

Recurrent Neural networks: A critical review of the standard training procedure

Why do the error gradients tend to explode or vanish during backpropagation-through-time of the error through a standard recurrent neural network?

Mathematics

Word Count: 3784

Table of Contents

INTRODUCTION	2
CONTEXT	2
FORWARD AND BACKPROPAGATION IN RECURRENT NEURAL NETWORKS.....	6
EXPLODING AND VANISHING GRADIENTS.....	8
NONLINEAR DYNAMICS AND ITS CONTRIBUTION	9
GEOMETRICAL INTERPRETATION	12
CONCLUSION: CONSIDERATIONS AND FUTURE DIRECTIONS.....	13
REFERENCES.....	15
APPENDIX A: STANDARD BACKPROPAGATION DETAILS	19
1 Backpropagation	19
1.1 Intrinsic use of the chain rule	20
1.2 Updates	21
APPENDIX B: DERIVATION OF BPTT LONG TERM DEPENDENCY EXPLOSION/VANISHING FORMULA	23

Introduction

Connectionism – a school in machine learning – has first made a presence back in the 50s. Due to the constant shift in foresight, the frequency of research with regards to connectionism has oscillated. According to *Deep learning* “there have been three waves of development” (40-60, 80-90, 06-) and that although neural networks fit under the category of connectionism the reference ‘deep learning’ has been popularized (Goodfellow, et al., 2001, p.12). This third wave was largely sparked by the sudden increase in computing power and the provision of development languages such as CUDA, a C-like language allowing one to write programs that run directly on the GPU (Ebersole, 2012, ‘CUDA’, 2019). Over the course of the development as researchers started to specialize on tasks, so did the algorithms. In this extended essay I will be exploring the backpropagation-through-time algorithm with the intention of answering the question: **Why do the error gradients tend to explode or vanish during backpropagation-through-time of the error for long time intervals through a standard recurrent neural network?**

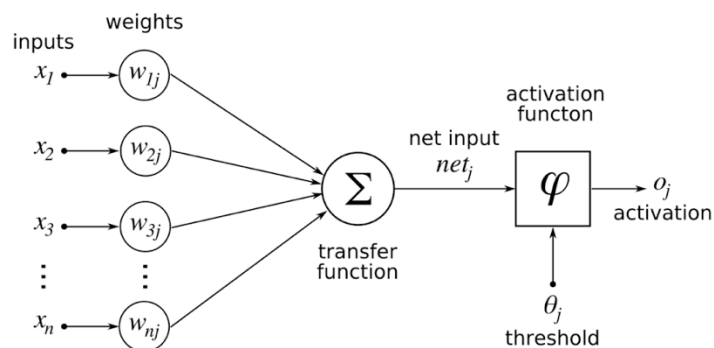


Figure 1 Source: <https://i.stack.imgur.com/ibYr3.png>

Context

To give context we should start with artificial neural networks (NNs). As indicated by its name artificial neural networks are derived from biological neural networks and inspired by neurobiological theories such as Hebbian theory (Arbib, et al., 2016, p.199, ‘Hebbian theory’, 2019). Neural networks are a set of organized neurons their arrangement depending on the framing of the problem. These neurons are interconnected using weighted connections. In the neuron as seen above these incoming weighted connections are summed and added to a bias before passing through an activation function. The purpose of the activation function is to non-linearize the incoming connection allowing for the representation of a wide range of distributions this high capacity neural networks exhibit can be qualitatively described as universal function approximation (Goodfellow, et al., 2016, p.192-194; ‘Universal approximation theorem’, 2019). To have this quality in a dense architecture (figure 1) requires a group of neurons (alternatively Maxout units could be used as discussed in Goodfellow, et al., 2013a). Neurons are most commonly grouped in what we call layers. Generally, there are three types of layers as seen in the figure below, the input layer (blue), hidden layer (orange), and output layer (purple). The input layer, in this view, is responsible for encoding the input data, the hidden layer for interpreting, and the output layer for renormalizing the data to the domain of the problem.

Keeping the aforementioned in mind, this interpretative unintuitive approach to representation is one of the reasons neural networks are deemed black box representors (i.e. the hidden layer). The nature of an output layer generally serves as a strong indicator as to the task class of network, where one with a single output unit is generally used to predict a value (regression; such as price of stock) and multiple to classify (say a flower species) where each unit is either 0 or 1. An example of such a network is depicted below. In this higher order regime, each layer consists of an intricately ordered set of operations these operations are customizable using parameters. Each of the variables on the diagram is a parameter and the interaction between all these parameters constitutes a neural network.

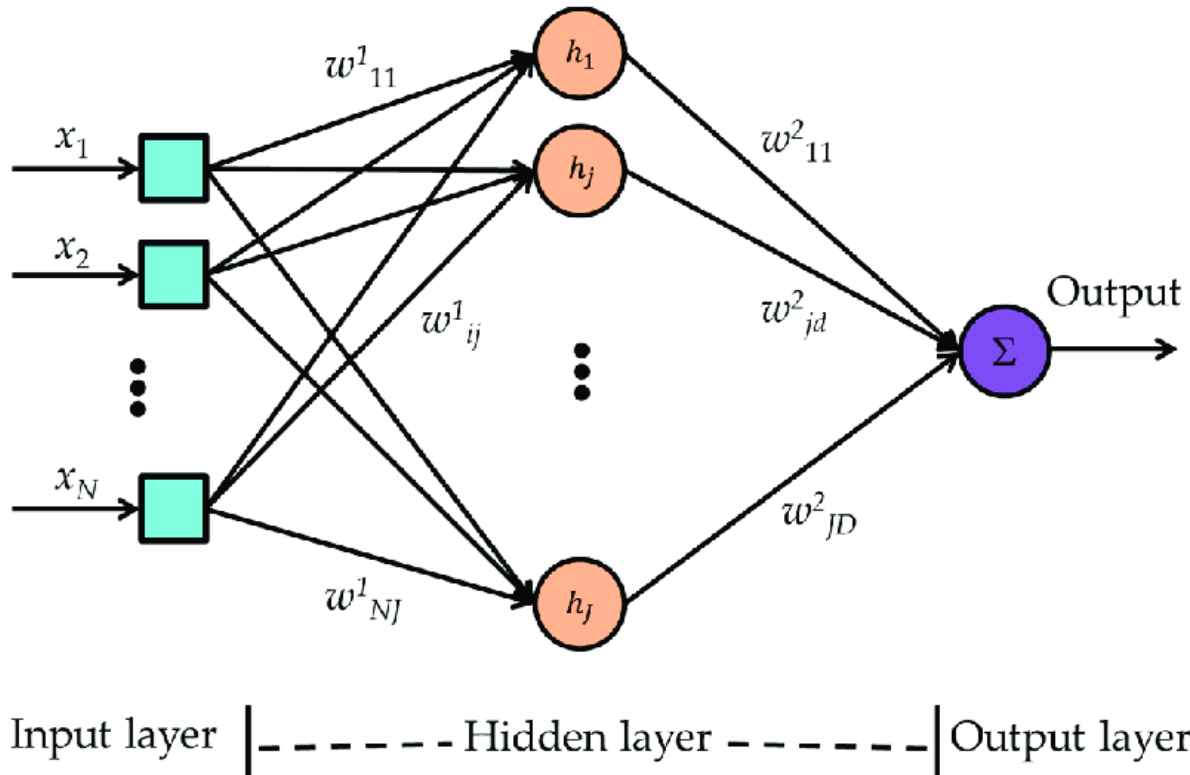


Figure 2: Neural network used for regression. Source: <https://www.researchgate.net/publication/319661423/figure/fig3/AS:538536061669377@1505408410171/Multilayer-perceptron-neural-network-block-diagram-i-is-the-sensor-index-j-is-the-index.png>

