Deep Learning: Autodiff, Parameter Tying and Backprop Through Time[*]

David Barber
Department of Computer Science
University College London

February 9, 2015

Abstract

How to do parameter tying and how this relates to Backprop through time.

1 Introduction

A common question when learning about Neural Nets (Deep Learning) is how to deal with parameter tying when calculating the gradient of the objective function. In particular, how does this relate to Backprop Through Time [3]? Whilst there is a large literature available on this, the explanations often seen (to my mind) rather over complicated. Below we explain how to deal with parameter tying in general (not just for Neural Nets).

2 Parameter Tying

Consider a simple objective such as

$$E(\theta) = [y - f(\theta g(x\theta))]^2 \tag{1}$$

One can think of this as a Neural Net squared loss objective corresponding to a single training input-output (x, y) with a scalar input x, single hidden layer $h = g(x\theta)$, with weight from input to hidden layer given by θ and output $f(\theta h)$, with weight θ from the hidden layer to the output layer. As a network diagram, this would look something like this $x \underset{\theta}{\rightarrow} h \underset{\theta}{\rightarrow} y$ in which the parameters from the input to hidden layer and hidden layer to output are tied.

We can calculate the gradient directly using the usual chain rule of calculus:

$$\frac{\partial E}{\partial \theta} = -2\left[y - f(\theta g(x\theta))\right] f'(\theta g(x\theta)) \left(\theta g'(x\theta)x + g(x\theta)\right) \tag{2}$$

where f' and g' denote the derivatives of f and g respectively.

Another way to do this is to consider

$$F(\theta_1, \theta_2) = [y - f(\theta_1 g(x\theta_2))]^2 \tag{3}$$

which is a Network with unconstrained parameters, $x \underset{\theta_1}{\rightarrow} h \underset{\theta_2}{\rightarrow} y$. Then

$$\frac{\partial F}{\partial \theta} = \frac{\partial F}{\partial \theta_1}\frac{\partial \theta_1}{\partial \theta} + \frac{\partial F}{\partial \theta_2}\frac{\partial \theta_2}{\partial \theta} \tag{4}$$

[*]UCL Department of Computer Science Technical Note

$$\begin{tikzcd}
x_1 \arrow[d, "A"'] & x_2 \arrow[d, "A"'] & x_3 \arrow[d, "A"'] \\
h_1 \arrow[r, "B"] \arrow[d, "C"'] & h_2 \arrow[r, "B"] \arrow[d, "C"'] & h_3 \arrow[d, "C"'] \\
y_1 & y_2 & y_3
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
x_1 \arrow[d, "A_1"'] & x_2 \arrow[d, "A_2"'] & x_3 \arrow[d, "A_3"'] \\
h_1 \arrow[r, "B_1"] \arrow[d, "C_1"'] & h_2 \arrow[r, "B_2"] \arrow[d, "C_2"'] & h_3 \arrow[d, "C_3"'] \\
y_1 & y_2 & y_3
\end{tikzcd}$$

(b)

Figure 1: **(a)**: A recurrent Neural Net here written for only 3 timesteps. **(b)**: Unconstrained version of (a) used to derive BPTT.

If we now constrain $\theta_1 = \theta_2 = \theta$, we obtain

$$\frac{\partial E}{\partial \theta} = \frac{\partial F(\theta, \theta)}{\partial \theta} = \left.\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_1}\right|_{\theta_1=\theta_2=\theta} + \left.\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_2}\right|_{\theta_1=\theta_2=\theta} \tag{5}$$

where $|_{\theta_1=\theta_2=\theta}$ means that we evaluate the resulting expression (after calculating the partial derivative) at the constrained values.

We can verify that this works:

$$\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_1} = -2\left[y - f(\theta_1 g(x\theta_2))\right] f'(\theta_1 g(x\theta_2)) g(x\theta_2) \tag{6}$$

and

$$\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_2} = -2\left[y - f(\theta_1 g(x\theta_2))\right] f'(\theta_1 g(x\theta_2)) \theta_1 x g'(x\theta_2) \tag{7}$$

Summing equation (6) and equation (7) and evaluating at $\theta_1 = \theta_2 = \theta$, we obtain equation (2).

