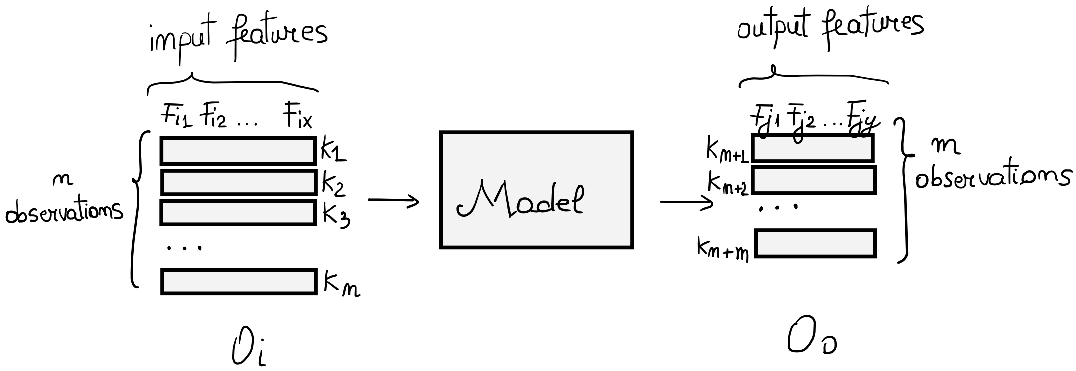
SensorML project

Preprocessing steps

When training NN models, some preprocessing steps that might be helpful include:

- Data standardization or normalization (to not treat differently the features that have values with a different order of magnitude or with higher/lower variances; it may also improve the model's stability)
- Creating the train & test (and maybe validation) datasets. For each type of dataset, the time series data should be split in pairs (O_i, O_o), where:
  - O_i represents a batch of consecutive observations given as input to make the prediction
  - O_o represents the batch observations outputted by the prediction



Hyperparameters

Examples of hyperparameters: number of epochs, learning rate, batch size, optimizer, number of hidden layers, etc.

The best model is impossible to guess … => solution: hyperparameter tuning

Examples of hyperparameter tuning strategies:

- Grid Search (check all combinations)
- Random Search (check random combinations)

Recurrent Neural Networks

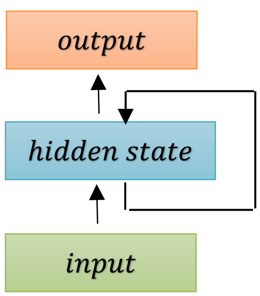
Why do we need them?

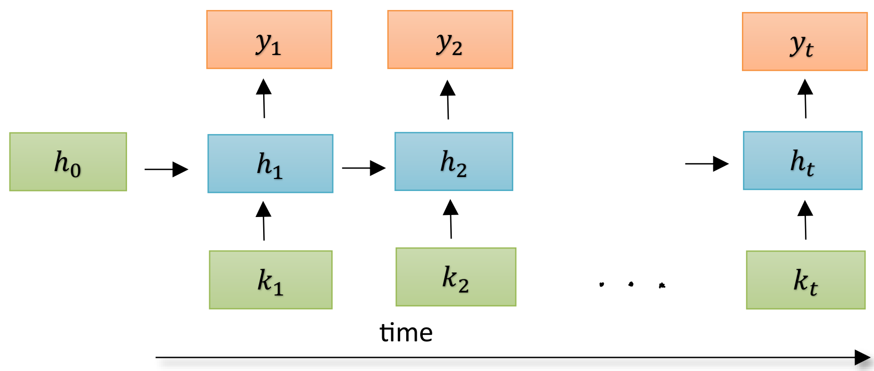
For solving problems that do not work with a fixed input and/or output size.

We could still use a traditional neural network for such problems (by considering an input layer with $n * x$ neurons, corresponding to the n observations given as input) but this is usually not feasible:

- How large should the input size be? If it is too small, there is not enough history to learn from; if it is too large => the weights matrix/matrices increase(s) in size => the network becomes very complex => training may take a very long time.
- Sometimes we inevitably have to deal with inputs of variables sizes (for e.g., when processing sentences with different lengths).

