Using Swarm Intelligence Algorithms to Optimize Text Classification Neural Networks

Jakob Germann Keegan Kerns

Abstract

Previous studies have shown an increase in convergence speed toward global optimums compared to Back-propagation on neural networks applied to non-linear functions. Here we aim to report the effectiveness of using Swarm Optimization algorithms as a method of training neural networks for Natural Language Processing applications. To understand the potential of swarm algorithms in neural networks, we replace Back-propagation with two similar swarming algorithms and apply them to a network used for sentiment analysis.

1 Introduction

Natural Language Processing is a popular field that has much potential for Neural Network models to use in different applications. One of the major difficulties when working with such systems is optimizing a model to maximize performance metrics.

In this experiment, the goal is to apply swarm based optimizers to a recurrent neural network, the LSTM model, and analyze the model's performance in comparison to the more traditional Back-propagation algorithm. We hypothesize that this new training method will provide better performance to the neural network because the use of particles will result in faster convergence to global optimums over Back-propagation. The experiment was conducted in a Google Colaboratory environment and written in Python 3. For the sake of time, we use Keras and its respective classes to create the LSTM and simplify integration of the training algorithm.

2 Sentiment Analysis

Sentiment analysis is a Natural Language Processing technique that uses machine learning to classify a document between two or more classes. Several traditional models can be applied to solve this problem depending on the complexity. Some popular baseline models include Multinomial Naive Bayes, N-Grams, and Hidden Markov Models. These models rely on probabilistic calculations to make their predictions making them relatively well suited for certain data sets. However, these probability based models rely on some assumptions that may not always be true which will reduce their performance. As such, Neural Networks have become another popular method of handling sentiment analysis problems. Recurrent Neural Networks and other similar models have some advantages over their more classical cousins, such as being able to handle far more complex data sets, especially those that are not linearly separable, or easily classifiable by traditional models.

3 Model and Data Selection

To test the swarm optimization algorithms, we decided to train a Long-Short Term Memory model on the IMDB Movie Review Dataset.

3.1 Long-Short Term Memory

A RNN is a type of neural network that forms connections between its nodes. This can be a directed or non-directed graph which allows for the recurrent neural network to take advantage of previously established knowledge and relate it to current incoming data. Essentially it creates context for information that passes through each node. The Long-Short Term Memory model, which is what we use for this experiment, is a more advanced and popular variant of the RNN.

Long-short term memory (LSTM) models take advantage of this property and specializes the structure of the nodes in the neural network. In utilizing this property, they are able to learn long term dependencies. Unlike an RNN, the repeating module is special to the LSTM. They share the same chain like node structure but the node is what makes the LSTM unique to the standard RNN. The reason that the LSTM node unique is that in a LSTM neuron there are four network layers at work instead of one. One of the major issues that accompanies a normal RNN is a vanishing gradient problem. This problem means as the number of epochs increases, the value of the last layer that is used during back-propagation becomes almost irrelevant. This can cause problems during training and lead to major inaccuracies.

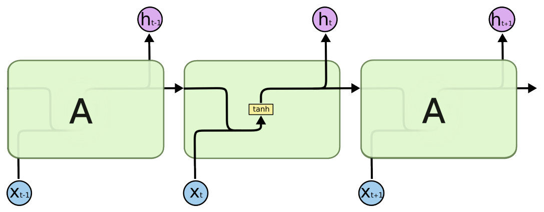

Figure 1: Depiction of a simple RNN with single network layer node. (1)

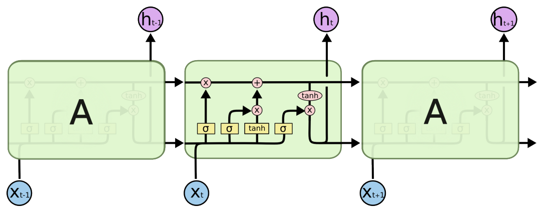

Figure 2: Depiction of a LSTM with four network layer nodes. (1)

