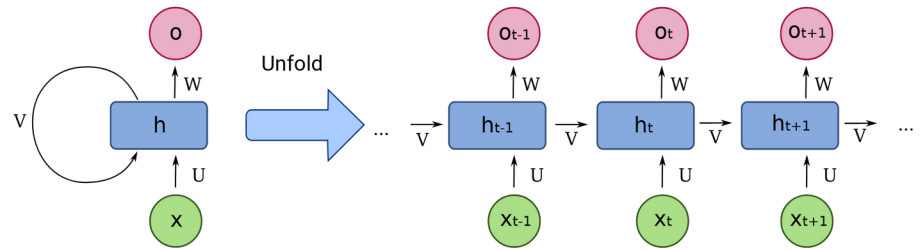
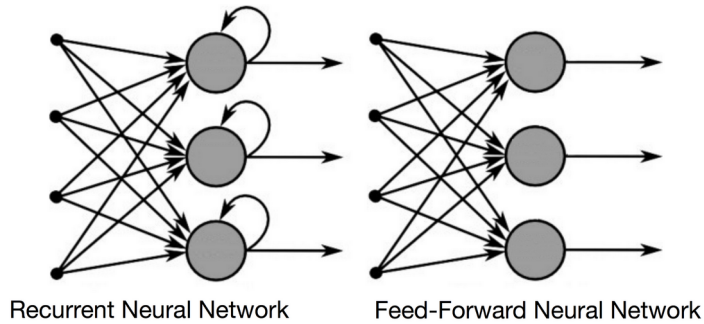


Recurrent Neural Networks (RNN)

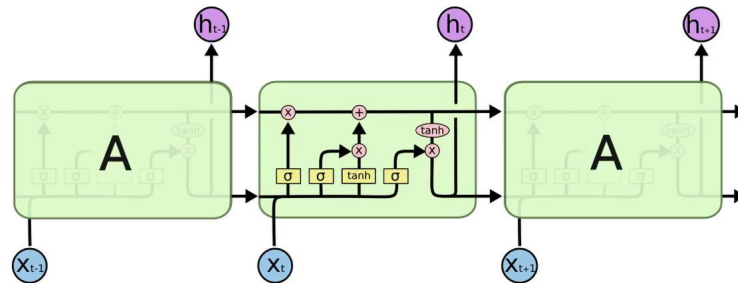
Content of this lecture

1. Time delay network
2. Recurrent networks, the general type
3. Recurrent networks, LSTM and GRU types
4. LSTM architectures ...
5. Combination of architectures

(Common google images when searching for RNN)



Long-Short Term Memory module: LSTM

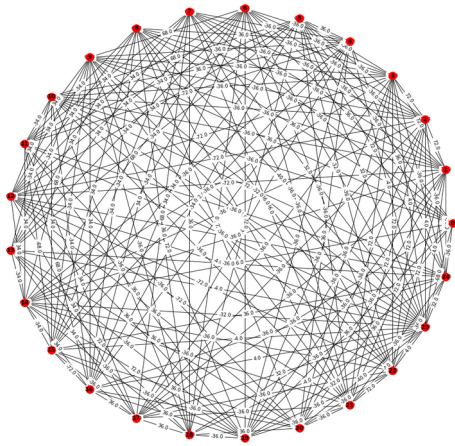


long-short term memory modules used in an RNN

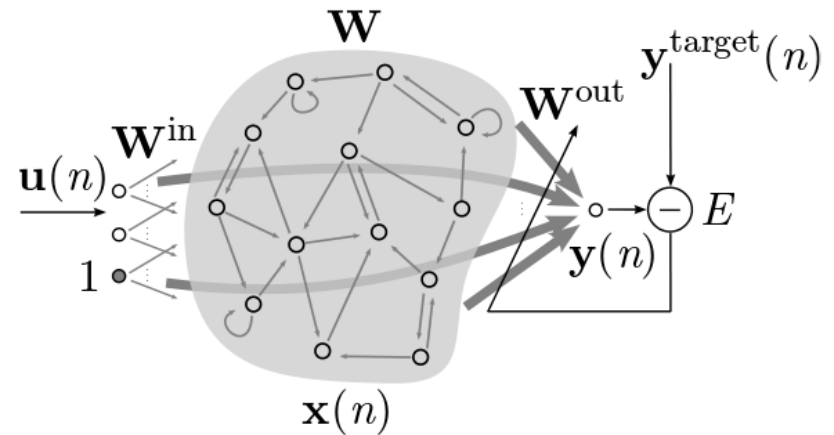


But these are also recurrent networks

Hopfield model



Echo state network



Recurrent Neural Networks

- Common for all neural network models we have studied so far is the lack of feedback connections. We will now study networks with such connections!
- Recurrent networks are typically used when we are dealing with sequence data. It can be text data, speech data, image data or numerical times series data coming from eg. sensors or stock markets. And combinations of all these!
- The feedback connections are used to capture the short and long term temporal dependencies in the data.

What is it that recurrent networks offer?

Sofar, one-to-one problems!

classification

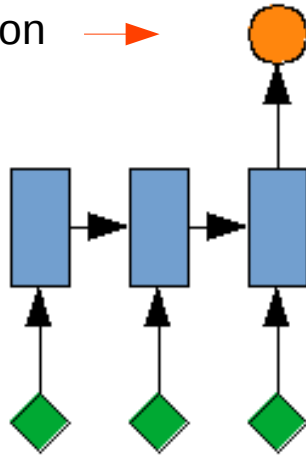


Image
(one object)

With recurrent networks

many-to-one

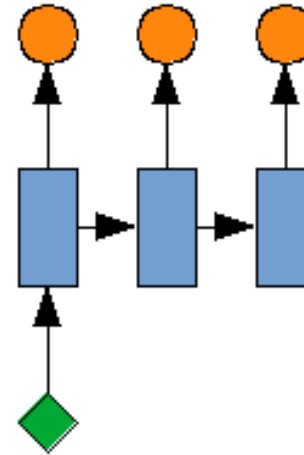
eg. classification



eg. text (sequence of words)

one-to-many

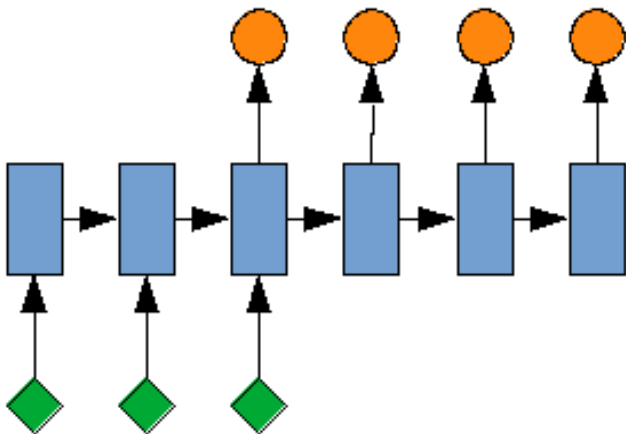
eg. caption



eg. image

many-to-many

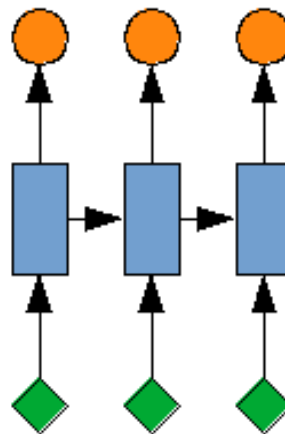
eg. text



eg. text

many-to-many

eg. another
sequence



eg. sequence

We will however start simple, by using an ordinary MLP for a specific task using sequence data.

Time delay networks (auto regressive model)

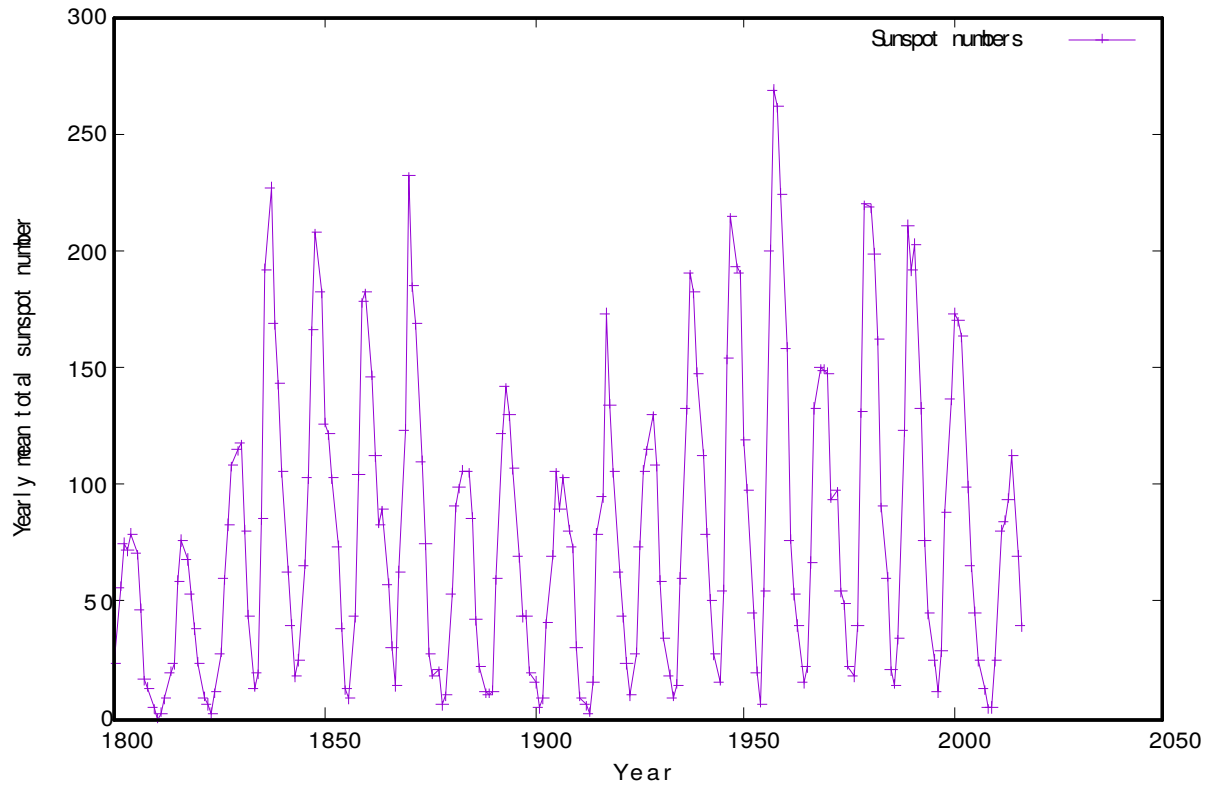
Suppose we have a time series:

$$x(1), x(2), \dots, x(t)$$

Task:

Predict $x(t + 1)$

An example: The yearly sunspot number!



