All Critical Gradient Descent Solutions Are Optimal

Anonymous Author
Anonymous Institution

Abstract

Within the theory of machine learning, we prove that when over-parameterized and gradient descent finds critical weights, i.e., a point $\vec{w}$ on the error surface with zero slope, the resulting neural network $NN_{\vec{w}}(x)$ provides perfect responses for each training data point. To put it differently, the error surface lacks local minima, maxima, and inflection points. This improves on work by Hui Jiang (2019) [5]. Their result requires the number of weights in the neural network to be as large as $(1/\epsilon)^N$. Using different techniques, we only require more nodes in a layer than training data. This number of weights is nearly optimal, because reducing the number of weights further would render the weight optimization problem NP-complete. Furthermore, our result establishes the first quantifiable link between the criticality of $\vec{w}$ and the accuracy of the machine's approximation to the supervisor's responses. As a theoretical paper, we assume that at $\vec{w}$ the sensitivity of the neural network itself has not gone flat, either because of a conscious effort to maintain a sufficient number of sigmoid/ReLU activation functions outside their flatter regions or due to the inherent property that sigmoid functions always retain at least some non-zero slope.

1 INTRODUCTION

In practice, it's often observed that with a sufficient amount of data and parameters, systems like ChatGPT and self-driving cars can be developed. However, as noted by John Tsotsos (2023)[6], a lingering concern remains: practical experience has yet to grasp when, why, and how the process might fail. When gradient descent leads us to a critical point denoted as $\vec{w}$, the question arises: is this a global minimum, and furthermore, can we guarantee that the resulting neural network will provide flawless answers for the provided training data? In this paper, we introduce a perspective in which we challenge the existence of disappointing critical points altogether.

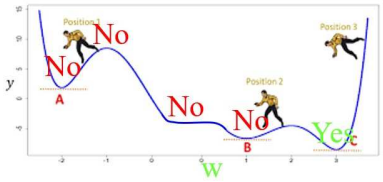

Figure 1: No local minimum

1.1 Historical Perspective

Our result requires over-parameterized. A longstanding debate within the realm of machine learning centers on the dilemma of whether the number of parameters, denoted as P, should surpass or fall short of the size of the training data set, represented as D. The former is essential to ensure the existence of weight configurations $\vec{w}$ that produce optimal computations on the training data, while the latter, as demonstrated by Avrim Blum and Ronald Rivest (1992) [3, 2] renders the task of finding such weights NP-complete. (See Section 2.6 for more). The former approach carries the risk of over-fitting, whereas the latter safeguards the machine's capacity to generalize to unseen data (a concept stemming from PAC learning). In contrast, a *smooth* machine also exhibits enhanced generalization capabilities. The fundamental idea is that when a new input $\vec{x}$ lies between training inputs $\vec{x}_d$ and $\vec{x}_{d'}$, its output is likely to fall within the range defined by their respective outputs. Bubeck and Sellke (2021) [4]

Preliminary work. Under review by AISTATS 2024. Do not distribute.

established that a machine attains smoothness only when the parameter count P satisfies $P \geq \Omega(ND)$. Additionally, practical observations substantiate the idea that having an increased number of weights enhances the maneuverability of gradient descent within this higher-dimensional search space, enabling it to navigate obstacles more effectively.

A well-known problem with gradient descent is that when the pre-activation values become very large, the slope of the sigmoid curve decays exponentially, which can severely slow down gradient descent. To circumvent this issue, practitioners have become adept at ensuring a sufficient proportion of sigmoid/ReLU activation functions remain outside their less responsive regions. As a theory paper, we make the assumption that, under the current weights, the sensitivity of the neural network is not flat either because people have succeeded at this or due to the inherent property that sigmoid functions always retain at least some non-zero slope.

Allen-Zhu, Li, and Song (2018) [1] gave an amazing paper proving that gradient descent starting with random weights $\vec{w}_0$ converges to an optimal solution in $D^{\mathcal{O}(1)}$ time as long as the machine has $D^{O(1)}$ parameters. Their technique proves that as long as the weights remain within a small ball around $\vec{w}_0$, each step of gradient descent makes a multiplicative decrease in the error until an optimal solution has been found. While this finding demonstrates the capacity of gradient descent to navigate away from local minima and flat regions, it falls short of conclusively asserting their non-existence. Hui Jiang pointed out that a limitation of the paper is that it only considers a small neighborhood around randomly selected weights where the convexity of the error space is evident. In practice, gradient descent often traverses much larger distances.

Jiang (2019) [5] made a significant contribution by proving the absence of local minima within the Error Surface. This achievement was accomplished by employing a Fourier ϵ-approximation of the function computed by the neural network. The error surface with respect to the Fourier coefficients is nicely convex, and this property translates back to the weight space. The requirement for the number of weights may be reasonable when the function to be learned possesses smooth characteristics. However, in the worst case, $(1/\epsilon)^N$ weights are needed.

The present study advances this understanding by reducing the requirements, specifically by stipulating that the number of hidden nodes in the widest layer need only match the amount of training data.

1.2 Mathematical Insight

We initiate our discussion with an intuitive overview of our approach. Our primary objective is to prove that only globally optimal critical points exist. When considering sigmoid activation functions, we make the assumption that their slope never reaches zero. Consequently, we contend that as the weights w_m undergo changes, the output of the neural network consistently varies. In practical terms, this implies that for any fixed input $\vec{x}$, the relationship between $NN_{\vec{w}}(x)$ and $\vec{w}$ does not encompass critical points. The error surface $Error(\vec{w}) = \sum_d (NN_{\vec{w}}(\vec{x}_d) - y_d^*)^2$ has local minimums only because of this imposed quadratic. We prove that all such critical points are optimal in that for each training input $\vec{x}_d$, the neural network produces an output $NN_{\vec{w}}(\vec{x}_d)$ equaling the supervisor's answer y_d^*.

