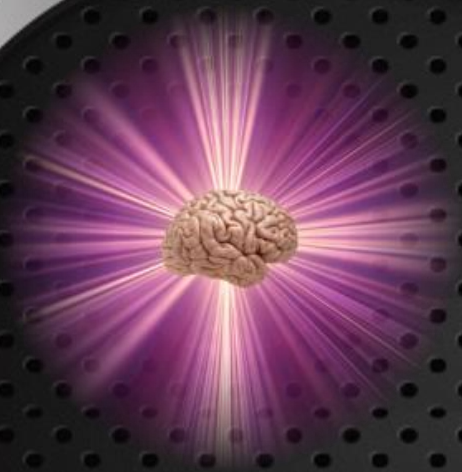
AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering
Department of Biocybernetics and Biomedical Engineering



Computational Intelligence

Recurrent Neural Networks for Learning Sequential Data

Adrian Horzyk
horzyk@agh.edu.pl

Google: Adrian Horzyk

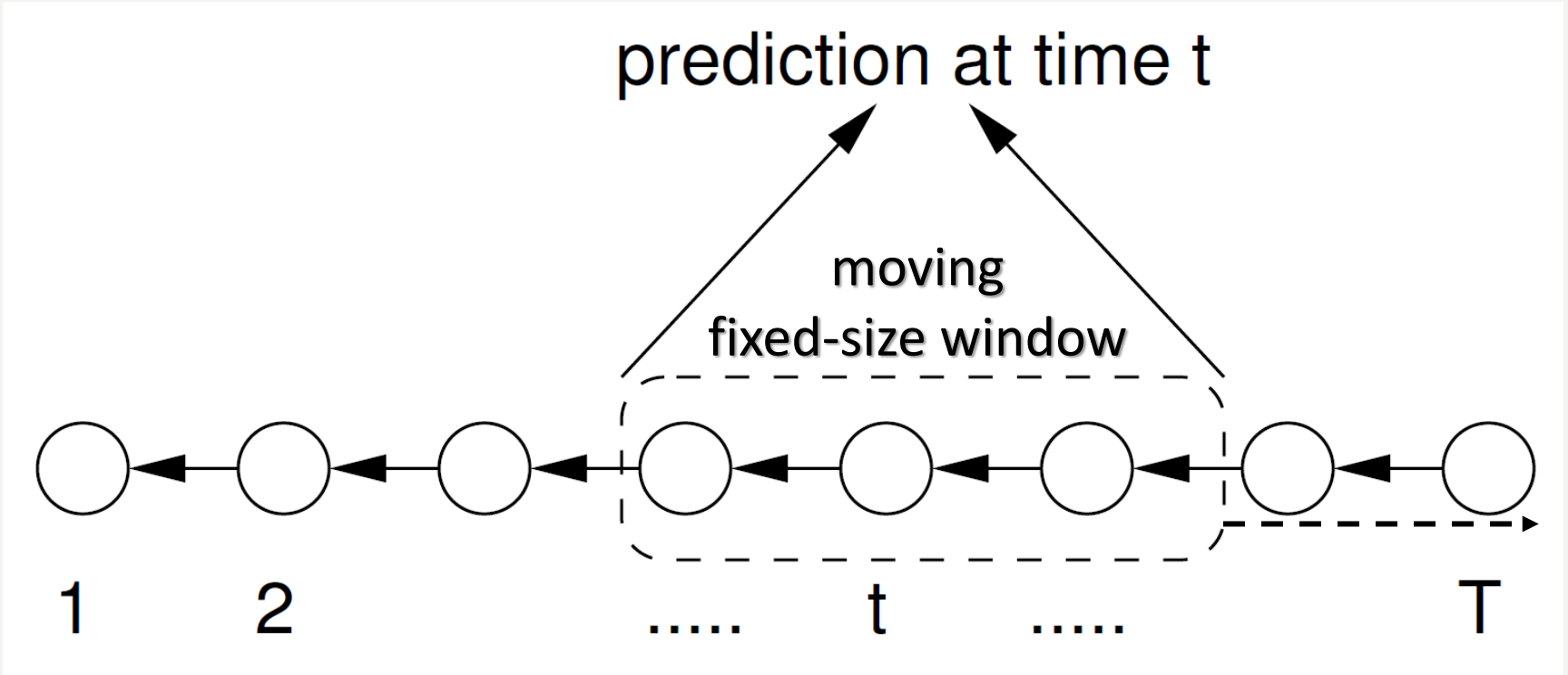
Learning Sequences with Recurrent Neural Networks

Why to use RNN to sequential data?

Fixed-size Windows

Sequences usually model processes (actions, movements) in time and are sequentially processed to predict next data (conclusions, reactions).

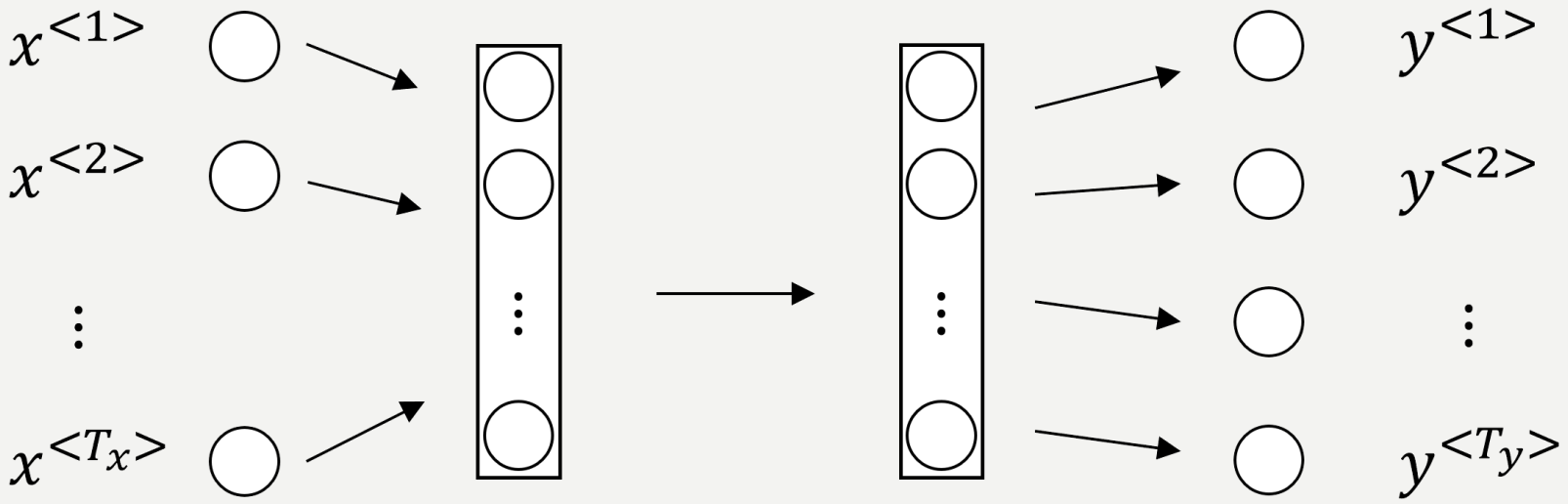
Sequences can have variable length, but typical machine learning models use a fixed number of inputs (fixed-size window) as a prediction context:



A standard network will not work!

When dealing with sequential data (like sentences of words):

- **Inputs and outputs of different examples can usually have different lengths.**
- **The same words in the different examples do not share the same inputs and features learned across different positions of text.**

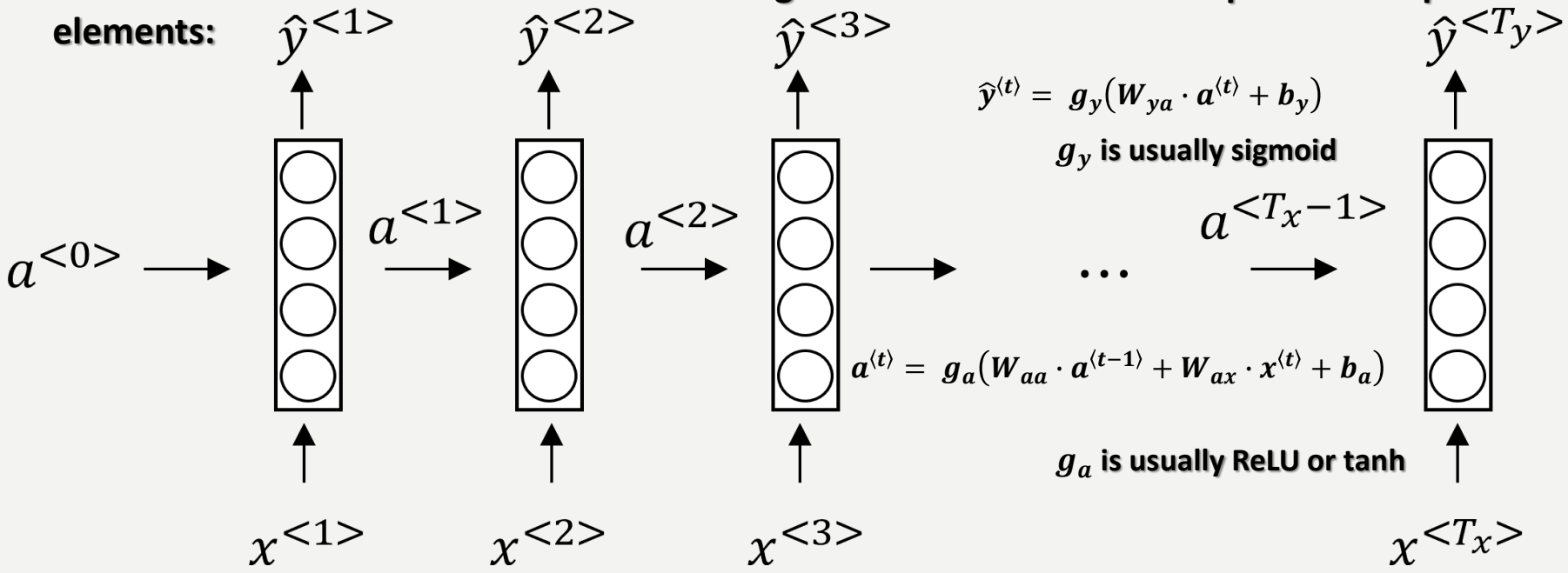


A standard network requires to associate inputs with given features, which cannot move or change over time to train the network well!

We need to find another neural network structure which can work with sequences of inputs (e.g. words) that can move the position in the sequences and take into account the context of previous inputs (like previous words).

Recurrent Neural Networks

We will use recurrent neural networks to overcome the presented difficulties and to allow the network to share features and weights and use the context of previous sequence elements:



In the above network, we put the subsequent elements (e.g. words) on the inputs of the subnetworks which share weights with the other subnetworks (in a nutshell, all these subnetworks are the same network), so the position of the element (word) in the sequence can be different without harm in the representation of this word by the neural network.

Thanks to the connections to the next subnetwork, we can use the context of the processed, previous elements (words) represented by the outputs of previous subnetworks $a^{\langle t \rangle}$.

Simplification of the notation

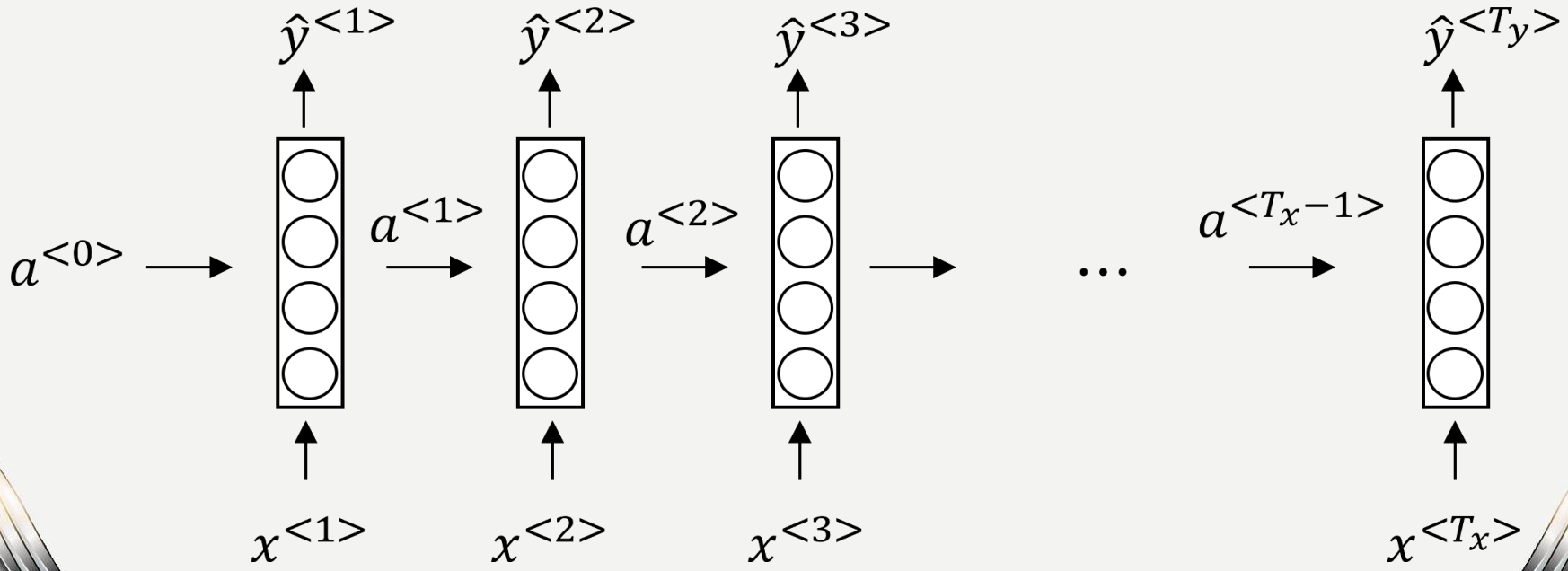
We often use a simplified notation to compute $a^{\langle t \rangle}$ and $\hat{y}^{\langle t \rangle}$ which stacks the weight matrices and also speed up computations a bit because we do not need to operate on two matrices and adding the multiplication results when computing $a^{\langle t \rangle}$ but multiplying only once in parallel:

