Gradient Descend

Luca Brodo
Hochschule Hamm-Lippstadt
Deep Learning
Lippstadt, germany
luca.brodo@stud.hshl.de

***Abstract*—The objective of this paper is to give to the reader a thorough overview of the Gradient Descent Algorithm applied the area of Machine Learning. In this paper the different variations of this algorithm will be introduced and explained. In order to offer a complete understanding, both the mathematical foundations and application will be presented. In order to offer a complete view to the reader, the algorithm will be applied both to a linear regression model and to a deep neural network model.**

I. Motivation

The main challenge in the area of Machine Learning for both researchers and practitioners is to train the model as efficiently as possible. In the context of Deep Neural Network models the training is based on the back propagation algorithm, which propagates the errors from the output layer to the front one, updating the variables layer by layer, and it's based on gradient descent optimization algorithms.A lot of different algorithms have been introduced to better improve the performances of the models,e.g., Newton's method, etc... These algorithms, however, are based on high-order derivatives and in practice tend to be slower than the first-derivative-based Gradient Descent and its variants. In the context of linear regression based models, performances are also important, and an optimized Gradient Descent algorithms may result vital when the data set is relatively big. Usually, they are often used as black-box optimizers by state-of-the-art Deep Learning library, e.g. lasagne, caffe, and keras documentation, but an ad-hoc implementation for a specific dataset may be more efficient. To achieve so, a full understanding of the algorithm is necessary.

II. Introduction

The aim of this paper is to provide a thorough understanding of the Gradient Descent algorithm for optimization in the area of Machine Learning. In this section, a first introduction to the algorithm will be provided. Subsequently, in section 2, the mathematical foundations of the algorithm will be layed down. In the next three sections, the three different versions of the algorithm, namely the Batch, Stochastic and Mini-Batch Gradient Descent, are discussed in detail along with a real world application. Finally, the 3 versions are compared and additional strategies which are helpful for optimizing gradient descent are introduced.

As already mentioned, the gradient descent is an algorithm that allows the optimization of various machine learning models. In order to achieve this, the algorithm is based on an iterative process for finding a local minima of a differentiable function. At every iteration, the step in the opposite direction of the gradient[1] of the function at the given point is taken. This algorithm is usually attributed to Cauchy, who first suggested it in 1847.

III. Mathematical Foundations

The algorithm moves from the observation that, taken a multi-variable,defined and differentiable in a point $\mathbf{a}$ function $F(x)$, it will decrease fastest whenever one goes from $\mathbf{a}$ towards $-\nabla F(x)$ (opposite to the gradient of $F(x)$). It follows, then:

$$\mathbf{a_n} = \mathbf{a_{n-1}} - \alpha \nabla F(\mathbf{a_{n-1}}) \tag{1}$$

where α is small enough, so that $F(a_{n-1} \geq a_n)$.

In other words, by subtracting $\alpha \nabla F(\mathbf{a_{n-1}})$ to $\mathbf{a_{n-1}}$, one moves against the gradient, toward the local minima as fast as possible.
By considering each point $\mathbf{a_0}, \mathbf{a_1}, \mathbf{a_2}, ...\mathbf{a_n}, \mathbf{a_{n+1}}$, such that $\mathbf{a_{n+1}} = \mathbf{a_n} - \alpha \nabla F(\mathbf{a_n})$, one will obtain the following monotonic sequence:

$$F(a_0) \geq F(a_1) \geq F(a_2) \geq F(a_3) \geq F(a_4) \geq ...$$

This sequence will converge to the local minima of the function. [3]

A. Real World Example

The above described algorithm can be better visualized with a real world example. Let's assume a situation in which one finds himself on the top of a mountain and needs to reach the valley. In case the visibility is very low, for example because of extreme fog, one solution can be to walk some steps and to always choose the steepest path. If one keeps going like this, eventually the valley will be reached. This is the main idea of the algorithm.

