Recurrent neural networks

Feedforward n/ws have fixed size input, fixed size output, fixed number of layers. RNNs can have variable input/output.

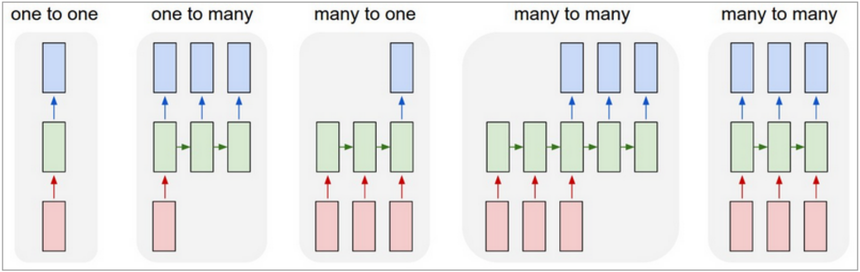

Figure: RNN schematic. Input: red, Output: blue, State: green. 1-1: Std. feed forward, 1-M: image captioning,M-1: sentiment identification, M-M: translation, M-Ms: Video frame labelling. *(Source: Karpathy blog)*

RNN applications

RNNs (and their variants) have been used to model a wide range of tasks in NLP:

- Machine translation. Here we need a sentence aligned data corpus between the two languages.
- Various sequence labelling tasks e.g. PoS tagging, NER, etc.
- Language models. A language model predicts the next word in a sequence given the earlier words.
- Text generation.
- Speech recognition.
- Image annotation.

Recurrent neural network - basic

- Recurrent neural n/ws have feedback links.

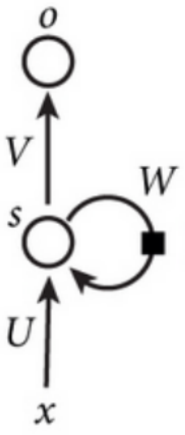

Figure: An RNN node[1]

[1]Source: WildML blog

Recurrent neural network schematic

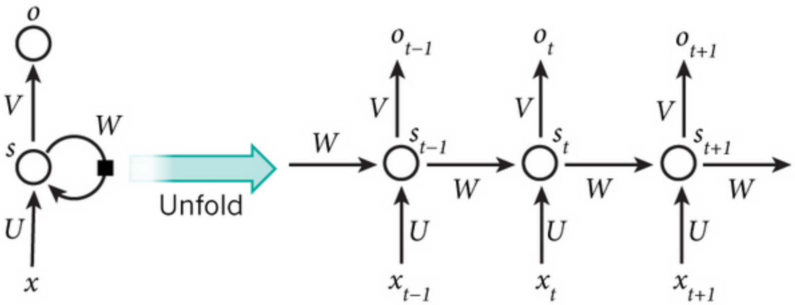

Figure: RNN unfolded[2]

[2]Source: WildML blog

RNN foward computation

- An RNN computation happens over an input sequence and produces an output sequence using the t^{th} element in the input and the neuron state at the $(t-1)^{th}$ step, ψ_{t-1}. The assumption is that ψ_{t-1} encodes earlier history.
- Let ψ_{t-1}, ψ_t be the states of the neuron at step $t-1$ and t respectively, w_t the t^{th} input element and y_t the t^{th} output element (notation is diffent from figure to be consistent with our earlier notation). Assuming U, V, W are known ψ_t and y_t are calculated by:

$$\psi_t = f_a(Uw_t + W\psi_{t-1})$$
$$y_t = f_o(V\psi_t)$$

  Here f_a is the activation function, typically *tanh*, *sigmoid* or *relu* which introduces non-linearity in the network. Similary, f_o is the output function, for example *softmax* which gives a probability over the vocabulary vector.

- Often inputs or more commonly outputs at some time steps may not be useful. This depends on the nature of the task.
- Note that the weights represented by U, V, W are the same for all elements in the sequence.

The above is a standard RNN. There are many variants in use that are better suited to specific NLP tasks. But in principle they are similar to the standard RNN.

Training RNNs

- RNNs use a variant of the standard back propagation algorithm called *back propagation through time* or BPTT. To calculate the gradient at step t we need the gradients at earlier time steps.
- The need for gradients at earlier time steps leads to two complications:
  a) Computations are expensive (see later).
  b) The vanishing/exploding gradients (or unstable gradients) problem seen in deep neural nets.

