Understanding Learning Rates of **cycle_policy** and How It Improves Performance in Deep Learning

2019年6月16日 7:46

Understanding the Learning Rates of cycle_policy and How It Improves Performance in Deep Learning

This post is an attempt to document understanding on the following topic:

What is the learning rate? What is it's significance?
How does one systematically arrive at a good learning rate?
Why do we change the learning rate during training?

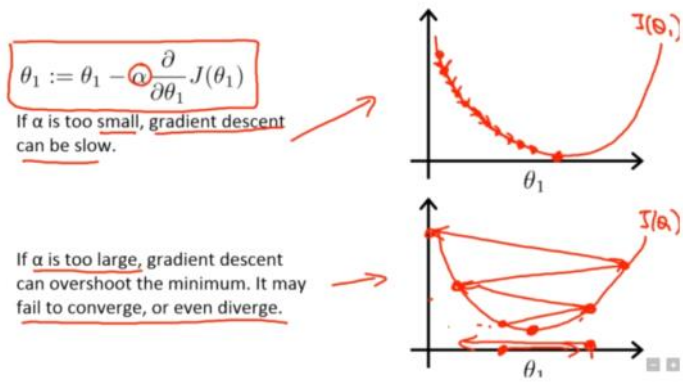
Much of this post are based on the stuff written by past fast.ai fellows [1], [2], [5] and [3] . This is a concise version of it, arranged in a way for one to quickly get to the meat of the material. Do go over the references for more details.

what is the learning rate?

Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient. The lower the value, the slower we travel along the downward slope. While this might be a good idea (using a low learning rate) in terms of making sure that we do not miss any local minima, it could also mean that we'll be taking a long time to converge — especially if we get stuck on a plateau region.

The following formula shows the relationship.

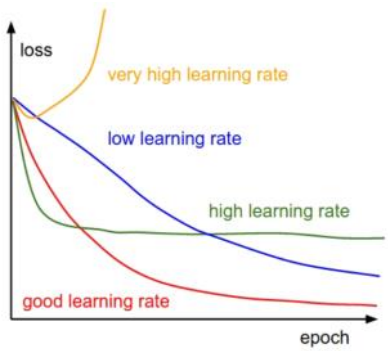
```
new_weight = existing_weight — learning_rate * gradient
```



Gradient descent with small (top) and large (bottom) learning rates. Source: Andrew Ng's Machine Learning course on Coursera

Typically learning rates are configured naively at random by the user. At best, the user would leverage on past experiences (or other types of learning material) to gain the intuition on what is the best value to use in setting learning rates.

As such, it's often hard to get it right. The below diagram demonstrates the different scenarios one can fall into when configuring the learning rate.



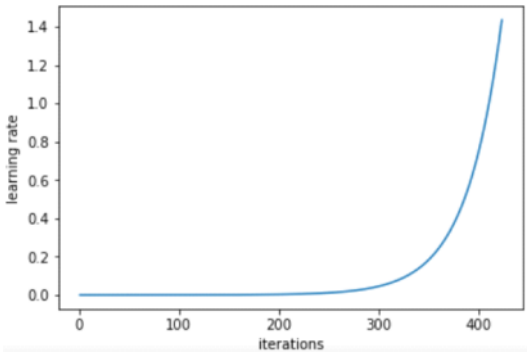
Effect of various learning rates on convergence (Img Credit: cs231n)

Furthermore, the learning rate affects how quickly our model can converge to a local minima (aka arrive at the best accuracy). Thus getting it right from the get go would mean lesser time for us to train the model.

```
Less training time, lesser money spent on GPU cloud compute. :)
```

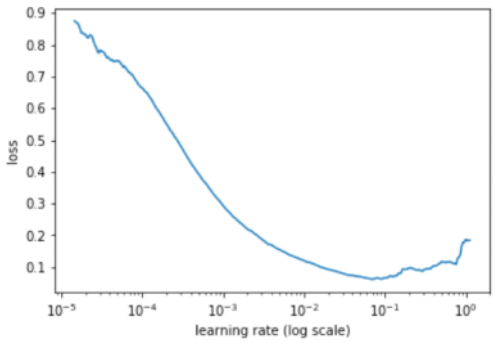
Is there a better way to determine the learning rate?

In Section 3.3 of **“Cyclical Learning Rates for Training Neural Networks.”** [4], Leslie N. Smith argued that you could estimate a good learning rate by training the model initially with a very low learning rate and increasing it (either linearly or exponentially) at each iteration.



