Advanced Machine Learning (Semester 1 2023)

Recurrent Neural Networks

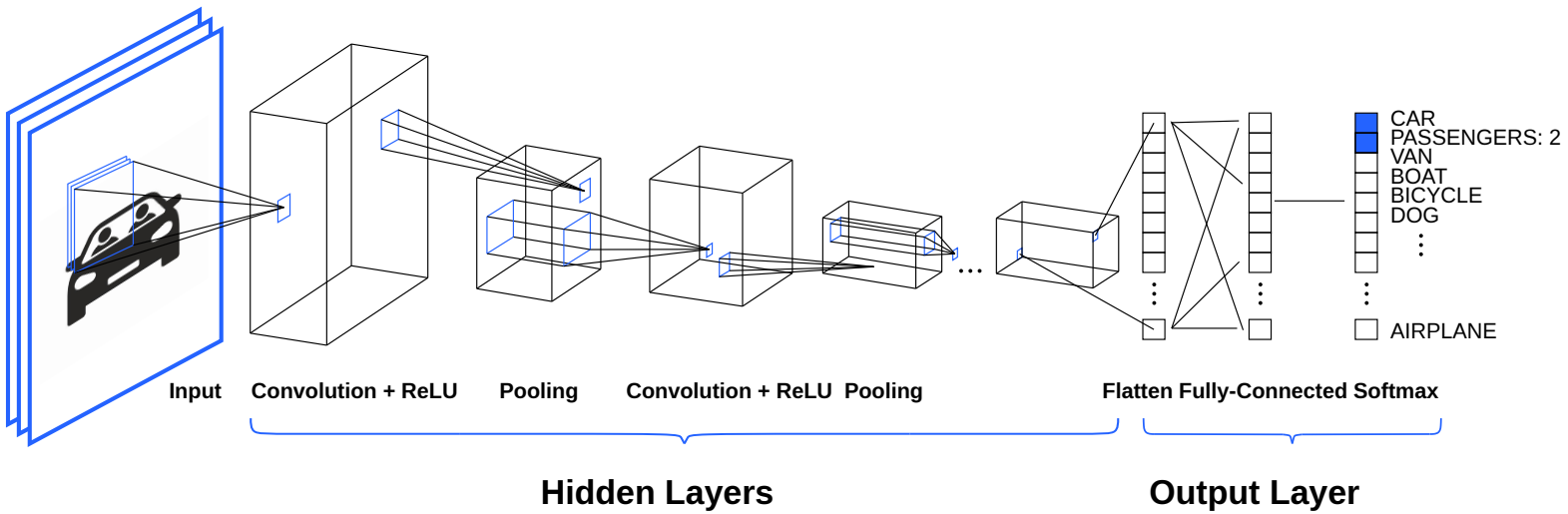
Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

May 29, 2023

Recap

A Convolutional Neural Network (CNN) consists of alternating **Convolutional layers** and **Pooling layers** that extract features. Finally, their output is flattened into a one-dimensional feature layer that is passed into a Multi-Layer Perceptron (fully-connected layers) to obtain a prediction.

In this way they preserve the input image's spatial and temporal dependencies, mimicking how animal (human) vision works.

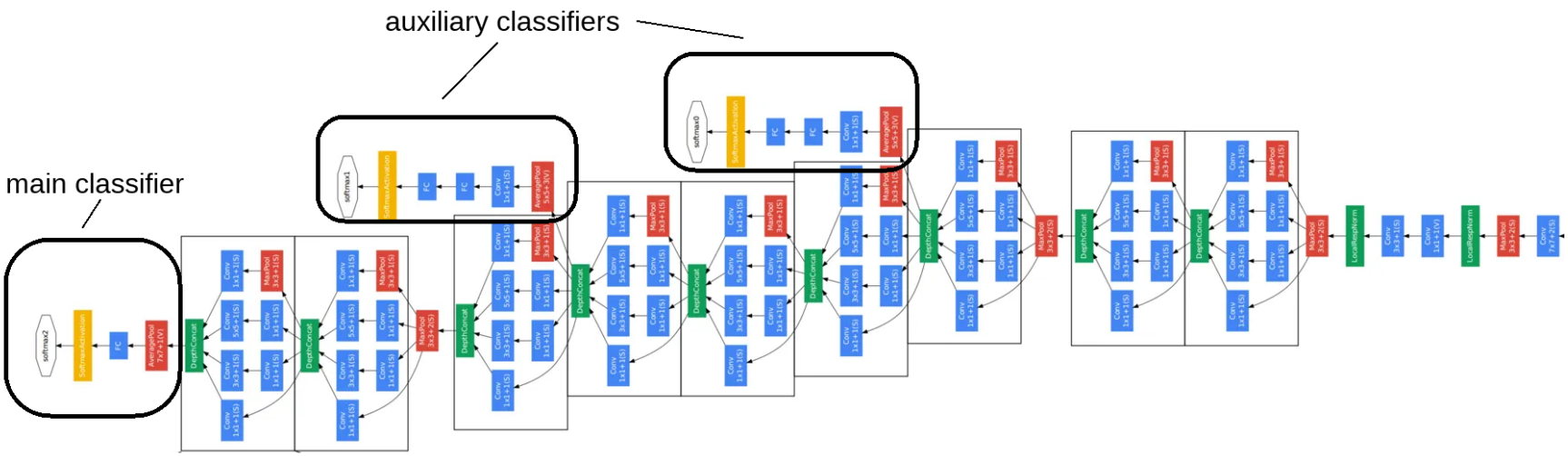


Recap

CNN architectures can show a **high degree of complexity**:

GoogLeNet, also called Inception-v1 (Szegedy et al. 2014) is built on the concept of an **inception module**: design a good local network topology and then **stack** these modules on top of each other (network within a network).

Auxiliary classifiers to prevent exploding/vanishing gradients start with a 5×5 average-pooling, followed by a convolution layer with 1×1 kernel size and two fully connected layers.



Motivation

different data types lead to different NN architectures:

classical NN
work on tabulated data

1	1325.328857	16.119175	16.119175	0.018178
2	1325.349731	16.116192	16.116192	0.018189
3	1325.370605	16.118525	16.118525	0.018218
4	1325.391357	16.119541	16.119541	0.018228
5	1325.412231	16.118353	16.118353	0.018226
6	1325.433105	16.118797	16.118797	0.018227
7	1325.453857	16.120758	16.120758	0.018256
8	1325.474731	16.121220	16.121220	0.018261
9	1325.495605	16.121134	16.121134	0.018249
10	1325.516357	16.120495	16.120495	0.018251
11	1325.537231	16.123848	16.123848	0.018310
12	1325.558105	16.123949	16.123949	0.018334
13	1325.578857	16.125397	16.125397	0.018335
14	1325.599731	16.124432	16.124432	0.018331
15	1325.620605	16.123493	16.123493	0.018288
16	1325.641479	16.126211	16.126211	0.018362
17	1325.662231	16.127548	16.127548	0.018384
18	1325.683105	16.126381	16.126381	0.018342
19	1325.703979	16.126745	16.126745	0.018367
20	1325.724731	16.127941	16.127941	0.018383
21	1325.745605	16.129608	16.129608	0.018395
22	1325.766479	16.129673	16.129673	0.018406
23	1325.787231	16.162720	16.162720	0.018971
24	1325.808105	16.221766	16.221766	0.020035
25	1325.828979	16.240532	16.240532	0.020410
26	1325.849731	16.239178	16.239178	0.020395
27	1325.870605	16.229338	16.229338	0.020203

Motivation

different data types lead to different NN architectures:

classical NN
work on tabulated data

1	1325.328857	16.119175	16.119175	0.018178
2	1325.349731	16.116192	16.116192	0.018189
3	1325.370605	16.118525	16.118525	0.018218
4	1325.391357	16.119541	16.119541	0.018228
5	1325.412231	16.118353	16.118353	0.018226
6	1325.433105	16.118797	16.118797	0.018227
7	1325.453857	16.120758	16.120758	0.018256
8	1325.474731	16.121220	16.121220	0.018261
9	1325.495605	16.121134	16.121134	0.018249
10	1325.516357	16.120495	16.120495	0.018251
11	1325.537231	16.123848	16.123848	0.018310
12	1325.558105	16.123949	16.123949	0.018334
13	1325.578857	16.125397	16.125397	0.018335
14	1325.599731	16.124432	16.124432	0.018331
15	1325.620605	16.123493	16.123493	0.018288
16	1325.641479	16.126211	16.126211	0.018362
17	1325.662231	16.127548	16.127548	0.018384
18	1325.683105	16.126381	16.126381	0.018342
19	1325.703979	16.126745	16.126745	0.018367
20	1325.724731	16.127941	16.127941	0.018383
21	1325.745605	16.129608	16.129608	0.018395
22	1325.766479	16.129673	16.129673	0.018406
23	1325.787231	16.162720	16.162720	0.018971
24	1325.808105	16.221766	16.221766	0.020035
25	1325.828979	16.240532	16.240532	0.020410
26	1325.849731	16.239178	16.239178	0.020395
27	1325.870605	16.229338	16.229338	0.020203

CNN
work on images
by mimicking vision

Motivation

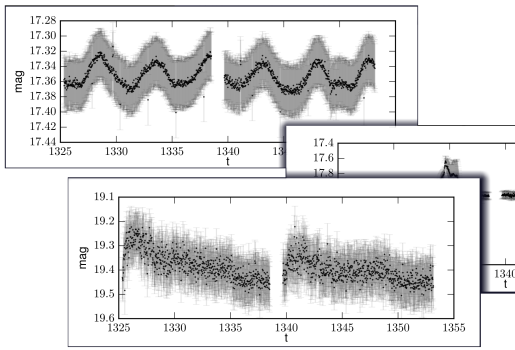
different data types lead to different NN architectures:

classical NN
work on tabulated data

1	1325.328857	16.119175	16.119175	0.018178
2	1325.349731	16.116192	16.116192	0.018189
3	1325.370605	16.118525	16.118525	0.018218
4	1325.391357	16.119541	16.119541	0.018228
5	1325.412231	16.118353	16.118353	0.018226
6	1325.433105	16.118797	16.118797	0.018227
7	1325.453857	16.120758	16.120758	0.018256
8	1325.474731	16.121220	16.121220	0.018261
9	1325.495605	16.121134	16.121134	0.018249
10	1325.516357	16.120495	16.120495	0.018251
11	1325.537231	16.123848	16.123848	0.018310
12	1325.558105	16.123949	16.123949	0.018334
13	1325.578857	16.125397	16.125397	0.018335
14	1325.599731	16.124432	16.124432	0.018331
15	1325.620605	16.123493	16.123493	0.018288
16	1325.641479	16.126211	16.126211	0.018362
17	1325.662231	16.127548	16.127548	0.018384
18	1325.683105	16.126381	16.126381	0.018342
19	1325.703979	16.126745	16.126745	0.018367
20	1325.724731	16.127941	16.127941	0.018383
21	1325.745605	16.129608	16.129608	0.018395
22	1325.766479	16.129673	16.129673	0.018406
23	1325.787231	16.162720	16.162720	0.018971
24	1325.808105	16.221766	16.221766	0.020035
25	1325.828979	16.240532	16.240532	0.020410
26	1325.849731	16.239178	16.239178	0.020395
27	1325.870605	16.229338	16.229338	0.020203

CNN
work on images
by mimicking vision

sequential
data



Motivation

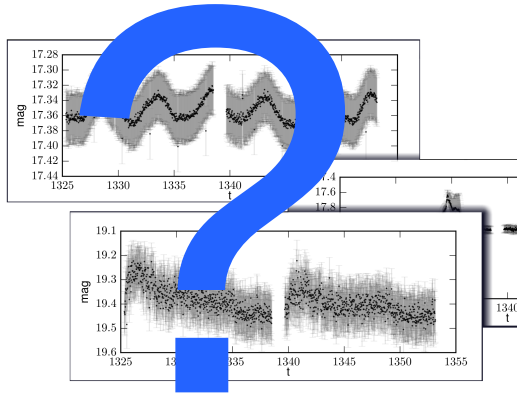
different data types lead to different NN architectures:

classical NN
work on tabulated data

1	1325.328857	16.119175	16.119175	0.018178
2	1325.349731	16.116192	16.116192	0.018189
3	1325.370605	16.118525	16.118525	0.018218
4	1325.391357	16.119541	16.119541	0.018228
5	1325.412231	16.118353	16.118353	0.018226
6	1325.433105	16.118797	16.118797	0.018227
7	1325.453857	16.120758	16.120758	0.018256
8	1325.474731	16.121220	16.121220	0.018261
9	1325.495605	16.121134	16.121134	0.018249
10	1325.516357	16.120495	16.120495	0.018251
11	1325.537231	16.123848	16.123848	0.018310
12	1325.558105	16.123949	16.123949	0.018334
13	1325.578857	16.125397	16.125397	0.018335
14	1325.599731	16.124432	16.124432	0.018331
15	1325.620605	16.123493	16.123493	0.018288
16	1325.641479	16.126211	16.126211	0.018362
17	1325.662231	16.127548	16.127548	0.018384
18	1325.683105	16.126381	16.126381	0.018342
19	1325.703979	16.126745	16.126745	0.018367
20	1325.724731	16.127941	16.127941	0.018383
21	1325.745605	16.129608	16.129608	0.018395
22	1325.766479	16.129673	16.129673	0.018406
23	1325.787231	16.162720	16.162720	0.018971
24	1325.808105	16.221766	16.221766	0.020035
25	1325.828979	16.240532	16.240532	0.020410
26	1325.849731	16.239178	16.239178	0.020395
27	1325.870605	16.229338	16.229338	0.020203

CNN
work on images
by mimicking vision

sequential
data



Motivation

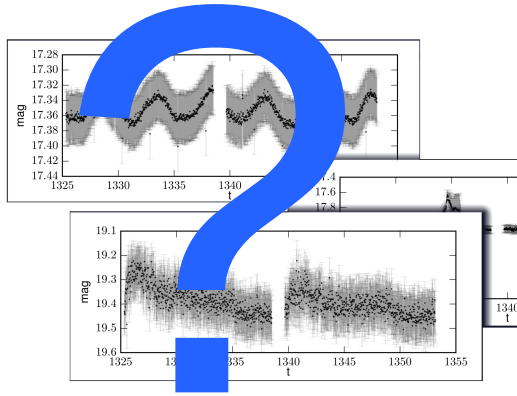
different data types lead to different NN architectures:

classical NN
work on tabulated data

1	1325.328857	16.119175	16.119175	0.018178
2	1325.349731	16.116192	16.116192	0.018189
3	1325.370605	16.118525	16.118525	0.018218
4	1325.391357	16.119541	16.119541	0.018228
5	1325.412231	16.118353	16.118353	0.018226
6	1325.433105	16.118797	16.118797	0.018227
7	1325.453857	16.120758	16.120758	0.018256
8	1325.474731	16.121220	16.121220	0.018261
9	1325.495605	16.121134	16.121134	0.018249
10	1325.516357	16.120495	16.120495	0.018251
11	1325.537231	16.123848	16.123848	0.018310
12	1325.558105	16.123949	16.123949	0.018334
13	1325.578857	16.125397	16.125397	0.018335
14	1325.599731	16.124432	16.124432	0.018331
15	1325.620605	16.123493	16.123493	0.018288
16	1325.641479	16.126211	16.126211	0.018362
17	1325.662231	16.127548	16.127548	0.018384
18	1325.683105	16.126381	16.126381	0.018342
19	1325.703979	16.126745	16.126745	0.018367
20	1325.724731	16.127941	16.127941	0.018383
21	1325.745605	16.129608	16.129608	0.018395
22	1325.766479	16.129673	16.129673	0.018406
23	1325.787231	16.162720	16.162720	0.018971
24	1325.808105	16.221766	16.221766	0.020035
25	1325.828979	16.240532	16.240532	0.020410
26	1325.849731	16.239178	16.239178	0.020395
27	1325.870605	16.229338	16.229338	0.020203

CNN
work on images
by mimicking vision

sequential
data



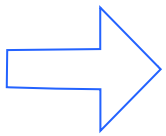
use a neural network that remembers data behavior from the past and applies it to further predictions

Recurrent Neural Networks

A **recurrent neural network** is a type of deep learning neural net that takes time and sequence into account.

