EECS 230 Deep Learning
Lecture 7: Optimization

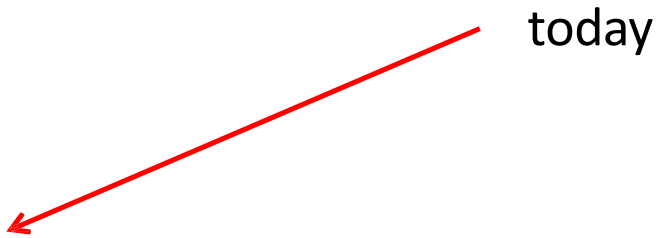
Some slides from C. Lee Giles, Sargur Srihari and Ankur Mali

Topics in Optimization

- Role of Optimization in Deep Learning
- Basic Algorithms
- Algorithms with adaptive learning rates
- Optimization strategies and meta-algorithms

Optimization is essential for DL

- Deep Learning is an instance of a recipe:
  1. Specification of a dataset
  2. A cost function
  3. A model
  4. An optimization procedure

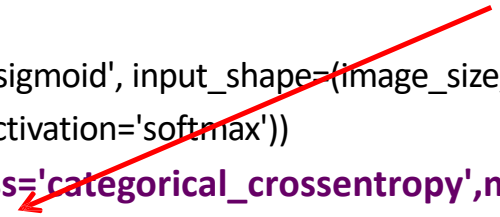


Our focus is on one case of optimization

- Find parameters $\boldsymbol{\theta}$ of a neural network that significantly reduces a cost function $J(\boldsymbol{\theta})$
  - It typically includes:
    - a performance measure evaluated on an entire training set as well as an additional regularization term

Keras for MNIST Neural Network

- # Neural Network
- import keras
- from keras.datasets import mnist
- from keras.layers import Dense
- from keras.models import Sequential
- (x_train, y_train), (x_test, y_test) = mnist.load_data()
- num_classes=10
- image_vector_size=28*28
- x_train = x_train.reshape(x_train.shape[0], image_vector_size)
- x_test = x_test.reshape(x_test.shape[0], image_vector_size)
- y_train = keras.utils.to_categorical(y_train, num_classes)
- y_test = keras.utils.to_categorical(y_test, num_classes)
- image_size = 784 model = Sequential()
- model.add(Dense(units=32, activation='sigmoid', input_shape=(image_size,)))
- model.add(Dense(units=num_classes, activation='softmax'))
- **model.compile(optimizer='sgd', loss='categorical_crossentropy',metrics=['accuracy'])**
- history = model.fit(x_train, y_train, batch_size=128, epochs=10, verbose=False,validation_split=.1)
- loss,accuracy = model.evaluate(x_test, y_test, verbose=False)



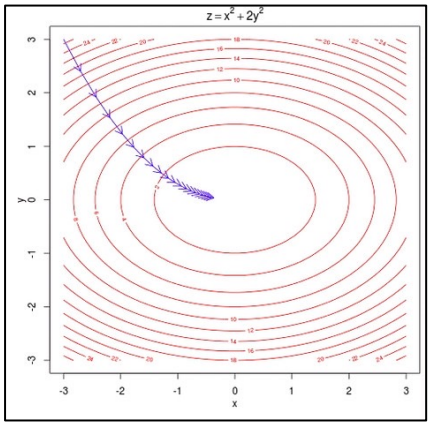
UCMERCED

Summary of Optimization Methods

- Movies:

http://hduongtrong.github.io/2015/11/23/coordinate-descent/

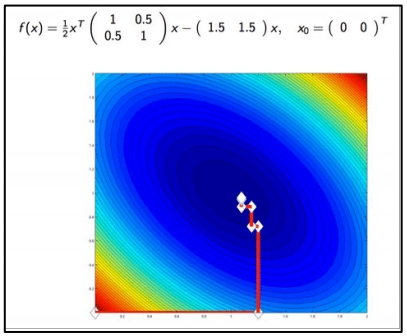
Gradient Descent



$$g = \frac{1}{M} \nabla_{\vartheta} \sum_{i=1}^{M} L\left(\boldsymbol{x}^{(i)}, y^{(i)}, \vartheta\right)$$

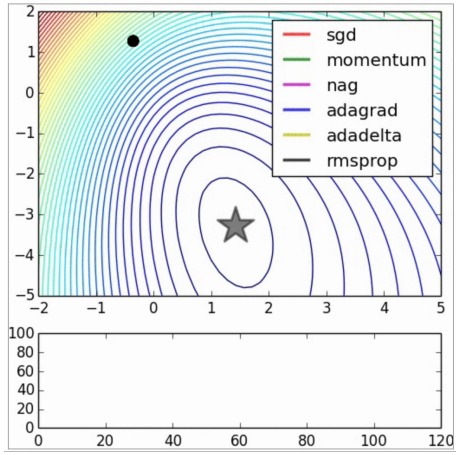
$$\vartheta \leftarrow \vartheta - \varepsilon g$$