Learning rate increases after each mini-batch

If we record the learning at each iteration and plot the learning rate (log) against loss; we will see that as the learning rate increase, there will be a point where the loss stops decreasing and starts to increase. In practice, our learning rate should ideally be somewhere to the left to the lowest point of the graph (as demonstrated in below graph). In this case, 0.001 to 0.01.



The above seems useful. How can I start using it?

At the moment it is supported as a function in the fast.ai package, developed by Jeremy Howard as a way to abstract the pytorch package (much like how Keras is an abstraction for Tensorflow).

One only needs to type in the following command to start finding the most optimal learning rate to use before training a neural network.

```
# learn is an instance of Learner class or one of derived classes like ConvLearner
learn.lr_find()
learn.sched.plot_lr()
```

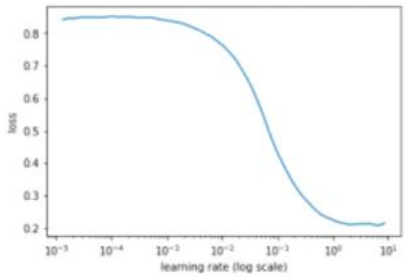
plot_loss.py hosted with ❤ by GitHub view raw

```
In [11]: learn = ConvLearner.pretrained(f_model, data, metrics=metrics)

In [15]: lrf=learn.lr_find()
         learn.sched.plot()
```

```
A Jupyter Widget

[ 0.       0.22404  0.31176  0.82044]
```



0.1 seems like a good learning rate

cycle_policy

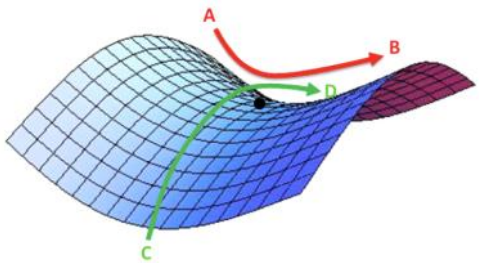
At this juncture we've covered what learning rate is all about, it's importance, and how can we systematically come to an optimal value to use when we start training our model.

Next we would go through how learning rates can still be used to improve our model's performance.

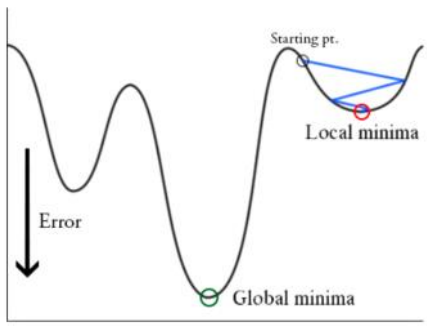
The conventional wisdom

Typically when one sets their learning rate and trains the model, one would only wait for the learning rate to decrease over time and for the model to eventually converge.

However, as the gradient reaches a plateau, the training loss becomes harder to improve. In [3], *Dauphin et al argue that the difficulty in minimizing the loss arises from saddle points rather than poor local minima.*



A saddle point in the error surface. A saddle point is a point where derivatives of the function become zero but the point is not a local extremum on all axes. (Img Credit: safaribooksonline)



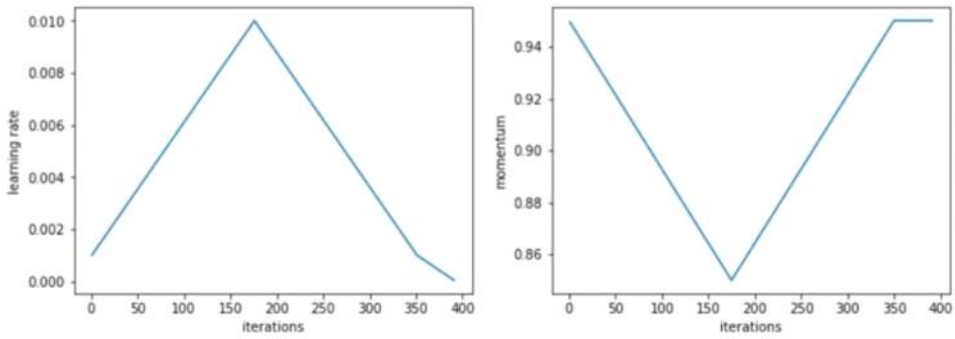
SGDR: Stochastic Grandient Descent with warm Restarted

So how do we escape from this?

