Chapter 1: Basic Setup

This Chapter introduces the basic nomenclature. Training/test error, generalization error etc. ≪Tengyu notes: todo: Illustrate using the curve seen during training. Mention some popular architectures (feed forward, convolutional, pooling, resnet, densenet) in a brief para each. ≫

We review the basic notions in statistical learning theory.

- A space of possible data points $\mathcal{X}$.
- A space of possible labels $\mathcal{Y}$.
- A joint probability distribution P on $\mathcal{X} \times \mathcal{Y}$. We assume that our training data consists of n points

  $$(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}) \overset{\text{i.i.d.}}{\sim} P$$

  each drawn independently from P.
- Hypothesis space: $\mathcal{H}$ is a family of hypotheses, or, a family of predictors. E.g., $\mathcal{H}$ could be the set of all neural networks with a fixed architecture $H = \{h_\theta\}$ where h_θ neural nets parameterized by parameters θ.
- Loss function: $\ell : (\mathcal{X} \times \mathcal{Y}) \times \mathcal{H} \to \mathbb{R}$.
  - E.g., in binary classification, we have $\mathcal{Y} = \{-1, +1\}$, and suppose the hypothesis is $h_\theta(x)$, then the logistic loss function is

    $$\ell((x, y), \theta) = \frac{1}{1 + \exp(-y h_\theta(x))} \tag{1}$$
- Expected loss: $L(h) = \mathbb{E}_{(x,y)\sim P}\left[\ell((x, y), h)\right]$, where P is a data distribution over $\mathcal{X} \times \mathcal{Y}$. Moreover, we define $h^* \in \operatorname{argmin}_{h \in \mathcal{H}} L(h)$ as the minimizer of the expected loss.
- Training loss (also known as empirical risk):

  $$\hat{L}(h) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(\left(x^{(i)}, y^{(i)}\right), h\right)$$

  where $\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(n)}, y^{(n)}\right)$ are n training examples drawn i.i.d. from P.

- Empirical risk minimizer (ERM): $\hat{h} \in \operatorname{argmin}_{h \in \mathcal{H}} \hat{L}(h)$.
- Regularization: Suppose we have a regularizer $R(h)$, then the regularized loss is

$$\hat{L}_\lambda(h) = \hat{L}(h) + \lambda R(h)$$

.

Chapter 2: Basics of Optimization

This chapter sets up the basic analysis framework for gradient-based optimization algorithms and relates how it applies to deep learning.

≪Tengyu notes: Sanjeev notes:

Suggestion: when introducing usual abstractions like Lipschitz constt, Hessian norm etc. let's relate them concretely to what they mean in context of deep learning (noting that Lipschitz constt is wrt the vector of parameters). Be frank about what these numbers might be for deep learning or even how feasible it is to estimate them. (Maybe that discussion can go in the side bar.)

BTW it may be useful to give some numbers for the empirical liptschitz constt encountered in training.

One suspects that the optimization speed analysis is rather pessimistic.≫

Gradient descent

Suppose we want to optimize a function $f(x)$

$$\min f(x) \tag{2}$$

The gradient descent algorithm is

$$x_0 = \text{initializaiton} \tag{3}$$

$$x_{t+1} = x_t - \eta \nabla f(x_t) \tag{4}$$

where η is the step size or learning rate.

One of the motivation or justification of the GD is that the $-\nabla f(x_t)$ is the steepest descent direction locally. Consider the Taylor expansion at a point x_t

$$f(x) = f(x_t) + \underbrace{\langle \nabla f(x_t), x - x_t \rangle}_{\text{linear in } x} + \cdots \tag{5}$$

Suppose we optimize the first order approximation of the function in a neighborhood of x_t

$$\underset{x}{\text{argmin}} f(x_t) + \langle \nabla f(x_t), x - x_t \rangle \tag{6}$$

$$\text{s.t.} \|x - x_t\|_2 \le \epsilon \tag{7}$$

Then, the optimizer the program above is equal to $x + \delta$ where

$$\delta = -\alpha \nabla f(x_t) \tag{8}$$

for some positive scalar α. In other words, to locally minimize the first order approximation of $f(x)$ around x_t, we should move towards the direction $-\nabla f(x_t)$.