$$a^{\langle t \rangle} = g_a\left(W_{aa} \cdot a^{\langle t-1 \rangle} + W_{ax} \cdot x^{\langle t \rangle} + b_a\right) = g_a\left(W_a \cdot \left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_a\right)$$

g_a is usually ReLU or tan

$$\hat{y}^{\langle t \rangle} = g_y\left(W_{ya} \cdot a^{\langle t \rangle} + b_y\right) = g_y\left(W_y \cdot a^{\langle t \rangle} + b_y\right)$$

g_y is usually sigmoid

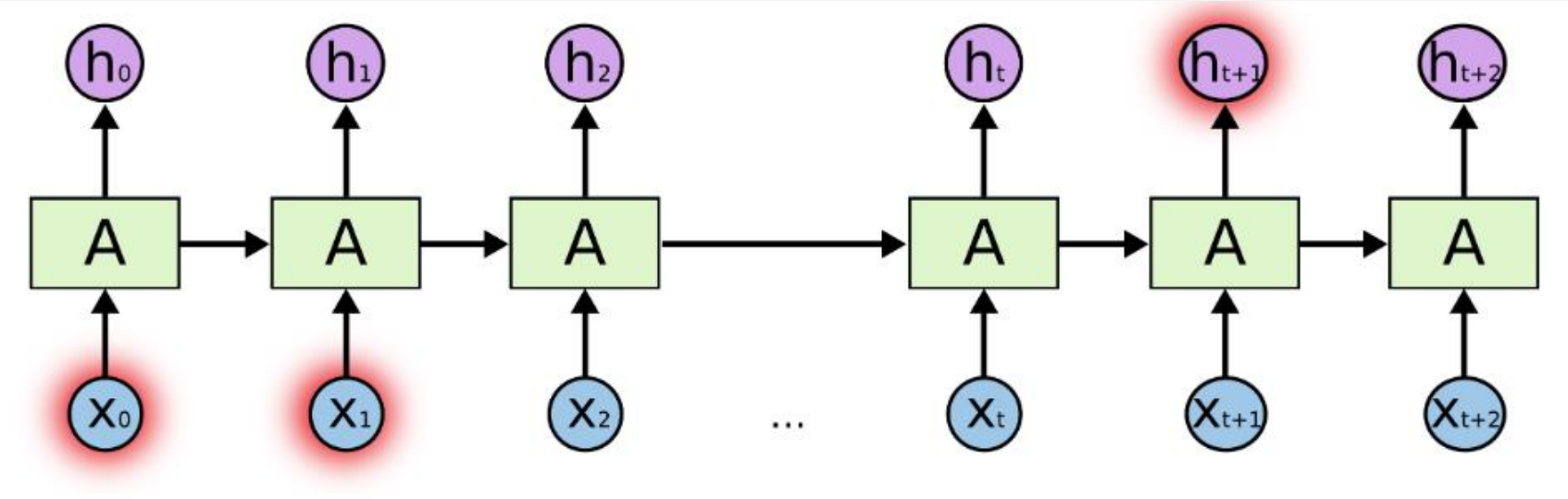


Prediction of Sequence Elements

We can try to predict a next word in a sentence, more generally,
a next element in a sequence, we usually use a few previous words, e.g.:

„I grew up in England. Thanks to it, I speak fluent ……” (English)

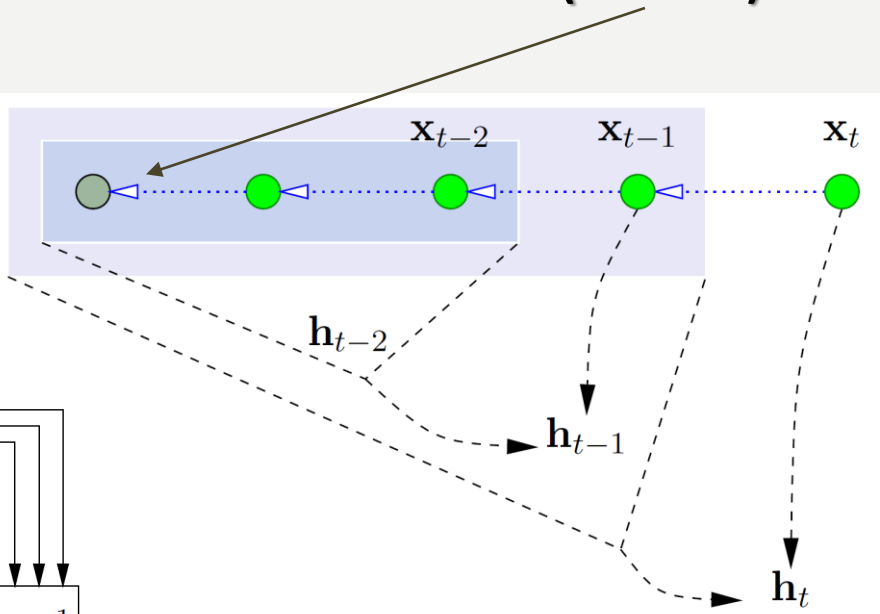
RNNs (e.g. LSTM, GRU) are capable of handling such long-term dependencies.

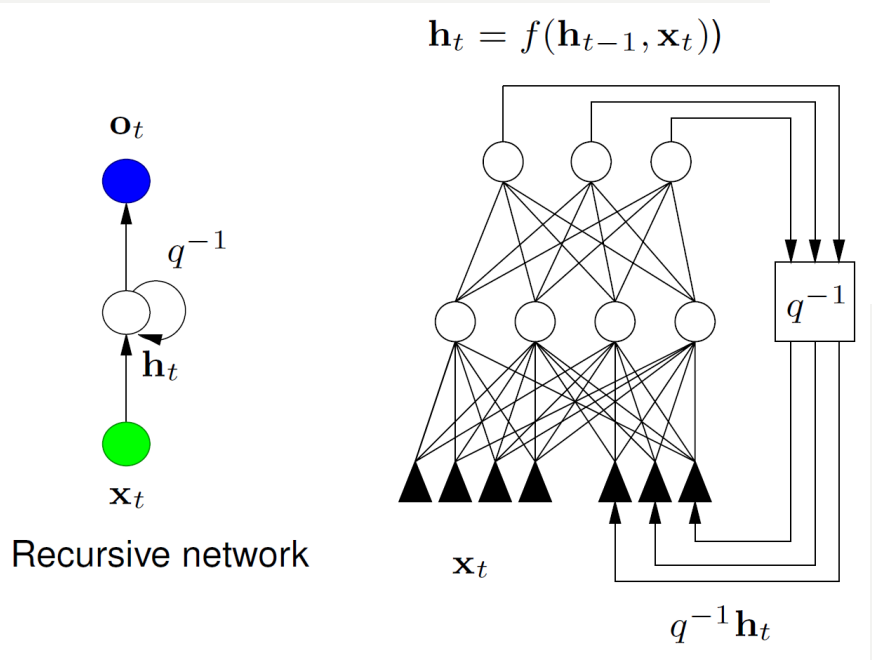


State Transition Function

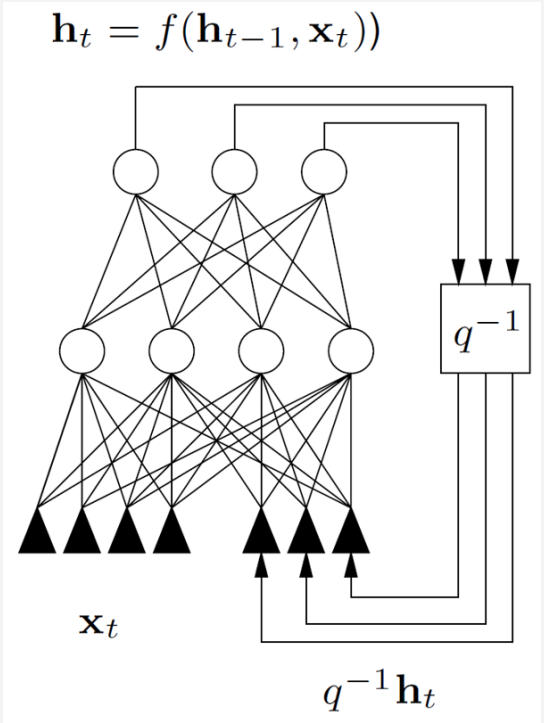
The state transition function defining a single time step can be defined by the shift operator q^{-1}:

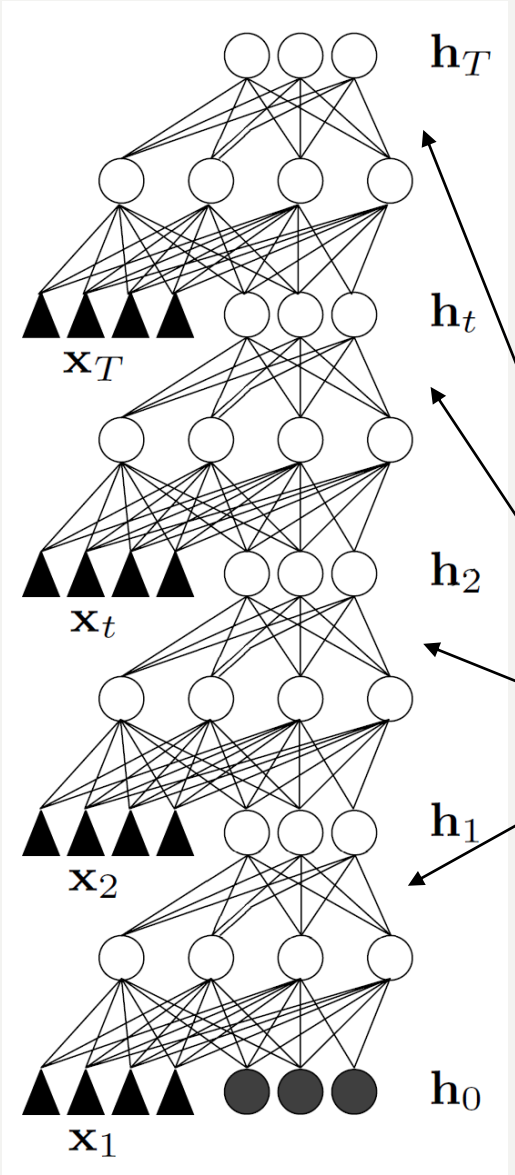
- h_0 – an initial step (at t=0) associated with the external vertex (frontier)
- $h_t = f(h_{t-1}, x_t)$ – t-step
- $q^{-1} h_t = h_{t-1}$ – unitary time delay
- o_t – output (predicted value)





Unfolding Time and Next Sequence Elements



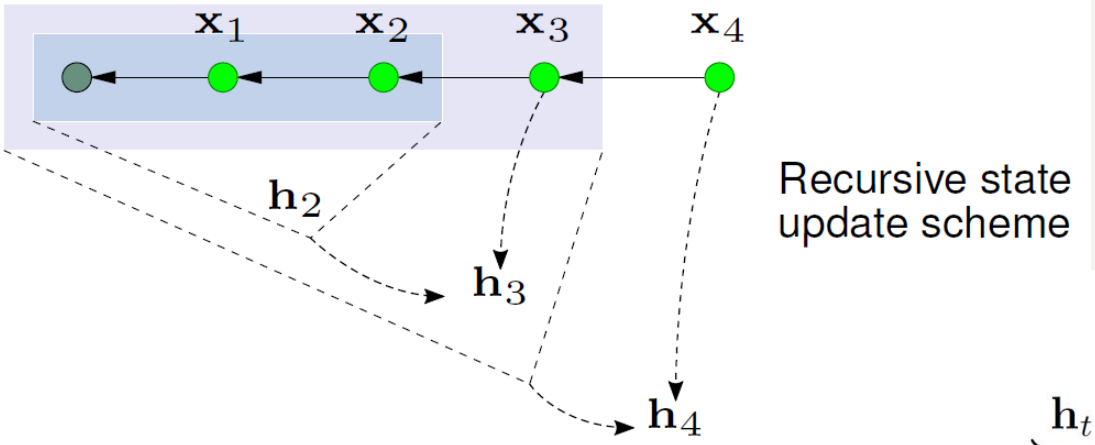


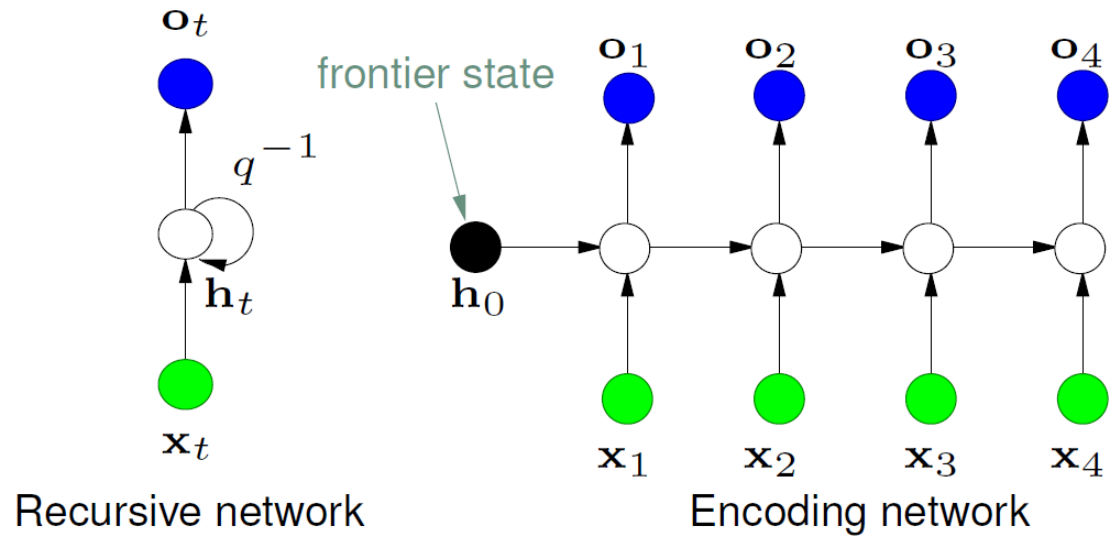
The sequence can be modeled by a deep feedforward neural network which weights can be computed using backpropagation:

- h_T – is the last state of the whole sequence,
- **W – weights are shared** (replicated, the same) between layers.

