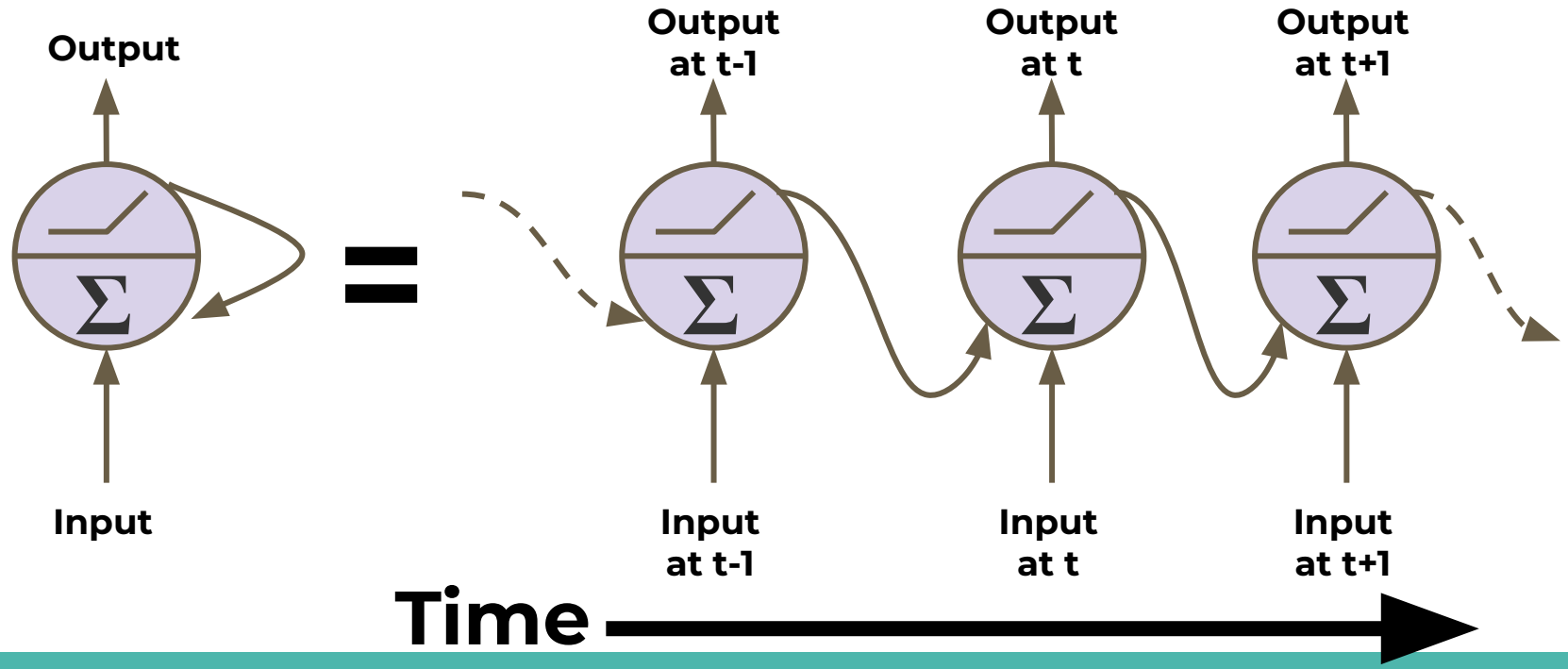
Recurrent Neural Networks

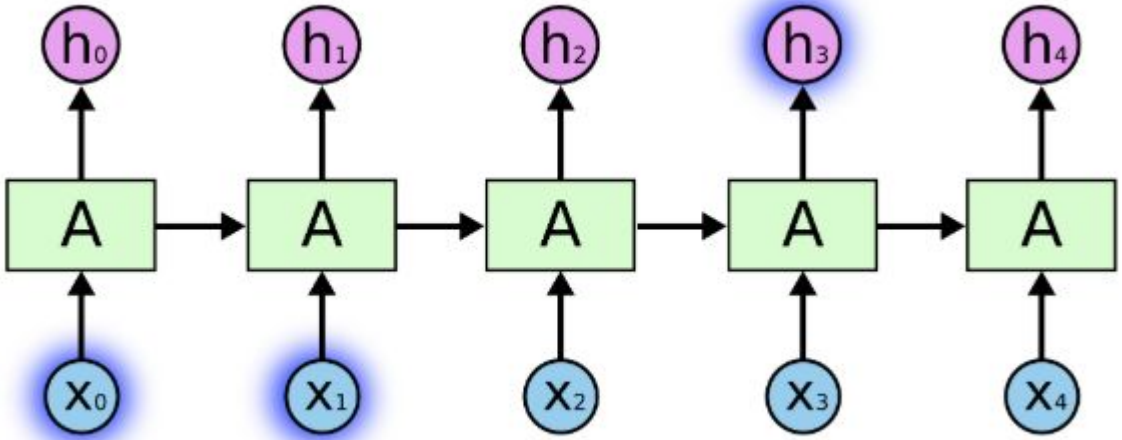
Recurrent Neuron

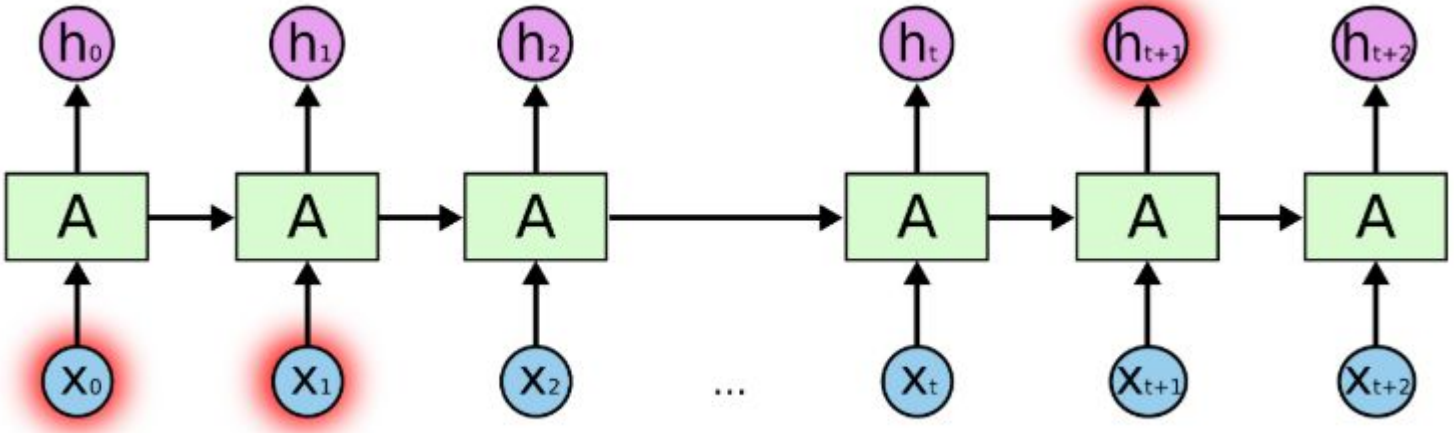
- memory cells



Applications

- applications in Computer Vision (CV) , speech recognition, language modeling, translation, image captioning
- modeling of time-dependent and sequential data tasks
- Shortcoming :- where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information.