Formalizing the Taylor Expansion. We will state a lemma that characterizes the descent of function values for GD. We make the assumption that the eigenvalues of $\nabla^2 f(x)$ is bounded between $[-L, L]$ for all x. We call function satisfying it L-smooth function. This allows us to approximate the function using Taylor expansion accurately in the following sense:

$$f(x) \le f(x_t) + \langle \nabla f(x_t), x - x_t \rangle + \frac{L}{2}\|x - x_t\|_2^2 \tag{9}$$

Descent lemma for gradient descent The following says that with gradient descent and small enough learning rate, the function value always decrease unless the gradient at the iterate is zero.

Lemma 1 (Descent Lemma)**.** *Suppose f is L-smooth. Then, if $\eta < 1/(2L)$, we have*

$$f(x_{t+1}) \le f(x_t) - \eta/2 \cdot \|\nabla f(x_t)\|_2^2 \tag{10}$$

The proof uses the Taylor expansion. The main idea is that even using the upper provided by equation (9).

Proof. We have that

$$\begin{aligned} f(x_{t=1}) &= f(x_t - \eta \nabla f(x_t)) && (11)\\ &\le f(x_t) - \eta \langle \nabla f(x_t), -\eta \nabla f(x_t)\rangle + \frac{L}{2}\|\eta^2 \nabla f(x_t)\|_2^2 \\ & && \text{(by equation (9))}\\ &= f(x_t) - (\eta - \eta^2 L/2)\|\eta^2 \nabla f(x_t)\|_2^2 \\ &\le \eta/2 \cdot \|\nabla f(x_t)\|_2^2 && \text{(by } \eta \le L/2) \end{aligned}$$

□

Stochastic gradient descent

- Motivation: Computing the gradient of a loss function could be expensive. Recall that

$$\hat{L}(h) = \frac{1}{n}\sum_{i=1}^{n} \ell\left(\left(x^{(i)}, y^{(i)}\right), h\right) \tag{12}$$

  Computing the gradient $\nabla \hat{L}(h)$ scales linearly in n. Stochastic gradient descent (SGD) estimate gradient by sampling a mini-batch of gradients. Especially when the gradients of examples are similar, the estimator can be reasonably accurate.

For simplicity, we simplify the notations a bit. We consider optimizing the function

$$\frac{1}{n}\sum_{i=1}^{n} f_i(x) \tag{13}$$

At each iteration t, the SGD algorithm first sample $i_1, \ldots, i_B$ uniformly from $[n]$, and then compute the estimated gradient

$$g_S(x) = \frac{1}{B}\sum_{k=1}^{B} \nabla f_{i_k}(x_t) \tag{14}$$

Here S is a shorthand for $\{i_1, \ldots, i_B\}$. The SGD algorithm updates

$$x_{t+1} = x_t - \eta g_S(x_t) \tag{15}$$

Accelerated Gradient Descent

Heavy-ball algorithm has the following update rule:

$$x_{t+1} = x_t - \eta \nabla f(x_t) + \beta(x_{t+1} - x_t) \tag{16}$$

Here $\beta(x_{t+1} - x_t)$ is the momentum term.

≪Tengyu notes: perhaps mention the ODE connection≫ ≪Tengyu notes: missing a figure≫

Another equivalent way to write the algorithm is

$$u_t = -\nabla f(x_t) + \beta u_{t-1} \tag{17}$$
$$x_{t+1} = x_t + \eta u_t \tag{18}$$

Exercise: verify the two forms of the algorithm are indeed equivalent.

Another variant of the heavy-ball algorithm is called due to Nesterov

$$u_t = -\nabla f(x_t + \beta(u_t - u_{t-1})) + \beta u_{t-1} \tag{19}$$
$$x_{t+1} = x_t + \eta u_t \tag{20}$$

One can see that u_t stores a weighed sum of the all the historical gradient.

Nesterov gradient descent works similarly to the heavy ball algorithm empirically for training deep neural networks. It has the advantage of stronger worst case guarantees on convex functions. Both of the two algorithms can be used with stochastic gradient, but little is know about the theoretical guarantees about stochastic accelerate gradient descent.