Encoding Networks

For a given sequence s, the encoding network associated to s is formed by unrolling (time unfolding) the recursive network through the input sequence s:





In linear dynamical systems, we can define:

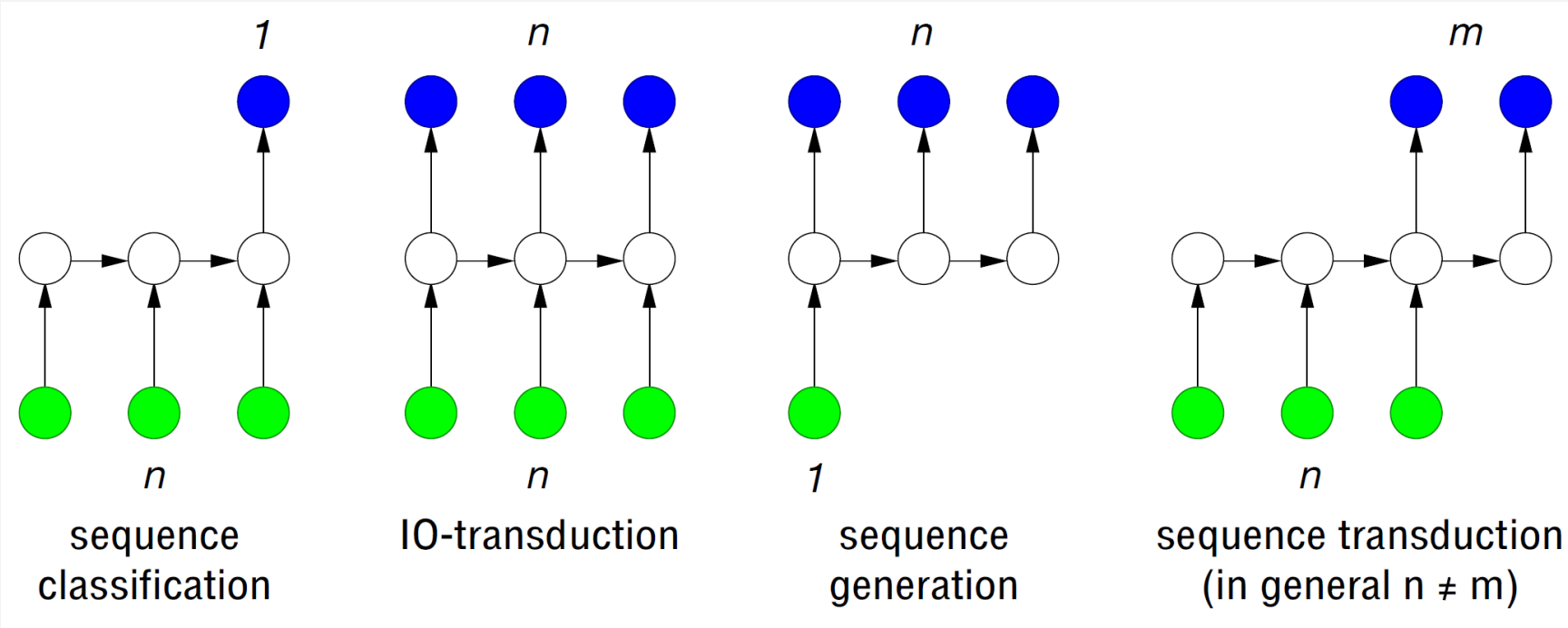
$$\underbrace{\mathbf{h}_t}_{state} = \overbrace{\mathbf{A}}^{\substack{input\\weights}} \underbrace{\mathbf{x}_t}_{input} + \overbrace{\mathbf{B}}^{\substack{hidden\\weights}} \mathbf{h}_{t-1} \qquad \mathbf{h}_0 = \mathbf{0}$$

$$\underbrace{\mathbf{o}_t}_{output} = \underbrace{\mathbf{C}}_{\substack{output\\weights}} \mathbf{h}_t$$

Variety of Sequential Transductions

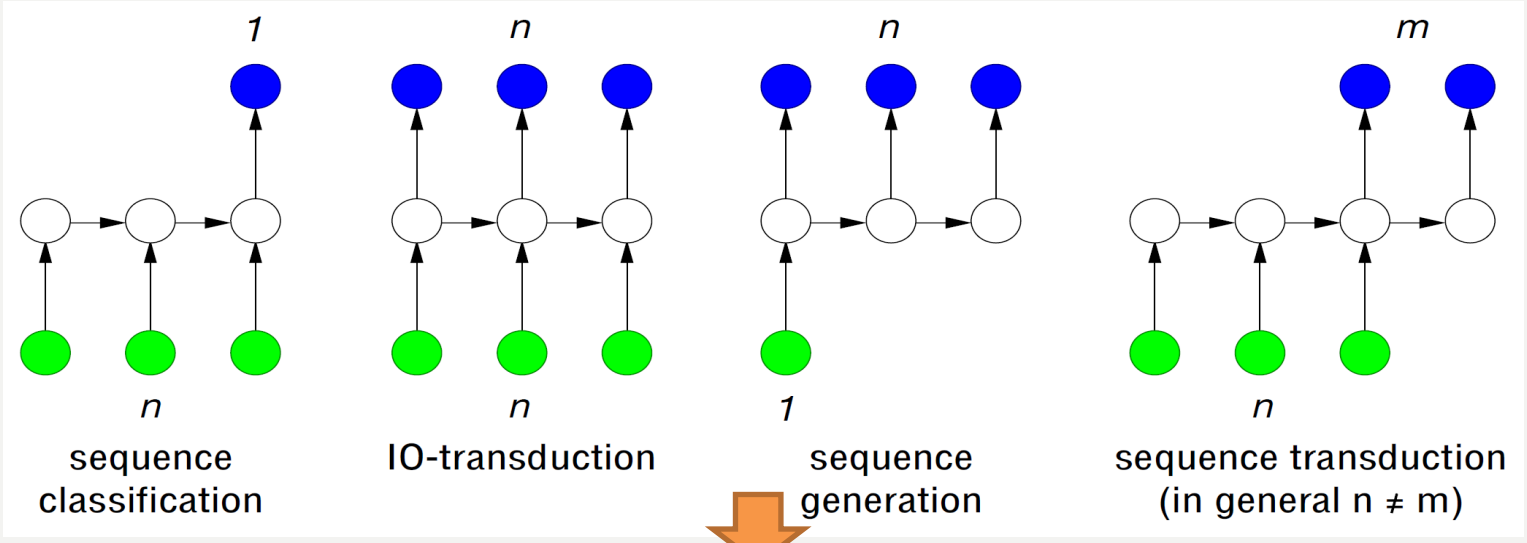
Due to the solved task, we can distinguish various unfolded network structures for:

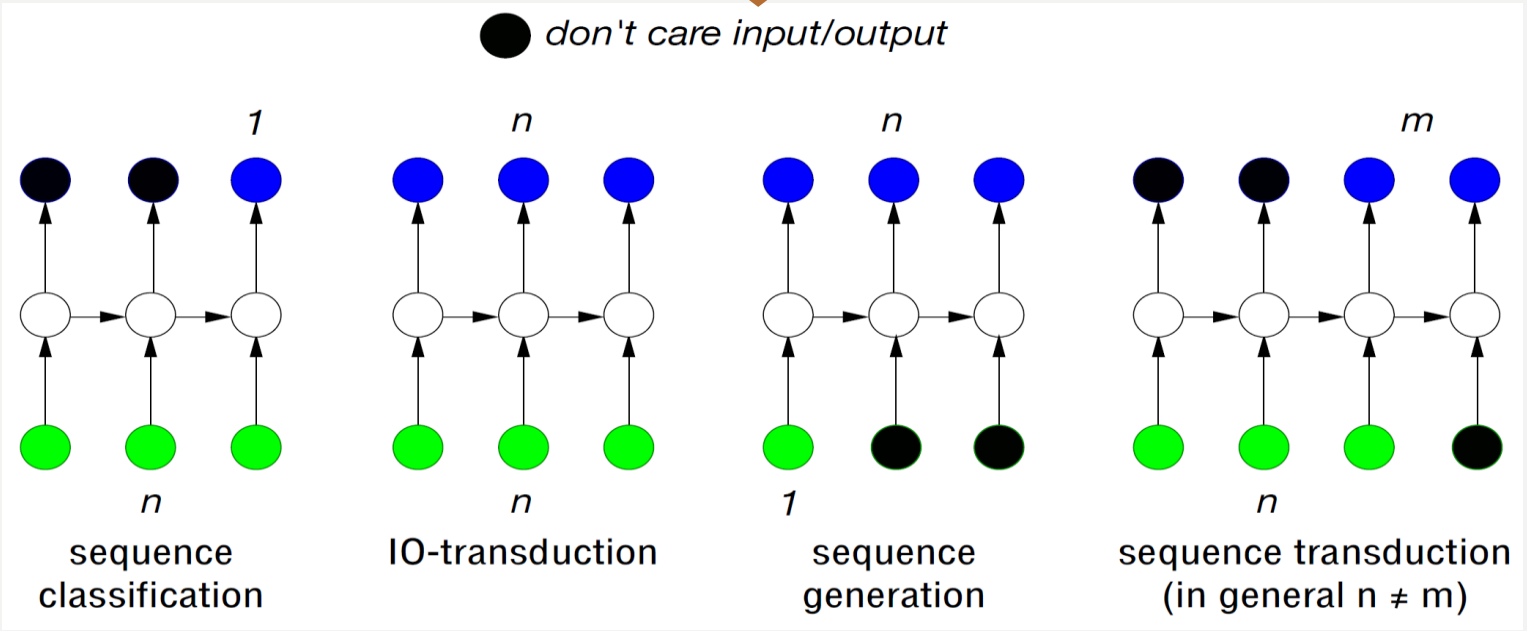
- **Sequence classification (e.g. sentiment classification)**
- **IO transduction (e.g. conversion, transfer)**
- **Sequence generation (e.g. music generation)**
- **Sequence transduction (from one to another, e.g. sequence translation)**



Unification of Various Sequence Tasks

We can easily unify all the presented tasks:



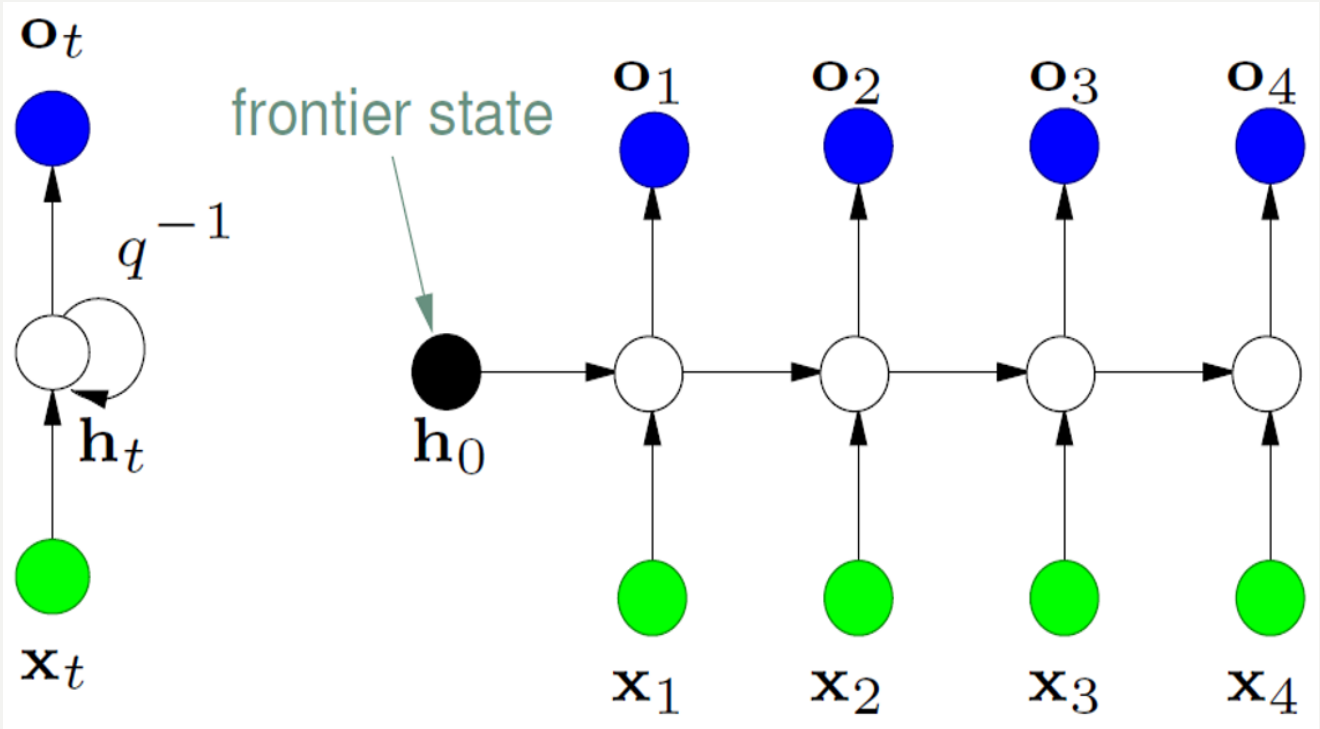


Shallow Recurrent Neural Networks

A shallow Recurrent Neural Network (RNN) defines a non-linear dynamical system:

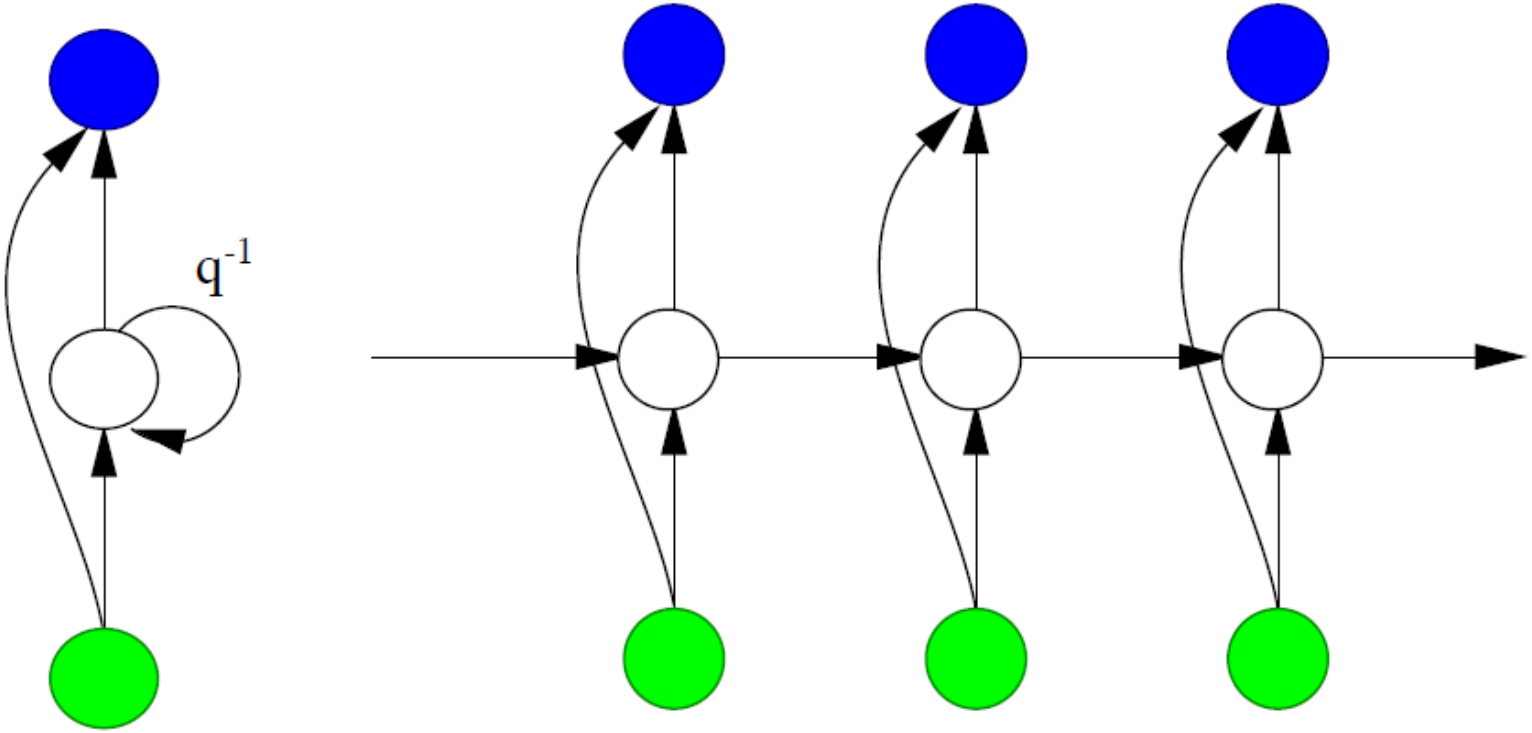
$$\mathbf{h}_t = f(\mathbf{A}\,\mathbf{x}_t + \mathbf{B}\,\mathbf{h}_{t-1})$$
$$\mathbf{o}_t = g(\mathbf{C}\,\mathbf{h}_t)$$

where the functions *f* and *g* are non-linear functions (e.g. *tanh*), and $\mathbf{h}_0 = 0$ or can be learned jointly with the other parameters.



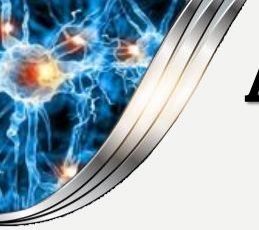
Additional Architectural Features of RNN

We can use additional short-cut connections between inputs and outputs:

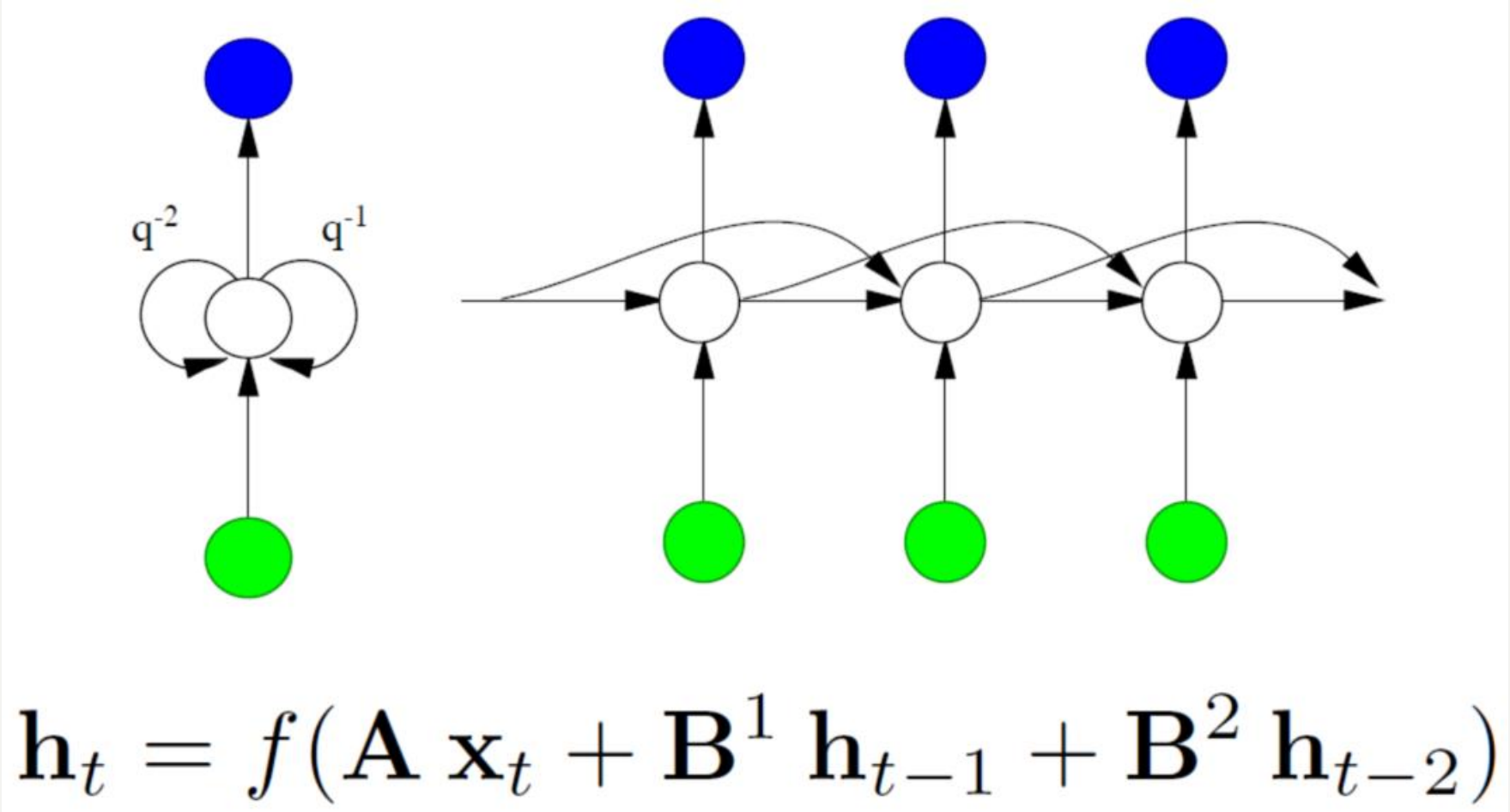


$$\mathbf{o}_t = g(\mathbf{C}\,\mathbf{h}_t + \mathbf{D}\,\mathbf{x}_t)$$

Additional Architectural Features of RNN



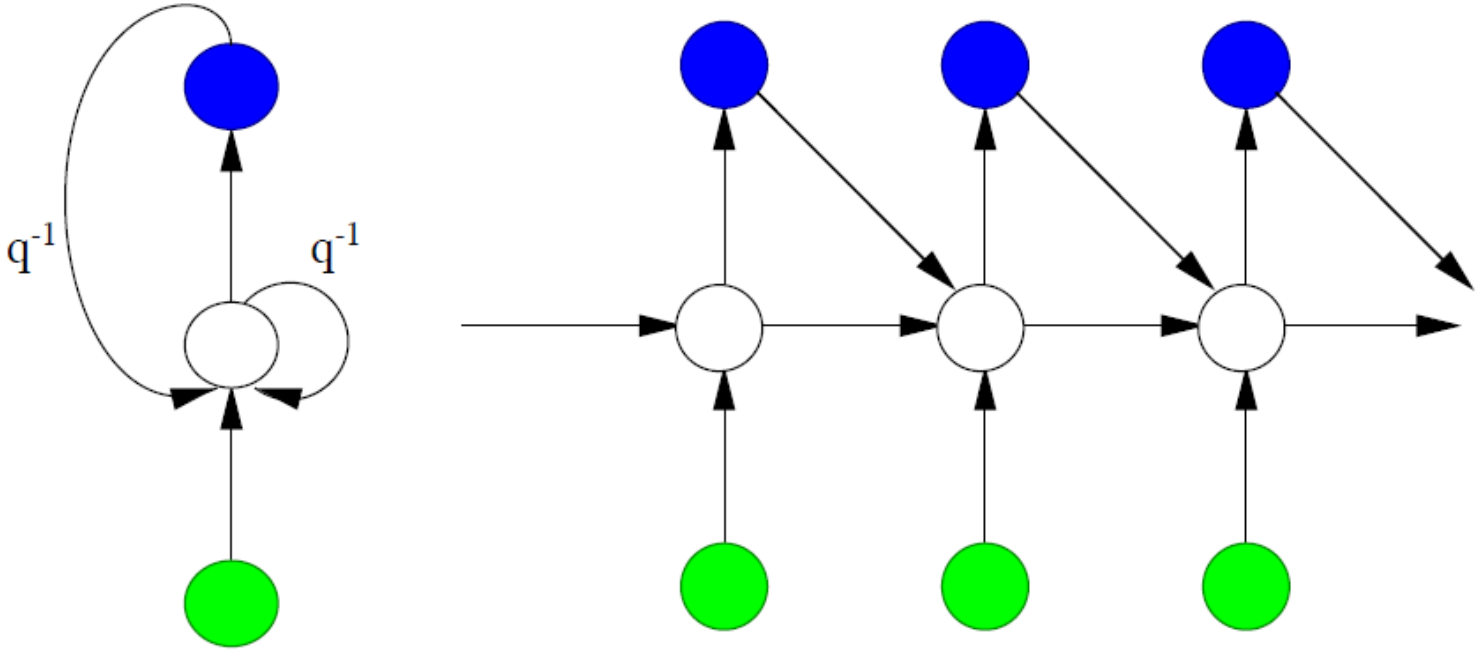
We can use higher-order states and connections between them, e.g. the 2nd order states:



$$\mathbf{h}_t = f(\mathbf{A}\,\mathbf{x}_t + \mathbf{B}^1\,\mathbf{h}_{t-1} + \mathbf{B}^2\,\mathbf{h}_{t-2})$$

