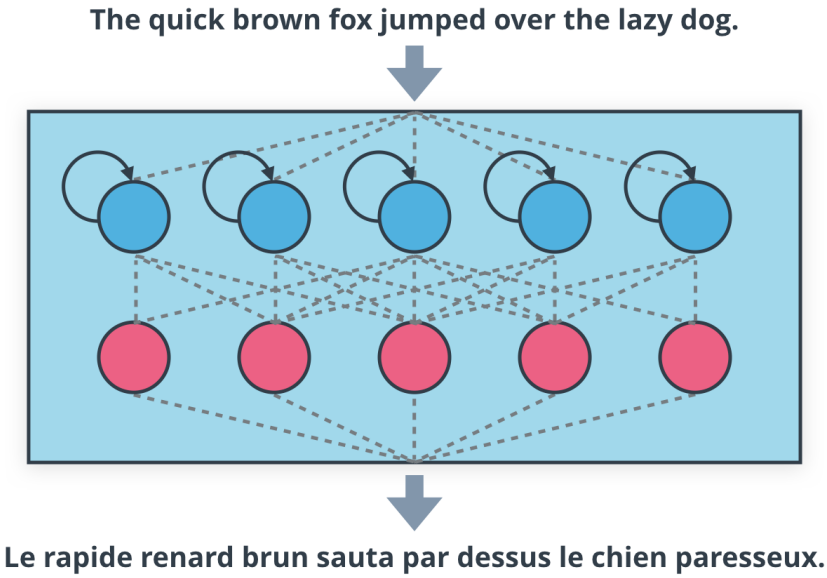
Machine Translation can be thought of as a sequence-to-sequence learning problem.



Machine Translation: A Sequence-to-Sequence Learning Problem

You have one sequence going in, i.e. a sentence in the source language, and one sequence coming out, its translation in the target language.

This seems like a very hard problem - and it is! But recent advances in Recurrent Neural Networks have shown a lot of improvement. A typical approach is to use a recurrent layer to encode the meaning of the sentence by processing the words in a sequence, and then either use a dense or fully-connected layer to produce the output, or use another decoding layer.

Experimenting with different network architectures and recurrent layer units (such as LSTMs, GRUs, etc.), you can come up with a fairly simple model that performs decently well on small-to-medium size datasets. Commercial-grade translation systems need to deal with a much larger vocabulary, and hence have to use a much more complex model, apply different optimizations, etc. Training such models requires a lot of data and compute time.

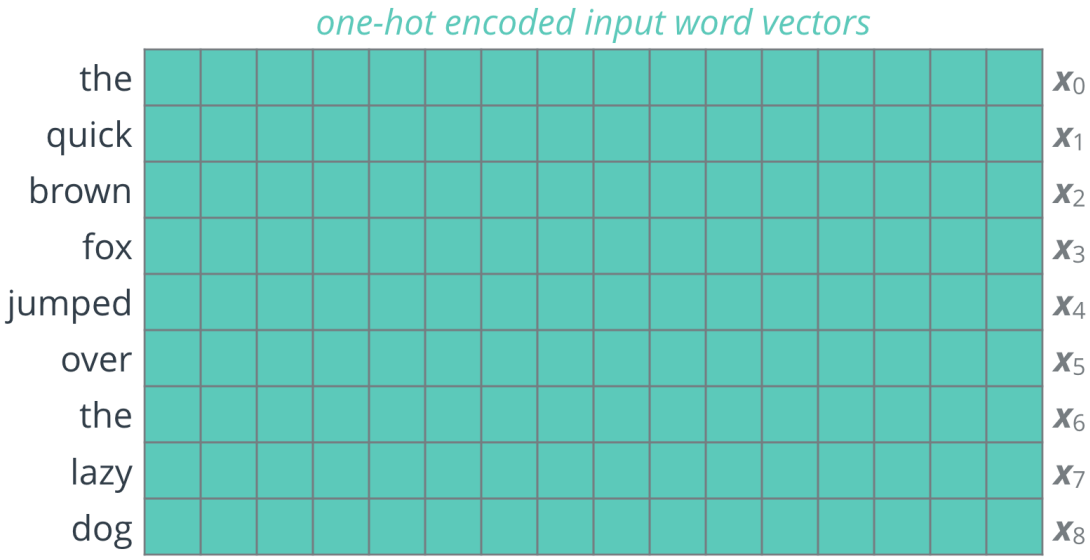
Neural Net Architecture for Machine Translation

Let's develop a basic neural network architecture for machine translation.