Hui Jiang (2019) [5] effectively introduces the following framework: Consider a vector space of dimension D, where each dimension corresponds to a training input $\vec{x}_d$. Given the current set of weights $\vec{w}$, we can measure the discrepancy between the neural network's response $NN_{\vec{w}}(\vec{x}_d)$ and the supervisor's answer y_d^* using the vector $\vec{Loss}$ defined as $\vec{Loss} = \langle NN_{\vec{w}}(\vec{x}_d) - y_d^* \,|\, d \in [D] \rangle$. For each weight $\vec{w}_m$, we define $\vec{Dir}_m$ as the vector of derivatives $\langle \frac{\delta NN_{\vec{w}}(\vec{x}_d)}{\delta \vec{w}_m} \,|\, d \in [D] \rangle$. We can assemble a matrix **Dir** using these vectors, where each row corresponds to $\vec{Dir}_m$. This framework enables us to express the direction of descent, i.e., the vector of derivatives of the Error function with respect to each $\vec{w}_m$, as follows:

$$\begin{aligned}
Error(\vec{w}) &= \textstyle\sum_d \left(NN_{\vec{w}}(\vec{x}_d) - y_d^*\right)^2 \\
\tfrac{\delta Error(\vec{w})}{\delta \vec{w}_m} &= 2 \textstyle\sum_d \left(NN_{\vec{w}}(\vec{x}_d) - y_d^*\right) \cdot \tfrac{\delta NN_{\vec{w}}(\vec{x}_d)}{\delta \vec{w}_m} \\
&= 2\ \vec{Loss} \cdot \vec{Dir}_m \\
\nabla Error(\vec{w}) &= \left\langle \tfrac{\delta Error(\vec{w})}{\delta \vec{w}_1}, \ldots, \tfrac{\delta Error(\vec{w})}{\delta \vec{w}_P} \right\rangle \\
&= 2\ \mathbf{Dir} \cdot \vec{Loss}
\end{aligned}$$

If the neural network has more weights than data, $P \geq D$, the matrix **Dir** has more rows than columns. Our task is to prove that it has full rank D. This then gives

$$\vec{Loss} = \tfrac{1}{2}\mathbf{Dir}^{-1} \cdot \nabla Error(\vec{w})$$

Having the weights $\vec{w}$ situated at a critical point implies that each of these Error derivatives is zero, namely $\nabla Error(\vec{w}) = \vec{0}$. This results in $\vec{Loss} = \langle NN_{\vec{w}}(\vec{x}_d) - y_d^* \mid d \in [D] \rangle = \vec{0}$, signifying that the neural network provides the optimal answers for all of the training data points.

Hui Jiang (2019) [5] establishes that his version of the matrix **Dir** indeed possesses a rank of D. This achievement is accomplished through the application

of Fourier ϵ-approximation to the function computed by the neural network. However, it's worth noting that in the worst-case scenario, **Dir** could have a staggering number of rows/weights on the order of $(1/\epsilon)^N$. In addressing Hui Jiang's challenge, we have managed to demonstrate that this considerably smaller version of **Dir** also has rank of D.

To provide some more context to our result, if one were to perform gradient descent with respect to only a single training input $\vec{x}_d$, it would be clear that the optimal solution for this input would be found. The challenge we faced in obtaining this result revolved around handling the interactions among these various inputs. Our proof successfully finds a way to do this. It's important to note that in deriving this result, we do assume another matrix has linearly independent columns, this assumption is much more reasonable for this different matrix, thereby bolstering the validity of our overall argument.

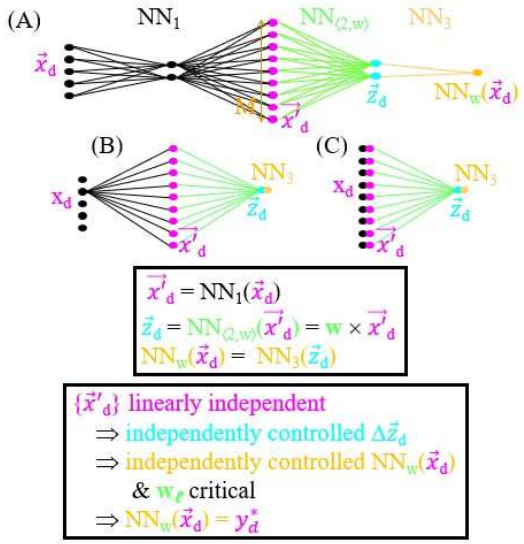

Figure 2: Three examples are given of the machine $NN_{\vec{w}}(\vec{x}_d) = \left[NN_3 \circ NN_{\langle 2,\vec{w}\rangle} \circ NN_1\right](\vec{x}_d)$. Machine A is a typical one. The input to Machine-B is a singleton value. Stage NN_1 of machine-C is the 1-1 function $\vec{x}'_d = \vec{x}_d$. Stage NN_3 of machines B and C are the 1-1 function $NN_{\vec{w}}(\vec{x}_d) = \vec{z}_d$. The top box gives the functions computed by the stages of the machine. The bottom box gives the stages of the proof.

2 RESULTS

2.1 Statement of Theorem

Theorem 1 *(Critical is Optimum): Consider any computational machine $NN_{\vec{w}}(\vec{x}_d)$ expressed as $\left[NN_3 \circ NN_{\langle 2,\vec{w}\rangle} \circ NN_1\right](\vec{x}_d)$, with NN_1 and NN_3 satisfying requirements 1 and 3, as specified below. For any given set of training data $\{\langle \vec{x}_d, y^*_d\rangle \mid d \in [D]\}$, there exists a set of weights $\vec{w}$ that enables the machine to produce correct answers. Furthermore, during gradient descent to learn these weights $\vec{w}$, the existence of no local minima (or maxima) is guaranteed. Specifically, if the weights reach a state of sufficient criticality such that the magnitude of the gradient vector does not exceed ϵ, then they are nearly optimal in that $\forall d \in [D], |NN_{\vec{w}}(\vec{x}_d) - y^*_d| \leq \frac{\epsilon}{2qr}$. The parameters r and q, both of which should be approximately equal to one, are defined below.*