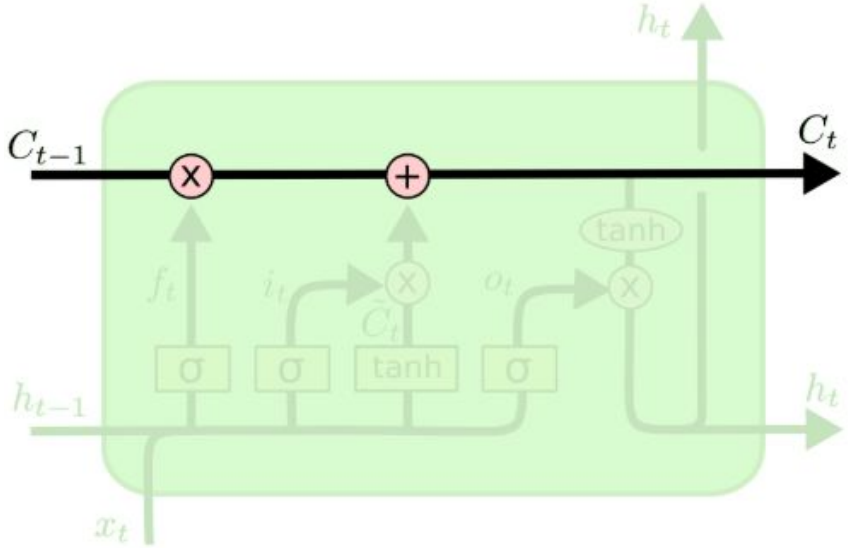

h0
h1
h2
h3
h4
A
A
A
A
A
x0
x1
x2
x3
x4


h0
h1
h2
ht
ht+1
ht+2
A
A
A
A
A
A
x0
x1
x2
...
xt
xt+1
xt+2

Long Short Term Memory networks (LSTM)

- explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior

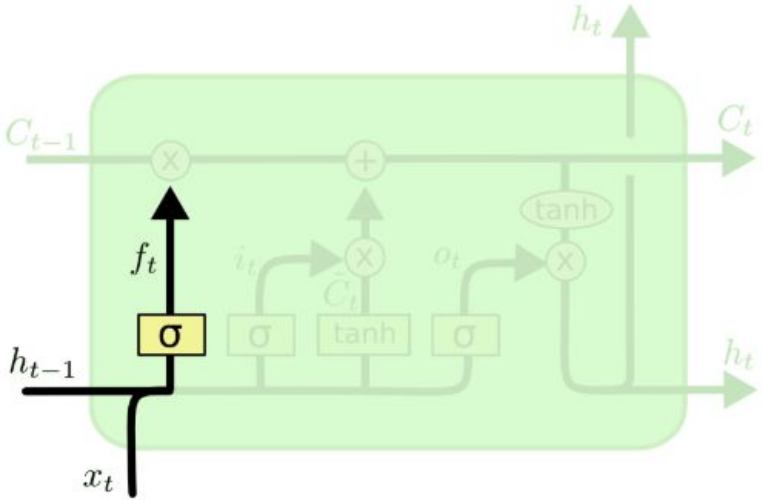
- key to LSTMs is the **cell state**



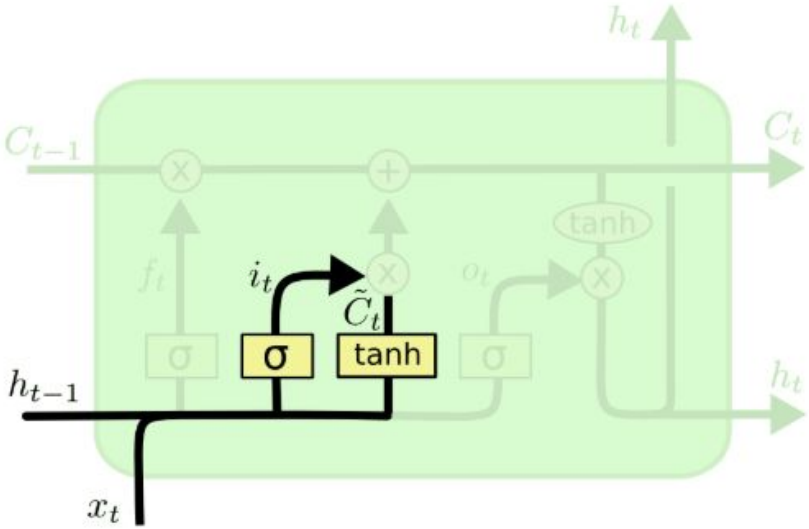
LSTM has three **gates**, to protect and control the cell state. Each gate is composed out of a sigmoid neural net layer and a pointwise multiplication operation. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates

- “forget gate layer”

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



- decide what new information to store in the cell state
- “input gate layer” : decides which values to update
- “tanh layer”: creates a vector of new candidate values, C̃ t, that could be added to the state

