50.039 – Theory and Practice of Deep Learning

Alex

Week 02: Gradient methods

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

1 Gradient Descent Methods

Key takeaways:

- recapitulation of the meaning of gradients, in particular gradient and directional derivatives
- gradients for the optimization (max/min) of differentiable functions
- coding: implement gradient descent, see the sensitivity to stepsize and initialization choices, the difficulties of gradient optimization
- SGD and minibatches as noisy version of the batch gradient

Last class: explicit solution for linear regression with bias. Often there is no explicit solution known.

Example 1: when added an explicit bias to linear regression, no explicit solution is known anymore:

$$\begin{aligned} f(x) &= w \cdot x + b \\ \ell(f(x), y) &= (f(x) - y)^2 \\ (w^*, b^*) &= \text{argmin}_{w,b} L(f, \ell, D_n) = \text{argmin}_{w,b} \frac{1}{n} \sum_{(x_i, y_i) \in D} (w \cdot x_i + b - y_i)^2 \end{aligned}$$

Example 2: the linear model for classification with the hinge loss

$$\begin{aligned} f_{w,b}(x) &= w \cdot x + b \\ \ell(f_{w,b}(x), y) &= \max(0, 1 - y f_{w,b}(x)) \\ (w, b) &= \text{argmin}_{w,b} L(f, \ell, D_n) = \text{argmin}_{w,b} \frac{C}{n} \sum_{(x_i, y_i) \in D} \max(0, 1 - y f_{w,b}(x)) + \frac{1}{2}\|w\|^2 \end{aligned}$$

has no explicit solution, too

Input space: $\mathbb{R}^d$, **output space:** $\{-1, +1\}$ respectively $\mathbb{R}^1$

Model: use linear model for classification

$$f_{w,b}(x) = w \cdot x + b$$

– has real-valued outputs.

$$g(x) = sign(f_{w,b}(x)) \in \{-1, +1\}$$

can be used for classification with the linear model.

Loss function:
the first idea is to count the number of misclassified samples by the zero-one loss

zero-one loss

$$\ell(f(x), y) = 1[sign(f(x)) \neq y] = 1[f(x)y < 0]$$

Note here the possibility to rewrite the condition
$1[sign(f(x)) \neq y]$ as $1[f(x)y < 0]$ without any sign – that is: we have an error if the prediction function $f(x)$ has opposite sign of the ground truth label y.

$$f(x)y = \begin{cases} f(x) > 0, y > 0 & \text{no error} \\ f(x) < 0, y < 0 & \text{no error} \\ f(x) > 0, y < 0 & \text{error} \\ f(x) < 0, y > 0 & \text{error} \end{cases}$$

hinge loss

The hinge-loss for classification is given as

$$\ell(f(x), y) = \max(0, 1 - f(x)y)$$

The hinge-loss is an upper bound on the zero-one-loss.

The hinge loss is an upper bound on the zero one loss. The logic: if the hinge loss is low, the zero-one loss must also become low. Therefore it is ok to minimizes an average over the hinge loss instead of the average of zero-one losses over training samples

$$\text{argmin}_{w,b} \frac{1}{n} \sum_{(x_i, y_i) \in D} \max(0, 1 - y_i f_{w,b}(x_i))$$

Finally we add $L2$-regularization with a constant λ

$$\text{argmin}_{w,b} \frac{1}{n} \sum_{(x_i,y_i)\in D} \max(0, 1 - y_i f_{w,b}(x_i)) + \lambda \|w\|^2$$

For the sake of optimization it does not matter, whether one multiplies the weight with either the regularizer $\|w\|^2$ or with the loss term. Then one arrives at the formulation

$$\text{argmin}_{w,b} \frac{C}{n} \sum_{(x_i,y_i)\in D} \max(0, 1 - y_i f_{w,b}(x_i)) + \|w\|^2$$

What is the common general problem in example 1 and example 2? The general problem setting that one tries to solve in both examples:

given a class of functions f_w that depend on some parameters w, goal is to minimize

$$\frac{1}{n} \sum_i \ell(f_w(x_i), y_i) + \lambda R(f_w)$$

where $R(f_W)$ is a regularizer term. Since every mapping f_w is defined by its parameters w in a one-by-one relationship, this amounts to

$$w^* = \text{argmin}_w \frac{1}{n} \sum_i \ell(f_w(x_i), y_i) + \lambda R(f_w)$$

