How Learning Differs from Optimization

Sargur N. Srihari srihari@cedar.buffalo.edu

Topics in Optimization

- Optimization for Training Deep Models: Overview
- How learning differs from optimization
 - Risk, empirical risk and surrogate loss
 - Batch, minibatch, data shuffling
- Challenges in neural network optimization
- Basic Algorithms
- Parameter initialization strategies
- Algorithms with adaptive learning rates
- Approximate second-order methods
- Optimization strategies and meta-algorithms

Topics in Learning vs Optimization

- Learning vs Pure Optimization
- Empirical Risk Minimization
- Surrogate Loss Functions and Early Stopping
- Batch and Minibatch Algorithms

Learning vs Pure Optimization

- Optimization algorithms for deep learning differ from traditional optimization in several ways:
 - Machine learning acts indirectly
 - We care about some performance measure P defined wrt the training set which may be intractable
 - We reduce a different cost function $J(\theta)$ in the hope that doing so will reduce P
- Pure optimization: minimizing J is a goal in itself
- Optimizing algorithms for training Deep models:
 - Includes specialization on specific structure of ML objective function

Typical Cost Function

Typically the cost function can be written as an average over a training set

$$J(\boldsymbol{\theta}) = E_{(\boldsymbol{x},y) \sim \hat{p}_{\text{data}}} \left(L(f(\boldsymbol{x};\boldsymbol{\theta}),y) \right)$$

- Where
 - L is the per-example loss function
 - $f(x; \theta)$ is the predicted output when the input is x
 - \hat{p}_{data} is the empirical distribution
- In supervised learning y is target output

Typical Cost Function

- We consider the unregularized supervised case
 - where arguments of L are $f(x; \theta)$ and y
- Trivial to extend to cases:
 - Where parameters θ and input x are arguments or
 - Exclude output y as argument
 - For regularization or unsupervised learning

Objective wrt data generation is risk

Objective function wrt training set is

$$J(\theta) = E_{(x,y) \sim \hat{p}_{\text{data}}} \left(L(f(x;\theta),y) \right)$$
 L is the per-example loss function

 We would prefer to minimize the corresponding objective function where expectation is across the data generating distribution p_{data} rather than over finite training set

$$J^*(\boldsymbol{\theta}) = E_{(\boldsymbol{x},y) \sim p_{\text{data}}} \left(L(f(\boldsymbol{x};\boldsymbol{\theta}),y) \right)$$

- The goal of a machine learning algorithm is to reduce this expected generalization error
- This quantity is known as risk

Empirical Risk

• True risk is
$$J^*(\theta) = E_{(x,y) \sim p_{\text{data}}} (L(f(x;\theta),y))$$

- If we knew $p_{data}(x,y)$ it would be optimization solved by an optimization algorithm
- When we do not know $p_{data}(x,y)$ but only have a training set of samples, we have a machine learning problem
- Empirical risk, with m training examples, is

$$\left|J(\boldsymbol{\theta}) = E_{(\boldsymbol{x},y) \sim \hat{p}_{\text{data}}} \left(L(f(\boldsymbol{x};\boldsymbol{\theta}),y) \right) = \frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),y^{(i)}) \right|$$

Empirical Risk Minimization

Empirical risk, with m training examples, is

$$\left| J(\boldsymbol{\theta}) = E_{(\boldsymbol{x}, y) \sim \hat{p}_{\text{data}}} \left(L(f(\boldsymbol{x}; \boldsymbol{\theta}), y) \right) = \frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}) \right|$$

- Which is the average training error
- Still similar to straightforward optimization
- But empirical risk minimization is not very useful:
 - 1. Prone to overfitting: can simply memorize training set
 - 2.SGD is commonly used, but many useful loss functions have 0-1 loss, with no useful derivatives (derivative is either 0 or undefined everywhere)
- We must use a slightly different approach
 - Quantity we must optimize is even more different from what we truly want to optimize

Surrogate Loss: Log-likelihood

- Exactly minimizing 0-1 loss is typically intractable (exponential in the input dimension) even for a linear classifier
- In such situations use a surrogate loss function
 - Acts has a proxy but has advantages
- Negative log-likelihood of the correct class is a surrogate for 0-1 loss
 - It allows model to estimate conditional probability of classes given the input
 - If it does that well then can pick the classes that yield the least classification error in expectation