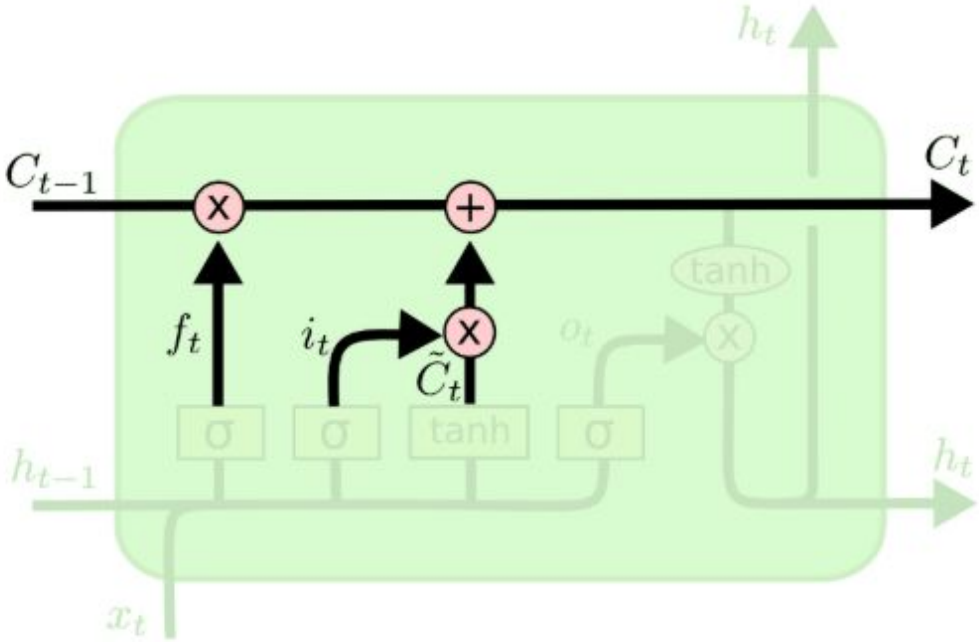


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

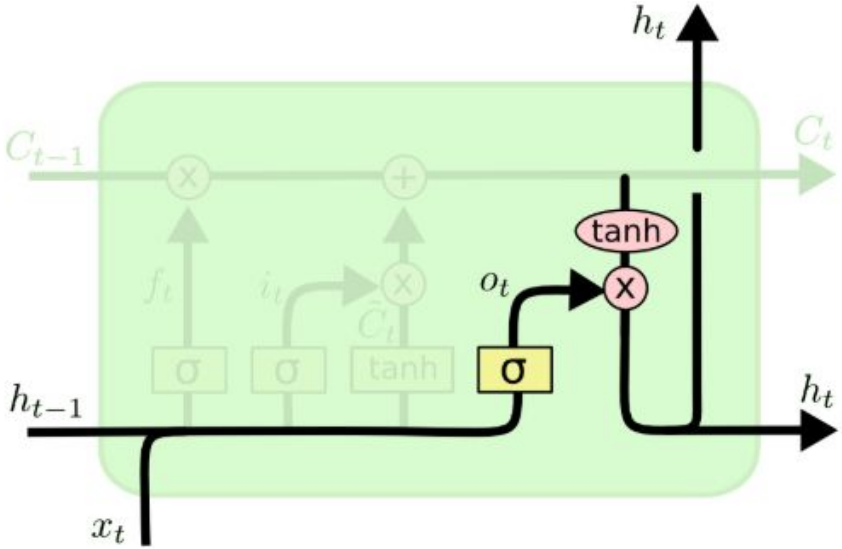
update the old cell state, Ct−1, into the new cell state Ct

- multiply the old state by ft, forgetting the things we decided to forget earlier
- add it∗C̃ t. This is the new candidate values, scaled by how much we decided to update each state value.



