Applied Deep Learning

Chapter 6: Recurrent NNs

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Fall 2025

Learning from Sequence Data

In many applications, we have *sequence data*, *e.g.*,

- *speech data which is usually a long time series*
- *text data which is sequence of words and letters*
- *financial data that is typically a time-dependent sequence of values*

Learning from such data can be inefficient via FNNs, i.e., MLPs and CNNs

- *On one hand, we have long sequence*
  ↳ *This can easily make the NN size infeasible*
- *On another hand, we do not have so much features*
  ↳ *Just think of a long text, where we need to predict the next word in it*

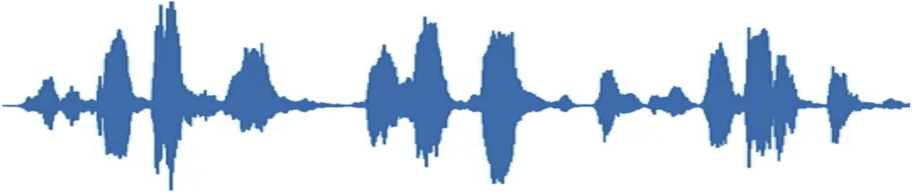
We need to develop some techniques to handle such data

First, let's see some examples!

Learning from Sequence Data: *Example I*

Assume we listen to a 15-minute talk: we want to find out whether it is about sport or science

- *This is a classification problem*
  - ↳ *The data-point, i.e., talk is classified either as sport or science*
- *What about the data?*
  - ↳ *We sample the audio signal at rate 44.1 kHz and quantize the samples*
  - ↳ *We put every N successive samples in a frames $\equiv$ vector of samples*
  - ↳ *We store the 15-minute talk as a sequence of time frames*
  - ↳ *We may also store frequency frames from the Fourier transform*



$\mathbf{x}[1]$ $\mathbf{x}[2]$ $\mathbf{x}[3]$ $\cdots$ $\mathbf{x}[t]$ $\mathbf{x}[t+1]$ $\cdots$

Learning from Sequence Data: *Example I*

Assume we listen to a 15-minute talk: we want to find out whether it is about sport or science

- *This is a binary classification problem*
- *What about the data? each data-point is a sequence of vectors*

Say we make frames of 512 samples; then, we roughly have

$$77{,}587 \text{ frames} = 39{,}690{,}000 \text{ samples}$$

But do we need to pass them all together through an NN?

- *It does not seem to be!*
  - ↳ *We can classify based of simple words and expressions in the talk*
  - ↳ *Processing all samples together seems to be an unnecessary hardness*
- *It does not generalize well*
  - ↳ *We want to classify shorter and longer talks as well*

Learning from Sequence Data: *Example II*

Now let's consider another example: *we have a long text and want to learn what is the next word in the sentence*

- *This is a prediction task*
  - ↳ *Given previous text we predict the next outcome*
- *What about the data?*
  - ↳ *We parse the text into a sequence of words*
  - ↳ *We represent the letters of each word with their numeric, e.g., ASCII*
  - ↳ *We save each word into an N-dimensional frame*

`...therapy.  UofT undergraduate students explore the use of AI to treat speech`

$\mathbf{x}[1] \quad \mathbf{x}[2] \quad \mathbf{x}[3] \quad \cdots \quad \mathbf{x}[t] \quad \mathbf{x}[t+1] \quad \cdots \quad \mathbf{x}[T] \quad \mathbf{y} = ?$

Learning from Sequence Data: *Example III*

Another example: *we have a sequence of stock prices and are interested in the future price*

- *This is again a prediction task*
  - ↳ *Given previous prices we predict the future price*
- *What about the data?*
  - ↳ *We put daily prices in form of a sequence*
  - ↳ *We collect every couple of prices along with other indicators into a vector*

In all these problems: *we have a sequence of data and we intend to learn from them in a generalizable way*

↳ *We clearly need a memory component that can potentially be infinite*

↳ *We should keep track of this infinitely large memory via limited storage*

Predicting Next Word

Let's start with a simple example: *we want to train a neural network that gets a sentence and complete the next word*

...	$\mathbf{x}[t-6]$	$\mathbf{x}[t-5]$	$\mathbf{x}[t-4]$	$\mathbf{x}[t-3]$	$\mathbf{x}[t-2]$	$\mathbf{x}[t-1]$	$\mathbf{x}[t]$	$\mathbf{y} = \mathbf{x}[t+1]$
...	`Julia`	`has` `been`	`nominated`	`to` `receive`	`Alexander`	`von` `Humboldt`	`Prize` `for`	`her`

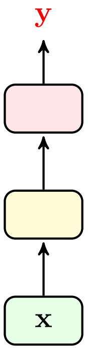
- + *How does the training dataset look like?*
- – We are given with several long texts in the same context: *at each entry of sequence in each of these texts the whole sequence is the data-point and the next word is label*

Predicting Next Word

Let's make some specification to clarify the problem

- *Each entry is a vector of dimension N, i.e., $\mathbf{x}[t] \in \mathbb{R}^N$*
- *We have an NN with some hidden layers to train*