Local Runtime Analysis of GD

When the iterate is near a local minimum, the behavior of gradient descent is clearer because the function can be locally approximated by a quadratic function. In this section, we assume for simplicity that we are optimizing a convex quadratic function, and get some insight on how the curvature of the function influence the convergence of the algorithm.

We use gradient descent to optimize

$$\min_x \frac{1}{2} x^\top A x \tag{21}$$

where $A \in \mathbb{R}^{d\times d}$ is a positive semidefinite matrix, and $x \in \mathbb{R}^d$.
Remark: w.l.o.g, we can assume that A is a diagonal matrix. **Diagonalization is a fundamental idea in linear algebra.** Suppose A has singular vector decomposition $A = U\Sigma U^\top$ where Σ is a diagonal matrix. We can verify that $x^\top Ax = \hat{x}^\top \Sigma \hat{x}$ with $\hat{x} = U^\top x$. In other words, in a difference coordinate system defined by U, we are dealing with a quadratic form with a diagonal matrix Σ as the coefficient. Note the diagonalization technique here is only used for analysis.

Therefore, we assume that $A = \mathrm{diag}(\lambda_1, \ldots, \lambda_d)$ with $\lambda_1 \geq \cdots \geq \lambda_d$. The function can be simplified to

$$f(x) = \frac{1}{2}\sum_{i=1}^{d} \lambda_i x_i^2 \tag{22}$$

The gradient descent update can be written as

$$x \leftarrow x - \eta \nabla f(x) = x - \eta \Sigma x \tag{23}$$

(Here we omit the subscript t for the time step and use the subscript for coordinate.) Equivalently, we can write per-coordinate update

$$x_i \leftarrow x_i - \eta \lambda_i x_i = (1 - \lambda_i \eta_i) x_i \tag{24}$$

Now we see that if $\eta > 2/\lambda_i$ for some i, then the update the absolute value of x_i will blow up exponentially and lead to instable behavior. Thus, we need $\eta \lesssim \frac{1}{\max \lambda_i}$. Note that $\max \lambda_i$ corresponds to the smoothness parameter of f because λ_1 is the largest eigenvalue of $\nabla^2 f = A$. This is consistent with the condition in Lemma 1 that η needs to be small.

Suppose for simplicity we set $\eta = 1/(2\lambda_1)$, then we see that the convergence for x_1 coordinates is very fast — the coordinate x_1 is halved every iteration. However, the convergence of the coordinate x_d is slower, because it's only reduced by a factor of $(1 - \lambda_d/(2\lambda_1))$ every iteration. Therefore, it takes $O(\lambda_d/\lambda_1 \cdot \log(1/\epsilon))$ iterations to converge to an error ϵ.

The condition number is defined as $\kappa = \sigma_{\max}(A)/\sigma_{\min}(A) = \lambda_1/\lambda_1$, which governs the convergence rate of GD.

≪Tengyu notes: add figure≫

Pre-conditioners

From the toy quadratic example above, we can see that it would be more optimal if we can use a different learning for different coordinate. In other words, if we introduce a learning rate $\eta_i = 1/\lambda_i$ for each coordinate, then we can achieve faster convergence. In the more general setting where A is not diagonal, we don't know the coordinate system in advance, and the algorithm corresponds to

$$x \leftarrow x - A^{-1}\nabla f(x) \tag{25}$$

in the even more general setting where f is not quadratic, this corresponds to the Newton's algorithm

$$x \leftarrow x - \nabla^2 f(x)^{-1}\nabla f(x) \tag{26}$$

Computing the hessian $\nabla^2 f(x)$ can be computational difficult because it scales quadratically in d (which can be more than 1 million in practice). Therefore, approximation of the hessian and its inverse is used:

$$x \leftarrow x - \eta Q(x)\nabla f(x) \tag{27}$$

where $Q(x)$ is supposed to be a good approximation of $\nabla^2 f(x)$, and sometimes is referred to as a pre-conditioner. In practice, often people first approximate $\nabla^2 f(x)$ by a diagonal matrix and then take its inverse. E.g., one can use $\text{diag}(\nabla f(x)\nabla f(x^\top))$ to approximate the Hessian, and then use the inverse of the diagonal matrix as the pre-conditioner.

≪Tengyu notes: more on adagrad?≫