

6.036 Spring 2018 : Week 10

April 5, 2018

11 Recurrent Neural networks

Our goal here is to model sequences such as natural language sentences using neural networks. We will try to first adapt feed-forward networks for this purpose by incorporating one, two, or more previous words as inputs, aiming to always to predict the next word in the sequence. We will then generalize the simple feedforward model to a recurrent neural network (RNN) which has greater flexibility of storing key aspects of the sequence seen so far. RNNs maintain a notion of "state" (continuous vector) that evolves along the sequence and serves as the memory or summary of what has been seen so far. Next word is then predicted relative to the information stored in the state.

While RNNs are powerful, they are not trivial to train. Essentially you are trying to adapt a dynamical system to your bidding. In terms of the algorithms, the problem is very similar to deep feed-forward neural networks: we use gradient descent, and calculate the necessary gradients by propagating the gradient information backwards in the model (now backwards in time/sequence). RNNs use the same functional mapping (same parameters) to transform the the previous state and any new information into a new state vector. As the same parameters are used at different times / positions along the sequence, there is quite a bit of parameter sharing. Indeed, if we unfold an RNN explicitly in time, i.e., treat each state update essentially as a layer in a feed-forward architecture, we would have a lot of parameter sharing between the layers. You will see this as one difference when learning the parameters.

Basic RNNs suffer from vanishing/exploding gradient problems just as any deep neural network. The problem is a bit more pronounced with RNNs since typically the number of time steps can be fairly large. The issue can be mitigated by introducing "gating". Gates are simple networks that are used to control the flow of information into the state (memory) as well as what is read from the state for the purpose of predictions (e.g., next symbol in the sequence). Most modern neural network sequence models are variants of so called Long-Short-Term-Memory (LSTM), introduced already two decades ago. We will begin with simple versions of these models, pruning away details when possible for clarity of exposition, illustrating the complex version only at the end.

11.1 From feed-forward to RNN language models

As a starting point, consider a simple English sentence

$$\begin{array}{cccccccc} y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 & y_8 \\ \text{midterm} & \text{will} & \text{have} & \text{a} & \text{neural} & \text{networks} & \text{question} & \text{<end>} \end{array} \tag{1}$$

where we will represent words by discrete variables $y_i \in \{1, \ldots, V\}$ where V is the size of the vocabulary we consider plus one (the <end> symbol). Note that in order to predict a sentence we also need to determine when it ends. For this reason, we always have the <end> symbol appearing at the end of each sentence. All of our models predict sentences by generating words in a sequence until they predict <end> as the next symbol. Formally, we will specify a probability distribution over the sentences, unfolded sequentially as in

$$P(y_1)P(y_2|y_1)P(y_3|y_2, y_1) \cdots P(y_{n-1}|y_{n-2}, \ldots, y_1)P(y_n|y_{n-1}, \ldots, y_1) \tag{2}$$

where y_n is necessarily <end> if $y_1, \ldots, y_n$ represents a complete sentence. The distribution involves parameters that come from the neural network that is used to realize the individual probability terms. For example, we could use a simple feed-forward neural network model that takes in just the previous word and predicts a probability distribution over the next word. As such, it will not be able to capture dependences beyond the previous symbol. In this model, $P(y_3|y_2, y_1)$ could only be represented as $P(y_3|y_2)$ (dependence on y_1 is lost). Such a model would then assign probability

$$P(y_1)P(y_2|y_1)P(y_3|y_2) \cdots P(y_{n-1}|y_{n-2})P(y_n|y_{n-1}) \tag{3}$$

to the sentence $y_1, \ldots, y_n$. Figure below illustrates a possible feed-forward neural network for modeling $P(y_t|y_{t-1})$. The same model is applied for all $t = 1, \ldots, n$ where the symbol y_0 can be taken to be <end> (end of previous sentence). We first map index y_{t-1} to a one-hot vector $\phi(y_{t-1})$ of dimension V that can be used as an input vector x_t. Note that, in this representation, only one of the input units is on (one) while all the rest are zero. With this input, we obtain hidden unit activations s_t (vector of dimension m), and finally softmax output probabilities p_t (a vector of dimension V). Thus the kth component of p_t represents $P(y_t = k|y_{t-1})$.
Algebraically, our model corresponds to equations

$$x_t = \phi(y_{t-1}), \quad s_t = \tanh(W^{s,x}x_t), \quad p_t = \text{softmax}(W^o s_t) \tag{4}$$

where, for clarity, we have omitted any offset parameters, and the equations are "vectorized" in the sense that, e.g., tanh is applied element-wise to a vector of inputs. Here $W^{s,x}$ is a matrix of dimension $m \times V$ and W^o is a $V \times m$ matrix.