We show our NN compactly with the following diagram

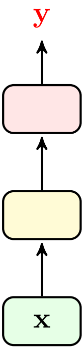


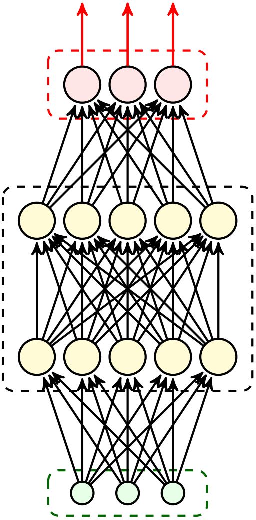
In this diagram

- *Green box shows the input layer*
- *Yellow box includes hidden layers*
  - ↳ *It could be several layers*
- *Red box is the output layer*
- *Arrows refer to all links between the layers*
  - ↳ *They could be learnable*

Predicting Next Word

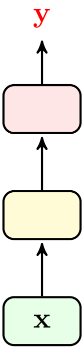
For instance, we could think of following equivalence





Predicting Next Word: *MLP*

Let's try solving this problem with a simple MLP



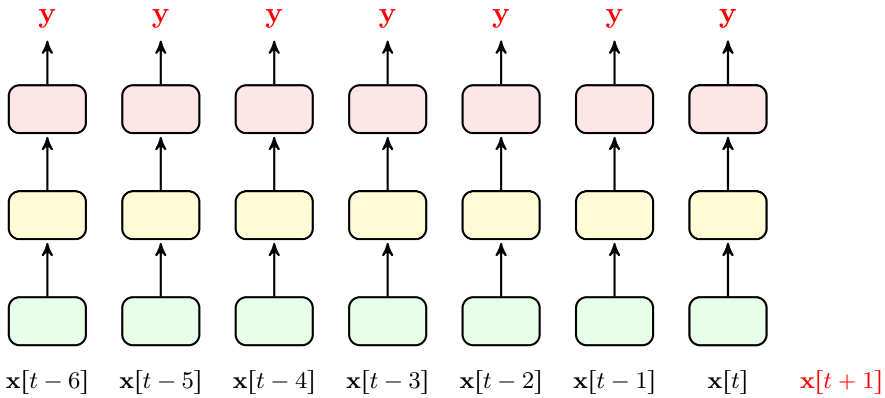
We have a fully-connected FNN that

- *takes N inputs, i.e., single entry*
- *returns N outputs, i.e., predicted next word*

We train this MLP

- *we go over all text*

Predicting Next Word: *MLP*



`... Julia has been nominated to receive Alexander von Humboldt Prize for her`

Does FNN predict "`her`*"? No! How can it remember we are talking about Julia?!*

Predicting Next Word: *MLP*

$\mathbf{x}[t-6]$ $\mathbf{x}[t-5]$ $\mathbf{x}[t-4]$ $\mathbf{x}[t-3]$ $\mathbf{x}[t-2]$ $\mathbf{x}[t-1]$ $\mathbf{x}[t]$ $\mathbf{x}[t+1]$

`... Julia has been nominated to receive Alexander von Humboldt Prize for her`

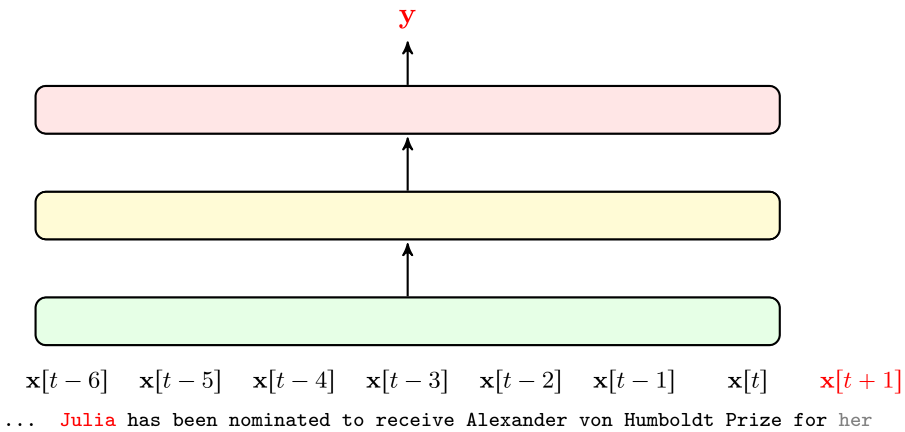
Each time the FNN gets trained with a new sample, *it forgets previous text*

- *At the end, it has a set of weights that are average over all predictions*
  - ↳ *many of these predictions are irrelevant, e.g.,*
    - ↳ "`nominated to`" *is followed by* "`receive`": *has nothing to say about* "`her`"
- *By the time we get to* $\mathbf{x}[t]$, *the FNN gets no input that connects it to Julia*

> This indicates that we need to *make a memory component for our NN*

Predicting Next Word: *Large MLP*

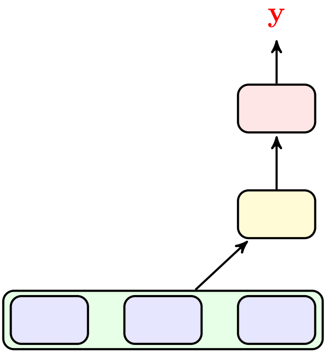
Maybe we could give more inputs to the FNN!