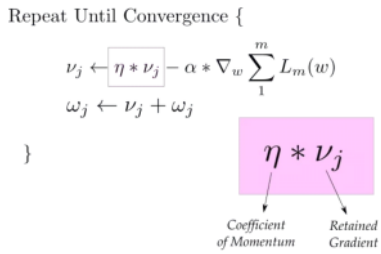
There are a few options that we could consider. In general, taking a quote from [1],

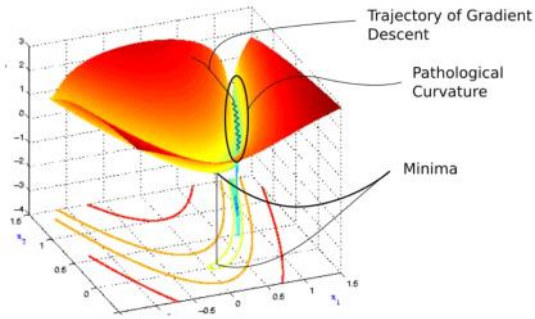
… instead of using a fixed value for learning rate and decreasing it over time, if the training doesn't improve our loss anymore, we're going to be changing the learning rate every iteration according to some cyclic function f. *Each cycle has a fixed length in terms of number of iterations. This method lets the learning rate cyclically vary between reasonable boundary values.* ***It helps because, if we get stuck on saddle points, increasing the learning rate allows more rapid traversal of saddle point plateaus.***

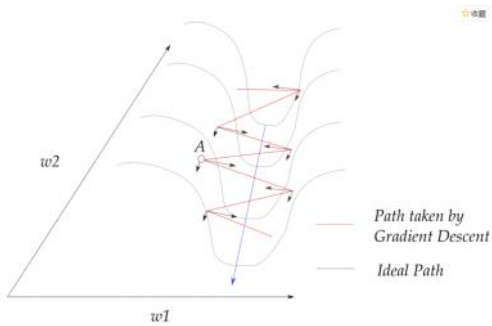
Fit_one_cycle:



Why do learning rates and momentum need to be adjusted at the same time like this?



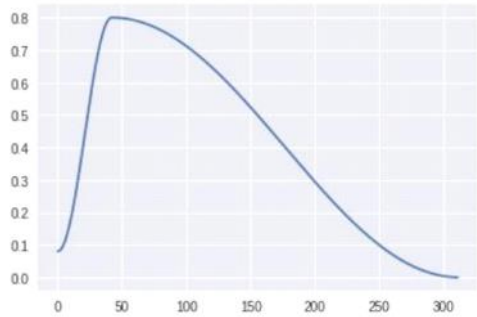


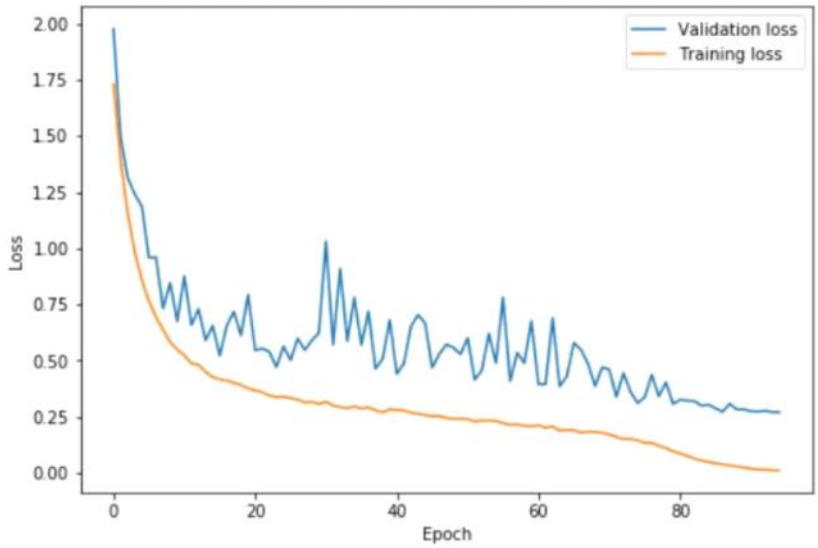


Experimen 1

[ ]: cyc_len = 95

[ ] :learn.fit_one_cycle(cyc_len=cyc_len, max_lr=0.08,div_factor=10, pct_start=0.1368,moms=(0.95, 0.85), wd=1e-4)