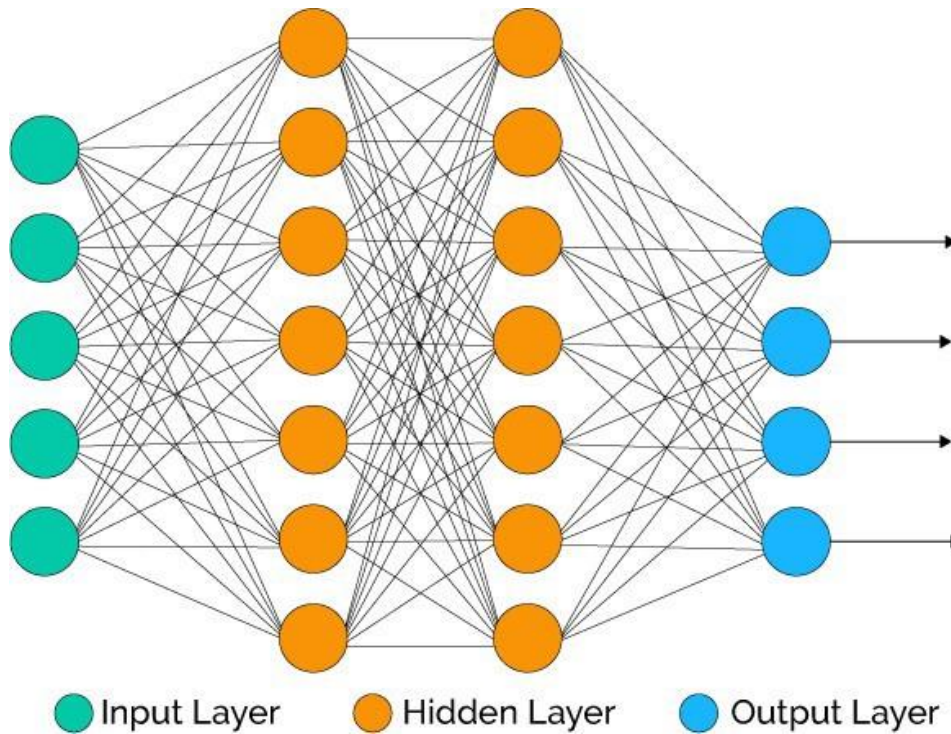


Figure 3: Neural network architecture generally used for classification. Source: https://cdn-images-1.medium.com/max/1200/1*DW0Ccmj1hZ0OvSXi7Kz5MQ.jpeg

To practicalize a network, training is required. The training process can easily be compared to pedagogy, although the ‘teaching’ process for these networks tends to be heuristic. To train the network, we would ideally want to learn the distribution of the given data but in practice the network learns to imitate it, this is partially due to the model capacity needing to be specified where an ill-specified capacity can lead to underrepresentation or overrepresentation (which are akin to overfitting and underfitting respectively in machine learning nomenclature). To learn the data distribution, backpropagation is commonplace. Backpropagation is an approach used to calculate the error gradient with respect to each parameter in neural networks. This error is intrinsic to the training process. A detailed description of backpropagation is presented in appendix A. In essence backpropagation continually updates what begins as a random initialization of the network parameters by sending the input values through the network, calculating the distance from the real solution (error) and updating each of the parameters in the network using this error, using the chain rule and the magnitude of the input to each unit to scale the error to the scale of the parameters in each unit.

The above was utilized to introduce the actual topic of this paper the backpropagation-through-time (BPTT) dilemma. BPTT was introduced alongside Recurrent Neural Network (RNN) by Rumelhart, et al. (1986a). RNNs can be seen as NNs with the addition of a recurrent connection that takes the output of that unit at the previous state (analogized as timestep due to its common use in temporal problems) in with the input. This can be unfolded as seen below to create a network-like structure similar to that of traditional NNs where each row represents a NN, these being connected. These NNs all share the same parameters, the only difference being that each

hidden unit receives the output of the unit at the previous state (timestep). Therefore, RNNs can be seen as a chain of NNs, with identical parameters, coupled through the hidden layer.

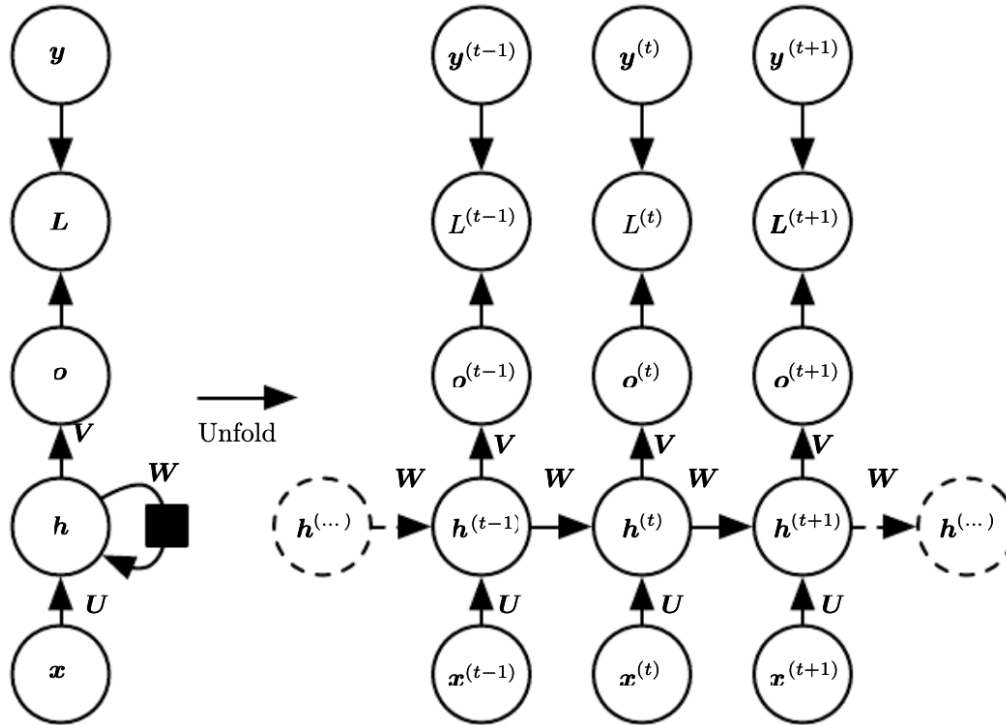


Figure 4: Source Deep learning by Ian Goodfellow

As with the NN the RNN also needs a training algorithm, backpropagation-through-time as it is called. This is simply backpropagation with error propagation through the additional edges and parameters. Unfortunately, nonlinear dynamics dictate this approach to training to be precariously unstable in many circumstances introducing the fundamental problem of deep learning (Schmidhuber, 2013).

In this essay we will explore the fundamental problem of deep learning as present in RNNs mathematically. In particular we will mathematically justify the occurrence of this issue,

qualitatively describe it using the theory of nonlinear dynamics, and interpret the issue geometrically.

Forward and Backpropagation in Recurrent Neural Networks

The premise of forward and backpropagation in RNNs is similar to that in NNs only with the additional propagation-through-time component. Backpropagation-through-time can be expressed with the standard backpropagation algorithm nested within it.

```
Back_Propagation_Through_Time(a, y)  // a[t] is the input at time t. y[t] is the output
  Unfold the network to contain k instances of f
  do until stopping criteria is met:
    x = the zero-magnitude vector; // x is the current context
    for t from 0 to n - k           // t is time. n is the length of the training sequence
      Set the network inputs to x, a[t], a[t+1], ..., a[t+k-1]
      p = forward-propagate the inputs over the whole unfolded network
      e = y[t+k] - p;               // error = target - prediction
      Back-propagate the error, e, back across the whole unfolded network
      Sum the weight changes in the k instances of f together.
      Update all the weights in f and g.
    x = f(x, a[t]);                // compute the context for the next time-step
```

Figure 5: Backpropagation-through-time pseudocode as provided on Wikipedia. Source: https://en.wikipedia.org/wiki/Backpropagation_through_time

In principal, the BPTT algorithm would backpropagate error from each of the outputs to each of the matrices in the network, the change made to each matrix at each timestep for each output is then summed and an average is found (remember the same matrices are used at each timestep for the same unit) from there the similar updates to those described in appendix A are made.

