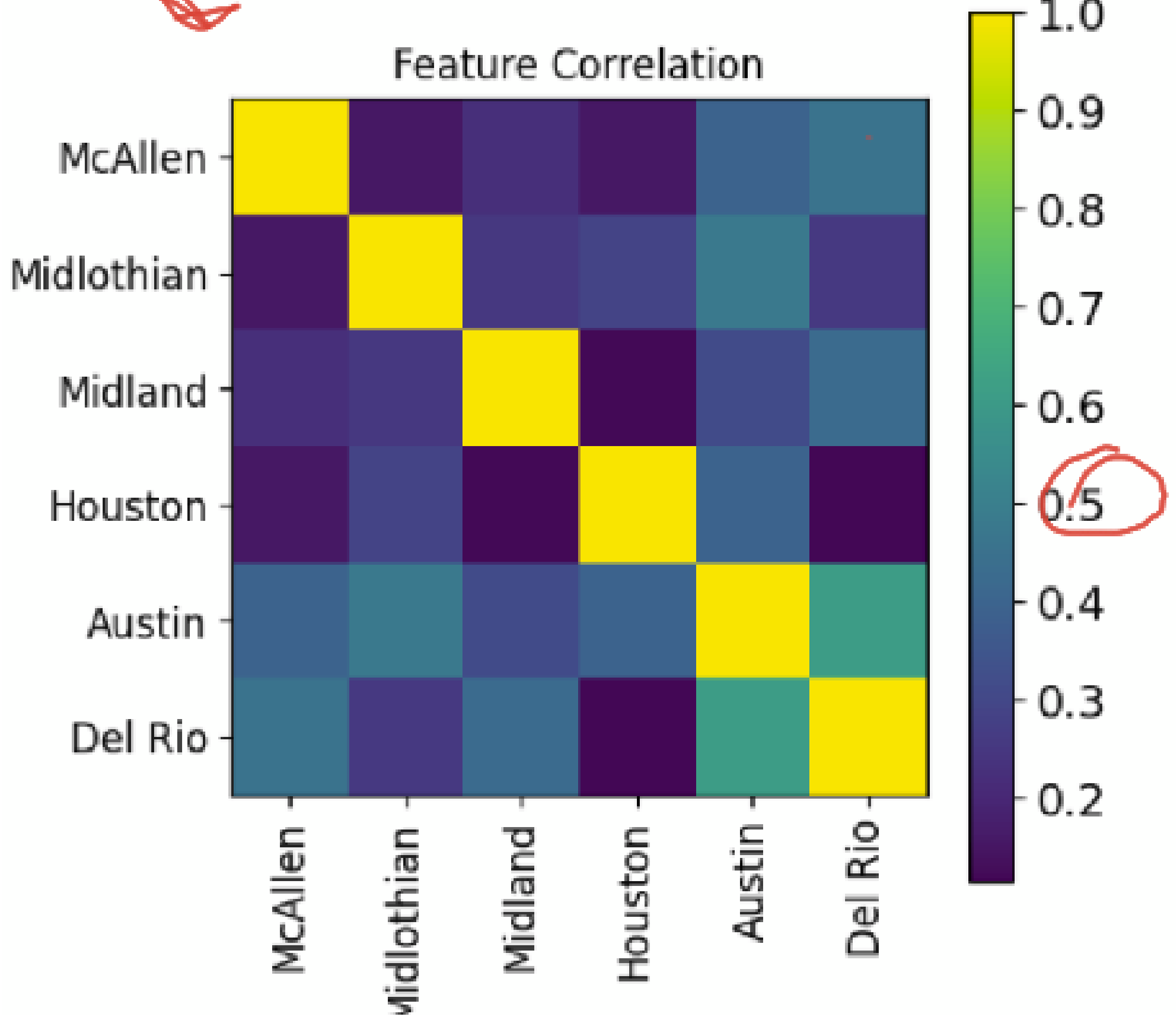
Predizione di serie temporali

1) Feature visualization
2) Preparazione del dataset
3) Divisione Test/Train
4) torch Dataset, Dataloader
5) Model definition
6) Train
7) Test

