Deep Learning Train-the-Trainer Workshop

Recurrent Neural Networks

Uwe Klemradt

RWTH Aachen University

Starting point: standard network

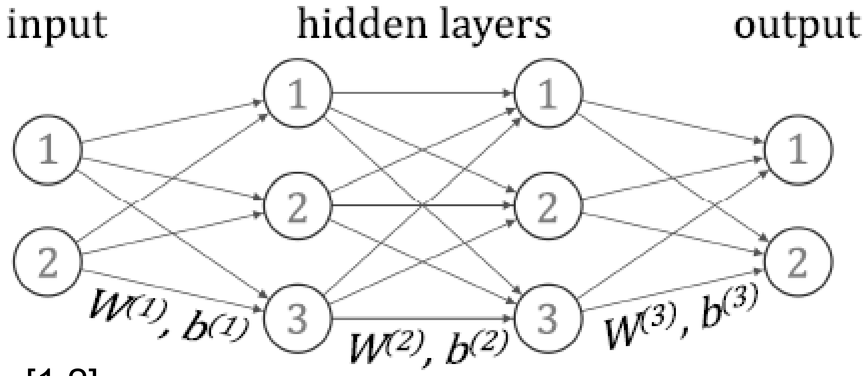
- Fully-connected networks:

  matrix multiplication between layers / bias vectors / nonlinear activation functions

  weights and biases are fixed by training

- Input data are processed from left to right (feedforward processing)

  network ready for new input after each output, results are not kept



[1,2]

Sequential data and time series

- Many forms of ordered data: time series, words, phrases, nucleobases in DNA…
- Time series important in physics: sensor readings (cf. data streams at CERN)

  everyday life: regularly updated weather forecasts

  financial data streams

  audio + video streams

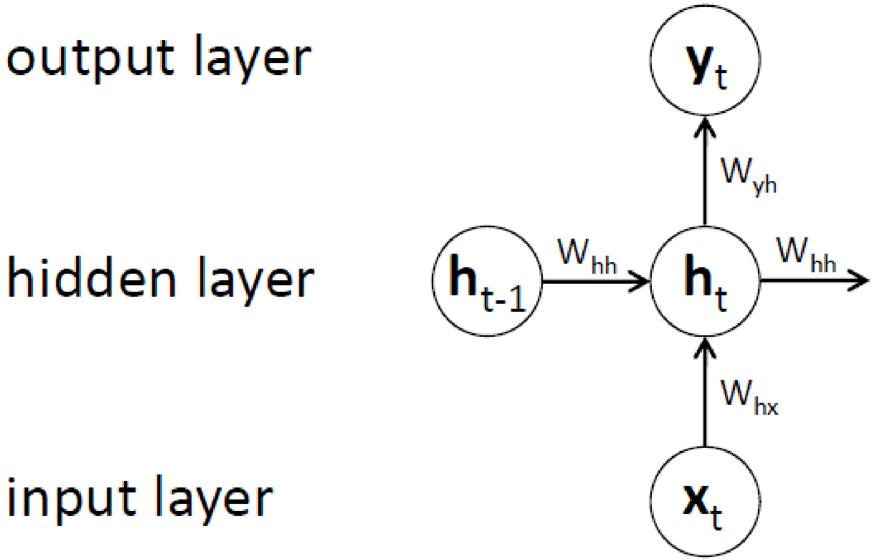
  voice assistants

  $\rightarrow$ time series are paradigm for physics
- Typical feature: variable input (= length of series not fixed, can vary during use)

No fixed input vector - standard matrix multiplication not well suited

New network design required when dealing with sequences: recurrent neural network

Basic unit of a recurrent network (RNN)



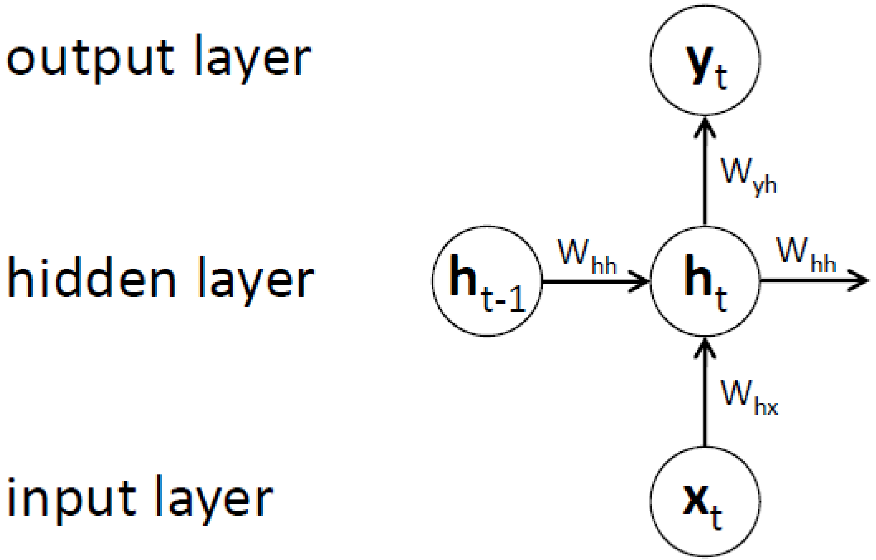
[2]

Output (e.g., forecast of the temperature at time step t+1)

Internal memory (to take into account previous input)

Input (e.g., reading of a temperature sensor at time step t)

Basic unit of a recurrent network (RNN)