But, what if Julia has been mentioned 10 pages ago? Forget about large MLPs!

Predicting Next Word: *CNN*

Let's now think about CNNs



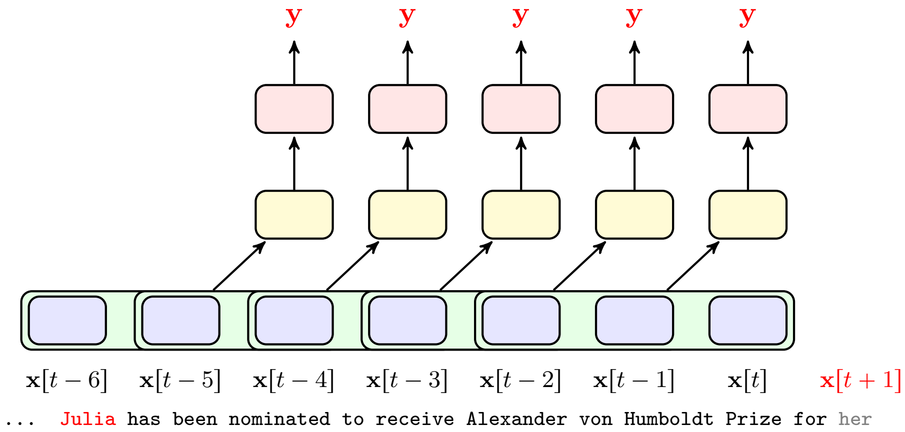
We have a fully-connected FNN that

- *takes N inputs, i.e., single entry*
- *returns N outputs, i.e., predicted next word*

We use convolution to

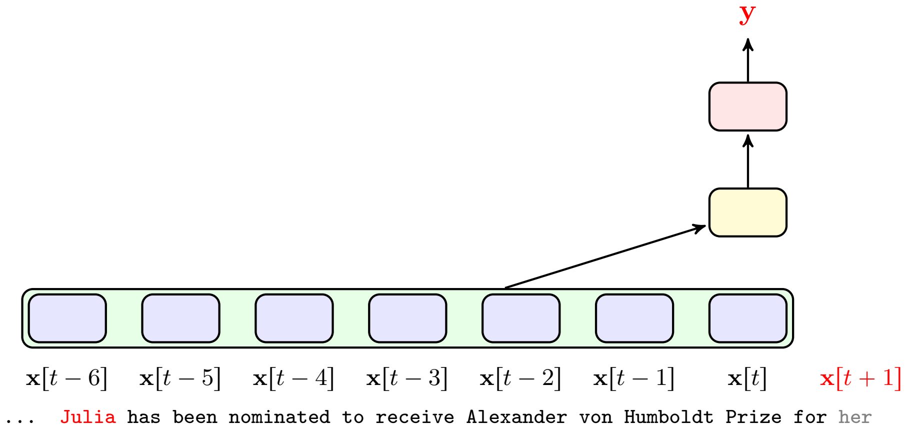
- *look into a larger input by a filter of size N*
- *extract features from a larger part of text and pass it to hidden layers*

Predicting Next Word: *CNN*



Yet, it doesn't seem to remember Julia! Unless we slide over the whole text!

Predicting Next Word: *Large CNN*



Though better than MLP, it is still infeasible to track the whole text

Finite Memory: *Root Problem*

$\mathbf{x}[t-6]$ $\mathbf{x}[t-5]$ $\mathbf{x}[t-4]$ $\mathbf{x}[t-3]$ $\mathbf{x}[t-2]$ $\mathbf{x}[t-1]$ $\mathbf{x}[t]$ $\mathbf{x}[t+1]$

`... Julia has been nominated to receive Alexander von Humboldt Prize for her`

The problem with *all architectures we know* is that *they have finite memory*

- *They can only remember from their input*
  - ↳ *we always give them independent inputs with similar features*
  - ↳ *they gradually learn to connect any of such inputs to their label*
- *If we need to remember for long time we have to give them huge inputs*
- *But the memory component does not seem to be so huge*
  - ↳ *We may only remember that Julia is a "single person" and "female"*
  - ↳ *If text now switches to Theodore we should refresh our memory that we are talking about a "single person" and "male"*

Finite Memory Component *with Infinite Response*

Component We Miss

We need to extract a memory component from our data that is finite in size but has been influenced (at least theoretically) infinitely

State-space model can help us building such memory component: *it is widely used in control theory to describe evolution of a system over time*

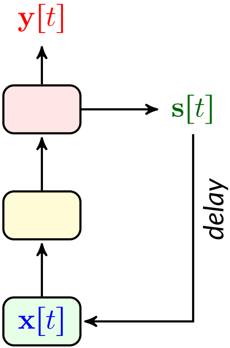
Assume $\mathbf{x}[t]$ is an input to a system at time t: the system returns an output $\mathbf{y}[t]$ to this input and a state variable $\mathbf{s}[t]$. The output in the next time, i.e., $t+1$, depends on the new input and current state, i.e.,

$$\mathbf{y}[t+1], \mathbf{s}[t+1] = f\left(\mathbf{x}[t+1], \mathbf{s}[t]\right)$$

