**Long Short-Term Memory Units**

The basic principle behind the architecture was that the network would be designed for the purpose of reliably transmitting important information many time steps into the future. The designs considerations resulted in the architecture shown below [1, p. 213]

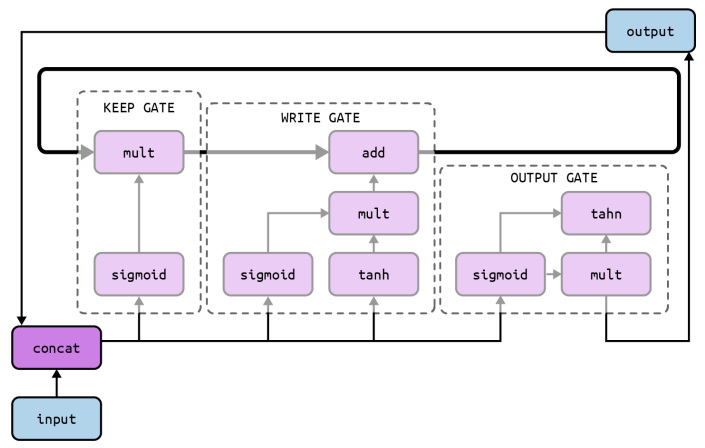


Figure 1 The architecture of an LSTM unit, ilustrated at a tensor (designated by arrows) and operation (designated by the inner blocks) level

**The Keep gate**

One of the core components of the LSTM architecture is the memory cell, a tensor represented by the bolded loop in the centre of the figure. The memory cell holds critical information that it has learned over time, and the network is designed to effectively maintain useful information in the memory cell over many time steps. This is determined by the *keep gate*, shown in detail in figure 2.

The basic idea of the keep gate is simple. The memory state tensor from the previous time step is rich with information, but some of the information may be stale (and therefore might need to be erased). We figure out which elements in the memory state tensor are still relevant and which elements are irrelevant by trying to compute a bit tensor (a tensor of zeros and ones) that we multiply with the previous state. If a particular location in the bit tensor holds a 1, it means that location in the memory cell is still relevant and ought to be kept. If that particular location instead holds a 0, it means that the location in the memory cell is no longer relevant and ought to be erased. The bit tensor is approximated by concatenating the input of the current time step and the LSTM output of the previous time step and applying a sigmoid layer to the resulting tensor.

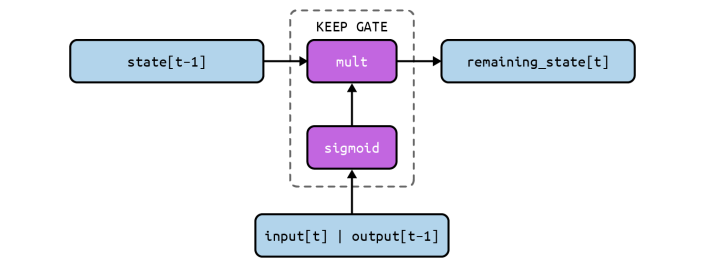


Figure 2 Architecture of the keep gate of an LSTM unit

**The write gate**

Once we’ve figured out what information to keep in the old state and what to erase, we’re ready to think about what information we’d like to write into the memory state. This part of the LSTM unit is called the *write gate*, and is depicted in figure 3.

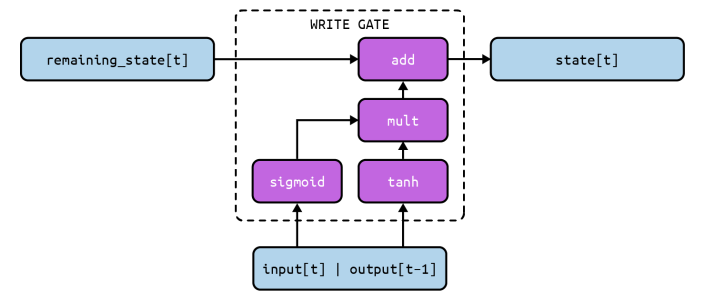


Figure 3 Architecture of the output gate of an LSTM unit

This is broken down into two major parts. The first component is figuring out what information we’d like to write into the state. This is computed by the tanh layer to create an intermediate vector. The second component is figuring out which components of this computed tensor we actually want to include into the new state and which we want to toss before writing. We do this by approximating a bit vector of 0’s and 1’s using the same strategy (a sigmoid layer) as we used in the keep gate. We multiply the bit vector with our intermediate tensor and then add the result to create the new state vector for the LSTM.