Input Representation

The key thing to note here is that instead of a single word vector or document vector as input, we need to represent each sentence in the source language as a sequence of word vectors.

Therefore, we convert each word or token into a one-hot encoded vector, and stack those vectors into a matrix - this becomes our input to the neural network.

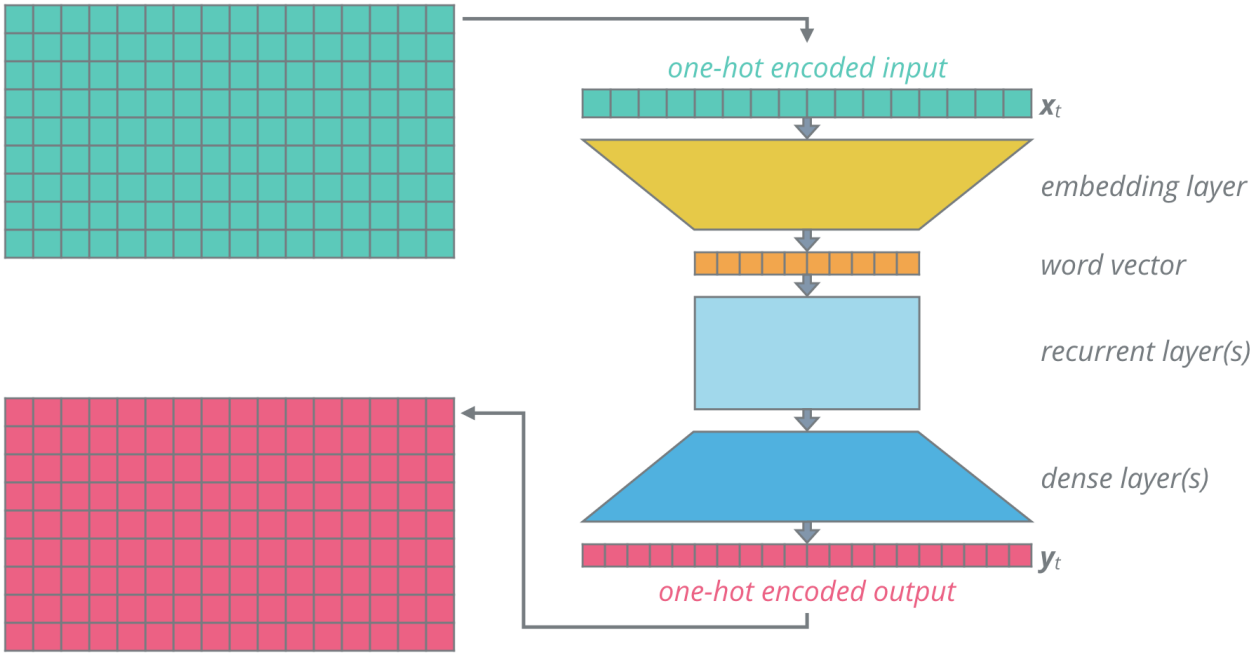


Input Representation

You may be wondering what to do about sequences of different length: One common approach is to simply take the sequence of maximum length in your corpus, and pad each sequence with a special token to make them all the same length.

Basic RNN Architecture

Once we have the sequence of word vectors, we can feed them in one at a time to the neural network.



Basic RNN Architecture for Machine Translation

Embedding Layer

The first layer of the network is typically an embedding layer that helps enhance the representation of the word. This produces a more compact word vector that is then fed into one or more recurrent layers.

Recurrent Layer(s)

This is where the magic happens! The recurrent layer(s) help incorporate information from across the sequence, allowing each output word to be affected by potentially any previous input word.

Note: You could skip the embedding step, and feed in the one-hot encoded vectors directly to the recurrent layer(s). This may reduce the complexity of the model and make it easier to train, but the quality of translation may suffer as one-hot encoded vectors cannot exploit similarities and differences between words.

