CS 194-129 Deep Neural Networks
Spring 2018 Canny Lecture 9

1 Reading

10. Sequence Modeling: Recurrent and Recursive Nets

RNNs are a family of neural networks for processing a sequence of values $x^{(1)}, ..., x^{(\tau)}$. They can scale to much longer sequences than other networks; most variants can also process sequences of variable length. To do so, an RNN shares parameters across multiple time steps (e.g. for different positions in a sentence). Specifically, each member of the output is a function of previous members of the output, and this function is the same for every member. This recurrent formulation enables the sharing of parameters throughout the entire computational graph.

We will refer to RNNs as operating on a sequence containing vectors $x^{(t)}$ with the time step index t ranging from 1 to τ. (The time step need not specify actual time; it may refer only to the position in the sequence.) RNNs can be modeled as a dynamical system driven by an external signal $x^{(t)}$:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

where h is the state (i.e. the hidden units of the network) and θ are the parameters for f. Typical RNNs will also add extra architectural features such as output layers that read information out of the state h to make predictions.

The network typically learns $h^{(t)}$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t.

The learned model is specified in terms of a transition from one state to another state. Therefore, it is 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 $g^{(t)}$ for all possible time steps. This allows generalization to sequence lengths that did not appear in the training set.

We can design RNNs in many ways. One representative design involves an RNN that produces an output at each time step and has recurrent connections between hidden units. Note that any function computable by a Turing machine can be computed by such a recurrent network of a finite size.

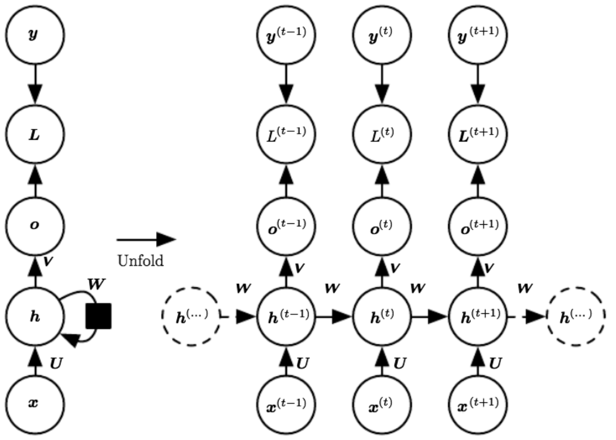

Figure 1: An RNN mapping an input sequence of x values to a corresponding sequence of output o values. L is the loss. The right graph is time-unfolded, such that each node is associated with one particular time instance. Source: *Deep Learning* by Goodfellow et al.

Bidirectional RNNs combine an RNN that moves forward through time, beginning from the start of the sequence, with another RNN that moves backward through time, beginning from the end of the sequence. This allows the network's predictions of $y^{(t)}$ to depend on the entire input sequence, which is useful in applications such as speech and handwriting recognition.

In a typical bidirectional RNN, each output unit $o^{(t)}$ can benefit from a relevant summary of the past in its $h^{(t)}$ input and from a relative summary of the future in its $g^{(t)}$ input. Here, $h^{(t)}$ stands for the sub-RNN that moves forward through time and $g^{(t)}$ stands for the state of the sub-RNN that moves backward through time. (Although they take input from both the past and the future, output units will be most sensitive to input values around time t.)

Sequence-to-Sequence

Sometimes we might want our RNNs to map an input sequence to an output sequence that is not necessarily of the same length. This comes up in applications such as speech recognition and question answering.

Some terminology: we call the input to the RNN the *context*. We want to produce a representation of this context, C. The context C might be a vector or sequence of vectors that summarize the input sequence $X = (x^{(1)}, ..., x^{(n_x)})$.

The simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence is the **encoder-decoder (sequence-to-sequence)** architecture. The idea is simple. First, an encoder RNN processes the input sequence, emitting the context C as a simple function of its final hidden state. Next, a decoder RNN is conditioned on that fixed-length vector to generate the output sequence $Y = (y^{(1)}, ..., y^{(n_y)})$. The innovation is that the lengths n_x and n_y can vary from each other.

The two RNNs are trained jointly to maximize the average of $\log P(y^{(1)}, ..., y^{(n_y)} \mid x^{(1)}, ..., x^{(n_x)})$ over all the pairs of x and y sequences in the training set. The last state h_{n_x} of the encoder RNN is typically used as a representation C of the input sequence that is provided as input to the decoder RNN.

