Calculus for Deep Learning, with Vectors, Matrices, and a few Tuples

Keith Dillon

This note derives the gradients useful for training a typical neural network consisting of a stack of layers. We start from vector calculus and matrix algebra fundamentals, and apply them to components of the model successively. We utilize a minimal extension to these structures in the form of tuples, to minimize use of additional mathematical theory. These tuples represent lists for bookkeeping of multiple data terms, for which an operation can be repeated, implying a software implementation external to the mathematical calculations. This allows us to avoid the need for extensions such as Kronecker products or tensor operations.

Background

Deep learning is based on fundamentally-simple operations. This may seem surprising the increasingly-impressive results achieved with the method. But in fact this simplicity is key to its success, making it relatively simple to scale to very large datasets and models. The most central and intensive operations are matrix algebra and calculus.

Ultimately, matrix operations are an elegant notation for describing particular combinations of scalar operations. One could choose instead to perform all calculations with scalar or vector algebra. The choice is a trade-off in terms of ease-of-use, both in theory and application (i.e., programming); some operations may appear simpler or easier to implement with different structures. In deep learning, popular software packages such as Tensorflow ([1]) and Pytorch ([6]) opted to utilize a more general notation yet, based on tensors (more-accurately called multiway arrays in this case). This allows for the most abstraction of course, while leaving the problem of achieving efficiency in the hands of the framework's programmer and their compiler, who must operate with this relatively-esoteric notation.

Restricting deep learning to vector and matrix operations, as in ([5]), is enough to elegantly describe most but not all operations in deep learning. Additional structure is often addressed by vectorizing or matricizing, reshaping matrices or tensors into lower-dimensional structures ([2]). This leads to use of Kronecker products to correspondingly reshape the mathematical operations, which can quickly become complicated ([4],[7]).

Here will will follow a programming-minded strategy that separates problematic higher-order information from the mathematical operations. We will minimize the use of any new structures or operations, except for one, the tuple. A tuple is a simple ordered list of structures, which themselves may be scalars, vectors or matrices. The goal of doing this is to support efficient implementation with a standard matrix framework such as NumPy ([3]), while relegating additional complexity to "control code".

Vectors, Matrices, and Tuples

Vector and Matrix algebra are abstract descriptions that encapsulate many scalar algebra operations into single vector or matrix operations. A sectors may be viewed as a tuple or list of scalar variables, e.g.,

$$\mathbf{v} = [v_1, v_2, ..., v_M]_1. \tag{1}$$

where the subscript "1" is a label to simply indicate that this is our first list (more will come). The ordering of scalar elements x_i is not necessarily meaningful, but the identity of the ith element must be maintained. For example, it may represent a particular pixel in an image. So the most straightforward approach is to maintain the ordering. Basic operations with vectors such as addition and multiplication by a scalar are simply scalar operations applied to all elements in the vector. Of course, many new properties (e.g., norms) and operations (e.g., the inner product) also arise when dealing with multiple variables, which may be important to a given task. Hence vector algebra goes far beyond just tuples.

A matrix may be similarly viewed as a book-keeping tool to start, in this case as a list of vectors, e.g.,

$$\mathbf{M} = [\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_N]_2, \tag{2}$$

with matrix instructions simply as repeated vector instructions for all vector elements. Again, of course, new properties and operations become possible, leading to the field of matrix algebra. Alternatively, we can view matrix data structures as lists of lists. In forming lists of matrices, we reach what we might view as rank-3 tuples, lists of lists of lists. For example,

$$\mathbf{T} = [\mathbf{M}_1, \mathbf{M}_2, ..., \mathbf{M}_K]_3, \tag{3}$$

Again, we may find a deep field of new properties and functions in dealing with such structures, called tensors (though perhaps not accurately) or multi-way arrays, though they are far less widely used compared to vectors and matrices.

In this report we will minimize our use of this third rank of structure, except for bookkeeping. So we will use the well-known algebra for vectors and matrices, but utilize tuples whenever vectors or matrices are not sufficient.

Vector Calculus

A vector argument is a compact notation for describing a multi-variable function.

$$f(\mathbf{x}) = f(x_1, x_2, ..., x_N) \tag{4}$$

The so-called “total derivative” of such a function with respect to its vector argument, similarly, is a compact notation to represent the vector of all its partial derivatives with respect to each scalar variable. The notation given here as a row vector is common though not universally used.

$$\frac{df(\mathbf{x})}{d\mathbf{x}} = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_N}\right) = \nabla_{\mathbf{x}}^T f \tag{5}$$

This is the transpose of the gradient from vector calculus.