Additional Architectural Features of RNN

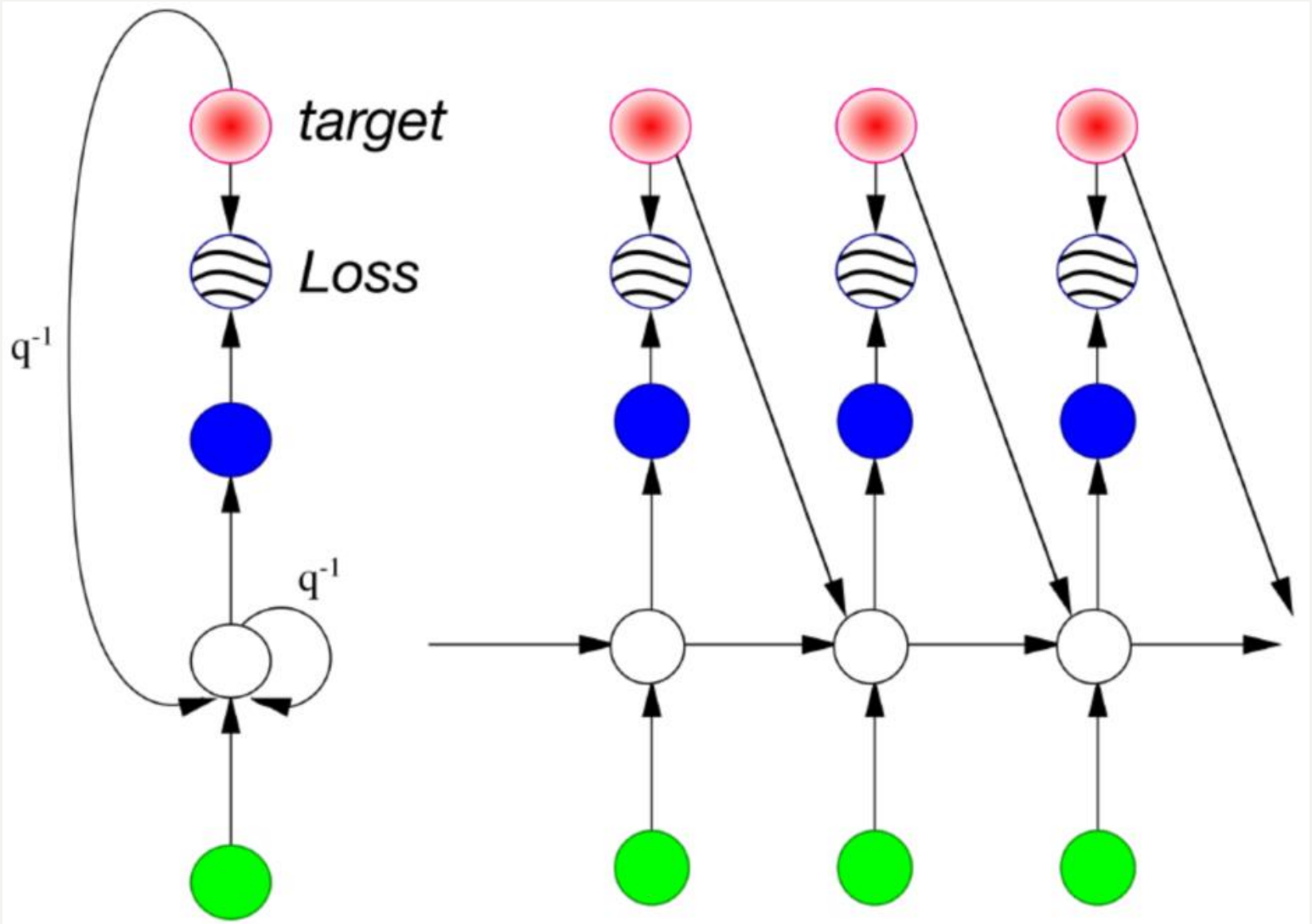
We can use the output to convey contextual information of the previous state:



$$\mathbf{h}_t = f(\mathbf{A}\,\mathbf{x}_t + \mathbf{B}^i\,\mathbf{h}_{t-1} + \mathbf{B}^o\,\mathbf{o}_{t-1})$$

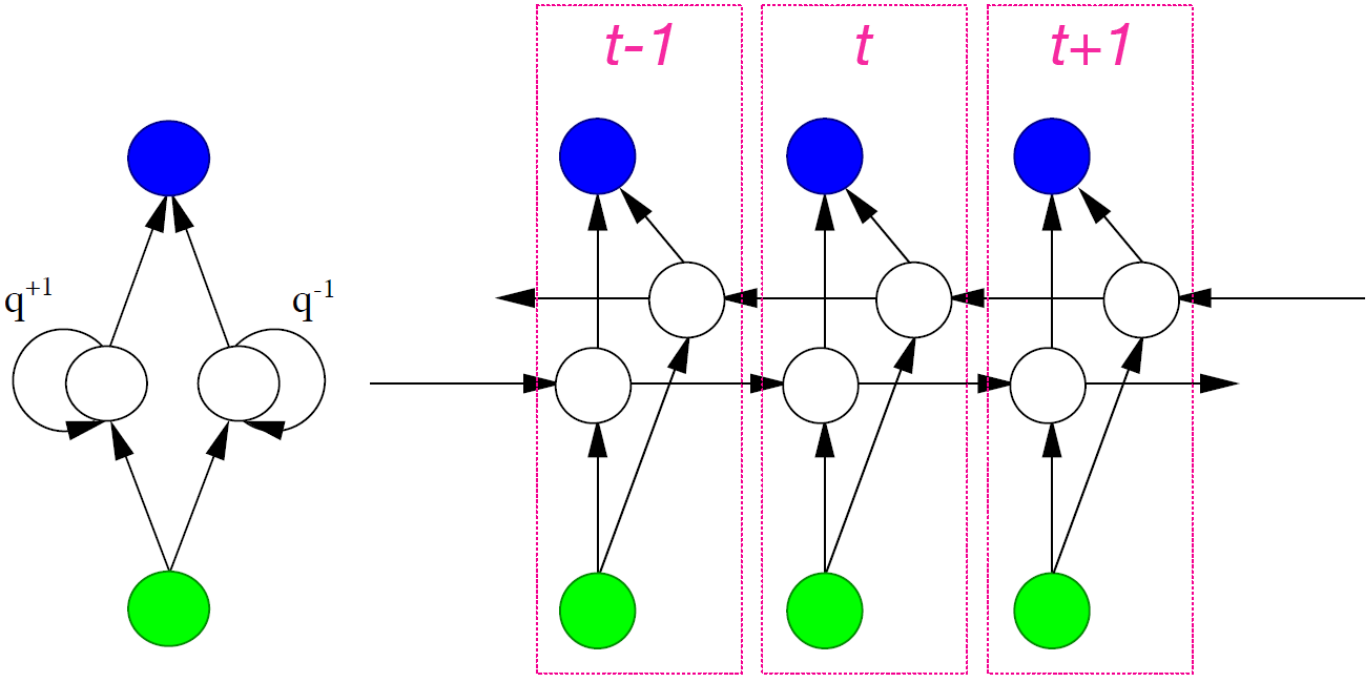
Additional Architectural Features of RNN

We can also force the target signal (presented by a teacher):



Additional Architectural Features of RNN

We can create Bidirectional Recurrent Neural Networks (BRNN) for off-line processing or when the sequences are not temporal to predict not only next but also previous sequence elements:

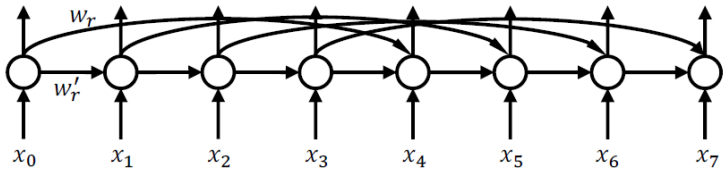


$$\mathbf{o}_t = g(\mathbf{C}^p\,\mathbf{h}_t^p + \mathbf{C}^f\,\mathbf{h}_t^f)$$

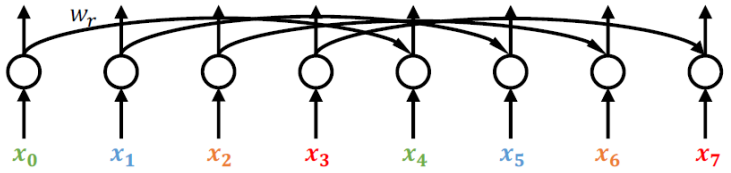
$$\mathbf{h}_t^p = f(\mathbf{A}^p\,\mathbf{x}_t + \mathbf{B}^p\,\mathbf{h}_{t-1}) \qquad \mathbf{h}_t^f = f(\mathbf{A}^f\,\mathbf{x}_t + \mathbf{B}^f\,\mathbf{h}_{t+1})$$

Deep Dilated Recurrent Neural Networks

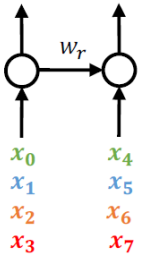
We can also stack recurrent layers and combine various approaches:



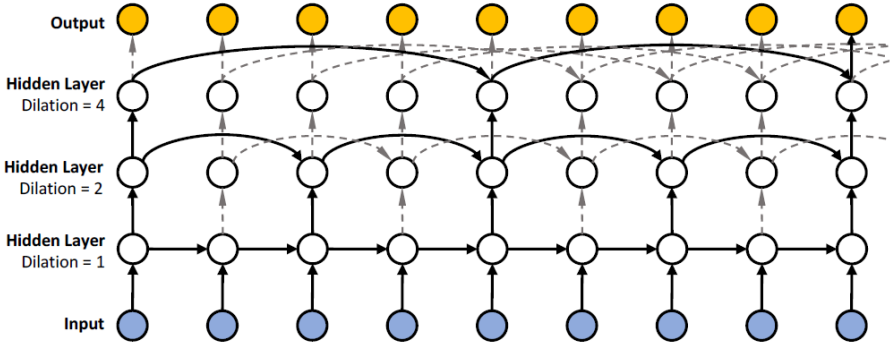
single-layer RNN with recurrent skip connections



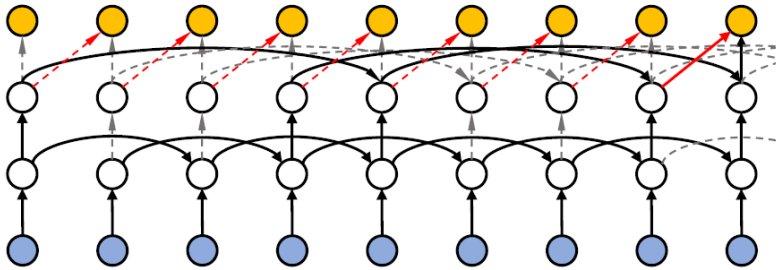
same RNN with dilated recurrent skip connections



Deep Dilated Recurrent Neural Networks



$$s^{(l)} = M^{l-1}, l = 1, \cdots, L$$



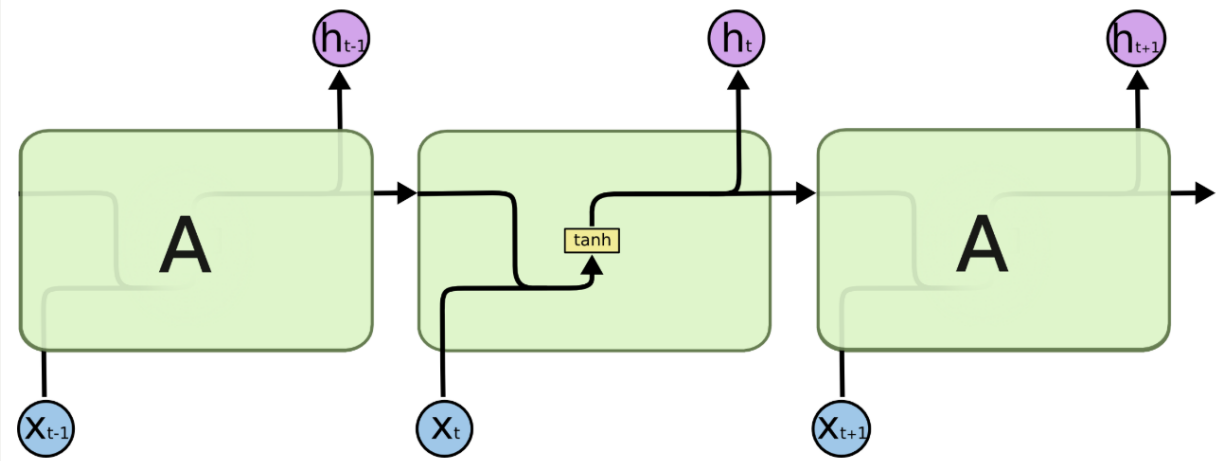
$$s^{(l)} = M^{(l-1+l_0)}, l = 1, \cdots, L \text{ and } l_0 \geq 0$$

convolutional layer is appended as the final layer (red arrows)

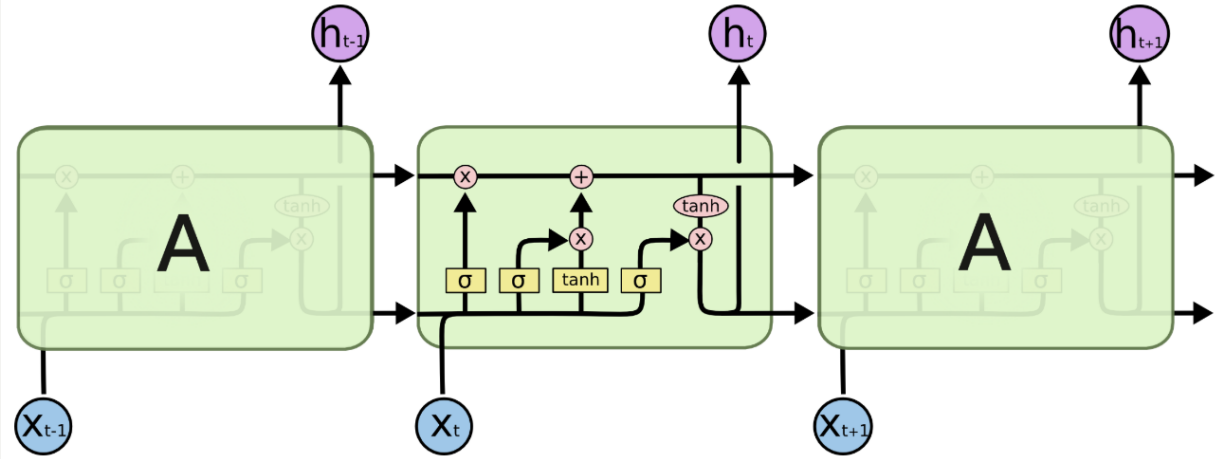
Dilated connections can help to carry the context of previous states.

