RECURRENT NEURAL NETWORKS (RNNS): A GENTLE INTRODUCTION AND OVERVIEW

Nguyen Hoang Tan
Department of Computer Science
University of Information Technology
Linh Trung, Thu Duc
21521313@gm.uit.edu.vn

ABSTRACT

State-of-the-art solutions in the areas of "Language Modelling & Generating Text", "Speech Recognition", "Generating Image Descriptions" or "Video Tagging" have been using Recurrent Neural Networks as the foundation for their approaches. Understanding the underlying concepts is therefore of tremendous importance if we want to keep up with recent or upcoming publications in those areas. In this work we give a short overview over some of the most important concepts in the realm of Recurrent Neural Networks which enables readers to easily understand the fundamentals such as but not limited to "Backpropagation through Time" or "Long Short-Term Memory Units" as well as some of the more recent advances like the "Attention Mechanism" or "Pointer Networks". We also give recommendations for further reading regarding more complex topics where it is necessary.

1 Introduction & Notation

Recurrent Neural Networks (RNNs) are a type of neural network architecture which is mainly used to detect patterns in a sequence of data. Such data can be handwriting, genomes, text or numerical time series which are often produced in industry settings (e.g. stock markets or sensors). However, they are also applicable to images if these get respectively decomposed into a series of patches and treated as a sequence. On a higher level, RNNs find applications in Language Modelling & Generating Text, Speech Recognition, Generating Image Descriptions or Video Tagging. What differentiates Recurrent Neural Networks from Feedforward Neural Networks also known as Multi-Layer Perceptrons (MLPs) is how information gets passed through the network. While Feedforward Networks pass information through the network without cycles, the RNN has cycles and transmits information back into itself. This enables them to extend the functionality of Feedforward Networks to also take into account previous inputs $X_{0:t-1}$ and not only the current input X_t. This difference is visualised on a high level in Figure 1. Note, that here the option of having multiple hidden layers is aggregated to one Hidden Layer block H. This block can obviously be extended to multiple hidden layers.

1.1 Model Architecture

RNNs add recurrent layers to the NN (Neural Network) model and consists of three sets of layers (input, recurrent, and output). Input layers take the sensor output and convert it into a vector that conveys the features of the input. These are followed by the recurrent layers, which provide feedback. In most recent recurrent layer models, memory cells exist as well. Subse- quently, the model completes similarly to most NN models with Fully Connected (FC) layers and an output layer that can be a softmax layer. FC layers and the output layer are grouped into the set of output layer.

We can describe this process of passing information from the previous iteration to the hidden layer with the mathematical notation proposed in. For that, we denote the hidden state and the input at time step t respecively as $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ and $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ where n is number of samples, d is the number of inputs of each sample and h is the number of hidden units. Further, we use a weight matrix $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, hidden-state-to-hidden-state matrix $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ and a bias parameter $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$. Lastly, all these informations get passed to a activation function ϕ which is usually a logistic sigmoid or tanh function to prepair the gradients for usage in backpropagation. Putting all these notations together yields Equation

1 as the hidden variable and Equation 2 as the output variable.

$$\mathbf{H}_t = \phi_h(\mathbf{X}_t \, \mathbf{W}_{xh} + \mathbf{H}_{t-1} \, \mathbf{W}_{hh} + \mathbf{b}_h) \tag{1}$$

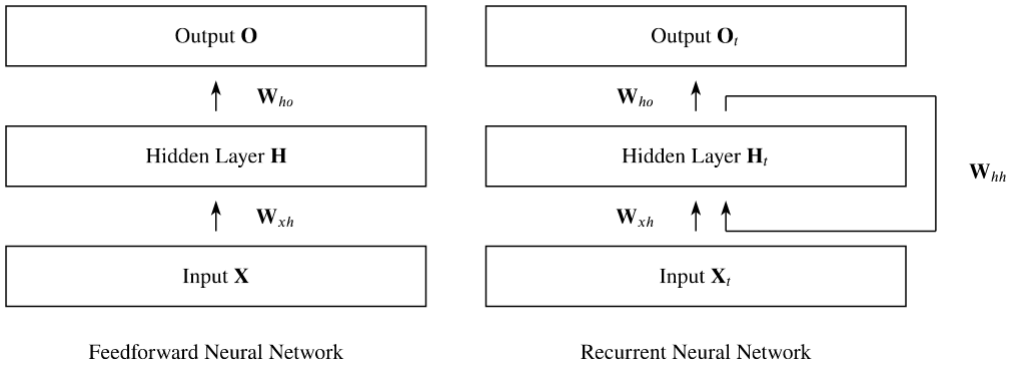

Figure 1: Visualisation of differences between Feedfoward NNs and Recurrent NNs

$$\mathbf{O}_t = \phi_o(\mathbf{H}_t \, \mathbf{W}_{ho} + \mathbf{b}_o) \tag{2}$$

Since $\mathbf{H}_t$ recursively includes $\mathbf{H}_{t-1}$ and this process occurs for every time step the RNN includes traces of all hidden states that preceded $\mathbf{H}_{t-1}$ as well as $\mathbf{H}_{t-1}$ itself.