The problem with the simple feed-forward model is that the prediction about what comes next is made only on the basis of the previous word. We can, of course, expand the model to use the previous two or three words as inputs rather than just one. However, even though the resulting model clearly becomes more powerful with any additional input added, it still lacks the ability to memorize an important cue early on in the sentence and store it for use later on. We need a way to carry the information we knew before forward as well. To address this, we will feed the previous hidden state vector s_{t-1} (summary prior to seeing

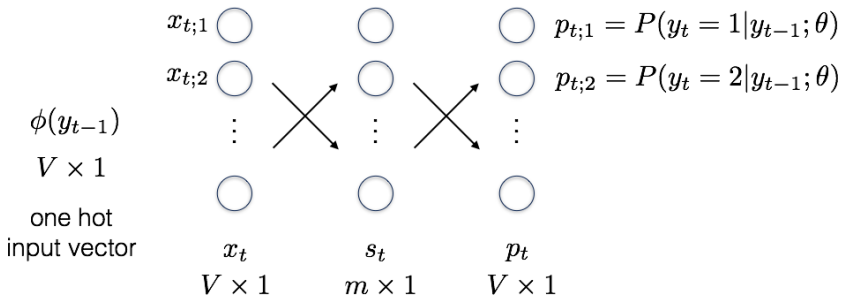


Figure 1: Simple feed-forward language model where the previous word is fed in as a one-hot vector, and the output is a vector of probabilities over the possible next words (softmax layer). Layer to layer transformations are dense, fully connected. The hidden layer dimension m is typically considerably smaller than the size of the vocabulary V. For example, we might have $m = 100$ and $V = 10,000$.

y_{t-1}) as an input to the model along with $\phi(y_{t-1})$ as in the feed-forward model. The idea of recurrent neural networks is precisely to incorporate previous "state" as an input. Algebraically, the simple RNN model is characterized by

$$x_t = \phi(y_{t-1}), \quad s_t = \tanh(W^{s,x}x_t + W^{s,s}s_{t-1}), \quad p_t = \text{softmax}(W^o s_t) \tag{5}$$

where we simply added the previous state s_{t-1} as an additional input to the state update. The associated new parameters are represented by an $m \times m$ matrix $W^{s,s}$. Such a recurrent neural network model has the ability to generate quite complex behaviors. Indeed, note that since s_{t-1} was calculated when y_{t-2} was fed as an input, the distribution p_t must depend on y_{t-2} as well. Going further, s_{t-1} also depends on s_{t-2} which was determined in part by y_{t-3}, and so on. As a result, the distribution over the possible next words p_t is affected by the whole history $y_{t-1}, \ldots, y_1$. Of course, the way this dependence is manifested results from the specific parameterization in the update equations. The recurrent model does not have enough parameters to specify any arbitrary $P(y_t|y_{t-1}, \ldots, y_1)$ (this would require $(V-1)V^{t-1}$ parameters) unless we increase m quite a bit. But it nevertheless gives us a parametric way to leverage the whole sequence seen so far in order to predict the next word.

11.1.1 Back-propagation through time

Training a recurrent neural network model doesn't differ much from training a simple feed-forward model. We can back-propagate the error signals measured at the outputs, and update parameters according to the resulting gradients. It is often helpful to unfold the RNN in time so that it looks more like a feed-forward structure. For example, we can represent each application of the update equations as a block, taking x_t, s_{t-1} as inputs and producing s_t, p_t. In other words, we feed $\phi(y_{t-1})$ as an input x_t together with the previous state s_{t-1}, calculate the updated state s_t and the distribution p_t over possible words y_t. Once y_t is revealed, we can measure $\text{Loss}(y_t, p_t)$, and proceed a step forward. The overall goal is to minimize the sum of losses incurred as the model is unfolded along the sequence.