Architecture of a vanilla (one-layer) RNN





Training

Feed forward:

$$h_t = f_1(W_1 \cdot k_t + W_1' \cdot h_{t-1} + b_1)$$

$$y_t = f_2(W_2 \cdot h_t + b_2)$$

where:

$k_t = input/instance\ give\ to\ network\ at\ timestamp\ t$

$h_t = hidden\ state\ from\ timestamp\ t$

$y_t = output\ from\ timestamp\ t$

$f_1, f_2 = activation\ functions\ for\ the\ hidden\ and\ the\ output\ layer$

$W_1, W_1', W_2, b_1, b_2 = \ matrices\ of\ weights\ and\ vectors\ of\ biases$

Obs1: The matrices of weights and vectors of biases are shared across timestamps.

Obs2: There are multiple input-output scenarios (e.g., some of the outputs of the network can be ignored –> take into account only y_t for computing the error and making the backpropagation)

Obs3: $k_t, h_t, y_t, b_1, b_2\ are\ vectors; W_1, W_1', W_2\ are\ matrices$

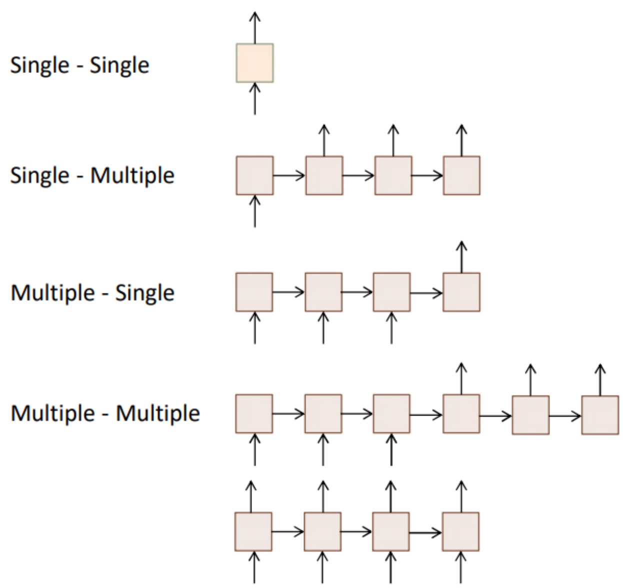
Backpropagation Through Time (BTT)

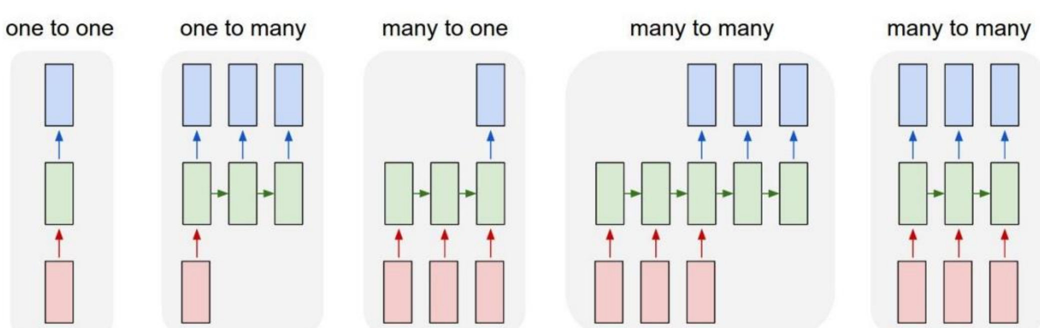
Idea:

- The unfolded network is considered as a big feed-forward network, which takes the entire sequence $k_1 k_2 \ldots k_t$ as an input.
- The gradients are computed as in a normal backpropagation.
- The weights and biases updates are computed for each "copy", then aggregated (e.g., averaged). These final updates are applied to the weight matrices and to the bias vectors.

