Optimization in Machine Learning

Venkata Sai Muktevi December 17, 2020

Introduction

Optimization is at the heart of all machine learning and drives the learning process forward. We usually must predict our dependent variable by feeding independent variables into an algorithm. This will result in an error term which can be portrayed as the difference in the dependent variable values and the prediction values. The optimization process takes over in this context to reduce the error in prediction as speedily and efficiently as possible to get closer to predicting the right values for the dependent variable. In this way the model fits the data. Optimization can be described as tuning the parameters involved in the learning process to achieve the highest possible standard of prediction according to the decided metrics.

While pursuing machine learning, it is critical to understand the importance of the optimization techniques being used. Often, we are introduced to these topics very gently while discussing the steps involved in a machine learning algorithm. I have not seen the depth and breadth of optimization being explored in these initial stages of learning which I believe is crucial knowledge to gain a better understanding of the algorithms that work in machine learning. This was the main reason why I chose to dig deeper into optimization techniques.

To appreciate the underlying principles of which powerful, effective, and generalized machine learning models are built upon, to look at the evolution of these techniques and to also learn more about Backpropagation and other such significant algorithms that have greatly contributed to machine learning today.

This essay is aimed at diving into optimization methods pertaining to machine learning algorithms. This essay is targeted for those who are learning machine learning techniques and would like to understand the breadth of optimization methods and fundamentals. We will also have a brief discussion on Backpropagation towards the end. The scope of this essay is bounded by my study and may be biased according to the references used. The aim is to compile the readings and insights of various research papers, book extracts, blogs and articles covering this topic that I have come across during my study. I also apologize for the use of personal opinions without knowledgeable backing or any ambiguity if found.

This is essay will discuss the following topics. First, we will try to get a glimpse of the evolution of problem-solving using optimization before filtering in on the machine learning context. We will then build a foundation on some of the basics of optimization and look into popular Cost Functions which are objective functions being optimized in machine learning algorithms. Then we will take a brief look at numerical analysis with Newton’s method, the ultimate predecessor of optimization algorithms. After that, we will explore gradient-based optimization methods, which are the most widely used type in machine learning. This will conclude our summary on the fundamentals and the background of optimization techniques. Then we can freely indulge in discussion of notable and interesting topics integrated with optimization in machine learning. The all-powerful Backpropagation will be the center of our discussion. Finally, we conclude with some findings, personal thoughts and next steps in further studies. It is important to note that all topics in this essay will not be explained in great detail but I will try to lay out the most interesting material and some basics that I have learned from this study. I hope to answer questions and discuss topics that are on the minds of fresh technological aspirants as well as capture the interests of those with prior knowledge.

Background

Glimpses of Optimization History and Evolution

Optimization problems have existed long before machine learning became a popular area of study. Obviously, optimization uses mathematical tools extensively. Therefore, it is not surprising to observe the early roots of optimization in mathematical methods. Antoniou & Lu (Practical Optimization: Algorithms and Engineering Applications, 2007) in their book explain that Optimization Theory is the branch of mathematics encompassing the quantitative study of optima methods and for finding them. They go on to explain how optimization problems occur in most disciplines but how they are abundant in engineering domains. They describe that most real-life problems have several solutions and occasionally an infinite number of solutions may be possible. If the problem at hand admits more than one solution, optimization can be achieved by finding the best solution of the problem in terms of some performance criterion. If the problem admits only one solution, which means only a unique set of parameter values is acceptable, then optimization cannot be applied.

The earliest optimization approach can be regarded as a point on a one-variable function with its first derivative equal to zero gives either maximum or minimum of that function (Wang, 2018). Wang (2018) in his blog on the history of optimization mentions that Pierre De Fermat and Joseph-Louis Lagrange first found calculus-based formulae for identifying optima. Isaac Newton and Johann C.F. Gauss first proposed iterative methods to search for an optimum. According to Wang, Formal optimization came from as far back as 1939 with “linear programming” started by Leonid Kantorovich. Optimization as a mathematical branch has multiple sub-branches of study. It was initially known as mathematical programming which was further classified into dealing with linear and non-linear optimization methods depending on the objective or the cost function. We mostly work with non-linear optimization problems in machine learning, which will be the scope of our discussion.

Wang (2018) divides optimization into 3 generations of approaches based on some salient features.

