CS 189 Introduction to Machine Learning
Fall 2017 **DIS7**

## 1 Backpropagation

In this discussion, we will explore the chain rule of differentiation, and provide some algorithmic motivation for the backpropagation algorithm. Those of you who have taken CS170 may recognize a particular style of algorithmic thinking that underlies the computation of gradients.

Let us begin by working with simple functions of two variables.

(a) Define the functions $f(x) = x^2$ and $g(x) = x$, and $h(x,y) = x^2 + y^2$. Compute the derivative of $\ell(x) = h(f(x), g(x))$ with respect to $x$.

**Solution:** We can write $\ell(x) = x^4 + x^2$, and from there, this is just a simple differentiation problem, and we have $\ell'(x) = 4x^3 + 2x$. It is also helpful to think of this from the chain rule perspective, where we have $\ell(x) = f(x)^2 + g(x)^2$, and differentiating using the chain rule yields

$$\begin{aligned}\ell'(x) &= 2f(x)f'(x) + 2g(x)g'(x)\\ &= 2x^2(2x) + 2(x)\cdot 1\\ &= 4x^3 + 2x.\end{aligned}$$

(b) Chain rule of multiple variables: Assume that you have a function given by $f(x_1, x_2, \ldots, x_n)$, and that $x_i = g_i(w)$ for a scalar variable $w$. How would you compute $\frac{\partial f}{\partial w}$?

The function graph of this computation is given below:

**Solution:** This is the chain rule for multiple variables. In general, we have

$$\frac{\partial f}{\partial w} = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}\frac{\partial x_i}{\partial w}.$$

If you're confused about this, look at how it played out in the previous part, where we had two variables $x_1$ and $x_2$.

(c) Let $w_1, w_2, \ldots, w_n \in \mathbb{R}^d$, and we refer to these variables together as $W$. We also have $x \in \mathbb{R}^d$ and $y \in \mathbb{R}$. Consider the function

$$f(W,x,y) = \left(y - \sum_{i=1}^{n} \phi(w_i^\top x + b_i)\right)^2.$$

Write out the function computation graph (also sometimes referred to as a pictorial representation of the network). This is a directed graph of decomposed function computations, with

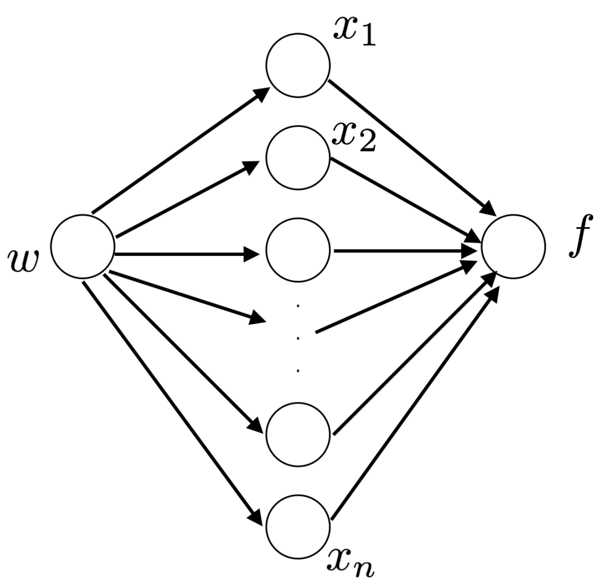

Figure 1: Example function computation graph

the function at one end (which we will call the sink), and the variables $W, x, y$ at the other end (which we will call the sources).

**Solution:**

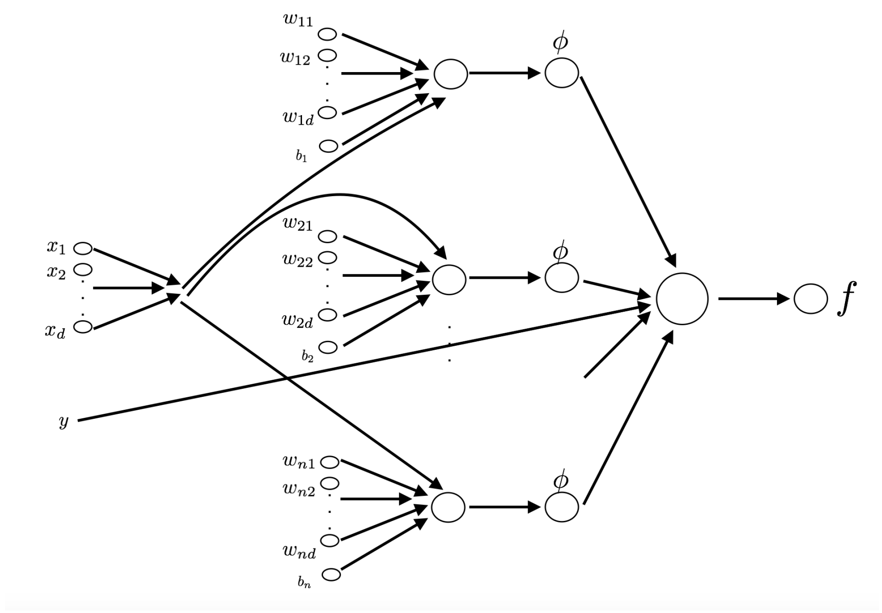

Figure 2: Example function computation graph

Alternatively, you could put the weights $w$ on the edges, but the way we've drawn it is easier to understand intuitively. Also, auto-differentiation packages in neural network libraries use precisely these computation graphs.

(d) What is the interpretation of taking a partial derivative of the output with respect to a particular node of this function graph? How can we use these partial derivatives to "move downhill"?