[1]The gradient represents in which point the function changes the most

IV. Machine Learning Application

When training a model, a so called ”Cost function” is used to express the average loss over the data set.Calculating the slope, or gradient, of this function at a given point defines how to change the parameters of the model to make it more accurate.By finding the minima of this function,the perfect value of the parameters that minimizes the cost of the model is found. Recalling the previous section, the concept of taking steps opposite to the gradient has been introduced. These steps, in the context of machine learning, are the learning rate. Choosing the correct value for the learning rate will produce a behavior like the one shown in Fig 1,where the minima of the function is found. On the other hand, Choosing distant steps will introduce overshooting and imprecision, while using close steps is more precise, but introduces calculation overhead since the gradient is recalculated very frequently, thus making the learning process slower [1]. This behavior can be seen in Fig 3.

Recalling the real world example in the previous section, let's imagine that the person on the mountain had with them a tool capable of measuring the steepness of the mountain at a specific point. by walking too much in one direction, it may be the case that that the value measured at the destination point is bigger that the one measured at the starting point. However, if the person uses the tool too many times, the process will be slowed down by the continuous calculations of the tool. Finally, not all cost functions will be parabolas like the ones shown in the previous pictures. This means that, instead of the local minimum, the gradient descent might stop at one local minimum, which is not as good as the global minimum [1]. This result can be seen in the right side of Fig 2. In addition, if an improper learning rate is chosen, this might result in a situation like the left side of Fig 2, where the pace of gradient descent is not fast enough to reach the global minimum[2]. To solve this problem, a cost function without local minimas should be used. To insure so, the Rolle Theorem can be used [2]. For the scope of this paper, the Mean Squared Error (MSE) cost function will be used and it has been proven not to have local minimas [1].

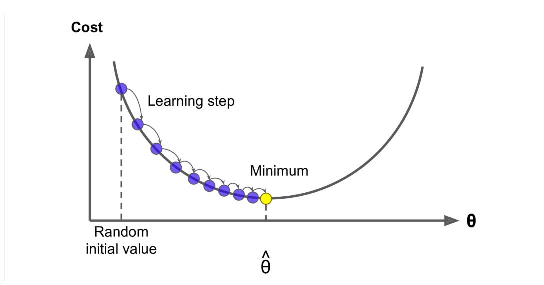


Fig. 1. Gradient descent Normal Behavior [1]

[2]The steps are not far enough and the algorithm is stuck in a plateau

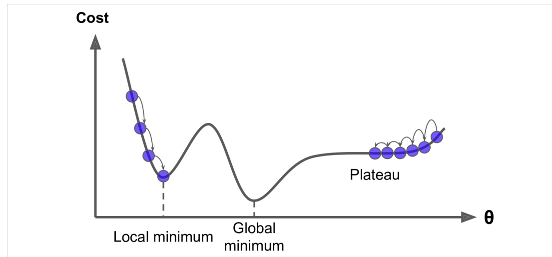


Fig. 2. Gradient descent pitfalls [1]

A. Development Environment

An implementation example will be developed for all the versions of the algorithm using Python 3.9 and Scikit Learn 0.24.1. The data set called ”Pima.tr” will be used to test the various implementation and it will be split beforehand into a training batch consisting of around 150 elements and a test batch of around 50. Fig n. 4 shows the data set.

V. Batch Gradient Descent Application

A. Theory behind the algorithm

The first variation of the algorithm is called Batch Gradient Descent Application, or Vanilla gradient descent, and can be thought as the naïve implementation of the algorithm. [4] The idea of this algorithm is to translate the mathematical representation into code. Given a training set τ, the *Batch Gradient Descent algorithm* optimizes the model variables with the following equation:

$$\Theta^{(i)} = \Theta^{(i-1)} - \alpha \nabla \mathscr{L}(\Theta; \tau) \tag{2}$$

where:

1) $\Theta^{(\tau)}$ denotes the model parameters vector at iteration i
2) $\nabla \mathscr{L}(\Theta; \tau)$ denotes the gradient of the loss function $\mathscr{L}(\Theta; \tau)$.The notation $(\Theta; \tau)$ indicates that the parameters Θ are taken with the whole τ. The loss function for the purpose of this paper will be defined as the Mean Squared Error (MSE) and will take the form of $\mathscr{L}(\Theta) = \frac{1}{N}\sum_{i=1}^{n}(\theta x_i - y_i)^2$.
3) α denotes the *learning rate* in the gradient descent algorithm, which is usually very small (e.g. 10^{-4})