In the above representation: *the state is a finite-size variable that carries information for infinitely long time*

State-Space Model for *NNs*

We can look at a NN as a state-dependent system



In this architecture, the NN

- *takes* $\mathbf{x}[t]$ *and previous state* $\mathbf{s}[t-1]$ *as inputs*
- *returns* $\mathbf{y}[t]$ *and new state* $\mathbf{s}[t]$ *as outputs*

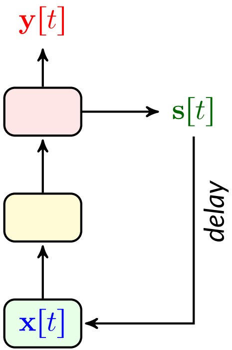
This NN captures temporal behavior of data: starting from $t = 1$*, the NN*

- *initiates some* $\mathbf{s}[0]$
- *computes* $\mathbf{y}[1]$ *and* $\mathbf{s}[1]$ *from time sample* $\mathbf{x}[1]$ *and* $\mathbf{s}[0]$
- *computes* $\mathbf{y}[2]$ *and* $\mathbf{s}[2]$ *from* $\mathbf{x}[2]$ *and* $\mathbf{s}[1]$

  $\ldots$

State-Space Model for *NNs*

Theoretically, this NN has infinite time response



Say NN initiates with some $\mathbf{s}[0]$. *It takes only sample* $\mathbf{x}[1]$ *and no other time samples is given to it. At time* t,

- $\mathbf{y}[t]$ *depends on* $\mathbf{s}[t-1]$
- $\mathbf{s}[t-1]$ *depends on* $\mathbf{s}[t-2]$

  $\cdots$
- $\mathbf{s}[1]$ *depends on* $\mathbf{x}[1]$

This means that $\mathbf{y}[t]$ *still remembers* $\mathbf{x}[1]$

State-Dependent NNs

It seems that for our purpose *state-space model helps extracting a good memory component. The challenge is to design a good state-dependent NN*

+ *Why is it a challenge? We make an NN with input* $(\mathbf{x}, \mathbf{s})$ *and output* $(\mathbf{y}, \mathbf{s}')$ *and then train it!*
- That sounds easy, but have some challenges

Design of state-dependent NNs has two main challenges

1. *Defining the state variable*
   ↳ *how should we specify the state as we do not have a clear clue about it*
2. *Training the NN over time: say we get a time sequence data and compute the empirical risk by averaging over time samples up to time* t
   ↳ *empirical risk depends on weights in the NN*
   ↳ *it also depends previous memory components which can be learnable*
      ↳ *if we want to train the model efficiently, we need to get back over time and update all those memory components!*

State-Dependent NNs

Several attempts have been done: *we look briefly into two of them*

- *Jordan Network*
  - ↳ *proposed by Michael Jordan*[1] *in 1986*
  - ↳ *it computes the state variable to be a simple moving average*
- *Elman Network*
  - ↳ *proposed by Jeffrey Elman*[2] *in 1990*
  - ↳ *it learns the state variable but does not track back completely over time*

These models are simple forms of what we nowadays know as

Recurrent NNs $\equiv$ *RNNs*

RNN

RNN is an NN with state-variable: the term recurrent refers to the connection between former state and new output

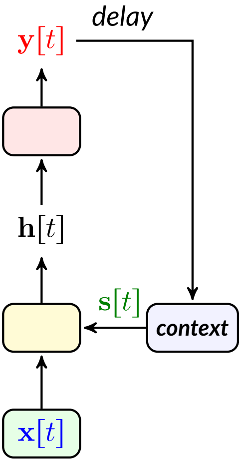
---

[1] Professor at CS Department of University of Berkeley; check his page

[2] Late Professor at UCSD ('48 – '18); check his page

Jordan Model

Jordan Network uses a simple moving average of output as the state



The proposal was a shallow NN

- *It starts with some initial state* $\mathbf{s}[1]$
- *In time* t, *it updates the state* $\mathbf{s}[t]$ *with fixed* μ *as*

$$\mathbf{s}[t] = \mu\mathbf{s}[t-1] + \mathbf{y}[t-1]$$

- *The hidden layer then computes*

$$\mathbf{h}[t] = f\left(\mathbf{W}_1\mathbf{x}[t] + \mathbf{W}_{\mathrm{m}}\mathbf{s}[t]\right)$$

- *The output is*

$$\mathbf{y}[t] = f\left(\mathbf{W}_2\mathbf{h}[t]\right)$$

Jordan Network

Jordan Network has a memory component with infinte response: *say we set initial state to zero, give $\mathbf{x}[1]$, and keep the input zero for the rest of time; then,*