**The output gate**

At every time step, we’d like the LSTM unit to provide an output. The LSTM unit is engineered to provide more flexibility by emitting an output tensor that is an “interpretation” or external “communication” of what the state vector represents. The gate is shown in figure 4. We use a nearly identical structure as the write gate: (1) the tanh layer creates an intermediate tensor from the state vector, (2) the sigmoid layer produces a bit tensor mask by using the current input and previous output, and (3) the intermediate tensor is multiplied with the bit tensor to produce the final output.

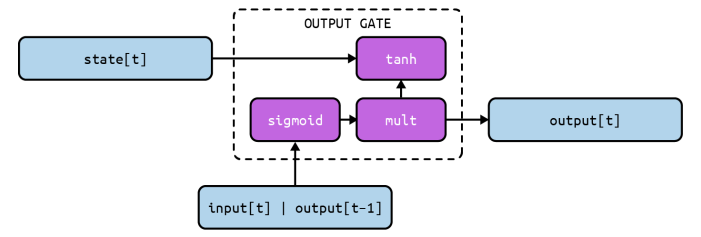


Figure 4 Architecture of the output gate of an LSTM unit

**Building LSTM units in pytorch**

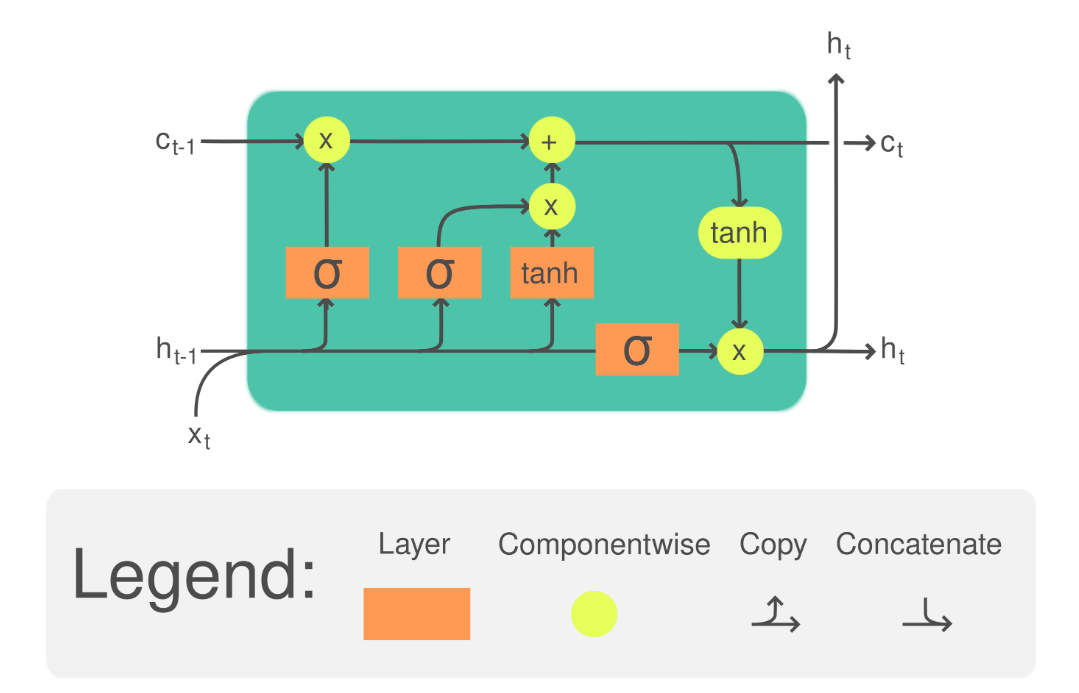
****

Figure 5 SImplified LSTM cell architecture

**Parameters**

* **input\_size** – the number of expected features in the input x
* **hidden\_size** – The number of features in the hidden state h
* **num\_layers** – Number of recurrent layers. E.g., setting num\_layers=2 would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results.
* **bias** – If False, then the layer does not use bias weights *b\_ih* and *b\_hh*.
* **batch\_first** – If True, then the input and output tensors are provided as (batch, seq, feature) instead of (seq, batch, feature). Note that this does not apply to hidden or cell states.
* **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout.
* **bidirectional** – If True, becomes a bidirectional LSTM.
* **proj\_size** – If > 0, will use LSTM with projections of corresponding size.

**Inputs: input, (h\_0, c\_0)**

* **input**: tensor of shape for unbatched input, when batch\_first = False or when batch\_first = True containing the features of the input sequence.
* **h\_0**: tensor of shape for unbatched input or containing the initial hidden state for each element in the input sequence.
* **c\_0**: tensor of shape for unbatched input or containing the initial cell state for each element in the input sequence.

**Outputs: output, (h\_n, c\_n)**

* **output**: tensor of shape for unbatched input when batch\_first = False or when batch\_first=True containing the outputs features (h\_t) from the last layer of the LSTM ,for each t.
* **h\_n**: tensor of shape for unbatched input or containing the final hidden state for each element in the sequence.
* **c\_n**: tensor of shape for unbatched input or containing the final hidden state for each element in the sequence.

