

Recurrent Neural Networks

Muchang Bahng

Spring 2023

Contents

1	Recurrent Neural Networks	2
1.1	Unidirectional RNNs	3
1.1.1	Loss Functions	4
1.1.2	Backpropagation Through Time	4
1.1.3	Stacked Unidirectional RNNs	5
1.2	Bidirectional RNNs	6
1.2.1	PyTorch Implementation	6
1.3	Long Short Term Memory (LSTMs)	7
1.3.1	Multilayer LSTMs	10
1.4	Gated Recurrent Units	11
	References	11

1 Recurrent Neural Networks

Let's focus on what is lacking in the vanilla feedforward neural net architecture. In a vanilla feedforward neural net architecture, we had a one to one map, where we take an input of fixed size and we map it to an output of fixed size. Perhaps we would want a one-to-many model, which takes in an image for example and outputs a variable-length description of the image. Or a many-to-many (e.g. machine translation from a sequence of words to a sequence of words) or many-to-one. Just as a convolutional neural network is specialized for processing a grid of values such as an image, a recurrent neural network is specialized for processing a sequence of values (e.g. audio, video, text, speech, time series).

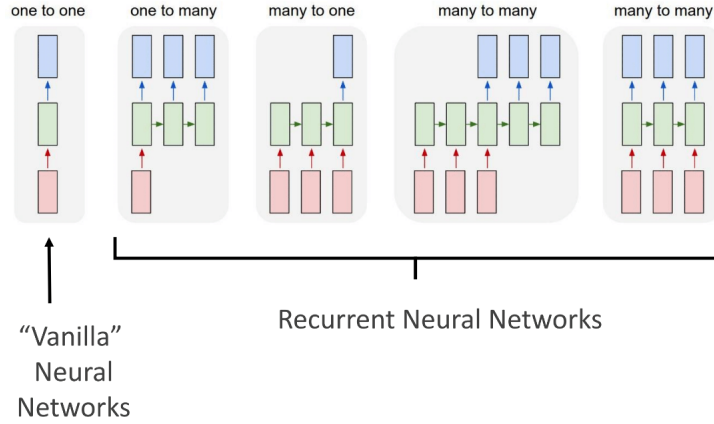


Figure 1: It is not limited to a fixed size of inputs and outputs.

Now to build such a model where the input or output elements are unbounded, we must take advantage of *weight sharing* (as seen in the CNN architecture) to control the size of our neural net. Furthermore, the fact that we should take in a sequence of inputs means that we may want to introduce some recursive structure in our neural net. Consider the classical form of a dynamical system driven by an external signal \mathbf{x} as

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta}) \quad (1)$$

which defines a recurrent relationship. Similarly, we can write \mathbf{h} to represent hidden neurons and write

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta}) \quad (2)$$

which indicates that the state of a hidden neuron is dependent on both the previous neuron and an input at time t . Through recursion, the hidden state \mathbf{h}_t contains all information about the inputs $\mathbf{x}_1, \dots, \mathbf{x}_t$ in the form of a complex function \mathbf{g} .

$$\mathbf{h}_t = \mathbf{g}_t(\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_1) \quad (3)$$

$$= f(\mathbf{h}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta}) \quad (4)$$

The fact that we can factorize \mathbf{g}_t into a repeated application of function \mathbf{f} gives us two advantages:

1. Regardless of the sequence length, the learned model always has the same input size because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.
2. It is possible to use the same transition function f with the same parameters at every time step. Since we do not have a growing number of parameters to optimize as our sequential data grows, training an RNN is still computationally feasible.

These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model \mathbf{g}_t for all possible time steps.

1.1 Unidirectional RNNs

A single layer unidirectional RNN is a direct application of the idea mentioned in the previous section. We can first look at its computational graph

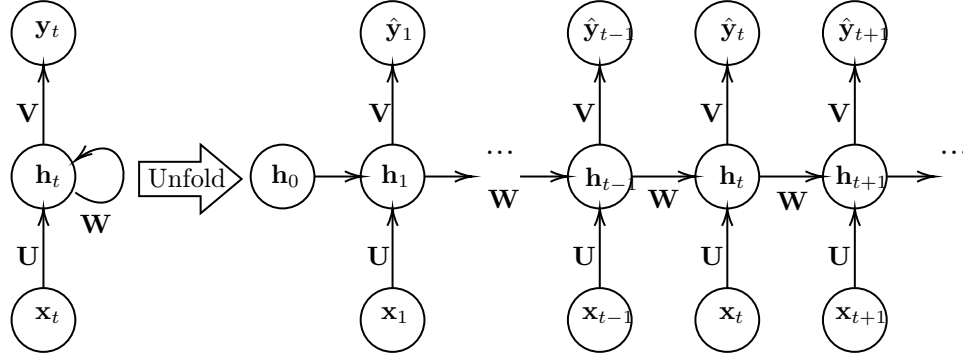


Figure 2: The circular arrow is notation for an RNN that “unfolds” to the graph on the right.