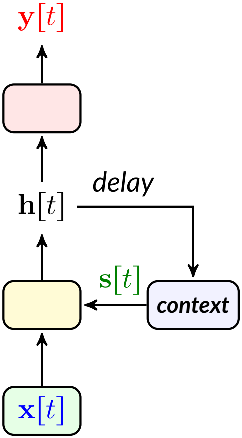
$$
\begin{aligned}
\mathbf{y}[1] &= f\left(\mathbf{W}_2 f\left(\mathbf{W}_1 \mathbf{x}[1]\right)\right) \\
\mathbf{y}[2] &= f\left(\mathbf{W}_2 f\left(\mathbf{W}_{\mathrm{m}} \mathbf{y}[1]\right)\right) = f\left(\mathbf{W}_2 f\left(\mathbf{W}_{\mathrm{m}} f\left(\mathbf{W}_2 f\left(\mathbf{W}_1 \mathbf{x}[1]\right)\right)\right)\right) \\
&\vdots
\end{aligned}
$$

But, the network *does not learn how to remember*

- *It takes a fixed moving average as memory component*
- *It only learns how to use this pre-defined memory for prediction*
  - ↳ *We could say that it implicitly learns to remember by learning* $\mathbf{W}_{\mathrm{m}}$

Elman Network

Elman Network uses output of hidden layer as state: also called hidden state



The proposal was a shallow NN

- *It starts with some initial hidden state $\mathbf{h}[0]$*
- *In time t, it updates the state $\mathbf{s}[t]$ as*

$$\mathbf{s}[t] = \mathbf{h}[t-1]$$

- *The hidden layer then computes*

$$\mathbf{h}[t] = f\left(\mathbf{W}_1 \mathbf{x}[t] + \mathbf{W}_{\mathrm{m}} \mathbf{s}[t]\right)$$

- *The output is*

$$\mathbf{y}[t] = f\left(\mathbf{W}_2 \mathbf{h}[t]\right)$$

Elman Network

Similar to Jordan Network, Elman Network has a memory component with infinte response: *with zero initial hidden state, we have*

$$
\begin{aligned}
\mathbf{y}[1] &= f\left(\mathbf{W}_2 f\left(\mathbf{W}_1 \mathbf{x}[1]\right)\right) \\
\mathbf{y}[2] &= f\left(\mathbf{W}_2 f\left(\mathbf{W}_{\mathrm{m}} \mathbf{h}[1]\right)\right) = f\left(\mathbf{W}_2 f\left(\mathbf{W}_{\mathrm{m}} f\left(\mathbf{W}_1 \mathbf{x}[1]\right)\right)\right) \\
&\vdots
\end{aligned}
$$

Elman network also learns how to remember only implicitly

Challenge of Learning Through Time

Though Jordan and Elman Networks had memory, *they did not get trained accurately over time, i.e., they simplified the solution to the second challenge*

+ *What is really this challenge?*
– We are going to deal with it in next sections, but let's see it on these simple networks first

---

Let's assume a simple setting: *we are to train our NN on single data sequence*

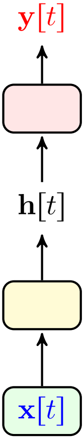
- *We have the sequence* $\mathbf{x}[1], \ldots, \mathbf{x}[T]$ *as the data-point*
- *For each entry of this sequence we have the true label*
  - ↳ *We have sequence* $\mathbf{v}[1], \ldots, \mathbf{v}[T]$ *with* $\mathbf{v}[t]$ *being label of* $\mathbf{x}[t]$
- *We are able to compute the loss between outputs and true labels as*[3]

$$\hat{R} = \sum_{t=1}^{T} \mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[t]\right)$$

[3]We will see that this is not always this easy!

Recap: *Basic FNN*

For sake if comparison, let's first train a basic FNN on this data sequence



We have a shallow FNN

- *The hidden layer computes*

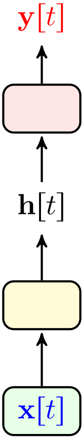
$$\mathbf{h}[t] = f\left(\mathbf{W}_1 \mathbf{x}[t]\right)$$

- *The output is*

$$\mathbf{y}[t] = f\left(\mathbf{W}_2 \mathbf{h}[t]\right)$$

Recap: *Basic FNN*

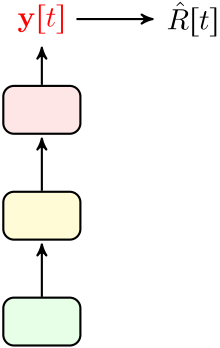
For sake if comparison, let's first train a basic FNN on this data sequence



How do we train this FNN?