For LSTM to combat the vanishing gradient, the nodes make use of three different gates. These gates also make the nodes able to retain their long term memory of dependencies. First of the gates is the forget gate. The forget gate is to release the previously stored memory in case of a change in subject focus or a pattern in the incoming data (4). The mathematical expression for the forget gates operation is shown below. (4)

$$\Gamma_f^t = \sigma(W_f[a^{t-1}, x^t + b_u])$$

The second of the gates is known as the update gate. The update gate is also known as an input gate. This gate takes in content and updates the subject. For this to occur, new vectors are needed to be applied to the last state since the last state is a vector as well. Three mathematical expressions are need to accomplish this task. By modifying $\tilde{C}_t$, which is the current memory value.

(4)

$$\Gamma_u^t = tanh(W_u[a^{t-1}, x^t] + b_u)$$

$$\tilde{C}^t = tanh(W_c[a^{t-1}, x^t] + b_c)$$

$$C^t = \Gamma_f^t * c^{t-1} + \gamma_u^t * \tilde{C}^t$$

Finally there is the output gate. The output gate decides which output should be processed by the next two equations before being passed out of the node. (4)

$$\Gamma_o^t = \sigma(W_o[a^{t-1}, x^t] + b_0)$$

$$a^t = \Gamma_o^t \times tanh(c^t)$$

In the world of Natural Language Processing, it is important to discern which types of vectors we will be using in the LSTM. There are dense and sparse vectors. Each are designed to handle different scenarios. Dense vectors are short (roughly 50-100 elements in length) where most elements are non-zero values. These vectors are better at defining features since there are less weights to tune, and are better at generalizing. Sparse vectors are more suited for handling synonyms and more commonly used in neural networks that conduct Natural Language Processing tasks.

3.2 IMDB Dataset

The data we chose for this project is the IMDB Movie Review Dataset (5). It is a collection of approximately 25,000 movie reviews split between a train and test set. Each review is assigned to a directory depending on if it is classified as positive or negative. We decided to use this dataset because it is an effective and popular dataset for testing models on baseline data before migrating to more advanced data. Due to storage constraints, the amount of data in the train and test sets was cut in half, resulting in 6000 positive and negative instances for each set. From that set we grab 3000 random samples where half are positive and half are negative. For the test data, we remove 30% of the training data.

4 LSTM and Sentiment Analysis

Previously it was discussed on how the LSTM model functions structurally. Now it is relevant to relate it to the Natural Language processing task of sentiment analysis. Sentiment analysis is the classification of what a document, review, tweet, or any piece of text intends to display as a sentiment. This means that it can be positive, negative, hurtful, sarcastic etc. Since the basis of sentiment analysis is a classification process, we can apply LSTM to try and identify the sentiment of a given piece of text.

First, we need to normalize and tokenize the data. Each unique word should be isolated as a keyword and the number of its occurrences should be taken into account to establish a pattern. This pattern will be associated with the given labels from the dataset to determine that a given token occurs more when positive or negative.

5 Classical Optimization Methods

5.1 How Optimizers Works

The purpose of an optimizer is to manipulate data to its most accurate form. To determine the amount of data the optimizer needs to navigate the gradient depends on the loss function. A loss function acts as a guide to show the optimizer what to change and if it needs to change back if loss is heavily altered. Once the loss is minimized the optimizer returns the data in a cleaner and accurate form. A key thing to remember about all optimizers is that their goal is to minimize the loss that is generated.

5.2 Adam Optimizer

The acronym Adam stands for Adaptive Moment Estimation. Unlike its counterpart, gradient descent, Adam implements the concept of momentum by taking parts of old gradients and applying them to current gradients. Adam is one of the most popular optimizers used in Artificial Intelligence and will serve as our benchmark for performance measurement. (6)

5.3 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is very similar in operation to gradient descent itself. However, gradient descent computes gradients for all the training examples where as SGD looks at batches or a random subset of examples from the training data. This is used to prevent over fitting and also reduces the chance of hitting a local minima. (6)

6 Proposed Optimization Methods

Below, we describe our optimization method. We begin by discussing Particle Swarm Optimization and its properties, provide a brief introduction to Grey Wolf optimization, then explain our implementation and the results gained. The algorithm for the PSO variant we used is provided after the discussion.

