RNNbow: Visualizing Backpropagation Gradients in Recurrent Neural Networks

Dylan Cashman*
Tufts University

Genevieve Patterson†
Microsoft Research

Abigail Mosca‡
Tufts University

Remco Chang§
Tufts University

Abstract

We present RNNbow, an interactive tool for visualizing the gradient flow during backpropagation training in recurrent neural networks. RNNbow is a web application that displays the relative gradient contributions from Recurrent Neural Network (RNN) cells in a neighborhood of an element of a sequence. By visualizing the gradient, as opposed to activations, it offers insight into *how* the network is learning. We use it to explore the learning of an RNN that is trained to generate code in the *C* programming language. We show how it uncovers insights into the *vanishing gradient* as well as the evolution of training as the RNN works its way through a corpus. We describe some future work in using RNNbow to illustrate the differences between RNN architectures and cell types.

Index Terms: I.2.6 [Computing Methodologies]: Artificial Intelligence—Learning; G.1.6 [Mathematics of Computing]: Numerical Analysis—Optimization; H.1.2 [Information Systems]: Models and Principles—User/Machine Systems

1 Introduction

Artificial Neural Networks (ANNs) have made revolutionary improvements in classification in many domains, with particular attention given to their ability to classify images using convolutional filters [15]. A commonly-cited issue with all ANNs is that they act as a black box, with large numbers of hidden layers each individually learning their own weights resulting in a massive parameter space. Problems of interpretability are compounded by non-linear transformations which obfuscate interactions between each layer. Visualizations of activations within convolutional neural networks have seen some success in illuminating the inner workings of networks to both help understanding and to assist in hyperparameter settings [21, 24, 25, 28]. However, such activation visualizations are specific to the domain of image processing, and primarily offer insight into how a network is functioning after training. In this work, we present RNNbow, a tool for aiding in understanding of and providing insight into the training of Recurrent Neural Networks (RNNs).

A key insight that differentiates this work from other visualizations for deep learning is that it visualizes the *gradients*, not the *activations*. Activations are the responses of the network during inference - when fed an input, what neurons are firing? While this is instructive in comprehending how the network *makes decisions*, it offers little insight into how the network *learns*. Learning is accomplished via gradient descent, a method which minimizes loss over a training set by iteratively updating parameters in the direction dictated by the gradient of that loss. Thus, to analyze how the network is learning (or if it is learning at all), we must inspect the gradients.

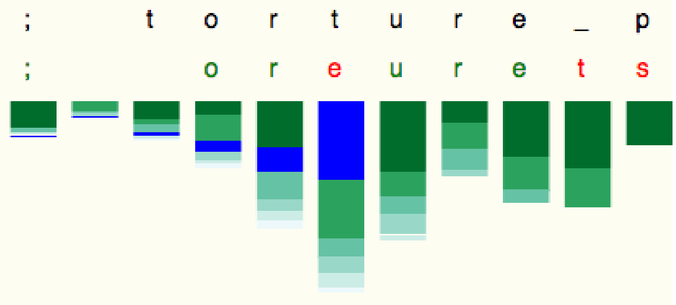
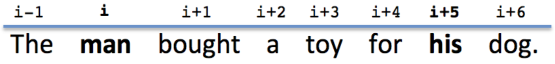


Figure 1: RNNbow helps the user see the flow of gradients due to an individual cell's loss during training of a Recurrent Neural Network. Here, we see highlighted in blue the gradient resulting from loss due to predicting the character "e" when the true character was "t".

RNNs are a particular class of ANNs that map input sequences to output sequences. As with all ANNs, their function depends on what they are fed in as inputs and what they are fed as desired outputs. They can learn to label each item in a sequence if their training data includes labels; a good example of this is training an RNN to do part-of-speech tagging [20]. Alternatively, RNNs can be used to generate sequences that look like the training data. This is accomplished in a technique first proposed by Elman [5] in which, for a given training set $s_1 \dots s_n$ the input sequence is set to $s_1 \dots s_{n-1}$ and the output sequence is set to $s_2 \dots s_n$. In this way, the RNN learns to predict the succeeding element of a sequence. RNNs are behind some of Deep Learning's most astonishing results, including language translation [27], generating image captions [13], and predicting medical outcomes [16].

RNNs have been called both "unreasonably effective" [12] and "difficult to train" [2, 19]. One of the major issues with training RNNs is ensuring that the gradient descent updates propagate far enough back that long-term dependencies can be learned. Consider an RNN that tried to produce the following sentence.



In order for the RNN to be able to know the gender of the pronoun **his**, it must remember the gendered noun **man** 5 time steps earlier. Since RNNs learn via gradient descent, the only way to learn time dependencies of that distance is to have the gradient due to a loss propagate back a far distance. In other words, if the word **his** at $t = i+5$ is a function of the word **man** at $t = i$, then the gradient at $t = i$ with respect to the loss at $t = i+5$ must be greater than 0.

If an RNN is parameterized poorly, it may fall victim to the well-studied *vanishing gradient* problem [2, 8, 19], in which gradient only flows a few cells back, at which point the network may be no more capable than using frequency counts over the training set. To try to address this problem, the user must not only select numerical parameters like the size of the hidden layer or the number of layers, but also must choose between different architectures (stacks or grids)

*e-mail: dylan.cashman@tufts.edu
†e-mail:gen@microsoft.com
‡e-mail: abigail.mosca@tufts.edu
§e-mail:remco@cs.tufts.edu

and different RNN cell types such as Long Short-Term Memory cells (LSTMs) [9] or Gated Recurrent Units (GRUs) [4]. The theoretical distinctions involved in making these choices can be mystifying to many users of RNNs, and it can be confusing to try to diagnose learning issues resulting from poor RNN design. Tools are needed to reveal endemic issues in gradient flow in RNNs so that the user has evidence of whether their network architecture is able to learn or not.

RNNbow is a tool to visualize the gradient flow during training of an RNN. By breaking down the gradient update at each cell by each component's origin, it makes the vanishing gradient apparent. It helps users assess their parameterization of their network during training. It also provides an illustration of the change in gradient behavior as a network trains. In the case of the earlier example, RNNbow can help the user detect if the loss incurred during the update of the word **him** is successfully propagated 5 steps back to the word **man**. At the time of writing, it is the only neural network visualization that visualizes gradient flow that the authors are aware of.

Further, because RNNbow visualizes the gradient and not the input space, the use of the tool is agnostic to the domain of the problem. In contrast to many of the prevalent ANN visualizations that focus on convolutional neural networks that operate on images, RNNbow can be used to visualize the gradient of any RNN. In this paper we use a character-level RNN as a demonstration, but RNNbow can be applied to show learning of other problems, including video frames, words, and other types of sequences.

