



# RECURRENT NEURAL NETWORKS

A QUICK OVERVIEW

Geilo 2017 – Winter School

*Filippo Maria Bianchi – [filippo.m.bianchi@uit.no](mailto:filippo.m.bianchi@uit.no)  
Machine Learning Group – Department of Physics and Technology  
Universitet i Tromsø*

# PRESENTATION OUTLINE

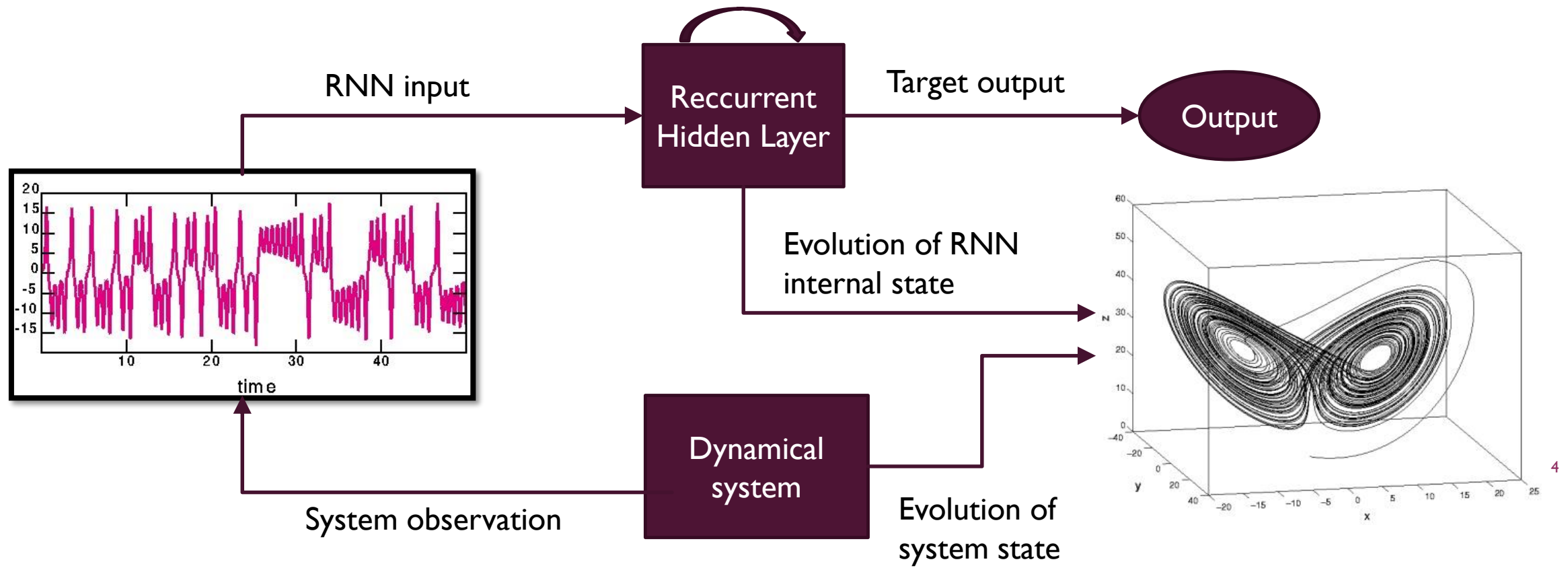
1. Introduction
2. The Recurrent Neural Network
3. RNN applications
4. The RNN model
5. Gated RNN (LSTM and GRU)
6. Echo State Network




# I. INTRODUCTION

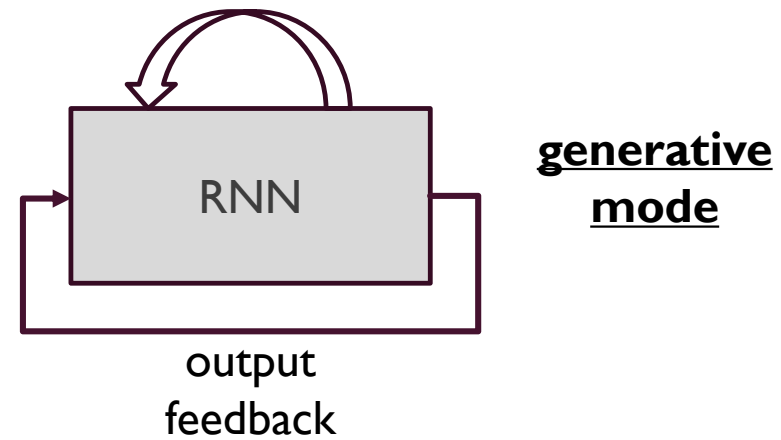
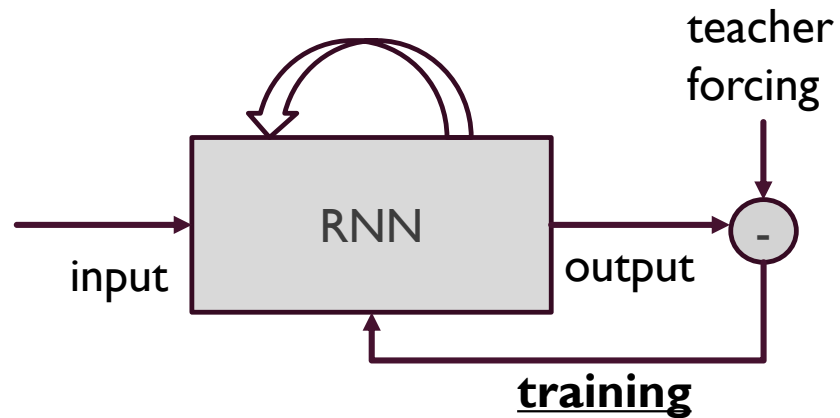
# THE RECURRENT NEURAL NETWORK

- A recurrent neural network (RNN) is a universal approximator of dynamical systems.
- It can be trained to reproduce any target dynamics, up to a given degree of precision.



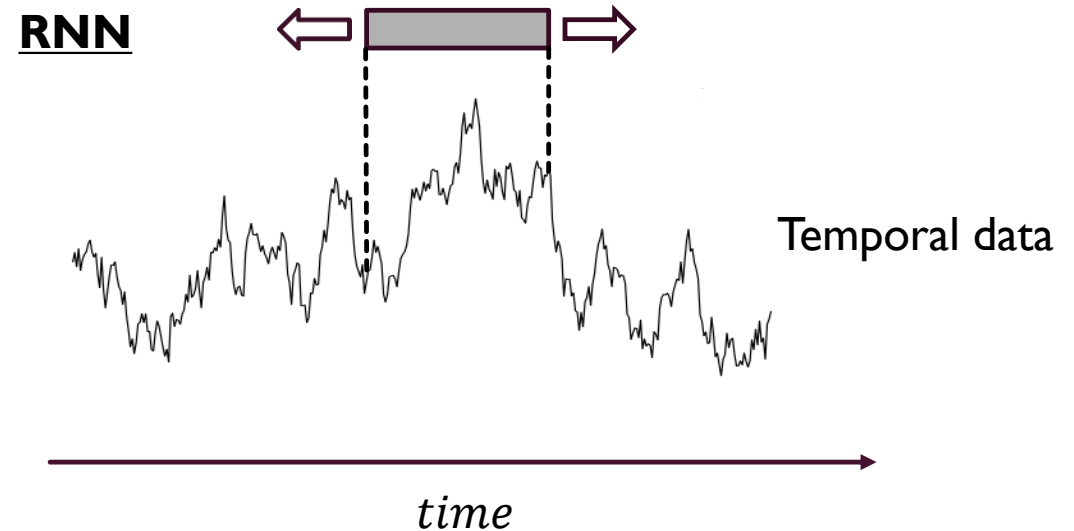
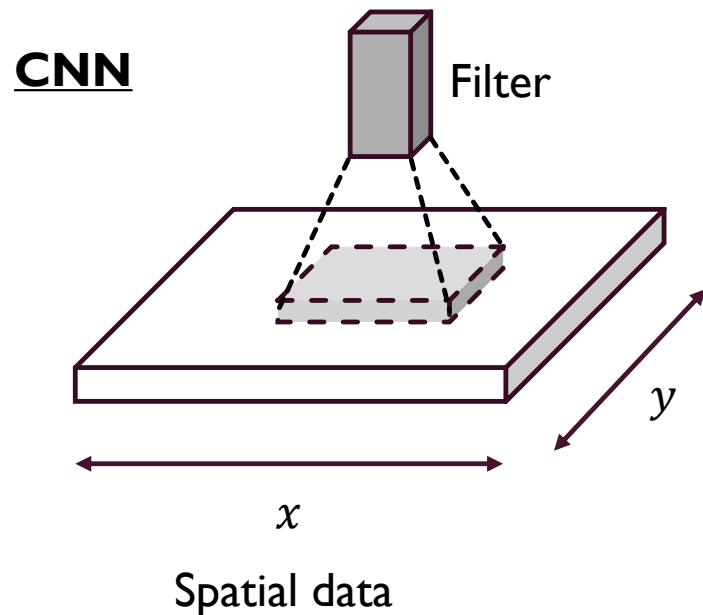
- 
- An RNN generalizes naturally to **new inputs** with any lengths.
  - An RNN make use of **sequential information**, by modelling a **temporal dependencies** in the inputs.
    - Example: if you want to predict the **next word** in a sentence you need to know which words came before it
  - The **output** of the network depends on the **current input** and on the value of the **previous internal state**.
  - The **internal state** maintains a (vanishing) memory about **history** of all past inputs.
  - RNNs can make use of information coming from **arbitrarily long** sequences, but in practice they are limited to look back only a **few time steps**.

- RNN can be trained to predict a **future value**, of the driving input.
- A side-effect we get a generative model, which allows us to **generate new elements** by sampling from the output probabilities.



# DIFFERENCES WITH CNN

- **Convolution** in space (CNN) VS convolution in time (RNN) .
- CNN: models **relationships in space**. Filter slides along  $x$  and  $y$  dimensions.
- RNN: models **relationships in time**. “Filter” slides along time dimension.



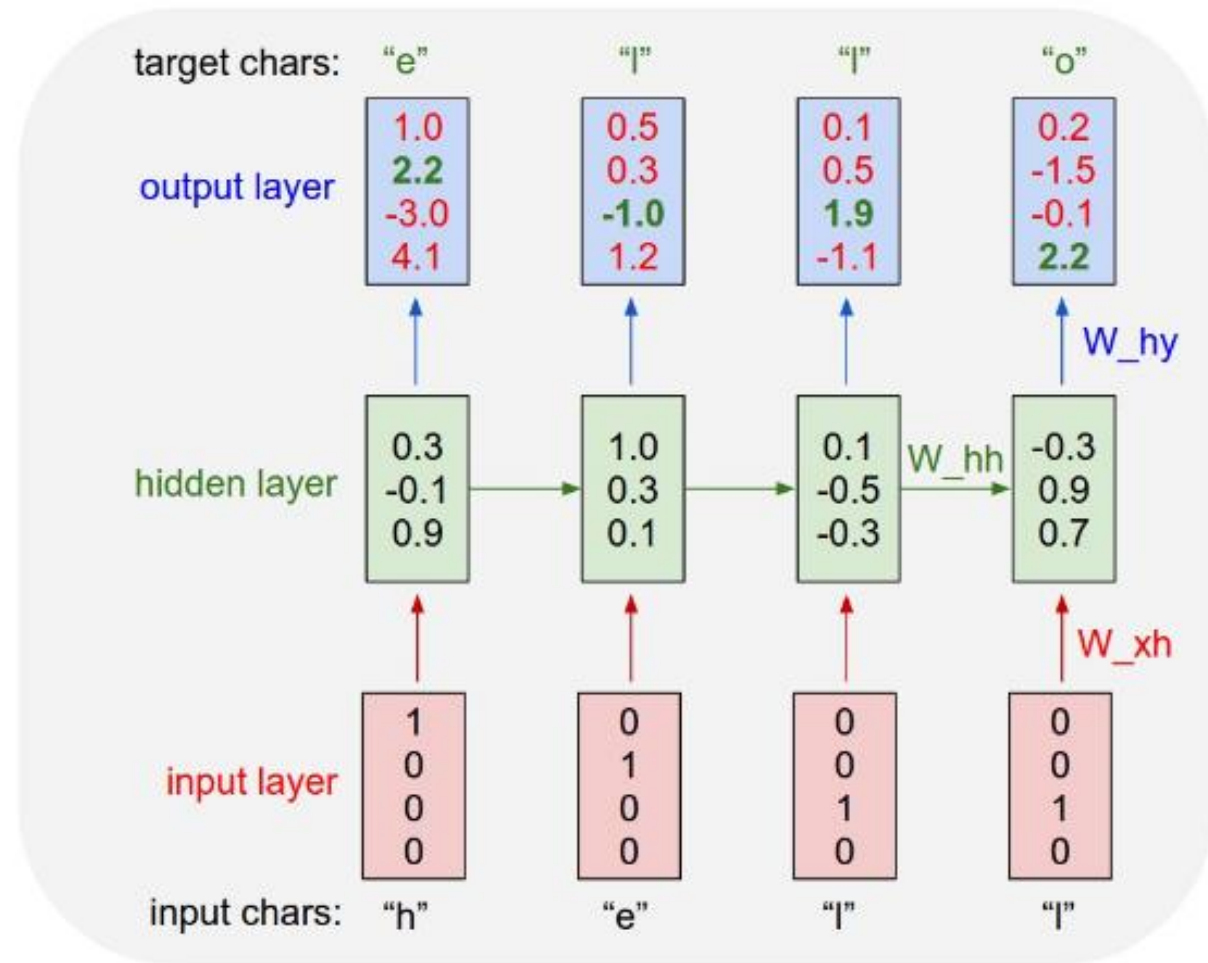


## 2. RNN APPLICATIONS



## APPLICATION I: NATURAL LANGUAGE PROCESSING

- Given a sequence of words, RNN predicts the probability of next word given the previous ones.
- Input/output words are encoded as one-hot vector.
- We must provide the RNN all the dictionary of interest (usually, just the alphabet).
- In the output layer, we want the green numbers to be high and red numbers to be low.