where:

A diagram of a machine

AI-generated content may be incorrect.

Figure 6 Visualization of the LSTM inputs and outputs in pytorch. num\_layers renamed to w, batch dimension ignored. Source: https://i.sstatic.net/SjnTl.png

**Example of an LSTM implemented in pytorch**

class LSTM(nn.Module):

  def \_\_init\_\_(self, input\_size, hidden\_size, num\_stacked\_layers):

    super().\_\_init\_\_()

    self.input\_size = input\_size

    self.hidden\_size = hidden\_size

    self.num\_stacked\_layers = num\_stacked\_layers

    self.layer\_1 = nn.LSTM(input\_size=input\_size,

                           hidden\_size=hidden\_size,

                           batch\_first=True,

                           num\_layers=num\_stacked\_layers,

                           dropout=0.5)

    self.layer\_2 = nn.Linear(hidden\_size, 1)

  def forward(self, x):

    batch\_size = x.size(0)

    h0 = torch.zeros(self.num\_stacked\_layers, batch\_size, self.hidden\_size) #(n\_layers, batch\_size, hidden\_size)

    c0 = torch.zeros(self.num\_stacked\_layers, batch\_size, self.hidden\_size) #(n\_layers, batch\_size, hidden\_size)

    out, (h0, c0) = self.layer\_1(x, (h0, c0))

    out = self.layer\_2(out[:,-1,:])

    return out

A graph of a graph

AI-generated content may be incorrect.

**Encoder-Decoder Network**

A diagram of a process flow

AI-generated content may be incorrect.

Figure 7 A simple machine translation model

Let’s take a look at a simple neural machine translation model that will translate English sentences to French (see Figure 7). In short, the English sentences are fed to the encoder, and the decoder outputs the French translations. Note that the French translations are also used as inputs to the decoder, but shifted 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 outputs). [2]

A diagram of a machine

AI-generated content may be incorrect.

Figure 8 Example of an encoder-decoder or sequence-to-sequence RNN architecture

Here we discuss how an RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length. This comes up in many applications, such as speech recognition, machine translation or question answering, where the input and output sequences in the training set are generally not of the same length (although their lengths might be related). We often call the input to the RNN the “context.” We want to produce a representation of this context, . The context might be a vector or sequence of vectors that summarize the input sequence .