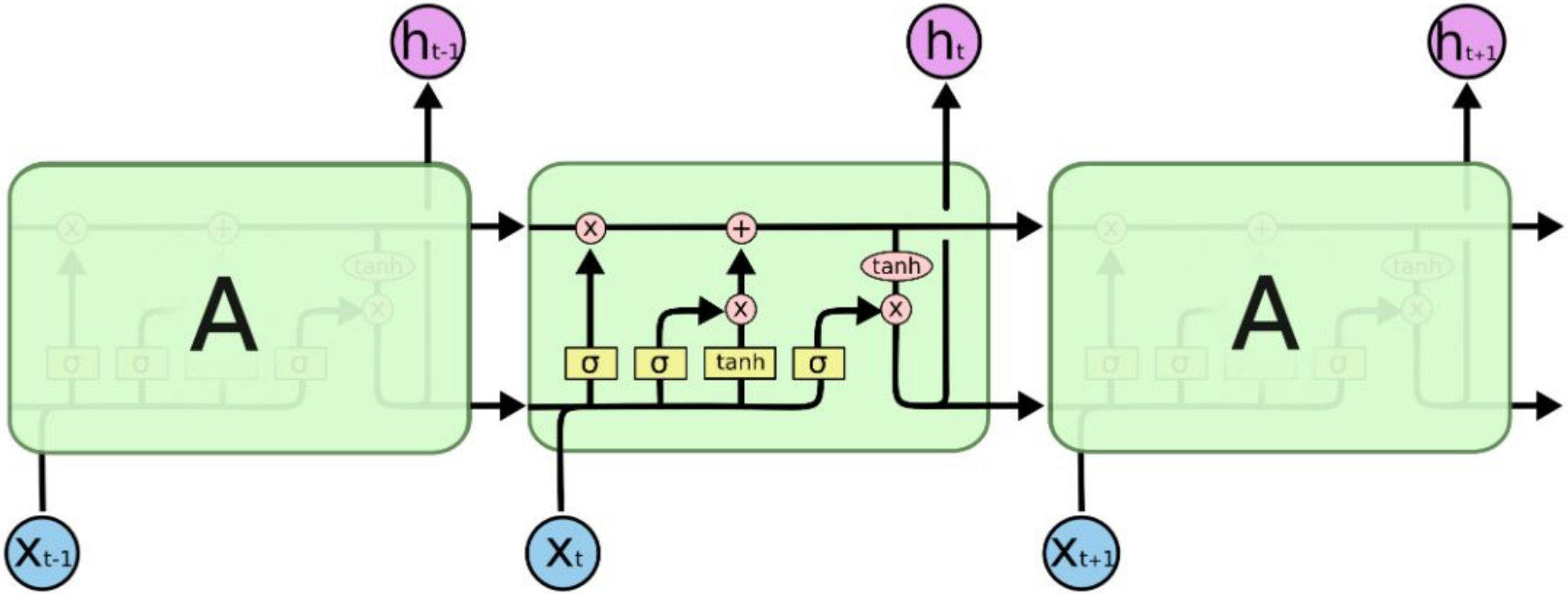
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- decide what to output
  - First, we run a sigmoid layer which decides what parts of the cell state we’re going to output.
  - we put the cell state through tanh (to push the values to be between −1 and 1)
  - multiply it by the output of the sigmoid gate, so that we only output the parts we decided to



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

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


h_{t-1}
h_t
h_{t+1}
A
A
tanh
σ
σ
tanh
σ
tanh
X_{t-1}
X_t
X_{t+1}

LSTM

Unlike other recurrent neural networks, the network's internal gates allow the model to be trained successfully using backpropagation through time, or BPTT, and avoid the vanishing gradients problem.

In the Keras deep learning library, LSTM layers can be created using the LSTM() class.

Each unit or cell within the layer has an internal cell state, often abbreviated as "*c*", and outputs a hidden state, often abbreviated as "*h*".

Example to model RNN Data

t=0	t=1	t=2	t=3	t=4
[The,	brown,	fox,	is,	quick]
[The,	red,	fox,	jumped,	high]

words_in_dataset[0] = [The, The]
words_in_dataset[1] = [brown, red]
words_in_dataset[2] = [fox,fox]
words_in_dataset[3] = [is, jumped]
words_in_dataset[4] = [quick, high]