[Image:Andrej Karpathy]

- 
- Once trained, the RNN can work in **generative mode**.
  - In NLP context, a generative RNN can be used in Natural Language Generation.
  - Applications:
    - Generate text (human readable data) from database of numbers and log files, not readable by human.
    - *What you see is what you meant.* Allows users to see and manipulate the continuously rendered view (NLG output) of an underlying formal language document (NLG input), thereby editing the formal language without learning it.

## NATURAL LANGUAGE GENERATION: SHAKESPEARE

- Dataset: all the works of Shakespeare, concatenated them into a single (4.4MB) file.
- 3-layer RNN with 512 hidden nodes on each layer.
- Few hours of training.

VIOLA:

```
Why, Salisbury must find his flesh and thought  
That which I am not apt, not a man and in fire,  
To show the reining of the raven and the wars  
To grace my hand reproach within, and not a fair are hand,  
That Caesar and my goodly father's world;  
When I was heaven of presence and our fleets,  
We spare with hours, but cut thy council I am great,  
Murdered and by thy master's ready there  
My power to give thee but so much as hell:  
Some service in the noble bondman here,  
Would show him to her wine.
```

KING LEAR:

```
O, if you were a feeble sight, the courtesy of your law,  
Your sight and several breath, will wear the gods  
With his heads, and my hands are wonder'd at the deeds,  
So drop upon your lordship's head, and your opinion  
Shall be against your honour.
```

[Source:Andrej Karpathy]

# TEXT GENERATION: WIKIPEDIA

- Hutter Prize 100MB dataset of raw Wikipedia.
- LSTM
- The link does not exist 😊

Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be said to be directly in Cantonese Communication, which followed a ceremony and set inspired prison, training. The emperor travelled back to [[Antioch, Perth, October 25|21]] to note, the Kingdom of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known in western [[Scotland]], near Italy to the conquest of India with the conflict. Copyright was the succession of independence in the slop of Syrian influence that was a famous German movement based on a more popular servicious, non-doctrinal and sexual power post. Many governments recognize the military housing of the [[Civil Liberalization and Infantry Resolution 265 National Party in Hungary]], that is sympathetic to be to the [[Punjab Resolution]] (PJS)[<http://www.humah.yahoo.com/guardian.cfm/7754800786d17551963s89.htm> Official economics Adjoint for the Nazism, Montgomery was swear to advance to the resources for those Socialism's rule, was starting to signing a major tripad of aid exile.]]

[Source:Andrej Karpathy]

# TEXT GENERATION: SCIENTIFIC PAPER

- RNN trained on a book (LaTeX source code of 16MB).
- Multilayer LSTM

For  $\bigoplus_{n=1, \dots, m}$  where  $\mathcal{L}_{m\bullet} = 0$ , hence we can find a closed subset  $\mathcal{H}$  in  $\mathcal{H}$  and any sets  $\mathcal{F}$  on  $X$ ,  $U$  is a closed immersion of  $S$ , then  $U \rightarrow T$  is a separated algebraic space.

*Proof.* Proof of (1). It also start we get

$$S = \text{Spec}(R) = U \times_X U \times_X U$$

and the comparicoly in the fibre product covering we have to prove the lemma generated by  $\coprod Z \times_U U \rightarrow V$ . Consider the maps  $M$  along the set of points  $Sch_{fppf}$  and  $U \rightarrow U$  is the fibre category of  $S$  in  $U$  in Section, ?? and the fact that any  $U$  affine, see Morphisms, Lemma ?? . Hence we obtain a scheme  $S$  and any open subset  $W \subset U$  in  $Sh(G)$  such that  $\text{Spec}(R') \rightarrow S$  is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that  $f_i$  is of finite presentation over  $S$ . We claim that  $\mathcal{O}_{X,x}$  is a scheme where  $x, x', s'' \in S'$  such that  $\mathcal{O}_{X,x'} \rightarrow \mathcal{O}'_{X',x'}$  is separated. By Algebra, Lemma ?? we can define a map of complexes  $\text{GL}_{S'}(x'/S'')$  and we win.  $\square$

To prove study we see that  $\mathcal{F}|_U$  is a covering of  $\mathcal{X}'$ , and  $\mathcal{T}_i$  is an object of  $\mathcal{F}_{X/S}$  for  $i > 0$  and  $\mathcal{F}_p$  exists and let  $\mathcal{F}_i$  be a presheaf of  $\mathcal{O}_X$ -modules on  $\mathcal{C}$  as a  $\mathcal{F}$ -module. In particular  $\mathcal{F} = U/\mathcal{F}$  we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\text{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1} \mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

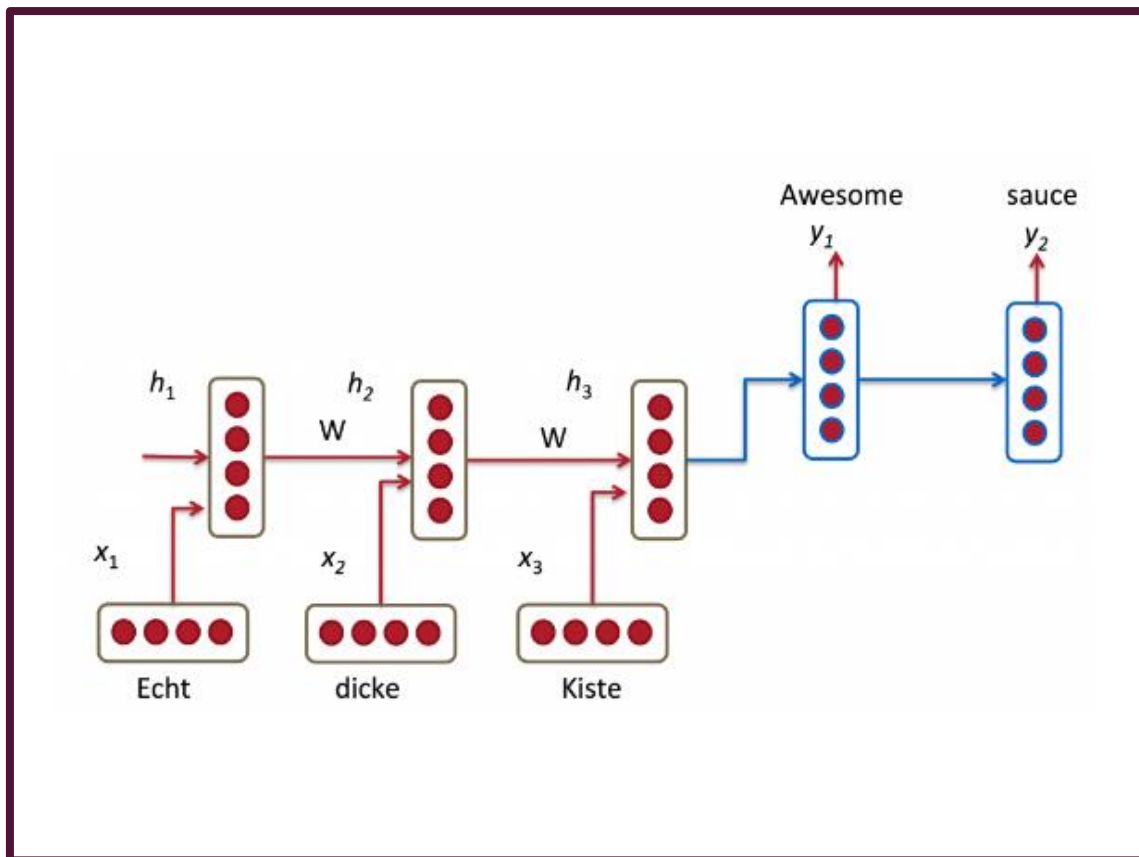
$$\text{Arrows} = (Sch/S)_{fppf}^{opp}, (Sch/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longmapsto (U, \text{Spec}(A))$$

is an open subset of  $X$ . Thus  $U$  is affine. This is a continuous map of  $X$  is the inverse, the groupoid scheme  $S$ .

# APPLICATION II: MACHINE TRANSLATION

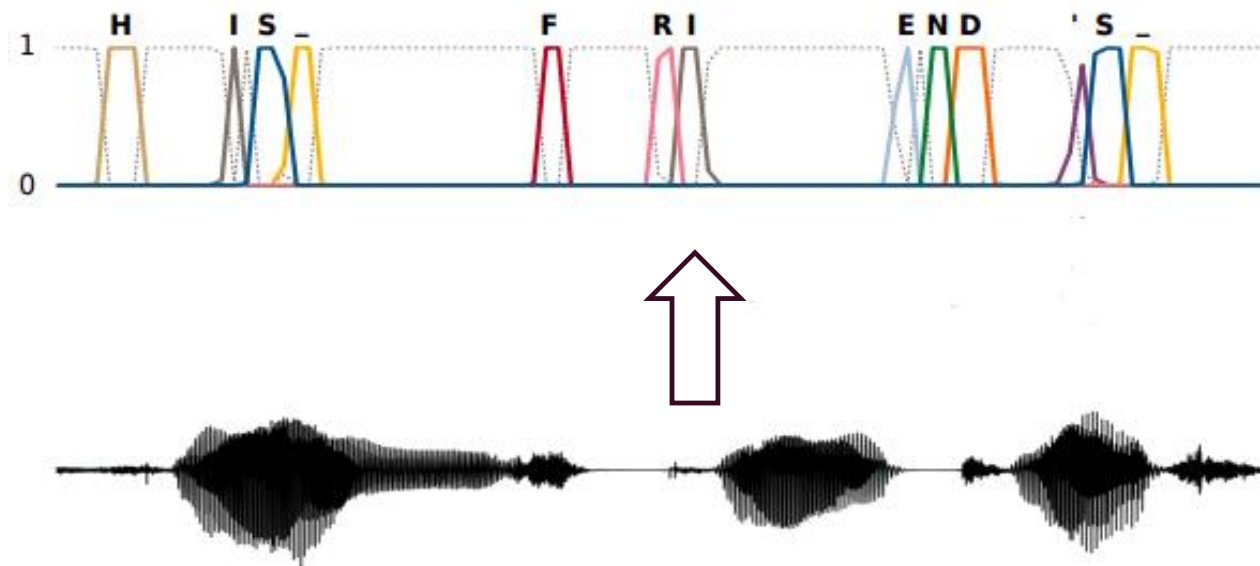


[Image: Richard Socher]

- Similar to language modeling.
- Train 2 different RNNs.
- Input RNN: trained on a source language (e.g. German).
- Output RNN: trained on a target language (e.g. English).
- The **second** RNN computes the output from the hidden layer of the **first** RNN.
- Google translator.

# APPLICATION III: SPEECH RECOGNITION

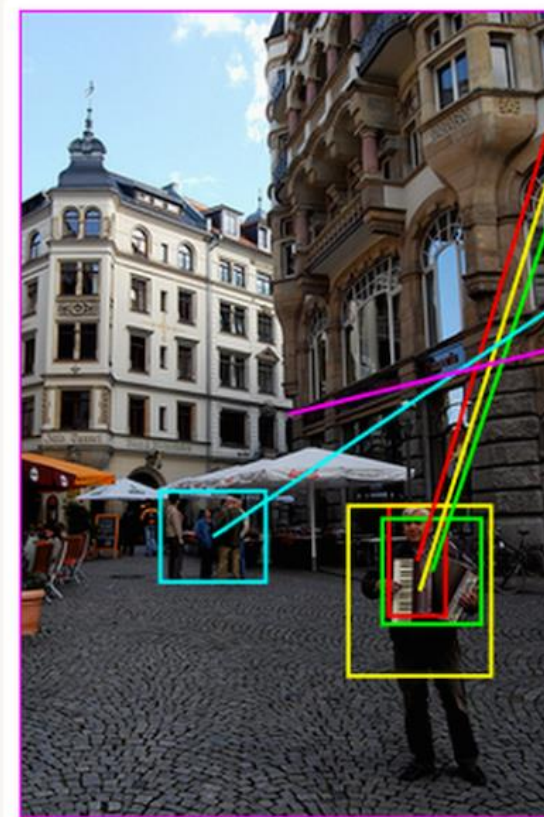
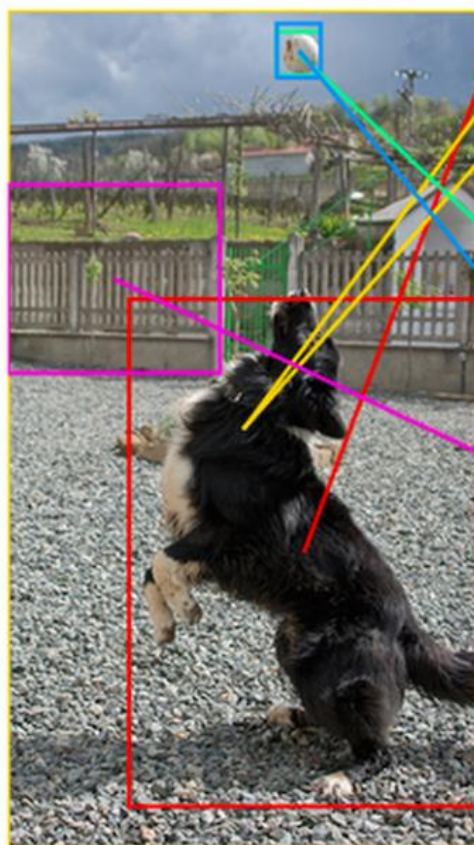
- Input: input sequence of acoustic signals.
- Output phonetic segments.
- Necessity of encoder/decoder to transit from digital/analogic domain.
- *Graves, Alex, and Navdeep Jaitly. "Towards End-To-End Speech Recognition with Recurrent Neural Networks.", 2014.*