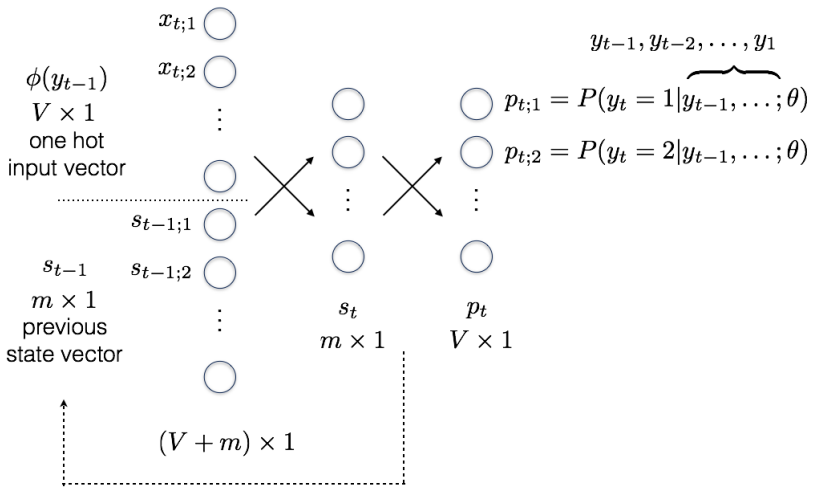
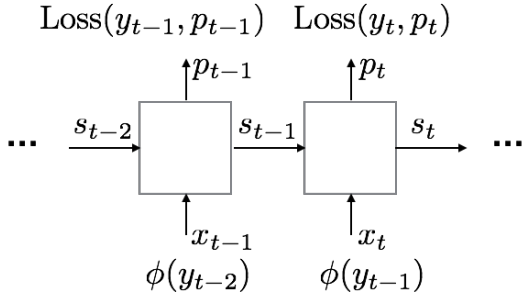


Figure 2: Simple recurrent neural network language model. The previous hidden layer activations – state – are routed as additional input coordinates when calculating the current activation and the output probabilities.

Note that the weights are shared across the blocks, i.e., each block uses exactly the same parameters $W^{s,x}$, $W^{s,s}$, W^o.

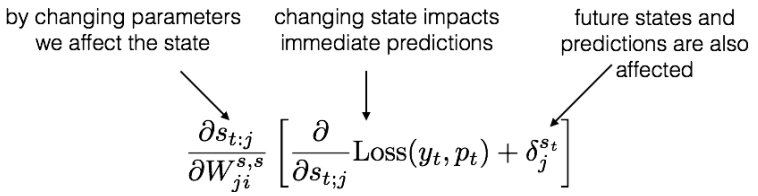


Unfolded block view of the simple RNN

Let's begin with a simple summary of how we get the gradients in these models. Suppose we are interested in adjusting $W^{s,s}$, i.e., parameters specifying how the model uses the previous state to update the new one. The goal is to minimize the prediction loss along the whole sequence, so we have to evaluate

$$\frac{\partial}{\partial W^{s,s}_{ji}} \left[\sum_{l=1}^{n} \text{Loss}(y_l, p_l) \right] \tag{6}$$

The same parameters $W^{s,s}$ are used in each block along the sequence and therefore any changes in the parameters impact each of the terms in the sum. The effect is also compounded by the fact that parameters affect the next state which in turn affects future states and losses. If we focus on a single block, at position t, the derivative we want is



The derivative in Eq.(6) is therefore just a sum of these terms, across time/positions. It remains to figure out how future losses depend on changing the current state s_t, the effect that is captured in δ^{s_t}