For a use case, we repeat a well-known RNN experiment [14] to learn and generate statements in the C programming language via a character-level RNN. We present some insights that can be gleaned via RNNbow. We explain how traditional implementations of backpropagation can be modified to collect the itemized gradients visualized by RNNbow, and discuss complexity implications. We discuss the advantages of visualizing gradient over activation, and conclude by considering future work in using RNNbow to compare different architectures.

2 Related Work

Prior to the recent explosion in big-data neural networks, artificial neural networks were generally small enough to allow for overview visualizations of all nodes and edges in their computation graph. Early work in visualizing neural network activity focused on "opening the black box" in this complicated computation graph. A good example can be found in Tzeng and Ma's work to display three-layer networks as node-link diagrams, using the size and color of the nodes and edges to encode activation magnitude and uncertainty [24]. As the number of layers increases, such visualizations did not scale, and visualizations began to focus on either aggregate views of activations on particular inputs, or by viewing inputs that maximize activation of particular nodes [28]. Some visualizations of the popular Convolutional Neural Network take advantage of the visual form of the input space, integrating images into overviews of the node activations [17]. Some visualizations treat neural networks like other similar high-dimensional classifiers, visualizing 2-D projections of their classifications to provide insight into their decision boundaries [10, 21].

Because of their sequential nature, RNNs proffer an opportunity for more concrete temporal visualizations. In an influential blog post [12] and accompanying publication [14], Karpathy et al. used a variety of visualizations that overlayed some representation of node activation over subsets of the input space to show how different hidden nodes are responsible for different decision logic. More recent research has visualized activation over time, in conjunction with other visualizations of the input domain [23, 25].

Most of the tools listed are used after training to attempt to render interpretable the state of a trained network at test time. In contrast, RNNbow is used to visualize how the network has learned. Thus, it could be useful to view gradients during training, to know whether hyperparameters need to evolve or if the experiment needs to be rerun with a different set of hyperparameters. Mid-training visualization is one of the features of *Tensorboard*, a visualization tool built on top of Google's *Tensorflow* [1]. *Tensorboard* allows users to write out values calculated during training to a log, and then generates basic visualizations, such as line graphs and bar charts, of those values throughout training. While it would be possible to log gradients by patching the backpropagation calculation in a *Tensorflow* project, there is minimal support for visualizing those gradients beyond line graphs and bar charts at the time of writing.

Our tool is unique in several ways. First, it visualizes gradient flow as opposed to activations. RNNbow is also used to assess the learning of a network during training, to determine if a change in hyperparameters is needed, as opposed to analyzing an ANN's response during test time. Lastly, in contrast to the cited RNN visualizations, RNNbow is agnostic to type of input sequence (text characters, movie frames, medical records) since it does not use the input domain as a fundamental part of the visualization.

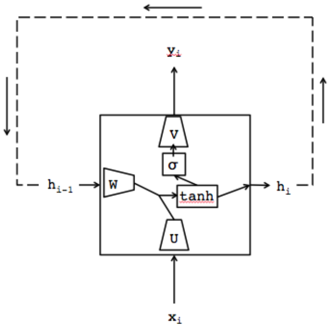


Figure 2: A simple one-cell recurrent neural network, seen as a cyclic computation graph. Trapezoids are linear transformations by a weight matrix. Rectangles are element-wise scalar functions. The RNN produces an output y_i for every input x_i, passing on the calculated hidden state h_i back to itself to use for the next input.

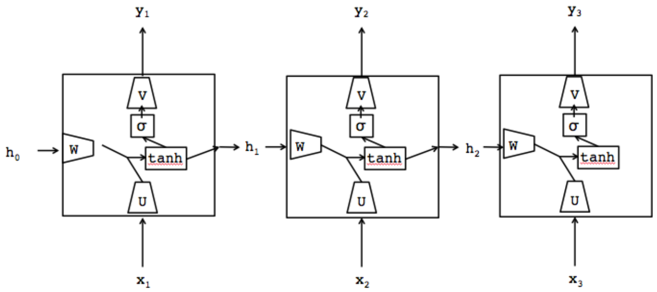


Figure 3: To make calculation over an input sequence well-defined, the RNN from Fig 2 is unrolled once for each element in the input sequence. This RNN is unrolled three times to make three cells. It takes three inputs, and produces three outputs. The weight matrices U, W, and V are shared in each cell.

3 Recurrent Neural Networks

The goal of an RNN is to produce output given sequence input. Their advantage over other sequential learners such as Markov chains or Maximum Entropy Classifiers is that they are able to learn long-term dependencies via nonlinear dynamics in their hidden layer. The basic RNN architecture can be viewed as a graph with cycles, as seen in Figure 2. At any given point in inference, the input x_i and the previous hidden state h_{i-1} are used to calculate the new hidden state h_i, which is then used to calculate the emission y_i. Mathematically, this can be described as:

$$h_i = \tanh(Wh_i + Ux_i) \tag{1}$$

$$y_i = \sigma(Vh_i) \tag{2}$$

Here, W, U, and V are weight matrices, and σ is the sigmoid function $\sigma(x) = \frac{1}{1-e^{-x}}$. Both tanh and σ are common *activation functions* in the deep learning literature. Intuitively, the weight matrices perform a linear transformation on the data, and then the activation functions squash the values back to an interpretable range, with tanh bounded by $(-1,1)$, and σ bounded by $(0,1)$. In addition, these activation functions add a nonlinearity into computation so that the RNN can fit more than polynomial functions.

During training, the training data set is partitioned into regular *batches*. An RNN trains on one batch at a time, in sequential order, by unrolling for a number of steps equal to the size of the batch. A batch size of 3 is seen in Figure 3; however, a typical batch size might range from a dozen elements to around a hundred. Within a batch, the RNN steps through input in order, taking in an input, calculating a hidden state, emitting an output, and then passing on the hidden state to be used for the next item in the sequence. The inputs are any data that can be sequenced (characters, words, frames, etc.), and the outputs can be classifications of or regression on those inputs, or distributions of potential classifications over the output range. The outputs are compared to the true labels, and a loss is calculated. The objective of training is to minimize the loss by choosing the optimal weight matrices W, U, and V. After total loss has been calculated for an entire batch, the gradient of the loss with respect to these weight matrices is calculated and they are then updated via gradient descent. These gradients are typically calculated using *backpropagation*, an efficient algorithm for calculating gradients in computational graphs. The newly updated weight matrices are used for the next batch. The batch size, the size of the hidden layer, and certain constants used in the updating of the weight matrices are all hyperparameters set by the user.