Recurrent Neural Networks

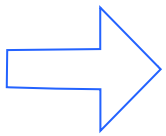
A **recurrent neural network** is a type of deep learning neural net that takes time and sequence into account.



temporal dimension

Recurrent Neural Networks

A **recurrent neural network** is a type of deep learning neural net that takes time and sequence into account.



temporal dimension

Traditional Neural Networks assume that inputs and outputs are independent of each other. The outputs of Recurrent Neural Networks depend on the prior elements within the sequence. They have an inherent **memory** as they take information from prior inputs to influence the current input and output.

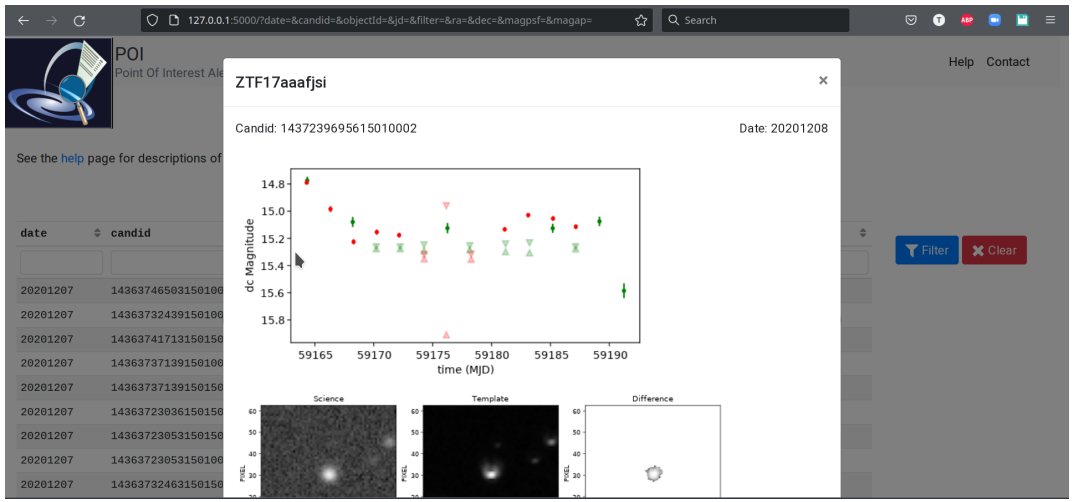
Recurrent Neural Networks

Applications: Speech recognition (Google Voice Search), Machine translation (Google Translate, deepl), Time series forecasting (stock market, weather) etc.

Recurrent Neural Networks

Applications: Speech recognition (Google Voice Search), Machine translation (Google Translate, deepl), Time series forecasting (stock market, weather) etc.

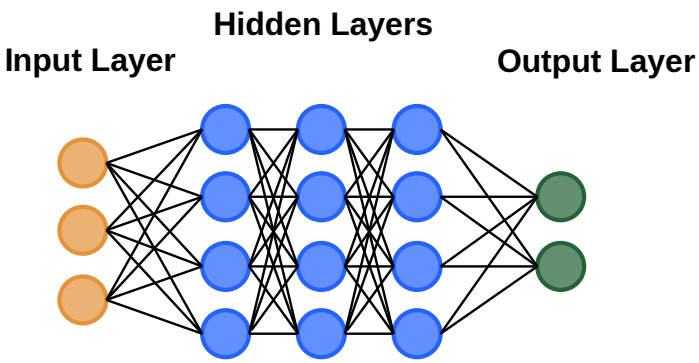
use case in astronomy: time series data, especially such that is constantly updated (not from a catalog but **live data**)



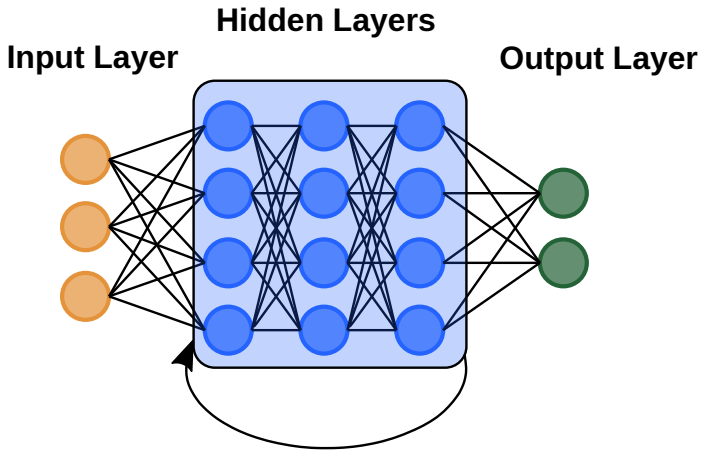
Point of Interest, a LSST broker

Recurrent Neural Networks vs. Feedforward Neural Networks

Feed-Forward Deep Learning Network



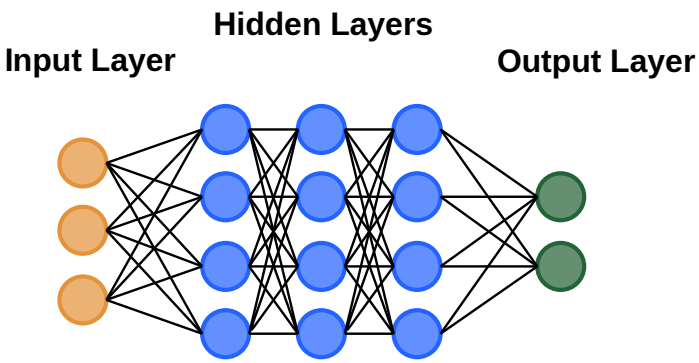
Recurrent Neural Network



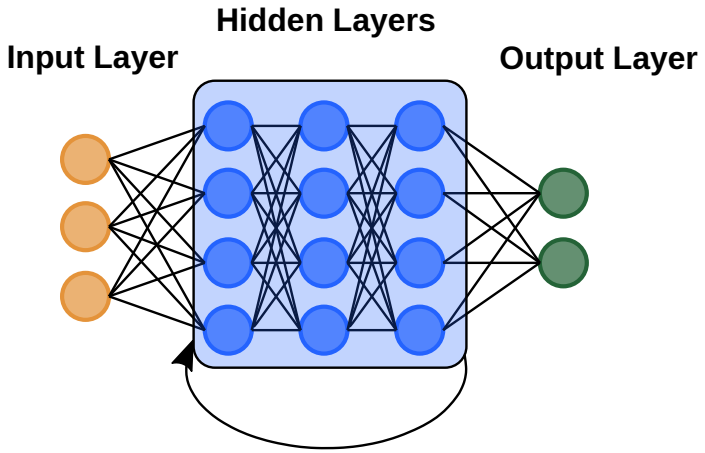
Feedforward Neural Networks follow a top-down approach: Data flows **only in one direction**, i.e. from input to output.
Recurrent Neural Networks use **feedback loops**: Features derived from earlier input are fed back into the network allowing them to **memorize.**

Recurrent Neural Networks vs. Feedforward Neural Networks

Feed-Forward Deep Learning Network



Recurrent Neural Network

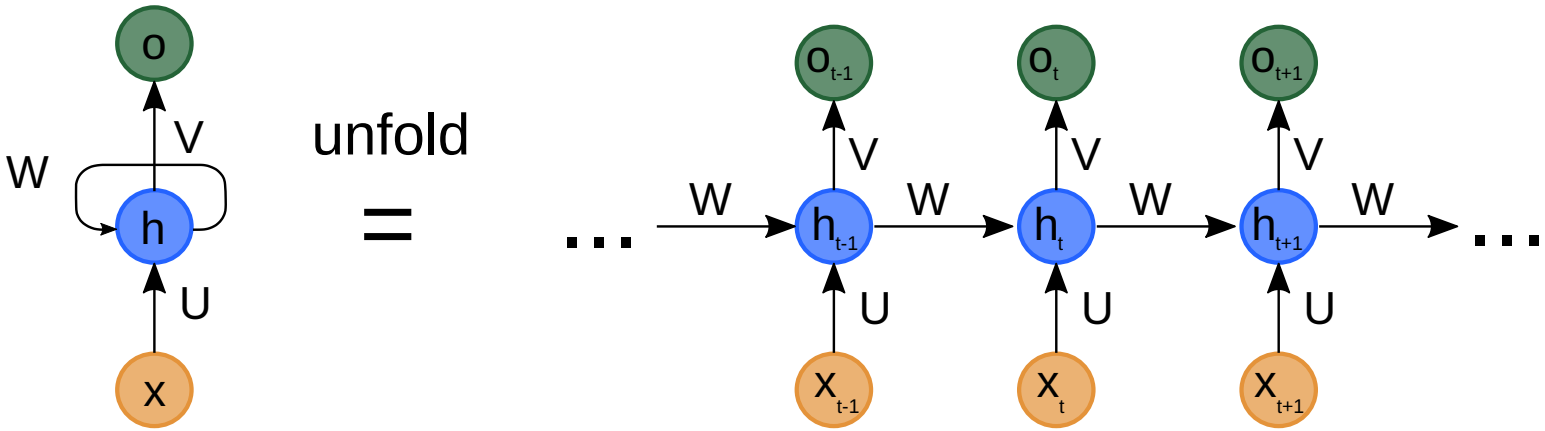


Feedforward Neural Networks follow a top-down approach: Data flows **only in one direction**, i.e. from input to output.
Recurrent Neural Networks use **feedback loops**: Features derived from earlier input are fed back into the network allowing them to **memorize.**

caution: memory is different from learning

Unfolding Recurrent Neural Networks

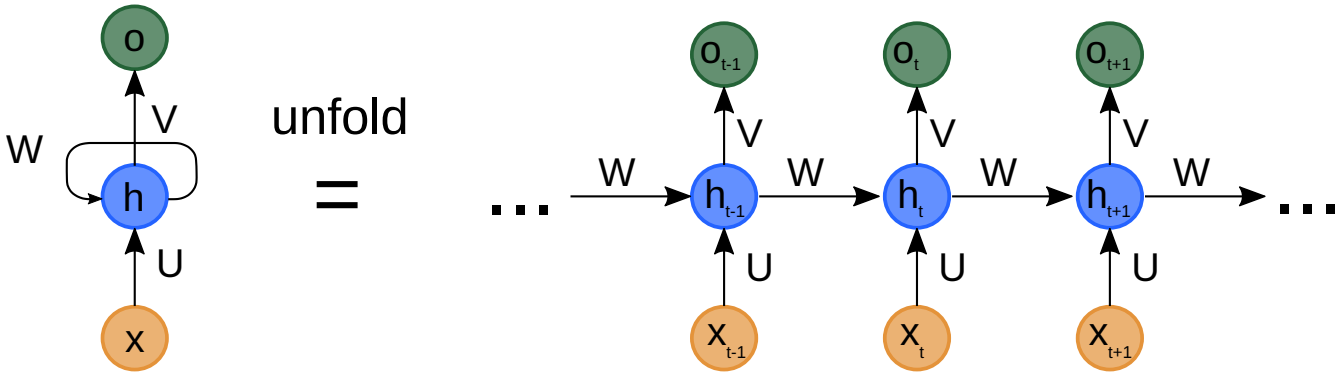
A simplified way of **representing the Recurrent Neural Network** is by unfolding/unrolling the RNN over the input sequence. For example, if we feed a 10-element sentence as input to the Recurrent Neural Network, the network would be unfolded such that it has 10 neural network layers.



Notice: the same function and the same set of parameters are used at every time step.

Functionality of a RNN

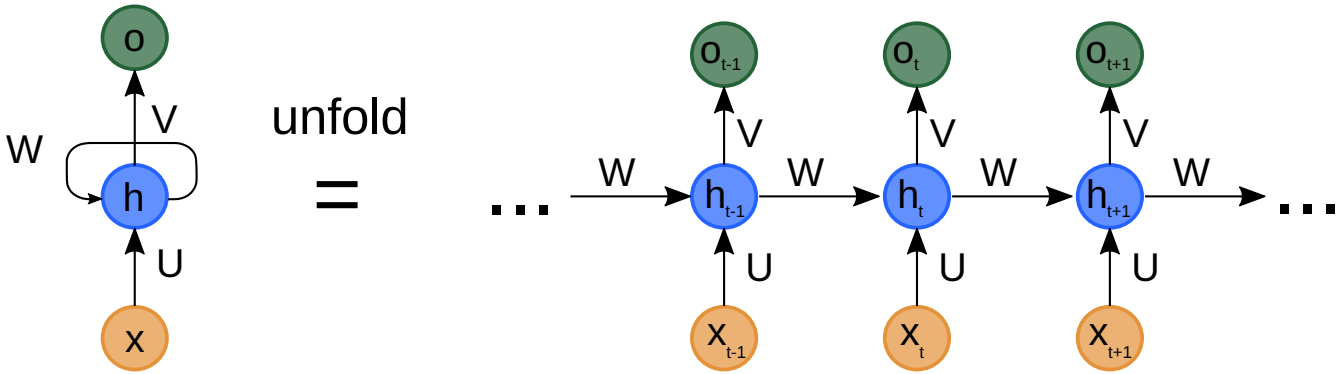
more technical:



Sequential information is preserved in the recurrent network's **hidden state**. It is finding **long-term dependencies**: correlations between events separated by many time steps. This is necessary as an event downstream in time depends upon, and is a function of, one or more events in the past.

