# Optimization: Stochastic Gradient Descent

Convolutional Neural Networks for Visual Recognition
`http://cs231n.github.io/`

Simon Bihel, `simon.bihel@ens-rennes.fr`

Friday 7th July, 2017

COINSE Lab, KAIST

## Table of contents

## Introduction

## Last week with Jinhan

A **score function** mapping the raw image pixels to class scores, e.g.

$$f(x_i, W) = Wx_i$$

With $W$ being a set of parameters.

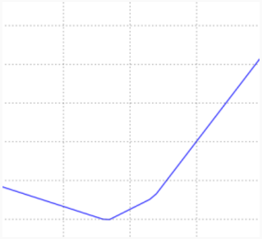
A **loss function** measuring the quality (how the scores agree with training data) of a set of parameters, e.g. SVM

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[ \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) \right] + \alpha R(W)$$

We want to minimize the loss function by tweaking $W$.

## Visualizing the loss function

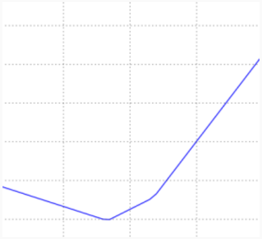
Difficult to visualize due to high-dimensional spaces. Gain some intuition by choosing a random $W$ and moving in a random direction ($L(W + aW_1)$)

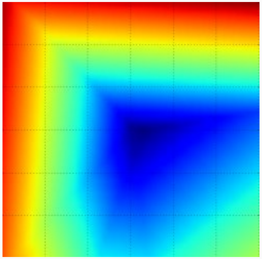


1 dim, 1 example

## Visualizing the loss function

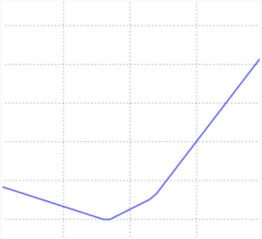
Difficult to visualize due to high-dimensional spaces. Gain some intuition by choosing a random $W$ and moving in a random direction ($L(W + aW_1)$) or 2 directions ($L(W + aW_1 + bW_2)$).

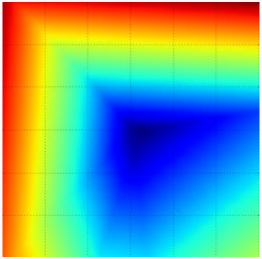

1 dim, 1 example

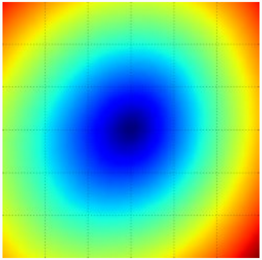

2 dim, 1 example

## Visualizing the loss function

Difficult to visualize due to high-dimensional spaces. Gain some intuition by choosing a random $W$ and moving in a random direction ($L(W + aW_1)$) or 2 directions ($L(W + aW_1 + bW_2)$).


1 dim, 1 example

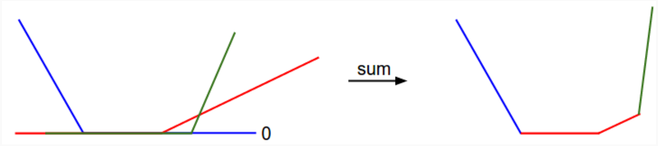

2 dim, 1 example


2 dim, 100 examples

## Piecewise-linear structure of the loss function

For a single example we have

$$L_i = \sum_{j \neq y_i} \left[ \max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$



In our examples the loss function is convex but it is not in practice with neural networks, thus we need more general minimization techniques.

## Optimization

## Strategy #1: Random Search

Obviously not the most accurate method on its own. But it is a good starting point to iteratively refine it.

## Strategy #2: Random Local Search

Start from a random $W$ and add a random perturbation $W + \delta W$ and perform an update if the loss is lower.

Still wasteful and computationally expensive.

## Strategy #3: Following the gradient

No need to randomly search for a good direction: we can compute the *best* direction with the steepest descend. This direction is called the **gradient** of the loss function

The gradient is just a vector of slopes (i.e. **derivatives**)

For a 1-D function it is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

For vectors it is simply the vector of derivatives for each dimension (called **partial derivatives**)

## Computing the gradient

## Numerically with finite differences

Slow, approximate but easy way.

For each dimension, make a small change and compute the partial derivative along that dimension.

Choosing the step size (aka *learning rate*) is one of the most important hyperparameter in training a NN. Large changes can miss minimums but are faster.

## Analytically with calculus

Derive a direct formula for the gradient. No approximations and very fast to compute but error-prone in the implementation.

Lets use the example of the SVM loss function for a single datapoint:

$$L_i = \sum_{j \neq y_i} \left[ \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta) \right]$$

$$\nabla_{w_{y_i}} L_i = - \left( \sum_{j \neq y_i} \mathbb{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) \right) x_i$$

You can check the correctness by comparing with the numerical gradient (**gradient check**).

## Gradient descent

It is the procedure of repeatedly evaluating the gradient and then performing a parameter update. This is the core of all NN libraries.

For large training data, a common approach is to compute the gradient over **batches** of data. This works well because training data are correlated.

**Stochastic Gradient Descent (SGD)** is the process of computing the gradient multiple times for only one example.