COMP 642 — Machine Learning Jan 13, 2021

Lecture 2: Optimization for Most ML

Lecturer: Anshumali Shrivastava *Scribe By: Anshumali Shrivastava*

1 Supervised Learning

We will start with the most popular form of machine learning. Supervised learning subsumes all attractive models and applications that we hear.

1.1 Supervised Data and Problem Definition

We will continue with our house price prediction example. Data is part of the problem definition and input to the program. We will first define the problem, and the notion of data naturally follows.

A dataset of n pairs of objects and labels $\{x_i,\ y_i\}_{i=1}^n$. Here x_i represents the ***Features*** of object x_i, typically it is a vector in d (for some d) dimensions $\mathbb{R}^d$ or $x_i = \{x_{i1},\ x_{i2}, ..., \ x_{id}\}$. In our house price prediction problem object x_i will denote the house and x_{i1} can be say no of bedrooms, x_{i2} can be no of bathrooms, x_{i3} could be heating type in the house, x_{i4} could be lot size of house x_i, etc. y_i is the ***Target*** variable. It can be vector or scalar. For simplicity we can start with a single variable. In our house price prediction problem, y_i is the price that we want to predict.

The Problem: Given a dataset of n pairs of object and labels $\{x_i,\ y_i\}_{i=1}^n$, design a program $\mathcal{F}$ that takes a feature vector as input, such that given a new object with feature vectors given by x_{test} when we can calculate the value of the program (or function) $\mathcal{F}(x_{test})$, it is a very good estimate of the target variable y_{test}.

Few Notes:

- Dataset is part of problem definition and the design process.
- The success of program $\mathcal{F}$ depends on the quality of data. *Garbage in Garbage out principle.*
- Most machine learning courses define data as i.i.d draw from some distribution. The assumption that a fixed distribution exists is itself restrictive but needed to define generalization formally. Also, data samples being i.i.d drawn seems far from reality. For this course, there is no need to define the data formally.

1.2 Model

Note, our program is a function, it is consuming an input vector x_{test} and returns a number $\widehat{y_{test}}$ which is an estimate for the actual target y_{test}. The function, or the program $\mathcal{F}$, is called the **Model**.

Thus machine learning can be formulated as a search problem. From a set of, say, all exciting functions that can be designed, find a function $\mathcal{F}$ that gives a "good (or best)" result. So far, we haven't defined "good (or best)" yet, and we will do that next.

Example: Say we have only three possible functions (oversimplification) $\{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3\}$, can you device a strategy to pick the best function for a given data-set $\{x_i,\ y_i\}_{i=1}^n$? (We will need a loss function.)

1.3 Loss Function as a measure of goodness. Validation and Test Data set for Evaluations

We need a measure to compare two models $\mathcal{F}_1$ and $\mathcal{F}_2$.
Natural Idea: We know that $\mathcal{F}(x_i)$ is the models prediction for object features x_i and it should be "close" to y_i, and this should hold for all the examples x_i in the dataset. We can define the measure of goodness of a function $\mathcal{F}$ as follows

$$L(\mathcal{F}) = \frac{1}{n}\sum_{i=1}^{n} dist(\mathcal{F}(x_i), y_i),$$

here distance is any distance. In our house price prediction, we know that y_i is a one dimensional number. We can use the standard euclidean distance where $dist(\mathcal{F}(x_i), y_i) = (\mathcal{F}(x_i) - y_i)^2$. The distance is problem dependent, for example later we will see for classification we use cross entropy loss which is closer to KL divergence distance.

Clearly, we prefer $\mathcal{F}_1$ or $\mathcal{F}_2$ depending on whichever has a lower value of the loss.
Caveat: Remember, the model $\mathcal{F}$ is any function or program and hence can even return a simple map of key-value pairs given by $(x_i, y_i)\ \forall\ i$. It will give us 0 for the loss value, the best possible value. However, it is useless because it cannot predict anything outside the dataset. (Dumb memorization).