Functionality of a RNN

more technical:



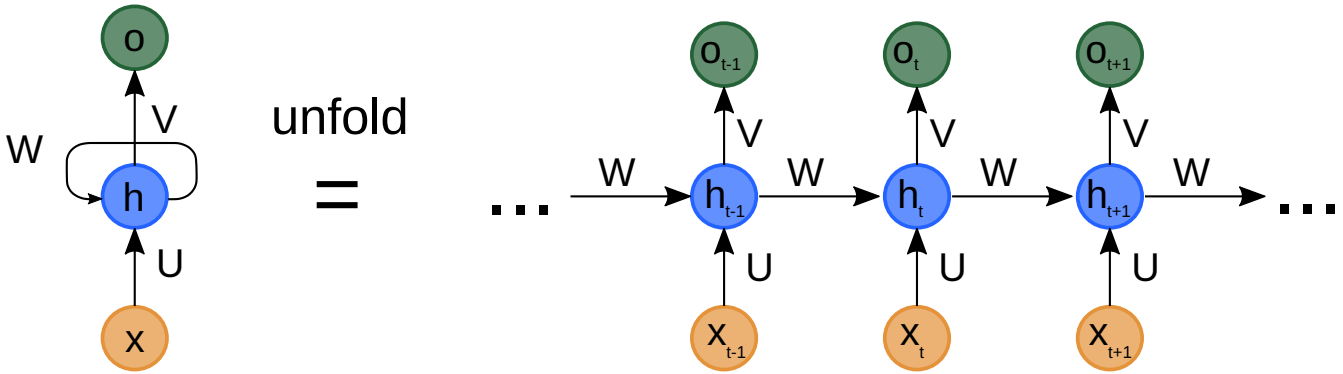
Sequential information is preserved in the recurrent network's **hidden state**. It is finding **long-term dependencies**: correlations between events separated by many time steps. This is necessary as an event downstream in time depends upon, and is a function of, one or more events in the past.

We can process a sequence of vectors **x** by applying a **recurrence equation** at every time step:

$$\mathbf{h}_t = \varphi(W\mathbf{x}_t + U\mathbf{h}_{t-1})$$

Functionality of a RNN

more technical:

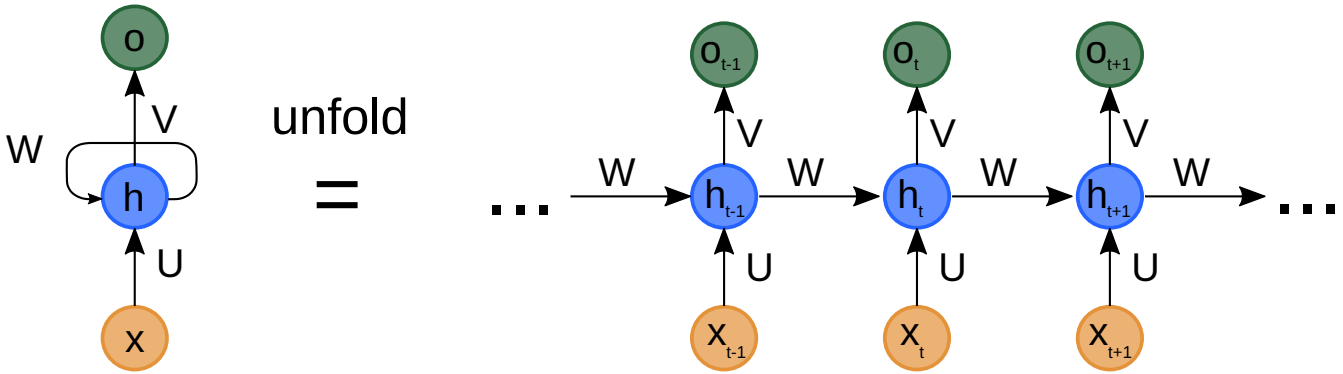


$$\mathbf{h}_t = \varphi(W\mathbf{x}_t + U\mathbf{h}_{t-1})$$

The **hidden state** at time step t is $\mathbf{h}_t$. It is a function of the input at time step x_t, modified by a **weight matrix** W (like used for feed-forward NN) added to the previous hidden state $\mathbf{h}_{t-1}$ multiplied by its own **hidden-state-to-hidden-state matrix** U. The matrices determine how much importance to give to the present input and the past hidden state.

Functionality of a RNN

more technical:

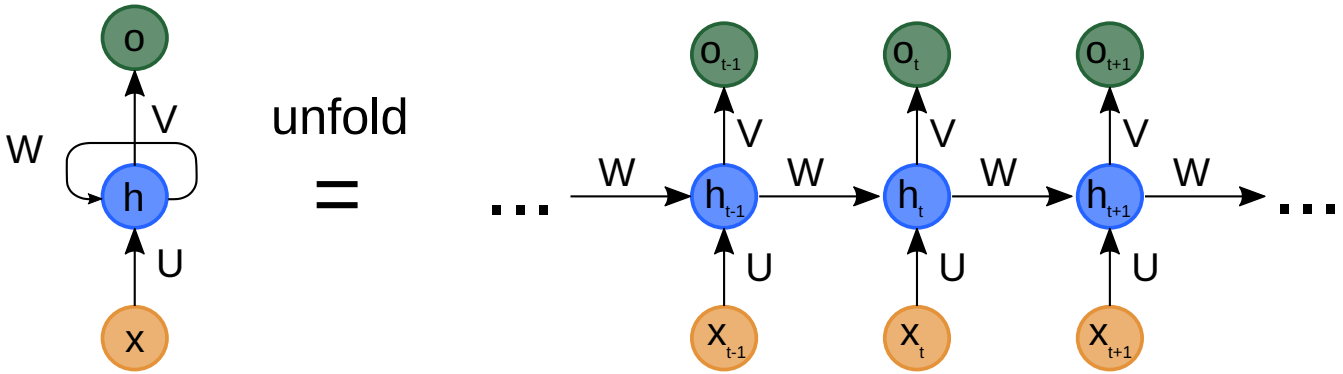


$$\mathbf{h}_t = \varphi(W\mathbf{x}_t + U\mathbf{h}_{t-1})$$

The **hidden state** at time step t is $\mathbf{h}_t$. It is a function of the input at time step x_t, modified by a **weight matrix** W (like used for feed-forward NN) added to the previous hidden state $\mathbf{h}_{t-1}$ multiplied by its own **hidden-state-to-hidden-state matrix** U. The matrices determine how much importance to give to the present input and the past hidden state. Finally, an **activation function** φ is applied.

Functionality of a RNN

more technical:



$$\mathbf{h}_t = \varphi(W\mathbf{x}_t + U\mathbf{h}_{t-1})$$

The **hidden state** at time step t is $\mathbf{h}_t$. It is a function of the input at time step x_t, modified by a **weight matrix** W (like used for feed-forward NN) added to the previous hidden state $\mathbf{h}_{t-1}$ multiplied by its own **hidden-state-to-hidden-state matrix** U. The matrices determine how much importance to give to the present input and the past hidden state. Finally, an **activation function** φ is applied.
Because of this feedback loop, each hidden state contains traces of all those that preceded h_{t-1} for as long as memory persists.

Training RNN

we have already seen:

Training a Neural Network is done by defining a loss function that measures the error/deviation between the predicted value y and the ground truth $\hat{y}$. The goal is to reduce the loss function in order to reach global minima. This is achieved in **backpropagation** by using optimizers to adjust the weights.

Training RNN

we have already seen:

Training a Neural Network is done by defining a loss function that measures the error/deviation between the predicted value y and the ground truth $\hat{y}$. The goal is to reduce the loss function in order to reach global minima. This is achieved in **backpropagation** by using optimizers to adjust the weights.

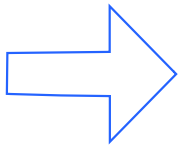
The second part of the training is the backward pass where the various derivatives are calculated.

Training RNN

we have already seen:

Training a Neural Network is done by defining a loss function that measures the error/deviation between the predicted value y and the ground truth $\hat{y}$. The goal is to reduce the loss function in order to reach global minima. This is achieved in **backpropagation** by using optimizers to adjust the weights.

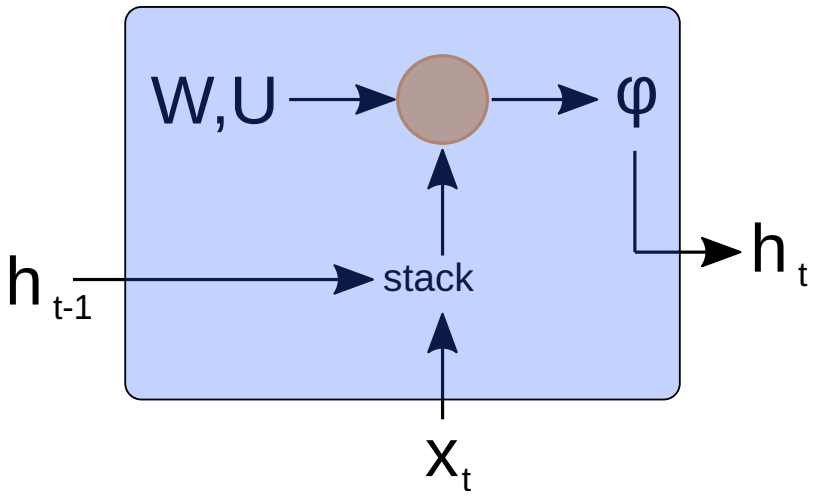
The second part of the training is the backward pass where the various derivatives are calculated.



More complex in RNN as the model backpropagate the gradients through all the hidden layers and also through time: in each time step it has to sum up all the previous contributions until the current timestamp.

Training RNN

Gradient Flow in a RNN

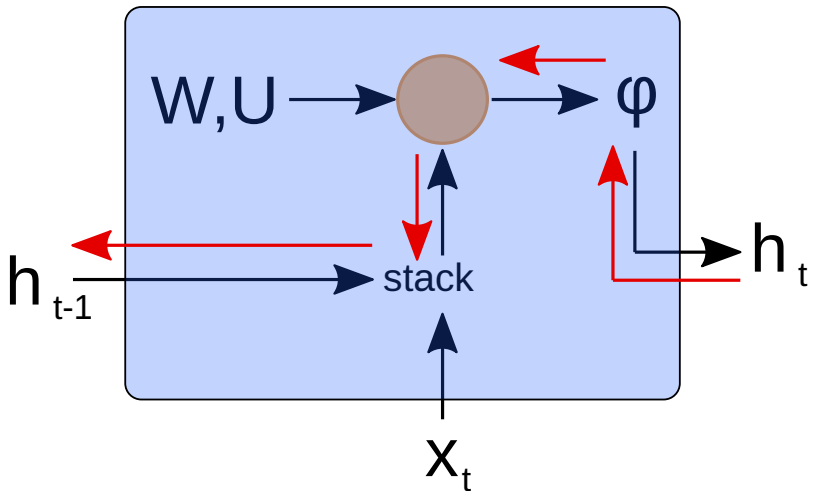


$$h_t = \varphi(W\mathbf{x}_t + U\mathbf{h}_{t-1})$$
$$= \varphi\left((W\ U)\begin{pmatrix}\mathbf{x}_t\\ \mathbf{h}_{t-1}\end{pmatrix}\right)$$

credit:
Bengio et al. *Learning long-termin dependencies with gradient descent is difficult*, IEEE Transactions of Neural networks, 1994
Pascanu et al., *On the difficulties of training recurrent neural networks*, ICML, 2013

Training RNN

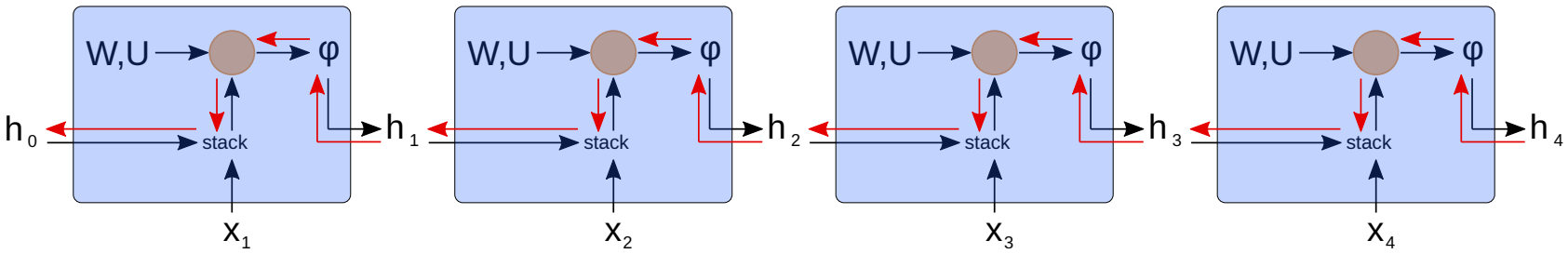
Gradient Flow in a RNN



backpropagation from $\mathbf{h}_t$ to $\mathbf{h}_{t-1}$ multiplies by U^T

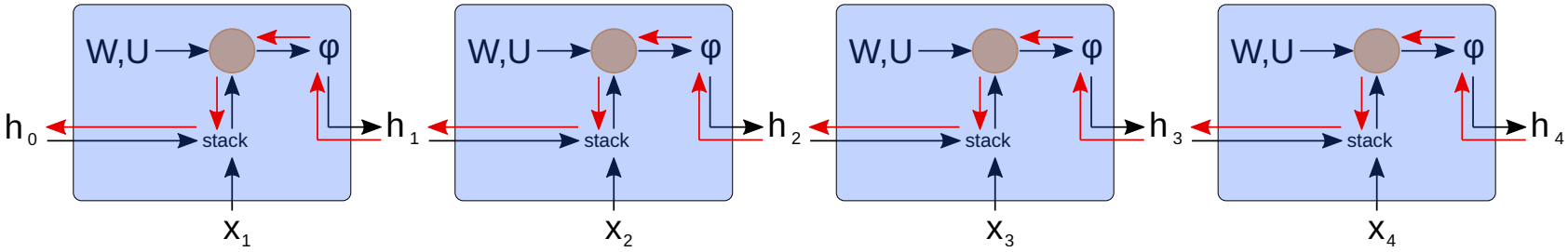
Training RNN

Gradient Flow in a RNN



Training RNN

Gradient Flow in a RNN



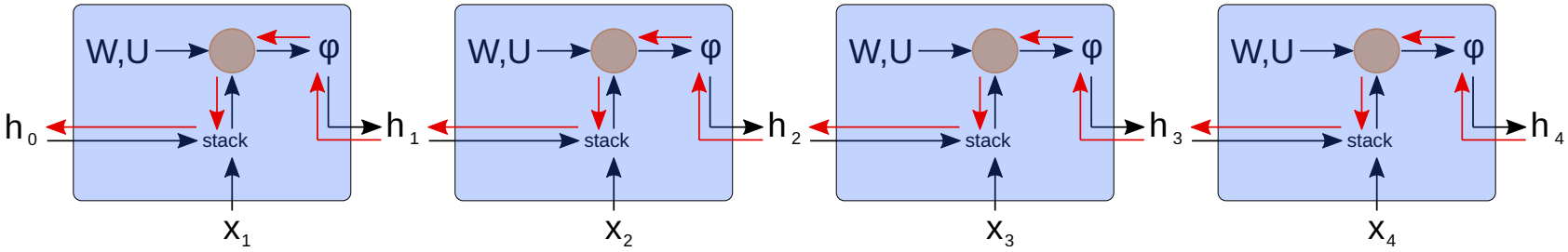
computing the gradient of h_0 involves many factors of W, U and repeated φ

largest singular value > 1: exploding gradients

largest singular value < 1: vanishing gradients

Training RNN

Gradient Flow in a RNN



computing the gradient of h_0 involves many factors of W, U and repeated φ

largest singular value > 1: exploding gradients

largest singular value < 1: vanishing gradients

gradient clipping:
scale gradient if its norm is too big

change RNN architecture

Training RNN

We've seen: the purpose of RNN is to accurately classify sequential input.

Training RNN

We've seen: the purpose of RNN is to accurately classify sequential input.

