Recurrent Neural Networks

Eilif Solberg

12.09.2018

Contents

1 Introduction

1.1 The dimension of time

We have so far seen ways of dealing with problems of finding functions $f\colon X \to Y$, where X and Y have had fixed sizes. We have also seen situations where X have had a spatial dimension associated with it, e.g. through

image data. In the case of image segmentation we have in addition seen spatial dimensions associated with Y. Through *parameter sharing* in convolutional networks we have to some extent been able to handle the challenge of variable-sized X and Y data. Now we will take a closer look at the situation where the data are arranged in a *sequence*, i.e. a linearly ordered set. The index dimension in the sequence will in many cases correspond to, or be closely associated with, *time*.

Sequential data is everywhere. When we look at the world around us, we not only capture independent glimpses, but observe phenomena as *movement* and *actions*. Some phenomena are in fact *only* meaningful when taking the time dimension into consideration. Sampling a sound wave at a single point in time is not very useful, only when integrating information over a window of time are we able to recognize the sound of an ambulance behind us or convey the meaning of speech. For humans, not only does our input arrive in a sequence, but our actions and responses are also spread out in time. Indeed it seems that our whole lives are a more or less continuous *stream* of inputs and outputs. We have a *representation* of the world around us which we continuously *update* based on *new information*. Based on our representation of the world and our goals and motivation we may also *act* upon it, possibly influencing the new information that we acquire.

We will now try to formalize this into a model we can work with. We shall use a *discrete* notion of time, introducing a sequence $t = 1, 2, \ldots, T$.

- Let $S \in \mathbb{R}^d$ represent our *state*. The state contains our current representation of the world around us. As this will change over time we add a superscript to it, so that S^t represents the state at time t.
- Let $X^t \in \mathbb{R}^m$ denote the input at time t, and $X = (X^1, \ldots, X^T)$ be the input sequence.
- Let $Y^t \in \mathbb{R}^n$ denote the output at time t, and $Y = (Y^1, \ldots, Y^T)$ be the output sequence.

The input sequence is given to us, but how are the outputs produced? Output decisions should be made based on our current state. We denote the *output* function by f, and we have

$$Y^t = f(S^t) \tag{1}$$

How do we update our beliefs as we move along? We define the *update* function h, to be a function of the new information at the current time step

X^t, our previous state S^{t-1} and also the output at the previous time step Y^{t-1}. We have

$$S^t = h(X^t, S^{t-1}, Y^{t-1}) \tag{2}$$

It may seem superflous to have Y^{t-1} as an input, as from equation (1) we have that this is just itself a function of S^{t-1}. One reason we might want to include it though is that f could potentially be a *random*, i.e. nondeterministic, function of S^{t-1}. If we are e.g. in the middle of a sentence, there might be many possible completions that will convey our message. Each of these completions will have a possibly different next word associated with it. So f ends up drawing from this distribution over words, and it will be useful for us to know which word eventually got picked. Even for deterministic f, it might still be easier to just feed the output directly as input in the next time step, rather then trying to figure this out by somehow learning a replica of the output function in the update function.

Note that equation (2) describes the general situation, there are situations where not all of the inputs to h are present. It is not uncommon to only have an output at the last step, genre classification of a song would be an example. We call this a **sequence-to-vector** RNN. Even when there is an output there are situations where we may not want to feed it back into the network. One such case would be if the output elements $Y^1, \ldots, Y^t$ are *conditionally independent* given the input $X^1, \ldots, X^t$. This in particular means that we can model the conditional distribution of each of the Y^t values as a function of $X^1, \ldots, X^t$ only.

Another special case worth mentioning, is cases where we only have a single input element X^1. It may seem strange to use a recurrent neural network in this case, as it appears we are not dealing with sequence data at all. However we can have sequence in the *output* data even for static inputs. Describing an image (called *image captioning* in the machine learning literature) would be such an example. We denote this special case as a **vector-to-sequence** RNN. Note that we in these situations might still feed in X^1 at each time step so that we don't have to store everything that we need from X^1 in our state vector.

Lastly, we may have that both the input and output are sequence, but where they may be of different lengths. Speech interpretation and machine translation are examples of this.