6.1 Particle Swarm Optimization

The PSO algorithm is one of the most popular swarm optimization algorithms in use today. It is a nature-inspired algorithm that uses a pre-defined population of simple particles to search an N-dimensional space. The search space is defined by some objective function, which the particles traverse and interact with each other to find the optimal point. Individual particles contain no intelligence or significant properties and are only defined by a current position, personal best position, and velocity vector. The overall swarm tracks the global best position of all particles and takes this into account when updating particle parameters. (9)

6.1.1 PSO Particles

Particle information is stored in matrices and iterated through to update each particle's information. (9)

$$P = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \qquad V = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Where we define position *p* and velocity *v* as, (9)

$$p_i = (d_1(t), d_2(t), d_3(t), \dots, d_k(t))$$
$$v_i = (d_1(t), d_2(t), d_3(t), \dots, d_k(t))$$

We further define *n* as the number of particles in the swarm, *t* as the current time step out of *T* time-steps, and *k* as the number of dimensions in the search space. The goal is to find an emergent intelligence from the collection of particles that will find a global optimum. The algorithm can be configured to find different optimums depending on context, such as a global minimum or global maximum. (9)

6.1.2 Update Functions

When initialized, each particle is assigned a random position in the search space. These particles update their positions using the velocity vector which is also updated. Both functions are repeated for T iterations, or epochs.

Below is the update function for a particle's position. (9)

$$p_i(t+1) = p_i(t) + v_i(t+1)$$

Prior to updating the position of the next time step, $p_i(t+1)$, we must update the velocity vector. (9)

$$v_i(t+1) = v_i(t) + c_1 r_1(t)(y_i(t) - p_i(t)) \\ + c_2 r_2(t)(\hat{y}(t) - x_i(t))$$

where c_1, c_2 are "acceleration" constants to help scale each particle's influence on the entire swarm, $r_1(t), r_2(t)$ are random values from a uniform distribution between $[0, 1]$ to make PSO more stochastic in nature, y_i is the personal best position of the current particle, and $\hat{y}_i$ is the global best position of all particles in the swarm. (9)

6.1.3 Personal and Global Best Positions

When searching for a global optimum, we want to continually track the progress of particles as they conduct their search. Over time, we will find that a particle's position is in fact the closest thus far to the approximate optimum. We want to store these positions and use them to update each particle's movement so they begin to converge toward the global optimum we're looking for. (9)

Part of the PSO algorithm is to continually track each particle's movement history and as such, remember the best position it ever visited. The algorithm continually compares this position with each position the particle visits. These positions are passed to the objective function and evaluated, the results of which are defined below. (9) (12)

$$y_i(t+1) = \begin{cases} y_i(t), & \text{if } f(x_i(t+1)) \geq f(y_i(t)). \\ x_i(t+1), & \text{if } f(x_i(t+1)) < f(y_i(t). \end{cases}$$

where f is the fitness function that each particle position is evaluated against. As the particles traverse the search space throughout the time steps, we extract the global best position from all personal best positions and use this in future updates. (9)

$$\hat{y}_i = \min \{f(y_0(t)), \ldots, f(y_n(t))\}$$

where n is the number of particles in the swarm.

