Time series analysis: Neural Networks

ΕΠΛ 428: IOT PROGRAMMING

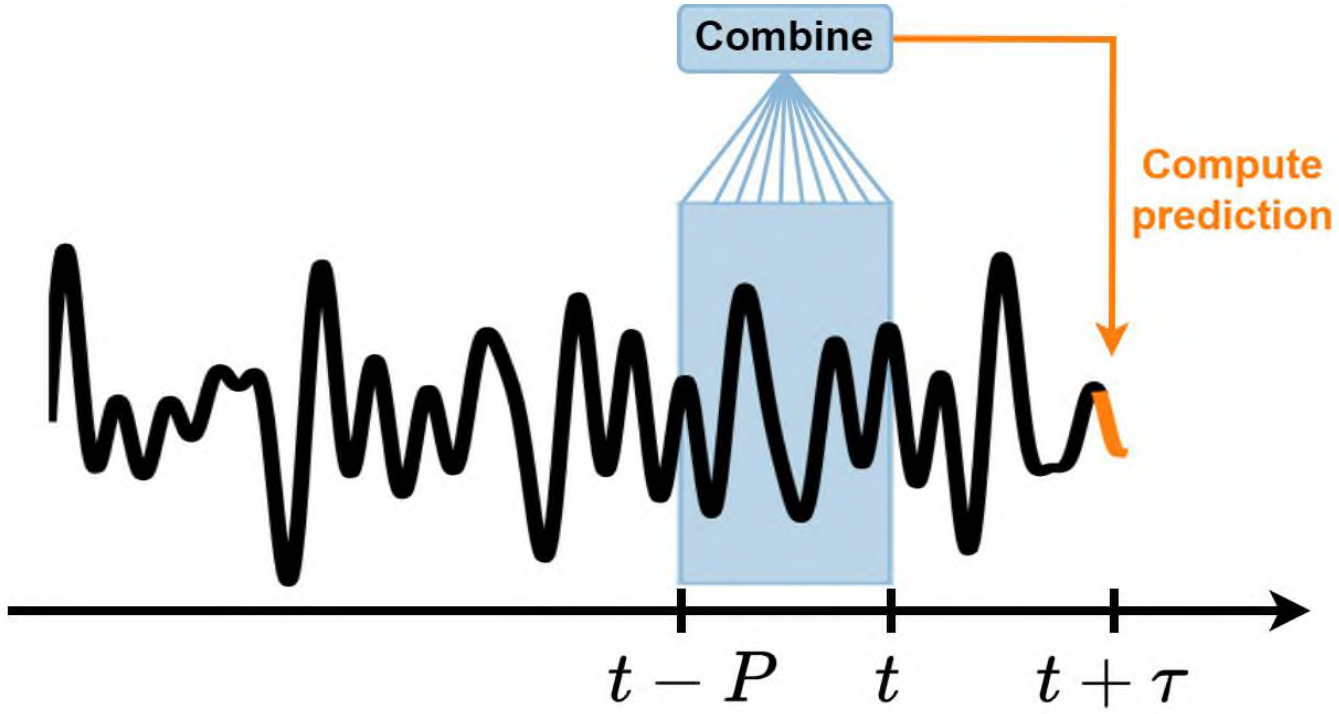
Dr. Panayiotis Kolios
Assistant Professor, Dept. Computer Science,
KIOS CoE for Intelligent Systems and Networks
Office: FST 01, 116
Telephone: +357 22893450 / 22892695
Web: https://www.kios.ucy.ac.cy/pkolios/

Πανεπιστήμιο Κύπρου

- Windowed approaches and nonlinear models for time series forecasting
- Neural networks and the Multi-Layer Perceptron
- A brief overview of Recurrent Neural Networks
- ESN, a randomized RNN from the family of Reservoir Computing
- An introduction to dimensionality reduction with Principal Component Analysis
- Examples of forecasting with MLP and ESN on real-world electricity data

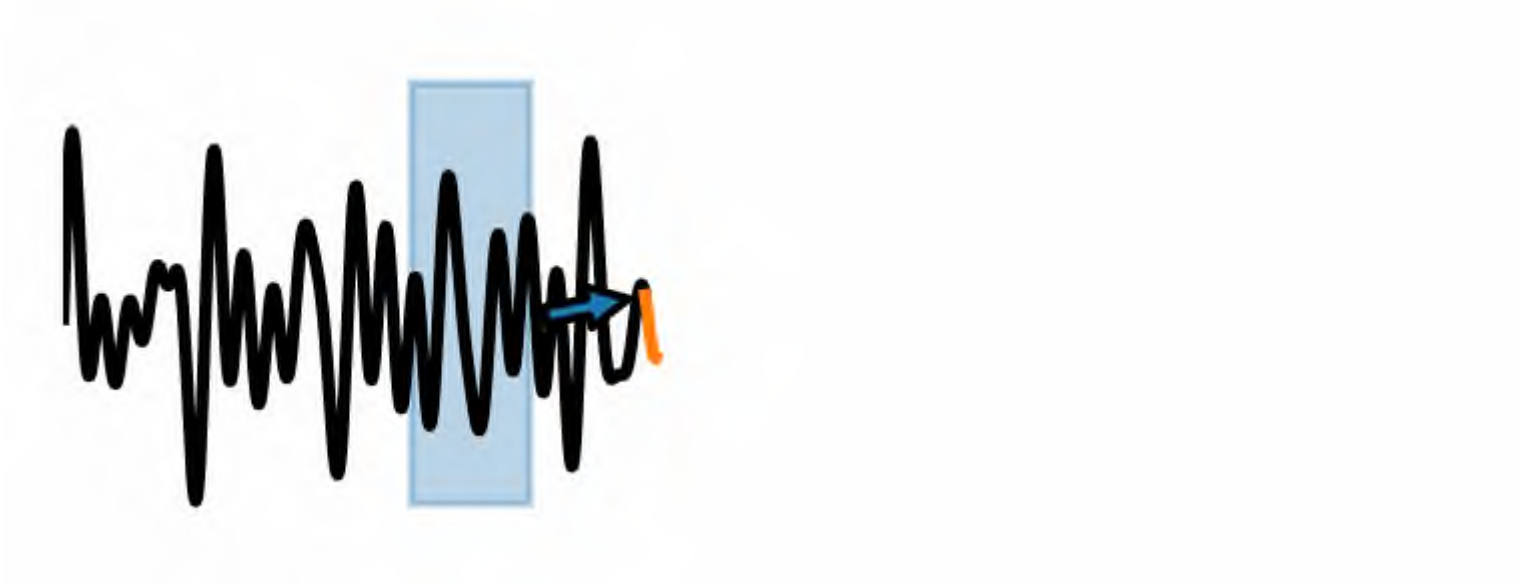
```
# Imports
import numpy as np
import matplotlib.pyplot as plt
import time
import plotly.graph_objects as go
import statsmodels.api as sm
from sklearn.datasets import make_circles
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.datasets import make_blobs
from sklearn.decomposition import PCA
from sklearn.linear_model import Ridge
from sklearn.ensemble import HistGradientBoostingRegressor
from reservoir_computing.reservoir import Reservoir
from reservoir_computing.utils import make_forecasting_dataset
from reservoir_computing.datasets import PredLoader
```

- Windowed methods consider a fixed window of size P
- Data of time series within the window $x(t-P), \ldots, x(t)$ used to compute the prediction $x(t+\tau)$,
- Different algorithms combine the elements of the window in different ways to make predictions

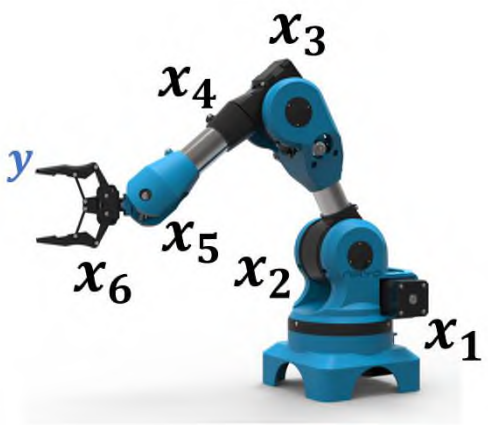


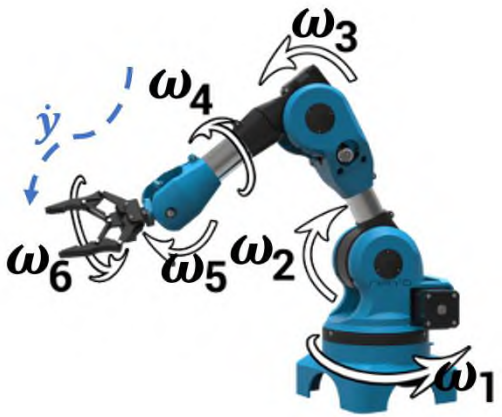
- An example of windowed approach is the linear AR model of order P

$$\hat{y}(t+1) = \beta_0 + \beta_1 \mathrm{x}(\mathrm{t}-1) + \beta_2 x(t-2) + \cdots + \beta_P x(t-P) + \varepsilon_t$$



- Linear models are widely used for their simplicity and interpretiveness
- However, linearity assumption between input and output often fails in practice (cannot capture nonlinearities and interactions effectively)
- Consider the following examples:
  - Compute the position y of a robotic arm given the positions $x_1, x_2, \ldots$ of the joints
  - Compute a trajectory $\hat{y}$ given the angular velocities of the joints





- This is a classic example where linear models fail
- Electricity demand is often governed by complex and nonlinear interactions:
- Temperature and Demand
    - Relationship between temperature and electricity demand is typically nonlinear and can exhibit a U-shaped curve
    - Demand is low at moderate temperatures but increases sharply at high or low temperatures due to heating and cooling needs
- Time and Demand
    - Demand patterns vary significantly throughout:
        - the day (peak hours in the morning and evening)
        - week (workdays vs. weekends)
        - year (summer vs. winter)