If we compare that notation for RNNs with similar notation for Feedforward Neural Networks we can clearly see the difference we described earlier. In Equation 3 we can see the computation for the hidden variable while Equation 4 shows the output variable.

$$\mathbf{H} = \phi_o(\mathbf{X} \, \mathbf{W}_{xh} + \mathbf{b}_h) \tag{3}$$

$$\mathbf{O} = \phi_o(\mathbf{H} \, \mathbf{W}_{ho} + \mathbf{b}_o) \tag{4}$$

A RNN uses a simple nonlinear activation function in every unit. However, such simple structure is capable of modelling rich dynamics, if it is well trained through timesteps.

1.2 Activation Function

For linear networks, multiple linear hidden layers act as a single linear hidden layer. Nonlinear functions are more powerful than linear functions as they can draw nonlinear boundaries. The nonlinearity in one or successive hidden layers in a RNN is the reason for learning input-target re- lationships. .The “sigmoid”, “tanh”, and rectified linear unit (ReLU) have received more attention than the other activation functions recently. The “sigmoid” is a common choice, which takes a real-value and squashes it to the range [0, 1]. This activation function is normally used in the output layer, where a cross-entropy loss function is used for training a classifica- tion model. The “tanh” and “sigmoid” activation functions are defined as [1]

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \quad \sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

Respectively, the “tanh” activation function is in fact a scaled “sigmoid” activation function such as

$$\sigma(x) = \frac{tanh(x/2) + 1}{2} \tag{6}$$

ReLU is another popular activation function, which is open- ended for positive input values, defined as

$$y(x) = max(x, 0) \tag{7}$$

[1] Tanh and sigmoid, both functions require floating-point division and exponential operations, which are expensive in terms of hardware resources. In order to have an efficient implementation for an RNN, non-linear function approximations are implemented in hardware. This approximation should satisfy a balance between high accuracy and low hardware cost.
Look-up tables (LUTs), Piecewise linear approximation, Hard tanh / Hard sigmoid, RALUT.

Selection of the activation function is mostly dependent on the problem and nature of the data. For example, "sigmoid" is suitable for networks where the output is in the range [0, 1]. However, the "tanh" and "sigmoid" activation functions saturate the neuron very fast and can vanish the gradient. Despite "tanh", the non-zero centered output from "sigmoid" can cause unstable dynamics in the gradient updates for the weights. The ReLU activation function leads to sparser gradients and greatly accelerates the convergence of stochastic gradient descent (SGD) compared to the "sigmoid" or "tanh" activation functions. ReLU is computationally cheap, since it can be implemented by thresholding an activation value at zero. However, ReLU is not resistant against a large gradient flow and as the weight matrix grows, the neuron may remain inactive during training.

1.3 Loss Function

Loss function evaluates performance of the network by comparing the output $\mathbf{y}_t$ with the corresponding target $\mathbf{z}_t$ defined as

$$L(\mathbf{y}, \mathbf{z}) = \sum_{t=1}^{T} L_t(\mathbf{y_t}, \mathbf{z_t}) \tag{8}$$

that is an overall summation of losses in each timestep. Selection of the loss function is problem dependent. Some popular loss function are Euclidean distance and Hamming distance for forecasing of real-values and cross-entropy over probablity distribution of outputs for classification problems.

2 Training Recurrent Neural Networks

Efficient training of a RNN is a major problem. The difficulty is in proper initialization of the weights in the network and the optimization algorithm to tune them in order to minimize the training loss. The relationship among network parameters and the dynamics of the hidden states through time causes instability. A glance at the proposed methods in the literature shows that the main focus is to reduce complexity of training algorithms, while accelerating the convergence. However, generally such algorithms take a large number of iterations to train the model. Some of the approaches for training RNNs are multi-grid random search, time-weighted pseudonewton optimization, GD, extended Kalman filter (EKF), Hessian-free, expectation maximization (EM), approx- imated Levenberg-Marquardt, and global optimization algorithms.

