Back-propagation Learning in Multilayer Perceptron Neural Networks

Keith M. Chugg, Antonio Ortega
version 0.3

March 31, 2023

This document derives the back-propagation learning algorithm for neural networks. In particular we focus on the case of Multilayer Perceptrons (MLPs) where each layer in the network is a fully-connected (or dense) layer. Back-propagation learning is really just the repeated application of the chain rule for derivatives from calculus. Specifically, each parameter w is the neural network is updated by $-\eta\frac{\partial C}{\partial w}$ at each iteration, where C is the loss, or loss, function. Thus, the entire algorithm comprises the steps and recursions required to calculate this partial derivative for each trainable parameter in the network. It is very useful to express these steps/recursions in vector-matrix form so we start with a summary of the required vector calculus results.

1 Vector Calculus Preliminaries

Vector calculus is simply a set of book-keeping conventions for tracking many related partial derivatives in a compact notation. The use of vector calculus notation is entirely optional for deriving and implementing back-propagation learning, but it substantially simplifies both and therefore we adopt it herein.

We are familiar with the gradient of a function $f(\mathbf{x})$ that maps a vector valued argument to a scalar:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \frac{\partial f(\mathbf{x})}{\partial x_3} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_M} \end{bmatrix} \tag{1}$$

Note that for $\mathbf{x} \in \mathbb{R}^M$, the gradient $\nabla_{\mathbf{x}} f(\mathbf{x}) \in \mathbb{R}^M$ is a column vector.

The gradient can be viewed as the derivative of a scalar valued function with respect to a vector argument. To derive backprop learning, we desire to extend this notion to the derivative of a vector-valued function with respect to a vector argument. Again, we emphasize that this is simply all partial derivatives involved, so it is more accurate to say that we seek a book-keeping convention for storing these partial derivatives. For example consider the vector-valued function

$\mathbf{f}(\mathbf{x})$ that maps $\mathbb{R}^M$ into $\mathbb{R}^K$

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ f_3(\mathbf{x}) \\ \vdots \\ f_K(\mathbf{x}) \end{bmatrix} \tag{2}$$

where each $f_k(\mathbf{x})$ maps $\mathbb{R}^M$ into $\mathbb{R}$. We would like to compute all of the partial derivatives $\left\{ \frac{\partial f_k(\mathbf{x})}{\partial x_m} \right\}_{k,m}$. These are naturally stored in a matrix which can be thought of as $\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}}$. However, there are two conventions for doing so. In the *denominator convention* this is considered a $(M \times K)$ matrix with $(m,k)^{\text{th}}$ entry $\frac{\partial f_k(\mathbf{x})}{\partial x_m}$. In the *numerator convention* this is considered a $(K \times M)$ matrix with $(k,m)^{\text{th}}$ entry $\frac{\partial f_k(\mathbf{x})}{\partial x_m}$. Clearly, these two conventions are related by a simple transpose. While each of these conventions has desirable properties, **we adopt the denominator convention**.

For this denominator convention, for a scalar-valued function we have

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \nabla_{\mathbf{x}} f(\mathbf{x}) \tag{3}$$

and for a vector-valued function we have

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \nabla_{\mathbf{x}} f_1(\mathbf{x}) & \nabla_{\mathbf{x}} f_2(\mathbf{x}) & \cdots & \nabla_{\mathbf{x}} f_K(\mathbf{x}) \end{bmatrix} \tag{4}$$

$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \frac{\partial f_2(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_K(\mathbf{x})}{\partial x_1} \\ \frac{\partial f_1(\mathbf{x})}{\partial x_2} & \frac{\partial f_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_K(\mathbf{x})}{\partial x_2} \\ \frac{\partial f_1(\mathbf{x})}{\partial x_3} & \frac{\partial f_2(\mathbf{x})}{\partial x_3} & \cdots & \frac{\partial f_K(\mathbf{x})}{\partial x_3} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_1(\mathbf{x})}{\partial x_M} & \frac{\partial f_2(\mathbf{x})}{\partial x_M} & \cdots & \frac{\partial f_K(\mathbf{x})}{\partial x_M} \end{bmatrix} \tag{5}$$

