Advanced Section 3
Echo-State Reservoir Computing

Marios Mattheakis

CS109B Data Science 2

Pavlos Protopapas, Mark Glickman and Chris Tanner

Outline

- Exploding and Vanishing Gradients
- Reservoir Computing: An echo-state RNN

Memory

A decision that a biological and an artificial neural network takes may or may not depend on an earlier decision.

I like reading any time. (**No memory is needed**, all times are the same)

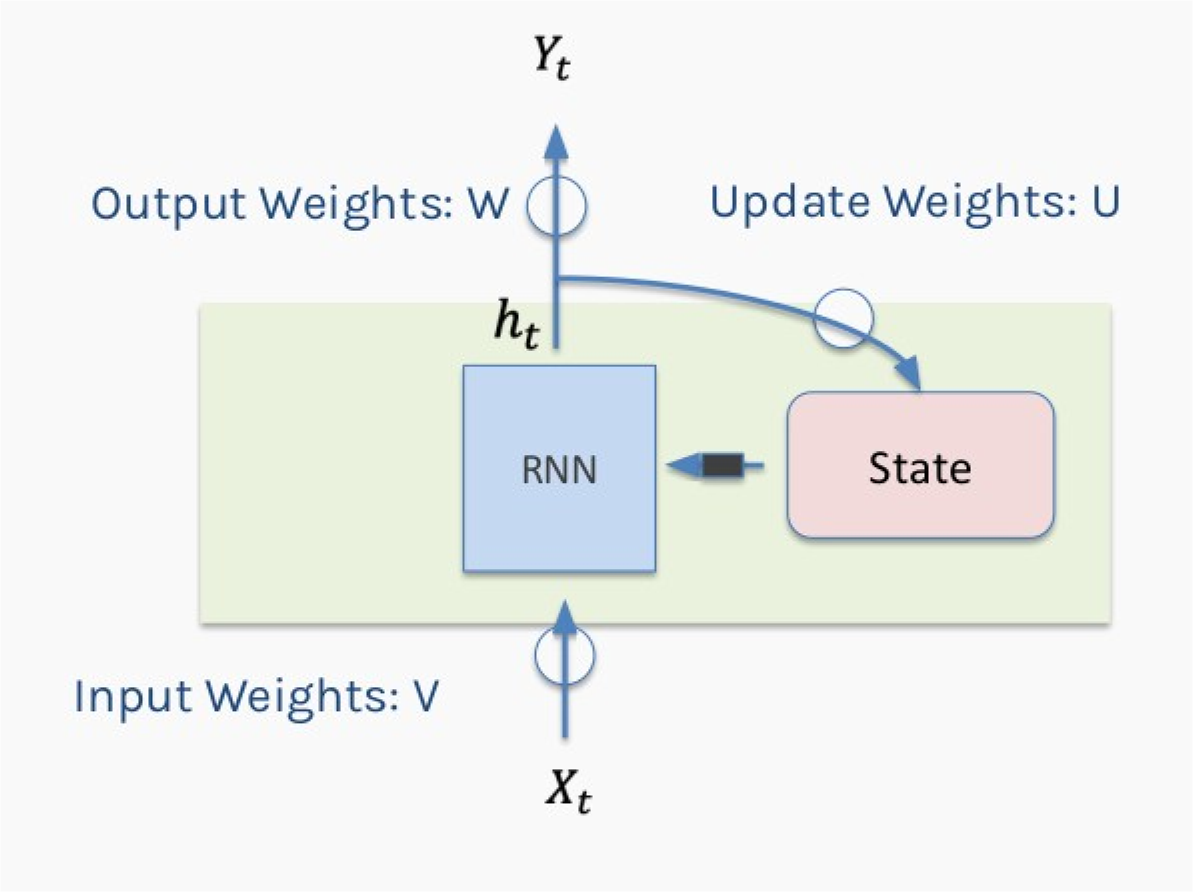
I like running before the lunch but I do not like running after the lunch. (**Memory is needed**, I need to remember if I had lunch or not)

We need an *artificial memory.* The recurrent connections in RNNs allow dynamical behavior. These recurrent hidden states provide **memory**.

Outline

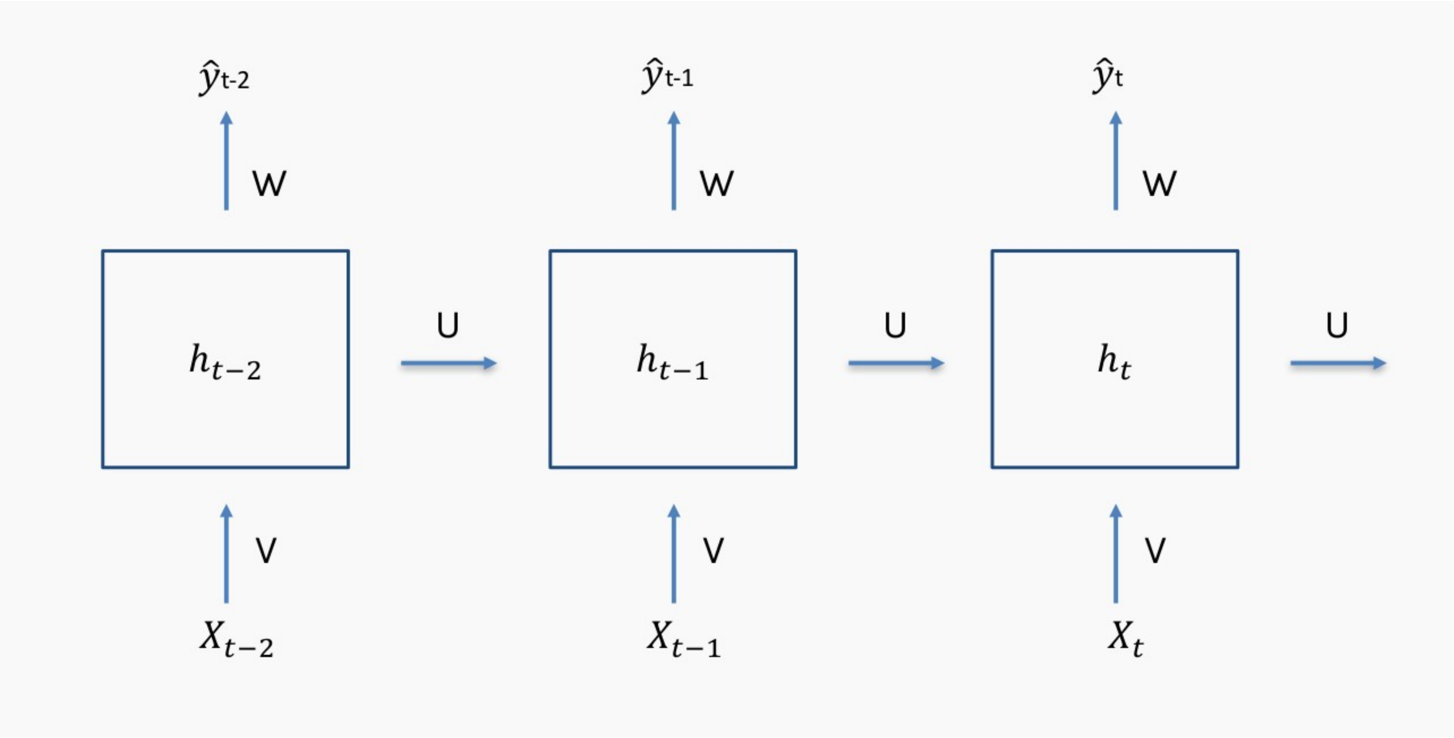
- **Exploding and Vanishing Gradients**
- Reservoir Computing: An echo-state RNN

