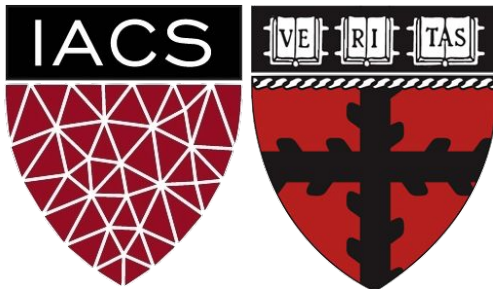


Advanced Section #4:  
Recurrent Neural Networks:  
Exploding, Vanishing Gradients & Reservoir Computing

AC 209B: Data Science 2

Marios Mattheakis

Pavlos Protopapas



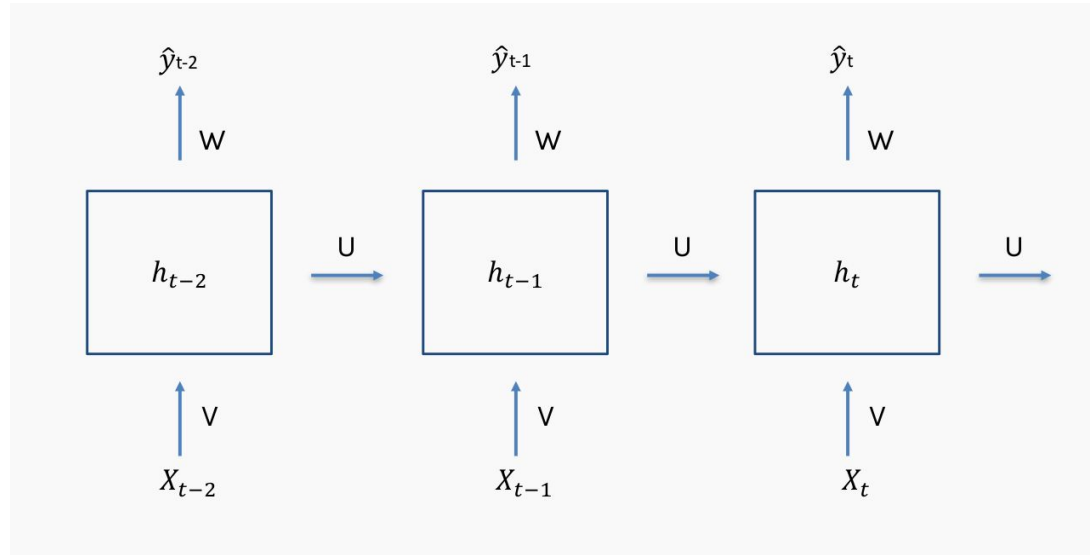
# Outline

- Exploding and Vanishing Gradients
- Reservoir Computing RNN

# Outline

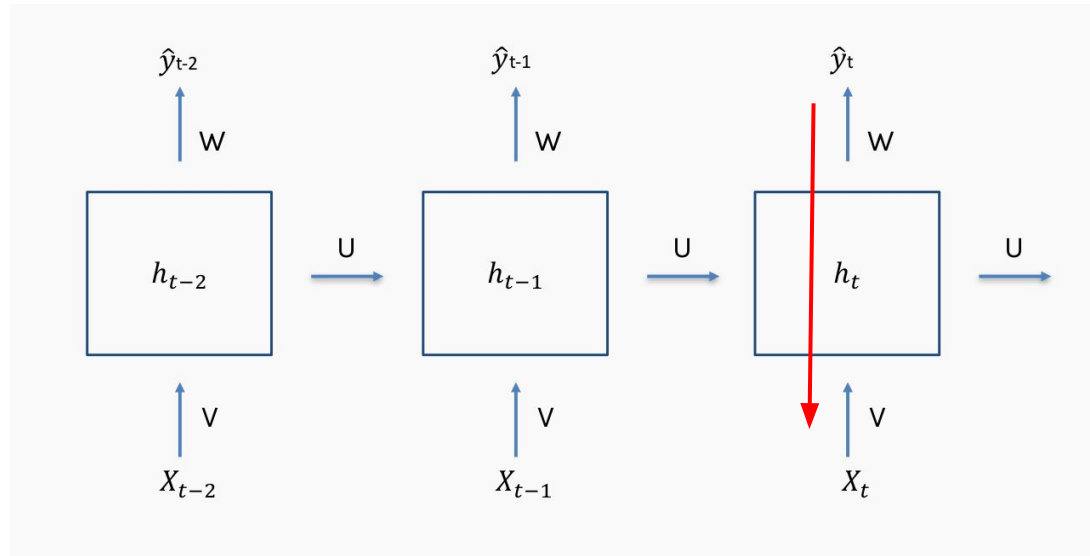
- Exploding and Vanishing Gradients
- Reservoir Computing RNN