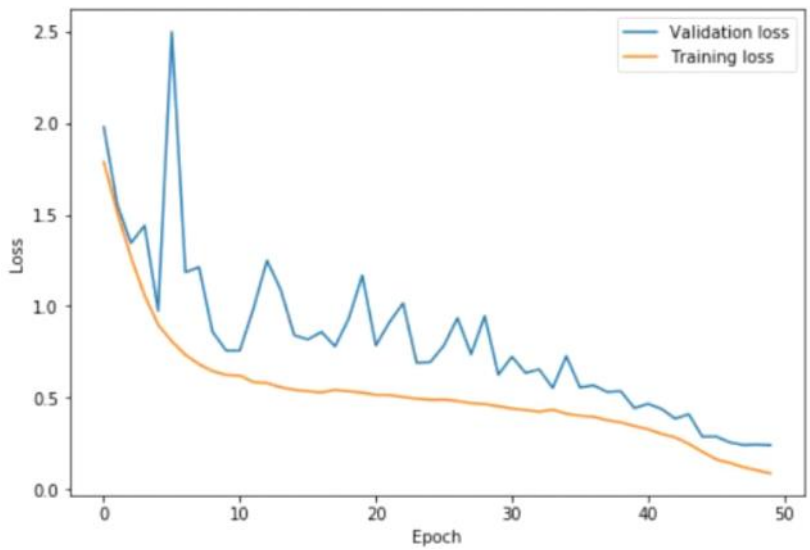
As shown in the figure, the learning rate rises from 0. 08 to 0. 8, and finally drops to a very low learning rate. The loss of the verification set becomes unstable in the middle, but it decreases in the end, and the loss difference between the verification set and the training set is not significant.

Experimen2:Higher learning rate / smaller cycle

[ ] # adjust cyc_len and div_factor, pct_start

[ ] learn.fit_one_cycle(cyc_len=50, max_lr=0.8,div_factor=10, pct_start=0.5,moms=(0.95, 0.85), wd=1e-4)

At the same time, due to the attenuation of one-cycle learning rate, we are allowed to use a higher learning rate.
But to ensure that the loss is not too big, a longer cycle may be needed for training



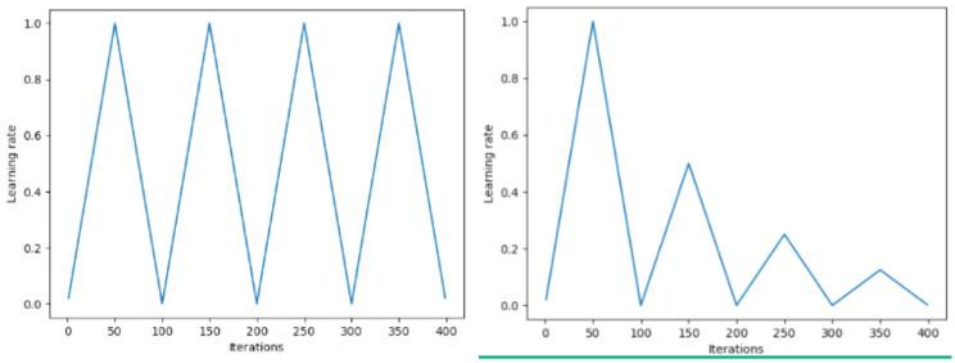
Experiment 3： invariant momentum

learn.fit_one_cycle(cyc_len=cyc_len, max_lr=0.08,div_factor=10, pct_start=0.1368,moms=(0.9, 0.9), wd=1e-4)

In Leslie's experiment, reducing momentum leads to better results.
In training, the momentum of 0.85 to 0.95 is the better value in experience.
In the experiment, it is found that if the momentum uses a constant, the result will be close to the accuracy of the changing momentum, but the loss will be larger. In time, the constant momentum will be faster.

Cyclical Learning Rates:

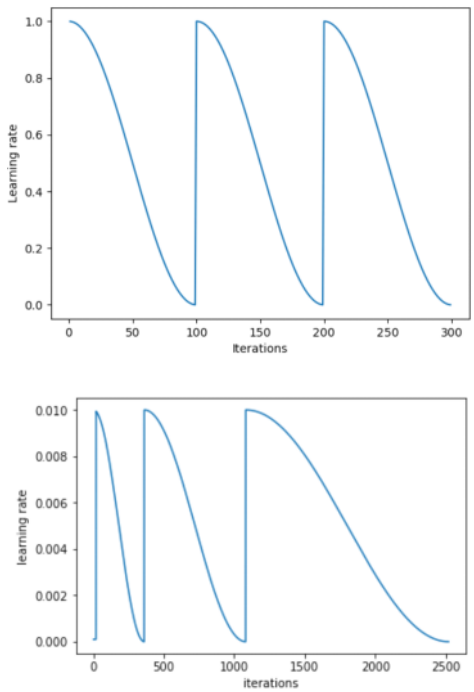
In [2], Leslie proposes a 'triangular' method where the learning rates are restarted after every few iteration



'Triangular' and 'Triangular2' methods for cycling learning rate proposed by Leslie N. Smith. On the left plot min and max lr are kept the same. On the right the difference is cut in half after each cycle.