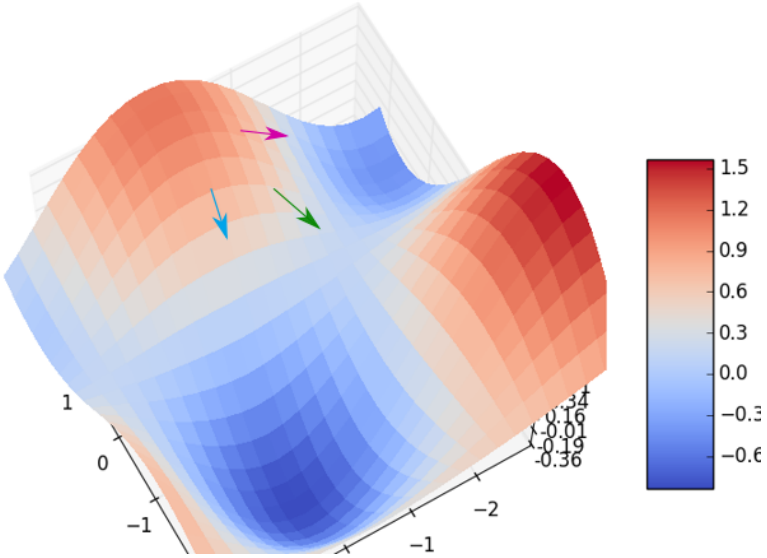
Gradient methods is a set of approaches to solve such problems, provided that the assumption holds that the function to be minimized is differentiable.

Problem setting for application of gradient-based minimization

The Problem Setting, in which gradient methods can be used:

- given some function $g(w)$
- goal: find $w^* = \text{argmin}_w g(w)$
- assumption: can compute $\nabla g(w)$ - the gradient in point w.

Idea: negative gradient at a point w is the direction of locally steepest function decrease from w.



Note: the direction of locally steepest decrease does not point to a global or local minimum.

Gradient Descent

Basic Algorithm: name: **Gradient Descent**:

- initialize start vector w_0 as something, step size parameter η
- run while loop, until function value changes very little (δ_g), do at iteration t:
  - $w_{t+1} = w_t - \eta \nabla_w g(w_t)$
  - compute change to last value: $\delta_g = \|g(w_{t+1}) - g(w_t)\|$

Properties and problems of gradient descent:
When does it converge ? When change is small, that is when $\eta \|\nabla_w g(w_t)\|$ becomes small. That means $\nabla_w g(w_t) \approx 0$, so a local optimum. Since we always went down, it must be local minimum, or a saddle point.

possible problems:

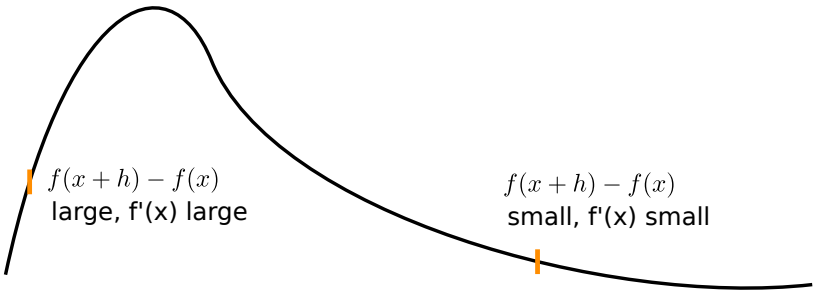
- we find a local minimum, not the global minimum of a function, can be good or bad.
- effects of bad stepsize: divergence – no solution, or slow convergence
- effects of starting point – which ones?

Lets explore these effects in class.

- in `learnThu8.py` run `tGD([stepsize])` to see the effect of different stepsizes. Why a too large stepsize can lead to numeric overflows? This is a common effect in deep learning training: too large stepsize, then training error will not go down.
- run `tGD2([initvalue])` with *initvalue* $\in [-4, +4]$ to see the effect of a constant stepsize, but different starting points – see in what minimum you end up.