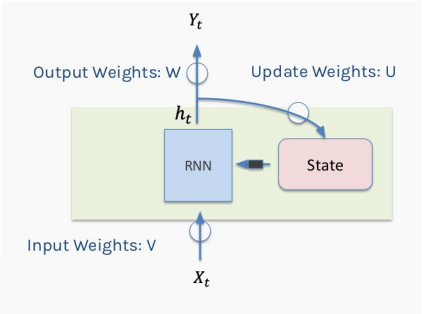
Train an RNN



Y_t
Output Weights: W
Update Weights: U
h_t
RNN
State
Input Weights: V
X_t

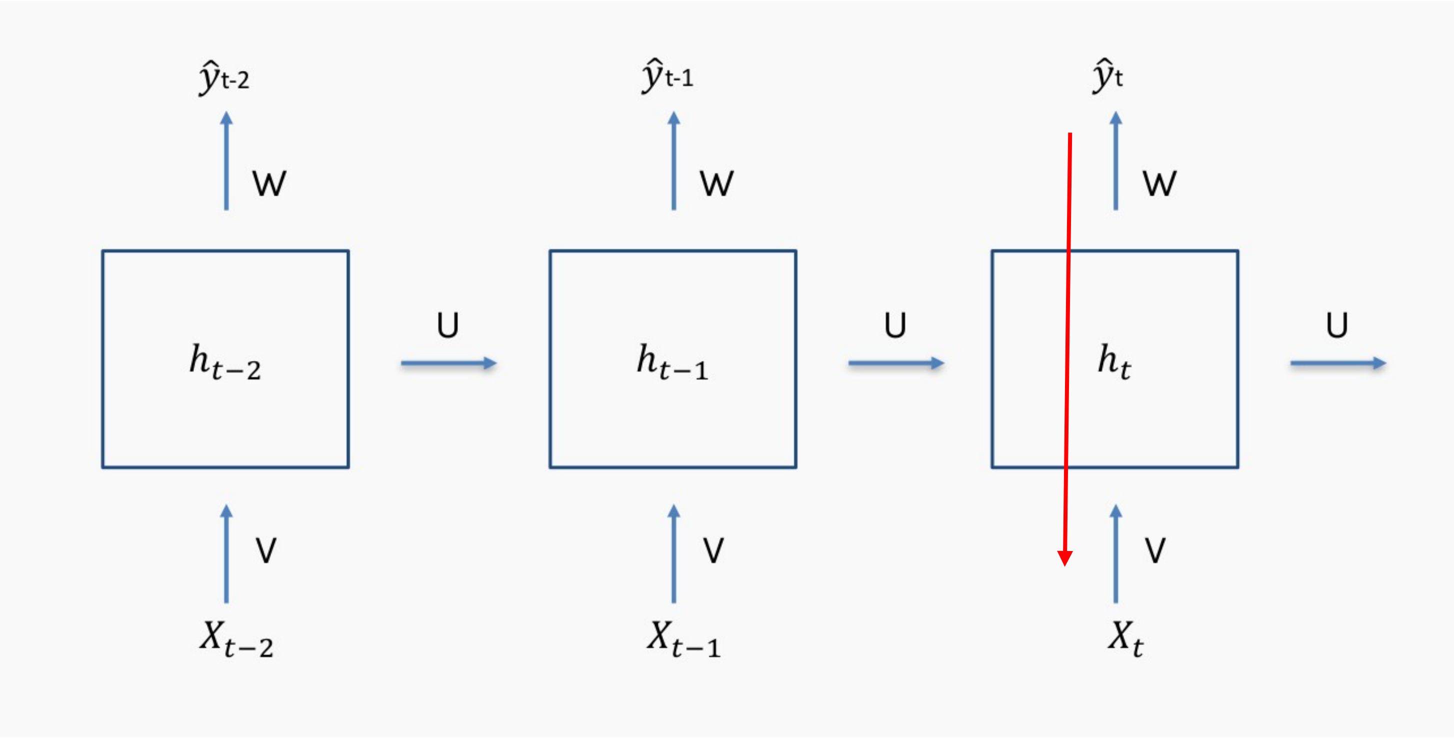
Train an RNN



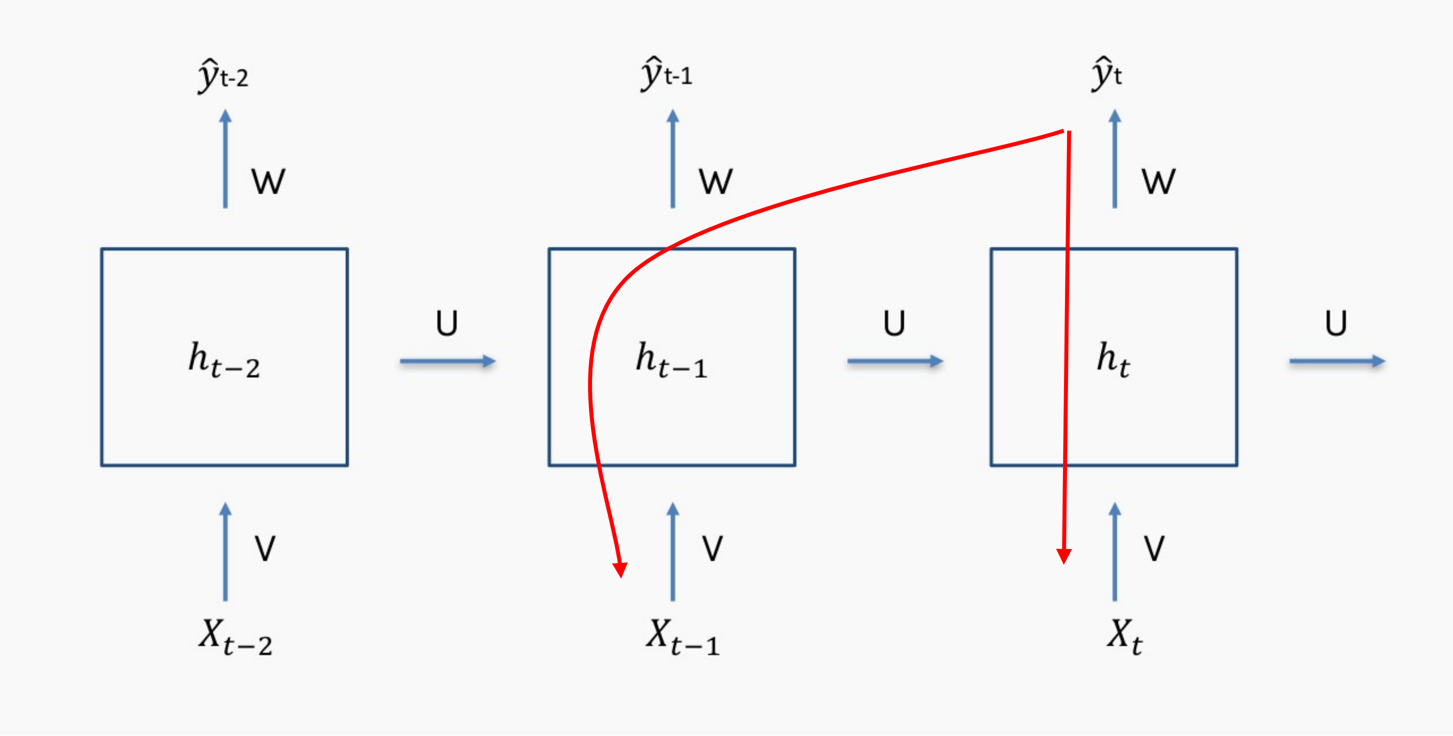


The training is extremely difficult due to the recurrent connections

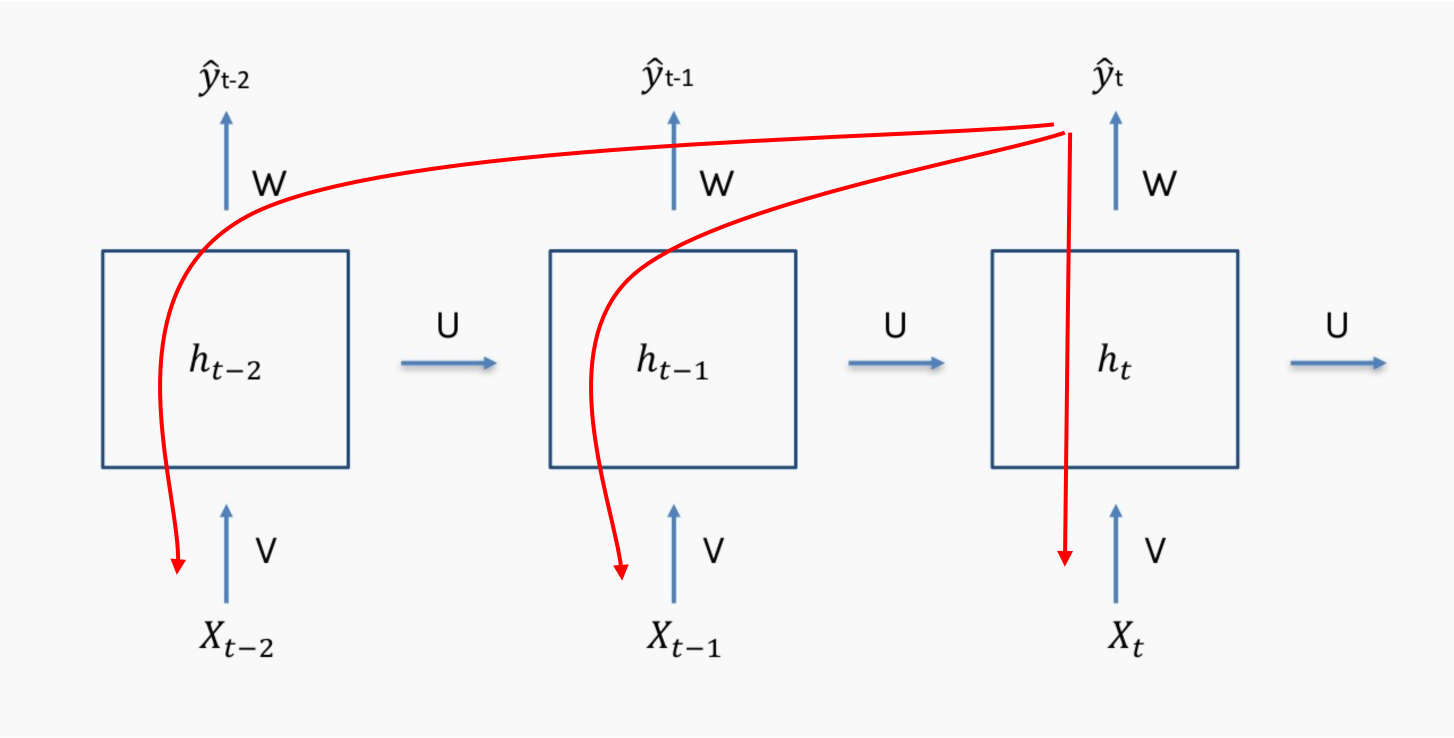
Back propagation



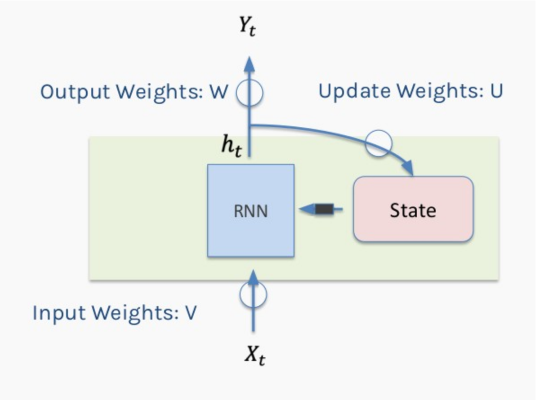
Back propagation



Back propagation



In the forward pass, we multiply the h in each time step. Similar in back-propagation. Getting deeper back in time, the gradients might become too strong or too weak



RNN Formulation

$$h_t = g_h\left(Vx_t + Uh_{t-1} + b'\right),$$

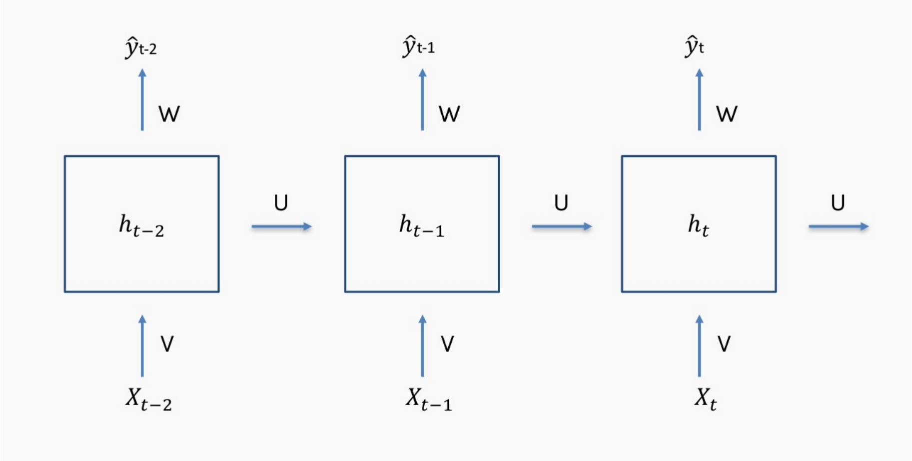
$$\hat{y}_t = g_y\left(Wh_t + b\right)$$

$g_h(\cdot)$: Activation for hidden states

$g_y(\cdot)$: Activation of the output layer

Total loss is the sum of loss in each time step

$$L = \sum_{t=1}^{T} L_t$$



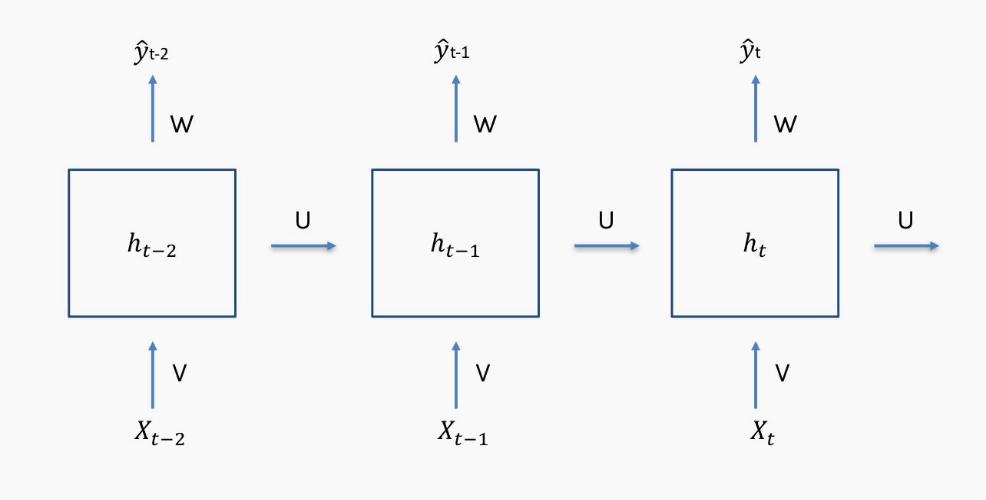
Consider a known sequence in the interval t = [1,T], this is the training set

Optimization

As usual, we have to minimize the total loss wrt all the weights *W, U, V*.

But let's focus only on the recurrent weights

$$\frac{dL}{dU} = \sum_{t=1}^{T} \frac{dL_t}{dU}$$



Explore the chain rule just for a single time step

Chain rule for a single time step

$$\frac{dL_t}{dU} = \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial U}$$

Chain rule for a single time step

$$\frac{dL_t}{dU} = \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial U}$$

$$= \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial U}$$

Chain rule for a single time step

$$
\begin{aligned}
\frac{dL_t}{dU} &= \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial U} \\
&= \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial U} \\