Coding of input, output, dimensions of vectors, matrices

It is instructive to see how the input, output is represented and calculate the dimensions of the vectors, matrices. Assume $|\mathcal{V}|$ is vocabulary size, h is hidden layer size. Assume we are training a recurrent network language model. Then:

- Let m be the number of words in the current sentence. The input sentence will look like:

  [START-SENTENCE, $w_1, \ldots, w_i, w_{i+1}, \ldots, w_m$, END-SENTENCE].
- Each word w_i in the sentence will be input as a 1-hot vector of size $|\mathcal{V}|$ with 1 at index of w_i in $\mathcal{V}$ and rest 0.
- Each desired output y_i is a 1-hot vector of size $|\mathcal{V}|$ with 1 at index of w_{i+1} in $\mathcal{V}$ and 0 elsewhere. The actual output $\hat{y}$ is a softmax output vector of size $|\mathcal{V}|$.
- The dimensions are:
  Sentence: $m \times |\mathcal{V}|$, U: $h \times |\mathcal{V}|$, V: $|\mathcal{V}| \times h$, W: $h \times h$

Update equations

At step t let y_t be the true output value and $\hat{y}_t$ the actual output of the RNN for input w_t then using cross-entropy error (often used for such networks instead of squared error) we get:

$$\mathcal{E}_t(y_t, \hat{y}_t) = -\mathcal{Y}_t \, log(\hat{y}_t)$$

For the sequence from 0 to t which is one training sequence we get:

$$\mathcal{E}(\hat{y}, y) = -\sum_t \mathcal{Y}_t \, log(\hat{y}_t)$$

BPTT will use backprop from 0 to t and use (stochastic) gradient descent to learn appropriate values for U, V, W. Assume $t = 3$ then the gradient for V is $\frac{\partial \mathcal{E}_3}{\partial V}$

Using chain rule:

$$\begin{aligned}\frac{\partial \mathcal{E}_3}{\partial V} &= \frac{\partial \mathcal{E}_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial V} \\ &= \frac{\partial \mathcal{E}_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial V} \\ &= (\hat{y}_3 - y_3) \otimes \psi_3\end{aligned}$$

$z_3 = V\psi_3$, $\mathbf{u} \otimes \mathbf{v} = \mathbf{u}\mathbf{v}^T$ (outer/tensor product).

Observation: $\frac{\partial \mathcal{E}_3}{\partial V}$ depends only on values at $t = 3$.

For W (and similarly for U)

$$\frac{\partial \mathcal{E}_3}{\partial W} = \frac{\partial \mathcal{E}_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial \psi_3} \frac{\partial \psi_3}{\partial W}$$

$\psi_3 = tanh(Uw_t + W\psi_2)$ and ψ_2 depends on W, ψ_2 etc. So, we get:

$$\frac{\partial \mathcal{E}_3}{\partial W} = \sum_{t=0}^{3} \frac{\partial \mathcal{E}_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial \psi_3} \frac{\partial \psi_3}{\partial \psi_t} \frac{\partial \psi_t}{\partial W}$$

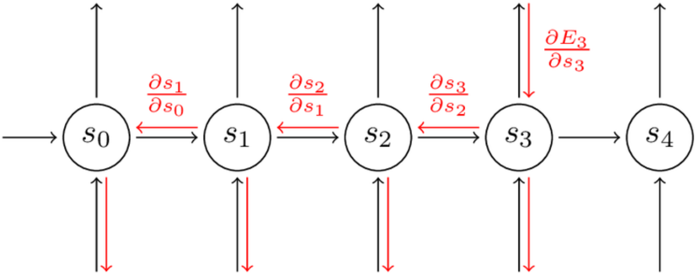

Figure: Backprop for W.[3]

[3] Source: WildML blog

Vanishing gradients

- Neural networks with a large number of layers using sigmoid or tanh activation functions face the unstable (typically vanishing gradients since explosions can be blocked by a threshold).
- $\frac{\partial \psi_3}{\partial \psi_t}$ in the equation for $\frac{\partial \mathcal{E}_3}{\partial W}$ is itself a product of gradients due to the chain rule:

$$\frac{\partial \psi_3}{\partial \psi_t} = \prod_{i=t+1}^{3} \frac{\partial \psi_i}{\partial \psi_{i-1}}$$

So, $\frac{\partial \psi_3}{\partial \psi_t}$ is a Jacobian matrix and we are doing $O(m)$ matrix multiplications (m is sentence length).

