Binomial Checkpointing in Autograd

Kyle Sargent
Department of Computer Science, Harvard University

Reverse-mode Automatic Differentiation

In a sequential setting, the forward mode of automatic differentiation works by recursive computation of

$$\dot{v}_i = \frac{\partial v_i}{\partial v_{i-1}} \dot{v}_{i-1}$$

For a given x_j, and beginning at $i = 1$, so that we have

$$\dot{v}_i = \frac{\partial v_i}{\partial x_j}$$

Binomial Checkpointing

The reverse mode of automatic differentiation requires saving intermediate steps so that the chain rule can be applied. In problems with large computational graphs, such as backpropagation-through-time, the saving of intermediate steps can become prohibitive from a memory perspective. Griewank [1] an optimal algorithm for checkpointing, whereby some intermediate steps of the computational graph are saved and the others are recomputed on-the-fly during reverse-mode AD to save memory. The saved steps are referred to as checkpoints, and Griewank developed a provably optimal scheme for their placement in sequential differentiable programs. Recently, the optimal checkpointing algorithm was independently rediscovered by Gruslys et al. [2] et. al, using a dynamic programming approach.

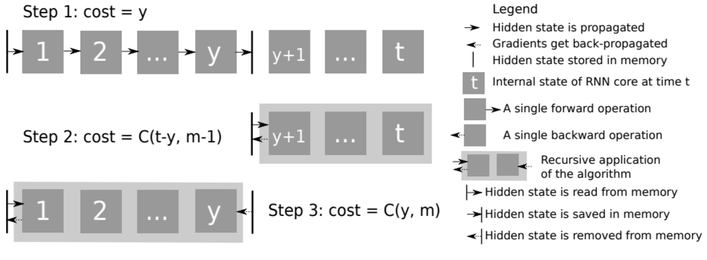
The appeal of checkpointing is that, when done optimally, it offers an extremely favorable (logarithmic) tradeoff between memory usage and computation time. In the Gruslys et al. [2], they determine that checkpointing could theoretically offer a 20x reduction in memory usage at the cost of a 1.3x increase in computation time.

Placement of Checkpoints

Suppose that the cost of performing reverse mode AD on a sequential program with t steps and ability to save m checkpoints under an optimal checkpoint policy is given by $C(t, m)$. Then we have the recurrence

$$C(t, m) = y + C(y, m) + C(t - y, m - 1)$$

We can solve this program directly to determine the optimal policy for placing checkpoints, as the border cases are solvable analytically.



Our Implementation

Our implementation of checkpointing takes as its input functions which behave like parametrized finite automata; they read input and have a persistent internal state. Given such a function, a number of checkpoints and a sequence length, it returns an autograd primitive with a hardcoded gradient.

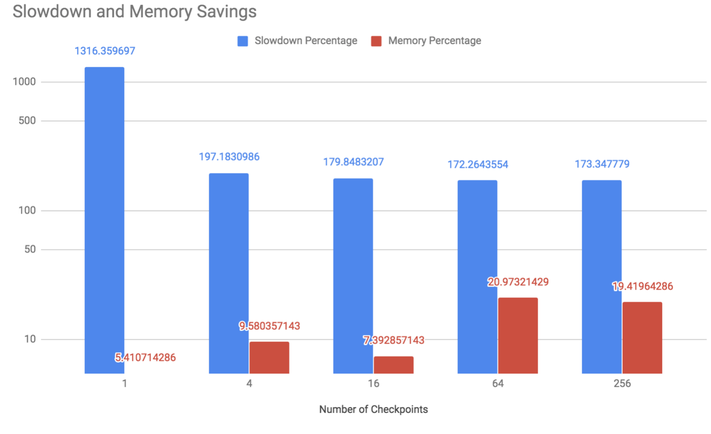
```
def binomial_checkpoint(function, sequence_length, num_checkpoints):
    """
    Args:
        function: takes parameters, condition, and input and produces output.
        sequence_length: sequence length
        num_checkpoints: number of checkpoints we can save

    Returns:
        wrapped: a new primitive whose gradient, when called, performs the checkpointing algorithm of Gruslys et. al (2016)
    """
    def loop_primitive(parameters, initial_condition, inputs):
        conditions = ag_list([initial_condition])
        for i, input in enumerate(inputs):
            conditions += ag_list([function(parameters, conditions[i], input)])
        return conditions

    wrapped_grad = make_bc_vjpmaker(function, sequence_length, num_checkpoints)
    wrapped = primitive(loop_primitive)
    defvjp_argnum(wrapped, wrapped_grad)
    return wrapped
```

As designed, the checkpointing wrapper can accept, for example, an rnn, and will return a checkpointed loop. In the case of a truly iterated and sequential program, the inputs can just be taken as None.

Results and Benchmarks

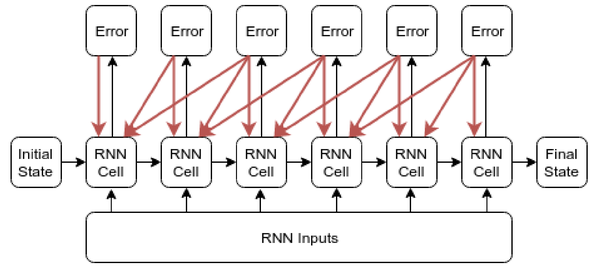
Gruslys et al. [2] declared the theoretical possibility of a 95% reduction in memory with only a 33% increase in computation time. Concretely, we have been able to attain even more significant reductions in memory use, but the overhead imposed by the checkpointing algorithm limited slowdown to around 60% in the best case.

We benchmarked our algorithm with a simple RNN operating on sequential input. We experimented with batch size, sequence length, number of checkpoints, and hidden/output/input sizes. For some configurations, the checkpointing algorithm provided significant memory savings (sometimes on the order of 98%). Still, for most choices of number of checkpoints, we observed roughly a 1.7x slowdown, though obviously the slowdown exploded as the number of checkpoints approached 1. Almost immediately after $n\_checkpoints = 1$, other computational costs began to dominate. This may be a sign that additional optimization is necessary.



Backpropagation through time as a sequential differentiable program

Backpropagation through time is a way of training a recurrent neural net by unfolding it over a given set of inputs and treating it as a feedforward network. A diagram of this procedure is given below.



We can must regard backpropagation through time as a sequential differentiable program, as the outputs at the final element of the sequence depend on the states at the beginning of the sequence, which in turn depend on the parameters of the network — thus, we need to pass gradients all the way through this chain.

Autograd

Autograd is a Python package for automatically differentiating pure Python and numpy code. It was designed by Harvard Intelligent Probabilistic Systems in 2014, and the backend is comprised of a tracing mechanism and associating computational nodes with a VJP (vector jacobian product) function that gets called during the backward pass. To implement binomial checkpointing, we coded a custom VJP for the differential loop given to the left.

Acknowledgements

Thanks to Alex Wiltschko and Dougal Maclaurin for advising.

References

[1] Andreas Griewank. "Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation". In: *ACM Transactions on Mathematical Software* ().

Audrunas Gruslys et al. "Memory-Efficient Backpropagation Through Time". In: *CoRR* abs/1606.03401 (2016). arXiv: 1606.03401. URL: http://arxiv.org/abs/1606.03401.