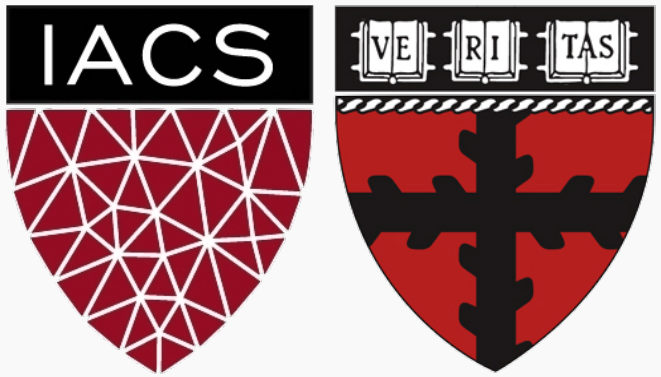
Advanced Section 2
Echo-State Reservoir Computing

Marios Mattheakis

CS109B Data Science 2

Pavlos Protopapas, Mark Glickman and Chris Tanner



Outline

- Exploding and Vanishing Gradients
- Reservoir Computing: An echo-state Recurrent Neural Network

Memory

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

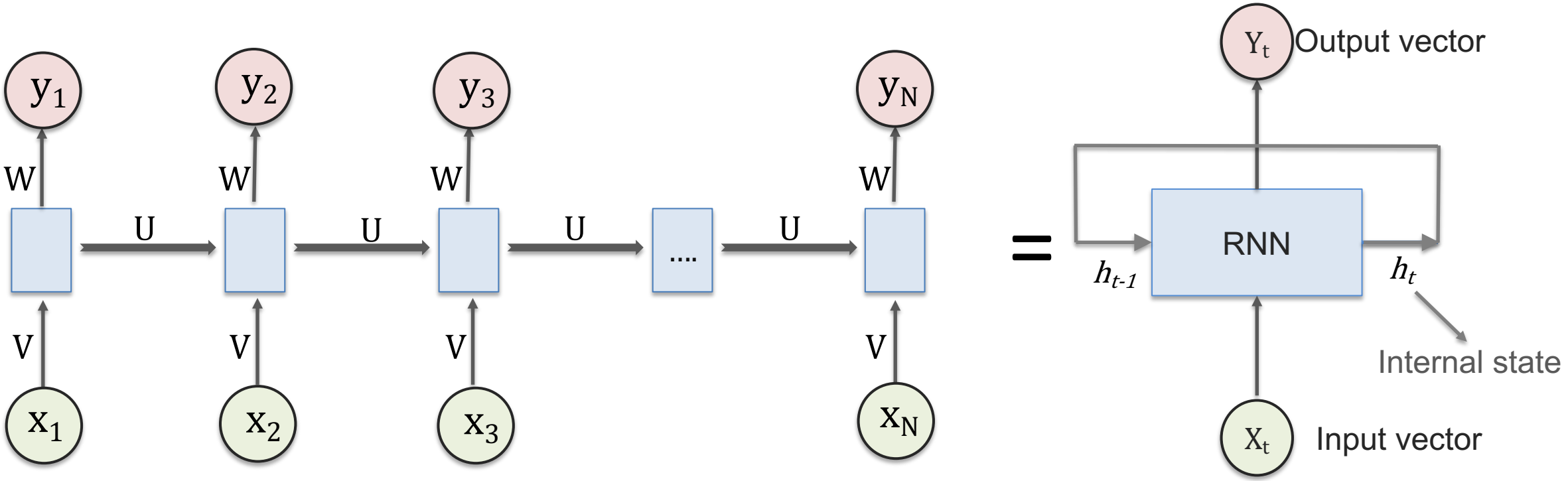
I like reading any time. **No memory is needed**
I like running before the lunch, but not after. **Memory is needed**

Artificial memory
The recurrent connections in RNNs allow dynamical behavior.
These recurrent hidden states provide **memory**.

Outline

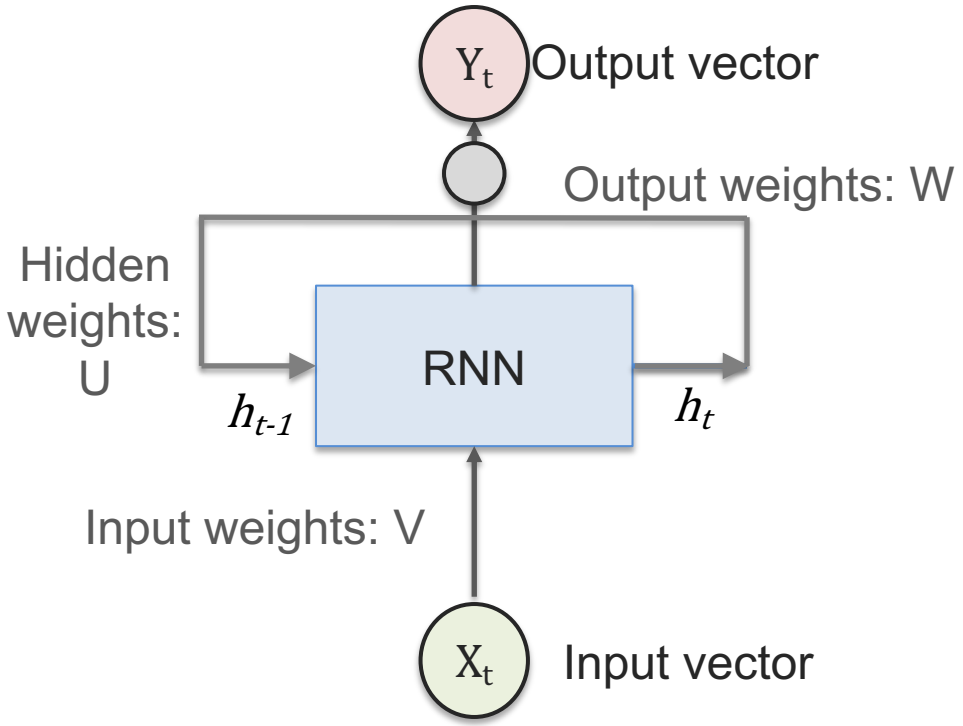
- **Exploding and Vanishing Gradients**
- Reservoir Computing: An echo-state RNN

Recurrent Neural Network (RNN)



At each time step the RNN is fed the current input and the previous hidden state.

