1 Problem

We've done a forward pass in a neural network f_θ. But backprop for parameter update needs $\nabla_\theta J$, and that's a lot of derivation via chain rule. We want to automate this — after all, it's just differentiation.

However, the limit definition of the derivative $\lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$ for computing $\nabla_\theta J$ has some issues: it's numerically unstable (floating point errors for small h) and slow (each parameter θ_i requires its own full forward pass through the network). We need something faster and more reliable.

2 Solution

Big-picture idea: we'll hard-code a set of primitive functions along with its derivatives with respect to each of its arguments: a primitive function $f(a_0, ..., a_{n-1})$ hard-codes n derivative functions (one for each positional argument) $d_{f,i} = \frac{\partial f}{\partial a_i}$ for each i. Then, we'll represent bigger functions (like $J(\theta)$, which involves a full forward pass and then a cost function on the network's final output) as simply compositions of these primitive functions; the gradient $\nabla_\theta J$ is computable via chain rule across these smaller primitives.

Construct a computational graph that's centered around primitive functions: each node represents a function call.

- Each node u owns:
  - A **primitive function** f_u (which primitive function it is: addition, multiplication, relu — these must be hard-coded)
  - An **arg list** $\text{adj}(u)$ (the nodes whose results are used as arguments for this node's fn) — this is the node's adjacency list, store edges in order of the positional args.
  - A cached **result** $\hat{u}$ (the value as a result of its function taking in its args)
  - A cached **gradient** $\text{grad}(u)$ (which will be used in backprop; if this node represents $\hat{u} = f_u(\hat{\text{adj}}(u))$, then the gradient value holds $\frac{\partial \text{root}}{\partial \hat{u}}$)
- This will form a DAG where the cost function call J roots the DAG, and the parameters θ_i are the leaves of the DAG.

The graph and its nodes are all constructed as computation occurs. Then, we can start from any node s and compute $\frac{\partial \hat{s}}{\partial \hat{u}}$ for all u reachable from s. We must traverse the graph in topological order when computing gradients.

- In our neural network case, we'd want to start from node s where $J = \hat{s}$ and end at all nodes t_i where $\theta_i = \hat{t}_i$, in order to compute $\nabla_\theta J$. Obviously, $\frac{\partial J}{\partial J} = 1$ to start, and initialize all grad values to zero.
- Then, suppose we're sitting at node v. For each edge that exists in $\text{adj}(v)$, let the child be u, and suppose it's specifically passed in as the ith positional argument which is denoted as $a_i = \hat{u}$. a_i only appears as the ith argument, even if u appears among multiple positional args; $\frac{\partial J}{\partial \hat{u}} += \sum_{a_i} \frac{\partial J}{\partial a_i}$ for a single parent by multivariate chain rule, and we do this for all parents of u for the full $\frac{\partial J}{\partial \hat{u}}$.
  - We need to find $\frac{\partial J}{\partial a_i} = \frac{\partial J}{\partial \hat{v}} \frac{\partial \hat{v}}{\partial a_i}$. Equivalently: $\frac{\partial J}{\partial a_i} = \text{grad}(v) \frac{\partial \hat{v}}{\partial a_i}$. (notice: needing $\frac{\partial J}{\partial \hat{v}}$ to be finalized is why we need topo traversal — v can't explore children before $\text{grad}(v)$ is finalized.) But what's $\frac{\partial \hat{v}}{\partial a_i}$?
  - Remember that $\hat{v} = f_v(\hat{\text{adj}}(v))$, so differentiating both sides w.r.t the positional arg, we get
$$\frac{\partial \hat{v}}{\partial a_i} = \frac{\partial}{\partial a_i} f_v(\hat{\text{adj}}(v)) = d_{f_v,i}(\hat{\text{adj}}(v))$$
  - So then in the end, we do $\boxed{\text{grad}(u) += \text{grad}(v) \cdot d_{f_v,i}(\hat{\text{adj}}(v))}$.
- We traverse until we've finished at the leaves. Then, we pick out the gradients of the nodes we care about (aka the nodes t_i such that $\hat{t}_i = \theta_i$, which gives us $\frac{\partial J}{\partial \theta_i}$).

3 Extending to tensors

Any tensor operations generate giant autograd graphs, and switching our "level of primitives" from scalars to tensors reveals another problem: $C = \text{matmul}(A, B) = AB$ for example gives a giant 4D Jacobian for $d_{\text{matmul},i}$, where something like $\frac{\partial C}{\partial A}$ would have a matrix of $\frac{\partial C_{ij}}{\partial A_{i'j'}}$ for all i', j' given a fixed i, j — a matrix

of partials for **each** entry in C. So: we'll fold $d_{f_v,i}$ into a broader chain rule function: $g_{f_v,i} = \frac{\partial J}{\partial \hat{v}} d_{f_v,i}$ (tensor contraction, not scalar multiplication). We'll figure out what $\frac{\partial J}{\partial \hat{v}} d_{f_v,i}$ is for each of our hard-coded primitives in order to directly compute our result without touching any intermediates.

