Chapter 1

Analyzing Neural Network Optimization with Gradient Tracing

Contributed by Brian DuSell

1.1 Introduction

Neural networks have led to state-of-the-art results in fields such as natural language processing [3] and computer vision. However, a persistent shortcoming of neural networks is their lack of interpretability. That is, although neural networks often outperform other machine learning techniques when trained to solve a task, it is typically impossible to draw out an intuitive explanation for the solution that the network learns from its complex network of connections. Research has been increasingly focused on developing specialized neural network architectures which include components that impose an inductive bias on the model that is particularly suited to the task at hand, and additionally may lead to more interpretable solutions. For instance, recent neural machine translation models have made heavy use of "attention" mechanisms, whereby a model learns to translate a sentence word-by-word by focusing attention on select source words at each step [1].

Adding specialized components to a network typically has one or more of the following purposes: (a) to increase the network's modeling power, (b) to make the network more feasible to train, and (c) to make some aspect of it inherently interpretable. Stack RNNs, Queue RNNs, and Neural Turing Machines are all examples of adding stack, queue, or tape data structures to a recurrent neural network (RNN) in order to increase its modeling power, allowing it to learn algorithmic solutions that generalize to sequences much longer than those encountered during training [4, 7, 5]. Long short-term memory networks (LSTMs) and gated recurrent units (GRUs) are specifically designed to address trainability issues of RNNs [6, 2]. Finally, the aforementioned attention mechanism produces readily interpretable alignments between words in a source and target sentence.

In this chapter, we explore the effect of neural network components on trainability with a graph-traversal technique dubbed "gradient tracing," whereby we analyze the flow of gradient during backpropagation – the procedure whereby neural networks are trained – through those components. Our goal is to analyze the extent to which certain components influence parameter updates and facilitate learning.

1.2 The Problem as a Graph

We can model the training of a neural network as the propagation of error through a directed acyclic graph (DAG) known as a computation graph. Neural network components exist as subgraphs of this computation graph. By examining the amount of gradient that flows through a particular neural network component, we can measure its influence on the network's parameters during training. We first review gradient descent for neural networks and then relate this to the propagation of gradients through the computation graph.

Neural networks are typically trained by minimizing a loss function on a set of training data using some form of gradient descent algorithm. A neural network is defined by a set of learnable parameters and a function that uses those parameters to map input data to some desired output. We denote the parameters as the vector θ; the parameters often represent the strengths of connections between activation units. The function which makes use of the parameters defines the architecture of the network. The model is "trained" in the sense that its parameters are adjusted so that its predicted outputs better fit the desired outputs, where the notion of "fit" is defined by a loss function that decreases when the predicted and desired outputs match. Gradient descent automatically optimizes the model by repeatedly computing the gradient of the loss function with respect to θ and nudging θ in the opposite direction by a small amount. We can express this as the following update rule:

$$\theta' = \theta - \eta \nabla_\theta L(f(\mathbf{x}; \theta), \hat{\mathbf{y}}) \tag{1.1}$$

where η is a learning rate, f is the neural network (i.e., a function of some input $\mathbf{x}$ parameterized by θ), $\hat{\mathbf{y}}$ is the desired output of the network, and L is the loss function quantifying the dissimilarity between the predicted and desired output. Note that the use of the gradient restricts the functions L and f to functions that are differentiable with respect to θ.

Backpropagation is the name of the specific algorithm for computing the gradient of L with respect to θ, although it can be used for any differentiable function. The task is to compute the gradient of L with respect to θ for a constant $\mathbf{x}$ and $\hat{\mathbf{y}}$. In the case of a multi-layer feed-forward network, the process of computing this gradient can be viewed as the propagation of error from the loss function backward through the layers of the network, updating connection weights in proportion to the amount that they increased the loss function when applied to $\mathbf{x}$ and $\hat{\mathbf{y}}$. The gradient function can be programmed manually or computed using "auto-differentiation," which exploits the chain rule of calculus to automatically compute gradients of complex functions using the gradients of simpler functions. Indeed, several popular neural network libraries, such as PyTorch and DyNet, are based on auto-differentiation.

Given a specific $\mathbf{x}$ and $\hat{\mathbf{y}}$, any neural network, including recurrent models, can be unrolled into a computation graph. A computation graph is a representation of the mathematical operators that define the neural network and loss function as an abstract syntax tree. The leaves of the tree correspond to constants or learned parameters, and interior vertices represent functions on sub-expressions; the root vertex is the top-most operator of L, which we denote ℓ. Edges between vertices correspond to the usage of expressions as function arguments higher up the tree. Since sub-expressions can be shared, the "tree" is actually a graph in the general case. Additionally, since it is a representation of a mathematical expression, it is always acyclic, forming a DAG.