2 RNN

2.1 Basic RNN

So far we have been looking very abstractly at the problem, but how would we go about actually implementing the update function h? Assume our state s is of dimension d, and the input x and output y are of dimensions m and n respectively. A simple choice would be to have just a linear[1] function followed by an activation function, i.e.

$$h(x, s, y) = a(Ux + Vs + Wy + b) \tag{3}$$

where $U \in \mathbb{R}^{d \times m}$, $V \in \mathbb{R}^{d \times d}$ and $W \in \mathbb{R}^{d \times n}$ are all matrices and b is a vector of dimension d. Tanh, sigmoid or ReLU are all common choices for the activation function a and are applied elementwise. Note that the above formulation is equivalent to concatenating the weight matrices U, V, W into the matrix M and applying this to the concatenated vector [x, s, y], i.e. $h(x, s, y) = a(M[x, s, y] + b)$. We shall call the representation given by (3) above is the basic RNN model.

Note that the form in (3) need not be followed strictly. Clearly x need not be the raw data, but any preprocessing may have been applied to it. Furthermore, if the preprocessing function is differentiable we could learn this as well in an end-to-end fashion by passing the gradients on. If our inputs are e.g. images it will in many cases be natural to first transform the pixel values into a semantically more meaningful representation using a convolutional neural network. In addition we allow for a, possibly learnable, preprocessing of the output y. If we are e.g. generating sentences, one word at a time, y^t may be encoded as a one-hot vector or a soft distribution over words. In this case applying a *word embedding* to y would be logical. The idea is the same in both cases; transforming the data into a domain where meaningful decisions can be made with a simple linear function. Note that applying a similiar preprocessing to s does not make sense in the same way, s is *already* assumed to be at the right level of abstraction.

2.2 Gradients of RNN

The equations in (3) tells us for given weight matrices and a sequence of inputs $x_1, \ldots, x_\tau$, how to compute a sequence of outputs $y_1, \ldots, y_\tau$. How do we get the appropriate weight matrices in the first place? This will

[1] We will use the term *linear* even if we have a bias term, where *affine* would really be the correct term

of course depend on the particular problem setting, but as usual or go-to method will be to turn the problem into an optimization problem and then use some sort of gradient descent. How can we find the gradients for RNNs though? Now we suddenly have loops in our graphs, and these are bound to cause us some problems, right? It turns out they do not. If we look at the graph that unfolds when we add a new layer for each step in our computation, we just get an instance of a deep *feed-forward* network. We call this the *computational graph* or the *unrolled graph*. Note that the depth of this graph may be *random*, and could thus be different between samples. There is nothing wrong with that from a theoretical perspective, but it will cause us some headaches when trying to batch samples during training and inference.

Note that the same parameter may appear several times in the unrolled graph, indeed exactly once for each time step. This does not really complicate things much though, we deal with this in the same way as with other forms of parameter sharing; by summing over the derivatives of all the replicas. For the interested reader we will here lay out the basic argument for why this is.

2.2.1 Derivative of a shared parameter

Let f be a real-valued function of a parameter λ occuring K times in f. Let $\lambda_1, \lambda_2, \dots \lambda_K$ denote the different occurences of λ in f, and assume that we have already calculated $\frac{\partial}{\partial \lambda_i} f$ for $i = 1, \dots, K$. Each of these derivatives tells us how much change in f we can expect per unit change in λ_i, for small changes to λ_i. If we change λ by a small amount, say δ, all replicas λ_i will change by δ. It is perhaps not surprising that the total change in f will be close to the sum of all the changes that each λ_i causes individually. If this is exact in the limit for small δ we would have that the derivative with respect to λ is the sum of the derivatives with respect to all the replicas. Indeed it is so. Mathematically we have

$$\begin{aligned}
\frac{d}{d\lambda} f(\lambda) &= \lim_{\delta \to 0} \frac{f(\lambda + \delta) - f(\lambda)}{\delta} \\
&= \lim_{\delta \to 0} \frac{f((\lambda_1 + \delta, \lambda_2 + \delta, \dots, \lambda_K + \delta)) - f(\lambda_1, \lambda_2, \dots, \lambda_K)}{\delta} \\
&= \lim_{\delta \to 0} \frac{f((\lambda_1, \lambda_2, \dots, \lambda_K) + \delta r) - f(\lambda_1, \lambda_2, \dots, \lambda_K)}{\delta}
\end{aligned}$$