Visualization

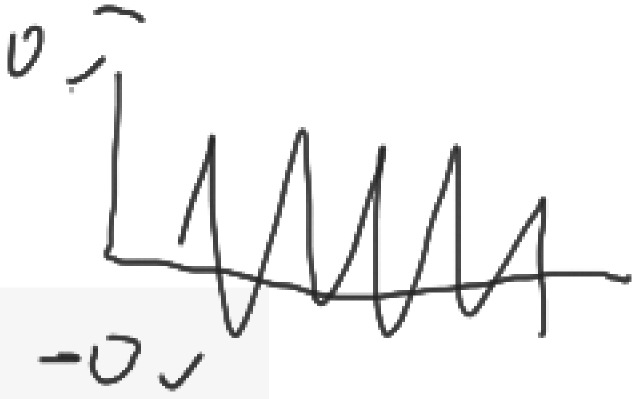


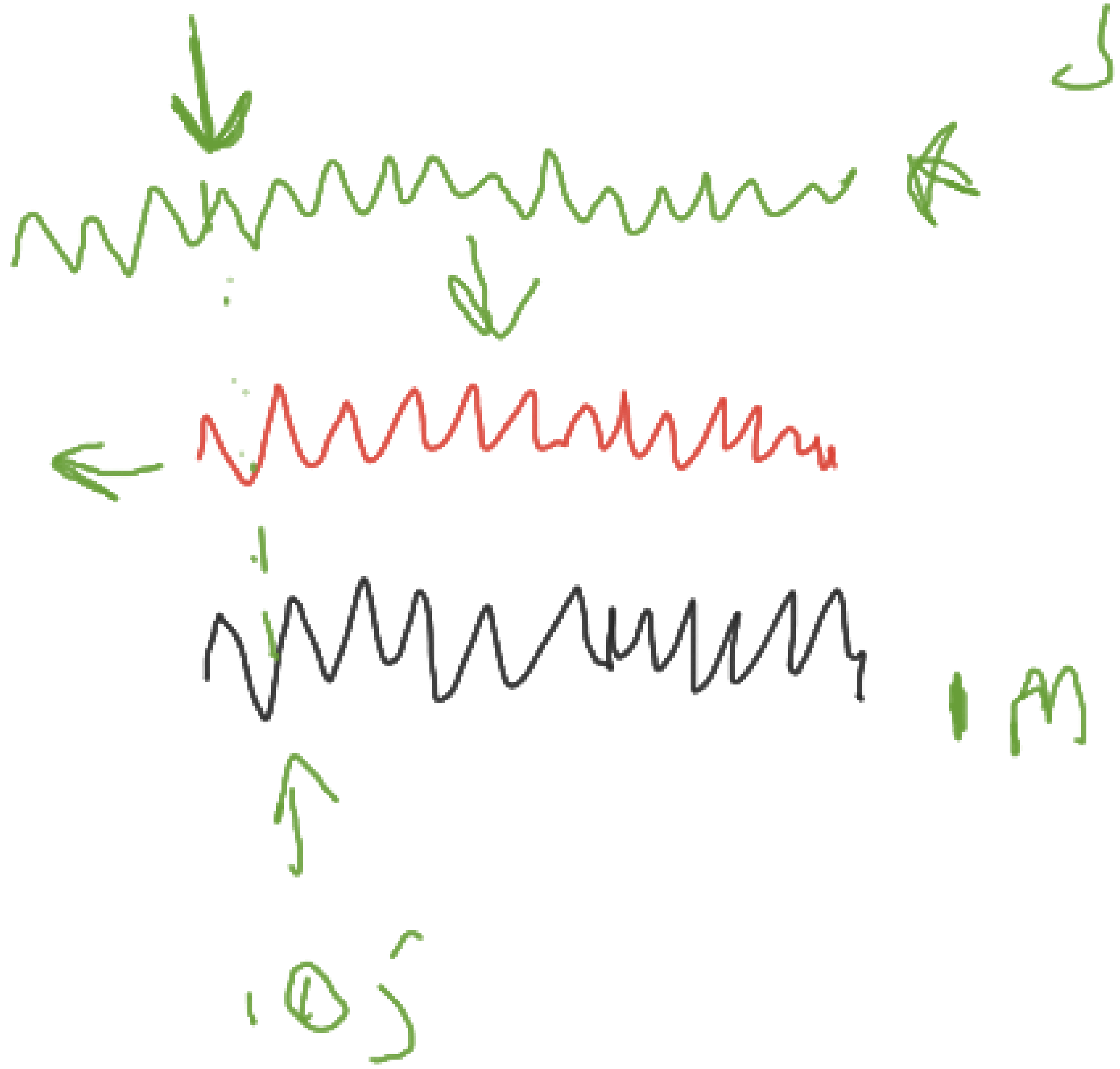


Dataset preparation

```python
# normalizzazione delle feature e del target, di norma migliora la convergenza della RNN e le pre
# finali

# standardizziamo feature e target sottranedo la media e dividendo per la deviazione standard

# salviamo per uso successivo mean e stddev del target
target_mean = df_train[target].mean()
target_stdev = df_train[target].std()

for c in df_train.columns:
  mean = df_train[c].mean() #usiamo solo i dati di training per ricavare media e stdev
  stdev = df_train[c].std()
  df_train[c] = (df_train[c] - mean) / stdev
  df_test[c] = (df_test[c] - mean) / stdev

display(df_train.describe())
display(df_test.describe())
```

```python