&= \sum_{k=1}^{t}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial U}
\end{aligned}
$$

Even computing the gradient in one time step requires a huge chain rule application because it demands all the previous times steps

Chain rule for a single time step

$$
\begin{aligned}
\frac{dL_t}{dU} &= \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial U} \\
&= \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial U} \\
&= \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial U}
\end{aligned}
$$

The *bad* term

Let's explore deeper ...

The “bad” term

Chain rule again

$$\frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_k}{\partial h_{k-1}}$$

$$= \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

The “bad” term

Chain rule again

$$\frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_k}{\partial h_{k-1}}$$

$$= \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

Jacobian matrix of the state to state transition.

The gradients is a huge product of Jacobian matrices.
Explore this term (almost done)

Explore the Jacobian Matrix

Remind:

$$h_t = g_h \left(V x_t + U h_{t-1} + b' \right)$$

Hence:

$$\frac{\partial h_j}{\partial h_{j-1}} = U^T g' \qquad \left(g' = \frac{\partial g_h}{\partial h_{j-1}} \right)$$

The Norm:

$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| = ||U^T g'|| \leq ||U^T|| \; ||g'|| = \beta_U \; \beta_h$$

Vanishing & Exploding gradients

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_U \ \beta_h)^{t-k}$$

$$(\beta_U \ \beta_h)$$

> 1: Exploding gradients

< 1: Vanishing gradients

It happens very fast as the time increases

Vanishing & Exploding gradients

Vanishing gradients:

Difficult to know in which direction to move for improving the loss function

Exploding gradients:

The learning becomes unstable

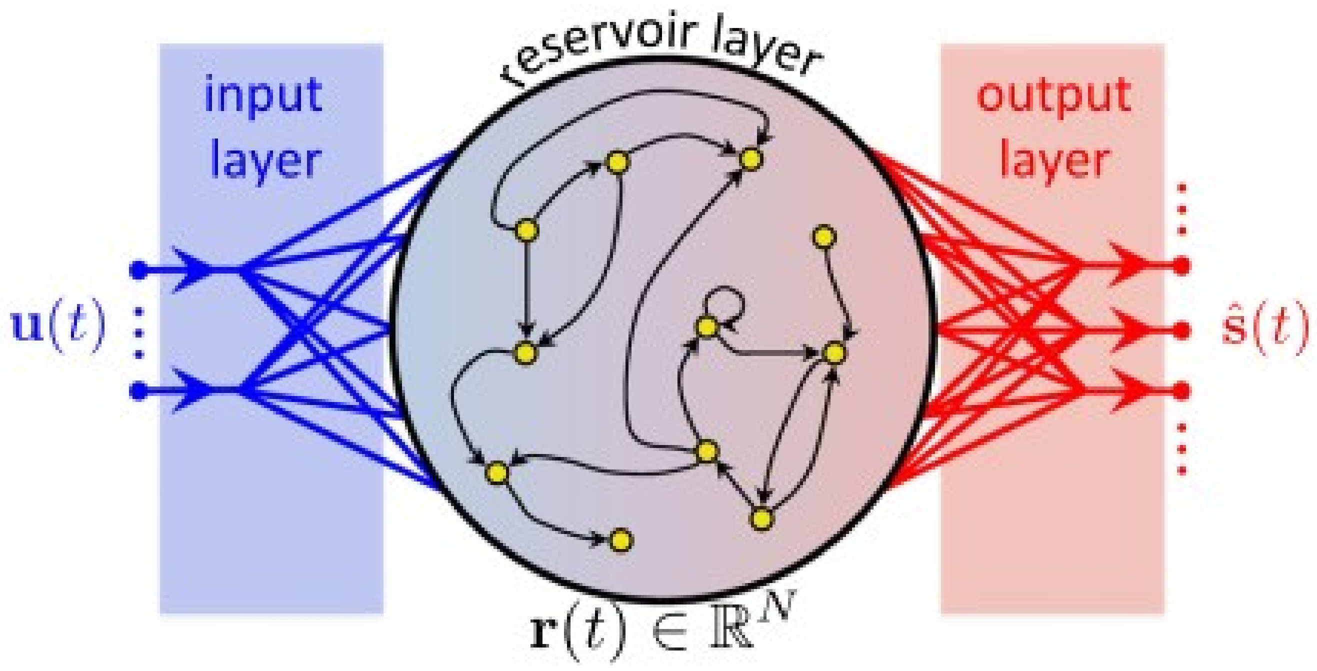
Solutions

- Clipping gradients
- Special RNN with leaky units such as:
  - Long-Short-Term-Memory (LSTM) and Gated Recurrent Units (GRU)
- Echo-state RNNs

Outline

- Exploding and Vanishing Gradients
- **Reservoir Computing: An echo-state RNN**

Reservoir Computing



Extremely fast training

Forecast, analyze, and control complex dynamical systems (weather, stock market etc...)

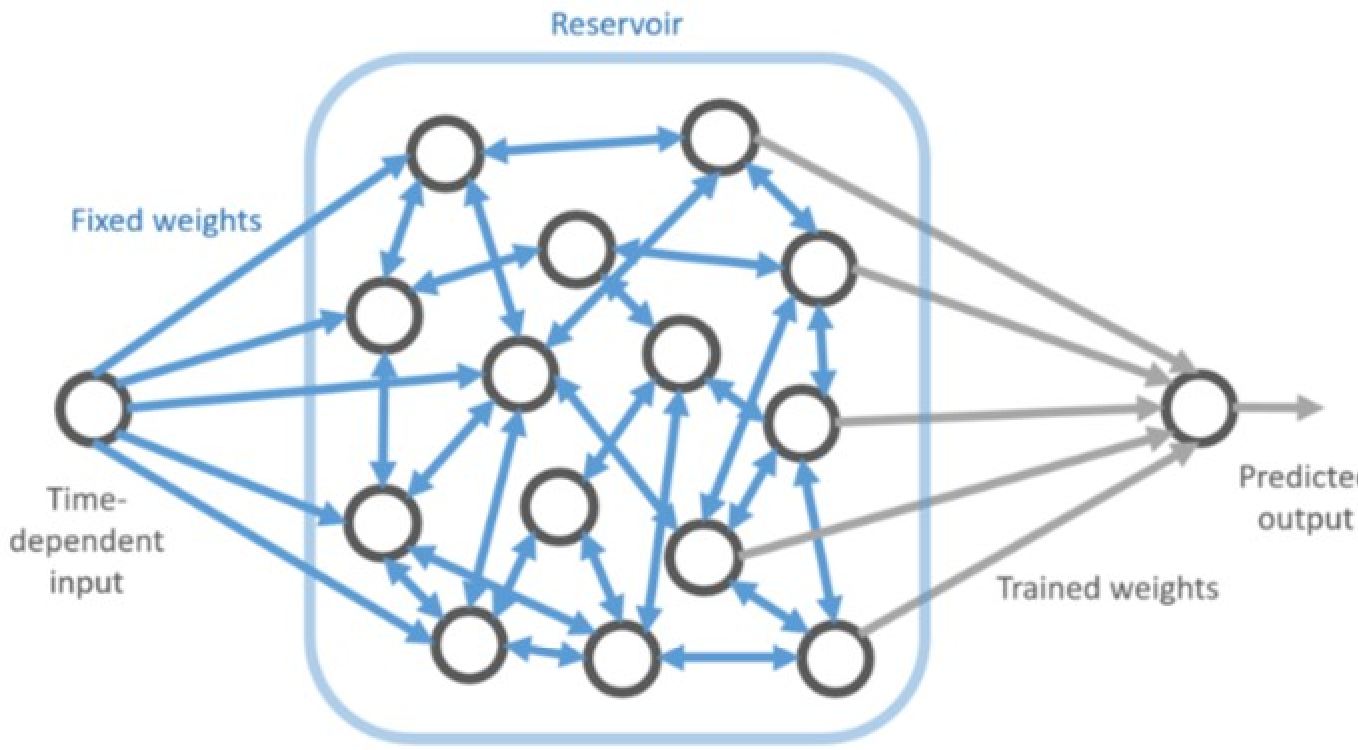
Echo-State RNNs

RNNs is extremely difficult to be trained:
Due to the recurrent and the input weights mapping to hidden states

Echo State Networks:
Fix the recurrent and input weights and learn only the output weights:

Reservoir: The hidden units form a reservoir of temporal features that capture different aspects of the history inputs.

RC Architecture



Input: An arbitrary length sequence input vector

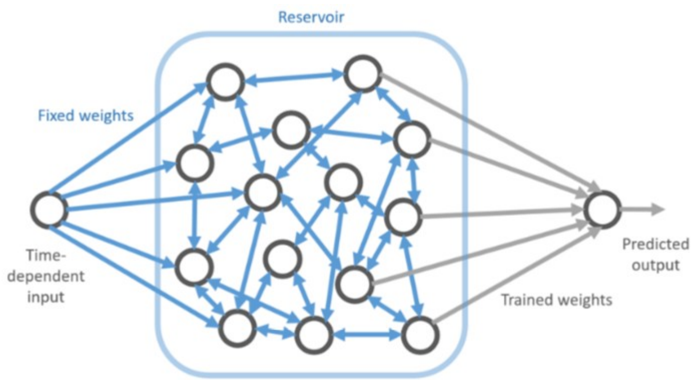
Reservoir: Mapping the input in a high-dimensional temporal feature space

Output: A linear or a classification layer depending on the task

RC Formulation

Input layer

$$W_{in}\mathbf{u}(t) + \mathbf{b}$$



Hidden recurrent dynamical nodes

$$\mathbf{r}(t+\Delta t) = (1-\alpha)\mathbf{r}(t) + \alpha f\left(W_{in}\mathbf{u}(t) + \mathbf{b} + W_r\mathbf{r}(t)\right)$$

Output layer

$$\hat{\mathbf{y}}(t) = g\left(W_{out}\mathbf{r}(t)\right) + \mathbf{b}'$$

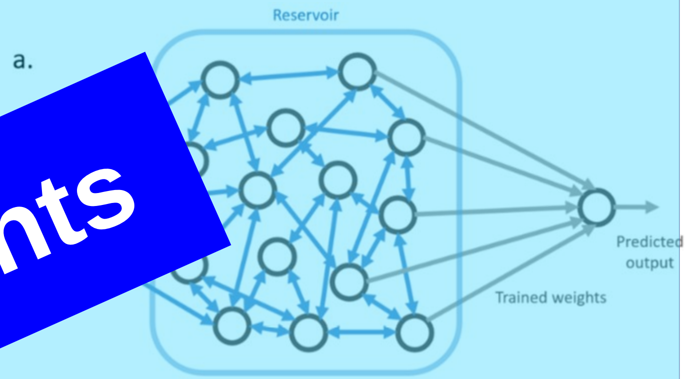
$$g() = \text{Identity}$$

$$g() = \text{softmax}$$

RC Formulation

Input layer

$$W_{in}\mathbf{u}(t) + \mathbf{b}$$



Hidden recurrent dynamical r

$$\mathbf{r}(t + \Delta t) = (1 - \alpha)\mathbf{r}(t) + \alpha f\left(W_{in}\mathbf{u}(t) + \mathbf{b} + W_r\mathbf{r}(t)\right)$$

