Recurrent Neural Networks: Exploding, Vanishing Gradients & Reservoir Computing

Authors: M. Mattheakis, P. Protopapas

1 Exploding and Vanishing Gradient

Training a Recurrent Neural Network (RNN) seems to simple since we have just a set of weight matrices, however, it is extremely hard due to its recurrent connections. We can analyze and understand the reason of this hardness by using the tool of back propagation and chain rule. For instance, as we multiply all the weight matrices in forward propagation we need to do the same in the back propagation. As we go backward, the signal may become too strong or too weak; this is the *gradient exploding* or *vanishing* problem, respectively. Essentially, as we go further back in time the gradients become stronger or weaker, hence for few times steps we do not observe the exploding or vanishing problem but it appears rabidly as the time sequence increase. Vanishing gradients make it difficult to know which direction the parameters should move to improve the loss functions, while exploding gradients make the learning unstable. The forward propagation for an input sequential x_t and for an output $\hat{y}_t$ of an RNN is graphically demonstrated in Fig. 1 and written:

$$h_t = g_h\left(V\, x_t + U\, h_{t-1} + b'\right), \tag{1}$$

$$\hat{y}_t = g_y\left(W\, h_t + b\right) \tag{2}$$

where Eq. (1) is the hidden-to-hidden recurrence with an arbitrary activation function g_h and Eq. (2) accounts to the output layer with activation function g_y. The weight matrices V and W associate, respectively, to the input and output layers while U is the recurrent weight matrix on which we focus on this notes.

In order to calculate the total loss function L with respect to the entire sequence in the interval $t = (1, T)$ we essentially have to sum up the loss function in all the time steps, hence

$$L = \sum_{t=1}^{T} L_t. \tag{3}$$

If we choose to minimize the loss function with respect to the hidden-to-hidden recurrence weights U we need to calculate the derivative

$$\frac{dL}{dU} = \sum_{t=1}^{T} \frac{dL_t}{dU}, \tag{4}$$

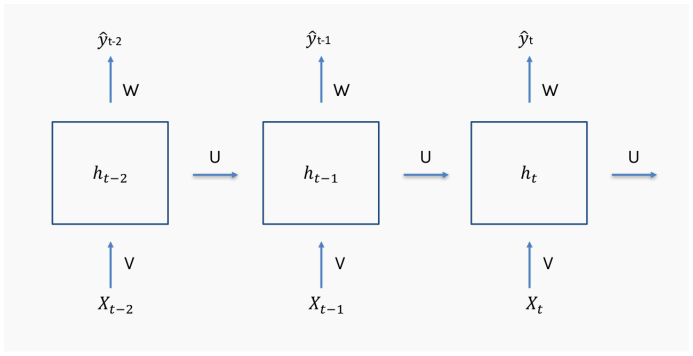

Figure 1: Recurent Neural Network.

Even computing the derivative for the loss function in a single time step it requires a very large chain rule application because it essentially demands all the previous time step calculations. That becomes obvious by using the chain rule

$$\begin{aligned}\frac{dL_t}{dU} &= \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial U}\\ &= \sum_{k=1}^{t}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial U}. \end{aligned} \tag{5}$$

We explore deeper the Eq. (5) by investigating the term $\partial h_t/\partial h_k$. To compute this term we have to use again the chain rule as:

$$\begin{aligned}\frac{\partial h_t}{\partial h_k} &= \frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial h_{t-2}}\cdots\frac{\partial h_k}{\partial h_{k-1}}\\ &= \prod_{j=k+1}^{t}\frac{\partial h_j}{\partial h_{j-1}}. \end{aligned} \tag{6}$$

The expression in the product of Eq. (6) is a derivative between two vectors and thus, it is the Jacobian matrix of the state to state transition function. Hence, the gradient $\partial h_t/\partial h_k$ is a product of Jacobian matrices each associated with a step in the forward computation. We explore further the term in the product (6) by using Eq. (1), then we obtain

$$\frac{\partial h_j}{\partial h_{j-1}} = U^T g', \tag{7}$$

with prime denotes derivate with respect to h_{t-1}. Taking the norm of the Jacobian (7) yields

$$\left\|\frac{\partial h_j}{\partial h_{j-1}}\right\| = \left\|U^T g'\right\| \leq \left\|U^T\right\| \left\|g'\right\| = \beta_U\, \beta_h, \tag{8}$$

where we assumed that β_U, β_h are the values of the norms $\left\|U^T\right\|$, $\left\|g'\right\|$, respectively. Combining Eqs. (6) and (8) we get

$$\left\|\frac{\partial h_t}{\partial h_k}\right\| = \left\|\prod_{h=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}\right\| \leq (\beta_U\, \beta_h)^{t-k}\,. \tag{9}$$

Subsequently, as the sequence becomes larger and larger ($t \to \infty$) the term 9 blows up or becomes very small whether the product ($\beta_U\, \beta_h$) is larger or smaller than one, respectively. This is the *exploding* or *vanishing* gradient problem and happens very quickly since t is on the exponent.