1.1 Useful Relations

With the denominator convention adopted above, we have

$$\frac{\partial (\mathbf{A}\mathbf{x})}{\partial \mathbf{x}} = \mathbf{A}^T \tag{6}$$

with special case

$$\frac{\partial (\mathbf{b}^T \mathbf{x})}{\partial \mathbf{x}} = \mathbf{b} \tag{7}$$

Also, the key identity used in the derivation of backprop is the *right-to-left chain-rule*[1], that is associated with the denominator convention. Specifically, let $\mathbf{f} : \mathbb{R}^M \to \mathbb{R}^K$ and $\mathbf{g} : \mathbb{R}^K \to \mathbb{R}^L$, then

$$\underbrace{\frac{\partial \mathbf{g}(\mathbf{f}(\mathbf{x}))}{\partial \mathbf{x}}}_{(M \times L)} = \underbrace{\frac{\partial \mathbf{f}}{\partial \mathbf{x}}}_{(M \times K)} \underbrace{\frac{\partial \mathbf{g}}{\partial \mathbf{f}}}_{(K \times L)} \tag{8}$$

[1] In the numerator convention, the chain rule is left-to-right and all the vector derivatives are transposes of those obtained with the denominator convention.

Identities: vector-by-vector $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$

Condition	Expression	Numerator layout, i.e. by y and $\mathbf{x}^\mathsf{T}$	Denominator layout, i.e. by $\mathbf{y}^\mathsf{T}$ and x
a is not a function of **x**	$\frac{\partial \mathbf{a}}{\partial \mathbf{x}} =$	$\mathbf{0}$	
	$\frac{\partial \mathbf{x}}{\partial \mathbf{x}} =$	$\mathbf{I}$	
A is not a function of **x**	$\frac{\partial \mathbf{A}\mathbf{x}}{\partial \mathbf{x}} =$	$\mathbf{A}$	$\mathbf{A}^\top$
A is not a function of **x**	$\frac{\partial \mathbf{x}^\top \mathbf{A}}{\partial \mathbf{x}} =$	$\mathbf{A}^\top$	$\mathbf{A}$
a is not a function of **x**, **u** = **u**(**x**)	$\frac{\partial a\mathbf{u}}{\partial \mathbf{x}} =$	$a\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$	
v = v(**x**), **a** is not a function of **x**	$\frac{\partial v\mathbf{a}}{\partial \mathbf{x}} =$	$\mathbf{a}\frac{\partial v}{\partial \mathbf{x}}$	$\frac{\partial v}{\partial \mathbf{x}}\mathbf{a}^\top$
v = v(**x**), **u** = **u**(**x**)	$\frac{\partial v\mathbf{u}}{\partial \mathbf{x}} =$	$v\frac{\partial \mathbf{u}}{\partial \mathbf{x}} + \mathbf{u}\frac{\partial v}{\partial \mathbf{x}}$	$v\frac{\partial \mathbf{u}}{\partial \mathbf{x}} + \frac{\partial v}{\partial \mathbf{x}}\mathbf{u}^\top$
A is not a function of **x**, **u** = **u**(**x**)	$\frac{\partial \mathbf{A}\mathbf{u}}{\partial \mathbf{x}} =$	$\mathbf{A}\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$	$\frac{\partial \mathbf{u}}{\partial \mathbf{x}}\mathbf{A}^\top$
u = **u**(**x**), **v** = **v**(**x**)	$\frac{\partial (\mathbf{u}+\mathbf{v})}{\partial \mathbf{x}} =$	$\frac{\partial \mathbf{u}}{\partial \mathbf{x}} + \frac{\partial \mathbf{v}}{\partial \mathbf{x}}$	
u = **u**(**x**)	$\frac{\partial \mathbf{g}(\mathbf{u})}{\partial \mathbf{x}} =$	$\frac{\partial \mathbf{g}(\mathbf{u})}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$	$\frac{\partial \mathbf{u}}{\partial \mathbf{x}}\frac{\partial \mathbf{g}(\mathbf{u})}{\partial \mathbf{u}}$
u = **u**(**x**)	$\frac{\partial \mathbf{f}(\mathbf{g}(\mathbf{u}))}{\partial \mathbf{x}} =$	$\frac{\partial \mathbf{f}(\mathbf{g})}{\partial \mathbf{g}}\frac{\partial \mathbf{g}(\mathbf{u})}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$	$\frac{\partial \mathbf{u}}{\partial \mathbf{x}}\frac{\partial \mathbf{g}(\mathbf{u})}{\partial \mathbf{u}}\frac{\partial \mathbf{f}(\mathbf{g})}{\partial \mathbf{g}}$