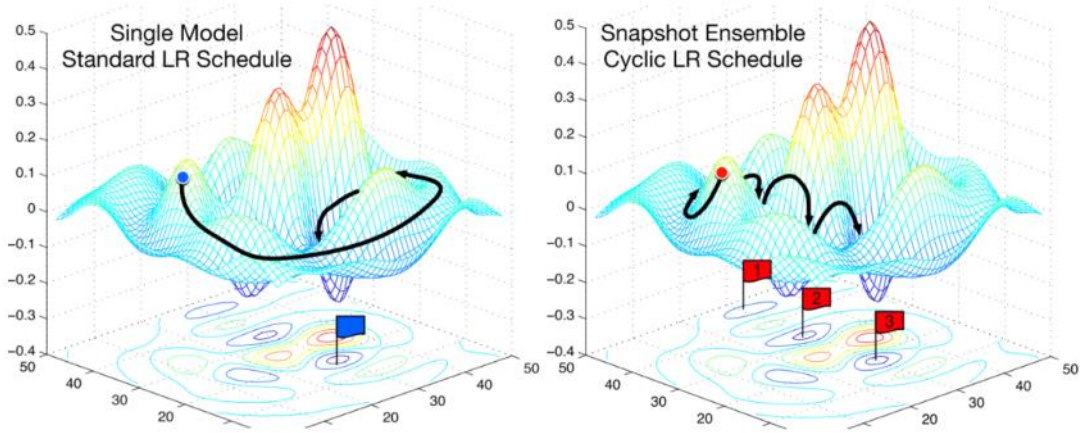
Another method that is also popular is called *Stochastic Gradient Descent with Warm Restarts, proposed by Loshchilov & Hutter* [6]. This method basically uses the cosine function as the cyclic function and restarts the learning rate at the maximum at each cycle. The "warm" bit comes from the fact that when the learning rate is restarted, it does not start from scratch; but rather from the parameters to which the model converged during the last step [7].

While there are variations of this, the below diagram demonstrates one of its implementation, where each cycle is set to the same time period.



SGDR plot, learning rate vs iteration.

Thus we now have a way to reduce the training time, by basically periodically jumping around *"mountains" (below).*



Comparing fixed LR and Cyclic LR (img credit: https://arxiv.org/abs/1704.00109)

Aside from saving time, research also shows that using these method tend to improve classification accuracy without tuning and within fewer iteration.

References:

[1] Improving the way we work with learning rate.

[2] The Cyclical Learning Rate technique.

[3] Transfer Learning using differential learning rates.

[4] Leslie N. Smith. Cyclical Learning Rates for Training Neural Networks.

[5] Estimating an Optimal Learning Rate for a Deep Neural Network

[6] Stochastic Gradient Descent with Warm Restarts

[7] Optimization for Deep Learning Highlights in 2017

[8] Lesson 1 Notebook, fast.ai Part 1 V2

[9] https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10

[10] https://hacpai.com/article/1552929502787

[11]https://blog.csdn.net/u013166817/article/details/86503899

深度学习 英文版： http://www.deeplearningbook.org/

8.2 Challenges in Neural Network Optimization

8.2.1 ill-condition

We can make a second-order Taylor series approximation to the function f (x) around the current point x(0) :

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{g} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^{(0)}), \tag{4.8}$$

where $\boldsymbol{g}$ is the gradient and $\boldsymbol{H}$ is the Hessian at $\boldsymbol{x}^{(0)}$. If we use a learning rate of ϵ, then the new point $\boldsymbol{x}$ will be given by $\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}$. Substituting this into our approximation, we obtain

$$f(\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \epsilon \boldsymbol{g}^\top \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}. \tag{4.9}$$

$$\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g} - \epsilon \boldsymbol{g}^\top \boldsymbol{g} \tag{8.10}$$