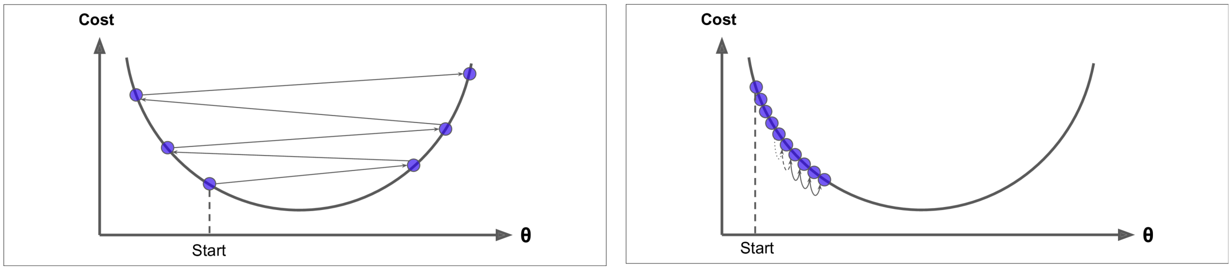
The implementation of this type can be seen in Algorithm N.1.

The algorithm, however, can be optimized by working on Equation n. 2. Recalling that the loss function was defined as:

$$\mathscr{L}(\Theta) = \frac{1}{N}\sum_{i=1}^{n}(\theta x_i - y_i)^2$$

Let's find its derivative:

$$\frac{d\mathscr{L}}{d\Theta} = \frac{2}{N}\sum_{i=1}^{n} x_i(\theta x_i - y_i)$$



(a) Gradient Descent Overshooting

(b) Gradient Descent using small steps

Fig. 3. Different behavior of the gradient descent with different type of learning rate [1]

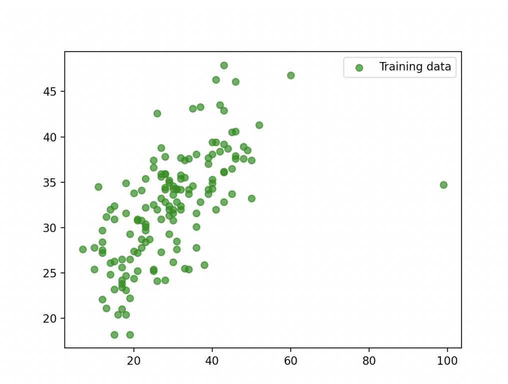


Fig. 4. Training data set

Algorithm 1 Vanilla Gradient Descent

Require: Training set τ, Learning Rate α, Mean Squared Error(MSE): $\mathscr{L}(\Theta;\tau)$

Ensure: Model Parameter θ

1: **for** i in *iterations* **do** ▷ The amount of iterations is arbitrary

2: Compute the gradient $\nabla\mathscr{L}(\theta;\tau)$

3: Update Variables $\Theta = \Theta - \alpha\nabla\mathscr{L}(\theta;\tau)$

4: **end forreturn** model variable Θ ▷ The vector Θ contains the update version of w and b

---

If we rewrite this in vector form, we find:

$$\nabla\mathscr{L}(\Theta;\tau) = \begin{bmatrix} \nabla_\Theta\mathscr{L}(\Theta;\tau)_1 \\ \nabla_\Theta\mathscr{L}(\Theta;\tau)_2 \\ \vdots \\ \nabla_\Theta\mathscr{L}(\Theta;\tau)_m \end{bmatrix} = \frac{2}{N}\tau^{\mathbf{T}}(\tau\mathbf{\Theta} - \mathbf{y}) \qquad (3)$$

Using (3) the dot product is computed and it allows to compute all the partial derivatives in one go [1]. Although this is faster than the previous one, the last formula will not solve the main problem of this version. According to the description, using this variation of the algorithm, to update the model variables, the computation of the whole gradient of the loss function at every iteration is needed. This introduces computation overhead by redundancy which makes the training slow and does not allow online training.