First Generation of Optimization is said to be characterized by:

1. Local optimization: dealing with local optimal point or a valley found for reducing loss.
2. Sequential / Iterative Search: which is akin to choosing the next optimal point using the information of a previously explored point.
3. Reliance on gradients or higher order derivatives.

It is easy to contemplate that the biggest issue with this generation was to explain why the optimal is the optimal and where to find the next optimal design in case the current solution was unusable.

Wang (2018) talks about the **Second Generation of Optimization** as Metaheuristic approaches. It is essential to note the difference between characteristic of heuristic and metaheuristic approaches at this point. In loose terms, Heuristics are often “problem-dependent” and are hence defined for a specific problem. Metaheuristics are “problem-independent” and can be applied to a broad range of problems. Talbi (Metaheuristics: From Design to Implementation, 2009), in his book, clearly tries to define Metaheuristics and their advantages. Such approaches include interesting global optimization algorithms like Genetic Algorithms (in 1960) invented by John Holland and Simulated Annealing (in 1983). Other popular approaches include: Particle Swarm Optimization, Ant Colony Optimization, Tabu Search, and Artificial Bee Colony. This group of approaches is inspired by nature or other heuristics and have the following features:

1. They are global optimization approaches in essence.
2. They support parallel computation.
3. They do not require gradients or higher derivatives.

Hence, we can see that these are complimentary to the previously discussed First Generation. Although, Wang (2018) discusses that the main problem with these approaches is the need for enormous amounts of trial points before reaching the global optimum. This is a major cause for concern.

Wang (2018) introduces the **Third Generation of Optimization** as including AI-based approaches that started in the late 1990s. He explains that the idea is to use a limited number of design trials (called sample points) to construct a machine-learning model between the design variables and objectives/constraints. The AI-based approaches are global optimization methods with the following advantages:

1. They support parallel computation.
2. They do not need gradients.
3. They use fewer design trials than before.
4. They offer insight and knowledge about the design problem.

These methods try to overcome the problems from the Second Generation by reducing the amount of data required. However, a big drawback of the AI-based approaches is the “curse of dimensionality” which we will soon discuss.

Fundamental Design of Optimization Problems in Machine Learning

For any optimization problem there are 3 basic elements:

1. Parameters
2. Constraints
3. Cost Function

Parameters are the different variables that describe a measure. The constraints set boundaries to the values that can be assigned to these parameters. The cost function describes the error in prediction of the measure which uses parameters and is to be minimized (or maximized) to obtain the optimal set of parameters that describes the system.

Optimization problems in machine learning arise through the definition of prediction and loss functions that appear in measures of expected and empirical risk that one aims to minimize, as given in the paper for Optimization Methods for Large Scale Machine Learning (Bottou, Curtis, & Nocedal, 2018). According to this source, consider a function h parameterized by a real vector $w \in \mathbb{R}^d$ over which the optimization is to be performed. Say a given input-output pair is (x, y), the prediction function is responsible for generating y' such that

$$h(x; w) = y'.$$

Once we have our prediction y', we use it to calculate a loss function l, where the loss is given by

$$Loss \; = \; l(h(x; w), y).$$

This loss function gauges the error in our prediction compared to the output. We optimize the loss function to minimize error in our prediction. When we have n input-output samples given by $\{(x_i, y_i)\}_{i=1}^{n}$ one may define a cost function as

$$Cost(w) = \frac{1}{n} \sum_{i=1}^{n} l(h\,(x_i; w), y_i).$$

This is also called the Empirical Risk function (Bottou, Curtis, & Nocedal, 2018). This is only one of the multiple cost models that can be applied to a variety of cost functions that we can encounter in optimization problems. Minimizing loss in this form is usually encountered in supervised learning models.

The cost function can have different forms. Common cost functions we come across in supervised learning for Regression problems are:

1. Mean Absolute Error (aka L1 Loss). Measured as the average sum of absolute differences between predictions y' and actual observations y.

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |\, y_i - y_i{}' |$$

2. Mean Squared Error (aka L2 Loss). Measured as the average sum of squares.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - y_i')^2$$

The difference between the two is that MSE heavily penalizes the errors due to squaring which makes it susceptible to outliers, whereas MAE is the absolute error and a bit more robust to outliers.