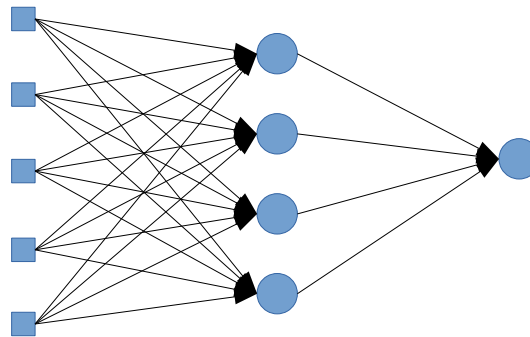
The approach is to use a fixed number of "previous" values of $x(t)$ in order to predict $x(t+)$

As an example we can create the following input-output dataset for the sunspot data using 5 history data points:

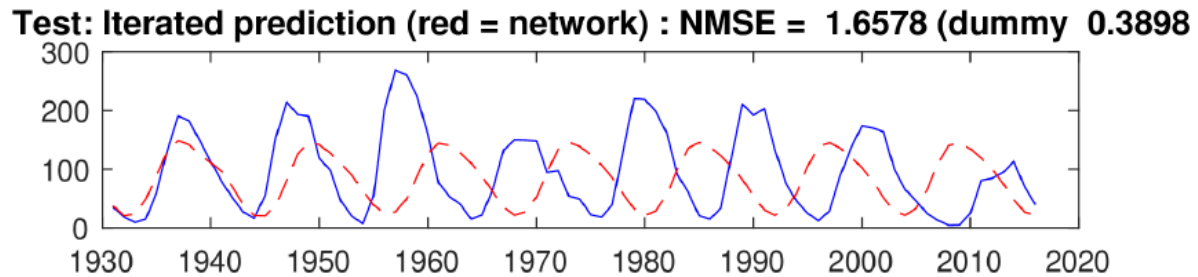
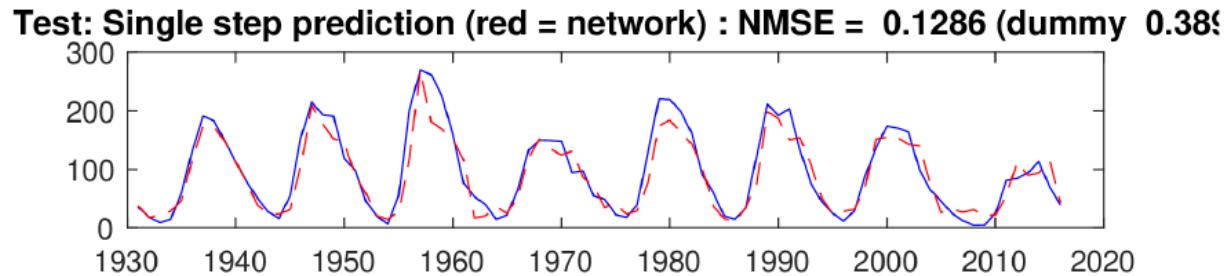
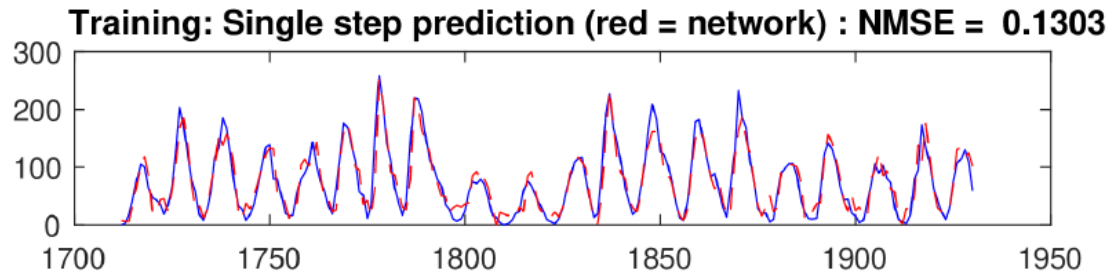
Data no	Input	Target
1	$x(t - 1), x(t - 2), x(t - 3), x(t - 6), x(t - 12)$	$x(t)$
2	$x(t - 2), x(t - 3), x(t - 4), x(t - 7), x(t - 13)$	$x(t - 1)$
3	$x(t - 3), x(t - 4), x(t - 5), x(t - 8), x(t - 14)$	$x(t - 2)$
...

Data no	Input	Target
1	$x(t - 1), x(t - 2), x(t - 3), x(t - 6), x(t - 12)$	$x(t)$
2	$x(t - 2), x(t - 3), x(t - 4), x(t - 7), x(t - 13)$	$x(t - 1)$
3	$x(t - 3), x(t - 4), x(t - 5), x(t - 8), x(t - 14)$	$x(t - 2)$
...

We can train an ordinary MLP
for this regression problem



MLP: 5 inputs, 4 hidden nodes, a single linear output node.

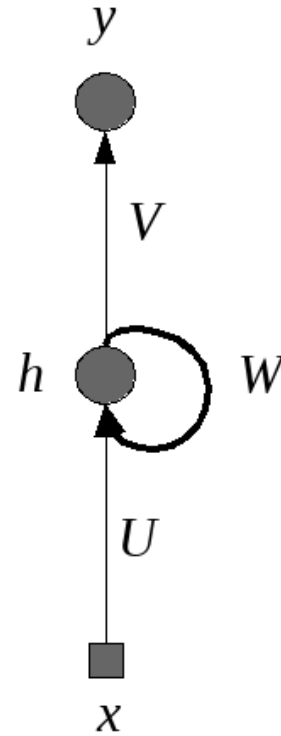
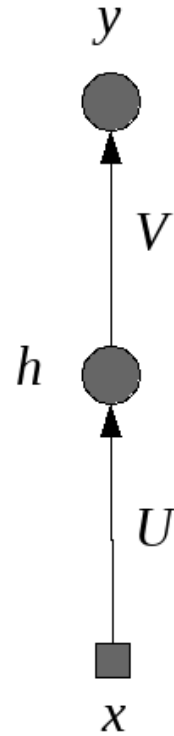


NMSE = Normalized mean squared error

Data no	Input	Target
1	$x(t - 1), x(t - 2), x(t - 3), x(t - 6), x(t - 12)$	$x(t)$
2	$x(t - 2), x(t - 3), x(t - 4), x(t - 7), x(t - 13)$	$x(t - 1)$
3	$x(t - 3), x(t - 4), x(t - 5), x(t - 8), x(t - 14)$	$x(t - 2)$
...

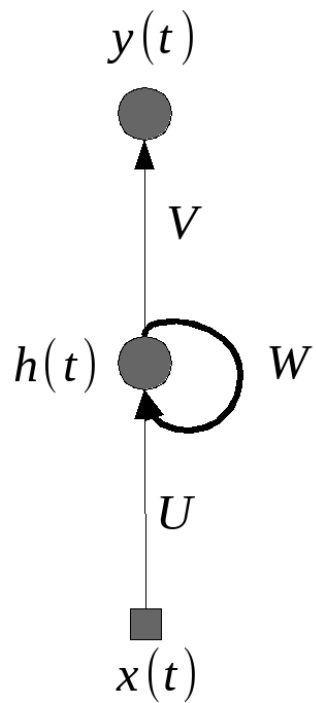
Obvious drawback! Need to specify the number of history values to use!

Simple recurrent networks, general type!



A simple 1-1-1 MLP

This network has a “true” feedback connection from the hidden output feeding into itself.



Given: $x(0), x(1), \dots, x(T)$

$$y(t) = g_o (Vh(t))$$

$$h(t) = g_h (Ux(t) + Wh(t - 1))$$

Let's be explicit for a few time steps!

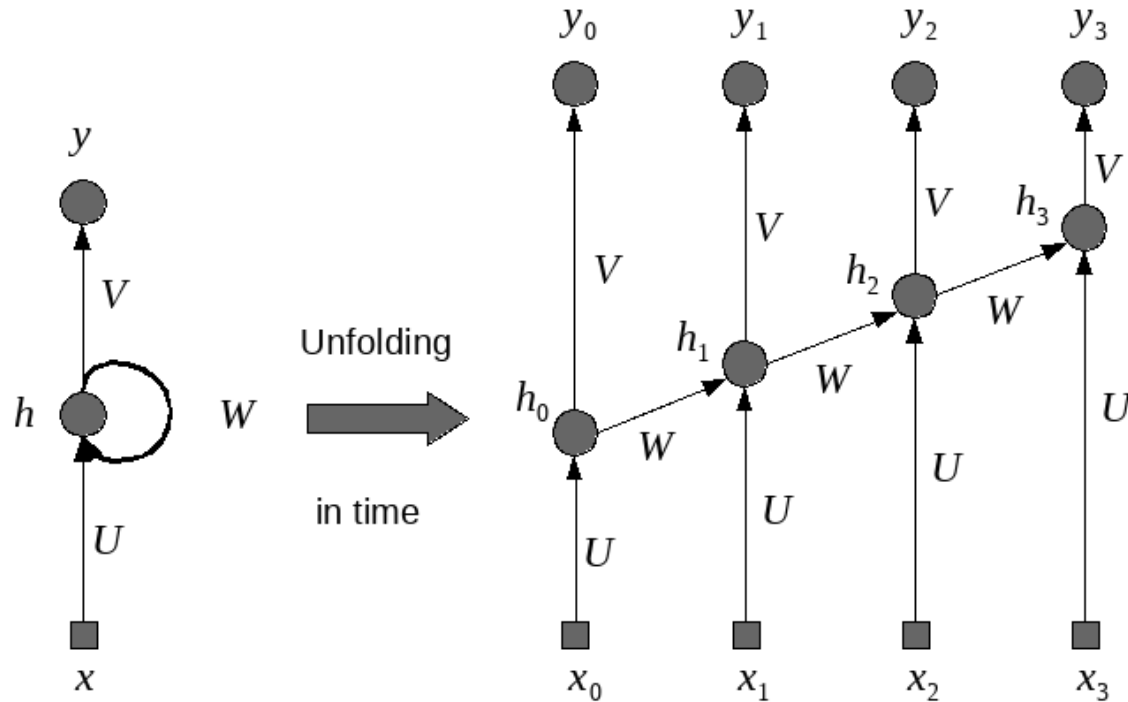
(to avoid clutter: $x(t) \rightarrow x_t, h(t) \rightarrow h_t$)

$$y(0) = g_o[Vh_0] = g_o[Vg_h(Ux_0)] \quad (\text{using the initial condition } h(-1) = 0)$$

$$y(1) = g_o[Vh_1] = g_o[Vg_h(Ux_1 + Wh_0)] = g_o[Vg_h(Ux_1 + Wg_h(Ux_0))]$$

$$\begin{aligned} y(2) &= g_o[Vh_2] = g_o[Vg_h(Ux_2 + Wh_1)] = \\ &= g_o[Vg_h(Ux_2 + Wg_h(Ux_1 + Wh_0))] = \\ &= g_o\left[Vg_h\left(Ux_2 + Wg_h\left(Ux_1 + Wg_h(Ux_0)\right)\right)\right] \end{aligned}$$

Do we recognize this structure?




$$y(2) = g_o \left[V g_h \left(U x_2 + W g_h \left(U x_1 + W g_h \left(U x_0 \right) \right) \right) \right]$$

Unfolding in time!

The unfolded network is an MLP with 4 hidden layers, sparsely connected and shared weights.

Backpropagation through time (BPTT) (= basically training the unfolded MLP)

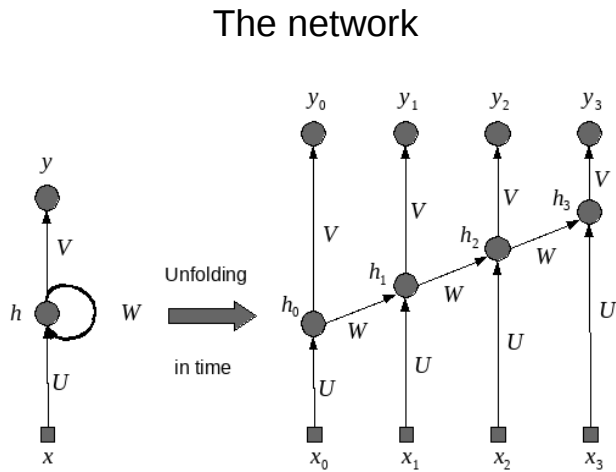
Assume an error of this kind!

$$E(U, W, V) = \sum_{t=0}^T E_t(U, W, V)$$


Can measure the difference between
a target sequence and an input sequence

But many other possibilities, eg. classification of sequences.

To do gradient descent we need to compute derivatives!



$$\frac{\partial E}{\partial V} = \sum_t \frac{\partial E_t}{\partial V}$$

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

$$\frac{\partial E}{\partial U} = \sum_t \frac{\partial E_t}{\partial U}$$

The V weight is rather simple!

$$\frac{\partial E_t}{\partial V} = \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial V} = \frac{\partial E_t}{\partial y_t} g'_o(V h_t) h_t$$

(Remember)

$$y_t = g_o(V h_t)$$

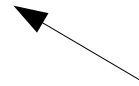
$$h_t = g_h(U x_t + W h_{t-1})$$

The other two are more complicated! For the W weight we get:

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial W}$$

Now,

$$h_t = g_h(Ux_t + Wh_{t-1})$$



Depends on W

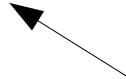
The correct expression:

$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

Is this a problem?

