Approximating Dense ReLU Networks Using Linear Regression

Max Khanov
Computer Science Department
University of Wisconsin Madison
Madison, US
maximkha@outlook.com

Esra Akbas
Computer Science Department
Oklahoma State University
Stillwater, US
eakbas@cs.okstate.edu

***Abstract*—With increasing model sizes and more complex problems, training of Neural network models has become a significantly resource intensive task taking up to hundreds of hours. In this paper, we propose a new efficient method to train dense ReLU-based neural networks. Our method estimates a layer's weight matrix using linear regression. We evaluate our models on 3 regression tasks using the UCI Machine Learning Repository's Auto MPG Data Set and Concrete Compressive Strength Data Set [1] and a synthetic dataset generated with $y = sin(x) + 1$ function. We compare our results to traditional gradient-based optimization methods. In both shallow (2-layers) and wide (10 hidden neurons) networks, this method significantly outperforms traditional gradient descent methods like AdaGrad in terms of speed, generalization, and accuracy.**

***Index Terms*—Machine Learning, Deep Neural Networks, Training and Optimization, ReLU, Artificial Intelligence**

I. Introduction

With the availability of large datasets, the application of machine learning has greatly increased in last decade. However, with the added size and complexity of the data a traditional machine learning may not be able to learn the complex patterns in the data. As a solution to this problem, deep learning models have gained attention in recent years. Different deep learning models has been proposed. While these complex models demonstrate great performance for complex problems, training these models has become a significantly resource intensive task taking up to hundreds of hours and huge amounts of compute. As an example, GPT-3 [2], a general-purpose NLP (natural language processing) model designed by OpenAI took 3.14E23 flops of computing power and multiple weeks to be trained. Solving this issue has always been a core aspect of the machine learning community, with hardware accelerators that make training faster and more efficient to better optimizers like AdaGrad which practically ensure convergence and speed up training times enormously.

In deep neural networks, an activation function plays an important role transforming the output of a neuron to learn more complex patterns. There are several activation functions such as a linear activation function which applies no transformation, the sigmoid activation function, which applies the sigmoid function to every component of the vector, and the hyperbolic tangent function, which applies the hyperbolic tangent function to every component of the vector. The Rectified Linear Unit (ReLU) [3], [4] is the most successful and widely-used activation function because of its simplicity and effectiveness. There are also many imrpoved ReLU functions such as Elu [5], leaky ReLu [6].

A regular neural networks model include layers of neuron and learn weights of layers that map inputs of the layer to output. It learns the parameters with minimizing loss function. Some popular neural network training methods are AdaGrad [7], SGD [8], and Adam [9]. They have been proven to work well on many different tasks. All of them are gradient-descent-based methods, which require significant computational resources. As a solution to this problem, in this paper, we propose a new neural network training model that approximates the weight of a layer by assuming successive layers have no ReLU activation. The proposed method differs from gradient descent methods because it is not iterative, nor does it use gradients to inform it on how to improve the next prediction. Our goal is train neural network model more efficiently without losing effectiveness.

We further demonstrate detailed experiments on 3 different data sets. One of them are generated with the function $y = sin(x) + 1$ and other two are real world datasets. Our results show that the proposed method show better efficiency and effectiveness than the current baseline activation functions.

II. Preliminaries

A. ReLU Based Dense Neural Networks

A regular neural network include layers of neurons and learns weights of layers to map inputs to outputs.

For each layer, the inputs are multiplied by the weights in a neuron, summed together and then transformed via an activation function σ to obtain the specific output or "activation" of the node. Output of a layer is defined as

$$l_n(\overrightarrow{x}) = \sigma((W_n)^T \overrightarrow{x} + \overrightarrow{B}_n)$$

where $\overrightarrow{x}$ is the input vector and W_n is the weight matrix of the layer and $\overrightarrow{B}_n$ is the bias vector.

In $ReLU$ based neural networks, σ is the $ReLU$ function. The $ReLU$ function performs the operation $max(0, x)$ on every component of the vector, essentially zeroing out the

negative components of the vector. The layer functions are then composed to form a neural network:

$$f(\overrightarrow{x}) = (l_n \circ l_{n-1} \cdots l_2 \circ l_1)(\overrightarrow{x})$$

The number of compositions of the layer functions determines the layer count, and the number of components of the resulting vector from each layer is known as its neuron count.