# Training an RNN



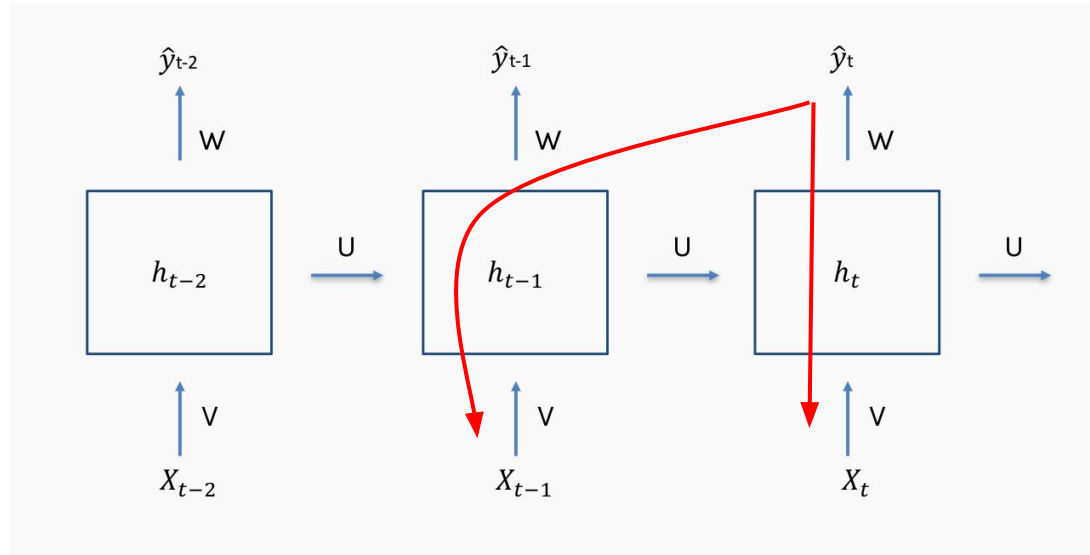
The training is extremely difficult due to the recurrent connections

# Training an RNN



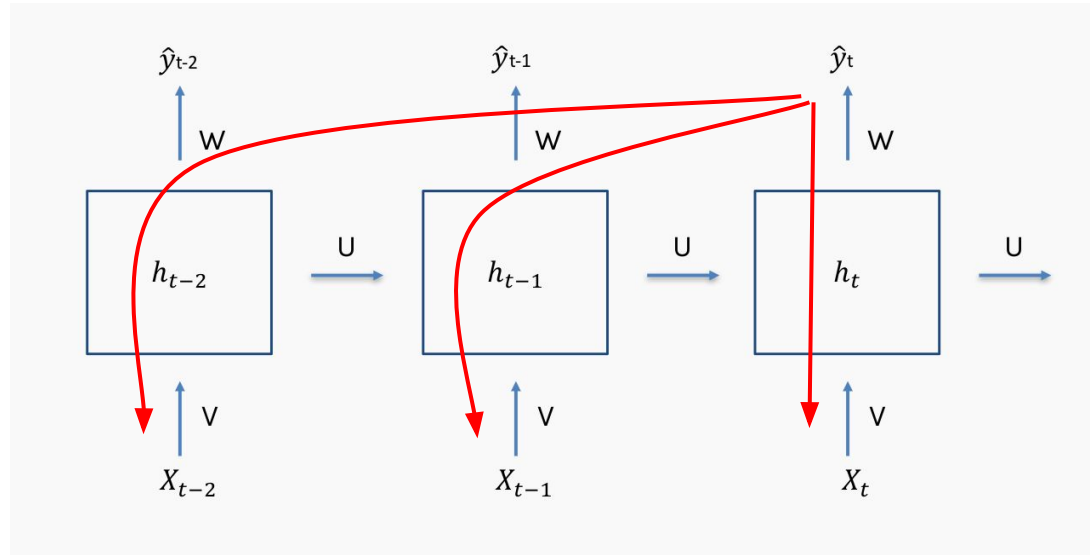
As we multiply the same matrix ( $h$ ) in each time in forward, we have to do the same in the back propagation