$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

This is also a chainrule



For example

$$\frac{\partial h_3}{\partial h_1} = \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1}$$

Now each

$$\frac{\partial h_{t+1}}{\partial h_t} = g'_h(\cdot)W$$

To conclude: Training a simple recurrent network like this, for long sequences, involves multiplication of many terms like,

$$g'_h(\cdot)W$$

Two numerical problems can occur:

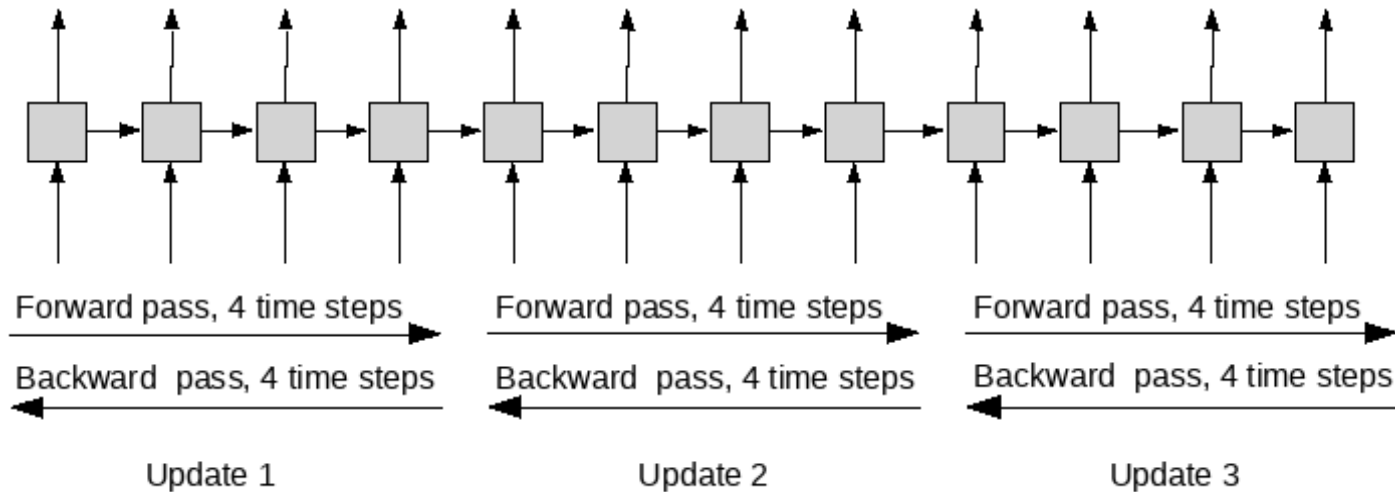
- Vanishing gradients
- Exploding gradients

Where vanishing gradients are probably most common!

As a result it is difficult to learn long term dependencies!

How do we solve this?

First approach: Truncated backpropagation through time!

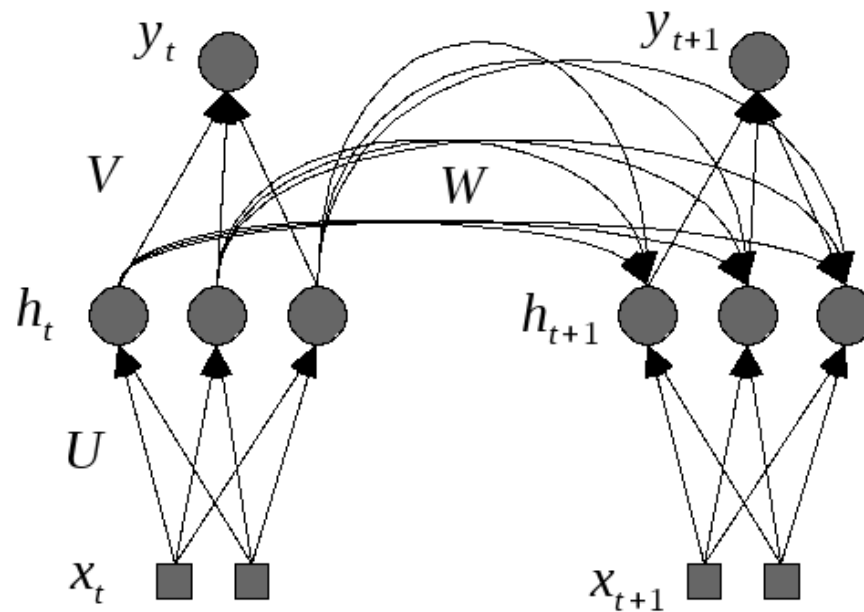


Here it is important to preserve the state between updates!

Second approach: Change the behavior of the hidden node as to have a more or less constant value of the recursive derivative.

→ **LSTMs**

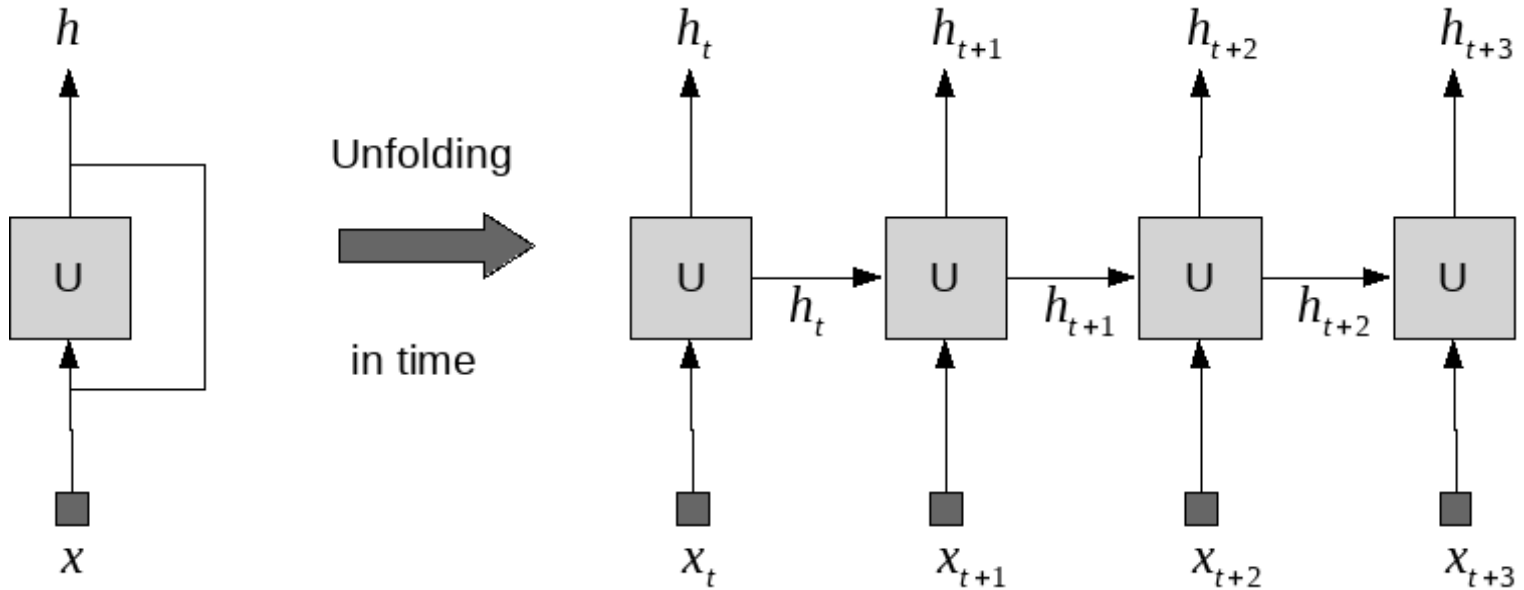
Before that let's just conclude by noting that the simple recurrent network can be more complicated than what we have showed here. As an example:



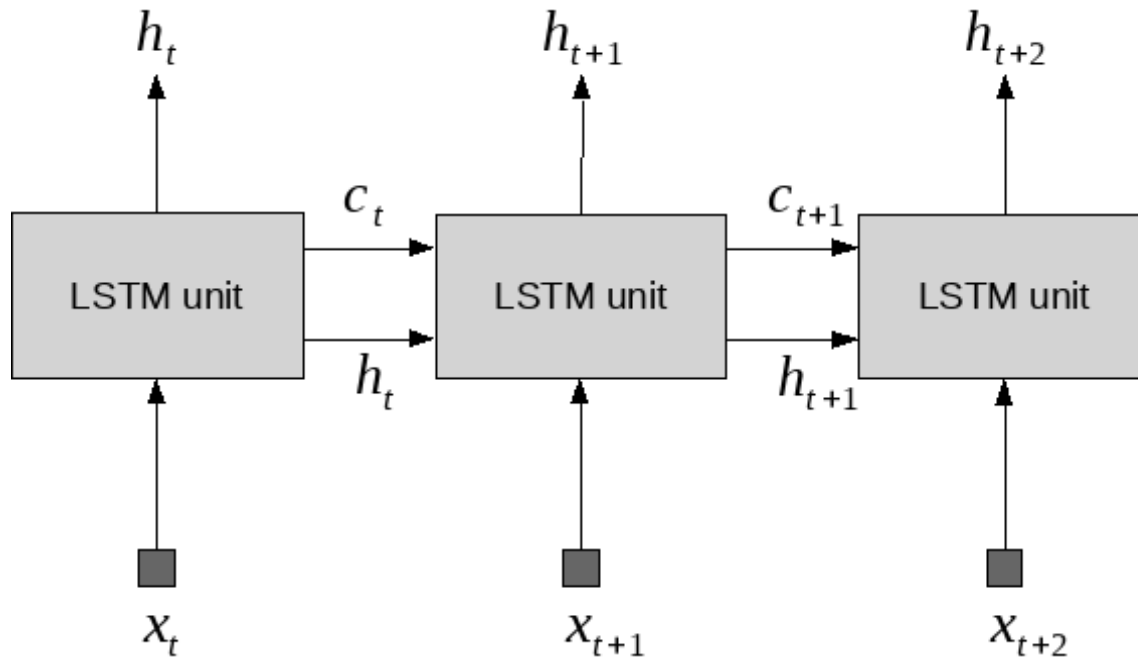
Two input, three hidden nodes. Unfolded one time step!

Long short-term memories (LSTM)

Previously we had networks like this, where the “U” box represents the simple recurrent unit.

