Recurrent neural networks

Deep learning - BMDC 2025-2026

Gheorghe Cosmin Silaghi

Universitatea Babeș-Bolyai

October 20, 2025

Time series

Definition

- any data obtained via measurements at regular intervals, like the daily price of a stock, the hour electricity consumption of an entity, weekly sales of a store etc.
- timeseries involves understanding of the *dynamics* of a system: its periodic cycles, how it trends over time, its regular regime, its sudden spikes etc.

Common tasks

- **forecasting**: predicting what happens next in the series. The most common task
- **anomaly detection**: detect anything unusual happening with a continuous data stream.
- **classification**: assign one or more category labels to a timeseries. E.g. given the activity of a visitor on a webiste, classify whether the visitor is human or bot
- **event detection**: identify the occurrence of a specific, (eventually expected event), within a continuous data stream

Temperature forecasting example

- problem: predict the temperature 24 hours in advance, given a timeseries of hourly measurements of quantities, such as atmospheric pressure and humidity, recorded in the recent past by a set of sensors.
- dataset: Jena dataset, 14 different quantities, recorded every 10 minutes over several years

Items to consider, regarding this problem

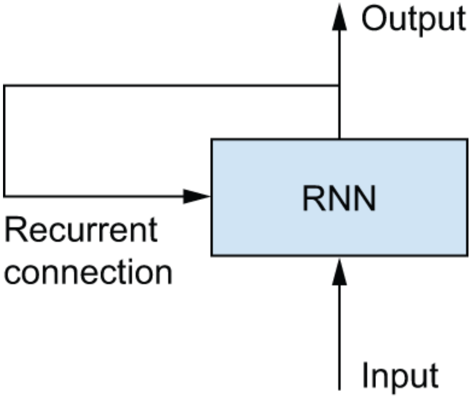
- how to prepare the training, validation and test sets: validation and test sets should be after the train set (in time)
- usage of *timeseries_dataset_from_array* utility
- usage of shuffle parameter: examples in the datasets should be suffled, before passing them to the ML model
- construction of the baseline: predict the same temperature value for 24 hours in advance
- usage of MAE as the assessment metric and MSE as the loss
- standard classifiers constructed with Dense and Conv1D blocks could not outperform the baseline

Recurrent neural networks

- dense connected NNs remove the notion of time from the input data - they have little chance to work on such sequenced data
- CNNs treat each segment of the data in the same way like the DNNs
- RNNs are designed exactly to look at the data as a sequence: each piece of data is processed at a moment, keeping in memory what we have seen up to now
- biological intelligence processes information incrementally, while maintaining an internal model of what it's processing, built on the past information and constantly updated as new information comes in

Recurrent neural networks

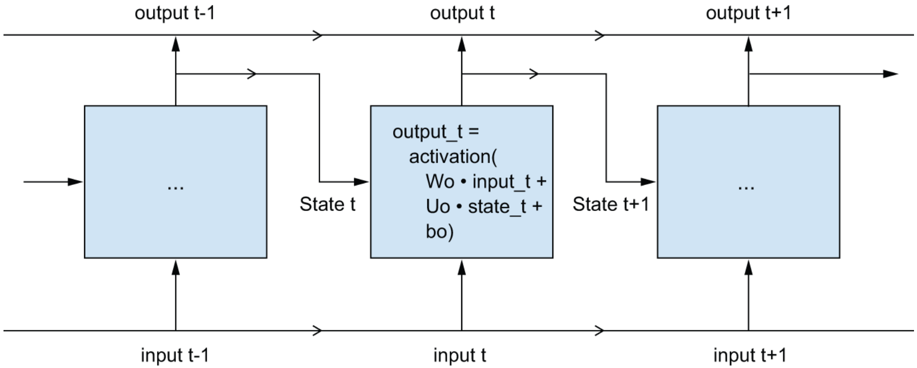
- RNN processes sequences by iterating through the sequence elements and maintaining a state containing information relative to what is has seen so far
- RNN is a type of network that has an internal loop.
- the state of the RNN is reset between processing two different independent sequences



