**ormNotes of Study on Dec 21-22, 2022 (Wed - Thu)**

Based on lecture notes of Stanford

Module 1: Neural Networks

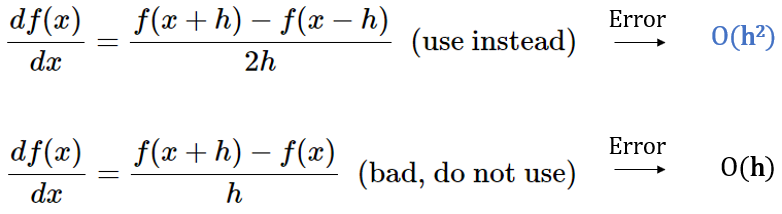
[Neural Networks Part 3: Learning and Evaluation](https://cs231n.github.io/neural-networks-3/)

gradient checks, sanity checks, babysitting the learning process, momentum (+Nesterov), second-order methods, Adagrad/RMSprop, hyperparameter optimization, model ensembles.

This section mainly discusses setting learning rates and finding good parameters.

**Gradient checks**

1. **Use the centered formula**



Where **h** is a very small number, in practice ~ **1e-5**. Note **h** can also be 1e-4 or 1e-6. It depends on feedbacks of the gradient check.

1. **Use relative error to compare numerical gradient and analytical gradient.**

Within a single layer, if the relative error is

< 1e-7: great

< 1e-4 and > 1e-7: normal

< 1e-2 and > 1e-4: bad

> 1e-2: too bad

However, for a deep learning network of 10 layers, relative error > 1e-2 is okay when the gradient checking the input data.

1. **Use double precision floating points instead of single precision ones.**
2. **Be careful of kinks.**

Kinks: **non-differentiable parts** of an objective function, introduced by functions such as ReLU (max(0,x)), or the SVM loss, Maxout neurons, etc.

Consider gradient checking the ReLU function at x=−1e6. Since x<0, the analytic gradient at this point is exactly zero. However, **the numerical gradient would suddenly compute a non-zero gradient because f(x+h) might cross over the kink** (e.g. if h>1e−6) and introduce a non-zero contribution. In this case, the gradient is not exact.

Solution: Use only a few datapoints. Fewer kinks with fewer datapoints.

1. **Gradient check after the first iteration of learning**, instead of doing it immediately.
2. **Turn off Regularization until data loss check is completed.**
3. **Turn off dropout/augmentations when gradient checking.**

Otherwise, huge errors can be introduced. Remember during dropout some neurons are completely silent with no parameter updates?

1. **Check only a few dimensions of parameters**: you know it’s not possible to check them all if there are millions of them.
2. Check the ratio of weights updates, which is defended as:

**A ratio of change around 1e-3 is appropriate.**

1. First-order methods: **SGD, standard momentum and Nesterov momentum**
   1. SGD (stochastic gradient descent)  
      テキスト

      自動的に生成された説明

If the loss function has a local minimum or a saddle point, the gradient becomes zero (gradient descent get stuck), like the red ball sticks as follows.

四角形

中程度の精度で自動的に生成された説明

* 1. Standard momentum (nothing to do with momentum in physics actually … but it does have better convergence rate than that of SGD)

Consider the gradient decent as a ball rolling down a hill, which is at the position **x** with a velocity **v.**

To update the velocity and the position (ignoring the time):

v **=** **mu** **\*** v **-** learning\_rate **\*** dx *# integrate velocity*

x **+=** v *# integrate position*

Where **mu** is a hyperparameter, the so-called momentum. However, it has nothing to do with momentum.

It’s supposed to be **the coefficient of fraction that defines the rate by which velocity decays (otherwise the particle would never come to a stop at the bottom of a hill)**.

Typical values of **mu** can be 0.5, 0.9, 0.95, 0.99.

A typical setting is to start with momentum of about 0.5 and anneal it to 0.99 or so over multiple epochs.

* 1. Nesterov momentum (Nesterov accelerated gradient, or NAG): look ahead and correct the actual next step!

A straightforward way to calculate Nesterov momentum:

*# evaluate next step by standard momentum*

x\_ahead **=** x **+** mu **\*** v

v **=** mu **\*** v **-** learning\_rate **\*** **dx\_ahead** *# velocity after correction by gradient at next step by standard momentum*

x **+=** v

To calculate Nesterov momentum:

At first we have a look at the step ahead by standard momentum whose position is x + mu\*v, and by which we have its gradient dx\_ahead.

To update the position (the actual one by Nesterov momentum), we do it by **replacing dx with dx\_ahead to make a correction**.

It does have better convergence than standard momentum especially for convex functions.

A demonstration of Nesterov momentum step as bellow.

文字が書かれている

中程度の精度で自動的に生成された説明

Another form to calculate Nesterov momentum (in the way like vanilla SGD):

v\_prev **=** v *# back this up*

v **=** mu **\*** v\_prev **-** learning\_rate **\*** dx *# velocity updates*

x **+=** **-**mu **\*** v\_prev **+** (1 **+** mu) **\*** v *# position update changes form*

(The derivation is to do… may check [ADVANCES IN OPTIMIZING RECURRENT NETWORKS](https://arxiv.org/pdf/1212.0901v2.pdf) at section 3.5)

1. A second-order method:

Without learning rate hyperparameters, positions can be updated as:

: gradient vector of function f at position x

: Hessian Matrix of f(x), a square matrix partial derivative of function f at position x, where

テーブル が含まれている画像

自動的に生成された説明

**Too time and memory consuming. Not recommended.**

1. **Adaptive learning rate methods**
   1. Adagrad

*# Assume the gradient dx and parameter vector x*

cache **+=** dx**\*\***2

x **+=** **-** learning\_rate **\*** dx **/** (np.sqrt(cache) **+** eps)

where

cache: to store the sum of square gradients

eps: to avoid division by zero. Usually set in range of [1e-8, 1e-4].

* 1. RMSprop

A simple modification of Adagrad, with a new hyperparameter decay\_rate (remember **mu** in momentum update? They have quite similar roles.)

cache **=** decay\_rate **\*** cache **+** (1 **-** decay\_rate) **\*** dx**\*\***2

x **+=** **-** learning\_rate **\*** dx **/** (np.sqrt(cache) **+** eps)

Simple yet robust, especially adjusting the Adagrad method where weights can reduce too aggressively.

Typical values of decay\_rate are [0.9, 0.99, 0.999].

* 1. **Adam (recommeded)**

The currently recommended learning rate adaptation method as the default algorithm.

m **=** beta1**\***m **+** (1**-**beta1)**\***dx

v **=** beta2**\***v **+** (1**-**beta2)**\***(dx**\*\***2)

x **+=** **-** learning\_rate **\*** m **/** (np.sqrt(v) **+** eps)

where

m: modified dx

v: modified cache (remember the Adagrad?)

Recommended values for the hyperparameters are:

eps = 1e-8

beta1 = 0.9

beta2 = 0.999

The updated Adam with a bias correction mechanism:

m **=** beta1**\***m **+** (1**-**beta1)**\***dx

mt **=** m **/** (1**-**beta1**\*\***t)

v **=** beta2**\***v **+** (1**-**beta2)**\***(dx**\*\***2)

vt **=** v **/** (1**-**beta2**\*\***t)

x **+=** **-** learning\_rate **\*** mt **/** (np.sqrt(vt) **+** eps)

where **mt** and **vt** are modified m and v respectively, and **t** is a iteration counter going from 1 to infinity.

As for impalements of codes, refer to:

<https://github.com/mantasu/cs231n/blob/master/assignment2/cs231n/optim.py>.