Training a Multilayer Neural Network
with Stochastic Gradient Descent

Landon Buell

27 January 2020

1 Introduction

Neural Networks are inspired from architecture of the human brain. With their first conception in the 1940's, these networks were designed to mimic patterns and processes that occur between neurons [4]. However, instead of several billions of neurons, each connected to hundreds of thousands of other neurons, we tend to build neural networks that exist in discrete layers, connected through a series of consecutive linear transformations. This architecture for artificial neural networks is was proposed in the in 1950's and is often dubbed the *multi-layer perceptron* model [1].

In order to train this model to produce desired results, some algorithm needs to find a set of parameters of each linear transformation that achieves a desired result. In the case of mutli-layer neural network, the goal is to produce an appropriate set of entries in the network's final layer. We measure the validity of these entries with something called a *cost* function, or *error* function [2]. The way to minimize the output of this function to find a set of parameters within the model that achieves this minimal value. The algorithm that we will explore to find these set of parameters is called *Stochastic Gradient Descent*.

Gradient Descent algorithms lie at the heart of several machine learning algorithms [2]. In general, the goal of the algorithm is find a set of parameters that minimize the value of a particular function, which we call the *cost* or *objective* function [3]. To do this, we apply a procedure from multivariate calculus that repeats a set of steps until a local minimum of that particular function is found. Note that while we ideally want to find a global minimum (the minimum of the whole function), this is computationally unrealistic, so local minima of the function are often used in it's place.

Suppose we have some scalar defined function, f of n independent variables. We can express this a function of n entries or as a single vector of n entries. We notate this as:

$$f = f(x_0, x_1, ..., x_{n-2}, x_{n-1}) = f(\vec{x}) \tag{1}$$

Althought the function is multivariate, it is still scalar-defined. The gradient of $f(\vec{x})$, returns a vector, with each element given by the partial derivative of f with respect to the i-th variable:

$$\nabla\big[f(x_0, x_1, ..., x_{n-2}, x_{n-1})\big] = \big[\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_{n_2}}, \frac{\partial f}{\partial x_{n-1}}\big] \tag{2}$$

Geometrically, the gradient provides a vector that points in the direction that causes the value of f to increase the fastest. In a two dimensional function, where the output is the 3rd dimension, this gives the direction of steepest ascent. Note that if the function of of n variables, then this operation take place in n-dimensional vector space.

The general procedure of the Gradient Descent Algorithm is:

1. Pick a 'starting point', p_0 in the n-d space. This is done by evaluating the function f for each of it's n input variables.
2. Compute the gradient of f, eq. (2) at the point p_0. Multiply the gradient by -1. This is the direction of steepest *descent*.
3. Follow the negative gradient a certain distance, η (called the *learning rate*[1]) to reach a new point p_1.
4. Repeat steps 2 and 3 until the value of $-\nabla f$ returns 0 withina certain tolerance. This indicates that a local minimum, or a saddle point has been found.

The set of parameters at the final point p_f are the set of parameters in each linear transformation that in general, minimize the value outputted by the objective function.

Some gradient descent algorithms may repeat this whole procedure with a series of initial points, each time tracking the minimum and corresponding parameters. This way, each time the algorithm is repeated, a new minimum is attained, which allows for a greater chance to find a successively lower value. While it seems appealing at first thought, it is work noting that this whole algorithm as outlined is *very* computationally expensive.

2 Considerations and Conventions

2.1 Structure

It is important to understand the architecture of a multi-layer neural network (again, also called a *multilayer perception* or MLP) at a very basic level. In it's simplest form as well will describe, a neural network is a collection of discrete layers of functions (often called *neuron* or *nodes*) [2]. The exact amount of layers in a network and the exact number of neurons in each layer depends on the type of task that the network seeks to accomplish. Currently, there is no formal rule to outline this, and is often determined on a emperical basis [1].

The entry point of a neural network, called *Layer 0* is the set of functions (or values) that receives an initial piece of data. The number of these inputs neurons in this layer corresponds to the number of *features* in the base data set. Thus if a sample of data has n_0 features, then there are n_0 neurons in this layer 0 of the neural network [1].

To move to the next layer, an operation is applied to all of the values in the previous layer. We model this linear transformation with standard matrix multiplication. Notation conventions for this will outlined in the next section. After each matrix multiplication is applied, the input is effectively transformed into the next layer of the network. It is important to know that although layer 0 has n_0 neurons, any other layer may have a different number of neurons in it, This difference is handled by the dimension of the matrix that allows for the transformation between adjacent layers.

In a network with L layers, there are $L-1$ matrix multiplications required, the last one usually labeld $L-2$. (due to 0 - indexing). Once the $L-2$ matrix has been applied to layer $L-2$, the resultant layer, $L-1$, is the output of the network, as it is the last layer. In the case of a network being used for a classification problem, the number of nodes in the final layer of the network, corresponds to the number of target classes [1]. Thus if we built a *K-Folds* classifier, the output layer of the CNN would have $n_{(l-1)} = K$ neurons.