6.1.4 Global vs. Local PSO

There are also two distinct forms of classical Particle Swarm Optimization. The most commonly used is Global PSO, or G-PSO, which has the particle velocity update function use a global best that is defined for all particles in the swarm. Alternatively, there is also Local-PSO which introduces the concept of particle neighborhoods. In this instance, global best positions are replaced by local best, each contained to a single neighborhood of particles that all influence each other. Both the global and local algorithms are similar in that their inter-agent update functions for the velocity components cause particles to shift toward the global optimum over time. However the key distinction is that the global version converges faster toward an optimum whereas the local version has greater coverage over the entire search space, resulting in decreased likelihood of becoming trapped by local optima. For the purposes of this paper, we use both the Global and Local PSO algorithms to determine which variant has better convergence given the vast amount of parameters. (9)

For local PSO we update the particles' velocity using the best position found by the local neighborhoods. (9)

$$v_i(t+1) = v_i(t) + c_1 r_1(t)(y_i(t) - x_i(t)) \\ + c_2 r_2(t)(\hat{y}_i(t) - x_i(t))$$

where $\hat{y}_i$ is the local best position found by neighborhood which is defined as (9)

$$\hat{y}_i(t+1) \in N_i | f(\hat{y}_i(t+1)) = \min f(x)$$

Below we introduce the pseudocode for Global and Local PSO.

Algorithm 1 Global PSO (9)

```
Require: $N > 0$
  while Stopping condition is not true do
    for each particle  $i = 1, \ldots, N$ do
      if $f(x_i) < f(y_i)$ then
        $y_i = x_i$
      end if
      if $f(y_i) < f(\hat{y})$ then
        $\hat{y} = y_i$
      end if
    end for
    for each particle  $i = 1, \ldots, N$ do
      $v_i = \text{vUpdate}(v_i)$
      $p_i = \text{pUpdate}(p_i)$
    end for
  end while
```

6.1.5 PSO Implementation

By default, Keras uses Back-propagation in tandem with optimizers like SGD or Adam to update a neural network. Back-propagation is an algorithm

Algorithm 2 Local PSO (9)

```
Require: $N > 0$
  while Stopping condition is not true do
    for each particle $i = 1, \ldots, N$ do
      if $f(x_i) < f(y_i)$ then
        $y_i = x_i$
      end if
      if $f(y_i) < f(\hat{y}_i)$ then
        $\hat{y} = y_i$
      end if
    end for
    for each particle $i = 1, \ldots, N$ do
      $v_i = \text{vUpdate}(v_i)$
      $p_i = \text{pUpdate}(p_i)$
    end for
  end while
```

that updates the gradients of individual parameters within the network as a form of feedback. The re-calculation of this gradient is what allows the network to increase in accuracy over time. However, Particle Swarm Optimization does not use any gradient in its calculations. Thus, our implementation required us to restructure the neural network so PSO would be compatible to optimize the new weights of the network. (8)

6.2 Gray Wolf Optimization

Gray Wolf Optimization (GWO) is a particle-based optimization algorithm similar in structure to PSO. It uses particles that converge to a global optimum over time. Unfortunately, we were unable to properly implement the algorithm due to incompatibility issues that could not be resolved in the given time. For future investigation, we have included an introduction to the topic. The distinction from PSO is that GWO introduces a leadership hierarchy to the particles which significantly changes the update functions and the amount of influence particles have on each other. The algorithm was originally proposed in 2014 and sought to emulate the hierarchy and hunting behaviors of Grey Wolf packs. The cornerstone of GWO is that three particles within the swarm each have a higher 'rank' than all other particles in the swarm, and thus have greater influence on them. These leader particles will then lead the lower ranked members to the global optimum, emulating the hunting behavior over several time steps. (10)

7 Results

Note that both tables have a learning rate of 0.001 and a batch size of 120.

Back Propagation		
Optimizer	Accuracy	Loss
Adam	0.53	0.98
SGD	0.52	0.69

Particle Swarm Implementation	
Accuracy	Loss
0.508	1.18

The current parameters showed that Adam and SGD performed more favourably when using Back-propagation rather than PSO. We expect this outcome because Particle Swarm Optimization is standalone in its training mechanism whereas Back-propagation works with Adam or SGD to update the network gradients.