Also, at the tensor level, we'll be doing broadcasting (common example: broadcasting b across the columns of WX). The idea is that inside g, the ops naturally broadcast, yielding a broadcast gradient internally; then, reduce the broadcasted gradient back to the shape of $\hat{u}$ to collect the gradient contributions of each original entry.

- For example, when we broadcast b across the columns of WX by creating the broadcasted tensor B, changing any b_i entry changes all B_{ij} for $0 \leq j < \text{cols}(WX)$. So J's change wrt a small b_i change is just the sum of all the ways b_i's change reaches J: via all the same exact changes in B_{ij} in that row i.

So in practice, if you're doing tensor autograd, replace $d_{f_v,i}(\text{pos-args})$ with $g_{f_v,i}(\text{grad}(v), \text{pos-args})$ — if visiting child u from parent v, do $\boxed{\text{grad}(u) \mathrel{+}= g_{f_v,i}\left(\text{grad}(v), \hat{\text{adj}}(v)\right)}$.

In practice, these might be the primitives pre-defined for a simple FNN ($G = \frac{\partial J}{\partial \text{result}}$, always, and $g_{f,i}$ always implicitly un-broadcasts its result at the end to match the dimensions of a_i):

- **Matmul**: $C = AB$, derived via the vector-Jacobian product.
  - $g_{\text{matmul},0}(G, A, B) = GB^t$
  - $g_{\text{matmul},1}(G, A, B) = A^t G$
- **Add**: $C = A + B$. These element-wise ops are simpler to derive; for example, think $\frac{\partial J}{\partial A_{ij}} = \frac{\partial J}{\partial C_{ij}} \frac{\partial C_{ij}}{\partial A_{ij}} = \frac{\partial J}{\partial C_{ij}} \cdot 1$, so $\frac{\partial J}{\partial A} = G$.
  - $g_{\text{add},0}(G, A, B) = G$
  - $g_{\text{add},1}(G, A, B) = G$
- **Element-wise mul**: $C = A \odot B$. Think $\frac{\partial J}{\partial C_{ij}} \frac{\partial C_{ij}}{\partial A_{ij}} = G_{ij} \cdot B_{ij}$.
  - $g_{\text{mul},0}(G, A, B) = G \odot B$
  - $g_{\text{mul},1}(G, A, B) = G \odot A$
- **Element-wise division**: $C = A \oslash B$. Think element-wise quotient rule for the second one.
  - $g_{\text{div},0}(G, A, B) = G \oslash B$
  - $g_{\text{div},1}(G, A, B) = -G \odot A \oslash B^{\odot 2}$
- **ReLU**: $B = \text{ReLU}(A)$. Think $\frac{\partial J}{\partial B_{ij}} \frac{\partial B_{ij}}{\partial A_{ij}} = G_{ij} \cdot \text{ReLU}'(A_{ij})$.
  - $g_{\text{ReLU},0}(G, A) = G \odot 1[A > 0]$
- **Exp**: $B = \exp(A)$.
  - $g_{\exp,0}(G, A) = G \odot \exp(A)$
- **Log**: $B = \log(A)$. Think $\frac{\partial J}{\partial B_{ij}} \frac{\partial B_{ij}}{\partial A_{ij}} = G_{ij} \cdot \frac{1}{A_{ij}}$.
  - $g_{\log,0}(G, A) = G \odot A^{\odot(-1)}$
- **Sum-reduce along axis** k: $B = \text{sum}(A, \text{axis} = k)$. For example for 2x2 A, $\frac{\partial J}{\partial A_{ij}} = \frac{\partial J}{\partial B_i} \frac{\partial B_i}{\partial A_{ij}} = G_i \cdot 1$. G is const across all j since G depends on i, and j is the sum-reduced axis, so just broadcast back to shape(A).
  - $g_{\text{sum},0}(G, A) = \text{broadcast}(G, \text{shape}(A))$.
- **Max-reduce along axis** k: $B = \max(A, \text{axis} = k)$. Think of it like this: A is composed of a bunch of vectors that stretch along axis k (for example, if axis k is rows, then all the vectors are rows). B squashes axis k, only keeping the max. So changing an entry in A only changes the corresponding B entry of that A axis-k vector if that A entry is the max of that axis-k vector. To formalize this with gradients, in $\frac{\partial B}{\partial A}$, put a 1 at the argmax pos for each axis k vector, and 0 otherwise. If multiple argmax positions, choose one.
  - Not going to formalize this with math — just implement it programmatically, it'll be easier.