The activation functions that map to the hidden nodes and the outputs will be labeled σ_h and σ_y , respectively. In general the W will represent the left and right mappings between hidden nodes, the U will represent the map going up from the input or hidden node to a hidden node, and V is the final mapping from a hidden node to an output. We only label the arrows with the matrices, though a bias term and the nonlinear activation function are still there. That is, we can summarize our network as

$$\mathbf{h}_t = \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta}) = \sigma_h(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_h) \quad (5)$$

$$\mathbf{y}_t = \sigma_y(\mathbf{V}\mathbf{h}_t + \mathbf{b}_y) \quad (6)$$

for $t = 1, \dots, \tau$, where \mathbf{h}_0 is initialized to be zeroes or some small vector. The dimensions of the maps and the variables are listed for clarification:

1. $\mathbf{x}_t \in \mathbb{R}^d$ for all t
2. $\mathbf{h}_t \in \mathbb{R}^h$ for all t
3. $\mathbf{b}_h \in \mathbb{R}^h$
4. $\mathbf{U} \in \mathbb{R}^{h \times d}$
5. $\mathbf{W} \in \mathbb{R}^{h \times h}$

As we can see, the hidden node from the previous time step provides a form of memory, or context, that encodes earlier processing and informs the decisions to be made at later points in time. Adding this temporal dimension makes RNNs appear to be more complex than non-recurrent architectures, but in reality, they’re not all that different. Consider the rearranged architecture of an RNN below.

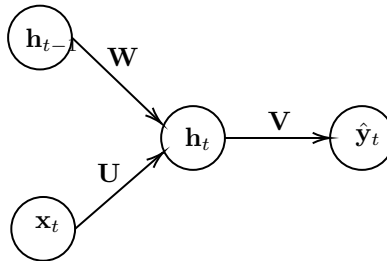


Figure 3: RNNs as an MLP.

1.1.1 Loss Functions

The form of the loss for a RNN will have to be slightly modified, since we can have multiple outputs. If we have a given input-output pair $\mathbf{x}^{(n)}, \mathbf{y}^{(n)}$, and we are interested producing a single output, then this is similar to what we already do with regular NNs. If we are interested in producing a sequence of outputs, then we can average the loss functions individually so that equal weight is placed on the prediction at each relevant timestep. This is called

$$L = \frac{1}{|T|} \sum_{t \in T} L_t \quad (7)$$

Sometimes, even with single inputs it may be good to include other intermediate terms in the loss so that we can direct the neural net to converge faster to what the correct answer should be.