Direction of data processing from input to output

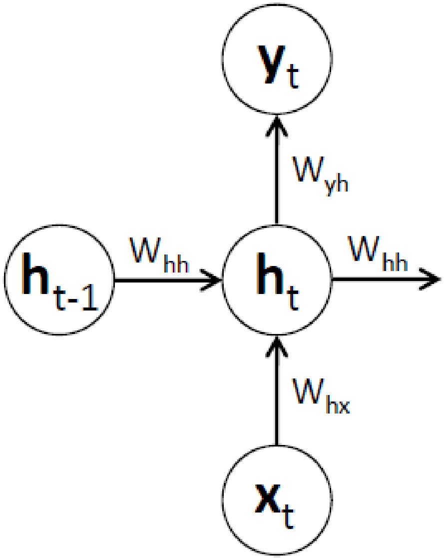
Timeline
(processing of previous input kept in internal memory)

Basic unit of a recurrent network (RNN)

output layer

hidden layer

input layer



Formal calculation:

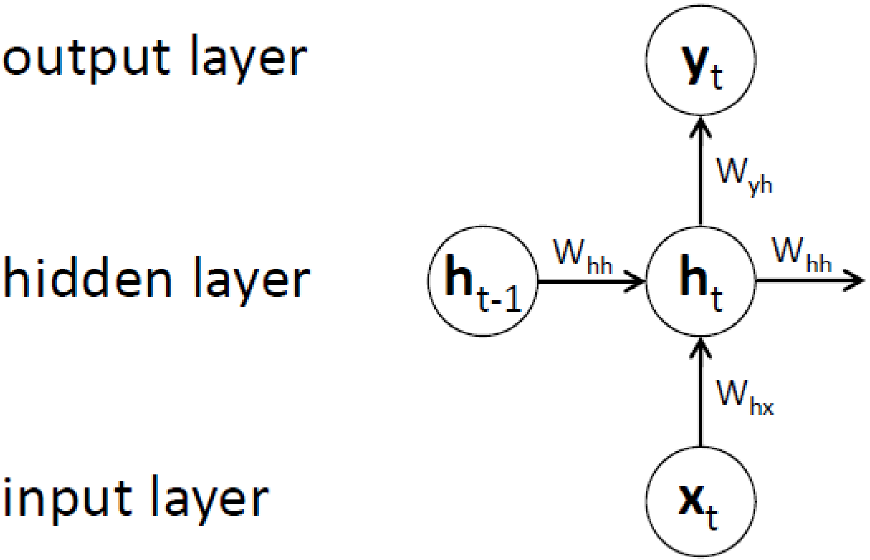
$$\vec{y}_t = g(\vec{h}_t)$$

$$\vec{h}_t = f(\vec{x}_t, \vec{h}_{t-1})$$

- Functions *f, g,* remain the same for the entire series
- Internal state at time t depends on *all* previous inputs:

$$\begin{aligned}\vec{h}_t &= f(\vec{x}_t, \vec{h}_{t-1}) \\ &= f(\vec{x}_t, f(\vec{x}_{t-1}, \vec{h}_{t-2})) \\ &= u(\vec{x}_t, \vec{x}_{t-1}, \vec{x}_{t-2}, \ldots)\end{aligned}$$

Basic unit of a recurrent network (RNN)

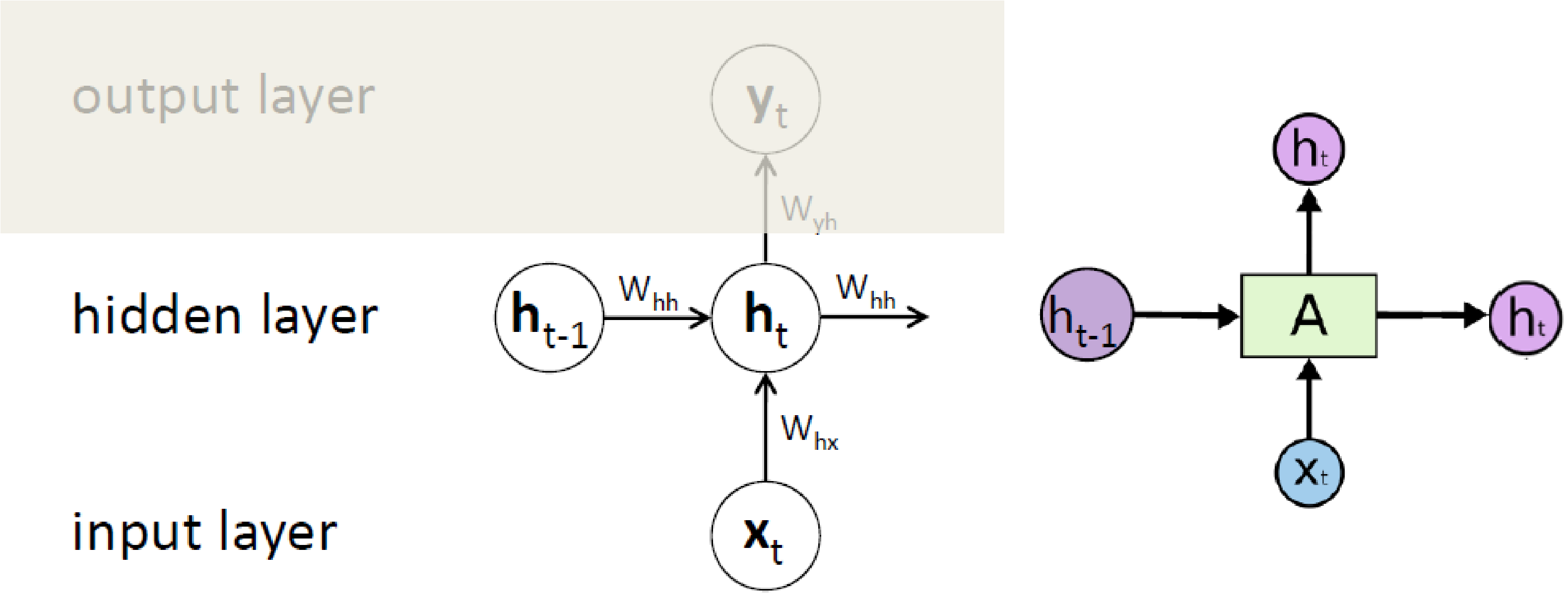


Example:

$$\vec{y}_t = \sigma(\mathbf{W}_{\mathrm{yh}}\,\vec{h}_t + \vec{b}_y)$$

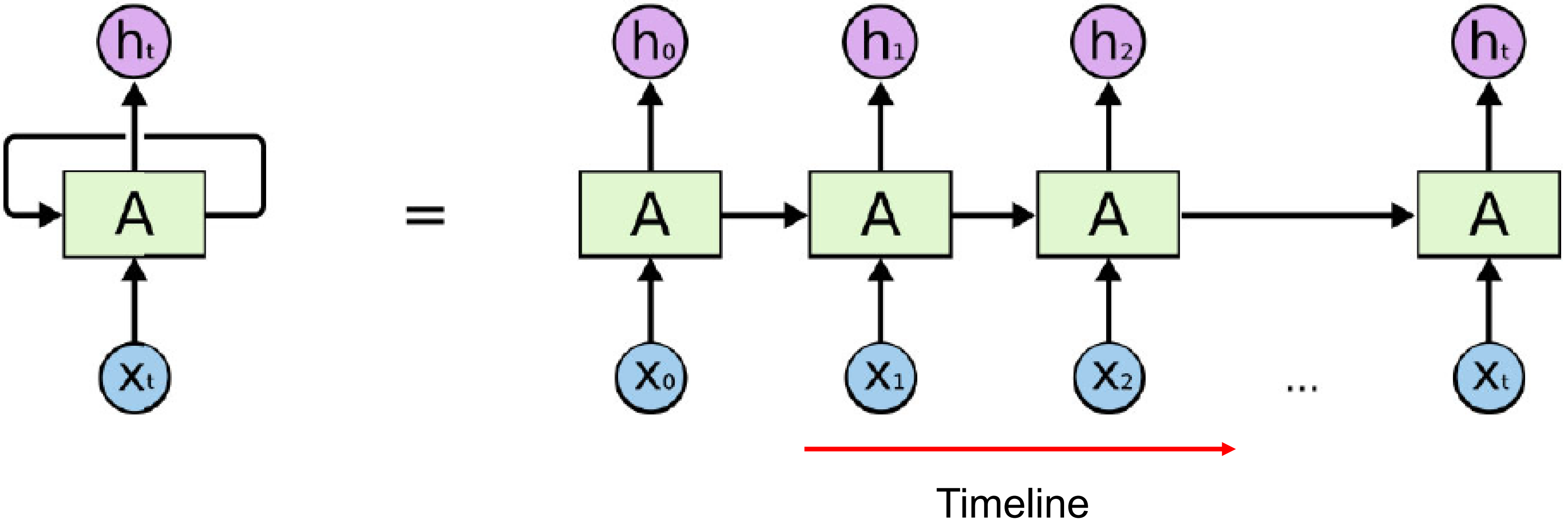
$$\vec{h}_t = \tanh\left(\mathbf{W}_{\mathrm{hx}}\,\vec{x}_t + \mathbf{W}_{\mathrm{hh}}\,\vec{h}_{t-1} + \vec{b}_{\mathrm{h}}\right)$$

Core building block of a RNN



[2,3]

Unrolling of a RNN



[3]

- Unrolled RNN with similar appearance as conventional feedforward network
- Training through backpropagation through time (BPTT) = usual backpropagation process for weights and biases adapted to RNN

Problem: short-term memory good, but long-term memory fails

Long short-term memory (LSTM)

- Introduced in 1997 by Hochreiter und Schmidhuber
- Eliminates sensitivity to gap length between important events of a series
  $\rightarrow$ training becomes possible for much more useful sequence lengths
- Game changer for RNNs: applicable in real-world situations
- Until today “gold standard“ for recurrent networks

Key features:
- Additional internal memory called cell state
- Gates that control the cell state actively
- Closely controlled update procedure of the internal memories

Gate overview:
- *Forget gate (= keep gate):* decides whether to forget or keep elements stored in the cell
- *Input gate:* decides whether a new value flows into the cell
- *Output gate:* controls whether the updated cell value contributes to the hidden state

Long short-term memory (LSTM)

Layer calculations and intermediate vectors:

- Direct update of the hidden state $\vec{h}_t$ replaced by a complex interaction of four intermediate vectors at each time step *t* (for simplicity here in 1D):

$$\begin{aligned}
\text{forget layer :} \quad & f_t = \sigma\left(W_f\, x_t + U_f\, h_{t-1} + b_f\right) \\
\text{input layer :} \quad & i_t = \sigma\left(W_i\, x_t + U_i\, h_{t-1} + b_i\right) \\
\text{output layer :} \quad & o_t = \sigma\left(W_o\, x_t + U_o\, h_{t-1} + b_o\right) \\
\text{cell input layer :} \quad & \tilde{C}_t = \tanh\left(W_c\, x_t + U_c\, h_{t-1} + b_c\right)
\end{aligned}$$

Updates from intermediate vectors:

- Cell state C_t
- Hidden state h_t

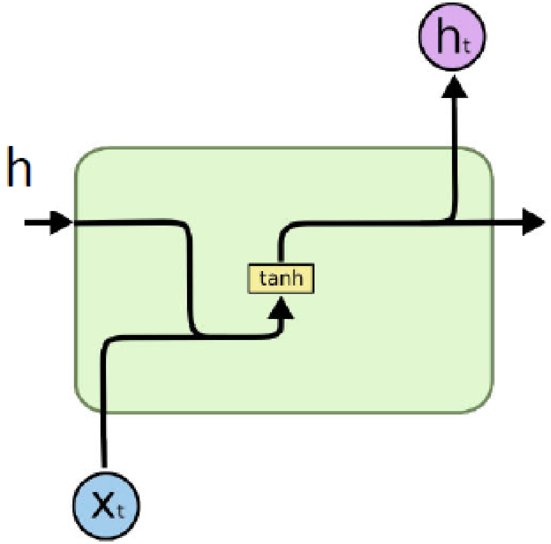
Example
Complete reset of cell state
$\rightarrow f_t = 0$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \qquad h_t = o_t \cdot \tanh\left(C_t\right)$$