State update



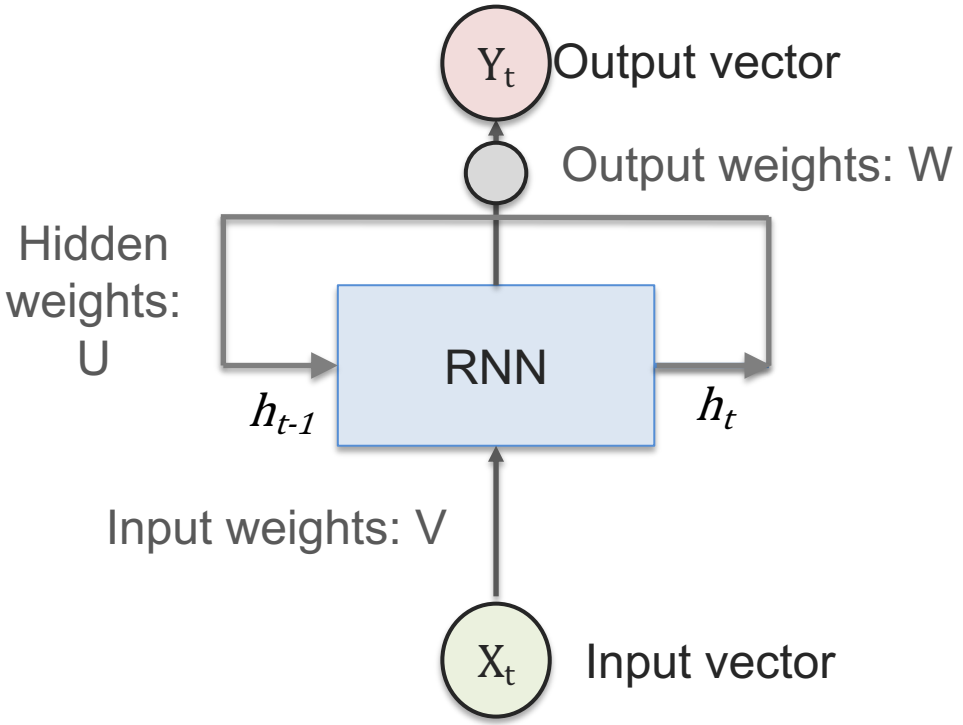
$$Y_t = \sigma\,(W h_t + \beta_2)$$

$$h_t = \tanh\,(U\,h_{t-1} + V X_t + \beta_1)$$

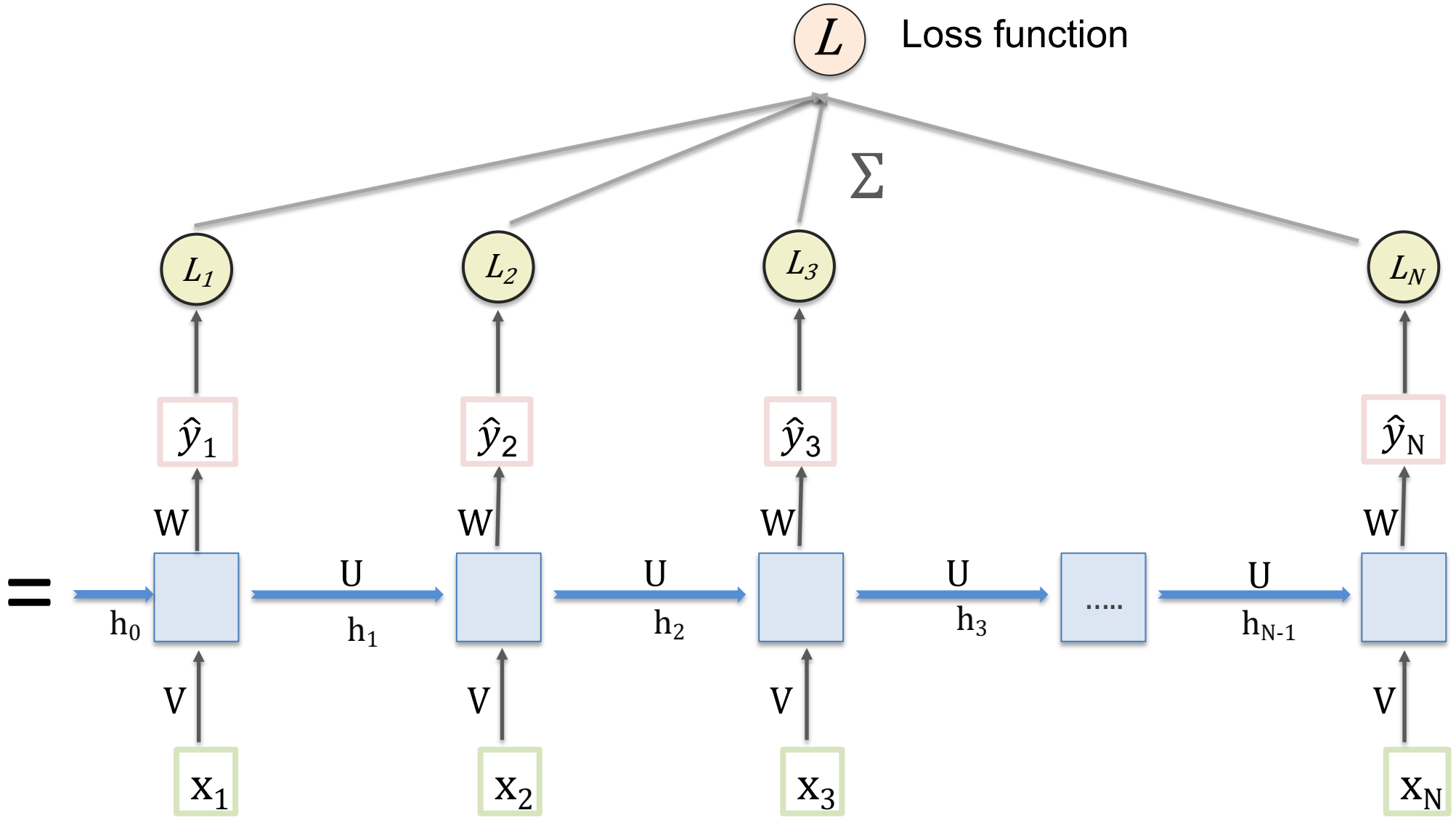
X_t Input vector

U, V and W are three different weight matrices learned during training

Train an RNN


Y_t
Output vector
Output weights: W
Hidden weights: U
RNN
h_{t-1}
h_t
Input weights: V
X_t
Input vector

Training in RNN: Forward Pass and constructing the loss function



RNN Formulation

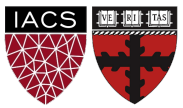
$$h_t = g_h\left(Vx_t + Uh_{t-1} + b'\right), \qquad g_h(\cdot): \text{Activation for hidden states}$$

$$\hat{y}_t = g_y\left(Wh_t + b\right) \qquad g_y(\cdot): \text{Activation of the output layer}$$

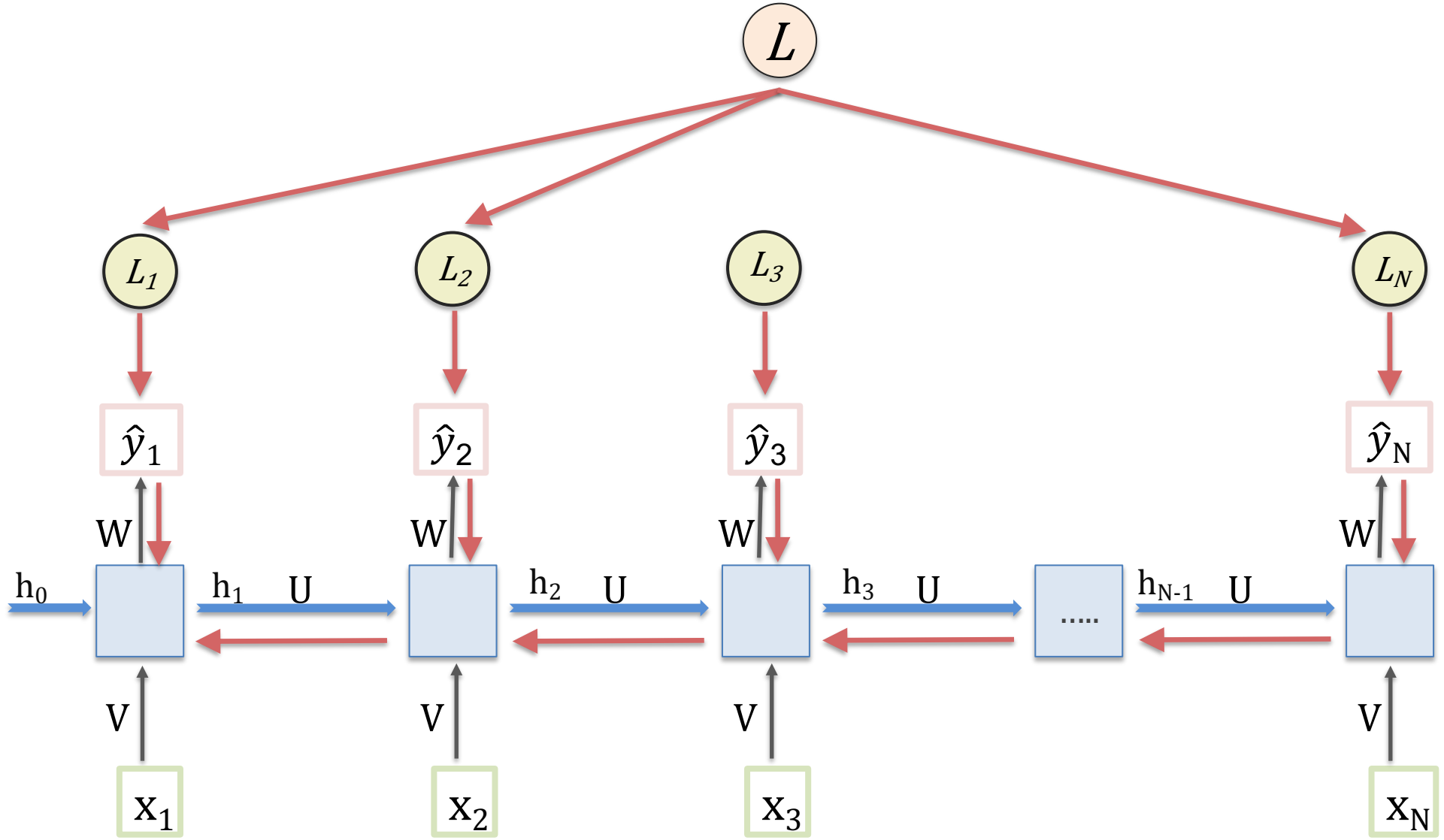
The total loss is the summation of all the individual losses in each time step