B. Application

In Fig 5 a line-to-line implementation of the algorithm using python can be seen. This implementation is a revised implementation of the one found in [1]. The main difference is that [1] uses a random dataset.

```
def vanilla_gradient(x_train,y_train):
    eta = 0.001  # learning rate
    n_iterations = 40000
    y = y_train.values.reshape(-1, 1)
    m = len(x_train.values)
    X = x_train.values.reshape(-1, 1)
    X_b = np.c_[np.ones((len(x_train.values), 1)), X]
    theta = np.ones((2,1))

    iteration =0
    while iteration < n_iterations :
        gradients = 2 / m * X_b.T.dot(X_b.dot(theta) - y)

        theta = theta - eta * gradients
        iteration+=1

    return theta
```

Fig. 5. Implementation of the vanilla gradient descent

C. Results

The algorithm in Fig 5 is measured against the standard implementation found in the Sklearn library. The result are demonstrated in Fig 6. On top of the graph the time taken by the algorithm is shown. The graph shows clearly that the two implementations produce the same model. However, this result depends mainly on the number of iterations and the eta chosen beforehand. These two values change dependently on each other as shown in Fig 7 and Fig 8. In Fig 7 an eta = 0.0001 is chosen. This means that a higher number of iterations (in this case around 16000) needs to be chosen to obtain a good enough model. [1] This results in a longer time of execution (around 0.6709s compared to 0.1574s of the previous implementation). In Fig 8 the result of the opposite approach is shown. With an

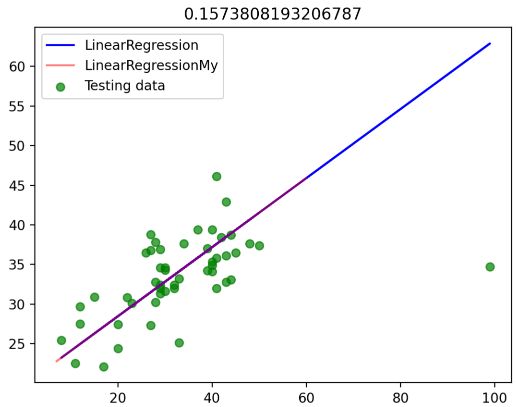


Fig. 6. Result of the implementation with a correct value of eta and the correct number of iterations

eta bigger than the proper one (eta = 0.001) the algorithm will not converge to a suitable value [1], therefore only a few iterations are necessary to show the model it produces (in this case, around 10). As a result, the time of execution turns out to be lower (around 0.00015s).

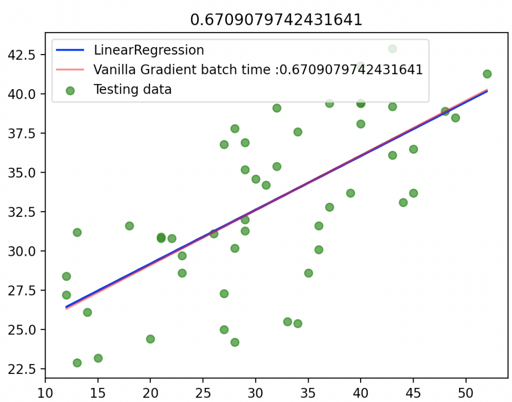


Fig. 7. Result of the implementation with a small value for eta and a big number of iterations

D. *Convergence Rate*

Since MSE is convex and does not change abruptly, the Batch Gradient Descent with a fixed learning rate will eventually converge to the optimal solution. It will take, however, $0(1/\epsilon)$ iterations to reach the optimum within a range of ϵ depending on the shape of the cost function. By dividing the tolerance by N, the algorithm will need to run 10 times longer. [1]

VI. Stochastic Gradient Descent Application

To improve efficiency, the *Stochastic Gradient Descent Application* updates the parameters by computing the loss function gradient instances by instances. In other words,

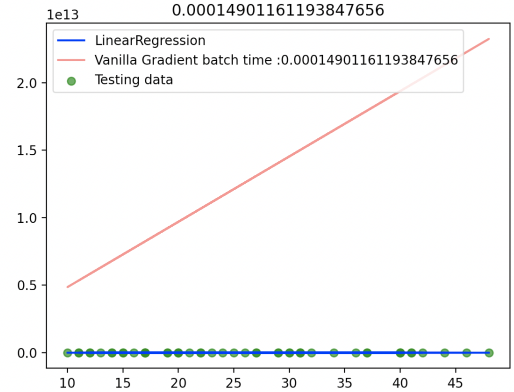


Fig. 8. Result of the implementation with a big value for eta and a small number of iterations

the gradient will be computed only once for every instance of our dataset. This means, equation (2) will become:

$$\Theta^{(i)} = \Theta^{(i-1)} - \alpha \nabla \mathscr{L}(\Theta; (x_i, y_i)) \tag{4}$$