Coordinate Descent



Minimize $f(\boldsymbol{x})$ wrt a single variable, x_i, then wrt x_j etc

SGD



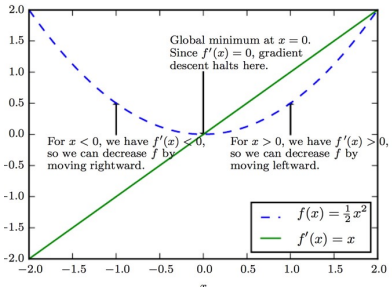
$$g = \frac{1}{m'} \nabla_{\vartheta} \sum_{i=1}^{m'} L\left(\boldsymbol{x}^{(i)}, y^{(i)}, \vartheta\right)$$

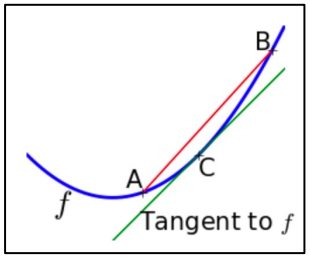
$$\vartheta \leftarrow \vartheta - \varepsilon g$$

UCMERCED

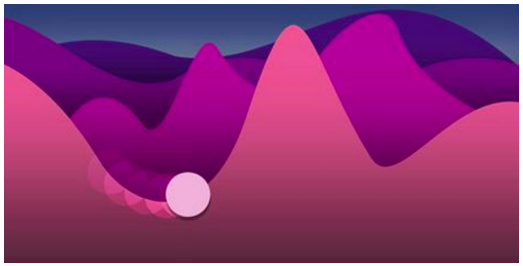
Optimization Problem in DL

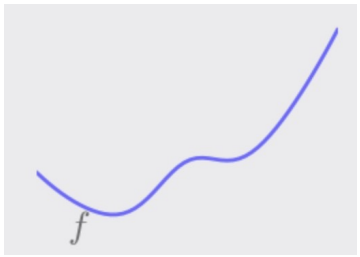
- Optimization is an extremely difficult task for DL
  - Traditional ML: careful design of objective function and constraints to ensure convex optimization





  - When training neural networks, we must confront nonconvex cases





UCMERCED

Batch Gradient Methods

- Batch or deterministic gradient methods:
  - Optimization methods that use all training samples are batch or deterministic methods

- *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 drawn from a stream rather than multiple passes over a fixed size training set
- Deep learning algorithms usually use more than one but fewer than all samples
  - Methods traditionally called minibatch or minibatch stochastic now simply called stochastic

Ex: (stochastic gradient descent - SGD)

Minibatch Size

- Driven by following:
  - Larger batches→more accurate gradient
  - 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 sizes power of 2
    - Range from 32 to 256, sometimes with 16 for large models

Regularizing Effect of Small Batches

- Small batches offer regularization due to noise added in the process
- Generalization is best for batch size of one
- Small batch sizes require a small learning rate
  - Maintains stability due to high variance in estimate of gradient
- Total run time can be high
  - Due to reduced learning rate that requires more time to observe entire training set

Random Selection of Minibatches

- Crucial to select minibatches randomly for an unbiased estimate
- Computing expected gradient from a set of samples requires 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 in shuffled fashion

Example of Simple Random Sampling

- Define the population
- Let the training set have 10,000 examples
- Choose 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 and shuffle
  - Failing to shuffle can seriously impact the training

SGD and Generalization Error

- Minibatch SGD follows the gradient of the true generalization error

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

  - as long as the examples are repeated
- Implementations of minibatch SGD
  - Shuffle once and pass through multiple number of times

Topics

- Importance of Optimization in machine learning
- Basic Optimization Algorithms
  - SGD, Momentum, Nesterov Momentum
- Algorithms with adaptive learning rates
  - AdaGrad, RMSProp, Adam

SGD Follows Gradient Estimate Downhill

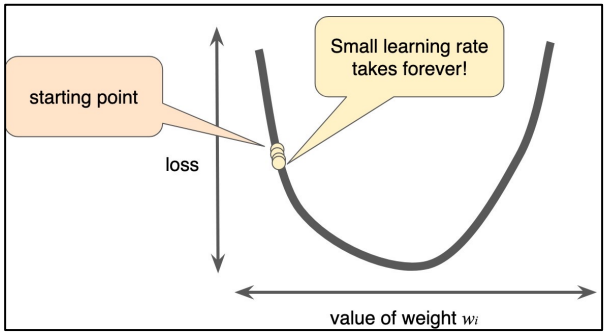
Algorithm: SGD update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