# dobbiamo shiftare di 15 step

forecast_step = 15
target = f"{target_sensor}_future{forecast_step}"

df[target] = df[target_sensor].shift(-forecast_step)

# eliminiamo le ultime 15 righe del dataframe pandas che dopo lo shift non avranno valori a causa
# dello shift
df = df.iloc[:-forecast_step]

display(df.head(20))
```



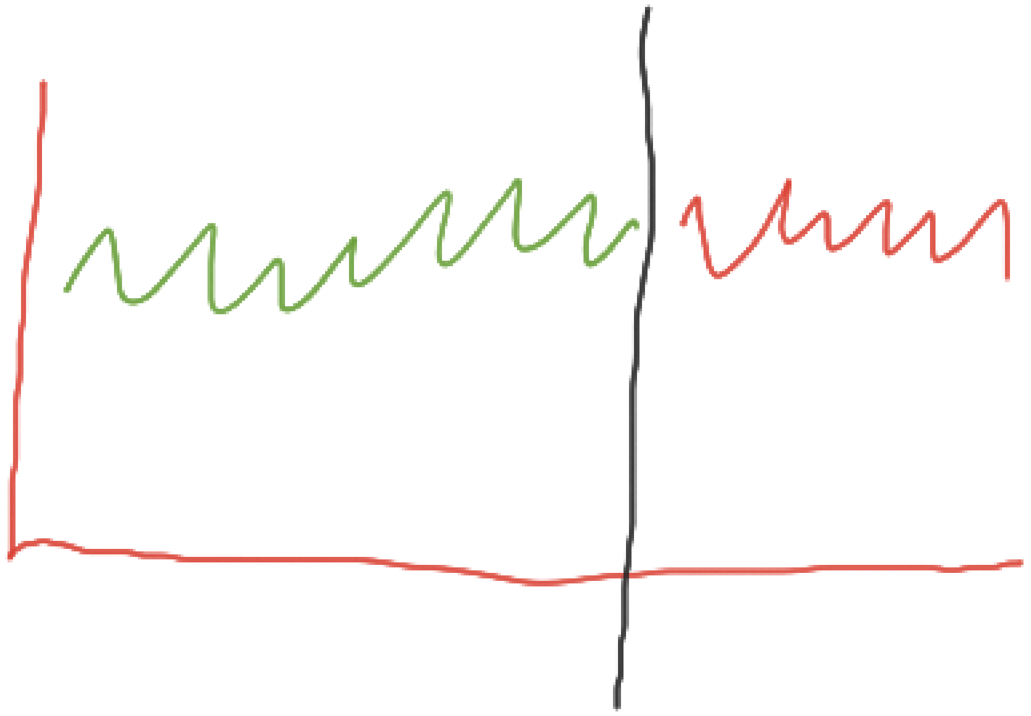


$$L = \underline{\text{MSE}}\,(ss(t)\ \ell n)$$

$$\frac{\partial L}{\partial w}$$

Split

```python
# creazione di un test set per valutare le prestazioni del modello

# il dataset va dal 1.9.2021 al 21.10.2021, prendiamo come test set
# le acquiszioni dal 10.10.2021 al 21.10.2021
test_start = "2021-10-10"

df_train = df.loc[:test_start].copy()
df_test = df.loc[test_start:].copy()

print("Test set fraction:", len(df_test) / len(df))
```