B. Neural Network Training

The goal of neural network training is to learn the parameters $W_1, W_2 \cdots W_n$ and $\overrightarrow{B}_1, \overrightarrow{B}_2 \cdots \overrightarrow{B}_n$ by minimizing some error function $L(y, \hat{y})$, commonly known as a loss function, where y is the true value, and $\hat{y}$ is the estimated value calculated by $\hat{y} = f(\overrightarrow{x})$. In regression tasks, MSE (mean squared error) is a commonly used loss function defined as:

$$L_{MSE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

This is also the function usually minimized when fitting a linear regression model, more specifically, the ordinary least squares method. In traditional neural network training methods, we take the derivative of the loss function with respect to the neural network parameters. Then we take some step based on the derivatives, recalculate the loss, and repeat this process. Adam, AdaGrad, and SGD are all gradient based methods and differ based on how they update the weights using the gradients.

III. Related Works

In machine learning, gradient-based-methods, like Adam, AdaGrad and SGD, are almost exclusively used to train neural networks. Most training methods require knowledge of the underlying model such as gradient or the symbolic expression. These methods are also known as local optimization methods. In addition to the mentioned methods, non-gradient based methods have been proposed such as PSO (particle swarm optimization) [10], surrogate loss function optimization [11], and simulated annealing [12]. These methods were developed specifically for tasks where the underlying model is not known and has multiple local optima. The proposed method is similar to the gradient based methods, where the underlying model is known, however, the gradient is not calculated. This allows us to exploit the model structure to efficiently estimate the model parameters, giving us both the benefits of speed of the non-gradient-based methods and improved accuracy of the gradient-based-methods.

IV. Methodology

In this section, we explain our proposed method Linstorch. We develop a novel method to train a neural network model with the ReLU activation function by linear assumption.

The core idea of the proposed method is to approximate the weight of the i-th layer by assuming successive layers have no ReLU activation. This means that they considered as linear. By using a generalized form of matrix inverses (the Moore–Penrose inverse [13]) and the linearity assumption, we can calculate the expected input to the $(i+1)$-th layer, e.g. the output of the current layer called as $backward$. We can also forward-propagate the neural network up until the $(i-1)$-th layer, which is called as $forward$. Since we assume that the current layer, $l_i(\overrightarrow{x})$, is linear layer which is the form of $W_i\overrightarrow{x} + \overrightarrow{B_i}$, solving the layer becomes a multivariate linear regression problem where we estimate $\overrightarrow{B_i}$, the bias vector, and W_i, which is the coefficient matrix, i.e.

$$backward = W_i \times forward + \overrightarrow{B_i}$$

The proposed method differs from gradient descent methods because it is not iterative, nor does it use gradients to inform it on how to improve the next prediction.

The proposed training method, named Linstorch, is detailed in Algorithm 1 that estimates the weight matrix and bias vector for a single layer. First, we propagate input to the current layer i with a `ForwardTo` procedure, and propagates the output to the current layer with `BackwardTo` procedure.

Remaning steps are depends on whether there is a bias or not. Without bias a layer can be defined as

$$l_i(\overrightarrow{x}) = ReLU(W_i\overrightarrow{x})$$

This procedure calculates the $forward$ and $backward$ as described in the beginning of the section. If the layer has a bias term, we apply the `AppendOne` procedure to the forward matrix. With `AppendOne`(A), we expand the