For example, in our use case described in section 5, we build an RNN to generate code for the *C* programming language. Before training, W, U, and V are initialized randomly. If we used our RNN with these randomly-initialized weight matrices to generate text, it would be the same as sampling from a uniform distribution over all characters, and thus it would not look like code. By feeding our RNN input data that looks like valid code, we gradually update our weight matrices so that our RNN generates sequences that better match not only the distribution of characters in the training set but the transitions between characters as well. Examples of code generated by this RNN before and after training can be seen in Figure 4. For more examples of character level RNNs, including much more in-depth analysis of the generation of *C* code, see [12, 14].

```
Batch 0:       0cu     |nv"M$R/m^u tt+^CeU@x>Uh
Batch 10000:      s ged->bat ag_Me_sertaket())
```

Figure 4: Text generated by a character level RNN as used in our use case. The first line was generated before any training, and seems like a random sampling of characters. The second line was generated after training on 250000 characters of the Linux Kernel, and seems to have captured some understanding of the semantic rules of the *C* programming language, such as function calls, underscore separators in function names, and pointer accessing.

Our full training dataset is the Linux kernel, which we split into batches of 25 characters. This also corresponds to unrolling our RNN 25 steps. Referring to the equations defined in (1) and (2), the character input x_i could be encoded densely using a mapping such as ASCII, or it could use one-hot encoding. We use a hidden layer of 100 nodes, meaning that the weight matrix U transforms our input x_i into a 100-dimensional vector representation of the original input character. The hidden state vector is also 100 dimensions. Wh_{i-1} and Ux_i are added together and then squashed through the tanh activation function. That result is then multiplied by the weight matrix V, which projects back into the character-space encoding, providing a multinomial distribution over all possible characters.

To train on a given batch, the RNN starts with the first character as input, calculates a hidden state, and outputs a multinomial distribution for what it thinks the next character could be. In RNNbow, we show the most-likely character from that multinomial distribution, $\max(y_i)$, as shown in area 1 of Figure 5. The true label for the first character in the batch is the second character in the batch, since in this experiment, we want the RNN to predict succeeding characters based on the current character and its context. We use a softmax loss, which generates high loss if our RNN suggests there is a low probability of the true label and a high probability of other labels. That loss is then used to calculate the gradient update for the weight matrices. In RNNbow, we only visualize the gradients of W, since W is what controls the memory of the RNN.

4 RNNbow

RNNbow is a web application that visualizes the gradients used to update parameters during training of a recurrent neural network. In this section, we describe the interface, then we review how the gradient data is harvested during training via backpropagation through time (BPTT) [26].

4.1 Interface

The user interface of RNNbow is a coordinated multiple view implemented in *d3.js* that provides both an overview of the data as well as details of particular elements of the training set. The interface can be seen in Figure 5. It takes in data on gradients recorded during a pass over a training set. The specifics of the data it reads in are described in section 4.2. It visualizes the gradients from a single batch at a time, so that the gradients at individual locations in the training set can be seen clearly. Descriptions of the interface below will make repeated use of the numeric labels from Figure 5.

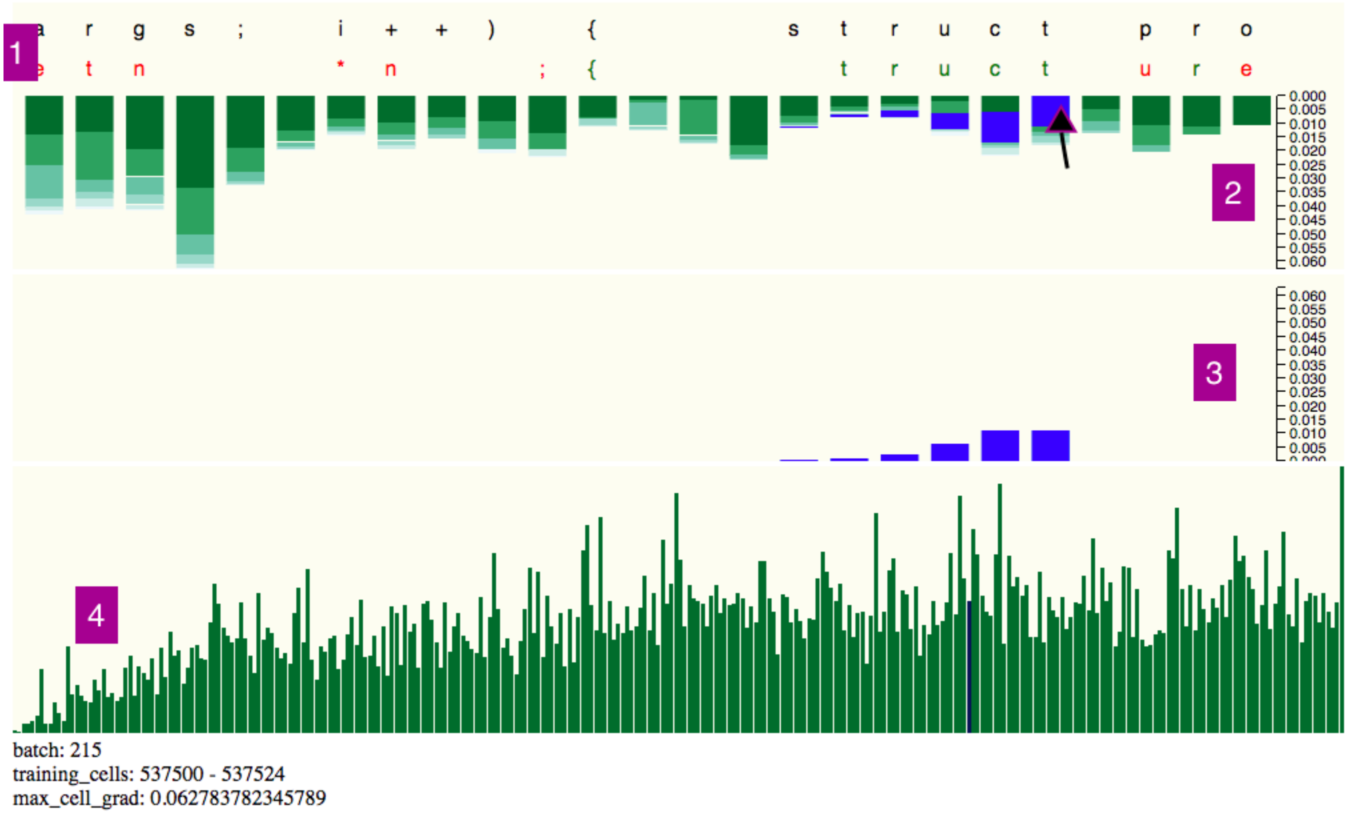
4.1.1 Area 1: Prediction and True Labels

In area 1, in the top row of the visualization, we can see the training set labels per element in the batch. Immediately underneath those labels, we see our RNN's prediction for the label of that element. These predictions are colored according to whether they are correct (green) or incorrect (red). In this figure, the training set batch begins with the five characters "`args; `", and our RNN predicts the five characters "`etn  `". Showing the true and predicted labels helps ground the user in their data.

