The Fly & Anti-Fly Missile

Rick Tilley
Florida State University (USA)
rt05c@my.fsu.edu

Abstract

Linear Regression with Gradient Descent are used in many machine learning applications. The algorithms are visually based in a dimensional coordinate system. The dimensional space can also involve not so visually orientated data. Deep-learning involving neural networks use similar weight algorithms as a main basis for artificial neurons. Usually neural networks use many neurons in sequence that each produce weights or weighted output that can be fed into the next neuron, and so on. Since a lot of the learning material was ambiguous on the "controlling" neurons that determine the flow and neural use, I was interested in seeing how intuitive it would be for me. I wanted to focus mostly on linear regression in a single neuron. The program I made does a linear regression of a 2D data plot, uses this to estimate a future position, and then does a second linear regression of an intercept path from a 2rd party location. This could be viewed as 3 neurons. 2 linear regression neurons and 1 neuron in-between that calculates an intercept. To apply this, I am simulating a fly as one party, and an anti-fly missile as a second party. The fly is avoiding the missile, and the missile is chasing the fly.

Introduction

There are many machine learning methods in AI including clustering, regression, and logic. Neural networks can use many techniques, but the most foundational and of interest for this article is linear regression. This could also include logistic regression as the method uses only a different cost formula. Specifically, I am interested in using gradient descent as a method of solving the regression. Machine learning has been used for robotic imitation and learning movement, natural language processing, optical character recognition, and many other uses. Deep learning is used by combining layers of artificial neurons, and controlling neural connections by other coding neurons (Haiqin Yang, 2010). The non-coding neurons have input and output values. The I/O can chain with other neurons allowing for multiple processing of the same weighted values. Each non-coding neuron may perform a wide range of functions. Often that function is some sort of gradient regression algorithm. Coding neurons on the other hand can turn on and off non-coding neurons to control logic flow and determine neural pathways for processing of the data (Alexander Grubb, 2010).

Background

Learning methods often have to deal with a large dimensionality of data. Regression can be used on practically anything, and often certain features of the data can be used rather than the entire data (Mahdi Milani Fard, 2012). Often the data is sparse, meaning the

graphed location of the data falls in clusters, and a large portion of the domain is void of data (Haiqin Yang, 2010). Often, such as in (Mahdi Milani Fard, 2012), the data is broken into equal training and testing segments. This allows training data to be reused and tested against itself in multiple folds, making training more efficient on the amount of data available. In a layered neural network, many of the neurons are performing some sort of gradient regression. This method is very useful in minimizing losses between predicted and actual weights (Ofer Meshi, 2010). These chained neurons are composed of inputs and outputs to create the neural linking aspect. There are other transforming modules that turn on and off the worker neurons, which can change neural flow of input/output connections (Alexander Grubb, 2010). Optimizations can be performed by the transforming modules, and even back-propagation of gradient data for dormant neurons can be used to speed up the system (Alexander Grubb, 2010). An important detail that must be addressed is the perils of over fitting. If the regression works too hard to match the exact form of data, it may be fitting to a specific instance that is much different than the general rule of the data form. Because of this, it is usually better for regression formulas to limit fitting and give a more general regression line. Gradient descent algorithms are a method of supervised learning because they are trying to match a pattern. Linear regression formulas can also solve in an unsupervised environment. The line is fit based on the mean values over all the data points. In gradient descent, the algorithm must check the divergence from its predictions to an actual expected result. It works in a loop by slowly incrementing or decrementing the weight values until it to a local minima. This local minima should represent a good weighted regression fit.

Gradient descent involves iterations over a sub-gradient, using a differential function of cost (Haiqin Yang, 2010). The descent starts at a random weight selection, and checks the cost derivative. This tells the algorithm if the chosen weight is converging or diverging from a local minimum. The weight is adjusted accordingly, modified by only a fraction of the cost derivative. This iterates for a maximum amount of iterations, or until the cost differential drops below a certain threshold. Assuming the latter is true, the chosen weights should now make a good regression line. This can be applied in logistic regression to divide two cluster categories in a similar manner. Other factors such as error bounds and n-vector bias-variance can be used to ensure successful descent (Mahdi Milani Fard, 2012). The cost function affects the type of regression, and a derivative of that function is essential for gradient descent to work. It is also useful to convert the data being regressed to the least dimensions. This can be done through rotations (Minyoung Kim, 2010). A 2D scatter plot can be rotated so that each plot is seen as a distance from its rotation axis. This makes the data much easier to analyze as we can use a $y=f(x)$ rather than calculating for two independent variables.

Fly vs. Anti-Fly

Because linear regression works best in a visual sense, I chose to use regressions on trajectories on a 2D space. In this case, the data is the past n data points of an entity's trajectory. I have two

entities, a Fly and an Anti-Fly Missile. Both operate the same way, except one tries to maximize distance and the other tries to minimize distance. Both the fly and the missile start out flying in random directions. At each time-"round", all entities move based on their weighted trajectory, and their attention counter is decremented. Once the entity's attention reaches zero, it is reset to a default value, and the entity determines a new trajectory based on the opponent. The entities continue to fly around a 2D space until the user ends the program. Should the distance between the two entities reach a collision minimum, the missile blows up the fly and the simulation is over.

Determining a new trajectory is done with 3 neurons. The first neuron loads in the opponent's last n positions and does a linear regression fit to the data. These weights are passed to a second neuron that takes the distance between both entities and based on the "self" entities current speed, estimates the location of its opponent in the time it would take to travel between both entities currently. This is the self's destination. In the case of the missile this is straight forward. The fly however practices avoidance, so the destination is estimated as a greater distance from the opponent's future position rather than towards it. It then draws a line of n_max data points between self's current position and its destination. This line is modified by minimal random noise to prevent a perfect fit. The data points are then passed to a 3rd neuron that performs another linear regression of the data. The weights found are then used for the self's new trajectory. Their attention counter has been set to non-zero, and they will travel in their new trajectory until the next time their attention counter reaches zero.