Figure 1: Table of derivatives of vector quantities with respect to vector variables taken from the Wikipedia page for matrix calculus. In this document we adopt the denominator layout convention. This Wikipedia page uses bold lower case for vectors and bold upper case for matrices. The chain rule is summarized in the last two entries.

These book-keeping conventions can be extended to define, for example, the derivative of a vector-valued function with respect to matrix variable. Surprisingly, one of the best references for this is the Wikipedia page for matrix calculus. A table of relations for vector derivatives of vector-valued functions from this page is shown in Figure 1.

1.2 Vectorized Scalar Functions

One special case of the above is a vector-valued function that is a "vectorized scalar function." This is a vector valued function that is obtained by applying the same scalar-valued function to each component of a vector argument

$$\mathbf{f}(\mathbf{x}) = \underline{f}(\mathbf{x}) = \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_M) \end{bmatrix} \tag{9}$$

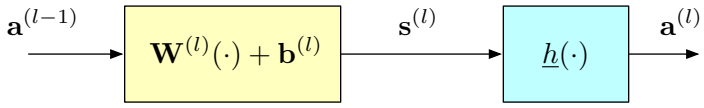

Figure 2: The processing for layer l where a vectorized scalar function has been assumed for the activation function.

where we have introduced the notation $\underline{f}(\mathbf{x})$ to denote this special case of a vector valued function. Note that this is the special case of (2) with $f_k(\mathbf{x}) = f(x_k)$. For the special case of a vectorized scalar function we have

$$\frac{\partial \underline{f}(\mathbf{x})}{\partial \mathbf{x}} = \text{diag}\left(\frac{d}{dx_1}f(x_1), \frac{d}{dx_2}f(x_2), \ldots \frac{d}{dx_M}f(x_M)\right) = \mathbf{diag}\left(\underline{\dot{f}}(\mathbf{x})\right) \tag{10}$$

where diag$(\cdot)$ indicates a diagonal matrix with the specified diagonal elements and $\underline{\dot{f}}(\mathbf{x})$ is the vectorization of the scalar function $\dot{f}(v) = \frac{df(v)}{dv}$.

As a special case of the chain rule in (8) when $\mathbf{y} = \mathbf{g}(\mathbf{f}(\mathbf{x})) = \mathbf{g}(\underline{f}(\mathbf{x}))$ – *i.e.,* when $\mathbf{f}(\cdot) = \underline{f}(\cdot)$ is a vectorized scalar function is

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \underline{f}}{\partial \mathbf{x}}\frac{\partial \mathbf{g}}{\partial \underline{f}} = \mathbf{diag}\left(\underline{\dot{f}}(\mathbf{x})\right)\frac{\partial \mathbf{g}}{\partial \underline{f}} \tag{11}$$