# APPLICATION IV: IMAGE TAGGING

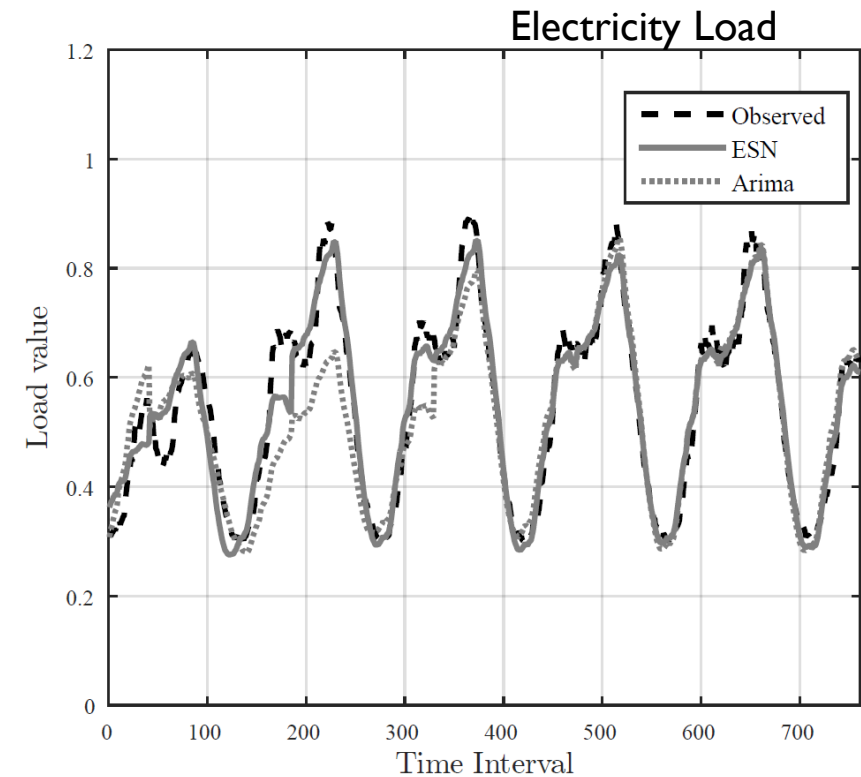
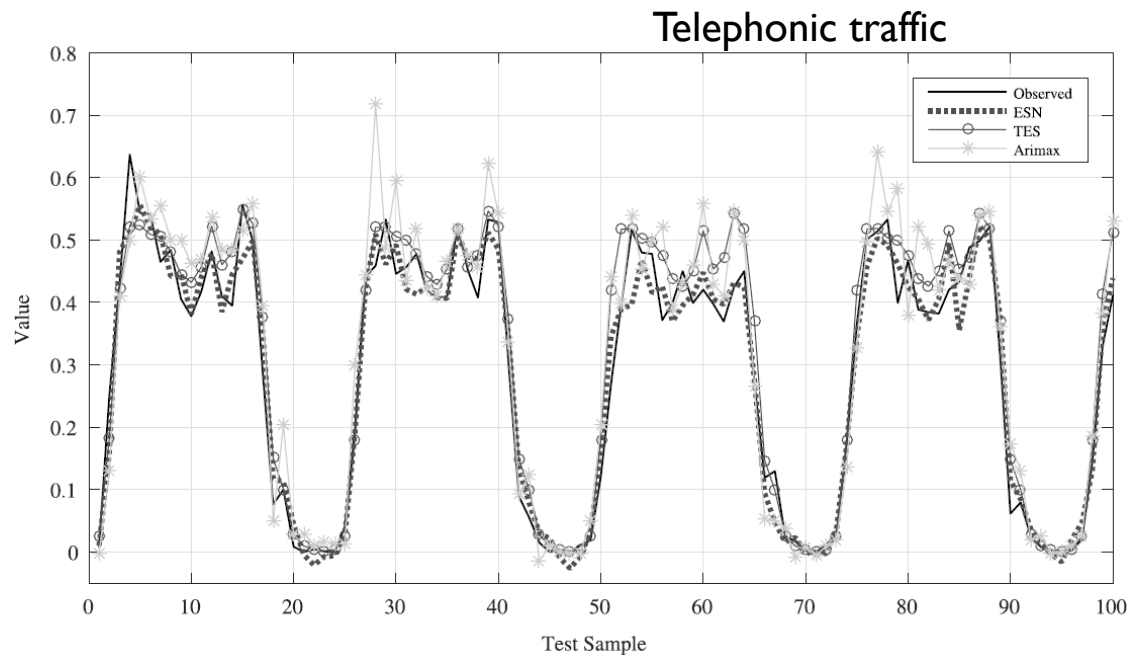
- RNN + CNN jointly trained.
- CNN generates features (hidden state representation).
- RNN reads CNN features and produces output (end-to-end training).
- Aligns the generated words with features found in the images
- *Karpathy, Andrej, and Li Fei-Fei. "Deep visual-semantic alignments for generating image descriptions.", 2015.*





# APPLICATION V: TIME SERIES PREDICTION

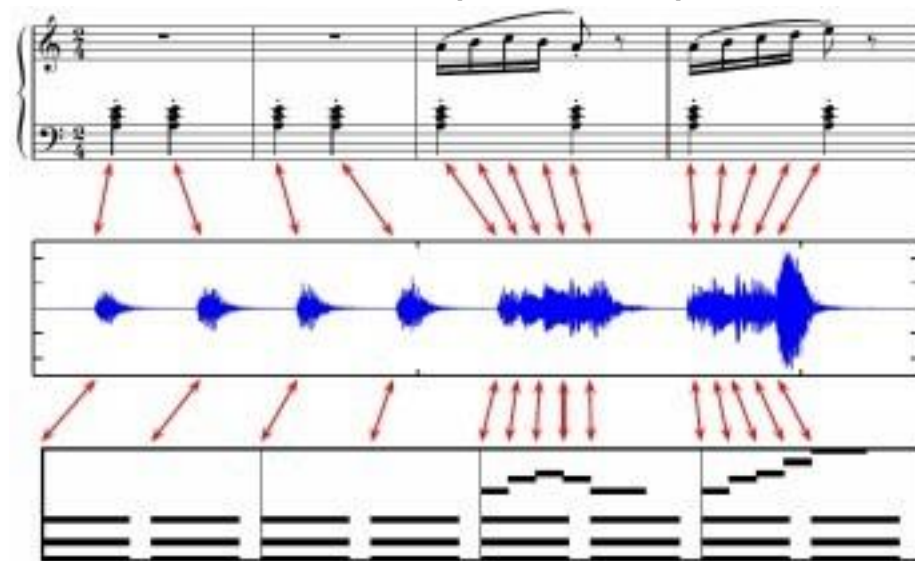
- Forecast of future values in a time series, from past seen values.
- Many applications:
  - Weather forecast.
  - Load forecast.
  - Financial time series.



# APPLICATION VI: MUSIC INFORMATION RETRIEVAL

- MIR: identification of songs/music
  - Automatic categorization.
  - Recommender systems.
  - Track separation and instrument recognition.
  - Music generation. 🗣️
  - Automatic music transcription.

Music transcription example




[Source: Meinard Müller]

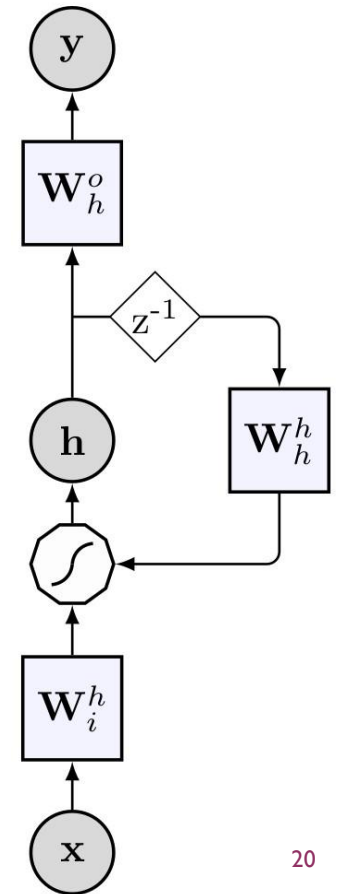




## 3. DESCRIPTION OF RNN MODEL

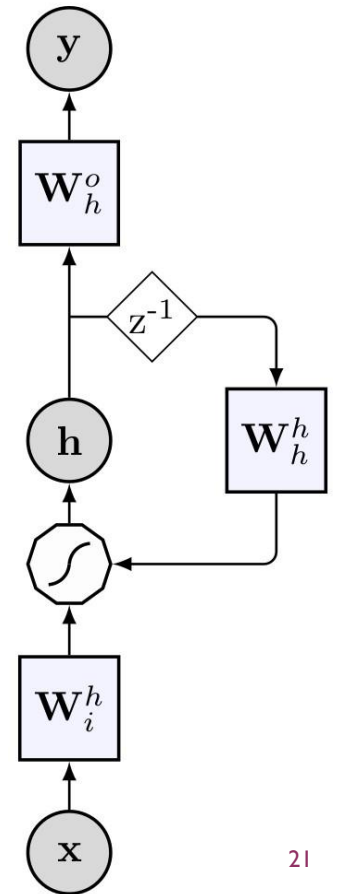
# ARCHITECTURE COMPONENTS

- $x$ : input
- $y$ : output
- $h$ : internal state (memory of the network)
- $W_i^h$ : input weights
- $W_h^h$ : recurrent layer weights
- $W_h^o$ : output weights
- $z^{-1}$ : time-delay unit
-  : neuron transfer function



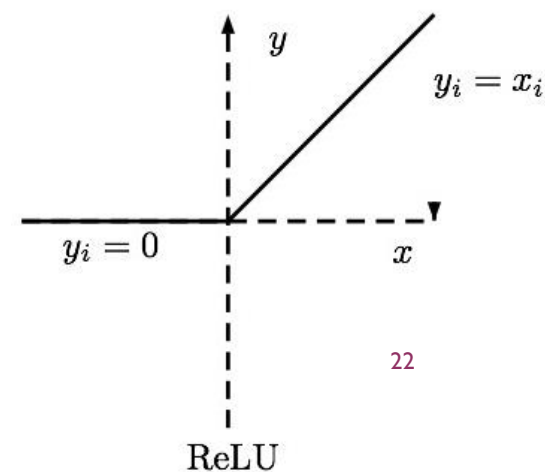
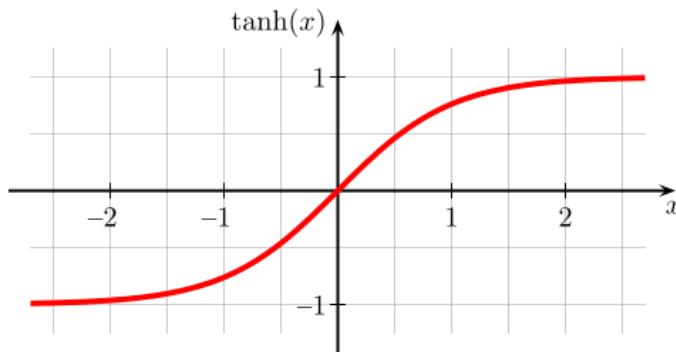
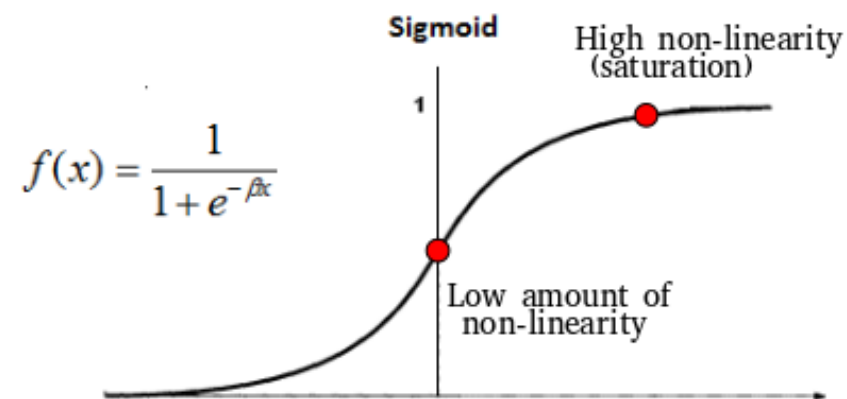
# STATE UPDATE AND OUTPUT GENERATION

- An RNN selectively **summarize an input sequence** in a fixed-size state vector via a **recursive update**.
- Discrete, time-independent **difference equations** of RNN state and output:  
$$h[t + 1] = f(W_h^h h[t] + W_i^h x[t + 1] + b_h),$$
$$y[t + 1] = g(W_h^o h[t + 1] + b_o).$$
- $f()$  is the **transfer function** implemented by each neuron (usually the same non-linear function for all neurons).
- $g()$  is the **readout** of the RNN. Usually is the identity function - all the non-linearity is provided by the internal processing units (neurons) – or the softmax function.



# NEURON TRANSFER FUNCTION

- The activation function in a RNN is traditionally implemented by a sigmoid.
  - Saturation causes vanishing gradient.
  - Non-zero centering produces only positive outputs, which lead to zig-zagging dynamics in the gradient updates.
- Another common choice is the tanh.
  - Saturation causes vanishing gradient.
- ReLU (not very much used in RNN).
  - Greatly accelerate gradient convergence and it has low demanding computational time.
  - No vanishing gradient.
  - Large gradient flowing through a ReLU neuron could cause the its “death”.