Anti-Fly in C++

Coding this was more problematic than I first thought. At first I tried a gradient descent linear regression algorithm on separate dimensions X and Y, and used a third foundational variable T for time. I had problems with the cost function so I reduced gradient descent to 1 dimension where y = f(x). I did this by rotating by the start and end points, and calculating the regression based off this rotation. This worked better, but I still had problems with my cost function, possibly due to scaling issues. I am currently using the solved linear regression formula to do the linear regression and using offsets only for X and Y based on trajectory changes. Both neurons 1 and 3 are the same linear regression function, and are pretty straight forward. Neuron 2 is part of my pickpath() function and is hardcoded. It takes the weights from neuron 1 and predicts the future location, picks a destination of there, or away from there, and plots a course for the destination. It passes the plotted course to neuron 3 that simply does a regression on these plots to determine the new path to take. Because this is to work on a remote Linux server using G++, I did not use any graphics libraries. Instead the pixel locations of each entity and their distance is printed out as plain text. For testing purposes I also printed out each entity's actual weights.

Findings of this Program

Due to time constraints and the difficulties of this material, I was unable to get my original plan operational. The gradient descent algorithm looked perfect functionally, but almost never

worked. The cost function would sky-rocket to infinity and cause my weights to overflow. I tried various normalizations, and the other outcome was to have my weights always converge to a very small number ($\sim 10^{-13}$) every time. There was no middle ground so in interest of getting a working prototype, I used the algebraic solution for linear regression. Whereas in the previous methods I was doing rotation, this formula doesn't require rotations. I was able to pass the weights to the second neural stage, where it accurately calculates a trajectory from the entities current position to a prediction destination. It then does a 3rd stage of linear regression against this new path, and assigns the entity new weights for this path. As a result, each time pickpath() is called for an entity, it changes directions. I still had a problem with regression picking weights that were too big after a while. I see this as a problem of attempting to accelerate to a destination with no physical constraints. This could cause entities to shift locations unnaturally at unrealistic speeds, even transporting suddenly to new locations. I used a normalization constraint on my y-weight so that it was never greater than ± 5. This allows the two entities to fly around each other without accelerating away too rapidly.

The user interface is very simple. I would like to have had a graphic animation of the two entities in a 2D space, but limited myself text based output in this project. The output is as follows:

T: 1 distance = 6.13418
Fly:
{ 3.07564, -2.47673 }
{ [w0:-0.17::w1:-0.75::deg:286 or 4.99164] }
Missile:
{ -2.86163, -0.934966 }
{ [w0:0.41::w1:0.47::deg:296 or 5.16617] }
Continue? [y,n]:

The T: 1 corresponds to time = 1. Time is an integer value corresponding to rounds. Each round increments time by 1. Distance = 6.13418 is the distance between the two entities. The two entities are listed in the same format. After Fly: the first two numbers are the current (x,y) coordinate locations of the entity. The rest is testing output: w0 and w1 are the actual weight values for the entity, and deg: gives degrees and radians (*d or r*). The prompt simply takes a dummy string input, and anything other than 'n' continues the loop another round. This will continue until you exit, the program will not stop upon collision, although this could be easily introduced. For now I am just paying attention to distance when testing. Future versions will have a graphical interface, which is more ideal for evaluating this sort of data.

Conclusion

Although the theory is fairly straightforward and I understand all the methods involved in this project, getting it to actually work has been quite a challenge. I feel that I have re-coded this program 3 or 4 times, and even at its current state, it is debatable whether the algorithms are giving optimal results. If this project has taught me anything, it is the difficulties of simple algorithms that have powerful machine learning applications in application. When applying them to real world situations, data analysis becomes more complicated than expected, and issues of normalization, scale, and bounds come into play. I don't believe over-fitting

was an issue, but under fitting may have cause problems since objects are changing direction frequently. Getting real life data to conform to results I have experienced previously, from our pre-generated environment in homework assignments, has not been a piece of cake. I plan to revisit this project in the future on my own to see how things can be improved. I would like to see a graphic animation as the output for this program. I would also like to experiment more with gradient descents and perhaps use Bezier curves to simulate smoother directional changes rather than abrupt changing linear vectors. My findings suggest also that regression on observations are useful, but when using regression on a theoretical trajectory, the regression will make up its own velocity - which is very unrealistic to the limits of the physical world. Perhaps using Bezier curves and applying weights to curves rather than formulas could reduce velocity anomalies. Moreover this project has taught me the importance of fitting your algorithms to your data, and dealing with relative locations and rotations. There are a million ways to do this, and often the simplest methods are the most efficient. I believe the answer is to use more neurons and use additional neurons to analyze the data and deal with logical events.

References

Alexander Grubb, J. A. (2010). Boosted Backpropogation Learning for Traning Deep Modular Networks - Paper Id: 451. *International Conference on Machine Learning*.

Haiqin Yang, Z. X. (2010). Online Learning for Group Lasso - Paper Id: 473. *International Conference on Machine Learning*.

Mahdi Milani Fard, Y. G. (2012). Compressed Least-Squares Regression on Sparse Spaces. *Conference on Artificial Intelligence (AAAI) - 26th*.

Minyoung Kim, F. D. (2010). Local Minima Embedding - Paper Id: 374. *International Conference on Machine Learning*.

Ofer Meshi, D. S. (2010). Learning Efficiently with Approximate Inference via Dual Losses - Paper Id: 587. *International Conference on Machine Learning*.