Input-output scenarios

Some of the outputs of the RNN can be ignored. Also, the network can receive one or multiple inputs.





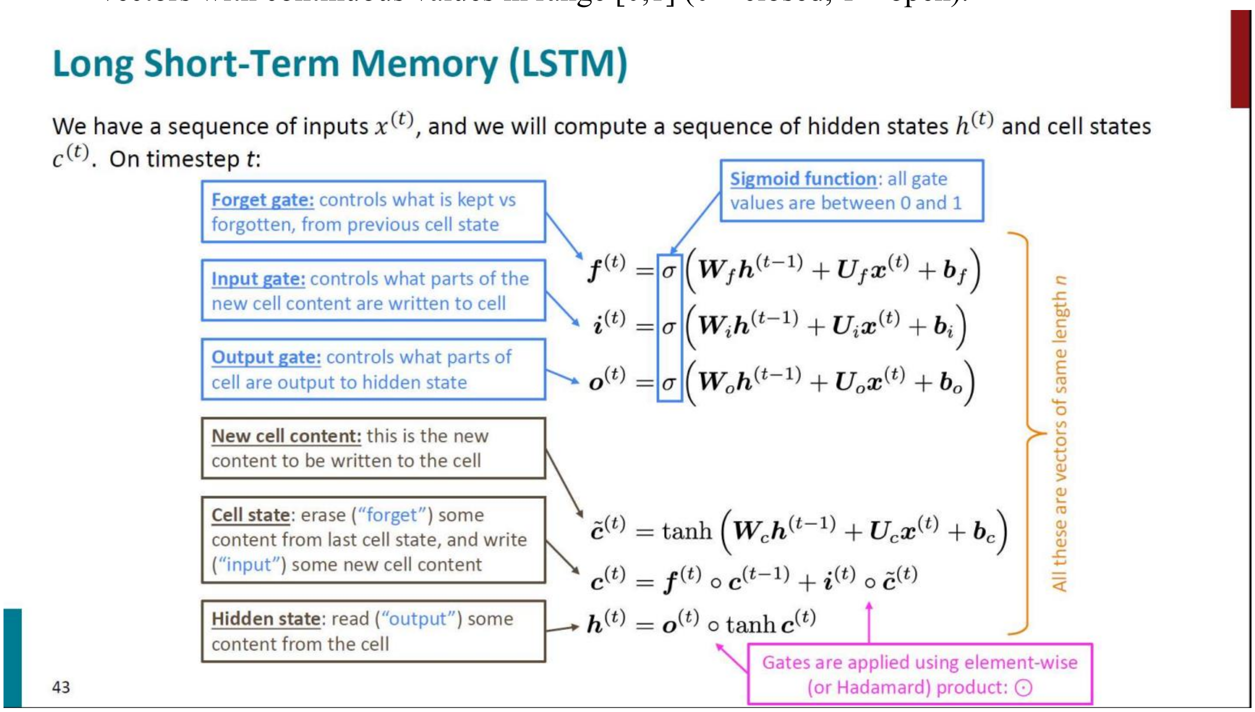
Drawbacks of RNNs

- Backpropagation through time can be slow
- *Vanishing/exploding gradient* (Long term dependencies are hard to capture)
  - ➔ common solution for exploding gradient: *gradient clipping*
  - ➔ common solution for vanishing gradient: using other architectures (LSTM, GRU)

LSTM & GRU

LSTM

- LSTM aims to (partially) solve the vanishing gradient problem, such that the network can learn better long-term dependencies => **a new cell state c_t is introduced to store long-term information**
- Information can be **read, erased or written** from/to the cell state
- To select what is read/erased/written, **three corresponding gates** are used. The gates are vectors with continuous values in range [0,1] (0 – closed, 1 – open).