# TRAINING

- Model's parameters are **trained** with **gradient descent**.
- A loss function is evaluated on the **error** performed by the network on the training set and, usually, also a **regularization** term.

$$L = E(y, \hat{y}) + \lambda R$$

Where  $E()$  is the error function,  $y$  and  $\hat{y}$  are target and estimated outputs,  $\lambda$  is the regularization parameter,  $R$  is the regularization term.

- The **derivative of the loss function**, with respect to the **model parameters**, is **backpropagated** through the network.
- Weights are adjusted until a **stop criterion** is met:
  - Maximum number of **epochs** is reached.
  - Loss function **stop decreasing**.

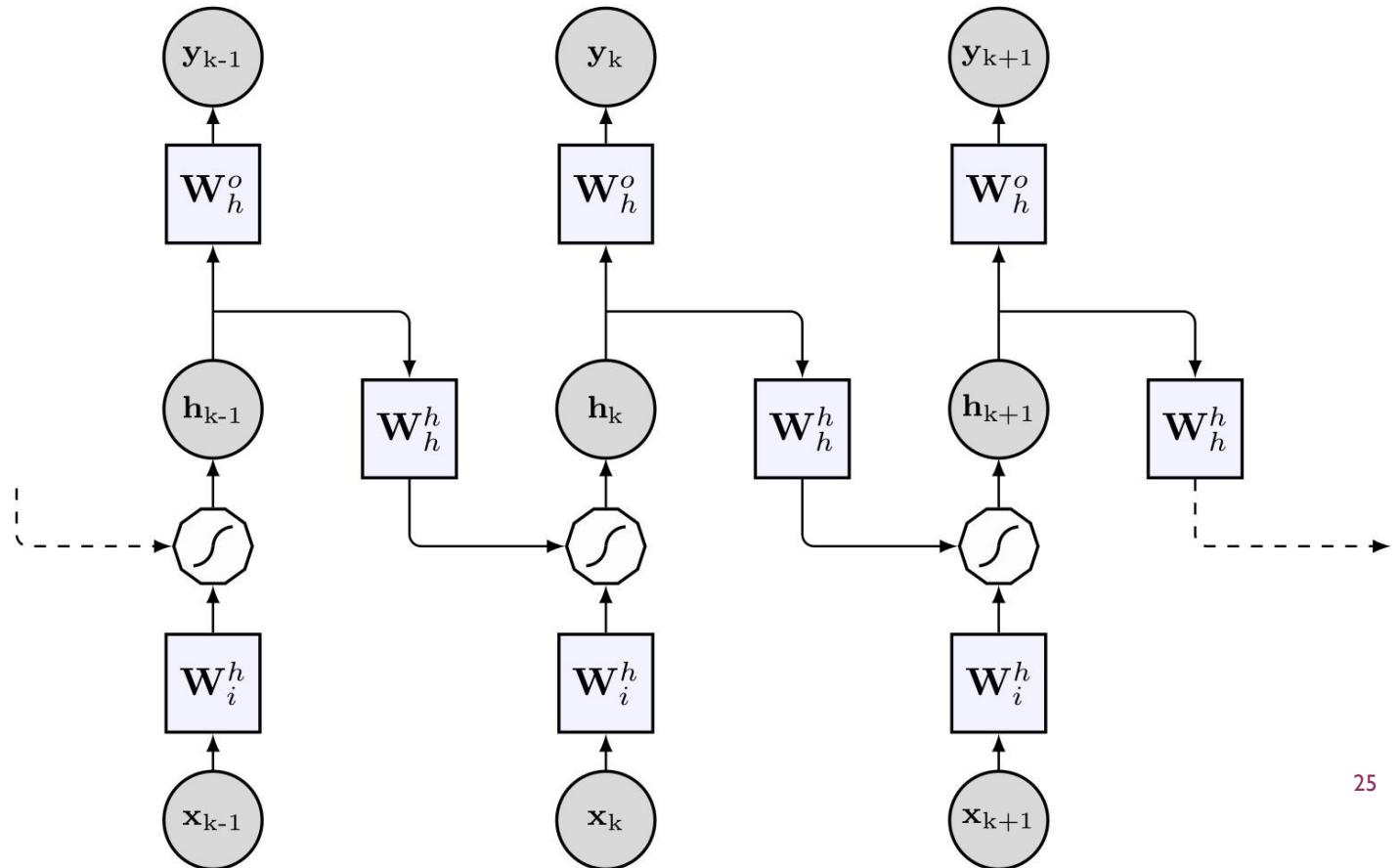
# REGULARIZATION

- Introduce a **bias**, necessary to prevent the RNN to **overfit** on training data.
- In order to **generalize** well to unseen data, the **variance** (complexity) of the model should be **limited**.
- Common regularization terms:
  1.  $L_1$  regularization of the weights:  $\|W\|_1$ . Enforce sparsity in the weights.
  2.  $L_2$  regularization of the weights:  $\|W\|_2$ . Enforce small values for the weights.
  3.  $L_1 + L_2$  (elastic net penalty). Combines the two previous regularizations.
  4. **Dropout**. Done usually only on the output weights. Dropout on recurrent layer is more complicated (the weights are constrained to be the same in each time step by the BPPT) → requires **workaround**.



# RNN UNFOLDING

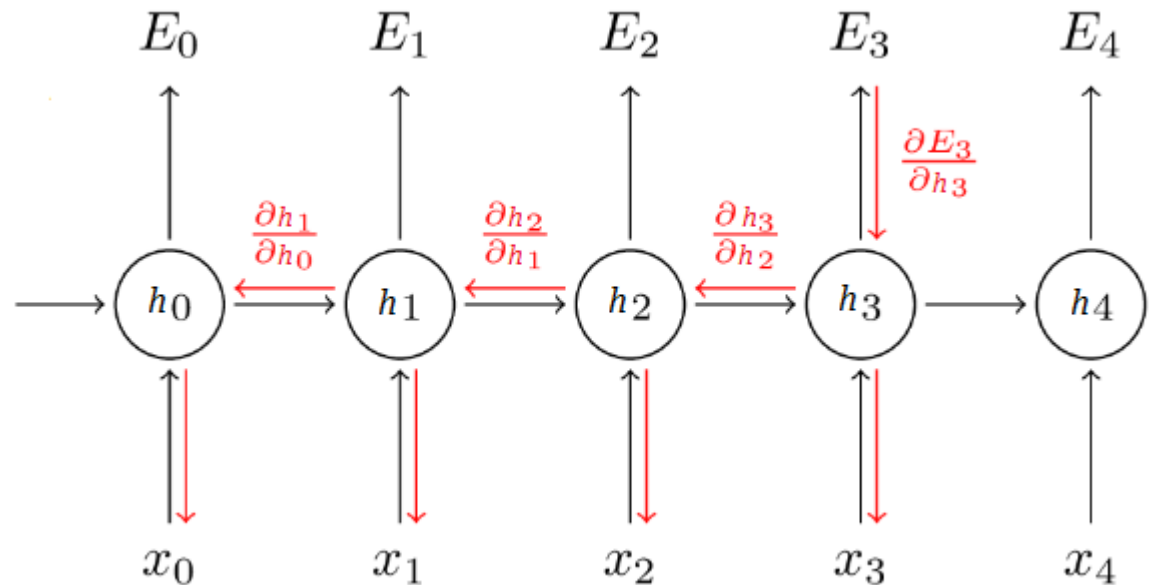
- In order to train the network with gradient descent, the RNN must be **unfolded**.
- Each **replica** of the network is relative to a **different time** interval.
- Now, the architecture of the network become **very deep**, even starting from a shallow RNN.
- The **weights** are constrained to be the **same**.
- **Less parameters** than in other deep architectures.



# BACK PROPAGATION THROUGH TIME

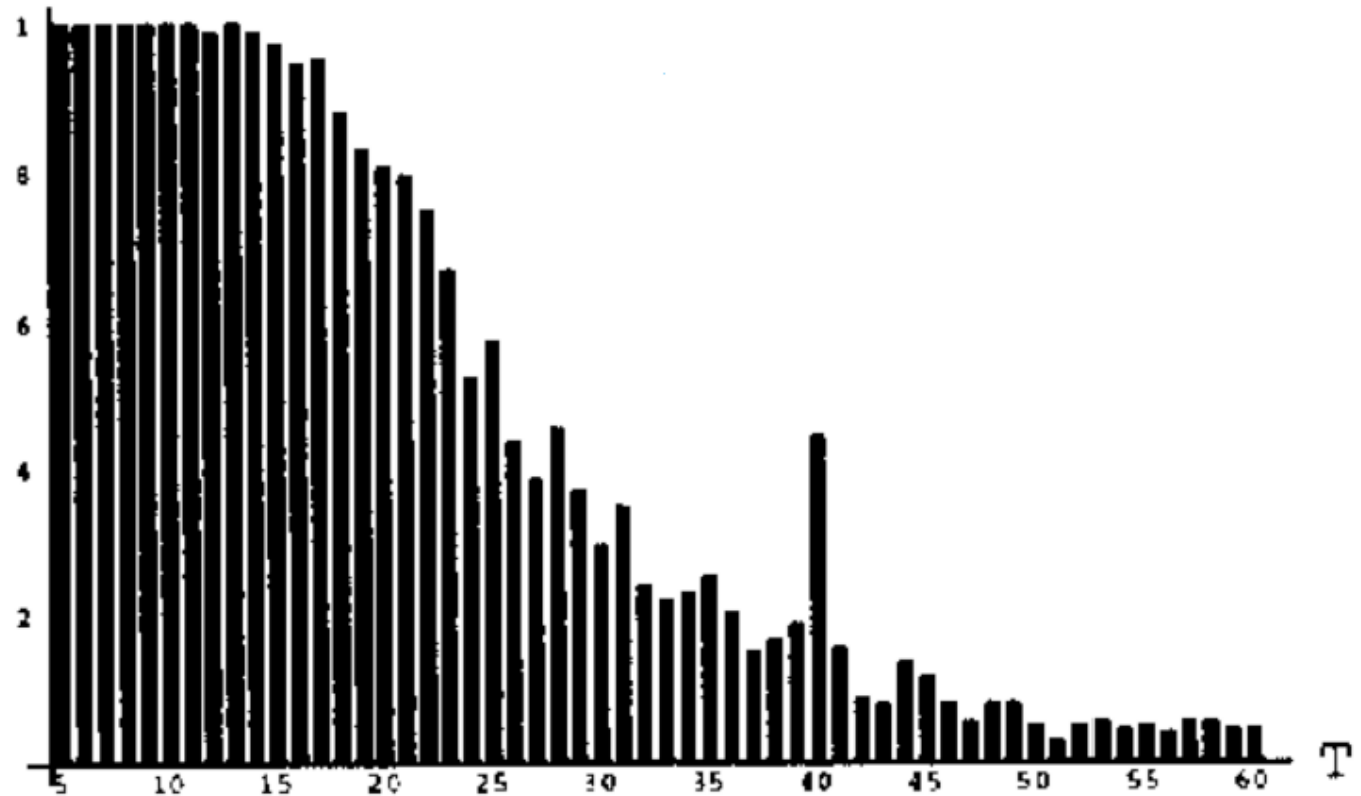
- In the example, we need to backpropagate the gradient  $\frac{\partial E_3}{\partial W}$  from current time ( $t_3$ ) to initial time ( $t_0$ )  $\rightarrow$  **chain rule** (eq. on the right).
- We **sum** up the **contributions** of **each time** step to the gradient.
- With of **very long** sequence (possibly infinite) we have **untreatable depth**.
- Repeat the procedure only up to a given time (**truncate** BPPT).
- Why it works? Because each **state** carries a little bit of **information** on each **previous input**.
- Once the network is **unfolded**, the procedure is analogue to **standard backpropagation** used in deep Feedforward Neural Networks.

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^3 \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial h_3} \frac{\partial h_3}{\partial h_k} \frac{\partial h_k}{\partial W}$$



## VANISHING GRADIENT: SIMPLE EXPERIMENT

- Bengio, 1991.
- A simple RNN is trained to keep 1 bit of information for  $T$  time steps.
- $P(\text{success}|T)$  decreases exponentially as  $T$  increases.



[Image:Yoshua Bengio]

# VANISHING GRADIENT: TOO MANY PRODUCTS!

- In order to have (local) **stability**, the **spectral radius** of the matrix  $W_h^h$  must be **lower than 1**.
- Consider **state update** equation  $h[t + 1] = f(h[t], u[t + 1])$ . We can see it is a **recursive** equation.
- When input sequence is given, the previous equation can be rewritten **explicitly** as:

$$h[t + 1] = f_t(h[t]) = f_t(f_{t-1}(\dots f_0(h[0])). \quad (1)$$

- The resulting gradient, relative to the loss at time  $t$  will be:

$$\frac{\partial L_t}{\partial W} = \sum_{\tau} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_{\tau}} \frac{\partial h_{\tau}}{\partial W}. \quad (2)$$

- The Jacobian of matrix **derivatives**  $\frac{\partial h_t}{\partial h_{\tau}}$  can be **factorized** as follows

$$\frac{\partial h_t}{\partial h_{\tau}} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_{\tau+1}}{\partial h_{\tau}} = f'_t f'_{t-1} \dots f'_{\tau+1} \quad (3)$$

- In order to **reliably** “store” information in the **state** of the network  $h_t$ , RNN **dynamics** must remain close to a **stable attractor**.
- According to **local stability analysis**, the latter condition is met when  $|f'_t| < 1$
- However, the previous product  $\frac{\partial h_t}{\partial h_\tau}$ , expanded in (3) rapidly (exponentially) **converges to 0** when  $t - \tau$  **increases**.
- Consequently, the sum in (2) is **dominated** by terms corresponding to **short-term** dependencies.
- This effects is called “**vanishing gradient**”.
- As an effect, weights are **less and less updates**, as the gradient flows backward through the architecture.
- On the other hand, when  $|f'_t| > 1$  we obtain an opposite effect called “**exploding gradient**”, which leads to **instability** in the network.