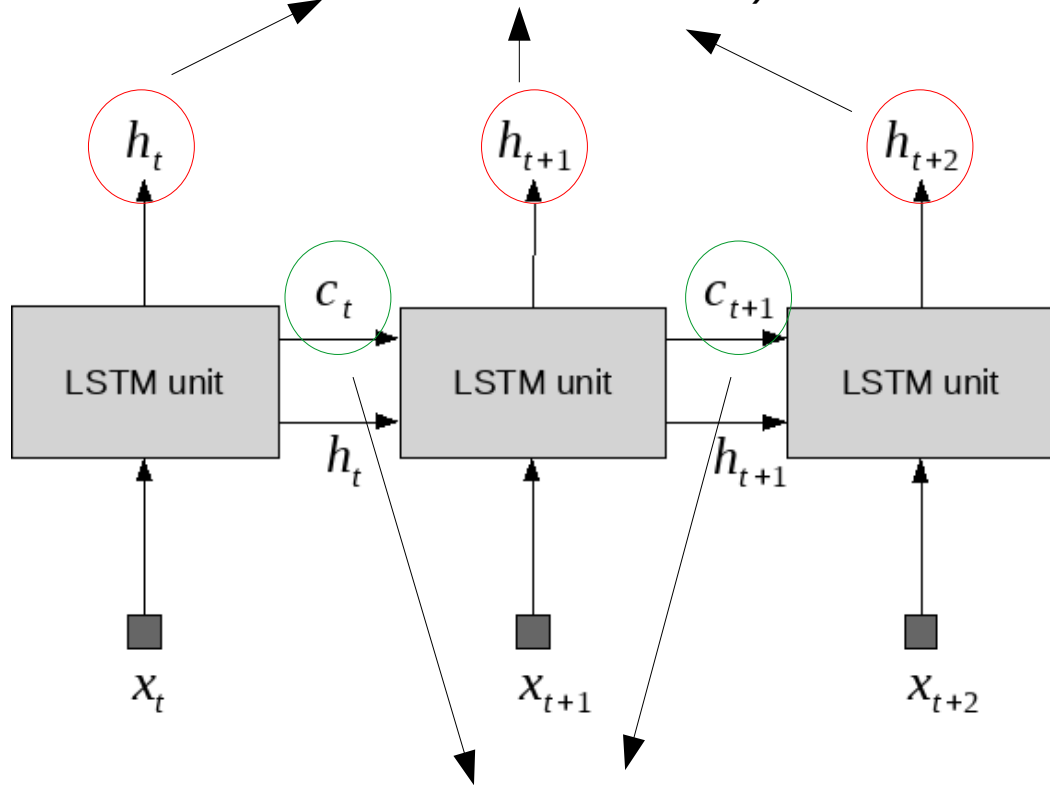


For a LSTM network we simply replace the “U” box with a LSTM box and add a new connection,



So in a sense it is similar to the previous network (apart from the "c" value. However the LSTM box is more complicated than the "U" box!

The usual hidden state (or the output of the hidden node)



Internal memory of the LSTM node. Is not "exported"!

Details of the LSTM node!

In addition to the internal memory c_t we have so called gates

$$\begin{aligned}\text{input gate } i &= \sigma(x_t U^i + h_{t-1} W^i + b_i) \\ \text{forget gate } f &= \sigma(x_t U^f + h_{t-1} W^f + b_f) \\ \text{output gate } o &= \sigma(x_t U^o + h_{t-1} W^o + b_o)\end{aligned}$$

They are all number between 0 and 1, and are used to "filter" new values for the memory and hidden state.

$$(U^i, U^f, U^o), (W^i, W^f, W^o), (b_i, b_f, b_o) \quad = \text{new weights!}$$

The internal calculation is done as follows:

1. New candidate value for the internal memory.

$$\tilde{c}_t = \tanh(x_t U^c + h_{t-1} W^c + b_c)$$

2. New internal memory

$$c_t = c_{t-1} f + \tilde{c}_t i$$

3. New hidden value

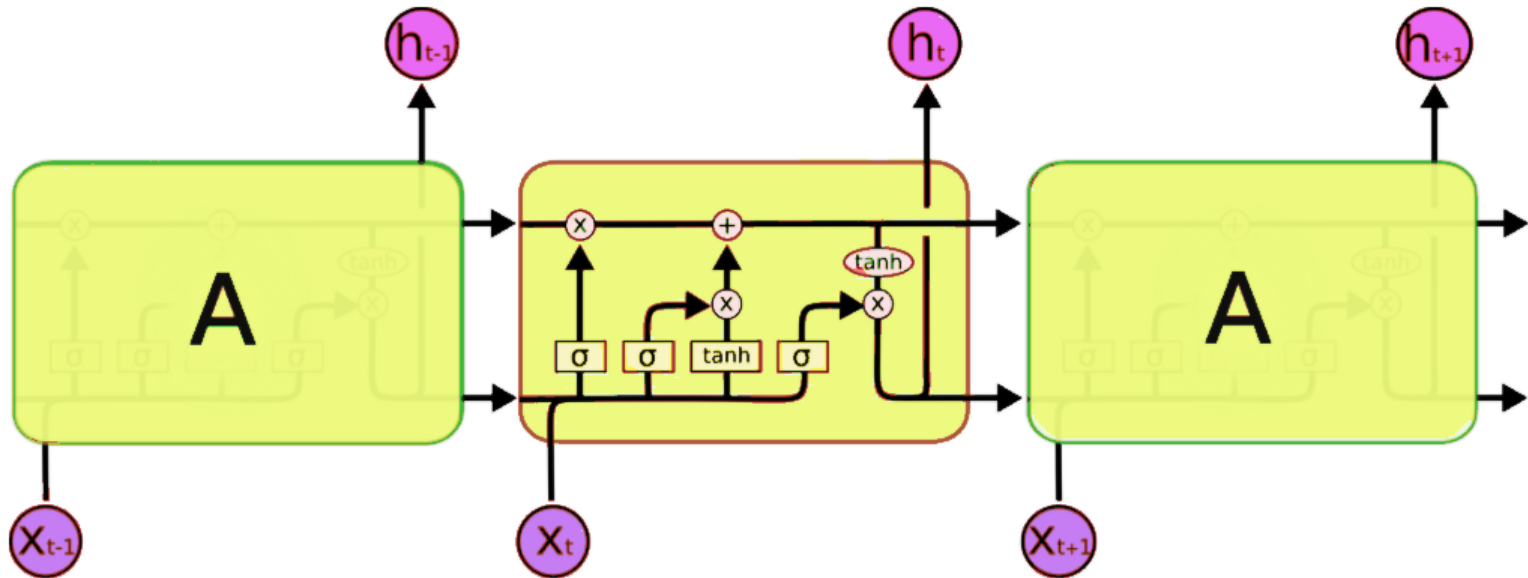
$$h_t = \tanh(c_t) o$$

The new memory value is a combination of the previous value filtered by the forget gate and the candidate value filtered by the input gate.

The new output of the LSTM node (h_t) is the memory “squashed” by the $\tanh()$ and filtered by the output gate.

It is the gating mechanism that allows the LSTM network to model long-term dependencies.

There are many illustrations of the LSTM node, here is one:



From a good blog:

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

Can we understand why the LSTM handles the vanishing gradient problem?

Previously we looked at $\frac{\partial h_{t+1}}{\partial h_t}$

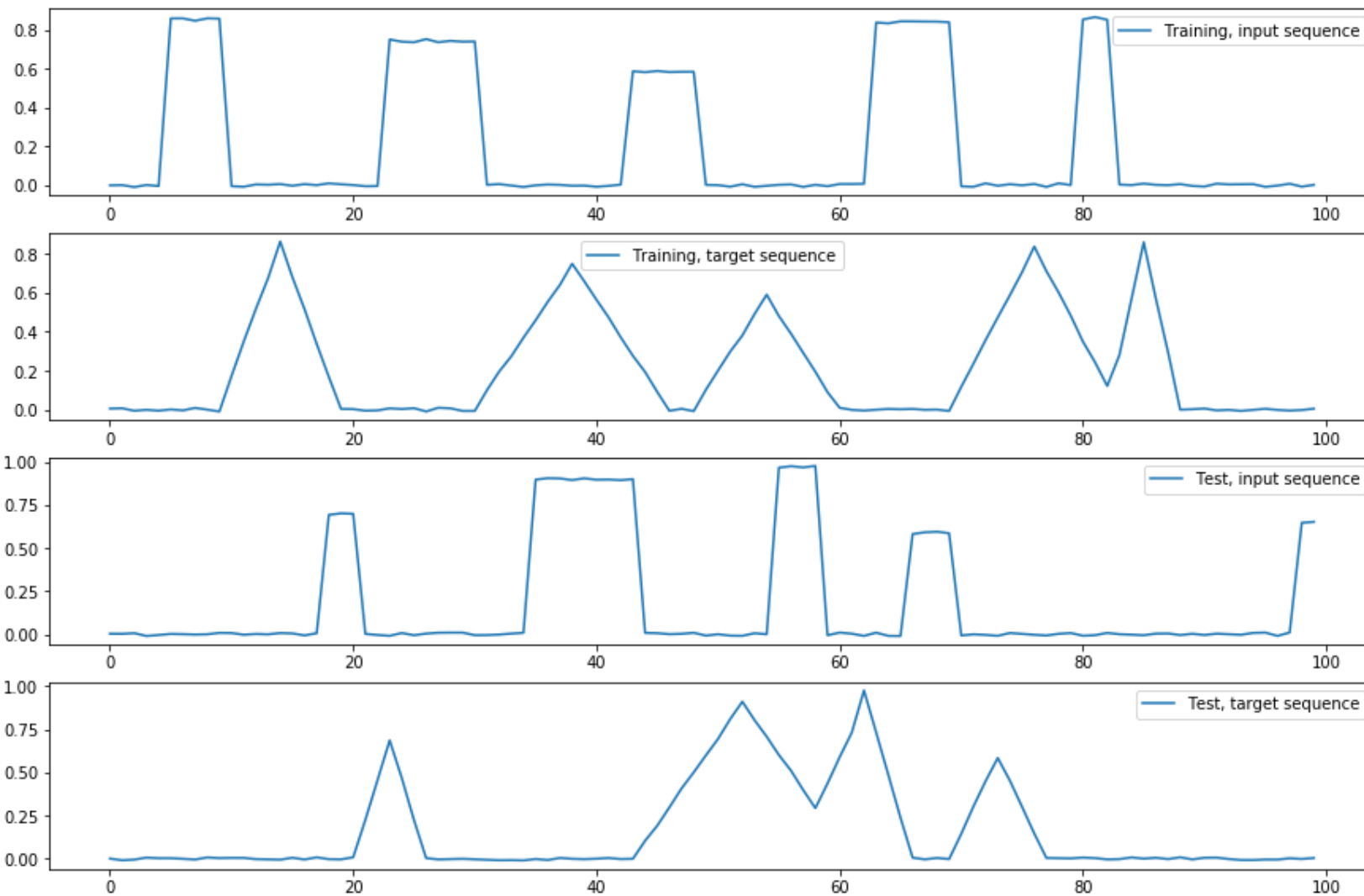
A corresponding derivative is now: $\frac{\partial c_t}{\partial c_{t-1}}$

$$\begin{aligned}
\frac{\partial c_t}{\partial c_{t-1}} &= c_{t-1} \sigma'(\cdot) W^f o_{t-1} \tanh'(c_{t-1}) \\
&\quad + \tilde{c}_t \sigma'(\cdot) W^i o_{t-1} \tanh'(c_{t-1}) \\
&\quad + i_t \tanh(\cdot) W^c o_{t-1} \tanh'(c_{t-1}) \\
&\quad + f_t
\end{aligned}$$

The details are not important, but:

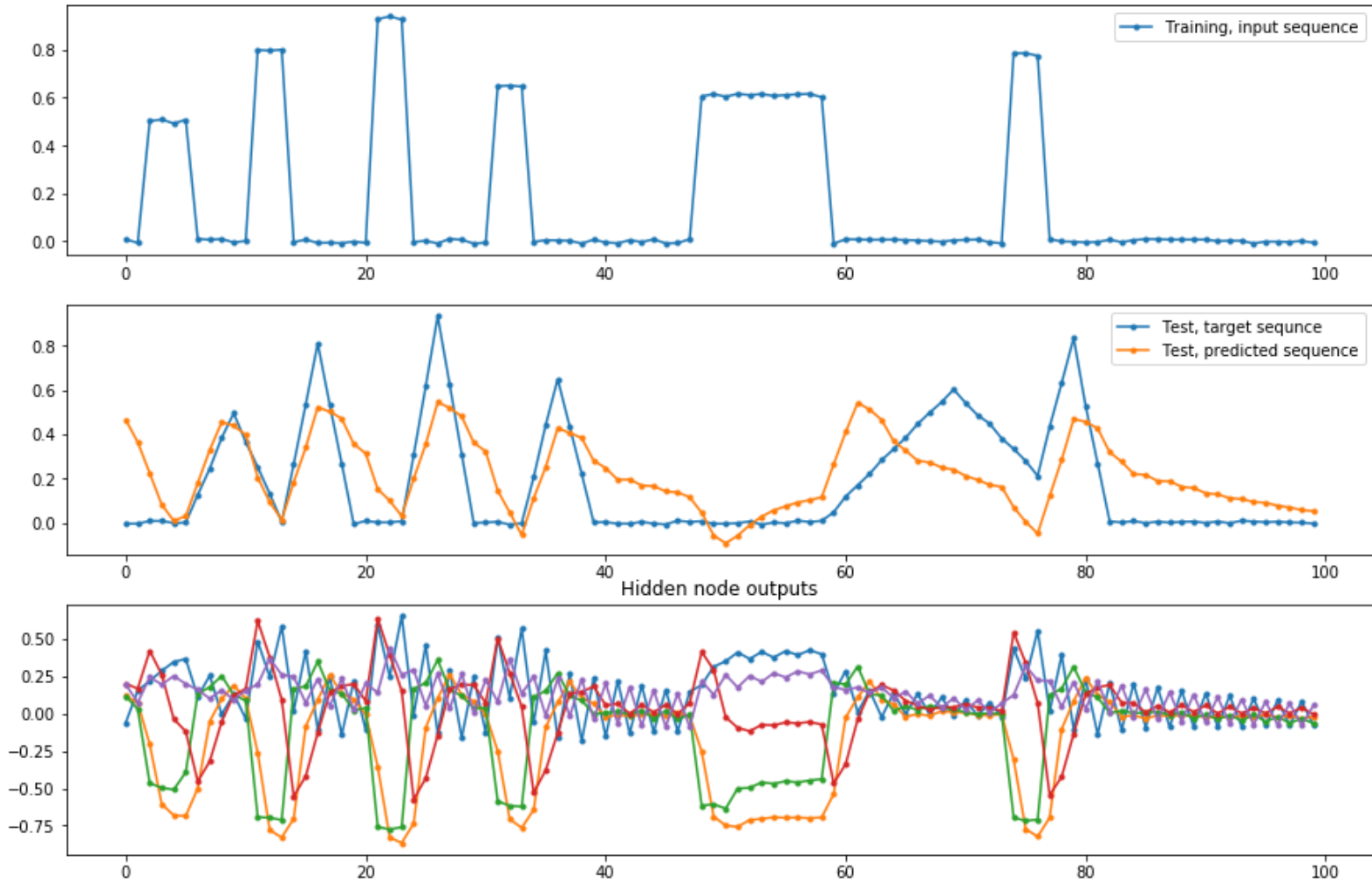
- We will multiply many such terms
- The terms can be both > 1 and < 1
- In a sense the network can *learn* when gradients should vanish!!

Numerical example! RNN as a pulse converter



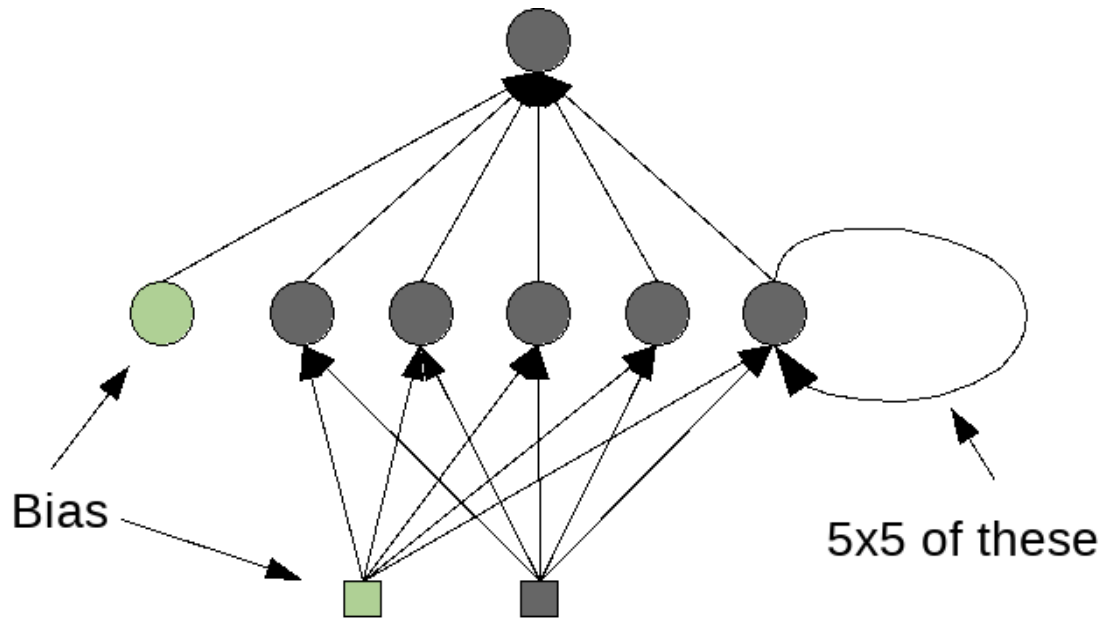
Simple recurrent network with 5 hidden nodes (no LSTM)

(41 trainable weights! How?)



Test error: 0.47

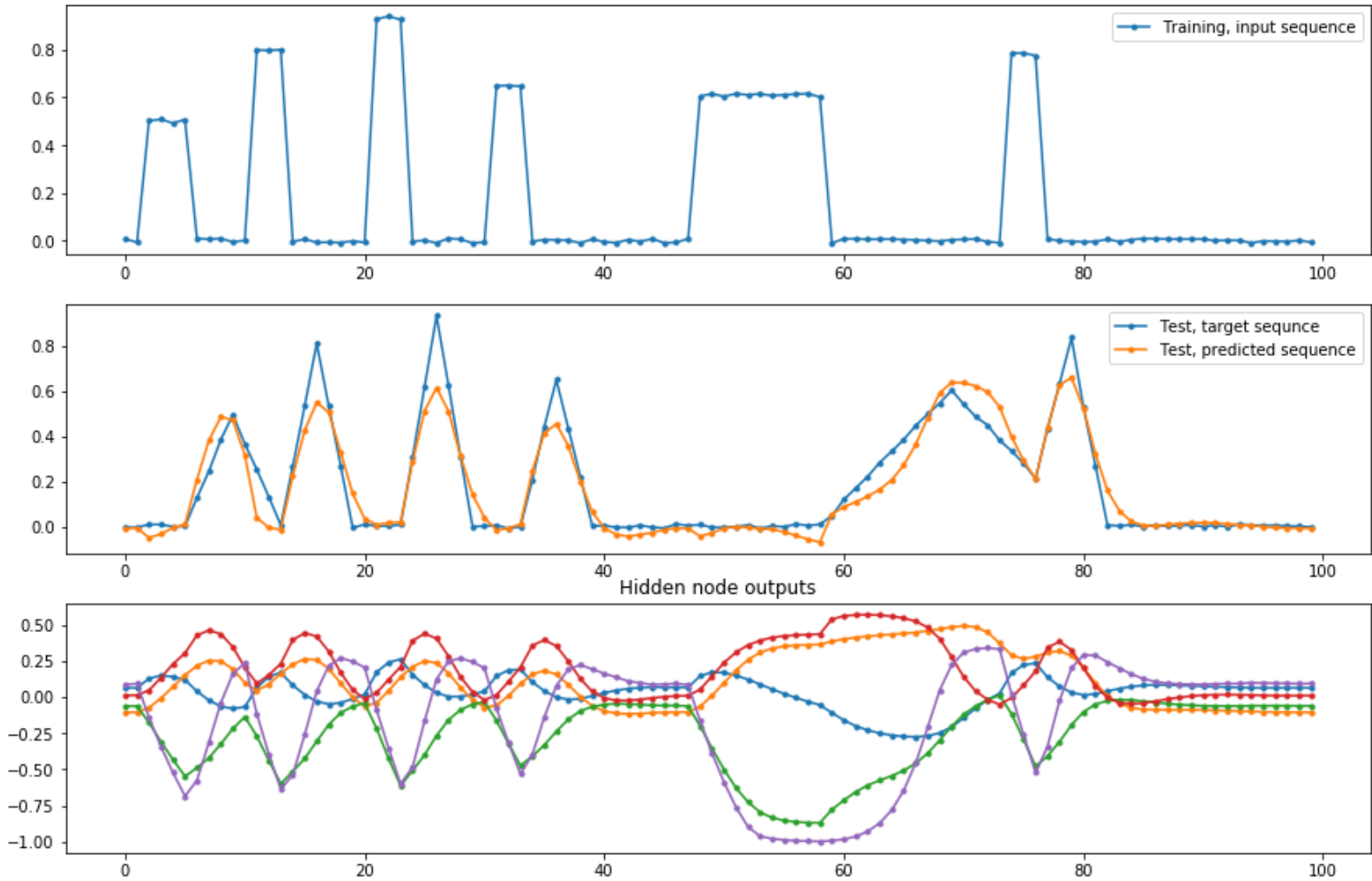
(41 trainable weights! How?)



$$(5+1) + 5 \times 5 + 5 \times (1+1) = 41$$

LSTM network with 5 hidden nodes!

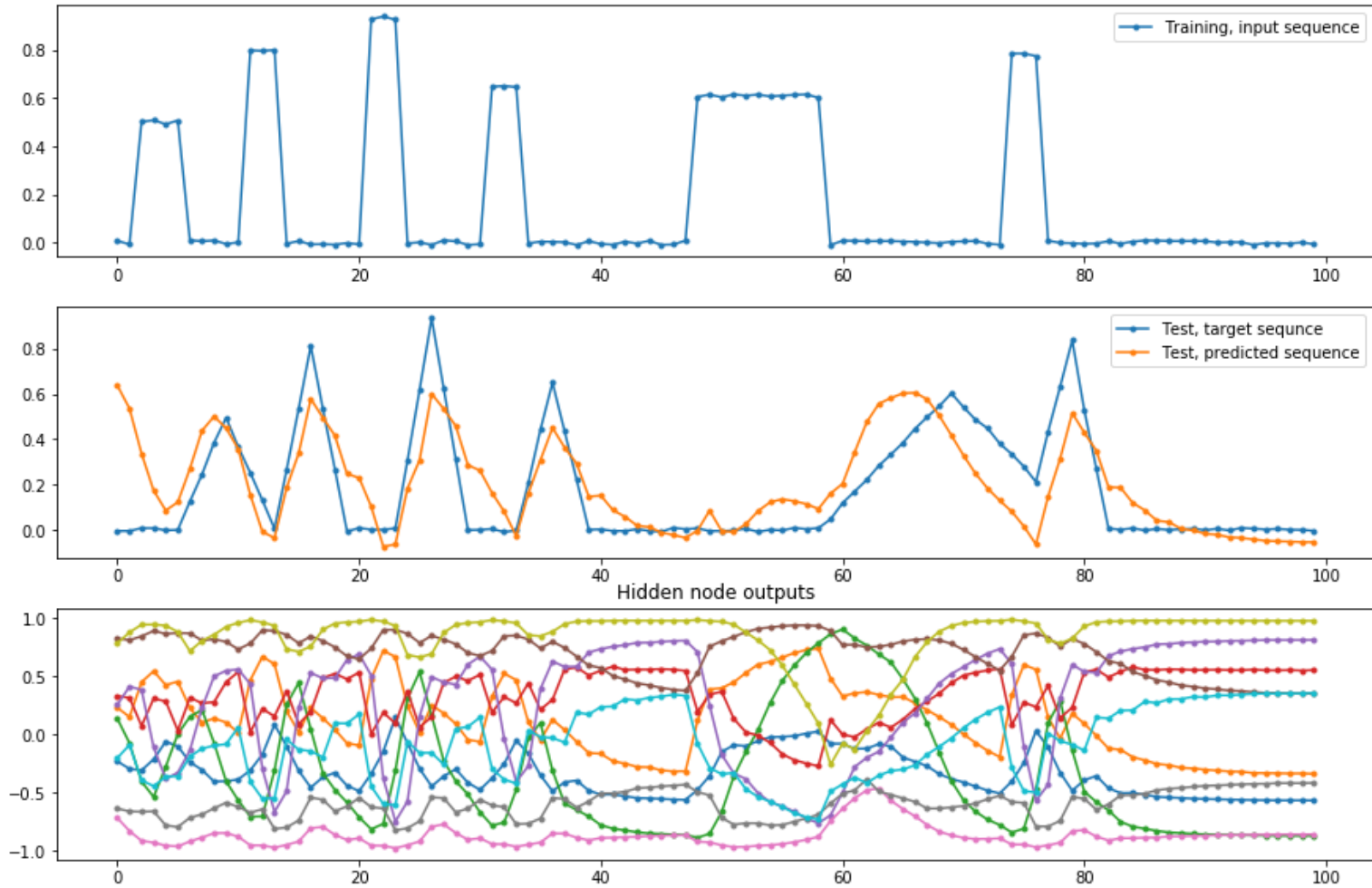
(146 trainable weights!)