Frozen Weights

Output layer

$$(t) = g\left(W_{out}\mathbf{r}(t)\right) + \mathbf{b}'$$

Learnable Weights

$$g() = \text{Identity}$$

$$g() = \text{softmax}$$

Recurrent states

The recurrent nodes consist a dynamical system. The states are determined by a system of autonomous nonlinear differential equations.

$$\frac{d\mathbf{r}(t)}{dt} = -\alpha\mathbf{r}(t) + \alpha f\left(W_{in}\mathbf{u}(t) + \mathbf{b} + W_r\mathbf{r}(t)\right)$$

$$\mathbf{r}(t+\Delta t) = (1-\alpha)\mathbf{r}(t) + f\left(W_{in}\mathbf{u}(t) + \mathbf{b} + W_r\mathbf{r}(t)\right)$$

Leaking Unit:

Decay constant α

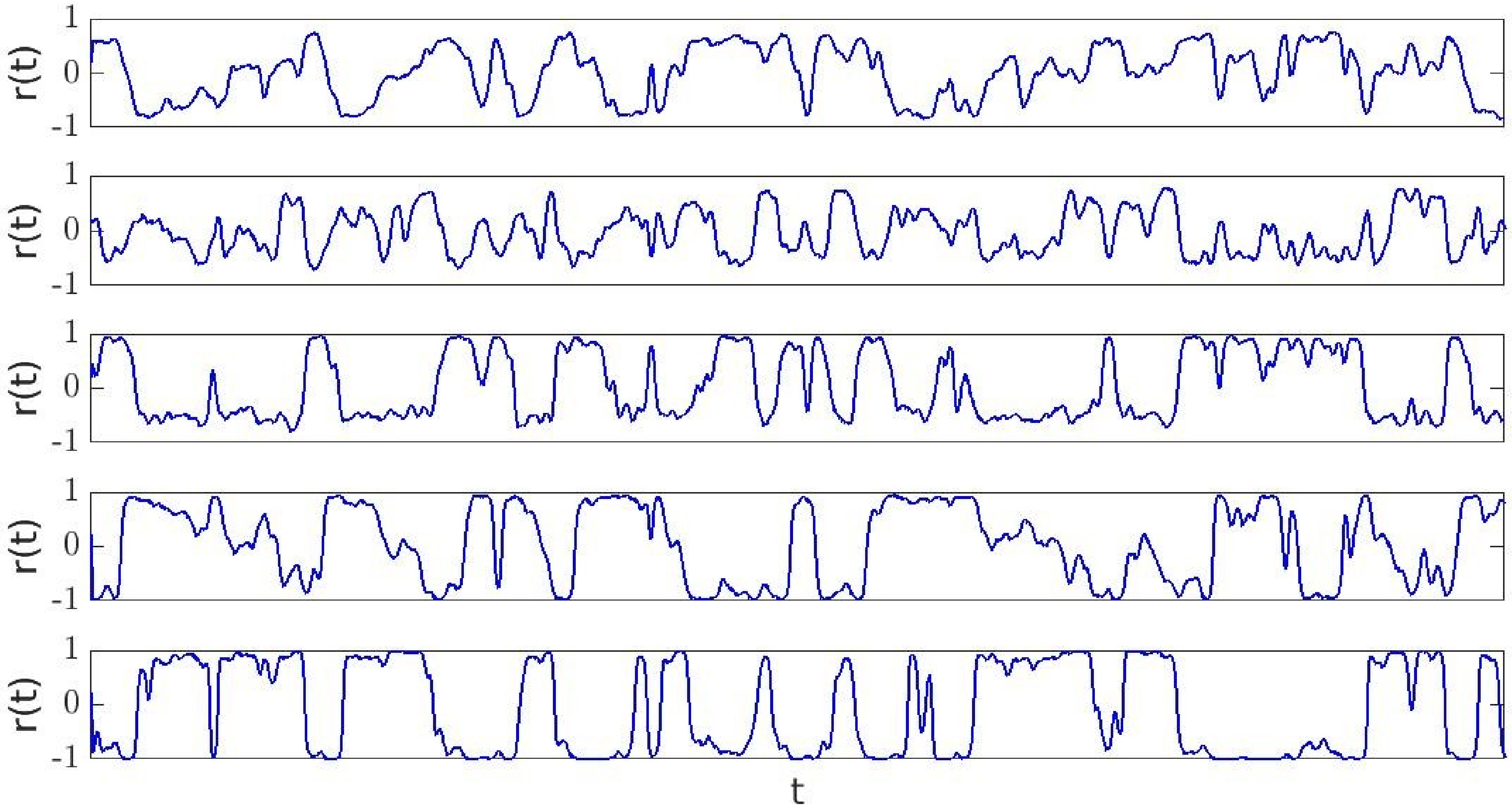
The nonlinear dynamics:

Activation function: tanh, sigmoid ...

Everything is fixed here

Recurrent states

An example of some hidden states



Recurrent states

Generalizations:

Include delay in the link times (same or different for each state)

$$\mathbf{r}(t+\Delta t) = (1-\alpha)\mathbf{r}(t) + \alpha f\left(W_{in}\mathbf{u}(t) + \mathbf{b} + W_r\mathbf{r}(t-\tau)\right)$$

More general:

$$\mathbf{r}(t+\Delta t) = h(\mathbf{r}(t-\tau), \mathbf{u}(t))$$

$$\text{for a nonlinear activation function } h(\cdot)$$

Hyper-Parameters Initialization

There are many hyper-parameters that we need to initialize

- The **leaking constant,** the **input bias**, and **link-times delay**
- The **number of hidden nodes** N (typically N > 500)
- W_{in} is randomly initialized by a **uniform distribution** in [-σ, σ]
- The **sparsity factor** N and the **spectral radius** ρ
- W_r is a sparse matrix randomly initialized by a normal distribution with D/N non-zero elements and with largest eigenvalue ρ

As usual, there is no systematic method to optimize the hyper-parameters

Training

We optimize **only** the output layer.

Essentially, it is a simple linear regression or a simple classification task

$$\hat{\mathbf{y}}(t) = g\left(W_{out}\mathbf{r}(t)\right) + \mathbf{b}'$$

Extremely fast training.

In many cases we know the exact solutions for the optimization process

Training: In a period [0, T] where all the signals (time-series) are known

Prediction: For t>T, where we know the input signals and predict the output

Forecasting

Linear regression

$$g(\cdot) = \text{Identity}$$

Loss Function: Mean Square Error + Ridge regularization

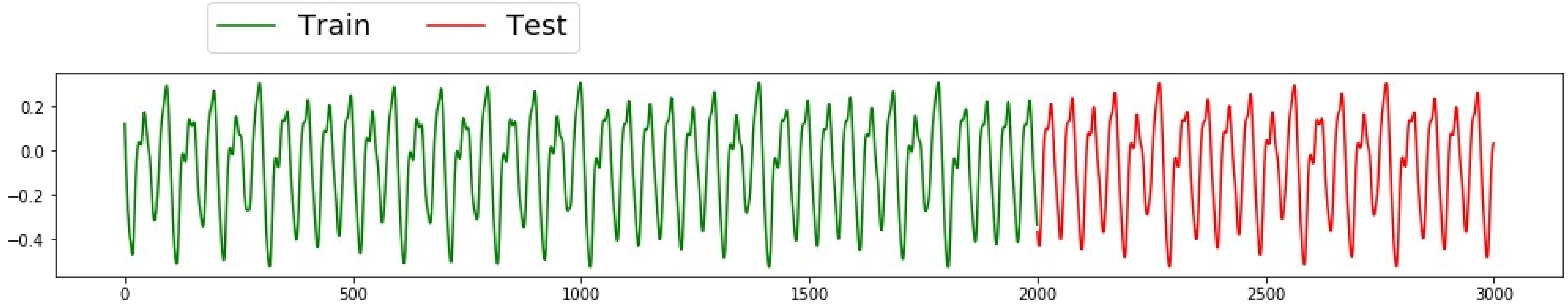
$$L = \sum_{t=0}^{T} \left(\mathbf{y}(t) - \hat{\mathbf{y}}(t)\right)^2 + \beta \mathrm{Tr}\left[W_{out} W_{out}^T\right]$$