4.1.2 Area 2: Per Batch Gradients

Underneath each label, in area 2, gradients at each time step of the selected batch of training data are visualized as a stacked bar chart. The height of each bar represents the magnitude of the gradient used to update the weights of the RNN *at that step in time*, relative to the gradients within that specific batch. The bars are partitioned according to how far in the future that portion of the gradient resulted from. The highest, darkest portion of the bar is the gradient due to the loss of the current point in time (due to the loss of the label immediately above the bar). Stepping down in the bar, each new partition is gradient due to the loss accrued due to the succeeding label.

For example, the total gradient is large for the fourth character in the batch shown in area 2 of Figure 5, which we'll call $t = 4$.



batch: 215
training_cells: 537500 - 537524
max_cell_grad: 0.062783782345789

Figure 5: The user interface to RNNbow. In (1), the top row holds the true labels from the training set, and the bottom row holds the prediction from the RNN at training time. The prediction is colored green if it is correct, and red if it is incorrect. (2) shows the magnitude of the gradients being used to update the weights at each point in time. Each bar is decomposed into the different sources of the loss that created that gradient. On mousing over a particular gradient, signified by the purple and black cursor, we see the gradient due to a single loss highlighted in blue, and projected down to (3) for easy inspection. In (4), the user is shown the maximum gradient per batch, and by mousing over different batches, the user can view gradient data from each batch in the training set. The blue bar seen $3/4$ of the way through (4) indicates the currently selected batch, and some information for that batch is seen printed below (4). Different batches can be previewed in (1), (2), and (3) by hovering over their respective bars in (4), and selected by clicking on those bars.

The RNN predicted a white space character, but the true label was the character ‘s‘. The darkest green portion of the bar immediately below the character s represents the gradient due to that particular prediction. However, predictions made in succeeding steps $t = 5, 6, \ldots$ are a function of the hidden state calculated at $t = 4$, there is also gradient in this node due to poor predictions madein the succeeding time steps. Thus, the slightly lighter green portion of the bar beneath the s is the amount of gradient that is the result of the prediction made at $t = 5$, where the RNN predicted a white space character but the true label was the character ‘;‘. Likewise, since the hidden state is passed along as the memory of the RNN to each succeeding character in the sequence, we receive some gradient from all succeeding time steps. In RNNbow, each vertical bar can show the gradient contribution from up to 5 time steps in the future. The number of steps was chosen empirically based on this use case; for other datasets, a larger horizon may be necessary. For more discussion, see section 4.2.2.

4.1.3 Area 3: Gradients Due to Individual Prediction

While seeing the breakdown of the sum of the gradient at each step may be informative, it is also interesting to see how gradient flows backwards from a particular time step - this would show how the network was learning long-term dependencies. By hovering over any portion of a bar in area 2, we highlight all portions in neighboring bars due to the same loss. For the sake of analysis, these portions are projected down into area 3 to highlight the rate at which the gradient decays. For example, in the figure, the purple and black cursor is hovering over the topmost component of the gradient bar below the true label ‘t‘. In area 3, we can see that the gradient due to this decision propagated back 5 time steps, albeit diminishing in magnitude. We call this projected bar chart the *gradient horizon*, as it aims to show when the gradient contribution vanishes as it passes back. As the user sweeps the mouse up and down and across bars, area 3 changes which gradient horizon it displays, allowing the user to quickly navigate the decomposition of gradients.

4.1.4 Area 4: Overview of Max Gradient of All Batches

Area 4 of the interface is a bar chart overview of the maximum gradient within each batch. Our data comprised of 300 batches of data. Each bar in area 4 represents the maximum gradient of any time step within that batch. By showing the maximum gradient in each batch, RNNbow cues the user to navigate to parts of their training set where the most learning is happening. While visualizing the mean would also be valid in that it would show which batches were the most informative, visualizing the max instead shows which individual elements of the training set are the most informative. Since batches are a construct of the training process, it was deemed that maximal batches would be of less interest to the user than maximal elements, and the max was favored.

This area is also interactive: by hovering over different bars within that plot, areas 1, 2, and 3 in the interface change to display information about the corresponding batch. When the user's cursor exits area 4, areas 1,2, and 3 return to displaying the original selected batch. The user can click on a bar in area 4 to select that batch to be displayed when the cursor leaves area 4. The currently selected batch is signified as a blue bar in area 4. Some basic information about the batch being visualized is displayed in text below area 4, including the batch number, which training cells it corresponded to, and the maximum gradient.

4.2 Generating Gradient Data

Given a loss function, backpropagation passes that loss back to any parameters involved in the loss's calculation, via the chain rule. In a Convolutional Neural Network, where one prediction is made, there is generally a single loss calculation, which is then passed along via gradients. In an RNN, there are multiple losses; loss is calculated at each output y_i. Calculating the gradient of W is a difficult task since each hidden state and each output are compounded functions of W.

To account for the multiple losses, RNNs use a special form of an algorithm called backpropagation through time (BPTT) [8]. To use BPTT, RNNs are *unrolled* - that is, each cycle in the computation graph is represented as an additional copy of the RNN, to create a directed acyclic computation graph that backpropagation can then be used on, as seen in Figure 3.

Backpropagation is designed to be as computationally fast as possible, making extensive use of dynamic programming to memoize intermediate calculations so that the gradient can be calculated in a single pass backwards through time. However, fully utilizing dynamic programming will cause us to lose track of some of the intra-sequence effects that RNNbow aims to illuminate. Thus, we remove one level of dynamic programming, trading off increased computational complexity for the ability to record itemized gradients. To motivate this, in the following section we fully derive an expression for the gradient, pointing out what the mapping is between RNNbow and the terms in that expression. We also show how our implementation is equivalent to backpropagation.

4.2.1 Derivation of Itemized Gradients

We are concerned with $\frac{\partial L}{\partial W}$, the rate at which the loss (L) changes with respect to the weights of the hidden layer (W). In training, we use that quantity to update W via gradient descent. Because we are interested in the gradient contributions from each time step, we decompose the loss into loss contributed from the prediction made at each time step. Here, L_t is defined as the loss due to predicting y_t, and n is the size of the batch.

