50.021 -AI

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.]

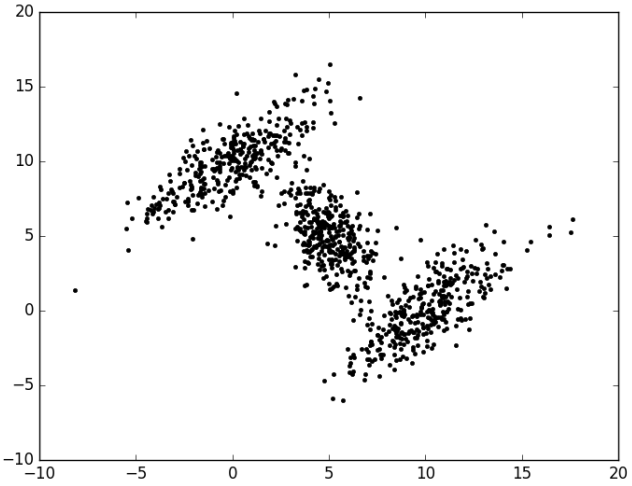
A review of last weeks in class coding

Many students were confused about C and y.

As stated, we want to predict y from x. C serves only as a helper to simulate data. Why do we use such a C ? Lets give a more clear example.

Suppose x would be a 2-dimensional vector of patient temperature (fever), and a measure of inflammation like erythrocyte sedimentation rate $x = (fev, esr)$.

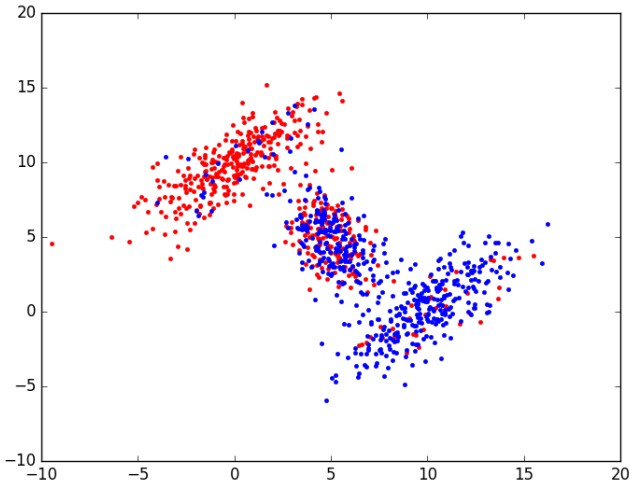
Now imagine you would look at the measurements of x of many patients. One can imagine, that when plotting it, one could see clusters of patients with similar values of x. As in the graphics with the black dots - three clusters are visible.



We want to predict the therapy outcome from above x. One can imagine, that the therapy success of treatment of patient may depend

on the values of x, right ? High fever is not a good sign. Let be $y = 0$ - therapy failure, and $y = 1$ therapy success.

Now we want to model/simulate data , where the probability of outcomes y depends on x indirectly - namely by the fact from which cluster the sample x came. The graphics shows such a case - red dots are failures.



We see one cluster with high fail rate, one cluster with medium fail rate, one cluster with high success rate.

This is a fairly general simulation model for data: x is concentrated in several clusters. The outcome $y \in \{0, 1\}$ depends on from which cluster x came.

For modeling that we define a cluster membership random variable C. This is what we did in the data generation example. We do not want to predict the cluster membership, we use it only as a helper to generate a dependency between x and y.

So during generation of a data sample (x, y) we must do the following:

- draw a cluster membership from a random variable $C \in \{1, 2\}$ - the probability for that is given by $P(C = 1) + P(C = 2) = 1$
- draw x depending on the value of C: the density for that is given by $f(x|C = 1), f(x|C = 2)$
- draw x depending on the value of C: the probability for that is given by $P(y = 0|C = 1), P(y = 0|C = 2)$. Note: in general this probability would depend on x, too, but here we assume in our model that the probability depends only on the value of C

See this general modelling idea. Why is this useful ? Sometimes you will need to test algorithms with data where you know $P(x, y)$ – when the algorithm runs but produces wrong results. Also: reviewing how to fraw from Gaussians might turn out later to be useful.

Gradient Descent Methods

Key takeaways:

- optimization by gradients
- SGD and minibatches as noisy version of the batch gradient
- coding: implement gradient descent, see the sensitivity to stepsize and initialization choices, the difficulties of gradient optimization

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

$$\sum_i L(f_w(x_i), y_i) + \lambda R(f)$$

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

$$w^* = \text{argmin}_w \sum_i L(f_w(x_i), y_i) + \lambda R(f)$$

For ordinary least squares, and ridge regression we had found a closed form solution. How to solve it when no such solution possible?

Problem Setting for gradient methods:

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

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

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

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

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.

Better versions: stepsize η not constant, decreases slowly as a function of iterations.

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 too slow
- effects of starting point – different starting points, different local minima as results.

Lets explore these effects in class.

in class task - try to finish in class, and submit as homework:

- Implement `gd` in `learnThu8.py` – the algorithm from above. check note on returning functions in python.
- 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.
  3 Takeaways
  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 stepsize of the gradient step depends on *the norm of the gradient*, too. So when starting in a steep region, even a small stepsize can bring trouble.
  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.

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 descent

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

in class task:

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

Batch Gradient Descent versus stochastic gradient descent

Reexamine: the `df` used in `t6()` computes a gradient using all samples of the data set: This is so called **batch gradient descent**. 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 20 samples. This is the default in deep learning.

Recap: **Batch gradient descent** is:

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

Stochastic gradient descent when starting at index m and using the next k samples is:

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

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.

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

The batch gradient can be seen as an empirical 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:

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

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

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

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 this expectation!

This is central limit theorem!

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

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 $E_{(x,y)\sim P}[L(f(X), Y)] \approx \frac{1}{n} \sum_{(x_i,y_i)\in D_n} L(f_w(x_i), y_i)$.

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

in class task:

- check `sgdLinReg` run `t7()`
- run `t8()` – here gradient descent has trouble to find a good solution

- run `t9()` – the SGD version of that, can sometimes find a better solution due to the noise induced by approximation

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.