Feed-forward NN: Backpropagation moves backward from the final error through the outputs, weights w and inputs of each hidden layer, calculating their partial derivatives $\partial E/\partial w$ which are used by gradient descent to adjust the weights.

Training RNN

We've seen: the purpose of RNN is to accurately classify sequential input.

Feed-forward NN: Backpropagation moves backward from the final error through the outputs, weights w and inputs of each hidden layer, calculating their partial derivatives $\partial E/\partial w$ which are used by gradient descent to adjust the weights.

RNN: Extension of backpropagation called **Backpropagation Through Time (BPTT)** were time is expressed by a well-defined, ordered series of calculations linking one time step to the next.

Training RNN

We've seen: the purpose of RNN is to accurately classify sequential input.

Feed-forward NN: Backpropagation moves backward from the final error through the outputs, weights w and inputs of each hidden layer, calculating their partial derivatives $\partial E/\partial w$ which are used by gradient descent to adjust the weights.

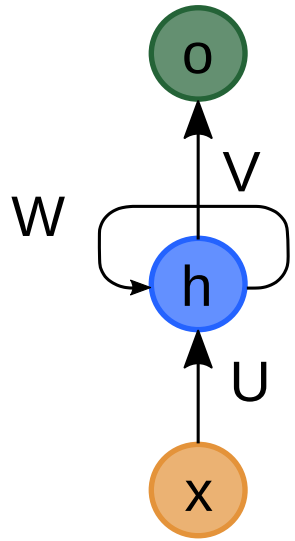
RNN: Extension of backpropagation called **Backpropagation Through Time (BPTT)** were time is expressed by a well-defined, ordered series of calculations linking one time step to the next.

BPTT works by **unfolding** a RNN in time. The unfolded network contains the individual inputs and outputs, but every copy of the network shares the same parameters. Each timestep of the unrolled recurrent neural network may be seen as an additional layer given the order dependence of the problem and the internal state from the previous timestep is taken as an input on the subsequent timestep.

Training RNN with BPTT

We use the following **definitions**:

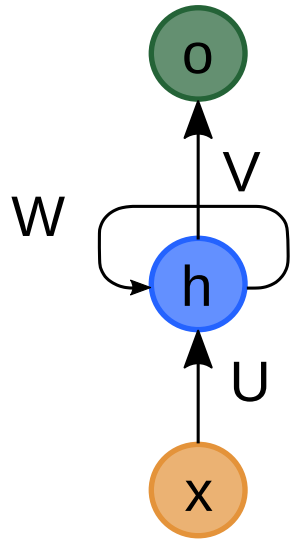
h_t: hidden state at time t
x_t: input at time t
o_t: output at time t
$\widehat{o}_t$: desired output (ground truth) at time t
U: weights at input layer
V: weights at output layer
W: weights at hidden layer



Training RNN with BPTT

We use the following **definitions**:

h_t: hidden state at time t
x_t: input at time t
o_t: output at time t
$\widehat{o_t}$: desired output (ground truth) at time t
U: weights at input layer
V: weights at output layer
W: weights at hidden layer



The hidden states and outputs at each time are:

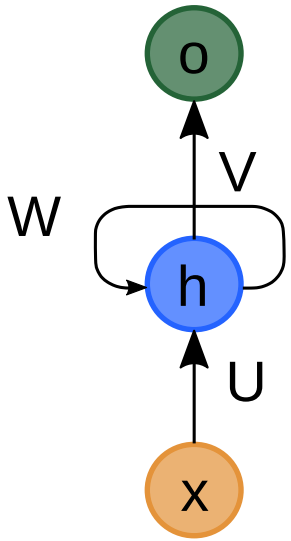
$$h_t = f(x_t, h_{t-1}, W, U)$$
$$o_t = g(h_t, V)$$

where f and g are the activations of the hidden and output layer, respectively.
Hence we have a chain of values $\ldots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \ldots$ that depends on each other via recurrent computation.
Forward propagation is straightforward: we only need to loop trough the (x_t, h_t, o_t) one time step at a time.

Training RNN with BPTT

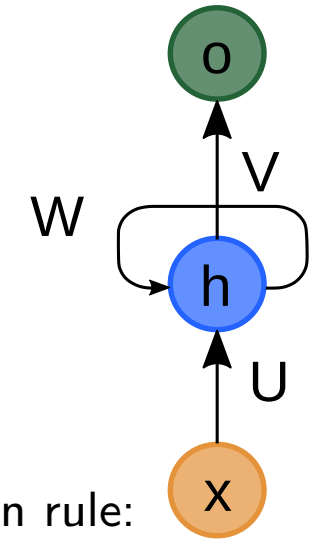
Backpropagation:
The discrepancy between output o_t and desired output $\widehat{o}_t$ is evaluated by a loss function E across all T time steps as:

$$E(x_1, ..., x_T, \widehat{o}_1, ..., \widehat{o}_T, W, V) = \frac{1}{T} \sum_{i=1}^{T} l(\widehat{o}_t, o_t)$$



Training RNN with BPTT

Backpropagation:
The discrepancy between output o_t and desired output $\widehat{o}_t$ is evaluated by a loss function E across all T time steps as:



$$E(x_1, ..., x_T, \widehat{o}_1, ..., \widehat{o}_T, W, V) = \frac{1}{T} \sum_{i=1}^{T} l(\widehat{o}_t, o_t)$$

What is tricky with BPTT is computing the gradients w.r.t. the parameter W of the objective function E. With the chain rule:

$$\frac{\partial E}{\partial W} = \frac{1}{T} \sum_{t=1}^{T} \frac{\partial l(\widehat{o}_t, o_t)}{\partial W}$$

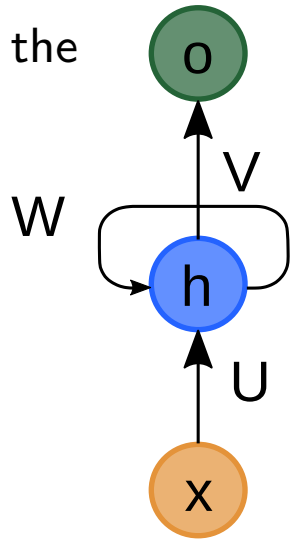
$$= \frac{1}{T} \sum_{t=1}^{T} \frac{\partial l(\widehat{o}_t, o_t)}{\partial o_t} \frac{\partial g(h_t, V)}{\partial h_t} \frac{\partial h_t}{\partial W}$$

The first and second factors of the product are easy to compute. The third factor $\partial h_t / \partial W$ is where it comes tricky as we need to recurrently compute the effect of W on h_t. According to the recurrent computation, h_t depends on both h_{t-1} and W, where computation of h_{t-1} also depends on W.

Training RNN with BPTT

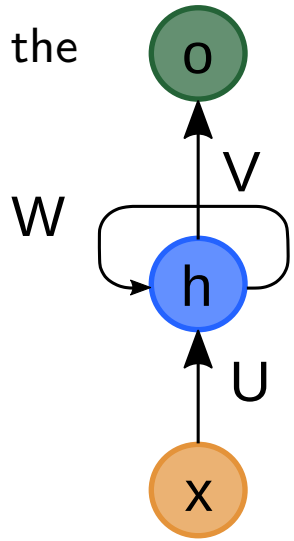
Note that the weight matrices U, V, W are shared across all the time sequence. Therefore, we can differentiate w.r.t them at each time step and sum all together.
We use the chain rule:

$$\frac{\partial h_t}{\partial W} = \frac{\partial f(x_t, h_{t-1}, W)}{\partial W} + \frac{\partial f(x_t, h_{t-1}, W)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W}$$



Training RNN with BPTT

Note that the weight matrices U, V, W are shared across all the time sequence. Therefore, we can differentiate w.r.t them at each time step and sum all together.
We use the chain rule:



$$\frac{\partial h_t}{\partial W} = \frac{\partial f(x_t, h_{t-1}, W)}{\partial W} + \frac{\partial f(x_t, h_{t-1}, W)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W}$$

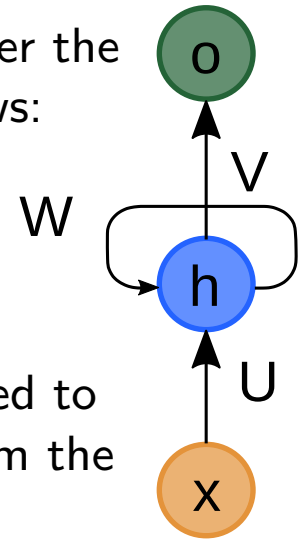
We get (derivation not shown here):

$$\frac{\partial h_t}{\partial W} = \frac{\partial f(x_t, h_{t-1}, W)}{\partial W} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial f(x_j, h_{j-1}, W)}{\partial h_{j-1}} \right) \frac{\partial f(x_i, h_{i-1}, W)}{\partial W}$$

While we can use the chain rule to compute $\partial h_t / \partial W$ recursively, this chain can get very long for large t.

Training RNN with BPTT

We then compute the gradient w.r.t. U. Similarly, we consider the time step $t+1$ and calculate the gradients w.r.t. U as follows:



$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \hat{o}_{t+1}} \frac{\partial \hat{o}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial U}$$

Because h_t and x_{t+1} both make contributions to h_{t+1}, we need to backpropagate h_t as well. If we consider the contribution from the time step, we get further:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \hat{o}_{t+1}} \frac{\partial \hat{o}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial U} + \frac{\partial E_{t+1}}{\partial \hat{o}_{t+1}} \frac{\partial \hat{o}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} \frac{\partial h_t}{\partial U}$$

Thus, summing up all the contributions from $t+1$ to 1 via Backpropagation, we can yield the gradient at the time step $t+1$ and then take the derivative with respect to U over the whole sequence as:

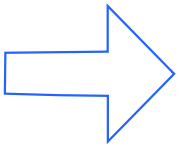
$$\frac{\partial E}{\partial U} = \sum_{t}^{T} \sum_{k=1}^{t+1} \frac{\partial E_{t+1}}{\partial \hat{o}_{t+1}} \frac{\partial \hat{o}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial U}$$

Training RNN with BPTT

BPTT can be computationally expensive as the number of timesteps increases.

Training RNN with BPTT

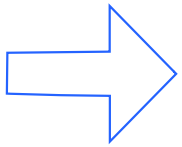
BPTT can be computationally expensive as the number of timesteps increases.



we are looking for robust estimators that generalize well. Hence this strategy is almost never used in practice.

Training RNN with BPTT

BPTT can be computationally expensive as the number of timesteps increases.



we are looking for robust estimators that generalize well. Hence this strategy is almost never used in practice.

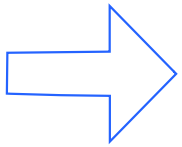
Truncating Time Steps

We can truncate the summation after τ steps. This leads to an approximation of the true gradient, simply by terminating the sum at $\frac{\partial h_{t-\tau}}{\partial W}$. This method is commonly referred to as truncated backpropgation through time (Jaeger, 2002).

One of the consequences of this is that the model focuses primarily on short-term influence rather than long-term consequences. This is actually desirable, since it biases the estimate towards simpler and more stable models.

Training RNN with BPTT

BPTT can be computationally expensive as the number of timesteps increases.



we are looking for robust estimators that generalize well. Hence this strategy is almost never used in practice.

Randomized Truncation
Alternatively, we can replace $\partial h_t/\partial W$ by a random variable which is correct in expectation but truncates the sequence. This is achieved by using a sequence of ζ_t with predefined $0 \leq \pi_t \leq 1$, where $P(\zeta_t = 0) = 1 - \pi_t$ and $P(\zeta_t = \pi_t^{-1}) = \pi_t$. We use this to replace the gradient $\partial h_t/\partial W$ with

$$z_t = \frac{\partial f(x_t, h_{t-1}, W)}{\partial W} + \zeta_t \frac{\partial f(x_t, h_{t-1}, W)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W}$$

Whenever $\zeta_t = 0$ the recurrent computation terminates at that time step t. This leads to a weighted sum of sequences of varying lengths, where long sequences are rare but appropriately overweighted. This idea was proposed by Tallec and Ollivier (2017).

Training RNN

more on **Truncated Backpropagation Through Time (TBPTT)**:

This is a modified BPTT training algorithm for RNN where the sequence is processed one timestep at a time and periodically (k_1 timesteps) the BPTT update is performed back for a fixed number of timesteps (k_2 timesteps).

Training RNN

more on **Truncated Backpropagation Through Time (TBPTT)**:

This is a modified BPTT training algorithm for RNN where the sequence is processed one timestep at a time and periodically (k_1 timesteps) the BPTT update is performed back for a fixed number of timesteps (k_2 timesteps).

We can summarize the algorithm as follows:

1. Select a sequence of k_1 timesteps of input-output pairs.
2. Unfold the RNN, calculate & accumulate errors over k_2 timesteps.
3. Fold up the RNN and update weights.
4. Repeat

Training RNN

more on **Truncated Backpropagation Through Time (TBPTT)**:

This is a modified BPTT training algorithm for RNN where the sequence is processed one timestep at a time and periodically (k_1 timesteps) the BPTT update is performed back for a fixed number of timesteps (k_2 timesteps).

We can summarize the algorithm as follows:

1. Select a sequence of k_1 timesteps of input-output pairs.
2. Unfold the RNN, calculate & accumulate errors over k_2 timesteps.
3. Fold up the RNN and update weights.
4. Repeat

Influence of the parameters on TBPTT:

k_1: Number of forward-pass timesteps between updates. This influences the training speed, given how often weight updates are performed.

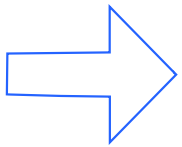
k_2: Number of timesteps to which to apply BPTT. It should be large enough to capture the temporal structure in the problem for the network to learn. Too large values result in vanishing gradients.

Training RNN

more on **Truncated Backpropagation Through Time (BPTT)**:

Using the notation from Williams & Peng (1990) for common configurations:

TBPTT(n,n): Updates are performed at the end of the sequence across all n timesteps in the sequence (e.g. classical BPTT).
TBPTT(1,n): timesteps are processed one at a time followed by an update that covers all timesteps seen so far (e.g. classical TBPTT by Williams & Peng (1990)).
TBPTT(k_1,1): The network likely does not have enough temporal context to learn, relying heavily on internal state and inputs.
TBPTT(k_1,k_2), where $k_1 < k_2 < n$: Multiple updates are performed per sequence which can accelerate training.
TBPTT(k_1,k_2), where $k_1 = k_2$: A common configuration where a fixed number of timesteps are used for both forward and backward-pass timesteps (e.g. 10s to 100s).