To avoid that, we create a partition of the data $\{x_i,\ y_i\}_{i=1}^n$ into the train and test set. We develop and find the model $\mathcal{F}$ using only the train set, and we evaluate the loss on the test set. The loss of $\mathcal{F}$ on the train is called train loss, and the loss on the test set is called test or holdout loss. Since the model never sees x_i from the test set, dumb memorization will not work on test loss. The loss on the test set also measures how well the model generalizes to data samples it has never seen before.

In practice, we also use a validation set, a third partition, and its use will be more apparent when we go into experiments.

2 How to find a "good" Model? Iterative Descent Algorithms

If someone gives me a small finite set of potential models, we can always find the best. However, we want to search for a much larger class. Ideally, all possible functions, but that is impossible. We want to search through a broad class of functions likely to contain a "good enough" function (or model).

So we pick a family of models: Linear, Neural Networks, Decision Trees, etc. (they will become clearer later). We usually define a family of models. Then we "try" to pick the best model from the family. Usually, the easiest way to find a "reasonable" class of models is to define a parameterized family.

For example: We can find the best model $\mathcal{F}$ from a linear family $\mathcal{F}_w(x) = w^T x = \sum_{i=1}^{d} w_i \times x_i$, or say a quadratic family $\mathcal{F}_w(x) = \sum_{i=1}^{d} w_i \times x_i + \sum_{i,j=1}^{i,j=d} w_{ij} \times x_i \times X_j$. here w is any real vector. Thus, our class of functions have infinite functions (or models).

Once, we pick the model class, parameterized by w, the goal can be formally stated: **Goal:** Find a w which minimized the loss $L(\mathcal{F}_w) = \frac{1}{n}\sum_{i=1}^{n} dist(\mathcal{F}_w(x_i), y_i)$ or formally

$$w^* = \arg\min_w \sum_{i=1}^{N} dist(\mathcal{F}_w(x_i), y_i),$$

we then return $\mathcal{F}_w$ as our program or model.

2.1 How do I find a minimum of a parametric function?

Clearly, we are looking at finding an assignment to the parameters w that minimizes the loss function. We should note that the variable is w and not x. x is data and is given to us.

High School Memory: Compute Gradient with respect to w, and equate to 0, etc., etc. Thinkable with well-chosen functions. We won't be able to find a closed-form solution with almost all interesting models. When we equate to 0, we won't be able to solve that equation. In high school, we were only given solvable things, and most optimization in real machine learning is not solvable in closed form.

Savior: Iterative Methods In High School: Single variable functions, finding roots, tangent-secant, Netwon-Raphson, etc. All these methods, start from w_0, find successive w_t, such that $f(w_t) < f(w_{t-1})$, the process will always converge. We can discuss if the convergence is good or not later. But it will converge or stop making reasonable progress after a while.

In one dimension there are only two directions: Assuming f is continuous. We have only two directions to move w_t, positive or negative, $w_t + \delta$ or $w_t - \delta$ ($\delta > 0$). A simple idea is to pick δ small enough such that we can always find a better w_{t+1} unless w_t is already local minimum.

$$w_{t+1} = \begin{cases} w_t + \delta & \text{if } \; f(w+\delta) \le f(w) \\ w_t - \delta & otherwise. \end{cases} \quad (1)$$

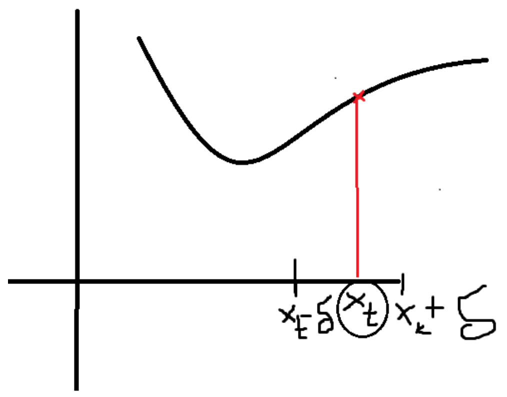

Figure 1: .