where r is $(1, 1, \dots, 1)$, i.e. the vector of all ones. The last equation is what we call a *directional derivative*. From a basic result in calculus we know

that the for a differentiable function f the directional derivative at a point λ along a vector r is equal to the inner product of the gradient at λ and r. Let $\cdot$ denote the inner product function. It follows that

$$\begin{aligned}
&\lim_{\delta \to 0} \frac{f((\lambda_1, \lambda_2, \ldots, \lambda_K) + \delta r) - f(\lambda_1, \lambda_2, \ldots, \lambda_K)}{\delta} \\
=&(\nabla_{(\lambda_1, \lambda_2, \ldots, \lambda_K)} f) \cdot r \\
=&(\frac{\partial}{\partial \lambda_1} f, \frac{\partial}{\partial \lambda_2} f, \ldots, \frac{\partial}{\partial \lambda_K} f) \cdot (1, 1, \ldots, 1) \\
=&\sum_{i=1}^{K} \frac{\partial}{\partial \lambda_i} f
\end{aligned}$$

which proves the result.

2.3 LSTM and other gated recurrent units

Although the basic RNN model seemed to have great learning potential in theory, it was discovered early that it had great difficulties learning long-term dependencies. Simply storing information over long sequences was very difficult to accomplish. A lot of early research was focused on improved optimization techniques for training RNNs. It proved very challenging however, and in some cases one had just as much success with random weight-guessing! Thorough analysis identified *exploding* or *vanishing* gradients as an inherent problem of RNNs [1]. As in many other cases, more progress was made by changing the model than improving the optimization.

Long short-term memory (LSTM) was a model designed to address the shortcomings of RNNs for long sequences [2]. The model has several components and we will introduce them one at a time. Arguably the most important contribution of the LSTM is the skip connections they introduce between layers. We have seen this in feed-forward residual networks (aka. ResNet [3]). Keep in mind, however, that the ideas we are presenting here were introduced two decades earlier. Our update equation with the introduced skip connection takes the form

$$s^t = s^{t-1} + r(x^t, s^{t-1}, y^{t-1}) \tag{4}$$

where r is a linear function possibly followed by an activation function, i.e. it is of the form in equation (3). To simplify notation, let r^t denote $r(x^t, s^{t-1}, y^{t-1})$. We then have

$$r^t = g(U_r x^t + V_r s^{t-1} + W_r y^{t-1} + b_r) \tag{5}$$

where g is often taken to be *tanh*, but other activation functions could be used instead.

The form in (4) makes it easier to keep information across time steps, as it now in some sense is the *default* behaviour. Instead of learning a new representation at each time step we only learn *changes*. The authors however did not stop there. They introduced what they called an *input gate* which is a further protection of the information contained in the state. The input gate decides whether we are allowed to update the neurons in the state vector or not. The r function now serves merely as a *proposed update*, but a particular state neuron will not change unless the input gate allows it. We will take the input gate to be of the form

$$i^t = \sigma(U_i x^t + V_i s^{t-1} + W_i y^{t-1} + b_i) \tag{6}$$

where the *sigmoid* function is used as the activation function. Combined with the residual we so far have

$$s^t = s^{t-1} + i^t \odot r^t \tag{7}$$

where $\odot$ denotes elementwise multiplication.

A few years after the original LSTM paper the *forget gate* was introduced [4]. This proved useful in continual learning tasks, i.e. tasks that stretches over time with no beginnings or ends. In these situations some of the information may be useful to accomplish a particular task, but then have no further value. It's not hard to imagine such situations. Imagine that you are asked to add 23 and 8, then these numbers are certainly important to remember. After you have provided the answer '31', however, and then asked to add 43 and 36, still having to remember 23 and 8 will not be helpful and could potentially take up memory we could have used for other purposes. We define the forget gate as

$$f^t = \sigma(U_f x^t + V_f s^{t-1} + W_f y^{t-1} + b_f) \tag{8}$$

The update equation is then transformed into

$$s^t = f^t \odot s^{t-1} + i^t \odot r^t \tag{9}$$

Of course we could learn to reset the state by adding an *additive inverse* to the current state. An explicit reset gate may however make the behavior easier to learn.