Surrogate may learn more

- Using log-likelihood surrogate,
 - Test set 0-1 loss continues to decrease for a long time after the training set 0-1 loss has reached zero when training
 - Because one can improve classifier robustness by further pushing the classes apart
 - Results in a more confident and robust classifier
 - Thus extracting more information from the training data than with minimizing 0-1 loss

Learning does not stop at minimum

- Important difference between Optimization in general and Optimization for Training:
 - Training does not halt at a local minimum
 - Early Stopping: Instead Learning algorithm halts on an early stopping criterion
 - Based on a true underlying loss function
 - Such as 0-1 loss measured on a validation set
 - Designed to cause algorithm to stop overfitting
- Often stops when derivatives are still large
 - In pure optimization, algorithm considered to converge when derivatives are very small

Decomposition: Batch Algorithms

- Objective function decomposes as a sum over training examples
 - This is unlike pure optimization
- Optimization for learning:
 - update the parameters based the expected value of cost using only a subset of the terms of of the full cost function

Srihari

Ex: Decomposition into a sum

- Maximum likelihood estimation problem
 - In log-space estimated parameter decomposes into a sum over each example

$$\left| \boldsymbol{\theta}_{\text{ML}} = \underset{\boldsymbol{\theta}}{\text{arg}} \max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log \, p_{\text{model}} \Big(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}; \boldsymbol{\theta} \Big) \right|$$

- Maximizing this sum is equivalent to maximizing the expectation over the empirical distribution defined by the training set $J(\theta) = E_{x,y \sim \hat{p}_{data}} \log p_{model} \big(x, y; \theta \big)$
- Commonly used property of $J(\theta)$ is its gradient

$$\boxed{ \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \!=\! E_{\boldsymbol{x}, \boldsymbol{y} \sim \hat{\boldsymbol{p}}_{data}} \nabla_{\boldsymbol{\theta}} \!\log p_{\text{model}} \! \left(\boldsymbol{x}, \! \boldsymbol{y}; \! \boldsymbol{\theta} \right) }$$

- Computing this expectation is very expensive
 - Requires summation over every training sample
- Solution: randomly sample small no. of samples

Quality of sampling-based estimate

Standard error for mean from n samples is



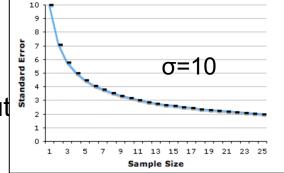
– where σ is std dev of samples

- Denominator shows that error decreases less than

linearly with no. of samples



- Computation increases by factor of 100 but
- Error decreases by only a factor of 10



- Optimization algorithms converge much faster
 - if allowed to rapidly compute approximate estimates
 - rather than slowly compute exact gradient

A motivation for sampling: Redundancy

- Training set may be redundant
 - Worst case: all m examples are identical
 - Sampling based estimate could use m times less computation
 - In practice
 - unlikely to find worst case situation but
 - likely to find large no. of examples that all make similar contribution to gradient

Batch gradient methods

- Batch or deterministic gradient methods:
 - Optimization methods that use all training samples in a large batch
- Somewhat confusing terminology
 - Batch also used to describe minibatch used by minibatch stochastic gradient descent
 - Batch gradient descent implies use of full training set
 - Batch size refers the size of a minibatch

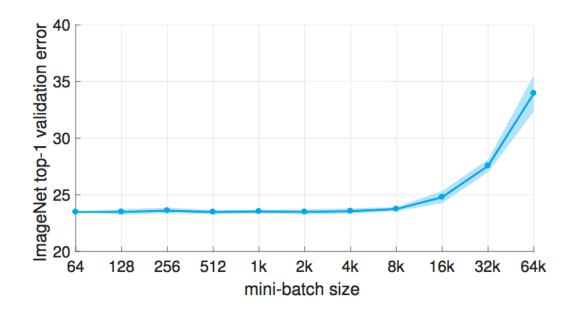
Stochastic or online methods

- Those using a single sample are called stochastic or on-line
 - On-line typically means continually created samples, rather than multiple passes over a fixed size training set
- Deep learning algorithms use more than 1 but fewer than all
- Traditionally called minibatch or minibatch stochastic or simply stochastic

Minibatch Size

- Driven by following factors
 - Larger batches > more accurate gradient but with less than linear returns
 - Multicore architectures are underutilized by extremely small batches
 - Use some minimum size below which there is no reduction in time to process a minibatch
 - If all examples processed in parallel, amount of memory scales with batch size
 - This is a limiting factor in batch size
 - GPU architectures more efficient with power of 2
 - Range from 32 to 256, sometimes with 16 for large models

Distributed synchronous SGD



ImageNet dataset large minibatches cause optimization difficulties, but when these are addressed the trained networks exhibit good generalization. Specifically, no loss of accuracy when training with large minibatch sizes up to 8192 images.