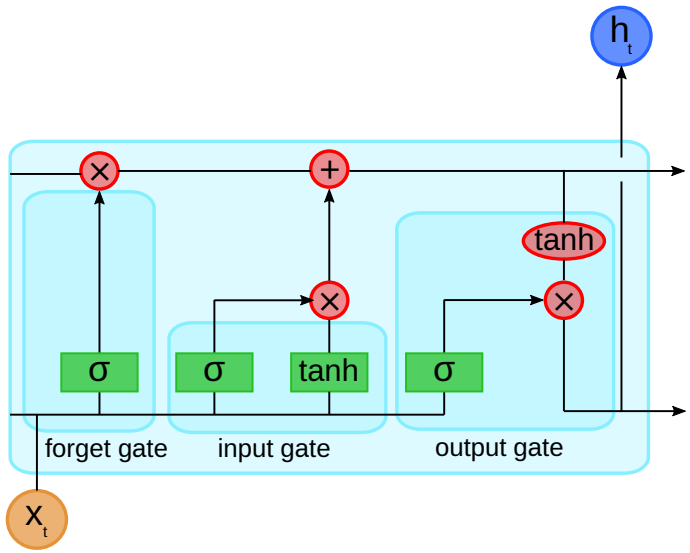


all configurations attempt to approximate BPTT with faster training and more stable results

Long Short Term Memory (LSTM)

This type of RNN can **learn long-term dependencies**. Unlike an RNN equipped with a simple layer in a network block, a LSTM unit carries out additional operations to remember crucial information and forget unnecessary information while learning.

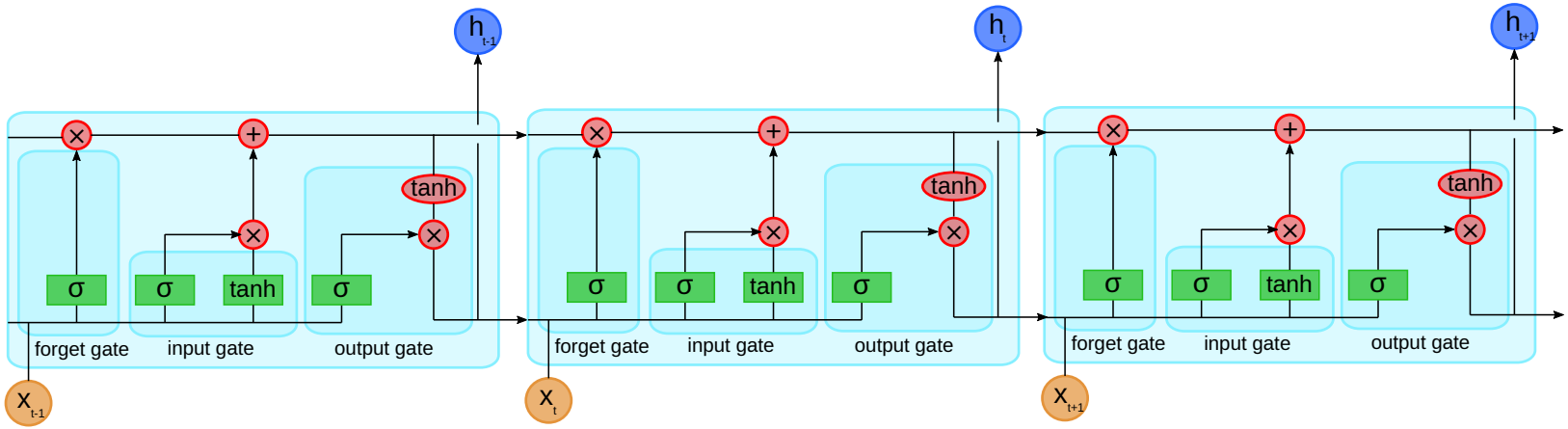
- The **input gate** identifies new information.
- The **forget gate** learns what details to be discarded from the block.
- The **output gate** passes on information to the next cell.



Long Short Term Memory (LSTM)

This type of RNN can **learn long-term dependencies**. Unlike an RNN equipped with a simple layer in a network block, a LSTM unit carries out additional operations to remember crucial information and forget unnecessary information while learning.

- The **input gate** identifies new information.
- The **forget gate** learns what details to be discarded from the block.
- The **output gate** passes on information to the next cell.



Long Short Term Memory (LSTM)

Long Short-Term Memory units were proposed by S. Hochreiter and J. Schmidhuber (1997) as a solution to the vanishing gradient problem.

Long Short Term Memory (LSTM)

Long Short-Term Memory units were proposed by S. Hochreiter and J. Schmidhuber (1997) as a solution to the vanishing gradient problem.

LSTMs help **preserve the error** that can be backpropagated through time and layers. In this way, they allow recurrent nets to continue to learn over many time steps (over 1000).

Long Short Term Memory (LSTM)

Long Short-Term Memory units were proposed by S. Hochreiter and J. Schmidhuber (1997) as a solution to the vanishing gradient problem.

LSTMs help **preserve the error** that can be backpropagated through time and layers. In this way, they allow recurrent nets to continue to learn over many time steps (over 1000).

LSTMs contain information outside the normal flow of the recurrent network in a gated cell. Information can be stored in, written to, or read from a cell, **similar to data in a computer's memory**. The cell makes decisions about what to store, and when to allow read, write and delete operations, via gates that open and close. These **gates are analog**, implemented with element-wise multiplication by sigmoids, which are in the range [0,1]. The advantage over digital is that the **signal is differentiable**, and thus suitable for backpropagation.

Long Short Term Memory (LSTM)

Long Short-Term Memory units were proposed by S. Hochreiter and J. Schmidhuber (1997) as a solution to the vanishing gradient problem.

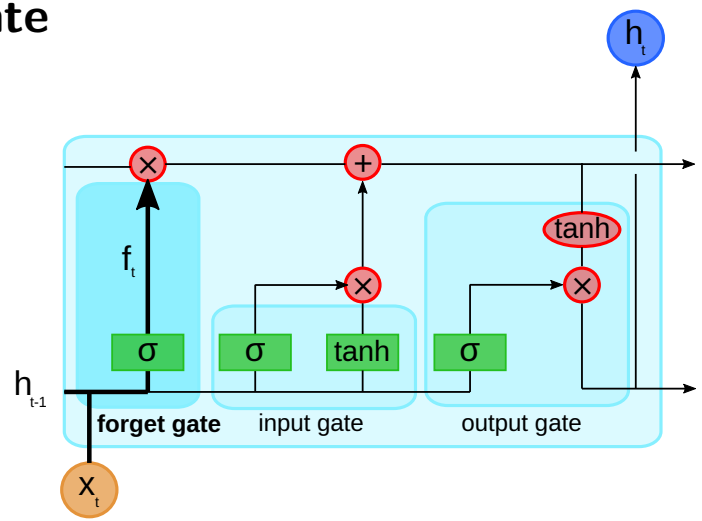
LSTMs help **preserve the error** that can be backpropagated through time and layers. In this way, they allow recurrent nets to continue to learn over many time steps (over 1000).

LSTMs contain information outside the normal flow of the recurrent network in a gated cell. Information can be stored in, written to, or read from a cell, **similar to data in a computer's memory**. The cell makes decisions about what to store, and when to allow read, write and delete operations, via gates that open and close. These **gates are analog**, implemented with element-wise multiplication by sigmoids, which are in the range [0,1]. The advantage over digital is that the **signal is differentiable**, and thus suitable for backpropagation.

The cells learn when to allow data to enter, leave or be deleted through backpropagating error and adjusting weights via gradient descent.

Long Short Term Memory (LSTM)

Forget Gate

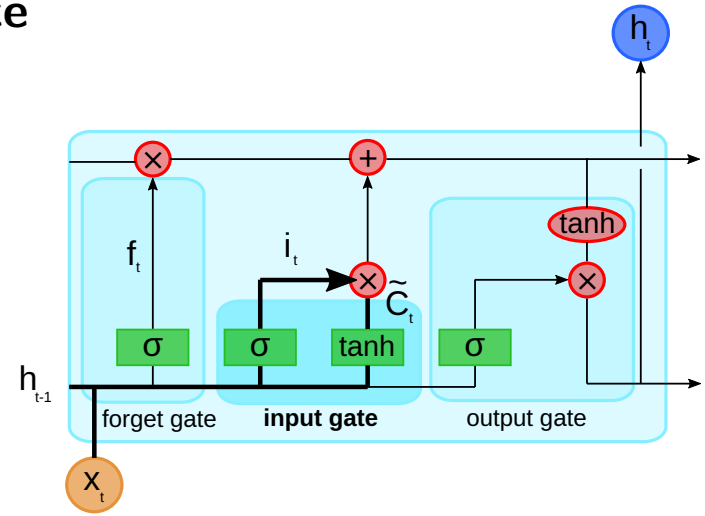


The inputs h_{t-1} and x_t are passed to the sigmoid activation function which outputs values between 0 and 1. The sigmoid function acts as a gate: 0 means completely forget and 1 means completely retain information.
With the bias b_f and W_f is the combined weight of the 2 inputs, we get:

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

Long Short Term Memory (LSTM)

Input Gate



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

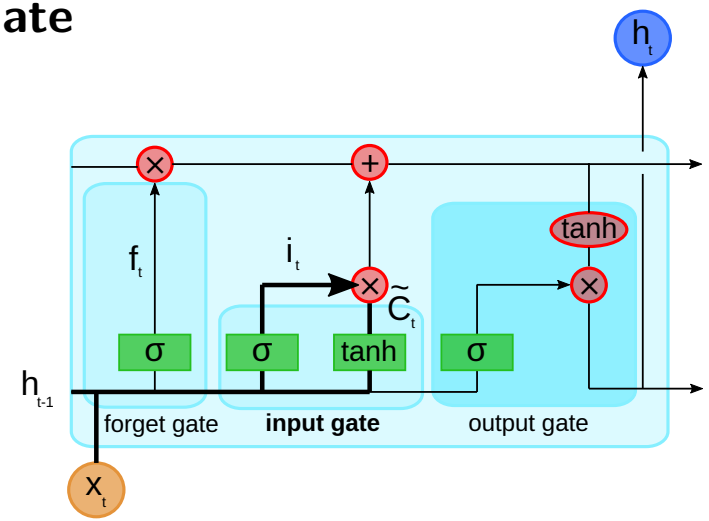
This gate identifies and adds new information to the cell state.

Step 1: The sigmoid layer outputs a value between 0 and 1 based on the inputs h_{t-1} and x_t. These inputs will also be passed to the tanh layer which outputs values between -1 and 1.

Step 2: The output of the sigmoid layer and tanh layer is multiplied Now, the cell state is updated from C_{t-1} (previous LSTM cell output) to C_t (current LSTM cell output).

Long Short Term Memory (LSTM)

Output Gate

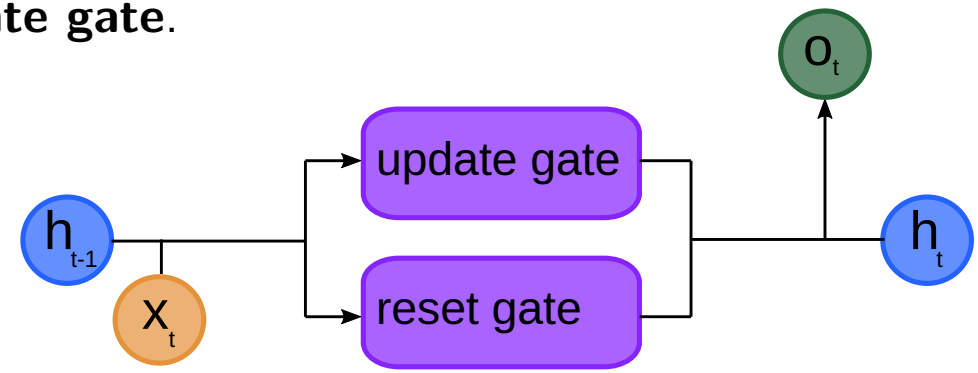


$$\sigma_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = \sigma_t \times \tanh(C_t)$$

First, the cell state is passed through the tanh function and simultaneously inputs h_{t-1} and x_t were sent to the sigmoid function layer.
Then multiplication takes place and h_t is the output of this memory cell which is passed on to the next cell.

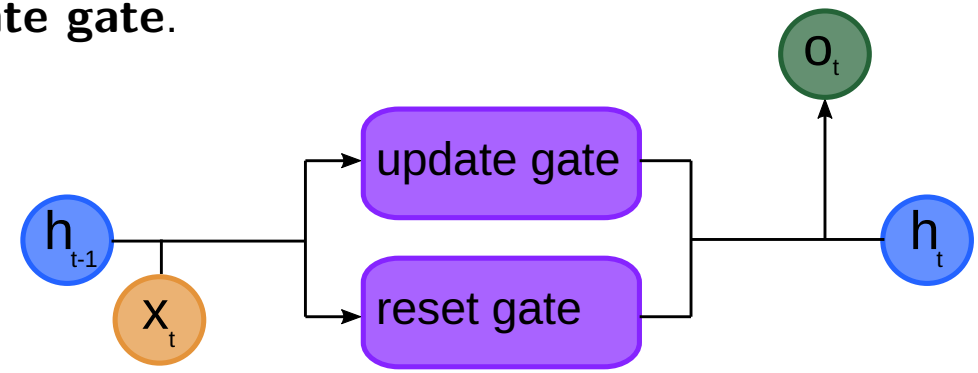
Gated Recurrent Unit (GRU)

Gated Recurrent Units (GRU) are a streamlined version of the LSTM, introduced by Cho et al. (2014).
The LSTM's three gates are replaced by two: **the reset gate and the update gate**.



Gated Recurrent Unit (GRU)

Gated Recurrent Units (GRU) are a streamlined version of the LSTM, introduced by Cho et al. (2014).
The LSTM's three gates are replaced by two: **the reset gate and the update gate**.



As with LSTMs, these gates have sigmoid activations.
The **reset gate** controls how much of the previous state we might still want to remember. The **update gate** allows us to control how much of the new state is just a copy of the old state. The outputs of two gates are given by two fully connected layers with a sigmoid activation function.
GRU's performance on certain tasks like speech signal modeling and natural language processing was found to be similar to that of LSTM.

Gated Recurrent Unit (GRU)

Update Gate: Amount of information that must be passed forward

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

Where $W^{(z)}$ is the weight associated with x_t, $U^{(z)}$ is the weight associated with input from the previous state that is h_{t-1} and σ is the sigmoid activation function. The output z_t will be between 0 and 1 based on which information will be passed on.