How RNNs are functioning

- it takes an input a tensor of size **(timesteps, input_features)**.
- loops over timesteps.
  - at each timestep it considers its current state at **t** and the input at **t** - both of shape (input_features) and combines them to obtain the output at **t**
  - this output is considered the state at the next time **t+1**
- at the first timestep, the output is not defined - there is no current state
- the transformation over the input and the state are parametrized by two matrices: W and U and a bias vector, and is similar with the one operated by a densely connected layer in a feed-forward network
- $output\_t = tanh(input\_t \times W + state\_t \times U + b)$

The RNN unrolled



RNN implementation in Keras

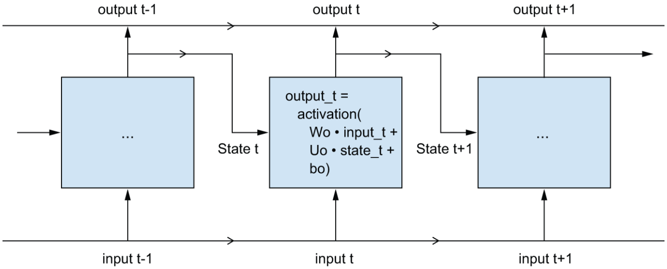
- RNNs could process batches of data, therefore the input is of shape `(batch_size, timesteps, input_features)`.
- timestep entry to the RNN could be set to `None`. in this case, model.summary() would not be available
- the network could return (*return_sequences* argument)
    - only the last output for each input sequence: a rank-2 tensor of shape `(batch_size, output_features)`
    - or could produce a rank-3 vector with the full successive outputs at each timestep: shape `(batch_size, timesteps, output_features)`
- stacking RNN layers one after the other increases the representational power of the network - `return_sequences` should be set `True`

SimpleRNN

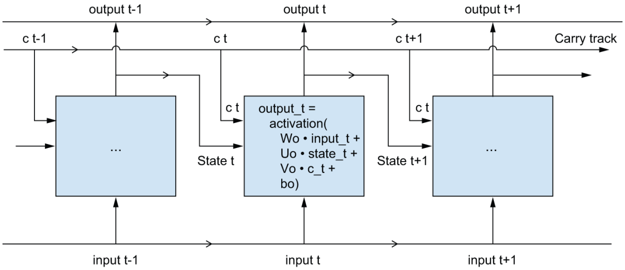
- is to simplistic to be of real use.
- theoretically, it should be able to retain at time t information about the inputs seen before
- but such long-term dependencies are impossible to learn because of vanishing gradient problem.

Long Short-Term Memory - LSTM

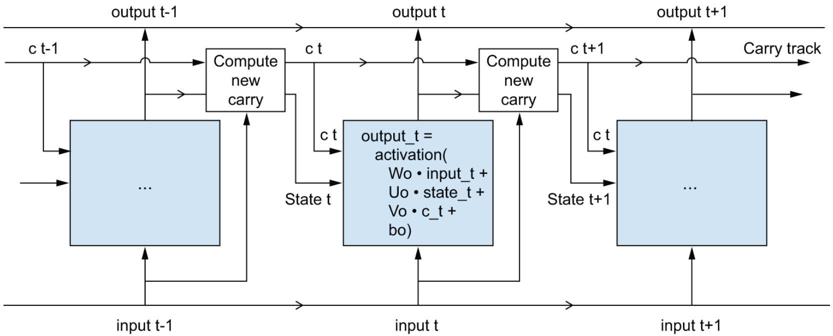
- it adds a way to carry information across many timesteps
- imagine a conveyor belt running parallel to the sequence you are processing
- information from the sequence jumps on the conveyor belt at any point, be transported to a latter time, and jump off, intact, when you need it
- please see the LSTM as a way to allow past information to be re-injected at later time in the processing, fighting the vanishing gradients problem



- add the *carry track* c_t to the SimpleRNN



Computation of the carry track



$$output_t = activation(state_t \times Uo + input_t \times Wo + C_t \times Vo + bo)$$
$$i_t = activation(state_t \times Ui + input_t \times Wi + bi)$$
$$f_t = activation(state_t \times Uf + input_t \times Wf + bf)$$
$$k_t = activation(state_t \times Uk + input_t \times Wk + bk)$$
$$c_{t+1} = i_t * k_t + C_t * f_t$$