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

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



Training in RNN: Back propagation and weights update



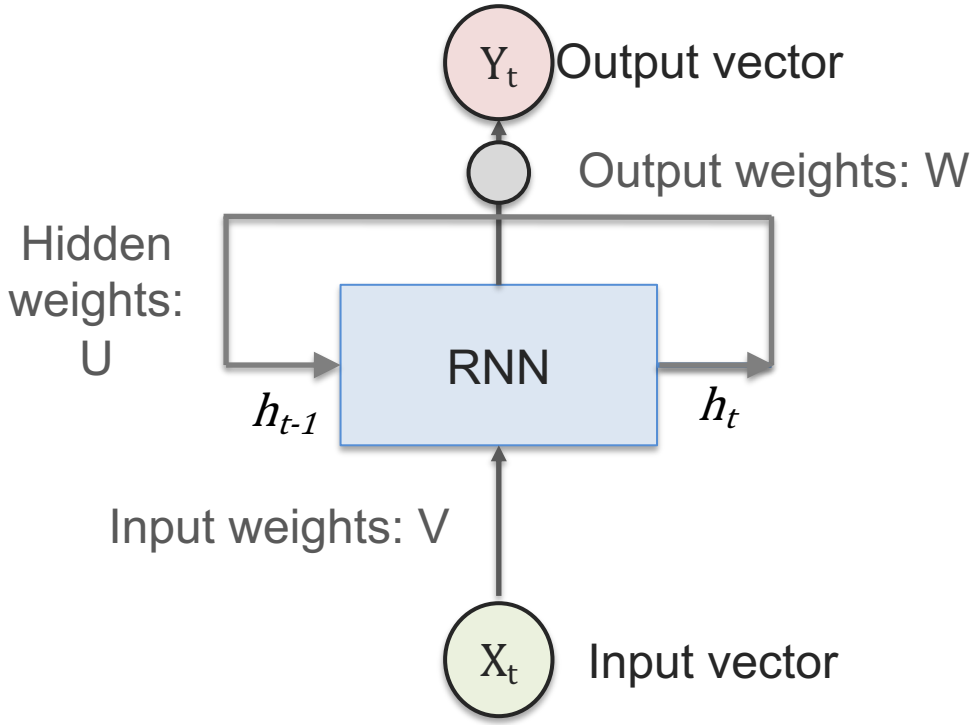
Getting deeper back in time, the gradients might become too strong or too weak

Optimization

Minimize the total loss with respect to 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

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

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

$$= \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial U}$$

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

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

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

$$= \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial U}$$

The *bad* term

Let's explore deeper ...

$$\sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial U}$$

The "bad" term

Chain rule again

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

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

$$\sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial U}$$

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:

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

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

$(\beta_U \ \beta_h)$ → < 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.

The weights are updated extremely slow

Exploding gradients:

The learning becomes unstable (overflow problem)

Vanishing & Exploding gradients

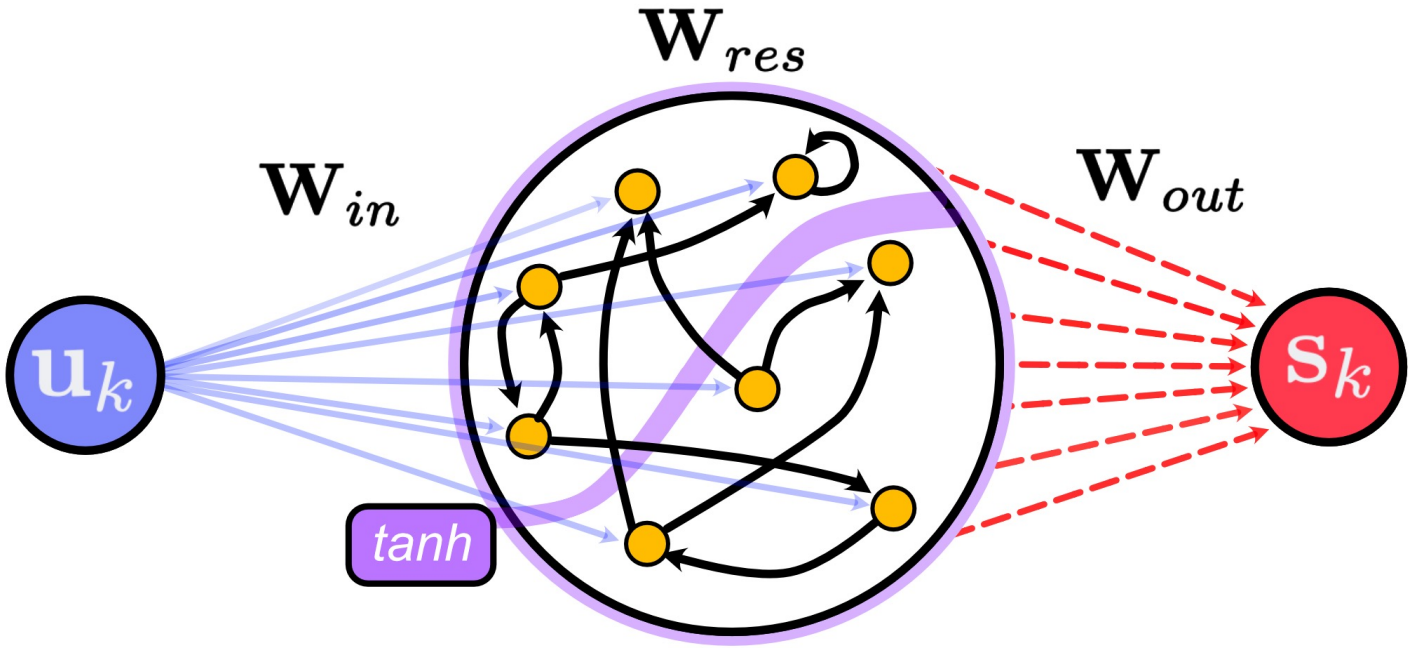
Solutions

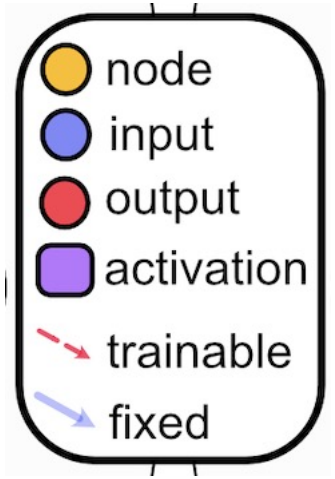
- Clipping gradients
- Special RNN with leaky units such as
- Gated Recurrent Units (GRU)
- Long-Short-Term-Memory (LSTM)
- **Echo-state RNNs**

Outline

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

Reservoir Computing





Extremely fast training

Forecast, analyze, and control complex dynamical systems

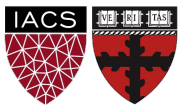
(weather, stock market etc...)

Echo-State RNNs

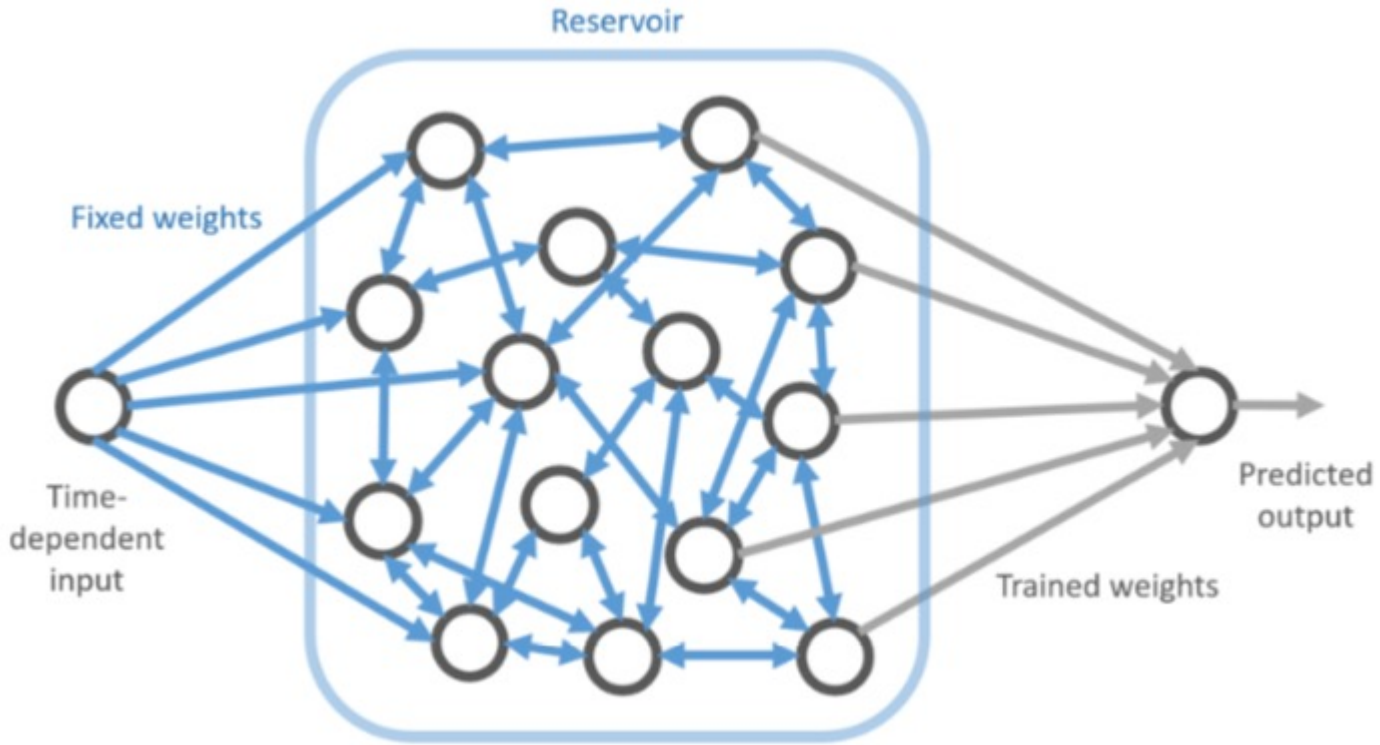
RNNs is extremely difficult to be trained:
Due to the recurrent and the input weights mapping to hidden states