How to address unstable gradients?

- Do not use gradient based weight updates.
- Use *Relu* which has a gradient of 0 or 1. ($relu(x) = max(0, x)$).
- Use a different architecture. LSTM (Long, Short Term Memory) or GRU (Gated Recurrent Unit). We go foward with this option - the most popular in current literature.

Schematic of LSTM cell

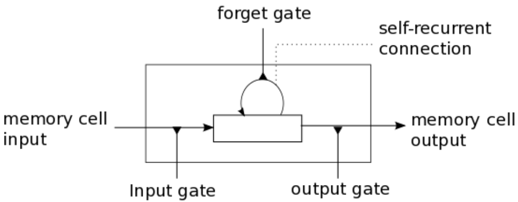

Figure: A basic LSTM memory cell. (From deeplearning4j.org).

LSTM: step by step

- The key data entity in an LSTM memory cell is the hidden internal state C_t at time step t, which is a vector. This state is not visible outside the cell.
- The main computational step calculates the internal state C_t which depends on the input $\mathbf{x}_t$ and the previous visible state h_{t-1}. But this dependence is not direct. It is controlled or modulated via two gates: the forget gate g_f and the input gate g_i. The output state h_t that is visible outside the cell is obtained by modulating the hidden internal state C_t by the output gate g_o.

The gates

- Each gate gets as input the input vector $\mathbf{x}_t$ and the previous visible state h_{t-1}, each linearly transformed by the corresponding weight matrix.
- Intuitively, the gates control the following:
  g_f: what part of the previous cell state C_{t-1} will be carried forward to step t.
  g_i: what part of $\mathbf{x}_t$ and visible state h_{t-1} will be used for calculating C_t.
  g_o: what part of the current internal state C_t will be made visible outside - that is h_t .
- C_t does depend on the earlier state C_{t-1} and the current input $\mathbf{x}_t$ but not directly as in an RNN. The gates g_f and g_i modulate the two parts (see later).

Difference between RNN and LSTM

The difference with respect to a standard RNN is:

1. There is no hidden state in an RNN. The state h_t is directly computed from the previous state h_{t-1} and the current input $\mathbf{x}_t$ each linearly transformed via the respective weight matrices U and W.
2. There is no modulation or control. There are no gates.

The gate equations

Each gate has as input the current input vector $\mathbf{x}_t$ and the previous visible state vector h_t.

$$
\begin{aligned}
g_f &= \sigma(W_f \mathbf{x}_t + W_{h_f} h_{t-1} + \mathbf{b}_f) \\
g_i &= \sigma(W_i \mathbf{x}_t + W_{h_i} h_{t-1} + \mathbf{b}_i) \\
g_o &= \sigma(W_o \mathbf{x}_t + W_{h_o} h_{t-1} + \mathbf{b}_o)
\end{aligned}
$$

The gate non-linearity is always a sigmoid. This gives a value between 0 and 1 for each gate output. The intuition is: a 0 value completely forgets the past and a 1 value fully remembers the past. An in between value partially remembers the past. This controls long term dependency.

Calculation of C_t, h_t

- There are two vectors that must be calculated. The hidden internal state C_t and the visible cell state h_t that will be the propagated to step $(t + 1)$.
- The C_t calculation is done in two steps. First an update $\hat{C}_t$ is calculated based on the two inputs to the memory cell, $\mathbf{x}_t$ and h_{t-1}.