The conclusion is that we can deal with parameter tying in any objective by the following procedure:

1. Treat all parameters as independent and calculate the gradient with respect to each independent parameter.
2. Sum all the resulting independent gradients together.
3. Evaluate the expression by setting all the independent parameters to the same value.

Note that this is a general result and can be used to deal with parameter tying in any objective, not just Deep Learning and Neural Nets.

3 Backprop Through Time

A recurrent network can be thought of as a deterministic temporal model with inputs x_t, hidden values h_t and outputs y_t. All inputs, hidden values and outputs may be vectors. For a squared loss, we would have an objective of the form

$$E(A, B, C) = \sum_t (y_t - f(h_t; C))^2, \qquad h_t = g(x_t, h_{t-1}; A, B) \tag{8}$$

which means that the output of the network at time t is some function f (parameterised by a matrix C) of the hidden state h_t. Similarly, the hidden value at time t is some function g (parameterised by input to hidden weights A and hidden to hidden weights B) of the input x_t and previous hidden value h_{t-1}, see fig(1a).

To train a recurrent network we need to calculate the gradient with respect to A, B, C. There are several ways to do this with varying storage and time performances, see [4] for a detailed comparison. Perhaps the most obvious approach is to directly calculate the gradient by a forward-propagation algorithm (RTRL) which is also the same technique discussed in [1] and extends recurrent networks to probabilistic models.

A more common approach is Backprop Through Time (BPTT) [3]. We assume that the reader is familiar with the standard backprop algorithm (see for example [2]). Given our understanding about parameter tying, we see that we can calculate the gradient of the recurrent NN objective by first treating all parameters as independent, and then running standard backprop on the resulting architecture, see fig(1b). Finally, we simply sum all the corresponding gradients (for example with respect to the matrices A_1, A_2, A_3 to calculate the gradient with respect to A) and subsequently set the parameters to be equal. This is an efficient exact algorithm which, in contrast to RTRL, runs backwards in time. In practice, to save on storage, backprop is typically run over a fixed window (rather than going back to time 1) which results in an approximation to the true gradient.

3.1 Conclusion

One can deal with parameter tying simply by treating parameters as if they are independent, and then summing all the corresponding gradients. This is particularly useful for objectives such as Neural Nets since one can then make use of standard (and efficient) backprop routines to calculate the unconstrained gradients.

4 AutoDiff

A more general viewpoint on backprop and parameter tying can be established through Automatic Differentiation. AutoDiff takes a function $f(\mathbf{x})$ and returns an exact value (up to machine accuracy) for the gradient

$$g_i(\mathbf{x}) \equiv \frac{\partial}{\partial x_i} f \bigg|_{\mathbf{x}} \tag{9}$$

Note that this is not the same as a numerical approximation (such as central differences) for the gradient. One can show that, if done efficiently, one can always calculate the gradient in less than 5 times the time it takes to compute $f(\mathbf{x})$. This is also *not* the same as symbolic differentiation.

In Symbolic Differentiation, given a function $f(x) = \sin(x)$, symbolic differentiation returns an algebraic expression for the derivative. This is not necessarily efficient since it may contain a great number of terms. As an (overly!) simple example, consider

$$f(x_1, x_2) = \left(x_1^2 + x_2^2\right)^2 \tag{10}$$

$$\frac{\partial f}{\partial x_1} = 2\left(x_1^2 + x_2^2\right) 2x_1, \qquad \frac{\partial f}{\partial x_2} = 2\left(x_1^2 + x_2^2\right) 2x_2 \tag{11}$$

The algebraic expression is not computationally efficient. However, by defining

$$y = 4(x_1^2 + x_2^2) \tag{12}$$

then

$$\frac{\partial f}{\partial x_1} = y x_1, \qquad \frac{\partial f}{\partial x_2} = y x_2 \tag{13}$$

Which is a more efficient *computational* expression. Also, more generally, we want to consider computational subroutines that contain loops and conditional `if` statements; these do not correspond to simple closed algebraic expressions. We want to find a corresponding subroutine that can return the exact derivative efficiently for such subroutines. There are two main flavours of AutoDiff, namely Forward and Reverse mode.