# Training an RNN



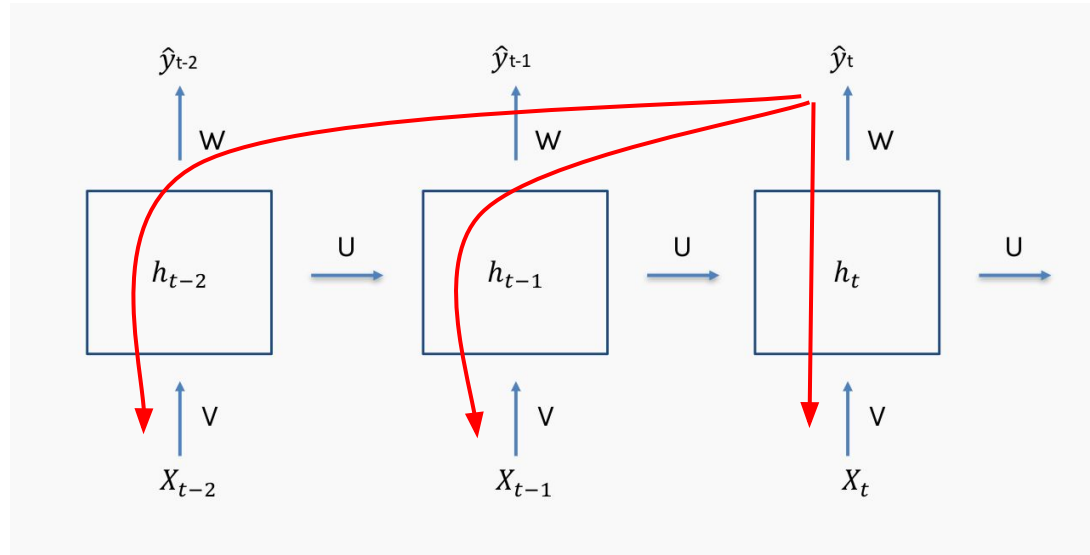
As we multiply the same matrix ( $h$ ) in each time in forward, we have to do the same in the back propagation

# Training an RNN



As we multiply the same matrix ( $h$ ) in each time in forward, we have to do the same in the back propagation

# Training an RNN



The first update of the gradients will be fine. But, as we are going further back in time the signals (gradients) might become too strong or too weak