The idea is very simple: (1) an **encoder** or **reader** or input **RNN** processes the input sequence. The encoder emits the context , usually as a simple function of its final hidden state. (2) a **decoder** or **writer** or **output** RNN is conditioned on that fixed-length vector (just like in figure 10.9) to generate the output sequence . The innovation of this kind of architecture over those presented in earlier sections of this chapter is that the lengths and can vary from each other, while previous architectures constrained . In a sequence-to-sequence architecture, the two RNNs are trained jointly to maximize the average of log over all the pairs of and sequences in the training set. The last state of the encoder RNN is typically used as a representation of the input sequence that is provided as input to the decoder RNN. [3]

**Conditional Language Models**

Sequence-to-sequence models need to estimate the conditional probability of a sequence given a source That’s why sequence-to-sequence tasks can be modelled as Conditional Language Model (CLM) – they operate similarly to LMs, but additionally receive source information [4]

Since the only difference from LMs is the presence of source , the modelling and training is very similar to language models. In particular, the high-level pipeline is as follows:

* feed source and previously generated target words into a network;
* get vector representation of context (both source and previous target) from the network decoder;
* from this vector representation, predict a probability distribution for the next token.

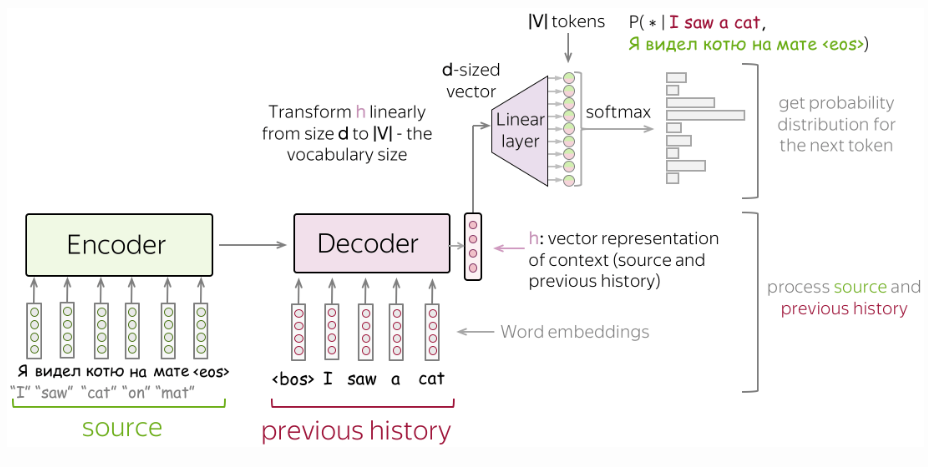


Figure 9 seq-to-seq high-level pipeline

**Training: The cross-entropy loss (once again)**

Similarly to neural LMs, neural seq2seq models are trained to predict probability distributions of the next token given previous context (source and previous target tokens). Intuitively, at each step we maximize the probability a model assigns to the correct token.

Formally, let’s assume we have a training instance with the source and the target . Then at the timestep , a model predicts a probability distribution . The target at this step is , i.e., we want a model to assign probability 1 to the correct token, , and zero to the rest.

The standard loss function is the cross-entropy loss. Cross-entropy loss for the target distribution and the predicted distribution is

Since only one of is non-zero (for the correct token ), we will get

**Inference: Greedy Decoding and Beam Search**

Now when we understand how a model can look like and how to train this model, let's think how to generate a translation using this model. We model the probability of a sentence as follows:

Now the main question is: how to find the argmax?

Note that we cannot find the exact solution. The total number of hypotheses we need to check is , which is not feasible in practice. Therefore, we will find an approximate solution.

**Greedy Decoding: At each step, pick the most probable token**

The straightforward decoding strategy is greedy - at each step, generate a token with the highest probability. This can be a good baseline, but this method is inherently flawed: the best token at the current step does not necessarily lead to the best sequence.

**Beam Search: Keep track of several most probably hypotheses**

Instead, let's keep several hypotheses. At each step, we will be continuing each of the current hypotheses and pick top-N of them. This is called beam search.

Usually, the beam size is 4-10. Increasing beam size is computationally inefficient and, what is more important, leads to worse quality.

**Example seq2seq for time-series forecasting**

Here we will focus on the earliest modern seq2seq model type to be popularized: RNN Encoder to RNN Decoder.