Algorithm 1 SolveLayer: Learning One layer's parameter

procedure SOLVELAYER($l, i, \overrightarrow{x}, \overrightarrow{Y}$)
  $forward \leftarrow$ `ForwardTo`$(l, i, \overrightarrow{x})$
  $backward \leftarrow$ `BackwardTo`$(l, i, \overrightarrow{Y})$
  if l_i has bias **then**
    $paddedForward \leftarrow$ `AppendOne`$(forward)$
    $inverse \leftarrow$ `pinv`$(paddedForward)$
    $solutionMatrix \leftarrow (inverse \times backward)^T$
  ▷ Now extract the bias and weight matrix and update the weight and bias of the layer.
    $\overrightarrow{B_i} \leftarrow$ `lastColumnVector`$(solutionMatrix)$
  ▷ Since we padded $forward$ with ones on the right most column, the bias vector corresponds with the rightmost column vector in $solutionMatrix$.
    $W_i \leftarrow$ `exceptLastColumnVector`$(solutionMatrix)$
  ▷ The matrix without the rightmost column vector is the correct weight matrix.
  else
    $inverse \leftarrow$ `pinv`$(forward)$
    $solutionMatrix \leftarrow (inverse \times backward)^T$
    $W_i \leftarrow solutionMatrix$ ▷ Update the layer's weight matrix.
  end if
end procedure

Algorithm 2 Evaluate all layers up to l_i

procedure FORWARDTO($l, i, \overrightarrow{x}$)
  if i equals 0 **then**
    return $\overrightarrow{x}$ ▷ The input of the first layer should be $\overrightarrow{x}$
  end if
  $\overrightarrow{out} \leftarrow \overrightarrow{x}$
  for $j \leftarrow 1$ to $i-1$ **do**
    $\overrightarrow{out} \leftarrow l_j(\overrightarrow{out})$
  end for
  return $\overrightarrow{out}$
end procedure

forward matrix by adding a column vector of ones to the right of the last column and returns the new matrix e.g.:

$$\texttt{AppendOne}\left(\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \\ \vdots & \vdots \\ a_n & b_n \end{bmatrix}\right) \rightarrow \begin{bmatrix} a_1 & b_1 & 1 \\ a_2 & b_2 & 1 \\ \vdots & \vdots & \vdots \\ a_n & b_n & 1 \end{bmatrix}$$

The resulting matrix is called $paddedForward$. When adding a constant feature of 1, the multipliers of the constant feature become bias terms. This is a common technique when fitting linear regression models. To estimate the layer parameters, we need to solve the equation:

$$backward = W_i \times forward + \overrightarrow{B_i}$$

, which can be expressed using $paddedForward$ as following:

$$backward = S_i \times paddedForward$$

Finally, Moore–Penrose inverse is used to solve the matrix equation:

$$S_i = \texttt{pinv}\,(paddedForward) \times backwards$$

We take the rightmost column vector of the solution matrix S_i to get $\overrightarrow{B_i}$. We do that because we appended the constant 1 feature to the rightmost column of our data, meaning everything in the rightmost column of S_i are constant multiples of 1 representing the bias term $\overrightarrow{B_i}$. The matrix W_i is equal to the solution matrix S_i excluding the rightmost constant column vector.

The ForwardTo procedure evaluates the neural network up to the before l_i as regular with ReLU activation function, which is detailed in Algorithm 2.

The BackwardTo procedure assumes that all layers after l_i are linear and iterates backward calculating the product of all the linear matrices. Then, using the Moore–Penrose inverse, it calculates a possible input to the linear approximation of the layers which is the expected output of l_i, so that the successive linear approximations result in $\overrightarrow{Y}$ and returns the expected output:

The procedure `Mat` generates a matrix representation of l_i in the following form:

Algorithm 3 Evaluate all layers up to l_i

procedure BACKWARDTO($l, i, \overrightarrow{y}$)
  if i equals N **then**
    return $\overrightarrow{y}$ ▷ The output of the last layer should be $\overrightarrow{y}$
  end if
  $wProduct \leftarrow I$ ▷ Initialize the weight product with an identity matrix with the size being the number of output features
  for $j \leftarrow N$ to $i+1$ **do** ▷ **Note:** this iterates backward
    $layerMatrix \leftarrow \texttt{Mat}\,(l_j)$
    $wProduct \leftarrow wProduct \times layerMatrix$
  end for
  $inverseMatrix \leftarrow \texttt{pinv}\,(wProduct)$
  $expectedOutput \leftarrow (\overrightarrow{y} \times inverseMatrix)^T$
  return $expectedOutput$
end procedure

$$\texttt{Mat}\,(l_i) \rightarrow \begin{bmatrix} M_{11} & M_{12} & \dots & M_{1n} & \overrightarrow{C}_1 \\ M_{21} & M_{22} & \dots & M_{2n} & \overrightarrow{C}_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ M_{m1} & M_{m2} & \dots & M_{mn} & \overrightarrow{C}_m \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

Where for convenience the following variables are used:

$$M = W_i$$

$$C = B_i$$

M has a size of $m \times n$ and $\overrightarrow{C}$ has a size of m.