- Nonlinear models, include **neural networks**, **SVM**, tree-based methods, etc

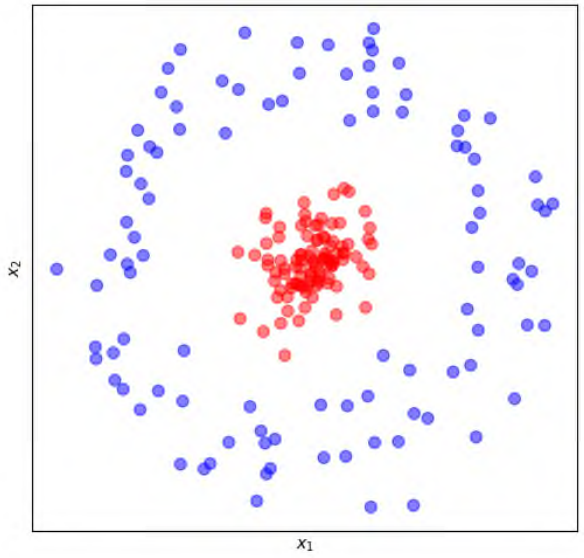
- Model the nonlinear relationships between demand and factors like temperature, capturing the U-shaped curve accurately

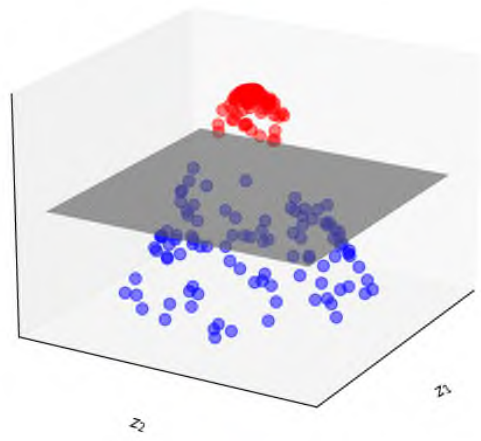
- Take into account the interactions between different variables, such as the combined effect of time, holidays, and temperature on demand

- Adapt to various patterns and changes in trends over time, making them more robust in dynamic environments

- Neural Nets are universal approximators, i.e., they can learn to approximate any function
- Basic idea:
  - Map the data into a high dimensional space (bless of dimensionality)
  - Apply a nonlinear transformation
  - Data become linearly separable

```
plot_3D()
```

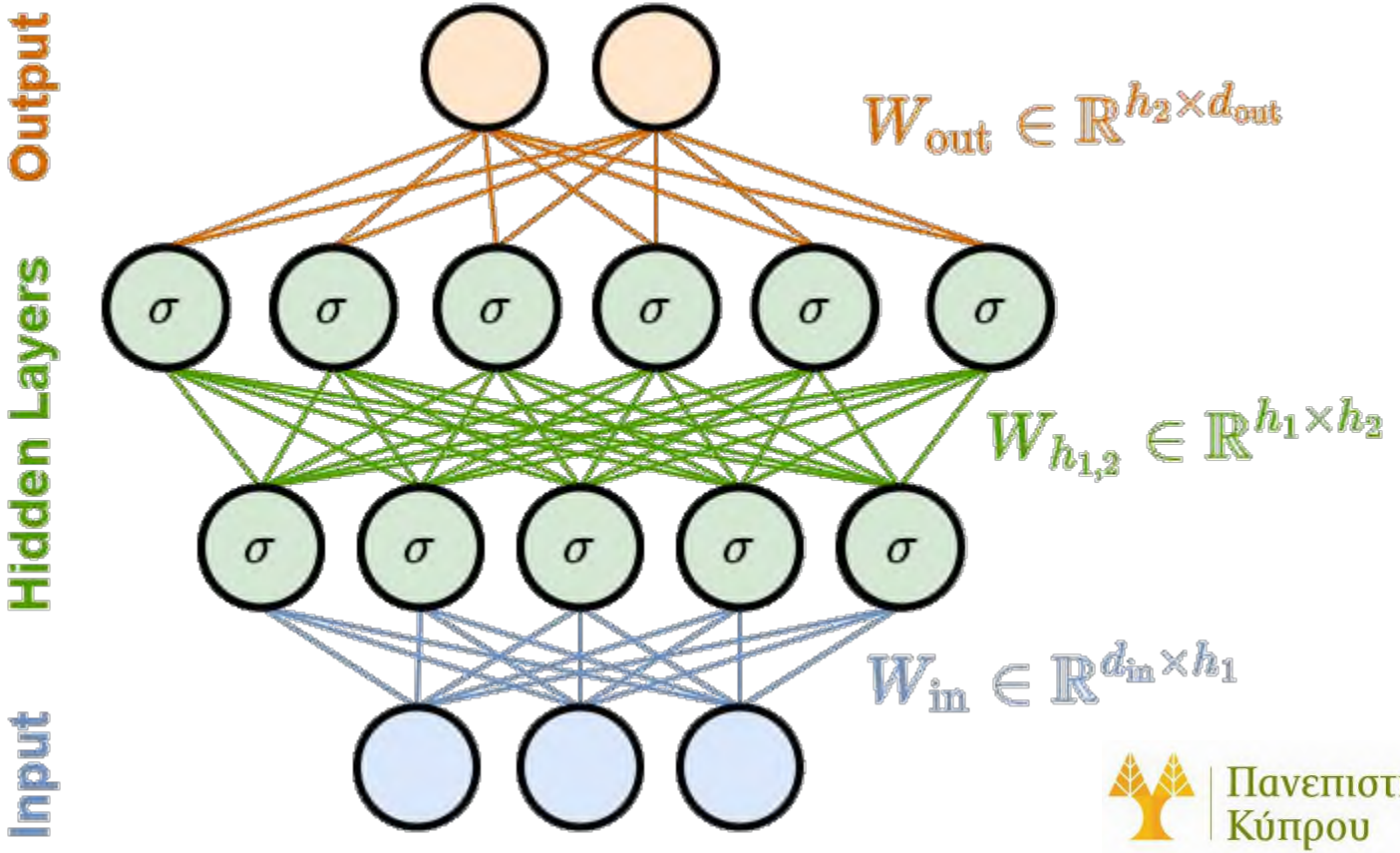




:data from a lower dimension (2D) to a higher dimension (3D) can become linearly separable by a hyperplane

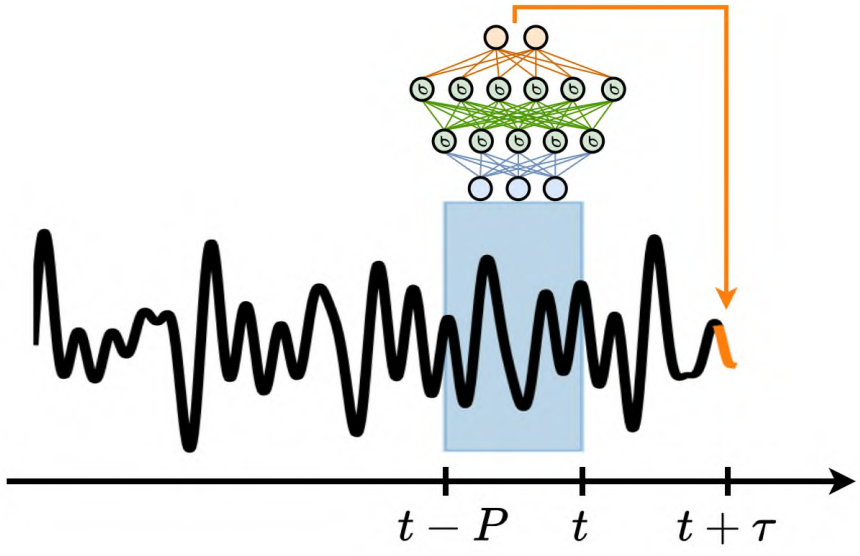
- A Multi-Layer Perceptron (MLP) is a simple Neural Network that consists of at least three layers of nodes:
  - An input layer
  - One or more hidden layers
  - An output layer
- The nodes in the hidden layers apply a nonlinear activation function σ
- MLP are generally trained with a supervised learning technique called backpropagation
- The MLP's layers are fully connected, meaning each node in one layer connects with a certain weight to every node in the following layer
- The first layer receives the input.
- The output of each layer is the input for the next layer until the final layer produces the output of the MLP
- The weights are stored in matrices W and are the trainable parameters of the model
- The MLP can learn complex mappings from inputs to outputs to perform a wide range of data modeling and prediction tasks

- Consider an example of an MLP with one input layer, two hidden layers (with h_1 and h_2 units respectively), and one output layer
- The input vectors are of size d_{in}, and the output vectors are of size d_{out}.



