Optimization Methods for Deep Feed-Forward Neural Networks

Landon Buell

25 June 2020

1 Introduction

1.1 Deep Feed-Forward Neural Network Structure

A Deep-Feed forward neural network is a class of architectures built around layers of operations that are called in a fixed sequence. Each layer, (l) takes some input object $x^{(l-1)}$, applies some operation, $f^{(l)}$, and produces some output $x^{(l)}$, where the super-script indicates a layer index. The object $x^{(l)}$ then becomes the input for the next layer of the network [1, 2]. This repeated process can be shown mathematically by:

$$x^{(l)} = f^{(l)}\left[x^{(l-1)}\right] \tag{1}$$

A model with L layers requires an input object $x^{(0)}$, which is then transformed by $L-1$ layers to produce object $x^{(L-1)}$, which we refer to as the output of the network.

Each layer of the network takes the form of a function-like object f that takes an input $x \in \mathbb{R}^N$ and transforms it into an output $y \in \mathbb{R}^M$. Typically, f can take the form of a matrix-vector equation, a k-dimensional convolution operation, a pooling operation, a non-linear activation function, or any other general transformation. Often, these operations may be combined to produce composite functions in each layer. In each case, there as a series of parameters associated with the function f. In the case of a matrix-vector equation, there are the elements inside each matrix and bias vector, and in the case of convolution, there are the weighting values within the kernel itself.

1.2 Optimization Motivation

Ultimately any output is function of the input, and the parameters in each layer. By convention, these parameters can be concatenated into a single object called Θ. For smaller neural networks, Θ may only have several hundred elements, but for increasingly complex models, there can upwards of several million elements within the object. The goal of the network then is to find a particular set of elements $\{\theta_0, \theta_1, ...\}$ that allow a network to produce a desirable output based on that input. To do this, we require a set of samples $X = \{x_0, x_1, x_2, ...\}$ and a set of corresponding labels $Y = \{y_0, y_1, y_2, ...\}$, such that element x_i should produce result y_i, when passed into the network model.

In reality, a network will not produce output y_i from samples x_i exactly, but will instead produce an *approximation* of y_i which we denote as y_i^*. We can use a function J called the *objective function* which compares the expected output, y_i, to the actual output, y_i^* and produces a numerical score ranking how much the two values differ from each other. This score is called the *cost* of the sample [3], lower numbers being favorable, showing a low difference or low cost of the sample.

The value of the cost function is then directly dependent on the expected output, y_i and the given output y_i^*. The cost of a single sample can be shown as $J = J(y_i, y_i^*)$. However, the output is implicitly dependent on the the network input x_i and the parameters in each layer, given by Θ. Since the latter two are fixed objects, we can then only adjust the elements in the object Θ. We do so to allow for increasingly lower values of the objective function, $J(\Theta)$, which indicates that on average, the network model is producing more and more accurate predictions. This process is called *optimization.*

It is worth noting that machine learning does differ from explicit multi-dimensional optimization. In machine learning, the performance of a neural network is typically measured by a standard statistical metric such as precision score, recall score, accuracy rating, etc. [1]. These measures are far more interpretable to a human, and often more indicative that a model is performing as desired than the cost function itself. Thus, we optimize the elements in $J(\Theta)$ in hopes that doing so will also improve these related performance metrics [2].Despite this indirect optimization, it is the basis for much of machine-learning.

1.3 The Θ Object

It is worth reinforcing that although we have implied Θ to be a single dimensional column-vector-like object, it can in reality, hold just about any shape. It may be more convenient to express it as a *set*, containing the parameters for each layer's kernels, weights and biases in the form of vectors or matrices. It is often reasonable to convey Θ not as a tensor-like object, but rather a set of objects, concatenated from each layer:

$$\Theta = \left\{W^{(0)}, b^{(0)}, W^{(i)}, b^{(i)}, W^{(l)}, b^{(l)}, ..., W^{(L-1)}, b^{(L-1)}\right\} \tag{2}$$

Where $W^{(i)}$, $b^{(i)}$ gives a weighting and element in the i-th layer.

