AIML - CS 337

Lecture 20: Gradient Descent

27 October 2022

Lecturer: Abir De *Scribe: Group 41 & Group 42*

Till now, we haven't discussed minimization of **Supervised Learning Algorithms yet**. In this lecture, we cover the optimisation technique **Gradient Descent**; the working of the algorithm, the cause of stochastic behaviour of gradient descent and how to utilise this stochastic nature to our advantage. We also see applications of gradient descent in an upcoming technology known as *MLaaS* (*M*achine *L*earning *a*s *a S*ervice). This lecture is mainly theoretical in nature where many concepts are just introduced at a very high level without getting into the Maths and detailed theory involved.

1 Introduction

Optimisation is a central requirement in machine learning. For instance, consider we have a dataset $\mathcal{D}$ with the i^{th} data point $(x_i, y_i) \in \mathcal{D}$. Then, for a given model $f(x) = \theta^T x_i$ loss function, we have

$$L(\theta) = \sum_{i \in \mathcal{D}} L(\theta^T x_i, y_i) \tag{1}$$

And, we seek to solve the following optimisation problem

$$\min_{\theta} L(\theta) \tag{2}$$

Let us represent the initial value of the model (the starting point) as θ^0. We seek the following assurance from our optimization algorithm

$$\theta^0 \xrightarrow{???} \theta^* = \arg\min_{\theta} L(\theta)$$

Thus, whatever the initial value of the weights/learnable parameter (θ^0), we obtain θ^*

We now answer some crucial questions about θ^0 such as:

- How do we choose θ^0 and what effect does this choice have on our results?
- Do we converge to 'the' minima on starting from any θ^0?

2 The Gradient Descent Setup

Gradient Descent begins with a starting value, θ^0. Then, at each intermediate value of the model, it computes the gradient of the loss function with respect to the model parameters and updates the

model a fixed constant (called learning rate) times the gradient in the 'direction' opposite to that of the gradient. Thus, we express

$$\theta^t = \theta^{t-1} - \eta_t \nabla_\theta L(\theta) \tag{3}$$

where η_n is the learning rate at time step 't' and θ^t represents the model weights/parameters after 't' iterations and we terminate when a suitable convergence condition is met. The value of θ thus obtained is claimed to be θ^*

The aim of gradient descent (GD) algorithms is to minimize function l with respect to parameters $\mathbf{w}$ by updating them towards negative gradient. Whether the algorithm converges to a global or local minimum, or at all, depends on the initialization of parameters, step size (= learning rate), and properties of the loss function such as convexity.
The pseudocode for GD is as follows :

Algorithm 1 Gradient Descent

$\mathbf{w}_0 \leftarrow \mathbf{r}$ ▷ $\mathbf{r}$ is a randomly initialised vector
$i \leftarrow 0$
while ($\mathbf{w}$ `has not converged`) **do**
    $\mathbf{w}_{i+1} \leftarrow \mathbf{w}_i - \eta_i \cdot \text{Grad}(\ell, \mathbf{w}_i, \mathbf{x}, \mathbf{y})$
    $i \leftarrow i + 1$
end while

Here, Grad$(\cdot)$ is substituted according to the algorithms being GD, Mini-batch GD or SGD. Note that the hyperparameters $\mathbf{w}_0$ and η can be tuned and affect the speed as well as convergence result of the algorithm (in finite steps).

2.1 Vanilla/Batch Gradient descent

When the number of training samples is relatively small, an effective method to update the parameters is to compute the gradient using the whole training set at each iteration:

$$\begin{aligned}\mathbf{w}_{i+1} &= \mathbf{w}_i - \eta_i \nabla_w l(\mathbf{w}_i) \\ &= \mathbf{w}_i - \eta_i \sum_{(x,y)\in\mathcal{D}} \nabla_w \ell(\mathbf{w}_i, \mathbf{x}, \mathbf{y})\end{aligned}$$

where η_i is the learning rate. Note that the learning rate does not necessarily have to be constant during the search. Even though batch GD converges with comparably small number of iterations, calculating the gradient with a large training set can be computationally very inefficient.

Note: This is also referred to as Batch Gradient Descent in multiple texts.