In more than one dimension there are infinite directions: If w_t is a vector, we have infinitely many possible δ choices. **Good news:** It turns out half of them will lead to a decrease in the function value. Why? (see next, either they have a positive projection with the gradient vector or a negative projection), but we can randomly pick a few directions and test them. We saw a python notebook (in Assignment 1) in the class where we randomly probed from 8 random directions to get a direction of decrease.

2.2 Gradients: Direction of Steepest Descent

It turns out that there is a sure-shot way to find a very good direction to reduce the function value, provided we can compute its gradients. Since we have a function over vectors, we can approximate as small change in the direction $\vec{\delta}$, where $\vec{\delta}$ is a unit vector, via Taylor series as

$$f(\vec{w_t} + \eta\vec{\delta}) = f(\vec{w_t}) + \eta(\vec{\delta} \cdot \nabla f(\vec{w})|_{w_t}) + higherorderterms$$

Here $\nabla f(\vec{w})|_{w_t} = \nabla f(\vec{w_t})$ is the gradient vector evaluated at w_t, assuming it exits. This approximation clearly states that the magnitude of change is given by $(\vec{\delta} \cdot \nabla f(\vec{w_t}))$. Thus, if the unit vector's direction $\vec{\delta}$ is aligned with $-\nabla f(\vec{w_t})$ we will see the smallest value of $f(\vec{w_t} + \eta\vec{\delta})$.

Thus, **for small enough** η, we have a simple iterative **Gradient Descent Algorithm** to minimize a function f(w), where we start with random w_0 and iteratively calculate w_t, given by

$$w_{t+1} = w_t - \eta \nabla f(w_t)$$

Some points of ponder:

- Gradient Descent wont make progress if you reach any (Local) Minima (where the gradient $\nabla f(w)|_{w_t} = 0$
- If you reach any minima, gradients will be zero, and you will not make any update.
- Close of minima, the gradients will be small, and progress will be slow.
- The reduction is guaranteed if the step size is infinitesimal, but then the progress will also be infinitesimal. Large step size may overshoot.

2.3 Ideal Step Size and Oscillations

We have been saying η is small enough to ensure that we find a better $f(w_{t+1}) < f(w_t)$. We can ask, what is the best η and the answer is another optimization given by,

$$\eta^* = \arg\min_{\eta} f(w_t + \eta \nabla f(w_t))$$

The above optimization, which simply calculates the maximum possible decrease that can happen in a given direction $\nabla f(w_t)$ may be even harder than the original optimization.

2.3.1 Higher order approximation and why they are almost always prohibitive.

It is possible to get a better mathematical update by doing higher order Taylor series approximation

$$f(\vec{w_t} + \eta\vec{\delta}) = f(\vec{w_t}) + \eta(\vec{\delta} \cdot \nabla f(\vec{w})|_{w_t}) + \frac{\eta^2(\nabla^2)vec(f(w))}{2} + \sum_{n=3}^{\infty} \frac{\eta^n(\nabla^n)vec(f(w))}{n!}$$

If we truncate the Taylor series at second order (quadratics), we get what is called **Newtons update**. Note we cannot go beyond quadratics because we don't have closed-form solutions anymore. **High School Wisdom:** We can always minimize quadratics in closed form but not beyond that. This is the only reason why second-order methods are the only ones we find in books and not beyond that. Computation-wise, even second-order methods are barely worth the efforts.

In a head-to-head comparison, the second-order method is likely to be slower than first-order: Second-order methods require computing the inverse of a second derivative matrix (also called Hessian a dxd matrix). This cost is almost always not justifiable over first-order methods. When we do one update with the second-order method (require O(d^3) operations due to inversion), we can do d^2 steps of straightforward gradient descent, which will almost always result in a faster algorithm. The best way is to verify the wall clock time execution of these two methods (left to exercise if you are interested)