The chain rule of calculus for functions of one real variable x is

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \frac{d}{dx} g(x) \tag{1.2}$$

This rule states that the derivative of a composite function $f(g(x))$ can be computed automatically given that the function f' is known. For more complex cases where g is also a composite function, the rule can be applied recursively. This principle generalizes to functions of multiple variables, where the rule takes the form

$$\frac{\partial}{\partial x_i} f(\mathbf{g}(\mathbf{x})) = \nabla f(\mathbf{g}(\mathbf{x})) \cdot \frac{\partial}{\partial x_i}\mathbf{g}(\mathbf{x}) \tag{1.3}$$

$$= \sum_k f'_k(\mathbf{g}(\mathbf{x})) \frac{\partial}{\partial x_i} g_k(\mathbf{x}) \tag{1.4}$$

where f'_k is the kth element of ∇f. Auto-differentiation applies this rule recursively, "propagating" the gradient from the root vertex for the loss function back to the vertices for the parameters of the network in topological order. Backpropagation computes the values $\nabla_f L$ for each function f in the computation graph. Backpropagation is complete when it reaches all of the (leaf) vertices for the parameters θ, at which point it has computed the needed values of $\nabla_\theta L$.

We can express backpropagation as an operation on a computation graph $G = (V, E)$. Let each vertex $u \in V$ be a mathematical operator with derivative u'; the subtree rooted at the vertex represents a function U of the parameters θ. Let each edge $(u, v) \in E$ indicate that the output of operator v is used as an input to operator u. Suppose that the output of every operator u has already been computed during a "forward" pass through the network and is available as c_u. For each parameter θ_i, let the weight $w_i(u, v)$ of edge (u, v) be defined as

$$w_i(u, v) = u'_i(c_v) \tag{1.5}$$

Let g_v be the gradient of L with respect to the function U whose subtree is rooted at vertex/operator v. The backpropagation graph kernel consists of computing

$$g_v = \nabla_U L = \sum_{(u,v)\in E} w(u, v) g_u \tag{1.6}$$

for all $v \in V$ as necessary to compute g_{θ_i} for each θ_i. In the base case, $g_\ell = 1$.

Finally, the goal of gradient tracing is to identify which paths through G, and thereby which architectural components in the network, contribute most to the parameter update $\nabla_\theta L$. For a given θ_i, this is done by starting at the vertex for θ_i and greedily traversing edges with the weight with the greatest absolute value until reaching ℓ. Let us denote the most influential parent operator on operator v as t_v. This simply corresponds to the term in Equation 1.6 with the greatest absolute value. The gradient tracing graph kernel computes

$$t_v = \operatorname*{argmax}_{u|(u,v)\in E} |w(u, v) g_u| \tag{1.7}$$

for all $v \in V$ necessary to connect a path from θ_i to ℓ.

We deem a component C, a subgraph of G, to have more influence on θ_i when the gradient tracing path passes through C for more $(\mathbf{x}, \hat{\mathbf{y}})$ pairs.

1.3 Some Realistic Data Sets

We can extract a computation graph from any neural network architecture being trained on any data set. Indeed, the results of gradient tracing may differ depending on the data set and task at hand, even for the same network architecture.

Suppose that we are training an RNN language model on the Penn Treebank corpus, which contains some one million words. A model of modest size might have a vocabulary size of 10,000 words and 1,000 hidden units. This would result in roughly 2.1 million parameters and $30 \times n$ million edges in the computation graph, where n is sequence length.

Neural network computation graphs typically consist of a series of fully-connected layers of neurons, resulting in sparse graphs with very dense sub-regions. In practice, the length of the gradient tracing path is likely to be far shorter than the number of vertices and edges in the computation graph. Indeed, in a simple multi-layer feed-forward network, its length will be proportional to the number of layers.

To evaluate the effectiveness of gradient tracing in identifying important components of a neural network, we can generate networks which connect multiple components together in competition, then ablate the most- or least-influential components and observe whether the number of samples that the model requires to solve the task increases or decreases accordingly. For instance, in the context of language modeling, we might run an RNN, LSTM, and GRU on the same input sequence $\mathbf{x}$ in parallel and average their outputs together. We ablate whichever component has the least influence during training, then the second least, and so on, verifying that the dip in performance of the model on the given task is proportional to the importance of the removed component. For more comprehensive testing, we can randomly sample artificial neural network architectures by plugging components together using random graph generators.

1.4 GT-A Key Graph Kernel

The gradient tracing graph kernel is given in Algorithm 1. The procedure presumes that back-propagation has already been run on the computation graph G, and that $w(u, v)$ and g_u have been computed for all $u, v \in V$.

Note that this algorithm simply traverses the DAG from one of its leaf nodes to its root; its time complexity is $O(|V| + |E|)$. For a model with k parameters, the procedure will need to be repeated k times, resulting in $O(k(|V| + |E|))$.

Algorithm 1 Gradient tracing

```
1: procedure GradientTracing($G, \theta_i$)   ▷ $G$ is a computation graph with root vertex $\ell$, $\theta_i$ is a parameter
2:     $p \leftarrow$ an empty path
3:     $v \leftarrow \theta_i$
4:     while $v \neq \ell$ do
5:         $v \leftarrow \operatorname{argmax}_{u|(u,v)\in E} |w(u, v) g_u|$
6:         append $v$ to $p$
7:     return p
```

Bibliography

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[2] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[3] Yoav Goldberg. A primer on neural network models for natural language processing. *CoRR*, abs/1510.00726, 2015.

[4] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[5] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. *CoRR*, abs/1506.02516, 2015.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[7] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *CoRR*, abs/1503.01007, 2015.