GRU

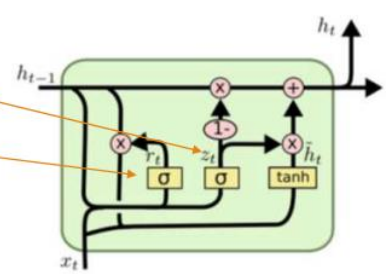
A simplification over LSTMs
- Eliminate the cell state (memory cell)
- Replace forget (f) and input (i) gates with an update gate (z)
- Introduce a reset gate (r) that modifies $h^{(t-1)}$

$z^{(t)} = \sigma(W_z h^{(t-1)} + U_z x^{(t)} + b_z)$ update gate

$r^{(t)} = \sigma(W_r h^{(t-1)} + U_r x^{(t)} + b_r)$ reset gate

$$\tilde{h}^{(t)} = tanh(W(r_t \circ h^{(t-1)}) + Ux^{(t)} + b)$$

$$h^{(t)} = (1 - z_t) \circ h^{(t-1)} + z_t \circ \tilde{h}^{(t)}$$



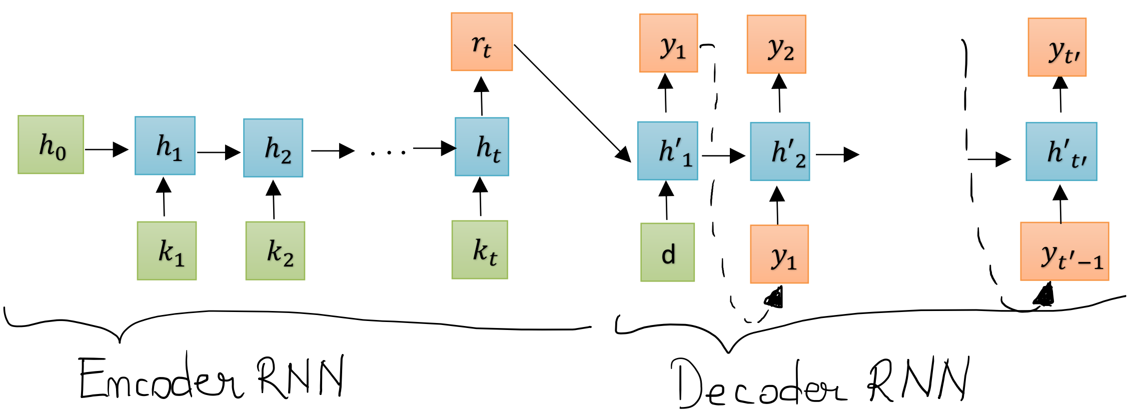
GRUs and LSTMs have comparable performance.

The Seq2seq model

- RNNs (including variations such as LSTM and GRU) usually struggle with capturing long-term dependencies
- Seq2Seq ("sequence to sequence") models are able to handle inputs and outputs of variable length and capture complex dependencies between input and output sequences
- The Seq2Seq model contains two components:
  - **An encoder network** (usually a RNN) used to build a **representation** of the given inputs. In the example below
  - A decoder network (usually a RNN) takes the representation and generates outputs, one at a time

Classical Seq2seq model (using two vanilla RNNs)

- r_t is the representation outputted by the encoder, storing all information about the given inputs
- r_t acts as initial hidden state (h'_0) for the decoder
- The decoder receives d as an initial dummy input (that indicates the start of the generation)
- The decoder generates an output, one at a time, using the previous hidden state and the last generated output
- Disadvantage of the classical Seq2seq model: the bottle neck problem; solution: using an attention mechanism



Other resources

1. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#overview
2. https://profs.info.uaic.ro/~nlp/documente/C7.%20RNN-1.pdf
3. https://profs.info.uaic.ro/~nlp/documente/C9.%20RNN-2.pdf
4. http://cs231n.stanford.edu/slides/2023/lecture_8.pdf
5. LSTM Training
6. Seq2Seq model