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

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{n} \frac{\partial L_t}{\partial W} \tag{4}$$

For a given time step $t = i$, we have the following decomposition, via the chain rule.

$$\begin{aligned} \frac{\partial L_i}{\partial W} &= \frac{\partial L_i}{\partial y_i} \cdot \frac{\partial y_i}{\partial W} \\ &= \frac{\partial L_i}{\partial y_i} \cdot \frac{\partial y_i}{\partial h_i} \cdot \frac{\partial h_i}{\partial W} \end{aligned} \tag{5}$$

We can further decompose the rightmost term, $\frac{\partial h_i}{\partial W}$, but we must be careful: h_i is a function of W, but it is also a function of h_{i-1} which is in turn a function of W, so we must use the product rule.

$$\frac{\partial h_i}{\partial W} = \tanh'(Ux_i + Wh_{i-1}) \left[h_{i-1} + W \frac{\partial h_{i-1}}{\partial W} \right] \tag{6}$$

The leftmost term is the derivative of $\tanh(x)$, evaluated at $Ux_i + Wh_{i-1}$. Notice that we still must further expand the rightmost term, just like we had to with $\frac{\partial h_i}{\partial W}$. In the following derivations, the term $\tanh'(Ux_i + Wh_{i-1})$ is truncated to $\tanh'_i$ for the sake of readability.

$$\frac{\partial h_i}{\partial W} = \tanh'_i \left[h_{i-1} + W \tanh'_{i-1} \left[h_{i-2} + W \frac{\partial h_{i-2}}{\partial W} \right] \right] \tag{7}$$

We would then have to expand $\frac{\partial h_{i-2}}{\partial W}$, and $\frac{\partial h_{i-3}}{\partial W}$, and on until we end up with the term $\frac{\partial h_1}{\partial W}$ which does not expand since h_0, our initialized hidden state, does not depend on W - it is a hyperparameter set by the user. In this way, the loss due to our prediction at $t = i$ ends up propagating all the way back through the input sequence to $t = 1$.

$$\frac{\partial h_i}{\partial W} = \tanh'_i \left[h_{i-1} + W \tanh'_{i-1} \left[h_{i-2} + \ldots + W \frac{\partial h_1}{\partial W} \right] \right] \tag{8}$$

If we were to expand out all of the products in (8), we would end up with summands that were only dependent on values available in ordered subsets of the sequence.

$$\frac{\partial h_i}{\partial W} = \tanh'_i h_{i-1} + \tanh'_i W \tanh'_{i-1} h_{i-2} + \ldots \tag{9}$$

Note that calculating the first summand only requires knowing h_{i-1} and the additional arguments to $\tanh'_i$, U, W, and x_i. Then, we can memoize the value of $\tanh'_i$, and when calculating the next summand, we only need that memoized value and U, W, h_{i-2}, and x_{i-1}. Let M_j be the $i-j$th summand of (9), so $M_i = \tanh'_i h_{i-1}$, $M_{i-1} = \tanh'_i W \tanh'_{i-1} h_{i-2}$. Note that calculating M_j depends only on the values U, W, x_j, h_{j-1}, h_j, and M_{j+1}.

$$\frac{\partial h_i}{\partial W} = M_i + M_{i-1} + \ldots + M_1 \tag{10}$$

$$M_j = \frac{M_{j+1}}{h_j} W \tanh'_j h_{j-1} \qquad ; \qquad 0 < j < i \tag{11}$$

$$M_i = \tanh'_i h_{i-1} \tag{12}$$

Then we can rewrite (5) in a way that clarifies our implementation of its calculation.

$$\frac{\partial L_i}{\partial W} = \sum_{j=1}^{i} \frac{\partial L_i}{\partial y_i} \cdot \frac{\partial y_i}{\partial h_i} \cdot M_j \tag{13}$$

In order to calculate the gradient for the entire batch, we would need to sum this over each time step, so we substitute (13) into (4).

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{n} \sum_{j=1}^{t} \frac{\partial L_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot M_j \tag{14}$$

In order to use RNNbow, we record the summand of (14) for each value of (t, j). We call these summands the *itemized gradients*, as they are itemized by the time step that was the source of their loss.

To calculate this in $O(n^2), \{M_j\}$ can be implemented as a one-dimensional dynamic programming table that is filled in from right to left. Then each can be calculated in a single backward pass of the batch, from $j = t$ down to $j = 1$. As an example, suppose that we were training a character-level RNN, and had a batch to train on that was the six characters `g u i t a r`, but our RNN instead predicted the six characters `b a n a n a`. To calculate our gradient, we start at the last character, $t = 6$, and see that we predicted a instead of r, and so we incorporate some loss. We record the gradient due to that prediction at $t = 6$, and then pass back that loss via M_6 to $t = 5, \cdots, t = 1$. Once we have calculated all itemized gradients due to predicting a instead of r at $t = 6$, we jump to $t = 5$, calculate the loss due to predicting n instead of a, and pass a different set of M_j back. This is based on an implementation of BPTT from [3].

The full area of the batch view seen in area 2 of Figure 5 represents the full value of $\frac{\partial L}{\partial W}$. The full area of the detailed gradient horizon seen in area 3 of Figure 5 represents the full value of $\frac{\partial L_i}{\partial W}$ described in (13), and each bar within area 3 corresponds to an individual summand. A vanishing gradient would correspond to the summands decreasing as j decreases, which can be seen in Figure 8.

The calculation of (14) takes $O(n^2)$, where n is the size of the batch. Traditional backpropagation only takes $O(n)$, but it doesn't expose the itemized gradients that we need to record for RNNbow. A further exploration of the relationship between our calculation and traditional backpropagation is given in the appendix.

4.2.2 Computational Concerns vs. Estimation

In practice, it may be impractical and unadvised to calculate the itemized gradients throughout all of training. To begin with, this algorithm generates an immense amount of data, storing $O(HNn)$ gradients in a single pass over the training set, where H is the number of nodes in the hidden layer, N is the size of the training set, and n is the batch size. In our use case, we used small batches and a small hidden layer ($n = 25, H = 100$) compared to many networks, and if we had used the entire training set, we would have created data that was 2500 times the size of our training data.

The problem of data size can be ameliorated by only calculating the itemized gradients periodically - in our use case, we only store the gradients every 100 batches, reverting to the optimized version of backpropagation for the other 99% of batches. Lastly, gradients are averaged between all nodes in the hidden layer, as our goal is to see the general rate of training, rather than drilling down into individual hidden nodes. Since this data is used for visualizations, visualizing data from all nodes would lead to occlusion problems, and the general trends of gradient can be viewed in the average.