Long Short-Term Memory (LSTM)

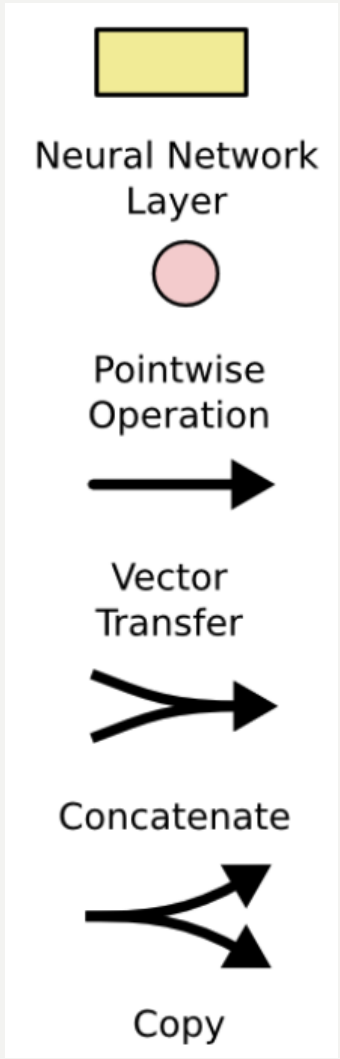
Long Short-Term Memory networks are a special kind of Recurrent Neural Networks, containing four (instead of one) interacting layers and capable of learning long-term dependencies.



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



The repeating module in an LSTM contains four interacting layers.

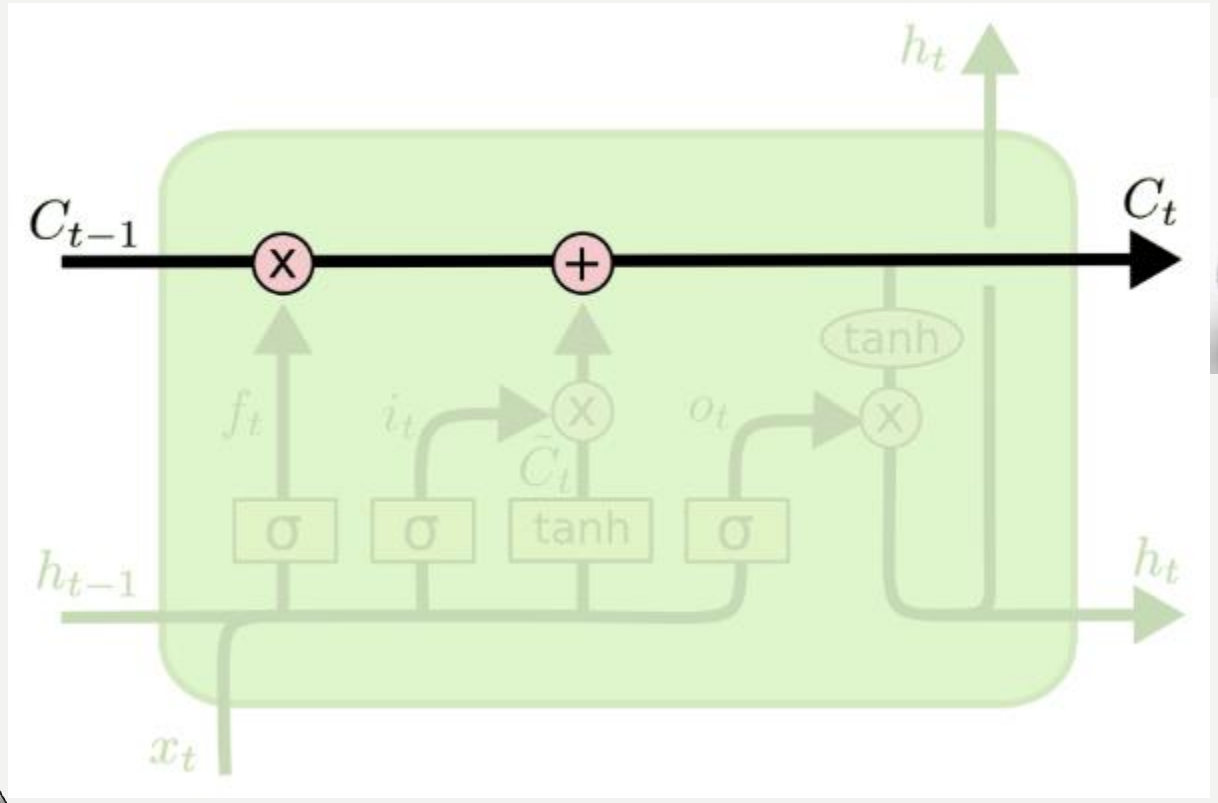


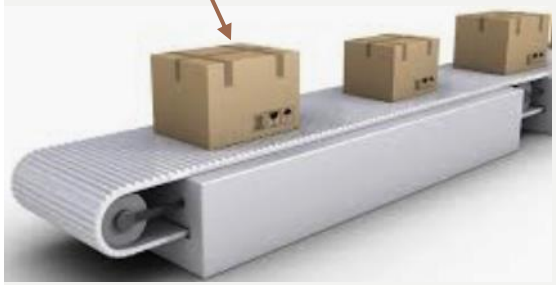
Cell State of LSTMs

The key to LSTM is the cell state represented by the horizontal line running through the top of the diagram. It is a kind of conveyor belt.

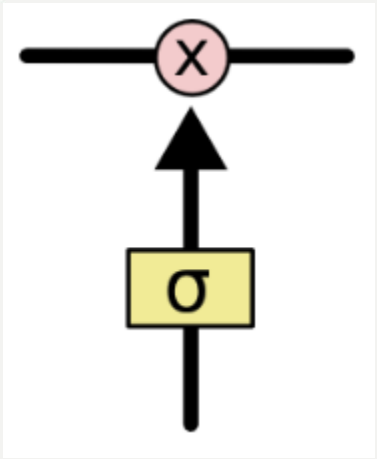
It runs straight down the entire chain, with only some minor linear interactions.

The LSTM has the ability to remove or add information to the cell state, carefully regulated by structures called gates.





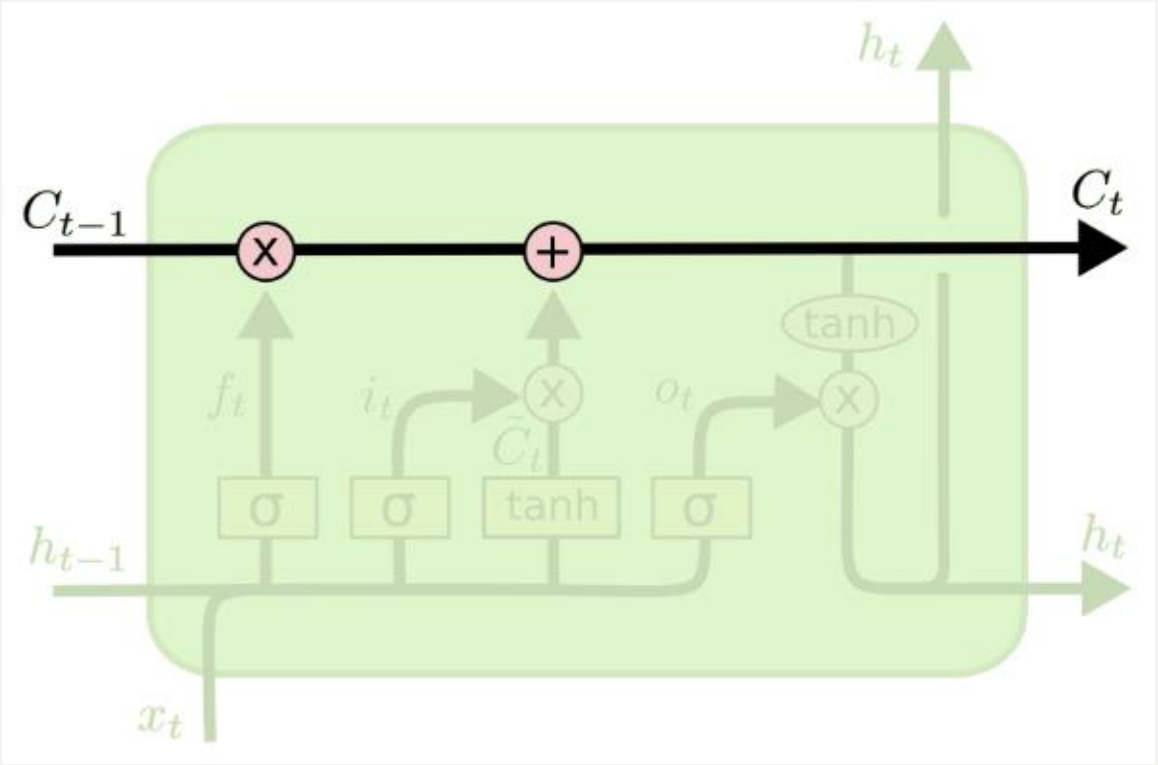
Gates of LSTMs



Gates are a way to optionally let information through.

They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of **zero** means "**let nothing through**," while a value of **one** means "**let everything through!**"

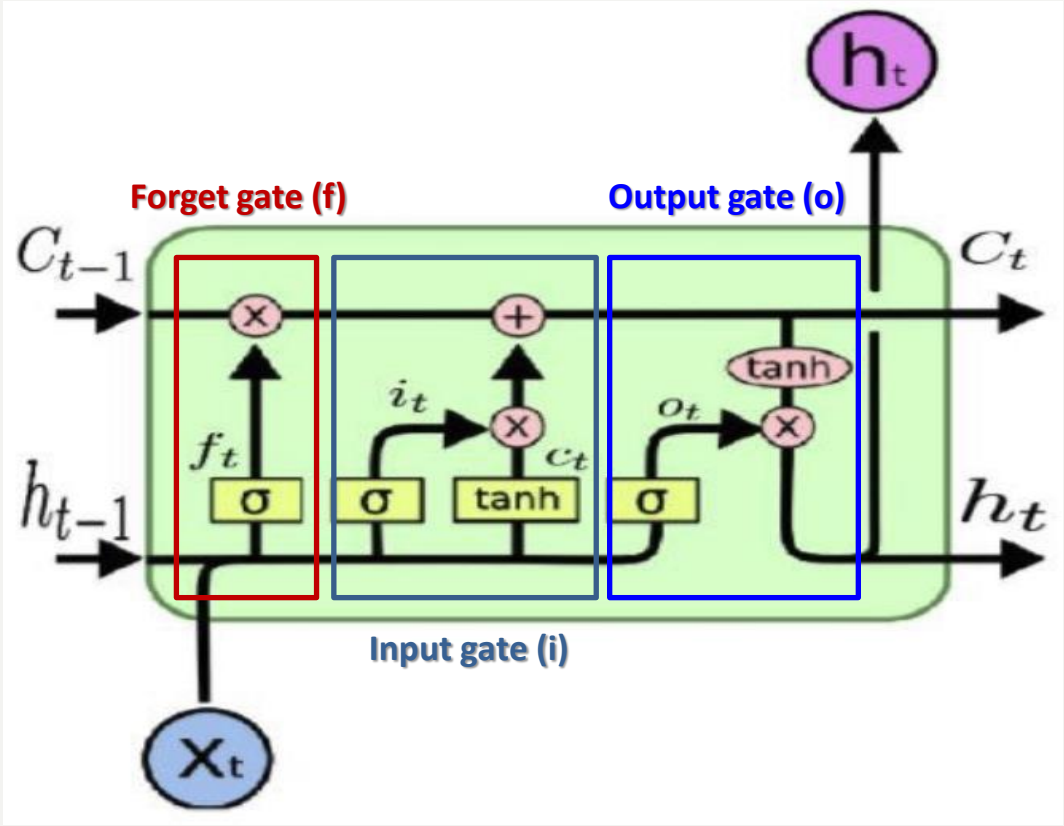


An LSTM has three of these gates, to protect and control the cell state.

