Optimization schemes

Zhihan Hu

March 2025

1 Introduction

Currently, the size of the data is drastically increased, leading to low efficiency in model training. You may have already read the document that talks about Xavier and He initialization, which refines the training data set so that the training process will be faster. In this document, I will talk about some techniques that refine the training process.

2 Mini Batch Gradient descent

The workflow of mini batch gradient descent is:

- Define a mini batch size d
- Shuffle the training data set and then split into $\frac{m}{d}$ batches, where m is the number of data points.
- Iteratively using each mini batch to do the gradient descent, where the current weight state is the result from last mini batch
- When all the mini batches have been thrown into the model, we call this an epoch. We can do several epochs to further improve model performance.

Now we have two hyperparameters, epoch and batch size.

When I first learned about this technique. I did an analysis of the complexity, which caused me to question that how mini batch improved the training efficiency.

Suppose the training data set has m data points, what will be the computational complexity in one layer of back-propagation? Let's write out the formula and analyze:

$$dZ^{(l)} = (W^{(l+1)})^T dZ^{(l+1)} \circ \sigma'(Z^{(l)}) \tag{1}$$

$$dW^{(l)} = dZ^{(l)}(A^{(l-1)})^T \tag{2}$$

$$db^{(l)} = \textbf{np.sum}(dZ^{(l)}, \textbf{axis=}1) \tag{3}$$

In equation 1, the complexity will be $O(n_{l+1} \times n_l \times m + n_l \times m)$.
In equation 2, the complexity will be $O(n_l \times n_{l-1} \times m)$.
In equation 3, the complexity will be $O(n_l \times m)$

When m is really large, the complexity of the training process will be linear to m. In one epoch, we will finally run through all m data points, then why does mini batch can improve the efficiency?

The reason lies in the reduction overhead of memory access, since the size of a mini batch is much smaller than the entire dataset. From the perspective of hardware, a better cache efficiency and a reduction in memory switching. Besides this, mini batch gradient descent enables a faster convergence due to a higher frequency of gradient update. Furthermore, mini batch implicitly prevents overfitting.

3 Adam gradient descent

In a non-trivial machine learning model, e.g, deep learning model, the shape of the loss function is complex, which includes a lot of local minima, saddle points and flat region. If we use simple gradient descent without any modification of the step, the optimization may have these following behaviors:

- Fluctuation. The gradient computed may only contain information of local minima, which will have a different direction pointing to the global minimum. Instead of going directly to the global minima, the optimization process will fluctuate along the direction of different local minima, which slows the optimization process.
- Trapped in local minima. There is also an potential that the optimization process will trapped in local minima, meaning that it fluctuates around the local minima instead of stepping out.
- Trapped in saddle point. Unlike local minima, if the optimization is trapped in saddle point, there will be no chance for the process to step out. Since on saddle point, the gradients are 0, and there will be no update if a simple gradient descent scheme is used.
- Slows down in flat region. Although flat region will not stop the training the process, the little gradient update in this region will make the process really inefficient.

Thus, we need an optimization scheme to avoid these issues. And this requirement brings the topic of this section: Adam Gradient Descent.

Adam gradient descent is not named after someone whose name is Adam, instead, Adam stands for Adaptive moment estimation. The intuition behind this scheme is that the gradient of this iteration is averaged by previous gradients, such that

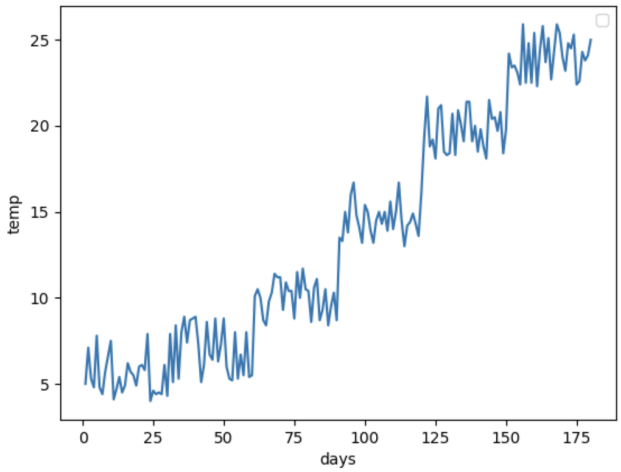
- When approaching local minimum, the gradient update will not go directly to the minimum since the direction depends on the gradient computed previously. In such manner, the moving average of previous gradient smooths out sudden changes, which reduced fluctuation
- For the same reason, when in saddle point and flat region, the averaged gradient will not be zero even if the true gradient is 0, and thus trapping and slow down will be reduced.

