Recurrent Neural Networks

The contents of this material not pretend to be an original work and are mostly a scheme of Chapter 10 from the book "Deep Learning" by Ian Goodfellow, Yoshua Bengio and Aaron Courville.

Contents

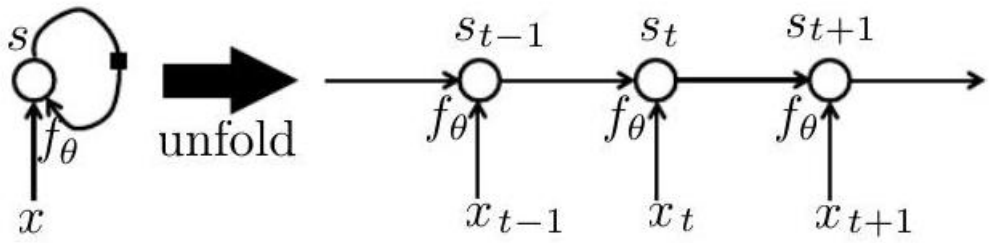
Motivation

One of the early ideas found in machine learning and statistical models is that of sharing parameters across different parts of a model, allowing to extend and apply the model to examples of different forms and generalize across them. In general any problem with a determined order parameter (e.g. time, word order in a text) where the different parts of the sequence share certain similarities may be suitable for RNN. This allows one to model variable length sequences, whereas if we had specific parameters for each value of the order parameter, we could not generalize to sequence lengths not seen during training, nor share statistical strength across different sequence lengths and across different positions in time. Moreover, parameter sharing implicitly means that the absolute time step at which an event occurs is meaningless: it only makes sense to consider the event in some context that somehow captures what has happened before. These characteristics can be seen in surprisingly huge amount of different problems as speech and handwriting recognition, text processing, prediction of market evolution, image analysis...

Flow graph for representation of RNN

A flow graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. There are two

ways of representing the set of computations, the folded and unfolded representations. As an example consider the following network,



At every step t, the network receives a new variable input $\mathbf{x}_t$ and the processed function f_θ which encodes the information of all the previous steps. Writing it explicitly,

$$\mathbf{s}_t = f_\theta(\mathbf{x}_t, \mathbf{s}_{t-1}) \tag{1}$$

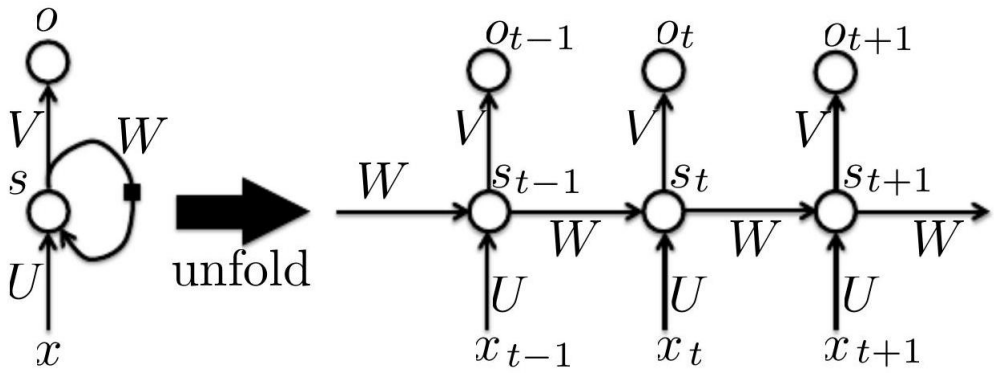
Where the parameters θ of the problem are shared for all steps. This function could be written also as,

$$\mathbf{s}_t = g_{\theta,t}(\mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, ..., \mathbf{x}_2, \mathbf{x}_1) \tag{2}$$

From this expression is clear the power of the recurrence since we do not need to create a different function for every step. Instead of that a vector of finite length $\mathbf{s}_t$ encodes all the relevant information from the previous steps relevant for the next step. It is clear that since the vector s_t is finite, there must be a loss of information during the process.