- *We compute the gradients* $\nabla_{\mathbf{W}_1} \hat{R}$ *and* $\nabla_{\mathbf{W}_2} \hat{R}$
  - ↳ *We do it via backpropagation*
- *We apply gradient descent*

Learning Through Time: *FNNs*



$\mathbf{x}[1] \quad \mathbf{x}[2] \quad \cdots \quad \mathbf{x}[t-1] \quad \mathbf{x}[t] \quad \cdots \quad \mathbf{x}[T-1] \quad \mathbf{x}[T]$

$$\hat{R} = \sum_{t=1}^{T} \underbrace{\mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[t]\right)}_{\hat{R}[t]} = \sum_{t=1}^{T} \hat{R}[t] \rightsquigarrow \nabla_{\mathbf{w}_i} \hat{R} = \sum_{t=1}^{T} \nabla_{\mathbf{w}_i} \hat{R}[t]$$

Learning Through Time: *FNNs*

Let's learn $\mathbf{W}_1$ and to ease computation we use our cheating notation, i.e., *use $\circ$ to show any product: to compute the gradient we start with the output*

$$\nabla_{\mathbf{W}_1} \hat{R}[t] = \nabla_{\mathbf{W}_1} \mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[t]\right)$$
$$= \nabla_{\mathbf{y}[t]} \hat{R}[t] \circ \nabla_{\mathbf{W}_1} \mathbf{y}[t]$$

We know that $\mathbf{y}[t] = f\left(\mathbf{W}_2 \mathbf{h}[t]\right)$, so we can write

$$\nabla_{\mathbf{W}_1} \mathbf{y}[t] = \nabla_{\mathbf{h}[t]} \mathbf{y}[t] \circ \nabla_{\mathbf{W}_1} \mathbf{h}[t]$$
$$= \left(\dot{f}\left(\mathbf{W}_2 \mathbf{h}[t]\right) \circ \mathbf{W}_2\right) \circ \nabla_{\mathbf{W}_1} \mathbf{h}[t]$$

What about $\nabla_{\mathbf{W}_1} \mathbf{h}[t]$? We keep on backward!

Learning Through Time: *FNNs*

Up to now, we have

$$\nabla_{\mathbf{W}_1} \hat{R}[t] = \nabla_{\mathbf{y}[t]} \hat{R}[t] \circ \nabla_{\mathbf{h}[t]} \mathbf{y}[t] \circ \nabla_{\mathbf{W}_1} \mathbf{h}[t]$$

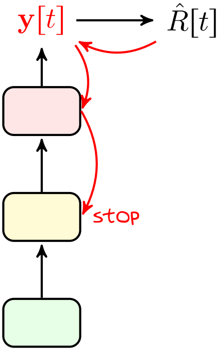
We use the fact that $\mathbf{h}[t] = f(\mathbf{W}_1 \mathbf{x}[t])$

$$\begin{aligned}\nabla_{\mathbf{W}_1} \mathbf{h}[t] &= \nabla_{\mathbf{W}_1} f(\mathbf{W}_1 \mathbf{x}[t]) + \nabla_{\mathbf{x}[t]} \mathbf{h}[t] \circ \nabla_{\mathbf{W}_1} \mathbf{x}[t] \\ &= \dot{f}(\mathbf{W}_1 \mathbf{x}[t]) \circ \mathbf{x}[t] + \left(\dot{f}(\mathbf{W}_1 \mathbf{x}[t]) \circ \mathbf{W}_1 \right) \circ \underbrace{\mathbf{0}}_{\mathbf{x}[t] \text{ is not a function of } \mathbf{W}_1}\end{aligned}$$

Therefore, we end chain rule here

$$\nabla_{\mathbf{W}_1} \hat{R}[t] = \nabla_{\mathbf{y}[t]} \hat{R}[t] \circ \nabla_{\mathbf{h}[t]} \mathbf{y}[t] \circ \nabla_{\mathbf{W}_1} \mathbf{h}[t]$$

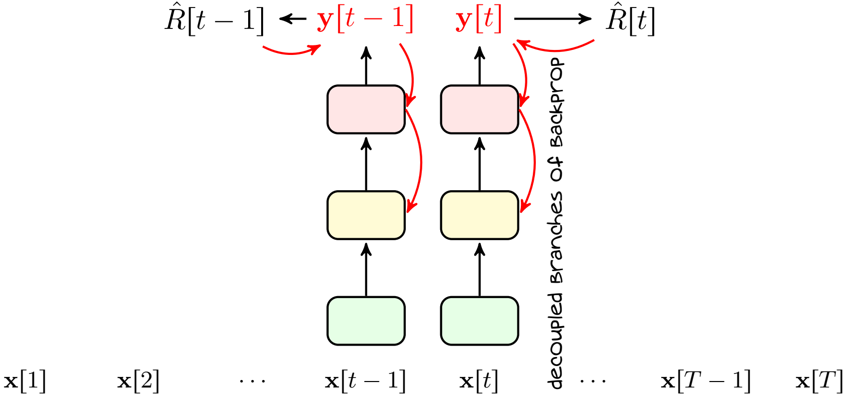
Learning Through Time: *FNNs*



$\mathbf{x}[1]$ $\mathbf{x}[2]$ $\cdots$ $\mathbf{x}[t-1]$ $\mathbf{x}[t]$ $\cdots$ $\mathbf{x}[T-1]$ $\mathbf{x}[T]$

$$\nabla_{\mathbf{W}_1} \hat{R}[t] = \nabla_{\mathbf{y}[t]} \hat{R}[t] \circ \nabla_{\mathbf{h}[t]} \mathbf{y}[t] \circ \nabla_{\mathbf{W}_1} \mathbf{h}[t]$$