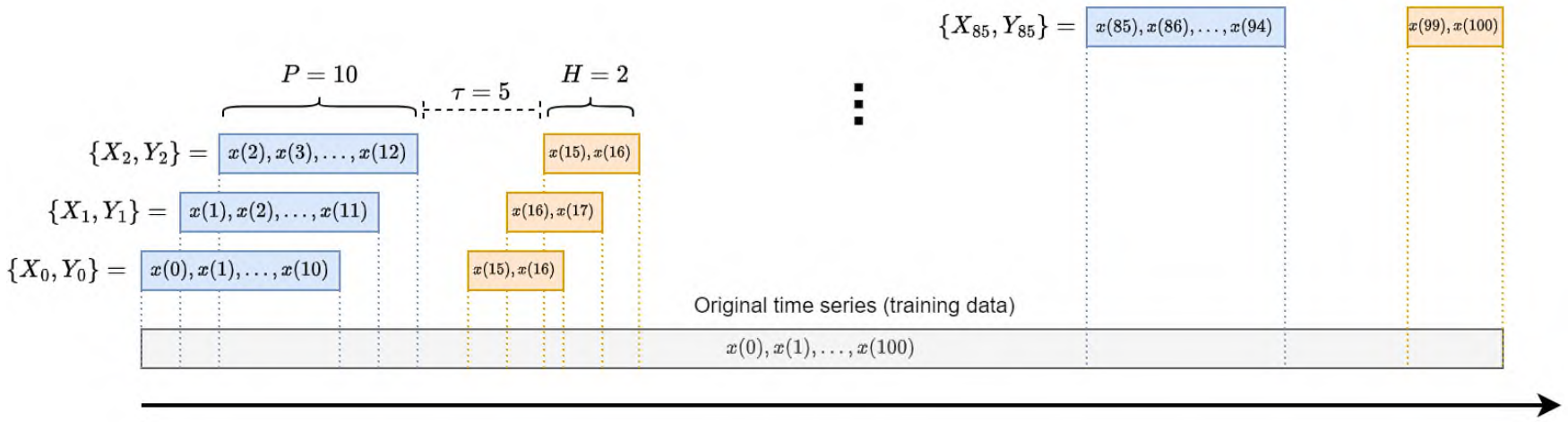
- Input layer
  - The size of the input layer corresponds to the size of the input vectors, d_{in}
  - This layer simply passes the input to the first hidden layer without applying any transformation
- First hidden layer
  - This layer has h_1 nodes
  - Each node connects to all the d_{in} inputs.
  - It applies a weight matrix $W_{in} \in \mathbb{R}^{d_{in} \times h_1}$ that transforms the input from vectors in $\mathbb{R}^{d_{in}}$ to vectors in $\mathbb{R}^{h_1}$
  - Then, it applies a nonlinear activation function σ, and outputs intermediate results
- Second hidden layer
  - This layer has h_2 nodes
  - It takes the outputs h_1 from the first hidden layer.
  - Applies another set of weights and a nonlinear activation function to produce h_2 intermediate results
- Output Layer
  - The output layer has d_{out} units.
  - It takes the h_2 outputs from the second hidden layer, applies weights and possibly a different activation function that depends on the task (usually, softmax for classification or no activation for regression)
  - The final output vector is of size d_{out}

- The MLP can be used as a windowed technique to perform time series forecasting
- The sequence of past values $x(t-P), \ldots, x(t)$ will represent our input vector ($d_{in} = \mathrm{P}$)
- The future value $x(t+\tau)$ will be the output that the model will learn to predict ($d_{out} = 1$)
- We can also train the MLP to predict a sequence $d_{out} = H$ of future predictions, e.g., $x(t+\tau), x(t+\tau+1), \ldots, x(t+\tau+H)$



- Key for the training of the MLP is the creation of input-output pairs $\{x_i, y_i\}$ used for training
- Given the input x_i the model must learn to predict the output y_i
- This is done by adapting the model weights to minimize the discrepancy between the prediction $\hat{y}_i$ and the desired output y_i
- The weights are modified according to the gradient taken with respect to a loss function $\mathcal{L}(\hat{y}_i, y_i)$ such that the MSE, e.g., $W \leftarrow W + \delta \frac{\partial \mathcal{L}}{\partial W}$ is a small constant defining the gradient step (learning rate)

- Training samples are generated from chunks of time series
- Each input-output pair consists of:
  - a window of past time series values $x(t-P), \ldots, x(t)$
  - a window of future values $x(t+\tau), \ldots, x(t+\tau+H)$



Πανεπιστήμιο Κύπρου

- Consider a time series of electricity consumption registered on a backbone of the energy distribution network of Rome
- The original time series has 10min resolution
- For this example, we will resample it to 1h resolution as it will become more smooth (and easier to predict)
- To train our models faster, we will consider only the first 3000 samples

```
ts_full = PredLoader().get_data('ElecRome')

# Resample the time series to hourly frequency
ts_hourly = np.mean(ts_full.reshape(-1, 6), axis=1)

# Use only the first 3000 time steps
time_series = ts_hourly[0:3000]
time_steps = np.arange(0, len(time_series))
```

```
# Split the time series into training and test sets
train_size = int(0.9*len(time_series))
tr = time_series[:train_size]
te = time_series[train_size:]
```

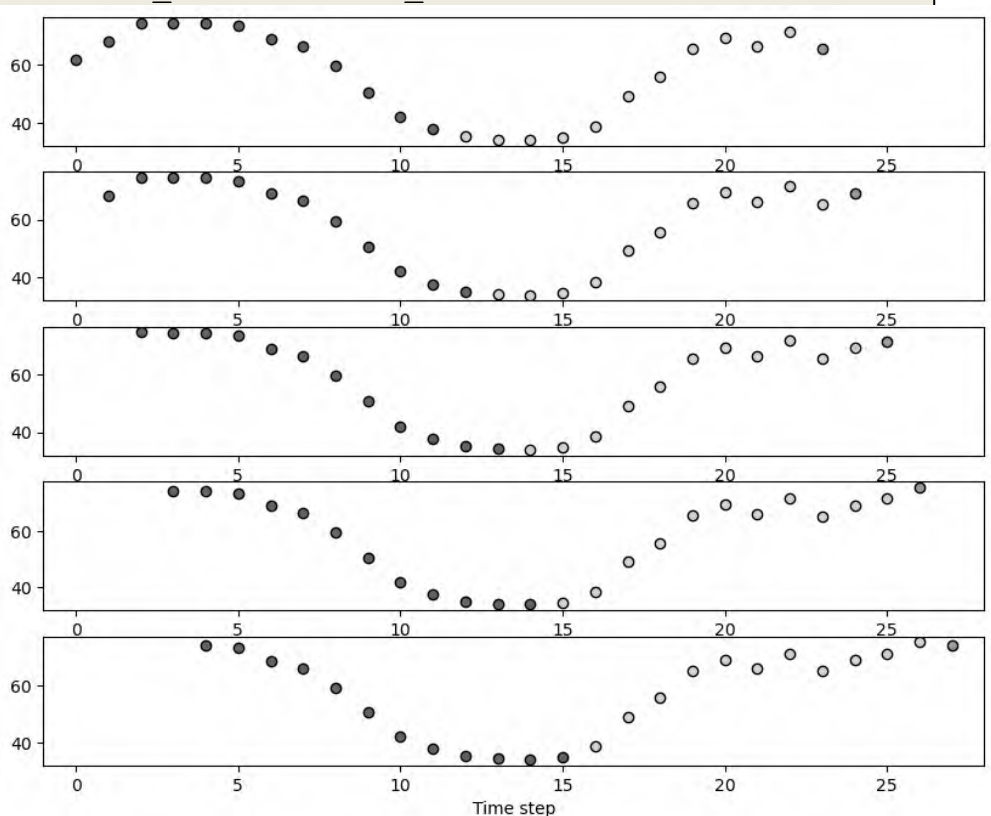
- Next, we define the function to create input-output pairs.
- For example, we create as input windows of size $P = 12$ by specifying window_size=12
- Similarly, we create as output windows of size forecast_horizon=12
- If we are interested in predicting only one sample, e.g., $\tau = 12$ and $H = 1$, we simply take the last element of the output

```
def create_windows(data, window_size, forecast_horizon):
    X, y = [], []
    for i in range(len(data) - window_size - forecast_horizon + 1):
        X.append(data[i:(i + window_size)])
        y.append(data[i + window_size:i + window_size + forecast_horizon])
    return np.array(X), np.array(y)

# Define window size and forecast horizon
window_size = 12
forecast_horizon = 12

# Create input-output pairs
X_train, y_train = create_windows(tr, window_size, forecast_horizon)
X_test, y_test = create_windows(te, window_size, forecast_horizon)
```

EXAMPLE

- Input-output pair visualization (blue input, orange output)
- $P = 12\ \tau = 12$

```
# Plot some pairs of input-output
fig, axes = plt.subplots(5,1,figsize=(10,8))
for i in range(5):
    axes[i].scatter(range(i, window_size+i), X_train[i], color='tab:blue',
edgecolor='k')  # Input
    axes[i].scatter(range(window_size+i, i+window_size+forecast_horizon-1),
y_train[i,:-1], color='lightgray', edgecolor='k')
    axes[i].scatter(i+window_size+forecast_horizon-1, y_train[i,-1],
color='tab:orange', edgecolor='k')
    axes[i].set_xlim(-1,28)
plt.xlabel('Time step')
plt.show()
```



- Predict energy consumption 1-day ahead by looking at the consumption of the previous 12 hours, $\text{forecast\_horizon} = 24$

```
# Define window size and forecast horizon
window_size = 12
forecast_horizon = 24
# Create input-output pairs
X_train, y_train = create_windows(tr, window_size, forecast_horizon)
X_test, y_test = create_windows(te, window_size, forecast_horizon)
# We are only interested in the last time step of the horizon
y_train = y_train[:, -1]
y_test = y_test[:, -1]
```

- Most neural nets, including the MLP, want the data to be normalized in a small range
  - Apply StandardScaler() from the sklearn library
  - scaler.fit_transform(X_train) will subtract from X_train its mean and divide by its variance
  - scaler.transform(X_test) will subtract from X_test the mean of X_train and divide by the variance of X_train

```
# Normalize the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

- Next, we define the MLP
  - Specify the size of the hidden layers as $h_1 = 16$ and $h_2 = 8$
  - Using larger values increases the model capacity, but can lead to overfit
- As the activation function we use a ReLU
- As the algorithms to compute the gradients and update the values of the parameters W we use Adam
- Finally, we train the model for 1000 iterations