Deep Recurrent Networks

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

1. from the input to the hidden state
2. from the previous hidden state to the next hidden state
3. from the hidden state to the output

Generally, each block is associated with a single weight matrix and corresponds to a single layer in a deep network (a learned affine transformation followed by a fixed nonlinearity). It is likely advantageous to add depth to these mappings.

Strategies for Multiple Time Scales

One problem with long-term dependencies in RNNs is that gradients propagated over many stages tend to either vanish or explode. We can deal with this by designing model that operates at multiple time scales, s.t. some parts of the model operate at fine-grained time scales and can handle small details, and others operate at coarse time scales and transfer information from the distant past to the present more efficiently.

We might obtain coarse time scales by adding direct connections from variables in the distant past to variables in the present (**skip connections**). In an ordinary RNN, a recurrent connection goes from a unit at time t to a unit at time $t + 1$, but it is possible to construct RNNs with longer delays. This helps, since gradients tend to vanish or explode exponentially with respect to the *number of time steps*.

Another way to obtain paths on which the product of derivatives is close to 1 is to have units with linear self-connections (**leaky units**) and a weight near one on these connections.

LSTM and Other Gated RNNs

The most effective sequence models have often been **gated RNNs**, which include the **long short-term memory** and networks based on the **gated recurrent unit**. Gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode. They are a generalization of leaky units for which the connection weights can change at each time step.

Leaky units allow networks to accumulate information over a long duration. However, we might want to forget some of the old state after it has been used. In gated RNNs, we let the neural network decide when to forget things (i.e. set the old state to zero).

The LSTM model introduces self-loops to produce paths where the gradient can flow for long durations. Note that the weight on the self-loop is not fixed, but conditioned on the context. Furthermore, the weight of the self-loop is gated (controlled by another hidden unit), so the time scale of integration can be changed based on the input sequence. (The time constraints are output by the model itself.)

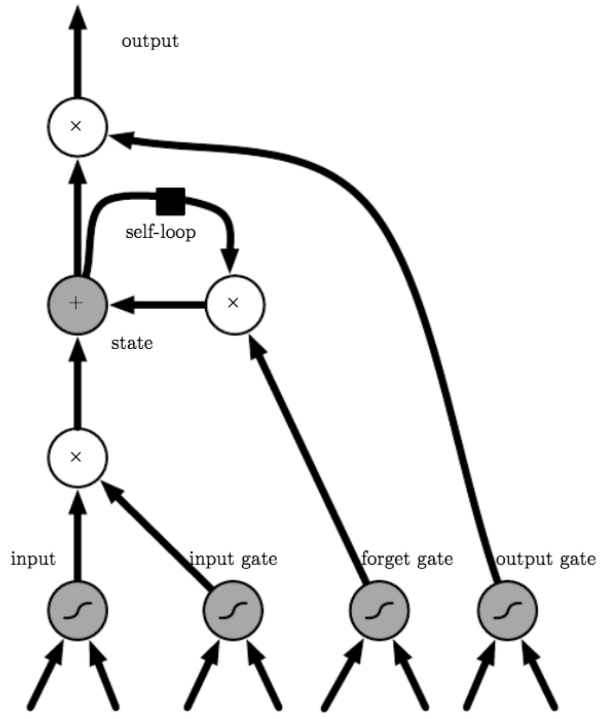

Figure 2: An LSTM cell. Cells are connected recurrently to each other, replacing the usual hidden units of RNNs. An input feature can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All gating units have a sigmoid nonlinearity, while the input unit can have any kind of squashing nonlinearity. The black square indicates a delay of a single time step. Source: *Deep Learning* by Goodfellow et al.

Instead of a unit that simply applies an element-wise nonlinearity to the affine transformation of inputs and recurrent units, LSTM RNNs have "LSTM cells" with an internal recurrence (a self-loop) in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information.

LSTM networks have been shown to learn long-term dependencies more easily than simpler architectures.

Understanding LSTM Networks

RNNs can use past events in a series to reason about future events in a series. They are networks with loops in them, which allows information to persist. In other words, they have some semblance of memory.

An RNN can be thought of as multiple copies of the same network, each passing a message to a successor. Below is a block diagram of a basic RNN (from Christopher Olah's blog, which also serves as the source for these notes).

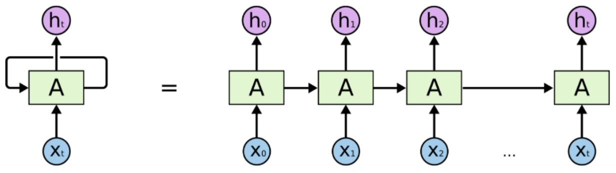

Figure 3: An unrolled RNN. Source: "Understanding LSTM Networks" by Christopher Olah.