# The vanishing/exploding gradient

- RNN formulation

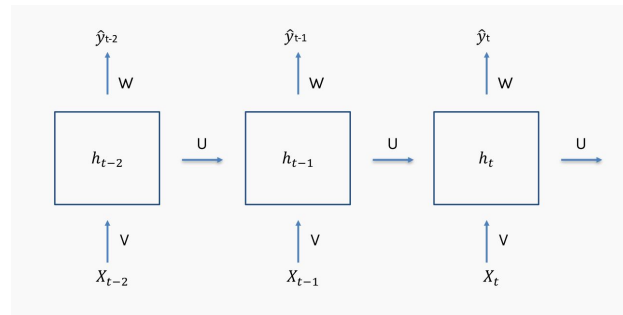
$$h_t = g_h (V x_t + U h_{t-1} + b'),$$

$$\hat{y}_t = g_y (W h_t + b)$$

- Total Loss is the sum of each loss in time

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

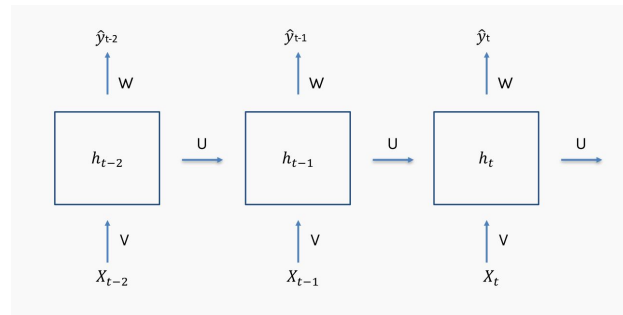
For the loss function wrt to the entire sequence in the interval  $t = [1, T]$  we need to sum up the loss function in all the time steps



# Loss minimization

- Minimize the total loss wrt the recurrent weights

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



- Let explore with chain rule just 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

$$\begin{aligned}\frac{dL_t}{dU} &= \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial U} \\ &= \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial U}\end{aligned}$$

# Chain rule for a single time step

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

# 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, it requires a huge chain rule application because it demands all the previous times steps

# Chain rule for a single time step

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

Let's explore deeper ...

The bad term:  $\partial h_t / \partial h_k$

- Chain rule again

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



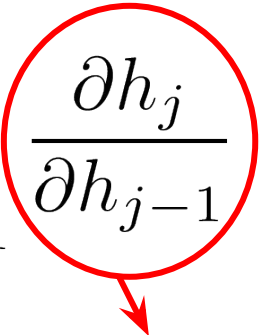
The bad term:  $\partial h_t / \partial h_k$

- Chain rule again

$$\begin{aligned} \frac{\partial h_t}{\partial h_k} &= \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_k}{\partial h_{k-1}} \\ &= \prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}} \end{aligned}$$

The bad term:  $\partial h_t / \partial h_k$

- Chain rule again

$$\begin{aligned} \frac{\partial h_t}{\partial h_k} &= \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_k}{\partial h_{k-1}} \\ &= \prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}} \end{aligned}$$


Jacobian matrix of the state to state transition.

The gradients is a product of Jacobian matrices (huge product)

The bad term:  $\partial h_t / \partial h_k$

- Chain rule again

$$\begin{aligned} \frac{\partial h_t}{\partial h_k} &= \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_k}{\partial h_{k-1}} \\ &= \prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}} \end{aligned}$$

Explore further each the Jacobian matrix (almost done)

# Explore the Jacobian matrices

- Remind:

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

# Explore the Jacobian matrices

- Remind:

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

- Hence:

$$\frac{\partial h_j}{\partial h_{j-1}} = U^T g'$$

# Explore the Jacobian matrices

- Remind:

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

- Hence:

$$\frac{\partial h_j}{\partial h_{j-1}} = U^T g'$$

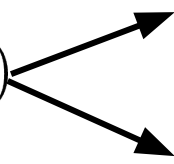
- The Norm:

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

Where  $\beta$ s are the values of the norms (numbers)

# The vanishing/exploding gradient

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

$(\beta_U \beta_h)$    $> 1$ : Exploding gradients  
 $< 1$ : Vanishing vanishing

It happens very fast as the time increases

# What happens?

- Vanishing gradient:  
Make it difficult to know in which direction to move for improving the loss function
  
- Exploding gradient:  
The learning becomes unstable



# Solutions

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

# Outline

- Exploding and Vanishing Gradients
- Reservoir Computing RNN

# Reservoir Computing

- The recurrent weights from hidden to hidden state ( $h_{t-1}$  to  $h_t$ ) and the input weights mapping to hidden ( $x_t$  to  $h_t$ ) are extremely difficult to be trained