A vector-valued-function, conversely, is a compact notation for simultaneously describing multiple functions over the same domain; the domain itself may be scalar or vector. Here we start with an example over a scalar domain, t.

$$\mathbf{v}(t) = \begin{pmatrix} v_1(t) \\ v_2(t) \\ \vdots \\ v_M(t) \end{pmatrix} \tag{6}$$

The derivative of the vector-valued function is simply the vector of derivatives of the multiple functions.

$$\frac{\partial \mathbf{v}(t)}{\partial t} = \begin{pmatrix} \frac{\partial}{\partial t} v_1(t) \\ \frac{\partial}{\partial t} v_2(t) \\ \vdots \\ \frac{\partial}{\partial t} v_M(t) \end{pmatrix} \tag{7}$$

Note that for the scalar case, the total derivative is the same as the partial derivative. Now we can see the convenience of defining a gradient as a row-vector, since it leaves the column dimension for independently describing the different scalar functions. Given a vector-valued function over a vector domain,

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_M(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} f_1(x_1, x_2, ..., x_N) \\ f_2(x_1, x_2, ..., x_N) \\ \vdots \\ f_M(x_1, x_2, ..., x_N) \end{pmatrix}, \tag{8}$$

the total derivative can be described by combining the derivatives for each function as rows into a matrix, also known as the Jacobian,

$$\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \begin{pmatrix} \nabla_{\mathbf{x}}^T f_1 \\ \nabla_{\mathbf{x}}^T f_2 \\ \vdots \\ \nabla_{\mathbf{x}}^T f_M \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_M}{\partial x_1} & \frac{\partial f_M}{\partial x_2} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix} = \mathbf{J} \tag{9}$$

Thus far we have recalled three forms of derivatives, $\frac{\partial}{\partial x_i}$, $\nabla_{\mathbf{x}}$, and $\mathbf{J}$, which produce scalars, column vectors, and matrices, respectively. The total derivative $\frac{d\mathbf{f}}{d\mathbf{x}}$ is, for our purposes, merely a book-keeping convenience to describe how we represent the collection of partial derivatives over all of a function's parameters, leading to each of the above as a special case.

Therefore, from now on we will denote a general function as $\mathbf{f}(\mathbf{x})$, where the domain and range are each vectors of of sizes, $M \times 1$ and $N \times 1$, respectively, with $M \geq 1$ and $N \geq 1$ so that all three cases are included. It will also be convenient at times to use an even more abridged shorthand to represent the total derivative, namely the "d" operator. For example, given a function $\mathbf{f} : \mathbf{x} \to \mathbf{y}$, we define $d\mathbf{f} = \frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}}$.

Activation Function

An activation function is used in neural network models to implement the nonlinearity at neural outputs. It is a one-to-one mapping using the scalar function σ, which may represent various possible alternatives, applied element-wise to each input.

$$\boldsymbol{\sigma}(\mathbf{x}) = \begin{pmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \vdots \\ \sigma(x_M) \end{pmatrix} \tag{10}$$

A common choice is the sigmoid function which has the nice property that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

$$\text{sigmoid}(z) = \frac{1}{1 + \exp(-z)}. \tag{11}$$

A more modern alternative is the rectified linear unit or "ReLU" function,

$$\text{ReLU}(z) = \alpha \max(0, z) = \begin{cases} 0, & \text{if } z < 0 \\ \alpha z, & \text{if } z \geq 0 \end{cases} \tag{12}$$

The derivative can also be computed efficiently,

$$\sigma'(z) = \begin{cases} 0, & \text{if } z < 0 \\ \alpha, & \text{if } z \geq 0 \end{cases} \tag{13}$$

For this simple function, we can see from Eq. (9) that the derivative will yield a (square) diagonal matrix,

$$\frac{\partial \boldsymbol{\sigma}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial}{\partial x_1}\sigma(x_1) & \frac{\partial}{\partial x_2}\sigma(x_1) & ... & \frac{\partial}{\partial x_N}\sigma(x_1) \\ \frac{\partial}{\partial x_1}\sigma(x_2) & \frac{\partial}{\partial x_2}\sigma(x_2) & ... & \frac{\partial}{\partial x_N}\sigma(x_2) \\ \vdots & \vdots & & \vdots \\ \frac{\partial}{\partial x_1}\sigma(x_N) & \frac{\partial}{\partial x_2}\sigma(x_N) & ... & \frac{\partial}{\partial x_N}\sigma(x_N) \end{pmatrix} = \begin{pmatrix} \sigma'(x_1) & 0 & ... & 0 \\ 0 & \sigma'(x_2) & ... & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & ... & \sigma'(x_N) \end{pmatrix} \tag{14}$$

Note that it is also common to indicate the extension of scalar functions to vector versions by simply applying the function to a vector. E.g., $\sigma(\mathbf{x})$, or $\sin(\mathbf{x})$, $\log(\mathbf{x})$, etc., implying a vector-valued function defined with the corresponding scalar function applied element-wise.

The Chain Rule

In this section will derive some basic cases of matrix calculus in detail, to show how they may be addressed elegantly with the total derivative. One of the simplest multi-variable uses of the chain rule is a scalar-valued function composed with a vector-valued function of a scalar variable.

$$f(\mathbf{v}(t)) = f\left(v_1(t), v_2(t), ..., v_N(t)\right) \tag{15}$$

The chain rule for multi-variable functions yields,

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial t} + \frac{\partial f}{\partial v_2}\frac{\partial v_2}{\partial t} + ... + \frac{\partial f}{\partial v_N}\frac{\partial v_N}{\partial t} = \nabla_{\mathbf{v}}^T f \cdot \frac{\partial \mathbf{v}}{\partial t}. \tag{16}$$

Note that this is an inner product, with $\nabla_{\mathbf{v}}^T f$ as a row-vector and $\frac{\partial}{\partial t}\mathbf{v}$ as a column-vector.