In practice, we use LSTMs – a special kind of RNN which works much better than the standard version. You may ask: what's wrong with the standard version? Well, sometimes the gap between relevant information and the point where it is needed is very large (think of an essay where the author mentions he's from France at the beginning, and then much later writes "I speak fluent..." where we want to predict the next word in the sentence). As that gap grows (s.t. we have a long-term dependency), RNNs become unable to learn to connect the information.

LSTMs, on the other hand, *can* learn long-term dependencies. Hooray! In fact, they are specifically designed to remember information for long periods of time.

All RNNs have the form of a chain of repeating modules of neural network. An LSTM's repeating module contains four neural network layers which interact in a very special way.

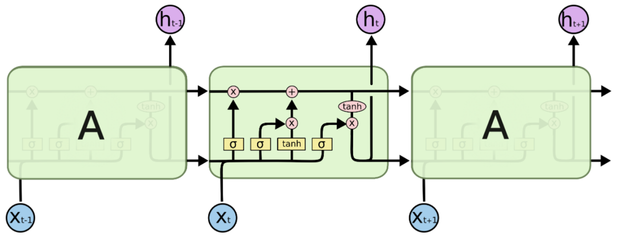

Figure 4: LSTM block diagram. The yellow boxes are learned neural network layers. The pink circles are pointwise operations. Source: "Understanding LSTM Networks" by Christopher Olah.

The key to LSTMs is the cell state – the horizontal line running through the top of the diagram. This is a kind of conveyor belt along which information can flow. The LSTM will add or remove information to/from the cell state, and this process will be carefully regulated by structures called gates.

Gates are designed to optionally let information through. They are composed of a sigmoid neural network layer and a pointwise multiplication operation. The sigmoid outputs numbers between 0 and 1, which describes how much of each component should be let through.

In an LSTM, the first step is to decide which information we should throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer," which looks at h_{t-1} and x_t and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

The next step is to decide what new information we will store in the cell state. This involves a sigmoid layer called the "input gate layer," which decides which values we'll update, and a tanh layer which creates a vector of new candidate values $\tilde{C}_t$ for potential addition to the state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_C)$$

To make the update from C_{t-1} (the old cell state) to C_t (the new cell state), we first multiply the old state by f_t, forgetting the things we decided to forget earlier. We then add $i_t * \tilde{C}_t$ (the new candidate values, scaled by how much we decided to update each state value).

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then we squash the cell state through a tanh and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

$$\begin{aligned}o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)\\ h_t &= o_t * \tanh(C_t)\end{aligned}$$

So far, this has described a pretty standard LSTM. However, note that there are many variants of LSTMs, and most look slightly different!

2 Lecture

Recurrent Networks

So far we've been focusing on directed acyclic networks, which have a topological ordering that is used for activation propagation and gradient backpropagation.

Recurrent networks (RNNs) introduce a time dependence and a notion of recursion. Now we'll have a unit with two inputs x_t and h_{t-1} (hidden state from the previous time step), and two outputs y_t and h_t (hidden state from the current time step). In other words, the hidden state will come back with a one-step delay and be aligned with the next step of x.

RNNs are an amazingly versatile and powerful tool for sequential data (e.g. text or moving image data). Just as CNNs are designed to process a grid of values, RNNs are designed to process streams of data $x_1, ..., x_n$. They are also capable of producing streams of outputs $y_1, ..., y_m$.

We can't really process cyclic networks, but we can unroll them in time – which produces an acyclic network from the cyclic network. This is necessary, since backpropagation requires an acyclic network. Note that parameters are shared within the acyclic network; as with any network with tied states, the gradient will be summed across all copies of the same weight. Therefore, we'll take a gradient from each node and add it to the cumulative gradient for the set of parameters.

We can also make layered (deep) RNNs, which essentially stacks RNNs on top of each other. Each output becomes an input to the next layer, and each layer shares the same parameters. As we go from one RNN to the next, we ascend to a higher level of features.

We will process our sequence of vectors x by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

where h_t is the new state, f_W is some function with learnable parameters W, h_{t-1} is the old state, and x_t is the input vector at some time step. This formula is fixed for all t.