# HOW TO LIMIT VANISHING GRADIENT ISSUE?

- Use ReLU activations (in RNN however, they cause the “dying neurons” problem).
- Use LSTM or GRU architectures (discussed later).
- Use a proper **initialization** of the weights in  $W$ .

# WEIGHTS INITIALIZATION

- A suitable initialization of the weights permits the gradient to **flow quicker** through the layers.
- A smoother flow ensures **faster convergence** of the training procedure (faster reach of the minimum).
- It also helps to **reduce** the issue of **vanishing gradient**.
- When using **sigmoids** or **hyperbolic tangent** neurons, use the following weight initialization:

$$w = \text{np.random.randn}(n) / \text{sqrt}(n)$$

- This has to be **repeated** for **each layer**. The value  $n$  is the number of neurons in each layer.
- This ensures that **all neurons** in the network initially have approximately the **same output distribution**.
- The **biases** instead should be initialized to **0**.
- Random initialization is important to break symmetry (prevents coupling of neurons).

# LEARNING RATE

- The standard weights-update procedure is called Stochastic Gradient Descent (SGD).
- SGD depends on a **learning rate** hyperparameter  $\gamma$ :

$$W_i^h = W_i^h - \gamma \frac{\partial L}{\partial W_i^h}, \quad W_h^h = W_h^h - \gamma \frac{\partial L}{\partial W_h^h}, \quad W_h^o = W_h^o - \gamma \frac{\partial L}{\partial W_h^o}.$$

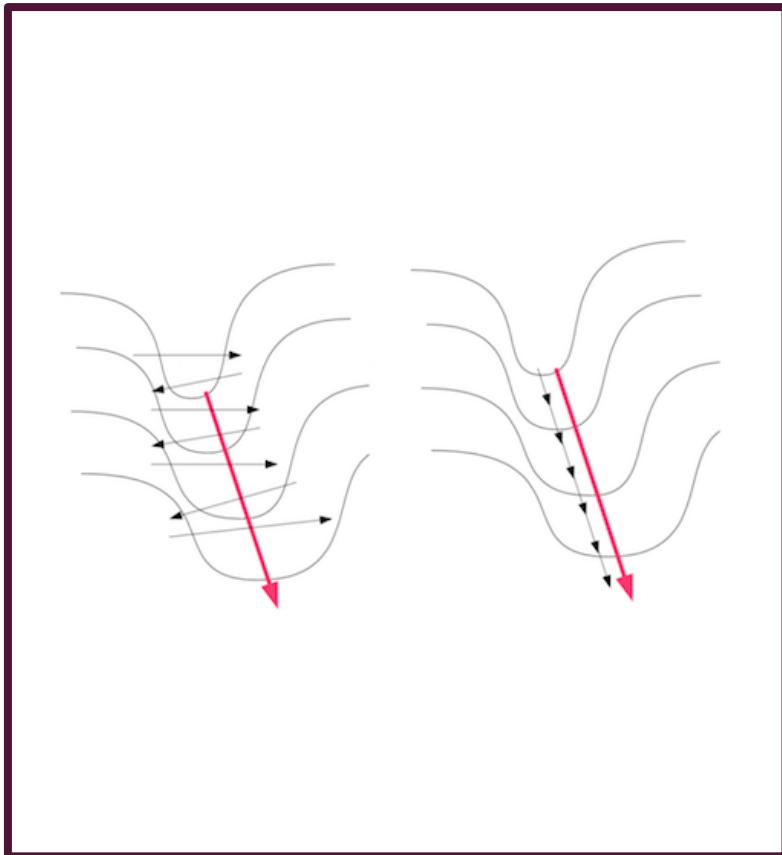
- $\gamma$  is a **critical** parameter. How to set it?
- Use **cross-validation** to find optimal value.
- Several strategies can be used to **improve learning**:
  1. **Annealing** of the learning rate.
  2. **Second order** methods.
  3. Add **momentum** to SGD
  4. **Adaptive** learning rate methods.



# ANNEALING OF THE LEARNING RATE

- With a **high learning** rate, the system has **too much kinetic energy** and the parameter vector bounces around chaotically, **unable** to settle down into deeper, but **narrower** parts of the loss function.
- Possible solution: **anneal** the learning rate **over time**.
- Decay it **slowly** and you'll be **wasting computation** bouncing around chaotically with little improvement for a long time.
- Decay it **too aggressively** and the system will **cool too quickly**, unable to reach the best position it can.
- 3 common ways to decay learning rate:
  1. **Step** decay: half  $\gamma$  every few epochs.
  2. **Exponential** decay:  $\gamma = \gamma_0 e^{-kt}$ , where  $\gamma_0$  and  $k$  are hyperparameters and  $t$  is epoch number.
  3. **Multiplicative** decay:  $\gamma = \frac{\gamma_0}{1+kt}$ , where  $\gamma_0$  and  $k$  are hyperparameters and  $t$  is epoch number.

# SECOND ORDER METHODS



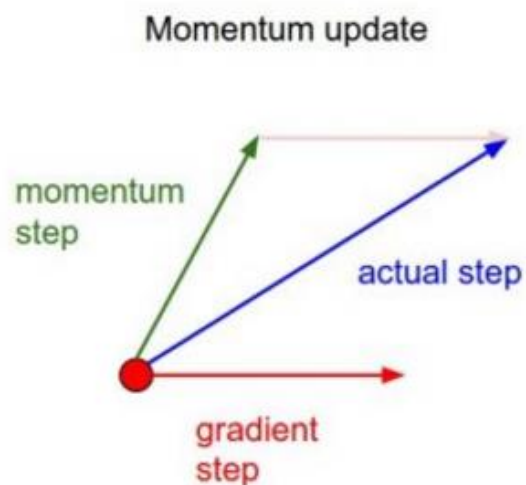
- Weights are updated as follows

$$h \leftarrow h - [Hf(h)]^{-1} \nabla f(h)$$

- Where  $Hf(h)$  is the Hessian matrix, containing **second-order partial derivatives** of  $f$ .
- Hessian describes the **local curvature** of the loss function.
- More **aggressive steps** in directions of **shallow curvature** and **shorter steps** in directions of **steep curvature**.
- Perform a **more efficient update**.
- **Impractical** for most deep learning applications: computing (and inverting) the Hessian is a very **costly** process in both **space** and **time**.
- *Martens, James. "Deep learning via Hessian-free optimization.", 2010: reduces cost, but still much slower than first-order methods.*

# SGD WITH MOMENTUM

- In **SGD** update, gradient directly **integrates the position**.
- With **momentum**, the gradient only directly **influences the velocity**, which in turn has an **effect on the position**.
- Momentum update:  $v \leftarrow \mu \cdot v - \gamma \frac{\partial f(h)}{\partial h}$ ,  $h \leftarrow h + v$
- Nesterov momentum (NAG):  $h^* \leftarrow h + \mu \cdot v$ ,  $v \leftarrow \mu \cdot v - \gamma \frac{\partial f(h^*)}{\partial h^*}$ ,  $h \leftarrow h + v$



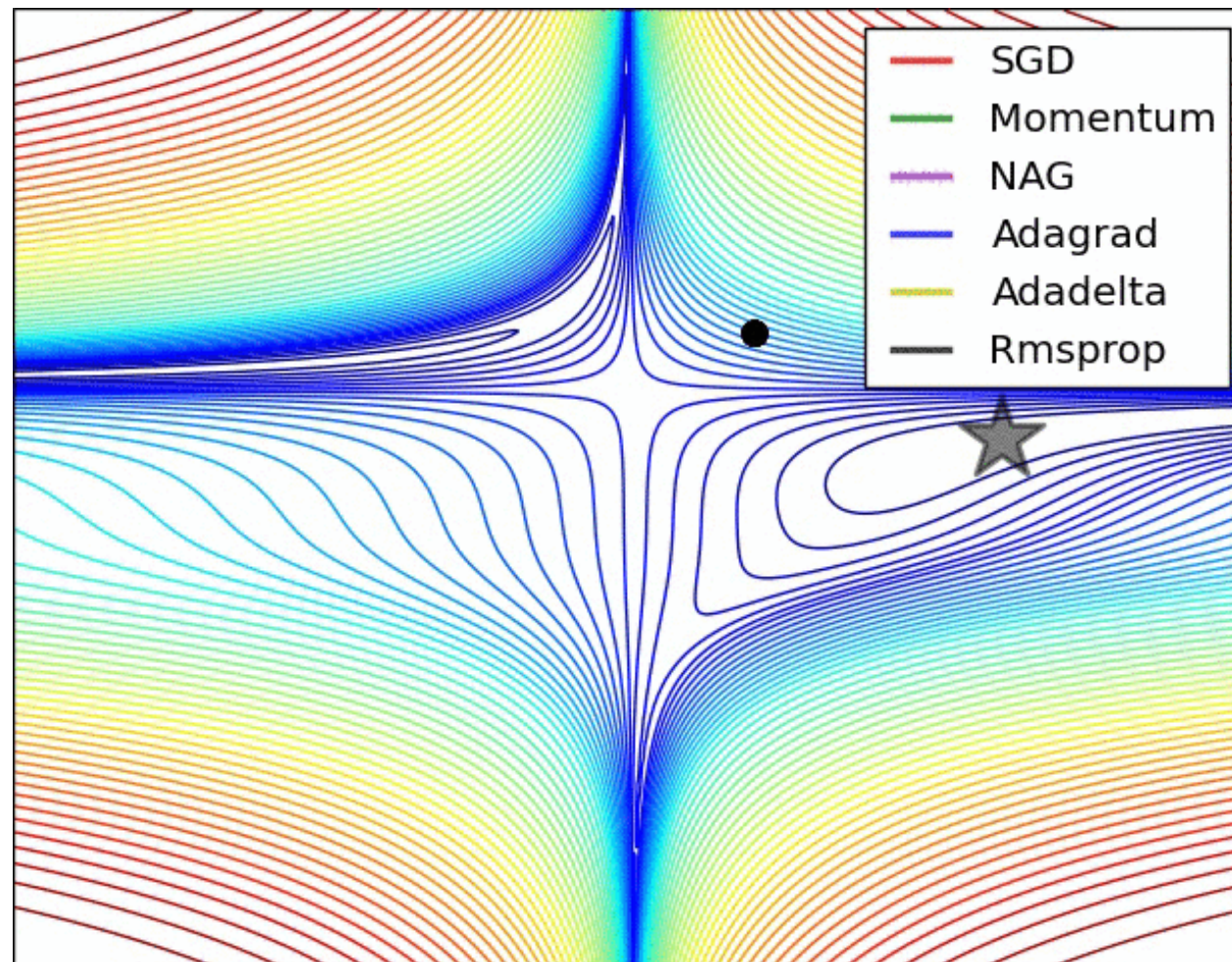
[Image:Andrej Karpathy]

# ADAPTIVE LEARNING RATE

- Adapts the learning rate to the parameters, performing **larger updates** for **infrequent** parameters and **smaller updates** for **frequent** parameters.
- Adagrad:  $g_{i,t} = \nabla_{W} J(w_i)$ ,  $w_{i,t+1} = w_{i,t} - \frac{\gamma}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{i,t}$ , where  $G_t$  is a diagonal matrix whose elements are the sum of the **squares of the gradients** w.r.t. individual weights  $w_i$  at time step  $t$  (they might be updated differently) and  $\epsilon$  is a smoothing term that avoids division by zero.
- Other adaptive learning rate methods are:
  1. Adadelta and Rmsprop: reduce the aggressive, monotonically decreasing learning rate of Adagrad, by restricting the window of accumulated past gradients to some fixed size.
  2. Adam: adaptive learning rate + momentum.
  3. Nadam: adaptive learning rate + Nesterov momentum.

## COMPARISON OF LEARNING PROCESS DYNAMICS (1/2)

- Contours of a loss surface and time evolution of different optimization algorithms.

