CSC311: Optimization for Machine Learning
Fall 2020

Based on Slides by Eleni Triantafillou, Ladislav Rampasek, Jake Snell, Kevin Swersky, Shenlong Wang, and others
Contents

- Overview
- Gradient Descent
- Convexity
Overview of Optimization
An informal definition of optimization

Minimize (or maximize) some quantity.
Applications

- Engineering: Minimize fuel consumption of an automobile
- Economics: Maximize returns on an investment
- Supply Chain Logistics: Minimize time taken to fulfill an order
- Life: Maximize happiness
Goal: find $\theta^* = \arg\min_{\theta} f(\theta)$, (possibly subject to constraints on $\theta$).

- $\theta \in \mathbb{R}^n$: *optimization variable*
- $f: \mathbb{R}^n \rightarrow \mathbb{R}$: *objective function*

Maximizing $f(\theta)$ is equivalent to minimizing $-f(\theta)$, so we can treat everything as a minimization problem.
Optimization is a large area of research

The best method for solving the optimization problem depends on which assumptions we want to make:

- Is $\theta$ discrete or continuous?
- What form do constraints on $\theta$ take? (if any)
- Is $f$ “well-behaved”? (linear, differentiable, convex, submodular, etc.)
Often in machine learning we are interested in learning the parameters $\theta$ of a model.

Goal: minimize some loss function

- For example, if we have some data $(x, y)$, we may want to maximize $P(y|x, \theta)$.
- Equivalently, we can minimize $-\log P(y|x, \theta)$.
- We can also minimize other sorts of loss functions

log can help for numerical reasons
Gradient Descent
Gradient Descent: Motivation

From calculus, we know that the minimum of $f$ must lie at a point where $\frac{\partial f(\theta^*)}{\partial \theta} = 0$.

- Sometimes, we can solve this equation analytically for $\theta$.
- Most of the time, we are not so lucky and must resort to iterative methods.

Review

- Gradient: $\nabla_\theta f = \left( \frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \ldots, \frac{\partial f}{\partial \theta_k} \right)$
Outline of Gradient Descent Algorithm

Where $\eta$ is the learning rate and $T$ is the number of iterations:

- Initialize $\theta_0$ randomly
- for $t = 1 : T$:
  - $\delta_t \leftarrow -\eta \nabla_{\theta_{t-1}} f$
  - $\theta_t \leftarrow \theta_{t-1} + \delta_t$

The learning rate shouldn’t be too big (objective function will blow up) or too small (will take a long time to converge)
Gradient Descent with Line-Search

Where $\eta$ is the learning rate and $T$ is the number of iterations:

- Initialize $\theta_0$ randomly
- for $t = 1 : T$:
  - Finding a step size $\eta_t$ such that $f(\theta_t - \eta_t \nabla \theta_{t-1}) < f(\theta_t)$
  - $\delta_t \leftarrow -\eta_t \nabla \theta_{t-1} f$
  - $\theta_t \leftarrow \theta_{t-1} + \delta_t$

Require a line-search step in each iteration.
Gradient Descent with Momentum

We can introduce a momentum coefficient \( \alpha \in [0, 1) \) so that the updates have “memory”:

- Initialize \( \theta_0 \) randomly
- Initialize \( \delta_0 \) to the zero vector
- for \( t = 1 : T \):
  - \( \delta_t \leftarrow -\eta \nabla \theta_{t-1} f + \alpha \delta_{t-1} \)
  - \( \theta_t \leftarrow \theta_{t-1} + \delta_t \)

Momentum is a nice trick that can help speed up convergence. Generally we choose \( \alpha \) between 0.8 and 0.95, but this is problem dependent
Outline of Gradient Descent Algorithm

Where $\eta$ is the learning rate and $T$ is the number of iterations:

- Initialize $\theta_0$ randomly
- Do:
  - $\delta_t \leftarrow -\eta \nabla_{\theta_{t-1}} f$
  - $\theta_t \leftarrow \theta_{t-1} + \delta_t$
- Until convergence