The longer route: Suppose we have already evaluated

$$\delta^{s_t}_j = \frac{\partial}{\partial s_{t;j}} \sum_{l=t+1}^{n} \text{Loss}(y_l, p_l), \quad j = 1, \ldots, m \tag{7}$$

i.e., the impact of state s_t on the future losses (after t). Here j indexes the m units that are used to represent the state vector. Note that $\delta^{s_n}_j$ is simply zero as there are no future losses from that point on. We wish to calculate $\delta^{s_{t-1}}_i$, $i = 1, \ldots, m$, i.e., go one block backwards. Note that in our simple RNN, s_{t-1} affects $\text{Loss}(y_t, p_t)$ as well as the future losses only via the next state s_t. As a result,

$$\begin{aligned}
\delta^{s_{t-1}}_i &= \frac{\partial}{\partial s_{t-1;i}} \left[\text{Loss}(y_t, p_t) + \sum_{l=t+1}^{n} \text{Loss}(y_l, p_l) \right] && (8)\\
&= \sum_{j=1}^{m} \frac{\partial s_{t;j}}{\partial s_{t-1;i}} \frac{\partial}{\partial s_{t;j}} \left[\text{Loss}(y_t, p_t) + \sum_{l=t+1}^{n} \text{Loss}(y_l, p_l) \right] && (9)\\
&= \sum_{j=1}^{m} \frac{\partial s_{t;j}}{\partial s_{t-1;i}} \left[\frac{\partial}{\partial s_{t;j}} \text{Loss}(y_t, p_t) + \frac{\partial}{\partial s_{t;j}} \sum_{l=t+1}^{n} \text{Loss}(y_l, p_l) \right] && (10)\\
&= \sum_{j=1}^{m} \frac{\partial s_{t;j}}{\partial s_{t-1;i}} \left[\frac{\partial}{\partial s_{t;j}} \text{Loss}(y_t, p_t) + \delta^{s_t}_j \right] && (11)
\end{aligned}$$

where, for example,

$$\begin{aligned}
\frac{\partial s_{t;j}}{\partial s_{t-1;i}} &= \frac{\partial}{\partial s_{t-1;i}} \tanh([W^{s,x} x_t]_j + [W^{s,s} s_{t-1}]_j) && (12)\\
&= (1 - s^2_{t;j}) \frac{\partial}{\partial s_{t-1;i}} [W^{s,s} s_{t-1}]_j && (13)\\
&= (1 - s^2_{t;j}) W^{s,s}_{ji} && (14)\\
\frac{\partial}{\partial s_{t;j}} \text{Loss}(y_t, p_t) &= -\sum_{k=1}^{V} \left([\![k = y_t]\!] - p_{t;k} \right) W^{o}_{k,j} && (15)
\end{aligned}$$

Now, suppose we have calculated all $\delta^{s_t}_j$, $t = 1, \ldots, n$, $j = 1, \ldots, m$. We would like to use these to evaluate the derivatives with respect to the parameters that are shared across the

unfolded model. For example, $W^{s,s}_{ji}$ is used in all the state updates and therefore it will have an impact on the losses through $s_1, \ldots, s_n$. We will decompose this effect into a sum of immediate effects, i.e., how $W^{s,s}_{ji}$ impacts s_t (and losses there on) when s_{t-1} is assumed to be constant. So, when we write $\partial s_{t;k}/\partial W^{s,s}_{ji}$, we mean this immediate impact (as if the parameter was specific to the t^{th} box, just set equal to $W^{s,s}_{ji}$). Accordingly,

$$\frac{\partial}{\partial W^{s,s}_{ji}}\left[\sum_{l=1}^{n}\text{Loss}(y_l, p_l)\right] = \sum_{t=1}^{n}\sum_{k=1}^{m}\frac{\partial s_{t;k}}{\partial W^{s,s}_{ji}}\frac{\partial}{\partial s_{t;k}}\left[\sum_{l=1}^{n}\text{Loss}(y_l, p_l)\right] \quad (16)$$

$$= \sum_{t=1}^{n}\sum_{k=1}^{m}\frac{\partial s_{t:k}}{\partial W^{s,s}_{ji}}\left[\frac{\partial}{\partial s_{t;k}}\text{Loss}(y_t, p_t) + \frac{\partial}{\partial s_{t;k}}\sum_{l=t+1}^{n}\text{Loss}(y_l, p_l)\right] \quad (17)$$

$$= \sum_{t=1}^{n}\sum_{k=1}^{m}\frac{\partial s_{t:k}}{\partial W^{s,s}_{ji}}\left[\frac{\partial}{\partial s_{t;k}}\text{Loss}(y_t, p_t) + \delta^{s_t}_{k}\right] \quad (18)$$

$$= \sum_{t=1}^{n}\frac{\partial s_{t:j}}{\partial W^{s,s}_{ji}}\left[\frac{\partial}{\partial s_{t;j}}\text{Loss}(y_t, p_t) + \delta^{s_t}_{j}\right] \quad (19)$$

$$= \sum_{t=1}^{n}(1 - s^2_{t:j})s_{t-1;i}\left[\frac{\partial}{\partial s_{t;j}}\text{Loss}(y_t, p_t) + \delta^{s_t}_{j}\right] \quad (20)$$