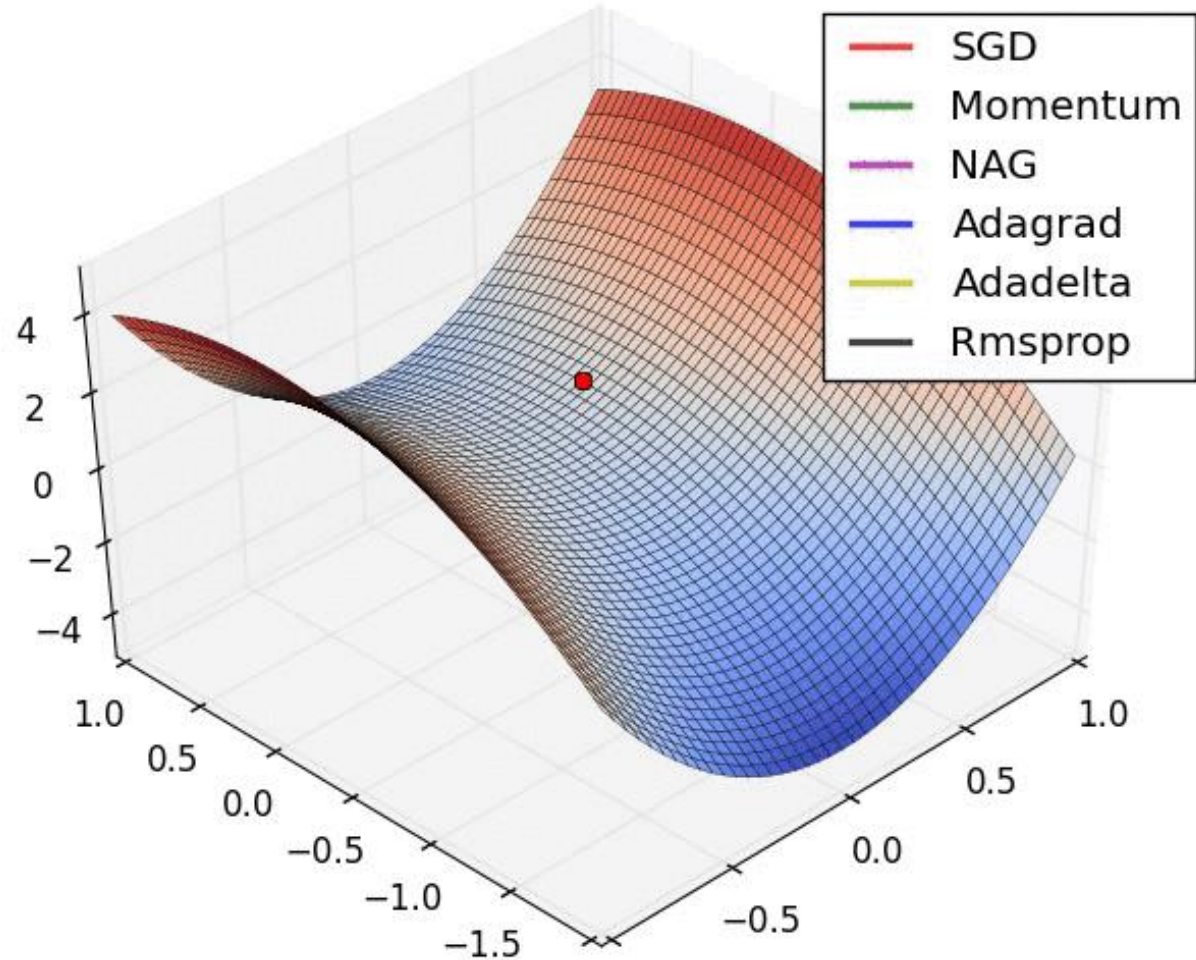


[Image:Alec Radford]



## COMPARISON OF LEARNING PROCESS DYNAMICS (2/2)

- A visualization of a saddle point in the optimization landscape.
- The curvature along different dimension has different signs (one dimension curves up and another down – very common in deep learning!).
- SGD has a very hard time breaking symmetry and gets stuck on the top.
- RMSprop will see very low gradients in the saddle direction



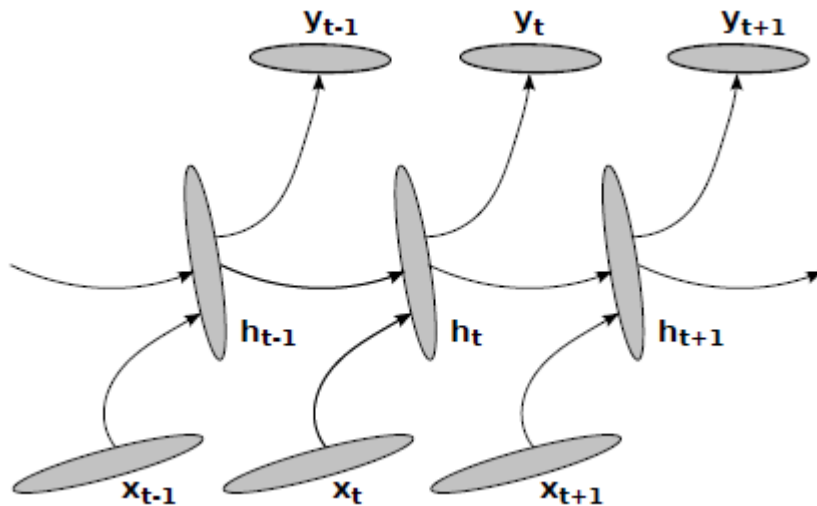
[Image:Alec Radford]



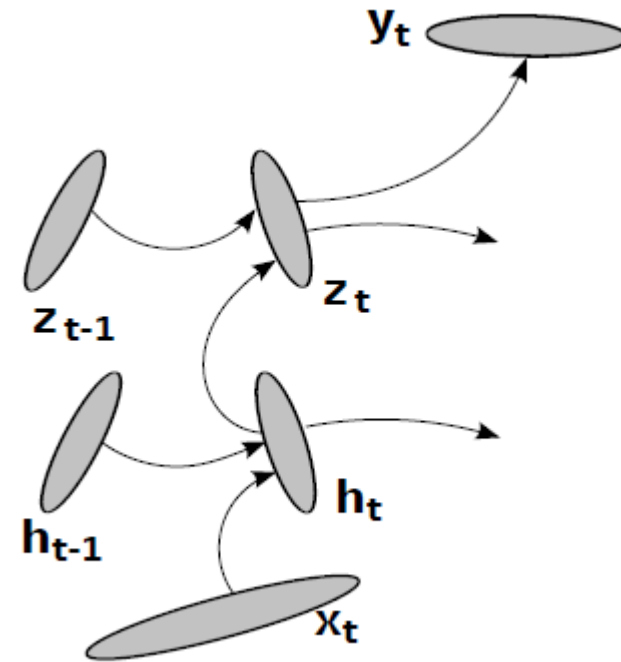
## 4. RNN EXTENSIONS

# DEEP RNN (1/2)

- Increase the **depth** of RNN to increase **expressive** power.
- N.B. here we add depth in **SPACE** (like FFNN), not in **TIME**.
- Pascanu, R., Gulcehre, C., Cho, K., & Bengio, Y, “How to construct deep recurrent neural networks”, 2013.



Standard RNN

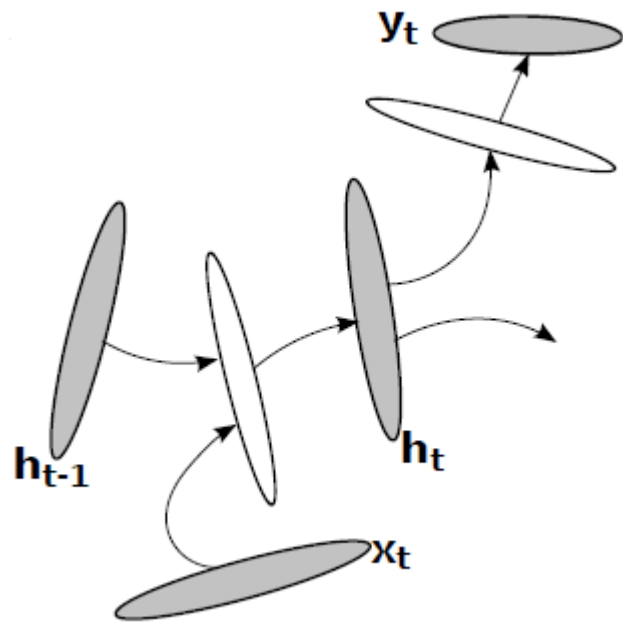


Stacked RNN

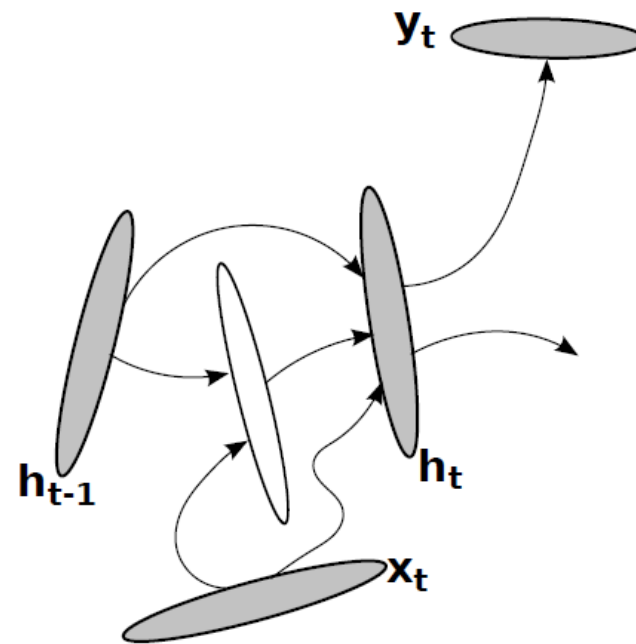
(learns different time-scales at each layer – from fast to slow dynamics)



## DEEP RNN (2/2)

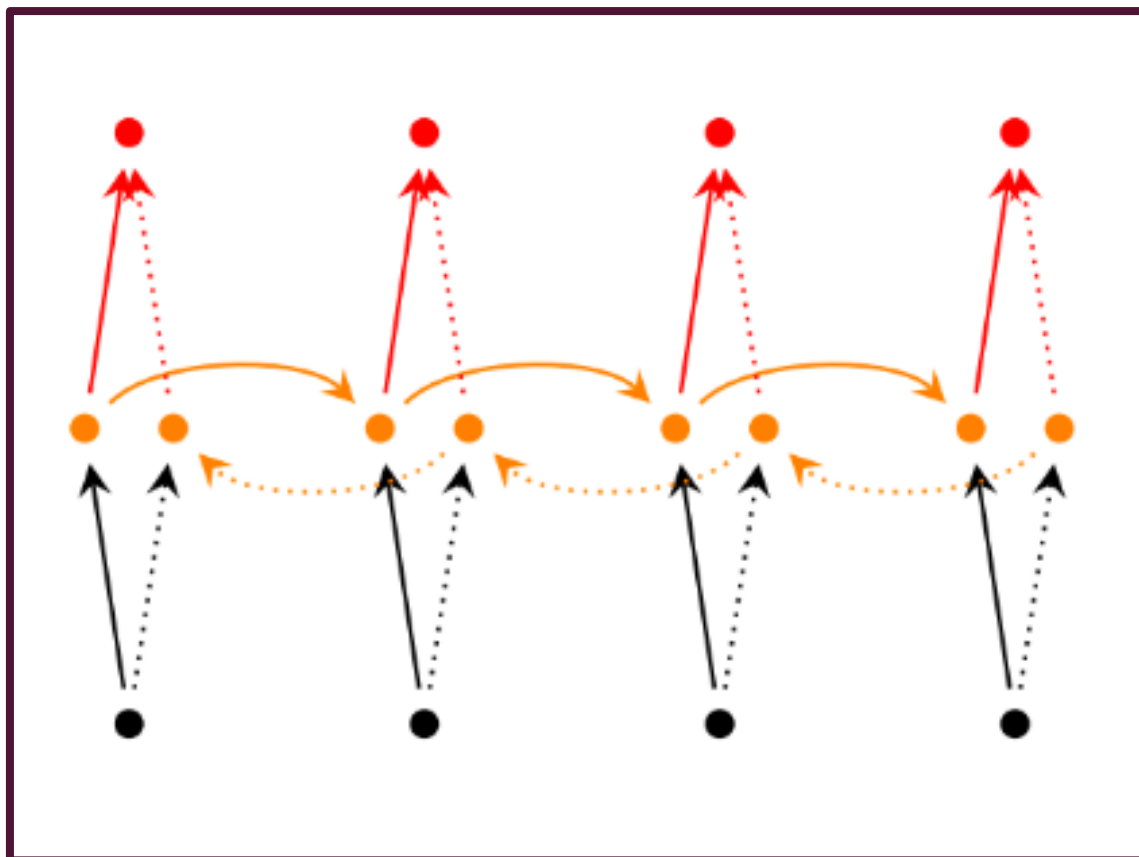


Deep input-to-hidden +  
Deep hidden-to-hidden +  
Deep hidden-to-output



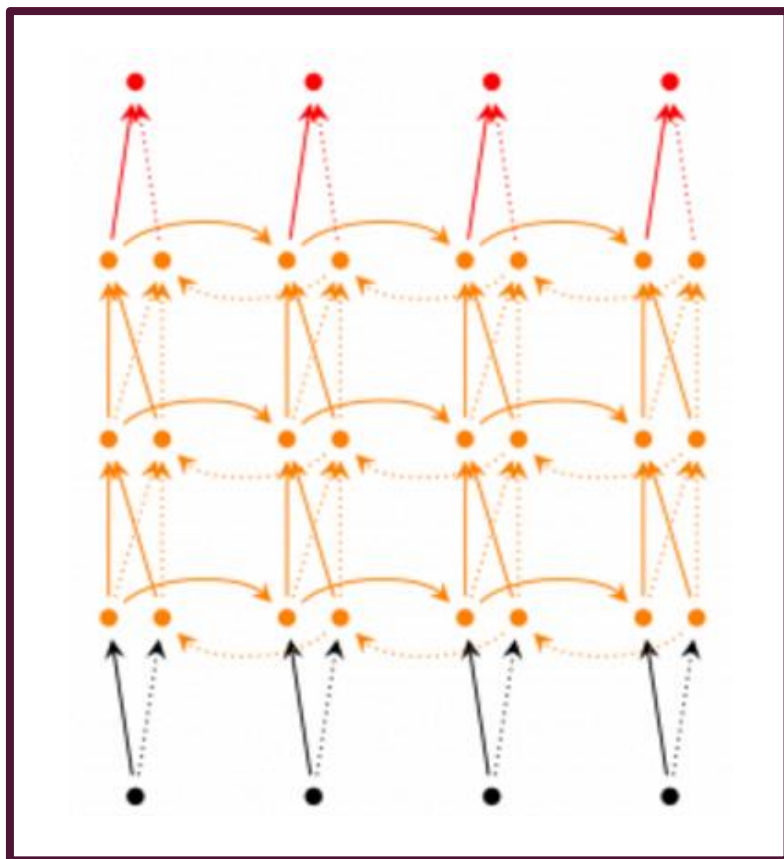
Deep input-to-hidden +  
Deep hidden-to-hidden +  
Shortcut connections (useful for letting the  
gradient flow faster during backpropagation).