The extension to a vector-valued function $\mathbf{f}$ is straightforward, based on treating the function as a collection of scalar-valued functions

$$\mathbf{f}(\mathbf{v}(t)) = \begin{pmatrix} f_1\left(v_1(t), v_2(t), ..., v_N(t)\right) \\ f_2\left(v_1(t), v_2(t), ..., v_N(t)\right) \\ \vdots \\ f_M\left(v_1(t), v_2(t), ..., v_N(t)\right) \end{pmatrix} \tag{17}$$

Simply combine the results similarly for all scalar functions f_i, producing a matrix-vector product,

$$\frac{\partial \mathbf{f}}{\partial t} = \begin{pmatrix} \nabla_{\mathbf{v}}^T f_1 \frac{\partial \mathbf{v}}{\partial t} \\ \nabla_{\mathbf{v}}^T f_2 \frac{\partial \mathbf{v}}{\partial t} \\ \vdots \\ \nabla_{\mathbf{v}}^T f_M \frac{\partial \mathbf{v}}{\partial t} \end{pmatrix} = \frac{\partial \mathbf{f}}{\partial \mathbf{v}} \cdot \frac{\partial \mathbf{v}}{\partial t} \tag{18}$$

Alternatively, if the variables are vector-valued, as in

$$\begin{aligned} f(\mathbf{v}(\mathbf{x})) &= f\left(v_1(x_1, x_2, ..., x_p), v_2(x_1, x_2, ..., x_p), ..., v_N(x_1, x_2, ..., x_p)\right), \\ &= g(\mathbf{x}) \end{aligned} \tag{19}$$

In this case the chain rule produces a vector-matrix product. To derive this using only steps we have derived thus far, we start from the gradient of the composed function g, the basic gradient,

$$\begin{aligned} \frac{df(\mathbf{v}(\mathbf{x}))}{d\mathbf{x}} &= \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} = \nabla_{\mathbf{x}}^T g \\ &= \left(\frac{\partial g}{\partial x_1}, \frac{\partial g}{\partial x_2}, ..., \frac{\partial g}{\partial x_N}\right) \\ &= \left(\frac{\partial f(\mathbf{v}(x_1))}{\partial x_1}, \frac{\partial f(\mathbf{v}(x_2))}{\partial x_2}, ..., \frac{\partial f(\mathbf{v}(x_p))}{\partial x_p}\right) \end{aligned} \tag{20}$$

Note that the terms $\frac{\partial}{\partial x_i} f(\mathbf{v}(x_i))$ ignore the dependence on the other $x_j, j \neq i$ purely to highlight the form of Eq. (16). Using this result, we get

$$\begin{aligned}\frac{df(\mathbf{v}(\mathbf{x}))}{d\mathbf{x}} &= \left(\nabla_{\mathbf{v}}^T f \cdot \frac{\partial \mathbf{v}}{\partial x_1}, \nabla_{\mathbf{v}}^T f \cdot \frac{\partial \mathbf{v}}{\partial x_2}, ..., \nabla_{\mathbf{v}}^T f \cdot \frac{\partial \mathbf{v}}{\partial x_p}\right) \\ &= \nabla_{\mathbf{v}}^T f \cdot \left(\frac{\partial \mathbf{v}}{\partial x_1}, \frac{\partial \mathbf{v}}{\partial x_2}, ..., \frac{\partial \mathbf{v}}{\partial x_p}\right) \\ &= \nabla_{\mathbf{v}}^T f \cdot \frac{\partial \mathbf{v}}{\partial \mathbf{x}} \end{aligned} \tag{21}$$

Perhaps the last simple case, which includes the previous results as special cases, is the composition of vector-valued functions over a vector domain, as in

$$\mathbf{f}(\mathbf{v}(\mathbf{x})) = \begin{pmatrix} f_1(\mathbf{v}(\mathbf{x})) \\ f_2(\mathbf{v}(\mathbf{x})) \\ \vdots \\ f_M(\mathbf{v}(\mathbf{x})) \end{pmatrix} \tag{22}$$

As written above, we see that this can be described as a vector with elements of the form of Eq. (19). The derivative is similarly a vector of gradients of the form of Eq. (21).

$$\frac{d\mathbf{f}(\mathbf{v}(\mathbf{x}))}{d\mathbf{x}} = \begin{pmatrix} \frac{d}{d\mathbf{x}} f_1(\mathbf{v}(\mathbf{x})) \\ \frac{d}{d\mathbf{x}} f_2(\mathbf{v}(\mathbf{x})) \\ \vdots \\ \frac{d}{d\mathbf{x}} f_M(\mathbf{v}(\mathbf{x})) \end{pmatrix} = \begin{pmatrix} \nabla_{\mathbf{v}}^T f_1 \cdot \frac{d\mathbf{v}}{d\mathbf{x}} \\ \nabla_{\mathbf{v}}^T f_2 \cdot \frac{d\mathbf{v}}{d\mathbf{x}} \\ \vdots \\ \nabla_{\mathbf{v}}^T f_M \cdot \frac{d\mathbf{v}}{d\mathbf{x}} \end{pmatrix} = \begin{pmatrix} \nabla_{\mathbf{v}}^T f_1 \\ \nabla_{\mathbf{v}}^T f_2 \\ \vdots \\ \nabla_{\mathbf{v}}^T f_M \end{pmatrix} \cdot \frac{d\mathbf{v}}{d\mathbf{x}} = \frac{d\mathbf{f}}{d\mathbf{v}} \cdot \frac{d\mathbf{v}}{d\mathbf{x}}, \tag{23}$$

resulting in a matrix-matrix product.

