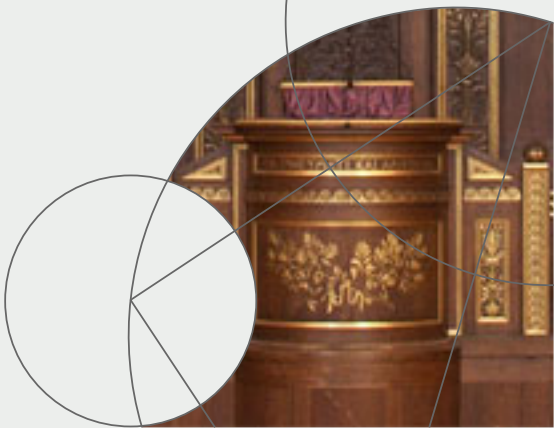
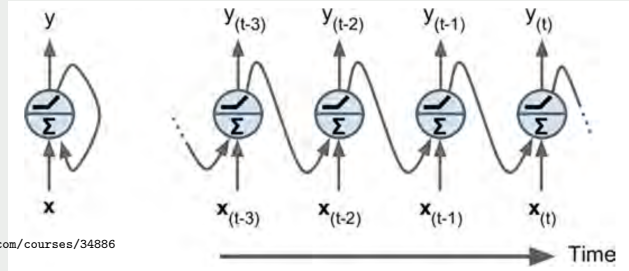
Deep Learning: Recurrent Neural Network (RNN)

Rensheng Wang,
https://sit.instructure.com/courses/34886
November 16, 2019



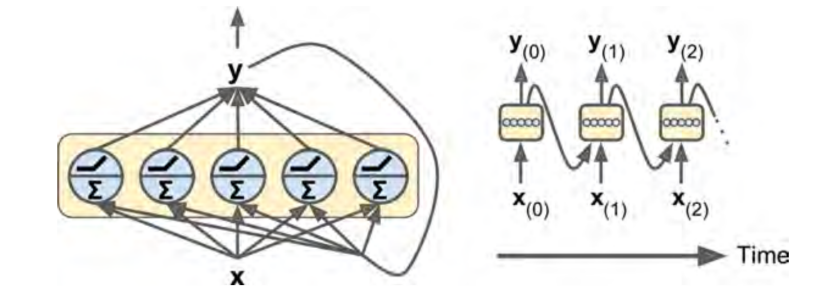
Recurrent Neurons

- Up to now we have mostly looked at feedforward neural networks, where the activations flow only in one direction, from the input layer to the output layer.
- A recurrent neural network (RNN) looks very much like a feedforward neural network, except it also has connections pointing backward.
- The simplest possible RNN, composed of just one neuron receiving inputs, producing an output, and sending that output back to itself.
- At each time step t (also called a frame), this recurrent neuron receives inputs $x_{(t)}$ as well as its own output from the previous time step, $y_{(t-1)}$. We can represent this tiny network against the time axis, This is called unrolling the network through time.



Recurrent Neurons

- To create a layer of recurrent neurons, at each time step t, every neuron receives both the input vector $\mathbf{x}_{(t)}$ and the output vector from the previous time step $\mathbf{y}_{(t-1)}$, as shown below:



- Note that both the inputs and outputs are vectors now (when there was just a single neuron, the output was a scalar).
- Each recurrent neuron has two sets of weights: one for the inputs $\mathbf{x}_{(t)}$ and the other for the outputs of the previous time step, $\mathbf{y}_{(t-1)}$. Lets call these weight vectors $\mathbf{w}_x$ and $\mathbf{w}_y$.

Recurrent Neurons

- The output of a single recurrent neuron can be computed as

$$\mathbf{y}(t) = \phi(\mathbf{x}_t^T \cdot \mathbf{w}_x + \mathbf{y}_{t-1}^T \cdot \mathbf{w}_y + b)$$

  where b is the bias term and $\phi(\cdot)$ is the activation function, e.g., ReLU.

- Just like for feedforward neural networks, we can compute a whole layers output in one shot for a whole mini-batch using a vectorized form of the previous equation

$$\begin{aligned}\mathbf{Y}(t) =& \phi(\mathbf{X}_t \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b} \\ =& \phi\left(\begin{bmatrix}\mathbf{X}_t & \mathbf{Y}_{(t-1)}\end{bmatrix} \cdot \mathbf{W} + \mathbf{b}\right)\end{aligned}$$

  where $\mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}$

Long Short-Time Memory (LSTM) Cell

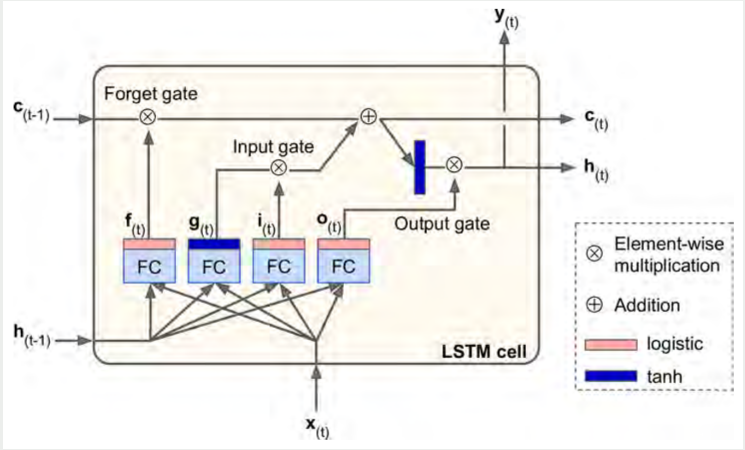
- The Long Short-Term Memory (LSTM) cell was proposed in 1997 by Sepp Hochreiter and Jurgen Schmidhuber, and it was gradually improved over the years by many other researchers.
- If you consider the LSTM cell as a black box, it can be used very much like a basic cell, except it will perform much better; training will converge faster and it will detect long-term dependencies in the data.
- In TensorFlow, you can simply use a `BasicLSTM` Cell instead of a `BasicRNNCell`:

$$\texttt{lstm\_cell} = \texttt{tf.contrib.rnn.BasicLSTMCell}(\texttt{num\_units} = \texttt{n\_neurons})$$

- LSTM cells manage two state vectors, and for performance reasons they are kept separate by default. You can change this default behavior by setting `state_is_tuple` = `False` when creating the `BasicLSTMCell`.

LSTM Cell

- The architecture of a basic LSTM cell is shown below:



- the LSTM cell looks exactly like a regular cell, except that its state is split in two vectors: $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$ (c stands for cell).
- You can think of $\mathbf{h}_{(t)}$ as the short-term state and $\mathbf{c}_{(t)}$ as the long-term state.

tanh is Rescaled Logistic **sigmoid** Function

- The logistic `sigmoid` function, a.k.a. the inverse logit function, is

$$g(x) = \frac{e^x}{1 + e^x}$$

- Its outputs range from 0 to 1, and are often interpreted as probabilities (in, say, logistic regression).
- The **tanh** function, a.k.a. hyperbolic tangent function, is a rescaling of the logistic sigmoid, such that its outputs range from -1 to 1.

