EECS 189 Introduction to Machine Learning
Fall 2020 Note 14

1 Training Neural Networks

We have seen that first-order optimization techniques such as **gradient descent** and **stochastic gradient descent** are effective tools for minimizing differentiable cost functions. In order to implement these techniques, we need to be able to compute the gradient of the cost function with respect to the weights. The chain rule allows us to compute these derivatives in principle, but as we will see, the order of the computations matters in neural networks. The **backpropagation** algorithm takes advantage of the directed acyclic graph (DAG) nature of feedforward neural networks to calculate these derivatives efficiently.

1.1 Computational graphs

We assume that the our network can be expressed as a finite directed acyclic graph $G = (V, E)$, sometimes called the **computational graph** of the network. Each vertex $v_i \in V$ represents the result of some differentiable[1] computation. Each edge represents a computational dependency: there is an edge $(v_i, v_j) \in E$ if and only if the value computed at v_i is used to compute v_j. We denote the set of outgoing neighbors of a node v_i by

$$\text{out}(v_i) = \{v_j \in V : (v_i, v_j) \in E\}$$

Furthermore, some of these vertices have special significance. There is a vertex $\ell \in V$, representing the loss function, which contains no outgoing edges (i.e. $\text{out}(\ell) = \varnothing$). There is also some subset of vertices $W \subset V$ representing the trainable parameters of the network. Our objective is to efficiently calculate $\frac{\partial \ell}{\partial w_i}$ for each $w_i \in W$.

The primary mathematical tool employed in backpropagation is the chain rule. This allows us to write

$$\frac{\partial \ell}{\partial v_i} = \sum_{v_j \in \text{out}(v_i)} \frac{\partial \ell}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

The intuition here is that the value computed at v_i affects potentially all of the vertices to which it is an input, and each of those vertices affects the loss in some way. The total contribution of v_i to the loss must be summed over these downstream effects.

We could expand recursively to get an expression for each weight:

$$\frac{\partial \ell}{\partial w_i} = \sum_{v_j \in \text{out}(w_i)} \frac{\partial \ell}{\partial v_j} \frac{\partial v_j}{\partial w_i}$$

[1] A number of common neural network operations, such as the ReLU activation function, are not everywhere differentiable. In practice it is sufficient to be differentiable except at finitely many points.

$$= \sum_{v_j \in \text{out}(w_i)} \sum_{v_k \in \text{out}(v_j)} \frac{\partial \ell}{\partial v_k} \frac{\partial v_k}{\partial v_j} \frac{\partial v_j}{\partial w_i}$$

$$\vdots$$

$$= \sum_{\text{paths } v^{(1)}, \dots, v^{(k)} \text{ from } w_i \text{ to } \ell} \frac{\partial \ell}{\partial v^{(k)}} \frac{\partial v^{(k)}}{\partial v^{(k-1)}} \cdots \frac{\partial v^{(2)}}{\partial v^{(1)}} \frac{\partial v^{(1)}}{\partial w_i}$$

However, computing the derivative by evaluating this expression is quite inefficient, as many terms appear in more than one path from w_i to ℓ, so we are doing more work than necessary.

1.2 Backpropagation

The backpropagation algorithm combines the chain rule with the principles of **dynamic programming**: dividing a large problem into simpler subproblems, solving these and storing their solutions, and combining the stored solutions to solve larger subproblems or the original problem. In this context, the large problem is computing $\nabla \ell(W)$, and the subproblems are computing the individual terms $\frac{\partial \ell}{\partial w_i}$. The key observation from the first chain rule expression above is that we can reuse work by computing $\frac{\partial \ell}{\partial v_i}$ in a "back to front" order. That is, before computing $\frac{\partial \ell}{\partial v_i}$, we should compute $\frac{\partial \ell}{\partial v_j}$ for each $v_j \in \text{out}(v_i)$. Because our computational graph is a DAG, such a **topological ordering** can always and efficiently[2] be computed via a **topological sort**.[3] Then the subproblem of computing $\frac{\partial \ell}{\partial v_i}$ can be easily accomplished by combining the stored values $\frac{\partial \ell}{\partial v_j}$ with the terms $\frac{\partial v_j}{\partial v_i}$, which can typically be computed analytically based on our knowledge of what mathematical computations each vertex performs. Let us consider a few examples of computations that the vertices of neural network computation graphs perform, to get a concrete sense of what these $\frac{\partial v_j}{\partial v_i}$ terms look like.