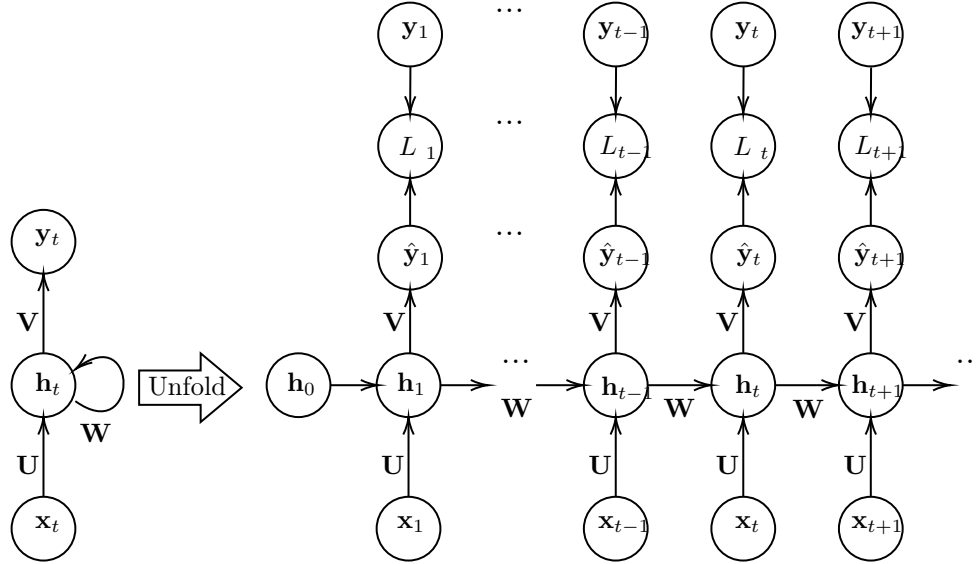


Figure 4

Note that one problem is that the errors can build up as the RNN predicts outcomes. For example, if we predicted $\mathbf{x}_1 \mapsto \hat{\mathbf{y}}_1$ we can compute the loss as $L_1(\mathbf{y}_1, \hat{\mathbf{y}}_1)$. However, there are two ways to compute the second loss: with inputs $L_2(\mathbf{x}_1, \mathbf{x}_2)$ or with $L_2(\mathbf{x}_1, \hat{\mathbf{y}}_1)$. One just uses the ground truth while the other uses the previous prediction for the next prediction, which can accumulate error. Both ways are feasible for loss computation, but it is generally done in the former way, called **teacher forcing**. This is analogous to a human student taking a multi-part exam where the answer to each part depends on the answer to the preceding part. Rather than grading every answer in the end, with the risk that the student fails every single part even though they only made a mistake in the first one, a teacher records the score for each individual part and then tells the student the correct answer, to be used in the next part.

1.1.2 Backpropagation Through Time

Now if we wanted to backpropagate through this RNN, we can compute

$$\frac{\partial L_t}{\partial \mathbf{W}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}} \quad (8)$$

where the first term depends on the specific form of the loss and the second is simply the matrix \mathbf{V} . This all looks the same as backpropagation for a MLP, but since \mathbf{W}_{hh} is used at multiple layers, we can reduce the third term in the equation to

$$\frac{\partial L_t}{\partial \mathbf{W}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \left(\sum_{k=1}^t \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}} \right) \quad (9)$$

where

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{i=k+1}^t \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \quad (10)$$

is computed as a multiplication of adjacent time steps. Now this can be very problematic, since if we have a lot of multiplications, then depending on the randomness of these matrices the gradient may be highly unstable, causing the vanishing or exploding gradient problem. We can elaborate on this a little further. Note that the hidden linear maps are known to be square matrices. We can expand out the derivative without the constant terms on the left as such:

$$\sum_{k=1}^t \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}} = \sum_{k=1}^t \prod_{k < i \leq t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}} \quad (11)$$

and we can see if at some point one of the $\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}$ tend to be small just from randomness, then their product for all coefficients where $k \leq j$ will be small too. This means that all the information, or memory, from the j th hidden state and before will vanish. In fact, if the spectrum (the set of eigenvalues and eigenvectors) is less than 1, then the multiplication of these derivatives will converge to a 0 matrix, and so we have an exponential memory loss throughout the network.

Furthermore, we can compute these gradients in batches by splitting up the corpus into several sentences, and sampling the sentences for gradient computation. Therefore, a forward or backward pass has a runtime complexity of $O(\tau)$ and cannot be reduced by parallelization because the forward propagation graph is inherently sequential. Each time step may only be computed after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\tau)$.

1.1.3 Stacked Unidirectional RNNs

Note that since we really have three matrices to optimize in the regular RNN, this may not be so robust. Therefore, we would like more hidden layers to capture further nonlinearities in an RNN, which is why we introduce a **stacked RNN** as shown below:

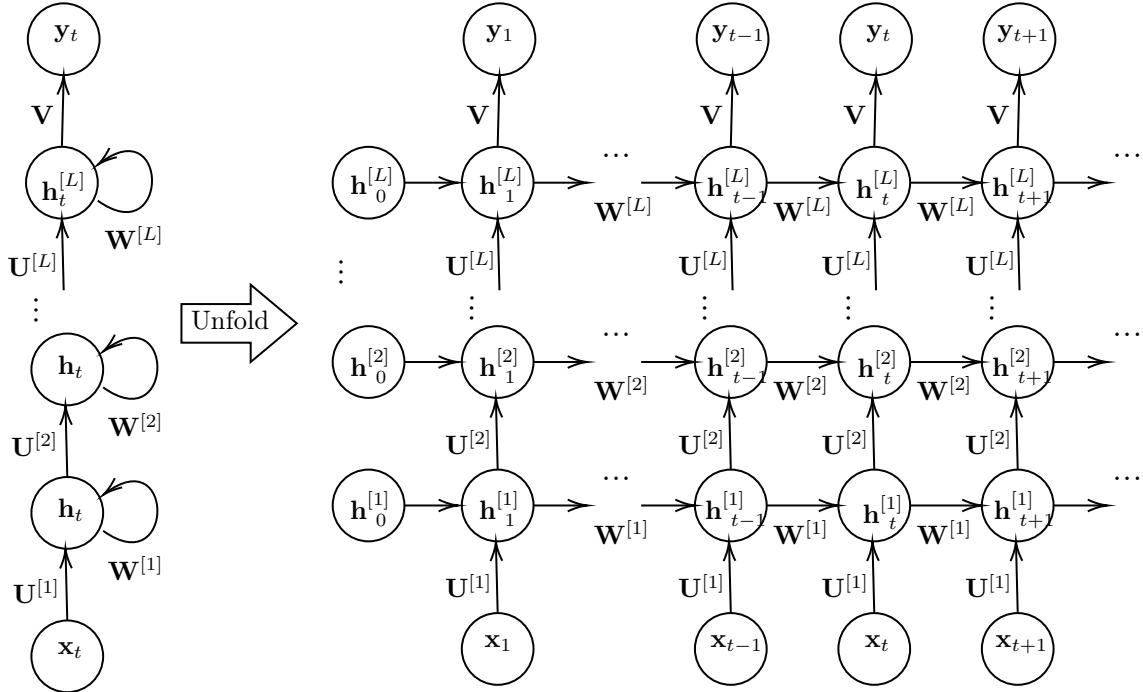


Figure 5: Stacked RNNs.

Now in this case, there are more layers of hidden nodes that an input must go through before it reaches the output node. We can expand out the computations as such, for $t = 1, \dots, \tau$, $l = 2, \dots, L$:

$$\mathbf{h}_t^{[1]} = \sigma_h(\mathbf{W}^{[1]}\mathbf{h}_{t-1}^{[1]} + \mathbf{U}^{[1]}\mathbf{x}_t + \mathbf{b}_h^{[1]}) \quad (12)$$

$$\mathbf{h}_t^{[l]} = \sigma_h(\mathbf{W}^{[l]}\mathbf{h}_{t-1}^{[l]} + \mathbf{U}^{[l]}\mathbf{x}_t + \mathbf{b}_h^{[l]}) \quad (13)$$

$$\mathbf{y}_t = \sigma_y(\mathbf{V}\mathbf{h}_t^{[L]} + \mathbf{b}_y^{[L]}) \quad (14)$$

or we could get rid of the first equation all together if we set $\mathbf{x}_t = \mathbf{h}_t^{[0]}$. Note that the hidden nodes $\mathbf{h}_t^{[l]}$ for all t and all $l \neq 0$ are all in \mathbb{R}^h , i.e. all hidden nodes will be h -dimensional. Therefore, most of the parameter matrices that we work with are square: $\mathbf{W}^{[l]} \in \mathbb{R}^{h \times h}$ and $\mathbf{U}^{[l]} \in \mathbb{R}^{h \times h}$ except for $\mathbf{U}^{[1]} \in \mathbb{R}^{h \times d}$.

1.2 Bidirectional RNNs

1.2.1 PyTorch Implementation

The implementation in PyTorch actually uses *two* bias terms $\mathbf{b}_{hW}^{[l]}$ and $\mathbf{b}_{hU}^{[l]}$ rather than just $\mathbf{b}_h^{[l]}$. This is technically not needed since the bias terms will just cancel out, but this is just how cuDNN (Cuda Deep Neural Network) is implemented.

$$\mathbf{h}_t^{[l]} = \sigma_h(\mathbf{W}^{[l]}\mathbf{h}_{t-1}^{[l]} + \mathbf{b}_{hW}^{[l]} + \mathbf{U}^{[l]}\mathbf{x}_t + \mathbf{b}_{hU}^{[l]}) \quad (15)$$

Let us look at a 2 layer RNN of sequence length 5. The input features will be set to 10, meaning that each $\mathbf{x} \in \mathbb{R}^{10}$. The hidden nodes will all be in \mathbb{R}^{20} .

```

1 input_features = 10
2 hidden_features = 20
3 num_layers = 2
4 sequence_length = 5
5
6 rnn = nn.RNN(input_features, hidden_features, num_layers)
7 input = torch.randn(sequence_length, input_features)
8 h0 = torch.randn(num_layers, hidden_features)
9 print(input.size(), h0.size())
10 # torch.Size([5, 10]) torch.Size([2, 20])
11
12 print([weight.data.size() for weights in rnn.all_weights for weight in weights])
13 # [torch.Size([20, 10]), torch.Size([20, 20]), torch.Size([20]), torch.Size([20]),
14 torch.Size([20, 20]), torch.Size([20, 20]), torch.Size([20]), torch.Size([20])]
15
16 output, hn = rnn(input, h0)
17 print(output.size(), hn.size())
18 # torch.Size([5, 20]) torch.Size([2, 20])

```

The corresponding diagram is shown below.