2.2 Notation

The mathematical description of neural networks and stochastic gradient descent (SGD) classifiers is largely based in linear algebra [2]. This means a great deal of matrix vector indexing is required to fully describe the procedure. For this work, I will be using a standard of 0 - indexing all objects. I outline a notional convention for this work:

- A scalar:
$$n_l \tag{3}$$
is the number of neurons (also called nodes or functions) in the l-th layer.

- A vector:
$$\vec{x}_i^l \tag{4}$$
is the i-th entry in the vector x in the l-th layer of a network. The vector that describes layer l of the network has n_l rows and 1 column. Thus the vector $\vec{x}^1$ has a n_l number of components in it.

- A matrix:
$$W_{i,j}^l \tag{5}$$
is the entry in the i-th row, and the j-th column of matrix W that operates on the l-th layer of a network. The matrix that operates on layer l has $n_{(l+1)}$ rows and n_l columns

Mathematically, each neuron, contains a (floating point) number that can be operated on.

3 Neural Network Procedure - Feed Forward

Suppose we have a linear neural network, with L layers of neurons, each layer containing n_l number of neurons in it. The first layer of neurons is called the *input layer*, which contains n_0 neurons. This corresponds to the number of input features given to the network. The final layer of neurons is called the *output layer* and corresponds to the number of target classes to place a sample into. Each sample input feature then fills in the entry of it's respective neuron. Thus, after presenting the network with an array of raw data, the x^0 vector then becomes:

$$\vec{x}^0 = \left[x_0^0, x_1^0, x_2^0, ..., x_{n_0-2}^0, x_{n_0-1}^0\right]^T \tag{6}$$

Understand that every entry in this vector is a numerical quantity. In the case of a computer, they are generally floating-point numbers, often normalized between -1 and $+1$ to ease the magnitude of the values.

4 Forward Propagation

The network must then feed this information in the form of a column vector to the next layer of itself. It must take the vector $\vec{x}^0$ and transform it to into the vector $\vec{x}^1$ containing the entries in the neurons in 1-th layer. Recall that these layers may have a different number of neurons. We perform this transformation in the form of a matrix of weights, which we denote as W^0.

The weighting matrix W^0 operates on vector $\vec{x}^0$ to produce the entries in the vector $\vec{x}^1$. More generally, matrix W^l operates on layer/vector x^l to produce layer/vector x^{l+1}. The dimensions of matrix W^l must then be n_{l+1} rows by n_l columns. In a network with L layers, there are L vectors (0 through $L-1$) and $L-1$ weighting matrices (0 through $L-2$).

The process of matrix multiplication makes up the bulk of the *feed-forward* algorithm for neural networks. We can generalize this reoccurring procedure by the matrix vector equation below.

$$x^{l+1} = W^l x^l \tag{7}$$

Where the superscript l indicates an arbitrary integer layer number. This process repeats for $L-1$ matrix multiplications, often making a network a very computationally expensive operation. Once the vector x^{L-1} is produced, the network has finished it's algorithm and a result is then the final output of the network. Recall that this layer is called the *output layer* and has n_{L-1} number of neurons in it.

4.1 The Weighting Matrices

The actual procedure of this type of neural network is essentially this basic recursive linear algebra problem. It then raises the questions, how is each matrix, W^l built? What are it's

elements and what do they do? If we examine the matrix-vector equation (7), we can break this down further into it's constituent parts.

For simplification of superscripts and subscripts, let the layer, x^{l+1} be notated by the vector y. This layer will contain $n_{l+1} = a$ neurons within it. Similarly, let the layer, x^l be the vector x with $n_l = b$ neurons in it. Thus the matrix W^l has a rows, and b columns. Each entry is denoted as $W_{i,j}$ for the i-th row and the j-th column. The matrix-vector equation expanded out then becomes:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{a-2} \\ y_{a-1} \end{bmatrix} = \begin{bmatrix} W_{0,0} & W_{0,1} & \dots & W_{0,b-2} & W_{0,b-1} \\ W_{1,0} & W_{1,1} & \dots & W_{1,b-2} & W_{1,b-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ W_{a-2,0} & W_{a-2,1} & \dots & W_{a-2,b-2} & W_{a-2,b-1} \\ W_{a-1,0} & W_{a-1,1} & \dots & W_{a-1,b-2} & W_{a-1,b-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{b-2} \\ x_{b-1} \end{bmatrix} \tag{8}$$

Keep in mind that a may not be equal to b. We can now use this to understand a little bit more specifically what the weighting matrix is doing, and what each entry in the operation does.

From linear algebra, the entry y_i in the column vector y is given by the dot product between the W_i row and the column vector x. We can denote this as:

$$y_i = \sum_{j=0}^{b-1} W_{i,j}(x_j) = (W_{i,0}x_0) + (W_{i,1}x_1) + ... + (W_{i,b-2}x_{b-2}) + (W_{i,b-1}x_{b-1}) \tag{9}$$

This equation tells us exactly how the elements in the layer x influence the elements produced in layer y. Thus we can see that it is a linear combination of all elements in the layer x, whose coefficient are given by the corresponding entries in the weighting matrix that produces the elements in the next layer. For example, neuron y_m is related to neuron x_n by a factor of $W_{n,m}$. If the element $W_{m,n}$ is very small or zero, then x_n has little or no affect on y_m. Conversely, if $W_{m,n}$ is very large, then changes in the value of x_n have a large impact on y_m.