Test error: 0.08

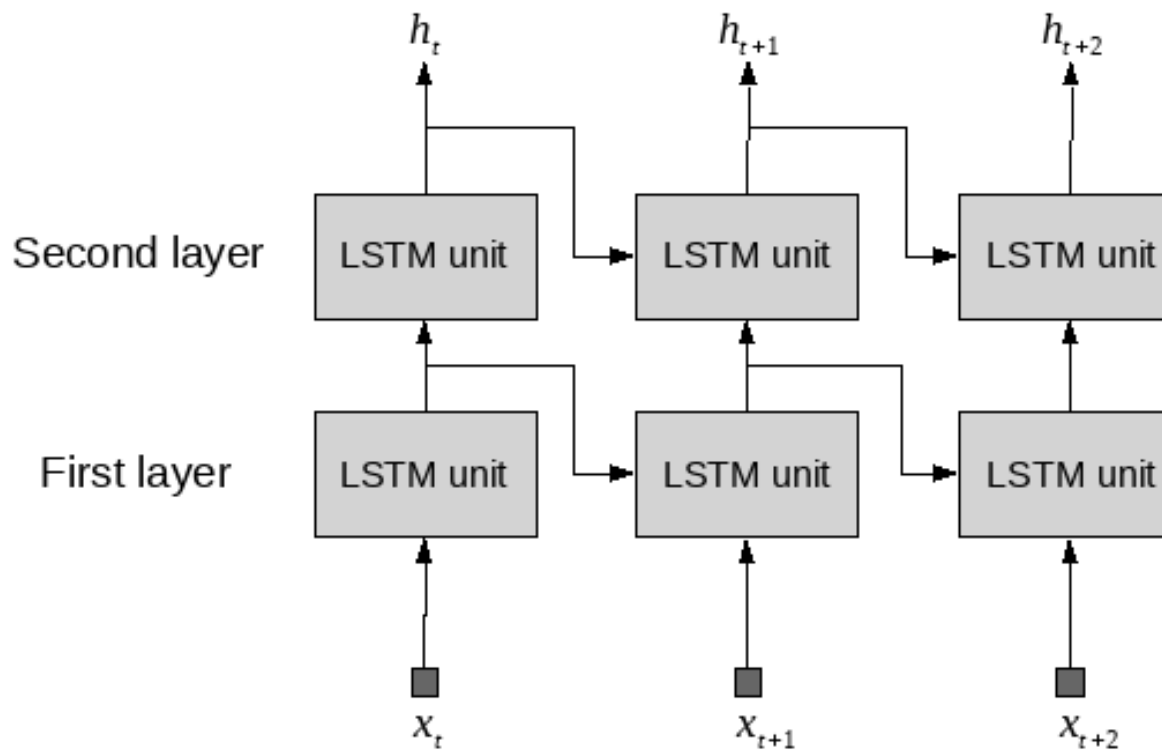
Simple recurrent network with 10 hidden nodes!

(131 trainable weights)



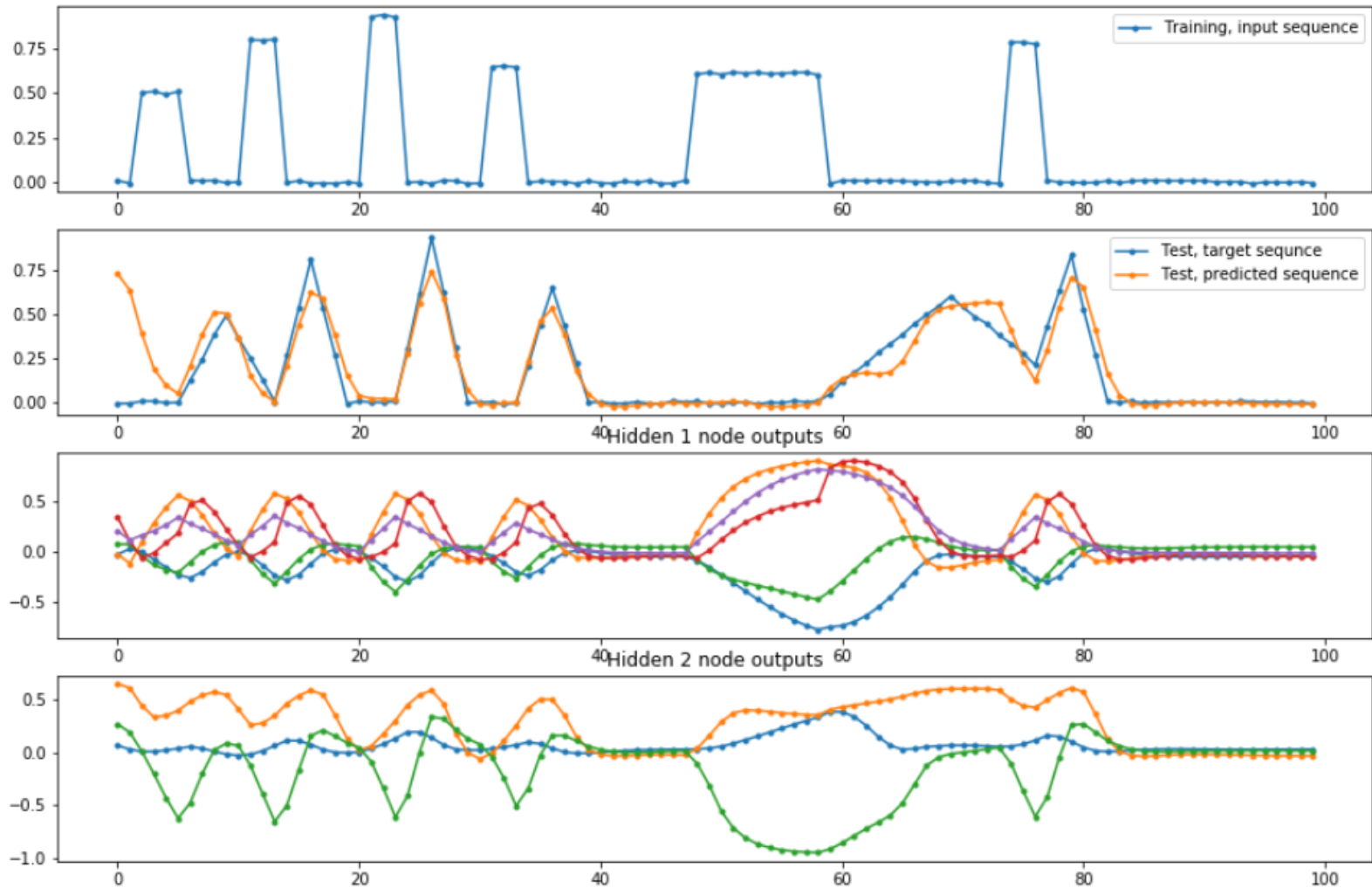
Test error: 0.27

Multilayer LSTMs



LSTM network with 5 – 3 nodes!

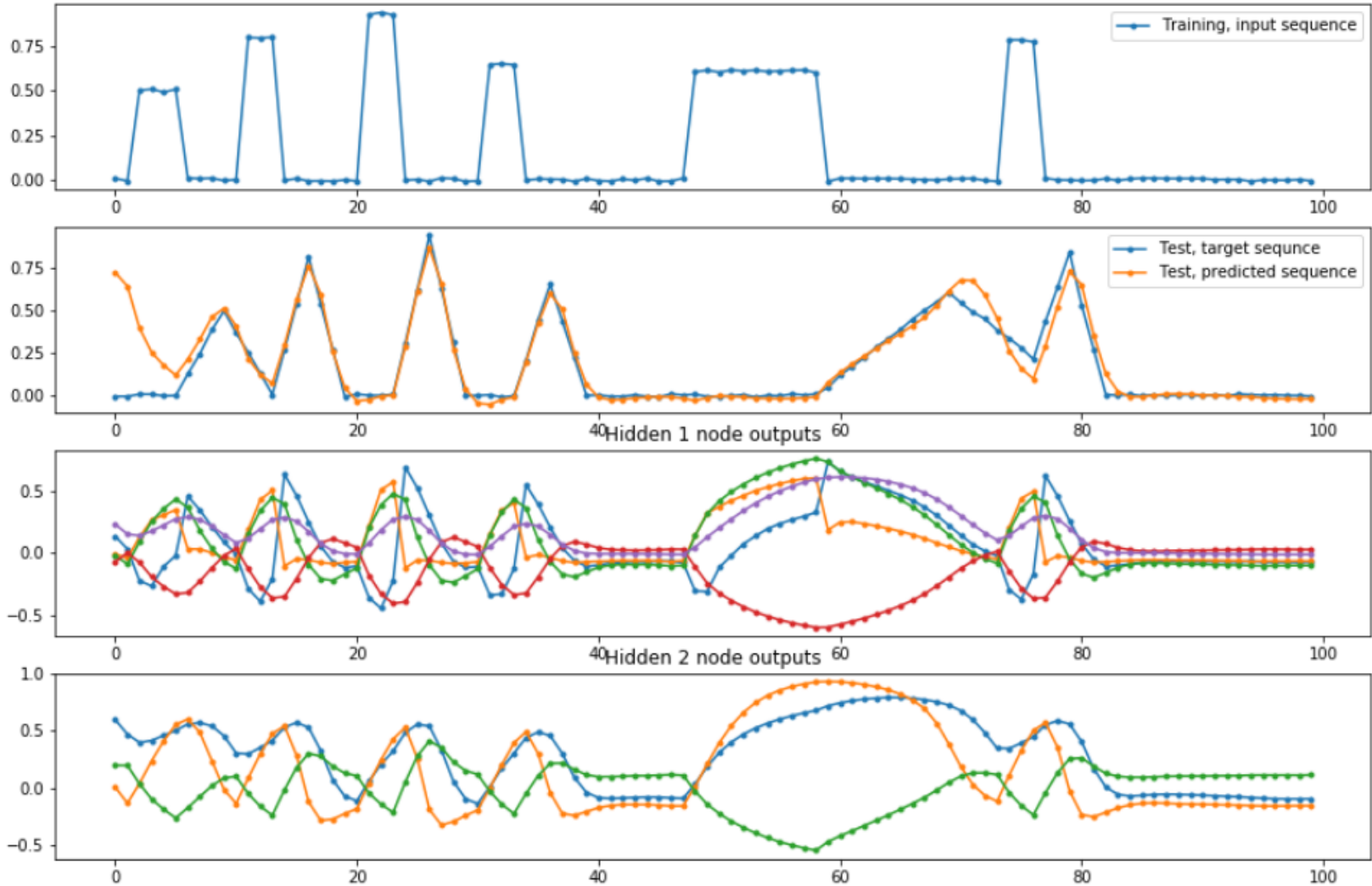
(252 trainable weights!)