Gated Recurrent Unit (GRU)

Reset Gate: Decides the amount of information to forget
The output of the reset gate is calculated as

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

Where $W^{(r)}$ is the weight associated with x_t, $U^{(r)}$ is the weight associated with input from the previous state that is h_{t-1} and σ is the sigmoid activation function. The output r_t, will be between 0 and 1 based on which information will be forgotten. Now, the important step is adding a memory component called the reset gate into the network.
We calculate this as

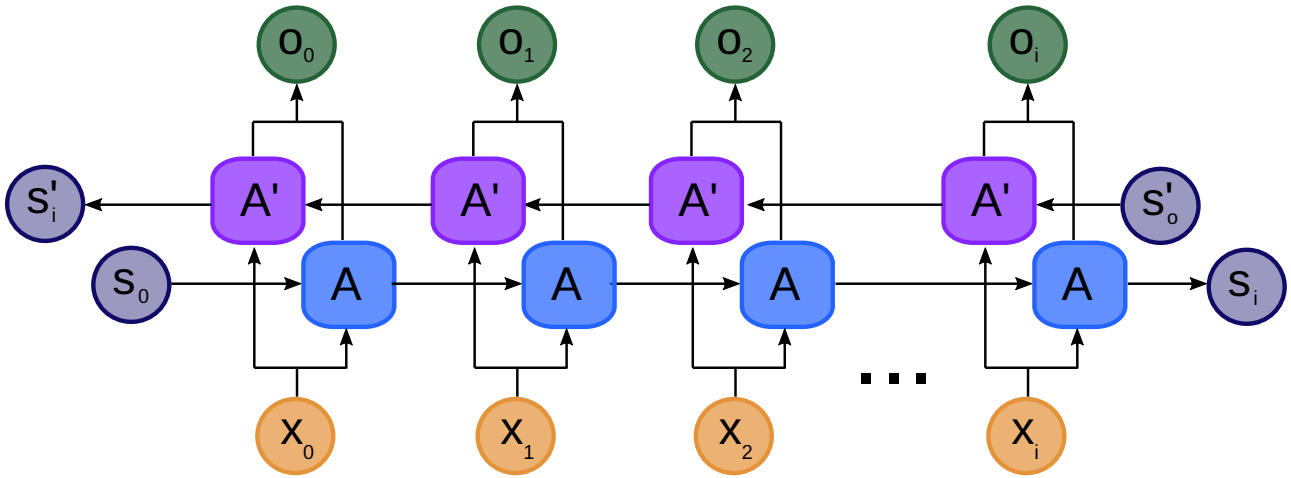
$$h'_t = \tanh(W x_t + r_t \cdot U h_{t-1})$$

Now finally, we calculate the current state h_t which will be passed on:

$$h_t = z_t \cdot h_{t-1} + (1 - z_t) \odot h'_t$$

Recurrent Neural Networks Architectures

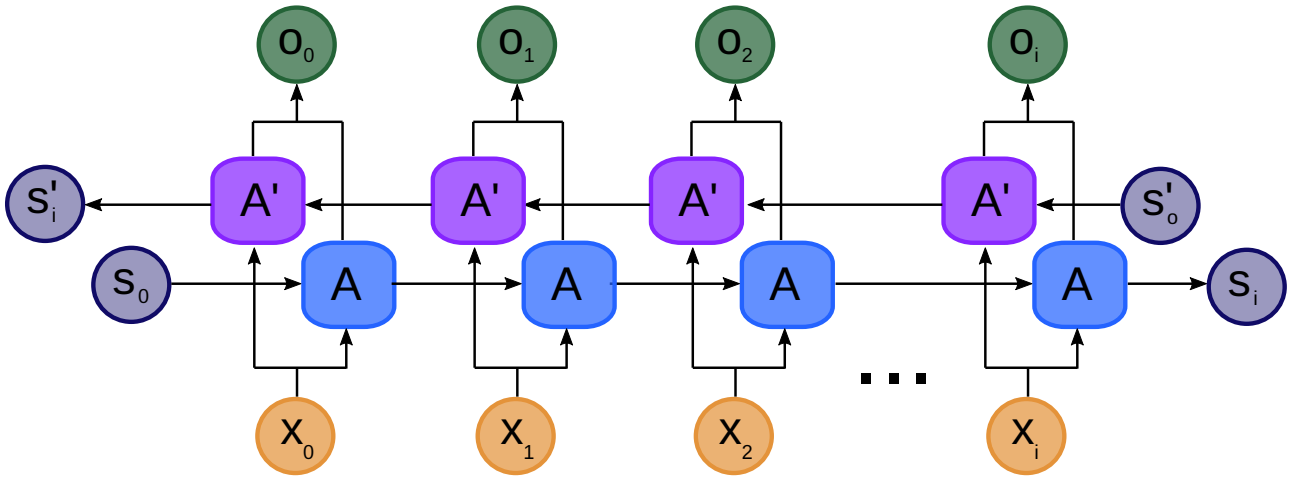
Bidirectional Recurrent Neural Networks (BRNN) rely not only only past and present events like conventional RNN, but also future events.
Example: predicting a word to be included in a sentence might require us to look into the future, i.e., a word in a sentence could depend on a future event. (You might have seen this when you use speech recognition: already "recognized" words might change later on!)



A BRNN involves **coupling two RNNs** moving in opposite directions. Their outputs are concatenated at each time step. The individual network blocks in a BRNN can either be a traditional RNN, GRU, or LSTM.

Recurrent Neural Networks Architectures

Bidirectional Recurrent Neural Networks (BRNN) rely not only only past and present events like conventional RNN, but also future events.
Example: predicting a word to be included in a sentence might require us to look into the future, i.e., a word in a sentence could depend on a future event. (You might have seen this when you use speech recognition: already "recognized" words might change later on!)



A BRNN involves **coupling two RNNs** moving in opposite directions. Their outputs are concatenated at each time step. The individual network blocks in a BRNN can either be a traditional RNN, GRU, or LSTM.

The **drawback** of BRNN is that it is slow.

Sequence to Sequence Learning

The idea behind Sequence to Sequence (Seq2Seq) learning is that input data that is received in one language is **translated into another language**. ("Language" can be very abstract; also codes are a language). They build an **Encoder-Decoder architecture**.

Sequence to Sequence Learning

The idea behind Sequence to Sequence (Seq2Seq) learning is that input data that is received in one language is **translated into another language**. ("Language" can be very abstract; also codes are a language). They build an **Encoder-Decoder architecture**.

Sequence to sequence learning was proposed by Mikolov (2012), further developed by Google.

Applications are speech recognition, machine translation, image captioning and question answering.

Sequence to Sequence Learning

The idea behind Sequence to Sequence (Seq2Seq) learning is that input data that is received in one language is **translated into another language**. ("Language" can be very abstract; also codes are a language). They build an **Encoder-Decoder architecture**.

Sequence to sequence learning was proposed by Mikolov (2012), further developed by Google.

Applications are speech recognition, machine translation, image captioning and question answering.

Types of Sequence to Sequence Learning:

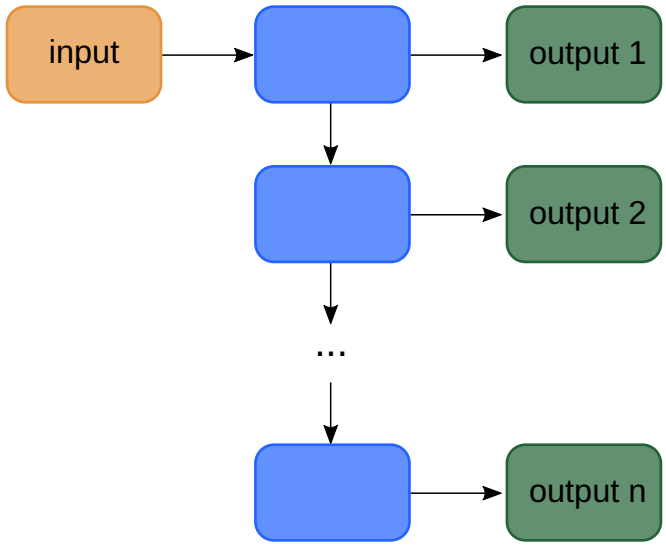
1. Sequence to Sequence: Outputs are equal to the number of inputs.
2. Sequence to Vector: A single output is given for n number of inputs.
3. Vector to Sequence: n number of outputs is received for 1 input.
4. Vector to Vector: single output is received for a single input.

Sequence to Sequence (One-to-one) RNN

The most straightforward type of RNN is One-to-One, which allows a single input and a single output. It has fixed input and output sizes and acts as a standard neural network. One-to-One RNNs are applied to image classification.

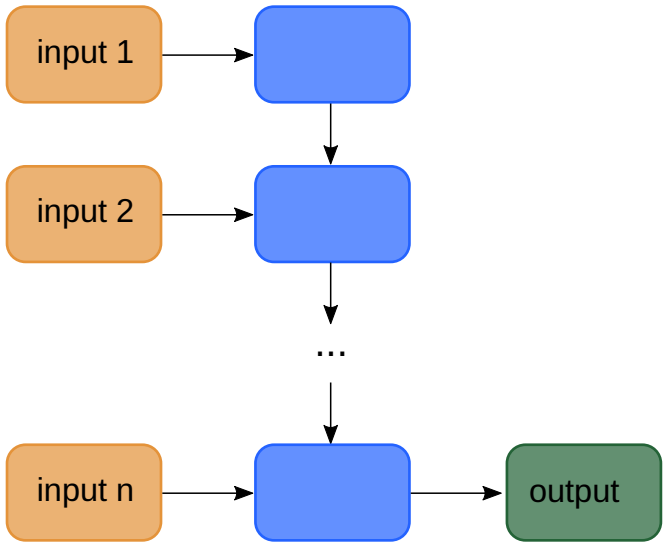
Sequence to Vector (One-to-many) RNN

One-to-Many is a type of RNN that expects multiple outputs on a single input given to the model. The input size is fixed and gives a series of data outputs. One-to-Many RNNs are used for e.g. music generation and image captioning.



Vector to Sequence (Many-to-one) RNN

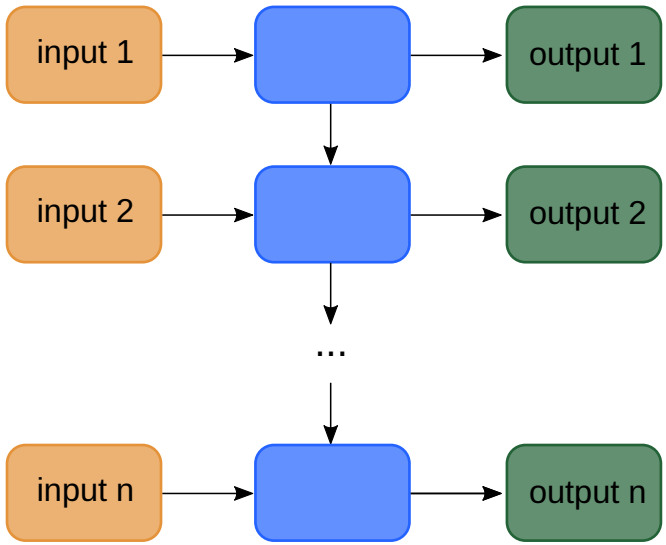
Many-to-One RNN converges a sequence of inputs into a single output by a series of hidden layers learning the features. Sentiment Analysis is a common example of this type of RNN.



Vector to Vector (Many-to-Many) RNN

This type of RNN is used to generate a sequence of output data from a sequence of input units. It is further divided into the following two subcategories:

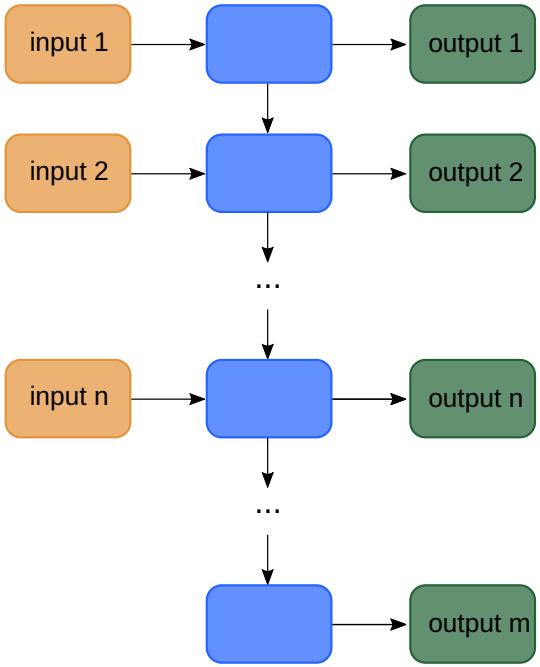
Equal Size: In this case, the input and output layer size is exactly the same.



Many-to-Many RNN

This type of RNN is used to generate a sequence of output data from a sequence of input units. It is further divided into the following two subcategories:

Unequal Size: In this case, inputs and outputs have different numbers of units. Its application can be found in Machine Translation.



Sequence to Sequence Learning

The building blocks (units) in Seq2seq learning are regular RNN, or a LSTM or GRU (avoid vanishing gradient). The primary components are one **encoder** and one **decoder network**. The encoder turns each item into a corresponding hidden vector containing the item and its context. The decoder reverses the process, turning the vector into an output item, using the previous output as the input context.

Recurrent Neural Networks Architectures

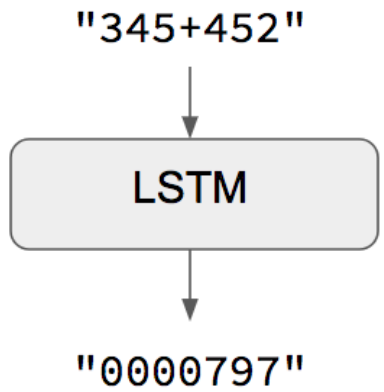
Sequence to Sequence Learning

The building blocks (units) in Seq2seq learning are regular RNN, or a LSTM or GRU (avoid vanishing gradient). The primary components are one **encoder** and one **decoder network**. The encoder turns each item into a corresponding hidden vector containing the item and its context. The decoder reverses the process, turning the vector into an output item, using the previous output as the input context.

The trivial case: sequence to sequence

we've seen: that's when input and output sequences have the same length

When both input sequences and output sequences have the same length, one can implement such models simply with a Keras LSTM or GRU layer(s). Caveat: In the general case, information about the entire input sequence is necessary in order to start generating the target sequence.

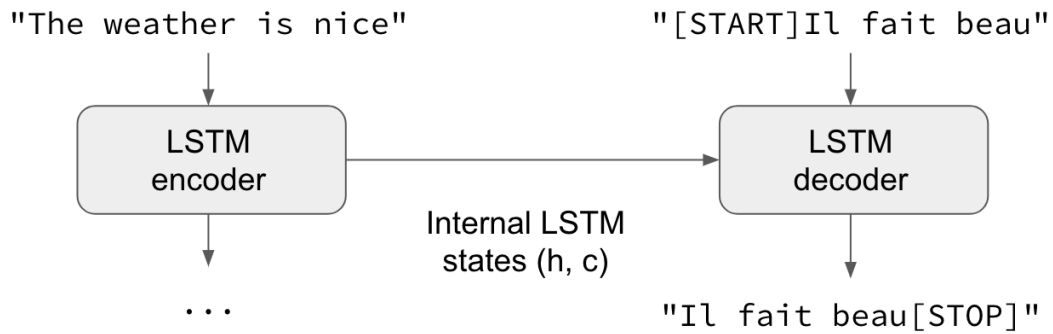


credit: https://blog.keras.io/

Sequence to Sequence Learning

The general case: canonical sequence-to-sequence
we've seen: that's when input and output sequences have arbitrary length

A RNN layer (or stack) acts as **encoder**: It processes the input sequence and returns its own internal state. Note that we discard the outputs of the encoder RNN, only recovering the state. This state will serve as the "context", or "conditioning", of the decoder in the next step.
Another RNN layer (or stack) acts as **decoder**: It is trained to predict the target sequence characters, given its previous characters and conditioned on the input sequence. The encoder uses as initial state the state vectors from the encoder.



credit: https://blog.keras.io/

Sequence to Sequence Learning

The general case: canonical sequence-to-sequence
we've seen: that's when input and output sequences have arbitrary length

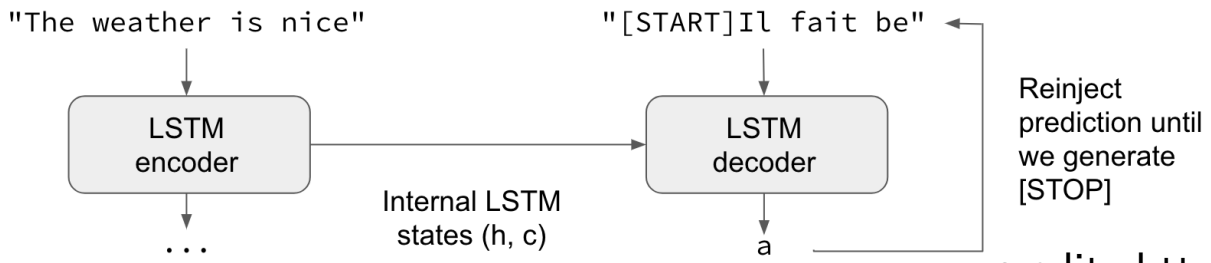
In inference mode, i.e. when we want to decode unknown input sequences, we go through a slightly different process:

Sequence to Sequence Learning

The general case: canonical sequence-to-sequence
we've seen: that's when input and output sequences have arbitrary length

In inference mode, i.e. when we want to decode unknown input sequences, we go through a slightly different process:

1) Encode the input sequence into state vectors.
2) Start with a target sequence of size 1.
3) Feed the state vectors and 1-char target sequence to the decoder to produce predictions for the next character.
4) Sample the next character using these predictions (argmax).
5) Append the sampled character to the target sequence.
6) Repeat until we generate the end-of-sequence character.



credit: https://blog.keras.io/

General Advantages and drawbacks of RNNs

Advantages of RNNs

- RNN architecture is designed to process inputs of any length.
- The weights of the hidden layers are shared across time steps.
- The internal memory of RNNs is an inherent property that is used for processing the arbitrary series of inputs which is not the case with feedforward neural networks.
- RNNs when combined with traditional CNNs gives an effective pixel neighborhood prediction.