```
# Define and train the neural network
mlp = MLPRegressor(hidden_layer_sizes=(16,8), activation='relu', solver='adam',
max_iter=1000)
mlp.fit(X_train_scaled, y_train)
```

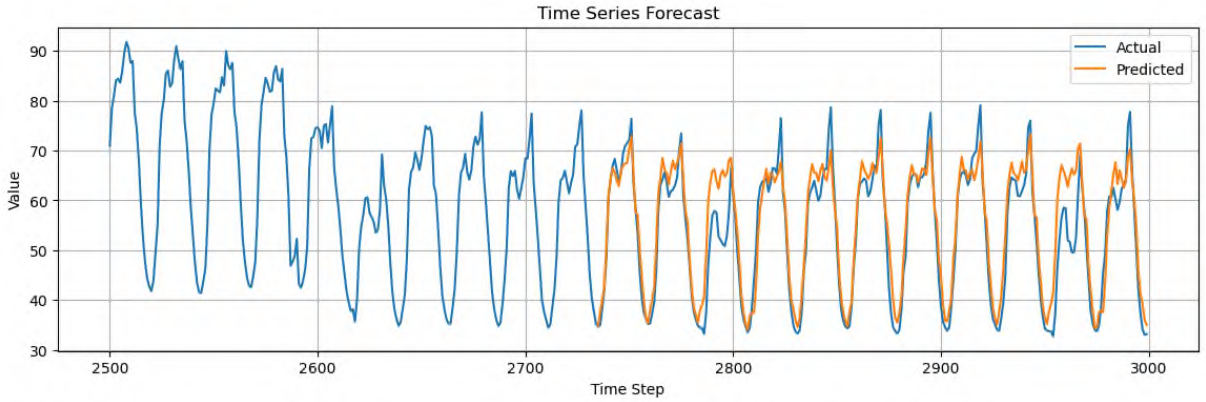
- Once the model is trained, we can compute the prediction on the test set
- We can also compute predictions beyond the whole dataset that we have available
- In this case, we do not have a way to check how well the model is doing since we do not have the actual data available
- However, it reflects a realistic forecasting scenario where we, indeed, do not know the future

```
# Predict on the test set
y_pred = mlp.predict(X_test_scaled)

# Forecast beyond the dataset using the model
last_window = time_series[-window_size:]
last_window_scaled = scaler.transform(last_window.reshape(1, -1))
next_step_pred = mlp.predict(last_window_scaled)

print(f"The next time step prediction is {next_step_pred[0]:.2f}")
```

- To check the performance of the model we can compute the MSE on the predictions of the test set
- We can also visualize the predictions against the real data

```
# Check performance with MSE
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")

plt.figure(figsize=(14, 4))
plt.plot(time_steps[2500:], time_series[2500:], label="Actual")
plt.plot(time_steps[-len(y_test):], y_pred, label="Predicted")
plt.title("Time Series Forecast")
plt.xlabel("Time Step")
plt.ylabel("Value")
plt.grid()
plt.legend()
plt.show()
```

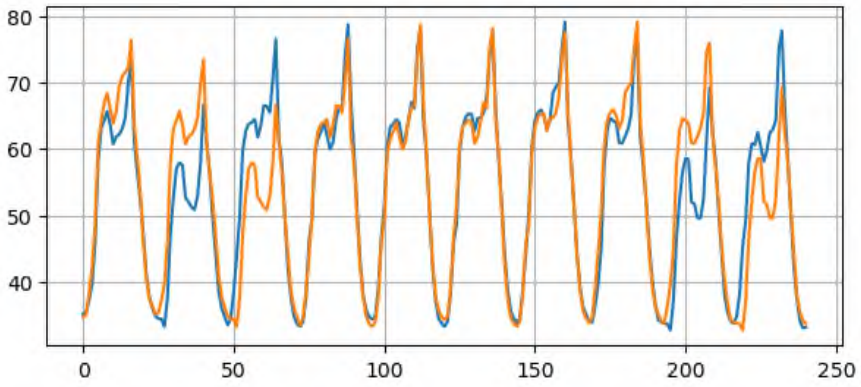
Mean Squared Error: 23.67



- Compare the performance against a simple baseline
- In this case, we are making a prediction at a forecast horizon equal to the main seasonality of the time series
- The most natural baseline is to use the values of the previous day as the prediction for the next day

```
# Compute the mse between the original time series and the time series shifhted
by 24 time steps
mse = mean_squared_error(y_test[24:], y_test[:-24])
print(f"Mean Squared Error: {mse:.2f}")