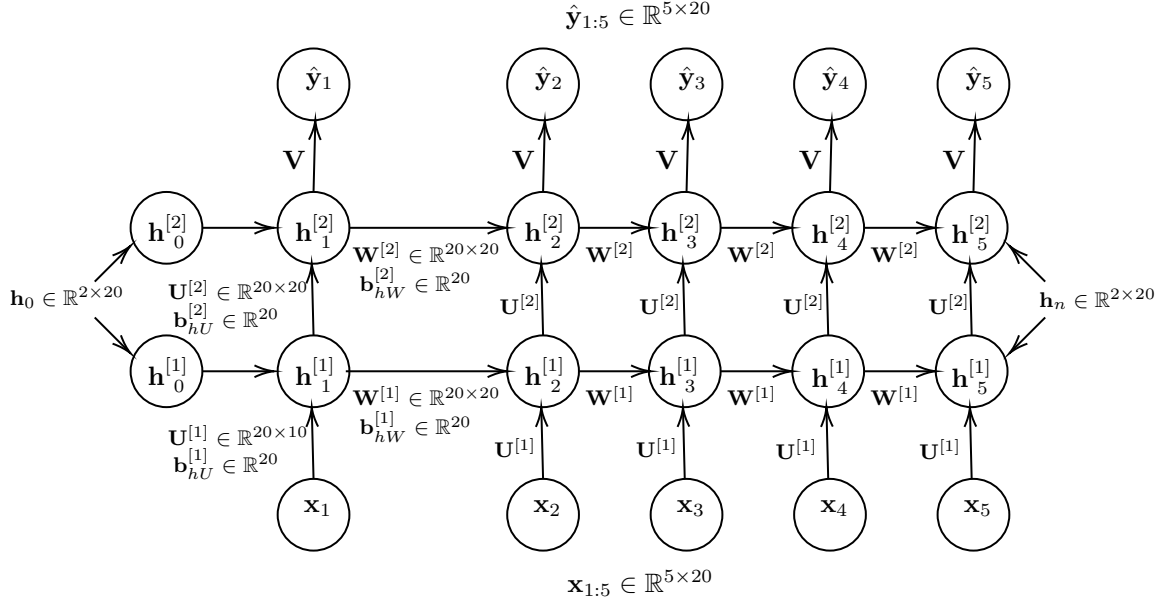


Figure 6

As we expect, there are 8 vectors/matrices we must optimize: $\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \mathbf{U}^{[1]}, \mathbf{U}^{[2]}, \mathbf{b}_{hU}^{[1]}, \mathbf{b}_{hU}^{[2]}, \mathbf{b}_{hW}^{[1]}, \mathbf{b}_{hW}^{[2]}$.

1.3 Long Short Term Memory (LSTMs)

In theory, RNNs are very beautiful and can be applied in all cases, but in practice they do not perform very well, mainly due to the vanishing/exploding gradient problem.

1. An exploding gradient is easy to fix, since we can just use the max-norm regularization, i.e. **gradient clipping**, to just set a max value for the gradients if they grow too large.
2. The **truncated backpropagation through time (TBPTT)** simply limits the number of times steps the signal can backpropagate after each forward pass, e.g. even if the sequence has 100 time steps, we may only backpropagate through 20 or so.
3. The **LSTM** model uses a memory cell for modeling long-range dependencies and avoids the vanishing gradient problems.

Historically LSTMs were used in achieving state-of-the-art results in 2013 through 2015, in tasks such as handwriting recognition, speech recognition, machine translation, parsing, and image captioning, as well as language models. They became to dominant approach for most NLP tasks, but in 2021, they have been overshadowed by transformer models, which we will talk about next.

LSTMs have a much more complicated unit to work with, so let's go through it slowly. Note that so far, a one-layer RNN consisted of recursive mappings of the form

$$(\mathbf{x}_t, \mathbf{h}_{t-1}) \mapsto (\mathbf{h}_t, \hat{\mathbf{y}}_t) \quad (16)$$

We can interpret the vector \mathbf{h}_{t-1} as the **short term memory**, or **hidden state**, that contains information used to predict the next output value. However, this can be corrupted (e.g. forgetting information from many steps ago), so we add an additional **long term memory**, or **cell state**, vector \mathbf{c}_t that should be preserved. Therefore, we have two arrows coming out of each hidden layer, as shown below in the one-layer LSTM.