These results can be extended to longer chains of functions by using the fact that function composition is associative. We can, for example, compute the gradient of the composition $\mathbf{a} \circ \mathbf{b} \circ \mathbf{c}$ by first computing the gradient of $\mathbf{f} = \mathbf{a} \circ \mathbf{b}$ using Eq. (23), then compute the gradient of $\mathbf{f} \circ \mathbf{c}$, again using Eq. (23). We can proceed to compute the gradient of any number of compositions in this fashion. In general, given a composition of vector-valued functions over vector domains,

$$\begin{aligned}\mathbf{f}(\mathbf{x}) &= \mathbf{f}^{(k)}(\mathbf{f}^{(k-1)}(\ldots(\mathbf{f}^{(0)}(\mathbf{x}))\ldots)) \\ &= \mathbf{f}^{(k)} \circ \mathbf{f}^{(k-1)} \circ \cdots \circ \mathbf{f}^{(0)} \circ \mathbf{x} \end{aligned} \tag{24}$$

we have the chain rule,

$$\begin{aligned}\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} &= \frac{d\mathbf{f}^{(k)}}{d\mathbf{f}^{(k-1)}} \frac{d\mathbf{f}^{(k-1)}}{d\mathbf{f}^{(k-2)}} \cdots \frac{d\mathbf{f}^{(0)}}{d\mathbf{x}} \\ &= d\mathbf{f}^{(k)} d\mathbf{f}^{(k-1)} \ldots d\mathbf{f}^{(0)} \end{aligned} \tag{25}$$

For programming consistency, in Eq. (24) we have included $\mathbf{x}$ itself as the final function in the composition. This $\mathbf{x}$ is a simple function that takes no input and returns $\mathbf{x}$. The total derivative of this function is $\frac{d\mathbf{x}}{d\mathbf{x}}$, which is the identity matrix. Then we can view chain rule terms as resulting from the function composition operators.

Single-layer Perceptron

The mathematical model for a Perceptron (consisting of a single-layer unless otherwise indicated) is a scalar function formed by composing an activation function σ with an affine function $\phi(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$.

$$\begin{aligned} f(\mathbf{x}) &= \sigma(\mathbf{w}^T\mathbf{x} + b) \\ &= \sigma \circ \phi_{(\mathbf{w},b)} \circ \mathbf{x} \end{aligned} \tag{26}$$

where we have indicated the parameters for the affine function with a subscript.

Historically, a step function was used for the activation σ. However this function is not differentiable, and so is generally not used in deep learning. We will assume an alternative choice of activation (one of the previous examples) is used instead.

In model optimization (i.e., training) contexts, we ultimately need derivatives with respect to the parameters in order to optimize them. We commonly indicate parameters in functions as $f(\mathbf{x}; \mathbf{w}, b)$ with a semicolon to separate the parameters from inputs. The domain of f in this notation is still $\mathbf{x}$, so we may still use the shorthand $d\mathbf{f} = \frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$, while we must explicitly write $\frac{\partial}{\partial \boldsymbol{\theta}}\mathbf{f}(\mathbf{x}; \Theta)$ for some parameter $\boldsymbol{\theta}$.

The derivatives for this function are

$$\frac{d}{d\mathbf{x}} f(\mathbf{x}; \mathbf{w}, b) = \frac{d\sigma(\phi)}{d\phi} \frac{d\phi(\mathbf{x})}{d\mathbf{x}} = \sigma' \cdot \mathbf{w} \tag{27}$$

$$\frac{d}{d\mathbf{w}} f(\mathbf{x}; \mathbf{w}, b) = \frac{d\sigma(\phi)}{d\phi} \frac{d\phi(\mathbf{x}; \mathbf{w}, b)}{d\mathbf{w}} = \sigma' \cdot \mathbf{x} \tag{28}$$

$$\frac{d}{db} f(\mathbf{x}; \mathbf{w}, b) = \frac{d\sigma(\phi)}{d\phi} \frac{d\phi(\mathbf{x}; \mathbf{w}, b)}{db} = \sigma' \tag{29}$$

Note that often we may not explicitly state the domain variable in terms such as σ', or df. But these are still functions and at implementation time we must input numbers to compute their values. For example, in the above, "σ'" is $\frac{d\sigma(\phi)}{d(\phi)}$ evaluated at $\phi = \mathbf{w}^T\mathbf{x} + b$, for a particular input $\mathbf{x}$ and parameters $\mathbf{w}$ and b. The back-propagation algorithm describes the efficient ordering of calculations for each function within a network.