Algorithm 2 Stochastic Gradient Descent Application

Require: Training set τ, Learning Rate α, Mean Squared Error(MSE): $\mathscr{L}(\theta; \tau)$

Ensure: Model Parameter θ

1: **for** i in *iterations* **do** ▷ The amount of iterations is arbitrary
2: Shuffle τ
3: **for** each instance $(x_i, y_i) \in \tau$ **do**
4: Compute the gradient $\nabla \mathscr{L}(\theta; (x_i, y_i))$
5: Update Variables $\Theta = \Theta - \alpha \nabla \mathscr{L}(\Theta; (x_i, y_i))$
6: **end for**
7: **end forreturn** model variable Θ ▷ The vector Θ contains the update version of w and b

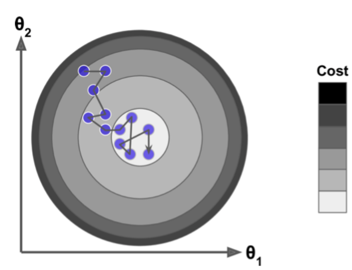


Fig. 9. Concept of the stochastic gradient algorithm

While this implementation reduces the number of computations, the main draw back is that, due to the random nature of the algorithm, this is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. However the possibility to overshoot and jump out of the local optimum is high,so*Stochastic Gradient Descent Application* has a better chance of finding the global minimum than *Batch Gradient Descent Application* does. Fig 9 shows the described behavior. Therefore randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum. One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum. This process is akin to simulated annealing, an algorithm inspired from the process of annealing in metallurgy where molten metal is slowly cooled down. The function that determines the learning rate at each iteration is called the learning schedule. If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum. If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early. [1].

A. *Application*

The application of this variation of the algorithm uses the same data-set as the one before with surprising the results. According to [1], the two variations should give more or less the same result. This is probably the case when the data-set does not contain a lot of noise and it presents a pretty linear behavior. However, as shown in fig n. 4, the data set used for this example contains a certain degree of noise that can influence the outcome of the algorithm. In addition, the stochastic gradient works better with non-convex, smooth error functions. The function used in this paper, as explained previously, is the MSE, which is a convex function, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve. This implies that there are no local minima, just one global minimum. [1]

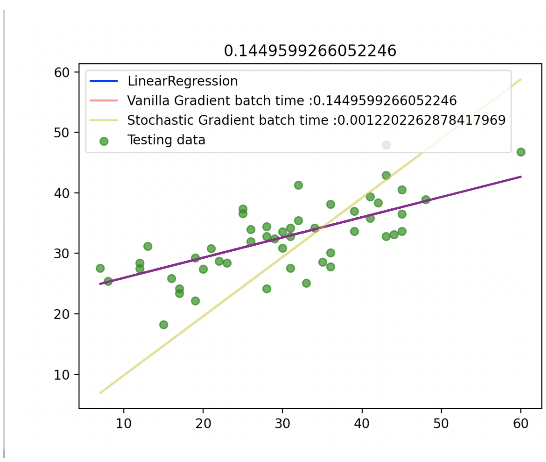


Fig. 10. Result of the implementation of the stochastic gradient descent

The time needed by this version to come up with a behavioral model, however, is significantly less than the one needed by the vanilla gradient descent.

VII. Mini-Batch Gradient Descent Application

To balance between the *Vanilla Gradient Descent Application* and the *Stochastic Gradient Descent Application* an approach which lies in between the two can be taken. Instead of taken the whole dataset to compute the gradient, we can divided it in sub-sets called Mini Batch. This approach is named *Mini-Batch Gradient Descent*. Formally, let $\beta \subset \tau$, we can rewrite (2) as:

$$\Theta^{(i)} = \Theta^{(i-1)} - \alpha \nabla \mathscr{L}(\Theta; \beta) \tag{5}$$

Algorithm 3 Mini-Batch Gradient Descent Application

Require: Training set τ, Learning Rate α, Mean Squared Error(MSE): $\mathscr{L}(\theta; \tau)$, Mini-batch size b