Each element $W^{(1)}$, $b^{(l)}$ is an array-like object which contains further elements (floating-point numbers). For example: $W^{(l)} \in \mathbb{R}^{N\times M} = \left\{\theta_0, \theta_1, ..., \theta_{N\times M-1}\right\}$. Then, Θ is an object that contains matrices and vectors corresponding to the numerous parameter contained within each layer. Each of those parameters can be further broken down into it's constituent elements which are $\theta_0, \theta_1, \theta_2, ...$

2 Gradient Based Learning

A large trouble in machine learning and deep learning arises from the size of the dimensionality of Θ. For example, in two immediately adjacent *Dense Layers*, which n units in layer (l) and n units in layer $(l+1)$, there are $(m \times n) + m$ elements added to Θ. Often, deep learning networks can contain dozens of densely connect layers with hundreds of neurons in each layer. This pushes the size of Θ to grow very quickly. The number of elements in Θ is the dimensionality of the network's parameter space [2]. This space contains all possible combinations of parameters of the networks.

We need to then find the appropriate position in parameter-space as to minimize J. This task becomes even more difficult when considering that we do not define J explicitly for each position in the space. Furthermore, J may often contain discontinuities, or ill-conditionings, and is not easily represented by a traditional analytical function, thus we cannot solve the optimization with standard whiteboard mathematics. For deep and complex nerual networks, even over a finite set of possible floating-point values, this can amount to trillions of possible locations in parameter space to test, which is unpractical for computational implementation. This prevents us from traditional brute-force testing methods in hopes to find a local or global minimum.

Instead, we brute-force a *guess* or series of guesses for a possible solution. This guess is a particular point in parameter space that either be chosen randomly, or derived from a previously found solution. We can then optimize $J(\Theta)$ based on the guess or collection of guesses by adjusting each value in Θ as the reduce the value of J by a small quantity. By repeating this process hundreds of times, under the right conditions, we can take a pseudo-random guess at the solution and slowly adjust each element to develop an approximation of the solution. This process is a simplification of *gradient descent* algorithm.

2.1 Gradient Descent

Gradient Descent aims to compute the numerical gradient of the cost function at some potion in parameter-space and use the negative gradient to update the elements in the space, as to take a *step* to reduce the value of the cost from the previous step. For example, suppose we guess a set of parameters, θ^*, in a k-dimensional parameter-space, $\theta^* \in \mathbb{R}^k$. We then compute the gradient of the cost function, with respect to each parameter in Θ, which we denote as $\nabla_\Theta\big[J(\Theta)\big]$. Recall the that gradient operator, ∇ returns a vector where each element is partial derivative of J with respect to that parameter:

$$\nabla_\Theta\big[J(\Theta)\big] = \nabla_\Theta\big[J(\theta_0, \theta_1, \theta_2, \theta_3, ...)\big] = \Big[\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \frac{\partial J}{\partial \theta_3}, ...\Big]^T \tag{3}$$

Both Θ^* and $\nabla_\Theta\big[J(\Theta)\big]$ exist in the same dimensional space, $\mathbb{R}^k$, and thus can be added together - or rather we add the *negative* gradient to decrease the cost. The elements in Θ^*

provides a sort of starting point, and the elements in $-\nabla_\theta\big[J(\Theta)\big]$ act as a sort of *step* to take, such that changing each element, θ_i^* by some amount $\frac{\partial J}{\partial \theta_i}$ results in a small decrease in the overall cost function. Generally, across all parameters in the parameters object, we note this update rule as:

$$\Theta^* = \Theta^* + -\alpha \nabla_\Theta \big[J(\Theta)\big] \tag{4}$$

Another trouble arises when we reconsider that each element in the gradient vector is often not well-defined by a traditional mathematical function, meaning we cannot simply use symbols and standard calculus rules to find the necessary values in eqn. (4), we need to use numerical approximations, and *back propagation* to compute each partial derivative.