It may also not be necessary to step all the way back through the batch when calculating itemized gradients. Equation (10) shows that the gradient due to a particular time step's loss is decomposable into a sum of sequence. We can use (11) to analyze the rate of decay of that sequence.

$$\frac{M_j}{M_{j+1}} = \frac{h_{j-1}}{h_j} W \tanh'_j \tag{15}$$

W is generally initialized close to 0, and regularization is used to keep it having small magnitude during gradient descent. $\tanh'$ has a range of $(0, 1]$, and $\frac{h_{j-1}}{h_j}$ should generally be close to one. Thus, the sequence $\{M_j\}$ should decay as j decrements. Then it would stand to reason that we might be able to choose a value k such that we only have to step back k steps to be close enough to the real gradient.

$$S_k = M_i + M_{i-1} + \ldots + M_{i-k} \tag{16}$$

$$S_k \approx \frac{\partial h_i}{\partial W} \tag{17}$$

In our use case, we empirically chose $k = 5$ based on manual inspection of the data. However, it is highly likely that acceptable k should vary with RNN architecture and cell choice. It might be possible to find a k such that $||S_k - \frac{\partial h_i}{\partial W}|| < \varepsilon$ globally across a validation set. Since k is loosely a measure of how far the gradient horizon is, it would stand to reason that a more sophisticated architecture would demand a larger k.

5 Use Case

To demonstrate the use of RNNbow, we trained a character-level RNN on the Linux Kernel to try to get it to generate code that looks like the C programming language, replicating an experiment done in Karpathy et al.'s seminal RNN work [14]. We used batches of 25 characters, and recorded gradients every 100 batches over the first 50000 batches of the training data. We use a hidden layer of 100 nodes, but we average the gradients across all nodes. In this section, we outline several insights that can be found via RNNbow.

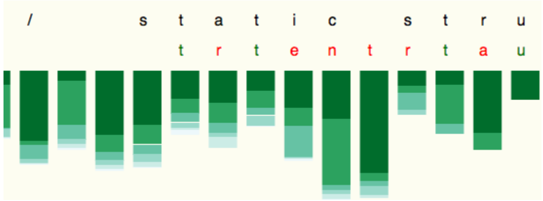


Figure 6: The gradients of a batch early in training. Note that the gradients are mostly composed of darker shades of color. This signifies that the gradient updates are primarily due to local loss, zero, one or two time steps away.

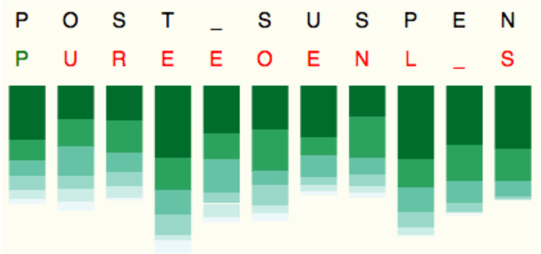


Figure 7: The gradients of a batch later in training. Here, the gradients are much more distributed across different shades, suggesting that longer-term dependencies are being learned.

5.1 Overview of Gradients Over Time

Figure 5 shows the result of training an RNN using our approach. Looking at the overview section, seen as area 4 in Figure 5, the first insight is that the magnitude of the gradient starts very small, and then appears to plateau, albeit with a fair amount of variance. This suggests that early in training, the weights update slowly - there may be some burn-in required before the parameters are updating efficiently. The overview also points the user to batches with maximal gradient. It makes it easy for the user to view the elements of the training set that the RNN learns the most from.

It can also be instructive to compare the batch visualizations (area 1 in Figure 5) as they change from early in training to late in training. At a glance, the darker the batch visualization is, the shorter the gradient horizon is; a larger portion of the update at each step comes from local losses. This tends to be a pattern early in training, as seen in Figure 6. Notice that most of the bars in the visualized batch are primarily composed of dark green bars. Compare that pattern to a batch later in training, seen in Figure 7, where the gradient is much lighter; this corresponds to longer gradient horizons for training in this batch.

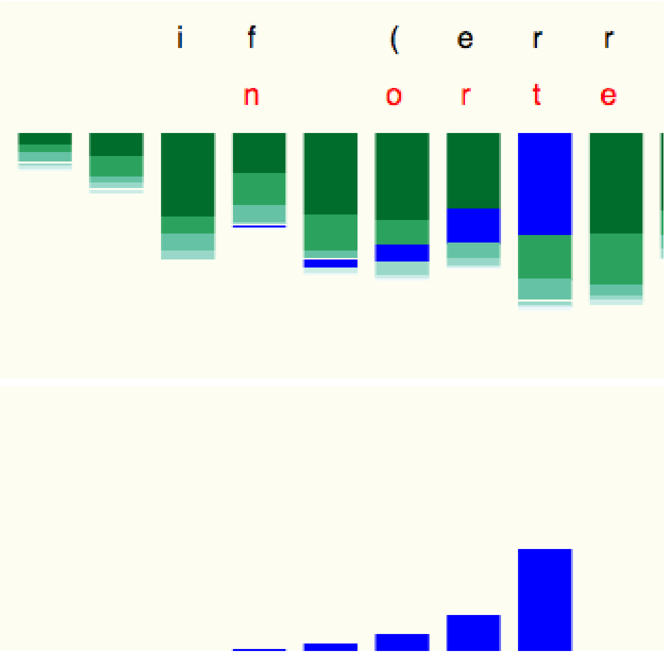


Figure 8: The detailed view of the gradients due to a single character's loss. Note that it makes the vanishing gradient effect very apparent.

5.2 Vanishing Gradient

A well-known consequence of the activation functions used in RNNs, tanh and σ, is that they result in a gradient that decreases as it is passed back in time. For large swaths of the hyperparameter space, the gradient may decay incredibly fast, restricting any long-term dependency learning [2, 8, 19].

The primary function of the projection of gradients in the visualization, area 2 in Figure 5, is to illustrate the rate at which the gradient decays. By mousing over a gradient bar, the user can see the rate at which that particular gradient due to a particular character's loss vanishes. This can be seen in Figure 8.