(1) $\mathbf{x'} = \{\boldsymbol{NN_1(\vec{x}_d)} \mid \boldsymbol{d \in [D]}\}$ is Linear Independent: *In the first stage of the machine, each training input $\boldsymbol{\vec{x}_d \in \{0,1\}^N}$ is transformed into a vector of hidden values $\boldsymbol{\vec{x}'_d \in \{0,1\}^M}$. This transformation can be achieved through various means, such as an arbitrary adversarial function, a multi-layered neural network learned via gradient descent, or a straightforward one-to-one mapping. The requirement here is that given the set of $\boldsymbol{D}$ training inputs $\boldsymbol{\{\vec{x}_d \mid d \in [D]\}}$, the resulting set of vectors of hidden values $\boldsymbol{\{\vec{x}'_d \mid d \in [D]\}}$ must exhibit linear independence with a scaling factor of at least $\boldsymbol{r}$, computed as $\boldsymbol{Min_{\vec{v}}\ |\mathbf{x'}\vec{v}| \geq r\,|\vec{v}|}$. Notably, this requirement necessitates that the number of nodes $\boldsymbol{M}$ in this hidden layer must be greater than or equal to the quantity of training data $\boldsymbol{D}$, i.e., $\boldsymbol{M \geq D}$. (In Section 2.6, we contrast the number of hidden nodes $\boldsymbol{M}$ with the number of parameters $\boldsymbol{P}$.)*

(2) $\boldsymbol{\vec{z}_d = NN_{\langle 2,\vec{w}\rangle}(\vec{x}'_d) = \mathbf{w} \times \vec{x}'_d}$ is Critical: *In the second stage, we have a fully connected linear neural network layer that maps the hidden values $\boldsymbol{\vec{x}'_d \in \{0,1\}^M}$ from the previous layer to the hidden values $\boldsymbol{\vec{z}_d \in \{0,1\}^{M'}}$ at the subsequent layer. Let $\mathbf{w}$ denote the weights $\boldsymbol{\vec{w}}$ rearranged into the transpose of an $\boldsymbol{M \times M'}$ weight matrix. The key requirement is that these weights, obtained through some gradient descent process, must be critical. This criticality is characterized by the condition that the gradient vector $\boldsymbol{\nabla Error(\vec{w}) = \left\langle \frac{\delta Error(\vec{w})}{\delta w_1}, \ldots, \frac{\delta Error(\vec{w})}{\delta w_{M\times M'}} \right\rangle}$ has a magnitude no greater than the parameter $\boldsymbol{\epsilon}$. It's important to note that while the proof primarily focuses on the specific error function $\boldsymbol{Error(\vec{w}) = \sum_d (NN_{\vec{w}}(\vec{x}_d) - y^*_d)^2}$, the only essential requirement is that when $\boldsymbol{\left|\frac{\delta Error}{\delta y_d}\right| \leq \mathcal{O}(\epsilon)}$, it implies $\boldsymbol{|y_d - y^*_d| \leq \mathcal{O}(\epsilon)}$.*

As an illustrative example, this result also holds true when the training data is categorical, the activation function is softmax, and the error function is cross-entropy.

(3) $\boldsymbol{NN_{\vec{w}}(\vec{x}_d) = NN_3(\vec{z}_d)}$**:** *In the final stage, we have a mapping from the hidden values* $\boldsymbol{\vec{z}_d \in \{0,1\}^{M'}}$ *to real-valued outputs. Similar to previous stages, this mapping can be achieved through various means, such as an arbitrary adversarial function, a multi-layered neural network learned via gradient descent, or, with* $\boldsymbol{M'=1}$*, a straightforward one-to-one mapping.*

Full Range: *The requirement for the first result is straightforward: the range of* $\boldsymbol{NN_3(\vec{z})}$ *must include all of the supervisor's answers* $\boldsymbol{y^*_d}$*.*

Sensitive: *The requirement for the second is that the final stage is sensitive. In other words, at each input* $\boldsymbol{\vec{z}_d}$ *under consideration, the function* $\boldsymbol{NN_3(\vec{z}_d)}$ *must exhibit a large slope. This sensitivity is quantified by the condition that the vector of derivatives* $\boldsymbol{\nabla NN_3(\vec{z}_d) = \left\langle \frac{\delta NN_3(\vec{z}_d)}{\delta [\vec{z}_d]_1}, \ldots, \frac{\delta NN_3(\vec{z}_d)}{\delta [\vec{z}_d]_{M'}} \right\rangle}$ *must have a magnitude of at least the parameter* $\boldsymbol{q}$*.*

Section 2.4 provides arguments to support the idea that these assumptions are not unreasonable.

2.2 Outline of the Proof

Suppose that the error surface $Error(\vec{w}) = \sum_d (NN_{\vec{w}}(\vec{x}_d) - y^*_d)^2$ is flat/critical at the current weights $\vec{w}$. Our goal is to prove that the answers $NN_{\vec{w}}(\vec{x}_d)$ given by the neural network are the correct ones y^*_d. To do this, consider your favorite training input $\vec{x}_d$. Let $Error_d(\vec{w}) = (NN_{\vec{w}}(\vec{x}_d) - y^*_d)^2$ denote the error contribution from this $\vec{x}_d$. Our objective is to find a direction $\Delta\vec{w}_d$ in weight space such that the output $NN_{\vec{w}+\gamma\Delta\vec{w}_d}(\vec{x}_d)$ changes with respect to γ while keeping $Error_d(\vec{w} + \gamma\Delta\vec{w}_d)$ flat/critical. If we can achieve this, we are done because $Error_d(\vec{w}+\gamma\Delta\vec{w}_d) = (NN_{\vec{w}+\gamma\Delta\vec{w}_d}(\vec{x}_d) - y^*_d)^2$ is only flat/critical with respect to γ when $NN_{\vec{w}}(\vec{x}_d) = y^*_d$.