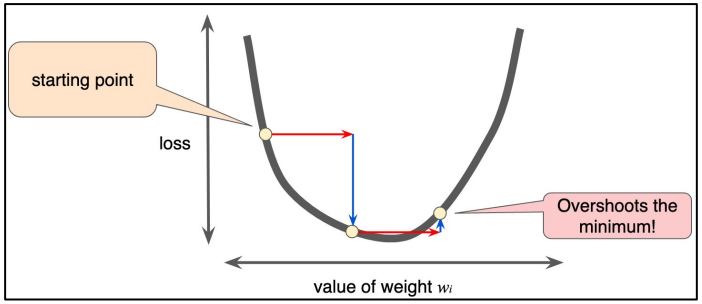
A crucial parameter is the learning rate ε

At iteration k it is ε_k

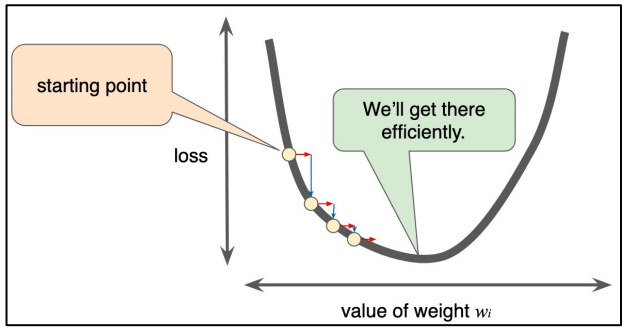
Choice of Learning Rate



Too small learning rate will take too long



Too large, the next point will perpetually bounce haphazardly across the bottom of the well



If gradient is small, then can safely try a larger learning rate, which compensates for the small gradient and results in a larger step size

https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate

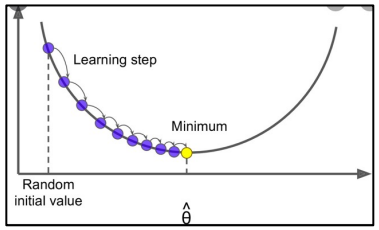
Need for Decreasing Learning Rate

- True gradient of total cost function
  - Becomes small and then 0
    - One can use a fixed learning rate
- But SGD has a source of noise
  - Random sampling of m training samples
    - Gradient does not vanish even when arriving at a minimum
  - Common to decay learning rate linearly until iteration τ: $\varepsilon_k=(1-\alpha)\varepsilon_0+\alpha\varepsilon_\tau$ with $\alpha=k/\tau$
  - After iteration τ, it is common to leave ε constant
    - Often a small positive value in the range 0.0 to 1.0

Learning Rate Decay

- Decay learning rate

  τ: $\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau$ with $\alpha = k/\tau$



- Learning rate is calculated at each update
  - (e.g. end of each mini-batch) as follows:

```
lrate = initial_lrate * (1 / (1 + decay * iteration))
```

  - Where *lrate* is learning rate for current epoch
  - *initial_lrate* is specified as an argument to SGD
  - *decay* is the decay rate which is greater than zero and
  - *iteration* is the current update number

```
from keras.optimizers import SGD
...
opt = SGD(lr=0.01, momentum=0.9, decay=0.01)
model.compile(..., optimizer=opt)
```