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

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



RC Architecture

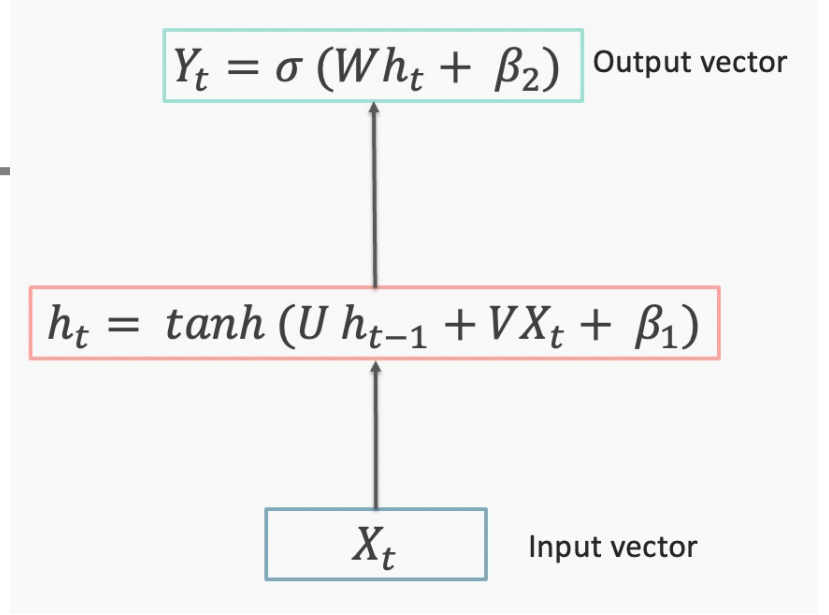


Input: An arbitrary length sequence input vector

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

$$\mathbf{W}_{\text{in}} \cdot \mathbf{u}_t + \mathbf{b}$$

Hidden recurrent dynamical nodes

$$\mathbf{h}_t = (1-\alpha)\,\mathbf{h}_{t-1} + \alpha f\,(\mathbf{W}_{\text{res}} \cdot \mathbf{h}_{t-1} + \mathbf{W}_{\text{in}} \cdot \mathbf{u}_t + \mathbf{b})$$

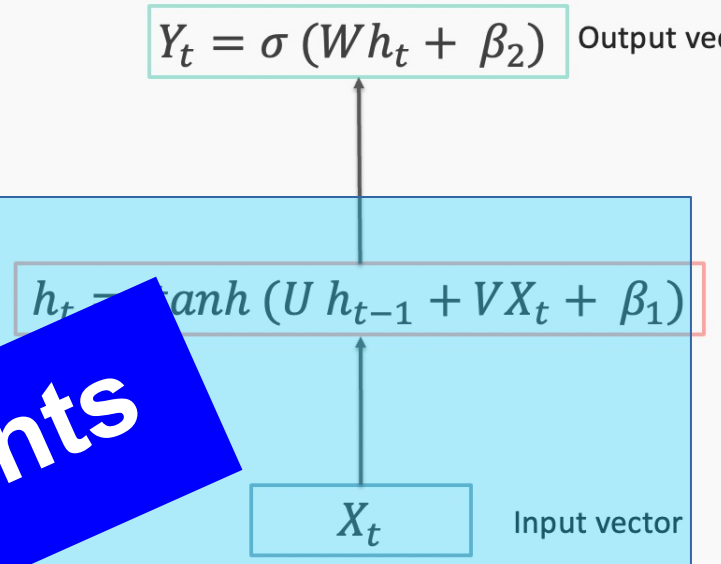
Output layer

$$\hat{\mathbf{y}}_t = g\,(\mathbf{W}_{\text{out}} \cdot \mathbf{h}_t) + \mathbf{b}'$$

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

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

RC Formulation



Input layer

$$\mathbf{W}_{\text{in}} \cdot \mathbf{u}_t + \mathbf{b}$$

Hidden recurrent dynamical node

$$\mathbf{h}_t = (1-\alpha)\,\mathbf{h}_{t-1} + \alpha\,(\mathbf{W}_{\text{res}} \cdot \mathbf{h}_{t-1} + \mathbf{W}_{\text{in}} \cdot \mathbf{u}_t + \mathbf{b})$$