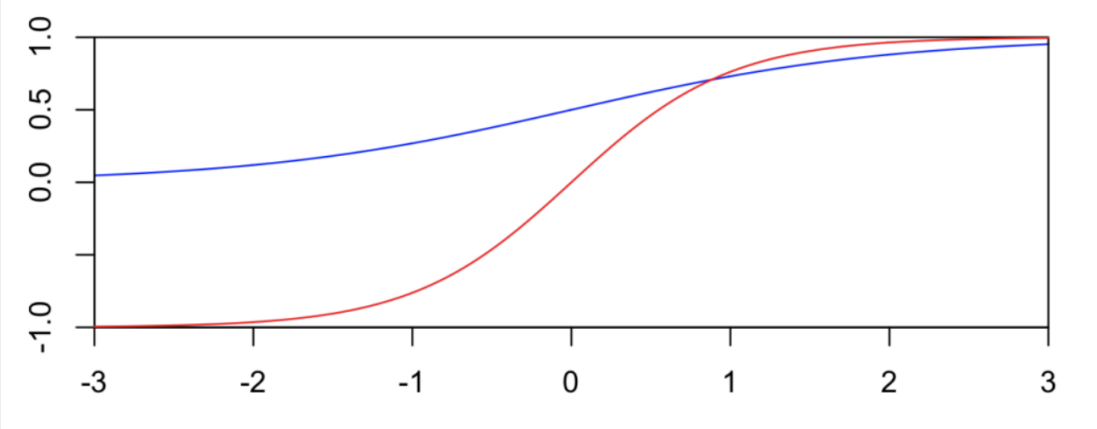
$$\tanh(x) = 2g(2x) - 1$$

- Its easy to show the above leads to the standard definition

$$\tanh = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} = \frac{2e^{2x}}{1 + e^{2x}} - 1$$

tanh is Rescaled Logistic **sigmoid** Function

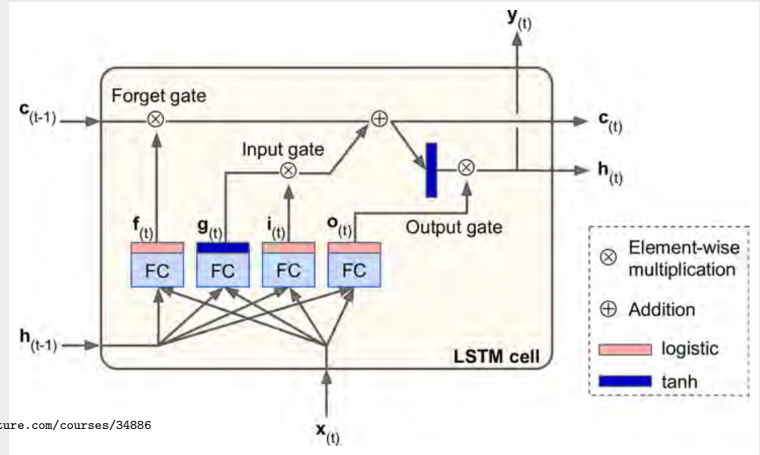
- The two functions are plotted below. Blue is the logistic function, and red is tanh.



☞ $\tanh \sim [-1, 1]$: sigmoid $\sim [0, 1]$

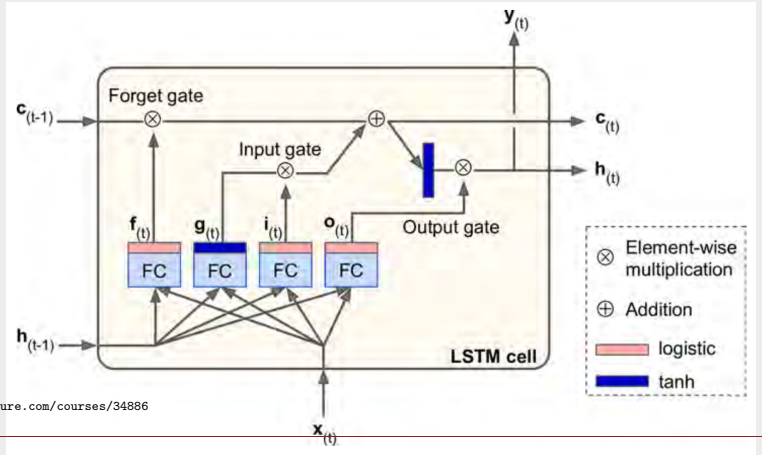
LSTM Cell

- The key idea of LSTM cell is that the network can learn what to store in the long-term state, what to throw away, and what to read from it.
- As the long-term state $\mathbf{c}_{(t-1)}$ traverses the network from left to right, you can see that it first goes through a *forget gate*, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*).



LSTM Cell

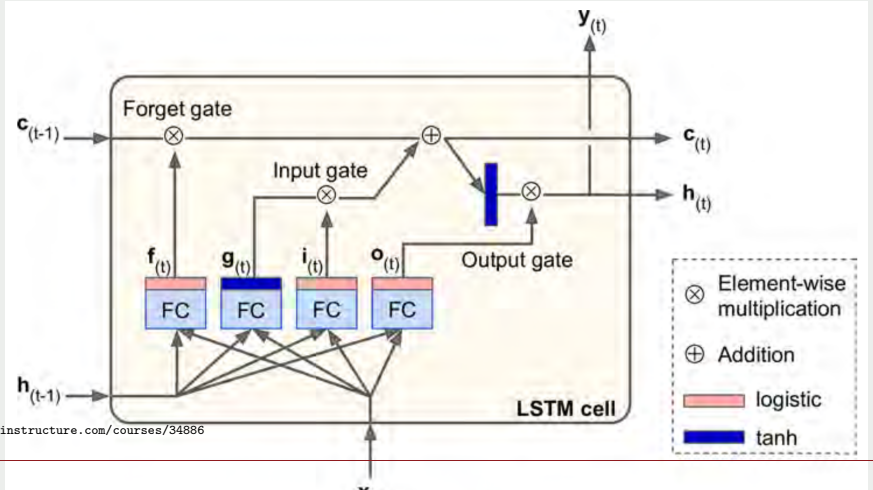
- The result $\mathbf{c}(t)$ is sent straight out, without any further transformation. So, at each time step, some memories are dropped and some memories are added.
- Moreover, after the addition operation, the long-term state is copied and passed through the **tanh** function, and then the result is filtered by the output gate.
- This produces the short-term state $\mathbf{h}_{(t)}$ (which is equal to the cells output for this time step $\mathbf{y}_{(t)}$).