The following are common loss functions in Classification problems:

3. Hinge Loss / Multi-class SVM Loss. The score of the correct category s_j should be greater than sum of scores of all incorrect categories s_{y_i} by some safety margin (usually one). Hence, hinge loss is used for maximum-margin classification, most notably for Support Vector Machines.

$$Loss = \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right)$$

4. Cross Entropy Loss / Negative Log Likelihood. Multiplying the log of the actual predicted probability for the ground truth class. An important aspect of this is that cross entropy loss heavily penalizes the predictions that are confident but wrong.

$$CrossEntropyLoss = \; -y_i \, log(\hat{y}i) + (1 - \hat{y}_i) \, log(1 - \hat{y}_i)$$

Notice the difference between cost and loss functions although sometimes they are used interchangeably. Usually, the loss function (or error) is for a single training example, while the cost function is over the entire training set (or mini-batch for mini-batch gradient descent).

The cost function therefore varies according to the optimization approach being taken and depending on the type of basic elements described in the machine learning model. Unsupervised learning cost functions are to be interpreted differently as there is no output variable y for training the model. For example, the cost function for k-means clustering is given by the Euclidean Distance metric. There are many other such objective functions used depending on the type of data being worked on.

Numerical Analysis: Newton's Method

This would be the perfect place to start as a "stepping-stone" into the world of optimization. Newton's Method is the ancestor to optimization methods. Newton being the "Father of all Calculus," attempted to solve the problem of finding polynomial roots by using the gradient perspective. Over the year's Newtons method has been developed and found its uses in numerous applications across various domains. "It's role in optimization cannot be overestimated: the method is the basis for the most effective procedures in linear and nonlinear programming" (Polyak, 2007). Before going further, I would like to mention Eddie Woo, a famous mathematics teacher from Sydney, whose videos helped me understand these concepts in a very intuitive and exciting way. I encourage the reader to take a look at his material from the references (Woo, 2016).

The Newton's method after being developed and worked upon by multiple mathematicians over the years has proven to be a pivotal discovery for the world of optimization. This method employs the use of derivatives and the Pythagoras theorem. It was initially posed as a problem of solving the roots of polynomial functions. Later developments of the method gave way to other forms of the function rather than just looking at polynomials. In the context of optimization, it is the finding of the optimal value of the parameters of a function to get the closest approximation of the function value. It is in a recursive iterative numerical estimation procedure where one gets closer to the answer with more precision on every step.

Let us dive into the mathematical interpretation of this algorithm. Assume we have a polynomial function $f(x)$ where $x \epsilon \mathbb{R}$ in an nth degree polynomial. The problem at hand is to find the roots of this polynomial or the most precise estimate of the root that can be found, i.e., $x : f(x) = 0$. The method states that we can arrive at the root by continuously calculating x_{k+1} in the recursive formula.

$$x_{k+1} = x_k - f'(x_k)^{-1} \cdot f(x_k), \quad k = 0, 1, \dots \quad (Newton's\ method\ equation)$$

To gain an intuition, let us look at the geometric interpretation of this method on a graph. The first step is to assume an x_0 which would be the first value plugged into $f(x_0)$. It would be a known value greater than the root. x_0 would ideally be the point at which the value of $f(x_0)$ is positive such that any other value taken by x close to x_0 may result in a negative value of $f(x_0)$. Once we assume this x_0 value, we know that $f'(x_0)$ results in a gradient or *tangent* that intersects the x-axis as shown in Figure 1.

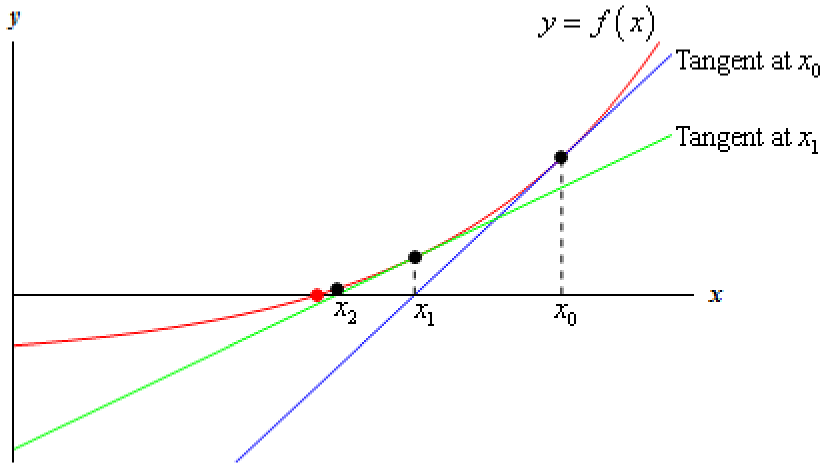

Figure 1: Netwon's Method (Paul's Notes)

Using the Pythagoras theorem on the right triangle formed between the tangent, the dotted line and the base (as shown in Figure 1) and the fact that the angle between the gradient and the x-axis is the slope itself, we arrive at the equation for Newton's method. Using this equation, we can now find x_1. Then we repeat the process, and we can see from Figure 1 that x gets closer and closer to the value of the root of $f(x)$.