1.3 Derivatives of common neural network elements

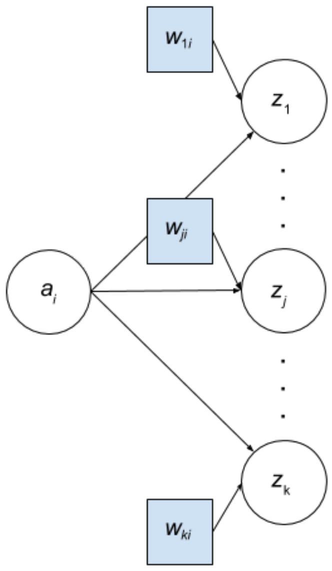
1.3.1 Fully connected layers

In a standard fully-connected layer, each vertex calculates z_j as a linear combination of the activations a_i of the previous layer, with weights w_{ji}:

$$z_j = \sum_i w_{ji} a_i$$

We have omitted layer indexing to keep the notation simple, but keep in mind that this a_i is the result of some computation performed at the previous layer[4], and these z_j are likely used as inputs to vertices at later layers. This part of the computational graph looks like

[2] In time linear in the size of the graph: $O(|V| + |E|)$.
[3] See CS 170!
[4] unless it is the input layer



In the image above, $\text{out}(w_{ji}) = \{z_j\}$, and it is straightforward to see that

$$\frac{\partial z_j}{\partial w_{ji}} = a_i$$

so

$$\frac{\partial \ell}{\partial w_{ji}} = \frac{\partial \ell}{\partial z_j} a_i$$

Observe that we must use the activations a_i that were previously computed in the forward pass.

We must also compute the derivatives z_j with respect to a_i so that we can pass these backward to earlier layers. In the image above, $\text{out}(a_i) = \{z_1, \dots, z_k\}$, and it is straightforward to see that

$$\frac{\partial z_j}{\partial a_i} = w_{ji}$$

so

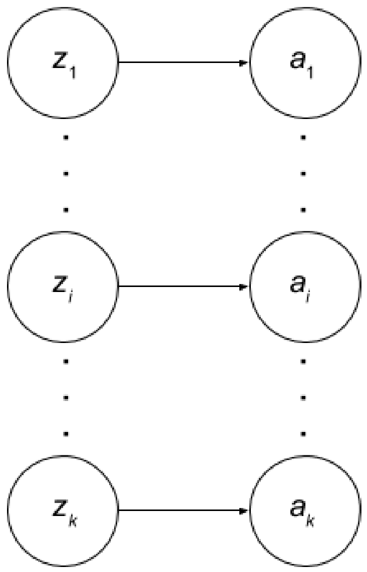
$$\frac{\partial \ell}{\partial a_i} = \sum_{j=1}^{k} \frac{\partial \ell}{\partial z_j} w_{ji}$$

1.3.2 Element-wise nonlinearities

After taking linear combinations, it is typical to insert a nonlinearity. (Recall from the previous note that nonlinearities are at the heart of neural networks' expressive power.) In most cases, this nonlinearity is applied elementwise. Again omitting layer indexing, we might write such a computation as

$$a_i = \sigma(z_i)$$

where z_i is the value from the previous layer, and σ is the activation function. This part of the computational graph looks like



In the image above, out$(z_i) = \{a_i\}$, and it is straightforward to see that

$$\frac{\partial a_i}{\partial z_i} = \sigma'(z_i)$$

so

$$\frac{\partial \ell}{\partial z_i} = \frac{\partial \ell}{\partial a_i}\sigma'(z_i)$$