General Advantages and drawbacks of RNNs

Advantages of RNNs

- RNN architecture is designed to process inputs of any length.
- The weights of the hidden layers are shared across time steps.
- The internal memory of RNNs is an inherent property that is used for processing the arbitrary series of inputs which is not the case with feedforward neural networks.
- RNNs when combined with traditional CNNs gives an effective pixel neighborhood prediction.

Disadvantages of RNNs

- Due to its recurrent nature, the computation becomes slow.
- Training of RNN models can be very difficult and time-consuming as compared to other Artificial Neural Networks.
- Processing very long sequences becomes very difficult if the activation functions used are ReLu or tanh.
- Prone to problems such as exploding and gradient vanishing.
- Standard RNNs are not able to keep track of long-term dependencies.

Hyperparameter Tuning

What to keep in mind when **optimizing hyperparameters for RNNs**:

- Watch out for overfitting - a neural network essentially memorizing the training data. It can be detected as the performance on training data is great but the model is useless for other predictions. For this reason a separate test set on which the network doesn't train is used.
- The larger the network, the more powerful, but it's also more prone to overfitting.
- Introducing more data is a good way to prevent overfitting.
- Train over multiple epochs (complete passes through the dataset).
- Evaluate test set performance at each epoch to know when to stop (early stopping).
- In general, stacking layers can help.
- For LSTMs, use the softsign (not softmax) activation function over tanh (it's faster and less prone to saturation).

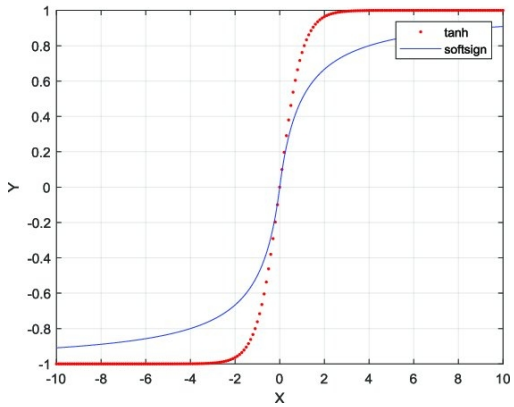


Image Captioning

Image captioning is the technique to automatically generate short descriptions (captions) of images.

Image Captioning

Image captioning is the technique to automatically generate short descriptions (captions) of images.

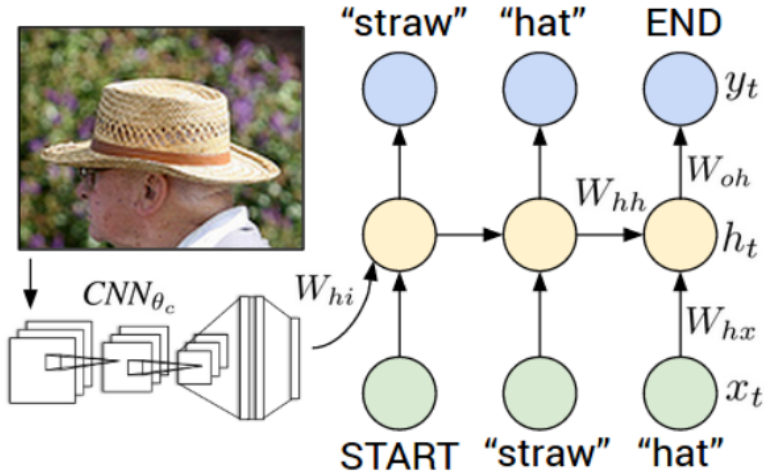


Diagram of multimodal Recurrent Neural Network generative model. The RNN takes a word, the context from previous time steps and defines a distribution over the next word in the sentence. The RNN is conditioned on the image information at the first time step.
credit: A. Karpathi & L. Fei-Fei (2014), *Deep Visual-Semantic Alignments for generating Image Descriptions*

Applications

Image Captioning

... working well:

a man is throwing a frisbee in a park

a black and white cat sitting in a bathroom sink

a parking meter is on the side of the street

Captions generated using neuraltalk2; credit: Karpathy et al.
`https://cs.stanford.edu/people/karpathy/neuraltalk2/demo.html`

Image Captioning

... gone wrong:

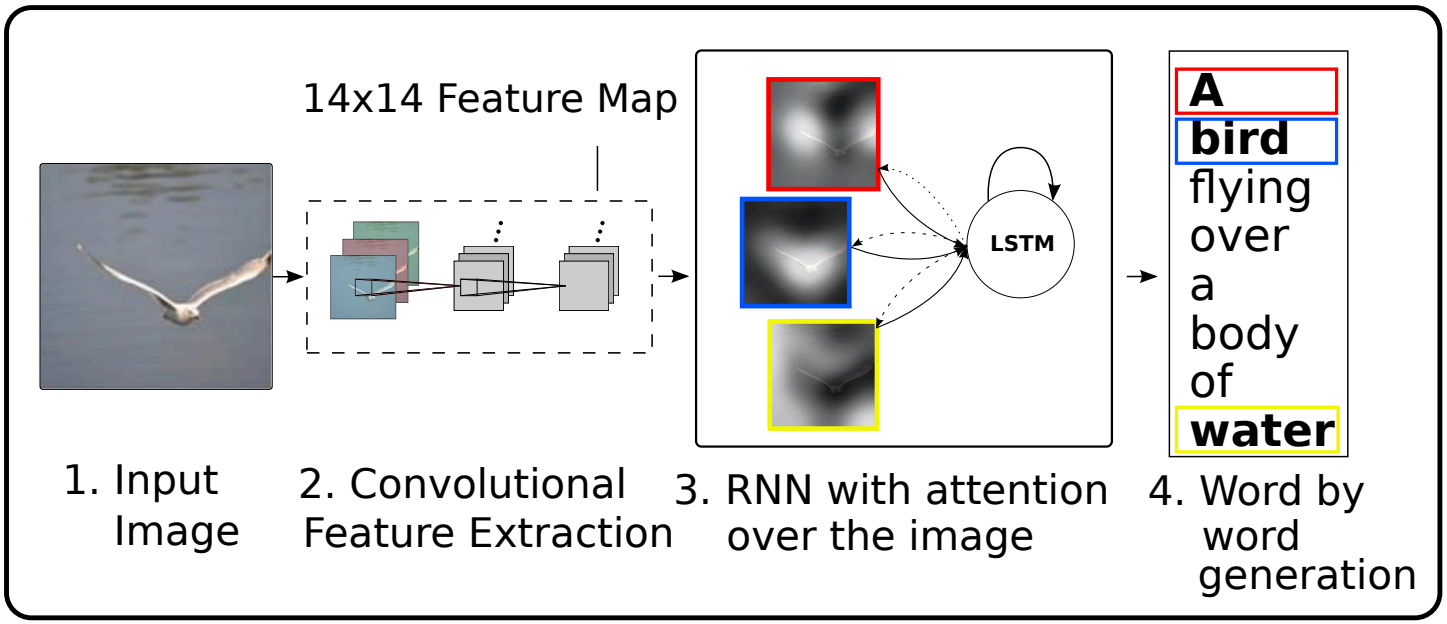
a laptop computer sitting on top of a wooden desk

a person holding a plate of food and a cup of coffee

Captions generated using neuraltalk2; credit: Karpathy et al.
`https://cs.stanford.edu/people/karpathy/neuraltalk2/demo.html`

Image Captioning with Attention

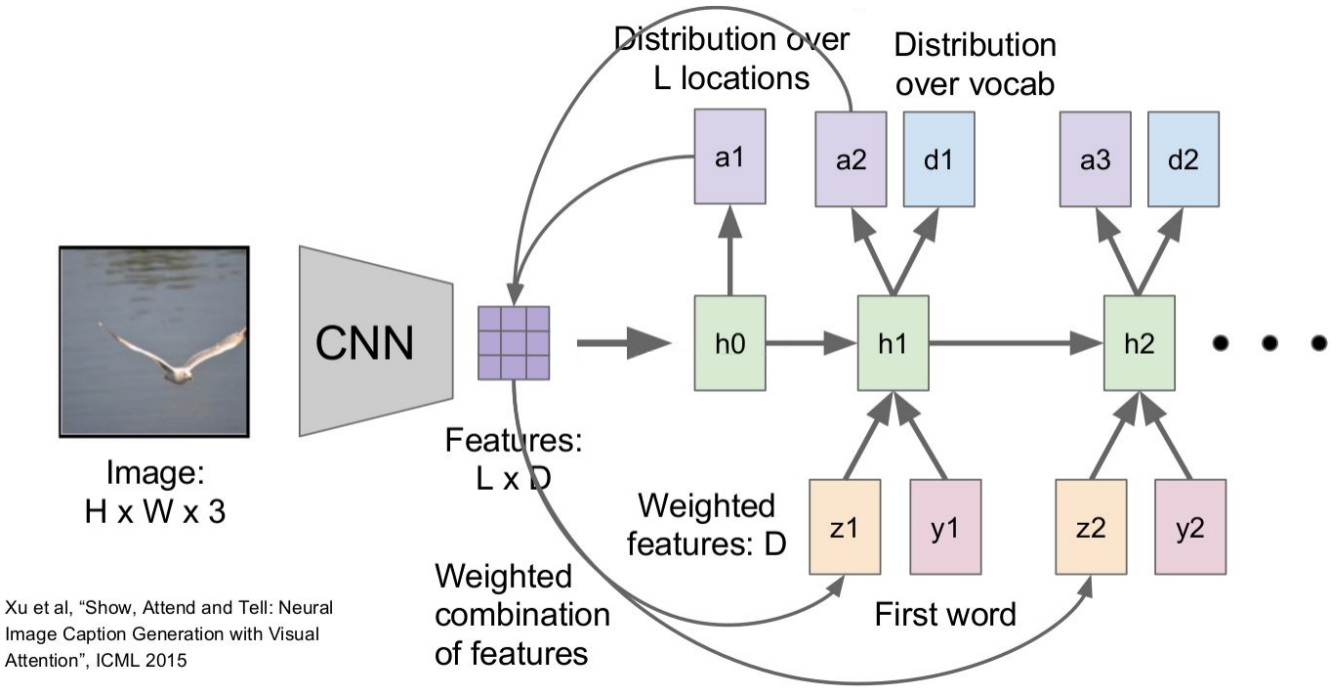
RNN focuses its attention at different spatial locations when generating each word



K. Xu, J. Ba et al, (2015), *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*

Image Captioning with Attention

RNN focuses its attention at different spatial locations when generating each word

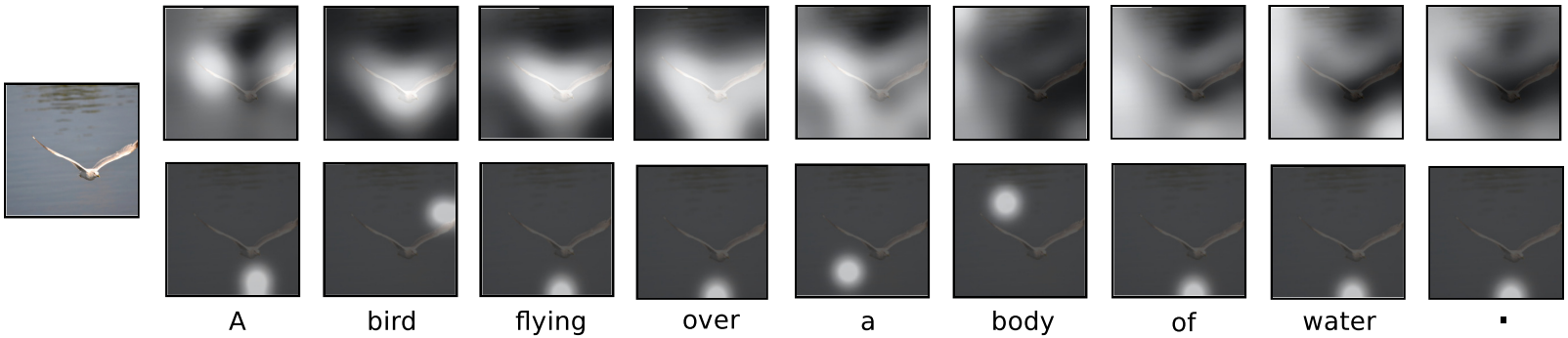


K. Xu, J. Ba et al, (2015), *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*

Applications

Image Captioning with Attention

RNN focuses its attention at different spatial locations when generating each word



Visualization of the attention for each generated word. The rough visualizations obtained by upsampling the attention weights and smoothing. (top) "soft" and (bottom) "hard" attention (note that both models generated the same captions in this example)

K. Xu, J. Ba et al, (2015), *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*

Image Captioning with Attention

... working well:

A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.

A stop sign is on a road with a mountain in the background.

A little girl sitting on a bed with a teddy bear.

A group of people sitting on a boat in the water.

A giraffe standing in a forest with trees in the background.

K. Xu, J. Ba et al, (2015), *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*

Image Captioning with Attention

... gone wrong:

A large white bird standing in a forest.

A woman holding a clock in her hand.

A man wearing a hat and a hat on a skateboard.

A person is standing on a beach with a surfboard.

A woman is sitting at a table with a large pizza.

A man is talking on his cell phone while another man watches.

K. Xu, J. Ba et al, (2015), *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*

Applications in Astronomy

There are many publications showing the successful application of RNN to astronomical data, especially time series data (e.g. variable stars, supernovae), telescope scheduling,local weather forecast (e.g. at observing sites) and others.