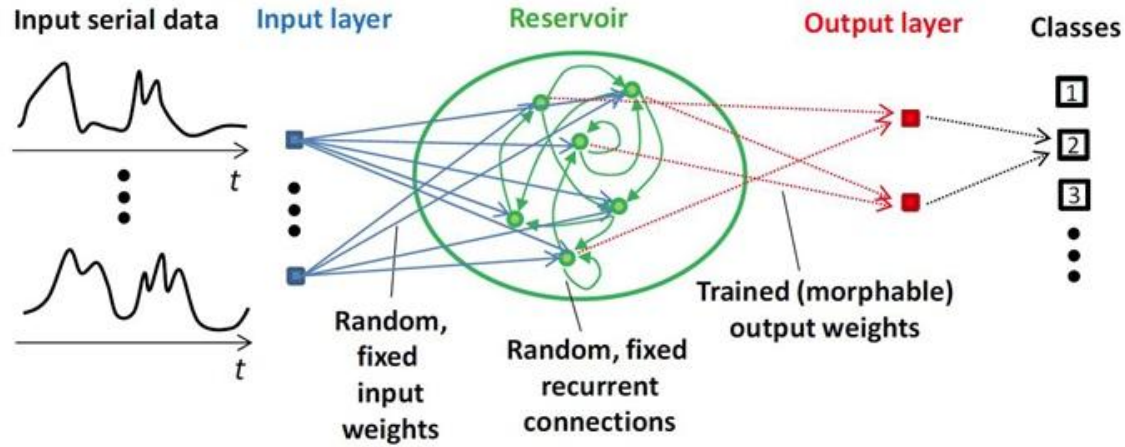
# Reservoir Computing

- The recurrent weights from hidden to hidden state ( $h_{t-1}$  to  $h_t$ ) and the input weights mapping to hidden ( $x_t$  to  $h_t$ ) are extremely difficult to be trained
- An approach to avoid this difficulty is to *fix* the recurrent and input weights and learn only the output weights: *Echo State Network* (ESN)

# Reservoir Computing

- The recurrent weights from hidden to hidden state ( $h_{t-1}$  to  $h_t$ ) and the input weights mapping to hidden ( $x_t$  to  $h_t$ ) are extremely difficult to be trained
- An approach to avoid this difficulty is to *fix* the recurrent and input weights and learn only the output weights: *Echo State Network* (ESN)
- The hidden units form a *Reservoir* of temporal features that capture different aspects from of the history inputs

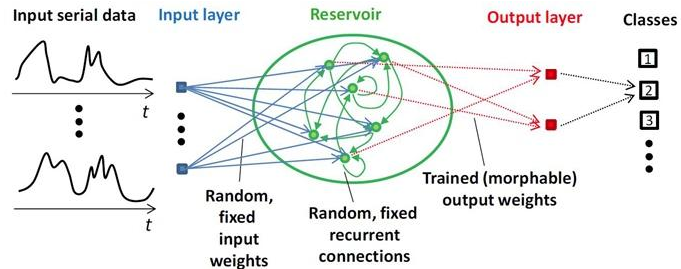
# Insight description of RC



- Takes an arbitrary length sequence input vector ( $\mathbf{u}_t$ )
- Mapping it into a high-dimensional feature space (recurrent state  $h_t$ )
- A linear predictor (lin. regression) is applied to find ( $\hat{\mathbf{y}}_t$ )

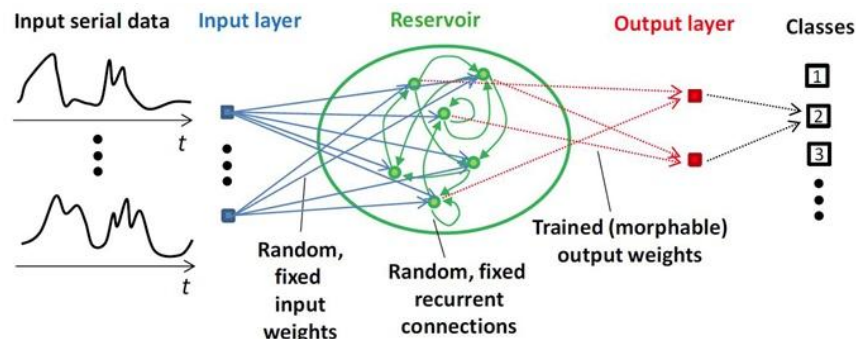
# Speed up the training

- We essentially train only the output weights:
  - Drastically speeds up the training
  - Great advantage of RC
- Set and fix the input and recurrent weights to represent a rich history:
  - Recurrent states as dynamical systems near to the stability
  - Stability means Jacobians close to one
  - Leaky hidden units that partially remember the previous state
    - ✓ Avoid exploding/vanishing gradients
    - ✓ No need of training