In the following content, I will provide a detailed explanation on Adam gradient descent.

3.1 Exponentially weighted averages

The scheme that we applied for averaging out gradients is called Exponentially weighted averages. Let's see through an example on how it works.

Suppose we have a dataset, which records the temperature at city Ningbo(Where I was born) in recent 180 days. We have the following plot to show the temperature change Now, we want to average the temperature on day n with the



temperatures of the previous days, what we do is shown below:

$$v_0 = 0 \tag{4}$$

$$v_1 = \beta v_0 + (1 - \beta)t_1 \tag{5}$$

$$v_2 = \beta v_1 + (1-\beta)t_2 \tag{6}$$

For example, suppose $\beta = 0.8$, then, To see why it is exponential weight average,

Days	**temperature**	**v**
		0
1	20	4
2	23	7.8
3	17	9.64

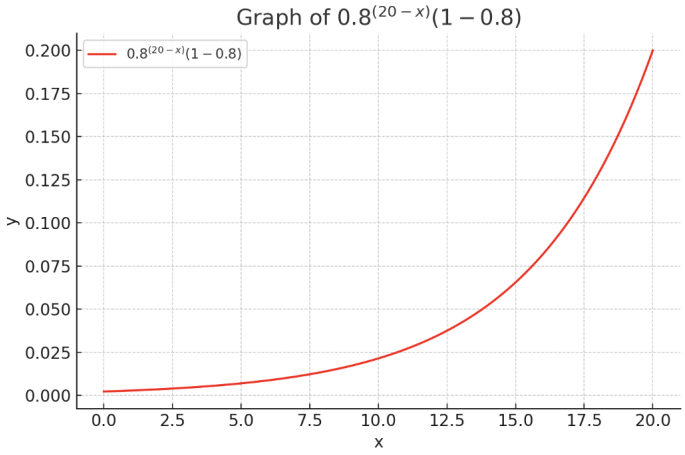
we can write out v_n by

$$v_n = \sum_{i=1}^{n} \beta^{n-i}(1-\beta)t_i \tag{7}$$

Thus, v_n is a weighted sum of temperature from day 1 to day n. The weight for t_i is

$$\alpha_i = \beta^{n-i}(1-\beta) \tag{8}$$

Suppose n is 20 and $\beta = 0.8$ we can have the weight for each t_i As you can see,

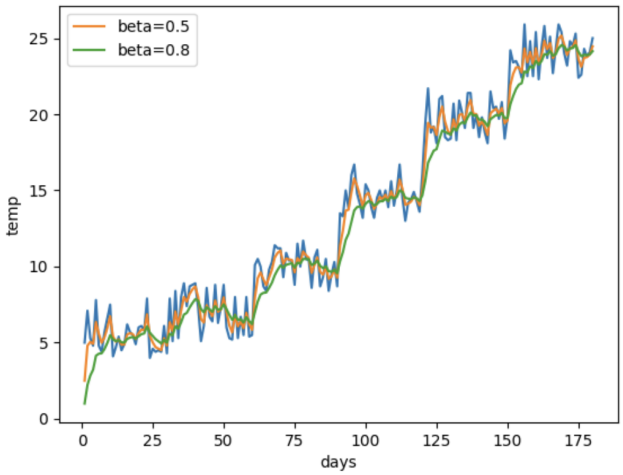


an exponential curve here for the weights.

The following graph shows the effect of different β values. We can see that when β is 0.8, the curve will be smoother than when β is 0.5. Intuitively, a smoother curve means that the current v depends on more previous ts such that even there is a sudden increase or decrease in temperature, the sharpness will be averaged out to become more smooth.

One important thing to know, as you may take it as for granted, is that we can assume we are averaging out the past

$$\frac{1}{1-\beta} \tag{9}$$



days of temperature for today's temperature.