2 Notation for MLP Processing

The processing in the l^{th} layer of a multilayer perceptron can be expressed in vector form as

$$\mathbf{a}^{(l)} = \underline{h}\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right) = \underline{h}\left(\mathbf{s}^{(l)}\right) \quad l = 1, 2, \ldots L \tag{12}$$

where $\mathbf{a}^{(l)}$ is the *activation* for layer l. Denoting the number of neurons (nodes) at layer l as M_l, the weight matrix $\mathbf{W}^{(l)}$ is $(M_l \times M_{l-1})$, and the bias vector $\mathbf{b}^{(l)}$ is $(M_l \times 1)$. The activation function $h(\cdot)$ is vectorized to $\underline{h}(\cdot)$ as described in Section 1.2. This is shown in Figure 2. The *linear activation* or *pre-activation* for layer l is implictly defined in (12) as

$$\mathbf{s}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{13}$$

The processing in (12) is initialized with $\mathbf{a}^{(0)} = \mathbf{x}$ where $\mathbf{x}$ is the network input (non-augmented). The output of the network is $\mathbf{a}^{(L)}$ for an L-layer network. Specifically, for a regression problem the final layer will typically have a linear activation and $\hat{\mathbf{y}}(\mathbf{x}) = \mathbf{a}^{(L)} = \mathbf{s}^{(L)}$. For a classification problem, the last layer typically contains a SoftMax operation and $\hat{\mathbf{p}} = \mathbf{a}^{(L)} = \text{SoftMax}(\mathbf{s}^{(L)})$ is the network's predictions of the class membership of the network input $\mathbf{x}$.[2]

[2]Note in the case of a SoftMax output layer, the activation is not a vectorized scalar function. Specifically, each coordinate of the SoftMax$(\mathbf{s})$ function is a function of every component of $\mathbf{s}$. This is discussed in detail in Section 3.3.2.

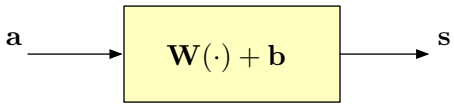

Figure 3: The mapping from activation to pre-activation for a given layer.

3 Back-Propagation Algorithm

In Section 2 we have defined the notation for processing a single input vector $\mathbf{x}$ to a single network output vector $\mathbf{a}^{(L)}$. In this section we will consider a single data-point gradient descent update for this model. In other words, we consider the input $\mathbf{x}_n$ and compute $\frac{\partial C_n}{\partial \theta}$ for every trainable parameter θ in the network – *e.g.,* θ can be one of the elements of $\mathbf{W}^{(l)}$ or $\mathbf{b}^{(l)}$ for each layer $l \in \{1, 2, \dots L\}$. Here, the total loss function is $C = \sum_{n=1}^{N} C_n$ and $C_n = C_n(\mathbf{x}_n; \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L})$. For notational simplicity, let us consider in this section a single input $\mathbf{x} = \mathbf{x}_n$ and denote the associated loss contribution as $C = C_n$. We generalize this to batch-based backprop learning in the section that follows.

The back-propagation learning algorithm is a "backward" recursion on the gradient of the loss with respect to the layer pre-activations – *i.e.,* computing $\nabla_{\mathbf{s}^{(l-1)}} C$ from $\nabla_{\mathbf{s}^{(l)}} C$. Specifically, defining

$$\boldsymbol{\delta}^{(l)} \triangleq \nabla_{\mathbf{s}^{(l)}} C \tag{14}$$

we seek a recursion computing $\boldsymbol{\delta}^{(l-1)}$ from $\boldsymbol{\delta}^{(l)}$. With this recursion, we only need two other components for backprop learning: (i) an initialization for $\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{s}^{(L)}} C$ and (ii) a way to express the partial derivatives of all trainable parameters in terms of $\boldsymbol{\delta}^{(l)}$. Let us consider each of these three components of backprop in the following subsections.