This was one of the simplest and first uses of a gradient descent-like approach which is optimizing the parameter x to approach a certain value by iteratively updating it with newly acquired knowledge. While studying gradient descent methods we can find their basis in many parallels to the Newton's method. This is how we find many extensions and developments of this method branching into more global optimization algorithms based on gradients.

Gradient-Based Optimization: Gradient Descent

Any textbook on nonlinear optimization mentions that the gradient descent method was found due to Louis Augustin Cauchy, in his *Compte Rendu à l'Académie des Sciences* of October 18, 1847 (Lemaréchal, 2010). Cauchy was involved with voluminous astronomic calculations aimed at describing the motion of heavenly bodies. In his journal article on Cauchy, Lemaréchal gives a rough translation and interpretation of how Cauchy utilizes the earliest form of gradient descent (Lemaréchal, 2010).

We have our cost function to measure the errors and now we need to estimate the parameters in our model. For estimating our parameters and making them better through iterative processing we can use **Gradient Descent**. Gradient descent is one of the most popular algorithms used in machine learning for optimization. Various forms of gradient descent have become widely used and have proven to be robust forms of optimization. Gradient descent is a way to minimize the cost function $J(\theta)$ where θ is a vector of the model parameters to be involved in optimization. The parameters W are updated in the opposite direction of the gradient of the cost function given by $\nabla_{\theta} J(\theta)$. Progressing in the direction according to the gradient will get us closer and closer to our local minimum. Notice that we had encountered a similar iterative approach while studying the Newton method.

An important feature of gradient descent is the **Learning Rate** (η). The learning rate determines the size of steps we take to reach the local minimum and is an important factor to be considered when looking into the larger or more complex deep learning optimization problems. The learning rate governs the rate of convergence of the algorithm to the optimal values.

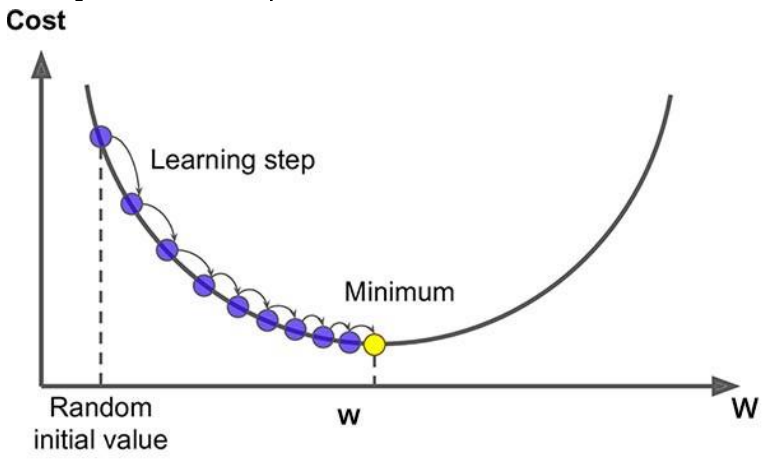

Figure 2. Gradient Descent

Figure 2 shows an illustration on how the gradient minimizes the cost function till it reaches the optimal point. The “learning step” is given by η and w gives the parameter value.

There are different ways in which we can implement gradient descent (Ruder, 2017):

Batch Gradient Descent

Batch Gradient Descent (aka Vanilla Gradient Descent) computes the gradient of the cost function with respect to the parameters for the entire training dataset. The parameter update step is given by

$$\theta new = \theta old - \eta \cdot \nabla J(\theta old).$$

Since the entire dataset is involved in calculating the gradient, this approach can be slow and complex due to the large scale of “real-world” datasets. This form of gradient descent also does not allow for ad-hoc updates with new data coming in which is also called *online learning*. This is a great algorithm for cases where the loss function is of a convex form which guarantees reaching the global minimum but reaches local minimum for non-convex loss functions. Figure 2. is a visual representation of the working of batch gradient descent (What Does Gradient Descent Actually Mean, 2020).

