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:

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

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, also denoted by the variable y^*

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.

The nature of the deep neural network is a multilayered function composition:

$$y^* = x^{(L-1)} = f^{(L-1)}\big[f^{(L-2)}...f^{(1)}\big[f^{(0)}\big[x^{(0)}\big]\big]\big] \tag{2}$$

Each function is composed of a series of variable parameters, called *weights*, W, and *biases*, b. Each layer-function may also contain a series of static parameters such as an nonlinear

activation function,a. In a densely connectivity deep neural network, we define a forward pass for the single layer (1) to take some variation of the form:

$$x^{(l)} = f^{(l)}\big[x^{(l-1)}\big] = a^{(l)}\big[W^{(l)}x^{(l-1)} + b^{(l)}\big] \tag{3}$$

For a network with L layers, we can define a forward pass computation with algorithm (1)

Algorithm 1 Forward propagation system in a standard deep neural network. Each iteration in the main *for-loop* represents the execution of a layer, and passing the result to the "next" layer function. A practical application of this algorithm should include batches of samples instead of a single sample.

Require: Network with L layers - $0, 1, 2, ..., L-2, L-1$
Require: Set of weighting parameters - $W^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of bias parameters - $b^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function - $a^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Input sample - $x^{(0)}$
Require: Arrays to hold pre-nonlinear activation - $Z = \{\}$
Require: Arrays to hold post-nonlinear activation - $X = \{\}$
  for $l = 1, 2, ..., L-2, L-1$ **do**
    Compute pre-nonlinear activation
    $z^{(l)} \leftarrow W^{(l)}x^{(l-1)} + b^{(l)}$
    Apply activation function
    $x^{(l)} \leftarrow a^{(l)}[z^{(l)}]$
    Store values for later use in optimizer
    $Z \leftarrow Z.add(z^{(l)})$
    $X \leftarrow X.add(x^{(l)})$
  end for
  Network final prediction is last object in 'act' array.
  $y^* \leftarrow x^{(L-1)} = X[-1]$
  Return the pre-nonlinear activations and final-nonlinear activations
  return Z, X

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, each containing dynamic variable parameters.

$$\Theta = \left\{ W^{(0)}, b^{(0)}, W^{(i)}, b^{(i)}, W^{(l)}, b^{(l)}, ..., W^{(L-1)}, b^{(L-1)} \right\} \tag{4}$$

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:

$$\nabla_\Theta\big[J(\Theta)\big] = \nabla_\Theta\big[J(\theta_0, \theta_1, \theta_2, \theta_3, ...)\big] = \Big[\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \frac{\partial J}{\partial \theta_3}, ...\Big]^T \tag{5}$$

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:

$$\Theta = \Theta + -\alpha \nabla_{\Theta}\big[J(\Theta)\big] \tag{6}$$

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. (**??**), 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:

$$\begin{aligned} z^{(l)} &= W^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} &= a^{(l)}\big[z^{(l)}\big] \end{aligned} \tag{7}$$

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].

Algorithm 2 Backwards propagation system, in a standard deep neural network. Each iteration in the first *for-loop* computes the gradient of the cost function J with respect to the weight and bias arrays. Each element in those arrays is then the discrete gradient of that parameter. A practical application of this algorithm should include batches of samples instead of a single sample

Require: Cost/Objective function $J(y, y^*)$ and learning rate α
Require: Set of weighting parameters - $W^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of bias parameters - $b^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function - $a^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function derivatives - $\partial a^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of pre-nonlinear activation - $Z^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of post-nonlinear activation - $X^{(i)}, i \in \{0, 1, ..., L-1\}$
Execute forward pass in algorithm (1) and compute the gradient of the cost with respect to the final layer activations
$dx \leftarrow \nabla_{y^*} J(y, y^*)$
for $L-1, L-2, ..., 2, 1$ **do**
Compute gradient w.r.t pre-nonlinear activation portion of layer function
$dx^{(l)} \leftarrow \nabla_{Z^{(l)}} J = dx^{(l)} \odot \partial a^{(l)}[Z^{(l)}]$
Compute gradient w.r.t weighting and bias elements
$db \leftarrow \nabla_{b^{(l)}} J = dx^{(l)}$
$dW \leftarrow \nabla_{W^{(l)}} J = dx^{(l)} \cdot X^{(l-1)}$
add layer gradients to respective objects
$W^{(l)} \leftarrow W^{(l)} + -\alpha(dW)$
$b^{(l)} \leftarrow b^{(l)} + -\alpha(db)$
end for
Return Updated weight and bias parameters
return W, b