  $$\hat{C}_t = tanh(W_c \mathbf{x}_t + W_{h_c} h_{t-1} + \mathbf{b}_c)$$

  C_t is calculated by modulating the previous state with g_f and adding to it the update $\hat{C}_t$ modulated by g_i giving:

  $$C_t = (g_f \otimes C_{t-1}) \oplus (g_i \otimes \hat{C}_t)$$

- Similarly, the new visible state or output h_t is calculated by:

  $$h_t = g_o \otimes tanh(C_t)$$

Detailed schematic of LSTM cell

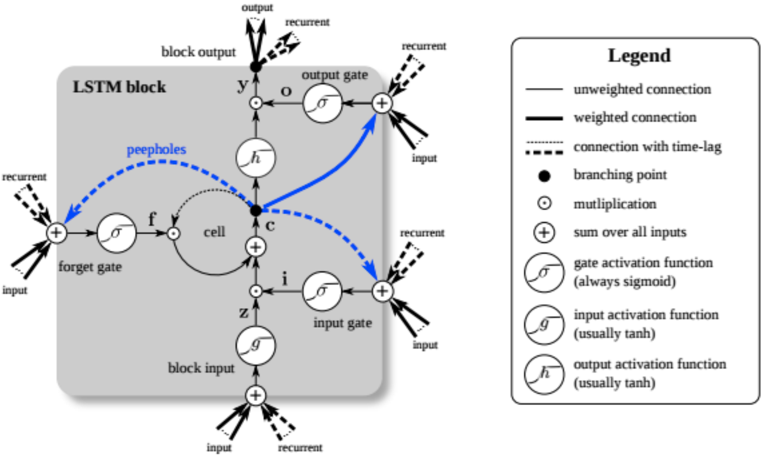

Figure: Details of LSTM memory cell. (From deeplearning4j.org).

Peepholes

- A variant of the standard LSTM has peep holes. This means that all the gates have access to the earlier and current hidden states C_{t-1} and C_t. So, the gate equations become:

$$
\begin{aligned}
g_f &= \sigma(W_f \mathbf{x}_t + W_{h_f} h_{t-1} + W_{c_f} C_{t-1} + \mathbf{b}_f) \\
g_i &= \sigma(W_i \mathbf{x}_t + W_{h_i} h_{t-1} + W_{c_i} C_{t-1} + \mathbf{b}_i) \\
g_o &= \sigma(W_o \mathbf{x}_t + W_{h_o} h_t + W_{c_o} C_t + \mathbf{b}_o)
\end{aligned}
$$

This is shown by the blue dashed lines and non-dashed line in the earlier diagram.

Gated Recurrent Units (GRU)

- A GRU is a simpler version of an LSTM. The differences from an LSTM are:
  - A GRU does not have an internal hidden cell state (like C_t). Instead it has only state h_t.
  - A GRU has only two gates g_r (reset gate) and g_u update gate. These together perform like the earlier g_f and g_i LSTM gates. There is no output gate.
  - Since there is no hidden cell state or output gate the output is h_t without a tanh non-linearity.

GRU equations

- The gate equations are:

$$g_r = \sigma(W_{r_i}\mathbf{x}_t + W_{r_h}h_{t-1} + \mathbf{b}_r)$$
$$g_u = \sigma(W_{u_i}\mathbf{x}_t + W_{u_h}h_{t-1} + \mathbf{b}_u)$$

- The state update is:

$$\hat{h}_t = tanh(W_{h_i}\mathbf{x}_t + W_{h_h}(g_r \otimes h_{t-1}) + b_h)$$

  The gate g_r is used to control what part of the previous state will be used in the update.
- The final state is:

$$h_t = ((1 - g_u) \otimes \hat{h}_t) \oplus (g_u \otimes h_{t-1})$$

  The update gate g_u controls the relative contribution of the update ($\hat{h}_t$) and the previous state h_{t-1}.

GRU schematic

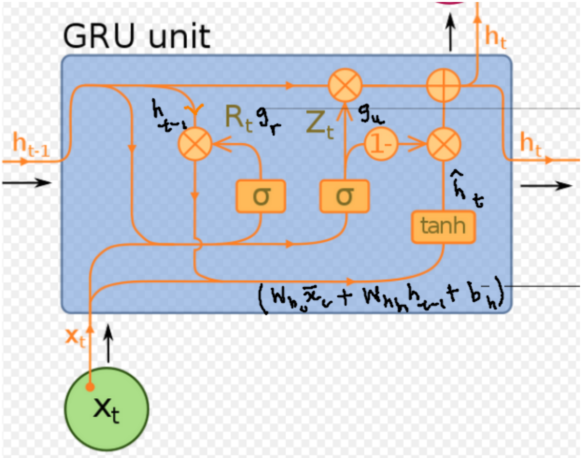

Figure: Details of GRU memory cell. (From wikimedia.org).