2.2 Mini-batch gradient descent

In mini-batch GD, the gradient is computed by using a smaller subsets of the training set at each iteration. If $B_1, B_2, ... \subset \mathcal{D}$, is a partition of the training set, then the parameters are updated by

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \eta_i \sum_{(x,y)\in \boldsymbol{B}_i} \nabla_w \ell(\mathbf{w}_i, \mathbf{x}, \mathbf{y})$$

It is likely that the algorithm does not converge before using all the batches, and in this case the batches are shuffled and re-iterated.
When the size of dataset is large it is a difficult job to store the gradients calculated using all of the training instances. This is a limitation caused due to the **GPU memory**.

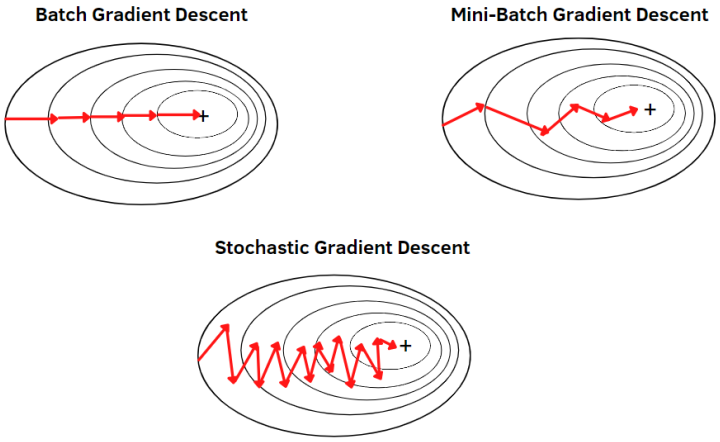
To resolve this issue, mini-batch GD is adopted with a feasible batch size (practically, it is around 10-20 for Computer Vision applications with 360p image resolutions and 2-4 in NLP applications involving documents). It improves the speed of convergence as well since quicker (but noisier) updates are made to the parameters.

2.3 Stochastic gradient descent

In stochastic GD, the training set is shuffled and the gradient is computed using a single (random) instance at each step. In every iteration, the training set is randomly shuffled and such updates are performed for all instances :

$$\mathbf{w} = \mathbf{w} - \eta_i \nabla_w \ell(\mathbf{w}, \mathbf{x}, \mathbf{y})$$

where $(\mathbf{x}, \mathbf{y})$ is the random instance for one of the updates. In this algorithm, it is not needed that the updates are along the direction of gradient vector but the expectation of updates (for every pass over the training set) is parallel to the gradient.



The figure above compares the three algorithms in their descent trends. To recall, Batch GD considers all of the training set, Mini-Batch GD considers small batches of training instances, Stochastic GD considers only single instances for gradient updates. Thus, the direction along which the update vector (Δw term) points and the smoothness of the descent can be explained.

2.4 Hyper-parameters

There are 2 hyperparameters which gradient descent needs, namely the starting point θ^0 and the learning rate η_t (possibly variable as a function of t)

2.4.1 Choice of Learning Rate η

This famous graph below[1] that shows how a learning rate that is too big or too small affects the loss during training.

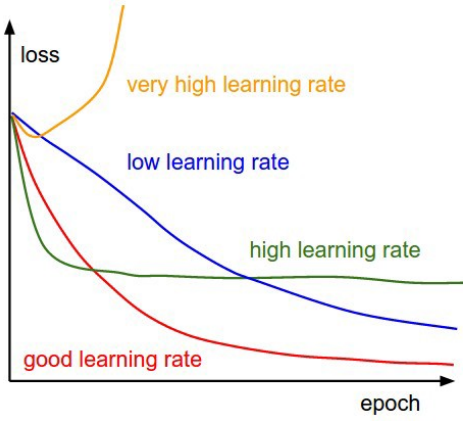


Figure 1: Choice of Learning Rate η

With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line).

2.4.2 Choice of θ^0

We observe, and can show, that for a given learning rate, the various values of θ^0 can give different values of θ^*. An important question lies ahead. Suppose we choose various values of the starting model, say

$$\Theta^0 = \{\theta_1^0, \theta_2^0 \cdots \theta_n^0\}$$

and run gradient descent to obtain the corresponding optimal values (for which loss is minimised) as

$$\Theta^* = \{\theta_1^*, \theta_2^* \cdots \theta_n^*\}$$

Should we choose the best value in Θ^* or the average value and why?