Given that the sum $Error(\vec{w}) = \sum_{d'} Error_{d'}(\vec{w})$ is flat, the component $Error_d(\vec{w})$ only fails to be flat when other $Error_{d'}(\vec{w})$ decrease to compensate for $Error_d(\vec{w})$ increasing. Hence, it is sufficient to prove that in the direction $\Delta\vec{w}_d$, these other $Error_{d'}(\vec{w} + \gamma\Delta\vec{w}_d)$ do not change. This is ensured by having the output $NN_{\vec{w}+\gamma\Delta\vec{w}_d}(\vec{x}_{d'})$ not change in this direction. The proof technique then is to gain "linearly independent control" over what the neural network outputs on each of the training data.

We have let $\vec{x'}_d \in \mathbb{R}^M$ denote the vector of values leaving the first stage NN_1. Having fewer such vectors than the number of their dimensionality, it is possible that they are linearly independent. Our first requirement is that this is the case, either because of properties of Vandermonde matrices or of random matrices.

In the second stage $NN_{\langle 2,\vec{w}\rangle}$, the relation is $\vec{z}_d = \mathbf{w} \times \vec{x'}_d$, where $\mathbf{w}$ denotes transpose of the $M \times M'$ weight matrix and $\vec{z}_d \in \mathbb{R}^{M'}$ denotes the vector of pre-activation values leaving this layer. The linear independence of $\vec{x}_d$ implies that for each training input $\vec{x}_d$, there is a direction in weight space $\boldsymbol{\Delta \mathbf{w}_d}$ that allows us to change $\vec{z}_d$ in any desired way $\Delta\vec{z}_d$ while leaving $\vec{z}_{d'}$ unchanged for other inputs.

Because the third state NN_3 is sensitive on input $\vec{z}_d$, our modification $\Delta\vec{z}_d$ of $\vec{z}_d$ can be chosen to change the output $NN_{\vec{w}}(\vec{x}_d)$. In contrast, if the $\vec{z}_{d'}$ does not change, then the output $NN_{\vec{w}}(\vec{x}_{d'})$ does not change. This completes the proof.

2.3 Proof

Proof: We compute the weights $\vec{w}$ (or its matrix form $\mathbf{w}$) for the first part of the theorem as follows. As allowed by Requirement 3, for every supervisor's answers y^*_d, let $\vec{z}_d$ denote a vector such that $NN_3(\vec{z}_d) = y^*_d$. For the purpose of linear algebra, form matrices $\mathbf{x}$ and $\mathbf{x}'$, and $\mathbf{z}$, and vector y^* with a column for each training input $\vec{x}_d$, namely $[\mathbf{x}]_d = \vec{x}_d$, $[\mathbf{x}']_d = \vec{x}'_d$, $[\mathbf{z}]_d = \vec{z}_d$, and $[y^*]_d = y^*_d$. This gives $\mathbf{x}' = NN_1(\mathbf{x})$, $\mathbf{z} = \mathbf{w} \times \mathbf{x}'$, and $y^* = NN_3(\mathbf{z})$. Recall that Requirement 1 assures that matrix $\mathbf{x}'$ has full rank. Hence, by Lemma 2, we can solve the $\mathbf{z} = \mathbf{w} \times \mathbf{x}'$ for $\mathbf{w}$.

We now prove the second part of the theorem. Consider a machine $NN_{\vec{w}}(\vec{x}_d) = [NN_3 \circ NN_{\langle 2,\vec{w}\rangle} \circ NN_1](\vec{x}_d)$ with critical weights $\vec{w}$ and training data $\{\langle \vec{x}_d, y^*_d \rangle \mid d \in [D]\}$ as required. Focus on the training input $\vec{x}_d$ and the corresponding hidden values $\vec{z}_d = \left[NN_{\langle 2,\vec{w}\rangle} \circ NN_1\right](\vec{x}_d)$. Requirement 3 gives that, at input $\vec{z}_d$, the function $NN_3(\vec{z}_d)$ has a large slope, i.e., the vector of derivatives $\nabla NN_3(\vec{z}_d) = \left\langle \frac{\delta NN_3(\vec{z}_d)}{\delta [\vec{z}_d]_1}, \ldots, \frac{\delta NN_3(\vec{z}_d)}{\delta [\vec{z}_d]_{M'}} \right\rangle$ has magnitude at least parameter q. Let $\Delta\vec{z}_d$ denote the length one vector in the steepest accent direction, i.e., the unit length scaling of $\nabla NN_3(\vec{z}_d)$. The slope of $NN_3(\vec{z}_d)$ in this direction is $\nabla_{\Delta\vec{z}_d} NN_3(\vec{z}_d) = \left|\frac{\delta NN_3(\vec{z}_d + \gamma\Delta\vec{z}_d)}{\delta\gamma}\right| = [\nabla NN_3(\vec{z}_d)] \cdot [\Delta\vec{z}_d] = |\nabla NN_3(\vec{z}_d)| \times |\Delta\vec{z}_d| \times cos(\theta) \geq q \times 1 \times 1 = q$.

Denote by $\Delta\vec{w}_d$ a direction to change the weights $\vec{w}$ to achieve the change $\Delta\vec{z}_d$, while at the same time making zero effective change when given any other training

input. Namely, for every real value γ,

$$\left[NN_{\langle 2,\vec{w}+\gamma\Delta\vec{w}_d\rangle} \circ NN_1\right](\vec{x}_d)$$
$$= \left[NN_{\langle 2,\vec{w}\rangle} \circ NN_1\right](\vec{x}_d) + \gamma\Delta\vec{z}_d = \vec{z}_d + \gamma\Delta\vec{z}_d$$
and $\forall d' \neq d$
$$\left[NN_{\langle 2,\vec{w}+\gamma\Delta\vec{w}_d\rangle} \circ NN_1\right](\vec{x}_{d'})$$
$$= \left[NN_{\langle 2,\vec{w}\rangle} \circ NN_1\right](\vec{x}_{d'}) + 0 = \vec{z}_{d'} + 0$$