Long Short-Term Memory Gates

A simple LSTM cell consists of three gates:

- **Forget gate (f)** – whether and to what extend to forget (erase) the previous C_{t-1} cell
- **Input gate (i)** – it controls writing to the cell and how strong the given input influence the output result and combines it with the previous cell output
- **Output gate (o)** – how much to reveal the cell and use for computing the output h_t



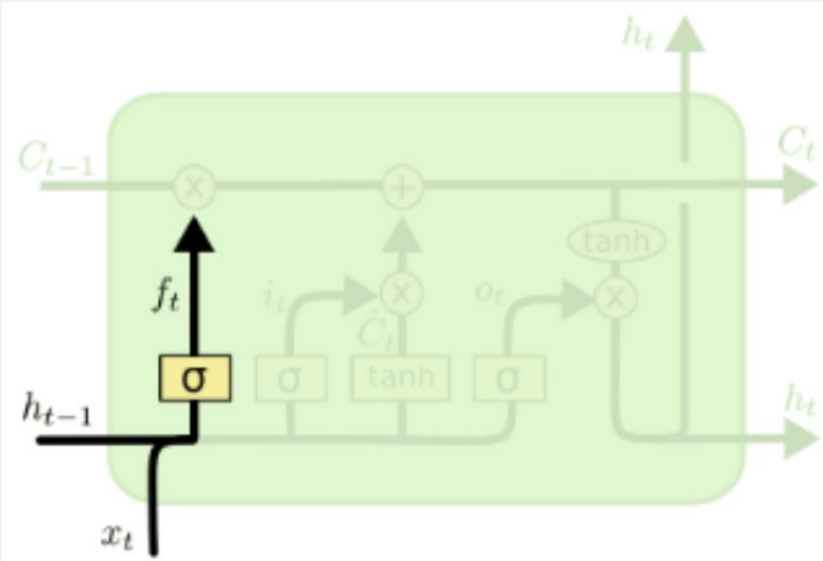
Long Short-Term Memory Gates

In the first step, the LSTM decides by a sigmoid layer called the "forget gate layer" what information is let to go throw away from the cell state.

The forget gate (o) of a simple LSTM cell takes the decision about what must be removed from the C_{t-1} state after getting the output of the previous state, and it thus keeps only the relevant stuff.

It is surrounded by a sigmoid function σ which crushes the input between [0, 1].

We multiply the forget gate with previous cell state to forget the unnecessary stuff from the previous state which is not needed anymore.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

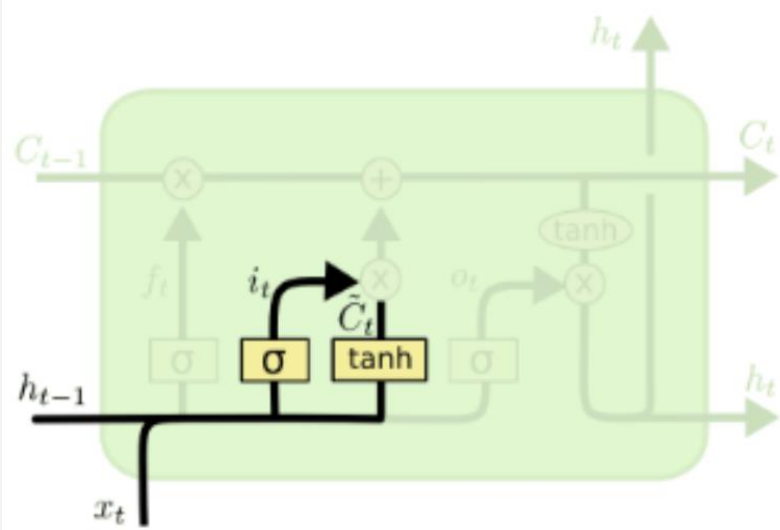
Long Short-Term Memory Gates

In the next step, the LSTM decides what new information will be stored in the cell state: First, a *sigmoid layer* σ called the input gate layer decides which values we shall update. Next, a *tanh layer* creates a vector of new candidate values $\widetilde{C}_t$ that could be added to the state.

In the next step, we shall combine these two to create an update to the state.

The input gate (i) of a simple LSTM decides about the addition of new stuff from the present input to our present cell state scaled by how much we wish to add them.

The sigmoid layer σ decides which values to be updated and *tanh* layer creates a vector for new candidates to added to the present cell state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

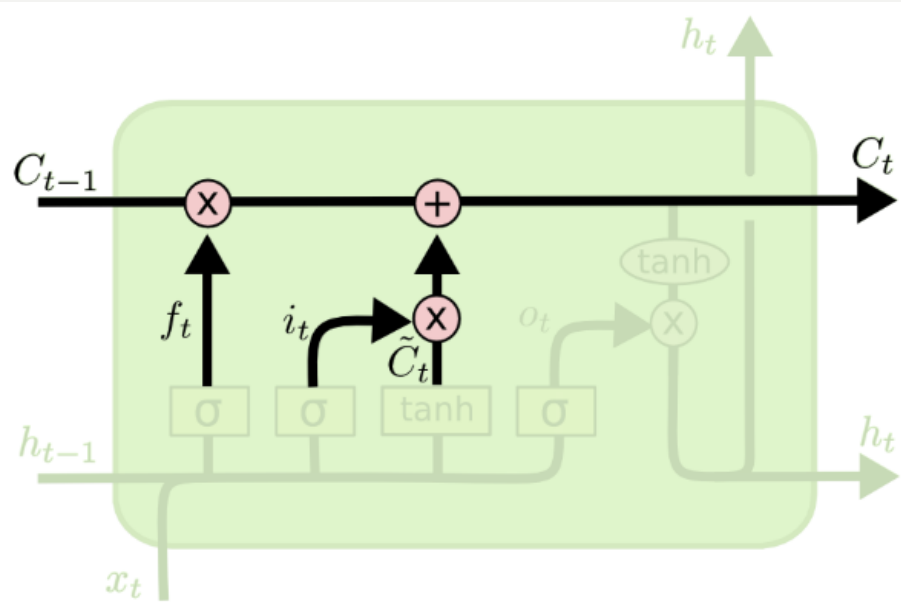
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Long Short-Term Memory Gates

In the third step, the LSTM updates the old cell state C_{t-1} into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

We can actually drop the information about the old subject's attribute and add the new information, as we decided in the previous steps.



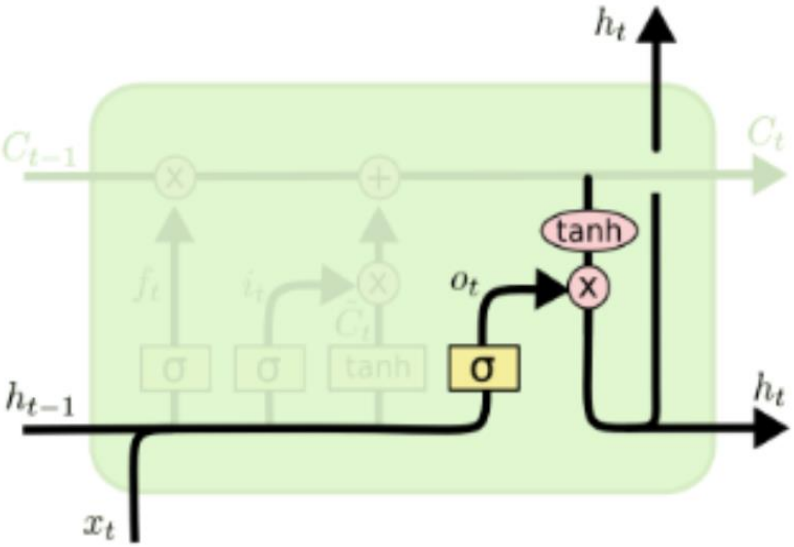
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Long Short-Term Memory Gates

Finally, the LSTM decides what is going to the output based on our cell state, but will be a filtered version. First, a sigmoid layer σ decides what parts of the cell state go to the output. Then, the cell state is put through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that only the parts are sent to the output.

The output gate (o) of a simple LSTM cell decides what to output from the cell state which will be done by the sigmoid function σ.

The input x_t is multiplied with *tanh* to crush the values between (-1,1) and then multiply it with the output of sigmoid function:

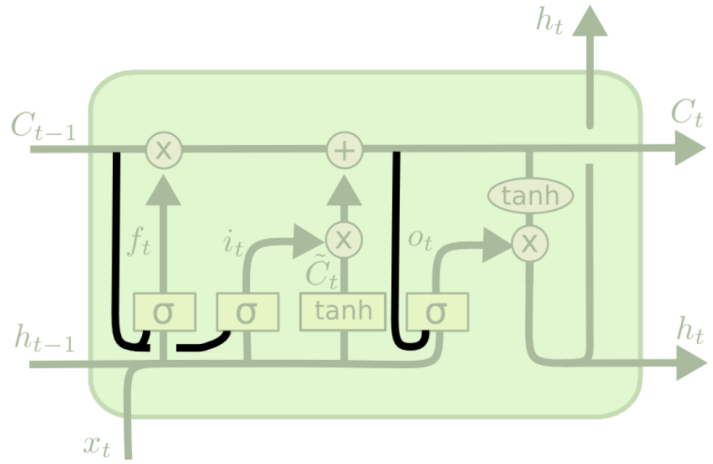


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

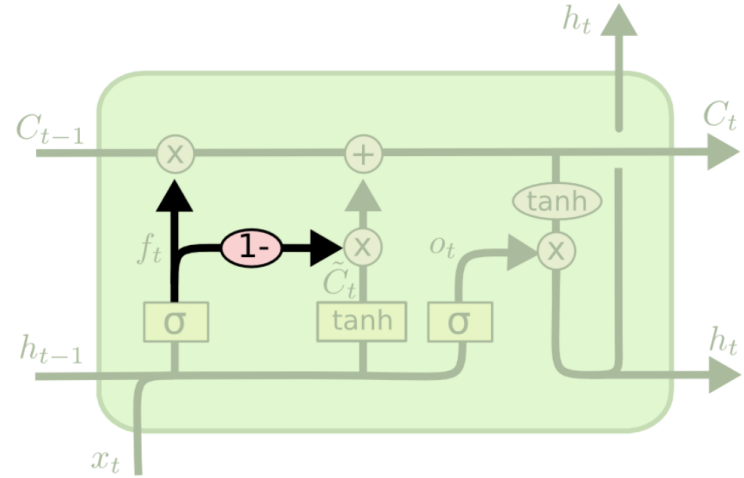
Variants of LSTM

Peephole connections can be added to some or all the gates of the LSTM cells:



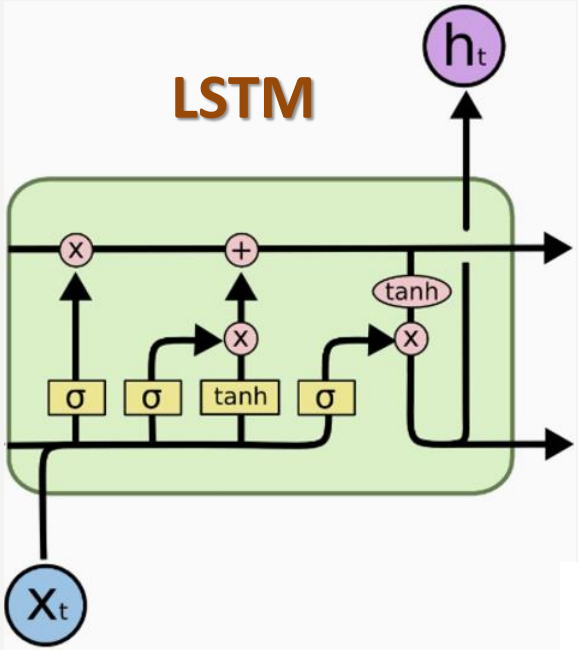
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

The forget gate can be coupled to forget only when we are going to put something in the place of the forgotten older state:



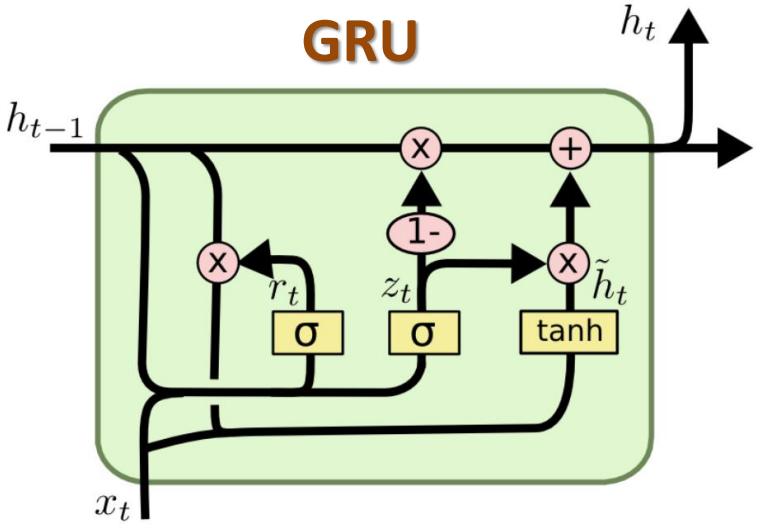
$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

Gated Recurrent Unit (GRU)



The gated recurrent unit combines the forget and input gates into a single update gate and merges the cell state and hidden state together with some other minor changes.

In result, the GRU units are simpler and computationally faster than the LSTM units:



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

RNN, LSTM, GRU in Keras

How to use recurrent layers in Keras?