Frozen Weights

Learnable Weights

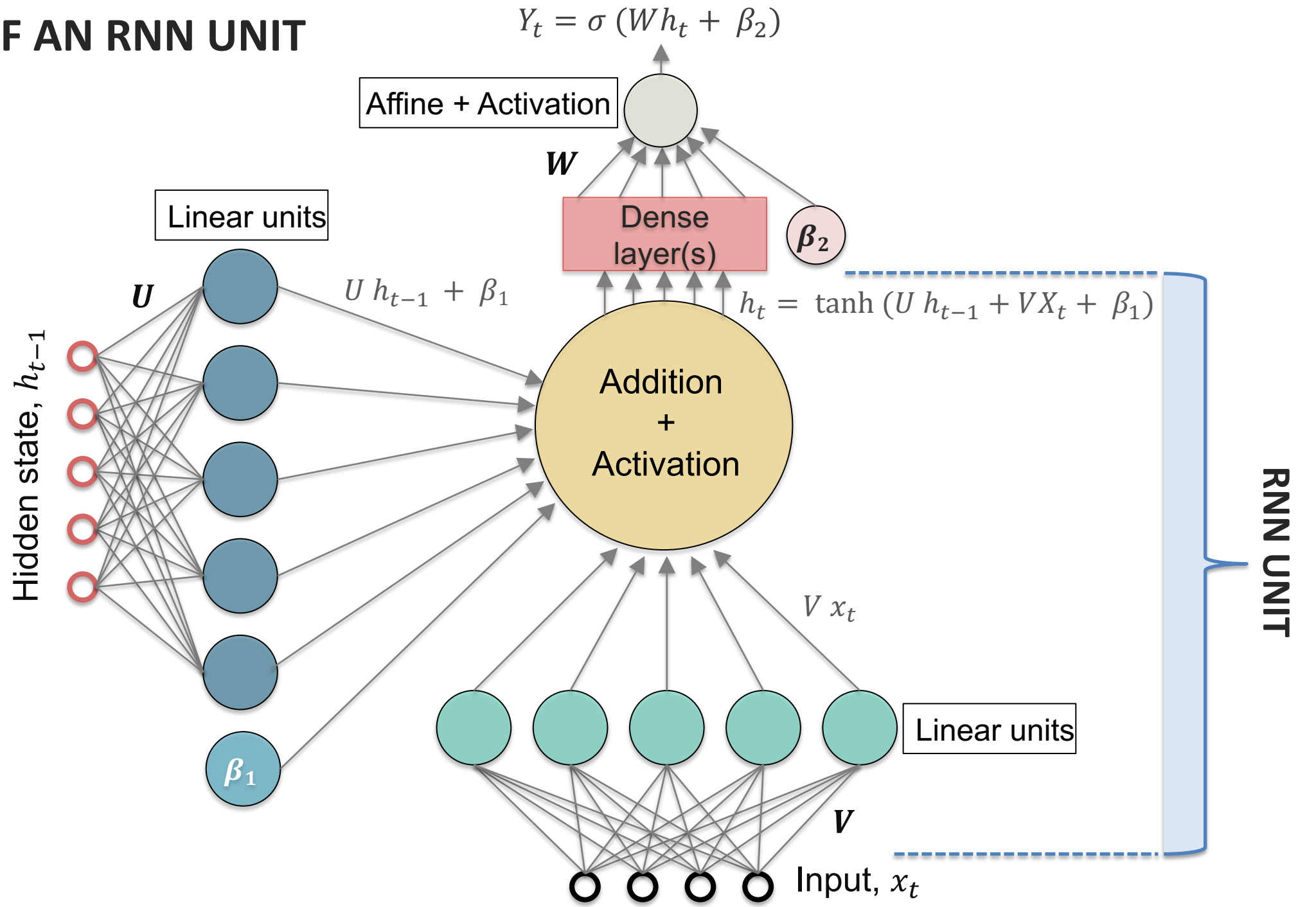
Output layer

$$= g\,(\mathbf{W}_{\text{out}} \cdot \mathbf{h}_t) + \mathbf{b}'$$

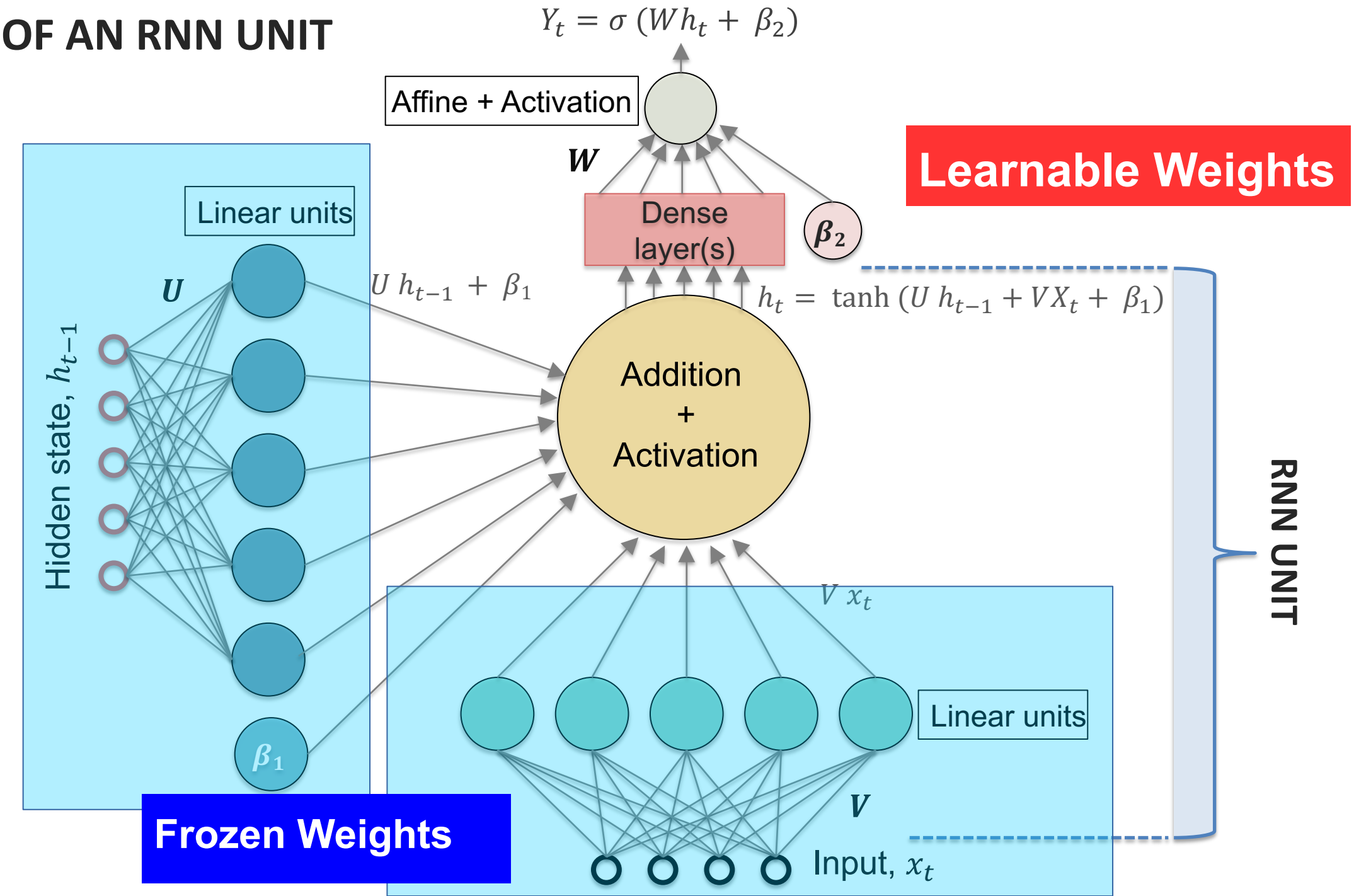
$$g() = \text{Identity}$$

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

ANATOMY OF AN RNN UNIT



ANATOMY OF AN RNN UNIT



Recurrent states

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

$$\frac{d\mathbf{h}(t)}{dt} = -\alpha\mathbf{h}(t) + \alpha f\left(\mathbf{W}_{\text{res}} \cdot \mathbf{h}(t) + \mathbf{W}_{\text{in}} \cdot \mathbf{u}(t) + \mathbf{b}\right)$$

$$\mathbf{h}(t + \Delta t) = (1 - \alpha\Delta t)\mathbf{h}(t) + \alpha\Delta t\ f\left(\mathbf{W}_{\text{res}} \cdot \mathbf{h}(t) + \mathbf{W}_{\text{in}} \cdot \mathbf{u}(t) + \mathbf{b}\right)$$

Leaking Unit:

Decay constant α

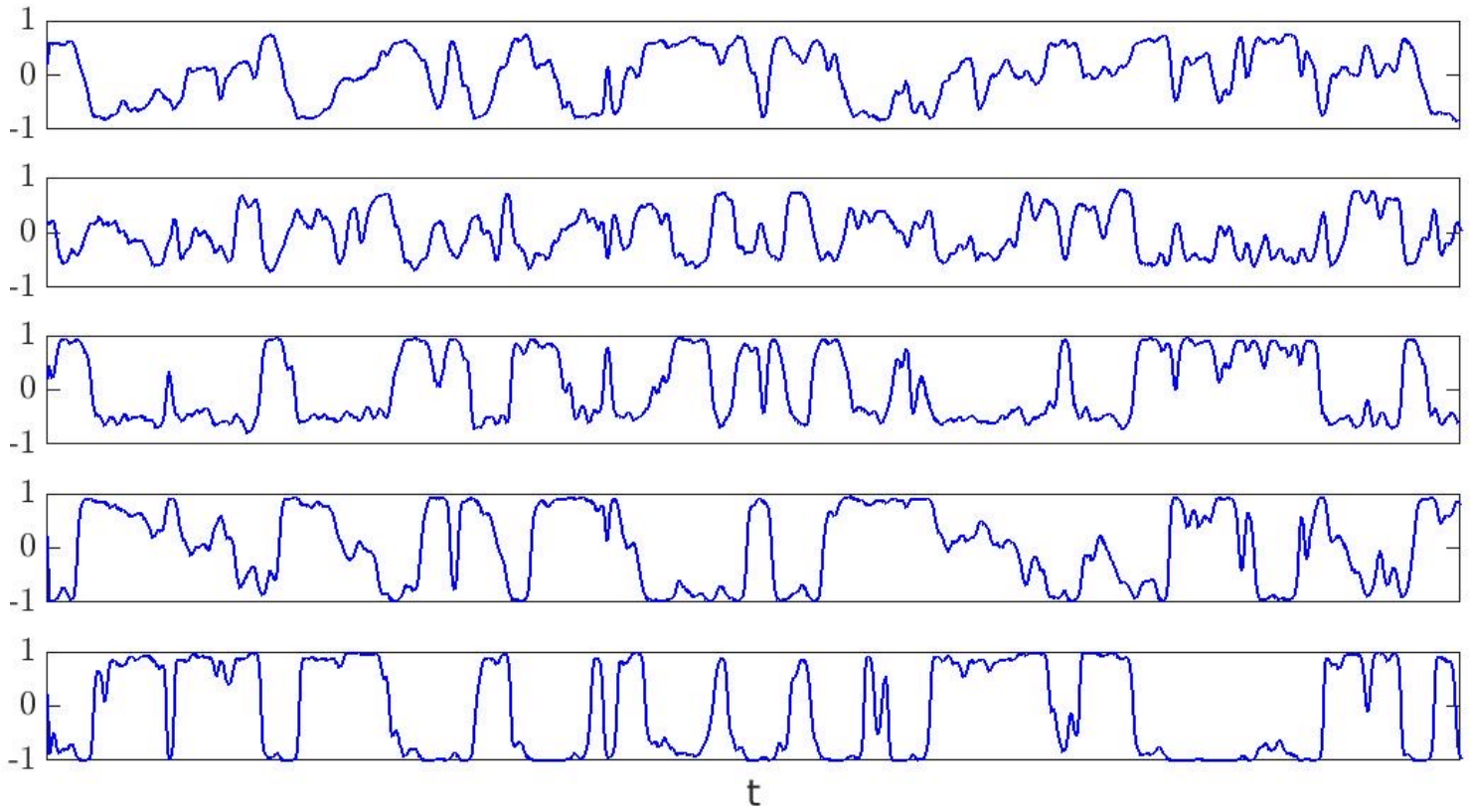
The nonlinear dynamics:

Activation function: tanh, sigmoid ...