# BIDIRECTIONAL RNN



- The **output** at time  $t$  may not only depend on the previous elements in the sequence, but also **future** elements.
- Example: to predict a missing **word** in a sequence you want to look at both the **left** and the **right** context.
- Two RNNs stacked on top of each other.
- Output computed based on the **hidden state** of **both** RNNs.
- *Huang Zhiheng, Xu Wei, Yu Kai. Bidirectional LSTM Conditional Random Field Models for Sequence Tagging.*

# DEEP (BIDIRECTIONAL) RNN



- Similar to Bidirectional RNNs, but with **multiple** hidden layers per time step.
- **Higher learning** capacity.
- Needs a **lot** of **training data** (the deeper the architecture the harder is the training).
- *Graves Alex, Navdeep Jaitly, and Abdel-rahman Mohamed. "Hybrid speech recognition with deep bidirectional LSTM.", 2013.*



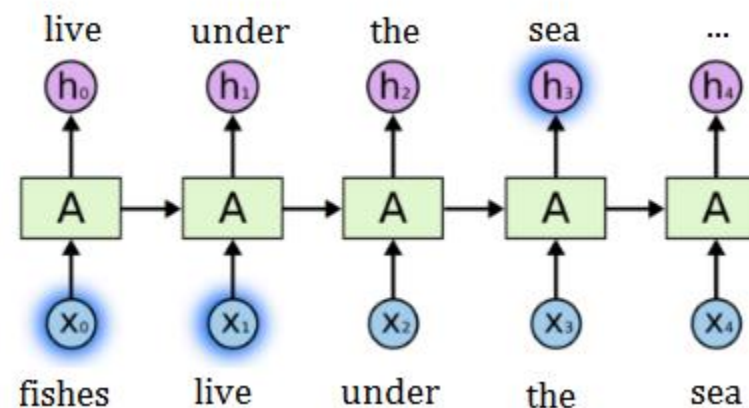
## 5. GATED RNNS

Inspired by: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

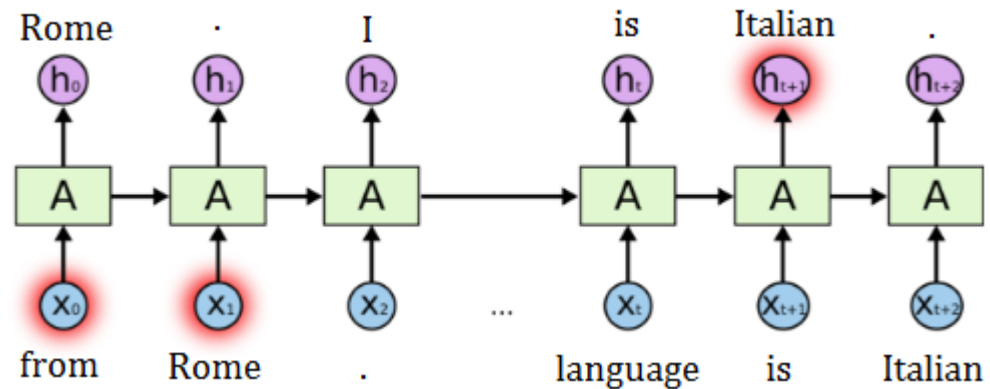
# LONG-TERM DEPENDENCIES

- Due to vanishing gradient, RNN are **incapable** of learning **long-term** dependencies.
- Some applications require **both short** and **long** term dependencies.
- Example: Natural Language Processing (**NLP**).
- In some cases, **short-term** dependencies are **sufficient** to make predictions.

- Consider to train the RNN to make 1-step ahead prediction.
- To predict the word '**sea**' it is sufficient to look only **3** step back in time.
- In this case, it is sufficient to **backpropagate** the **gradient 3** step back to successfully learn this task.



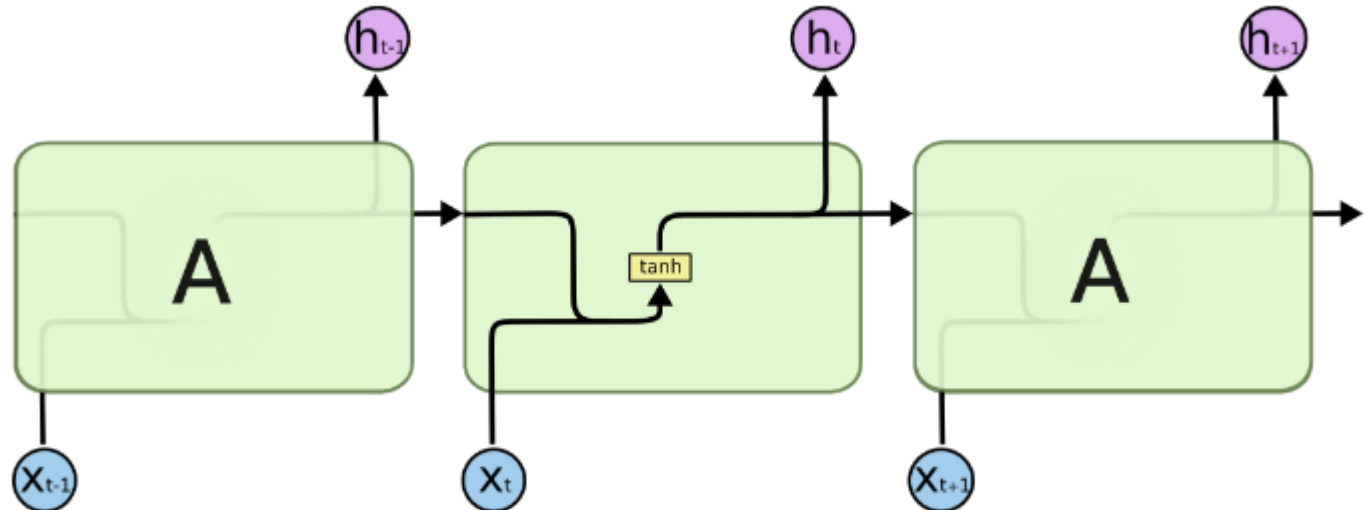
- Let's stick to 1-step ahead prediction.
- Consider the sentence: *I am from Rome. I was born 30 years ago. I like eating good food and riding my bike. My native language is Italian.*
- When we want to predict the word *Italian*, we have to look back **several time steps**, up to the word *Rome*.
- In this case, the short-term memory of the RNN would not do the trick.



- As the **gap** in **time** grows, the **harder** for an RNN become to **handle** the problem.
- Let's see how Long-Short Term Memory (**LSTM**) can handle this difficulty.

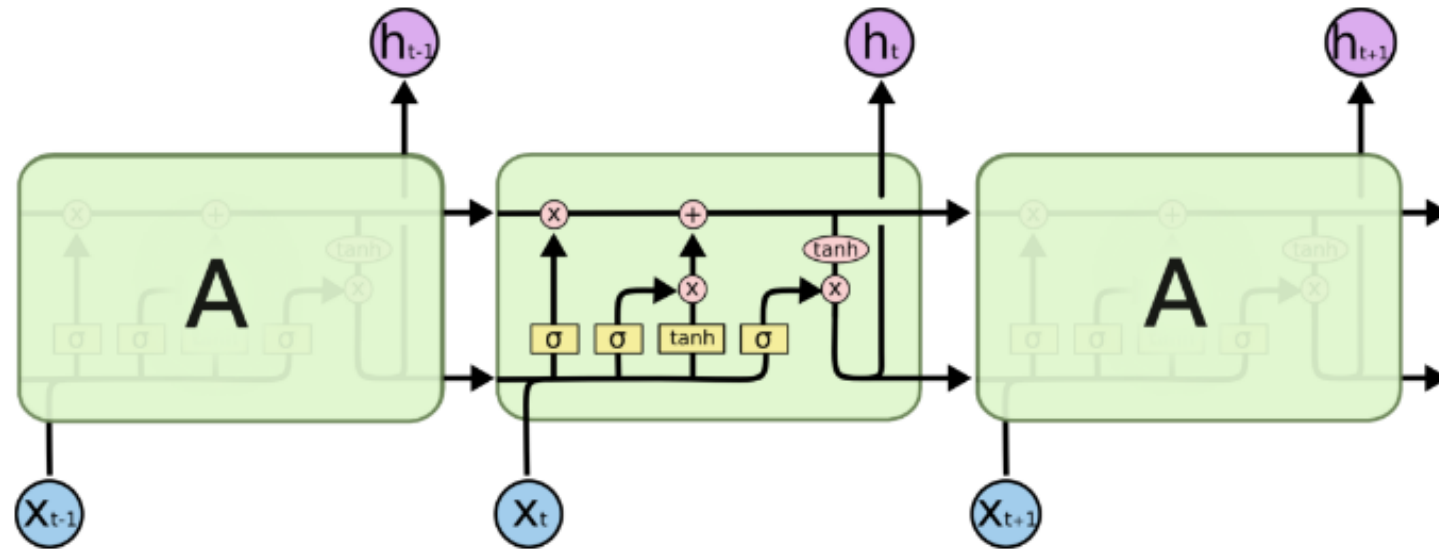
# LSTM OVERVIEW

- Introduced by Hochreiter & Schmidhuber (1997).
- Work very well on many different problems and are widely used nowadays.
- Like RNN, they must be unfolded in time to be trained and understood.
- Let's recall the unfolded version of a RNN.
- A very simple processing unit is repeated each time.

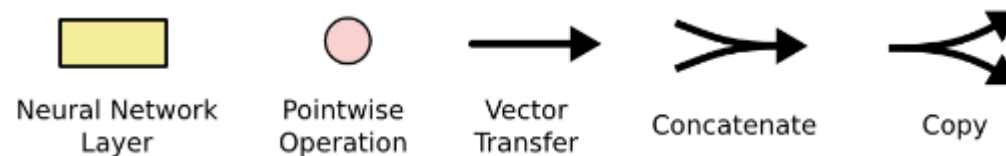


The repeating module in a standard RNN contains a single layer.

- The processing unit of the LSTM is more complex and is called *cell*.
- An LSTM cell is composed of 4 layers, interacting with each other in a special way.



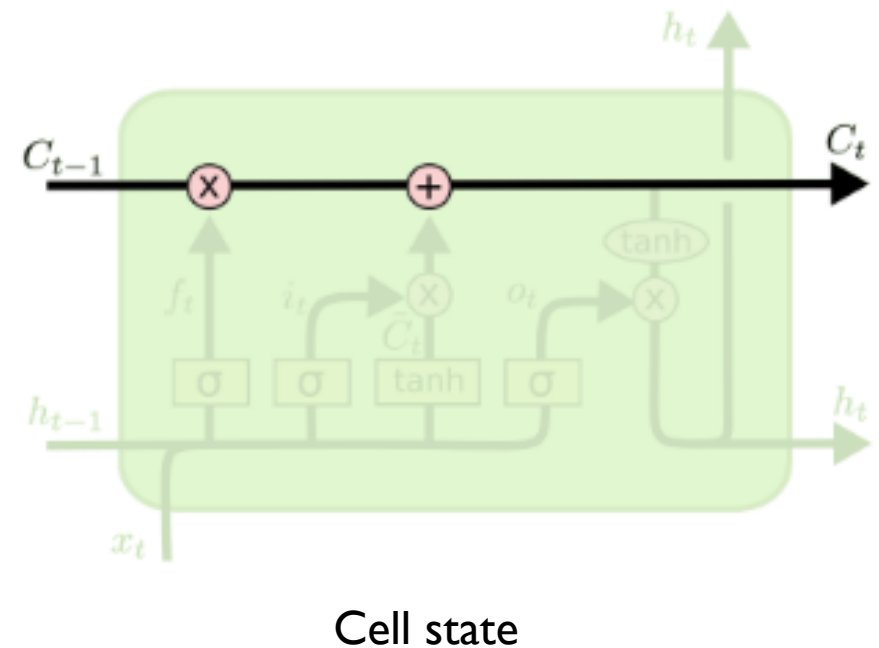
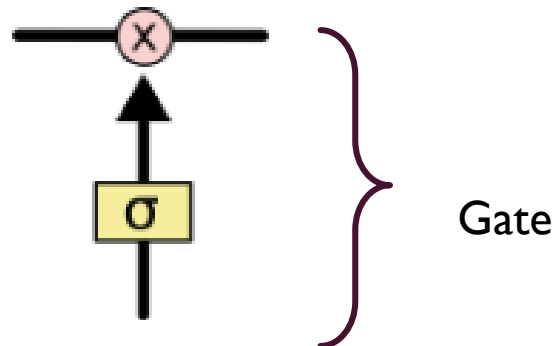
The repeating module in an LSTM contains four interacting layers.





# CELL STATE AND GATES

- The **state** of a cell at time  $t$  is  $C_t$ .
- The LSTM modify the state only through **linear interactions**: information flows **smoothly** across time.
- LSTM **protect** and **control** the information in the cell through 3 **gates**.
- Gates are implemented by a **sigmoid** and a **pointwise multiplication**.