Derivatives with respect to Tuples

At this point we have gone as far as we easily can using matrix calculus. To compute derivatives with respect to matrices would normally require reshaping operations and tricks like Kronecker products, or else full tensor index methods. Our approach will be to use tuples. For a general tuple, $\boldsymbol{\Theta} = [\theta_1, \theta_2, \dots \theta_N]$ and some vector-valued function $\mathbf{f}(\mathbf{x}; \boldsymbol{\Theta})$, we compute the derivative with respect to the tuple as simply the tuple of derivatives with respect to its elements

$$\frac{d\mathbf{f}}{d\boldsymbol{\Theta}} = \left[\frac{d\mathbf{f}}{d\theta_1}, \frac{d\mathbf{f}}{d\theta_2}, \dots, \frac{d\mathbf{f}}{d\theta_N}\right]. \tag{30}$$

Note that the elements of this tuple may be various kinds of structures, including even tuples themselves. If the elements are scalars, we'd expect (and see) that the math is effectively the same as if we replaced the tuple with a vector.

Derivatives with respect to Matrices

To differentiate with respect to matrices, we view the matrix as a tuple of column vectors. In this case, we will avoid a significant amount of additional mathematical structure (i.e., having to learn a lot of new notation and tricks) by sticking with our minimalist tuple approach. The derivative of a function with respect to a matrix will be defined as the tuple

$$\frac{d\mathbf{f}}{d\mathbf{A}} = \left[\frac{d\mathbf{f}}{d\mathbf{a}_1}, \frac{d\mathbf{f}}{d\mathbf{a}_2}, \dots, \frac{d\mathbf{f}}{d\mathbf{a}_N}\right] \tag{31}$$

where $\mathbf{a}_i$ refers to the ith column of $\mathbf{A}$.

As with vectors in Eq. (5), derivatives with respect to matrices are ultimately just an abstract description for multiple derivatives with respect to the scalar components of the matrix. So we can organize them as we choose, as long as we implement the proper scalar calculations ultimately when performing calculations.

The application of the chain rule here follows analogously. For our purposes, it will suffice to consider cases such as the following

$$\frac{d}{d\mathbf{A}}\mathbf{f}(\mathbf{g}(\mathbf{h}(\dots))) = \left[\frac{d}{d\mathbf{a}_1}\mathbf{f}(\mathbf{g}(\mathbf{h}(\dots))), \frac{d}{d\mathbf{a}_2}\mathbf{f}(\mathbf{g}(\mathbf{h}(\dots))), \dots, \frac{d}{d\mathbf{a}_N}\mathbf{f}(\mathbf{g}(\mathbf{h}(\dots)))\right] \tag{32}$$

where we separate the entire expression into a tuple of vector-derivatives and proceed using prior results.

Affine Function

Fine-grained connectivity within a neural network is typically described by affine functions such as

$$\boldsymbol{\phi}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{33}$$

The parameters consist of the weight matrix $\mathbf{W}$ and the bias vector $\mathbf{b}$, of size $M \times N$ and $M \times 1$, respectively. The total derivative of this function is the same as of the linear function $\mathbf{Wx}$, namely

$$\frac{d}{d\mathbf{x}}(\mathbf{W}\mathbf{x} + \mathbf{b}) = \frac{d\mathbf{W}\mathbf{x}}{d\mathbf{x}} = \mathbf{W} \tag{34}$$

The derivatives with respect to the parameters are $\frac{d}{d\mathbf{b}}$ and $\frac{d}{d\mathbf{W}}$. The former is simply the identity matrix,

$$\frac{d}{d\mathbf{b}}(\mathbf{W}\mathbf{x} + \mathbf{b}) = \frac{d\mathbf{b}}{d\mathbf{b}} = \mathbf{I} \tag{35}$$

The derivative with respect to $\mathbf{W}$ is a direct application of Eq. (31). If we write the affine function as a sum over columns of $\mathbf{W}$, as in

$$\boldsymbol{\phi}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} = \sum_{i=1}^{n} \mathbf{w}_i x_i + \mathbf{b} \tag{36}$$

then we can see that

$$\frac{d}{d\mathbf{w}_j}(\mathbf{W}\mathbf{x} + \mathbf{b}) = \frac{d}{d\mathbf{w}_j}\left(\sum_{i=1}^{N} \mathbf{w}_i x_i + \mathbf{b}\right) = \frac{d}{d\mathbf{w}_j}\mathbf{w}_j x_j = x_j\mathbf{I} \tag{37}$$

Plugging this into Eq. (31) we get,

$$\frac{d}{d\mathbf{W}}(\mathbf{W}\mathbf{x} + \mathbf{b}) = [x_1\mathbf{I}, x_2\mathbf{I}, \dots, x_N\mathbf{I}]. \tag{38}$$

Dense Layer

The Multi-Layer Perceptron (MLP) is a network model consisting of a composition of functions also known as "layers" with standard forms, primarily a vector-valued variant of the (single-layer) Perceptron from earlier. This is called a "dense" or "fully-connected" layer, as it has a weight for relating every input element to every output element.

$$\begin{aligned}\mathbf{n}^{(i)}(\mathbf{x}) &= \boldsymbol{\sigma}^{(i)}(\mathbf{W}^{(i)}\mathbf{x} + \mathbf{b}^{(i)}) \\ &= \boldsymbol{\sigma}^{(i)} \circ \boldsymbol{\phi}_{(\mathbf{W}^{(i)}, \mathbf{b}^{(i)})} \circ \mathbf{x}\end{aligned} \tag{39}$$

where $\mathbf{n}^{(i)}$ denotes the output of the i^{th} layer ("$\mathbf{n}$" here stands for "node"). The superscripts indicate that the parameters and activation function may very between layers. When in a network, the input would be $\mathbf{x} = \mathbf{n}^{(i-1)}$, but we can consider layers independently for now, and combine results later using the chain rule.

