Machine Learning (CSE 446): Variations on our Themes

Sham M Kakade

University of Washington
cse446-staff@cs.washington.edu

Hi!

Announcements

Allen Center front office.

- Midterm review in section.
  (pickup cheat sheets in ~~OHs~~)
- See class website for another grading scheme.
- HW3 "milestone" due thurs.
- lots of good extra credit!
- Today: Variations on our Themes:
  - sgd $\rightarrow$ mini-batch sgd
  - binary classification $\rightarrow$ multi-class classification
  - linear methods $\rightarrow$ non-linar methods

Our running example for the loss minimization problem

$$\operatorname*{argmin}_{\mathbf{w}} \frac{1}{N}\sum_{n=1}^{N}\frac{1}{2}\left(y_n - \mathbf{w}\cdot\mathbf{x}_n\right)^2 + \frac{1}{2}\lambda\|\mathbf{w}\|^2$$

- How do we run GD/SGD?

- how do we set the step size? λ? the "mini-batch" size?

Theory helps with guidance/(sometimes) auto-tuning. Ultimately, we just have try to tune these ourselves to get experience. HW3 helps.

review: GD for the square loss

Data: step sizes $\langle \eta^{(1)}, \ldots, \eta^{(K)} \rangle$
Result: parameter $\mathbf{w}$
initialize: $\mathbf{w}^{(0)} = \mathbf{0}$;
for $k \in \{1, \ldots, K\}$ **do**
$\quad \mathbf{w}^{(k)} = \mathbf{w}^{(k-1)} + \eta^{(k)} \left(\frac{1}{N} \sum_n (y_n - \mathbf{w}^{(k-1)} \cdot \mathbf{x}_n)\, \mathbf{x}_n \right)$; $+ \lambda w \,)$
end
return $\mathbf{w}^{(K)}$;

Algorithm 1: SGD

- the term in red is a costly to compute!
- Even by using matrix multiplications (and not explicitly doing the sum), it is often too slow.

Gradient Descent Tips

- how do we set the stepsize?
  - Remember: we diverge/unstable if the step size is too big!
  - you just set it a little lower (like $1/2$) less than when things start to diverge/error starts to drop.
- do we decay it?
  No: GD will converge just fine without decaying the learning rate.
- Is GD a good algorithm?
  if convex, then it is 'poly time'. but GD is often too slow:
  - computing the gradient of the objective function involves a sum over
- SGD: let's sample the gradient!

SGD: review

Expectation is

Data: step sizes $\langle \eta^{(1)}, \ldots, \eta^{(K)} \rangle$
Result: parameter $\mathbf{w}$
initialize: $\mathbf{w}^{(0)} = \mathbf{0}$;
for $k \in \{1, \ldots, K\}$ **do**
  Sample $n \sim \mathrm{Uniform}(\{1, \ldots, N\})$;
  $\mathbf{w}^{(k)} = \mathbf{w}^{(k-1)} + \eta^{(k)} \left[\left(y_n - \mathbf{w}^{(k-1)} \cdot \mathbf{x}_n\right) \mathbf{x}_n + \lambda w \right]$
end
return $\mathbf{w}^{(K)}$;

Algorithm 2: SGD

- the term in red is a "sampled" gradient.

"mini-batch" SGD for the square loss

Expectation is the gradient

Data: step sizes $\langle \eta^{(1)}, \ldots, \eta^{(K)} \rangle$
Result: parameter $\mathbf{w}$
initialize: $\mathbf{w}^{(0)} = \mathbf{0}$;
for $k \in \{1, \ldots, K\}$ **do**
  Sample m examples of (x, y) (uniformly at random) from the training set and let $\mathcal{M}$ be the set of these m points;
  $\mathbf{w}^{(k)} = \mathbf{w}^{(k-1)} + \eta^{(k)} \frac{1}{m} \sum_{(x,y)\in\mathcal{M}} (y - \mathbf{w}^{(k-1)} \cdot \mathbf{x})\, \mathbf{x}$; $+ \lambda w]$
end
return $\mathbf{w}^{(K)}$;

Algorithm 3: SGD

- the term in red is a lower variance, "sampled" gradient.
- how do we choose m?
  larger m means lower variance but more computation.
- Matrix algebra can make computing the term in red very fast!
  This is critical to get big performance bumps.

SGD Tips: stepsize

- Theory: If you turn down the step sizes at (some prescribed decaying method) then SGD will converge to the right answer.
  The "classical" theory doesn't provide enough practical guidance.
- Practice:
  - starting stepsize: start it "large":
    if it is "too large", then either you diverge (or nothing improves). set it a little less (like $1/4$) less than this point.
  - When do we decay it?
    When your training error stops decreasing "enough".
    OR based on a dev set.
- HW: you'll need to tune it a little. (a slow approach: sometimes you can just start it somewhat smaller than the "divergent" value and you will find something reasonable.)