In the simplest case, the RNN looks something like this:

$$\begin{aligned}h_t &= f_W(h_{t-1}, x_t) = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)\\ y_t &= W_{hy}h_t\end{aligned}$$

There are three sets of weights: incoming hidden weights, incoming x weights, and the output weights. Generally, W_{hh} is square while W_{xh} is not.

Example. In order to perform image captioning, we can take a CNN image classifier, adjust the dimension of its output, and feed it into an RNN as a hidden vector into the first recurrent unit (along with the input and output linear maps). We can then take the outputs and feed them in as next inputs so the RNN can recursively generate an English description (by repeatedly predicting the next word in the sequence). Note: the RNN in the slides is word-based rather than character-based.

Example. One application of RNNs is in visual attention. For example, we can have an RNN attend spatially to different parts of images while generating each word of a sentence. This mimics the way human beings process visual input. We have a localized visual system that sees only a relatively small region at high resolution, and then we typically scan around while interpreting scenes.

People have found that it helps to do the same thing with input images, especially for generating a text description. In other words, it's helpful if, while referring within a caption to a man, the network is actually looking at the man. Therefore, on top of a caption, we might want our networks to output an attention map (a set of weights in $[0, 1]$ at fairly low resolution across the image) which will act as a mask and resemble the visual field.

The point is, *recurrent neural networks can learn to focus their attention.*

LSTMs

RNNs are very powerful already, but they have a serious limitation: they have a lot of trouble with long-term dependencies. The issues here are similar to the ones that made it difficult for very deep convolutional networks to train. Once you start trying to propagate gradients over more than 20 units or so, it just doesn't seem to work ("there are too many degrees of freedom").

We saw that *residual networks* were able to train and keep improving with depth, as a result of giving them a hint (in the form of residual connections) that the identity would be a good approximate function between two layers.

Therefore, it might makes sense to do that with recurrent networks. We might want to give the recurrent unit something that's a gated identity map from some input to its output. This is the approach we take with **LSTMs**. This is called a memory cell, and it's basically an elementwise mapping from input to output. We will choose to remember either everything, nothing, or anything in between.

Unlike the simple recurrent units, there is no linear transformation (we can't mix memories around). However, we *can* change the cell elements in magnitude and add new memories. Therefore, in terms of going from cell input C_{t-1} to cell output C_t, we have an identity map with at most a scale applied to it.

On the other hand, between the hidden states h_{t-1} and h_t, our memory cells can at least do as much as the general RNN. (The hidden states are definitely still computed recurrently.) Holistically speaking, the cell now has a semblance of memory which works better with long-term dependencies.

Anyway, we now have two recurrent nodes: c_i and h_i.

- c_i is the cell's memory, and undergoes no transform (only elementwise forgetting). There is no linear transformation so that the cell can remember things over a long period of time.
- h_i plays the role of the output in the simple RNN, and is recurrent. For stacked arrays, the hidden layers (h_is) become the inputs (x_is) for the layer above.

Let's break an LSTM down into three parts. The following are all applied to the cell state c, which travels through on a line to the next unit in the chain, and also to the combination of h and x.

- **Decide what to forget.** The forget function f has a sigmoid applied to a linear map (its own weight matrix) applied to the left and below inputs. Therefore, its output will be in $[0, 1]$ – it will either turn fully on or turn off the memory of that memory line.
- **Decide what new things to remember (i.e. add to the cell state).** The next group of nodes are i, a sigmoid gate, and g (the former output from the recurrent layer), a hyperbolic tangent.
- **Decide what to output.** Finally, we have output generation. The output h is a tanh function of the cell's memory, including the additional state from the previous step. It is limited to $[-1, 1]$ and gated by o (another sigmoid).

h and c are both passed to the next stage. However, h is the output, so it also goes up to the next layer (if there is one). See the diagram on the slides for more details.

LSTMs are pretty much just sigmoids and linear weights! So they train well with backpropagation. In fact, they are easier to train than a general recurrent network. This is by design: in order to handle long-term dependencies, we wanted to prevent the gradients from vanishing or exploding. LSTMs accomplish this by constraining the cell value c (and the backpropagated gradient of c) to grow only linearly with time.

Note: The LSTM is at least as powerful as the RNN. An LSTM tends to be good at using its cell state for long-term memory, instead of trying to remember h values over long distances like the RNN.

LSTM Variants

We can combine the hidden state h and cell memory c into one vector. Also, instead of having both a forget value f and a remember value i, we can simply define i as $1 - f$. (We will either choose to forget and add something new, or retain the current information.)

References

[1] Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`

[2] Understanding LSTM Networks, `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`