Example: Vanilla RNN

The most basic machine learning program (MLP) is the Vanilla RNN. We will compute explicitly all the algorithm for this model and the rest of the discussion will rely more on the graphical representation and the intuition behind those models. Its representation is,



Where $\mathbf{V}$, $\mathbf{U}$, $\mathbf{W}$ are linear weighting vectors. The equations describing this model are,

$$\begin{cases} \mathbf{a}_t = \mathbf{b} + \mathbf{W}\mathbf{s}_{t-1} + \mathbf{U}\mathbf{x}_t \\ \mathbf{s}_t = \tanh(\mathbf{a}_t) \\ \mathbf{o}_t = \mathbf{c} + \mathbf{V}\mathbf{s}_t \\ \mathbf{p}_t = softmax(\mathbf{o}_t) \end{cases} \tag{3}$$

The first two equations are the same of an Adaline with two input sources and the hyperbolic tangent as a non-linearity not allowing the future generation to diverge through

propagation. The third equation is new layer which prepares the processed information for the output. In general, the inner layer dimensionality $\mathbf{s}_t$ will be huger since it is in charge of keeping the information from the past events and hence carry more information than that needed in the output. In that sense, it resembles the structure of the inner layers of neural networks. The fourth equation gives a probabilistic interpretation to the outcome of the network which is of great relevance for most part of the applications. The definition of the softmax function is,

$$softmax(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{4}$$

Loss and gradient update rule

Inspired in the probabilistic representation of the output it is straightforward to define a loss function which for a given tupple ($\mathbf{x}_t$,$\mathbf{y}_t$) of training data points associates a loss,

$$L_t = -\log p_{t,\mathbf{y}_t} \tag{5}$$

where the second subindex indicates the position entry of the vector $\mathbf{p}_t$ corresponding to the label $\mathbf{y}_t$. The total loss for a sequence will be hence,

$$L = \sum_t L_t \tag{6}$$

In order to update the parameters it is necessary to compute the Back-Propagating Through Time gradient. The set of equations required for its computation are,

$$\begin{cases} (\nabla_{\mathbf{o}_t} L)_i = p_{t,i} - \mathbf{1}_{\mathbf{i},\mathbf{y_t}} \\ \nabla_{\mathbf{s}_t} L = \nabla_{\mathbf{s}_{t+1}} L \; diag(1-\mathbf{s}_{t+1}^2)\mathbf{W} + \nabla_{\mathbf{o}_t} L \mathbf{V} \end{cases} \quad \begin{cases} \nabla_{\mathbf{c}} L = \sum_t \nabla_{\mathbf{o}_t} L \\ \nabla_{\mathbf{b}} L = \sum_t \nabla_{\mathbf{s}_t} L \; diag(1-\mathbf{s}_t^2) \\ \nabla_{\mathbf{V}} L = \sum_t \nabla_{\mathbf{o}_t} L \mathbf{s}_t^\intercal \\ \nabla_{\mathbf{W}} L = \sum_t \nabla_{\mathbf{s}_t} L \; diag(1-\mathbf{s}_t^2) s_{t-1}^\intercal \end{cases} \tag{7}$$

The computation of the gradient for the learning scheme lets us see a potential a problem we will have to deal with. As can be seen there are chains of Jacobians which in in the case of long sequences may lead to divergent or vanishing behaviour depending if $||J||_\infty > 1$ or $||J||_\infty < 1$.

Variations

Up to this point we have described the essential components composing a RNN. In the following section we will describe additions and variations to the basic model in order to asses different problems.