Test error: 0.04

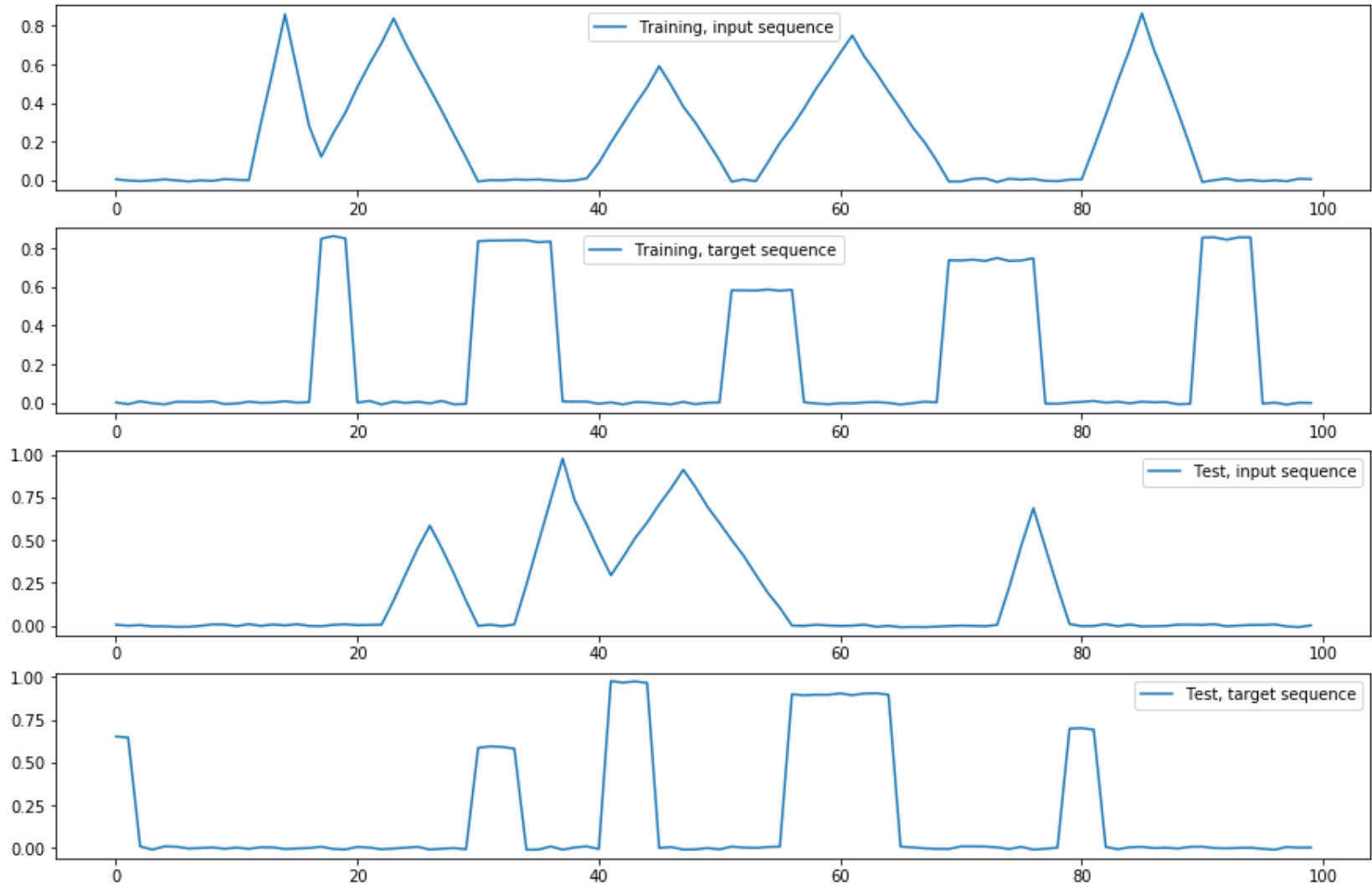
GRU network with 5 – 3 nodes!

(190 trainable weights!)



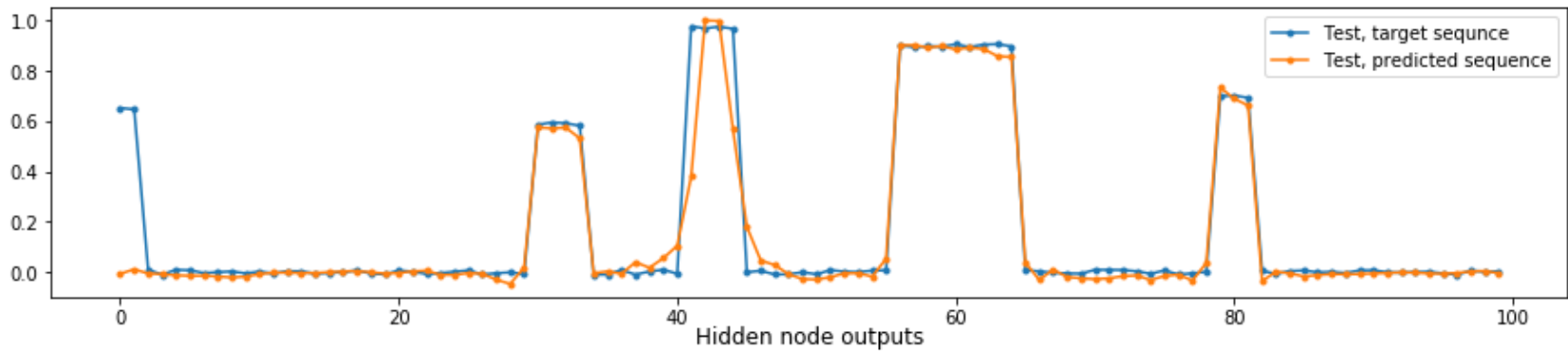
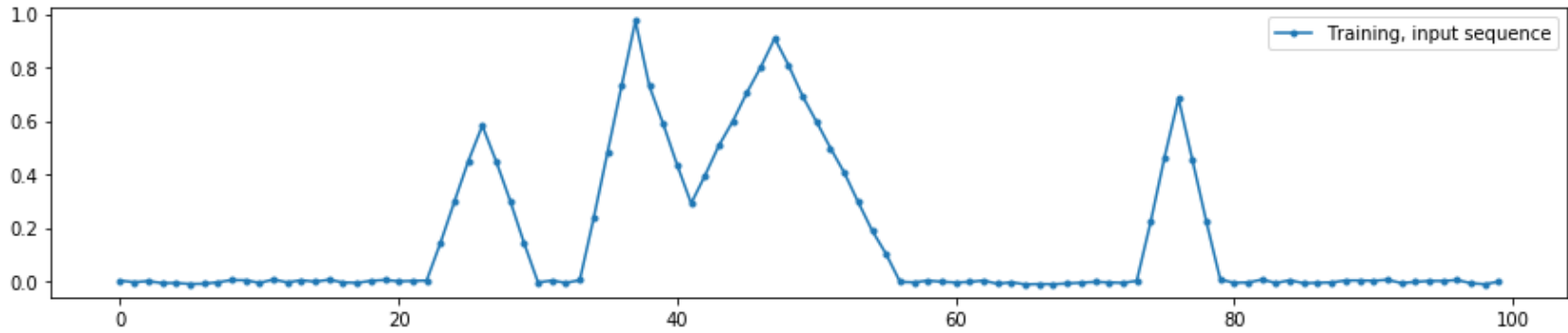
Test error: 0.03

The inverse problem is more difficult!

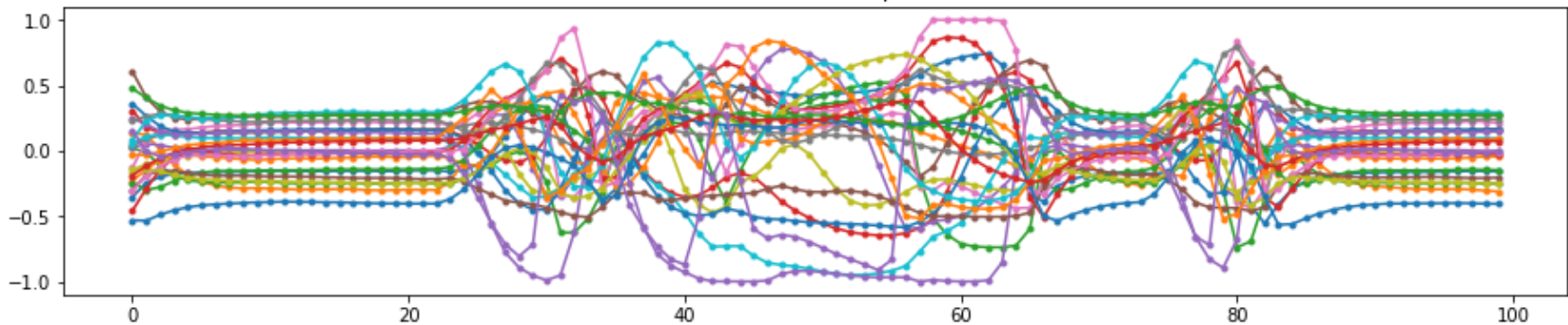


LSTM network with 25 hidden nodes!

(2726 trainable weights!)

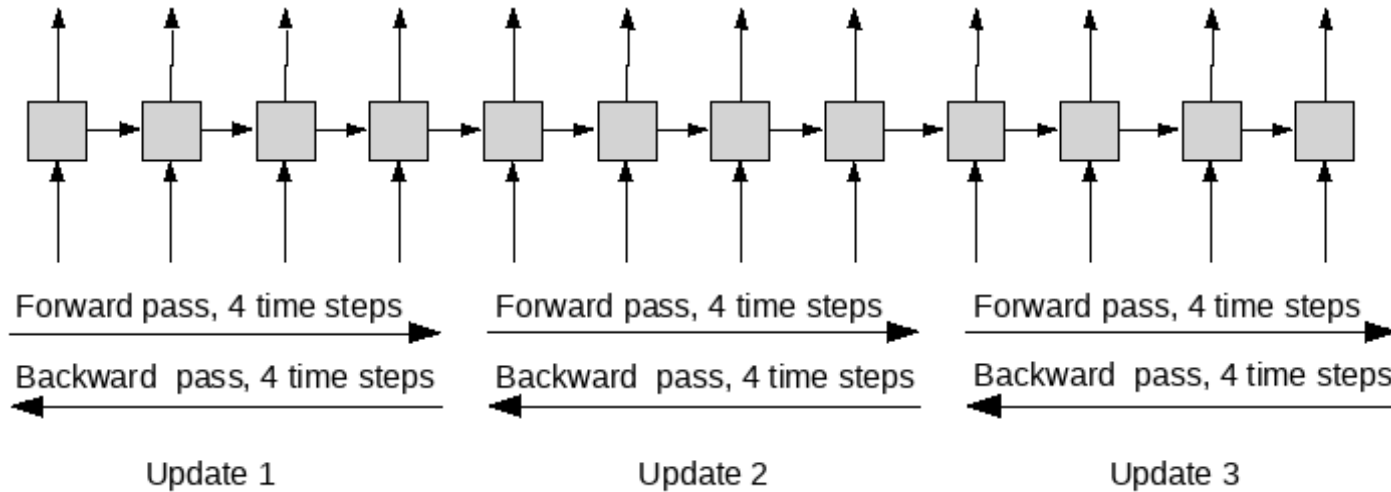


Hidden node outputs



Test error: 0.07

Even with LSTMs it is very common to use truncated BPTT!



But typically with more than 4 times steps!!!

LSTM variant, The Gated Recurrent Unit (GRU)

Simpler version of the LSTM unit. Introduced 2014!

The core idea is the same, but it is simpler.

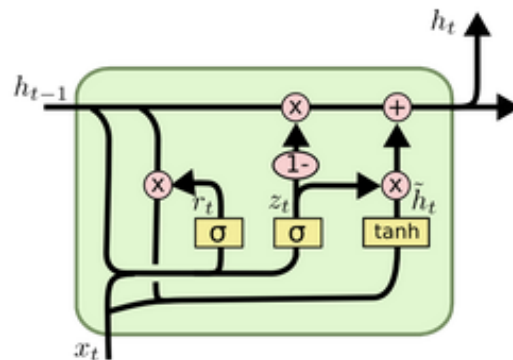
No memory and only two gates.