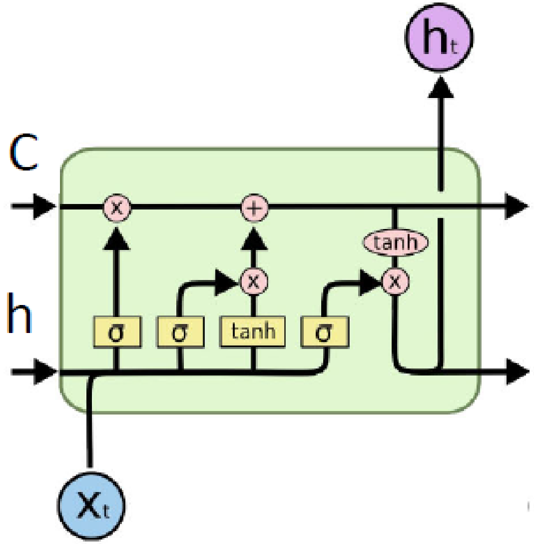
Long short-term memory (LSTM)

Graphical representation of the update process:

Standard RNN cell



LSTM cell

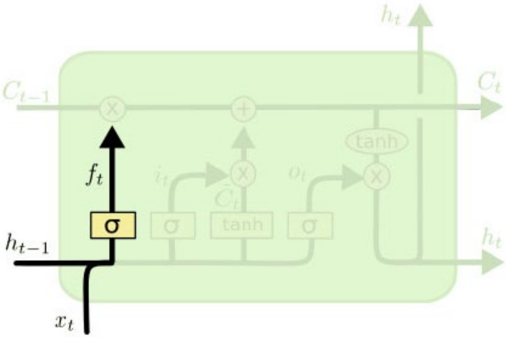


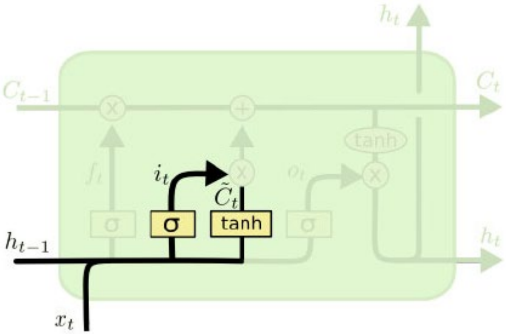
[3]

$$\vec{h}_t = \tanh\left(\mathbf{W}_{\mathrm{hx}}\, \vec{x}_t + \mathbf{W}_{\mathrm{hh}}\, \vec{h}_{t-1} + \vec{b}_{\mathrm{h}}\right)$$

Long short-term memory (LSTM)

LSTM walk-through: setup of intermediate variables





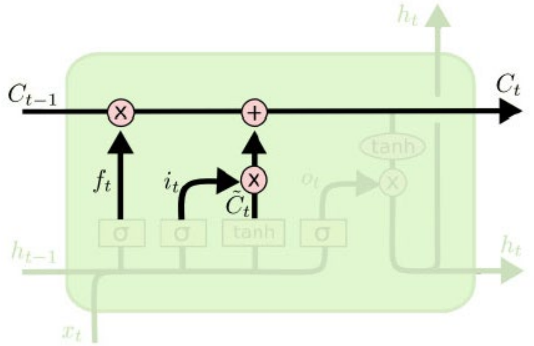
forget layer : $$f_t = \sigma\left(W_f\, x_t + U_f\, h_{t-1} + b_f\right)$$

input layer : $$i_t = \sigma\left(W_i\, x_t + U_i\, h_{t-1} + b_i\right)$$

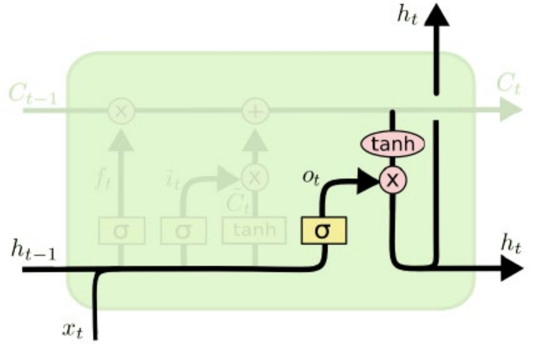
$$\tilde{C}_t = \tanh\left(W_c\, x_t + U_c\, h_{t-1} + b_c\right)$$

Long short-term memory (LSTM)

LSTM walk-through: cell and hidden state updates



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



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

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

Gated recurrent unit (GRU)

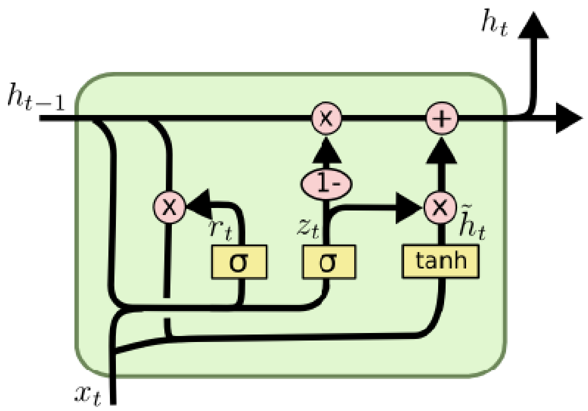
- Introduced in 2014 by Cho et al.
- Aim: simplification of the complicated LSTM update procedure
- Growing popularity in applications, needs less resources

Key features:
- No cell state
- Only two gates

Gate overview:
- *Reset gate:* decides whether to reset the hidden state
- *Update gate:* decides whether to update the hidden state