plt.figure(figsize=(7, 3))
plt.plot(y_test[24:])
plt.plot(y_test[:-24])
plt.grid()
plt.show()
```

Mean Squared Error: 27.14



- What is the optimal value of P?
- If P is fixed, how can we deal with different temporal dependencies?

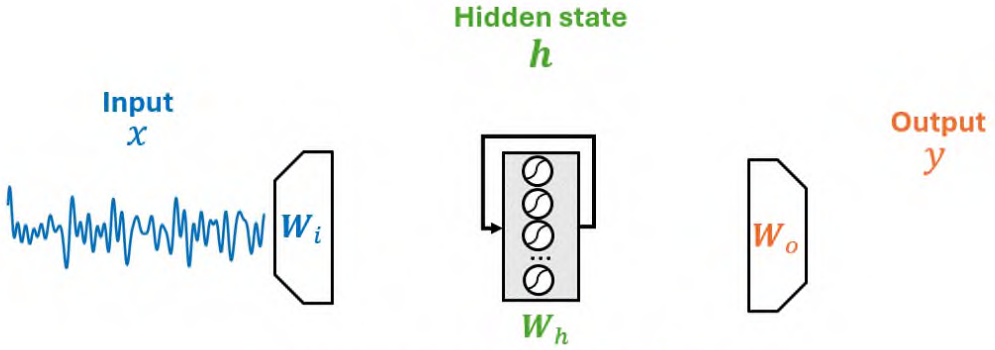
Fishes live under the

I am from Rome. I like mountain biking, my native language is

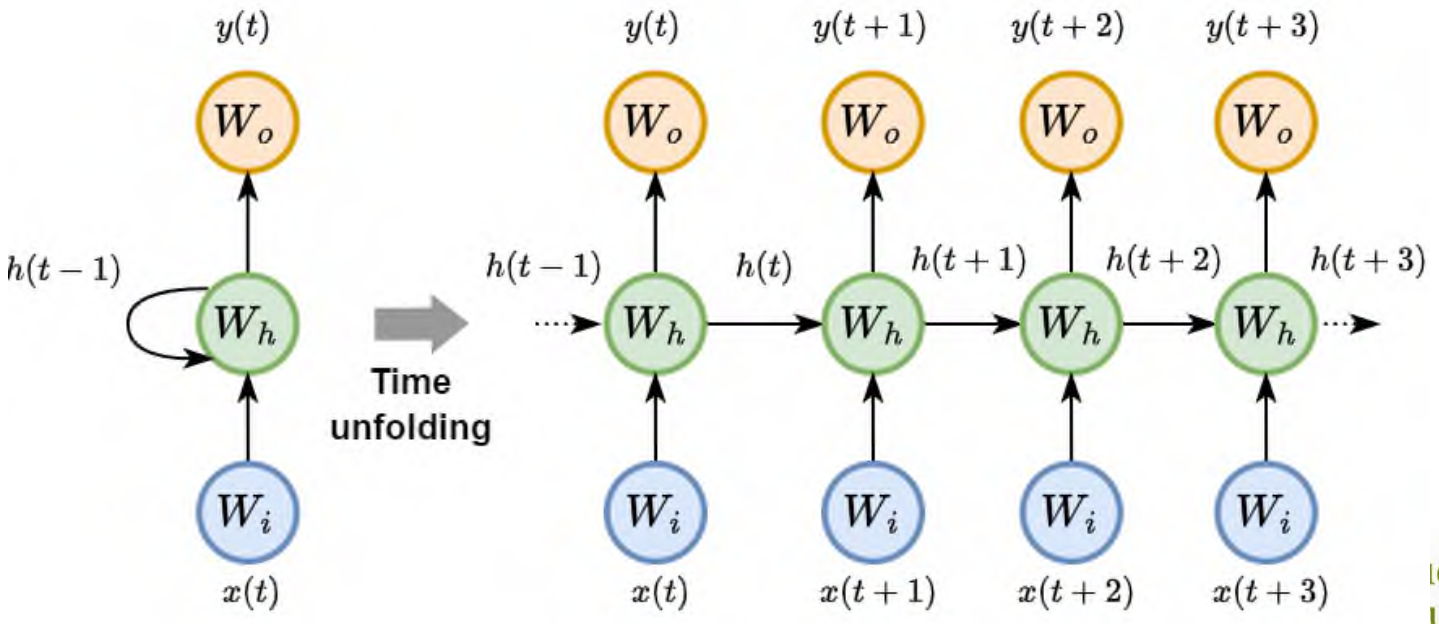
In the first case, we need to go back 4 time steps to retrieve the information we need. In the second, 9 steps.

- How to set P? What is the maximum memory we will ever need?
- As discussed previously, setting P too high will smooth our data too much
- Finding a good tradeoff is difficult…

- RNNs are a class of neural networks designed to recognize patterns in sequences of data, such as time series data
- Unlike traditional neural networks, RNNs have a memory that captures information about what has been calculated so far, essentially allowing them to make predictions based on past inputs
- As an RNN processes a sequence x maintains information in a hidden state h
- This process allows the RNN to use previous computations as a context for making decisions about new data

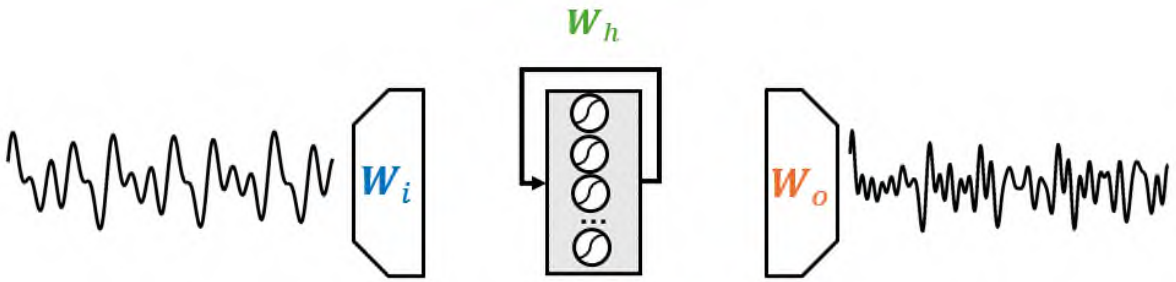


- BPTT is the algorithm used for training RNNs
- It involves unfolding the RNN through time to obtain a standard feed-forward neural network (like an MLP)
- After unfolding, one can apply the standard backpropagation algorithm
- BPTT computes gradients for each parameter across all time steps of the input sequence



- In BPTT the gradient has to go through all time steps and, due to the presence of nonlinearities, it can become too small and not reach distant time steps
    - This creates the problem called vanishing gradient
- Opposed, yet similar, is the problem of exploding gradient occurring when the gradients grow across the sequence.
- The vanishing gradient problem makes it hard for RNNs to learn long-range dependencies because updates to the weights become insignificantly small, causing the learning to stall
- Conversely, exploding gradients can cause weights to oscillate or diverge
- Techniques such as gradient clipping and gated units (e.g., LSTM, GRU) have been developed to mitigate these issues
- Another important limitation of the RNNs is that they fail to exploit hardware acceleration like other neural nets
- This is due to their recurrent nature that requires computations to be done sequentially rather than in parallel

- RC is a family of randomized RNNs, popularized in machine learning by Echo State Networks (ESNs)
- RC and ESNs are terms often used interchangeably
- There are two main differences that separate an ESN from an RNN:



State update: $\boldsymbol{h}(t) = \sigma\left(\boldsymbol{W}_i x(t) + \boldsymbol{W}_h \boldsymbol{h}(t-1)\right)$

output: $y(t) = \boldsymbol{W}_o\, \boldsymbol{h}(t)$

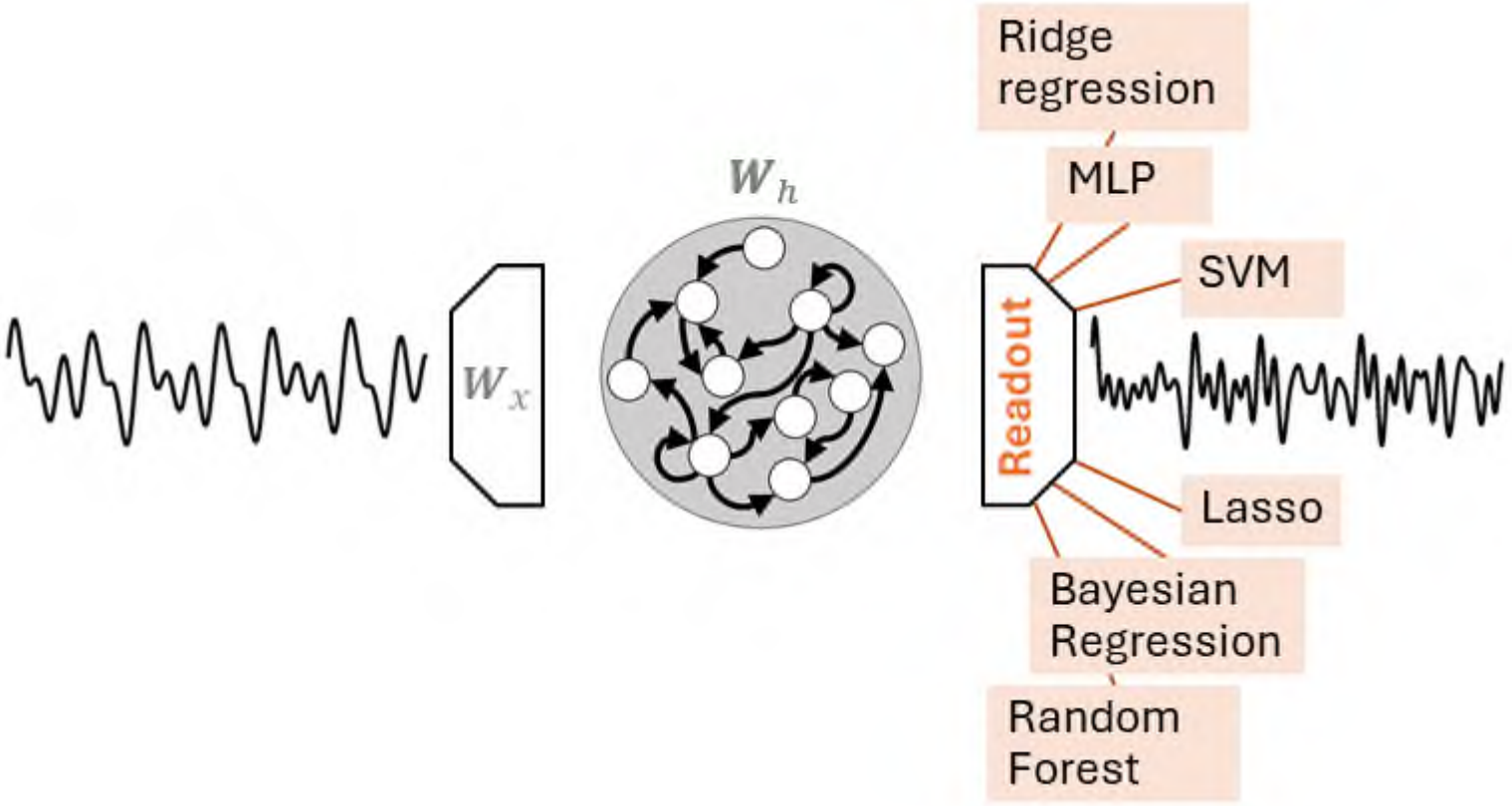
- The output weights W_o is the only part of the ESN that is trained

- Optimizing W_o can be done with a simple linear regression algorithm
- The workflow is as follows:
- Generate a sequence of reservoir states $H = \{h(0), \dots, h(t)\}$
- This is done by applying the state update equation

$$h_t = \sigma(W_i x(t) + W_h h(t-1))$$

- for each time step $t = 1, 2, \dots, T$ of the input sequence
- The nonlinearity σ is usually a hyperbolic tangent ($\tanh$).
- Apply a linear regression algorithm to compute a linear mapping $g(.)$ between the reservoir states and the desired output sequence $y = \{y(1), y(2), \dots, y(T)\}$
- The function $g(.)$ is called readout
- In a forecasting setting, $y(t)$ correspond to a future value of the input, e.g., $y(t) = x(t + \tau)$

- The readout $g(.)$ is usually implemented through a linear regression model, e.g., Ridge Regression
- In this case $g(.)$ corresponds to a weight matrix W_o
- However, any other regression model can be used to implement the readout, including an ML



- Fitting an MLP to the readout states is different from the window approach we saw before
  - Window approach:
    - The model predicts a future value from the fixed amount of temporal information contained in the window

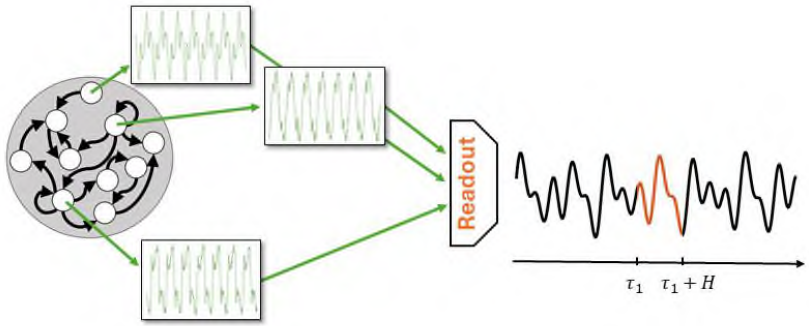
$$x(t+\tau) = g([x(t-P), \dots, x(t)])$$

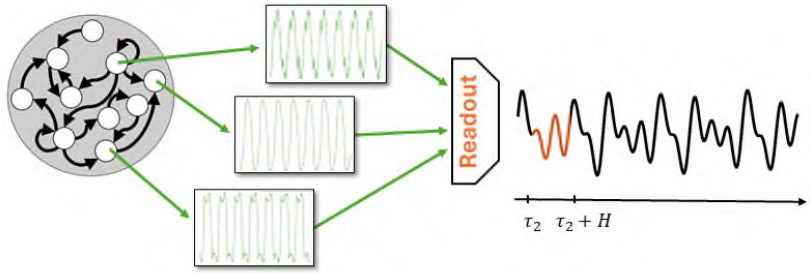
  - Reservoir approach:
    - The model predicts a future value from a single Reservoir state

$$x(t+\tau) = g(h(t))$$

- The Reservoir extracts a rich pool dynamical features from the time series
- These are embedded into the high-dimensional Reservoir state $h(t)$
- Contrary to a fixed window, $h(t)$ maintains a memory of all the previous inputs, back to the origin of the series $x(0)$
- Some of the features are relevant for the task at hand, while others are not
- The task of the readout is to select those features relevant for the task

- Say, we want make a forecast τ_1 steps ahead
- The readout will select a certain combination of dynamical features from the Reservoir





- To predict at a different horizon τ_2 the readout will select a different group of features
- Note that in both cases the Readout produces always the same pool of features!

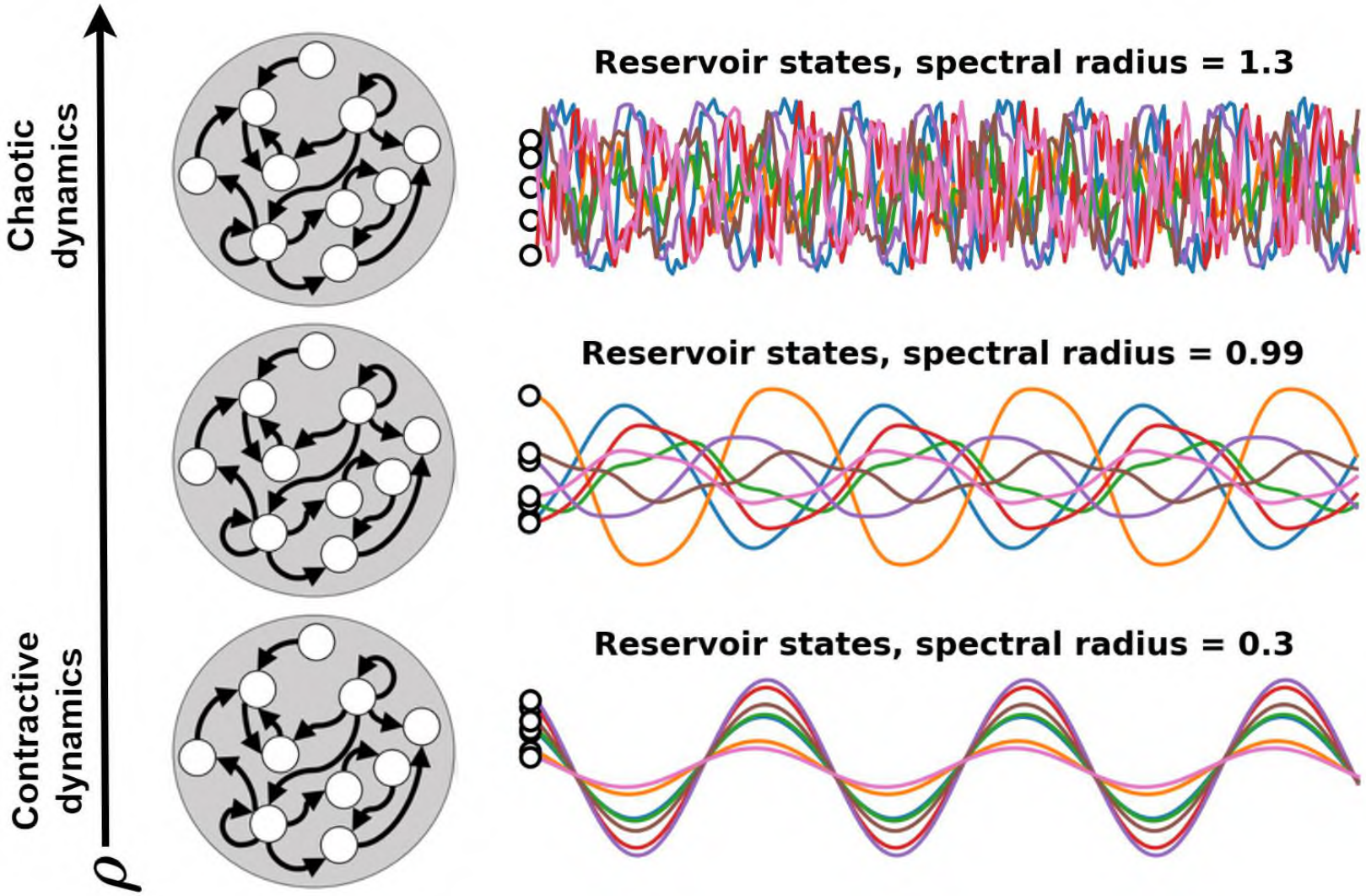
- The Readout is untrained and its states are generated without supervision, i.e., without an external guidance.

- Since it does not know what task it will have to solve, the Readout is configured to produce a pool of dynamic features that is most rich and varied as possible.

- In other words, the Readout trades the lack of training with a redundancy of generated features

Spectral radius

- More recent inputs must have a stronger influence on the current state
- So Reservoir should gradually forget its past states
- This is the echo state property
- It ensures to not model noise, to forget sporadic shocks, and the initial state that is uninformative
    - In control theory, this translate in having dynamics that are *contractive (*two initially different states eventually converge)
- On the other hand, we want the Reservoir to produce a rich pool of features
    - Does not happen if the dynamics of the Reservoir are too conctractive
- Need a sweet spot by tuning a parameter called spectral radius



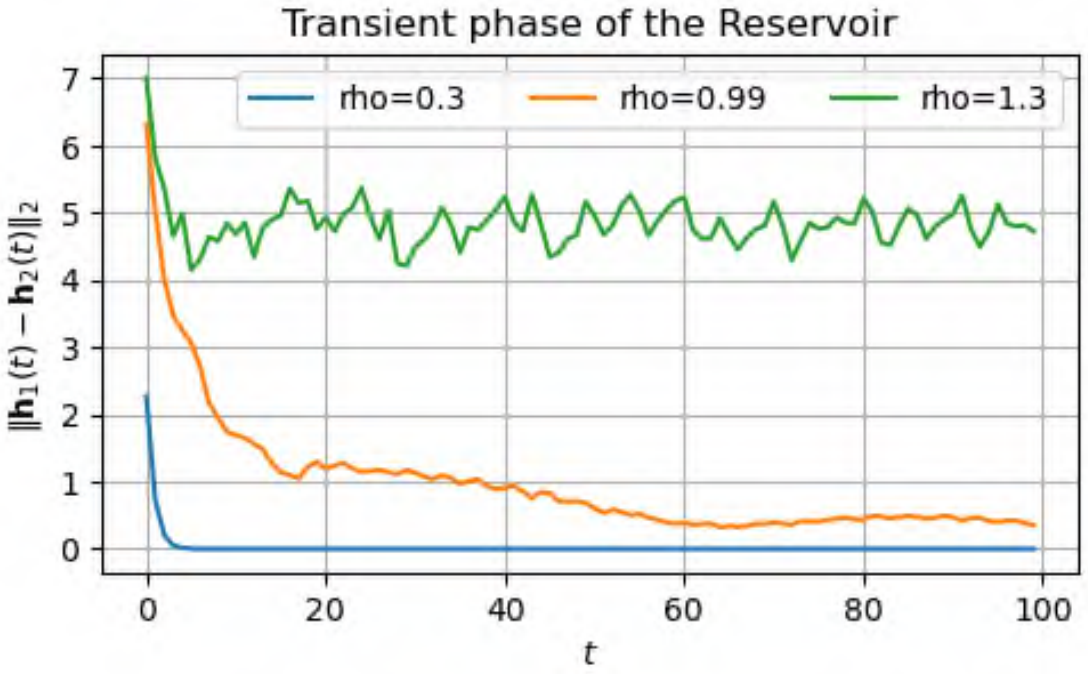
Πανεπιστήμιο Κύπρου

$$h_t = \sigma(W_i x(t) + W_h h(t-1))$$

- The spectral radius is the largest eigenvalue of the state transition matrix W_h
- We can set the spectral radius by computing the largest eigenvalue λ_{max} of W_h and then letting $W_h = \rho \frac{W_h}{\lambda_{max}}$
- A rule of thumb is to set ρ just below 1
- However, to achieve good performance it is often necessary to fine-tune ρ to values that can be lower or even higher than 1
- Another way of determining a good value of ρ is to look at the transient phase of the Reservoir
- This is how much time it takes to forget the initialization
- Assume two different initializations of the Reservoir state: $h_1(0)$ and $h_2(0)$
    - If the Reservoir dynamics is contractive, the effect of the different initializations will eventually fade
    - If it is chaotic, it will persist

- Two initializations: $h_1(0) = [0, \dots, 0]$ and $h_2(0) = [1, \dots, 1]$
- Set the input x to be always zero to not let it affect the evolution of the Reservoir state