num_batches = 5, batch_size = 2, time_steps = 5

Executing LSTM models

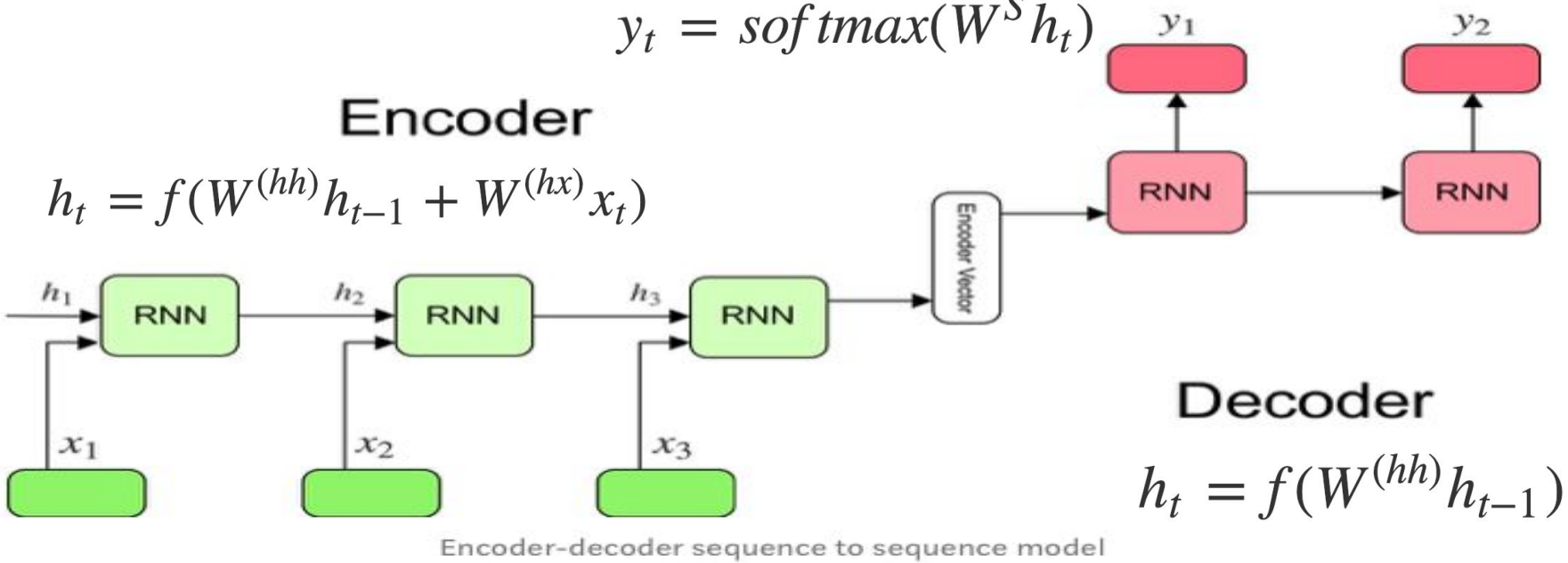
- Important functions
- Worked out examples
- Concepts

Seq2Seq Prediction using Encoder- Decoder Model

- type of sequence prediction problem that takes a sequence as input and requires a sequence prediction as output. These are called sequence-to-sequence prediction problems, or seq2seq for short.
- This architecture is comprised of two models: one for reading the input sequence and encoding it into a fixed-length vector, and a second for decoding the fixed-length vector and outputting the predicted sequence. The use of the models in concert gives the architecture its name of Encoder-Decoder LSTM designed specifically for seq2seq problems.
- The RepeatVector layer can be used like an adapter to fit the encoder and decoder parts of the network together. We can configure the RepeatVector to repeat the fixed length vector one time for each time step in the output sequence. the RepeatVector is used as an adapter to fit the fixed-sized 2D output of the encoder to the differing length and 3D input expected by the decoder.

```
model = Sequential()
model.add(LSTM(..., input_shape=(...)))
model.add(RepeatVector(...))
model.add(LSTM(..., return_sequences=True))
model.add(TimeDistributed(Dense(...)))
```

Source : https://machinelearningmastery.com/encoder-decoder-long-short-term-memory-networks/



Encoder-decoder sequence to sequence model

Encoder Vector : This is the final hidden state produced from the encoder . It acts as the initial hidden state of the decoder part of the model.

The power of this model lies in the fact that it can map sequences of different lengths to each other.