SGD Tips: mini-batching

- Theory: there are diminishing returns to increasing m.
  - As you grow m, your "improvements" tend to diminish.
  - mini-batch size m "small": you can turn it up and you will find that you are making more progress per update.
  - mini-batch size m "large": you can turn it up and you will make roughly the same amount of progress (so your code will become slower).
- Practice: there are diminishing returns to increasing m.
- How do we set it?
  Easy: just keep cranking it up and eventually you'll see that your code doesn't get any faster.

Regularization/complexity control: Tips.

- Theory: really just says that λ controls your "model complexity".
  - we DO know that "early stopping" for GD/SGD is very similar to L2 regularization for us.
  - i.e. if we don't run for too long, then $\|\mathbf{w}\|^2$ won't become too big.
- Practice:
  - Exact methods (like matrix inverse/least squares): always need to regularize or something horrible happens....
  - GD/SGD: sometimes it works just fine ignoring regularization remember: early stopping is a form complexity control
  - Other times we have to tune it on some dev set.
    Fortunately, it is pretty robust to tune, by trying out different "orders of magnitude" guesses.

SGD + momentum

Binary classification $\to$ Multi-class classification

- suppose $y \in \{1, 2, \ldots k\}$.
- MNIST: we have $k = 10$ classes. How do we learn?
- Misclassification error:
  the fraction of times (often measured in %) in which our prediction of the label does not agree with the true label.
- Like binary classification, we do not optimize this directly
  it is often computationally difficult

Multi-class classification: "one vs all"

$\hat{y} = w \cdot x$

$\hat{y}^1 = w^1 \cdot x$

$\vdots$

$\hat{y}^{10} = w^{10} \cdot x$

HW

$\boxed{W \in R^{d \times 10}}$

train jointly

- Simplest method: consider each class separately.
- make 10 binary prediction problems:
- Build a separate model of $\Pr(y^{\text{class}}) = 1 | x, \mathbf{w}^{\text{class}})$.
- Example: build $k = 10$ separate linear regression models. HW3!

use pred. with highest score.

misclassification error: one perspective...

- directly using misclassification error is a poor objective function anyways:
  - NP-Hard
  - it only gives feedback of "correct" or "not"
  - even if you don't predict the true label (e.g. you make a mistake), there is a major difference between your model still "thinking" the true label is likely v.s. thinking the true label is "very unlikely".
- how do give our model better 'feedback'?
  - Seek provide probabilities of **all** outcomes
  - Then we reward/penalize our model based on its "confidence" of the correct answer...

A better probabilistic model: the soft max

$\sum_{\ell} Pr_w(y=\ell|x)=1$

- $y \in \{1, \ldots k\}$: Let's turn the probabilistic crank....
- The model: we have k weight vectors, $w^{(1)}, w^{(2)}, \ldots w^{(k)}$. For $\ell \in \{1, \ldots k\}$,

$$p(y=\ell|x, w^{(1)}, w^{(2)}, \ldots w^{(k)}) = \frac{\exp(w^{(\ell)} \cdot x)}{\sum_{i=1}^{k} \exp(w^{(i)} \cdot x)}$$

- It is "over-parameterized":

$$p_W(y=k|x) = 1 - \sum_{i=1}^{k-1} p_W(y=i|x)$$

- max. likelihood estimation is still a convex problem!

Aside: why might square loss be 'ok' for binary classification?

- Using the square loss for $y \in \{0, 1\}$?
  - it doesn't look like a great surrogate loss.
  - also, it doesn't look like a faithful probabilistic model:
- What is the "Bayes optimal" predictor for the square loss?
- The Bayes optimal predictor for the square loss with $y \in \{0, 1\}$:
- Can we utilize something more non-linear in our regression?

Can We Have Nonlinearity *and* Convexity?

	expressiveness	convexity
Linear classifiers	☹	☺
Neural networks	☺	☹

Can We Have Nonlinearity *and* Convexity?

	expressiveness	convexity
Linear classifiers	☹	☺
Neural networks	☺	☹

Kernel methods: a family of approaches that give us nonlinear decision boundaries without giving up convexity.

Let's try to build feature mappings

- Let $\phi(x)$ be a mapping from d-dimensional x to $\tilde{d}$-dimensional x.
- 2-dimensional example: quadratic interactions
- What do we call these quadratic terms for binary inputs?

Another example

- 2-dimensional example: bias$+$linear$+$quadratics interactions

- What do we call these quadratic terms for binary inputs?

The Kernel Trick

- Some learning algorithms, like the (lin. or logistic) regression, only need you to specify a way to take *inner products* between your feature vectors.
- A **kernel** function (implicitly) computes this inner product:

$$K(\mathbf{x}, \mathbf{v}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v})$$

  for some ϕ. Typically it is *cheap* to compute $K(\cdot, \cdot)$, and we never explicitly represent $\phi(\mathbf{v})$ for any vector $\mathbf{v}$.
- Let's see!