2.3.2 Total Cost = Cost per Update x No of Updates

Reducing the no of steps to convergence need not be faster. For example, Method 1 can take 100 steps, but each step is 20x costly. This optimization will be slower than Method 2, taking 300 steps but with a less costly update step!. **Exception**: when we have additional parallelism while calculating the update. However, giving additional parallelism to computing gradients implies we are not using "enough data parallelism" which we will discuss later in the course. We can also state the same as follows:

Total Cost = Cost to find a direction and move x No of times we move

3 Batch Gradient Descent: Even a gradient computation may not be worth it.

We need a "good enough" direction to move: Coming back to the problem of minimizing loss. Our loss is a function over parameters w, which is an average over the n data points.

$$L(\mathcal{F})_w = \frac{1}{n}\sum_{i=1}^{n} dist(\mathcal{F}(x_i), y_i),$$

the gradient of this is another average

$$\nabla_w L(\mathcal{F})_w = \frac{1}{n}\sum_{i=1}^{n} \nabla_W dist(\mathcal{F}(x_i), y_i),$$

Clearly, the total cost to compute a gradient $O(n)$ as it requires computing the gradient over all the n data samples $\{x_i,\ y_i\}$ and averaging them. Clearly if n is large (big dataset), this is prohibitive. In many application reading the full data itself is the primary cost.

A simple idea, and one of the most powerful which makes machine learning practical on large dataset is the "heuristic" of approximating the average gradient by picking a small sample of data and only averaging the gradient over it. Thus, we pick a random small fraction of $k \ll n$ data samples $\{x_i, y_i\}_{i=1}^{k}$ and return

$$\nabla_w L(\mathcal{F})_w = \frac{1}{k}\sum_{i=1}^{k} \nabla_W dist(\mathcal{F}(x_i), y_i),$$

as a good enough direction for update. When $k = 1$ we have a popular name of the method known as *Stochastic Gradient Descent.*

Tradeoffs: When k, also called **Batch Size**, is big, the cost of update is more but the gradient direction is better and will require less total number of updates. On the other hand, when k is small, the cost of update is less but the direction is more noisy and may not always results in good decrease leading to more number of updates.

Ideally, we want more samples to average, but cannot afford to read and average over large data samples in many applications.

There is no good recommendation for k: However, just like every other hyperparameters in machine learning, including step size, the ideal choice of k is a mystery. What works best for the problem at hand is always determined by experimental evaluations. Having said that many popular pipelines use 128 to 256 most of the time.

4 Idea of Running Average and momentum as a cheap proxy to get better gradient estimates. We already calculated something in the last few steps that can be useful

So let say, we are at w_t we randomly sampled k data points, compute the gradient approximate with batch size k as $\nabla f(w_t)$. Then we make a move to $w_{t+1} = w_t - \eta \nabla f(w_t)$. Now we again used batch size k to calculate $\nabla f(w_{t+1})$. Here, assume that w_t and w_{t+1} are not very far $w_{t+1} \simeq w_t$, then $\frac{\nabla f(w_t)+\nabla f(w_{t+1})}{2}$ is a better gradient estimate. Why? (We are averaging over a larger random sample set of effectively $2k$, but we don't pay the cost of a bigger average as we already calculated $\nabla f(w_t)$ on a random sample before.). In fact, we can even think about $\frac{\nabla f(w_{t-1})+\nabla f(w_t)+\nabla f(w_{t+1})}{3}$.,

4.1 Decay to discount for old gradients. Stale gradients should be treated appropriately

Clearly, if we are at w_t, we want to give more weights to recent gradient $\nabla f(w_t)$ as compared to a slightly old $\nabla f(w_{t-1})$ because we have moved from w_{t-1} and that information is stale. Ideally, we can re weight the average to reflect this kind of intuition by giving more weights to recent gradients and less weight to older gradients as $\beta \nabla f(w_t) + (1-\beta)\nabla f(w_{t+1})$, with $beta < 0.5$. The older the gradient should decay more.