To optimize a network composed of such layers, we need the derivative of a layer with respect to its total input, and the derivative with respect to its parameters. The first can be computed by applying the chain rule from Eq. (25) to the composition of Eq. (39).

$$\frac{d\mathbf{n}^{(i)}}{d\mathbf{x}} = d\mathbf{n}^{(i)} = d\boldsymbol{\sigma}^{(i)}\, d\phi_{(\mathbf{W},\mathbf{b})} = d\boldsymbol{\sigma}^{(i)}\,\mathbf{W} \tag{40}$$

where $d\boldsymbol{\sigma}$ is the diagonal matrix given by Eq. (14). The derivative with respect to $\mathbf{b}$ is also simple,

$$\frac{d\mathbf{n}^{(i)}}{d\mathbf{b}} = d\boldsymbol{\sigma}^{(i)}\,\frac{d}{d\mathbf{b}}(\mathbf{W}\mathbf{x}+\mathbf{b}) = d\boldsymbol{\sigma}^{(i)} \tag{41}$$

The derivative with respect to $\mathbf{W}$ is a direct application of Eq. (32).

$$\frac{d\mathbf{n}^{(i)}}{d\mathbf{W}} = \left[\frac{d\mathbf{n}^{(i)}}{d\mathbf{w}_1}, \frac{d\mathbf{n}^{(i)}}{d\mathbf{w}_2}, \ldots \frac{d\mathbf{n}^{(i)}}{d\mathbf{w}_N}\right] = \left[x_1 d\boldsymbol{\sigma}^{(i)}, x_2 d\boldsymbol{\sigma}^{(i)}, \ldots x_N d\boldsymbol{\sigma}^{(i)}\right] \tag{42}$$

where we have used

$$\frac{d}{d\mathbf{w}_k}\boldsymbol{\sigma}(\mathbf{W}\mathbf{x}+\mathbf{b}) = \frac{d}{d\mathbf{w}_k}\boldsymbol{\sigma}\left(\sum_{j=1}^{N}\mathbf{w}_j x_j + \mathbf{b}\right) = d\boldsymbol{\sigma}\frac{d}{d\mathbf{w}_k}\mathbf{w}_k x_k = d\boldsymbol{\sigma}\,\mathbf{I}\,x_k \tag{43}$$

Loss Function Optimization

We now have almost all the tools we need to optimize a deep network. The final piece is the derivative of the chosen loss function, a simple scalar function of (in general) vector arguments, written as $L(\mathbf{f}(\mathbf{x}), \mathbf{y})$. To optimize a network we minimize this loss with respect to the parameters for some set of training data (a collection of input vectors $\mathbf{x}$), seeking the optimal choice of parameters. To do this using gradient descent, we update the weights using the derivative of the loss with respect to the parameters, e.g.,

$$\boldsymbol{\theta} \to \boldsymbol{\theta} - \mu\nabla_{\boldsymbol{\theta}} L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}), \mathbf{y}).$$

Where $\boldsymbol{\theta}$ represents the parameter we are optimizing, such as $\mathbf{W}^{(i)}$ or $\mathbf{b}^{(i)}$ for a particular layer i. We can treat the loss calculation as an additional function composition,

$$L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}), \mathbf{y}) = L \circ \mathbf{f}(\mathbf{x};\boldsymbol{\theta}) \tag{44}$$

$$\frac{d}{d\boldsymbol{\theta}}L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}), \mathbf{y}) = dL\frac{d}{d\boldsymbol{\theta}}\mathbf{f}(\mathbf{x};\boldsymbol{\theta}). \tag{45}$$

So we simply need to know the gradient of our loss function and use the chain rule to combine it with the gradients of our network layers, and we can optimize the network.

One common loss function is squared loss, for $\mathbf{y} = y \in \mathbf{R}$, used for predicting continuous-valued outputs, as in regression.

$$L(f(\mathbf{x}), y) = (f(\mathbf{x}) - y)^2 \tag{46}$$

As a scalar function with a scalar argument, the gradient here is particularly simple.

$$\frac{d}{d\boldsymbol{\theta}}(f(\mathbf{x};\boldsymbol{\theta}) - y)^2 = 2(f(\mathbf{x};\boldsymbol{\theta}) - y)\frac{d}{d\boldsymbol{\theta}}f(\mathbf{x};\boldsymbol{\theta}) \tag{47}$$

Another common choice is the Logistic loss, for $\mathbf{y} = y \in \{-1, +1\}$, used for predicting binary outputs as in classification.

$$L(f(\mathbf{x}), y) = \log\{1 + \exp(yf(\mathbf{x}))\} \tag{48}$$

For the derivative, we get

$$\frac{d}{d\boldsymbol{\theta}} \log\{1 + \exp(yf(\mathbf{x};\boldsymbol{\theta}))\} = \frac{1}{1+\exp(yf(\mathbf{x}))} \frac{d}{d\boldsymbol{\theta}} f(\mathbf{x};\boldsymbol{\theta}) \tag{49}$$