End of sequences

If the RNN is actually going to be used to generate sequences, one must also incorporate in the output information allowing to stochastically decide when to stop generating new output elements. This can be achieved in various ways. In the case when the output is a symbol taken from a vocabulary, one can add a special symbol corresponding to the end of a sequence. When that symbol is generated, a complete sequence has been generated. The target for that special symbol occurs exactly once per sequence, as the last

target after the last output element.
Another kind of solution is to model the integer T itself, through any reasonable parametric distribution, and use the number of time steps left (and possibly the number of time steps since the beginning of the sequence) as extra inputs at each time step. Thus we would have decomposed $p(x_1..., x_T)$ into $p(T)$ and $p(x_1..., x_T|T)$ where the sequence will be conditioned to the length T.

RNNs to Represent Conditional Probability Distributions

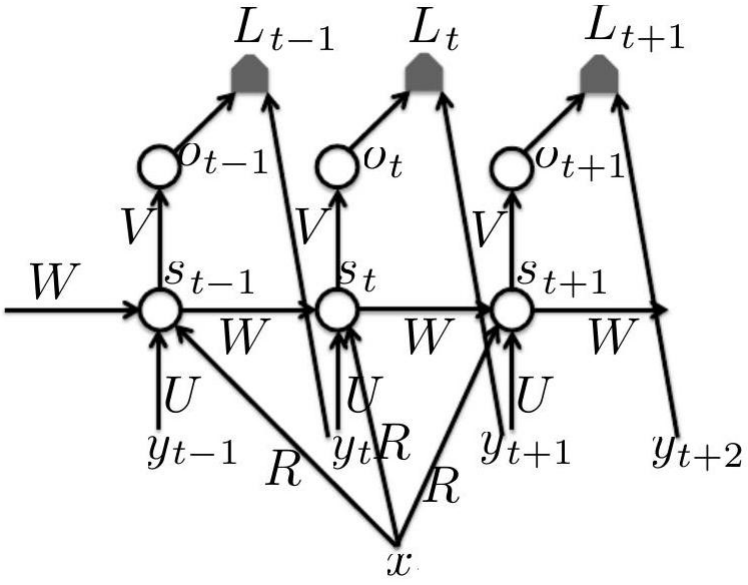
When we can represent a parametric probability distribution $p(y|\omega)$, we can make it conditional by making ω a function of the appropriate conditioning variable:

$$p(\mathbf{y}|\omega = f(\mathbf{x})) \tag{8}$$

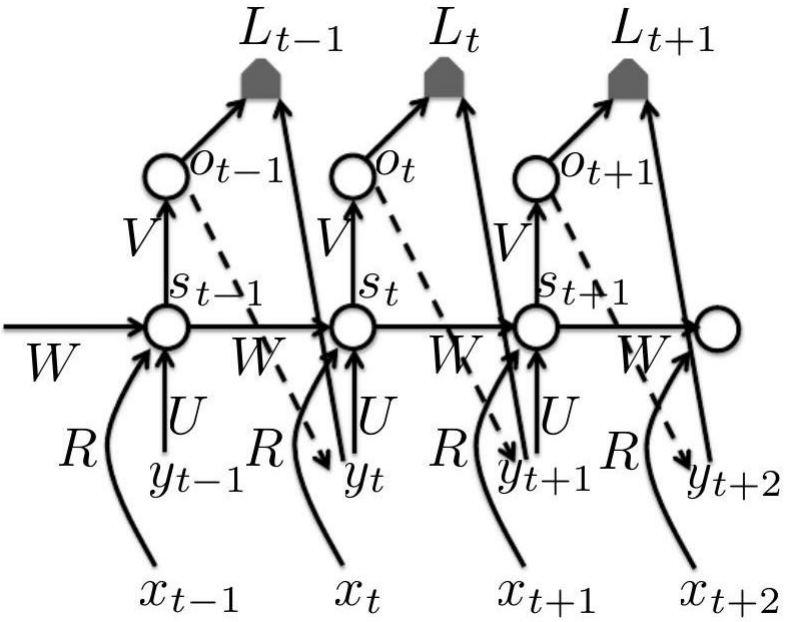
If $\mathbf{x}$ is a fixed-size vector, then we can simply make it an extra input of the RNN that generates the y sequence. Some common ways of providing an extra input to an RNN are:

- as an extra input at each time step, or
- as the initial state s 0 , or
- both.