Gated recurrent unit (GRU)

GRU hidden state update



[3]

reset layer : $r_t = \sigma\left(W_r\, x_t + U_r\, h_{t-1} + b_r\right)$

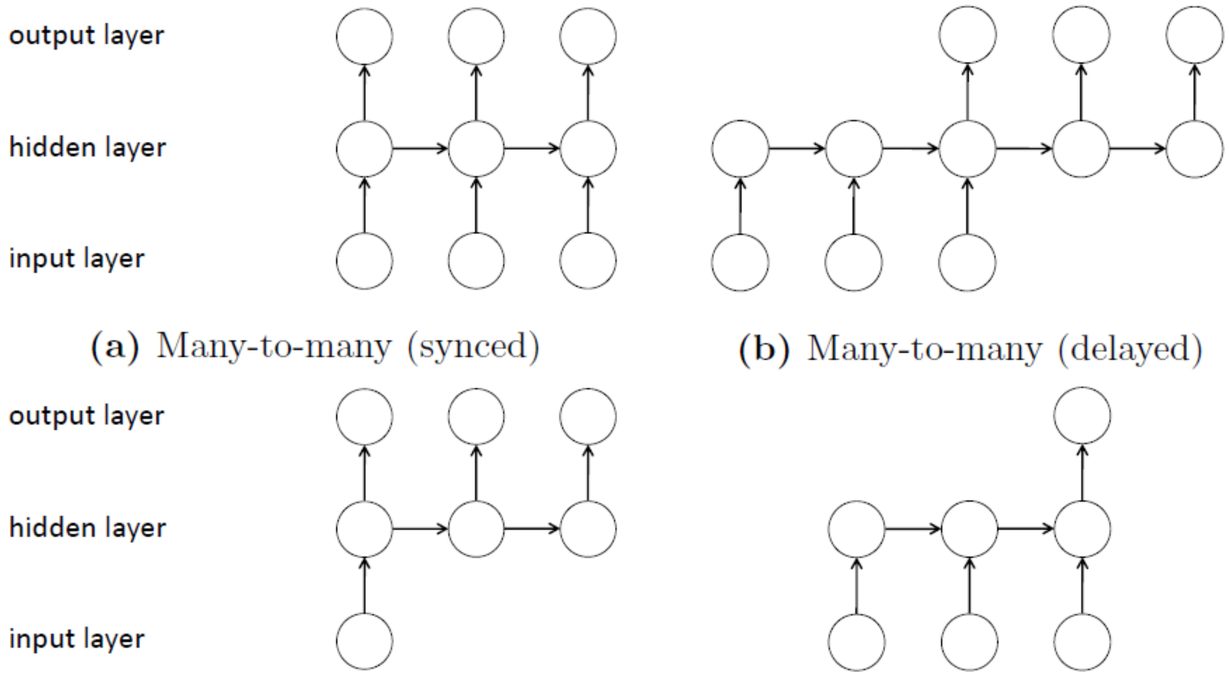
update layer : $z_t = \sigma\left(W_z\, x_t + U_z\, h_{t-1} + b_z\right)$

candidate layer : $\tilde{h}_t = \tanh\left(W_h\, x_t + U_h\left(r_t \cdot h_{t-1}\right) + b_h\right)$

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

Applications in current technology

Variants of sequence processing



(a) Many-to-many (synced)
(b) Many-to-many (delayed)
(c) One-to-many
(d) Many-to-one

Application examples:

(a) Temperature forecast
(b) Automatic translation
(c) Image captioning
(d) Text classification

[2]

Application example in physics

Nuclear Inst. and Methods in Physics Research, A 867 (2017) 40–50

Contents lists available at ScienceDirect

Nuclear Inst. and Methods in Physics Research, A

journal homepage: www.elsevier.com/locate/nima

ELSEVIER

Using LSTM recurrent neural networks for monitoring the LHC superconducting magnets

Maciej Wielgosz [a,*], Andrzej Skoczeń [b,c], Matej Mertik [d]

[a] Faculty of Computer Science, Electronics and Telecommunications, AGH University of Science and Technology, Kraków, Poland
[b] Faculty of Physics and Applied Computer Science, AGH University of Science and Technology, Kraków, Poland
[c] The European Organization for Nuclear Research — CERN, CH-1211 Geneva 23, Switzerland
[d] Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia

Wielgosz et al., Nucl. Inst. Meth. A, 867, 40 (2017)

A R T I C L E I N F O

Keywords:
LHC
Recurrent neural networks
LSTM
Deep learning
Modeling

A B S T R A C T

The superconducting LHC magnets are coupled with an electronic monitoring system which records and analyzes voltage time series reflecting their performance. A currently used system is based on a range of preprogrammed triggers which launches protection procedures when a misbehavior of the magnets is detected. All the procedures used in the protection equipment were designed and implemented according to known working scenarios of the system and are updated and monitored by human operators.

This paper proposes a novel approach to monitoring and fault protection of the Large Hadron Collider (LHC) superconducting magnets which employs state-of-the-art Deep Learning algorithms. Consequently, the authors of the paper decided to examine the performance of LSTM recurrent neural networks for modeling of voltage time series of the magnets. In order to address this challenging task different network architectures and hyper-parameters were used to achieve the best possible performance of the solution. The regression results were measured in terms of RMSE for different number of future steps and history length taken into account for the prediction. The best result of RMSE = 0.00104 was obtained for a network of 128 LSTM cells within the internal layer and 16 steps history buffer.