The bottommost row vector is added to the matrix in order to pass along the column vector of ones, added by the `AppendOne` procedure, necessary for the subsequent layers to implement the bias vector $\overrightarrow{C}$. This matrix form allows us to pass along the constant one column vector appended by the `AppendOne` procedure.

Note: If the layer has no bias $\overrightarrow{C}$ is set to the zero vector with the size of m.

Finally, Lintorch algorithm that trains a neural network model with the ReLU activation function by linear assumption is detailed in Algorithm 4. It runs for multiple layers and trains neural network with data. First it iterates from layer N to 1 to learn the parameters of all layers with SolveLayer 1. Then it iterates from layer 1 to N to update the weights of the layers.

V. RESULTS

In this section we report our experimental results on 3 different data sets. The first data set is the Auto MPG Data Set from the UCI Machine Learning Repository and is derived from real world data. The second data set is generated with the function $y = sin(x) + 1$. We compare the newly proposed method and state-of-the-art training methods. The third data

Algorithm 4 Fit the neural network to the data

procedure SOLVE($l, \overrightarrow{x}, \overrightarrow{Y}$)
  for $j \leftarrow N$ to 1 **do** ▷ Iterating backward through each layer of the neural network
    `SolveLayer` $(l, i, \overrightarrow{x}, \overrightarrow{Y})$
  end for
  for $j \leftarrow 1$ to N **do** ▷ Iterating forwards through each layer of the neural network
    `SolveLayer` $(l, i, \overrightarrow{x}, \overrightarrow{Y})$
  end for
end procedure

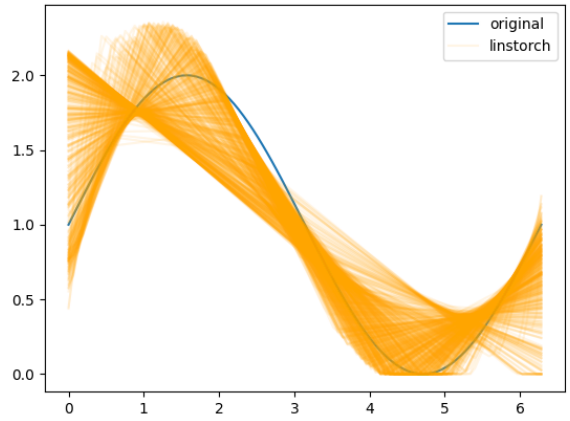

Fig. 1. 1000 randomly initialized neural networks fitted to the function $sin(x) + 1$ by the Linstorch algorithm.

set is the Concrete Compressive Strength Data Set from the UCI Machine Learning Repository and is derived from real world data.

Figure 1 shows the neural network outputs of the 1000 trained neural networks fitted to the $y = sin(x)+1$ data using the Linstorch algorithm. As one can see from this Figure, even though we use a linear regression-based training method, the resulting function is not strictly linear and approximates the shape of the data well. Figure 2 shows a histogram of the errors of 1000 Linstorch trained neural networks. The dotted line represents the error of a ordinary least squares based linear regression and the solid line represents the average error of the Linstorch training method. As we see, the proposed method easily outperforms linear regression and is always better than linear regression. Figure 3 shows the error of a random neural network approximation of the function $y = sin(x) + 1$ without any training. The random neural network approximation performs significantly worse than simple linear regression and Linstorch. All of the tested neural networks had 2-layers and 10 neurons in the intermediate layer.