Three Properties:

1. convergence to global optimum only if the function is convex and the stepsize is sufficiently small. In general one reaches only local minima, sometimes saddle points.
2. the size of the update step $w_{t+1} = w_t - \eta \nabla_w g(w_t)$ depends on the norm of the gradient, too. So when starting in a steep region, even a small stepsize can bring trouble.



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

the gradient stepsize depends not only on the stepsize parameter but also on the norm of the gradient

3. in practice: one starts with a learning rate, and decreases it over time, either with a polynomial decrease, or by a factor every N iterations. pytorch: `lr_scheduler`, gluon: trainer.set_learning_rate(...). This enforces convergence, but not necessarily to a good point.

What happens if one decreases the learning rate very fast?

2 Gradient Descent for linear regression

Found a closed form solution, but contains a matrix inverse. Can be too slow for high dimensions, why? Solving $X^T X w = X^T Y$ can be done in $O(d^3)$ – it is solving a linear equation system.

$$\sum_{i=1}^{n}(x_i \cdot w - y_i)^2 = (X \cdot w - Y)^T \cdot (X \cdot w - Y)$$

You can use the gradient for finding a good w:

$$D_w((X \cdot w - Y)^T \cdot (X \cdot w - Y)) = 2X^T \cdot (X \cdot w - Y)$$

in class task:

- check `gdLinReg`, run `t6()` this is gradient descent for linear regression.

3 Batch Gradient Descent versus stochastic gradient descent

Lets consider our setting where we have an average of losses over training samples:

$$\frac{1}{n}\sum_{i=1}^{n}\ell(f(x_i), y_i)$$

The application of vanilla gradient descent to this function results in the following algorithm:

batch gradient descent for an average of losses

- initialize start vector w_0 as something, choose step size parameter η
- run while loop, until function value changes very little (δ), do at iteration t:
  - $w_{t+1} = w_t - \eta\nabla_w(\frac{1}{n}\sum_{i=1}^{n}\ell(f_{w_t}(x_i), y_i))$
  - compute change to last value: $\delta = \frac{1}{n}\|\sum_{i=1}^{n}\ell(f_{w_{t+1}}(x_i), y_i)) - \ell(f_{w_t}(x_i), y_i))\|$

This is called batch gradient descent because it uses the set of **all training data samples** to compute the gradient in each step.
The alternative is **stochastic gradient descent** (SGD).

stochastic gradient descent computes in every iteration a gradient using only one sample every iteration, or, it uses a mini-batch like $k = 20$ samples – such that this set is chosen randomly from the set of all training samples. This is the default in deep learning.

Recap: **Batch gradient descent** uses as gradient:

$$\nabla_w \frac{1}{n}\sum_{i=1}^{n}\ell(f_w(x_i), y_i)$$

stochastic gradient descent for an average of losses

The core idea of **Stochastic gradient descent** is to compute the gradient only over a randomly selected subset of samples. Stochastic gradient descent when starting at index m and using the next k samples is:

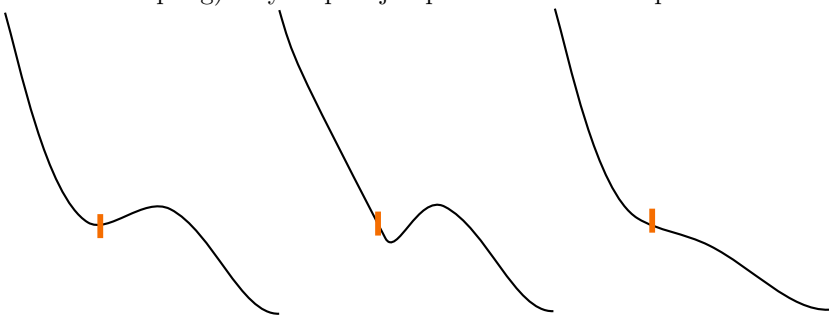
$$\nabla_w \frac{1}{k} \sum_{i=m+0}^{m+k-1} \ell(f_w(x_i), y_i)$$

- initialize start vector w_0 as something, choose step size parameter η
- run while loop, until function value changes very little (δ), do at iteration t:
  - select a random subset of k samples (usually by a random ordering of all training samples)
  - $w_{t+1} = w_t - \eta \nabla_w (\frac{1}{k} \sum_{i=m+0}^{m+k-1} \ell(f_w(x_i), y_i))$
  - compute change to last value: $\delta = \frac{1}{k} \| \sum_{i=m+0}^{m+k-1} \ell(f_{w_{t+1}}(x_i), y_i)) - \ell(f_{w_t}(x_i), y_i)) \|$

Advantages of stochastic gradient descent

- Full-batch is often too costly to compute a gradient using all samples when its more than tens of thousands
- SGD is a noisy, approximated version of the batch gradient.
- injecting small noise is one way to prevent overfitting! Sometimes SGD can be better than full batch gradient descent in finding *good* local optima.