Application example in physics

Wielgosz et al., Nucl. Inst. Meth. A, 867, 40 (2017)

- Superconducting magnet quench sets free huge energies
- Quenches occur regularly for many reasons

  frequently: release of local mechanical stress, e.g. from assembly
- Electronic monitoring for quench protection exists
- Voltage-time series for each magnet logged, time resolution 400 ms

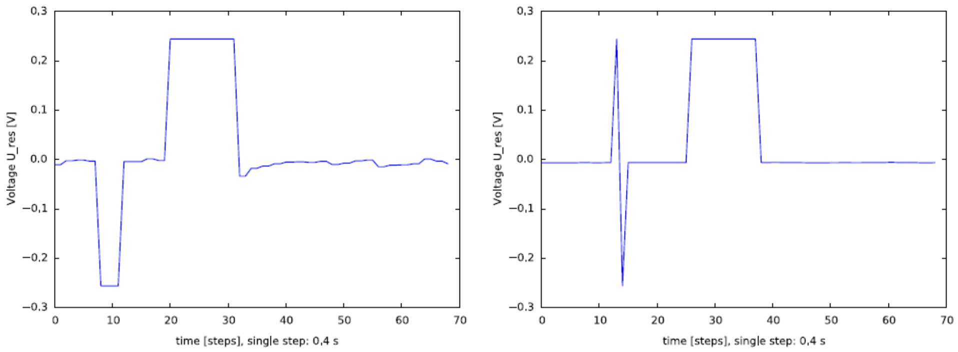


Fig. 8. The selected sample anomalies of 600 A magnets extracted from the LS database.

Quench unfolds gradually in time:
→ precursors herald instability

→ RNN training opportunity

Application example in physics

Wielgosz et al., Nucl. Inst. Meth. A, 867, 40 (2017)

Available data sets:

- 600 A magnet: 425 quench events between 2008 and 2016
- time window of 24 hours before quench selected for training

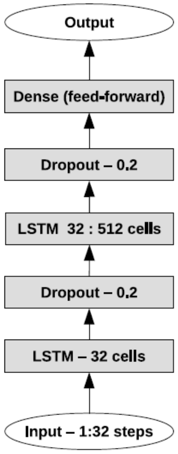


Fig. 10. The LSTM-based network used for the experiments.

Custom-design RNNs of various architectures trained

data split: 70% training, 30% testing

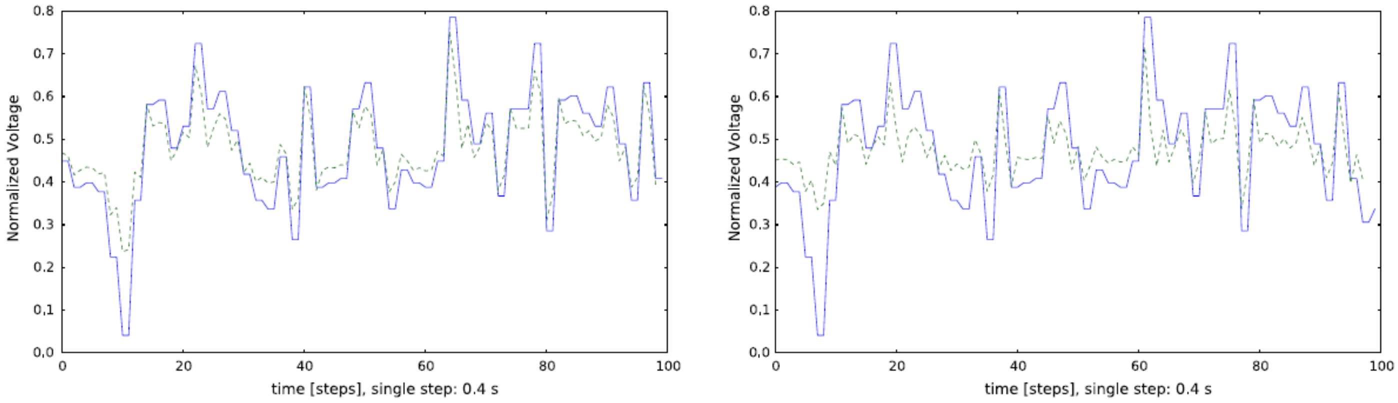
Table 5
The parameters of the LSTM network used to the experiments.

Parameter	Value
Number of layers	5
Number of epochs	6
Total number of the network parameters	21 025
Dropout	0.2
Max. number of steps ahead	32

Application example in physics

Wielgosz et al., Nucl. Inst. Meth. A, 867, 40 (2017)

Results:



(a) One step ahead.

(b) Two steps ahead.

Fig. 11. Two examples of prediction for one and two steps ahead in time. Predicted signal is plotted in a green broken line.

- Voltage-time series of magnets can be modeled by RNNs
- Prediction quality drops significantly with number of steps ahead
- Real time monitoring using RNNs?

Table 6
Performance of various approaches to LSTM hardware implementation (data from [45–47]).

Setup	Platform	Computation time [μs]
2 layers (128 cells), 32/16 bit	Xilinx Zynq 7020 (142 MHz), external memory — DDR3	~932
Compressed LSTM (20x), 1024 cells	Xilinx XCKU060 Kintex (200 MHz), external memory — DDR3	82.7
2 layers (30, 256 cells), 6-bit quantization	Xilinx Zynq XC7Z045 (100 MHz) 2.18 MB on-chip memory max, all in the internal memory	15.96

References

[1] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, Chapter 10, MIT Press, 2016
www. deeplearningbook.org

[2] M. Erdmann, J. Glombitza, G. Kasieczka, U. Klemradt, *Deep Learning for Physics Research,* World Scientific, 2021

[3] C. Olah, *Understanding LSTM networks*, 2015
colah.github.io/posts/2015-08-Understanding-LSTMs

RNNs in Keras

Basic usage, see https://keras.io/layers/recurrent/

```
from keras.layers import SimpleRNN, LSTM, GRU, Input

z0 = Input(shape=(100, 5))    # input: sequence of 100 steps, holding 5 features each
z = SimpleRNN(16, activation='tanh')(z0)
z = LSTM(16, activation='tanh', recurrent_activation='hard_sigmoid')(z0)
z = GRU(16, activation='tanh', recurrent_activation='hard_sigmoid')(z0)
```