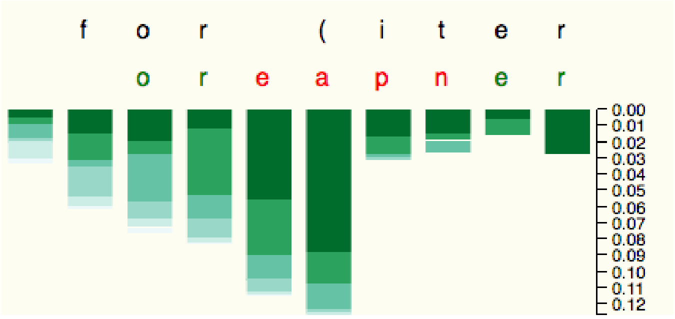


Figure 9: The maximal gradient in the training set. The RNN assumes a large gradient when predicting the character a instead of the character (. This may be due to a penalty of not learning the iterator grammar of the *C* programming language.

5.3 Batches With Maximal Gradient

The overview bar chart in area 4 of the interface shown in Figure 5 can be used to cue the user towards parts of the training set that the RNN learns the most from. For an example, we clicked on the rightmost bar of area 4 to change the focus of RNNbow to that batch, since that bar had the greatest height and thus the greatest maximal gradient. The stacked bar chart of the maximal gradient in that batch can be seen in Figure 9. The maximal gradient is due to predicting the character `a` instead of predicting the character `(`, in spite of the context of being in a `for` loop. Note that it also assumes some gradient from incorrectly predicting the subsequent characters as well. This suggests that our RNN has not learned the iterator grammar of the *C* programming language. It also confirms that our RNN is learning from actual mistakes and not overfitting the training set. As this maximal gradient comes late in our training set and is on a legitimate mistake, it cues the user that we have not trained enough and training must continue.

6 Discussion

One of the key insights of RNNbow is that it visualizes the gradient, as opposed to the activation. Considering that the gradient is what dictates updates to the model, i.e. what is learned, its visualization is revealing of the training process. It is particularly salient in a visualization of RNNs, as the many-to-many relationship between losses and time steps can be difficult to intuit about. It proved useful in discovering endemic properties like vanishing gradients; there may be other endemic qualities in network training that cannot be noticed in visualizations of activations. Visualizing gradient flow may still be helpful for other ANNs, including CNNs.

RNNbow is not intended to be a tool for power users of RNNs; such users would be better served with custom visualizations and custom analytics within their deep learning pipelines. RNNbow is most useful to the non-expert. The current implementation does have some scaling issues, both in the interface as well as in the implementation of backpropagation through time, described in section 4.2. However, it is more likely that a non-expert would use smaller networks that are executable on a personal computer; that is the scale we aim to currently support.

In future work, we hope to design for more sophisticated networks. This interface doesn't support more than a few hundred batches, although this should be solvable with some aggregation and drilldown. In practice, RNNs may train over hundreds of thousands of batches. A heuristic could be used to point the user to particularly interesting batches within the training data. Similarly, the stacked bar chart might not scale to batch sizes of 50 or more. In the use case given in this work, with a batch size of 25 and $k = 5$, the stacked bar chart was responsible for visualizing 125 pieces of data. A sophisticated RNN might have a batch size of 128 and would hopefully have a much larger memory; more iterations of design are needed to come up with an appropriate visualization of so many gradients. In addition, some form of aggregation would need to be defined for the predicted labels (in this case, characters) in large batch sizes. Perhaps the largest hurdle to supporting industry scale networks is the number of layers visualized. RNNbow currently supports a single layer; there are popular CNNs with more than a hundred layers [7], and RNNs are following suit [18]. It's unclear how the stacked bar chart would scale to even a dozen layers. It may be that a higher resolution visualization of gradients between layers is needed.

Prior to the design of the visualization, basic experiments were run to estimate the gradient horizon throughout the first 50000 batches of training, and it was found that it varied from 2 to 5 characters before the gradient dropped below a certain $\varepsilon > 0$. Production-level RNNs, with sophisticated architectures, may need gradient horizons of tens or even hundreds of time steps. The current algorithm used may not be scalable to record the gradient that many steps back.

This could be addressed by taking the gradients much less frequently than every 100 batches.

7 Future Work

The design space of RNNs is rich and constantly evolving, and much of the progress has the goal of having a longer gradient horizon. As visualizing this gradient horizon is the key feature of RNNbow, it should prove invaluable in comparing different approaches. One direction of RNN research focuses on making more sophisticated cells in which the calculation of the hidden state and outputs differs from the vanilla RNN equations, seen in (1) and (2). We would like to adapt RNNbow to visualize the gradient flow of these alternative cells. It should be illustrative in comparing a vanilla RNN to a Long Short Term Memory (LSTM). LSTMs have two hidden layers, one of which is supposed to hold short term memory, and one of which holds long term memory. It would be interesting to see if this is supported in visualizations of the gradients of each hidden layer. Another type of cell called a Gated Recurrent Unit (GRU) only has one hidden layer, but in practice accomplishes long-term dependencies similarly to LSTMs. It isn't particularly clear why this should be. In both LSTMs and GRUs, the calculations of emissions require additional computation with each type of cell having several gates that supposedly lengthen the memory. Perhaps RNNbow would be able to be extended to reveal the gradient flow within a cell, not just between cells.

Beyond different cell types, different arrangements of cells have also proved helpful in practical RNNs. One type of cell architecture is a bidirectional RNN [22], in which two recurrent neural networks are run in parallel for each batch, with one running through the input sequence from left to right and the other in reverse. Their outputs are then combined through a learned linear combination. The bidirectional RNN allows learning former and future context, and viewing the gradient flow in both directions may aid in the understanding of its training. More advanced architectures results from adding layers of RNNs, either to match to multidimensional sequences [6], or to use multiple layers to capture different levels of abstraction in the input sequence as is typical in CNNs [11]. These architectures are difficult to conceptualize; visualizing the gradient may help. However, their spatial complexity proves a challenge in the current layout of RNNbow; some cleverness will be necessary to determine a layout for such architectures.

In this work, we didn't consider the separate nodes in the hidden layer - we just averaged the gradients together. This is in contrast to many of the works on interpreting ANNs, in particular because the activation of a single node tends to be a binary decision maker. Previous work suggests that individual nodes have unique responsibilities; in the same experiment as used in this work, Karpathy et al. [14] found that a single node was responsible for remembering the state of generated *C* code as being in a parenthesis or out of a parenthesis. However, there is no scalable way to visualize all hidden nodes in RNNbow; some heuristic will need to be developed to cue the user to interesting nodes, and some overview of node performance other than the mean will need to be added.

8 Conclusion

We present RNNbow, a web-based visualization tool for analyzing gradient flow during training of a Recurrent Neural Network. We demonstrate how it can be used to find endemic properties in a network, and how it can provide insights into the learning process. We also show that it can be a useful educational tool for illuminating the vanishing gradient phenomenon. We review how to calculate the itemized gradients necessary for loading data into RNNbow. We discuss potential uses of the tool, especially the applications to other RNN architectures.

