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, their lack of interpretability remains a persistent and serious shortcoming. Even though neural networks often outperform other machine learning techniques, it is typically impossible to come up with an intuitive explanation for the solution that the network learns from its inscrutable 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. A recurrent neural network (RNN) is an extension to neural networks that can operate on sequences of variable length. 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." The standard technique for training neural networks, called backpropagation, involves the flow of "erro," or gradient, backward through the components of the network. Gradient tracing is a technique of analyzing the flow of gradient through those components; its 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. This process is known as "backpropagation." Neural network components form subgraphs of the computation graph. By examining the amount of gradient that flows through a particular neural network component, we can measure the component's 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. Then, we express backpropagation as a graph kernel and introduce gradient tracing as a closely related variant of backpropagation.

1.2.1 Gradient Descent

A neural network consists of a collection of real numbers called "parameters" (which we denote θ) and a function (say f) that uses those parameters to map a vector of inputs (say $\mathbf{x}$) to a vector of outputs (say $\mathbf{y}$). The parameters often, but not always, represent the strengths of "connections" between "activation units," which are analogous to synapses and neurons in a biological brain. The goal of neural network optimization, or "training," is to adjust the parameters so that they better approximate a desired function. This is particularly useful in scenarios where the desired function is poorly understood or extremely complex. Although the definition of the function f never changes during training, different parameter values cause the network to model different functions of $\mathbf{x}$.

Gradient descent is a standard technique for training neural networks. For each pair of vectors $(\mathbf{x}, \hat{\mathbf{y}})$ in a collection of training data, the algorithm attempts to minimize a scalar-valued loss function, say L, which quantifies the dissimilarity between the predicted output $\mathbf{y} = f(\mathbf{x})$ and the desired output $\hat{\mathbf{y}}$. The loss function 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 update rule

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

where η is a learning rate. Note that the use of the gradient restricts the functions L and f to functions that are differentiable with respect to θ. This update rule is applied for multiple iterations until the loss stops decreasing, performance on a held-out set of data stops improving, or some other stopping criterion. Note that gradient descent only guarantees finding a local minimum, not a global one.

For simplicity, we have described gradient descent on a single training pair. However, true gradient descent uses a loss function that quantifies error on the entire training set at once (which may simply be the sum of the losses for each training pair). In this case, the parameters are not changed until the entire training set has been seen (after one "epoch"), before starting another iteration. In practice, the training set is often partitioned into smaller "mini-batches," and the parameters are updated after each "mini-batch," or several times per epoch.

1.2.2 Automatic Differentiation and Backpropagation

Automatic differentiation is an efficient and general algorithm for computing gradients of complicated functions. The algorithm exploits the chain rule of calculus to compute the gradient of a function using a fixed library of simpler functions whose derivatives have already been programmed by hand. This algorithm is typically called "backpropagation" in the specific context of computing

the gradient of L with respect to θ in neural networks during gradient descent, but it can be used for any differentiable function. The backpropagation routines of several popular neural network libraries, such as PyTorch and DyNet, are based on automatic differentiation.

In gradient descent, 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 $\hat{\mathbf{y}}$. Given a specific $\mathbf{x}$ and $\hat{\mathbf{y}}$, any neural network, including feed-forward and 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. The ordering of incoming edges (function arguments) is significant, whereas outgoing edges are unordered. 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.

1.2.3 The Chain Rule

We now review the definition of the chain rule. 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 and functions of vectors, 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. Auto-differentiation avoids redundant gradient computations by saving previously computed gradients at each vertex rather than re-computing them for every input, allowing the algorithm to run in linear time with respect to graph size. In this sense, auto-differentiation is a dynamic programming algorithm. 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 $\nabla_\theta L$.

1.2.4 Graph Model of Backpropagation

We can express auto-differentiation/backpropagation as a graph kernel on a computation graph $G = (V, E)$. Let each vertex $u \in V$ be a variable or operator in the mathematical definition of the neural network model. The vertex u corresponds to the output of a function $u(x_1, \ldots, x_n)$. Let

u'_k be the function corresponding to the derivative of u with respect to x_k. The subtree rooted at vertex u represents a function U of the parameters θ. Let each edge $(u, v, k) \in E$ indicate that the output of operator v is used as the kth argument to operator u. Note that k is necessary in order to record the ordering of incoming edges. If the operator u has multiple arguments, then the gradient with respect to those arguments may differ depending on the argument, e.g. $u(x, y) = x/y$.