We compute $\Delta\vec{w}_d$ (or its matrix form $\Delta\mathbf{w}_d$) as follows. Let $\Delta\mathbf{z}_d$ be the matrix whose d^{th} column is the required change $\Delta\vec{z}_d$ and the rest of the columns are zero. Note that this mostly zero matrix and the vector have the same Euclidean magnitude $|\Delta\mathbf{z}_d|_2 = |\Delta\vec{z}_d|_2 = 1$, where we defining $|\mathbf{M}|_2 = \sqrt{\sum_{\langle i,j\rangle} \mathbf{M}_{\langle i,j\rangle}{}^2}$. Let $\Delta\mathbf{w}_d$ denote any matrix satisfying the equation $\Delta\mathbf{w}_d \times \mathbf{x}' = \Delta\mathbf{z}_d$. By Requirement 1, matrix $\mathbf{x}'$ is linearly independent with scaling factor at least r computed as $Min_{\vec{v}}\ |\mathbf{x}'\vec{v}| \geq r\,|\vec{v}|$. Hence by Lemma 1, $r\,|\Delta\mathbf{w}_d|_2 \leq |\Delta\mathbf{w}_d \times \mathbf{x}'|_2 = |\Delta\mathbf{z}_d|_2 = 1$ and hence $|\Delta\vec{w}_d| \leq \frac{1}{r}$. Then, by linearity, this weight change changes $\vec{z}_d$ as required, namely $(\mathbf{w}+\gamma\Delta\mathbf{w}_d)\times\mathbf{x}' = (\mathbf{z}_d+\gamma\Delta\mathbf{z}_d)$.

The machine's composition ensures that the weights $\vec{w}$ affect the machine's output $NN_{\vec{w}}(\vec{x}_d)$ only via the hidden values $\vec{z}$. We have shown that on input $\vec{x}_d$, changing the γ in the weights $\vec{w}+\gamma\Delta\vec{w}_d$ has the same effect as changing the γ in the hidden values $\vec{z}_d+\gamma\Delta\vec{z}_d$. It follows that $\left|\frac{\delta NN_{\vec{w}+\gamma\Delta\vec{w}}(\vec{x}_d)}{\delta\gamma}\right| = \left|\frac{\delta NN_3(\vec{z}_d+\gamma\Delta\vec{z}_d)}{\delta\gamma}\right|$, which by Requirement 3, is at least q. In contrast, on any other input $\vec{x}_{d'}$, changing the γ in the weights $\vec{w}+\gamma\Delta\vec{w}_d$ has zero effect the $\vec{z}_{d'}$. Hence, $\left|\frac{\delta NN_{\vec{w}+\gamma\Delta\vec{w}}(\vec{x}_{d'})}{\delta\gamma}\right| = 0$.

Let's examine how these weight changes affect the machine's error. Partition this error into components $Error(\vec{w}) = \sum_{d'} Error_{d'}(\vec{w})$, where $Error_{d'}(\vec{w}) = (NN_{\vec{w}}(\vec{x}_{d'}) - y^*_{d'})^2$. We have seen that on any other input $\vec{x}_{d'}$, changing the γ in the weights $\vec{w} + \gamma\Delta\vec{w}_d$ has zero effect on the machine's output $NN_{\vec{w}+\gamma\Delta\vec{w}}(\vec{x}_{d'})$ and hence has zero effect on the input's error component $Error_{d'}(\vec{w} + \gamma\Delta\vec{w})$. This means only the $Error_d$ component affects the error, i.e., $\left|\frac{\delta Error(\vec{w}+\gamma\Delta\vec{w})}{\delta\gamma}\right|$
$= \left|\frac{\delta Error_d(\vec{w}+\gamma\Delta\vec{w})}{\delta\gamma}\right| = \left|\frac{\delta\left(NN_{\vec{w}+\gamma\Delta\vec{w}_d}(\vec{x}_d)-y^*_d\right)^2}{\delta\gamma}\right|$
$= \left|2\left(NN_{\vec{w}}(\vec{x}_d)-y^*_d\right)\times\frac{\delta NN_{\vec{w}+\gamma\Delta\vec{w}_d}(\vec{x}_d)}{\delta\gamma}\right| \geq 2\,|NN_{\vec{w}}(\vec{x}_d)-y^*_d|\times q.$

Let's now bound this error derivative in the other direction. The gradient used for gradient descent is $\nabla Error(\vec{w}) = \left\langle \frac{\delta Error(\vec{w})}{\delta w_1}, \ldots, \frac{\delta Error(\vec{w})}{\delta w_{M\cdot M'}} \right\rangle$ giving the slope along each axis. For any other direction $\Delta\vec{w}$ in the weight space, the slope is $\nabla_{\Delta\vec{w}} Error(\vec{w}) = \frac{\delta Error(\vec{w}+\gamma\Delta\vec{w})}{\delta\gamma}$. It is computed as $\nabla Error(\vec{w}) \cdot \Delta\vec{w} = |\nabla Error(\vec{w})| \cdot |\Delta\vec{w}| \cdot cos(\theta)$. Requirement 2 gives that this first magnitude is at most ϵ; we bound this second magnitude to be most $\frac{1}{r}$, and $cos(\theta) \leq 1$.

Combining these two bounds $\frac{\epsilon}{r} \geq \left|\frac{\delta Error(\vec{w}+\gamma\Delta\vec{w})}{\delta\gamma}\right| \geq 2q\,|NN_{\vec{w}}(\vec{x}_d)-y^*_d|$ gives as required that $|NN_{\vec{w}}(\vec{x}_d) - y^*_d| \leq \frac{\epsilon}{2qr}$. ■

What remains for the proof is to prove some simple properties of matrices.

Definition: We say that matrix $\mathbf{M}$ has *scaling factor* at least r if $Min_{\vec{v}}\ |\mathbf{M}\vec{v}| \geq r\,|\vec{v}|$. Note if $r > 0$, then the matrix has full rank.