LSTM Cell

Let us look at where new memories come from and how the gates work:

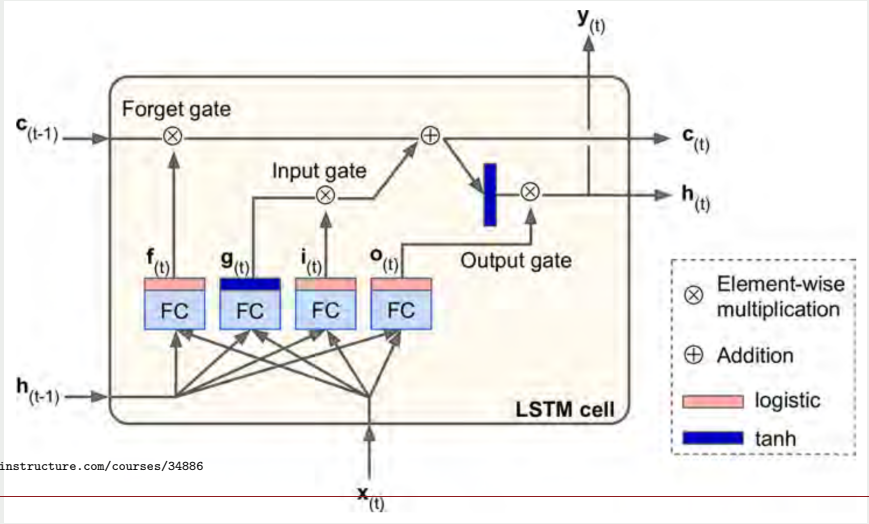
- The main layer is the one that outputs $\boldsymbol{g}_{(t)}$. It has the usual role of analyzing the current inputs $\mathbf{x}_{(t)}$ and the previous (short-term) state $\mathbf{h}_{(t-1)}$. In a basic cell, there is nothing else than this layer, and its output goes straight out to $\mathbf{y}_{(t)}$ and $\mathbf{h}_{(t)}$. In contrast, in an LSTM cell this layers output does not go straight out, but instead it is partially stored in the long-term state.



LSTM Cell

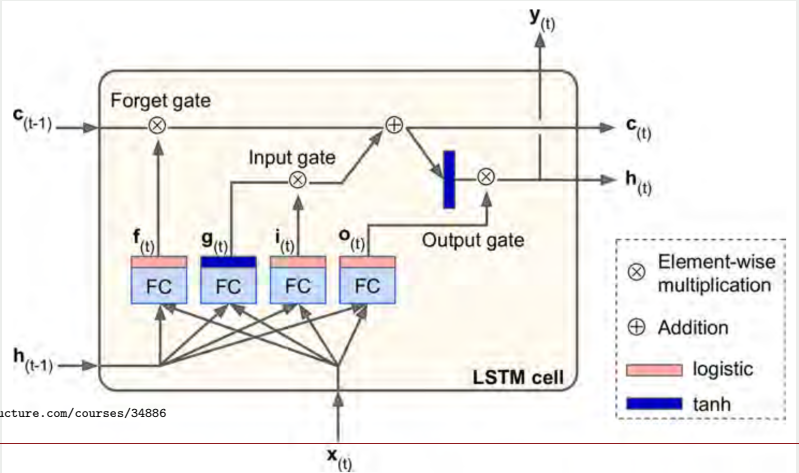
Let us look at where new memories come from and how the gates work:

- The three other layers are gate controllers. Since they use the logistic activation function, their outputs range from 0 to 1. As you can see, their outputs are fed to element-wise multiplication operations, so if they output 0s, they close the gate, and if they output 1s, they open it.



LSTM Cell

- The forget gate ($\mathbf{f}_{(t)}$) controls which parts of the long-term state should be erased.
- The input gate ($\mathbf{i}_{(t)}$) controls which parts of $\boldsymbol{g}_{(t)}$ should be added to the long-term state (this is why we said it was only partially stored).
- Finally, the output gate ($\mathbf{o}_{(t)}$) controls which parts of the long-term state should be read and output at this time step (both to $\mathbf{h}_{(t)}$ and $\mathbf{y}_{(t)}$).



LSTM Cell

- In short, an LSTM cell can learn to recognize an important input
- (thats the role of the input gate), store it in the long-term state,
- learn to preserve it for as long as it is needed
- (thats the role of the forget gate),
- and learn to extract it whenever it is needed.
- This explains why they have been amazingly successful at capturing long-term pat terns in time series, long texts, audio recordings, and more.

LSTM Computations

- Summarizes how to compute the cells long-term state, its short-term state, and its output at each time step for a single instance

$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\
\boldsymbol{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{w}_{hg}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \boldsymbol{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)})
\end{aligned}
$$

Peephole Connections

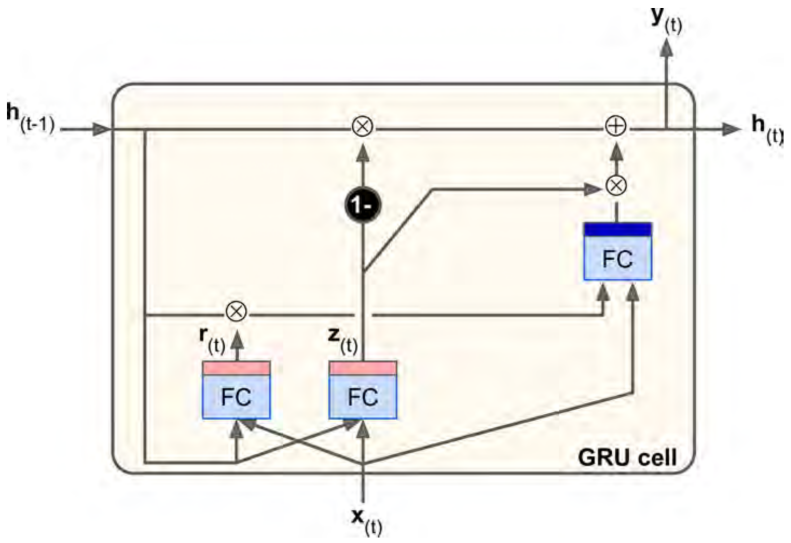
- In a basic LSTM cell, the gate controllers can look only at the input $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$.
- It may be a good idea to give them a bit more context by letting them peek at the long-term state as well. This idea was proposed by Felix Gers and Jurgen Schmidhuber in 2000.
- They proposed an LSTM variant with extra connections called peephole connections:

  The previous long-term state $\mathbf{c}_{(t-1)}$ is added as an input to the controllers of the forget gate and the input gate, and the current long- term state $\mathbf{c}_{(t)}$ is added as input to the controller of the output gate.
- To implement peephole connections in TensorFlow, you must use the `LSTMCell` instead of the `BasicLSTMCell` and set `use_peepholes=True`:

  $\texttt{lstm\_cell} = \texttt{tf.contrib.rnn.LSTMCell}(\texttt{num\_units} = \texttt{n\_neurons}, \texttt{use\_peepholes} = \texttt{True})$

GRU Cell

- The Gated Recurrent Unit (GRU) cell was proposed by Kyunghyun Cho et al. in a 2014 paper that also introduced the EncoderDecoder network we mentioned earlier.