8 Division of Labor

Between the two contributors Keegan Kerns and Jakob Germann, the work needed to be shared. Keegan handled the implementation of the LSTM model, and parsing the dataset. Whereas Jakob handled the formatting of the dataset and the swarm optimizers. Both members had to conduct a great deal of research to get the source code to where it needed to be. There was a lot of time committed to tuning the model since there was issues using base configurations.

9 Difficulties

Initially, there were extreme difficulties integrating Particle Swarm Optimization with traditional neural networks. This became apparent when we found that PSO is incompatible with gradient based training methods like Back-propagation. Thus, we refocused our efforts to replacing Back-propagation entirely with our own training function based on PSO. However, we found that they also yielded poor results for the validation data. Due to time constraints, we were unable to effectively fine-tune the model to acceptable metrics. Furthermore, excessive debugging also prevented us from completing our implementation of the Grey Wolf optimizer. We also would require more time to fine-tune the parameters of both Particle Swarm Optimization and Grey Wolf Optimization once implemented.

10 Conclusions

In this paper, we proposed the use of two swarm algorithms as a method for training LSTM variants of Recurrent Neural Networks for sentiment analysis problems. Gained results demonstrate that both versions of Particle Swarm Optimization perform less efficiently than Back-propagation. Given more time, we believe the algorithm and model parameters could be fine-tuned to provide competitive results.

References

[1] Olah, C., 2015. Understanding LSTM Networks – colah's blog. [online] Colah.github.io. Available at: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> [Accessed 26 April 2022].

[2] Analytics Vidhya. 2022. NLP | Sentiment Analysis using LSTM - Analytics Vidhya. [online] Available at: <https://www.analyticsvidhya.com/blog/2021/06/natural-language-processing-sentiment-analysis-using-lstm/> [Accessed 26 April 2022].

[3] Pytorch.org. 2019. torch.optim — PyTorch 1.11.0 documentation. [online] Available at: <https://pytorch.org/docs/stable/optim.html> [Accessed 26 April 2022].

[4] Atharva Sandeep Vidwans. 2021. Cognitive computing for Human-Robot Interaction:Recurrent Neural Network. [online] Available at: <https://www.sciencedirect.com/topics/computer-science/recurrent-neural-network> [Accessed 26 April 2022].

[5] Maas, A., 2022. Sentiment Analysis. [online] Ai.stanford.edu. Available at: <http://ai.stanford.edu/ amaas/data/sentiment/> [Accessed 26 April 2022].

[6] DataRobot. 2018. Introduction to Optimizers. [online] datarobot.com. Available at: <https://www.datarobot.com/blog/introduction-to-optimizers/> [Accessed 25 April 2022].

[7] Keras. API Docs. [online] keras.io Available at: <https://keras.io/api/models/> [Accessed 27 April 2022].

[8] Zuniga, E., 2022. Evolving Neural Networks with Particle Swam Optimization. [online] Medium. Available at: <https://medium.com/semantixbr/evolving-neural-networks-with-particle-swam-optimization-26a261f49d9f> [Accessed 18 April 2022].

[9] Engelbrecht, Andries P. 2007. Computational Intelligence : An Introduction. Hoboken, N.J: John Wiley ; Chichester.

[10] Mirjalili, Seyedali, Seyed Mohammad Mirjalili, and Andrew Lewis. 2014. "Grey Wolf Optimizer." Advances in Engineering Software 69 (March): 46–61. https://doi.org/10.1016/j.advengsoft.2013.12.007.

[11] Carvalho, Marcio, and Teresa B. Ludermir. 2007. "Particle Swarm Optimization of Neural Network Architectures And Weights." IEEE Xplore. September 1, 2007. https://doi.org/10.1109/HIS.2007.45.

[12] Gudise, V.G., and G.K. Venayagamoorthy. n.d. "Comparison of Particle Swarm Optimization and Backpropagation as Training Algorithms for Neural Networks." Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No.03EX706). Accessed December 18, 2019. https://doi.org/10.1109/sis.2003.1202255.