I. RECURRENT NEURAL NETWORKS

Consider an RNN which is designed to predict the output y from the input x. Suppose an input is a vector of dimension S and an output is a vector of dimension N. For instance, an input might be a letter i from the alphabet with S characters, represented as a vector $x_j = \delta_{ij},\ j = 1, \ldots, S$. In another example an input can be a column of features in the system with S features.

The layers are labeled by the index t which takes T values. The T is an input size, which can be interpreted as a length of the time series over which we train the RNN. Within this time series the subsequent inputs are influenced by the prior inputs through the memory propagation, as explained below.

Each layer t is characterized by the memory vector m_t of size M. The RNN is characterized hidden bias vector b, output bias vector e, input-to-weight matrix W, memory-to-memory matrix V, and memory-to-output matrix U. Layers are connected to each other sequentially, so that a memory vector at each layer influences both the output of that layer and the memory input of the next layer. The memory of the given layer is also influenced by the input of the given layer. Therefore the network propagates the knowledge of the previous inputs down the layers, in the form of the layer memory impact. These conditions are summarized as (we choose tanh as a neuron activation function for illustration purposes, the discussion can be generalized to other activations functions)

$$m_t = \tanh\left(W\, x_t + V\, m_{t-1} + b\right)\,, \tag{1}$$

$$y_t = U\, m_t + e\,, \tag{2}$$

that is, first we update the layer memory m_t using the input x_t at the given layer, and the memory m_{t-1} at the previous layer, and next we calculate an output of the given layer using the found memory m_t of the given layer. We will seed the memory layer m_{-1} by hand in order to start the processing.

Notice that W, V, U, b, and e parameters of the RNN are independent of t. This is unlike a typical feed-forward multi-layer NN, where at each layer we would have its own parameters. Therefore an RNN can be used on time series of different lengths, and we do not need to specify length in advance when designing RNN. Instead, the layer-specific information is contained in the memory vectors m.

In the feed-forward NN one input is converted into one output by propagation of outputs, front to back, through the hidden layers. An output of each layer serves as an input for the next layer. In RNN each layer both produces an output, and propagates the knowledge gained from the given input (in the form of the m memories) to the next layer. Therefore the RNN has two outputs from each layer: the y which we use as a prediction from x, and the m which we use to accumulate the information gained from x. The RNN can therefore process a causal chain of inputs, learning the time correlations between a different inputs in the sequence.

Therefore each layer of an RNN serves a double function of a hidden layer, and an input/output gate at a given time step. At each time step we can evaluate probabilities of various outputs as follows,

$$P_{it} = \frac{e^{y_{it}}}{\sum_j e^{y_{jt}}}\,, \quad i = 1, \ldots, N\,. \tag{3}$$

We want to train the RNN to produce an output R_t at time t. Then we calculate the loss function

$$L = -\sum_t \log\left(\sum_i P_{it}\, R_{it}\right)\,. \tag{4}$$

We will be considering the training problems in which at each time t we want to predict a vector

$$R_{it} = \delta_{i\, r_t}\,, \tag{5}$$

that is, for the given sequence of inputs $\{x_t\}$ we want to produce sequence of indices $\{r_t\}$ of the most significant components of the output vectors. The loss function is therefore given by

$$L = \sum_t \log\left(\sum_i e^{y_{it}}\right) - \sum_t y_{r_t\, t}\,, \tag{6}$$

and its gradient w.r.t. the outputs is

$$\left[\nabla^{(y)} L\right]_{mt} \equiv \frac{\partial L}{\partial y_{mt}} = P_{mt} - \delta_{m\, r_t}\,. \tag{7}$$

To train an RNN means to fit its parameters W, U, V, b, and e. We will do this using a backpropagation technique, shifting the values of those parameters in the direction opposite to the error function gradient. We need to find derivatives of the loss function w.r.t. these

parameters of the RNN. Let's write down the RNN equations in a more elaborate form

$$m_{it} = \tanh\left(\sum_j W^{ij}\, x_{jt} + \sum_j V^{ij}\, m_{j\,t-1} + b^i\right)\,, \tag{8}$$

$$y_{it} = \sum_j U^{ij}\, m_{jt} + e^i\,, \tag{9}$$

We can calculate the gradients of the loss function w.r.t. the parameters of the RNN.