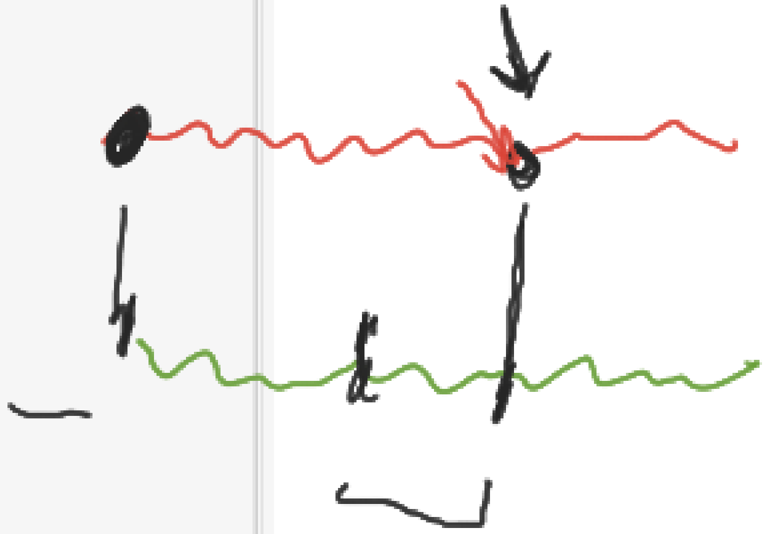
Dataset/Dataloader

DAT = SD(1RL IMT)

```python
class SequenceDataset(Dataset):
    def __init__(self, dataframe, target, features, sequence_length=5):
        self.features = features #features
        self.target = target #target
        self.sequence_length = sequence_length #lunghezza dela sequenza
        self.y = torch.tensor(dataframe[self.target].values).float()
        self.X = torch.tensor(dataframe[self.features].values).float()

    def __len__(self):
        return self.X.shape[0]

    # il metodo __getitem__ implementa la logica del dataset
    def __getitem__(self, i):
        if i >= self.sequence_length - 1:
            i_start = i - self.sequence_length + 1
            x = self.X[i_start:(i + 1), :]
        else:
            padding = self.X[0].repeat(self.sequence_length - i - 1, 1)
            x = self.X[0:(i + 1), :]
            x = torch.cat((padding, x), 0)

        outy = self.y[i]
        return x, outy
```

len (DAT)

DAT[i]



0 0 0 0 0 i

LSTM

CLASS torch.nn.LSTM($*args$, $**kwargs$) [SOURCE]

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

Parameters:

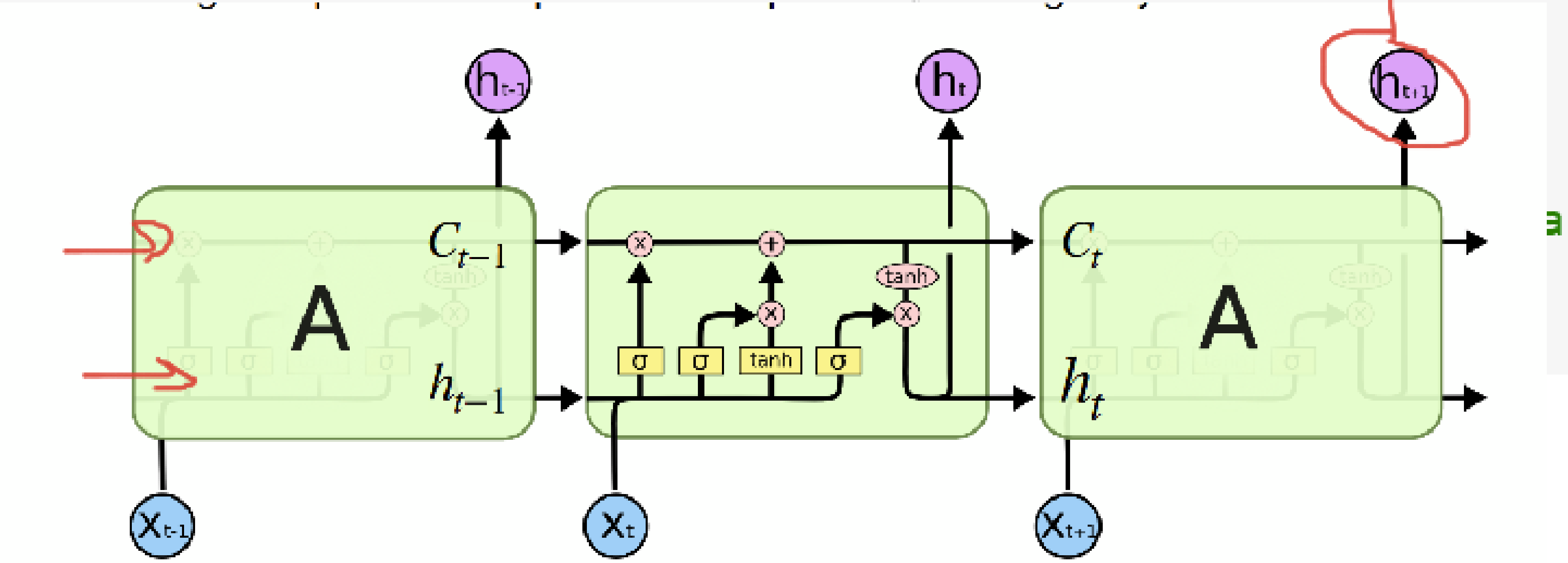
- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the initial cell state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.

Model

```python
class myLSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super().__init__()
        self.input_dim = input_dim  # this is the number of features
        self.hidden_dim = hidden_dim
        self.num_layers = 1 #usiamo una LSTM con un solo layer (non una stack LSTM)

        self.lstm = nn.LSTM(
            input_size=input_dim,
            hidden_size=hidden_dim,
            batch_first=True, # necessario se l'input ha shape (batch, seq. len., features) altrimenti (seq.len., batch, features)
            num_layers=self.num_layers
        )

        # uscita un layer denso con 1 neurone di uscita (il valore di PM2.5 di Austin)
        self.linear = nn.Linear(in_features=self.hidden_dim, out_features=1)
```

```python
def forward(self, x):
    batch_size = x.shape[0]
    # valore iniziale per hidden state e cell state (inzializziamo a zero)
    # NOTA: internamente per h e c pytorch usa la shape (numero di layers, batch size, hidden units)
    h0 = torch.zeros(self.num_layers, batch_size, self.hidden_dim).requires_grad_()
    c0 = torch.zeros(self.num_layers, batch_size, self.hidden_dim).requires_grad_()

    # ci interessa solo l'hidden state, quindi non consideriamo il cell state e l'output
    on, (hn, cn) = self.lstm(x, (h0, c0))

    # reshape the output on come (batch, seq, hidden)
    #on = on[:, -1, :]

    #print(on.shape)
    #print(hn.shape)

    out = self.linear(hn[0]).flatten()  #                                  a 1
    #out = self.linear(on)
    return out
```



```python
hist_vloss = []

# loop sulle epoche
for epoch in range(epochs):
    t0 = time.time()

    # training step (in cui aggiorniamo i pesi della rete neurale)
    model.train()
    train_loss = 0
    counter = 0

    for xb, yb in train_loader:
        counter += 1

        pred = model(xb) #predizione del modello

        # calcolo loss
        loss = loss_func(pred.squeeze(), yb)

        # aggiorno la loss totale
        train_loss += loss.item()

        # backpropagation
        opt.zero_grad() #resetta i gradienti prima di eseguire la backpropagation
        loss.backward() #calcola i gradeinti della loss
        opt.step() #aggiorna i pesi
```

sh: (1, 8, 1)

(8)