We will begin by describing the forward propagation procedure in a standard multi-layer RNN with a dense architecture, a standard training procedure (e.g., no regularization terms such as dropout as they will not be covered in this essay), and the same activation function throughout (i.e., the same network used in Hochreiter et al. 2001).

The forward propagation procedure for such an RNN can be described as:

$$y_t = \text{Sigmoid}(V \times h_t + c)$$

$$h_t = \text{ReLU}(W \times h_t + U \times x_t + b)$$

Where V, W, and U are matrices that scale values passing through the network, b, and c are biases, h_t is the hidden layer at timestep t, x_t is the input at timestep t, and y_t is the output at timestep t.

The ReLu function is defined as

$$\text{ReLU} = \max(0, x)$$

And the sigmoid is defined as (where e is the natural number)

$$\text{Sigmoid} = \frac{1}{1 + e^{-x}}$$

To make updates to the network, it is standard to use a procedure called gradient descent. Abstractly, gradient descent is designed to move towards a maximum or minimum in an n-dimensional function using the first derivative. Applied to NN's this means the matrices and biases will be adjusted towards the direction that at the current state seems to give the highest reward. This is simply done by encoding a gradient vector δ_e with n-1 real number error values, computed using the formula $\delta_i = y_t - \hat{y}_t$, where obviously we move in the direction of the negative gradient for the highest reward ($\hat{y}_t$ being the value inferred by the network and y_t being the solution). Each of the parameters in the network would then be adjusting using some fraction of this gradient with respect to themselves (how this gradient with respect to themselves is calculated is detailed below).

Let us ignore the fact that there will almost definitely be several local minima, see Martens (2010) or Kingma and Ba (2017) for methods that were developed to deal with this issue.

To calculate the gradient with respect to each location in the network using this gradient vector we simply apply BPTT. BPTT can be defined as a set of equations that express the changes made to each of the parameters with respect to the gradient of the error δ_e which can then be used by the gradient descent algorithm to update the parameters in the network. δ_p denotes the gradient with respect to a parameter p, however, we will only be defining δ_w as only this formula is required for the proof. The formula below was taken from *Deep learning* (p.376) which provides a more detailed look into BPTT for each of the individual components (Goodfellow, et al., 2016).

$$\delta_{h_t} = W^T \times \delta_{h_{t+1}} \times f'(h_{t+1}) + V^T \times \delta_t$$

Where f' is the derivative of the ReLu activation function defined as the linear piecewise function $f' = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$. As δ_{h_t} is dependent on its descendant, δ_{h_t} at a final time step f would be

$$\delta_{h_f} = V^T \times \delta_f$$

We notice that to find the gradient with respect to a hidden unit h, at timestep t, requires the use of a recursive relationship, otherwise known as the general term formula for a geometric series. Consider the following situation, a simplification of the RNN, in which we are given a chain of

infinite length, comprised of hidden units. We will omit dependence to δ_t , therefore removing all inputs and outputs except for the first input and final output, and obtain the following expression,

$$\delta_{h_t} = W^T \times \delta_{h_{t+1}} \times f'(h_{t+1})$$

We find an expression that depends on δ_{h_f} and obtain

$$\delta_{h_t} = W^{T^{f-t}} \times \delta_{h_f} \times \prod_t^f f'(h_{t+1})$$

If we factorize and assume $\delta_{h_f} = 1$ we obtain

$$\delta_{h_t} = \prod_t^f W^T f'(h_{t+1})$$

Note, do not mistake this with the update procedure which involves summing over all time intervals in which case it would look more like this

$$\Delta W = \sum_{f=1}^n \sum_{t=f-1}^1 f'(h_t) \left(\prod_{\tau=t}^f W^T f'(h_{\tau+1}) \right) h_{t-1}^T$$

Where h_{t-1}^T is used to scale the error to the particular unit and $f'(h_t)$ to pass it through the ReLu nonlinearity to the unit h_{t-1} . The transpose operator T is used as we assume vectors and require the dot product, we could extend this to matrices and tensors but do not for simplicity purposes.

For more details on how this equation was derived see appendix B.

Exploding and Vanishing Gradients

$$\delta_{h_t} = \prod_t^f W^T f'(h_{t+1})$$

The gradient equations above allow us to see purely just how critical the hidden layer weight matrix is in the backpropagation procedure, as δ for each of the parameters at timestep t simply bifurcates from δ_{h_t} , which can be analogized as the only mode of transportation to reach the parameters at that timestep. In our case this strong dependence becomes an issue as will be shown below.

To justify the issue with the strong dependence on matrix W mathematically, we resort to linear algebra, more specifically, Eigen decomposition. The process of Eigen decomposition involves decomposing a matrix into its Eigen values and Eigen vectors. Eigen values describe how a matrix scales when multiplied by any number n where $n > 0, n \in R$. An Eigen vector then is simply the vector by which the values are multiplied by to create the matrix $W = Q\Lambda Q^T$ where Q is the Eigen vector and Λ is the matrix of Eigenvalues. As one may have surmised from the use of Q on both sides of the Eigen vector, Eigen decomposition only works on real symmetric matrices, we could use Singular Value Decomposition, for universalities sake. However, for simplicity we will assume the matrix W abides by these properties. Now, to obtain δ_{h_t} we need to set W^T to a power f-t. We will neglect the transposition operator to obtain $W^{f-t} =$

$(Q\Lambda Q^T)^{f-t} = Q\Lambda^{f-t}Q^T$ as $Q^T Q = I$ and $IQ = Q$. Λ^{f-t} implies that any Eigenvalue < 1 converges to 0 and any Eigenvalue > 1 diverges, as Λ is a diagonal matrix. Moreover, this implies that the behavior of the network over long time intervals is determined primarily by the leading Eigenvalue, as can be justified by the success of the Softmax distribution (https://en.wikipedia.org/wiki/Softmax_function; Eigenvalues are conventionally ordered by size, largest to smallest).

As seen in the figure below our expression obtained above is very similar to that of Hochreiter et al (1991).

$$\dots \prod_{\tau=t-1}^{s+1} f'_{l_\tau} (net_{l_\tau}(\tau)) w_{l_\tau l_{\tau-1}} \dots$$

Figure 6: Source: Hochreiter et al. (2001, p.4)

Now that we have mathematically analyzed the issue, we will discuss alternative qualitative interpretations to the issue as provided by literature.

Nonlinear dynamics and its contribution

The theory of nonlinear dynamics can be used to describe any system that behaves nonlinearly. We will be looking specifically at a component of this theory, namely, bifurcation theory. Bifurcation theory according to Wikipedia is “the mathematical study of changes in the qualitative or topological structure of... the solutions of a family of differential equations” (‘Bifurcation theory’, 2019). Alternatively, using the theory of nonlinear dynamics, in our case, this can be described as a sudden change in the position of the attractor, the point to which the error gradient points, as once the bifurcation occurs the target value for the networks bias changes. In this section we will be considering two papers, Lorenz 1963, and Doya 1993 during our discussion of how nonlinear dynamics may contribute to the difficulty of training RNN’s.