An illustration why noise to the loss function (e.g. by randomized data sampling) may help to jump out of bad local optima



Left: a loss surfaces at some point (orange). Middle and Right: changes in the loss surfaces as different training data subsets are used. In the middle the gradient norm is much larger compared to the left case – allows to jump out of the local minimum.
A similar argument holds about adding noise to the gradient.

- One insight is that the stochastic gradient descent is an approximation to the batch gradient by using a subset of all samples.

(A) The batch gradient can be seen as an expected value with probability distribution having non-zero probability only for our observed n datapoints $(x_i, y_i) \in D_n$

$$P_{D_n}(x, y) = \begin{cases} \frac{1}{n} & \text{if } (x, y) = (x_i, y_i) \text{ for some } (x_i, y_i) \in D_n \\ 0 & \text{otherwise} \end{cases}$$

The empirical expectation is then:

$$\begin{aligned} \nabla_w \frac{1}{n} \sum_{(x_i,y_i)\in D_n} \ell(f_w(x_i), y_i) &= \sum_{(x_i,y_i)\in D_n} \nabla_w \ell(f_w(x_i), y_i) \ \frac{1}{n} \\ &= \sum_{(x_i,y_i)\in D_n} \nabla_w \ell(f_w(x_i), y_i) P_{D_n}((x_i, y_i)) \\ &= E_{(x_i,y_i)\sim P_{D_n}}[\nabla_w \ell(f_w(x_i), y_i)] \end{aligned}$$

This is an expectation of a gradient function $\nabla_w \ell(f_w(x_i), y_i)$. Remember here for a discrete set of values (x_i, y_i)

$$\sum_i r(x_i, y_i) P((x_i, y_i)) = E[r(x, y)]$$

(B) When performing stochastic gradient descent, we use a subset of samples from D_n in every step for computing the gradient. Using only a subset of samples from D_n is an approximation to the expectation shown in (A)!

This is law of large numbers/central limit theorem!

$$E_{(x_i,y_i)\sim P_{D_n}}[\nabla_w \ell(f_w(x_i), y_i)] \approx \sum_{i=m+0}^{m+k-1} \nabla_w \ell(f_w(x_i), y_i) \frac{1}{k}$$

The only thing that is necessary for the expectation to hold is that

- each $(x_i, y_i) \in D_n$ has equal probability $\frac{1}{n}$ to be used in the sum over k elements $\{m, \ldots, m + k - 1\}$
- drawing pairs (x_i, y_i),(x_k, y_k) is statistically independent

This is similar to the approximation $E_{(x,y)\sim P_{test}}[L(f(X), Y)] \approx \frac{1}{n} \sum_{(x_i,y_i)\in D_n} \ell(f_w(x_i), y_i)$ from last lecture – where we approximated the expectation under the unknown test distribution P_{test} by our training data set D_n

This noise from approximation can sometimes act as regularization – prevents to look at all the data too closely.

Still: without SGD, deep learning training would be not feasible in terms of time.

Another important regularization strategy used for gradient descent: early stopping of training when validation error gets sufficiently low.

4 Out of class: Outlook

How to optimize neural networks other than gradient?

- genetic algorithms – random walk in parameter space with guided selection of new parameter suggestions (option to keep old ones) on a validation data set. SoA Gradient provides a better guidance where to go than random choice of next parameters to evaluate.
- particle swarm / ant colony – related to above. Ants evolved in 2-D. Problem: volume/combinatorial explosion in high-dim spaces / bad coverage.
- Spiking neural nets - can be trained with gradient or other rules like STDP (spike timing dependent plasticity) see eg. equation(1) in `https://arxiv.org/pdf/1804.08150.pdf`. Currently (!), does not scale well to large problems. See SpykeTorch if you want to play.