3.1 Computing GD Updates from Deltas

Assume that we have available $\boldsymbol{\delta}^{(l)} = \nabla_{\mathbf{s}^{(l)}} C$ at each layer and we wish to compute the gradient descent updates for the bias vectors and weight matrices at each layer. To simplify the notation, let us drop the index l and focus on the mapping from activation $\mathbf{a}$ to pre-activation $\mathbf{s}$ as shown in Figure 3. Here the activation from the previous layer $\mathbf{a}$ is mapped to the pre-activation for the current layer via

$$\mathbf{s} = \mathbf{W}\mathbf{a} + \mathbf{b} \tag{15}$$

First, let us compute the gradient of the loss with respect to the bias vector. Using the right-to-left chain rule we have

$$\nabla_{\mathbf{b}} C = \frac{\partial C}{\partial \mathbf{b}} = \frac{\partial \mathbf{s}}{\partial \mathbf{b}} \frac{\partial C}{\partial \mathbf{s}} = \mathbf{I}\boldsymbol{\delta} = \boldsymbol{\delta} \tag{16}$$

where $\boldsymbol{\delta} = \nabla_{\mathbf{s}} C$.

To compute the GD updates for the matrix $\mathbf{W}$, we write out the expression in (15) for each component

$$s_m = \mathbf{w}_m^T \mathbf{a} + b_m \tag{17}$$

where b_m is the m^{th} element of the vector $\mathbf{b}$ and $\mathbf{w}_m^T$ is the m^{th} row of $\mathbf{W}$

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \mathbf{w}_3^T \\ \vdots \\ \mathbf{w}_M^T \end{bmatrix} \tag{18}$$

The gradient of C with respect to the $\mathbf{w}_m$ is

$$\frac{\partial C}{\partial \mathbf{w}_m} = \frac{\partial s_m}{\partial \mathbf{w}_m}\frac{\partial C}{\partial s_m} = \mathbf{a}\delta_m \tag{19}$$

It follows that the GD update for the m^{th} row of $\mathbf{W}$ is

$$\mathbf{w}_m^T(i+1) = \mathbf{w}_m^T(i) - \eta(i)\delta_m\mathbf{a}^T \tag{20}$$

Using this with (18) implies

$$\mathbf{W}(i+1) = \mathbf{W}(i) - \eta(i)\boldsymbol{\delta}\mathbf{a}^T \tag{21}$$

Using (16) and (21) and also identifying the layer indices from Figure 2 yields the method to use the $\boldsymbol{\delta}^{(l)}$ value to update the layer l weights and biases

$$\mathbf{W}^{(l)}(i+1) = \mathbf{W}^{(l)}(i) - \eta(i)\boldsymbol{\delta}^{(l)}\left(\mathbf{a}^{(l-1)}\right)^T \tag{22a}$$

$$\mathbf{b}^{(l)}(i+1) = \mathbf{b}^{(l)}(i) - \eta(i)\boldsymbol{\delta}^{(l)} \tag{22b}$$

3.2 The Delta Recursion

In order to get a recursion on $\boldsymbol{\delta}^{(l)} = \nabla_{\mathbf{s}^{(l)}}C$, consider the mapping from $\mathbf{s}^{(l-1)}$ to $\mathbf{s}^{(l)}$ as shown in Figure 4. The desired recursion follows from the right-to-left chain rule

$$\boldsymbol{\delta}^{(l-1)} = \frac{\partial C}{\partial \mathbf{s}^{(l-1)}} \tag{23}$$

$$= \frac{\partial \mathbf{a}^{(l-1)}}{\partial \mathbf{s}^{(l-1)}}\frac{\partial \mathbf{s}^{(l)}}{\partial \mathbf{a}^{(l-1)}}\frac{\partial C}{\partial \mathbf{s}^{(l)}} \tag{24}$$

$$= \text{diag}\left(\dot{\underline{h}}(\mathbf{s}^{(l-1)})\right)\left(\mathbf{W}^{(l)}\right)^T\boldsymbol{\delta}^{(l)} \tag{25}$$

$$= \dot{\underline{h}}(\mathbf{s}^{(l-1)}) \odot \left[\left(\mathbf{W}^{(l)}\right)^T\boldsymbol{\delta}^{(l)}\right] \tag{26}$$

$$= \dot{\mathbf{a}}^{(l-1)} \odot \left[\left(\mathbf{W}^{(l)}\right)^T\boldsymbol{\delta}^{(l)}\right] \tag{27}$$

where we have used (6) and (11) and implicitly defined

$$\dot{\mathbf{a}}^{(l)} \triangleq \dot{\underline{h}}(\mathbf{s}^{(l)}) \tag{28}$$