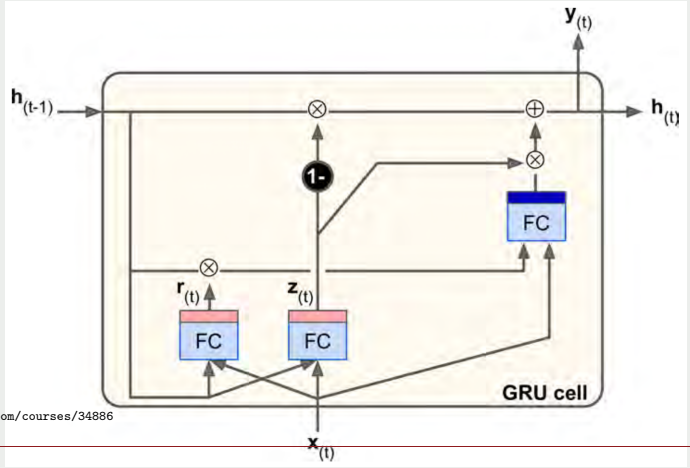
GRU Cell

- The GRU cell is a simplified version of the LSTM cell, and it seems to perform just as well (which explains its growing popularity).

  `gru_cell = tf.contrib.rnn.GRUCell(num_units=n_neurons)`

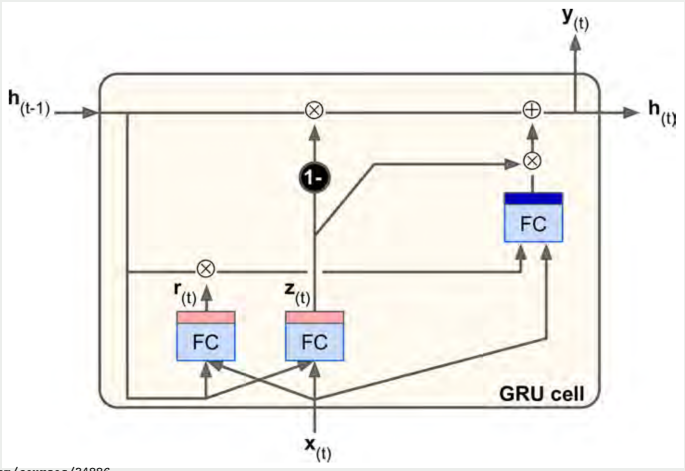
- The main simplifications are:

☞ Both state vectors are merged into a single vector $\mathbf{h}_{(t)}$.



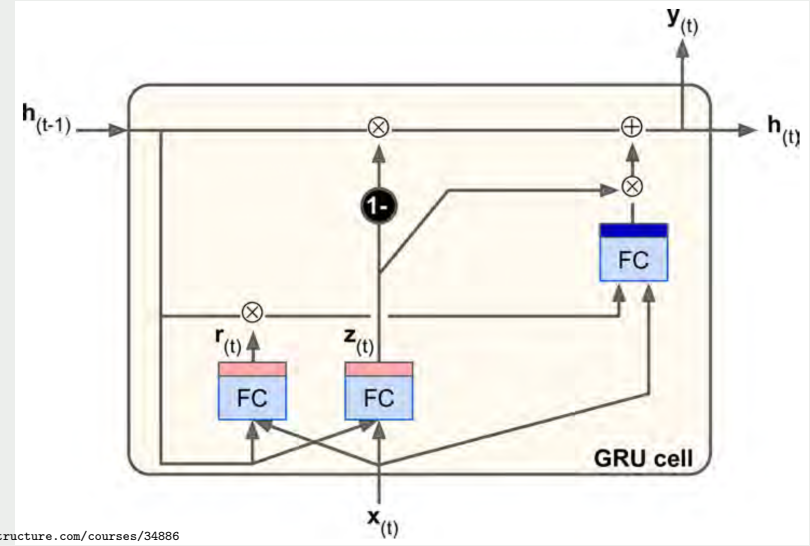
GRU Cell

☞ A single gate controller controls both the forget gate and the input gate. If the gate controller outputs a 1, the input gate is open and the forget gate is closed. If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself.



GRU Cell

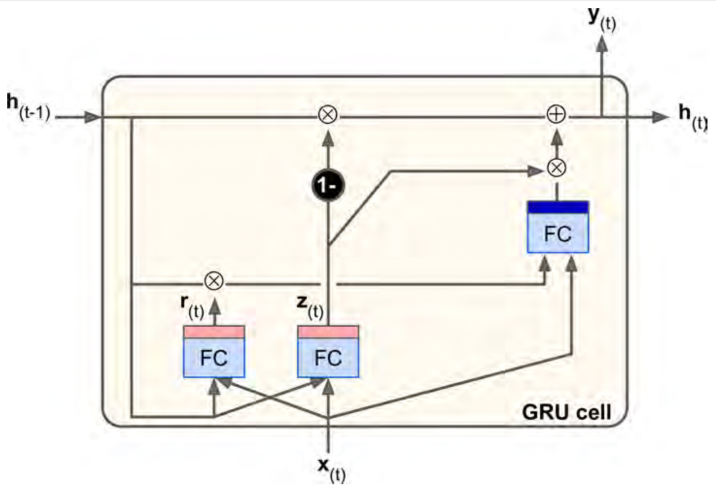
- There is no output gate; the full state vector is output at every time step. However, there is a new gate controller that controls which part of the previous state will be shown to the main layer.



GRU Cell States Computation

$$\mathbf{z}_{(t)} = \sigma\left(\mathbf{W}_{xz}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \cdot \mathbf{h}_{(t-1)}\right) \qquad \boldsymbol{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \cdot \left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right)\right)$$

$$\mathbf{r}_{(t)} = \sigma\left(\mathbf{W}_{xr}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \cdot \mathbf{h}_{(t-1)}\right) \qquad \mathbf{h}_{(t)} = (1 - \mathbf{z}_{(t)}) \otimes \tanh\left(\mathbf{W}_{xg}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{z}_{(t)} \otimes \boldsymbol{g}_{(t)}\right)$$



Natural Language Processing (NLP)

- LSTM or GRU cells are one of the main reasons behind the success of RNNs in recent years, in particular for applications in natural language processing (NLP).
- Most of the state-of-the-art NLP applications, such as machine translation, automatic summarization, parsing, sentiment analysis, and more, are now based (at least in part) on RNNs.
- There are two major components with RNN In NLP applications:
  - ☞ Word Embeddings
  - ☞ Sequence-to-Sequence

Word Embeddings

- To find a way for word representation, one option could be to represent each word using a one-hot vector. Suppose your vocabulary contains 50,000 words, then the nth word would be represented as a 50,000-dimensional vector, full of 0s except for a 1 at the n-th position.
- However, with such a large vocabulary, this sparse representation would not be efficient at all.
- Ideally, you want similar words to have similar representations, making it easy for the model to generalize what it learns about a word to all similar words.