You may noticed that, the beginning part of the weighted average curves for both β is really small. And it is simply because v_0 is 0 and the β is usually larger than 0.5, causing initial vs to be small. One way to solve this issue is called bias correction, by applying

$$v_t = \frac{v_t}{1 - \beta^t} \tag{10}$$

However, this correction may cause numerical instability, as $1 - \beta^t$ might be too small such that v_t at early stage becomes too large, which will even make v_t at later period to explode. For example, if β is 0.999, then v_t might explode. And in later stage, v_t depends mainly on v_{t-1}, and thus v_t on later stage will explode too even though $1 - \beta^t$ is close to 1. One other reason that many people don't use bias correction is that when in optimization process, one can regard days as iterations and temp as gradient computed at each iteration. It is less likely that we will have computed optimal gradient at early iterations. And thus we don't have to worry about the small value at early iteration since it will become better as the iterations keep going.

3.2 GD with averaged gradients

As mentioned above, we can regard days as iterations, and temp as gradient. And the gradient descent will become:

Listing 1: Gradient descent with weighted average

```
import numpy as np
```

```
# Initialize v_dw, v_db
v_dw = np.zeros(w.shape)
v_db = np.zeros(b.shape)
for i in runs:
    dw, db = backprop()
    v_dw = beta * v_dw + (1 - beta) * dw
    v_db = beta * v_db + (1 - beta) * db
    w = w - alpha * v_dw
    b = b - alpha * v_db
```

3.3 RMSprop

RMSprop is another scheme to solve the issues mentioned above. Although there are some mathematical relations in between RMSprop and exponential weighted average, their logic are different.

- exponential weighted average takes into account of the previous gradients, such that the optimization process will be smoother.
- RMSprop records the variance of the gradients from previous iterations, and regards variance as an indicator of the landscape the current process locates, and then dynamically change the learning rate based on it.

So how do we calculate the variance of the gradient? We know that our aim is to make the gradient of each parameter to be 0, and thus the variance will be W^2. Similar to weighted average gradient descent, we want stability of the variance to avoid cases like sudden increase or decrease in the gradient variance, and thus for current gradient variance, we applied exponential weighted average to smooth out the variance, i.e

$$s_{dw_n} = \beta s_{dw_{n-1}} + (1 - \beta) dw_n \tag{11}$$

Intuitively, if the weighted average variance of gradient is large, it indicates that for many iterations, the optimization process might be fluctuating back and forth around a local minima, or stepping among local minima and fluctuating slowly towards the global minimum. So we want to lower the learning rate such that the optimization process will be more stable. If the weighted average variance of gradient is small, it indicates for many iterations, the optimization process stuck into a flatten region such that there is not much gradient update. In such case we want the learning rate to be large to step out from these regions. In conclusion:

- Large variance - lower the learning rate
- Small variance - increase the learning rate

In RMSprop, it achieves this by doing the following:

$$W = W - \frac{\alpha}{\sqrt{s_{dw} + \epsilon}} dW \tag{12}$$

Where α is the original learning rate, and ϵ is usually 1e-8 to avoid divide by 0 error.

The implementation of RMSprop is shown as follows

Listing 2: RMSprop

```
import numpy as np

# Initialize s_dw, s_db
s_dw = np.zeros(w.shape)
s_db = np.zeros(b.shape)
for i in runs:
    dw, db = backprop()
    s_dw = beta * s_dw + (1 - beta) * (dw ** 2)
    s_db = beta * s_db + (1 - beta) * (db ** 2)
    w = w - (alpha / np.sqrt(s_dw+epsilon)) * dw
    b = b - (alpha / np.sqrt(s_db+epsilon)) * db
```

3.4 Adam gradient descent

Adam gradient descent integrates the benefits brought by weighted average GD and RMS prop such that, the gradient will be smoothed out and there will be flexibility in learning rate. The implementation will be

Listing 3: RMSprop

```
import numpy as np

# Initialize v_dw, v_db and s_dw, s_db
v_dw = np.zeros(w.shape)
v_db = np.zeros(b.shape)
s_dw = np.zeros(w.shape)
s_db = np.zeros(b.shape)
for i in runs:
    dw, db = backprop()
    v_dw = beta1 * v_dw + (1 - beta1) * dw
    v_db = beta1 * v_db + (1 - beta1) * db
    s_dw = beta2 * s_dw + (1 - beta2) * (dw ** 2)
    s_db = beta2 * s_db + (1 - beta2) * (db ** 2)
    w = w - (alpha / np.sqrt(s_dw+epsilon)) * v_dw
    b = b - (alpha / np.sqrt(s_db+epsilon)) * v_db
```