Regularization: Often we add weak random noise in the hidden units

The strength of the noise and the ridge parameter are also hyper-parameters

Example: Mackey Glass

Task: Predict the evolution of a non-linear time series



Use the python class pyESN taken by:

https://github.com/cknd/pyESN

Example: Mackey Glass

```python
from pyESN import ESN
```

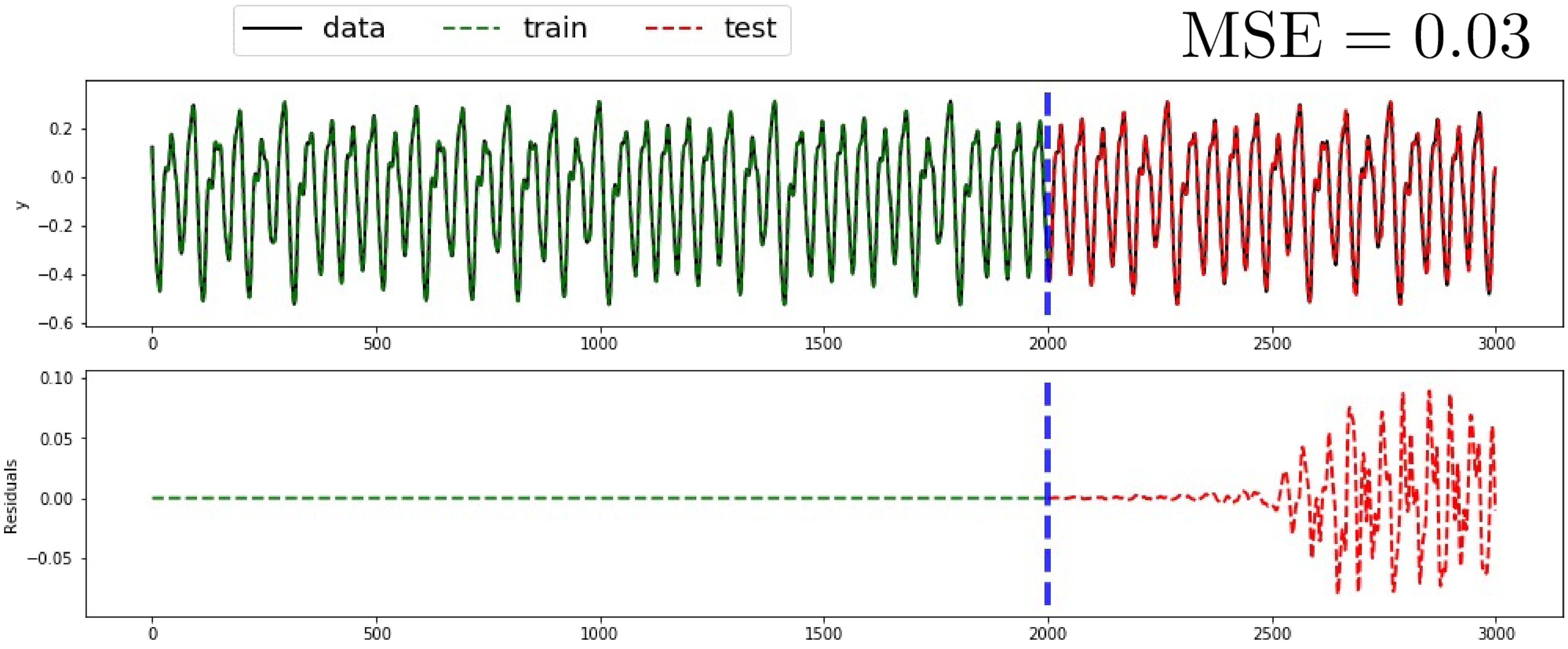
```python
# Setup the RC hyper-parameters
esn = ESN(n_inputs = 1,
          n_outputs = 1,
          n_reservoir = 1000,
          spectral_radius = 1.5,
          sparsity=.2,
          noise = 0.0001,
          random_state=42)

tTrain = np.ones(trainlen)
tTest  = np.ones(testlen)
# Output data
ytrain = data[:trainlen]
ytest = data[trainlen:trainlen+testlen]

# Train and predict
yfit = esn.fit(tTrain,ytrain)
yhat = prediction = esn.predict(tTest)
```

No input signal, so the input function is the identity

Example: Mackey Glass



Photonic Reservoir Computing

Besides the algorithms there are many **hardware setups for RC**

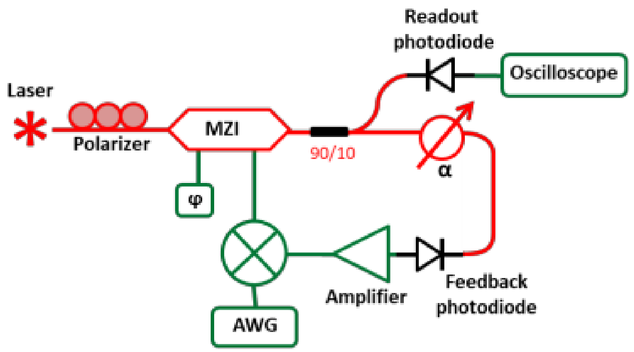


Fig. 1: Hardware schematic. Physical TDR consisted of Mach-Zehnder interferometer (MZI), arbitrary waveform generator (AWG), electrical amplifier, two photodetectors, fiber splitter (90/10), and tunable optical attenuator (α). Red lines denote optical components while green lines indicate electrical components.