Suppose that the output of every operator u has already been computed during a "forward" pass through the network and is available as the value c_u. Let the weight $w(u, v, k)$ of edge (u, v, k) be defined as

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

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

$$g_v = \frac{\partial L}{\partial U} = \begin{cases} \sum_{(u,v,k)\in E} w(u, v, k) g_u & v \neq \ell \\ 1 & v = \ell \end{cases} \tag{1.6}$$

for all $v \in V$ as necessary to compute g_{θ_i} for each θ_i, where ℓ is the root of computation graph.

Figure 1.1 shows an example computation graph extracted from a neural network implemented in PyTorch. Intuitively, backpropagation traverses the computation graph in topological order, starting at the root ℓ, and computes g_v upon visiting each vertex v. The only constraint is that the incoming vertices of v must have been traversed before reaching v. Note that this process can be used to compute the gradients for all θ_i simultaneously.

Backpropagation may alternatively be viewed as computing the sum of the weights of all paths leading into each parameter θ_i, where the weight of a path is defined as the product of all of its edges. Backpropagation can be turned into a different algorithm by changing the meaning of *sum* and *product*, thereby changing the *semiring* that it uses. We will see that gradient tracing is actually the backpropagation algorithm but with a different semiring.

1.2.5 Graph Model of Gradient Tracing

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 $\nabla_\theta L$. This is by done by identifying the path in the computation graph that transmits the most gradient to each parameter θ_i, and then identifying the components which the path intersects. 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

$$t_v = \begin{cases} \underset{u|(u,v)\in E}{\operatorname{argmax}} |w(u, v) g_u| & v \neq \ell \\ \text{nil} & v = \ell \end{cases} \tag{1.7}$$

for all $v \in V$ necessary to connect a path from θ_i to ℓ.[1] 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.

Note that gradient tracing is the backpropagation algorithm with a different *sum* operator. In place of computing the sum of incoming edge weights, it computes the edge with the maximum absolute value. Thus gradient tracing is a variant of backpropagation with a different semiring.

[1] The function $\underset{u|(u,v,k)\in E}{\operatorname{argmax}} |w(u, v, k) g_u|$ returns the vertex u of the edge (u, v, k) that maximizes the expression $|w(u, v, k) g_u|$.

1.2.6 Implementation Considerations

It should be noted that, in order to boost performance, serious implementations of backpropagation do not operate on scalar values, but instead on "tensors": multi-dimensional arrays of real values that are stored contiguously in memory. Processors and especially graphical processing units (GPUs) can take advantage of locality and parallelism to accelerate computation significantly. Consequently, the inputs and outputs of operators in the computation graph are usually tensors rather than scalars, and both forward values and gradients are computed using tensor-level operations. The same scalar-level mathematics described above still apply; the only difference is that the primitive operators in the computation graph are always tensor-level operations, which may encapsulate multiple edges and vertices in the equivalent scalar computation graph, a prime example being matrix multiplication.

Because individual input and output tensors can still be too small to make full utilization of the processor, tensor operations are also designed to work on "batches" of tensors, whereby multiple input or output tensors are packed into a single tensor along a new dimension. Processors can operate on the packed tensor faster than they could process the individual tensors in serial.

Finally, in real backpropagation implementations such as PyTorch and DyNet, the gradient is computed as follows. Each operator u defines a function h of the form

$$h_u(g_u, c_{v_1}, \ldots, c_{v_n}) = (g_{v_1}, \ldots, g_{v_n}) \tag{1.8}$$

The function h computes the outgoing gradient g_{v_k} for every argument v_k, given the value c_k of each argument. In contrast to Equation 1.6, h computes derivatives and multiplies them with the incoming gradient in the same step.

Even when dealing with tensor-level operations, the root loss operator ℓ must have a scalar output for the gradient to be well-defined.

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-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 most influence during training, then the second most, 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.

In section 1.5 we describe an experiment which uses stochastically-generated feed-forward neural networks.

1.4 GT-A Key Graph Kernel

Algorithm 1 defines the gradient tracing algorithm for computation graphs which consist of tensor-level operations. It simultaneously computes the gradients and most influential paths for each vertex v. It assumes that the forward values c_v have already been computed. The computation of derivatives is also changed to match the form of Equation 1.8.