Update gate $z = \sigma(x_t U^z + h_{t-1} W^z + b_z)$

Reset gate $r = \sigma(x_t U^r + h_{t-1} W^r + b_r)$

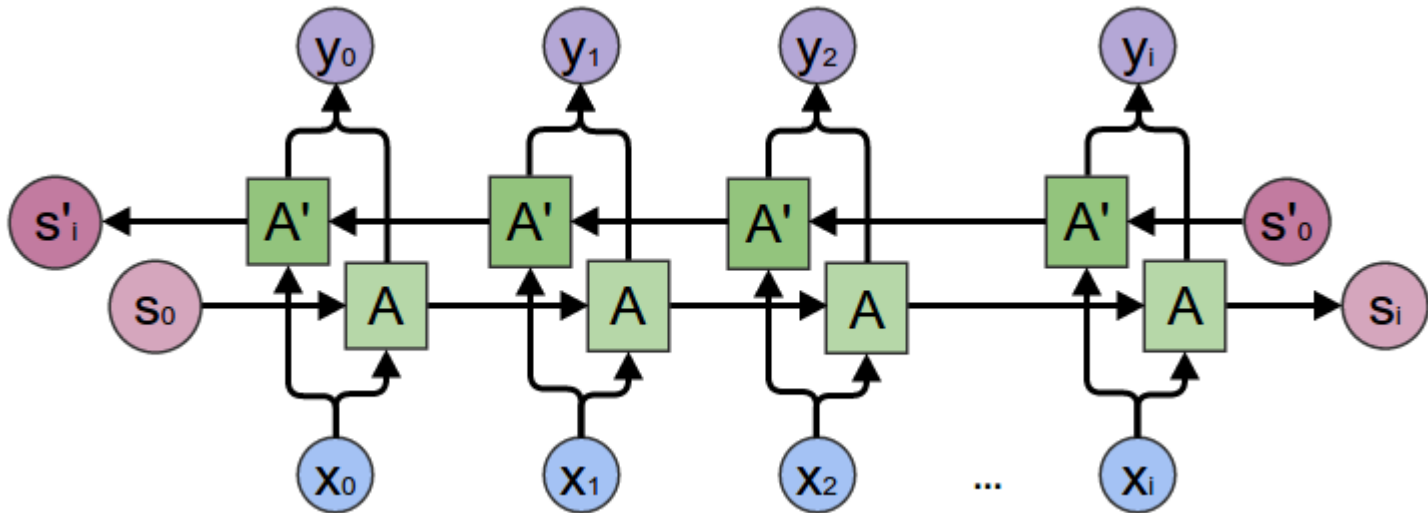
Candidate output $\tilde{h} = \tanh(x_t U^h + (h_{t-1} r) W^h + b_h)$

Output $h_t = (1 - z)\tilde{h} + z h_{t-1}$



Bidirectional LSTMs

Putting two independent LSTMs together, one for the forward sequence and one for the reverse sequence.



Useful when the context of the input is needed.

CNN LSTM architectures

These architectures combine the CNN for dealing with images and LSTM for the sequence part. Examples:

Activity recognition: image sequence to classification

Image description: image to text sequence (caption)

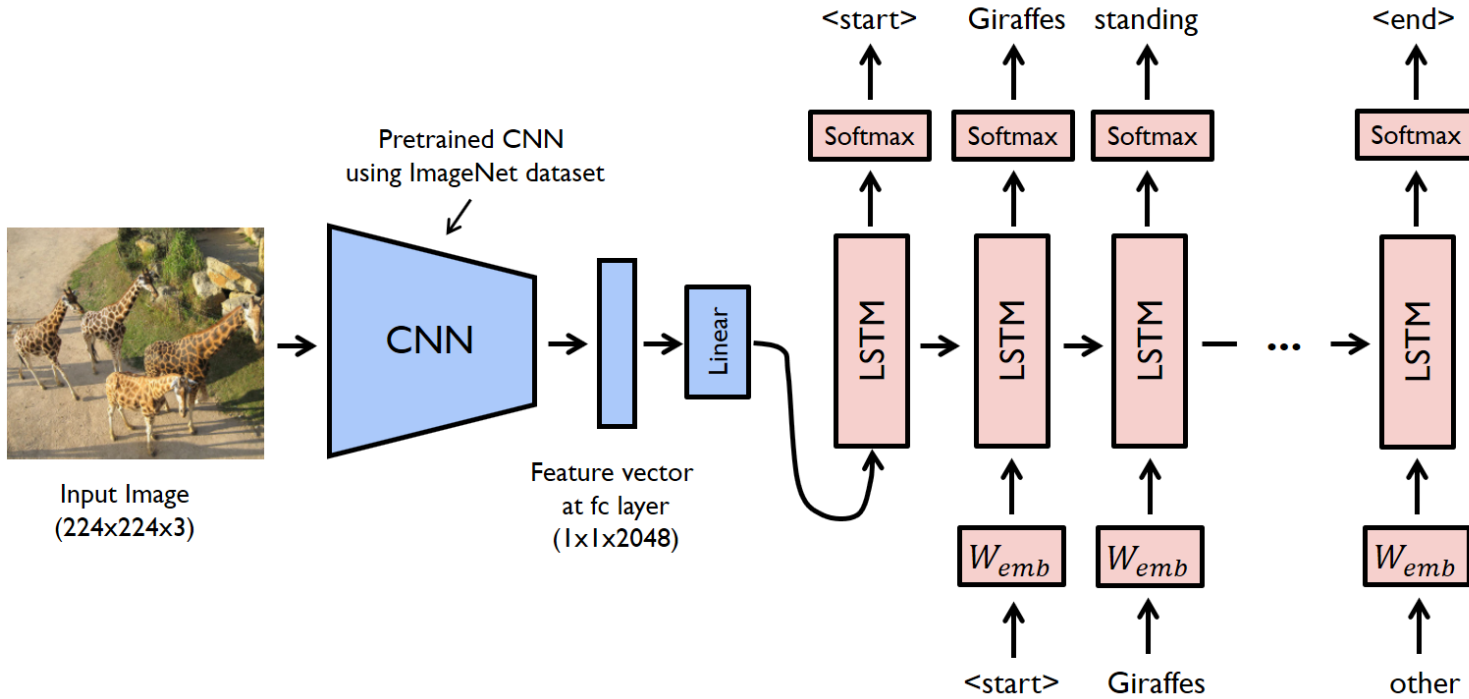
Video description: video sequence to sequence (description)

Two “early” papers:

Show and Tell: A Neural Image Caption Generator ([Link](#))

Long-term Recurrent Convolutional Networks for Visual Recognition and Description ([Link](#))

Example of a caption generator:



A last example: Sampling from a character model

Task: given a sequence of characters, predict the next character!

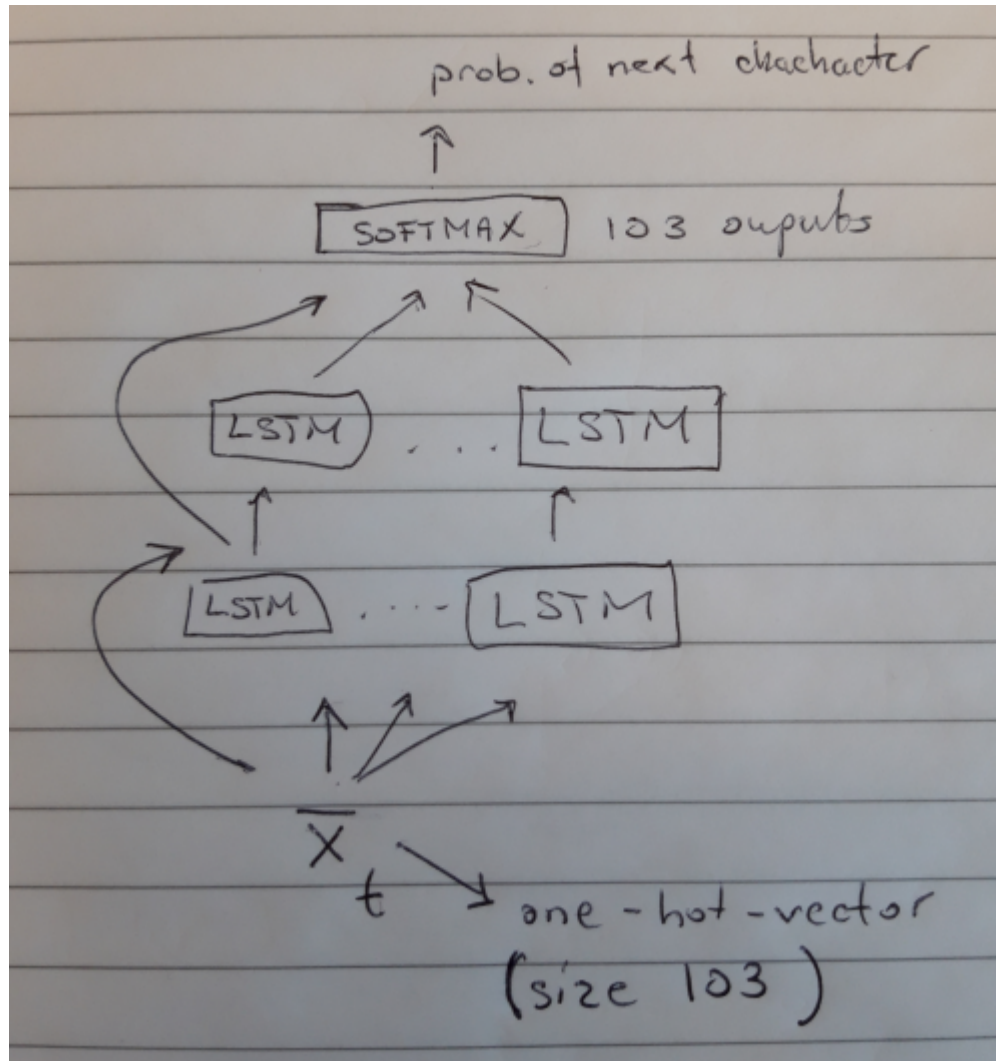
One, two, thre?

My name i?

In this specific example the training sequence
is the source code of (some earlier version) **Tensorflow**.
This is C++ code!

About 14 million characters in the sequence!

The model:



103 different characters
in the code

Skip layer connections

Two layers of 1024
LSTMs each

About 13 million
parameters

Since the output are probabilities for the 103 different characters, we can sample from the model:

Five characters (A, B, C, D, E)

As a simple example

[0.1, 0.3, 0.5, 0.1, 0.0]

The probability for 'C' is 50%, for 'B' 30% and so on.

The sampled character is then used as the new input to the (trained) model.

Seed: "void "

```
void AddShapeMemoryTranspose(const Key& key, const AllocatorAttributes attr) {
    VLOG(2) << "Instanding worker threads with dues to set cuSolveDDe gradient for: " << variant->devices[0]->env_;
    def.label_fills.push_back(TestMultipleWrites());
    std::unique_ptr<thire> locks(0);
    done(e->src(), cinfo->comp_device_context);
    return Status::OK();
}
Status GetStatusResponse::NewAppendable(string* stable_compress_ptr, class LoggingPand& BaseRendezvous,
    const CreateSing& conten
```

Seed: "printf("

```
printf("%s",
        strings::StrCat(true), ">");
}
if (!s.ok()) {
    for (Node* n = 0; node : merge.nodes n& y_noded && (n->dst_node == train_num ==
        CreateNodeDef(SIGEDMB, DT_FLOAT)) // Dsty in different device:
    for (int i = 0; i < N; ++i) {
        (*(start_times[idx].ptr = nullptr) || (i == 1) ? Padding == Padding::VALID:
        (*is_log);
    }
    output_memory_usage = strings::StrCat(prefix, ", successfully.");
}
}
list_type_list.set_select(tf_stat.m_def)
```

Alternative approaches to Recurrent Networks

- CNN models
([Comparison article 1](#),
[Comparison article 2](#))
- The Attention family of networks
(e.g. Transformer network).

[Ref1](#), [Ref2](#)

Example: “[Talk to transformer](#)”