Forward
- This is (usually) easy to implement
- However, it is not (generally) computationally efficient.
- It cannot easily handle conditional statements or loops.

Reverse
- This is exact and computationally efficient.
- It is, however, harder to code and requires a parse tree of the subroutine.
- If possible, one should always attempt to do reverse differentiation.
- As we will discuss, the famous backprop algorithm is just a special case of reverse differentiation.
- Reverse differentiation is also important since, with it, one can understand (for example) how to deal easily with calculating the derivative of a function subject to parameter tying.

4.1 Forward Differentiation

Consider $f(x) = x^2$ and that we wish to compute the derivative of this.

4.1.1 Central Differences

The most well known approach to *approximating* a derivative is to use

$$f'(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} = \frac{x^2 + 2\epsilon x + \epsilon^2 - x^2 + 2\epsilon x - \epsilon^2}{2\epsilon} \tag{14}$$

which in this case gives the exact result. More generally, the approximation is accurate up to order ϵ^3. Whilst this can be useful, in addition to it only being an approximation, it is also potentially slow since, for vector x, the calculation has to be repeated for each component of the vector.

4.1.2 Complex arithmetic

$$f(x + i\epsilon) = (x + i\epsilon)^2 = x^2 - \epsilon^2 + 2i\epsilon x \tag{15}$$

Hence

$$f'(x) = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \text{Im}\left(f(x + i\epsilon)\right) \tag{16}$$

This also holds for any smooth function (one that an be expressed as a Taylor series). For finite ϵ this gives an *approximation* only. More accurate approximation than standard finite differences since we do not subtract two small quantities and divide by a small quantity – the complex arithmetic approach is more numerically stable. To implement, we need to overload all functions so that they can deal with complex arithmetic.

4.1.3 Dual arithmetic

Consider $f(x) = x^2$. Define an idempotent variable, ϵ such that $\epsilon^2 = 0$.

$$f(x + \epsilon) = (x + \epsilon)^2 = x^2 + 2x\epsilon \tag{17}$$

Hence

$$f'(x) = \text{DualPart} f(x + \epsilon) \tag{18}$$

This holds for any smooth function $f(x)$ and non-zero value of ϵ. To implement this we need to overload every function in the subroutine to work in dual arithmetic. Also note that this is not an approximation – it is an *exact* numerical computation of the derivative (up to machine accuracy). Whilst exact, this is, however, not necessarily efficient.

4.2 Reverse Differentiation

A useful graphical representation is that the total derivative of f with respect to x is given by the sum over all path values from x to f, where each path value is the product of the partial derivatives of the functions on the edges:

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial g}\frac{dg}{dx}$$

$$\begin{tikzcd}
x \arrow[rd, "\frac{dg}{dx}"] \arrow[dd, "\frac{\partial f}{\partial x}"'] & \\
& g \arrow[ld, "\frac{\partial f}{\partial g}"] \\
f &
\end{tikzcd}$$

Example 1

For $f(x) = x^2 + xgh$, where $g = x^2$ and $h = xg^2$

$$\begin{tikzcd}
x \arrow[dd, "2x+gh"'] \arrow[rd, "g^2"] \arrow[rrd, "2x"] & & \\
& h \arrow[ld, "xg"] & g \arrow[l, "2gx"'] \arrow[lld, "xh"] \\
f & &
\end{tikzcd}$$

$$f'(x) = (2x + gh) + (g^2 xg) + (2x2gxxg) + (2xxh) = 2x + 8x^7 \tag{19}$$

Example 2

Consider

$$f(x_1, x_2) = \cos(\sin(x_1 x_2)) \tag{20}$$

We can represent this computationally using an Abstract Syntax Tree (AST), also known as the Computation Tree:

$$\begin{tikzcd}
x_1 \arrow[rd] & & x_2 \arrow[ld] \\
& f_1 \arrow[d] & \\
& f_2 \arrow[d] & \\
& f_3 &
\end{tikzcd}$$

$$\begin{aligned} f_1(x_1, x_2) &= x_1 x_2 \\ f_2(x) &= \sin(x) \\ f_3(x) &= \cos(x) \end{aligned}$$