Stochastic Gradient Descent

After understanding the computational complexity behind calculating the gradient using every point in the dataset, we look at alternatives to overcome this issue. *Stochastic* in simple terms means random. A stochastic process is one wherein we are determining the randomness of a parameter. We do not want to calculate gradient for all points and at the same time. We need to keep the information about the data points that give the optimal state of the gradient intact. For this reason, we choose a random data point from the dataset on each iteration for computing the gradient which reduces the computation involved enormously. This is the working of Stochastic Gradient Descent (SGD).

Since the data point for calculating gradient is chosen at random the cost would fluctuate and jump across multiple values for a while before stabilizing to a minimum. This is good for making sure we reach the global minimum for non-convex loss functions because we frequently test across a range of points for the best gradient. However, there is no guarantee that we will reach the closest estimation of the optimal value using this approach. We may overshoot the exact estimate of the minimum. Therefore, to overcome this problem we need to adjust our learning rate as we approach a more stable state for the gradient. Figure 3 shows us how learning rate affects the descent to find the minimum (Jordan, 2018).

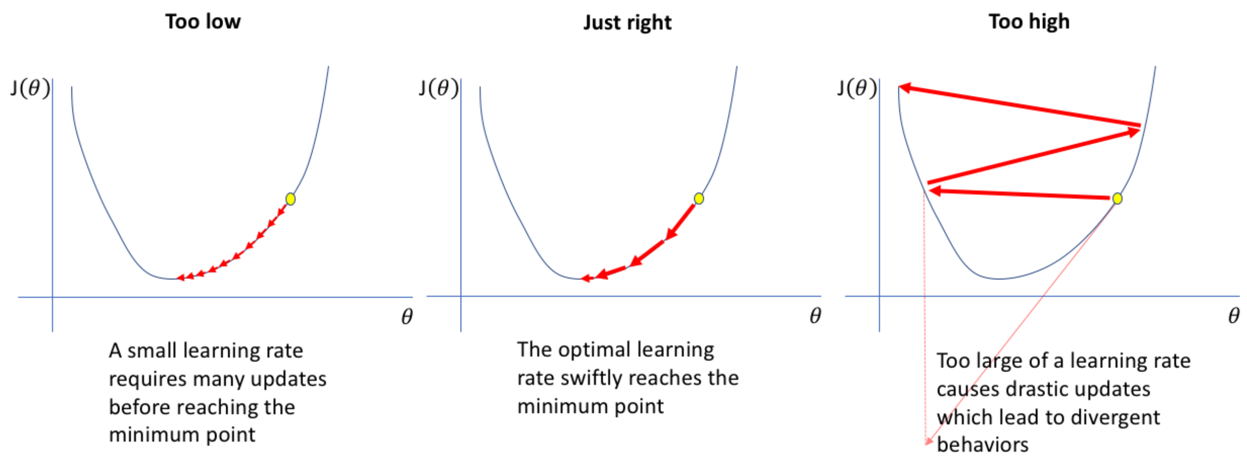
Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Figure 3. Finding the optimal learning rate.

The figure in the middle shows how we can reduce the learning rate while getting closer to the optimal minimum.

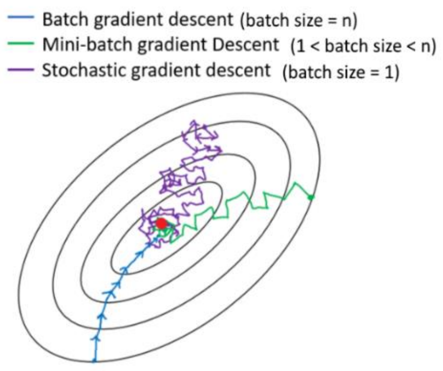
Batch gradient descent (batch size = n)
Mini-batch gradient Descent (1 < batch size < n)
Stochastic gradient descent (batch size = 1)

Figure 4. Showing here the path taken by the cost of the model for different variants of Gradient Descent.

Mini-Batch Gradient Descent

To get the advantages of both variants discussed for gradient descent and overcome the challenges in each of them, we can use a combination of both variants to build Mini-Batch Gradient Descent. This form uses smaller sets of random samples or "mini-batches" of data points from the dataset to compute the gradient. The batch size can be defined according to the size of the dataset.