Algorithm 1 Gradient Tracing

1: **procedure** TraceGradients(G) ▷ $G = (V, E)$ is a computation graph of tensor operations with root vertex ℓ
2:     **for** each vertex v **do**
3:         v.incoming_gradients $\leftarrow$ an empty list
4:     Topologically sort V so that every vertex v comes after its ancestors
5:     **for** each vertex v in topological order **do**
6:         **if** $v = \ell$ **then**
7:             $v.g_v \leftarrow 1$
8:             $v.t_v \leftarrow$ nil
9:         **else**
10:             $v.g_v \leftarrow \sum_i v.\text{incoming_gradients}[i]$
11:             $v.t_v \leftarrow \operatorname*{argmax}_i |v.\text{incoming_gradients}[i]|$
12:         **if** v is not a leaf node **then**
13:             **for** each edge (v, u, i) **do**
14:                 $x_i \leftarrow u.c_u$
15:             $(x'_1, \ldots, x'_n) \leftarrow h_v(g_v, x_1, \ldots, x_n)$
16:             **for** each edge (v, u, i) **do**
17:                 Append x'_i to v'.incoming_gradients
18:             v.internal_trace $\leftarrow$ an empty list
19:             **for** each edge (v, u, i) **do**
20:                 v.internal_trace[i] $\leftarrow$ vector of pointers to the output elements of v that contribute the most gradient to each output element of u at position

In this pseudocode, $\sum$ is an *element-wise* sum that operates over tensors, and argmax is likewise an *element-wise* argmax that operates over tensors. The value t_v will be set to a vector of indices, where each t_{vi} identifies the incoming edge which contributes the most gradient to the ith element of the output of v. The algorithm also computes "internal" pointers which indicate how the gradients flowing into the output elements of v contribute to the gradients flowing out of its input elements. Like the gradient function h_v, this must be implemented differently for each operator type. The most influential path for a parameter θ_i can be recovered by following the t_v and internal pointers up to the root ℓ.

Since $G = (V, E)$ represents a computation graph of tensor operations, $|V| + |E|$ does not correspond directly to the size of the graph. Let $G' = (V', E')$ represent the equivalent, conceptual *scalar* computation graphc corresponding to G. Algorithm 1 simply visits every edge and vertex of the conceptual scalar graph a fixed number of times, so its time complexity is $O(|V'| + |E'|)$.

1.5 A Sequential Algorithm

Although the pseudocode in Algorithm 1 does not parallelize the processing of parallel graph structures, it can already take advantage of efficient tensor-level operations to compute the gradient and gradient trace of each vertex. In this form, the algorithm can capitalize on the large number of efficient neural-network-oriented tensor operations that are implemented for CPUs and GPUs in major neural network libraries. A key aspect of this algorithm is that it relies on being able to store metadata in the form of g_v (a vector of real numbers), t_v (a vector of indices), v.internal_trace (a vector of indices), and v.incoming_gradients (a list of vectors) directly to each vertex v. Each vertex v should store an adjacency list of outgoing edges to make the lookup of all edges (v, u, i) efficient.

1.6 A Reference Sequential Implementation

We re-implemented the backpropagation algorithm in the Python programming language using the PyTorch library.[2] Figure 1.1 shows an illustration of a computation graph extracted from a two-layer feed-forward neural network. In order to handle computation graphs of several hundred nodes in depth, it is important to implement all graph traversals and topological sorts iteratively, without using recursion, in order to avoid hitting Python's built-in recursion limits.

The topological sort is implemented using Kahn's algorithm, whereby a counter of incoming edges is attached to each vertex. Starting from the root vertex, every time an edge is traversed, the counter of the adjacent edge is decremented. Whenever a vertex's edge reaches 0, it is emitted as the next element in the topological sort. This ensures that a vertex's ancestor vertices are always visited beforehand. Topological sorting takes time linear with respect to the number of edges and vertices. In principle, it should be possible to avoid a topological sort altogether, since the correct ordering simply corresponds to the order in which the computation graph expressions were created, in reverse.

1.7 Sequential Scaling Results

We now show scaling results for the backpropagation reimplementation on computation graphs of varying size. All experiments were run on a commodity Intel Core i7 processor without GPU acceleration. For reference, all experiments include running times for the heavily-optimized implementation of backpropagation built into PyTorch.

The first experiment entails running backpropagation on simple feed-forward neural networks with varying numbers of layers. In all cases, the number of hidden units per layer was fixed to 20, the input size was fixed to 10, and the output size was fixed to 15. Each layer used a tanh activation function. The input and target output vectors were randomly initialized (their values do not affect running time). Note that the number of layers is directly proportional to the number of scalar edges and vertices in the computation graph. Figure 1.2 shows that both the reimplementation and PyTorch implementation of backpropagation appear to scale linearly with respect to the number of layers.