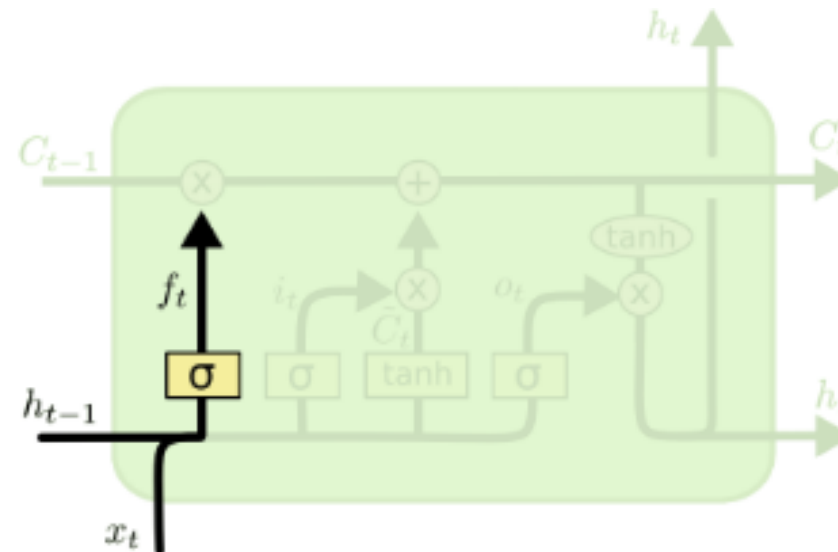


# FORGET GATE

- Decide what information should be **discarded** from the cell state.

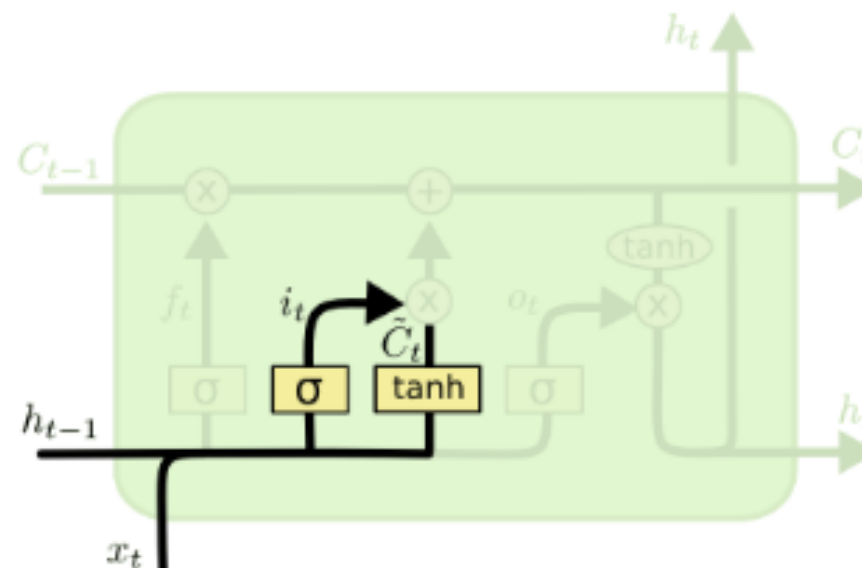
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) = \begin{cases} 0 & \rightarrow \text{completely get rid of content in } C_t \\ 1 & \rightarrow \text{completely keep the content in } C_t \end{cases}$$

- Gate controlled by current input  $x_t$  and past cell output  $h_{t-1}$ .
- NLP example: cell state keep the gender of the present subject to use correct pronouns.
- When sees a **new subject**, **forget the gender** of the old subject.



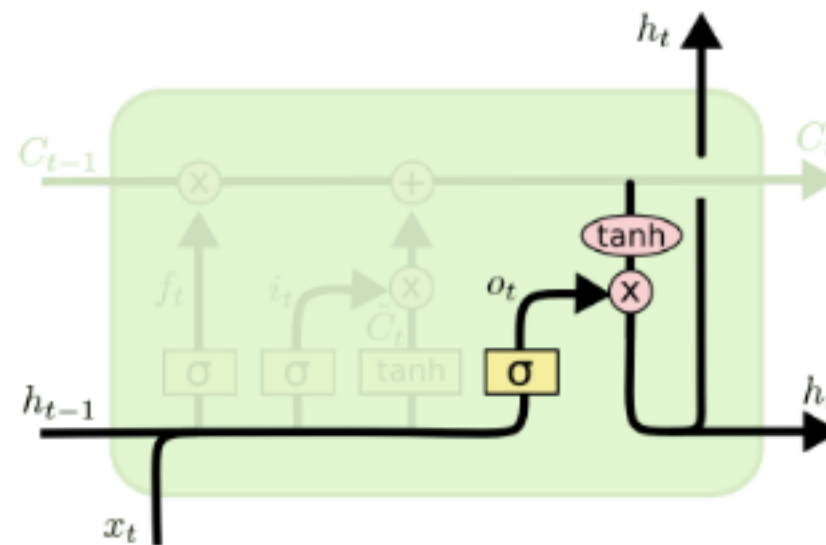
# UPDATE GATE

- With forget gate we decided whether or not to forget cell content.
- After, with update gate we decide **how much** to **update** the old state  $C_{t-1}$  with a **new candidate**  $\tilde{C}_t$ .
- Update gate:  $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) = \begin{cases} 0 \rightarrow \text{no update} \\ 1 \rightarrow \text{completely update} \end{cases}$
- New candidate:  $\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$ .
- New state:  $C_t = C_{t-1} + i_t * \tilde{C}_t$ .
- In the NLP example, we **update** the cell state as we **see** a **new subject**.
- Note that the new candidate is computed exactly like the **state in traditional RNN** (same difference equation).



# OUTPUT GATE

- The output is a **filtered** version of the **cell state**.
- Cell state is fed into a tanh, which squashed its values between -1 and 1.
- Then, the gate **select** the **part** to be returned as **output**.
- Output gate:  $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) = \begin{cases} 0 \rightarrow \text{no output} \\ 1 \rightarrow \text{return complete cell state} \end{cases}$
- Cell output:  $h_t = o_t * \tanh(C_t)$
- In NLP example, after having seen a subject, if a verb comes next, the cell outputs information about being singular or plural.



# COMPLETE FORWARD PROPAGATION STEP

- $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$  - forget gate
- $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$  - input gate
- $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$  - output gate
- $\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$  - new state candidate
- $C_t = C_{t-1} + i_t * \tilde{C}_t$  - new cell state
- $h_t = o_t * \tanh(C_t)$  - cell output
  
- Parameters of the model:  $\{W_i, W_c, W_o, b_i, b_c, b_o\}$
- Note that the weight matrix have a larger size than in normal RNN (they multiply the concatenation of  $x_t$  and  $h_t$ ).

# LSTM DOWNSIDES

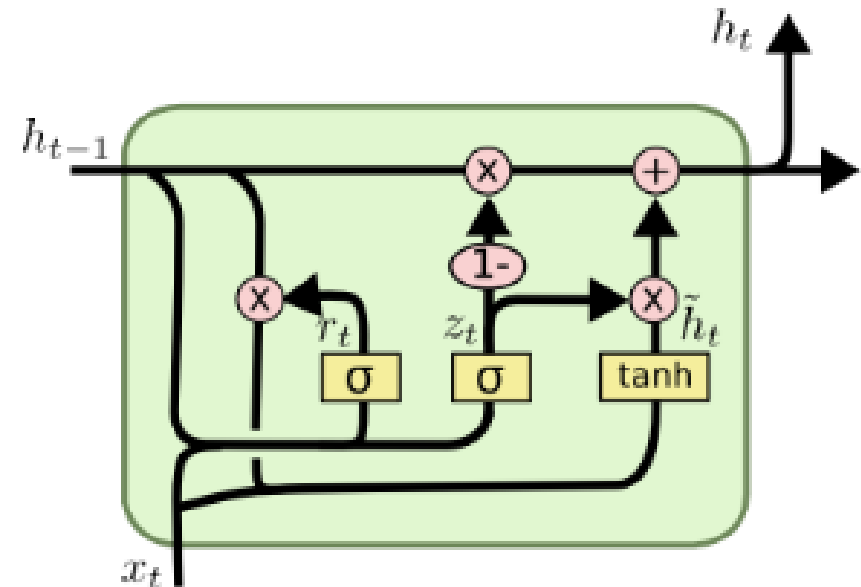
- Gates are never really 1 or 0. The content of the cell is inevitably corrupted after long time.
- Even if LSTM provides a huge improvement w.r.t. RNN, it still struggles with very long time dependencies.
- Number of parameters:  $4 \cdot (N_i + 1) \cdot N_o + N_o^2$ .
- Example: input = time series of 100 elements, LSTM units = 256 → 168960 parameters.
- Scales up quickly!! Lot of parameters = **lot of training data**.
- Rule of thumb: data elements must always be one order of magnitude greater than the number of parameters.
- **Memory** problems when dealing with lot of data.
- **Long training** time (use GPU computing).

# GATED RECURRENT UNITS

- Several LSTM variants exists.
- GRU is one of the most famous alternative architectures (Cho, et al. (2014)).
- It combines the **forget** and **input** gates into a single **update** gate.
- It also **merges** the **cell state** and **hidden state**, and makes some other changes.
- The cell is characterize by **fewer** parameters.

# GRU FORWARD PROPAGATION STEP

- $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$  - reset gate (merge of input and forget gate).
  - $z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$  - output gate.
  - $\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$  - new candidate output (merge internal state and output).
  - $h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$  - cell output
- 
- Performs better LSTM or GRU?
  - Depends on the problem at hand: *Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling.", 2014.*
  - 'No free lunch' theorem ☺







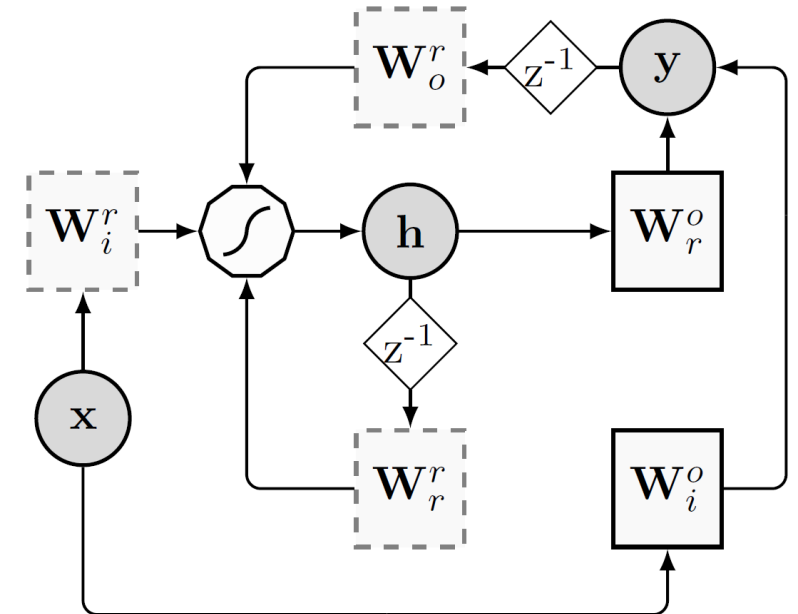
## 6. ECHO STATE NETWORKS

# ESN OVERVIEW

- Introduced by Jaeger (2001).
- With Liquid State Machine, they form the family of Reservoir Computing approaches.
- Large, untrained recurrent layer (reservoir).
- It has to be sparse and as much heterogeneous as possible to generate a large variety of internal dynamics.
- Linear, memoryless readout trained with linear regression, picks the dynamics useful to solve the task at hand.
- State of the art in real-valued time series prediction.
- Pros:
  - Fast and easy to train (no backpropagation).
  - Relatively small, they are used in embedded systems.
- Cons:
  - High sensitivity to hyperparameters (model parameters, usually set by hand).
  - Random initialization add stochasticity to the results.

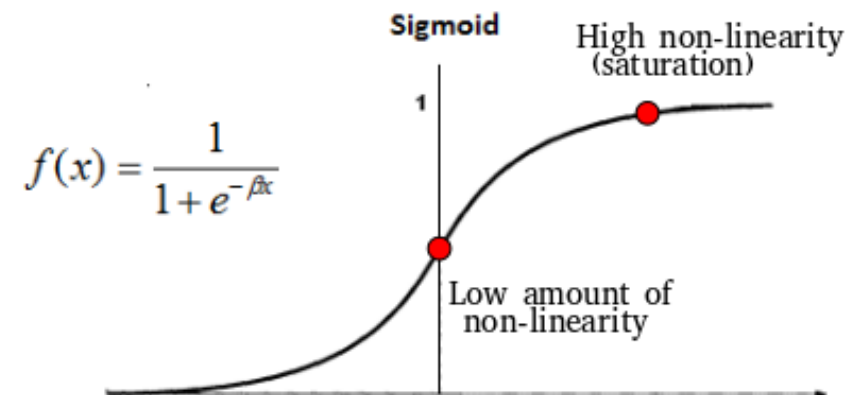
- Dashed boxes are randomly initialized weights, left untrained.
- Solid boxes are readout weights, usually trained with linear regression.
- Outputs of the network are forced to match target values.
- $L_2$  regularization is used in the linear regression to prevent overfitting.
- ESN state and output update:
 
$$h[t + 1] = f(W_r^r h[t] + W_i^r x[t + 1] + W_o^r y[t])$$

$$y[t + 1] = g(W_r^o h[t + 1] + W_i^o x[t + 1])$$
- $f()$  is usually implemented as a tanh.
- $g()$  is usually the identity function (more complicated readouts are possible though).



# PRINCIPAL ESN HYPERPARAMETERS

- Reservoir size:
  - Should be large enough to provide a great variety of dynamics.
  - If too large, there could be overfit (the readout just learn a simple mapping 1-to-1 from network state to output (effect is damped by regularization)).
- Reservoir spectral radius:
  - Is the largest eigenvalue of the reservoir matrix.
  - Controls the dynamics of the system. Should be tuned to provide a stable dynamics, yet sufficient rich (edge of chaos).
- Input scaling:
  - Controls the amount of nonlinearity introduced by the neurons.
  - High value → high amount of nonlinearity.



People whose work (partially) inspired this presentation.



Y. Bengio



G. Hinton



J. Schmidhuber



A. Graves



C. Olah



A. Karpathy



# THE END

THANKS FOR THE ATTENTION