  For example, if the model is told that I drink milk is a valid sentence, and if it knows that "milk" is close to "water" but far from "shoes", then it will know that "I drink water" is probably a valid sentence as well, while "I drink shoes" is probably not.

Word Embeddings

- The most common solution is to represent each word in the vocabulary using a fairly small and dense vector (e.g., 150 dimensions), called an embedding, and just let the neural network learn a good embedding for each word during training.
- At the beginning of training, embeddings are simply chosen randomly, but during training, back propagation automatically moves the embeddings around in a way that helps the neural network perform its task.
- Typically this means that similar words will gradually cluster close to one another, and even end up organized in a rather meaningful way.

  For example, embeddings may end up placed along various axes that represent gender, singular/plural, adjective/noun, and so on. The result can be truly amazing.

Word Embeddings

- In TensorFlow, you first need to create the variable representing the embeddings for every word in your vocabulary (initialized randomly):

```
vocabulary_size = 50000

embedding_size = 150

embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
embedding_size], -1.0, 1.0))
```

- Now suppose you want to feed the sentence "I drink milk" to your neural network. You should first preprocess the sentence and break it into a list of known words.
- For example you may remove unnecessary characters, replace unknown words by a pre defined token word such as "[UNK]", replace numerical values by "[NUM]", replace URLs by "[URL]", and so on.
- Once you have a list of known words, you can look up each words integer identifier (from 0 to 49999) in a dictionary, for example [72, 3335, 288].

Word Embeddings

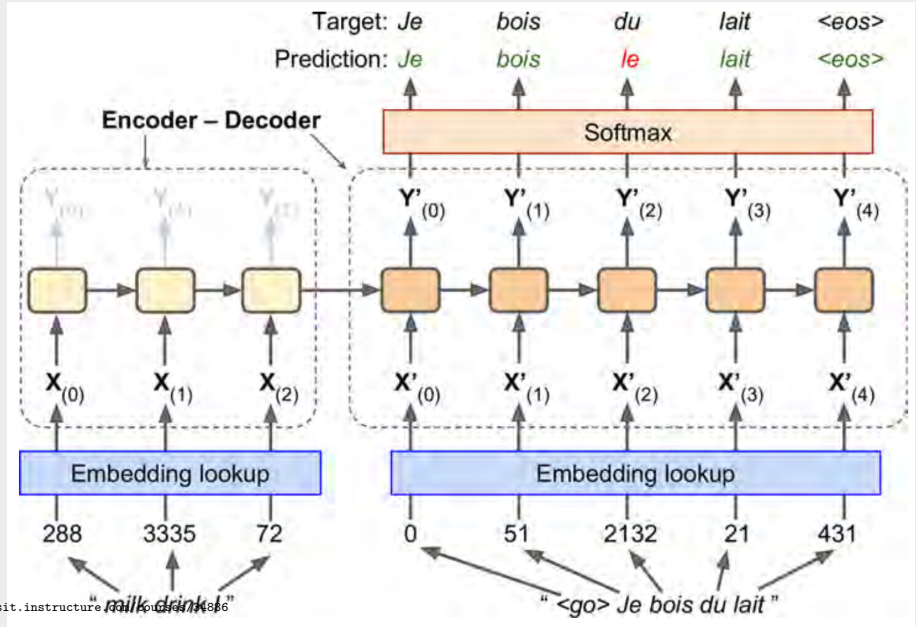
- Once you have a list of known words, you are ready to feed these word identifiers to TensorFlow using a placeholder, and apply the `embedding_lookup()` function to get the corresponding embeddings:

```
train_inputs = tf.placeholder(tf.int32, shape=[None]) # from ids ...
embed = tf.nn.embedding_lookup(embeddings, train_inputs) # ...to embeddings
```

- Once your model has learned good word embeddings, they can actually be reused fairly efficiently in any NLP application.
- In fact, instead of training your own word embeddings, you may want to download pre-trained word embeddings.

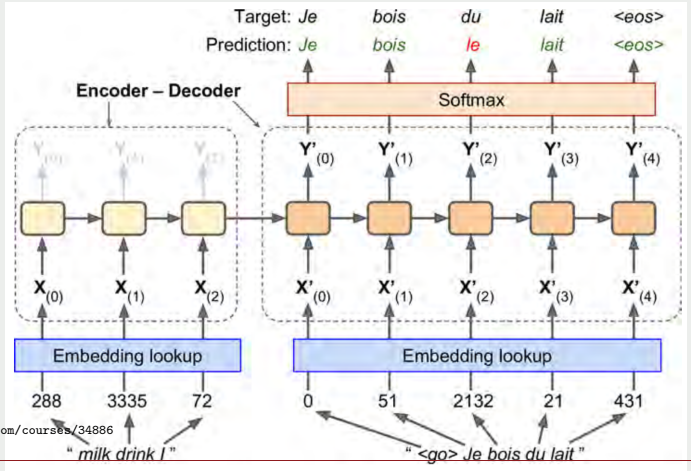
An EncoderDecoder Network for Machine Translation

- A simple machine translation model



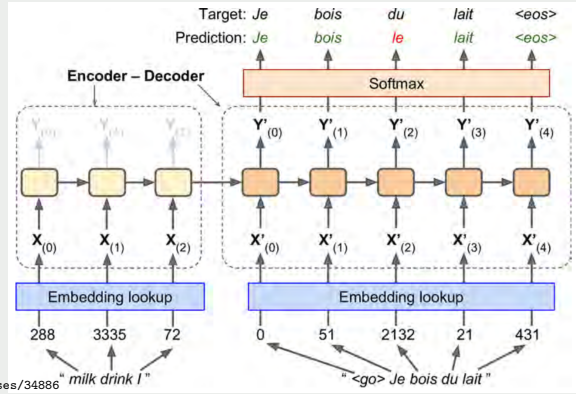
An EncoderDecoder Network for Machine Translation

- English sentences are fed to encoder, and decoder outputs French translations.
- Note that the French translations are also used as inputs to the decoder, but pushed back by one step.
- In other words, the decoder is given as input the word that it should have output at the previous step (regardless of what it actually output).



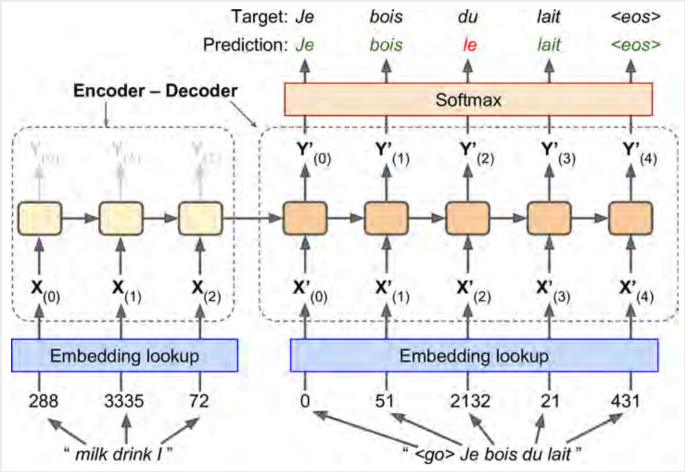
An EncoderDecoder Network for Machine Translation

- For the very first word, it is given a token that represents the beginning of the sentence (e.g., "<go>"). The decoder is expected to end the sentence with an end-of- sequence (EOS) token (e.g., "<eos>")
- Note that the English sentences are reversed before they are fed to the encoder. For example "I drink milk" is reversed to "milk drink I". This ensures that the beginning of the English sentence will be fed last to the encoder, which is useful because thats generally the first thing that the decoder needs to translate.