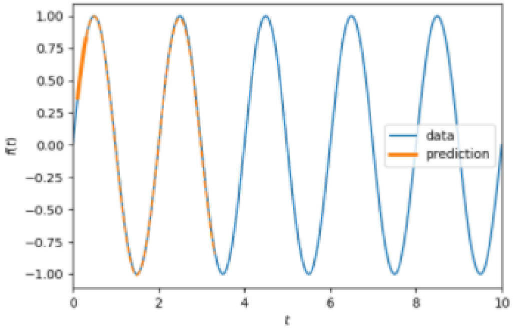
Usually CuDNN implementation is used (depending on your settings and hardware)

Common parameters with defaults:

- return_sequences=False - If True, return full sequences of states
- go_backwards=False - If True, RNN operates from back to front
- stateful=False - If True, reuse last states for each sample from previous batch
- unroll=False - Unroll graph: faster, but memory intensive (short sequences only!)

VISPA Examples:

Exercise 1:

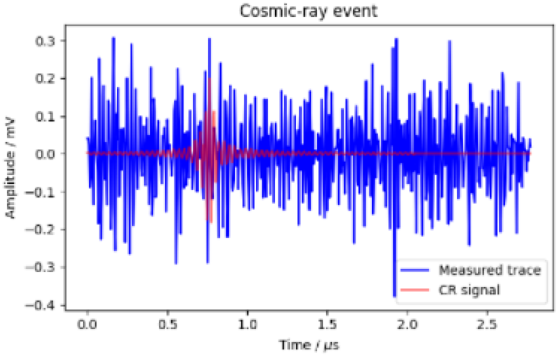


Introduction to (autoregressive) recurrent neural networks

In this example, we will introduce recurrent neural networks (RNNs) and implement them using keras. The aim is to train a network to predict the periodicity of a sine wave using the autoregressive characteristic of RNNs.

Open example

Exercise 2:



Cosmic-ray detection using recurrent networks

In this example, we will exploit RNNs in the context of sequence classification. Therefore, we use a simulation of cosmic-ray-induced air shower signals measured by radion antennas. The task is to design an RNN that can identify if the measured signal traces (shortened to 500 time steps) contains a signal or not.

Open example

Exercises walkthrough

http://deeplearningphysics.org/

Exercise 1:

Sinus forecasting

In this task, we will learn to implement RNNs in Keras. Therefore:

- Run the provided script and comment on the output.
- Vary the number and size of the LSTM layers and compare training time and stability of the performance.

In [13]:
```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
layers = keras.layers

print(keras.__version__)
```

```
2.4.0
```

Exercises walkthrough

http://deeplearningphysics.org/

Exercise 1:

Generation of data

We start by creating a signal trace: `t = 0-100` , `f = sin(pi * t)`

In [2]:
```
N = 10000
t = np.linspace(0, 100, N)  # time steps
f = np.sin(np.pi * t)  # signal
```

Split into semi-redundant sub-sequences of `length = window_size + 1` and perform shuffle

In [3]:
```
window_size = 20
n = N - window_size - 1  # number of possible splits
data = np.stack([f[i: i + window_size + 1] for i in range(n)])
```

Finally, split the data into features

In [4]:
```
X, y = np.split(data, [-1], axis=1)
X = X[..., np.newaxis]

print('Example:')
print('X =', X[0, :, 0])
print('y =', y[0, :])
```

Exercises walkthrough

http://deeplearningphysics.org/

Exercise 1:

Define and train RNN

In [5]:
```
z0 = layers.Input(shape=[None, 1])
z = layers.LSTM(16)(z0)
z = layers.Dense(1)(z)
model = keras.models.Model(inputs=z0, outputs=z)
print(model.summary())

model.compile(loss='mse', optimizer='adam')
```

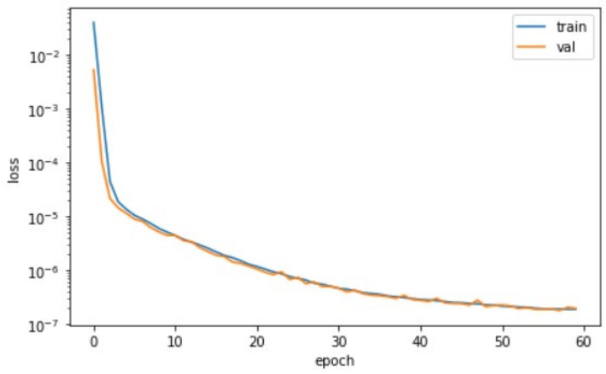
```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None, 1)]         0
_________________________________________________________________
lstm (LSTM)                  (None, 16)                1152
_________________________________________________________________
dense (Dense)                (None, 1)                 17
=================================================================
Total params: 1,169
Trainable params: 1,169
Non-trainable params: 0
_________________________________________________________________
None
```

Exercises walkthrough

http://deeplearningphysics.org/

Exercise 1:

```
In [6]: results = model.fit(X, y,
            epochs=60,
            batch_size=32,
            verbose=2,
            validation_split=0.1,
            callbacks=[
                keras.callbacks.ReduceLROnPlateau(factor=0.67, patience=3, verbose=1, min_lr=1E-5),
                keras.callbacks.EarlyStopping(patience=4, verbose=1)])
```