- let's interpret the operations making up a RNN cell as a *set of constraints* on the search into the hypothesis space

Fighting overfitting - dropout

- either a small LSTM starts overfit very early - use dropout to fight against
- same dropout mask (the same pattern of dropped units) should be applied at every timestep, instead of a dropout mask that varies randomly with the timestep to timestep - would allow the network to properly propagate its learning error through time
- to regularize the representations formed by the recurrent gates of layers (such as GRU or LSTM), a temporary dropout mask should be applied to the inner recurrent activations of the layer
- each recurrent layer has two dropout parameters:
  - dropout: specifying the dropout rate for input units of the layer
  - recurrent_dropout: the dropout rate of the recurrent units
- because of dropout, we can use the LSTM layer with twice more units, which should be hopefully more expressive

RNN runtime performance

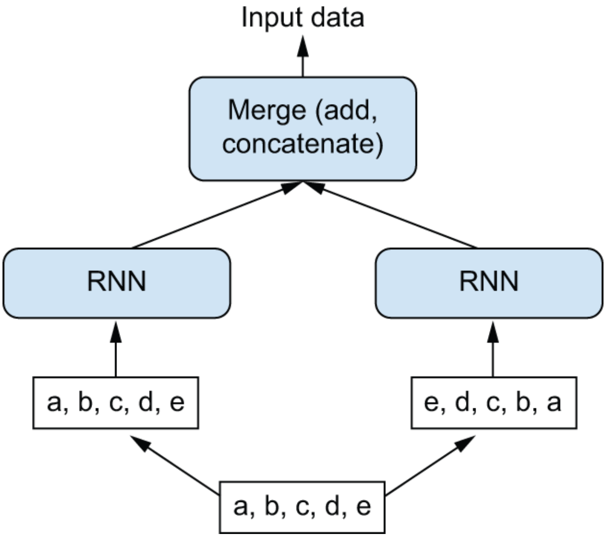
- RNN models with few parameters tend to be more faster on a multicore CPU than a GPU, because they involve small matrix multiplications and the chain of multiplications could not be well paralelized because of the for loop
- but larger RNNs benefit from the GPUs - they are using the cuDNN kernel highly optimized
- cuDNN kernel is very inflexible: if using operations that are not supported, they suffer a dramatic slowdown.
- recurrent dropout is not supported in cuDNN kernel: the runtime fall back to the regular Tensorflow implementation (about 5 times slower on GPU)
- to speed up the RNN layer you can use **unroll=True**. This will remove the for loops and in-line its content N times. Tensorflow will better optimize the computation graph.
- this option will also increase the memory consumption - therefore is only viable for small sequences (around 100 steps or fewer) and if the number of time steps is known in advance

Stacking recurrent layers

- increase the network capacity by stacking RNN layers
- to stack RNN layers on top of each other, all intermediate layers should return their full sequence of outputs (return_sequences=True)
- stacking 2 GRU layers with 32 units would increase the performance by 8%, to MAE=2.39

Using bidirectional RNNs

- A bidirectional RNN is frequently used in NLP
- RNNs are order dependent: they process the timesteps of their input sequences in order; shuffling or reversing the timesteps can completely change the representation extracted by the RNN.
- a bidirectional RNN exploits the order sensitivity of RNNs: it consists of two regular RNNs, each of which processes the input in one direction (chronological and anti-chronological) and then merging the representations
- they can catch patterns that may be overlooked by a unidirectional RNN.
- in our case, the anti-chronological LSTM will strongly underperform the baseline. Why?
- but for many other problems this is not true: the importance of a word in understanding a sentence is not strongly dependent on its position on the sentence. Although order matter, *which order* is not crucial.
- about 2016, before transformers, bidirectional LSTMs were considered SoA on many NLP tasks


Input data
Merge (add, concatenate)
RNN
RNN
a, b, c, d, e
e, d, c, b, a
a, b, c, d, e