Setting a convergence criteria.
Some convergence criteria

- Change in objective function value is close to zero:
  \[ |f(\theta_{t+1}) - f(\theta_t)| < \epsilon \]
- Gradient norm is close to zero: \[ \|\nabla_{\theta} f\| < \epsilon \]
- Validation error starts to increase (this is called *early stopping*)
Checkgrad

- When implementing the gradient computation for machine learning models, it’s often difficult to know if our implementation of $f$ and $\nabla f$ is correct.
- We can use finite-differences approximation to the gradient to help:

$$
\frac{\partial f}{\partial \theta_i} \approx \frac{f((\theta_1, \ldots, \theta_i + \epsilon, \ldots, \theta_n)) - f((\theta_1, \ldots, \theta_i - \epsilon, \ldots, \theta_n))}{2\epsilon}
$$

Why don’t we always just use the finite differences approximation?

- slow: we need to recompute $f$ twice for each parameter in our model.
- numerical issues
Stochastic Gradient Descent

- Any iteration of a gradient descent (or quasi-Newton) method requires that we sum over the entire dataset to compute the gradient.
- SGD idea: at each iteration, sub-sample a small amount of data (even just 1 point can work) and use that to estimate the gradient.
- Each update is noisy, but very fast!
- It can be shown that this method produces an unbiased estimator of the true gradient.
- This is the basis of optimizing ML algorithms with huge datasets (e.g., recent deep learning).
- Computing gradients using the full dataset is called batch learning, using subsets of data is called mini-batch learning.
The reason SGD works is because similar data yields similar gradients, so if there is enough redundancy in the data, the noise from subsampling won’t be so bad.

SGD is very easy to implement compared to other methods, but the step sizes need to be tuned to different problems, whereas batch learning typically “just works”.

Tip 1: divide the log-likelihood estimate by the size of your mini-batches. This makes the learning rate invariant to mini-batch size.

Tip 2: subsample without replacement so that you visit each point on each pass through the dataset (this is known as an epoch).
Convexity
Definition of Convexity

A function $f$ is **convex** if for any two points $\theta_1$ and $\theta_2$ and any $t \in [0, 1]$,

$$f(t\theta_1 + (1 - t)\theta_2) \leq tf(\theta_1) + (1 - t)f(\theta_2)$$

We can *compose* convex functions such that the resulting function is also convex:

- If $f$ is convex, then so is $\alpha f$ for $\alpha \geq 0$
- If $f_1$ and $f_2$ are both convex, then so is $f_1 + f_2$
- etc., see [http://www.ee.ucla.edu/ee236b/lectures/functions.pdf](http://www.ee.ucla.edu/ee236b/lectures/functions.pdf) for more
Why do we care about convexity?

- Any local minimum is a global minimum.
- This makes optimization a lot easier because we don’t have to worry about getting stuck in a local minimum.
Examples of Convex Functions

Quadratics

In [6]:

```python
import matplotlib.pyplot as plt
plt.xkcd()
theta = linspace(-5, 5)
f = theta**2
plt.plot(theta, f)
```

Out[6]: [matplotlib.lines.Line2D at 0x3ceae90]
Examples of Convex Functions

Negative logarithms

```
In [8]:
import matplotlib.pyplot as plt
plt.xkcd()
theta = linspace(0.1, 5)
f = -np.log(theta)
plt.plot(theta, f)
```

Out[8]: [<matplotlib.lines.Line2D at 0x3ef4a10>]
Convexity for logistic regression

Cross-entropy objective function for logistic regression is also convex!

\[
f(\theta) = -\sum_n t^{(n)} \log p(y = 1|x^{(n)}, \theta) + (1 - t^{(n)}) \log p(y = 0|x^{(n)}, \theta)
\]
Plot of \(-\log \sigma(\theta)\)

In [15]:
```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

theta = linspace(-5, 5)
f = -np.log(sigmoid(theta))
plt.plot(theta, f)
```

Out[15]: [<matplotlib.lines.Line2D at 0x4c453d0>]