Exercises walkthrough

http://deeplearningphysics.org/

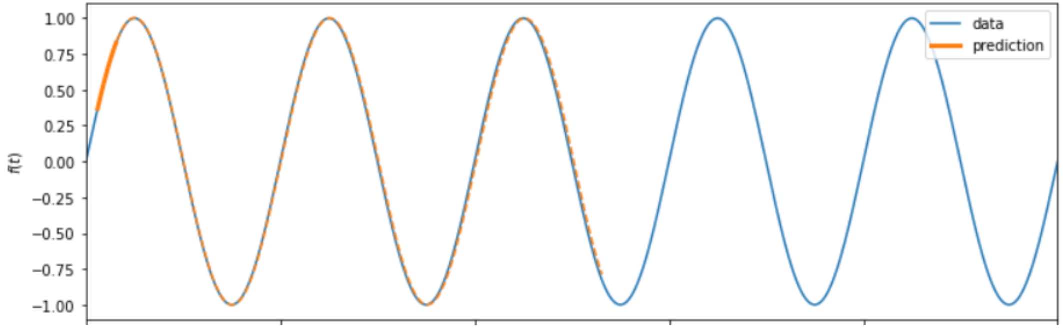
Exercise 1:

In [9]:
```
def plot_prediction(i0=0, k=500):
    """ Predict and plot the next k steps for an input starting at i0 """
    y0 = f[i0: i0 + window_size]  # starting window (input)
    y1 = predict_next_k(model, y0, k)  # predict next k steps

    t0 = t[i0: i0 + window_size]
    t1 = t[i0 + window_size: i0 + window_size + k]

    plt.figure(figsize=(12, 4))
    plt.plot(t, f, label='data')
    plt.plot(t0, y0, color='C1', lw=3, label='prediction')
    plt.plot(t1, y1, color='C1', ls='--')
    plt.xlim(0, 10)
    plt.legend()
    plt.xlabel('$t$')
    plt.ylabel('$f(t)$')
```

In [10]:
```
plot_prediction(12)
```



Exercises walkthrough

http://deeplearningphysics.org/

Based on https://arxiv.org/abs/1901.04079

Exercise 2:

Exercise 9.2

Large arrays of radio antennas can be used to measure cosmic rays by recording the electromagnetic radiation generated in the atmosphere. These radio signals are strongly contaminated by galactic noise as well as signals from human origin. Since these signals appear to be similar to the background, the discovery of cosmic-ray events can be challenging.

Identification of signals

In this exercise, we design an RNN to classify if the recorded radio signals contain a cosmic-ray event or only noise.

The signal-to-noise ratio (SNR) of a measured trace $S(t)$ is defined as follows:

$$\mathrm{SNR} = \frac{S^{\mathrm{signal}}(t)_{\mathrm{max}}}{\mathrm{RMS}[S(t)]},$$

where $S^{\mathrm{signal}}(t)_{\mathrm{max}}$ denotes the maximum amplitude of the (true) signal.

Typical cosmic-ray observatories enable a precise reconstruction at an SNR of roughly 3.

We choose a challenging setup in this task and try to identify cosmic-ray events in signal traces with an SNR of 2.
Training RNNs can be computationally demanding, thus, we recommend to use a GPU for this task.

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

layers = keras.layers
print("keras", keras.__version__)
```

```
keras 2.4.0
```

Exercises walkthrough

http://deeplearningphysics.org/

Exercise 2:

Plot example signal traces

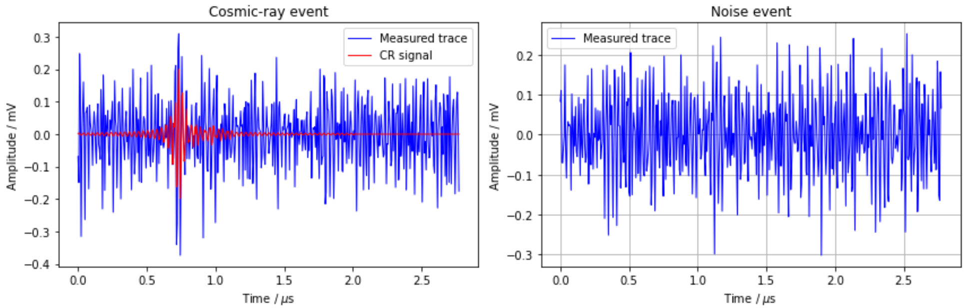
Left: signal trace containing a cosmic-ray event. The underlying cosmic-ray signal is shown in red, the backgrounds + signal is shown in blue. Right: background noise.

In [3]:
```
from matplotlib import pyplot as plt
fs = 180e6  # Sampling frequency of antenna setup 180 MHz
t = np.arange(500) / fs * 1e6
idx = np.random.randint(0, labels.sum()-1)
idx2 =  np.random.randint(0, n_train - labels.sum())

plt.figure(1, (12, 4))
plt.subplot(1, 2, 1)
plt.plot(t, np.real(f["traces"][labels.astype(bool)][idx]), linewidth = 1, color="b", label="Measured trace")
plt.plot(t, np.real(signals[labels.astype(bool)][idx]), linewidth = 1, color="r", label="CR signal")
plt.ylabel('Amplitude / mV')
plt.xlabel('Time / $\mu \mathrm{s}$')
plt.legend()
plt.title("Cosmic-ray event")
plt.subplot(1, 2, 2)

plt.plot(t, np.real(x_train[~y_train.astype(bool)][idx2]), linewidth = 1, color="b", label="Measured trace")
plt.ylabel('Amplitude / mV')
plt.xlabel('Time / $\mu \mathrm{s}$')
plt.legend()
plt.title("Noise event")

plt.grid(True)
plt.tight_layout()
```



Exercises walkthrough

http://deeplearningphysics.org/

Exercise 2:

Define RNN model

In the following, design a cosmic-ray model to identify cosmic-ray events using an RNN-based classifier.

In [ ]:
```
model = keras.models.Sequential()
model.add(...)

model.summary()
```

Pre-processing of data and RNN training

In [ ]:
```
sigma = x_train.std()
x_train /= sigma
x_test /= sigma
```

In [ ]:
```
model.compile(...)


results = model.fit(x_train[...,np.newaxis], y_train, ...)
```

In [ ]:
```
model.evaluate(x_test[...,np.newaxis], y_test)
```