Batch size is the number of data points used to train a model in each iteration. Choosing the right batch size is important to ensure convergence of the cost function and parameter values, and to the generalization of your model. Some research has considered how to make the choice, but there is no consensus (Katanforoosh, 2019). There is a balance to be attained depending on the available computational hardware and the task you are trying to achieve while choosing batch size.

Summary of Gradient Descent Optimization Algorithms

Newton's method breaks or has infeasible calculations involved to compute in practice for large complex high-dimensional data collections. Gradient descent is widely developed into multiple optimization algorithms for use. I think one of the simplest but most intuitive ways of describing optimization algorithms governed by gradient descent is the analogy of "a ball rolling down a hill" (Ruder, 2017).

It is very easy to understand that a ball rolling into a valley is how the gradient descent algorithm can be pictured. For SGD, imagine we dropped multiple balls at random locations and went to the valley where the balls ended up at the lowest of all valleys. SGD is only one of the more famous gradient descent optimization algorithms used. There are other such algorithms that are popular in various cases. I will try to quickly summarize them in this section using the analogy where relevant.

Momentum. This is the case where we push a ball down a hill and as it rolls down it accumulates momentum and accelerates. If I were to nudge the ball along the way in a different direction, because of its momentum, it would continue to move rapidly downhill towards the valley. Momentum increases for dimensions whose gradients point in the same directions and reduces otherwise. Hence, we gain faster convergence and reduce the oscillations.

Nesterov Accelerated Gradient (NAG). Imagine the ball was smarter. It had intelligence to reduce its speed upon detecting another hill approaching. NAG includes terms in the algorithm that do precisely this. It anticipates change in slope and reduces the rate of convergence. This avoids overshooting a minimum.

With the same aspect of intelligence from NAG, if we were to introduce terms to perform larger or smaller updates depending on the importance of individual parameters then we are using **Adagrad**. Adagrad uses a different learning rate for every parameter. **Adadelta** is an extension of Adagrad targeting some of its shortcomings regarding the accumulation of squared gradients which causes the learning rate to shrink too much. **RMSProp** was also built to combat the issue of reducing learning rate. Both Adadelta and RMSProp divide the learning rate by an exponentially decaying average of squared gradients. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients, like momentum. Where momentum is a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface. Adam is now the default optimizer in machine learning (Ruder, 2017).

A Study On Backpropagation

This is an attempt to briefly explain this method. Consider a simple example of taking logistic regression and applying it to a dataset for binary classification of a particular set of features. Essentially, using multiple logistic regressions to learn different attributes of various features with more and more details gives way for the usage of a neural network. In a neural network, each node in a layer has an activation function which in this case is the sigmoid function from logistic regression (with some bias included) that acts as the output function. The outputs of one layer are inputs to the next for the hidden layers. Features are learned in varying levels of detail across the network layers and are represented as the weights given to each connection between nodes across layers. The total loss is not computed for the outputs until we reach the last layer, so errors are accumulated throughout the forward progression of the network. To optimize these parameters and account for errors to update the weights appropriately, we can use mini-batch SGD. Such an algorithm would allow us to calculate the gradient of the current node's loss function. We also need to account for the accumulated effects of the gradient from previous layers. The process of calculating the gradients backwards, going from the current node to nodes in the previous layers to account for accumulated errors with the help of the chain rule, is called Backpropagation. After using backpropagation to calculate gradient, we update the weights for each layer.

Backpropagation was a revolutionary discovery and works robustly as a generalized optimization algorithm. Due to the availability of technology that we have today, a computationally heavy task like backpropagation can be used for massive datasets. Backpropagation is mostly now a mandatory step when we are dealing with feed-forward neural networks.

Backpropagation and SGD

Many a times SGD is often confused with Backpropagation due to the frequent use of the terms in different contexts. Backpropagation is an efficient method used for computing the gradients in neural networks that have a "feed-forward" nature. According to Czarnecki (2016), a research scientist at DeepMind, Backpropagation is not a learning method but just a complex implementation of chain rule of derivatives, which simply gives us the ability to compute all required partial derivatives in linear time in terms of the graph size. He goes on to say that SGD is the optimization method used here that is based on gradient of the cost function. Other optimization methods can be combined with Backpropagation if it is in the context of using a gradient or Jacobian. In loose terms, some still think it is okay to say "trained with backpropagation" for simplicity although we should be aware of the differences.