Ensure: Model Parameter θ

```
for i in iterations do
    Shuffle τ
    for each sub-set β ⊂ τ do
        Compute the gradient ∇ℒ(θ; β ⊂ τ)
        Update Variables Θ = Θ − α∇ℒ(Θ; β ⊂ τ)
    end for
end forreturn model variable Θ
```

In the *Mini-batch Gradient Descent Application* the mini batches are usually sampled sequentially from τ, which means that τ is divided in multiple subsets of size b[3], and these batches are picked one by one to train the model. On the other hand, some versions select the batches randomly and this is referred to as the random mini-batch generation process.Compared with vanilla gradient descent, the mini-batch gradient descent algorithm is much more efficient especially for the training set of an extremely large size. Meanwhile, compared with the stochastic gradient descent, the mini-batch gradient descent algorithm greatly reduces the variance in the model variable updating process and can achieve much more stable convergence. [4]

VIII. Conclusion

[1] presents a very well comparison between the three variations and it can be seen in Fig n.12. The Vanilla Gradient descent, or "Batch", converges smoothly towards the minimum value of the function, while the stochastic and mini-batch ones tend to over-shoot and never to settle into a proper minimum. This is due to their more random nature. These two last variations, however, tend to reach the zone of the minimum faster, i.e. with less iterations and less computational overhead. This is the trade-off that needs to be considered when choosing one of the variations.

[3]b should not be very large, usually 64,256 or 128

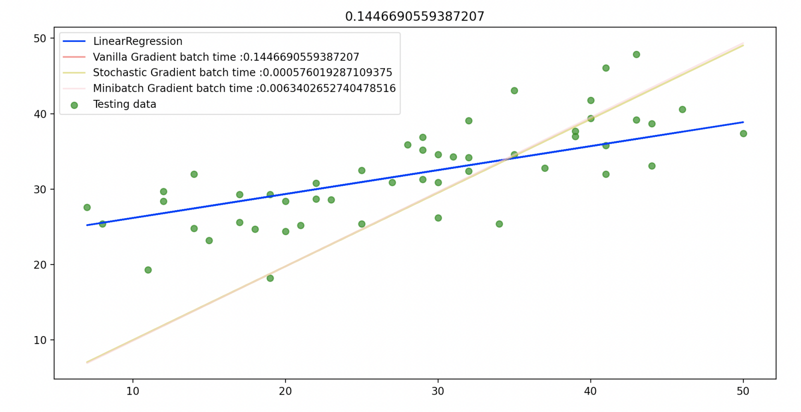


Fig. 11. Result of the mini batch gradient descent algorithm

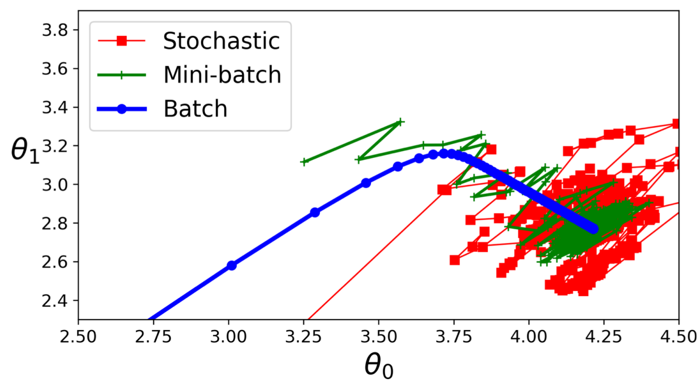


Fig. 12. Comparison between the 3 variations

IX. Declaration of Originality

I hereby confirm that I have written the accompanying paper by myself, without contributions from any sources other than those cited in the text and acknowledgements. This applies also to all graphics, drawings, maps and images included in the paper. The paper was not examined before, nor has it been published.

References

[1] A. Géron. Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems. CA: O'Reilly Media, Inc, 2019.
[2] Wikipedia. Rolle's theorem, 2021.
[3] Wikipedia. Wikipedia gradient descent, 2021.
[4] Jiawei Zhang. Gradient descent based optimization algorithms for deep learning models training. CoRR, abs/1903.03614, 2019.