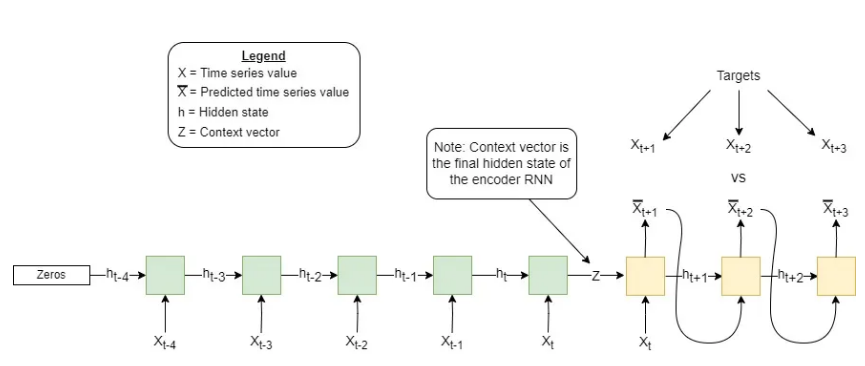


Figure 10 Fully reccurrent seq2seq model where the decoder is free running; note that the latest encoder input is the same as the first decoder input.

From this we can see the internals of the encoder/decoder black boxes. Importantly, the decoder here is “free running” or feeding its previous output back into itself at each timestep. This is used for inference since we would not know the future values (and evaluation since it is meant to mimic inference). However, for training we can take advantage of having this “future” data to help make the training more stable with teacher-forcing.

**Scheduled Sampling**

Teacher forcing is much easier than free running given that no matter how long the forecast horizon is, each decoder input keeps it on track. To counteract this difficulty gap we can use scheduled sampling. This simply means we use a decay function parametrized by time (either current training epoch or batch) to decide the probability of teacher forcing during training.

This function is sampled every decoder step, which can result in both free running and teacher forcing within the same decoder sequence rollout.

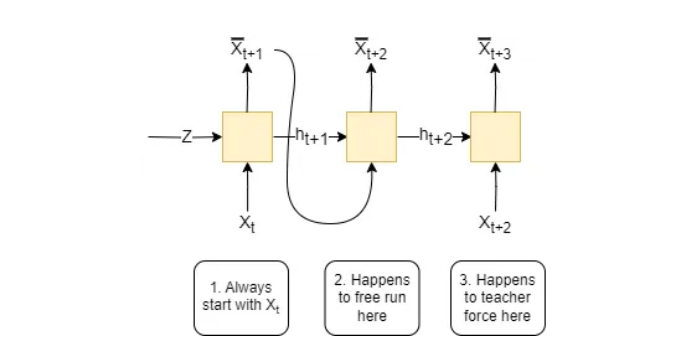


Figure 11 Shows that within a single rollout both free running and teacher forcing can occur

**Papers**

**An electricity load forecasting model based on multilayer dilated LSTM network and attention mechanism** **[5]**

“Generally speaking, for series with orders of magnitude less than 100, LSTM networks can perform the task of serial forecasting relatively well. Beyond this limit, the forecasting accuracy of the LSTM network will be decreasing. Secondly, in serial forecasting the input and output dimensions are often unequal, which requires the use of Seq2Seq models to map them. One of the two: in serial forecasting the input and output dimensions are often unequal, which requires the use of Seq2Seq model to map them. However, Seq2Seq has the generation of context vectors, which will lose the series information extracted by the Encoder layer to some extent, making the forecasting accuracy lower.”

**Proposed Solutions:**

(1) We used a multilayer extended LSTM network instead of traditional LSTM to extend the length of the historical series in load forecasting.

(2) In order to improve the accuracy of load forecasting, a new model has been established. This model uses a multilayer dilated LSTM structure as Encoder layer to extract long series information under the Seq2Seq framework, introduces attention mechanism to focus on key information, and finally LSTM are employed to decode the context vectors.

(3) Experiments have been conducted on two real load datasets in different regions, and the results show that the proposed model has a higher forecasting accuracy compared to the benchmark methods.

A table with text and words

AI-generated content may be incorrect.

**Building Energy Load Forecasting using Deep Neural Networks** [6]

“The main advantage of this architecture is that it allows inputs of arbitrary length. I.e. an arbitrary number of available load measurements of previous time steps can be used as inputs, to predict the load for an arbitrary number of future time steps.”