Problems with Backpropagation

Backpropagation can become quite complex to calculate and keep track of all the gradients as the neural network becomes "deeper". Even the well-known Andrew Ng, on his Coursera MOOC on Machine Learning, said that he had to keep revisiting this topic to fully gain intuition on its inner workings at the time of making his course (Ng, 2008). Grant Sanderson, in his video explaining Backpropagation, says that a lot of confusion arises due to the usage of many notations and symbols for representing different layers and variables in the equations involved (Sanderson, 2017). He goes on to explain the algorithm without using notations and relies on helpful visuals or animations for guiding the discussion.

From an algorithmic analysis perspective, we aim to look at the best- and worst-case scenarios of an algorithm before deciding how to optimize it. Let us look at some of the limitations of backprop.

Vanishing Gradients: Sigmoid and tanh non-linearities tend to saturate and entirely stop learning. This happens when the neuron outputs are at the extremes, causing the gradients to be 0 or close to 0, i.e., vanish. As more layers using certain activation functions are added to neural networks, the gradients of the loss function approach zero, making the network hard to train.

Exploding Gradients: In deep networks or RNNs, error gradients can accumulate during an update and result in very large gradients. These in turn result in large updates, to the network weights. At an extreme, the values of weights can become so large as to overflow reserved memory capacity.

There are also other problems with “backprop” (Backpropagation). It is biologically implausible. If neural networks are supposed to be analogs of biological neural networks then this analogy breaks down, or is at best very strained, if you use backprop because biological networks do not compute exact gradients and perform anything that is explicitly described as “back propagation” (James, 2017). There are also practical difficulties in applying backprop.

Backprop optimizes weights for a fixed target. Neural networks in this way are trained to excel at a predetermined task and the weights are fixed once trained. There is no room for additional learning to take place. Neural networks pride themselves in being generalized approaches that learn on all variations of the data before being deployed as trained models.

The use of gradient based models leads to loss of interpretability in complex neural architectures. It becomes tougher to explain why the weights have converged to their respective values for each of the nodes after optimization. This problem of interpretability has been prominent in deep learning as a whole, often resulting in it being treated as a “black box”.

Further Studies

Is Backpropagation (SGD) the only way?

For an algorithm to replace Backprop, it must solve the vanishing and exploding gradient problems, be computationally faster, converge faster, preferably reduce hyperparameters, work for multiple domains and most importantly be biologically plausible (Sally Robotics, 2020). There is still research being done for a generalized algorithm that could be better than Backpropagation. Although we do have some alternatives that could be used but work well in specific domains while failing to show the same results when applied in other cases. There are a couple of papers that discuss about some alternative approaches (Sally Robotics, 2020). Due to the excessive length of this essay, I have decided to leave out the details on those alternatives for future studies as they require extensive descriptions and further study in mathematical depth.

Although, to get a glimpse, **Evolutionary Algorithms** (aka Evolution Strategies, **ES**) seem to be on the rise in popular discussion among literature. According to OpenAI, early versions of these algorithms were inspired by the theory behind the biological evolution of dominant species. However, the mathematical details are so heavily abstracted that its best to think of ES as simply a class of black-box stochastic optimization techniques (Evolution Strategies as a Scalable Alternative to Reinforcement Learning, 2017). It is an optimization algorithm that works by selecting, say, 100 random directions in the parameter space "close" to the current parameter vector. This provides 100 candidate parameter vectors which can be evaluated. Then the 100 parameter vectors are combined by a weighted average with the weights being proportional to the reward each vector received. We can see that is likely to move the network in the direction of doing better.

Intuition probably would lead us to believe convergence would be incredibly slow - close to undetectable in reasonable computation. What is surprising is that it works and works very efficiently (James, 2017). However, the most important information about ES is that it is only effective in reinforcement learning. In supervised learning, where a trainer provides the correct target and where it is possible to compute an exact error, backprop is much faster than ES.

I would like to close this discussion with an excerpt from the conclusion of a research project that used Neuroevolution and Genetic Algorithms (GA) which come under the broader scope of ES:

"…We also showed that interesting algorithms developed in the neuroevolutionary community can now immediately be tested with deep neural networks, by showing that a Deep GA-powered novelty search can solve a deceptive Atari-scale game. It will be interesting to see future research investigate the potential and limits of GAs, especially when combined with other techniques known to Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning improve GA performance. More generally, our results continue the story – started by backprop and extended with ES – that old, simple algorithms plus modern amounts of computation can perform amazingly well. That raises the question of what other old algorithms should be revisited." – (Such, et al., 2018)

Conclusion

Optimization in hardware is racing with optimization in software. With Moore's Law holding true, hardware technology has been evolving at an exponential rate. We have seen such rapid advancements in compute power that were unfathomable decades ago. This power is being distributed to the fingertips of millions all over the world. These advancements are enough to eliminate what were once optimization problems as we have enough compute power to perform behemoth tasks.