Lemma 1 *(Scaling matrix): If matrix* $\mathbf{M}$ *has* scaling factor *at least* r *and* $\mathbf{w}$ *is not a vector but is a matrix, then* $|\mathbf{M}\mathbf{w}|_2 \geq r\,|\mathbf{w}|_2$*, where* $|\mathbf{w}|_2 = \sqrt{\sum_{\langle i,j\rangle} \mathbf{w}_{\langle i,j\rangle}{}^2}$.

Proof: $|\mathbf{M}\mathbf{w}|^2 = \sum_{columns\ j} |[\mathbf{M}\mathbf{w}]_j|^2 = \sum_j |\mathbf{M}[\mathbf{w}_j]|^2 \geq \sum_j r^2\,|\mathbf{w}_j|^2 = r^2\,|\mathbf{w}|^2$. ■

Lemma 2 *(Solving for Weights): If the columns of matrix* $\mathbf{x}$ *are linearly independent, then the equation* $\mathbf{w}\times\mathbf{x} = \mathbf{z}$ *can be solved for* $\mathbf{w}$.

Proof: Because the D columns of $\mathbf{x}$ are linearly independent, the row rank is also D. Separate the rows into $\mathbf{x} = \langle \mathbf{x_A}, \mathbf{x_B}\rangle$ where $\mathbf{x_A}$ is a square $D\times D$ invertible matrix and $\mathbf{x}_B$ contains the remaining rows. The equation $\mathbf{w}\times\mathbf{x} = \mathbf{z}$ is then rewritten and solved as follows: $\langle \mathbf{w_A}, \mathbf{w_B}\rangle\times\langle \mathbf{x_A}, \mathbf{x_B}\rangle = \mathbf{z}$; $\mathbf{w}_A\times\mathbf{x_A} + \mathbf{w_B}\times\mathbf{x_B} = \mathbf{z}$; $\mathbf{w_A}\times\mathbf{x_A} = \mathbf{z} - \mathbf{w_B}\times\mathbf{x_B}$; and $\mathbf{w_A} = (\mathbf{z} - \mathbf{w_B}\times\mathbf{x}_B)\times\mathbf{x_A}^{-1}$. Let $\mathbf{w_B}$ be zero/arbitrary and solve for $\mathbf{w_A}$. ■

2.4 Assumptions are Reasonable

We will now provide reasoning for why the assumptions in Theorem 1 can be considered reasonable.

(1) $\vec{x}'_d = NN_1(\vec{x}_d) \Rightarrow$ Linear Independence:
The requirement here is that, given the set of D training inputs $\{\vec{x}_d \mid d \in [D]\}$, the resulting set $\{\vec{x}'_d \mid d \in [D]\}$ of vectors of hidden values are sufficiently linearly independent.

Independent Input: One possibility is that the first stage NN_1 is the 1-1 function and the training inputs $\{\vec{x}_d \in \mathcal{R}^N \mid d \in [D]\}$ themselves are linear independent. This assumption is likely to hold as long as the number of bits in the input representation exceeds the number of training inputs, i.e., $D \geq N = M$. For instance, consider a scenario where the training data consists of millions of images, each of which can be compressed into a set of $N' < N$ values. If $D \geq N'$, then it's probable that the images are linearly dependent.

Features: If $D < N$, then each input $\vec{x}_d$ could be extended with sufficiently many features to make them independent.

Fix Random Layer: An alternative approach is to design the first stage NN_1 as a fully connected neural network layer with randomly set weights that remain unchanged during the learning process. With high probability, such a layer can produce an output $\{\vec{x}'_d \mid d \in [D]\}$ that is linearly independent. In an extreme case where the input consists of a single value $x_d = d$, each i^{th} hidden node will multiply x_d by some weight w_i and add a threshold w'_i before passing this through its activation function. For simplicity, let's set these values to $w_i = -1$ and $w'_i = i$ respectively, and assume a threshold activation function. This configuration results in an upper triangular 0/1 matrix defined by $[\vec{x}'_d]_i = 1$ if and only if $-d+i \geq 0$. This matrix is full rank.

Double Vandermonde Matrices: The following is another surprising example. In the previous example with single-valued inputs, consider the scenario where the multiplicative weights are $w_i = i$ and the threshold is set to zero. In this case, the pre-activation values $[d \times i]_{\langle i,d \rangle} = [i] \cdot [d]$ form a matrix with only rank one. However, the post-activation values $[sig(d \times i)]_{\langle i,d \rangle}$ create a double Vandermonde matrix with a sigmoid Fourier basis, and this matrix also has full rank. This phenomenon is similar to various other well-known matrices, such as the polynomial interpolation matrix $[i^d]_{\langle i,d \rangle}$, the Fourier transformation matrix $[sin(d \times i)]_{\langle i,d \rangle}$, and the almost sigmoid matrix $[e^{d \times i}]_{\langle i,d \rangle} = [(e^d)^i] = [(d')^i]$. The same would be true for any activation function that is non-linear enough to have a "vandermonde" property.

(3.1) Full Range: The first version of the third requirement is that the range of $NN_3(\vec{z})$ includes all of the supervisor's answers y^*_d. For this, it is sufficient that the function outputs both bigger and smaller values and is continuous.

(3.2) $NN_3(\vec{z}_d) \Rightarrow$ Sensitive: The second version of the third requirement is that the stage NN_3 is sensitive, i.e., at each input $\vec{z}_d$ considered, the function $NN_3(\vec{z}_d)$ has a large slope, i.e., the vector of derivatives $\nabla NN_3(\vec{z}_d) = \left\langle \frac{\delta NN_3(\vec{z}_d)}{\delta [\vec{z}_d]_1}, \ldots, \frac{\delta NN_3(\vec{z}_d)}{\delta [\vec{z}_d]_{M'}} \right\rangle$ has magnitude at least parameter q. Below we will justify why q should be approximately one.