```
# Initial states
initial_state_0 = np.zeros((1, 100), dtype=float)
initial_state_1 = np.ones((1, 100), dtype=float)

x = np.zeros((1, 100, 1)) # Zero input, it does not contribute to the state
rhos = [.3, 0.99, 1.3]    # We will use three different spectral radii

plot_states_evolution(x, rhos, initial_state_0, initial_state_1)
```

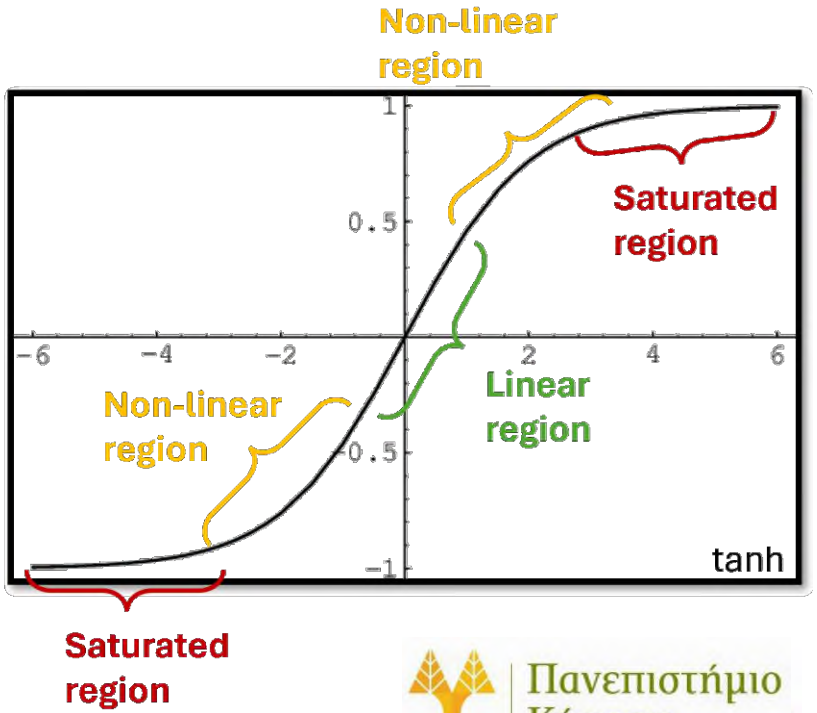


Πανεπιστήμιο Κύπρου

Input scaling

- Another critical value is the input scaling ω_{in}
- It multiplies the input weights W_i, changing their magnitude
- This is key to control the amount of nonlinearity in the model
- Reservoir units are usually equipped with a $\tanh$ activation
- A small value ω_{in} maps the Reservoir inputs towards the centre of the $\tanh$ where is more linear

- A small ω_{in} reduces the amount of nonlinearity
- A large ω_{in} translates into a more nonlinear behavior as the $\tanh$ is closer to saturation
- Good starting value around 0.1



Reservoir units

- Another hypeparameter N_h
- A larger value can give better performance at the cost of higher computation time
- A good starting point is usually $N_h = 300$, to be increased until there is no more gain in performance

Readout Sparsity

- There are also other hyperparameters in the ESN, such as the sparsity of the Readout and an optional noise to inject in the state update equation
- These are usually less critical than ρ and ω_{in}, and can be left to their default value in most cases
- Tuning hyperparameters in randomized architectures such as the ESN is much more important than in trainable neural networks, since there is no training that can compensate for poorly initialized models

- Use the Python library reservoir-computing
- Tasks in the library include classification, clustering, and forecasting