![Plot of the cross-entropy function](image)
More on optimization

- **Automatic Differentiation** Modern technique (used in libraries like tensorflow, pytorch, etc) to efficiently compute the gradients required for optimization. A survey of these techniques can be found here: https://arxiv.org/pdf/1502.05767.pdf

- **Convex Optimization** by Boyd & Vandenberghe Book available for free online at http://www.stanford.edu/~boyd/cvxbook/

- **Numerical Optimization** by Nocedal & Wright Electronic version available from UofT Library
Conditioning of Quadratic Losses
Consider a quadratic loss:

\[ \mathcal{L}(x) = x^T H x \]  

(1)

Consider the eigenvalues \( H \):

\[ \lambda_1, \ldots, \lambda_n \]  

(2)

If \( H \) symmetric, then the eigenvalues are real.

If \( H \) is positive definite, then every eigenvalue \( \lambda > 0 \) and our loss is strictly-convex.
\[ \mathcal{L}(x) = x^T H x \]

For positive definite \( H \) we can order the eigenvalues from smallest to largest

\[ 0 < \lambda_1 \leq \cdots \leq \lambda_n \] (3)

The ratio of largest to smallest eigenvalue characterizes difficulty of optimizing quadratic losses

\[ \kappa = \frac{\lambda_n}{\lambda_1} \] (4)
Gradient descent with an optimal step size converges with rate

\[
\frac{\kappa - 1}{\kappa + 1}
\]  

(5)

Gradient descent with momentum with optimal step size and momentum coefficient converges with rate:

\[
\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}
\]  

(6)

where (linear) rate of convergence means:

\[
\lim_{n \to \infty} \frac{\|x^{n+1} - x^*\|}{\|x^n - x^*\|}
\]  

(7)
We can calculate maximum/optimal step sizes given the eigenvalues of the matrix $H$ – See the lecture ipython notebook.

We can generalize this to $\mu$-strongly-convex, $L$-Lipschitz objectives.
Cross-Validation
Cross-Validation: Why Validate?

So far:

Learning as Optimization
Goal: Optimize model complexity (for the task) while minimizing under/overfitting

We want our model to **generalize well** without **overfitting**.

We can ensure this by **validating** the model.
Types of Validation

**Hold-Out Validation**: Split data into training and validation sets.

- Usually 30% as hold-out set.

Problems:
- Waste of dataset
- Estimation of error rate might be misleading
Types of Validation

- **Cross-Validation**: Random subsampling

  ![Diagram showing four runs of cross-validation](image)

  **Problem:**
  - More *computationally expensive* than hold-out validation.

Figure from Bishop, C.M. (2006). *Pattern Recognition and Machine Learning*. Springer
Variants of Cross-Validation

**Leave-p-out**: Use *p* examples as the validation set, and the rest as training; repeat for all configurations of examples.

e.g., for *p* = 1:

Problem:

- **Exhaustive**. We have to train and test \( \binom{N}{p} \) times, where *N* is the # of training examples.
Variants of Cross-Validation

**K-fold**: Partition training data into K equally sized subsamples. For each fold, use the other K-1 subsamples as training data with the last subsample as validation.
K-fold Cross-Validation

• Think of it like leave-$p$-out but without combinatoric amounts of training/testing.

Advantages:

• All observations are used for both training and validation. Each observation is used for validation exactly once.

• Non-exhaustive: More tractable than leave-$p$-out
K-fold Cross-Validation

Problems:

• **Expensive** for large $N, K$ (since we train/test $K$ models on $N$ examples).
  – But there are some efficient hacks to save time...

• Can still **overfit** if we validate too many models!
  – **Solution**: Hold out an additional test set before doing any model selection, and check that the best model performs well on this additional set (*nested cross-validation*).  =>  Cross-Validception
Practical Tips for Using K-fold Cross-Val

Q: How many folds do we need?
A: With larger $K$, ...

- Error estimation tends to be more accurate
- But, computation time will be greater

In practice:
- Usually use $K \approx 10$
- BUT, larger dataset => choose smaller $K$