$$\left[\nabla^{(U)} L\right]_{ij} \equiv \frac{\partial L}{\partial U^{ij}} = \sum_t \left[\nabla^{(y)} L\right]_{it} m_{jt}\,, \tag{10}$$

$$\left[\nabla^{(e)} L\right]_{i} \equiv \frac{\partial L}{\partial e^{i}} = \sum_t \left[\nabla^{(y)} L\right]_{it}\,. \tag{11}$$

Define an ancillary gradient

$$\left[\nabla^{(m)} L\right]_{it} \equiv \frac{\partial L}{\partial m_{it}} = \sum_j U^{ji} \left[\nabla^{(y)} L\right]_{jt} + \left[\nabla^{(m_n)} L\right]_{it}\,, \tag{12}$$

where the first term in the r.h.s. of the last equation comes from the dependence of L on m_{it} via y_t, and the second term comes from the dependence of L on m_{it} via $m_{j\,t+1}$ (for $t = T$ we seed $\left[\nabla^{(m_n)} L\right]_{iT} = 0$ by hand),

$$\left[\nabla^{(m_n)} L\right]_{it} = \sum_j \left[\hat{\nabla}^{(m)} L\right]_{j\,t+1} V^{ji}\,, \tag{13}$$

where we defined auxiliary quantities

$$m_{it} \equiv \tanh \hat{m}_{it}\,, \tag{14}$$

$$[\hat{\nabla}^{(m)} L]_{it} \equiv \frac{\partial L}{\partial \hat{m}_{it}} = \left[\nabla^{(m)} L\right]_{it} (1 - m_{it}^2)\,. \tag{15}$$

We see that $\left[\nabla^{(m)} L\right]$ requires knowing $\left[\nabla^{(m_n)} L\right]$, which requires knowing $\left[\hat{\nabla}^{(m)} L\right]$, which requires knowing $\left[\nabla^{(m)} L\right]$. To avoid logical loop we need to carefully resolve this calculation. This is done by backpropagating: we calculate the gradients in the descending t order, where for the largest t we use the vanishing next gradient $\left[\nabla^{(m_n)} L\right]_{iT} = 0$, since there are no layers/time steps afterwards, and update the next gradient in accord with (13) for the step $T-1$ (therefore at the step t in $[\hat{\nabla}^{(m)} L]_{jt}$ in the r.h.s. of (13) we use time index t and save this value for step $t-1$). This procedure is repeated down for smaller t.

In terms of these we can calculate

$$\left[\nabla^{(W)} L\right]_{ij} \equiv \frac{\partial L}{\partial W^{ij}} = \sum_t [\hat{\nabla}^{(m)} L]_{it}\, x_{jt}\,, \tag{16}$$

$$\left[\nabla^{(V)} L\right]_{ij} \equiv \frac{\partial L}{\partial V^{ij}} = \sum_t [\hat{\nabla}^{(m)} L]_{it}\, m_{j\,t-1}\,, \tag{17}$$

$$\left[\nabla^{(b)} L\right]_{ij} \equiv \frac{\partial L}{\partial b^{i}} = \sum_t [\hat{\nabla}^{(m)} L]_{it}\,. \tag{18}$$

Once we know the gradients we know by how much we need to shift each of the parameters W, V, U, b, e in order to decrease the loss function. If p is one of those parameters which we want to adjust then one way to do this is an adjusted gradient method,

$$p \to p - r\,[\nabla^{(p)} L]\,\frac{1}{\rho}\,, \tag{19}$$

where

$$\rho = \sqrt{\sum_n [\nabla^{(p)} L]^2}\,, \tag{20}$$

and the sum is done over all the iterations we have performed up till now, inclusive. This is an adaptive learning rate, which readjusts the coefficients in front of the gradient, besides the constant $r \simeq 0.1$.