Edward Lorenz is described as one of the earlier scientists to describe the bifurcation behavior nonlinear systems may exhibit (‘Chaos theory’, 2019). In his paper “Deterministic nonperiodic flow” Lorenz describes how certain hydrodynamical systems behave in this manner (Lorenz, 1963).

Doya (1993) acquaints one with the nonlinear dynamic formalism to describe the RNN training problem. He begins by explaining the domain of the problem and providing us with "a trivial example". Then introduces the theory as well as nomenclatures such as the bifurcations - saddle-node, Hopf, and homoclinic - and how they emerge, using the nonlinear formalism. Concluding by proposing alternative solutions to prevent the discussed issues. One potential concern is that this paper was never published in a journal, rather submitted, since this paper was considered supplementary material by the critically acclaimed textbook *Deep Learning* and numerous others (e.g. Pascanu et al., 2013) we will disregard this issue. We will utilize this source to prove the conjecture below to be true.

Conjecture: Bifurcations can be observed in the gradient-descent-powered update procedure for an RNN (as defined above).

Proof:

To give an overview the following qualities have been observed in systems that can exhibit chaotic behavior (a harbinger for bifurcative behavior):

- The system is linear, but the energy of the system is random (Lorenz, 1963, p. 131).
- “The surprising, erratic behavior comes from a nonlinear twist in the flow of energy in and out of this simple oscillator” (‘Chaos: Making a New Science’, p. 41).
- If “the governing equations are nonlinear... a constant forcing can lead to a variable response... both periodic and nonperiodic flow result from thermal forcing which... is constant” (Lorenz, 1963, p. 131, 140).

Following the points above, our system can be “governed” by a set of nonlinear differentiable equations (i.e. the activation function can be nonlinear and will be for the universal function approximation property). We will assume the networks nonlinear dynamics to be derived from its nonlinearities, such as the activation functions.

We will consider the above points as potential sources of the bifurcations observed below rather than assuming these to be the sources of error. To prove the possibility for bifurcation in the RNN we will be resorting to Doya 1993. While his work is with the continuous-time variant of RNN’s it was stated that “the discussion below also applies to discrete-time” RNN’s (p.2). The update procedure used in the work also coincides with the assumed update procedure present in our work.

Doya considered the following scenario in his proof. An RNN with a single neuron, one input, and one output has a fixed self-connecting weight of 5 and a Sigmoid activation function. We supply the network with an input of 1 and expect an output x : $0 \leq x \leq 1, x \in R$. To train the network we adjust the bias of the dense connection to the desired amount. The desired amount in this experiment is assumed to be the amount at which the error with respect to update number function begins to exhibit properties of convergence. This is important as the extent to which the training procedure continuous is decided by the author but will make negligible difference to the result, therefore unclarity in this part of the method is not beneficial for the analysis.

Following the experimental setup defined above Doya attained the following diagram, referred to as a bifurcation diagram due to its useful properties of being able to determine the points of bifurcation. This same diagram was later reviewed in Pascanu et al., 2013.

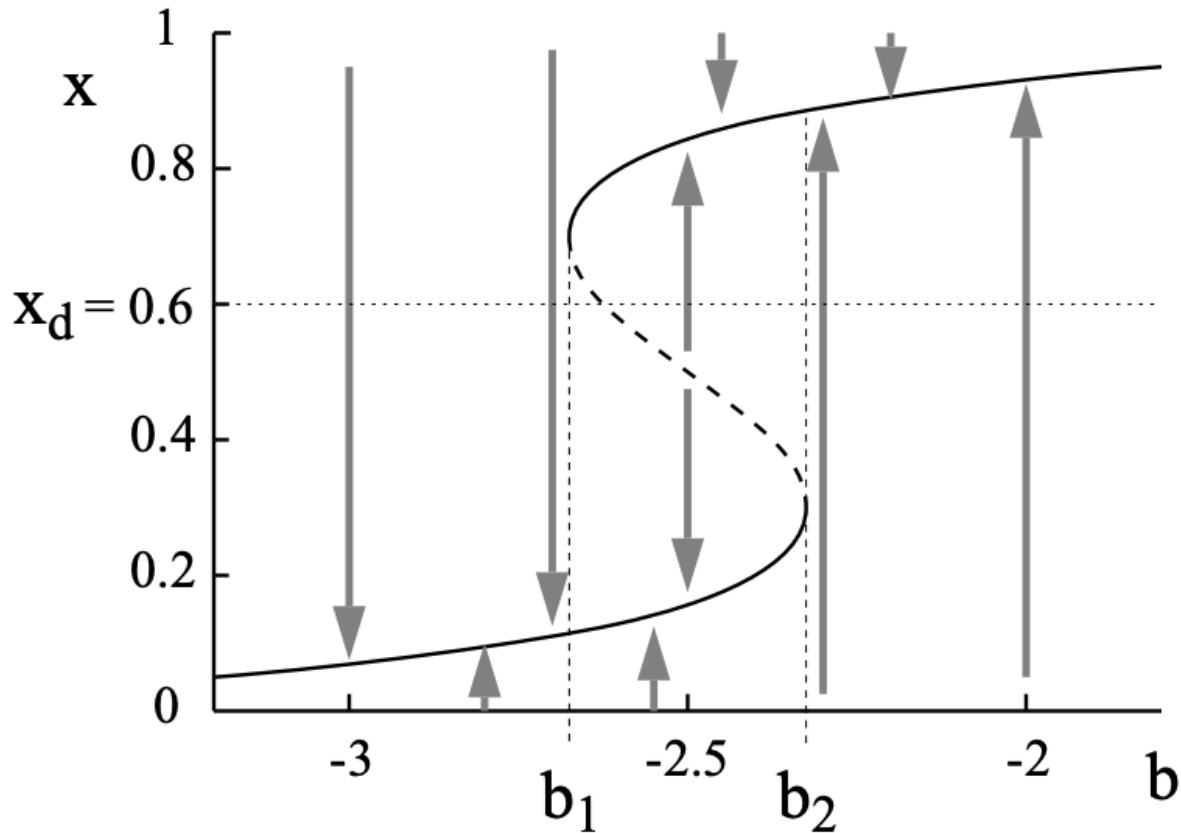


Figure 6: Bifurcation diagram taken from Doya 1993.

In the above diagram the x-axis represents the values of the bias at which RNN training converges, and the y-axis the target value of the output. In the diagram, the solid line represents the stable solutions to the problem, that is values of x exceeding approximately 0.75 or no greater than approximately 0.3. In remaining interval there is a dashed line representing a region of instability for the final values encompassed by it. In this region the output does not converge to a single value, rather as it nears an optimal value on the error surface the surface bifurcates. In this case it causes a feature that can be described as an attractor to jump from position, adapting the error surface in such a way that the target value as according to gradient descent is in a different location. This same behavior occurs as it nears the newly created attractor, in this case causing an infinite dissonance between attractors.



Figure 7: The following image depicts two basins of attraction, seperated by the maxima. Taken from wikipedia.org/wiki/Quartic_function

It is important noting that bifurcations can occur without an observable effect on the training procedure, rather it only has an observable effect if the attractor moves causes the current position of the gradient descent algorithm to be in another “basin of attraction” (Pascanu et al., 2013; i.e. drawn to another attractor). Therefore, our conjecture is only valid in the cases that the gradient descent path crosses to basin during the training of the network.

Geometrical interpretation

Alternatively, we can interpret this instability geometrically, as done by Pascanu et al to inspire an empirically backed solution that truncates the BPTT error when above a given threshold (Pascanu et al., 2013, p.4).