# RC formulation

- M given inputs:  $\mathbf{u}(t) \in \mathbb{R}^M$
- P desired outputs:  $\hat{\mathbf{y}}(t) \in \mathbb{R}^P$



- In the training period T with training set:  $\{\mathbf{u}(t), \mathbf{y}(t)\}_{t=1}^T$
- Suppose a reservoir with N hidden recurrent states  $\mathbf{r}(t) \in \mathbb{R}^N$
- Looking for a functional relationship:  $\hat{\mathbf{y}}(\mathbf{u}(t))$



# Weights and connections

- Reservoir nodes with recurrent connections:  $W_{\text{res}} \in \mathbb{R}^{N \times N}$

# Weights and connections

- Reservoir nodes with recurrent connections:  $W_{\text{res}} \in \mathbb{R}^{N \times N}$
- The inputs are connected to reservoir nodes through a linear layer:

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

$$W_{\text{in}} \in \mathbb{R}^{M \times N}$$

$$\mathbf{b} \in \mathbb{R}^N$$

# Weights and connections

- Reservoir nodes with recurrent connections:  $W_{\text{res}} \in \mathbb{R}^{N \times N}$
- The inputs are connected to reservoir nodes through a linear layer:

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

$$W_{\text{in}} \in \mathbb{R}^{M \times N}$$
$$\mathbf{b} \in \mathbb{R}^N$$

- The reservoir nodes are connected with output with a linear layer:

$$\hat{\mathbf{y}}(t) = W_{\text{out}} \mathbf{r}(t) + \mathbf{c}$$

$$W_{\text{out}} \in \mathbb{R}^{N \times P}$$
$$\mathbf{c} \in \mathbb{R}^P$$

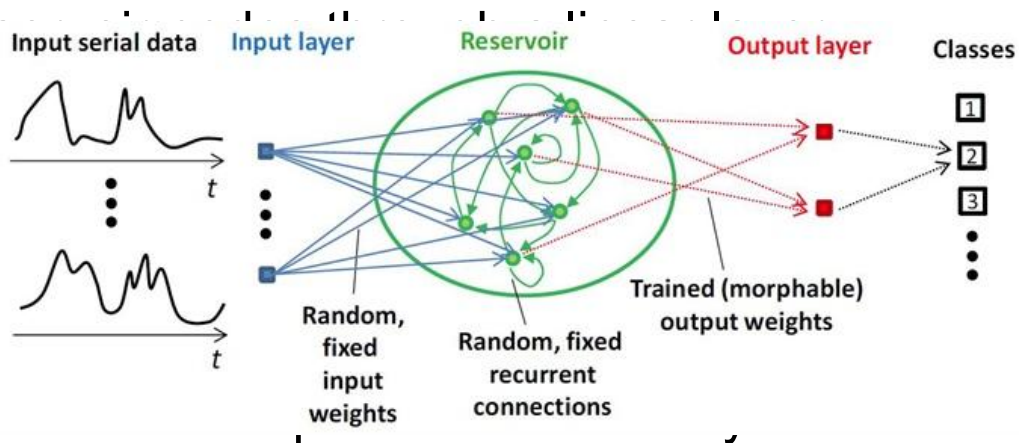
# Weights and connections

- Reservoir nodes with recurrent connections:

$$W_{\text{res}} \in \mathbb{R}^{N \times N}$$

- The inputs are connected to re

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



- The reservoir nodes are conne

$$\hat{\mathbf{y}}(t) = W_{\text{out}} \mathbf{r}(t) + \mathbf{c}$$

# Heuristic choice of weights

The input and recurrent weights are initialized randomly and then are fixed

So, we are not training them... boosting the training

How should we fix them to optimize the prediction? Later...