Regularizing effect of small batches

- Small batches offer regularizing effect due to noise added in process
- Generalization is best for batch size of 1
- Small batch sizes require small learning rate
 - To maintain stability due to high variance in estimate of gradient
- Total run time can be high
 - Due to reduced learning rate and
 - Requires more time to observe entire training set

Use of minibatch information

- Different algorithms use different information from the minibatch
 - Some algorithms more sensitive to sampling error
- Algorithms using gradient g are robust and can handle smaller batch sizes like 100
- Second order methods using Hessian H and compute updates such as H⁻¹g require much larger batch sizes like 10,000

Random selection of minibatches

- Crucial to select minibatches randomly
- Computing expected gradient from a set of samples requires that sample independence
- Many data sets are arranged with successive samples highly correlated
 - E.g., blood sample data set has five samples for each patient
- Necessary to shuffle the samples
 - For a data set with billions of samples shuffle once and store it in shuffled fashion

Simple random sampling

- Define the population. Say, training set has 10,000 examples
- Choose your batch size: say 100
- List the population and assign numbers to them
- Use a random number generator to generate a number in [1,1000]
- Select your sample

Parallelization of minibatches

- We can computer entire separate updates over different examples in parallel
 - Compute update that minimizes J(X) for one minibatch of examples X at the same time we compute update for several other minibatches
- Synchronous parallel distributed approaches discussed in Section 12.1.3

SGD and generalization error

Minibatch SGD follows the gradient of the true generalization error

$$\left| J^*(\boldsymbol{\theta}) = E_{(\boldsymbol{x},y) \sim p_{data}} \left(L(f(\boldsymbol{x};\boldsymbol{\theta}),y) \right) \right|$$

- As long as the examples are repeated
- Implementations of minibatch SGD
 - Shuffle once and pass through multiple no. of times
 - On the first pass: each minibatch computes unbiased estimate of true generalization error
 - Second pass: estimate is more biased because it is formed by resampling values already used rather than fair samples from data generating distribution

SGD minimizes generalization error

- Easiest to see equivalence in online learning
 - Examples/minibatches are drawn from a stream
 - Similar to living being who sees a new example at each instant
 - with each example (x,y) coming from data generating distribution $p_{\rm data}(x,y)$
 - » Examples never repeated, every example is a fair sample
- Equivalence is easy to derive when x and y are discrete
 - As seen next

Deep Learning

Discrete case with loss function

Generalization error (in terms of loss function)

$$J^*(\boldsymbol{\theta}) = E_{(x,y) \sim p_{data}} \left(L(f(x;\boldsymbol{\theta}),y) \right)$$

Which can be written as a sum

$$J^*(\boldsymbol{\theta}) = \sum_{x} \sum_{y} p_{data}(x,y) L(f(\boldsymbol{x};\boldsymbol{\theta}),y)$$

- with exact gradient

$$g = \nabla J^*(\boldsymbol{\theta}) = \sum_{x} \sum_{y} p_{data}(x, y) \nabla_{\theta} L(f(\boldsymbol{x}; \boldsymbol{\theta}), y)$$

Implies that derivative can be computed in batches

 We have already seen this (decomposition) demonstrated for log-likelihood in

$$J(\boldsymbol{\theta}) = E_{x, y \sim \hat{p}_{data}} \log p_{\text{mod}el}(x, y; \boldsymbol{\theta}) \quad \text{and} \quad \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = E_{x, y \sim \hat{p}_{data}} \nabla_{\boldsymbol{\theta}} \log p_{\text{mod}el}(x, y; \boldsymbol{\theta})$$

- Thus it holds for functions other than likelihood
- Similar result for when x and y are continuous

Use of multiple epochs

- SGD minimizes generalization error when samples are not reused
 - Yet best to make several passes through the training set
 - Unless training set is extremely large
- With multiple epochs, first epoch follows unbiased gradient of generalization error
- Additional epochs provide enough benefit to decrease training error
 - Although increasing gap between training and testing error

Impact of growing data sets

- Data sets are growing more rapidly than computing power
- More common to use each training example only once
 - Or even make an incomplete pass through the data set
- With a large training set overfit is not an issue
 - Underfitting and computational efficiency become predominant concerns