条件指数（Conditioning）：条件指数市值函数相对于输入的微小变化而发生变化的快慢程度

病态矩阵（Poorly conditioned matrices）：当输入被轻微的变化而导致输出结果发生很大变化的一个矩阵，称为病态矩阵。下面举了一个栗子：

现在有线性系统：Ax = b，解方程

$$\begin{bmatrix} 400 & -201 \\ -800 & 401 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 200 \\ -200 \end{bmatrix}$$

很容易得到解为：**x1 = -100, x2 = -200**. 如果在样本采集时存在一个微小的误差，比如，将 A 矩阵的系数 400 改变成 401：

$$\begin{bmatrix} 401 & -201 \\ -800 & 401 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 200 \\ -200 \end{bmatrix}$$

则得到一个截然不同的解：**x1 = 40000, x2 = 79800**.

https://www.jianshu.com/p/4095f1ee0819

8.2.2 Local Minima

8.2.3 Plateaus,Saddle points and Other Flat Regions

8.2.4 Cliffs and Exploding Gradients

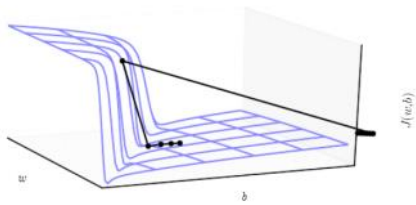


Figure 8.3: The objective function for highly nonlinear deep neural networks or for recurrent neural networks often contains sharp nonlinearities in parameter space resulting from the multiplication of several parameters. These nonlinearities give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly losing most of the optimization work that has been done. Figure adapted with permission from Pascanu *et al.* (2013).

8.2.5 Long-Term Dependencies

For example, suppose that a computational graph contains a path that consists of repeatedly multiplying by a matrix $\boldsymbol{W}$. After t steps, this is equivalent to multiplying by $\boldsymbol{W}^t$. Suppose that $\boldsymbol{W}$ has an eigendecomposition $\boldsymbol{W} = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}$. In this simple case, it is straightforward to see that

$$\boldsymbol{W}^t = \left(\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}\right)^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}. \tag{8.11}$$

8.2.6 Inexact Gradients

Most optimization algorithms are designed with the assumption that we have access to the exact gradient or Hessian matrix. In practice, we usually have only a noisy or even biased estimate of these quantities. Nearly every deep learning algorithm relies on sampling-based estimates, at least insofar as using a minibatch of training examples to compute the gradient.

8.2.7 Poor Correspondence between Local and Global Structure

未来的研究需要进一步探索影响学习轨迹长度和更好地表征训练过程的结果。
Future research will need to develop further understanding of the factors that influence the length of the learning trajectory and better characterize the outcome of the process.

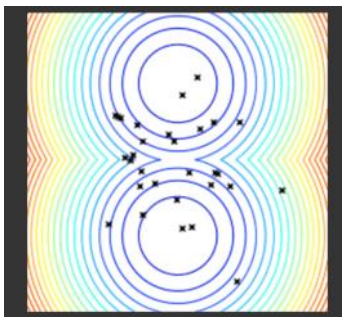
梯度下降和基本上所有的可以有效训练神经网络的学习算法，都是基于局部梯度或者Hessian矩阵信息更新。局部移动可能太过贪心， 朝着下坡方向移动，却和所有可行解南辕北辙。

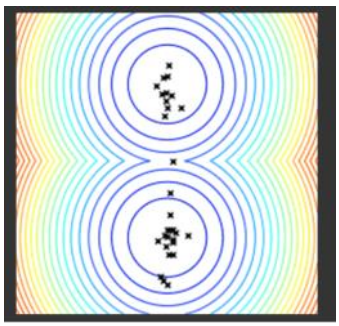
许多现有研究方法在求解具有困难全局结构的问题时，旨在寻求良好的初始点， 而不是开发非局部范围更新的算法。 最终的观点还是建议在传统优化算法上研究怎样选择更佳的初始化点，以此来实现 目标更切实可行。
Many existing research directions are aimed at finding good initial points for problems that have difficult global structure, rather than at developing algorithms that use nonlocal moves.

Facebook 创建了一个名叫 Nevergrad https://github.com/facebookresearch/nevergrad

在选择最佳初始点的尝试上，粒子群优化是一个很好的例子。
Particle swarm optimization is a good example in the attempt to select the best initial point.

粒子群优化算法（Particle swarm optimization）





Thank you for listening!