# Dynamics of hidden states

The hidden states evolve dynamically in time as:

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

# Dynamics of hidden states

The hidden states evolve dynamically in time as:

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

- $\alpha$  the leakage rate, controls the speed of evolution (leaky unit)
- $f(\cdot)$  the activation function (usually  $\tanh(\cdot)$  and sigmoid)

# Training

Since we have the evolution of the state, a linear layers gives the output

$$\hat{\mathbf{y}}(t) = W_{\text{out}}\mathbf{r}(t) + \mathbf{c}$$

Determine the weights and bias by minimizing the MSE loss

$$\mathcal{L} = \sum_{t=1}^T \|\mathbf{W}_{\text{out}}\mathbf{r}(t) + \mathbf{c} - \mathbf{y}(t)\|^2 + \beta \text{Tr} (W_{\text{out}} W_{\text{out}}^T)$$

- Linear Regression (simple!)
- $\beta$  is a ridge regression parameter (the last is a regularization term)



# No free lunch...

The training is very easy and fast but there are hyper-parameters

# No free lunch...

The training is very easy and fast but there are hyper-parameters

- Optimize for  $\alpha$ ,  $\mathbf{b}$

# No free lunch...

The training is very easy and fast but there are hyper-parameters

- Optimize for  $\alpha$ ,  $\mathbf{b}$
- Hyper-parameters that govern the random generation of the weights
  - The degree of a reservoir nodes  $D$ :
    - $W_{\text{res}}$  is sparse with  $D/N$  non-zero elements

# No free lunch...

The training is very easy and fast but there are hyper-parameters

- Optimize for  $\alpha$ ,  $\mathbf{b}$
- Hyper-parameters that govern the random generation of the weights
  - The degree of a reservoir nodes  $D$ :
    - $W_{\text{res}}$  is sparse with  $D/N$  non-zero elements
  - The spectral radius  $\rho$ :
    - The largest eigenvalue of  $W_{\text{res}}$  is  $\rho$

# No free lunch...

The training is very easy and fast but there are hyper-parameters

- Optimize for  $\alpha$ ,  $\mathbf{b}$
- Hyper-parameters that govern the random generation of the weights
  - The degree of a reservoir nodes  $D$ :
    - $W_{\text{res}}$  is sparse with  $D/N$  non-zero elements
  - The spectral radius  $\rho$ :
    - The largest eigenvalue of  $W_{\text{res}}$  is  $\rho$
  - $W_{\text{in}}$ : initialized by a uniform distribution in the  $[-\sigma, \sigma]$

# No free lunch...

The training is very easy and fast but there are hyper-parameters

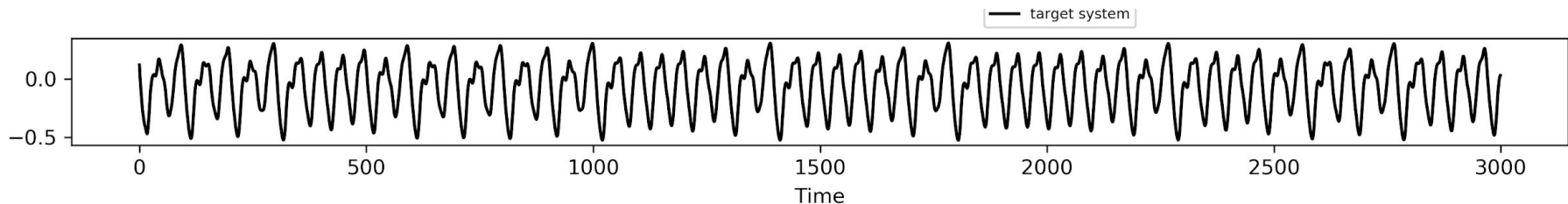
- Optimize for  $\alpha$ ,  $\mathbf{b}$
- Hyper-parameters that govern the random generation of the weights
  - The degree of a reservoir nodes  $D$ :
    - $W_{\text{res}}$  is sparse with  $D/N$  non-zero elements
  - The spectral radius  $\rho$ :
    - The largest eigenvalue of  $W_{\text{res}}$  is  $\rho$
  - $W_{\text{in}}$ : initialized by a uniform distribution in the  $[-\sigma, \sigma]$
- No systematic method to optimize the hyper-parameters

# Recap

- The network nodes each have distinct dynamical behavior
- Time delays of signal may occur along the network links
- The network hidden part has recurrent connections
- The input and internal weights are fixed and randomly chosen
- Only the output weight are adjusted during the training.