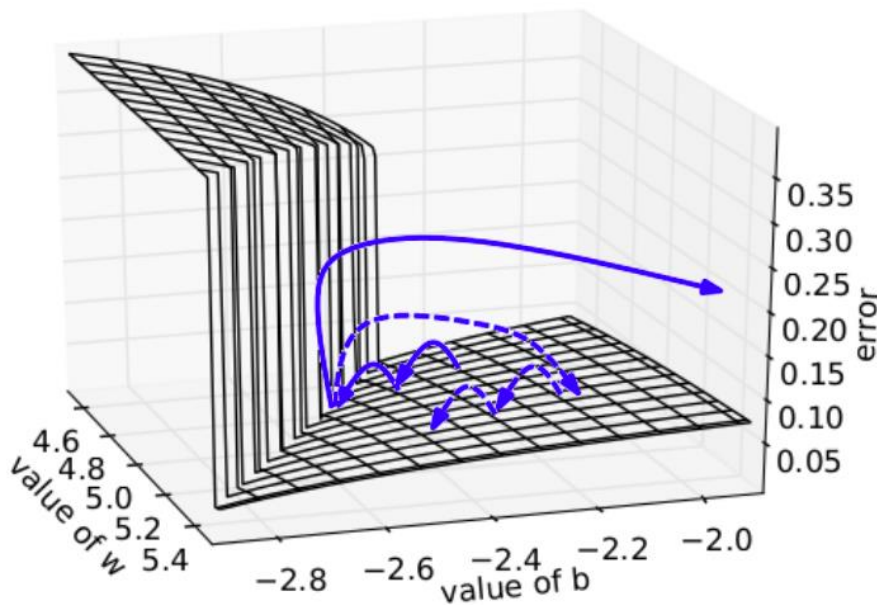


Figure 8: Plotted error surface from Pascanu et al 2013 highlighted with exemplar trajectories.

The above image describes this interpretation. Here we see a spike in error at a feature dubbed by the authors a “wall”. Using earlier formalism provided by Hochreiter’s thesis, the occurrence of this feature can be described as the boundary at which the leading Eigenvalue of the weight

matrix is closest to 1 (Hochreiter et al., 2001). The region of high error then is where the gradient has exploded (>1) and the low error region is where it vanished (<1), the magnitude of this difference implies a long-temporal dependency which in the case of this experiment is 50 timesteps. The diagram above can be recreated by following the following geometric series to $n=50$,

$$x_n = w \times Sigmoid(x_{n-1}) + b$$

$$Sigmoid = \frac{1}{1 + e^{-x}}$$

Each value for (w, b) is then plugged into the following equation to determine the y-coordinate or error:

$$E_{50} = (Sigmoid(x_{50}) - 0.7)^2$$

Here we scale the value using the activation function and then find the mean-squared-error (MSE) distance from the target value 0.7 (MSE is a standard metric used to calculate error for regression models; ‘Mean squared error’, 2019).

In the diagram we see superimposed to the error surface filled arrows which depict exemplar trajectories a standard gradient descent algorithm would follow when faced with this predicament. We see that the gradient descent algorithm attempts to get to the lowest point which is right at the edge of a steep wall, as soon as it reaches this wall, it takes a large step as standard gradient descent assumes convex shaped curvature (i.e. the higher the gradient the bigger the step).

Following this interpretation bring up alternative solutions such as truncating the gradient to decreases the size of this step taken by the standard gradient descent algorithm, as seen in the figure by the behavior outlined by the dashed line, suggested by the authors of the paper.

Conclusion: Considerations and future directions

It seems to be the case that the exploding and vanishing gradient problem can be attributed to primarily one component of the RNN, the use of the same weight matrix to represent the transformation from the same hidden unit from the previous timestep to the next timestep throughout, for long time dependencies this becomes an issue due to an exponential relationship.

One consideration in determining the validity of our analysis as brought up in Pascuna et al is that, “compared to the classical high curvature valley... we assume that the valley is wide” outlying one of the primary issues with using a one node simplification to emulate a realistic multi-layer model, in which the error surface w.r.t different model parameters will exhibit very different properties. The use of multi-layer models is less realistic as the parameter space increases exponentially as the size of the network increases. Experiment driven papers, however, have shown solutions according to this framing to be valid solutions, whereby is meant not arbitrary (Greff et al. 2017; Jozefowicz et al. 2015). Another issue we have not considered is the issue of size, as the size of the network increases the issue of ignoring how the change to one

weight matrix would another down the line is exacerbated. For further inquiry, this issue has been described by Sussillo and Abbott 2015.

A future direction may be to explore the remedies to this issue, such as the ones provided by Pascanu et al. 2013. Martens and Sutskevers work in second-order optimization methods that make use of a cheap approximation of the Hessian matrix supported by conjugate gradient approach (Martens, 2010; Martens and Sutskever, 2011; Martens and Sutskever, r.d.). More traditional work on improving the architecture such as that done by Hochreiter, et al. (1991) which is perhaps one of the most influential papers on gradient vanishing in RNNs. Gers et al. (1999), Gers et al. (2002), or Cho et al. (2014) for adapted versions of this architecture. Reservoir computing may also be an interesting direction (Jaeger, 2010).

References

- Bengio, Y., Simard, P. and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, [online] 5(2), p.157-166. Available at: <https://ieeexplore.ieee.org/document/279181>.
- Britz, D. (2019). *Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients*. [online] WildML. Available at: <http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/> [Accessed 24 Jul. 2019].
- Brownlee, J. (2018). *Long Short-Term Memory Networks Archives - Machine Learning Mastery*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/category/lstm/> [Accessed 26 Nov. 2018].
- Brownlee, J. (n.d.). *Long short-term memory networks with Python*. 1st ed. Jason Brownlee, p.All.
- Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *EMNLP*. [online] Available at: <https://arxiv.org/abs/1406.1078> [Accessed 24 Jul. 2019].
- Doya, K. (1993). Bifurcations of Recurrent Neural Networks in Gradient Descent Learning. *Institute of Electrical and Electronics Engineers*. [online] Available at: <https://www.semanticscholar.org/paper/Bifurcations-of-Recurrent-Neural-Networks-in-Doya/b57927b713a6f9b73c7941f99144165396483478> [Accessed 25 Mar. 2019].
- Ebersole, M. (2012). *What Is CUDA*. [online] The Official NVIDIA Blog. Available at: <https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/> [Accessed 24 Jul. 2019].
- En.wikipedia.org. (2019). *Bifurcation theory*. [online] Available at: https://en.wikipedia.org/wiki/Bifurcation_theory [Accessed 24 Jul. 2019].
- En.wikipedia.org. (2019). *Chaos theory*. [online] Available at: https://en.wikipedia.org/wiki/Chaos_Theory [Accessed 24 Jul. 2019].
- En.wikipedia.org. (2019). *CUDA*. [online] Available at: <https://en.wikipedia.org/wiki/CUDA> [Accessed 24 Jul. 2019].
- En.wikipedia.org. (2019). *Hebbian theory*. [online] Available at: https://en.wikipedia.org/wiki/Hebbian_theory [Accessed 24 Jul. 2019].