We can overpass the problem of exploding or vanishing gradients by using the *clipping gradient* method, by using special RNN architectures with leaky units such as Long-Short-Term-Memory (LSTM) and Gated Recurrent Units (GRU), or by using echo state RNNs. In the following section we discuss about a quite special echo state RNN which is called Reservoir Computing.

Exercise: Suppose a RNN with two time steps (2-sequence) and calculate the backpropogation by using the chain rule in the derivatives. Insightful exercise.

2 Reservoir Computing

The recurrent weights mapping from h_{t-1} to h_t hidden states and the input weights mapping from x_t to h_t are some of the most difficult parameters to learn in an RNN. One approach to avoid this difficulty is to fix the input and the recurrent weights such that the recurrent hidden units do a good job of capturing the history of the past inputs, and learn only the output weights. This is the backbone idea for the *echo state networks* (ESN) and *liquid state machines*. This networks are called *reservoir computing* (RC) to denote the fact that the hidden units form a reservoir of temporal features that may capture different aspects of the history inputs (Fig. 2). An RC RNN work very well in the prediction task including forecasting the weather, controlling complex dynamical systems, pattern recognition, and predicting time series.

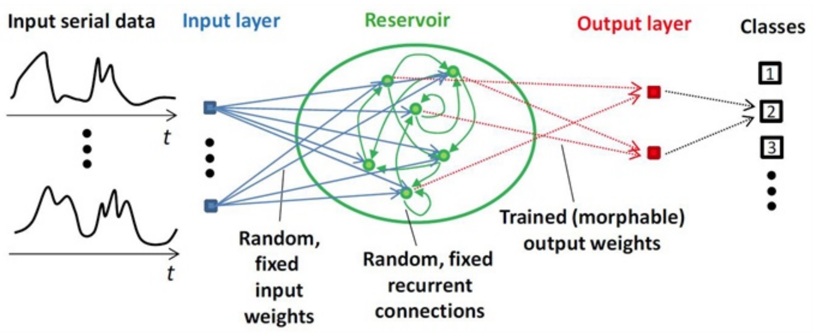
Input serial data
Input layer
Reservoir
Output layer
Classes
t
t
1
2
3
Random, fixed input weights
Random, fixed recurrent connections
Trained (morphable) output weights

Figure 2: Reservoir Computing Network.

An insight way to think RC RNNs is that they map an arbitrary length sequence (the history of the inputs up to the time t) into a high-dimensional fixed-length feature vector (the recurrent state h_t). Then a linear predictor, which is typically a linear regression, is applied to solve the problem of the interest. This makes RC RNNs very efficient and drastically speeds up the training since we only need to train the output weights by using the well known, from linear regression, learning algorithms. An important question that may naturally be asked is how to set the input and the recurrent weights so that a rich set of sequential history can be represented in the RNN state. The answer that is given, in the context of RC, is to view the recurrent net as a dynamical system and set the input and the recurrent weights such that the dynamical system is near the edge of stability. The stability is crucial for avoiding the exploding/vanishing gradients because it essentially means that the eigenvalues of the state to state Jacobian are close to one and hence the gradients do not explode or vanish (see Eq. (9)). Moreover, RC very often employs leaky hidden units that makes the hidden neurons to partially remember the its previous activation. This type of neurons performs a leaky integration of its activation from previous time steps and thus, they are called *leaky integrator neurons*. In summary, an RC is distinguished from traditional feed-forward NNs by the following qualities:

- The network nodes each have distinct dynamical behavior
- Time delays of signal may occur along the network links
- The network hidden part has recurrent connections
- The input and internal weights are fixed and randomly chosen
- Only the output weight are adjusted during the training.

Let us describe the formulation of the RC RNN of Fig. 2. We are seeking for a functional relationship between M given sequential inputs $\mathbf{u}(t) \in \mathrm{IR}^M$ and P desired outputs $\hat{\mathbf{y}}(t) \in \mathrm{IR}^P$, with $\{t = 1, 2, \cdots, T\}$ and T is the number of data points in the training dataset $\{\mathbf{u}(t), \mathbf{y}(t)\}_{t=1}^T$. We suppose a reservoir with N dynamical recurrent hidden nodes whose state vector is $\mathbf{r}(t) \in \mathrm{IR}^N$. The reservoir nodes have recurrent connections denoted by the matrix $W_{\text{res}} \in \mathrm{IR}^{N\times N}$. The inputs $\mathbf{u}$ are connected to the reservoir nodes through a linear input layer $W_{\text{in}} \in \mathrm{IR}^{M\times N}$. In turn, the reservoir nodes are connected to the outputs through a linear output layer $W_{\text{out}} \in \mathrm{IR}^{N\times P}$. The hidden states and the reservoir dynamics are given by

$$\mathbf{r}(t + \Delta t) = (1 - \alpha)\mathbf{r}(t) + \alpha\, f\,(W_{\text{res}}\mathbf{r}(t) + W_{\text{in}}\mathbf{u}(t) + \mathbf{b}) \tag{10}$$