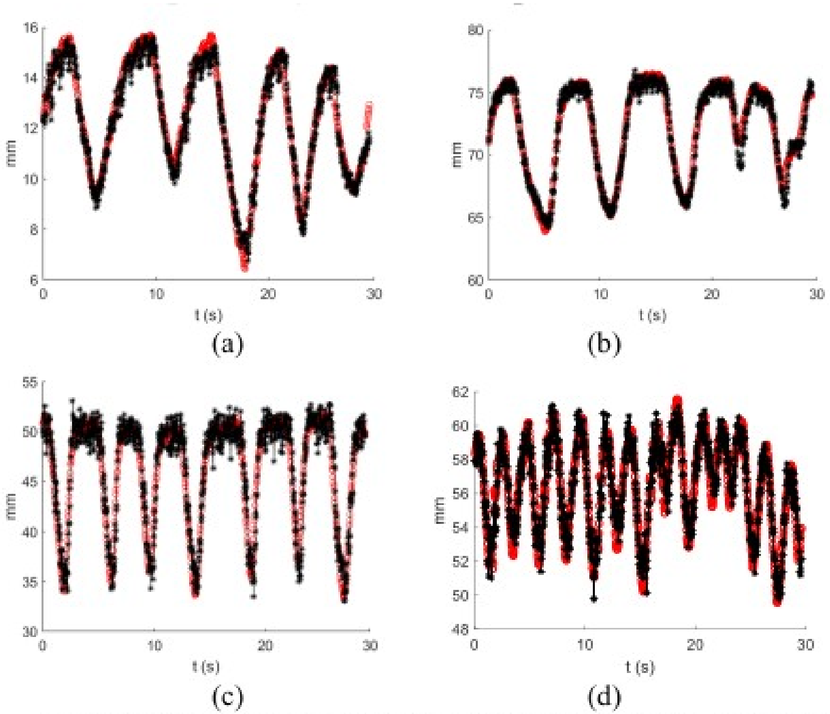


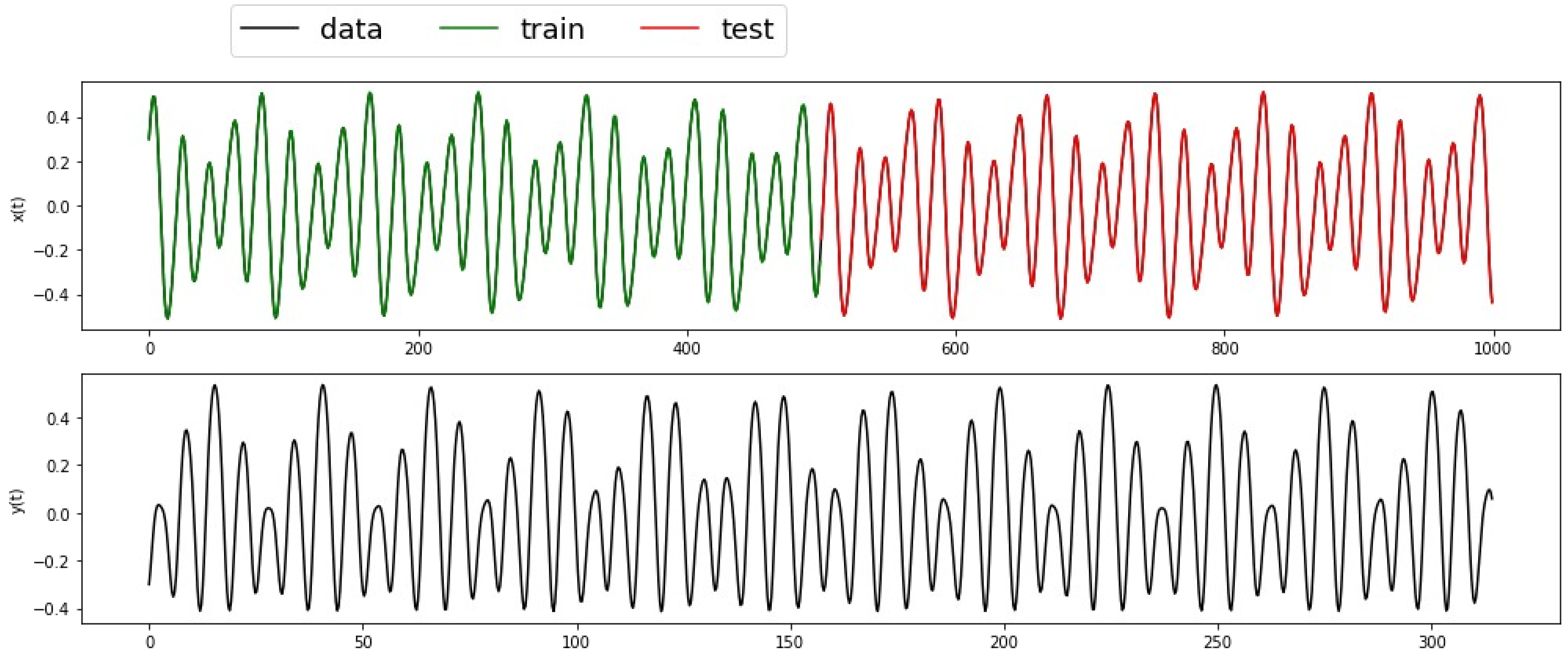
Fig. 7: Actual (red) and predicted (t+294ms) (black) tumor displacement curves for sample a) #440, b) #805, c) #642, and d) #817.

✔Extremely fast evaluation (light speed)

✔Integrated photonic circuit Neural Net

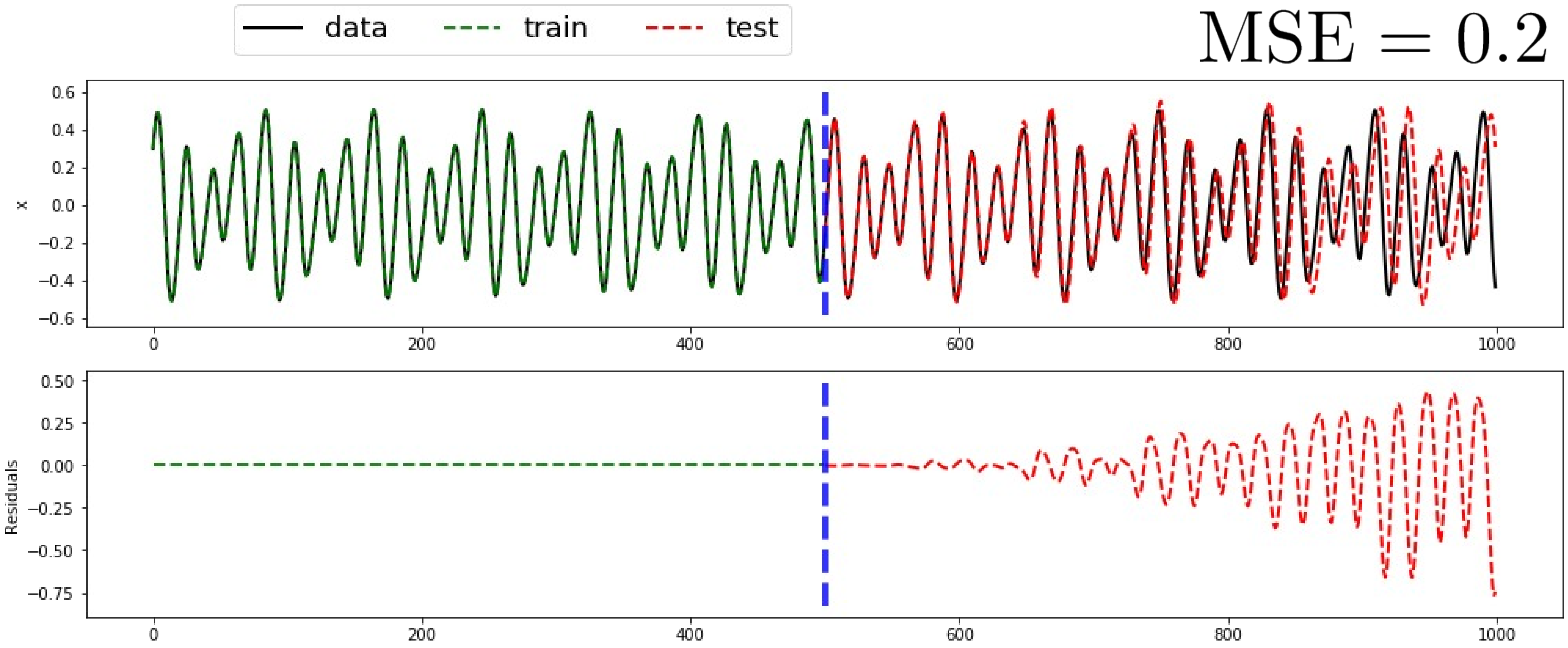
N. McDonald et al., "Analysis of an Ultra-short True Time Delay Line Optical Reservoir Computer," in Journal of Lightwave Technology.

Long-range forecasting



Prediction

$$\mathrm{MSE} = 0.2$$



We **do not** use the knowledge of the other signal

Inference

Using more signals as inputs drastically improves the prediction.

In that case we need to know the input signals for future times: **Inference**

The concept of **observers** that continuously provide partial information

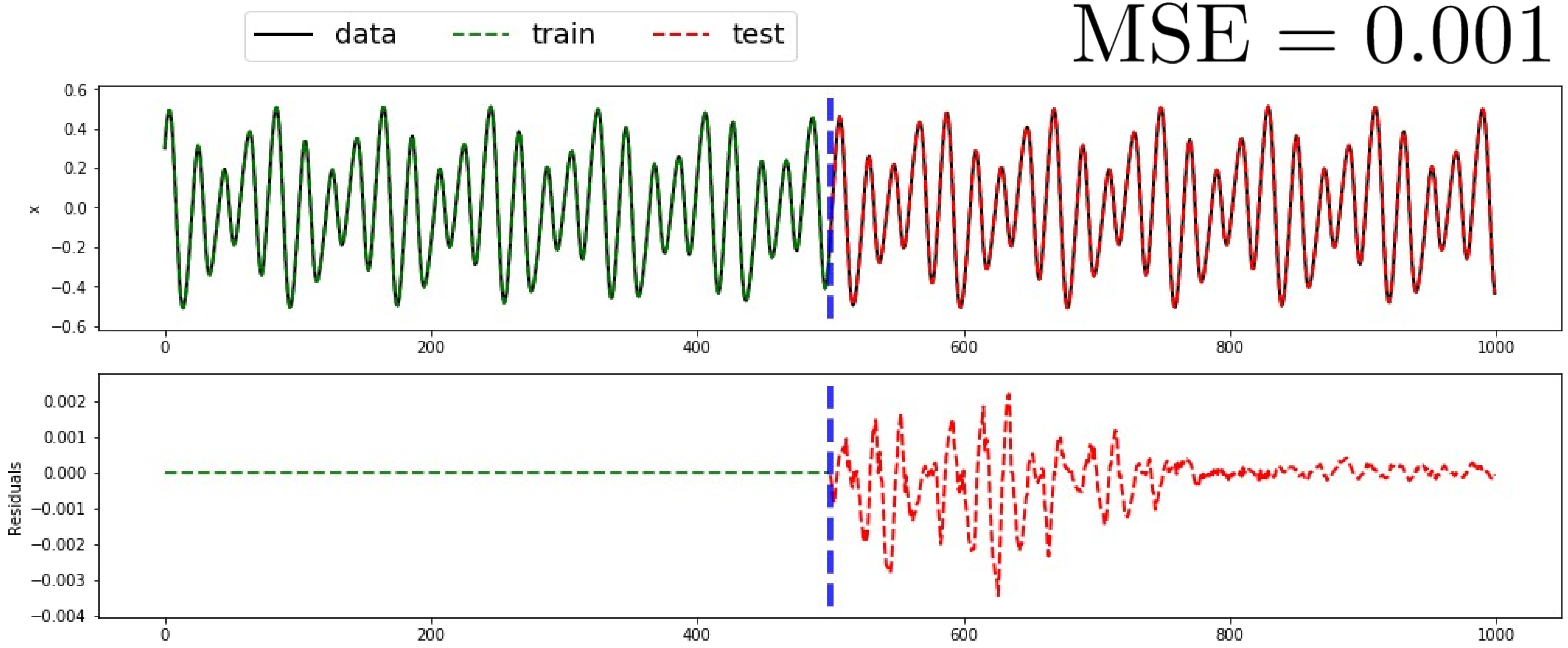
```
inputTrain = np.stack((ttrain,ytrain), axis=1)
inputTest = np.stack((ttest,ytest), axis=1)

esn = ESN(n_inputs = 2,
          n_outputs = 1,
          n_reservoir = 1000,
          spectral_radius = 1.5,
          sparsity=.2,
          noise =0.0001,
          random_state=42
         )

# Train and predict
xfit = esn.fit(inputTrain, xtrain)
xhat = prediction = esn.predict(inputTest)
```