As an example, we could imagine that $\mathbf{x}$ is encoding the identity of a phoneme and the identity of a speaker, and that $\mathbf{y}$ represents an acoustic sequence corresponding to that phoneme, as pronounced by that speaker.
Consider the case where the input x is a sequence of the same length as the output sequence $\mathbf{y}$, and the $\mathbf{y}_t$'s are independent of each other when the past input sequence is given, i.e., $p(\mathbf{y}_t|\mathbf{y}_{t-1}, ..., \mathbf{y}_1, \mathbf{x}) = p(\mathbf{y}_t|\mathbf{x}_t, \mathbf{x}_{t-1}, ..., \mathbf{x}_1)$. We therefore have a causal relationship between the $\mathbf{x}_t$'s and the predictions of the y_t's, in addition to the independence of the y_t's, given x. Under these (pretty strong) assumptions, we can interpret the t-th output $\mathbf{o}_t$ as parameters for a conditional distribution for $\mathbf{y}_t$, given $\mathbf{x}_t$, $\mathbf{x}_{t-1}$, ... , $\mathbf{x}_1$.



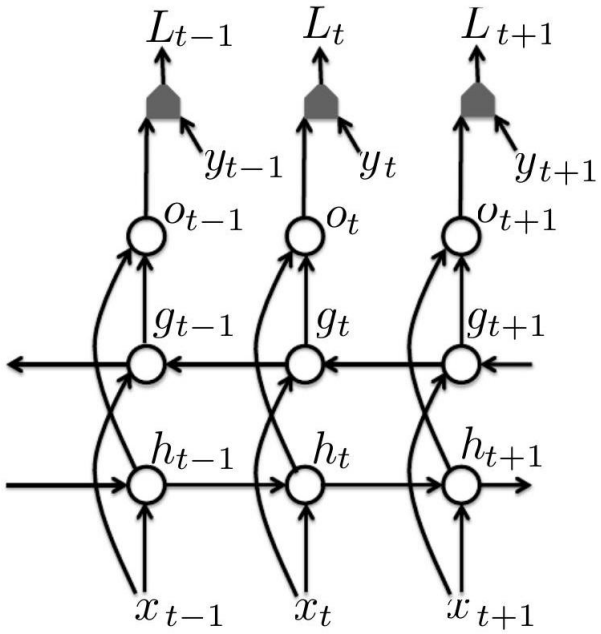
If we want to remove the conditional independence assumption, we can do so by making the past y_t's inputs into the state as well.



Recall that one of the variants of ending a sequence is a conditional probability of the length.

Bidirectional RNN

In many applications we want to output at time t a prediction regarding an output which may depend on the whole input sequence.
For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation and potentially may even depend on the next few words because of the linguistic dependencies between nearby words.

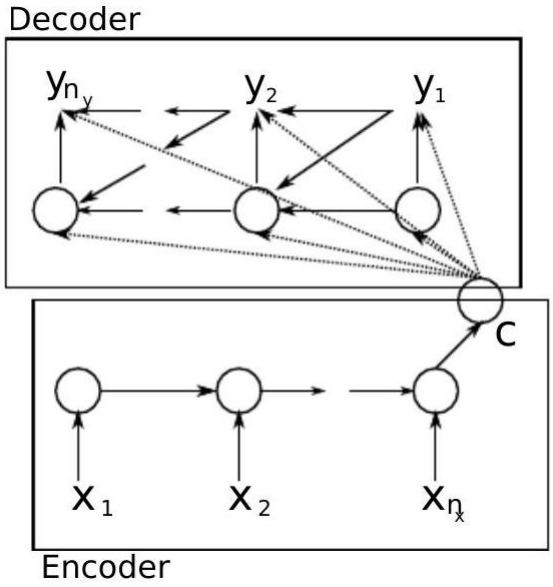


The basic idea behind bidirectional RNNs is to combine a forward-going RNN and a backward-going RNN. This allows the units $\mathbf{o}_t$ to compute a representation that depends

on both the past and the future but is most sensitive to the input values around time t, without having to specify a fixed-size window around t.