where $\Delta t \ll 1$, $\mathbf{b} \in \mathrm{IR}^N$ is a bias vector, and $f(.)$ is the activation function usually chosen to be sigmoid or tanh(). Equation (10) describes leaky units with α is the leakage rate parameter ($0 < \alpha \leq 1$) that controls how fast the reservoir evolves; for instance, $\alpha \to 0$ the reservoir evolves slowly. The weighted adjacent matrix W_{res}, the input matrix, and the bias vector are initially randomly drawn and then they are fixed. The output vector is taken to be a linear function of the reservoir state and defined as:

$$\hat{\mathbf{y}}(t) = W_{\text{out}}\mathbf{r}(t) + \mathbf{c} \tag{11}$$

where $\mathbf{c} \in \mathrm{IR}^P$ is the bias vector of the output layer. For the prediction task, we adjust W_{out} and $\mathbf{c}$ by using a finite duration training data samples so that the resulting output represents the input data in a least-square sense. In other words, we determine the W_{out} and $\mathbf{c}$ by minimizing the loss function

$$\mathcal{L} = \sum_{t=1}^{T} ||W_{\text{out}}\mathbf{r}(t) + \mathbf{c} - \mathbf{y}(t)||^2 + \beta \operatorname{Tr}\left(W_{\text{out}} W_{\text{out}}^T\right) \tag{12}$$

in the training set, where the second term is a regularization included to avoid overfitting with β is the *ridge regression parameter* which typically takes small values, and the norm of a vector reads $||\mathbf{q}|| = \mathbf{q}^T\mathbf{q}$. The great advantage of RC comparing to other RNNs is that we have to estimate only the weights of the output layer, hence the training becomes very efficient and computationally feasible for relatively large N.

In addition to the parameters α, Δt, $\mathbf{b}$ in Eq. (10), the reservoir dynamics depends on the parameter D, ρ, and σ, which govern the random generation of the W_{in} and W_{res}. In particular, the adjacency matrix W_{res} is typically sparse with density of non-zero matrix elements given by D/N, so the average degree of a reservoir node is D. The values of the non-zero elements are randomly drawn independently from a uniform distribution between -1 and 1. We then uniformly rescale all the elements of the W_{res} so that the largest value of the magnitude of its eigenvalues becomes ρ, which is referred as the *spectral radius* of the W_{res}. The elements of the W_{in} are randomly chosen from a uniform distribution in the range $[\sigma, \sigma]$. Unfortunately, there is not yet any method for optimizing these parameters, so we usually chose them in a heuristic way (no free lunch). This is the disadvantage of the RC network.

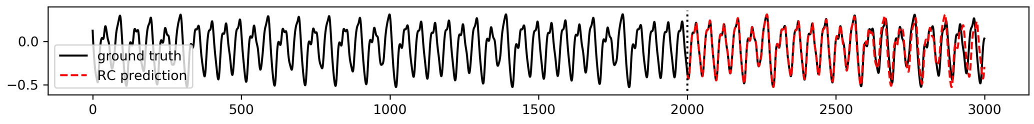

Figure 3: Predict the chaotic time-series of the Mackey-Glass system using RC. Black line is the ground truth (data) whereas red is the RC prediction. For the implementation we use the python RC code of the Ref. [5].

As implementation we employ an RC to predict a chaotic time-series of the Mackey-Glass system which is a standard benchmark system for time series prediction and also this is the example that was discussed in the seminal paper [2] where the RC was introduced. We have one input $u(t)$, one output $\hat{y}(t)$, and employ $N = 1000$ reservoir neurons. A 2000 step sequence is used for training the output layer and predict the time series for 1000 future step. We found that the hyper-parameters $D/N = 0.2$, $\rho = 1.5$, and $\sigma = 1$, work well in the specific task. The results are represented by Fig. 3 where the Mean Square Error (MSE) in the testing data given by

$$\text{MSE} = \sqrt{\langle(\hat{y}(t) - y(t))^2\rangle}$$

where $\langle . \rangle$ shows averaging in the testing time sequence, is calculated to be MSE = 0.09 verifying the good predictions of the RC network.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press (2017).

[2] H. Jaeger, and H. Haas, *Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication*, Science **304**, 78-80 (2014).

[3] M. Lukosevicius, and H. Jaeger,*Reservoir computing approaches to recurrent neural network training*, Computer Science Review **3** (3), 127-149 (2009).

[4] Z. Lu, J. Pathak, B. Hunt, M. Girvan, R. Brockett, and E. Ott, *Reservoir observers: Model-free inference of unmeasured variables in chaotic systems*, Chaos **27**, 041102 (2017).

[5] https://github.com/cknd/pyESN

[6] G. N. Neofotistos, M. Mattheakis, G. Barbaris, J. Hitzanidi, G. P. Tsironis, and E. Kaxiras, *Machine learning with observers predicts complex spatiotemporal behavior*, Frontier of Phys. - Quantum Computing, **7**, 24, (2019).