Full Network

A graph describing the compositions of functions in a loss calculation for a network is depicted in Fig. 1, where we have included nodes explicitly indicating the parameters, target (known) $\mathbf{y}$ value, and loss computation itself

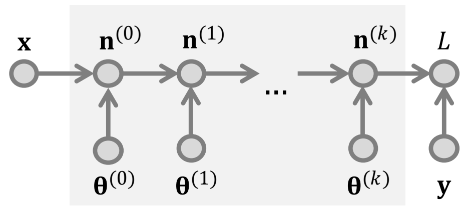

Figure 1: Graph representing deep feed-forward network loss calculation. The input sample is $\mathbf{x}$; the true label for the sample is $\mathbf{y}$; The nodes $\mathbf{n}^{(i)}$ represent layer functions, and $\boldsymbol{\theta}^{(i)}$ represent layer parameters. The shaded region is the feed-forward network, which computes an estimate of $\mathbf{y}$ given $\mathbf{x}$.

To compute the derivative with respect to the input $\mathbf{x}$, we apply the chain rule for the entire stack of layers,

$$\begin{aligned}\frac{dL}{d\mathbf{x}} &= \frac{dL}{d\mathbf{n}^{(k)}} \frac{d\mathbf{n}^{(k)}}{d\mathbf{n}^{(k-1)}} \frac{d\mathbf{n}^{(k-1)}}{d\mathbf{n}^{(k-2)}} \cdots \frac{d\mathbf{n}^{(2)}}{d\mathbf{n}^{(1)}} \frac{d\mathbf{n}^{(1)}}{d\mathbf{n}^{(0)}} \frac{d\mathbf{n}^{(0)}}{d\mathbf{x}} \\ &= \frac{dL}{d\mathbf{n}^{(k)}} \Big(\prod_{l=0}^{k} \frac{d\mathbf{n}^{(k-l+1)}}{d\mathbf{n}^{(k-l)}} \Big)\end{aligned} \tag{50}$$

To compute derivatives with respect to particular parameters, we only need use subsets of nodes. The graph indicates dependencies of each node, informing us of the terms needed in a chain rule calculation. For example, we can see from the graph that

$$\frac{dL}{d\boldsymbol{\theta}^{(k)}} = \frac{dL}{d\mathbf{n}^{(k)}} \frac{d\mathbf{n}^{(k)}}{d\boldsymbol{\theta}^{(k)}} + \frac{dL}{d\mathbf{y}} \frac{d\mathbf{y}}{d\boldsymbol{\theta}^{(k)}} = \frac{dL}{d\mathbf{n}^{(k)}} \frac{d\mathbf{n}^{(k)}}{d\boldsymbol{\theta}^{(k)}}, \tag{51}$$

using the fact that $\frac{d\mathbf{y}}{d\boldsymbol{\theta}^{(k)}} = \mathbf{0}$ as can be seen from the graph (since there's no dependency relating $\boldsymbol{\theta}^{(i)}$ to $\mathbf{y}$, and no further dependencies for $\mathbf{y}$ to consider). Applying the chain rule in this fashion to each subsequent dependency while discarding terms which do not depend on the variables of interest, we get the general result

$$\begin{aligned}\frac{dL}{d\boldsymbol{\theta}^{(i)}} &= \frac{dL}{d\mathbf{n}^{(k)}} \frac{d\mathbf{n}^{(k)}}{d\mathbf{n}^{(k-1)}} \frac{d\mathbf{n}^{(k-1)}}{d\mathbf{n}^{(k-2)}} \cdots \frac{d\mathbf{n}^{(i+2)}}{d\mathbf{n}^{(i+1)}} \frac{d\mathbf{n}^{(i+1)}}{d\mathbf{n}^{(i)}} \frac{d\mathbf{n}^{(i)}}{d\boldsymbol{\theta}^{(i)}} \\ &= \frac{dL}{d\mathbf{n}^{(k)}} \Big(\prod_{l=1}^{k-i} \frac{d\mathbf{n}^{(k-l+1)}}{d\mathbf{n}^{(k-l)}} \Big) \frac{d\mathbf{n}^{(i)}}{d\boldsymbol{\theta}^{(i)}}.\end{aligned} \tag{52}$$

For each dense layer, the parameters are $\mathbf{W}^{(i)}$ and $\mathbf{b}^{(i)}$. For the bias vector we use $\boldsymbol{\theta}^{(i)} = \mathbf{b}^{(i)}$ and plug Eq.

(41) into Eq. (52),

$$\frac{dL}{d\mathbf{b}^{(i)}} = \frac{dL}{d\mathbf{n}^{(k)}}\Big(\prod_{l=1}^{k-i}\frac{d\mathbf{n}^{(k-l+1)}}{d\mathbf{n}^{(k-l)}}\Big)d\boldsymbol{\sigma}^{(i)} = \mathbf{u}_i^T d\boldsymbol{\sigma}^{(i)}. \tag{53}$$

where we have defined

$$\mathbf{u}_i^T = \frac{dL}{d\mathbf{n}^{(k)}}\Big(\prod_{l=1}^{k-i}\frac{d\mathbf{n}^{(k-l+1)}}{d\mathbf{n}^{(k-l)}}\Big).$$

Note that since $d\boldsymbol{\sigma}^{(i)}$ is a diagonal matrix. the product $\mathbf{u}^T d\boldsymbol{\sigma}^{(i)}$ is simply an element-wise product between $\mathbf{u}_i$ and the diagonal of $d\boldsymbol{\sigma}^{(i)}$. We call the latter vector $\mathbf{d}_\sigma$ with elements $(\mathbf{d}_\sigma)_i = \frac{\partial\sigma(x)}{\partial x}$.