Claim 2.1. *In case of various optimal values θ^* corresponding to various starting values, θ^0, we choose the best value of θ^* (for which loss is minimised)*

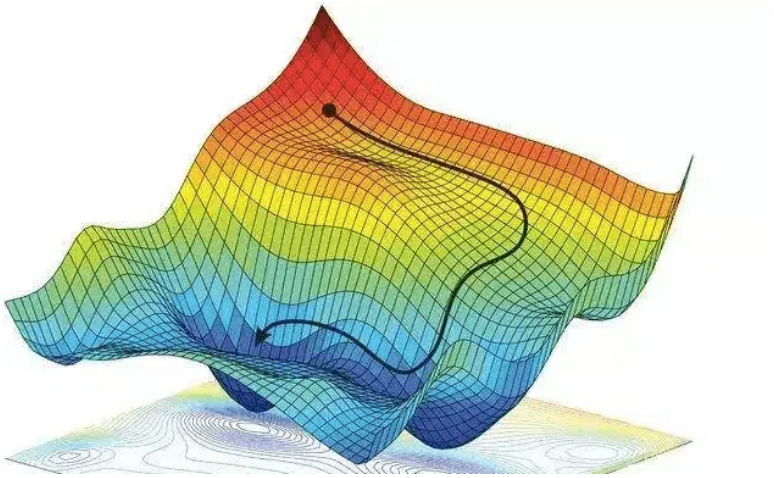


Figure 2: Plot of Loss(z-axis) vs θ_1(x-axis), θ_2(y-axis)

A justification to this is that, the choice of initial value can result in the gradient descent algorithm converging to a local minima and not being able to identify a "better" minima. Thus, the choice of various starting points allows the algorithm to explore various minima points and then choosing the best allows us to choose the more optimal value.
Note: The comparisons are done on the basis of performance on the **validation set**.

2.5 Initialisation During Coding

- Let us assume that we run optimisation multiple times, starting with random $\mathbf{w}_0^1, \mathbf{w}_0^2, \ldots$ each time resulting in validation errors as $\epsilon_1, \epsilon_2, \ldots$
  Question: How do we +decide which $\mathbf{w}_0^i$ to rely upon?
  Claim: Choose that $\mathbf{w}_0^i$ which gives least validation error ϵ_i.
  Justification: As explained above, lower validation error means better generalisation and weights giving least error signify the (yet) best found minima.

- In mini-batch gradient descent fix the initial weight parameter $\mathbf{w}_0$ and train over various batch sequences $\{\mathcal{B}_0^i, \mathcal{B}_1^i, \mathcal{B}_2^i, \ldots\}_{i=0}^N$
  Assume that $\mathbf{w}_0$ generates error ϵ_i for the ith sequence
  Question: While comparing various initialisations of $\mathbf{w}_0$, what metric should we use to judge them? Best error i.e. $\min(\epsilon_i)$ or average error i.e. $\bar{\epsilon}$?
  Claim: Use the average error i.e. $\bar{\epsilon}$ and not $\min(\epsilon_i)$ as a judging metric
  Justification: Relying on the sequence of batches is equivalent to using validation for training the model and this results in overfitting. Average error is equivalent to $\mathbb{E}[\epsilon]$ when a large #batches are tried and is a better way to judge the parameter.

- **Question**: Where exactly do ML libraries initialise the weights?

They would be initialised for an instance of the model when the model object constructor is called. The parameters can also be sampled from various distributions.[3]

```
class MyModel(self, ...):

    def __init__:
        self.linear = nn.Linear(in_features, out_features)
        #This is the line where initialization occurs with RANDOM weights

    def initialize(self):
        self.linear.weight.data = torch.randn(size)
        self.linear.bias.data = torch.randn(size)
        #This is the function where we initialize weights and biases
    according to the distribution of our choice

if __name__ == "__main__":
    model_instance = MyModel(...)
    #Writing this is fine, it still allows random initialization of
    weights
    model_instance.initialize()
    #This is optional, we write it only if we want to initialize weights
    with a distribution of our choice
```

We have mentioned that `Torch` automatically initializes weights of the Linear Layer when we create an instance of `nn.Linear`. One might ask what distribution did `Torch` use when it initialized the weights.
`Torch` uses $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$ to sample the weights.
Practically, we often observe a certain randomness in our results even when we use the same hyperparameters and model. What are the sources of this randomness?