Encoder-Decoder Sequence-to-Sequence Architectures

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. The idea is very simple: (1) an encoder or reader or input RNN processes the input sequence, producing from its last hidden state a representation C of the input sequence $\mathbf{X} = (\mathbf{x}_1, ..., x_{n_x})$; (2) a decoder or writer or output RNN is conditioned on that fixed-length vector to generate the output sequence $Y = (y_1, ..., y_{n_y})$, where the lengths n_x and n_y can vary from training pair to training pair. The two RNNs are trained jointly to maximize the average of $logP(Y = Y|X = X)$ over all the training pairs (X, Y).



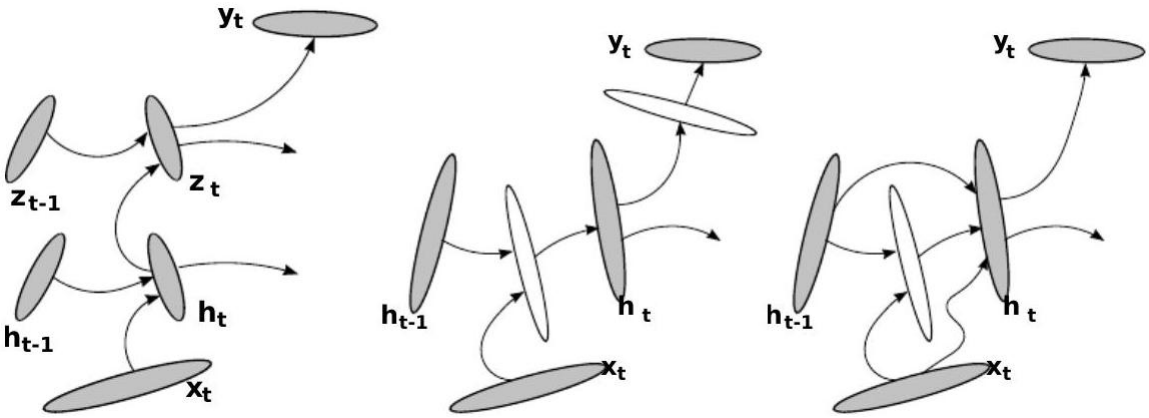
One clear limitation of this architecture is when the output of the encoder RNN has a dimension that is too small to properly summarize a long sequence.

Deep Recurrent Networks

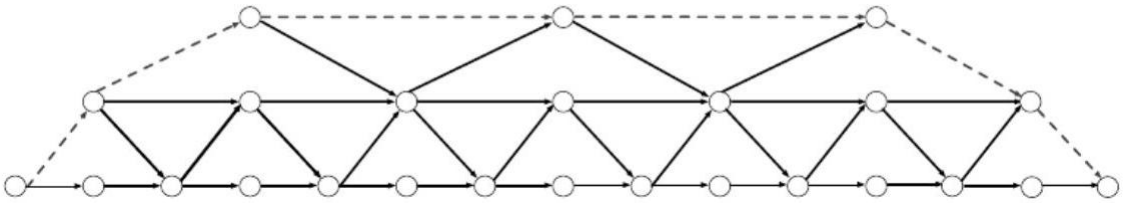
The computation in most RNNs can be decomposed into three blocks of param- eters and associated transformations:

- from input to hidden state,
- from previous hidden state to next hidden state, and
- from hidden state to output.