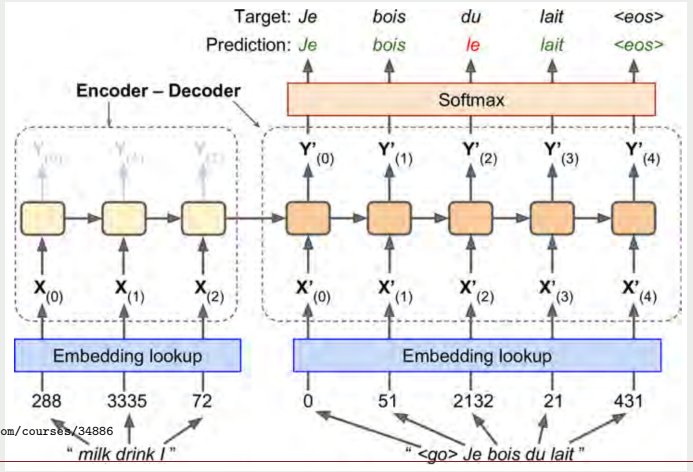
An EncoderDecoder Network for Machine Translation

- Each word is initially represented by a simple integer identifier (e.g., 288 for the word "milk"). Next, an embedding lookup returns the word embedding (as explained ear lier, this is a dense, fairly low-dimensional vector). These word embeddings are what is actually fed to the encoder and the decoder.



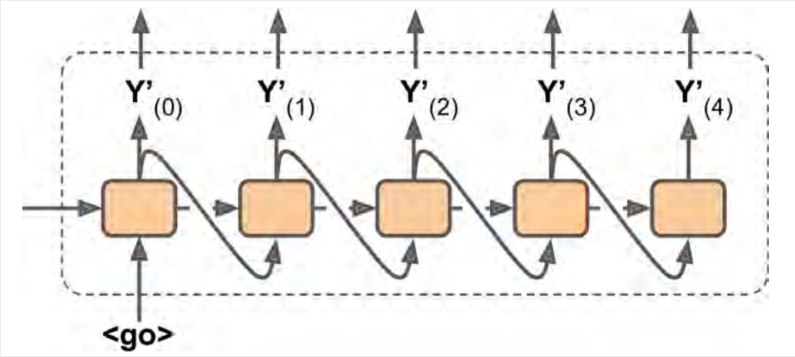
An EncoderDecoder Network for Machine Translation

- At each step, the decoder outputs a score for each word in the output vocabulary (i.e., French), and then the Softmax layer turns these scores into probabilities. For example, at the first step the word "Je" may have a probability of 20%, "Tu" may have a probability of 1%, and so on. The word with the highest probability is output. This is very much like a regular classification task, so you can train the model using the `softmax_cross_entropy_with_logits()` function.



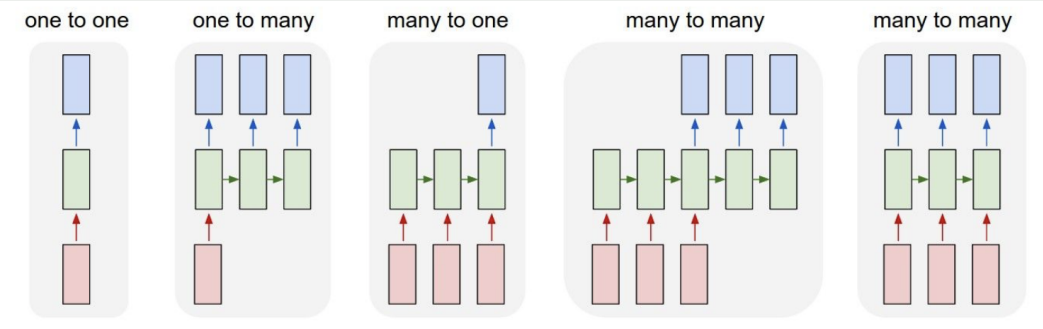
An EncoderDecoder Network for Machine Translation

- Note that at inference time (after training), you will not have the target sentence to feed to the decoder.
- Instead, simply feed the decoder the word that it output at the previous step, as shown below (this will require an embedding lookup that is not shown on the diagram).



CNN vs RNN

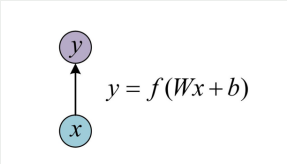
- CNN is with the fixed lengths for inputs and outputs, while RNN can have various lengths with the input and output.
- CNN only have one-to-one structure, while RNN has multiple scenarios.



CNN vs RNN

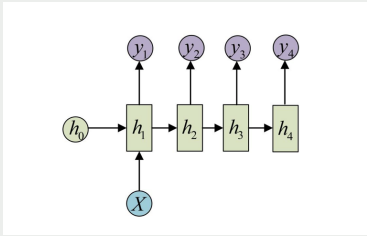
- One-to-one

  The base one layer network is for an input x, after the transform $\mathbf{W}\mathbf{x} + b$ and the activation function $f()$ to obtain the output y.



- One-to-n

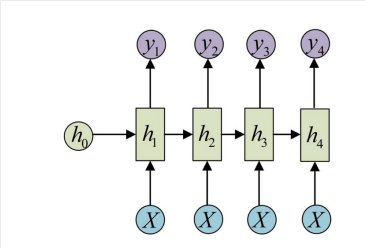
  The input is not a sequence but the output is.



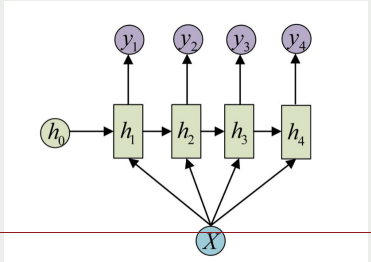
CNN vs RNN

- One-to-n

  Another structure is to have the information X as the input for each stage



or equivalently,

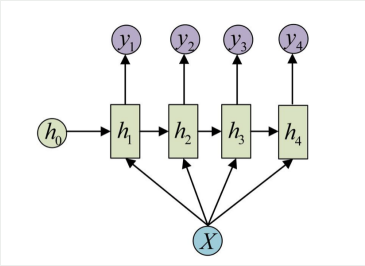


CNN vs RNN

- One-to-n

  This kind of one-to-n structure can handle situations as

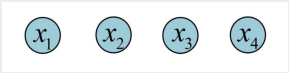
  - From images to create captions. The input X is the image feature and the output y sequence is a sentence, just like the subtitles in the movie.
  - From categories to create music or speech, etc.