1. When we use **Batch Gradient Descent**, we use the gradients of the *entire dataset* to move one step. That is, we will compute the gradient of loss with contribution from each point of the dataset and move in the direction opposite to this computed gradient. In such a scenario, the only source of randomness in a model (assuming we are not using other stochastic techniques such as **dropout**) is the *initialization of* θ^0
2. When we use **Mini-Batch Gradient Descent**, the choice of batches is also a source of randomness which has an effect on the final outcome. In each iteration, we choose different partitions of the data set to make batches and these partitions are often random, which results in non-determinism.

3 Mini-Batch Gradient Descent

Above, we would compute the loss function over the entire dataset, however, this suffers from one major problem; it is **slow to converge**[2]. Hence, it is prudent to use batching; training on chunks of the dataset at a time.

3.1 Dependence on Batch Order

We observe that the various orders of batches can give different values of θ^*. An important question lies ahead. Suppose we choose various batch orders of the starting model, say

$$\Omega = \{O_1, O_2 \cdots O_n\}$$

and run gradient descent to obtain the corresponding optimal values (for which loss is minimised) as

$$\Theta^* = \{\theta_1^*, \theta_2^* \cdots \theta_n^*\}$$

Should we choose the best value in Θ^*? Or the average value and why?

Claim 3.1. *In case of various optimal values θ^* corresponding to various batch orders, θ^0, we choose the average value of θ^* (for which loss is minimised)*

A justification to this is that, the choice of 'best' batch order depends on the validation dataset and can result in the gradient descent algorithm overfitting on the validation dataset. Even if we employ early stopping to prevent overfitting on the training set, we will overfit on the validation set, since we're trying to 'train' the batch choice on the validation set. For it to generalise well, we choose the average over all the optimal values obtained through the various batch orders.

We should have a **high generalization** on the validation set. If on changing batches, we get a huge change in validation error, that means the model is **not stable with respect to batches** and in that case, one should change the model or tune some other hyperparameters.

3.1.1 Questions to Think about on this topic

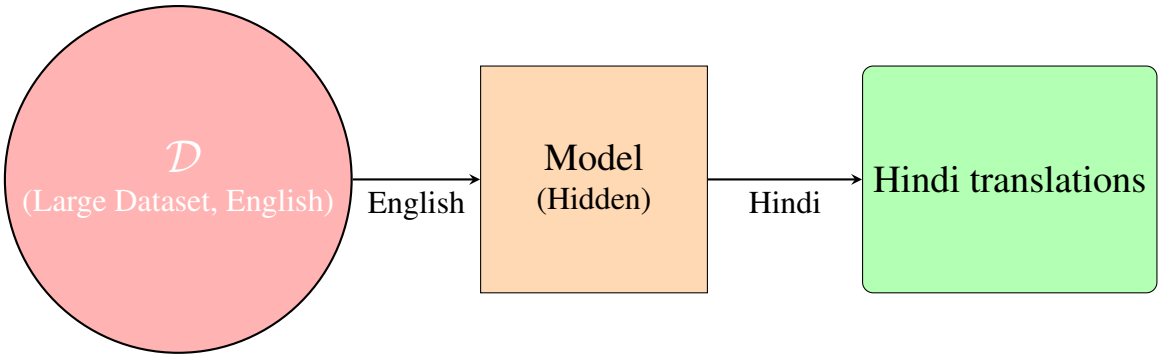
1. Can you prove an equivalence between L-2 Regularization and mini-batch gradient descent? Mini-batch gradient descent can also be used to prevent overfitting like L-2 regularization. We can say that under certain conditions, mini-batch gradient descent gives us similar results on running optimization algorithms. Can you find this strong connection?

2. Suppose you have a 24 Gigabyte GPU and you wish to run a training algorithm with 500 epochs at max. Assume that other parameters are given, then how would you choose your batch choice?

