Regularization

Zhihan Hu

March 2025

1 Introduction

In this document, I will talk about what is regularization in machine learning. Before we begin to dive into this topic, I would like to introduce some background that may help to better understand this idea: they are **bias-and-variance**, **Likelihood** and **Bayesian inference**

2 Bias and Variance

There are two terms that widely used to describe the performance of an ML model, **bias** and **Variance**. When we say a model has high bias, we mean the our model does not perform well on training dataset. When we say a model has high variance, we mean that our model does not perform well on testing dataset while having a good performance on the training dataset. We usually use two other terms to interchangeably represent high bias and high variance, they are

- Underfitting $< - >$ High bias
- Overfitting $< - >$ High variance

Personally speaking, I prefer Underfitting and Overfitting more since it is more semantically understandable, as Underfitting means the model does not capture enough relations within the features such that it performance, even on the training dataset, is low. Similarly, Overfitting means that the model pays too much attention on the training dataset, overestimating the underlying variance on the given data, and thus leading to a bad performance on the testing set. There are several ways to fix these two issues, and **regularization, as you may guess, is to solve the Overfitting issue**

3 Likelihood

Given a data set D and a learnable parameter set θ, the likelihood is defined as the probability of D to occur given θ, i.e, $P(D|\theta)$. Let us see an example on how likelihood is applied in ML. Suppose we have a deep learning model that

classifies whether a picture is showing a cat or a dog. We have a training dataset with 5 images, whose labels are Y = [dog, dog, cat, dog, cat]. Then, for the first image, our model will predict it to be a dog with a probability $P(dog|X_1, \theta)$, for the second image, the probability will be $P(dog|X_2, \theta)$, etc. Since we can assume that each image is independent from each other, the probability that our model will have a prediction of [dog, dog, cat, dog, cat] is

$$P(Y|\theta, X) = \prod_{i=1}^{5} P(Y_i|\theta, X_i) \tag{1}$$

Which, is the likelihood of D given some parameters θ.

Then, we hope that our model is as accurate as possible. Therefore, we want the likelihood as large as possible, i.e, the model has a larger probability to predict the correct output [dog, dog, cat, dog, cat]. And here comes the **Maximum likelihood**. i.e, for a dataset D, we want to find

$$max_\theta P(D|\theta) \tag{2}$$

Usually, we will use log to make the calculation simpler, and we say it as **Maximum log-likelihood**. Based on our example, we have

$logP(Y|\theta, X) = log \prod_{i=1}^{5} P(Y_i|\theta, X_i)$

$= \sum_{i=1}^{n} logP(Y_i|\theta, X_i)$

If we set label cat = 0 and dog = 1, we can rewrite the above equation as

$$\sum_{i=1}^{n} Y_i logP(Y_i|\theta, X_i) + (1 - Y_i) log(1 - P(Y_i|\theta, X_i)) \tag{3}$$

If we are going to maximize equation 3, we are actually minimize negative of equation 3, i,e

$$-logP(Y|\theta, X) = -\sum_{i=1}^{n} Y_i logP(Y_i|\theta, X_i) + (1 - Y_i) log(1 - P(Y_i|\theta, X_i)) \tag{4}$$

Let me use a notation from neural network, where $\hat{Y}_i$ is the probability of the model predicting image i as a dog, i.e, $\hat{Y}_i = P(Y_i = dog|\theta, X_i)$, then, the above equation can we written as

$$-logP(Y|\theta, X) = -\sum_{i=1}^{n} Y_i log\hat{Y}_i + (1 - Y_i) log(1 - \hat{Y}_i) \tag{5}$$

Do you find this equation familiar? Yes, it is the cross-entropy loss for a binary classifier. Here is the key takeaway: **Minimizing loss function is the same as maximizing log likelihood**. Maximizing log-likelihood can be considered as one of the most important mathematical foundations in ML, and there are way more details that I didn't cover, feel free to search it.

4 Bayesian inference

The Bayes law states that:

$$P(x|y) = \frac{P(y|x)p(x)}{p(y)} \tag{6}$$

If we have some prior knowledge on the learnable parameters distribution, we can then have:

$$P(\theta|D) \propto P(D|\theta)P(\theta) \tag{7}$$

Then, if we maximize equation 7, we get the largest potential of how θ will be. This will be really important to derive the equation for L2 regularization

5 L2 regularization

Overfitting means the model depends too much on the training dataset, and thus overestimating the underlying randomness in the data. To avoid this, one solution is to assume θ has some prior distribution, so to apply a limitation during the learning process. L2 regularization is derived by assuming θ is initially having a multi-variate Gaussian distribution with mean 0, and an identity covariance matrix, meaning each parameter is independent with each other. The following equation shows such distribution:

$$P(x) = \frac{1}{(2\pi)^{\frac{d}{2}}|\sum|^{\frac{1}{2}}} exp(-\frac{1}{2}(x-\mu)^T \sum (x-\mu)) \tag{8}$$

Then, we can start to use log to maximize the Bayesian.