Dense Layer(s)

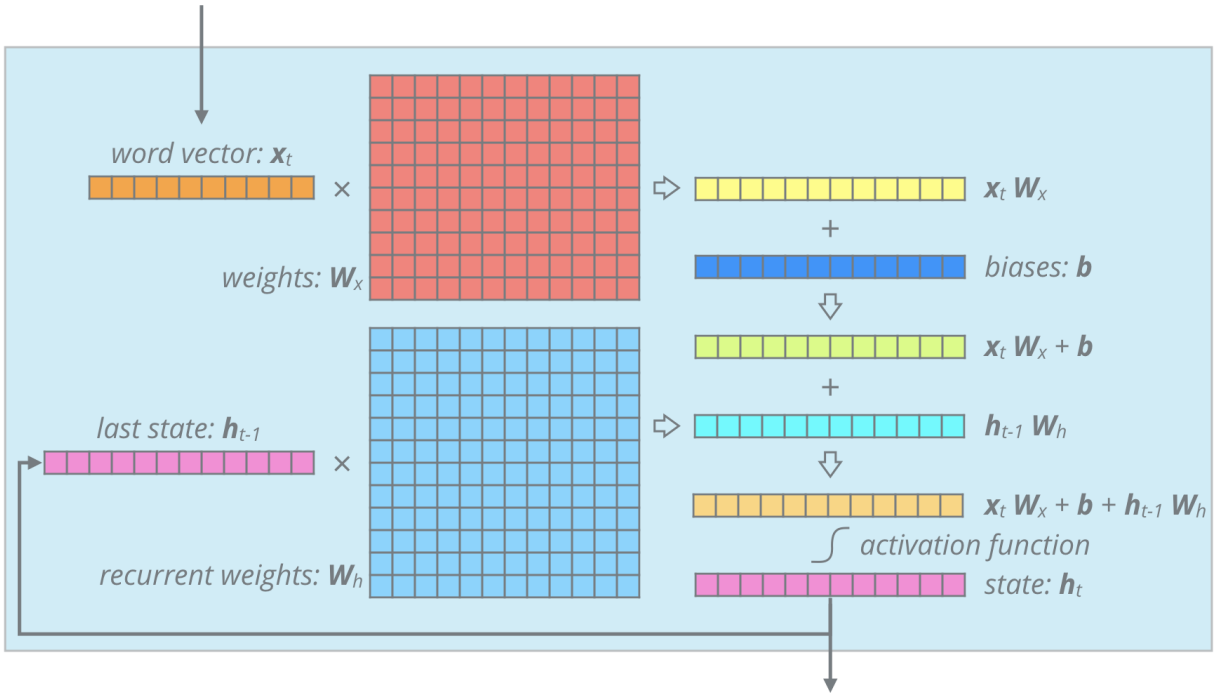
The output of the recurrent layer(s) is fed into one or more fully-connected dense layers that produce softmax output, which can be interpreted as one-hot encoded words in the target language.

As each word is passed in as input, its corresponding translation is obtained from the final output. The output words are then collected in a sequence to produce the complete translation of the input sentence.

Note: For efficient processing we would like to capture the output in a matrix of fixed size, and for that we need to have output sequences of the same length. Again, we can achieve this by using the same padding technique as we used for input.

Recurrent Layer: Internals

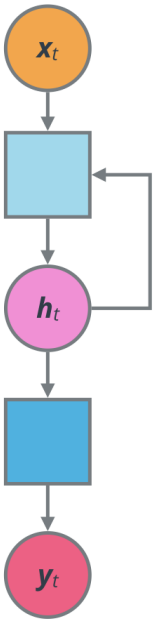
Let's take a closer look at what is going on inside a recurrent layer.



Recurrent Layer: Internals

- The input word vector x_t is first multiplied by the weight matrix: W_x
- Then bias values are added to produce our first intermediate result: $x_t W_x + b$
- Meanwhile, the state vector from the previous time step h_{t-1} is multiplied with another weight matrix W_h to produce our second intermediate result: $h_{t-1} W_h$
- These two are then added together, and passed through an activation function such as ReLU, sigmoid or tanh to produce the state for the current time step: h_t
- This state vector is passed on as input to the next fully-connected layer, that applies another weight matrix, bias and activation to produce the output: y_t

Let's simplify this diagram and look at the bigger picture again.



Basic RNN: Schematic

The key thing to note here is that the RNN's state h_t is used to produce the output y_t, as well as looped back to produce the next state.

In summary, a recurrent layer computes the current state h_t as:

> $$h_t = f(x_t W_x + h_{t-1} W_h + b)$$

Here $f(\cdot)$ is some non-linear activation function, x_t is the input vector, W_x is the input weight matrix, h_{t-1} is the previous state vector, W_h is the recurrent weight matrix and b is the bias vector.