On a closing note, we were able to cover much ground regarding our topic of discussion. We can also draw parallels between the generations of optimization mentioned and the different algorithms we have seen. Understanding how things started from Newton's method to gradient descent will help gain a better picture of how we formulate standard approaches to optimization. Due to the rapidly evolving hardware and the robustness of the algorithm, backprop has still not been dethroned and continues to surprise experts as a universal preference in common deep learning problems. I aim to further study alternatives to backprop and read the aforementioned papers in depth.

Exploring optimization methods and hyperparameter values can help one build intuition for optimizing machine learning tasks. During hyperparameter tuning, it is important to understand intuitively the optimization's sensitivity to learning rate, batch size, cost function, and so on based on experimentation. This intuitive understanding combined with the right method will help find the optimal model to fit the data.

References

Antoniou, A., & Lu, W.-S. (2007). *Practical Optimization: Algorithms and Engineering Applications.* Springer Science and Business Media.

Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization Methods for Large-Scale Machine Learning. *Society for Industrial and Applied Mathematics, 60*(2). Retrieved from https://coral.ise.lehigh.edu/frankecurtis/files/papers/BottCurtNoce18.pdf

Czarnecki, W. (2016, Jun 21). *What is the difference between SGD and back-propagation?* Retrieved from stackoverflow: https://stackoverflow.com/questions/37953585/what-is-the-difference-between-sgd-and-back-propagation#:~:text=Stochastic%20gradient%20descent%20(SGD)%20is,to%20minimize%20a%20loss%20function.&text=To%20do%20so%2C%20SGD%20needs,%22gradient%22%20that%20SGD%2

James, M. (2017, April 5). *Evolution Is Better Than Backprop?* Retrieved from I Programmer: https://www.i-programmer.info/news/105-artificial-intelligence/10659-evolution-is-better-than-backprop.html

Jordan, J. (2018). *Setting the learning rate of your neural network.* Retrieved from Jeremy Jordan.

Katanforoosh, K. (2019). *Parameter optimization in neural networks*. Retrieved from deeplearning.ai: https://www.deeplearning.ai/ai-notes/optimization/

Larsson, S. (2019). *Possible for batch size of neural network to be too small?* Retrieved from stackexchange.

Lemaréchal, C. (2010). Cauchy and The Gradient Method. *Documenta Math*. Retrieved from https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf

Ng, A. (2008). *MAcine Learning* . Retrieved from Coursera: https://www.coursera.org/learn/machine-learning/home/welcome

OpenAI. (2017, March 24). *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. Retrieved from openai.com: https://openai.com/blog/evolution-strategies/

Paul's Notes. (n.d.). Retrieved from tutorial.math.lamar: https://tutorial.math.lamar.edu/classes/calci/newtonsmethod.aspx

Polyak, B. T. (2007). Newton's method and its use in optimization. *European Journal of Operational Research*.

Ruder, S. (2017, September 15). *An overview of gradient descent optimization algorithms.* Retrieved from arxiv.org: https://arxiv.org/abs/1609.04747

Sally Robotics. (2020, August 25). *Backpropagation and its Alternatives*. Retrieved from medium.com: https://medium.com/@sallyrobotics.blog/backpropagation-and-its-alternatives-c09d306aae4chttps://medium.com/@sallyrobotics.blog/backpropagation-and-its-alternatives-c09d306aae4c

Sanderson, G. (2017, November 3). What is backpropagation really doing? | Deep learning, chapter 3. Retrieved from https://youtu.be/Ilg3gGewQ5U

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., & Clune, J. (2018, April 20). Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for. Retrieved from https://arxiv.org/pdf/1712.06567.pdf

Talbi, E.-G. (2009). *Metaheuristics: From Design to Implementation.*

Wang, G. (2018, December 7). *Brief History of Optimization*. Retrieved from empowerops.com: https://empowerops.com/en/blogs/2018/12/6/brief-history-of-optimization

What Does Gradient Descent Actually Mean. (2020). Retrieved from AnalyticsVidhya.

Woo, E. (2016). Retrieved from Youtube: https://www.youtube.com/watch?v=j6ikEASjbWE