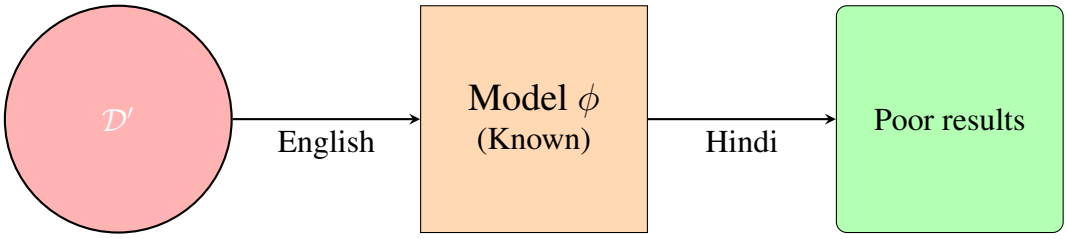
4 Machine Learning as a Service

Have you ever used a service which has an online chatbot? Say, on a banking website which employ models which use NLP techniques in creating such chatbots. MLaaS is a cloud computing service in which clients can use powerful models for their own use without having the hassle of research and development.

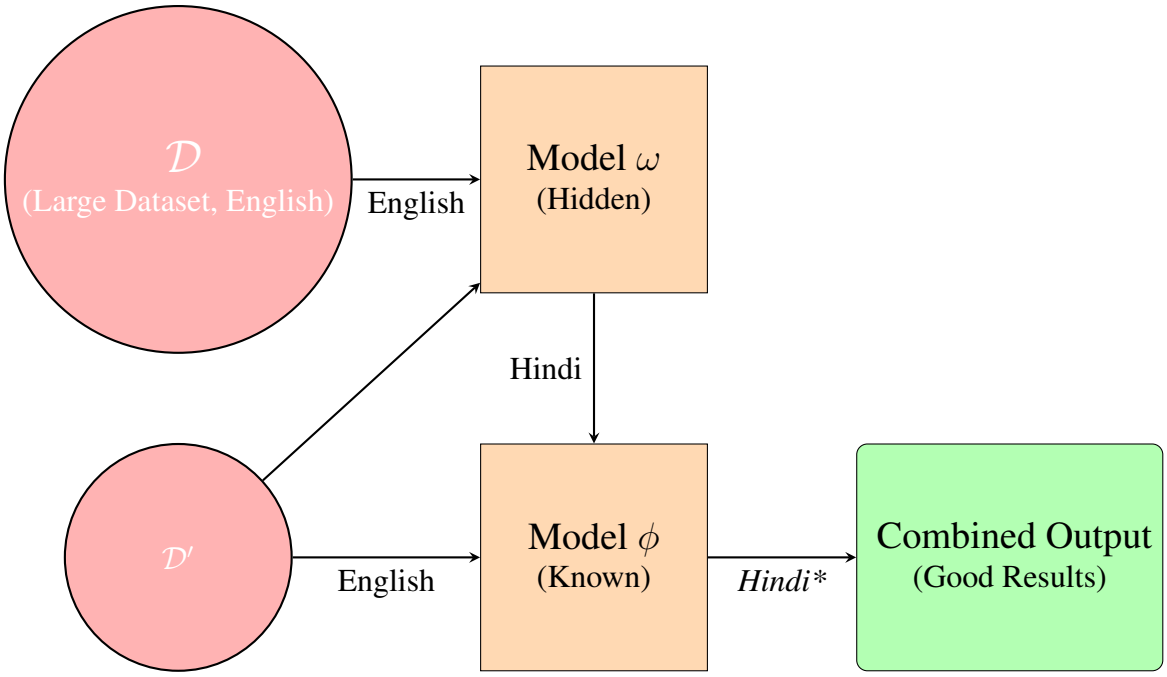
We will work with the simplified example of an online service which performs a certain task, say Google Translate, a service which takes an English sentence and converts it to Hindi.
The task is performed by a model M trained on a large dataset $\mathcal{D}$.



Now, we want to train another model *M'* on a smaller dataset $\mathcal{D}'$ supplementing training using model *M*, which acts as a black box since we do not have access to the internal structure and parameters of the model itself. The structure of *M* is the company's trade secret. We also do not have a dataset as large as Google does!



We wish to utilise the output of the model M during the training of M' to improve the results we would have got only using M'.
Our goal is to get a 'Gold Standard' output, which we call *Hindi. Note that this is different from the translations which just Google's model gives, which is why we were employing our model in the first place. Ideal output matching *Hindi** cannot be obtained with just our data and model alone as well though.**



$$\theta = [\phi, \omega]$$

but Learnable parameters are just ϕ

The issue is that, we cannot backpropagate through M since the gradient of the loss with respect to the parameters of M is not known, i.e. even though ϕ are known, ϕ and thus θ aren't. Are main problem is the lack of knowledge about the functional form of ω due to which we cannot compute gradients with respect to them. So what is the solution?