- En.wikipedia.org. (2019). *Mean squared error*. [online] Available at: https://en.wikipedia.org/wiki/Mean_squared_error [Accessed 24 Jul. 2019].
- En.wikipedia.org. (2019). *Universal approximation theorem*. [online] Available at: https://en.wikipedia.org/wiki/Universal_approximation_theorem [Accessed 24 Jul. 2019].
- Gers, F., Schmidhuber, J. and Cummins, F. (1999). *Learning to Forget: Continual Prediction with LSTM*. IDSIA-01-99. [online] Lugano: IDSIA. Available at: https://www.researchgate.net/publication/12292425_Learning_to_Forget_Continual_Prediction_with_LSTM [Accessed 25 Mar. 2019].
- Gers, F., Schraudolph, N. and Schmidhuber, J. (2002). Learning Precise Timing with LSTM Recurrent Networks. *Journal of Machine Learning Research*, [online] 3, p.115-143. Available at: <http://www.jmlr.org/papers/volume3/gers02a/gers02a.pdf> [Accessed 25 Mar. 2019].
- Gleick, J. (1987). *Chaos: Making a New Science*. New York: Viking Press, p.41.
- Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep learning*. 1st ed. Cambridge: MIT Press.
- Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A. and Bengio, Y. (2013). Maxout Networks. *JMLR WCP*, [online] 28(3), p.1319-1327. Available at: <https://arxiv.org/pdf/1302.4389.pdf> [Accessed 24 Jul. 2019].
- Greff, K., Srivastava, R., Koutnik, J., Steunebrink, B. and Schmidhuber, J. (2017). LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, [online] 28(10), p.2222-2232. Available at: <https://ieeexplore.ieee.org/document/7508408/>.
- Grestner, W. (2019). Hebbian Learning and Plasticity. In: M. Arbib and J. Bonaiuto, ed., *From Neuron to Cognition via Computational Neuroscience*. [online] Cambridge: MIT Press, p.199. Available at: https://books.google.nl/books/about/From_Neuron_to_Cognition_Via_Computational.html?id=jTtMDQAAQBAJ&printsec=frontcover&source=kp_read_button&redir_esc=y#v=onepage&q&f=false [Accessed 24 Jul. 2019].
- Hochreiter, S., Bengio, Y. and Frasconi, P. (2001). *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*. Undergraduate. Technical University Munich.

- Hochreiter, S. and Schmidhuber, J. (1997). LONG SHORT-TERM MEMORY. *Neural Computation*, [online] 9(8), p.N/a. Available at: <https://www.bioinf.jku.at/publications/older/2604.pdf> [Accessed 26 Nov. 2018].
- Jaeger, H. (2010). *The “echo state” approach to analysing and training recurrent neural networks – with an Erratum note*. [online] Berlin: German National Research Center for Information Technology. Available at: <https://pdfs.semanticscholar.org/8430/c0b9afa478ae660398704b11dca1221ccf22.pdf> [Accessed 24 Jul. 2019].
- Jozefowicz, R., Zaremba, W. and Sutskever, I. (2015). An Empirical Exploration of Recurrent Network Architectures. In: *International Conference on Machine Learning*. [online] Proceedings of Machine Learning Research. Available at: <http://proceedings.mlr.press/v37/jozefowicz15.pdf> [Accessed 25 Mar. 2019].
- Karpathy, A. (2015). *The Unreasonable Effectiveness of Recurrent Neural Networks*. [online] Karpathy.github.io. Available at: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/> [Accessed 24 Jul. 2019].
- Lipton, Z. and Berkowitz, J. (2015). A Critical Review of Recurrent Neural Networks for Sequence Learning. [online] 4, p.1-31. Available at: <https://arxiv.org/pdf/1506.00019.pdf> [Accessed 26 Nov. 2018].
- Lorenz, E. (1963). Deterministic Nonperiodic Flow. *Journal of the Atmospheric Sciences*, [online] 20, p.130-133, 140-141. Available at: <https://journals.ametsoc.org/doi/pdf/10.1175/1520-0469%281963%29020%3C0130%3ADNF%3E2.0.CO%3B2> [Accessed 24 Jul. 2019].
- Martens, J. (2010). Deep learning via Hessian-free Optimization. In: *International Conference on Machine Learning*. [online] Toronto: University of Toronto. Available at: http://www.cs.toronto.edu/~jmartens/docs/Deep_HessianFree.pdf [Accessed 25 Mar. 2019].
- Martens, J. and Sutskever, I. (2011). Learning Recurrent Neural Networks with Hessian-Free Optimization. In: *International Conference on Machine Learning*. [online] International Conference on Machine Learning. Available at: http://www.icml-2011.org/papers/532_icmlpaper.pdf [Accessed 25 Mar. 2019].
- Martens, J. and Sutskever, I. (n.d.). Training Deep and Recurrent Networks with Hessian-Free Optimization. [online] Available at: https://www.cs.utoronto.ca/~jmartens/docs/HF_book_chapter.pdf [Accessed 24 Jul. 2019].

Pascanu, R., Mikolov, T. and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In: *International Conference on Machine Learning*. [online] Atlanta: International Conference on Machine Learning. Available at: <http://proceedings.mlr.press/v28/pascanu13.pdf> [Accessed 25 Mar. 2019].

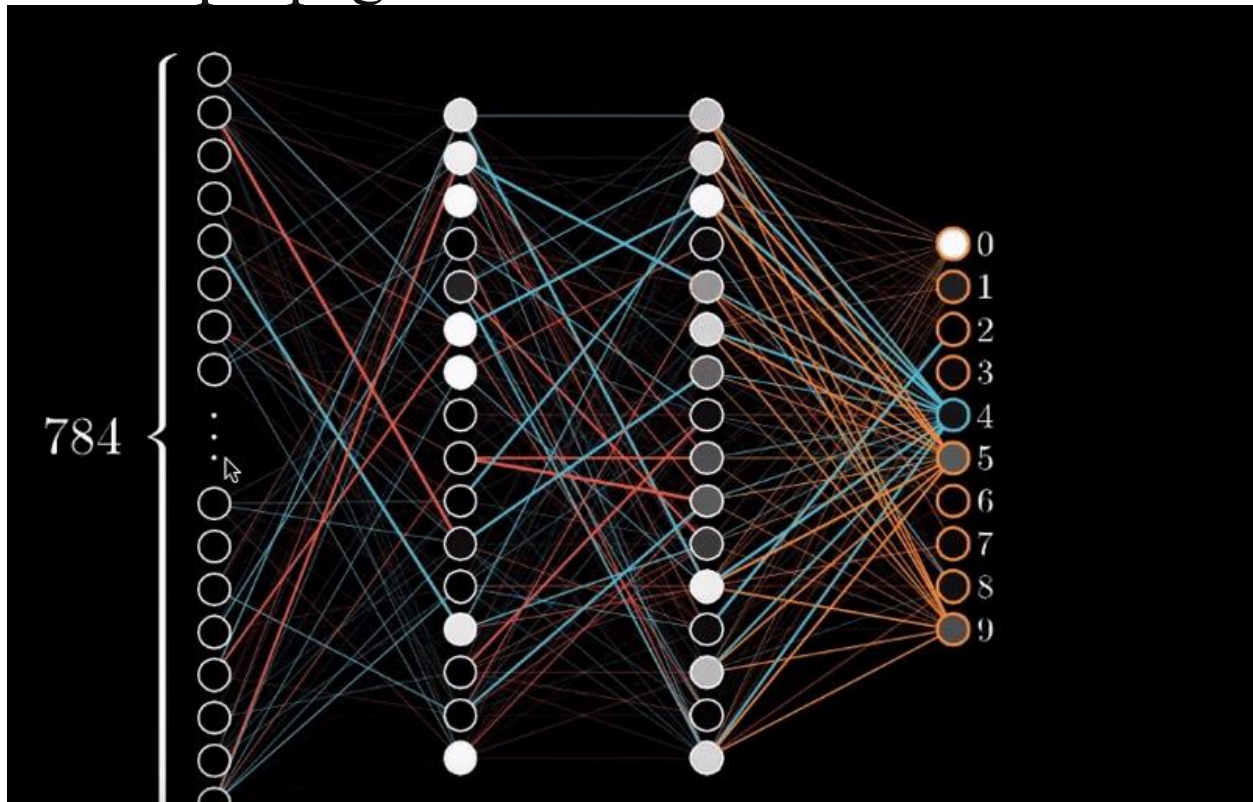
Rival, S. (2018). *Siraj Raval*. [online] YouTube. Available at: <https://www.youtube.com/channel/UCWN3xxRkmTPmbKwht9FuE5A> [Accessed 26 Nov. 2018].