Therefore we have, for the gradient,

$$\nabla_{\mathbf{b}^{(i)}}L = \Big(\frac{dL}{d\mathbf{b}^{(i)}}\Big)^T = d\boldsymbol{\sigma}^{(i)}\mathbf{u}_i = \mathbf{d}_\sigma \circ \mathbf{u}_i.$$

For the weight matrix term we use $\boldsymbol{\theta}^{(i)} = \mathbf{W}^{(i)}$. We must also utilize Eqs. (30) and (42),

$$\begin{aligned}\frac{dL}{d\mathbf{W}^{(i)}} &= \left[\frac{dL}{d\mathbf{w}_1^{(i)}}, \frac{dL}{d\mathbf{w}_2^{(i)}}, \dots, \frac{dL}{d\mathbf{w}_N^{(i)}}\right] \\ &= \left[\frac{dL}{d\mathbf{n}^{(k)}}\Big(\prod_{l=1}^{k-i}\frac{d\mathbf{n}^{(k-l+1)}}{d\mathbf{n}^{(k-l)}}\Big)\frac{d\mathbf{n}^{(i)}}{d\mathbf{w}_1}, \dots, \frac{dL}{d\mathbf{n}^{(k)}}\Big(\prod_{l=1}^{k-i}\frac{d\mathbf{n}^{(k-l+1)}}{d\mathbf{n}^{(k-l)}}\Big)\frac{d\mathbf{n}^{(i)}}{d\mathbf{w}_N}\right] \\ &= \left[x_1\mathbf{u}_i^T d\boldsymbol{\sigma}^{(i)}, \dots, x_N\mathbf{u}_i^T d\boldsymbol{\sigma}^{(i)}\right]. \end{aligned} \tag{54}$$

Forming the gradient and simplifying,

$$\begin{aligned}\nabla_{\mathbf{W}^{(i)}}L = \Big(\frac{dL}{d\mathbf{W}^{(i)}}\Big)^T &= \left[\Big(x_1\mathbf{u}_i^T d\boldsymbol{\sigma}^{(i)}\Big)^T, \dots, \Big(x_N\mathbf{u}_i^T d\boldsymbol{\sigma}^{(i)}\Big)^T\right] \\ &= \left[x_1 d\boldsymbol{\sigma}^{(i)}\mathbf{u}_i, \dots, x_N d\boldsymbol{\sigma}^{(i)}\mathbf{u}_i\right] \\ &= (\mathbf{d}_\sigma \circ \mathbf{u}_i)\mathbf{x}^T. \end{aligned} \tag{55}$$

The first step in Eq. (55) uses the fact that the transpose here applies to the elements of the tuple. The last step uses the fact that the tuple of vectors that results here can be described by a matrix, in this case an outer product.

Summary

We computed gradients for optimizing a network composed of dense layers with arbitrary activation functions. We used tuples for the problematic step of computing gradients with respect to weight matrices, which allowed us avoid flattening matrices or reshaping vectors, or delving into new branches of mathematics like tensor algebra. The benefit of an abstract description is that it allows one to use abstract rules of manipulation without needing to consider the internal workings of the abstract data structure. E.g., we can add vectors rather than having to consider how to add each element. However in the application here, this perspective from lower-level principles is ultimately necessary for efficient implementation. So we had to think through the meaning for our problem of each operation with tuples, such as derivative with respect to a tuple, and transpose of a tuple of vectors.

A few more common extensions are needed which are trivial extensions of the same approach. For example, batch-training means the input $\mathbf{x}$ must be treated as a tuple as well. Alternatively, layers often have multiple channels which would be represented by tuples of weight matrices. Inclusion of these extensions would multiply the sizes of equations here, but often amount to no more than tuples of the same form we have, without needing further simplifications or optimization of the calculations are needed. The only remaining optimization in a general one, to reuse terms which are repeated in multiple calculations. For example, the derivatives with respect to the loss and with respect to each node. Such considerations are typically incorporated into the backpropagation algorithm.

References

[1] Martín Abadi. TensorFlow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 1, New York, NY, USA, September 2016. Association for Computing Machinery.

[2] Brett W. Bader and Tamara G. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, December 2006.

[3] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. Number: 7825 Publisher: Nature Publishing Group.

[4] Charles F. Van Loan. The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, 123(1):85–100, November 2000.

[5] Terence Parr and Jeremy Howard. The Matrix Calculus You Need For Deep Learning, July 2018. arXiv:1802.01528 [cs, stat].

[6] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[7] D.S.G. Pollock. On Kronecker products, tensor products and matrix differential calculus. *International Journal of Computer Mathematics*, 90(11):2462–2476, November 2013. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/00207160.2013.783696.