Everything is fixed here

Recurrent states (features)

An example of some hidden states



Typically, we need many states (over 500)

Recurrent states

Generalization:

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

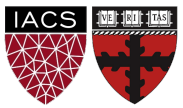
$$\mathbf{h}(t+\Delta t) = (1-\alpha\Delta t)\mathbf{h}(t) + \alpha\Delta t\ f\left(\mathbf{W}_{\text{res}}\cdot\mathbf{h}(t) + \mathbf{W}_{\text{in}}\cdot\mathbf{u}(t-\tau) + \mathbf{b}\right)$$

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** D 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

There is not any 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

$$\hat{\mathbf{y}}_t = g\left(\mathbf{W}_{\text{out}} \cdot \mathbf{h}_t\right) + \mathbf{b}'$$

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 (L_2) regularization

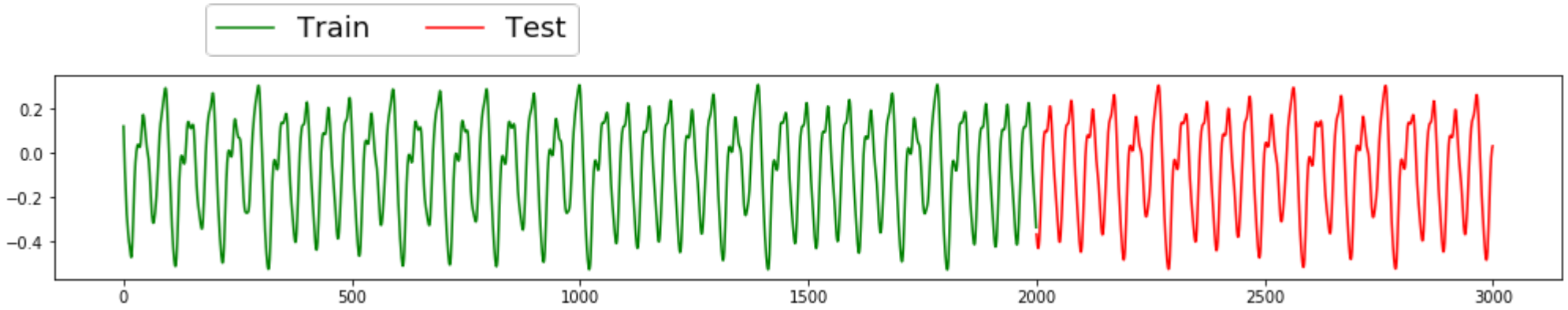
$$L = \sum_{t=0}^{T} \left(\hat{\mathbf{y}}_t - \mathbf{y}_t\right)^2 + \beta \ \text{Tr}\left[\mathbf{W}_{\text{out}} \mathbf{W}_{\text{out}}^T\right]$$

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

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

Example: Mackey Glass

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



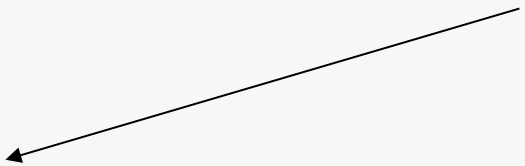
Use the python class pyESN taken by:

https://github.com/cknd/pyESN

Example: Mackey Glass

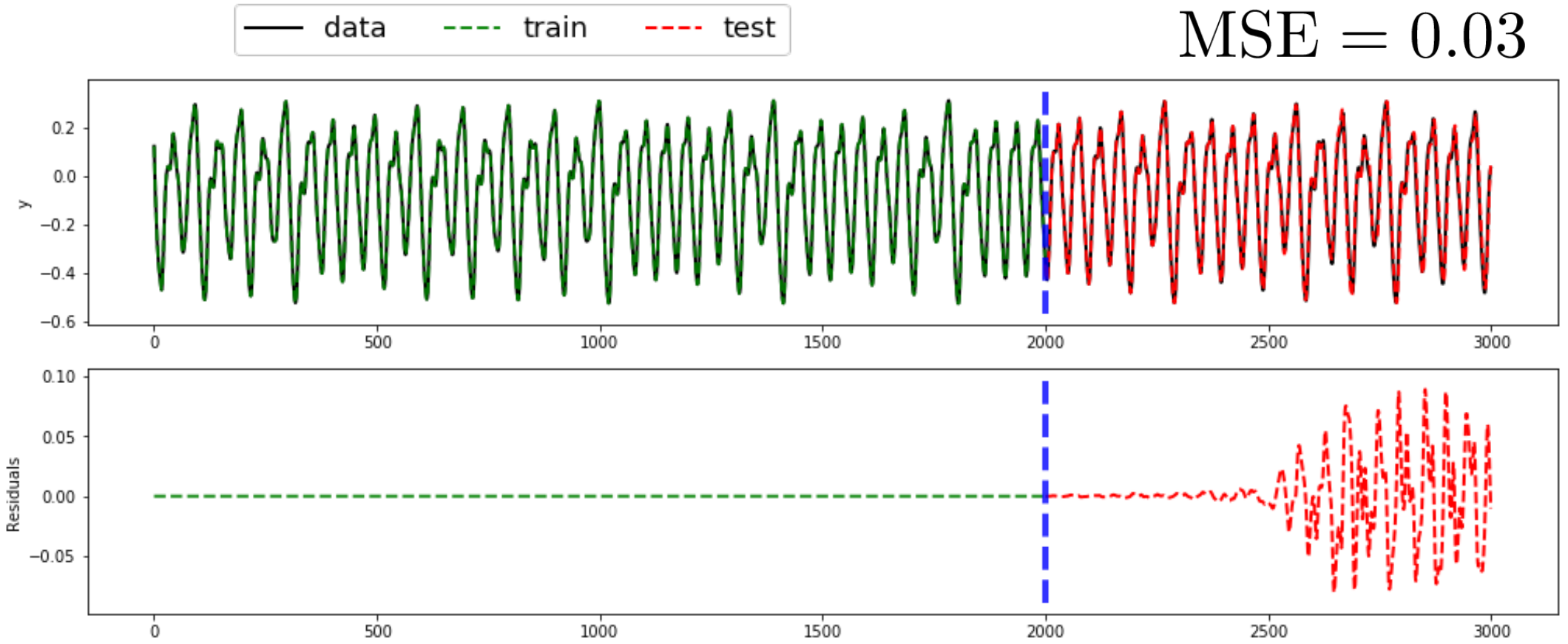
```
from pyESN import ESN
```

```
# 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**

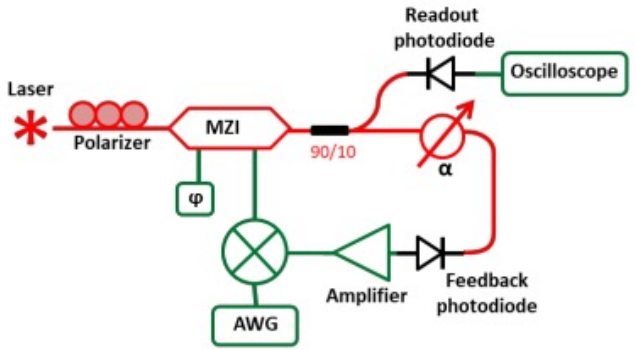


Fig. 1: Hardware schematic. Physical TDR consisted of Mach-Zehnder interferometer (MZI), arbitrary waveform generator (AWG), electrical amplifier, two photodetectors, fiber splitter (90/10), and tunable optical attenuator (α). Red lines denote optical components while green lines indicate electrical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

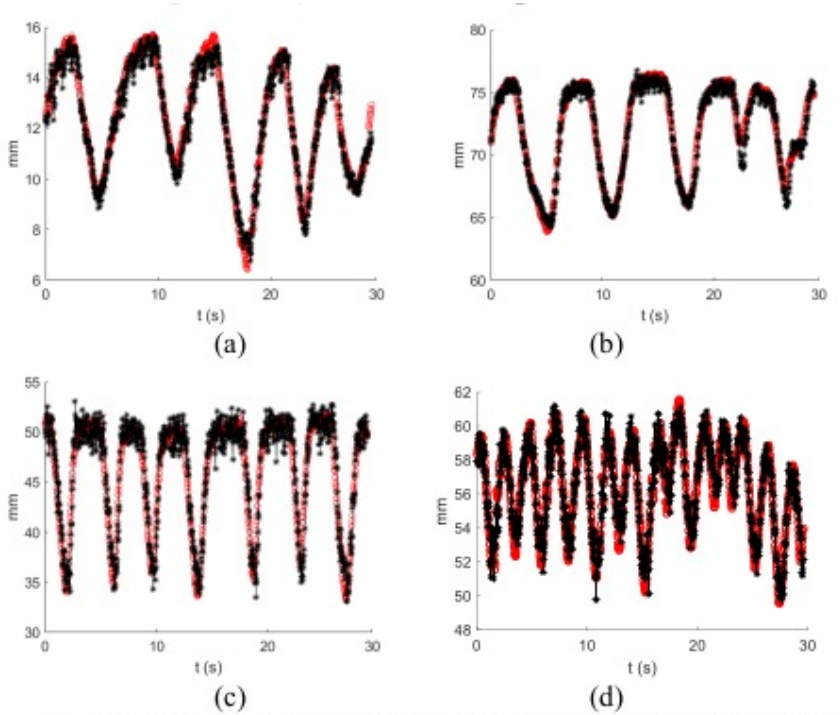
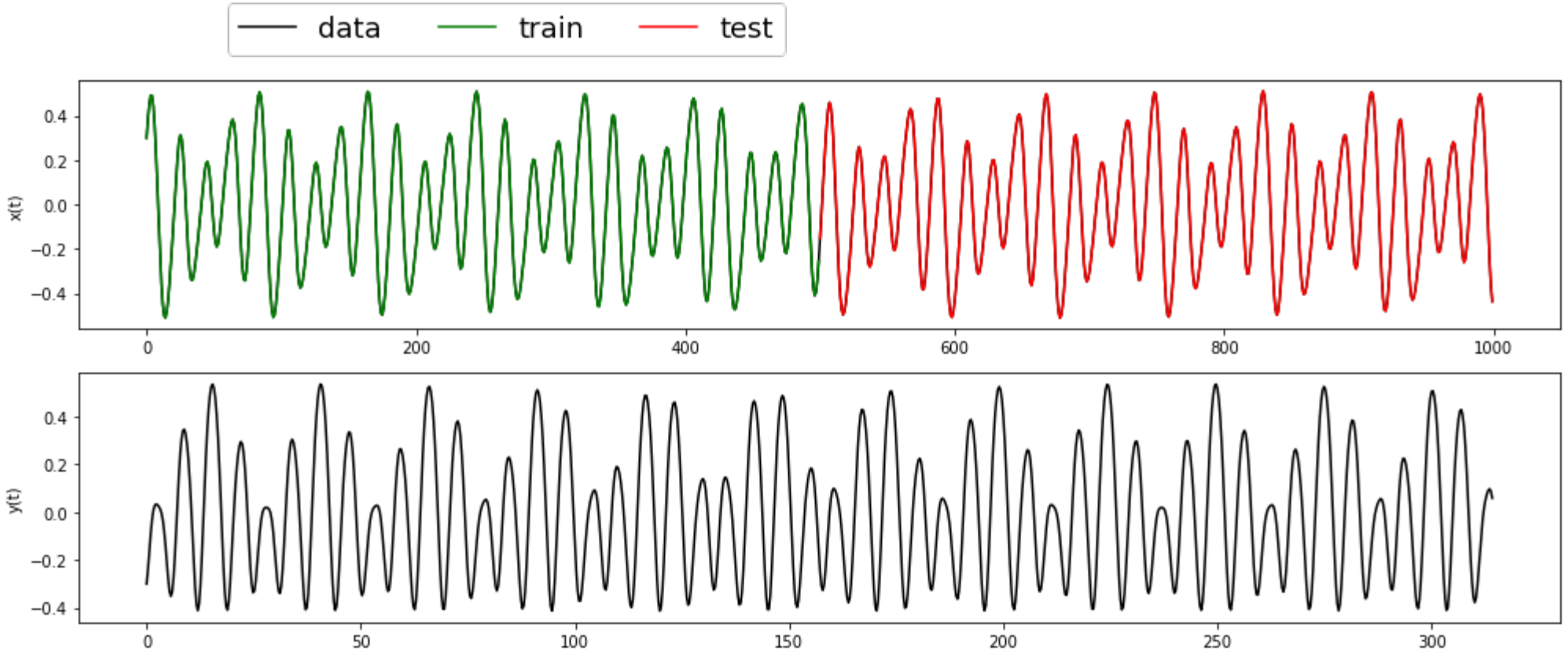


Fig. 7: Actual (red) and predicted (t+294ms) (black) tumor displacement curves for sample a) #440, b) #805, c) #642, and d) #817.

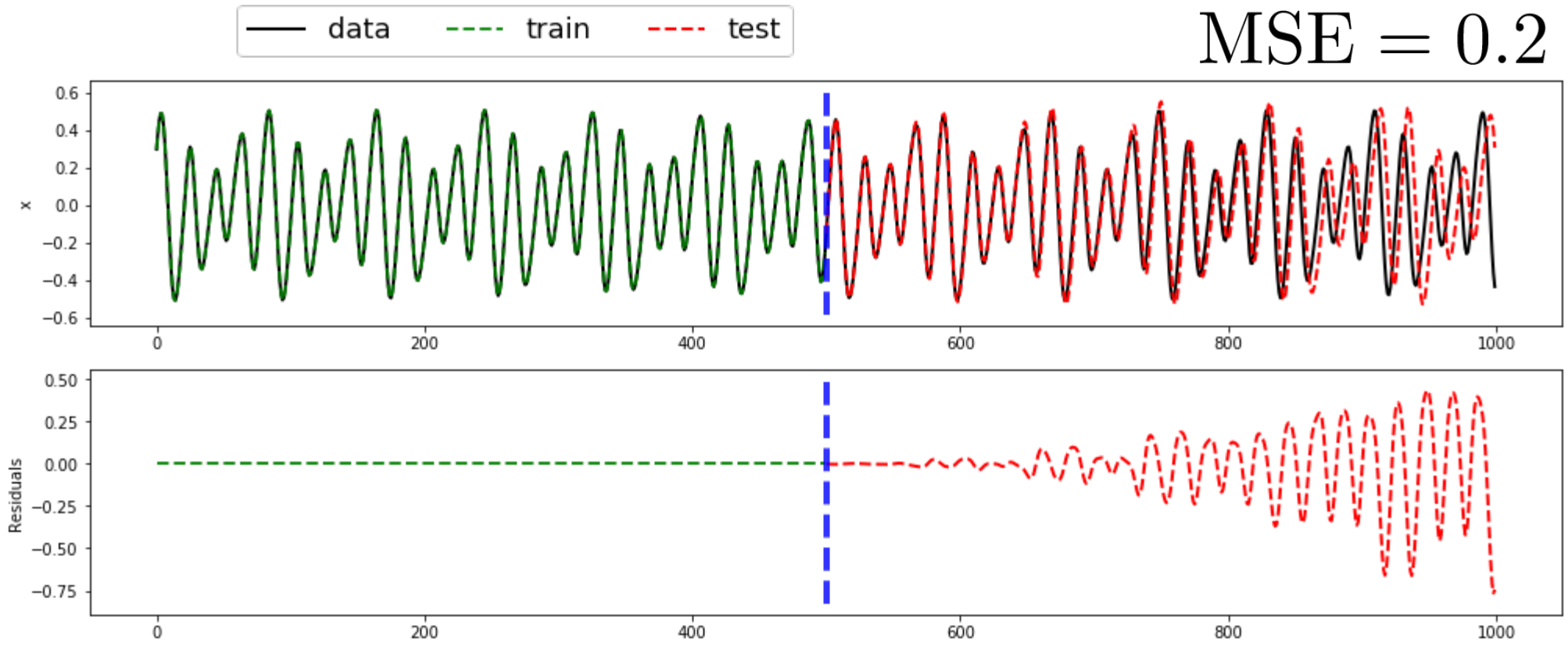
- ✓Extremely fast evaluation (light speed)
- ✓Integrated photonic circuit Neural Net

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



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

Inference

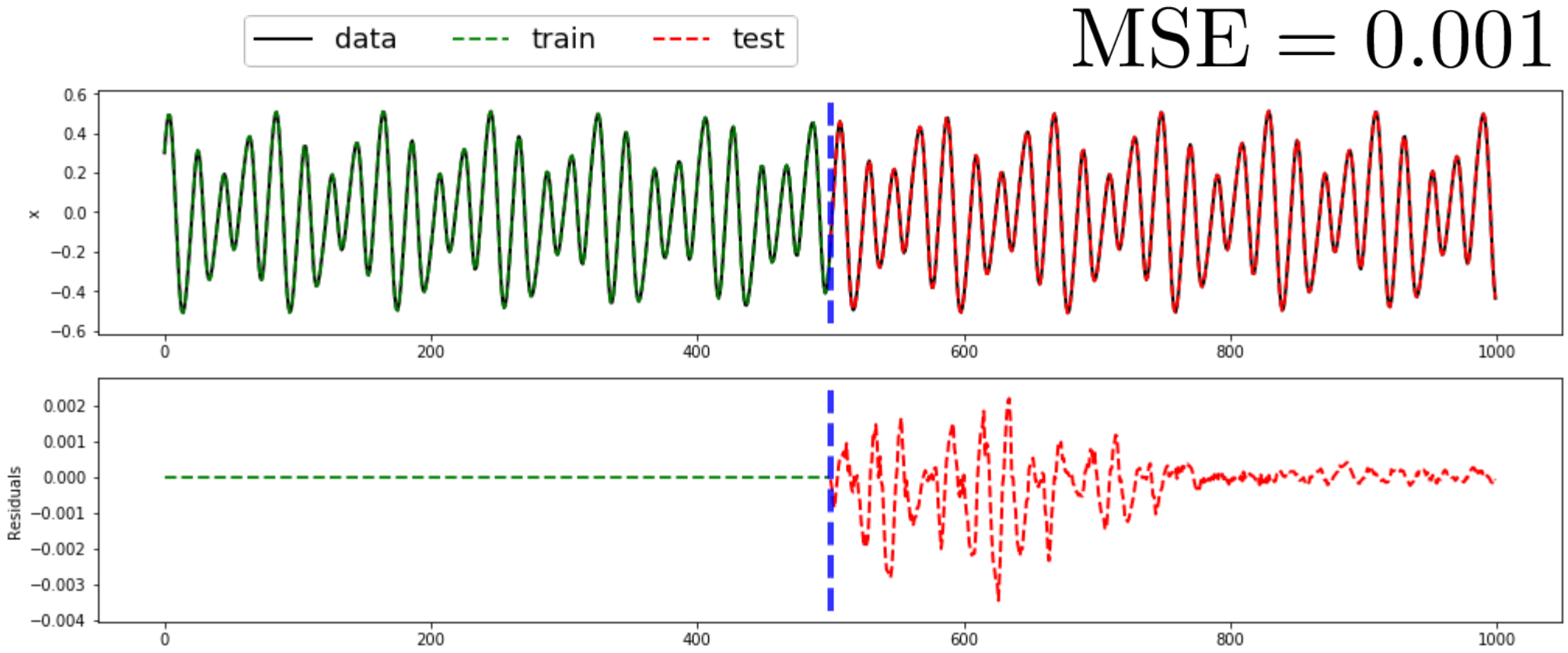
Using more signals as inputs drastically improves the prediction.

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

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

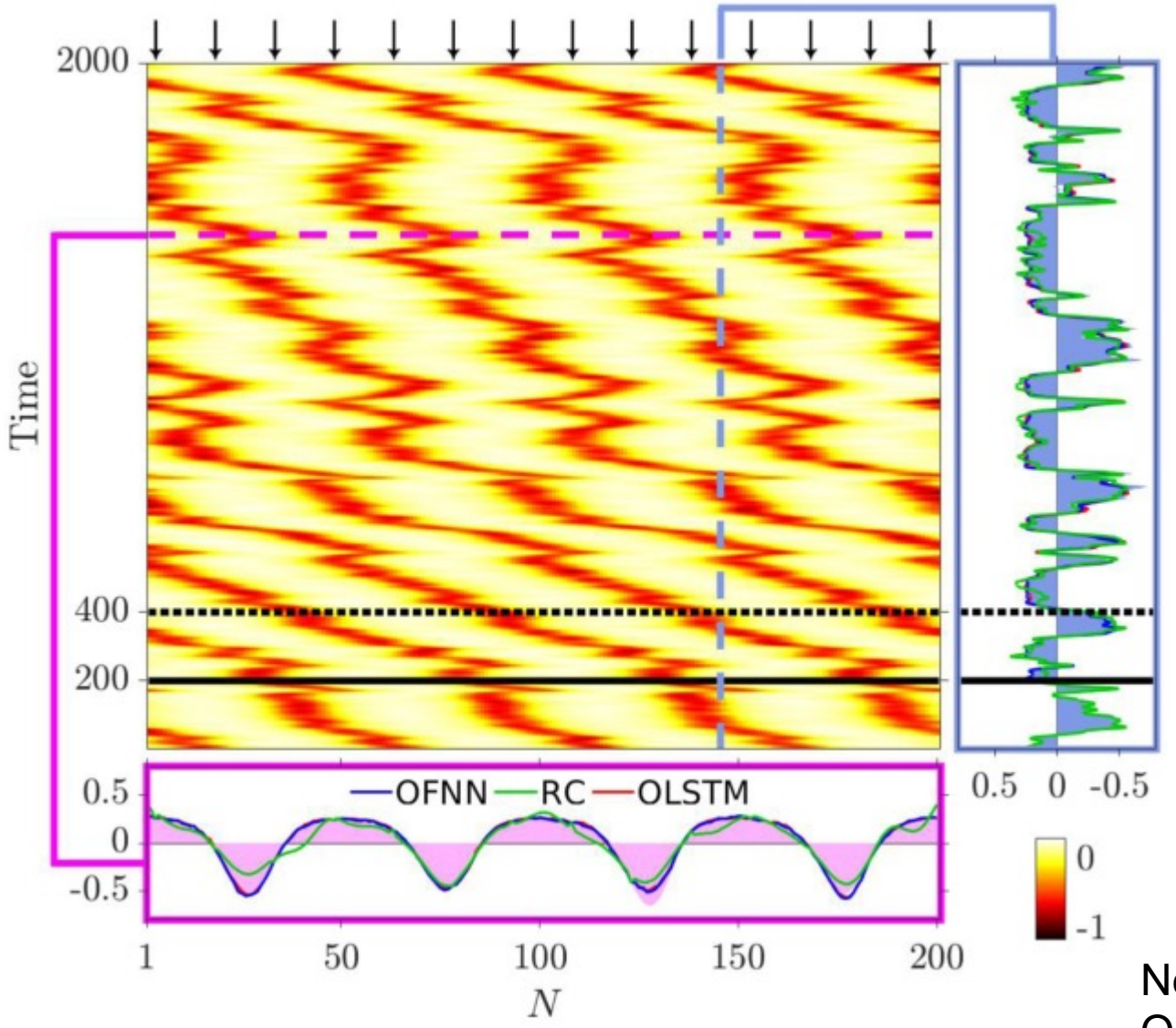
```
inputTrain = np.stack((ttrain,ytrain), axis=1)
inputTest = np.stack((ttest,ytest), axis=1)