In Figures 4 and 5, we show the MSE as a function of execution time, which allows us to compare errors and efficiency for epoch and non-epoch-based methods. We compare our

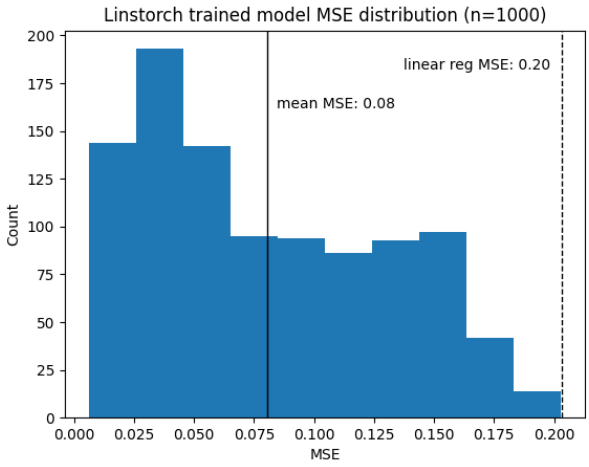

Fig. 2. Histogram MSE (mean squared error) of 1000 randomly initialized neural networks fitted to the function $sin(x) + 1$ by the Linstorch algorithm.

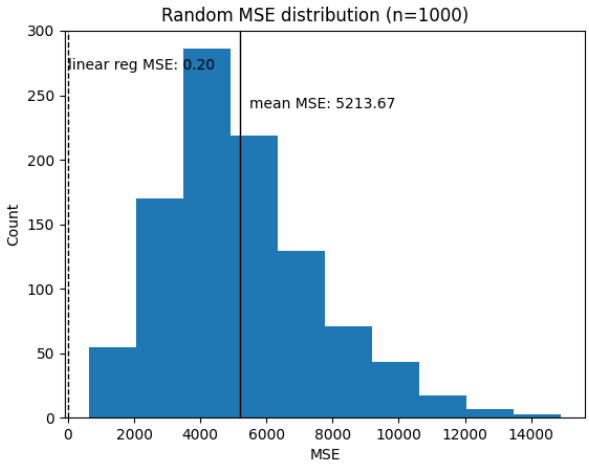

Fig. 3. Histogram MSE (mean squared error) of 1000 randomly initialized neural networks with the function $sin(x) + 1$.

results with the following training methods: SGD (stochastic gradient descent), Adam, AdaGrad (adaptive gradient algorithm), random neural network sampling, and Linstorch. For every training method, default hyper-parameters are used. For both the Random neural network sampling (RANDOM) and Linstorch methods, we do training by regenerating neural networks until a target error is achieved. For other methods, an iterative method is used. The tested neural networks had 2-layers. For both datasets, we used a 1/3 test train split, where 1/3 of the data was used for testing and the other 2/3 was used for training.

Figure 4 shows results for the function $y = sin(x)+1$. The proposed method reaches the lowest error in the least amount of elapsed time. Notably, Linstorch always outperforms random neural network sampling, meaning Linstorch actually fits the neural network to the data.

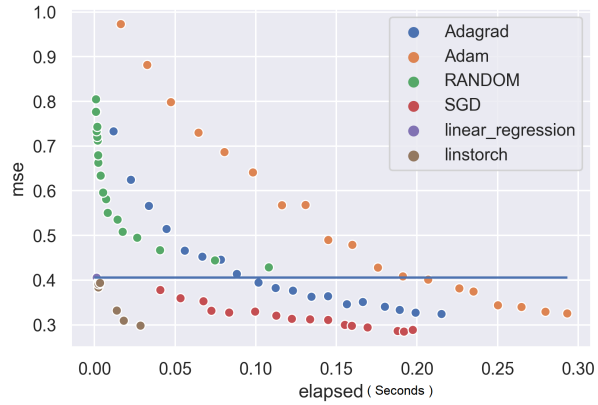

Fig. 4. Performance of different neural network training algorithms when fitting a 2-layer deep neural network to the function $sin(x) + 1$.

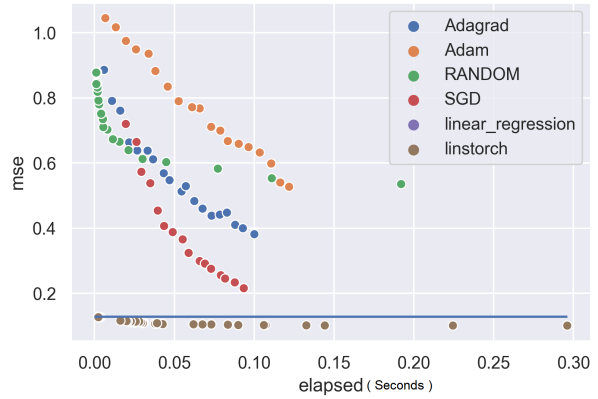

Fig. 5. Performance of different neural network training algorithms when fitting UCI Machine Learning Repository's Auto MPG Data Set.