Adding Simple RNN layers

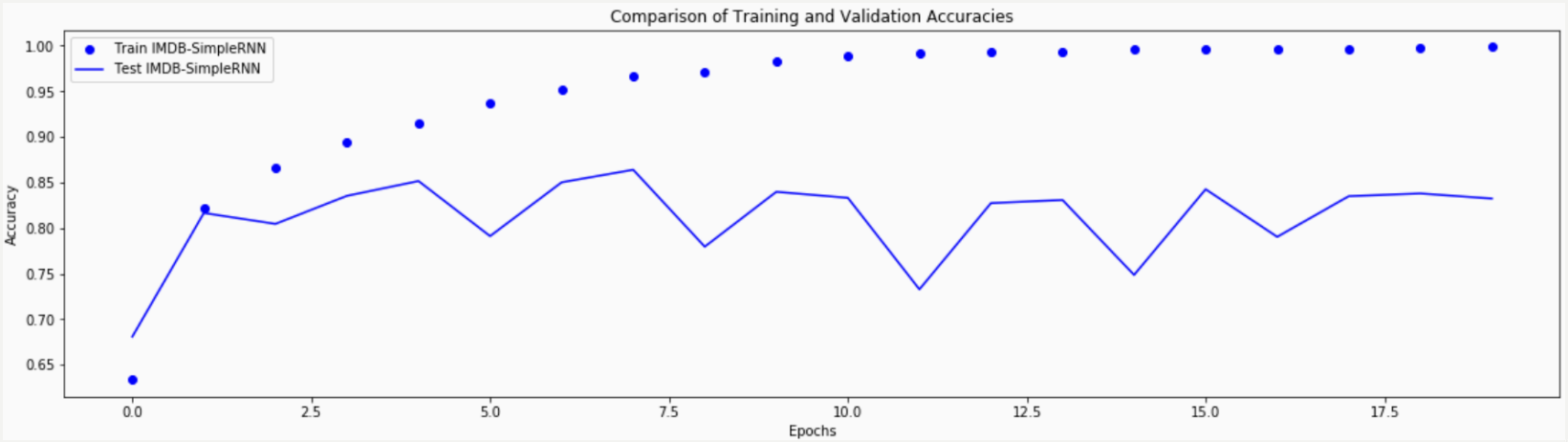
We add SimpleRNN layer(s) after the Embedding layer:

```
from keras.layers import Dense

epochs = 20
batch_size = 256

modelIMDBSimpleRNN = Sequential()
modelIMDBSimpleRNN.add(Embedding(max_features, 32))
modelIMDBSimpleRNN.add(SimpleRNN(32))
modelIMDBSimpleRNN.add(Dense(1, activation='sigmoid'))

modelIMDBSimpleRNN.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
historyIMDBSimpleRNN = modelIMDBSimpleRNN.fit(input_train, y_train,
                                              epochs = epochs,
                                              batch_size = batch_size,
                                              validation_split = 0.2)
```



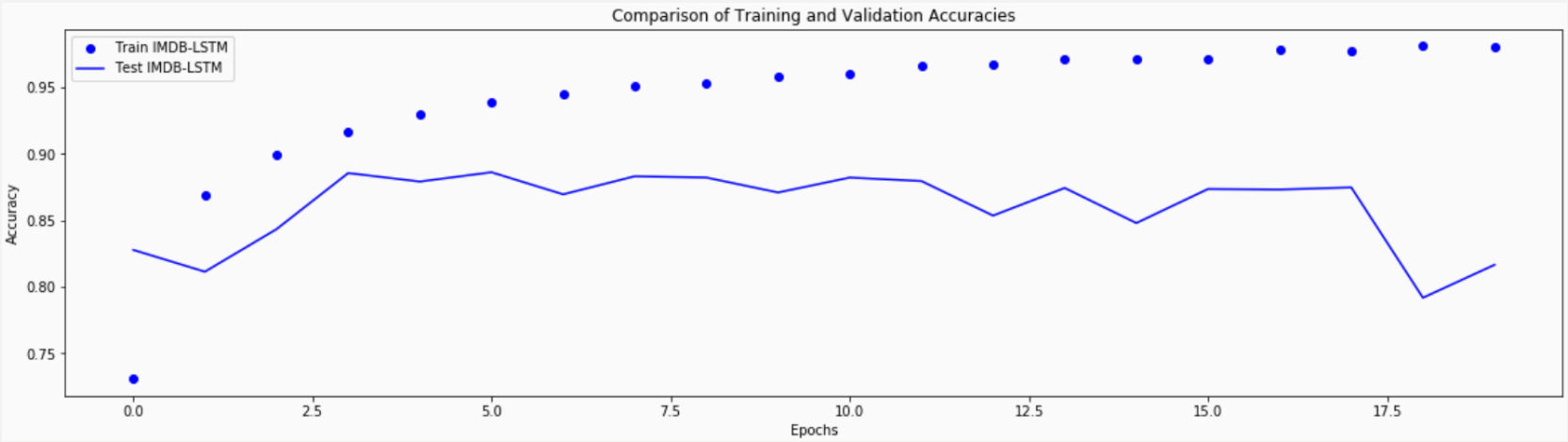
Adding LSTM layers

Similarly, we can also add LSTM layer(s):

```
from keras.layers import LSTM

modelIMDBLSTM = Sequential()
modelIMDBLSTM.add(Embedding(max_features, 32))
modelIMDBLSTM.add(LSTM(32))
modelIMDBLSTM.add(Dense(1, activation='sigmoid'))

modelIMDBLSTM.compile(optimizer='rmsprop',
                      loss='binary_crossentropy',
                      metrics=['acc'])
historyIMDBLSTM = modelIMDBLSTM.fit(input_train, y_train,
                                    epochs=epochs,
                                    batch_size=batch_size,
                                    validation_split=0.2)
```



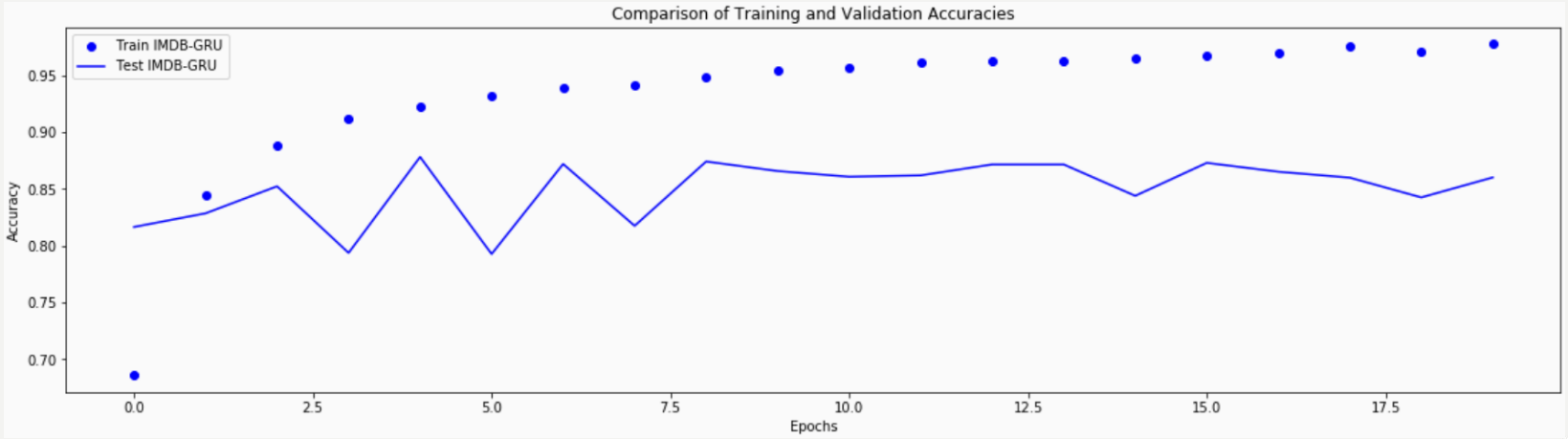
Adding GRU layers

Similarly, we can also add GRU layer(s):

```
from keras.layers import GRU

modelIMDBGRU = Sequential()
modelIMDBGRU.add(Embedding(max_features, 32))
modelIMDBGRU.add(GRU(32))
modelIMDBGRU.add(Dense(1, activation='sigmoid'))

modelIMDBGRU.compile(optimizer='rmsprop',
                     loss='binary_crossentropy',
                     metrics=['acc'])
historyIMDBGRU = modelIMDBGRU.fit(input_train, y_train,
                                  epochs=epochs,
                                  batch_size=batch_size,
                                  validation_split=0.2)
```



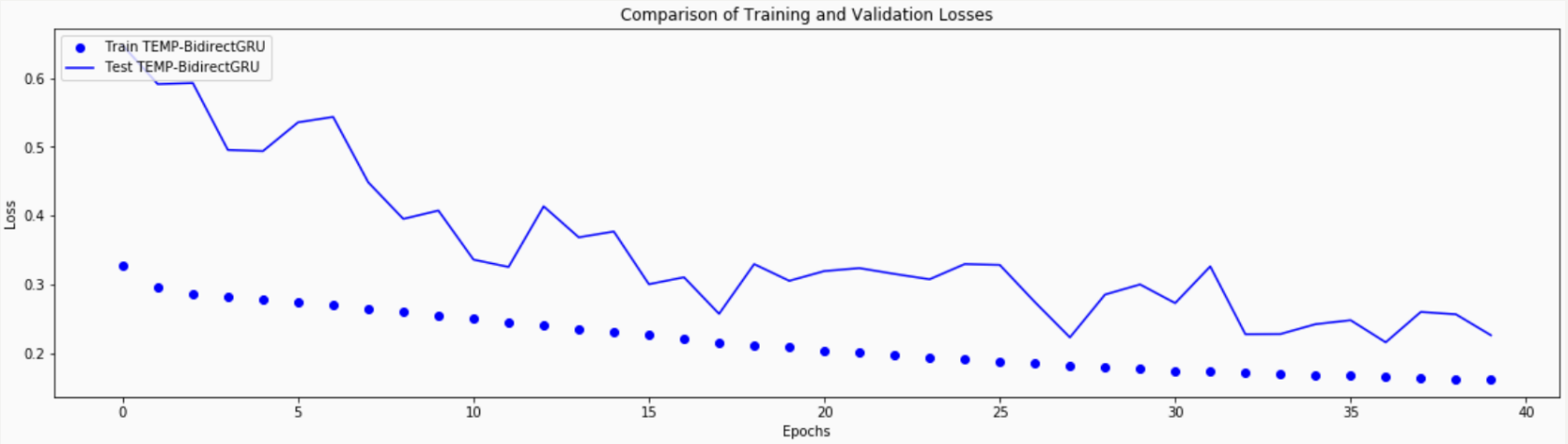
Using Bidirect GRU model

To achieve better results we can use a bidirect model(s):

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

modelTEMPBidirectGRU = Sequential()
modelTEMPBidirectGRU.add(layers.Bidirectional(layers.GRU(32, kernel_regularizer=regularizers.l2(0.001)),
                                              input_shape=(None, float_data.shape[-1])))
modelTEMPBidirectGRU.add(layers.Dense(1))

modelTEMPBidirectGRU.compile(optimizer=RMSprop(), loss='mae')
historyTEMPBidirectGRU = modelTEMPBidirectGRU.fit_generator(train_gen,
                                                            steps_per_epoch=500,
                                                            epochs=40,
                                                            validation_data=val_gen,
                                                            validation_steps=val_steps)
```



Training Difficulties

How to overcome some training difficulties?

NaN loss or validation loss

What can we do when NaN loss or validation loss happens to our training:

- **Try normalizing your data, or inspect your normalization process for any bad values introduced, i.e. Add(BatchNormalization()) layer to your network to prevent exploding gradients.**
- **Add regularization to add l1 or l2 penalties to the weights. Otherwise, try a smaller l2 reg. i.e. l2(0.001), or remove it if already exists.**
- **Try a smaller Dropout rate (0.2, 0.1, 0.05, or even less).**
- **Clip the gradients to prevent their explosion, e.g., you could use clipnorm=1. or clipvalue=1. as parameters for your optimizer.**
- **Check the validity of inputs (no NaNs or sometimes 0s). i.e. df.isnull().any()**
- **Replace optimizer with Adam which is easier to handle.**
  Sometimes also replacing sgd with rmsprop would help.
- **Use RMSProp with heavy regularization to prevent gradient explosion.**
- **Verify that you are using the right activation function (e.g. using a softmax instead of sigmoid for multiple class classification).**
- **Try to increase the batch size (e.g. 32 to 64 or 128) to increase the stability of your optimization.**
- **Check the size of your last batch which may be different from the batch size.**

BIBLIOGRAPY

1. Francois Chollet, “Deep learning with Python”, Manning Publications Co., 2018.
2. Ian Goodfellow, Yoshua Bengio, Aaron Courville, “Deep Learning”, MIT Press, 2016, ISBN 978-1-59327-741-3.
3. Home page for this course: http://home.agh.edu.pl/~horzyk/lectures/ahdydci.php
4. Nikola K. Kasabov, Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence, In Springer Series on Bio- and Neurosystems, Vol 7., Springer, 2019.
5. Holk Cruse, Neural Networks as Cybernetic Systems, 2nd and revised edition
6. R. Rojas, Neural Networks, Springer-Verlag, Berlin, 1996.
7. Convolutional Neural Network (Stanford)
8. Visualizing and Understanding Convolutional Networks, Zeiler, Fergus, ECCV 2014.

BIBLIOGRAPY

10. Home page for this course:
    http://home.agh.edu.pl/~horzyk/lectures/ahdydci.php
11. LSTM cells from scratch and the code
12. Understanding LSTM
13. Francois Chollet, Deep Learning with Python,
    Manning, Nov. 2017, ISBN 9781617294433
14. Understanding GRU