[2] Because the gradient tracing algorithm proper requires digging into the internals of the PyTorch library, it is still forthcoming. The cause of this is the "internal" edges that are encapsulated by the tensor-level operations, and the fact that the PyTorch API is very restrictive when it comes to accessing the computation graph. I am communicating with the PyTorch community to figure out how to get around this. Performance-wise, backpropagation serves as a decent proxy for gradient tracing for now since it (a) is a prerequisite for computing the gradient trace, and (b) is the same algorithm with a different semiring.

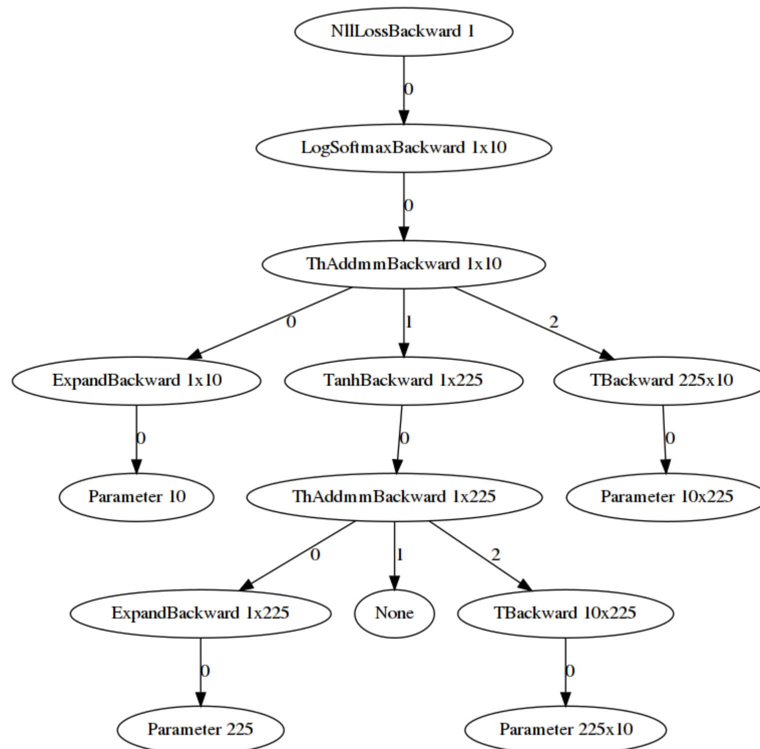

Figure 1.1: Example computation graph for a two-layer feed-forward neural network represented in PyTorch. The topmost vertex is the root of the computation graph and corresponds to the loss function. Each vertex is labeled with the dimensions of its output tensor. Leaf nodes correspond to tensors of parameters. Edge labels indicate the ordering of arguments. Each "ThAddmmBackward" vertex represents an affine transformation of the form $\mathbf{W}\mathbf{x} + \mathbf{b}$. The vertex with a label of "None" corresponds to an input that is not a function of any parameters and does not require its gradient to be computed.