CNN vs RNN

- n-to-n

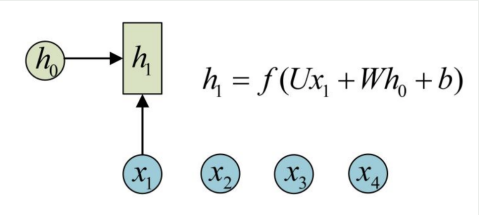
  This is the classic RNN structure where the input and output have the equal sequence length.



Suppose the input is $X = (x_1, x_2, x_3, x_4)$, and every x is a vector for a word.

To construct the model, RNN introduces a **hidden state** concept, where h can extract features from the input sequence and then convert to the outputs.

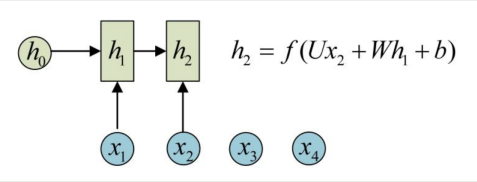
For example, the first h_1



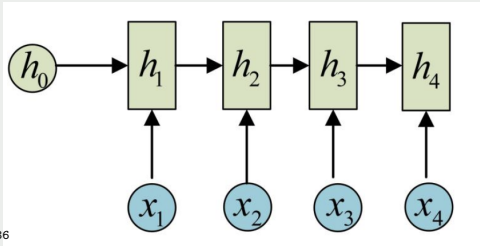
CNN vs RNN

- n-to-n

The second h_2 is similar to h_1. Note that herein every $\mathbf{U}, \mathbf{W}, b$ are the same. In other words, every step shares the same set of parameters, this is the most important feature with RNN.



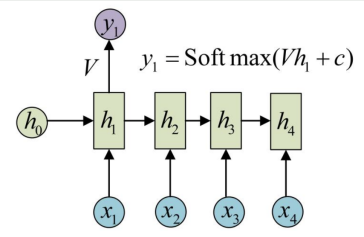
So on and so forth, with the same parameters, we have



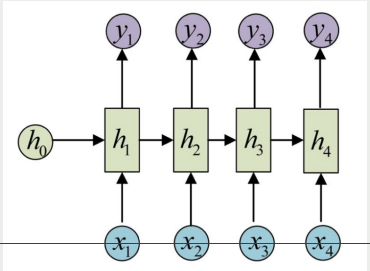
CNN vs RNN

- n-to-n

  The output can be directly calculated from h



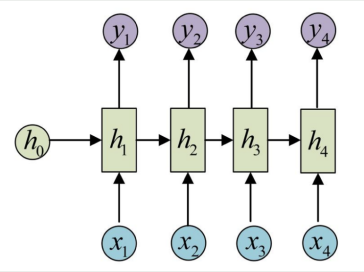
The following is similar,



CNN vs RNN

- n-to-n

  This classic RNN structure is with input $x_1, x_2, \cdots, x_n$ and the same length output $y_1, y_2, \cdots, y_n$.



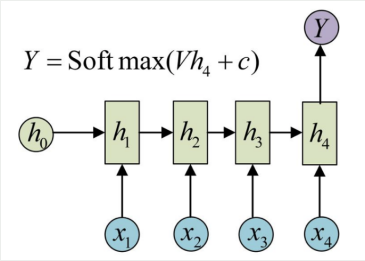
For this constrain, it has limited applications. However, for some special scenarios, it works well,

- ☞ Compute the category labels for each image frame.
- ☞ Input is the character and output is the next character probability. This is the **Char RNN**.

CNN vs RNN

- n-to-one

  The input is a sequence but the output is a scalar.



The only change is to have the output transform at the last h.

This structure usually is used to handle sequence classification question. Like using a paragraph to determine which category it belongs to, or using a sentence to determine its sentiment, or input a video to determine its label, etc.

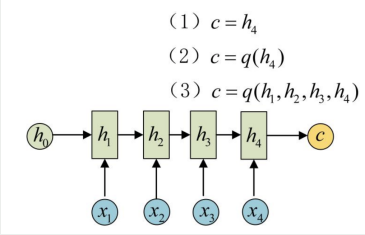
Encoder-Decoder

- n-to-m

  The input and output are sequences with different lengths.

  This structure is Encoder-Decoder, or Seq2Seq, one of the variant of RNN.

  The original n-ton RNN requires the equal length, however, most of the cases we meet in real life are with different sequence lengths, like machine translations, the original and target language sentences are not necessary with the same length.

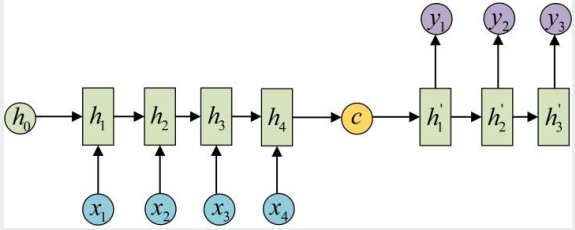


For this reason, Encoder-Decoder first converts input data as a context semantic vector C

Encoder-Decoder

- n-to-m

  Semantic vector C has multiple options for expressions. The simplest way is to put the last hidden state of the Encoder to C, or to have some transform of the last hidden state for C, or even have some transform of all the hidden states.



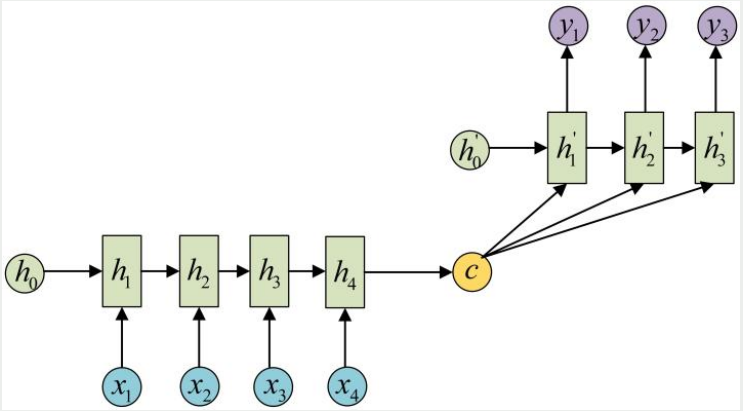
Once you have C, you can use another RNN to decouple the information, this part of RNN is named Decoder.

Decoder of RNN may be the same as the encoder, or may different. Specifically, you need to put C as the initial status h_0 to the Decoder.

Encoder-Decoder

- n-to-m

  Another solution is to use C as the input of every stage.