A recurrent neural network can be made deep in many ways. First, the hidden recurrent state can be broken down into groups organized hierarchically (left). Second, deeper computation (e.g., an MLP in the figure) can be introduced in the input-to-hidden, hidden-to-hidden, and hidden-to-output parts (Middle). However, this may lengthen the shortest path linking different time steps, but this can be mitigated by introduced skip connections (Right).



In the case of something-to-hidden, if there were no restriction at all and no pressure for some units to represent a slower time scale, then having N groups of M hidden units would be equivalent to having a single group of N M hidden units. Multiple time scales can be advantageous on several sequential learning tasks: each group of hidden unit is updated at a different multiple of the time step index e.g., at every time step, at every 2nd step, at every 4th step, etc. We can also think of the lower layers in this hierarchy as playing a role in transforming the raw input into a representation that is more appropriate, at the higher levels of the hidden state.



Long term dependencies

As introduced in the Vanilla section, a problem arises when training long sequences since they chain of Jacobians that naturally arises may provoke the vanishing or divergence of the gradient. In this section we will asses how to overcome those potential problems.

Divergence: Clipping gradient

The difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far, into a region where the objective function is larger, wasting a lot of the work that had been done to reach the current solution. This is because gradient descent is hinged on the assumption of small enough steps, and this assumption can easily be violated when the same learning rate is used for both the flatter parts and the steeper parts of the landscape.
One option is to clip the parameter gradient from a mini-batch element-wise just before the parameter update. Another is to clip the norm $||g||$ of the gradient g just before the parameter update:

$$if \;\; ||\mathbf{g}|| > v$$

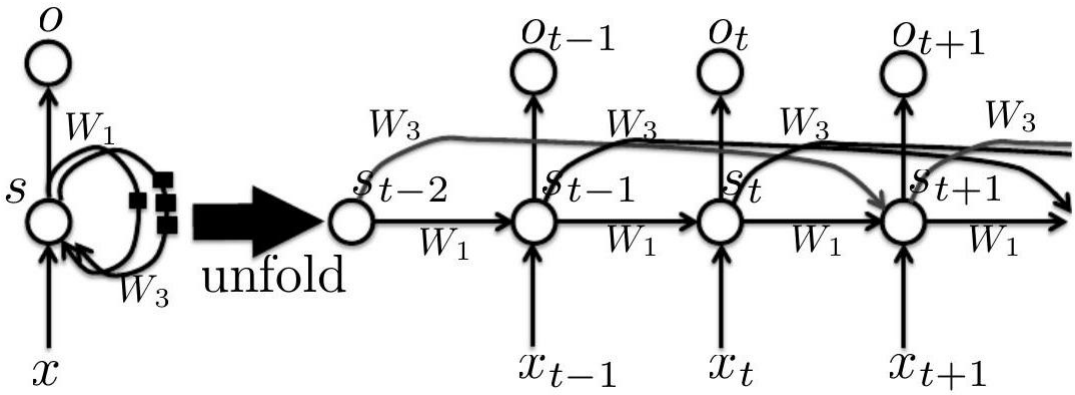
:

$$\mathbf{g} \leftarrow \frac{\mathbf{g}v}{||\mathbf{g}||} \tag{9}$$

It has also be shown that a random step in the case of a gradient that is surpasses the threshold also works fine.

Vanishing: Combining short and long paths

This idea has already come up when dealing with Deep Recurrent Networks. Gradients may vanish or explode exponentially with respect to the number of time steps. If we have recurrent connections with a time-delay of d, then instead of the vanishing or explosion going as $\mathcal{O}(\lambda^T)$ over T time steps (where λ is the largest eigenvalue of the Jacobians $\partial s_t/\partial s_{t-1}$), the unfolded recurrent network now has paths through which gradients grow as $\mathcal{O}(\lambda^{T/d})$ because the number of effective steps is T/d.