The problem with learning RNNs is not the equations but rather how the gradients tend to vanish/explode as the number of time steps increases. You can see this by examining how $\delta^{s_{t-1}}$ is obtained on the basis of δ^{s_t}. It involves multiplication with the weights as well as the derivative of the activation function (tanh). Performing a series of such multiplications can easily escalate or rapidly vanish. One way to remedy this problem is to change the architecture a bit.

11.1.2 RNNs with gates

We would like to make the state update more "additive" than multiplicative. The problem with multiplication is that such updates easily result in vanishing or exploding gradients. One way to keep the state updates more additive yet non-linear is to control the flow of information into the state by a gating network. A gate is a simple network that determines how much each coordinate of state s_t is updated. Specifically, we introduce

$$g_t = \text{sigmoid}(W^{g,x}x_t + W^{g,s}s_{t-1}) \quad (21)$$

where the activation function is again applied element-wise, and the gate vector g_t has the same dimension as the state, i.e., $m \times 1$. Recall that $\text{sigmoid}(z) = (1 + \exp(-z))^{-1}$ and therefore all of its coordinates lie in $[0, 1]$. As the first step, we combine gate and state updates as follows

$$x_t = \phi(y_{t-1}), \quad (22)$$

$$g_t = \text{sigmoid}(W^{g,x}x_t + W^{g,s}s_{t-1}) \quad (23)$$

$$s_t = (1 - g_t) \odot s_{t-1} + g_t \odot \tanh(W^{s,x}x_t + W^{s,s}s_{t-1}) \quad (24)$$

$$p_t = \text{softmax}(W^o s_t) \quad (25)$$

where $\odot$ refers to element-wise product of vector coordinates. Note that previously we simply overwrote the state with $\tanh(W^{s,x}x_t + W^{s,s}s_{t-1})$ whereas now s_t update is more graded, we add this contribution to the state to the degree specified by the gate. For example, if some of the coordinates of g_t are close to zero, s_t retains the same value as s_{t-1} for those coordinates. As a result, the model can easily carry information from the beginning of the sentence towards the end, and this property is directly controlled by the gate. Models of this kind are typically much easier to learn than simple RNNs with gradient descent algorithms and are therefore preferred.

LSTMs, General Recurrent Units (GRUs), and other variants involve further gating to better control the flow of information in and out of the state. In an LSTM, we have two states, an internal memory cell c_t and a visible state h_t. So, if we view LSTM as a block that transforms its state in light of new information, then the state that evolves is the pair $[c_t, h_t]$. We use gates to control what to forget from the memory cell (forget gate f_t), what to incorporate into it from observations (via input gate i_t), and what to read out from the memory cell into the visible state h_t (via output gate o_t). The memory cell c_t is additively accumulated and adjusted with gates. Specifically,

$$
\begin{aligned}
x_t &= \phi(y_{t-1}), \quad \text{(our language model example)} \\
f_t &= \text{sigmoid}(W^{f,h}h_{t-1} + W^{f,x}x_t) \quad \text{(forget gate)} \\
i_t &= \text{sigmoid}(W^{i,h}h_{t-1} + W^{i,x}x_t) \quad \text{(input gate)} \\
o_t &= \text{sigmoid}(W^{o,h}h_{t-1} + W^{o,x}x_t) \quad \text{(output gate)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W^{c,h}h_{t-1} + W^{c,x}x_t) \quad \text{(memory cell)} \\
h_t &= o_t \odot \tanh(c_t) \quad \text{(visible state)} \\
p_t &= \text{softmax}(W^o h_t)
\end{aligned}
$$

Note that we have again omitted all the offset parameter vectors from these updates. There are a number of variants of LSTMs though typically they perform comparably.