Figure 5 shows the results for the UCI Machine Learning Repository's Auto MPG Data Set. We used mpg, acceleration, and displacement as features and predicted horsepower. Again, Linstorch reaches the lowest error in the least amount of elapsed time. Linstorch still outperforms linear regression, but not by much. Linstorch also still outperforms the random neural network sampling.

Figure 6 shows the results for the UCI Machine Learning Repository's Concrete Compressive Strength Data Set. We used concrete age and the quantities of the ingredients. These ingredients include cement, blast furnace slag, fly ash, water, superplasticizer, coarse aggregate, and fine aggregate. Again, Linstorch reaches the lowest error in the least amount of elapsed time. Linstorch still outperforms linear regression and significantly outperforms the gradient-based methods. Linstorch also still outperforms the random neural network sampling.

Table I, shows errors which is obtained by running the `Solve` procedure either iterating backward or forwards first. As one can see from table I, the lowest error is achieved by

TABLE I
COMPARISON OF ERRORS FOR DIFFERENT SOLVE PROCEDURES.

MSE of different layer solve orders and iteration count		
Solve Calls	Backward First	Forward First
0	5331.501	5257.762
1	**0.084**	0.488
2	0.173	0.485

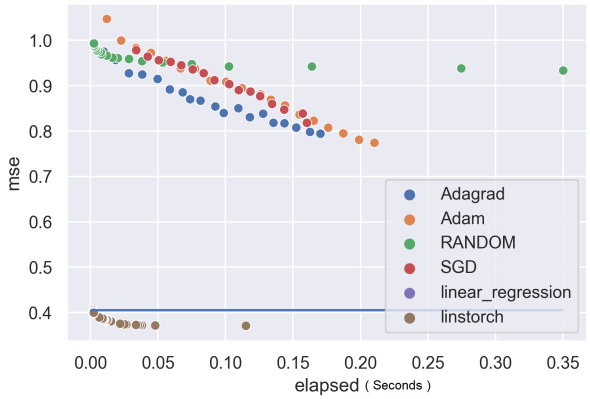

Fig. 6. Performance of different neural network training algorithms when fitting UCI Machine Learning Repository's Concrete Compressive Strength Data Set.

calling the `Solve` procedure once, and iterating through the layers backward first.

We change the number of layers to see the effect of it on the results. We see that increasing the layer number, decrease the mse bur increase the training time. In Figure 7, we give the results for 3-layer deep neural networks on $y = sin(x) + 1$ function. Notably, as more layers are added our training method decrease the efficiency of our method but increase the effectiveness with less error. In this case, our method still achieves the lowest error in the least amount of time.

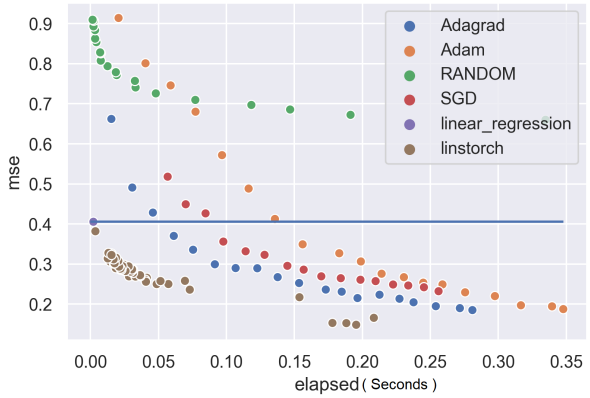

Fig. 7. Performance of different neural network training algorithms when fitting a 3-layer deep neural network to the function $sin(x) + 1$.