2.1 Initialization

Initialization of weights and biases in RNNs is critical. A general rule is to assign small values to the weights. A Gaussian draw with a standard deviation of 0.001 or 0.01 is a reasonable choice. The biases are usually set to zero, but the output bias can also be set to a very small value. However, the initialization of parameters is dependent on the task and properties of the input data such as dimensionality . Setting the initial weight using prior knowledge or in a semi-supervised fashion are other approaches.

2.2 Gradient-based Learning Methods

Gradient descent (GD) is a simple and popular optimization method in deep learning. The basic idea is to adjust the weights of the model by finding the error function derivatives with respect to each member of the weight matrices in the model. To minimize total loss, GD changes each weight in proportion to the derivative of the error with respect to that weight, provided that the non-linear activation functions are differentiable. The GD is also known as batch GD, as it computes the gradient for the whole dataset in each optimization iteration to perform a single update as

$$\theta_{t+1} = \theta_t - \frac{\lambda}{U} \sum_{k=1}^{U} \frac{\partial L_k}{\partial \theta} \tag{9}$$

where U is size of training set, λ is the learning rate, and θ is set of parameters. This approach is computationally expensive for very large datasets and is not suitable for online training

If you are familiar with training techniques for Feedforward Neural Networks such as backpropagation one question which might arise is how to properly backpropagate the error through a RNN. Here, a technique called Backpropagation Through Time (BPTT) is used. However, computing error-derivatives through time is difficult. This is mostly due to the relationship among the parameters and the dynamics of the RNN, that is highly unstable and makes GD ineffective. Gradient-based algorithms have difficulty in capturing depen- dencies as the duration of dependencies increases. The

derivatives of the loss function with respect to the weights only consider the distance between the current output and the corresponding target, without using the history information for weights updating. RNNs cannot learn long-range temporal dependencies when GD is used for training. This is due to the exponential decay of gradient, as it is back-propagated through time, which is called the vanishing gradient problem. In another occasional situation, the back-propagated gradient can exponentially blow-up, which increases the variance of the gradients and results in very unstable learning situation, called the exploding gradient problem.

2.3 Back-propagation through time (BPTT)

Backpropagation Through Time (BPTT) is the adaption of the backpropagation algorithm for RNNs. In theory, this unfolds the RNN to construct a traditional Feedfoward Neural Network where we can apply backpropagation. For that, we use the same notations for the RNN as proposed before.

Since we have three weight matrices $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$ and $\mathbf{W}_{ho}$ we need to compute the partial derivative to each of these weight matrices. With the chain rule which is also used in normal backpropagation we get to the result for $\mathbf{W}_{ho}$ shown in Equation 6.

$$\frac{\partial L}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{H}_t \tag{10}$$

For the partial derivative with respect to $\mathbf{W}_{hh}$ we get the result shown in Equation 7.

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{H}_t} \cdot \frac{\partial \mathbf{H}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \cdot \frac{\partial \mathbf{H}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{hh}} \tag{11}$$

For the partial derivative with respect to $\mathbf{W}_{xh}$ we get the result shown in Equation 8.

$$\frac{\partial L}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{H}_t} \cdot \frac{\partial \mathbf{H}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \cdot \frac{\partial \mathbf{H}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{xh}} \tag{12}$$

Since each Ht depends on the previous time step we can substitute the last part from above equations to get Equation 9 and Equation 10.

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} \frac{\partial \mathbf{H}_t}{\partial \mathbf{H}_k} \cdot \frac{\partial \mathbf{H}_k}{\partial \mathbf{W}_{hh}} \tag{13}$$

$$\frac{\partial L}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} \frac{\partial \mathbf{H}_t}{\partial \mathbf{H}_k} \cdot \frac{\partial \mathbf{H}_k}{\partial \mathbf{W}_{xh}} \tag{14}$$

The adapted part can then further be written as shown in Equation 11 and Equation 12.

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} (\mathbf{W}_{hh}^{T})^{t-k} \cdot \mathbf{H}_k \tag{15}$$

$$\frac{\partial L}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} (\mathbf{W}_{hh}^{T})^{t-k} \cdot \mathbf{X}_k \tag{16}$$

From here, we can see that we need to store powers of Wk hh as we proceed through each loss term ‘t of the overall loss function L which can become very large. For these large values this method becomes numerically unstable since eigenvalues smaller than 1 vanish and eigenvalues larger than 1 diverge. One method of solving this problem is truncate the sum at a computationally convenient size. When you do this, you’re using Truncated BPTT. This basically establishes an upper bound for the number of time steps the gradient can flow back to. One can think of this upper bound as a moving window of past time steps which the RNN considers. Anything before the cut-off time step doesn’t get taken into account. Since BPTT basically unfolds the RNN to create a new layer for each time step we can also think of this procedure as limiting the number of hidden layers.

Acknowledgments

.....