QUIZ QUESTION

Let's say you've decided to use a word vector (x_t) of length 200, and a state vector (h_t) of length 300. Treating these as single-row matrices, we can write the sizes as 1x200 and 1x300 respectively.

Now, what is the size of each parameter of the RNN? Match the correct sizes below.

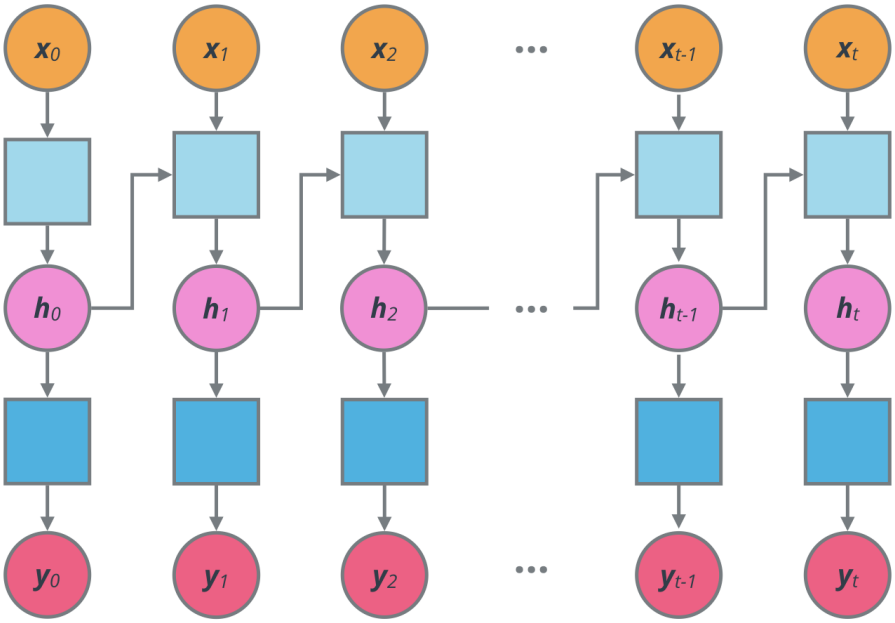
1x500 | 200x200 | 300x1 | 300x200 | 200x300 | 1x300 | 300x300

PARAMETER	SIZE (ROWS X COLS)
Input weight matrix (W_x)	
Recurrent weight matrix (W_h)	
Bias vector (b)	

SUBMIT

Unrolling an RNN

It’s easier to understand how this works over time if we unroll it.