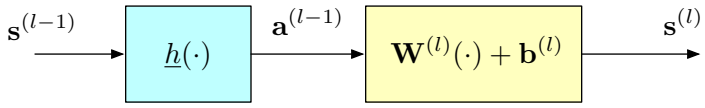

Figure 4: The mapping from pre-activation to pre-activation across a layer.

which are the *derivative activations* obtained by passing the pre-activations through the vectorized function $\dot{h}(\cdot)$ – *i.e.,* the derivative of the activation function. Also implicitly defined in (27) is the Hadamard (or element-wise) product of two vectors

$$\mathbf{x} \odot \mathbf{y} \stackrel{\Delta}{=} \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ x_3 y_3 \\ \vdots \\ x_m y_m \end{bmatrix} \tag{29}$$

3.3 Initializing the Delta Recursion

In the previous two subsections, we have derived a backward recursion on $\boldsymbol{\delta}^{(l)}$ and shown how $\boldsymbol{\delta}^{(l)}$ can be mapped to the GD updates for the layer weight matrix and bias vector. All that remains is to determine how to initialize the delta recursion. Specifically, we need an expression for $\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{s}^{(L)}} C$. This follows again by the right-to-left chain rule to the processing shown in Figure 5. Specifically,

$$\begin{aligned} \boldsymbol{\delta}^{(L)} &= \nabla_{\mathbf{s}^{(L)}} C && (30)\\ &= \frac{\partial C}{\partial \mathbf{s}^{(L)}} && (31)\\ &= \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial C}{\partial \mathbf{a}^{(L)}} && (32)\\ &= \dot{\mathbf{a}}^{(L)} \odot \nabla_{\mathbf{a}^{(L)}} C && (33) \end{aligned}$$

where it has been assumed that the final layer activation in a vectorized scalar function. Note that $\nabla_{\mathbf{a}^{(L)}} C$ is the standard gradient of the loss function with respect to the model prediction – *i.e.,* this is determined by the functional form of the loss function.

3.3.1 Special Case: MSE loss

For the MSE loss function, we have $C = \frac{1}{2}\|\mathbf{a}^{(L)} - \mathbf{y}\|^2$ where, for the typical case, $\mathbf{y}$ is the target vector for a multi-dimensional target regression problem. It follows that

$$\nabla_{\mathbf{a}^{(L)}} C = (\mathbf{a}^{(L)} - \mathbf{y}) = (\hat{\mathbf{y}} - \mathbf{y}) \tag{34}$$

Thus, for MSE regression, we have

$$\boldsymbol{\delta}^{(L)} = \dot{\mathbf{a}}^{(L)} \odot (\mathbf{a}^{(L)} - \mathbf{y}) \tag{35}$$

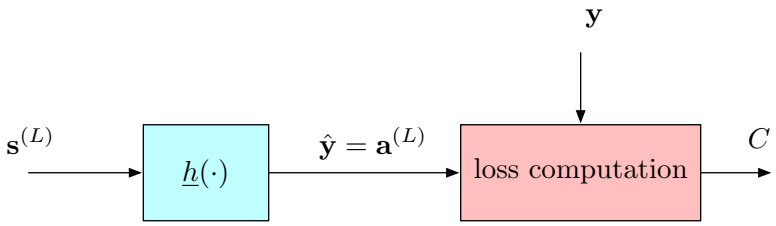

Figure 5: The computation of the loss function from the final layer output. A vectorized scalar activation function has been assumed for the final layer.