Rumelhart, D., Hinton, G. and Williams, R. (1986). *Learning Internal Representations by Error Propagation*. [online] San Diego: University of California, p.4-10, 28-30. Available at: <http://concepts.psych.wisc.edu/papers/711/RumelhartBackprop.pdf> [Accessed 24 Jul. 2019].

Schmidhuber, J. (2013). *Sepp Hochreiter's Fundamental Deep Learning Problem (1991)*. [online] People.idsia.ch. Available at: <http://people.idsia.ch/~juergen/fundamentaldeeplearningproblem.html> [Accessed 24 Jul. 2019].

Sussillo, D. and Abbott, L. (2015). Random Walk Initialization for Training Very Deep Feedforward Networks. [online] p.1-10. Available at: <https://arxiv.org/abs/1412.6558> [Accessed 24 Jul. 2019].

1 Backpropagation



Alt: 2 Cycles of forward/backward propagation taken from video linked in Additional Sources.

To explain the process of backpropagating let us start with an example to pragmatize the discussed. Let's say we want to classify a flower based on several of its attributes (see [Iris](#) classification for a developed example) for whatever reason, we decide it be best that a computer do this. To begin with we need to collect data describing the flower, including the petal and sepal length and width and other attributes such as color. After many treacherous hours of collecting, data that seems superfluous at first, we need to build model that is able to learn from the data what the characteristics of each flower species are. For this we decide to go with a more abstract “black box,” representation, the NN. The NN requires training as the network would not improve from its kernel initialization without feedback (the kernel initializer for the sake of this explanation assigns arbitrary weight values to each neuron). The training cycle goes as follows:

1. Input the first instance of data into the network, each attribute corresponding to a predetermined neuron.
2. Forward-propagate this information through the network.
3. Record output values.
4. Back Propagate the error through the network.
5. Repeat this process a certain number of times through the entire dataset (approximation due to **non-determinism**).

After forward-propagation we have a set of output values, say 0.5, 0.6, 0.89 (where the range is 0-1) and each of the values corresponds to the certainty of the input information being associating with a particular class, such as Iris setosa, Iris versicolor, or Iris virginica; 0.89: Iris virginica being the highest in our case. The data instance, on the contrary, argues that the combination of these characteristics should result in Iris setosa being chosen. As the network is still training it is now required to back propagate the error. So, what it does for each of the output neurons is first calculate the distance from the actual value (expected – output). The expected in our case is 1 while the first output value was 0.50 so the distance would be $1 - 0.50 = 0.50$.

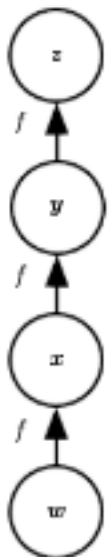
Now that we calculated the error of this forward pass with respect to the output unit, we want to obtain these errors for each neuron in the network such that it can be updated. To do this we apply the chain rule of calculus.

Practical knowledge: These errors are a lot of the time calculated in what we call batches. Batches consist of several instances (training examples is another commonly used name). For large networks, it then becomes especially apparent that this reduces the overall training cost as the algorithm passes through the network less often.

1.1 Intrinsic use of the chain rule

The following explanation is based heavily off of Ian Goodfellow's *Deep learning* p.204

We can express the calculated error discussed in the previous section using calculus, the obtained expression for a neuron z would be dE/dz where E is the error.

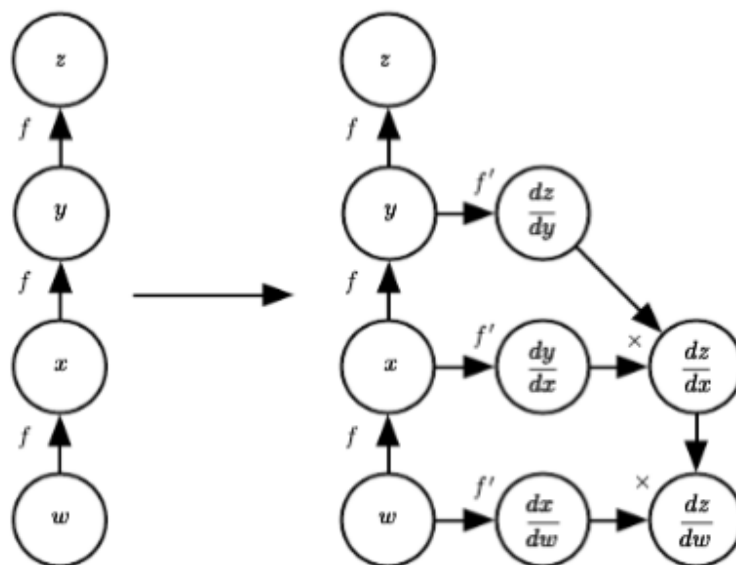


Goodfellow et al., 2016

Above is depicted a simple chain of operations similar to that one may see in a conventional MLP. These operations can simply be expressed as $f(f(f(w)))$ or $wf(wf(wf(aw+b) + b) + b) + b$ where b is the bias and w is the weight for each respective layer. One may see the reminiscence

this has to functions to which the chain rule can be applied. To obtain $dE/dy = dE/dz * dz/dy = dz/dE * f'(y)$. $f'(y)$ could intuitively be seen as adapting the error gradient such that it becomes compatible with y . This process is then repeated to obtain $dE/dw = dE/dz * dz/dy * dy/dx * dx/dw = dE/dz * f'(y) * f'(x) * f'(w)$. As outlined in *Deep learning* this can alternatively be expressed in terms of w $dE/dw = E * f'(f(f(w))) * f'(f(w)) * f'(w)$ (as dE/dz may also simply be referred to as E in this case due to their only being one output unit therefore not requiring distinction between output units in the notation). Rather than making the jump directly to dE/dw the backpropagation algorithm, of course, calculates the error gradient for all intermediary units.

Practical knowledge: Viewed this way this simply becomes recursion and anywhere where recursion is present, we may be able to apply dynamic programming to optimize. Popular deep learning libraries such as TensorFlow construct symbol maps where our original chain would become a chart clearly justifying the dynamical programming approach where we “memoize” the calculation of each derivative such that it does not need to be calculated multiple times. During the course of the backward pass.