2.3 Challenges of Gradient-Based Learning

3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is considered to be among the more powerful and versatile variations of basic gradient-based optimization methods [1]. It follows all of the same rules of previously introduced gradient descent and back propagation, but take advantage of a few extra tactics. The first and most obvious that we have compute the gradient over a *mini-batch* of m samples, and then average by the number of samples, and use this quantity to update the parameters in Θ. This ensures that no single sample creates a bias in the optimization procedure. Rather, a random batch of samples which are ideally share similar distributions [3].

Additionally, SGD often places more emphasis on the learning rate hyper-parameter α. Instead of a constant, it is standard practice to scale the learning rate to scale the learning rate according to the iteration number. Often, when the initial set of parameters are tested, the result gradient is large because it tends to me "far" from the local minimum. This, we begin with a larger learning rate to amplify the initial steps effects on the cost. As the solver proceeds, the loss tends to become smaller and smaller as it's gets "closer" to a local minimum, and thus the learning rate shrinks appropriately in order to ensure that a minimum is not overshot or skipped.

For a learning rate that is constant, we denote α and for one that changes with each iteration i, we denote α_i. Often the learning rate α or learning rate schedule, $\alpha(i)$ is chosen with "trial-and error" as a hyper parameter, and it's exact value or change is values variable by each problem [2]. Additionally, Neural networks may employ a *stopping criteria* S, which is either a numerical condition or a number of steps that if reached, the gradient descent optimizer stops executing and what ever set of parameters are chosen, is taken to be the solution. This is useful for ensuring that a network does not over fit on a set of data, and a network does not step and indefinite amount of time on a single batch.

Below, we outline a pseudo-code example for executing stochastic gradient descent.

Algorithm 3 Stochastic Gradient Descent (SGD) in a neural network

Require: Learning rate α or learning rate schedule, $\alpha(i)$.
Require: Subroutine/function to compute gradient of cost function.
Require: Mini-batch size, m
Require: Stopping criterion S
  while Stopping Criterion S is **false do**
    Extract a mini-batch of m samples from larger data set X. $[x^{(0)}, x^{(1)}, ..., x^{(m-1)}]$ and corresponding target values $[y^{(0)}, y^{(1)}, ..., y^{(m-1)}]$.
    Compute numerical gradient estimate of each sample in batch. This can be done with standard back-propagation in algorithm (2) and normalize by batch size m:
    $\Delta\Theta \leftarrow \frac{1}{m}\nabla_{\Theta}\Big[\sum_n J(y^{(n)}, y^{*(n)})\Big]$
    Apply update rule from batch, see eqn. (**??**)
    $\Theta \leftarrow \Theta + (-\alpha_i)\Delta\Theta$.
    Update Iteration number: $i \leftarrow i + 1$
  end while

3.1 Momentum Hyper-parameter

Stochastic Gradient Descent is a powerful and versatile learning algorithm, but in higher dimensional parameter spaces may take a long time to execute and converge on a reasonable set of values for Θ. Additionally, the learning rate, α can be chosen to accelerate this process

but the exact appropriate value and consequences may vary drastically with each problem. For this reason, we can choose to add a *momentum hyper-parameter* to the SGD optimizer method [1]. This enables a faster convergence when faced with high curvature.

Momentum seeks to use an rolling average or previous gradients and continues to move in that direction, while exponentially decaying. The name momentum is a loose reference to it's physical analog, in which a *mass* is used to scale a *velocity*. In this case, velocity is the gradient descent, being scaled by unit mass and additional hyper-parameter β. We denote momentum as a vector v, as it is reasonable to consider it as a velocity that moves through parameter space. When adding momentum to a SGD optimizer, the update rule changes to becomes:

$$\begin{aligned} v =& \beta v + (-\alpha_i)\nabla_\Theta\Big[\frac{1}{m}\sum_{n=1}^{m} J(y^{(n)}, y^{*(n)})\Big], \qquad \beta \in [0,1) \\ \Theta =& \Theta + v \end{aligned} \tag{8}$$

The velocity parameter works to accumulate the elements in the gradient vector with each iteration. A larger value of β compared to α is used to weight more previous iterations to calculate the affect on the next iteration.