Applications in Astronomy

There are many publications showing the successful application of RNN to astronomical data, especially time series data (e.g. variable stars, supernovae), telescope scheduling,local weather forecast (e.g. at observing sites) and others.

We'll take here a more detailed look at this paper:

DEEP RECURRENT NEURAL NETWORKS FOR SUPERNOVAE CLASSIFICATION

Tom Charnock[1] and Adam Moss[1]

[1] *School of Physics & Astronomy*
University of Nottingham, Nottingham, NG7 2RD, England

(Dated: May 9, 2017)

ABSTRACT

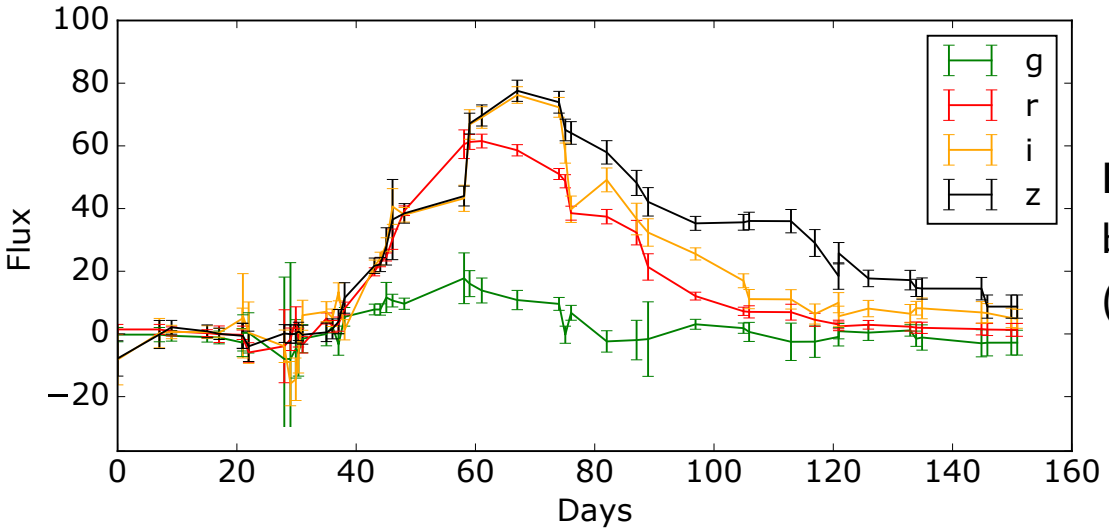
We apply deep recurrent neural networks, which are capable of learning complex sequential information, to classify supernovae[a)]. The observational time and filter fluxes are used as inputs to the network, but since the inputs are agnostic additional data such as host galaxy information can also be included. Using the Supernovae Photometric Classification Challenge (SPCC) data, we find that deep networks are capable of learning about light curves, however the performance of the network is highly sensitive to the amount of training data. For a training size of 50% of the representational SPCC dataset (around 10^4 supernovae) we obtain a type-Ia vs. non-type-Ia classification accuracy of 94.7%, an area under the Receiver Operating Characteristic curve AUC of 0.986 and a SPCC figure-of-merit $F_1 = 0.64$. When using only the data for the early-epoch challenge defined by the SPCC we achieve a classification accuracy of

Applications in Astronomy

Deep Recurrent Neural Networks for Supernovae Classification, T. Charnock & A. Moss (2017)

they use the Supernovae Photometric Classification Challenge (SPCC) data set (Kessler et al. 2010a):

A publicly released mix of 21319 simulated supernovae (SNe), with types (Ia, Ibc, and II) selected in proportion to their expected rates. The simulation was realized under realistic observing conditions (sky noise, point-spread function, and atmospheric transparency) based on years of recorded conditions at the DES site.



Example light curve in the 4 bands for SN ID 551675 (type-Ia) in the SPCC data

Applications in Astronomy

Deep Recurrent Neural Networks for Supernovae Classification, T. Charnock & A. Moss (2017)

method: applying deep recurrent neural networks (RNNs), which are capable of learning complex sequential information, to classify supernovae

Applications in Astronomy

Deep Recurrent Neural Networks for Supernovae Classification, T. Charnock & A. Moss (2017)

method: applying deep recurrent neural networks (RNNs), which are capable of learning complex sequential information, to classify supernovae

input: observational time and filter fluxes; additional data such as host galaxy information can be added:
The input vector to each sequential step consists of: time in days since the first observation; flux in each of the 4 bands; flux errors in each of the 4 bands; RA and Dec; dust extinction; and host photo-z if available.

Applications in Astronomy

Deep Recurrent Neural Networks for Supernovae Classification, T. Charnock & A. Moss (2017)

method: applying deep recurrent neural networks (RNNs), which are capable of learning complex sequential information, to classify supernovae

input: observational time and filter fluxes; additional data such as host galaxy information can be added:
The input vector to each sequential step consists of: time in days since the first observation; flux in each of the 4 bands; flux errors in each of the 4 bands; RA and Dec; dust extinction; and host photo-z if available.

findings: Performance of the RNN is highly sensitive to the amount of training data. For a training size of 50% of the representational SPCC dataset (around 10^4 supernovae) we obtain a type-Ia vs. non-type-Ia classification accuracy of 94.7 %. Applying a pre-trained model to obtain classification probabilities as a function of time shows it can give early indications of supernovae type. This method has applications for future large-scale photometric surveys (such as LSST).

Applications in Astronomy

Deep Recurrent Neural Networks for Supernovae Classification, T. Charnock & A. Moss (2017)

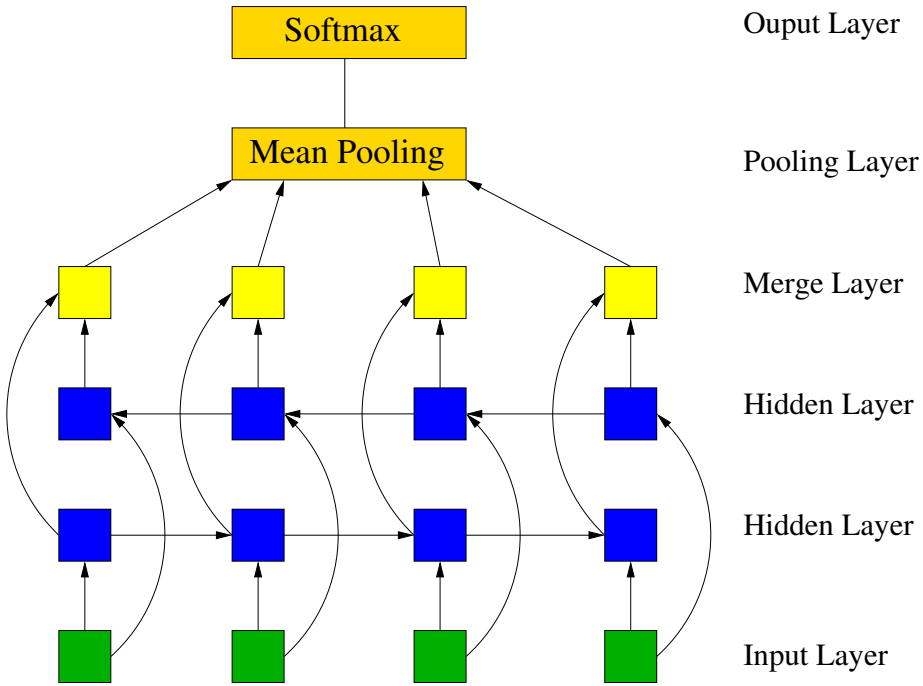


Figure 1. Bidirectional recurrent neural network for sequence classification. The input vectors at each sequential step are fed into a pair of bidirectional hidden layers, which can propagate information forwards and backwards. These are then merged to obtain a consensus view of the network, and finally a softmax layer computes classification probabilities.

Applications in Astronomy

Deep Recurrent Neural Networks for Supernovae Classification, T. Charnock & A. Moss (2017)

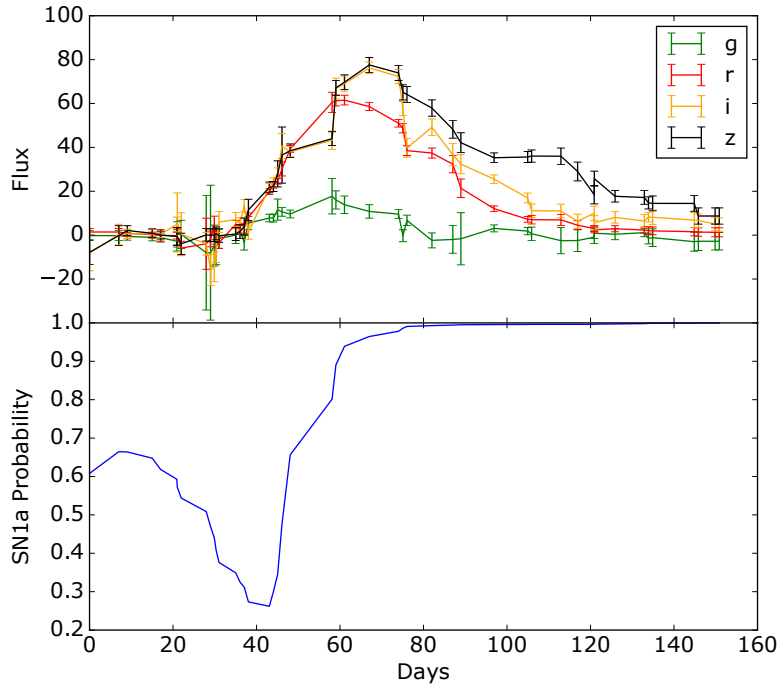


Figure 3. (Top) Example light curve in the 4 g, r, i, z bands for SN ID 551675 (a type-Ia) in the SPCC data. The data has been processed using augmentation so there is a g, r, i, z value at each sequential step.
(Bottom) Type-Ia probability as a function of time from a 2-layer LSTM model, trained with around 104 supernovae and SN 551675 excluded. The final probability gives 99.5% confidence that the supernovae is of type-Ia.

Applications in Astronomy

Deep Recurrent Neural Networks for Supernovae Classification, T. Charnock & A. Moss (2017)

conclusion:
As well as finding competitive classification performance, they have shown that with a deep RNN it is possible to give fast, early evaluation of supernovae type using pre-trained models. This is possible since the light curve can be fed to the model directly without needing any feature extraction.

Summary

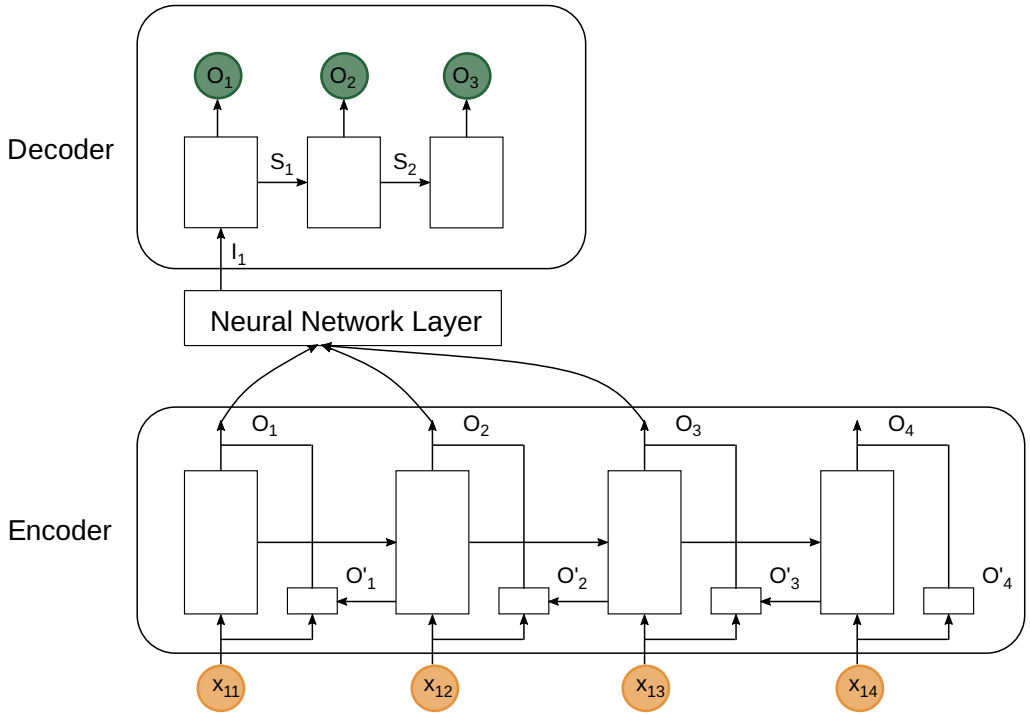
Recurrent Neural Networks, or RNNs, are a specialized class of neural networks used to process sequential data.

- RNNs model sequential data.
- RNNs implements Parameter Sharing by using same weights at every time stamp.
- One main limitation of standard RNN is that the gradient either explodes or vanishes; The network doesn't learn much from the data which is far away from the current position. To overcome this problem specialized versions of RNN are created like LSTM, GRU.
- Another limitation of standard RNN is that it processes inputs in a strict temporal order. This means current input has context of previous inputs but not the future.
- Bidirectional RNN (BRNN) duplicates the RNN processing chain so that inputs are processed in both forward and reverse time order.
- RNNs are widely used in the following domains/ applications: Machine Translation, Speech Recognition, Generating Image Descriptions, Video Tagging, Text Summarization etc.

Outlook

We have briefly looked at **encoders an decoders**: Input / output networks each consisting of LSTM or GRU cells.

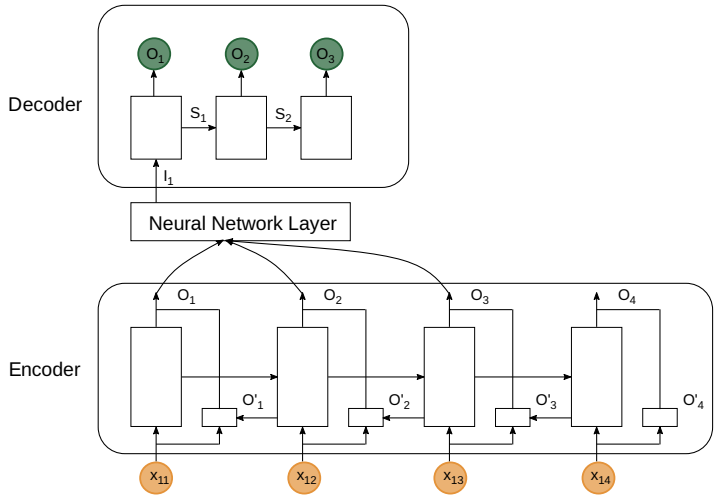
The drawback of Encoder and Decoder:
The context vector summarizes the input sequence but not all words will be valuable to include in the summary. This is overcome by using **Attention-Based models**. A neural network between encoder and decoder learns to filter relevant information - thus paying attention.



Outlook

We have briefly looked at **encoders an decoders**: Input / output networks each consisting of LSTM or GRU cells.

The drawback of Encoder and Decoder:
The context vector summarizes the input sequence but not all words will be valuable to include in the summary. This is overcome by using **Attention-Based models**. A neural network between encoder and decoder learns to filter relevant information - thus paying attention.



Next time we will look at **autoencoders**.