2.2 Backwards-Propagation

Given the recursive nature of the function, we cannot compute each element in the gradient vector by conventional derivatives. We have to use the chain rule from calculus to begin at the final output of the network and work backwards to the entry layer to compute each derivative. This process is called *back-propagation.*

To begin, we compute the gradient of the cost function with respect each element in the output layer. $\nabla_{y^*}\big[J(\Theta)\big]$. Next, recall that a layer's operation represented by the function $f^{(l)}$ may often be the composition of many functions, such as a dense layer having a linear matrix-vector product, which may then be subject to a non-linear activation function. We also need to apply the chain rule within each layer to ensure that each element is accurately computed.

By Computing the derivative in any given layer l, we can use the chain rule to recursively back-ward propagate through the network to ensure that every parameter in the Θ object is accounted for. Rather than explicitly derive the mathematical rule for a single network type, we outline a general back-propagation procedure for a general feed-forward network. Each layer is presumed to follow a two-step structure:

$$\begin{aligned} z^{(l)} &= W^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} &= a^{(l)}\big[z^{(l)}\big] \end{aligned} \tag{5}$$

Where $W^{(l)}$ is a weighting matrix, $b^{(l)}$ is a bias vector, and $x^{(l-1)}$ is the activation from the previous layer. The array $z^{(l)}$ represents the *pre-activations* to which the an activation function $a^{(l)}$ is applied to. This algorithm is adapted from Goodfellow, 2017 [2].

3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is considered to be amoung the more powerful and versatile variations of basic gradient-based optimization methods. It follows all of the same

Algorithm 1 Backwards Propagation System in a fully connected neural network model for a single sample.

```
Require: Network depth of $L$ layers , cost function $J$ , learning rate $\alpha$
Require: Object $\Theta$ containing all network parameters as in eqn. (2)
Require: Object of activation function derivatices: $\partial\vec{a}^{(i)}$
Require: Object of final activations by layer: $\vec{x}^{(i)}$ with $i \in [1, 2, ..., L-2, L-1]$
Require: Object of pre-activations by layer: $\vec{z}^{(i)}$ with $i \in [1, 2, ..., L-2, L-1]$
Require: Sample target output $y$ (usually one-hot-encoded)
  Array to hold weight updates: $dW \leftarrow \{\}$
  Array to hold bias updates: $db \leftarrow \{\})$
  Compute gradient of $J$ w.r.t output layer $\vec{x}^{(0)}$:
  $dx \leftarrow \nabla_{y^*}\big[J(y^*, y)\big]$
  for $l = L-2, L-3, ..., 2, 1$ do
    Compute gradient of pre-activation values:
    $dx \leftarrow \nabla_{z^{(l)}}\big[J\big] = dx \odot \partial\vec{a}^{(l)}\big[z^{(l)}\big]$
    Compute gradient w.r.t parameters in bais vectors and weighting arrays:
    $\nabla_{b^{(l)}}\big[J\big] = dx$
    $\nabla_{W^{(l)}}\big[J\big] = dx \cdot x^{(l-1)}$
  end for
  ASSERT: $dW^{(i)}$ has same dimension as $W^{(i)}$ for $i \in [0, 1, ..., L-3, L-2]$
  ASSERT: $db^{(i)}$ has same dimension as $b^{(i)}$ for $i \in [0, 1, ..., L-3, L-2]$
  for $l = 0, 1, ..., L-3, L-2$ do
    $W^{(i)} \leftarrow W^{(i)} + (\alpha)dW^{(i)}$
    $b^{(i)} \leftarrow b^{(i)} + (\alpha)db^{(i)}$
  end for
  return Object containing updated weights: $\hat{W}^{(i)}$ with $i \in [1, 1, ..., L-2, L-1]$
  return Object containing updated biases: $\hat{b}^{(i)}$ with $i \in [1, 1, ..., L-2, L-1]$
```

rules of previously introduce gradient descent, but add adds are sort of augmentation to it as well.

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.