Inference

$$\text{MSE} = 0.001$$



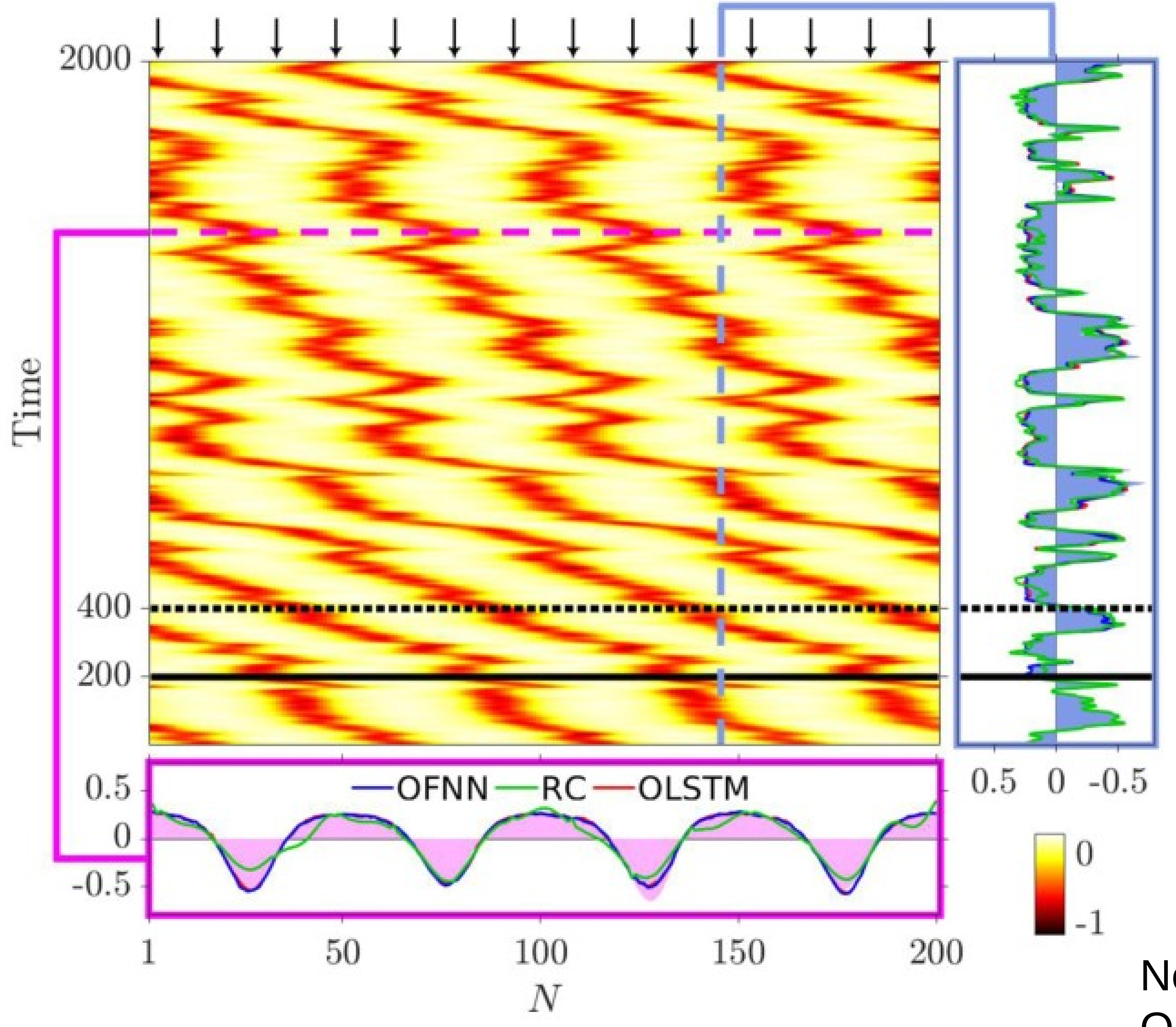
We **use** the knowledge of the other signal

Discussion

What is the RC learning?

- It is learning patterns and trends from the past sequential data
- Also, it is learning a real-time mapping between sequences (signals)

A very complex inference prediction



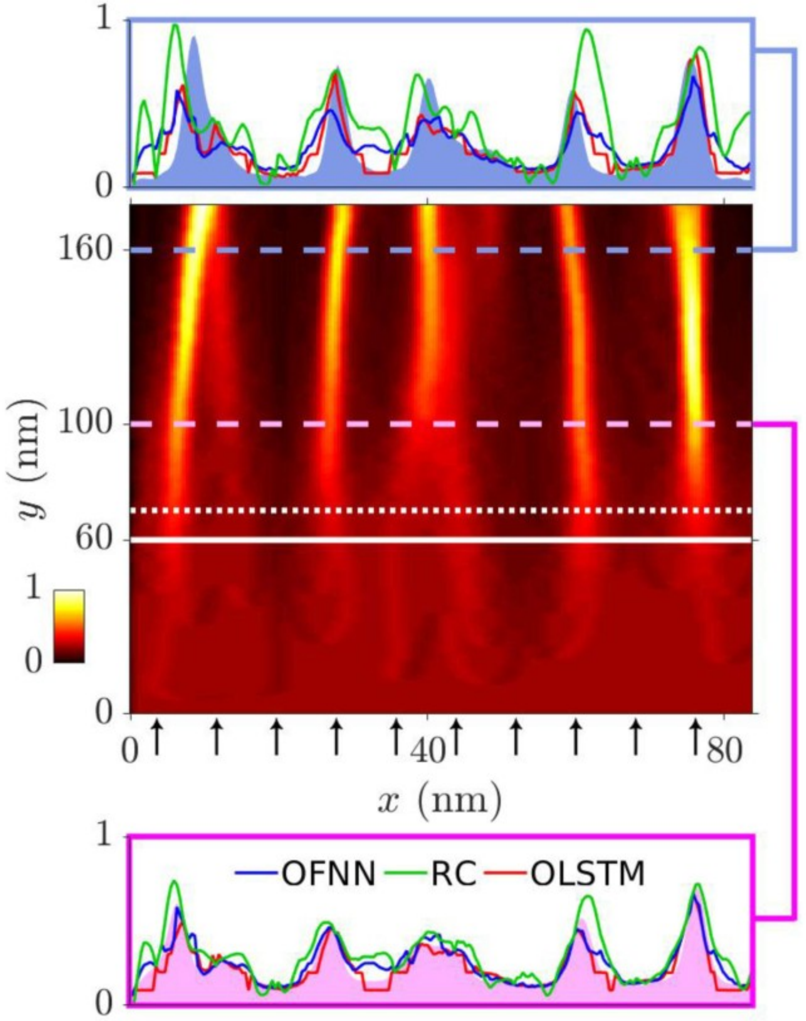
Neofotistos et al, Front. Phys. - Quantum Computing 7 (2019)

Research interest

Can we design **an observer-based photonic RC** with validation speed the *speed of light*?

It is loading ………

Are you interested in doing **research on RC**? Join us!



Neofotistos et al, Front. Phys. - Quantum Computing 7 (2019)

That’s it

- The training of the RC is fast, so for the homework you can use either the JupyterHub or your laptop.

Thank you!

References

- https://github.com/cknd/pyESN
- https://github.com/FilippoMB/Reservoir-Computing-framework-for-multivariate-time-series-classification
- https://towardsdatascience.com/gentle-introduction-to-echo-state-networks-af99e5373c68

1. H. Jaeger and H. Haas. Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication, Science **304** (2004)
2. Z. Lu, J. Pathak, B. Hunt, M. Girvan, R. Brockett, and E. Ott. Reservoir observers: Model-free inference of unmeasured variables in chaotic systems, Chaos **27** (2017)
3. G. N. Neofotistos, M. Mattheakis, G. Barmparis, J. Hitzanidi, G. P. Tsironis, and E. Kaxiras. Machine learning with observers predicts complex spatiotemporal behavior. Front. Phys. - Quantum Computing **7** (2019)