**Solution:** The partial derivative measures how much the function changes when that particular variable is changed, keeping all the other variables in our computation fixed. In particular, assume in the previous example that we are computing $\frac{\partial f}{\partial u_i}$ for some node of the graph $u_i$. In order to see what this looks like, we fix the values of all the parameters of the network that do not influence $u_i$, and vary $u_i$ by a small amount. The partial derivative then tells us the rate at which the function changes. In order to move downhill, we then see that we must toggle $u_i$ in the direction that decreases the function value.

(e) Assume that the function graph is given to you as input, and you wish to determine the partial derivatives of the sink with respect to the sources. Assume that it is a directed acyclic graph (DAG). Show how one would perform this by computing the partial derivatives $\frac{\partial v_j}{\partial v_i}$ for every pair of nodes in the network where $v_j$ is farther away from the sources than $v_i$. How much time (number of operations) does this take?

**Solution:** Let us stack up the nodes in some arbitrary order from left to right as above, with edges going from left to right. Notice that the nodes will form levels.

We are now going to compute $\frac{\partial v_j}{\partial v_i}$ for every pair of nodes in the network where $v_j$ is farther away from the sources than $v_i$. Clearly, this also computes $\frac{\partial f}{\partial w}$ for every source $w$. Let us do this in a feed-forward fashion by starting at the left.

Assume that we have obtained $\frac{\partial v_j}{\partial v_i}$ for every $v_j, v_i$ on the level up to and including level $t$. Then we can express (by inspecting the multivariate chain rule) the value $\frac{\partial v_\ell}{\partial v_i}$ for some $v_\ell$ at level $t+1$ as a weighted combination of values $\frac{\partial v_j}{\partial v_i}$ for each $v_j$ that has a direct input to $v_\ell$. This description shows that the amount of computation for a fixed $i, \ell$ pair is proportional to the in-degree of $v_\ell$. Therefore, in total for a fixed $i$, we do work proportional to the sum of all in degrees given by $2|E|$, where $E$ represents the set of edges in our graph. This amount of work happens for all $|V|$ values of $i$, letting us conclude that the total work in the algorithm is $O(|V||E|)$.

(f) Let us now use a different method to save computation. Starting from the output node (sink), recursively compute derivatives for nodes at distance $1, 2, \ldots$ from the sink. This is called the dynamic programming approach, and corresponds to the backpropagation algorithm. How long does this take?

**Solution:** Here, we look backwards. Look at some node $u$ with an outgoing edge to the sink node. The node $u$ receives a message along this outgoing edge from the sink at the other end of that edge, corresponding to the value $\frac{\partial f}{\partial f} = 1$. It then sends the following message to any node $z$ adjacent to it at a lower level: $1 \cdot \frac{\partial u}{\partial z}$. At $z$, we now sum all of the incoming messages to obtain $\sum_{u: z \to u} 1 \cdot \frac{\partial u}{\partial z} = \frac{\partial f}{\partial z}$.

This suggests a more general algorithm. Every node $u$ receives a message along each outgoing edge from the node at the other end of that edge. It sums these messages to get a number $S$ (if $u$ is the output of the entire net as above, then define $S = 1$) and then it sends the following message to any node $z$ adjacent to it at a lower level: $S\frac{\partial u}{\partial z}$. Eventually, we will have $\frac{\partial f}{\partial w}$ for each source $w$.

How much work is done here? Every node does work proportional to the sum of its out-degree (in summing the messages) and in-degree (in sending messages to lower levels). Thus, the total work is the sum of degrees, which is given by $2|E|$. The run-time of the algorithm is therefore $O(|E|)$, which can be a massive speedup from before.

(g) Discuss how gradient descent would work on the function $f(W,x,y)$ if we use backpropagation as a subroutine to compute gradients with respect to the parameters $W$ (with $x$ and $y$ given).

**Solution:** In order to compute a gradient update, we require $w_t = w_{t-1} - \gamma \nabla f(w_t)$. The gradient step can be computed by backpropagation above. However, we also require the function value in order to compute each of the gradients, and note that as $w$ changes, the function values at each of the nodes changes. Therefore, the gradient descent algorithm proceeds with a forward pass (in order to compute the funciton values at each of the nodes) followed by a backward pass via backpropagation.

(h) How might you make the backpropagation steps above more efficient by using vector operations?

**Solution:** We are computing partial derivatives one at a time, but this can be vectorized by using gradients. For instance, we can compute and pass gradients to the leaf nodes $d$ at a time instead of passing each partial derivative. Speedups are also obtained by using matrix vector multiplications in place of vector-vector multiplications, and via batching of data. You will see examples of these while implementing backpropagation in your next homework.

## 2 Derivatives of simple functions

Compute the derivatives of the following simple functions used as non-linearities in neural networks.

(a) $\sigma(x) = \frac{1}{1+e^{-x}}$

**Solution:** Taking the derivative via chain rule, we have

$$\sigma'(x) = -\frac{1}{(1+e^{-x})^2}(-e^{-x}) = \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) = \sigma(x)(1-\sigma(x)).$$

(b) $\mathsf{ReLu}(x) = \max(x,0)$

**Solution:** The derivative here is equal to 1 if $x > 0$, and 0 if $x < 0$. At 0, the function is not differentiable, so we must pick a "subgradient", which is some tangent to the function. It is typical to pick either 0 or 1.

(c) $\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

**Solution:** Notice that $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}} = \sigma(2x) - (1 - \sigma(2x)) = 2\sigma(2x) - 1$.

Hence, by chain rule, it is clear that the derivative is just $4\sigma'(2x) = 4\sigma(2x)(1 - \sigma(2x))$.

(d) Leaky ReLu: $f(x) = \max(x, -0.1x)$

**Solution:** This is similar to the first part, where the derivative takes value 1 and $-0.1$ when $x$ is positive and negative, respectively. At 0, the sub-gradient is anything between $-0.1$ and 1.