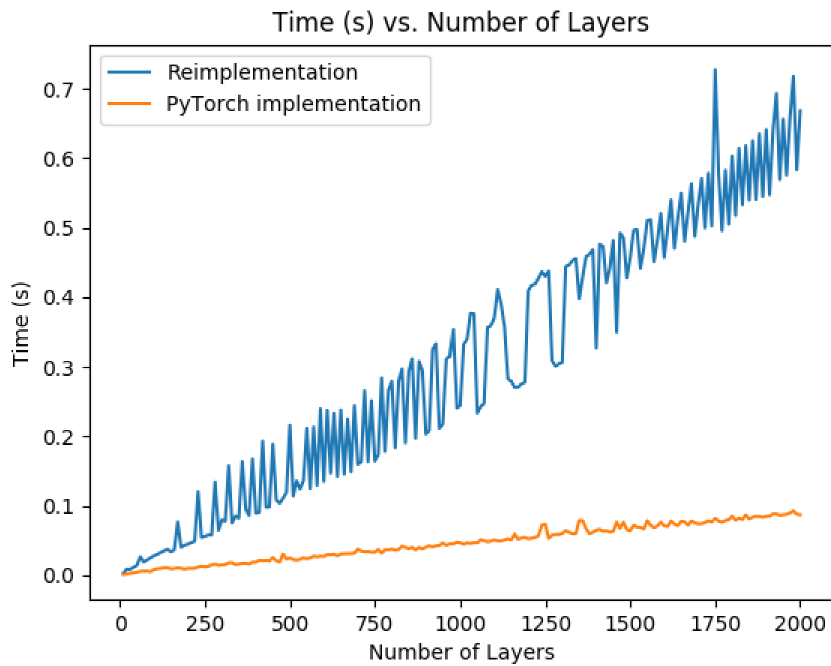

Figure 1.2: Running time of the backpropagation reimplementation as a function of number of layers. The running time of the built-in PyTorch backpropagation implementation is included for reference. Both algorithms exhibit linear time complexity. Note that even when running times are averaged over mulltiple runs, the spikes do not smooth out. Their cause is unclear.

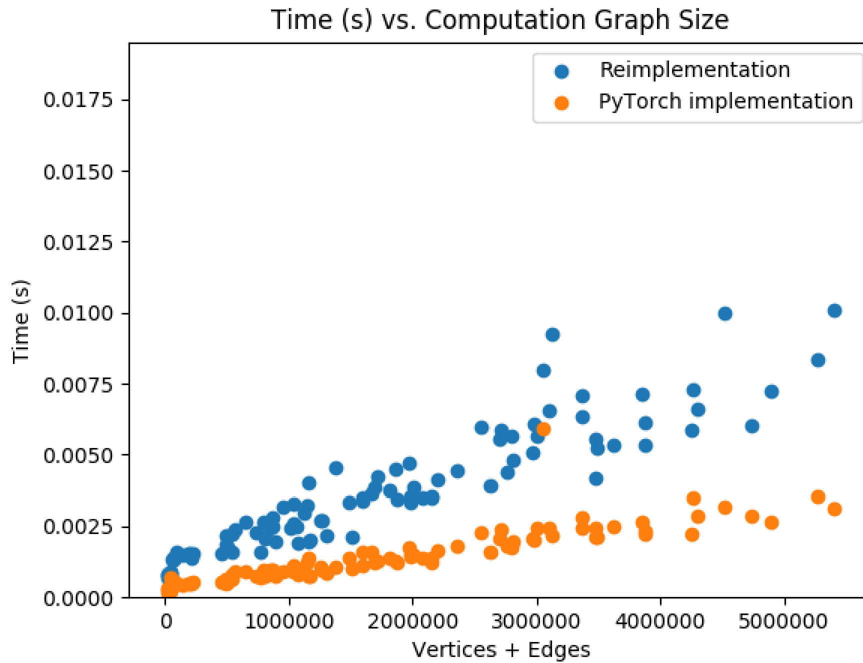

Figure 1.3: Running time of the backpropagation reimplementation as a function of graph size (vertices+edges). The running time of the built-in PyTorch backpropagation implementation is included for reference. Both algorithms exhibit linear time complexity.

We also ran both implementations on stochastically generated feed-forward networks containing branches. The architecture of each network was determined according to the following recursive procedure. The procedure starts with a single active "layer" which may be expanded into one of two patterns. With probability 0.95, the layer is expanded into a diamond configuration of sub-layers. In the diamond configuration, the input of the layer is re-used by two separate, intermediate layers. The outputs of the two intermediate layers are concatenated and used as the output of the layer. The same expansion procedure is run recursively on the two intermediate layers, except that the expansion probability is divided by two in order to prevent an explosion in graph size. With probability 0.05, the layer is expanded into a one-layer network. Every layer uses a tanh activation function. The number of hidden units in each layer is chosen uniformly from $[10, 500]$. The sizes of the output and input to the overall network are both 10. The inputs and outputs are randomly initialized.

Figure 1.3 show the running time of both backpropagation implementations as a function of the number of vertices and edges in the conceptual scalar computation graph These values are estimated based on the dimensions and types of the tensor operator vertices; for example, a matrix-vector multiplication with a matrix of size $m \times n$ has $O(mn)$ scalar vertices and edges. Again, both algorithms scale linearly with respect to the size of the scalar graph.

It is expected that the gradient tracing algorithm will have the same scaling behavior.

Response to Reviews

Section 1.2 was organized into multiple sections describing different points of background material. Attempted to explain the overall usefulness of gradient tracing better. Explicitly defined the meaning of argmax. Added a graphic with an example of a computation graph. Described stoachastic graph generating procedure in detail. Defined RNN in introduction. Tried to better explain the role of the neural network function and its parameters in section 1.2. Added a reference to the computation graph figure earlier in the chapter.

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.