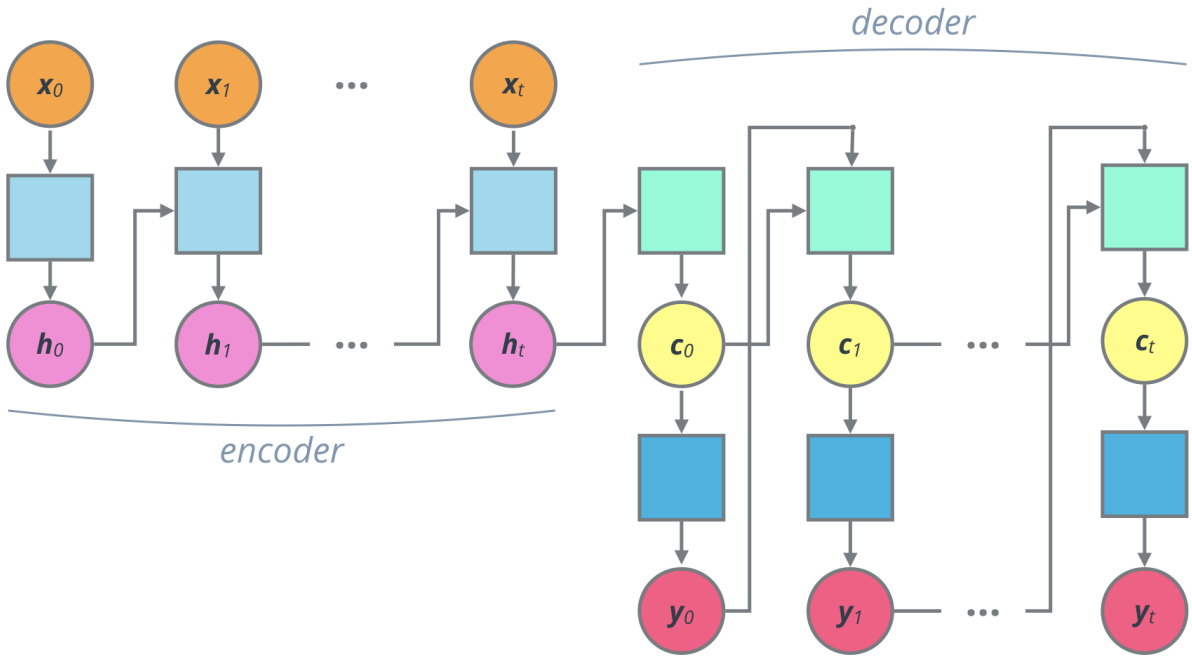
Basic RNN: Unrolled

Each copy of the network you see represents its state at the respective time step. At any time t, the recurrent layer receives input x_t as well as the state vector from the previous step, h_{t-1}. This process is continued till the entire input is exhausted.

The main drawback of such a simple model is that we are trying to read the corresponding output for each input word immediately. This would only work in situations where the source and target language have an almost one-to-one mapping between words.

Encoder-Decoder Architecture

What we should ideally do is to let the network learn an internal representation of the entire input sentence, and then start generating the output translation. In fact, you need two different networks in order to achieve this.



Encoder-Decoder: Unrolled

The first is called an Encoder, which accepts the source sentence, one word at a time, and captures its overall meaning in a single vector. This is simply the state vector at the last time step. Note that the encoder network is not used to produce any outputs.

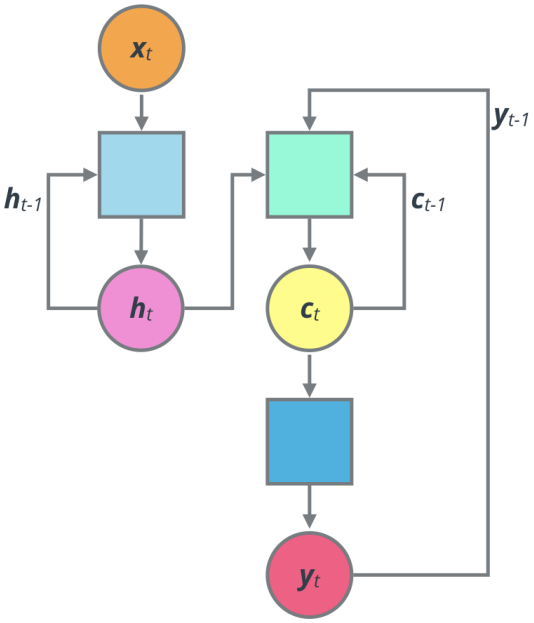
The second network is called a Decoder, which then interprets the final sentence vector and expands it into the corresponding sentence in the target language, again one word at a time.

The first time step for the decoder network is special. It is fed in the final sentence vector from the encoder h_t, and given a sentinel input to kickstart the process. The recurrent portion of the network produces a state vector c_0, and with that the fully-connected portion produces the first output word in the target language, y_0.

At each subsequent time step t, the decoder network uses its own previous state c_{t-1} as well as its own previous output y_{t-1}, in order to produce the current output, y_t.

This process is typically continued for a fixed number of iterations, with the idea that the network will start producing special padding symbols after all meaningful words have been generated. Alternately, the network could be trained to output a stop symbol, such as a period (.), to indicate that the translation is complete.

If we roll back the time steps, we can see what the overall architecture looks like.



Encoder-Decoder: Schematic

This encoder-decoder design very popular for several sequence-to-sequence tasks, not just Machine Translation.

Now, there are several variations of this design that can be used to enhance the performance of the network.

- One option is to use different kinds of recurrent neural network units, such as LSTMs, GRUs etc. instead of vanilla RNN units. That allows the network to better analyze the input sequence, at the cost of additional model complexity.
- Another dimension to explore is how many recurrent layers to use. Each layer effectively incorporates information from the input sequence, producing a compact state vector at each time step. Additional layers can essentially incorporate information across these state vectors.
- Other more innovative approaches include adding in a backward encoder (bidirectional encoder-decoder model), feeding in the sentence vector to each time step of the decoder (attention mechanism), etc.

Feel free to experiment with these different approaches to see what architecture works best for your task. Keep in mind that these mechanisms typically add to the model complexity, which means you need more data and time to train the additional parameters.