```
# Load energy data set
ts_full = PredLoader().get_data('ElecRome')

# Resample the time series to hourly frequency
ts_hourly = np.mean(ts_full.reshape(-1, 6), axis=1)[:, None]

# Use only the first 3000 time steps
time_series = ts_hourly[0:3000, :]
```

- Set input and target data Xtr and Ytr
- Use test data Xte and Yte to test the model
- Validation data Xval and Yval for hyperparameters tuning

- Given data X, the function forecasting_datasets computes:

1. Splits the dataset in consecutive chunks: train, val and test.
    - The size of the chunks is given by the values val_percent and test_percent
    - If we do not need validation data, set val_percent=0 (default) and the validation data will not be created
2. Create input data X and target data Y by shifting the data horizon time steps, where horizon is how far we want to predict
    - For example:
        - Xtr = train[:-horizon,:]
        - Ytr = train[horizon:,:]
    - Normalizes the data using a scaler from sklearn.preprocessing
    - If no scalers are passed, a StandardScaler is created
    - The scaler is fit on Xtr and then used to transform Ytr, Xval, and Xte
    - Note that Yval and Yte are not transformed

```
# Generate training and test datasets
Xtr, Ytr, Xte, Yte, scaler = make_forecasting_dataset(time_series,
                                                      horizon=24, # forecast
horizon of 24h ahead
                                                      test_percent = 0.1)
print(f"Xtr shape: {Xtr.shape}\nYtr shape: {Ytr.shape}\nXte shape:
{Xte.shape}\nYte shape: {Yte.shape}")
```

- Define the Reservoir hyperparameters and initialize

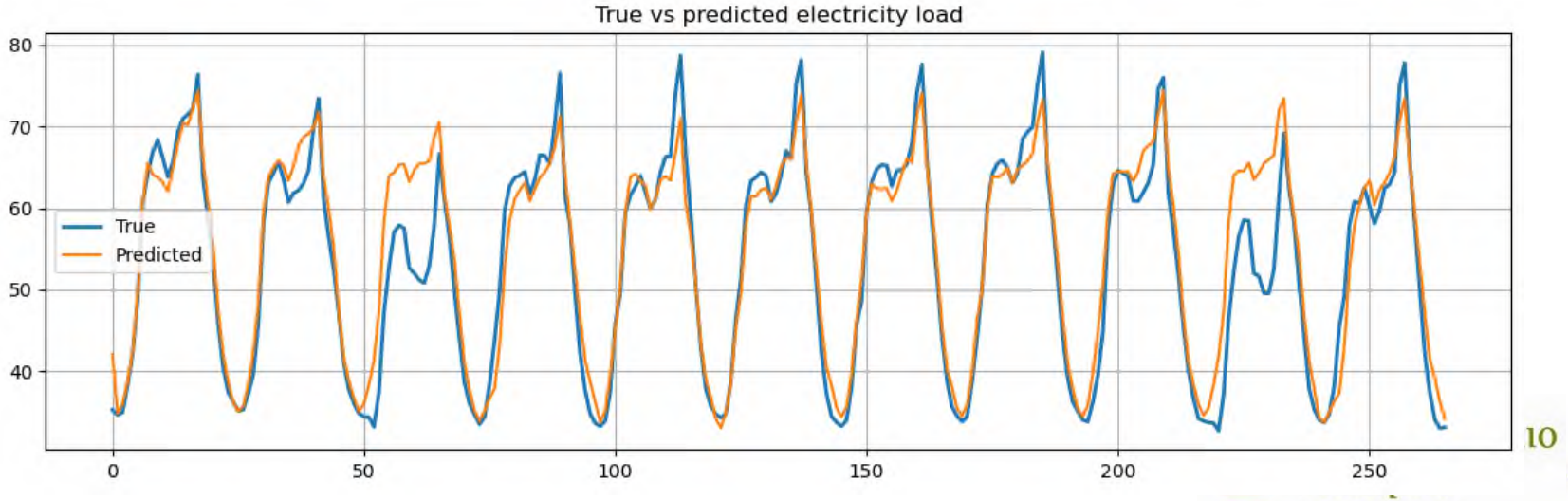
```
res= Reservoir(n_internal_units=900,
               spectral_radius=0.99,
               input_scaling=0.1,
               connectivity=0.25)
```

- Compute the sequence of the Reservoir states
  - Drop the initial states used for initialization $h(0)$

```
n_drop=10
states_tr = res.get_states(Xtr[None,:,:], n_drop=n_drop, bidir=False)
states_te = res.get_states(Xte[None,:,:], n_drop=n_drop, bidir=False)
print(f"states_tr shape: {states_tr.shape}\nstates_te shape: {states_te.shape}")
```

- Fit a linear readout implemented by Ridge regressor
- Use it to predict $\hat{Y}_{te}$

```
# Fit the ridge regression model
ridge = Ridge(alpha=1.0)
time_start = time.time()
ridge.fit(states_tr[0], Ytr[n_drop:,:])
print(f"Training time: {time.time()-time_start:.4f}s")

# Compute the predictions
time_start = time.time()
Yhat = ridge.predict(states_te[0])
print(f"Test time: {time.time()-time_start:.4f}s")

# Evaluate performance
mse = mean_squared_error(scaler.inverse_transform(Yhat), Yte[n_drop:,:])
```



- Reservoir states contain a rich, yet often redundant, pool of dynamics
- The readout job is to select only the dynamics that are useful for the task at hand
- However, training a readout on high-dimensional states is computational demanding, especially when using a sophisticated readout
- Also working with high dimensional data can increase the risk of multicollinearity, which destabilizes certain models, and overfitting
- May need to limit redundancy in the Reservoir states
- This can be done with an unsupervised dimensionality reduction procedure
- The most common and efficient dimensionality reduction procedure is PCA

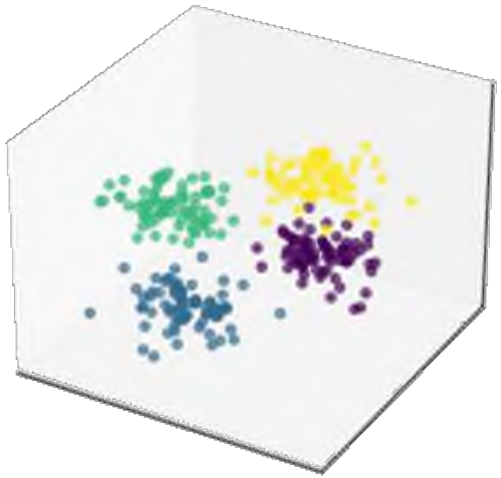
- PCA is a statistical procedure that utilizes an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components
- The number of principal components is less than or equal to the number of original variables
- By using a few components, PCA reduces the dimensionality of large data sets, by projecting the data onto a lower-dimensional space with minimal loss of information

- Our data is a sequence of length T Reservoir states, each one of size N_h
- They can be arranged in a matrix $H \in \mathbb{R}^{T \times N_h}$

- To illustrate the procedure, let's first consider a toy example with only $N_h = 3$ features
- In addition, we will create some structure in the data, by dividing the samples in 4 groups/clusters
- This will help us to see how PCA preserves the structure in the data

```
# Generate 4 clusters of points in 3 dimensions
T = 300 # number of samples
N_h = 3 # number of features
H, clust_id = make_blobs(n_samples=T, n_features=N_h,
                         centers=4, cluster_std=1.5, random_state=1)
```