Running Average, we don't need to store all past gradients: We can change our gradient descent to be something like this: start with w_0 at step t, we have w_t, calculate $\nabla f(w_t)$, keep one additional vector (they call is momentum, we remember our past direction) $m_t = \beta m_t + (1-\beta)\nabla f(x_t)$, for some appropriate $\beta < 1$, and the do gradient descent as

$$x_{t+1} = x_t - \eta m_t$$

Exercise: Note the decay is automatic. The weight of $\nabla f(w_{t-1})$ is in the average is β, $\nabla f(w_{t-1})$'s weight is β^2, $\nabla f(w_{t-2})$'s become β^3, etc. They are exponentially decaying. We really don't have much choice, because we don't have the resource to memorize every $\nabla f(w_t)$, we can keep a running sum thought.

4.2 The use of (almost) a standard deviation: Adaptive Gradient Methods are basically good normalization heuristics

Lets say we have two parameters $w1$ and $w2$. Imagine the last few gradients, say last 4 gradients, using batch gradient descent for $w1$ has been $w1_{t-3} = 10,\ w_{t-2} = 9.8,\ w_{t-2} = 10.2,\ w_{t-1} = 10,$

whereas the same updates for w_2 were $w2_{t-3} = 1,\ w_{t-2} = 19,\ w_{t-2} = -11,\ w_{t-1} = 21$.

The running average (without decay) for both $w1$ and $w2$ are the same 10. However, the gradient of +10 is very reliable for $w1$ as it is very consistent (small variation or standard deviation). While the gradient for $w2$ is unreliable because of fluctuations, which could be due to cheap approximate gradient computation for $w2$. As a result, we should not have a constant learning rate η for both $w1$ and $w2$. Ideally, updates for $w1$ can be accelerated significantly as we know its gradient has been positive ten all the time. However, we should decelerate for $w2$ because it oscillates.

A simple heuristic fix here is to standardise the mean of both $w1$ and $w2$ by dividing it with something like a standard deviation. If $m1$ and $m2$ are the means and $v1$ and $v2$ are the standard deviation (square root of variance (not exactly a variance as we don't subtract the mean)) then we replace the gradients by $\frac{m1}{v1}$ and $\frac{m1}{v1}$.

If we include the decay, then we recover the popular **ADAM algorithm**. As we have seen, we have m_t for a discounted running average of gradient (called momentum). In addition, we keep another discounted running average of something similar to variance v_t (sum of squares of discounted gradients), popularly called velocity (the name is confusing as it is merely used as a proxy of standard deviation). Our algorithm for update looks like the follows:

Start with w_0. At step t, we have w_t, calculate $\nabla f(w_t)$, keep two additional vector (they call is momentum and velocity), we can keep running average without remembering all the history) $m_t = \beta_1 m_t + (1-\beta_1)\nabla f(x_t)$, and $v_t = \beta_2 v_t + (1-\beta_2)(\nabla f(x_t) \cdot \nabla f(x_t))$ for some appropriate $\beta_1, \beta_2 < 1$, and the do gradient descent as

$$x_{t+1} = x_t - \eta \frac{m_t}{\sqrt{v_t} + \epsilon},$$

Here $\nabla f(x_t) \cdot \nabla f(x_t)$ is component wise vector multiplications (still a vector not an inner product) and $\epsilon = 10^{-8}$ is used to prevent divide by zero error.

If you have understood this far, you can now appreciate the ideas behind several other variants discussed in https://ruder.io/optimizing-gradient-descent/

5 Comments on Variety of Gradient Descent Ideas

Many Many More Similar Tricks with different twists: There are many many ideas but they essentially revolve around what we just saw. Please read this[1] nice blog to familiarize with a variety of tricks. Note, people have different way of explaining different things and they follow different lines of explanations. You are free to follow whatever resonates best with you. However, there is no concrete mathematical way yet to argue which among the ones will be better for problem at hand. Only real evaluation and experiments can identify the best performing idea. Note, there is no silver bullet gradient descent method yet. Having said that many software packages use Adam by default. If they show weird behavior on your problem, now you should have some ideas on what to try!

[1]https://ruder.io/optimizing-gradient-descent/