Activation Functions Not Flat: When the pre-activation values become very large, the slope of the sigmoid curve decays exponentially, which can severely reduce sensitivity and slow down gradient descent. To avoid this issue, it is typically desirable to keep pre-activation values within the middle range of the sigmoid, where the slope is not too small. ReLU (Rectified Linear Unit) functions have the advantage that the slope is big on one half but could be zero on the other half.

As a theory paper, we just assume that under the current weights, the sensitivity of the neural network has not gone flat either because people have managed to keep a sufficient number of its activation functions out of their flat regions or because, as is the fact, sigmoids always have at least some non-zero slope.

Import a Threshold: In neural networks, each node typically receives a pre-activation value z, which is computed as the weighted sum of its inputs. If all these weights are zero, then this sum does not depend on its inputs. Similarly, if all its inputs are zero, then this sum does not depend on the multiplicative weights. However, regardless of other factors, the pre-activation value z does depend on the additive threshold value. Given we have assumed that the activation functions are not flat, changing the threshold value affects the output of the node. If the third stage $NN_3(\vec{z}_d)$, as given, is not sensitive to its input, one could change it by including one such threshold as one of its inputs. Then when we do gradient decent setting the weights on the middle stage $NN_{\vec{w}}$, it could automatically adjust this threshold too.

Not Shrinking or Decaying: In practical machine learning scenarios, it is desirable to prevent the values across a layer from either blowing up or decaying as we move through the layers of the network. Specifically, machine learners aim to keep the magnitudes of each hidden vector $|\vec{z}_d|$ and the machine's output $NN = NN_3(\vec{z}_d)$ close to one. This expectation implies that if the vector $\vec{z}_d$ changes by a certain distance in some direction, the output should change by a similar amount. In mathematical terms, this can be expressed as $\frac{\delta NN_3(\vec{z}_d + \gamma \Delta \vec{z}_d)}{\delta \gamma} \approx q|\Delta \vec{z}_d|$, where q is a constant close to one. Using our previous calculation, $\frac{\delta NN_3(\vec{z}_d + \gamma \Delta \vec{z}_d)}{\delta \gamma} = |\nabla NN_3(\vec{z}_d)| \cdot |\Delta \vec{z}_d| \cdot \cos(\theta)$, it can be argued that $|\nabla NN_3(\vec{z}_d)|$ should be close to q, which should be some

constant close to one.

2.5 Long Gradient Descent Path

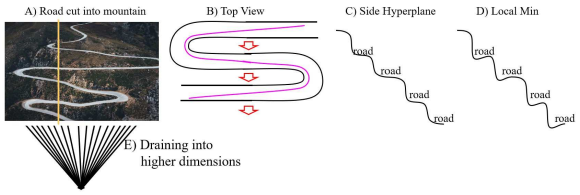

Figure 3: Long gradient decent path

A possible expectation is that having no local minima or maxima in the error surface would guarantee that the gradient descent path would traverse the space once, ensuring that the path's length matches the breadth of the space. Unfortunately, this does not hold true. Even when we obtain this result within every hyperplane of the space, it still doesn't guarantee a short path. To illustrate this, consider the example of a simple road winding down a mountain. As shown in Figure A, the road can still be arbitrarily long as it winds back and forth. Figures B-C demonstrate that the steepest descent path in such a scenario follows the road because the road's side-to-side variation is relatively flat. Figure C illustrates that all the plane cutting through the mountain road do not have local minima or flat regions. Moving to the extreme in which the neural network has a small number of hidden nodes per layer ($M << D$), the error space can contain exponentially many local minima, as seen in Figure D. Moving to the other extreme in which the number of parameters is $D^{\mathcal{O}(1)}$, gradient descent can compete in polynomial time, as demonstrated in Figure E. When gradient descent is allowed to break out of the subspace defined by A, it quickly drains into the optimal solution. These visual examples highlight the complex nature of the error surface and the behavior of gradient descent.

2.6 A Lower Bounds on (M or P) vs D

The number of hidden nodes M vs parameters P: One complaint about this paper is the focus on the number of hidden nodes M instead of on the number of parameters P. The first requirement is that the former is at least the number of training inputs D while guaranteeing there are weights with which the machine can compute the training data requires the latter to be at least D. This distinction is quite significant because the number of edges/parameters between two M node layers is M^2. We present two counterarguments to address this concern. First, our results can accommodate a scenario where $P = \mathcal{O}(M)$ parameters are utilized. This can be achieved by having just one layer with a minimum of $M \geq D$ nodes, while the other layers can consist of as few as one node. See Figure 1. Second, it remains uncertain whether D parameters or D nodes are the minimal requirements to guarantee the successful learning of the D training data.

Guaranteeing that one can learn the D training data: As stated in Section 1.1, if the number of neural network parameters P is less than the number D of training inputs, then, by simple counting, there are training outputs $\{y_d^* \in \mathcal{R} \mid d \in [D]\}$ that cannot be computed by any of the settings $\vec{w} \in \mathcal{R}^P$ of these parameters. This is tight. Given training data $\{\langle \vec{x}_d, y_d^* \rangle \mid d \in [D]\}$, the standard look-up table neural network has a hidden node for each possible input $\vec{x}_d$ which fires if this is the input. This requires $M = D$ and $P = NM$ parameters, but one can decrease this to $(N + D)$ parameters by building binary tree of nodes off the N input nodes that convert each input $\vec{x}_d$ into a unique value.

The author believes that the number of nodes M in this hidden layer is at least as important as the number of parameters P. To motivate this, consider now the architecture that contains L fully connected layers each with M nodes, giving a total of $P = LM^2$ parameters. If $LM^2 \gg D$, then there are sufficiently many parameters for the architecture to compute any training outputs. But if $D \gg M$, then as far as the author is aware, the question is unknown whether it can always perfectly compute the training data.