```
plot_data(H, clust_id, interactive=False)
```



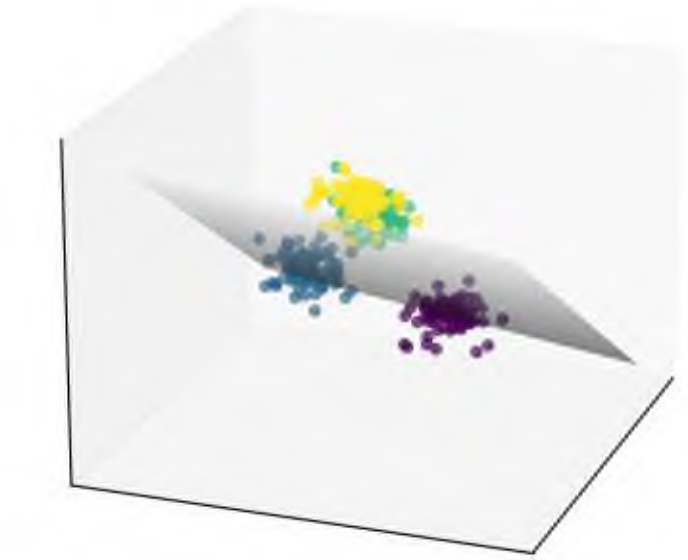
Reducing the data dimensionality with PCA invole the following steps:

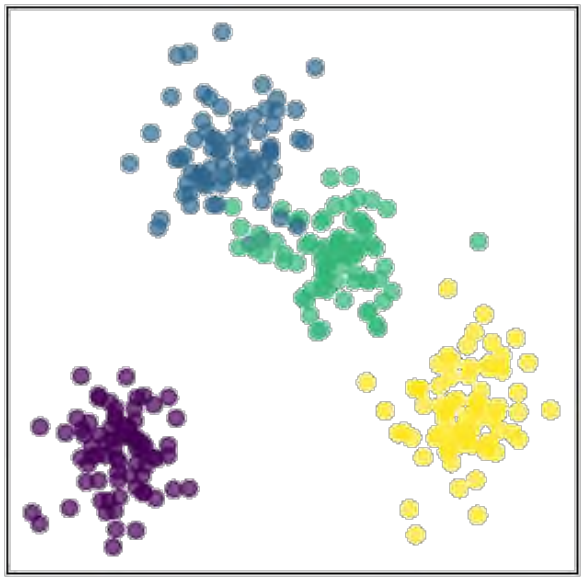
1. Standardization
    - Scale the data so that each feature has a mean of 0 and a standard deviation of 1
    - This is important because PCA is affected by scale
2. Covariance matrix computation:
    - Calculate the empirical covariance matrix $H^T H$
    - The matrix shows how changes in one variable are associated with changes in another variable
3. Eigenvalues and eigenvectors computation
    - The eigenvectors of the covariance matrix represent the directions of maximum variance
    - In the context of PCA, the eigenvectors are the principal components
    - The eigenvalues indicate the variance explained by each principal component.

4. Sorting eigenvectors:
    - The eigenvectors are sorted by decreasing eigenvalues
    - The top-k eigenvectors are selected, where k is the number of dimensions we want to keep
    - In our case, we keep $k = 2$ dimensions
5. Projection onto the new feature space:
    - The original data are projected onto the selected principal components
    - In our case, the 3-dimensional data are projected onto the plane spanned by the first two principal components
    - The projected data are the reduced-dimensional data

```
# Apply PCA to reduce to 2 components
pca = PCA(n_components=2)
H_pca = pca.fit_transform(H)
v1, v2 = pca.components_
```

```
# Plot the hyperplane spanned by the first two principal components
plot_pca_plane(H, clust_id, v1, v2, interactive=False)
```





```
# Plot the 2D projection
plt.figure(figsize=(4, 4))
plt.scatter(H_pca[:, 0], H_pca[:, 1], c=clust_id, cmap='viridis', alpha=0.7)
plt.xticks([], []), plt.yticks([], [])
plt.show()
```

- Can further reduce the number of dimensions
- In this case, we can go down to 1 dimension
- It boils down to projecting the data into the direction of maximum variation

```
H_pca_1d = PCA(n_components=1).fit_transform(H)

# Plot the 1D projection
plt.figure(figsize=(4, 1.5))
plt.scatter(H_pca_1d, np.zeros_like(H_pca_1d), c=clust_id, cmap='viridis',
alpha=0.7)
plt.xticks([], []), plt.yticks([], [])
plt.show()
```

- PCA for Reservoir states
- Since n_internal_units=900, we end up with a sequence of length T of vectors with size 900
- PCA used to reduce the dimensions to 75

```
# PCA for Reservoir states
pca = PCA(n_components=75)
states_tr_pca = pca.fit_transform(states_tr[0])
states_te_pca = pca.transform(states_te[0])
print(f"states_tr shape: {states_tr_pca.shape}\nstates_te shape:
{states_te_pca.shape}")
```

```
# Fit the ridge regression model
ridge = Ridge(alpha=1.0)
time_start = time.time()
ridge.fit(states_tr_pca, Ytr[n_drop:,:])
print(f"Training time: {time.time()-time_start:.4f}s")

# Compute the predictions
time_start = time.time()
Yhat_pca = ridge.predict(states_te_pca)
print(f"Test time: {time.time()-time_start:.4f}s")

# Compute the mean squared error
mse = mean_squared_error(scaler.inverse_transform(Yhat_pca), Yte[n_drop:,:])
print(f"Mean Squared Error: {mse:.2f}")
```

```
Training time: 0.0018s
Test time: 0.0001s
Mean Squared Error: 20.90
```

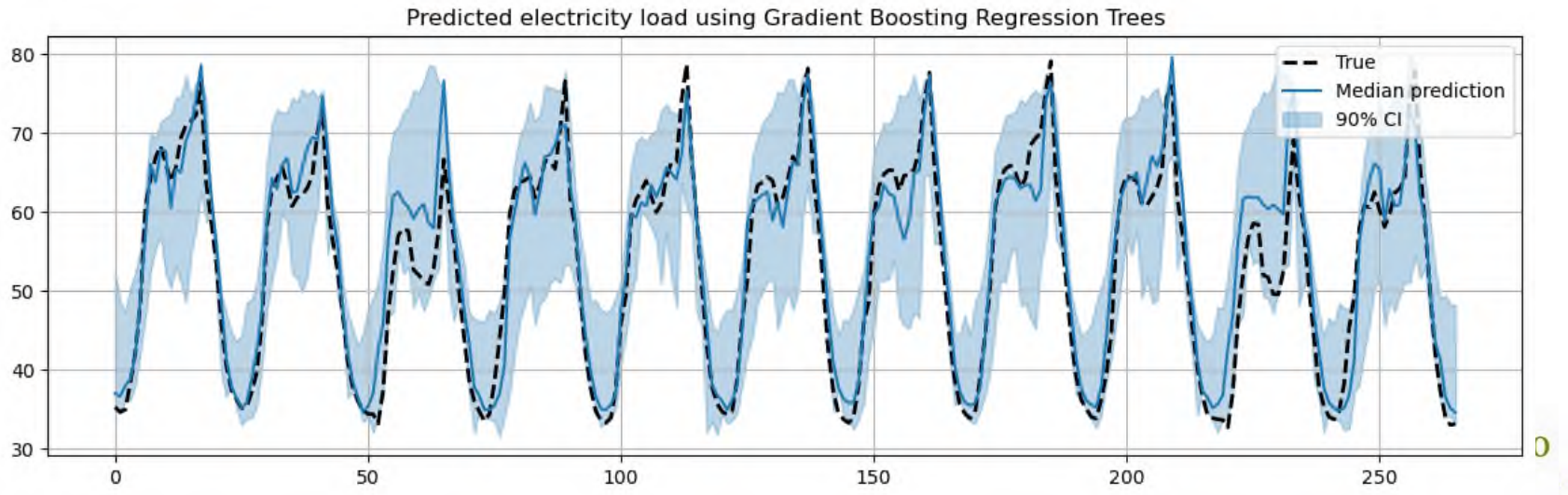
Performance was not impacted a lot but training and testing time reduced significantly

- Mapping the Reservoir states to the desired output is a standard regression problem, which can be solved by one of the many standard regression models in scikit-learn
    - For example, we can use a Gradient Boost Regression Tree (GBRT), which gives us predictions for different quantiles
    - In this way, we can compute confidence intervals in our predictions
    - This is a very simple way to implement probabilistic forecasting

- In the following example, we will fit a different model for the 0.5, 0.05 and 0.95 quantiles
- The 0.5 quantile will give us the most likely prediction for the future values
- The 0.05 and 0.95 quantiles together will us a 90% confidence interval for our prediction

FIT A GBRT READOUT

```
time_start = time.time()
# Quantile 0.5
max_iter = 100
gbrt_median = HistGradientBoostingRegressor(
    loss="quantile", quantile=0.5, max_iter=max_iter)
gbrt_median.fit(states_tr[0], Ytr[n_drop:,0])
median_predictions = gbrt_median.predict(states_te[0])
# Quantile 0.05
gbrt_percentile_5 = HistGradientBoostingRegressor(
    loss="quantile", quantile=0.05, max_iter=max_iter)
gbrt_percentile_5.fit(states_tr[0], Ytr[n_drop:,0])
percentile_5_predictions = gbrt_percentile_5.predict(states_te[0])
# Quantile 0.95
gbrt_percentile_95 = HistGradientBoostingRegressor(
    loss="quantile", quantile=0.95, max_iter=max_iter)
gbrt_percentile_95.fit(states_tr[0], Ytr[n_drop:,0])
percentile_95_predictions = gbrt_percentile_95.predict(states_te[0])
print(f"Training time: {time.time()-time_start:.2f}s")
```



```
time_start = time.time()
# Quantile 0.5
max_iter = 100
gbrt_median = HistGradientBoostingRegressor(
    loss="quantile", quantile=0.5, max_iter=max_iter)
gbrt_median.fit(states_tr_pca, Ytr[n_drop:,0])
median_predictions = gbrt_median.predict(states_te_pca)
# Quantile 0.05
gbrt_percentile_5 = HistGradientBoostingRegressor(
    loss="quantile", quantile=0.05, max_iter=max_iter)
gbrt_percentile_5.fit(states_tr_pca, Ytr[n_drop:,0])
percentile_5_predictions = gbrt_percentile_5.predict(states_te_pca)
# Quantile 0.95
gbrt_percentile_95 = HistGradientBoostingRegressor(
    loss="quantile", quantile=0.95, max_iter=max_iter)
gbrt_percentile_95.fit(states_tr_pca, Ytr[n_drop:,0])
percentile_95_predictions = gbrt_percentile_95.predict(states_te_pca)

print(f"Training time: {time.time()-time_start:.2f}s")
```

- Without PCA training time: 3.37s
- With PCA training time: 1.19s