Encoder-Decoder Applications

- As Encoder-Decoder structure has no limitations of the input and output sequence lengths, it has wider applications.
  - Machine translation
  - Context abstract
  - Reading comprehension
  - Speech recognition

Encoder-Decoder Framework

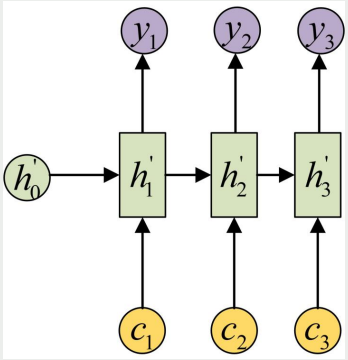
- Encoder-Decoder is not a specific model, but a framework.
  - ☞ Encoder: Convert the input sequence to a fixed length vector
  - ☞ Decoder: Convert a fixed length vector to the output sequence
  - ☞ Encoder and Decoder can be used independently or together

Drawback of Encoder-Decoder

- The largest limitation: The only connections between Encoder $\&$ Decoder are the fixed length semantic vectors C.
- Encoder has to convert the information of the whole sequence set into a fixed length semantic vector C.
- The semantic vector C may not carry the full information from the sequence.
- The information carried from the previous contents will be diluted by the following input information, or even overridden
- The longer the sequence, the worse the situation, this will make the Decoder does not obtain enough input information from the beginning, therefore it will degrade the Decoder performance.
- ☞ To solve the limitations from Encoder-Decoder, Researchers proposed Attention Mechanism

Attention Mechanism

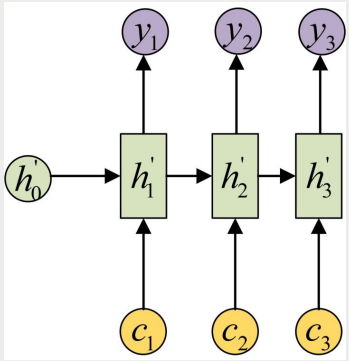
- Attention mechanism is a refined model of Encoder-Decoder.
- Attention mechanism utilizes inputing different c at different time slot to solve the problem.



- Every c will automatically select the best context information for the current output y.

Attention Mechanism

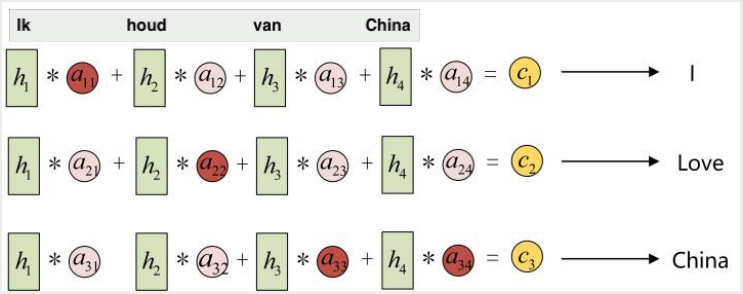
- Specifically, we use $a_{i,j}$ to evaluate the correlation between h_j at j-th stage of Encoder and i-th stage of Decoder.



- At last, the input context information c_i at i-th stage of Decoder comes from the weighted summation of all h_j over $a_{i,j}$.

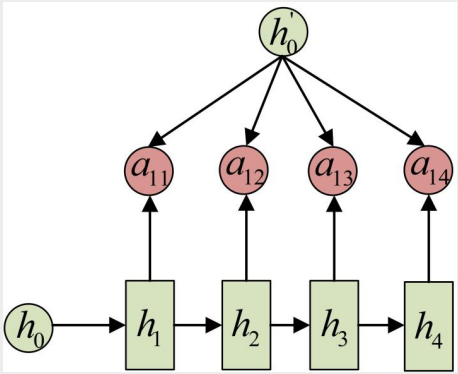
Attention Mechanism

- For example, the input Dutch sequence is "Ik houd van China", then, the $h_1,\ h_2,\ , h_3,\ h_4$ can be treated as information contained in "Ik", "houd", "van", "China".
- When we translate the Dutch sentence to English, the first context c_1 should be mostly related to the "Ik", and therefore the related $a_{1,1}$ should be larger, while $a_{1,2}, a_{1,3}, a_{1,4}$ smaller. c_2 should be more related to "houd", therefore, $a_{2,2}$ should be bigger. Lastly, c_3 are mostly related to h_3, h_4, then $a_{3,3}, a_{3,4}$ should be bigger



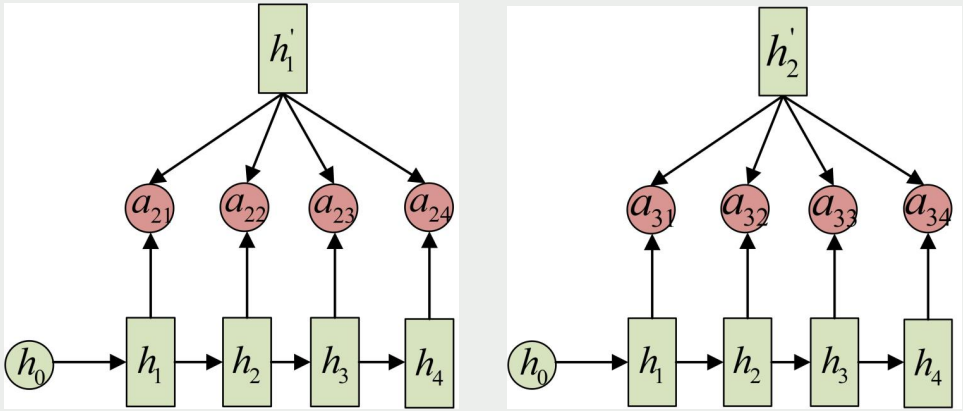
Attention Mechanism

- From the previous example, we have one more question: Where do these weights $\{a_{i,j}\}$ come from?
- In fact, $\{a_{i,j}\}$ are learned from the model. They are related to the $(i-1)$-th stage's hidden status with Encoder and the j-th stage's hidden status with Decoder.
- For example, to calculate $a_{1,j}$, (arrow means taking converting of h' and h_j at the same time)



Attention Mechanism

- For example, to calculate $a_{2,j}$, and $a_{3,j}$



Pros: Attention Mechanism

- During machine translations, let words not only care about the global context vector c, but also add the "attention range". It means the following output words will pay more attention to part of the input sequence. Based on the attention range to generate the next output.
- Do not require the Encoder to put all the information in a fixed-length vector.
- To make the input sequence encoded into a vector; During Decoder, every step to selectively pick a subset of the sequence to process.
- Every single output can fully take advantage of the carried information in the input. Different context vector c_i has a different attention focus.
-

Cons: Attention Mechanism

- You have to compute attention for every input and output combinations. Sequence with 50 words needs to compute 2500 attention.
- Before deciding which subset for attention, attention merchanism has to first scan all the history memory to determine what is the next output.
- ☞ Another alternative of attention is Reinforcement Learning, which can be used to predict the approximate location of the attention range.