4.0.1 Approach 1: Perturbation

A proposed solution was that we can perturb the inputs to M a little, thus effectively estimating the loss function around the inputs allowing us to backpropagate without knowing the gradient. This is like numerical estimation of the derivative. However, this is **very costly** since we need a huge number of queries to get smooth derivatives, numerical derivative computation is computationally expensive and thus **not a feasible solution**.

4.0.2 Approach 2: Gaussian Processes

The idea is to model M using a Gaussian process (any Bayesian model can suffice), i.e. replace the entire black box by a Gaussian Process. We use this to obtain correct results and reduce uncertainty. The procedure is roughly as follows

- Select an example sentence s through $\mathcal{D}$, pass it through M and get loss

- Make Gaussian process and train *M'* to fine tune for sentences *s'* close to original example sentence s

Using Gaussian Processes saves us the entire hassle of computing derivatives, since no derivatives are involved in Gaussian Process, thus solving our problem with backpropagation.

Note that for a large amount of data, the **interpolative nature** of Gaussian Process almost approximates the black box we are trying to model.

4.1 Prompt Engineering

This is a very interesting and upcoming area of NLP. Usually a model is designed for one task. But, we can make models which can generalise to many tasks. Say, a model is trained on some set of tasks (using 1 million training samples) but tested on other task, not in the set of tasks in the training set. It is possible to provide some examples (10^1) and retrain to make it 'adjust' to the new task.

As an example, suppose we are designing a complex model which processes a set of questions which are of a similar type:

1. Who is the Chancellor of Germany? *Ans: Olaf Scholz*
2. When did Armstrong land on the moon? *Ans: 1969*
3. When was Obama born? *Ans: 1961*

We have a 'training' data set of 1 million questions and label answers. While testing, now we give our model some 10 test examples which are nothing like what we've seen before.
As an example, we prompt the model, "What is the sentiment of the people towards an X political party?" Although our model will not be able to answer it accurately from the training it has received, when we retrain it with **just these 10 new 'test examples' (since they are not from an actual test set in the conventional set)**, the model is seen to perform surprisingly well on unseen examples.

Have you ever heard about the godly transformer model **GPT-3**?
It is a NL model that can do absolutely any task given to it in writing. There's a huge variety of tasks that it can perform. Keeping digression aside, here the input that we provided to the model "*Find the derivative of log(x)*" is called the **prompt** and then processes it.
In prompt engineering, the description of the task is embedded in the input, e.g., as a question instead of it being implicitly given. Prompt engineering typically works by converting one or more tasks to a prompt-based dataset and training a language model with what has been called "prompt-based learning" or just "prompt learning". Prompt engineering may work from a large "frozen" pretrained language model and where only the representation of the prompt is learned, with what has been called "prefix-tuning" or "prompt tuning".
Another way is, instead of a single sentence as a prompt, give few examples of **question-answer**

[1]Not exact numbers. Just an order of magnitude

pairs followed by the main question as your prompt. Then, the model interpretes the examples and owing to its huge recalling capability, is able to correctly answer your question. This is called **Few Shot Learning**.

Thus, there exists a certain robustness in the model by which when we train on a very small 'test set', we will be able to **update our weights** and still get very good answers for unseen questions. This is an interesting topic of current research known as **Prompt Engineering**.

5 Questions to ponder

- Why the configuration/permutation of the sequence of mini-batches is not a hyperparameter but the number of bathces (in the sequence) is?
- Show that Gradient Descent with L2 regularization is equivalent to Mini-batch Gradient Descent without L2 regularization with some appropriate batch size.
- If you have a GPU with 24GB RAM and you will train a model for 500 iterations at max, then what should be the batch size you would use?

6 Conclusion

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. It is a helpful tool in the machine learning toolkit which also works moderately well in practice. There exist many gradient descent algorithms[4] which further optimise gradient descent. We use gradient descent to optimise models in machine learning services and prompt engineering as well

References

[1] *CS231n: Deep Learning for Computer Vision*. Stanford University.

[2] B. Fehrman, B. Gess, and A. Jentzen. Convergence rates for the stochastic gradient descent method for non-convex objective functions. 2019.

[3] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.

[4] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs, 2016.