We are almost done describing the LSTM, there is only *one* more gate to describe. The original LSTM paper introduced the input gate to control *write access* to the state neurons. They also introduced an *output gate* to control the *read access* of the state neurons. At this point you might be thinking, "I was on board with the input gate, protecting the state so to make it easier to retain information over long periods. Not giving us full access to the state, which contains the information we need on the other hand...". You wouldn't be completely crazy having such thoughts. We will however put a slightly different perspective on it to appreciate the potential benefits of such a gate. Imagine that some of the neurons contains information that we will need at some point, but is not very useful in the current situation. By *blocking* the read access to this neurons we are not letting our decisions be affected by irrelevant information. In this setting we might look at the output gates as a soft *attention* mechanism, letting us focus on the state neurons that are the most useful to us for the decisions we need to make now. The form of the output gate follows the same pattern as the previous ones

$$o^t = \sigma(U_o x^t + V_o s^{t-1} + W_o y^{t-1} + b_o) \tag{10}$$

In addition to this attention mechanism, an activation function a is often applied to the state vector before we apply elementwise multiplication with the output gate. We thus have

$$\bar{s}^t = o^t \odot a(s^t) \tag{11}$$

where *tanh* again is a common choice of activation function. With the introduction of the output gate we will have to revisit everything we have done so far (including the output gate itself!), replacing the state s, which is no longer directly accesible, with the observeable $\bar{s}$.

$$r^t = g(U_r x^t + V_r \bar{s}^{t-1} + W_r y^{t-1} + b_r) \tag{12}$$
$$i^t = \sigma(U_i x^t + V_i \bar{s}^{t-1} + W_i y^{t-1} + b_i) \tag{13}$$
$$f^t = \sigma(U_f x^t + V_f \bar{s}^{t-1} + W_f y^{t-1} + b_f) \tag{14}$$
$$o^t = \sigma(U_o x^t + V_o \bar{s}^{t-1} + W_o y^{t-1} + b_o) \tag{15}$$

Note however that the form of the update of s^t remains unchanged, we still have

$$s^t = f^t \odot s^{t-1} + i^t \odot r^t \tag{16}$$

Although not clear from the expression, we of course have that s^t is a function of (x^t, s^{t-1}, y^{t-1}).

However [5] argued that in order to learn precise timings the gates need to be able to 'peek' into the state itself. For this purpose the concept of *peephole connections* was introduced. We will not go into that here, but see [6] for how these work.

A lot of different versions of the LSTM have been developed, and the term LSTM is often still used for this familiy of models with gated recurrent units. See [6] for a more thorough overview of the different variants. In the paper they also did a *local* search, changing one element from the vanilla LSTM, to see which parts of the LSTM are the most important. In [7] they defined a much broader search space, and was able to find gated recurrent units with good performance that looked quite different from the standard LSTM. The main takeaway is that residual connections and gated recurrent units have proved very helpful in learning long-term dependencies in RNNs. Exactly which model will work best for a particular problem may be hard to predict apriori, and must usually be determined experimenally.

2.4 Depth in RNN

We have seen different ways that recurrent neural networks implements the update function $h(x, s, y)$ of equation (2). In the basic RNN, the update function was that of equation (3), a simple linear function followed by an activation function. For the LSTM h was a bit more complex, but it was still a *shallow* function. How can we make the update function *deep*?

2.4.1 Multilayer perceptron

The first idea that comes to mind may be to use a multilayer perceptron[2] to implement h. One thing we should keep in mind though, is that if we make h have depth n then the path the error signal has to travel will be multiplied by a factor of n as well. One could of course use skip connections across several time steps to try to unset this effect.