Vanishing: Leaky Units

A related idea in order to obtain paths on which the product of derivatives is close to 1 is to have units with linear self-connections and a weight near 1 on these connections. Depending on how close to 1 these self-connection weights are, information can travel forward and gradients backward with a different rate of "forgetting" or contraction to 0, i.e., a different time scale. The basic idea comes from $continuous - time$ RNNs such as,

$$\dot{\mathbf{s}}\tau = -s + \sigma(\mathbf{b} + \mathbf{W}\mathbf{s} + \mathbf{U}\mathbf{x}) \tag{10}$$

where the σ represents the non-linearity and τ the relaxation time. Discretizing this model leads to,

$$\mathbf{s}_t = (1 - \frac{1}{\tau})\mathbf{s}_{t-1} + \frac{1}{\tau}\sigma(\mathbf{b} + \mathbf{W}\mathbf{s}_{t-1} + \mathbf{U}\mathbf{x}_t) \tag{11}$$

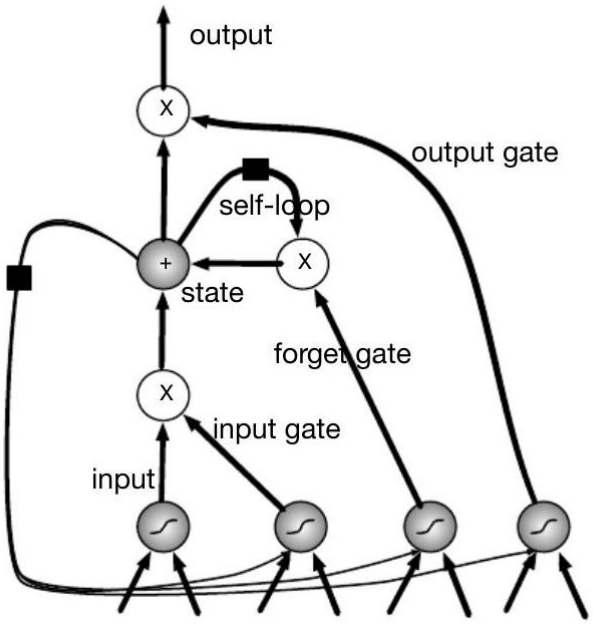
for $1 \leq \tau < \infty$. When $\tau = 1$, there is no linear self-recurrence, only the non-linear update which we find in ordinary recurrent networks. When $\tau > 1$, this linear recurrence allows gradients to propagate more easily. When τ is large, the state changes very slowly, integrating the past values associated with the input sequence. The extreme case where $\tau \to \infty$ because the leaky unit just averages contributions from the past, the contribution of each time step is equivalent and there is no associated vanishing or exploding effect. An alternative is to avoid the weightin front of the non-linear terms, thus making the state sum all the past values when ^{o}tau is large, instead of averaging them.

Long-Short-Term-Memory Architecture

Instead of a unit that simply applies a squashing function on the affine transformation of inputs and recurrent units, LSTM networks have "LSTM cells". Each cell has the same inputs and outputs as a Vanilla RNN, but has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $\mathbf{s}_t$ that has a linear self-loop similar to the leaky units described in the previous

section, but where the self-loop weight (or the associated time constant) is controlled by a forget gate unit, that sets this weight to a value between 0 and 1 via a sigmoid unit. Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit, and its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid non-linearity, while the input unit can have any squashing non-linearity. The state unit can also be used as extra input to the gating units. Schematically,

- Input: process the incoming information
- Input gate: makes the state ignore characteristics irrelevant of the input information
- Forget gate: makes the memory of the self-loop forget information not necessary any-more and adds the memory to the relevant information
- Output gate: removes information not necessary for the output purposes after it has been feedback the new information to memory and the next step



It is worth to note that a wide variety of variants has been developed with this concept of cells activated by logistic gates although the LSTM remains the most successful for generic purposes.