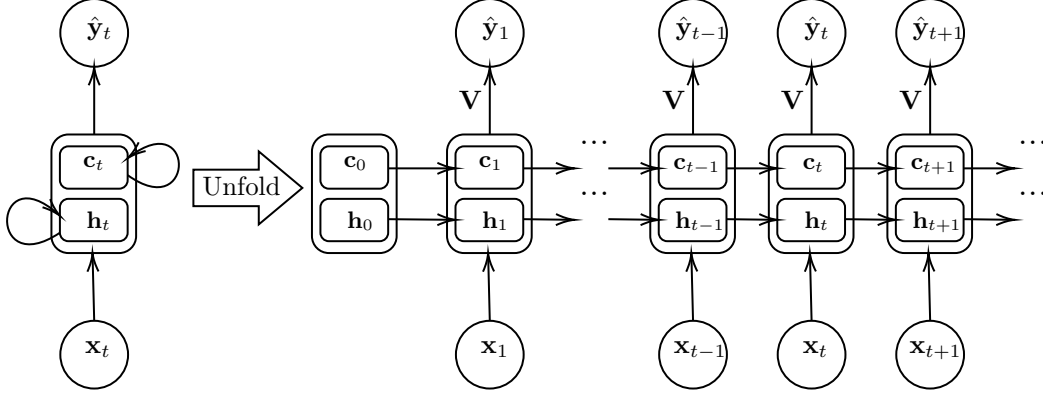


Figure 7

The mechanisms of the cell is quite complex, but the three basic steps are: (1) we forget a portion of the long term memory, (2) we add new long term memory, (3) we add new short term memory. Let us demonstrate this step by step. We are given three inputs: the previous long-term memory \mathbf{c}_{t-1} , the previous short-term memory \mathbf{h}_{t-1} , and the input at current time \mathbf{x}_t . In LSTMs, we only use the sigmoid and tanh activation functions, so we will denote them explicitly as σ and \tanh . For clarity, we will not write the matrix operations in the diagram anymore.

1. The **forget gate** (denoted by \mathbf{f}) takes an affine combination of \mathbf{h}_{t-1} and \mathbf{x}_t and puts it through the sigmoid activation function to generate a vector \mathbf{f}_t that has every element in $(0, 1)$. Then it element-wise multiplies it with \mathbf{c}_{t-1} , which essentially “forgets” a portion of the long-term memory.

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f) \quad (17)$$

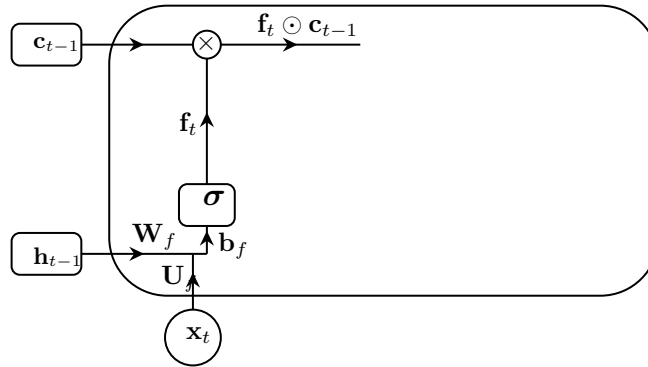


Figure 8

2. The **input gate** (denoted by \mathbf{i}) consists of two activations with the following operations.

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t + \mathbf{b}_i) \quad (18)$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c) \quad (19)$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \quad (20)$$

The layer \mathbf{i} can be seen as the filter that selects which information can pass through it and what information to be discarded. To create this layer, we pass the short-term memory and current input into a sigmoid function, which will transform the values to be between 0 and 1, indicating which information is unimportant. The second layer $\tilde{\mathbf{c}}$ takes the short term memory and current input and

uses the \tanh to transform the elements to be in $(-1,1)$, which allows us to add or subtract the necessary information from the long term memory.

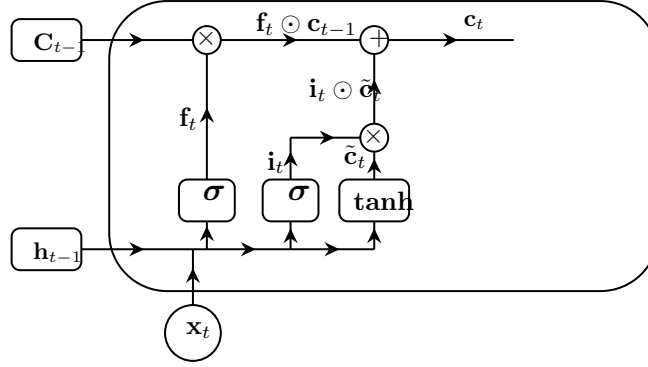


Figure 9

3. The **output gate** (denoted by o) consists of two activations with the following operations. This again creates a separate filter that selects the relevant information needed for the short term memory.