2.4.2 Stacking of RNNs

Another quite different approach for introducing depth into a recurrent neural network is to stack RNNs on top of each other. Assume that we have L RNNs. Let s_l^t be the state of RNN l at time t. Let $y_0^t = x^t$ and for $l > 0$

[2]The term *multilayer perceptron* is often used for a neural network consisting of several fully connected layers.

let y_l^t denote the output of RNN l at time t. We then define the update equation for RNN l at time t as

$$s_l^t = h_l(y_{l-1}^t, s_l^{t-1}, y_l^{t-1}) \tag{17}$$

In other words we feed the output of RNN $l-1$ as input to RNN l. In many practical applications the output of all but the top RNN is simply taken to be the state itself (possibly after e.g. an output gate has been applied, which is typical for LSTM models). In this case there is of course no reason to feed the output back in (as it is the same as the state) so that equation (17) simplifies to

$$s_l^t = h_l(s_{l-1}^t, s_l^{t-1}) \tag{18}$$

If we deviate from a strict stacking of RNNs (and why shouldn't we?) we could feed back the output from the topmost RNN into a lower layer, to allow for output-dependent processing in earlier layers.

2.5 Complexity of RNN

We will here take a look into how memory usage, number of compute operations and number of serieal computational steps scales with sequence length. We will do the analyzes both in the case of inference and training, as they differs on some of these statistics.

For inference the amount of computation we do is the same for each time step, so clearly this scales linearly with the sequence length. This is true for the number of serial steps as well, we have to wait for the previous time step to finish processing before we can start processing the next. For memory usage the situation is different, we do not need to remember old state values and can thus reuse the memory buffers from the states. The memory usage is thus constant, independent of the sequence length.

What about training? In some sense the question is meaningless in this general form. RNN defines a model and there are many potential training algorithms, of which some have not yet been imagined, each which may scale differently with sequence length. For a specific training procedure however we may be able to answer this question. By far the most common method of training is the backpropagation algorithm. For RNN the term *backpropagation through time* is often used, but this is just the normal backpropagation algorithm applied to the unrolled network as we observed in Section 2.2. We argued before that the both compute and serial steps scaled linearly with sequence length for the forward pass. For the same reasons this is also true

for the backward pass. What about memory usage? During training we can no longer discard old states during the forward pass, but have to keep them around until we have finished the backward pass. As each time step then adds a constant amount of required memory, the total memory usage scales linearly with T. There is however a special scenario where a training iteration actually can be parallelized accross time. If we do not feed back the previous state to the next iteration, only the new input and the output of the previous state, we have an update equation of the form

$$s_t = h(x^t, y^{t-1}) \tag{19}$$

If we are doing supervised learning we actually have the *correct* labels y^t. If we instead of feeding the output of RNN at the previos time step, feed in the correct outputs, the computations at each time step are independent and can thus be parallelized. At inference time we have to generate the outputs ourselves, thus the complexity will there scale in the same way as usual with sequence length.

The results are summarized in table 1.

Table 1: RNN complexity as a function sequence length

	Memory	Compute	Serial steps
Inference	O(1)	O(T)	O(T)
Training BPTT	O(T)	O(T)	O(T)
Training BPTT h(x, y*)	O(1)	O(T)	O(1)

3 Problems

3.1 How can we recover the basic RNN from an LSTM model?

3.1.1 Problem

To be more precices, assume that we have an RNN with an update equation of the form in equation (3), with weight matrices U, V and W and bias vector b. Let a be the activation function and s^0_{rnn} be the initial state of the RNN. We also assune a continuous, deterministic output function f. Assume we have the LSTM setup of equations (11) - (16). Let s^t and y^t denote the state and output at time t for the LSTM. How can we choose the parameters of

the LSTM such that $s^t \approx s^t_{rnn},\ y^t \approx y^t_{rnn}$ for all t, where the approximation could be made arbitrarily good?

3.2 Show that a stacked RNN is itself an RNN

3.2.1 Problem

We have proposed stacking of RNNs as a way to make RNNs deep. Can we show that the model we get when stacking RNNs actually is an RNN? With this we mean that given a sequence $X_1, \ldots, X_T$ we can write $Y_1, \ldots, Y_T$ as

$$\begin{aligned} Y^t &= f(S^t) \\ S^t &= h(X^t, S^{t-1}, Y^{t-1}) \end{aligned}$$

for some state space and functions f and h.

4 Bibliography

References

[1] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91:1, 1991.

[2] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[4] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[5] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000.

[6] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2017.

[7] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.