Acknowledgments

This material is partially based on research sponsored by the Air Force Research Laboratory and DARPA under agreement number FA8750-17-2-0107. This work is also supported in part by the National Science Foundation: IIS-1452977 (CAREER).

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166, Mar. 1994. doi: 10.1109/72.279181

[3] D. Britz. Recurrent neural networks tutorial, part 3 - backpropagation through time and vanishing gradients. `http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/`, 2015. Accessed: 2017-07-15.

[4] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[5] J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[6] A. Graves, S. Fernández, and J. Schmidhuber. Multi-dimensional recurrent neural networks. *CoRR*, abs/0705.2011, 2007.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[8] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 6(2):107–116, Apr. 1998. doi: 10.1142/S0218488598000094

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] M. Kahng, P. Andrews, A. Kalro, and D. H. Chau. Activis: Visual exploration of industry-scale deep neural network models. *CoRR*, abs/1704.01942, 2017.

[11] N. Kalchbrenner, I. Danihelka, and A. Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2016.

[12] A. Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, 2015. Accessed: 2017-07-15.

[13] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(4):664–676, Apr. 2017. doi: 10.1109/TPAMI.2016.2598339

[14] A. Karpathy, J. Johnson, and F. Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds., *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

[16] Z. C. Lipton, D. C. Kale, C. Elkan, and R. Wetzell. Learning to diagnose with lstm recurrent neural networks. *arXiv preprint arXiv:1511.03677*, 2015.

[17] M. Liu, J. Shi, Z. Li, C. Li, J. Zhu, and S. Liu. Towards better analysis of deep convolutional neural networks. *CoRR*, abs/1604.07043, 2016.

[18] R. Pascanu, Ç. Gülçehre, K. Cho, and Y. Bengio. How to construct deep recurrent neural networks. *CoRR*, abs/1312.6026, 2013.

[19] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pp. III–1310–III–1318. JMLR.org, 2013.

[20] J. A. Perez-Ortiz and M. L. Forcada. Part-of-speech tagging with recurrent neural networks. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, vol. 3, pp. 1588–1592. IEEE, 2001.

[21] P. E. Rauber, S. G. Fadel, A. X. Falcao, and A. C. Telea. Visualizing the hidden activity of artificial neural networks. *IEEE transactions on visualization and computer graphics*, 23(1):101–110, 2017.

[22] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[23] H. Strobelt, S. Gehrmann, B. Huber, H. Pfister, and A. M. Rush. Visual analysis of hidden state dynamics in recurrent neural networks. *CoRR*, abs/1606.07461, 2016.

[24] F.-Y. Tzeng and K.-L. Ma. Opening the black box-data driven visualization of neural networks. In *Visualization, 2005. VIS 05. IEEE*, pp. 383–390. IEEE, 2005.

[25] J. van der Westhuizen and J. Lasenby. Visualizing lstm decisions. *arXiv preprint arXiv:1705.08153*, 2017. preprint, `https://arxiv.org/abs/1705.08153`.

[26] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[27] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[28] J. Yosinski, J. Clune, A. M. Nguyen, T. J. Fuchs, and H. Lipson. Understanding neural networks through deep visualization. *CoRR*, abs/1506.06579, 2015.

A Itemized Gradients vs. Backpropagation

The calculation of (14) takes $O(n^2)$, where n is the size of the batch. It is possible to utilize dynamic programming further to speed it up to $O(n)$; this is used in most implementations of backpropagation. First, we expand both sums in (14).

$$\begin{aligned}\frac{\partial L}{\partial W} =& \frac{\partial L_n}{\partial y_n}\cdot\frac{\partial y_n}{\partial h_n}\cdot[M_n + M_{i-1} + \cdots + M_1]\\ &+ \frac{\partial L_{n-1}}{\partial y_{n-1}}\cdot\frac{\partial y_{n-1}}{\partial h_{n-1}}\cdot[M_{n-1} + \cdots + M_1]\\ &\cdots\\ &+ \frac{\partial L_1}{\partial y_1}\cdot\frac{\partial y_1}{\partial h_1}\cdot M_1 \end{aligned}\tag{18}$$

Next, we distribute, group the terms by M_j, and factor.

$$\begin{aligned}\frac{\partial L}{\partial W} =& M_n\left(\frac{\partial L_n}{\partial y_n}\cdot\frac{\partial y_n}{\partial h_n}\right)\\ &+ M_{n-1}\left(\frac{\partial L_n}{\partial y_n}\cdot\frac{\partial y_n}{\partial h_n} + \frac{\partial L_{n-1}}{\partial y_{n-1}}\cdot\frac{\partial y_{n-1}}{\partial h_{n-1}}\right)\\ &+\cdots\\ &+ M_1\left(\frac{\partial L_n}{\partial y_n}\cdot\frac{\partial y_n}{\partial h_n} + \ldots + \frac{\partial L_1}{\partial y_1}\cdot\frac{\partial y_1}{\partial h_1}\right)\end{aligned}\tag{19}$$

Let $N_j = \left(\frac{\partial L_n}{\partial y_n}\cdot\frac{\partial y_n}{\partial h_n} + \ldots + \frac{\partial L_j}{\partial y_j}\cdot\frac{\partial y_j}{\partial h_j}\right)$. Then $\{N_j\}$ can be implemented with a dynamic programming table as with $\{M_j\}$, and we can calculate the gradient in a single pass.

$$\frac{\partial L}{\partial W} = \sum_{i=1}^{n} M_i N_i \tag{20}$$

$$N_j = N_{j+1} + \frac{\partial L_j}{\partial y_j}\cdot\frac{\partial y_j}{\partial h_j} \quad ; \quad 0 < j < i \tag{21}$$

$$N_i = \frac{\partial L_i}{\partial y_i}\cdot\frac{\partial y_i}{\partial h_i} \tag{22}$$

In optimized versions of backpropagation through an RNN, we only make a single pass backwards through time, passing back our accumulation of the gradients N_j, and adding on the gradient of the current time step. For RNNbow, we can't use this method, however, because we lose track of the terms in the expanded product of (19) when we make use of the dynamic programming table for $\{N_j\}$. Thus, we need to use the $O(n^2)$ version described by (14), saving each summand as we accumulate the sum. It is possible that, depending on the implementation library, keeping track of the intermediate M_j and N_j, and then utilizing vector math, as is commonly used in the python library Numpy, could allow us to use traditional backpropagation.