and for MSE regression, typically the final layer activation is linear so that $\hat{\mathbf{y}} = \mathbf{a}^{(L)} = \mathbf{s}^{(L)}$ and $\dot{h}(v) = 1$. If follows that for MSE loss and a linear final layer activation

$$\boldsymbol{\delta}^{(L)} = (\hat{\mathbf{y}} - \mathbf{y}) \tag{36}$$

3.3.2 Special Case: SoftMax with Multiclass Cross-Entropy Loss

Another special case of interest is when the final layer has SoftMax activation and multiclass cross-entropy loss is used. Specifically, when

$$\hat{\mathbf{p}} = \mathbf{a}^{(L)} = \text{SoftMax}(\mathbf{s}^{(L)}) \tag{37}$$

and

$$C = \text{MCE}(\mathbf{p}, \hat{\mathbf{p}}) = -\sum_{m=1}^{C} p_m \ln(\hat{p}_m) \tag{38}$$

where p_m is the label probability for class m. In most cases $p_m = 1$ for the true class and $p_m = 0$ for all other classes. In the case of SoftMax output activation, the final activation is not a vectorized scalar function, as is illustrated in Figure 6. Again, applying the right-to-left chain rule, we obtain

$$\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{s}^{(L)}} C \tag{39}$$

$$= \frac{\partial C}{\partial \mathbf{s}^{(L)}} \tag{40}$$

$$= \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial C}{\partial \mathbf{a}^{(L)}} \tag{41}$$

where now $\frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{s}^{(L)}}$ is a non-diagonal matrix. In the case of the SoftMax output activation, this is a $(C \times C)$ matrix, where C is the number of classes.

For the special case of SoftMax final activation and multiclass cross-entropy loss, it can be shown that

$$\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - \mathbf{p} = \hat{\mathbf{p}} - \mathbf{p} \tag{42}$$

so that the delta recursion is initialized with the difference between the model prediction and the label/target vector.

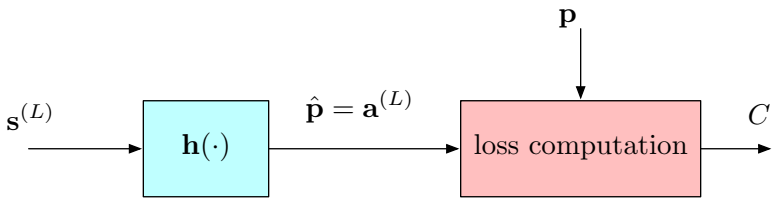

Figure 6: The computation of the loss function from the final layer output where the nonlinear activation of the final layer is assumed to be a general vector-valued function.

3.4 Single Data-point Back-Propagation Learning Summary

In summary, for a single input $\mathbf{a}^{(0)} = \mathbf{x}$ and output $\mathbf{a}^{(L)}$, the back-propagation learning algorithm is

$$\mathbf{a}^{(l)} = \underline{h}\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right) \quad \text{Layer Activation} \quad (43a)$$

$$\dot{\mathbf{a}}^{(l)} = \underline{\dot{h}}\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right) \quad \text{Layer Derivative Activation} \quad (43b)$$

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{s}^{(L)}} \nabla_{\mathbf{a}^{(L)}} C \quad \text{Delta Recursion Initialization} \quad (43c)$$

$$\boldsymbol{\delta}^{(l-1)} = \dot{\mathbf{a}}^{(l-1)} \odot \left[\left(\mathbf{W}^{(l)}\right)^T \boldsymbol{\delta}^{(l)}\right] \quad \text{Delta Recursion} \quad (43d)$$

$$\mathbf{W}^{(l)}(i+1) = \mathbf{W}^{(l)}(i) - \eta(i)\boldsymbol{\delta}^{(l)}\left(\mathbf{a}^{(l-1)}\right)^T \quad \text{Weight Matrix GD Update} \quad (43e)$$

$$\mathbf{b}^{(l)}(i+1) = \mathbf{b}^{(l)}(i) - \eta(i)\boldsymbol{\delta}^{(l)} \quad \text{Bias Vector GD Update} \quad (43f)$$