$logP(\theta|D) \propto logP(D|\theta)P(\theta)$

$= logP(D|\theta) + logP(\theta)$ The first term is the likelihood

$= logP(D|\theta) - \frac{1}{2}(\theta-\mu)^T \sum(\theta-\mu)$ assume $\mu = 0$ and parameters are independent

$= logP(D|\theta) - \frac{1}{2}\theta^T\theta$

To maximize the above equation, we are actually minimize the negative of it, i.e

$$-logP(D|\theta) + \frac{1}{2}\theta^T\theta \tag{9}$$

As always, when doing the actual optimization, we will divide equation 9 by total number of samples. Additionally, we will use a regularization coefficient λ as a tunable hyperparameter.

$$J(\theta) = -\frac{1}{m}logP(D|\theta) + \frac{\lambda}{2m}\theta^T\theta \tag{10}$$

For deep learning models, the regularization term need a small change, as each weight matrices contribute independently to the loss, we have

$$J(\theta) = -\frac{1}{m} logP(D|\theta) + \frac{\lambda}{2m} \sum_{l=1}^{L} (W^{(l)})^T W^{(l)} \tag{11}$$

Now we have mathematically seen how regularization term is derived, let's see why this term can help to reduce overfitting. Remember that, during gradient descent, the gradient w.r.t $W^{(l)}$ is computed from backpropagation, let's denote it as back($W^{(l)}$). When considering about regularization term, the gradient becomes

$$\frac{\delta J}{\delta W^{(l)}} = \text{back}(W^{(l)}) + \frac{\lambda}{m} W^{(l)} \tag{12}$$

Then, the new weight matrix after one iteration will be

$$W^{(l)} = W^{(l)} - \alpha(\text{back}(W^{(l)}) + \frac{\lambda}{m} W^{(l)}) = (1 - \frac{\alpha\lambda}{m}) W^{(l)} - \alpha \text{back}(W^{(l)}) \tag{13}$$

We can see that compared to normal gradient descent, the regularized version has a decay term $1 - \frac{\alpha\lambda}{2m}$ on the $W^{(l)}$, making it smaller. A smaller $W^{(l)}$ will cause Z to stay on the linear part the activation curve, making the model to be less complex.

6 Drop outs

One another widely used regularization technique in deep learning is called dropouts. The basic idea is to randomly drop out some of the neurons on a layer. For example, let's suppose we have m_l neurons on layer l and m_{l+1} neurons on layer l + 1. And we have a data set of n samples. When propagating from l to l+1, if no dropouts occurred on layer l+1, we will get an output Z $\in R^{m_{l+1} \times n}$ and then A with same dimension. If drop out occurs, A will still be in the same dimension, but with some of the neurons set to be 0 for each data point. three important properties that you need to notice are:

- dropout occurs data point wisely, i.e, on one layer, different data point has different dropped out neurons.
- Dropout occurs at output A.
- Dropout on $A^{(l)}$ will affect $W^{(l+1)}$
- Dropout on $A^{(l)}$ will not lead to unexpected drop on layer l+1, i.e, drop out on layer l will not influence the dimension of other layers.

It is complicated to prove that why drop out can reduce overfit, due to its random nature. And if you're interested, you can read through the paper *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. You will see that the idea is mainly from intuition, along with empirical result to show its effectiveness in reducing overfitting. And here I will talk about the intuition:

- Drop out logically makes the architect smaller, which helps to reduce overfitting.
- When you drop out neurons on $A^{(l)}$, you're actually dropping out weights in $W^{(l+1)}$. The randomly dropped out weights cause the neuron on layer l+1 to spread the incoming weights instead of concentrating on some of them, which has the same effect as weight decay (L2 regularization)

In order to avoid the issue of smaller A, which may lead to locate on saturated regimen on activation curve, we usually divide the dropped out A with keep_prob.

Now let's see what effect drop out has on the implementation.

In propagation, thing's are simple, we simply create a mask D^l for each layer, whose dimension is the same as A^l. The element in D^l will be initialized using a uniform distribution U(0, 1). And we set the elements that are smaller than keep_prob to 1, and the rest elements to 0. This will act like mask to drop out neurons for each data point. Then we apply $A^l * D^l$ for each layer, and everything else remain unchanged.

Some of you may think with drop out, backpropagation will be more complicated, since for each data point in one run of backpropagation, the architect of the model is different. And the dynamically changing architect cause the complexity. However, the implementation of backpropagation is still very simple. Remember that, drop out only logically change the architect of the model, and during propagation, we eliminate some neurons' contribution on the loss, and rescale the rest by a constant factor, and thus for each $dA^{(l)}$, we can also do the same, masking out the contribution of the dropped neuron, and rescale the rest by one over keep_prob, i.e, we can do the backpropagation as if no drop out has occurred, then apply mask, and rescale. i.e

$$dA^{(l)} = (W^{(l+1)})^T dZ^{(l+1)} \tag{14}$$

$$dA^{(l)} = dA^{(l)} \circ D^l / \text{keep\_prob} \tag{15}$$