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{x}_t + \mathbf{b}_o) \quad (21)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad (22)$$

$$\hat{\mathbf{y}}_t = \sigma_y(\mathbf{V} \mathbf{h}_t + \mathbf{b}_y) \quad (23)$$

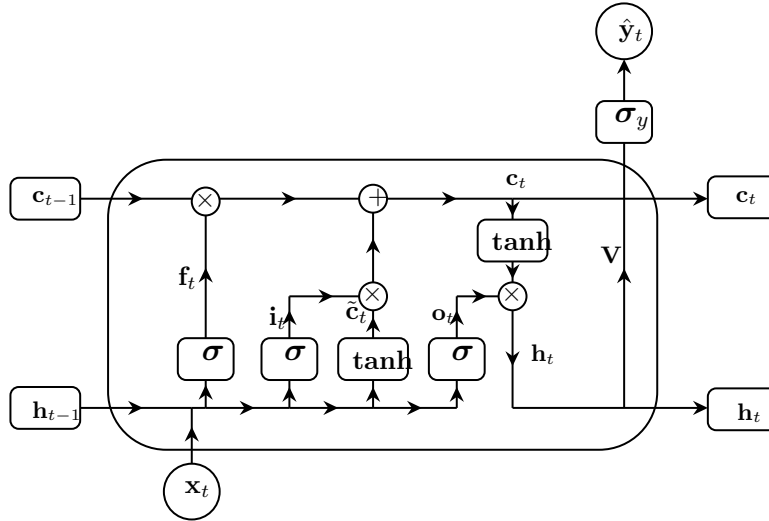


Figure 10

That is it! Now focusing on the cell state in the diagram above. Note that in order to go from cell state \mathbf{c}_{t-1} to \mathbf{c}_t , there was not a whole lot done to it. We really just multiply it once, which potentially deletes some content, and add it once, which adds new content, and we are done. The magic is this addition, since unlike multiplication, which can result in an exponential decay of knowledge, you are just constantly adding new numbers to update the storage, allowing the cell state to behave much more like RAM of a computer.

The LSTM architecture also makes it easier for the RNN to preserve information over many timesteps. For example, if the forget gate \mathbf{f}_t is set to $\mathbf{1}$ and the input gate set to $\mathbf{0}$, then the information of that cell is preserved indefinitely. In contrast, it's harder for a vanilla RNN to learn a recurrent weight matrix \mathbf{W} that

preserves information in the hidden state. In practice, a vanilla RNN would preserve memory up to maybe 7 timesteps (and increasing this is extremely difficult) while a LSTM would get about 100 timesteps, so in practice you should almost always just use a LSTM.

Unfortunately, LSTM doesn't *guarantee* that there is no vanishing or exploding gradients, but it does provide an easier way for the model to learn long-distance dependencies. Note that the gradient problem is not just a problem for RNNs; any neural architecture (including a feed-forward or convolutional) with very deep layers with multiple compositions of functions may suffer. Due to the chain rule and choice of nonlinearity function, these gradients can become vanishingly small and lower layers are learned very slowly. However, we can still implement residual connections to allow for more gradient flow such as ResNet, DenseNet, and HighwayNet.

1.3.1 Multilayer LSTMs

We can extend this architecture in the exactly same way for multilayer LSTMs. Note that we should be careful of the transformations each arrow represents. For the arrows going from $\mathbf{h}_t^{[l]} \mapsto \mathbf{h}_t^{[l+1]}$, there is no further transformation since we are just pushing this vector as an input to the next LSTM node. However, the arrow pushing from $\mathbf{c}_t^{[L]} \mapsto \hat{\mathbf{y}}_t$ does have an extra affine transformation with \mathbf{V} and \mathbf{b}_y , followed by some link function σ_y before we have the true prediction.