Given values for x_1, x_2, we first run forwards through the tree so that we can associate each node with an actual function value.

$$\frac{df_3}{dx_1} = \frac{\partial f_3}{\partial f_2}\frac{df_2}{dx_1} = \underbrace{\frac{\partial f_3}{\partial f_2}\frac{df_2}{df_1}}_{\frac{df_3}{df_1}}\frac{df_1}{dx_1} \tag{21}$$

Similarly,

$$\frac{df_3}{dx_2} = \underbrace{\frac{\partial f_3}{\partial f_2}\frac{df_2}{df_1}}_{\frac{df_3}{df_1}}\frac{df_1}{dx_2} \tag{22}$$

The two derivatives share the same computation branch and we want to exploit this.

$$\begin{tikzcd}
x_1 \arrow[dr, "\frac{\partial f_1}{\partial x_1} = x_2"'] & & x_2 \arrow[dl, "\frac{\partial f_1}{\partial x_2} = x_1"] \\
& f_1 \arrow[d, "\frac{\partial f_2}{\partial f_1} = \cos(f_1)"] & \\
& f_2 \arrow[d, "\frac{\partial f_3}{\partial f_2} = -\sin(f_2)"] & \\
& f_3 &
\end{tikzcd}$$

4.3 The AutoDiff Algorithm

1. Find the reverse ancestral (backwards) schedule of nodes (In example 2 above, this would be f_3, f_2, f_1, x_1, x_2).
2. Start with the first node n_1 in the reverse schedule and define $t_{n_1} = 1$.
3. For the next node n in the reverse schedule, find the child nodes $\text{ch}(n)$. Then define

$$t_n = \sum_{c \in \text{ch}(n)} \frac{\partial f_c}{\partial f_n} t_c$$

4. The total derivatives of f with respect to the root nodes of the tree (here x_1 and x_2) are given by the values of t at those nodes.

This is a general procedure that can be used to automatically define a subroutine to efficiently compute the gradient. It is efficient because information is collected at nodes in the tree and split between parents only when required.

4.3.1 Dealing with loops

```
f=function(x)
f=0
for i=1:10
  f=f+cos(f*x^i)
end
```

```
df=function(x)
f=0
df=0
for i=1:10
  f=f+cos(f*x^i)
  df=df-sin(f*x^i)*f*i*x^{i-1}+df*x^i
end
```

In the above example, we expanded the derivative of the cos term symbolically. In AutoDiff, unless the derivatives of such standard functions are given, we would need to replace this step with the computations on the AST and include these computations in the AutoDiff procedure.

4.4 Notes

Reverse AutoDiff generalises (and predates) Backprop since it holds for any computation tree. Another viewpoint of parameter tying is to write down the computation tree and carry out AutoDiff. In equation (1) the computation tree can be written as:

$$\begin{tikzcd}
y \arrow[dddrr] & x \arrow[dr] & & \theta \arrow[dl] \arrow[ddl] \\
& & g \arrow[d] & \\
& & f \arrow[d] & \\
& & E &
\end{tikzcd}$$

so that parameter tying can be seen as simply linking the same parameter to different nodes in the tree. One then carries out AutoDiff using the standard algorithm, which will have the same effect as we noted in section(2) – making two copies of θ, namely θ_1 and θ_2 and summing over them is equivalent to simply summing over the two child paths of θ in the computation tree.

References

[1] D. Barber. Dynamic Bayesian Networks with Deterministic Tables. In *Advances in Neural Information Processing Systems (NIPS)*, 2003.

[2] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Inc., New York, NY, USA, 1995.

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

[4] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin and D. E. Rumelhart, editors, *Back-propagation: Theory, Architectures and Applications*, chapter 13, pages 433–486. Hillsdale, NJ: Erlbaum, 1995.