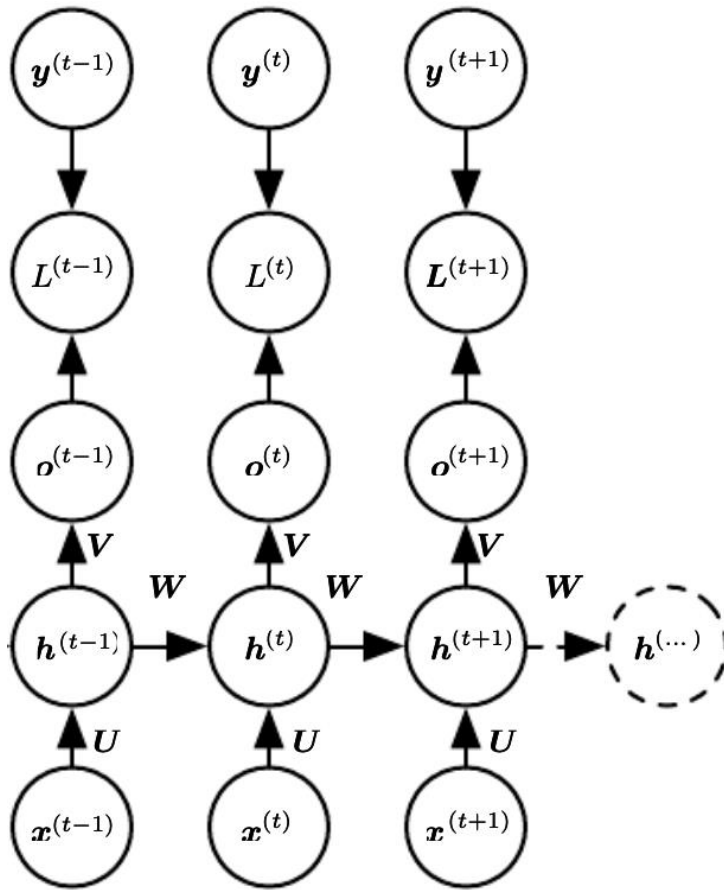
Goodfellow et al., 2016

1.2 Updates

Now that we calculated these error gradients are calculated they are used to update the network, using gradient descent. As mentioned previously we have two major components to each transition the weights (w ; also expressed as W in literature referring to all weights) and biases (b). If we were to update the weight and bias of y w_y and b_y respectively we denote their updated counterparts w_{yu} and b_{yu} . The update of w_y could simply be expressed as $w_{yu} = w_y + l_rate * error * input$ and $b_{yu} = b_y + l_rate * error$. The error being dE/dy , input the value multiplied by the weight and l_rate being a hyperparameter controlling the rate of update. The input partaking in the update process can be seen as a sort of scaling factor that makes larger updates to weights

whose outputs ($\text{weight} \times \text{input}$) are larger. This process is then repeated back up to the first layer, where the transfer derivatives of the neurons in ascending layer are the transfer derivatives of the net error calculated for the connecting descending neurons again the magnitude of the adaptations made by each neuron is proportional to the weights of the connections and the net adaptation made to that neuron is boosted by the Sigmoid function.

Appendix B: derivation of BPTT long term dependency explosion/vanishing formula



Goodfellow et al., 2016

We have a simple network similar to the one depicted above in which $y_t = o_t$

The following equations are defined for understanding purposes.

$$y_t = Sigmoid(V \times h_t + c)$$

$$h_t = ReLu(W \times h_t + U \times x_t + b)$$

Where V , W , and U are matrices that scale values passing through the network, b , and c are biases which are ideally responsible for ensuring that any additional information irrespective of size can be encoded, h_t is the hidden layer at timestep t which is connected to both the output at time t and the next hidden state, this allows for both decoding the information at that timestep and propagation of the same information to the next timestep, x_t is the input at timestep t allowing for additional changes in the system to be inputted such that the network would be able to capture features imperative to the big picture view it is designed to predict, and y_t is the output at timestep t .

The nonlinear components of the network ensure it is a universal function approximator. The nonlinear side in our case consists of a series of activation functions, there are many of such functions in the NN literature, however we will be using the linear and Sigmoid functions. These functions do not play much of a role in this proof as removing them makes no difference in the

conclusion, they play a more important role in the training of such a network which will be discussed later in the essay.

The ReLu function is defined as

$$ReLU = \max(0, x)$$

And the sigmoid is defined as

$$Sigmoid = \frac{1}{1 + e^{-x}}$$

Based on these definitions we now have a network which, as we will prove below is differentiable.

Given that we have an error e also expressed as the error gradient δ_e obtained after a full forward propagation we want to find δ_{h_t} . To do so let us make a deduction about any unit h_t which is as follows: every unit in a network follows the same regime, and in our network every hidden unit output its altered hidden state too two connecting units, an output, and a hidden unit in the next time step, therefore any hidden unit h_t has two scaled outputs one of which is an output and the other of which another hidden unit at timestep $t+1$.

Therefore the output of h_t , $h_{t,o}$ could be defined as

$$h_{t,o} = y_t + h_{t+1}$$

Using the definitions above, applying the derivative then we obtain,

$$\delta_{h_t} = (y_t + h_{t+1}) \frac{dh_t}{de} = Sigmoid'(V \times h_t + c) \times V^T \times \delta_{h_{t+1}} + W^T \times ReLu'(W \times h_t) \times \delta_t$$

Assuming no input into the hidden layer, subverting the possibility for a chaotic system local in time (the error signal can still pose chaotically to itself in time as the energy of the system – the error – is not constant and the differential equations describing the system are nonlinear, see Lorenz 1963 for more detailed criterion).

$$\delta_{h_t} = W^T \times ReLu'(h_{t+1}) \times \delta_{h_{t+1}} + V^T \times Sigmoid'(o_t) \times \delta_t$$

Where f' is the derivative of the ReLu activation function defined as the linear piecewise

function $f' = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$. As δ_{h_t} is dependent on its descendant, δ_{h_t} at a final time step f would be

$$\delta_{h_f} = V^T \times Sigmoid'(o_t) \times \delta_f$$

We notice that to find the gradient with respect to a hidden unit h , at timestep t , requires the use of a recursive relationship, similar to geometric sequences. Consider the following situation, a

simplification of the RNN, in which we are given a chain of infinite length, comprised of hidden units. We will omit dependence to δ_t and obtain the following expression,

$$\delta_{h_t} = W^T \times \delta_{h_{t+1}} \times \text{ReLu}'(h_{t+1})$$

We find an expression that depends on δ_{h_f} and obtain

$$\delta_{h_t} = W^{T^{f-t}} \times \delta_{h_f} \times \prod_t^f \text{ReLu}'(h_{t+1})$$

If we factorize, set $\delta_{h_f} = 1$, and simplify, we obtain

$$\delta_{h_t} = \prod_t^f W^T \text{ReLu}'(h_{t+1})$$

In about 50% of cases in a complex RNN $\delta_{h_t} = 0$ due to ReLu, as empirically found by (Sussillo and Abbott 2015).

Do not mistake the above expression with the update procedure which involves summing over all time intervals in which case it would look more like this – with respect to one source of error

$$\Delta W_f = \sum_{t=f-1}^1 f'(h_t) \left(\prod_{\tau=t}^f W^T f'(h_{\tau+1}) \right) h_{t-1}^T$$

Where the full weight update could be calculated by

$$\Delta W = \sum_{f=2}^n \Delta W_f$$

Where n is the number of timesteps with outputs in the network. This could then be substituted into the equation above to acquire,

$$\Delta W = \sum_{f=2}^n \sum_{t=f-1}^1 f'(h_t) \left(\prod_{\tau=t}^f W^T f'(h_{\tau+1}) \right) h_{t-1}^T$$

Which looks remarkably similar to the one presented in Hochreiter et al., 2001

$$\frac{\partial \delta_v}{\partial \delta_k} = \sum_{l_{t-1}=1}^n \dots \sum_{l_{s+1}=1}^n \left(w_{l_t l_{t-1}} \left(\prod_{\tau=t-1}^{s+1} f_{l_\tau}'(\text{net}_{l_\tau}(\tau)) w_{l_\tau l_{\tau-1}} \right) f_{l_s}'(\text{net}_{l_s}(s)) \right)$$

Which in our notation could be rewritten as,

$$\frac{\partial \Delta W}{\partial \delta_k} = \sum_{t=2}^1 \dots \sum_{t=n-1}^1 f'(h_t) \left(\prod_{\tau=t}^n W^T f'(h_{\tau+1}) \right) h_{t-1}^T$$

Where δ_k is the gradient for an output k irrespective of the timestep, this notation makes the expression applicable to a wider range of larger networks with multiple outputs per timestep.

Note that where we derive the error gradient per timestep, Hochreiter et al., 2001 implicitly make that step and immediately move on to the consideration of multi-output RNN's. Therefore, in our case $\frac{\partial \Delta W}{\partial \delta_k} = \Delta W$.