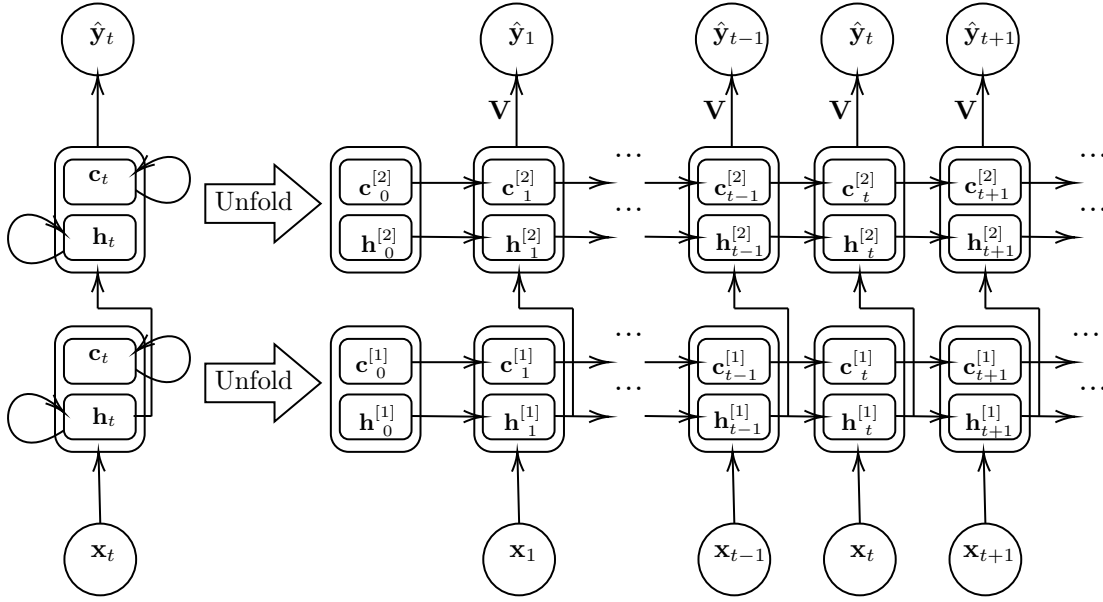


Figure 11

This follows the recursive equations, with $\mathbf{x}_t = \mathbf{h}_t^{[0]}$.

$$\text{Forget Gate } \begin{cases} \mathbf{f}_t^{[l]} = \sigma(\mathbf{W}_f^{[l]} \mathbf{h}_{t-1}^{[l]} + \mathbf{U}_f^{[l]} \mathbf{h}_t^{[l-1]} + \mathbf{b}_f^{[l]}) \end{cases} \quad (24)$$

$$\text{Input Gate } \begin{cases} \mathbf{i}_t^{[l]} = \sigma(\mathbf{W}_i^{[l]} \mathbf{h}_{t-1}^{[l]} + \mathbf{U}_i^{[l]} \mathbf{h}_t^{[l-1]} + \mathbf{b}_i^{[l]}) \\ \tilde{\mathbf{c}}_t^{[l]} = \tanh(\mathbf{W}_c^{[l]} \mathbf{h}_{t-1}^{[l]} + \mathbf{U}_c^{[l]} \mathbf{h}_t^{[l-1]} + \mathbf{b}_c^{[l]}) \\ \mathbf{c}_t^{[l]} = \mathbf{f}_t^{[l]} \odot \mathbf{c}_{t-1}^{[l]} + \mathbf{i}_t^{[l]} \odot \tilde{\mathbf{c}}_t^{[l]} \end{cases} \quad (25)$$

$$\text{Output Gate } \begin{cases} \mathbf{o}_t^{[l]} = \sigma(\mathbf{W}_o^{[l]} \mathbf{h}_{t-1}^{[l]} + \mathbf{U}_o^{[l]} \mathbf{h}_t^{[l-1]} + \mathbf{b}_o^{[l]}) \\ \mathbf{h}_t^{[l]} = \mathbf{o}_t^{[l]} \odot \tanh(\mathbf{c}_t^{[l]}) \end{cases} \quad (26)$$

$$\text{Output } \begin{cases} \hat{\mathbf{y}}_t = \sigma_y(\mathbf{V} \mathbf{h}_t^{[L]} + \mathbf{b}_y) \end{cases} \quad (27)$$

where

1. $\mathbf{x}_t \in \mathbb{R}^d$ for all t
2. $\mathbf{f}_t^{[l]}, \mathbf{i}_t^{[l]}, \mathbf{o}_t^{[l]} \in (0, 1)^h$
3. $\mathbf{h}_t^{[l]}, \tilde{\mathbf{c}}_t^{[l]} \in (-1, 1)^h$
4. $\mathbf{c}_t^{[l]} \in \mathbb{R}^h$

and we must optimize the parameters

$$(\mathbf{W}_f^{[l]}, \mathbf{U}_f^{[l]}, \mathbf{b}_f^{[l]}), (\mathbf{W}_i^{[l]}, \mathbf{U}_i^{[l]}, \mathbf{b}_i^{[l]}), (\mathbf{W}_c^{[l]}, \mathbf{U}_c^{[l]}, \mathbf{b}_c^{[l]}), (\mathbf{W}_o^{[l]}, \mathbf{U}_o^{[l]}, \mathbf{b}_o^{[l]}) \quad (28)$$

for $l = 1, \dots, L$. The fact that a LSTM uses the long term memory, in addition to the short term memory and the input, allows each cell to regulate the information to be kept or discarded at each time step before passing on the long-term and short-term information to the next cell. They can be trained to selectively remove any irrelevant information.

1.4 Gated Recurrent Units

References