Algorithm 4 Stochastic Gradient Descent (SGD) optimizer for a neural network

Require: Learning rate α or learning rate schedule, $\alpha(i)$.
Require: Initial velocity $v(0)$ hyper-parameter (usually left at 0).
Require: Hyper-parameter $\beta \in [0, 1)$
Require: Subroutine/function to compute gradient of cost function.
Require: Mini-batch size, m
Require: Stopping criterion S
  while Stopping Criterion S is **false do**
    Extract a mini-batch of m samples from larger data set X. $[x^{(0)}, x^{(1)}, ..., x^{(m-1)}]$ and corresponding target values $[y^{(0)}, y^{(1)}, ..., y^{(m-1)}]$.
    Compute numerical gradient estimate of each sample in batch. This can be done with standard back-propagation in algorithm (2) and normalize by batch size m:
    $\nabla J \leftarrow \frac{1}{m}\nabla_\Theta\Big[\sum_n J(y^{(n)}, y^{*(n)})\Big]$
    Compute velocity, and apply update from eqn. (8)
    $v \leftarrow \beta v + (-\alpha_i)\nabla J$
    $\Theta \leftarrow \Theta + v$
    Update Iteration number: $i \leftarrow i + 1$
  end while

3.2 Nesterov's Momentum

4 Adaptive Learning Algorithms

It has been experimentally shown that the learning rate α is an incredibly important hyper-parameter in the performance of an optimizer [1]. Instead of applying a fixed learning rate or learning rate schedule, we can choose to use an *adaptive learning rate* method. Furthermore since some parameters will be more sensitive to the learning rate than other, we can introduce a modification to the learning rate such that it is a scalar, but accompanied by a vector that scales each element in Θ differently [2]. We can then adapt each learning rate in the process of optimizing.

4.1 AdaGrad

The AdaGrad optimizer individually adapts learning rates of each network parameter based on the sum of previous values of the gradient cost function. Parameter learning rates are updated by scaling them inversely proportional to the square-root of the sum of all previous squared gradient values [2]. We notate with accumulated gradient r, and corresponding update rule, with stabilizing parameter δ, and learning rate α:

$$\begin{aligned} r =& r + \nabla_{\Theta}\Big[\frac{1}{m}\sum_{n=1}^{m} J(y^{(n)}, y^{*(n)})\Big] \odot \nabla_{\Theta}\Big[\frac{1}{m}\sum_{n=1}^{m} J(y^{(n)}, y^{*(n)})\Big] \\ \Theta =& \Theta + -\frac{\alpha}{\delta + \sqrt{r}} \odot \nabla_{\Theta}\Big[\frac{1}{m}\sum_{n=1}^{m} J(y^{(n)}, y^{*(n)})\Big] \end{aligned} \tag{9}$$

AdaGrad's method of accumulating past gradients to iterative update the learning rate, along with it's parameters allow for quick optimization problems with relatively convex solutions, but slower optimization with non-convex functions.

Algorithm 5 Adaptive Gradient (AdaGrad) optimizer for a neural network

Require: Learning rate α.
Require: Small constant δ for numerical stabilization, usually about $10^{-}7$.
Require: Subroutine/function to compute gradient of cost function.
Require: Mini-batch size, m
Require: Stopping criterion S
  while Stopping Criterion S is **false do**
    Extract a mini-batch of m samples from larger data set X. $[x^{(0)}, x^{(1)}, ..., x^{(m-1)}]$ and corresponding target values $[y^{(0)}, y^{(1)}, ..., y^{(m-1)}]$.
    Compute numerical gradient estimate of each sample in batch. This can be done with standard back-propagation in algorithm (2) and normalize by batch size m:
    $\nabla J \leftarrow \frac{1}{m} \nabla_{\Theta} \Big[\sum_n J(y^{(n)}, y^{*(n)}) \Big]$
    Compute accumulated squared gradient: $r \leftarrow r + \nabla J \odot \nabla J$
    Apply update rule from eqn. (9)
    $\Delta \Theta \leftarrow -\frac{\alpha}{\delta + \sqrt{r}} \odot \nabla J$
    $\Theta = \Theta + \Delta \Theta$
    Update Iteration number: $i \leftarrow i + 1$
  end while

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.