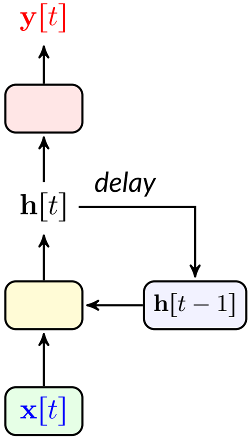
Learning Through Time: *FNNs*



$$\nabla_{\mathbf{w}_2} \hat{R} = \sum_{t=1}^{T} \nabla_{\mathbf{w}_2} \hat{R}[t]$$

Training a *Basic RNN*

Now, let's train Elman network on this sequence



The proposal was a shallow NN

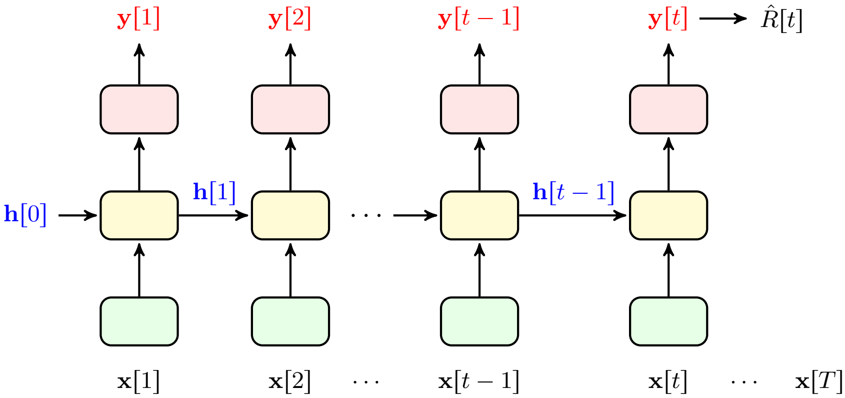
- *It starts with some initial hidden state* $\mathbf{h}[0]$
- *The hidden layer then computes*

$$\mathbf{h}[t] = f\left(\mathbf{W}_1 \mathbf{x}[t] + \mathbf{W}_{\mathrm{m}} \mathbf{h}[t-1]\right)$$

- *The output is*

$$\mathbf{y}[t] = f\left(\mathbf{W}_2 \mathbf{h}[t]\right)$$

Inferring Through Time: *Elman Network*



$$\hat{R} = \sum_{t=1}^{T} \underbrace{\mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[t]\right)}_{\hat{R}[t]} = \sum_{t=1}^{T} \hat{R}[t]$$

Learning Through Time: *Elman Network*

Let's again try to learn $\mathbf{W}_1$: *we start with the output*

$$\nabla_{\mathbf{W}_1} \hat{R}[t] = \nabla_{\mathbf{y}[t]} \hat{R}[t] \circ \nabla_{\mathbf{W}_1} \mathbf{y}[t]$$

We know that $\mathbf{y}[t] = f(\mathbf{W}_2 \mathbf{h}[t])$, *so we can write*

$$\begin{aligned}\nabla_{\mathbf{W}_1} \mathbf{y}[t] &= \nabla_{\mathbf{h}[t]} \mathbf{y}[t] \circ \nabla_{\mathbf{W}_1} \mathbf{h}[t] \\ &= \dot{f}(\mathbf{W}_2 \mathbf{h}[t]) \circ \mathbf{W}_2 \circ \nabla_{\mathbf{W}_1} \mathbf{h}[t]\end{aligned}$$

Next, we note that $\mathbf{h}[t] = f(\mathbf{W}_1 \mathbf{x}[t] + \mathbf{W}_{\mathrm{m}} \mathbf{h}[t-1])$

$$\begin{aligned}\nabla_{\mathbf{W}_1} \mathbf{h}[t] = \nabla_{\mathbf{W}_1} f(\mathbf{W}_1 \mathbf{x}[t] &+ \mathbf{W}_{\mathrm{m}} \mathbf{h}[t-1]) \\ &+ \nabla_{\mathbf{x}[t]} \mathbf{h}[t] \circ \underbrace{\nabla_{\mathbf{W}_1} \mathbf{x}[t]}_{\mathbf{0}} \\ &+ \nabla_{\mathbf{h}[t-1]} \mathbf{h}[t] \circ \underbrace{\nabla_{\mathbf{W}_1} \mathbf{h}[t-1]}_{?}\end{aligned}$$

Learning Through Time: *Elman Network*

Well! We know that $\mathbf{h}[t-1] = f\left(\mathbf{W}_1 \mathbf{x}[t-1] + \mathbf{W}_{\mathrm{m}} \mathbf{h}[t-2]\right)$