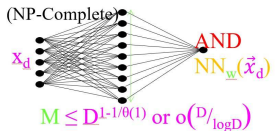

Figure 4: Architecture for NP result

NP-Complete: Avrim Blum and Ronald Rivest (1992) [3] prove that the neural network optimization problem is NP-complete when the neural network has one fully connected hidden layer with M threshold nodes and a single AND output node. Upon analyzing their work, we observe that this result remains valid when the amount of training data D is at least $\Omega(NM)$, which corresponds to the number of weights or edges in the network. To emphasize the importance

of the hidden layer's width M, we consider the scenario where N is small. Remarkably, the result still holds even when M is less than $D^{1-\frac{1}{\Theta(1)}}$. Taking this idea further, if M is just a little smaller than D, specifically $M \leq o\left(\frac{D}{\log D}\right)$, then the neural network problem does not have a poly-time algorithm unless 3SAT has a sub-exponential algorithm. This nicely balances the fact that our result applies when $M \geq D$. For completion, we discussed Blum and Rivest's reduction in more detail.

Lemma 3 *(NP-Complete): Given the training data as input, the computational problem is determining whether there are weights for which the neural network perfectly computes the D training data. The neural network considered has one fully connected hidden layer with M threshold nodes and a single AND output node. With $D \geq \Omega(NM)$ = number of weights/edges, this NN problem is NP-complete.*

Proof: Blum and Rivest prove that the *Neural Network* problem is NP-complete following the usual steps. Given a fast algorithm for the *Neural Network problem*, they design a fast algorithm for the *NOT-ALL-EQUAL 3SAT* problem. Given an instance to this 3SAT problem with $\Theta(N)$ variables and $\Theta(N)$ clauses, they construct an instance of the NN problem with N input nodes, some M hidden nodes, and $D = \Theta(NM)$ training data. In order to be a proper NP-reduction, we must keep the NN instance size $\Theta(ND)$ polynomial in the 3SAT instance size $\Theta(N)$. This imposes the relationship $D \leq N^{\Theta(1)}$. Rearranging gives NP-completeness as long as $M \leq \Theta\left(\frac{D}{N}\right) = D^{1-\frac{1}{\Theta(1)}}$. On the other hand, suppose we have a $D^{\Theta(1)}$ time algorithm for NN and want our 3SAT algorithm to run in time $2^{o(N)}$. Rearranging $D^{\Theta(1)} = 2^{o(N)}$ gives $N = \frac{\log D}{o(1)}$ and $M \leq \Theta\left(\frac{D}{N}\right) = o\left(\frac{D}{\log D}\right)$.

Each training input $\vec{x} \in \{0,1\}^N$ can be thought of as a point in the N-dimensional Boolean cube and the supervisor's answer $y_d^* \in \{0,1\}$ as a Boolean labeling of this point. Each hidden node with weights $\vec{w}_j \in \mathbb{R}^{N+1}$ can be thought of as defining an $(N-1)$-dimensional hyperplane in this space and returns zero or one depending on which side the input $\vec{x}$ lies, namely based on the sign of $\vec{w}_j \cdot \vec{x} = \sum_k [\vec{w}_j]_k \times [\vec{x}]_k$.

Avrim Blum and Ronald Rivest designed the training data to include $M-2$ spaced out copies of the following gadget designed to waste one hidden node. The gadget can be thought of as labeling one corner of the hyper-cube 0 and all its N neighbors 1. More formally, the gadget contains one training input $\vec{x}_d$ whose supervisor's answer is $y_d^* = 0$. It also contains the N inputs $\vec{x}_{d'}$ that are hamming distance one from $\vec{x}_d$ whose supervisor's answer is $y_{d'}^* = 1$. Recall that the output is the AND of these hidden nodes. Blum and Rivest argue that any neural network $NN_{\vec{w}}(\vec{x})$ getting the correct answers on this training data must use one of its hidden node's hyperplane to separate this 0 point from the rest of the hyper-cube. Note that the 2^N point N-cube has enough room to have up to $\frac{2^N}{N+1}$ of these gadgets. Adding $M-2$ of them adds a total of $D = (N+1)(M-2)$ items to our training data.

There will be two hidden nodes left. Blum and Rivest prove that determining how to use these two remaining hidden nodes is NP-complete. For this, they use $D = \Theta(N)$ additional training data. The proof is a reduction to the Set-Splitting problem, which has a simple reduction to the NOT-ALL-EQUAL 3SAT problem with $\Theta(N)$ variables and $\Theta(N)$ clauses.

The total amount of training data used is then $D = (N+1)(M-2) + \Theta(N)$. ■

3 Conclusion

We prove for a much smaller network than Hui Jiang (2019) [5] that, when gradient descent finds critical weights, i.e., a point $\vec{w}$ on the error surface with zero slope, the resulting neural network $NN_{\vec{w}}(x)$ provides perfect responses for each training data point. Our machine must possess at least one hidden layer with a node count M equal to or greater than the number of training data points, denoted as D. Furthermore, our findings establish the first quantifiable link between the criticality of $\vec{w}$ and the accuracy of the machine's approximation to the supervisor's responses. The number of weights needed to achieve this result is nearly optimal because reducing the number of weights further would render the weight optimization problem NP-complete.

References

[1] Z. Allen-Zhu, Y. Li and Z. Song. A Convergence Theory for Deep Learning via Over-Parameterization. ArXiv. /abs/1811.03962 *Proceedings of the 36th International Conference on Machine Learning,* 242–252, 2019.

[2] P. Auer, M. Herbster, and M.K. Warmuth. Exponentially many local minima for single neurons. In Proc. of Advances in Neural Information Processing Systems 8 (NIPS), 1995.

[3] Avrim L. Blum and Ronald L. Rivest, Training a 3-node neural network is NP-complete. In *Neural Networks, 5(1), 117-127. https://doi.org/10.1016/S0893-6080(05)80010-3* 209–213, 1992.

[4] S. Bubeck and M. Sellke A Universal Law of Robustness via Isoperimetry. ArXiv. /abs/2105.12806. 2021.

[5] H. Jiang, Why Learning of Large-Scale Neural Networks Behaves Like Convex Optimization. ArXiv/abs/1903.02140. 2019

[6] John Tsotsos. Personal communication