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



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

A very complex inference prediction



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

Discussion

What is the RC learning?

- It is learning patterns and trends from the past sequential data
- It is learning a real-time mapping between sequences (signals)
- Hyper-parameters optimization, the expensive part
  - Grid search: Works in low dimensions (up to 2D) but becomes extremely expensive in higher dimension
  - Bayesian optimization. Very efficient method in high dimensions

RcTorch library

We are building an RC library using the PyTorch package

PyTorch

The library includes an efficient Bayesian optimization for the hyper-parameters.
Parallel calculation on GPUs

https://github.com/blindedjoy/RcTorch

https://pypi.org/project/rctorch/

H. Joy, M. Mattheakis, P. Protopapas, *an RC PyTorch unit*, in preparation

Homework

- The training of the RC is fast. For the homework you can use either the JupyterHub-GPU or your laptop (~30 minutes for running the entire notebook)

- You can use either the **pyESN** or the **RcTorch** library. It is up to you

- **For the pyESN:** Instructions are included in the homework notebook. pyESN library does not support GPUs

- **For the rctorch:** GPUs are supported.
  In the jupyterhub-GPU, open a terminal and run

```
pip3 install rctorch==0.67
```

Check out the notebooks with the examples for both libraries

That's it

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

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)
4. J. R. Maat, N. Gianniotis, and P. Protopapas. Efficient Optimization of Echo State Networks for Time Series Datasets. International Joint Conference on Neural Networks (IJCNN), IEEE (2018)
5. P. Verzelli, C. Alippi, L. Livi, and P. Tino. Input-to-State Representation in linear reservoirs dynamics, arXiv:2003.10585