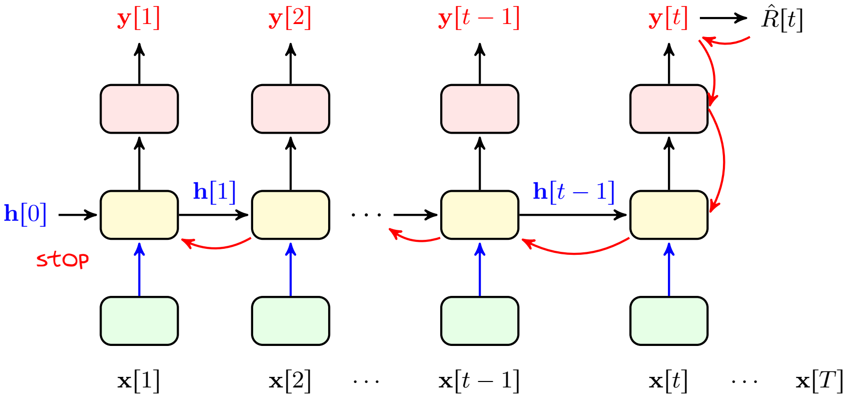
$$\begin{aligned}\nabla_{\mathbf{W}_1} \mathbf{h}[t-1] = \nabla_{\mathbf{W}_1} f\left(\mathbf{W}_1 \mathbf{x}[t-1] + \mathbf{W}_{\mathrm{m}} \mathbf{h}[t-2]\right) \\ + \nabla_{\mathbf{x}[t-1]} \mathbf{h}[t-1] \circ \underbrace{\nabla_{\mathbf{W}_1} \mathbf{x}[t-1]}_{\mathbf{0}} \\ + \nabla_{\mathbf{h}[t-2]} \mathbf{h}[t-1] \circ \underbrace{\nabla_{\mathbf{W}_1} \mathbf{h}[t-2]}_{?}\end{aligned}$$

We are not still done! We know $\mathbf{h}[t-2] = f\left(\mathbf{W}_1 \mathbf{x}[t-2] + \mathbf{W}_{\mathrm{m}} \mathbf{h}[t-3]\right)$

$$\begin{aligned}\nabla_{\mathbf{W}_1} \mathbf{h}[t-2] = \nabla_{\mathbf{W}_1} f\left(\mathbf{W}_1 \mathbf{x}[t-2] + \mathbf{W}_{\mathrm{m}} \mathbf{h}[t-3]\right) \\ + \nabla_{\mathbf{x}[t-2]} \mathbf{h}[t-2] \circ \underbrace{\nabla_{\mathbf{W}_1} \mathbf{x}[t-2]}_{\mathbf{0}} \\ + \nabla_{\mathbf{h}[t-3]} \mathbf{h}[t-2] \circ \underbrace{\nabla_{\mathbf{W}_1} \mathbf{h}[t-3]}_{?}\end{aligned}$$

Learning Through Time: *Elman Network*

We should in fact pass all the way back to the initial time interval time!



Note that all blue edges are representing $\mathbf{W}_1$

Learning Through Time

Moral of Story

To learn how to remember, we need to train our RNN through time: at each time interval, we should move all the way back to origin to find out how exactly we should change the weights!

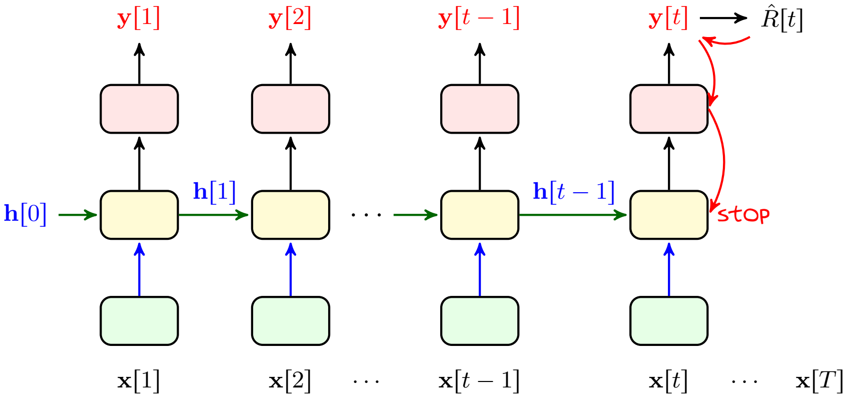
- + *Did Elman did so?*
- – Not really!

For training Elman treated the hidden state $\mathbf{h}[t-1]$ *as a fixed variable, i.e., he assumed* $\nabla_{\mathbf{W}_1}\mathbf{h}[t-1] \approx \nabla_{\mathbf{W}_\mathrm{m}}\mathbf{h}[t-1] = \mathbf{0}$*! So, he did not need to move backward in time!*

This means that Elman did not really addressed the second challenge!

Learning Through Time: *Elman's Approximation*

Elman treated it as a simple FNN with only one extra input!



Training $\mathbf{W}_1$ is exactly as FNN. We just have one extra $\mathbf{W}_\mathrm{m}$ here!

RNNs: *Need to Learn Memory*

Though appreciated, Elman and Jordan Networks did not do the job

1. *Their memory component is rather simple*
   ↳ *We should use deeper models that enable advanced memory components*
2. *They do not really learn how to remember*
   ↳ *We should train the memory component over time*

These led us to development of RNNs!

RNN: *Less Generic Definition*

An RNN can be designed with any known architecture by letting NN also learn from its past features and outputs. This new enabling is called recurrence