4 Extensions and Discussion

In the previous backprop development, we considered only a single data-point $\mathbf{x}$ along with the associated target/label. Consider the case where mini-batches (or full-batch) are used. In this case, we compute (43a) - (43d) for each data point and then average the updates in (43e) and (43f) over the mini-batch. Specifically, let the data be $\{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N_{\text{train}}}$ and denote the result of using the the

n^{th} data point by $(\cdot)_n$. Translating the above description to equations with this convention yields[3]

$$\mathbf{a}_n^{(l)} = \underline{h}\left(\mathbf{W}^{(l)}\mathbf{a}_n^{(l-1)} + \mathbf{b}^{(l)}\right) \quad \text{Layer Activations} \quad (44a)$$

$$\dot{\mathbf{a}}_n^{(l)} = \underline{\dot{h}}\left(\mathbf{W}^{(l)}\mathbf{a}_n^{(l-1)} + \mathbf{b}^{(l)}\right) \quad \text{Layer Derivative Activations} \quad (44b)$$

$$\boldsymbol{\delta}_n^{(L)} = \frac{\partial \mathbf{a}_n^{(L)}}{\partial \mathbf{s}_n^{(L)}} \nabla_{\mathbf{a}_n^{(L)}} C_n \quad \text{Delta Recursion Initializations} \quad (44c)$$

$$\boldsymbol{\delta}_n^{(l-1)} = \dot{\mathbf{a}}_n^{(l-1)} \odot \left[\left(\mathbf{W}^{(l)}\right)^T \boldsymbol{\delta}_n^{(l)}\right] \quad \text{Delta Recursions} \quad (44d)$$

$$\mathbf{W}^{(l)}(i+1) = \mathbf{W}^{(l)}(i) - \eta(i)\left[\frac{1}{|\mathcal{B}_i|}\sum_{n\in\mathcal{B}_i} \boldsymbol{\delta}_n^{(l)}\left(\mathbf{a}_n^{(l-1)}\right)^T\right] \quad \text{Weight Matrix GD Update} \quad (44e)$$

$$\mathbf{b}^{(l)}(i+1) = \mathbf{b}^{(l)}(i) - \eta(i)\left[\frac{1}{|\mathcal{B}_i|}\sum_{n\in\mathcal{B}_i} \boldsymbol{\delta}_n^{(l)}\right] \quad \text{Bias Vector GD Update} \quad (44f)$$

where $\mathcal{B}_i$ is the set of data indices for the i^{th} mini-batch and $|\mathcal{B}_i|$ is the number of data points in this mini-batch.

Another extension is the use of weight regularization. For example, if L2 regularization of the weights and biases is employed, with regularizer coefficeint λ, then the weight updates in (43e) and bias updates in (43f) are modified to

$$\mathbf{W}^{(l)}(i+1) = \mathbf{W}^{(l)}(i) - \eta(i)\left(\boldsymbol{\delta}^{(l)}\left(\mathbf{a}^{(l-1)}\right)^T + 2\lambda\mathbf{W}^{(l)}(i)\right) \quad (45a)$$

$$\mathbf{b}^{(l)}(i+1) = \mathbf{b}^{(l)}(i) - \eta(i)\left(\boldsymbol{\delta}^{(l)} + 2\lambda\mathbf{b}^{(l)}(i)\right) \quad (45b)$$

Finally, there are other types of layers used in modern neural networks. For example, convolutional layers, recurrent layers, and attention layers. the backprop equations for networks with these types of layers can be computed using the same principles employed herein. Specifically, the key step is to get a recursion on the pre-activations. In practice, most of the heavy lifting is done for us by auto-differentiation routines available in frameworks such as TensorFlow and PyTorch and one does not need to directly implement the backprop equations. However, for some applications (*e.g.,* imbedded programming, digital circuit acceleration) direct implementation of backprop using the equations derived herein is required.

[3]These updates can be vectorized over the variable n so that instead of looping over $|\mathcal{B}_i|$ matrix vector multiplications, one can do one matrix-matrix multiplication.