**Problem**

“For the first 60 hours, the actual load measurements on previous time steps were used as inputs to perform the prediction. Starting at hour 60, the predictions were introduced as inputs to generate a forecast for the next 60 hours. The figure shows how the model is incapable of providing accurate forecast for the last 60 hours, even with the forecast for one-time step ahead being very accurate. It can be assumed that the reasons behind this behaviour is that predicting the next step can be achieved with low error by simply bypassing the input from the current step straight to the output, this is because consequent measurements are very similar (when using one-minute resolution data). Therefore, if the network predicts that the load for the next time step is the same that the load on the current time. Thus, the neural network is learning a naïve mapping, where it generates an output equal to the input”

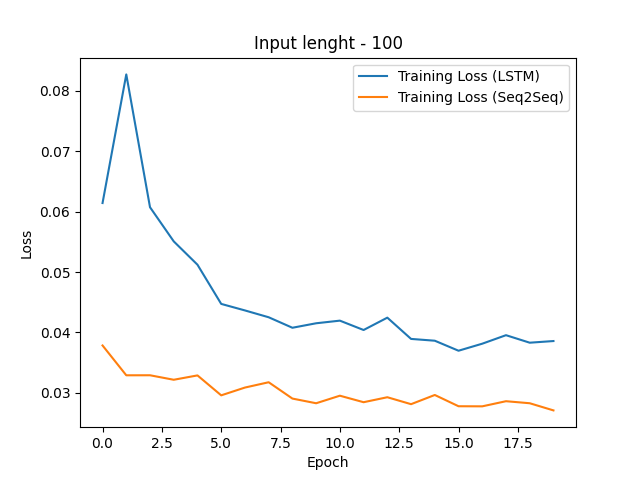
**Proposed solution:**

“Two approaches were investigated to solve the aforementioned problem. First was to introduce measurements from further in the past as inputs, for example 5 steps back, as opposed to inputting the load from the previous time step. This was done so that the input and output would different enough for the network to be able to learn a useful representation of the data. It can be seen that the architecture provides an estimation that follows the general trend of the future load. This method produced accurate results when used with hourly data, but failed to perform well with one-minute resolution data. Given that a delay of the input was not sufficient to provide a useful model for one-minute time steps, the second approach that was tested was to experiment with a S2S, LSTM-based architecture.”

A graph with blue and orange lines

AI-generated content may be incorrect.A graph with blue and orange lines

AI-generated content may be incorrect.

A graph with blue and orange lines

AI-generated content may be incorrect.

A graph of a graph

AI-generated content may be incorrect.A graph of loss of data

AI-generated content may be incorrect.

A graph of a graph

AI-generated content may be incorrect.A graph with blue lines and orange lines

AI-generated content may be incorrect.

A graph of a graph

AI-generated content may be incorrect.A graph with blue lines and orange lines

AI-generated content may be incorrect.

[1] N. Buduma, N. Buduma, Papa Joe, y N. Locascio, *Fundamentals of deep learning: designing next-generation machine intelligence algorithms*, Second edition. Sebastopol, CA: O’Reilly Media, Inc, 2022.

[2] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*, Third edition. Beijing: O’Reilly Media, Inc, 2023.

[3] I. Goodfellow, A. Courville, y Y. Bengio, *Deep learning*. en Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016.

[4] «Seq2seq and Attention». Accedido: 1 de octubre de 2025. [En línea]. Disponible en: https://lena-voita.github.io/nlp\_course/seq2seq\_and\_attention.html#related\_papers

[5] «Frontiers | An electricity load forecasting model based on multilayer dilated LSTM network and attention mechanism». Accedido: 6 de octubre de 2025. [En línea]. Disponible en: https://www.frontiersin.org/journals/energy-research/articles/10.3389/fenrg.2023.1116465/full

[6] D. L. Marino, K. Amarasinghe, y M. Manic, «Building Energy Load Forecasting using Deep Neural Networks», en *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, oct. 2016, pp. 7046-7051. doi: 10.1109/IECON.2016.7793413.