Dense()

Dense is the only actual network layer in that model and implements the operation:

output = activation(dot(input, kernel) + bias)

feeds all outputs from the previous layer to all its neurons, each neuron providing one output to the next layer

Expects 3D input : (batch_size, input_size) or (batch_size, optional,...,optional, input_size)

LSTM( )

- The input of the **LSTM** : *(batch_size, time_steps, seq_len)*
- Output is :
  a. If **return_sequence** is False : *(batch_size, units)*
  b. If **return_sequence** is True : *(batch_size, time_steps, units)*

```
model = keras.models.Sequential()

model.add(keras.layers.LSTM(units=3, input_shape=(2,10), return_sequences=False))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_58 (LSTM)               (None, 3)                 168
```

Summary()

Output shape refers to (batch_size, no of units in layer)

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential([
    Dense(32, activation='relu', input_shape=(input_size,)),
    Dense(64, activation='relu'),
    Dense(128, activation='relu'),
    Dense(num_classes, activation='softmax')
])

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 32)                4816928
_________________________________________________________________
dense_6 (Dense)              (None, 64)                2112
_________________________________________________________________
dense_7 (Dense)              (None, 128)               8320
_________________________________________________________________
dense_8 (Dense)              (None, 10)                1290
=================================================================
Total params: 4,828,650
Trainable params: 4,828,650
Non-trainable params: 0
_________________________________________________________________
```

These assertions show how the numbers of parameters of the layers depend on input, output, and each other: `output_size * (input_size + 1) == number_parameters`

```
assert 32 * (input_size + 1) == 4816928
assert 64 * (32 + 1) == 2112
assert 128 * (64 + 1) == 8320
assert num_classes * (128 + 1) == 1290
```

Difference

Model.fit( )

- for training the model with the given inputs (and corresponding training labels).

Model.evaluate( )

- evaluating the already trained model using the validation (or test) data and the corresponding labels. Returns the loss value and metrics values for the model.

model.predict( )

- for the actual prediction. It generates output predictions for the input samples.

EXAMPLE 1:

Let's look at `Dense(512, activation='relu', input_shape=(32, 32, 3))`.

Matrix multiplication:

```
(None, 32, 32, 3) * (3, 512)
```

EXPLANATION:

1. `None` is the number of pictures determined at model training, so it doesn't matter right now.
2. `(..., 32, 32, 3)` is the `input_shape` specified in the `Dense(...)`
3. `(3, 512)` comes from Keras seeing that you have the last dimension as a `(..., ..., ..., 3)` as your `input_shape`. So Keras takes that last `3` and combines that with the `512` to result in the final shape of `(3, 512)`. Taa-daa, automagic explained.

Results in:

```
(None, 32, 32, 512)
```

This is because those two `3`s cancel each other out because of the matrix multiplication.

The `Param #` comes from `(3 * 512) + 512 = 2048` as pointed out by grovina's answer. This is because of this equation:

`input * weights + bias`

1. `input` would be the `3` (aka number of params per neuron)
2. `weights` would be the `512` (aka number of neurons)
3. `bias` would be the `512` (aka one bias per neuron)

Stateless vs Stateful LSTM

- Stateless configuration is the default, LSTM cell memory is reset every batch.
  - This makes batch size a very important consideration. Stateless works best when the the sequences you're learning aren't dependent on one another.
  - Sentence-level prediction of a next word might be a good example of when to use stateless.

- The stateful configuration resets LSTM cell memory every epoch.
  - This configuration is most commonly used when each sequence in the training set depends on the sequence
  - In so doing, we must explicitly specify the batch size as a dimension on the input shape. This also means that when we evaluate the network or make predictions, we must also specify and adhere to this same batch size.
  - Set shuffle= False, to not shuffle the input and preserving the sequence of the input data
- PS：The LSTM networks are stateful. They should be able to learn the whole alphabet sequence, but by default the Keras implementation resets the network state after each training batch.

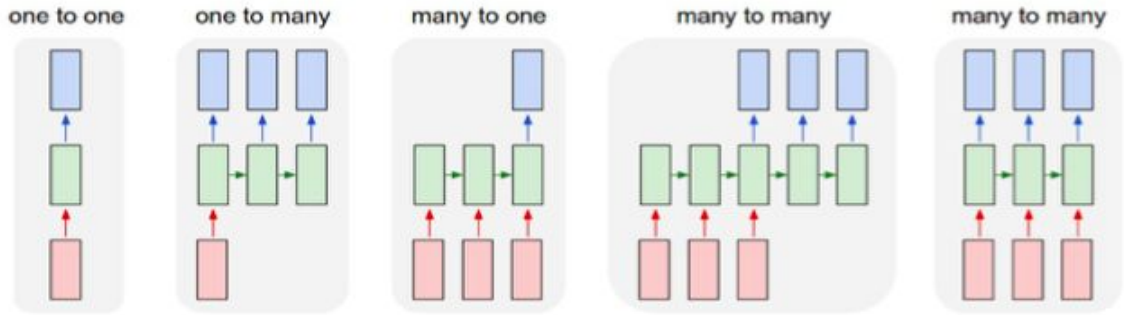
TimeDistributedDense Layer

- The input must be (at least) 3D.
- The output will be 3D. This means that if your TimeDistributed wrapped Dense layer is your output layer and you are predicting a sequence, you will need to resize your y array into a 3D vector.
- TimeDistributedDense applies a same Dense (fully-connected) operation to every timestep of a 3D tensor.
    - If you want to model this by Keras, you just need to used a TimeDistributedDense after a RNN or LSTM layer(with return_sequence=True) to make the cost function is calculated on all time-step output.
        - The output shape is (None, timesteps, LSTM_units)
    - If you don't use TimeDistributedDense and set the return_sequence of RNN=False, then the cost is calculated on the last time-step output and you could only get the last bN.

EdwardRaff commented on Nov 19, 2015 Contributor

@joetigger I think placebokkk did answer your question, but perhaps another way of saying the same thing will help. (And **@fchollet** please correct me if this wrong)

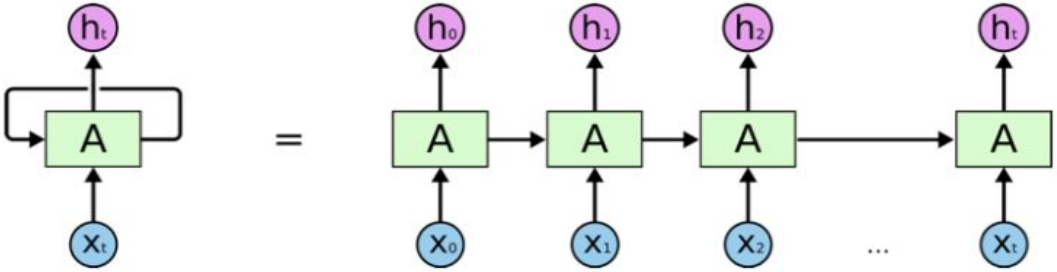
RNNs are capable of a number of different types of input / output combinations, as seen below



The `TimeDistributedDense` layer allows you to build models that do the *one-to-many* and *many-to-many* architectures. This is because the output function for each of the "many" outputs must be the same function applied to each timestep. The `TimeDistributedDense` layers allows you to apply that Dense function across every output over time. This is important because it needs to be the *same* dense function applied at every time step.

If you didn't not use this, you would only have one final output - and so you use a normal dense layer. This means you are doing either a *one-to-one* or a *many-to-one* network, since there will only be one dense layer for the output.

Let's say you have time-series data with N rows and 700 columns which you want to feed to a `SimpleRNN(200, return_sequence=True)` layer in Keras. Before you feed that to the RNN, you need to reshape the previous data to a 3D tensor. So it becomes a $N \times 700 \times 1$.



The image is taken from https://colah.github.io/posts/2015-08-Understanding-LSTMs

In RNN, your columns (the "700 columns") is the timesteps of RNN. Your data is processed from $t = 1 \; to \; 700$. After feeding the data to the RNN, now it have 700 outputs which are h_1 to h_{700}, not h_1 to h_{200}. Remember that now the shape of your data is $N \times 700 \times 200$ which is **samples (the rows) x timesteps (the columns) x channels**.

And then, when you apply a `TimeDistributedDense`, you're applying a `Dense` layer on each timestep, which means you're applying a `Dense` layer on each h_1, h_2,...,h_t respectively. Which means: actually you're applying the fully-connected operation on each of its channels (the "200" one) respectively, from h_1 to h_{700}. The 1st "$1 \times 1 \times 200$" until the 700th "$1 \times 1 \times 200$".

Why are we doing this? Because you don't want to flatten the RNN output.

Why not flattening the RNN output? Because you want to keep each timestep values separate.

Why keep each timestep values separate? Because:

- you're only want to interacting the values between its own timestep
- you don't want to have a random interaction between different timesteps and channels.