# Example: Chaotic time series prediction

## Data



## RC implementation

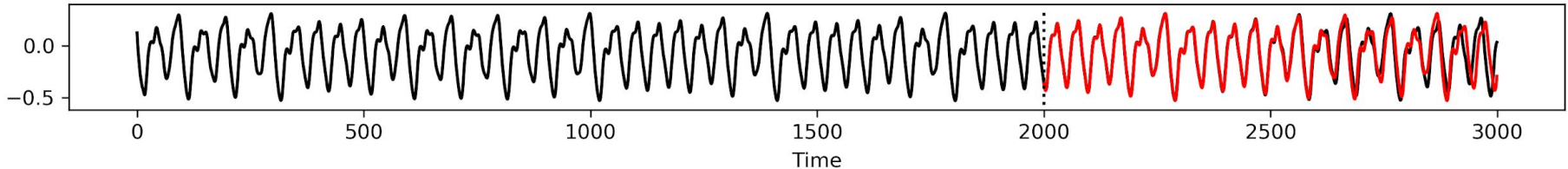
```
1 import numpy as np
2 from pyESN import ESN
3 from matplotlib import pyplot as plt
4 %matplotlib inline
5
6 data = np.load('mackey_glass_t17.npy') # http://minds.jacobs-university.de/mantas/code
7 esn = ESN(n_inputs = 1,
8           n_outputs = 1,
9           n_reservoir = 1000,
10          spectral_radius = 1.5,
11          sparsity=.2,
12          random_state=42)
13
```



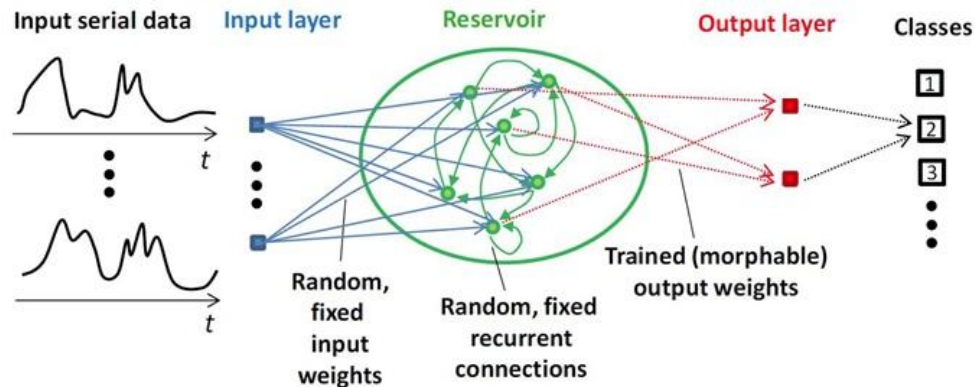
# Example: Chaotic time series prediction

```
14 trainlen = 2000
15 future = 1000
16 pred_training = esn.fit(np.ones(trainlen),data[:trainlen])
17
18 prediction = esn.predict(np.ones(future))
19 print("test error: \n"+str(np.sqrt(np.mean((prediction.flatten() - data[trainlen:trainlen+future])**2))))
20
21 plt.figure(figsize=(11,1.5))
22 plt.plot(range(0,trainlen+future),data[0:trainlen+future],'k',label="target system")
23 plt.plot(range(trainlen,trainlen+future),prediction,'r', label="free running ESN")
24 lo,hi = plt.ylim()
25 plt.plot([trainlen,trainlen],[lo+np.spacing(1),hi-np.spacing(1)],'k:')
26 # plt.legend(loc=(0.61,1.1), fontsize='x-small')
27
28 plt.tight_layout()
29 plt.savefig('MG_prediction.png', dpi = 300)
```

test error:  
0.09277461409740262



# Reservoir Computing



- Forecasting the weather
- Controlling complex dynamical systems
- Predicting and analyzing time series
- Pattern recognition
  - more...