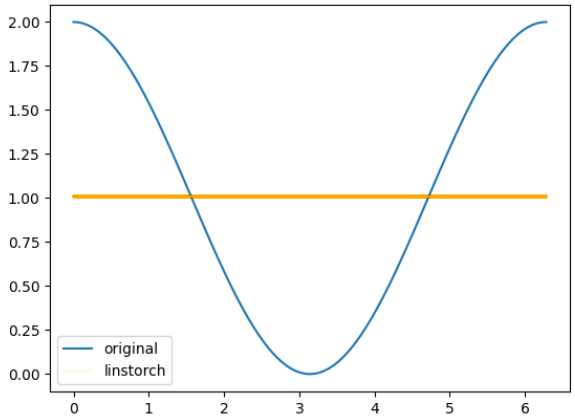

Fig. 8. Degenerate case where the slope of the regression line is zero and Linstorch does not outperform linear regression.

VI. Limitations of the Method

Studying this method we found that one particular case that represents a challenge for our method. When the best linear regression model has slopes of zero, the method cannot do better than the linear regression model. An example of this case is shown in Figure 8. The method also struggles with deep neural networks. This is related to the intrinsic assumptions that the Linstorch training method makes. Since it assumes successive layers are linear, the more layers there are, the worse that approximation is, which causes Linstorch to not get a low final error, even with re-sampling.

VII. Conclusions and Future Work

In this paper we introduced Linstorch, a novel non-gradient-based training method for ReLU activation dense neural networks. We compared the proposed method to widely used neural network training methods (Adam, SGD, AdaGrad) on three regression tasks: fitting the function $y = sin(x) + 1$, fitting the UCI Machine Learning Repository's Auto MPG Data Set, and fitting the UCI Machine Learning Repository's Concrete Compressive Strength Data Set. Our results show that Linstorch outperforms all methods in terms of elapsed time and MSE.

Unlike most training algorithms, Linstorch is not iterative, meaning it is trivial to parallelize. In the time it takes to run one iteration of the algorithm one could run the algorithm in parallel and obtain multiple samples granting a speedup factor proportional to however many cores are available. Another improvement to the method is using it in conjunction with gradient-based methods. Since Linstorch is incredibly fast to run and performs as well as linear regression or better, it could be a replacement for weight initialization in neural networks.

This method can be used in big data applications since it achieves low error quickly. It can play a part in edge computing, where low computing power devices need to train neural networks. This method can also be used in a hybrid approach, quickly creating a good initial model and using traditional gradient-based methods to finish optimizing the model. Developing a mathematically rigorous understanding of the proposed method could allow the limitations to be solved and further efficiency gains in training. Another possible future work is developing a parallel sampling method and evaluating the effectiveness of using Linstorch as weight initialization method. All the code and data are available at: https://github.com/maximkha/linfit.

Acknowledgment

This work is supported by the National Science Foundation under Grant 2050978. Some of the computing for this project was performed at the High Performance Computing Center at Oklahoma State University supported in part through the National Science Foundation grant OAC-1531128.

References

[1] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[3] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *nature*, vol. 405, no. 6789, pp. 947–951, 2000.

[4] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[5] Z. Zuo, J. Li, B. Wei, L. Yang, C. Fei, and N. Naik, "Adaptive activation function generation through fuzzy inference for grooming text categorisation," in *2019 IEEE International Conference on Fuzzy Systems*, 2019.

[6] Y. Liu, X. Wang, L. Wang, and D. Liu, "A modified leaky relu scheme (mlrs) for topology optimization with multiple materials," *Applied Mathematics and Computation*, vol. 352, pp. 188–204, 2019.

[7] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011. [Online]. Available: http://jmlr.org/papers/v12/duchi11a.html

[8] H. E. Robbins, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 2007.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[10] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in *Proceedings of the IEEE International Conference on Neural Networks*, 1995, pp. 1942–1948.

[11] N. Lee, H. Yang, and H. Yoo, "A surrogate loss function for optimization of f_β score in binary classification with imbalanced data," *CoRR*, vol. abs/2104.01459, 2021. [Online]. Available: https://arxiv.org/abs/2104.01459

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983. [Online]. Available: http://www.jstor.org/stable/1690046

[13] R. Penrose, "A generalized inverse for matrices," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 51, no. 3, p. 406–413, 1955.