In some cases, each step may be accompanied by the inclusion of a *bias function* , β, and a *sigmoid function*, $\sigma(x)$ [1]. The sigmoid is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{10}$$

Often, the bias function is just some scalar value that is used to increase or decrease the value of an entry in the y layer. This is typically tacked on at the end of the dot product equation in eqn. (9). Additionally, it is often in the best interest of a computer to work with values between -1 and $+1$ to prevent which is why we apply the sigmoid operator to

the matrix- vector equation. It takes any real number, and maps it to the open interval $(-1, +1)$. This prevents any matrix element or neuron element from eve getting to large to numerically handle withing reason. These two additions modify our matrix- vector equation (7) to become:

$$x^{l+1} = \sigma\big(W^l x^l + \beta\big) \tag{11}$$

Where the $\sigma(x)$ operation is applied to all elements in the output layer x^{l+1}.

It is the structure, elements and organization of these weighting matrices, W that determine how our network behaves. For an untrained network, these elements are often pseudo-randomly generated (called a *cold-start network*[1]). With each training sample introduced t the next work, we use the Stochastic gradient descent algorithm to adjust all elements in these $L-1$ matrices. After enough samples, we will have then ideally converges on a set of elements in all matrices that allow for the production of a local or global minimum as defined by some cost function [2].

5 Neural Network Procedure - Back Propagation

Suppose we have entered a single training data sample into a multi-layer neural network. Each input feature in layer x^0 is then transformed as prescribed above, until it reaches the output layer, x^{l-1}. The entries in each neuron in this layer correspond to the entries in a corresponding vector object of length n_{l-1}. We can now compare this output vector to the *desired* output, given by the corresponding label associated with that training sample.

5.1 The Objective Function

To determined how the network performed on this particular labeled sample, we define an objective function to minimize. Let the network bet a classifier over k independent classes. Thus the output later has $n_{l-1} - k$ neurons in it. For this example, we use the *means squared* error function [3]. It is defined:

$$MSE = \frac{1}{k}\sum_{i=0}^{k-1}\Big(y_i - x_i^{l-1}\Big)^2 \tag{12}$$

Where k is the number of possible output classes. y_i is the target vector, which is what this particular sample *should* cause the output layer to look like. x_i^{l-1} is the output vector as actually produce by the network for this particular sample.

Recall that the output of the network is the vector x^{l-1} of k elements, each a floating point number between -1 and $+1$. The target vector ,y also contains k entries, all either 0 or $+1$. If the training sample fed into the network belongs in class τ, then the element y_τ of the target vector contains a $+1$, and all other $k-1$ elements are all 0. Thus, a well-trained network would also ideally produce a vector x^{l-1} that has a very similar structure for that particular sample.

If the vectors are indeed similar, then the MSE (12) for that sample becomes quite small. If the network is still poorly trained, the output vector from the network may differ quite substantially from the target y. Thus the element-wise difference squared will take on a very high value. Thus now, the task becomes the minimization of this objective function.

5.2 Gradient Descent

The mean-squared objective function for a k-bins classifier at surface level only seems to be a dependent on the output vector x^{l-1} and the target vector y. This would only make it dependent on 2 vectors of length k, or $2k$ parameters. However, it took a special combination of all elements in all weighting matrices, and each bais function to produce that value. When considering $L-1$ matrices, with $(n_{l-1} \times n_l)$ elements in each, and a unique bias function, it becomes apparent that the object function becomes dependent on a great many parameters.

For smaller networks with only a few internal layers, this may only amalgamate to a few hundred parameters to adjust [2]. In the case of larger and more complex *deep learning models*, this may sum up to several ten or hundreds of thousands of parameters to adjust, making the task of minimization far more difficult.

In traditional single variable calculus, to find the minimum of a function, $f(x)$, and find the places where it's first derivative is equal to zero: $\frac{df}{dx} = 0$. In the case of a multivariate problem, where f is a scalar function of multiple variables, the problem is a little bit harder to solve, because it is very difficult to find where each partial derivative is equal to zero: $\frac{\partial f}{\partial x_1} = \frac{\partial f}{\partial x_2} = ... = 0$.

To attempt to find the minimum, we employ the gradient descent algorithm.

References

[1] Géron Aurélien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly, 2017.

[2] Goodfellow, Ian, et al.*Deep Learning*. MIT Press, 2017.

[3] James, Gareth, et al. An Introduction to Statistical Learning with Applications in R. Springer, 2017.

[4] McCulloch, Warren S., and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, 1943, pp. 115–133.

[5] Petrik, Marek. "Introduction to Machine Learning." Machine Learning. 22 Jan. 2020, Durham, New Hampshire.