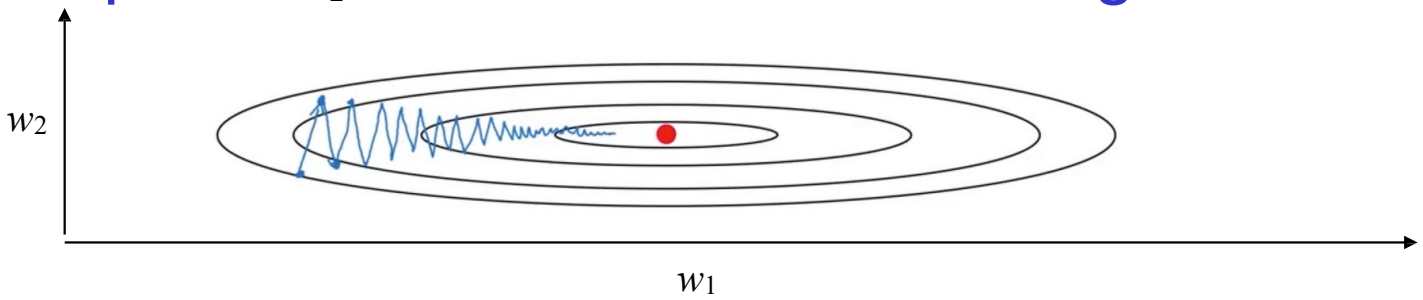
UCMERCED

Momentum Method

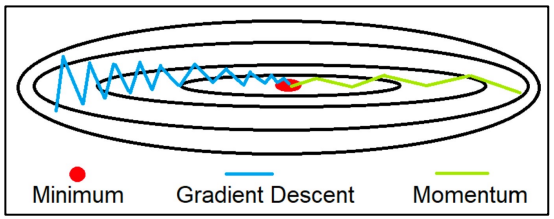
- SGD is a popular optimization strategy but it can be slow
- Momentum method accelerates learning, when:
  - Facing high curvature
  - Noisy gradients
- It works by accumulating the moving average of past gradients and moves in that direction while exponentially decaying

Gradient Descent with Momentum

- Gradient descent with momentum converges faster than standard gradient descent
- Taking large steps in w_2 direction and small steps in w_1 direction slows down algorithm



- Momentum reduces oscillation in w_2 direction



- Now can set a higher learning rate

https://www.andreaperlato.com/aipost/gradient-descent-with-momentum/

Momentum Definition

- Introduce velocity variable $\boldsymbol{v}$
- This is the direction and speed at which parameters move through parameter space
- Name momentum comes from physics & is mass times velocity
  - The momentum algorithm assumes unit mass
- A hyperparameter $\alpha \in [0,1)$ determines exponential decay of $\boldsymbol{v}$

Momentum Update Rule

- The update rule is given by

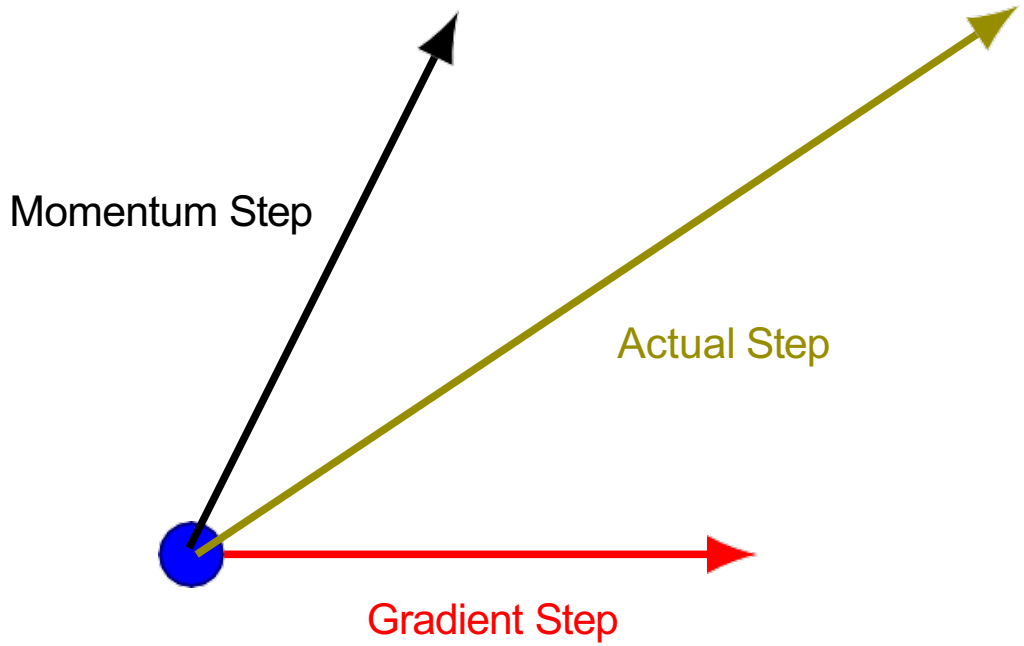
$$v \leftarrow \alpha v - \varepsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\left(f(x^{(i)}; \theta), y^{(i)} \right) \right)$$
$$\theta \leftarrow \theta + v$$

- The velocity $\boldsymbol{v}$ accumulates the gradient

$$\nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\left(f(x^{(i)}; \theta), y^{(i)} \right) \right)$$

  elements
- The larger α is relative to ε, the more previous gradients affect the current direction
- The SGD algorithm with momentum is next

Momentum



SGD Algorithm with Momentum

Algorithm: SGD with momentum

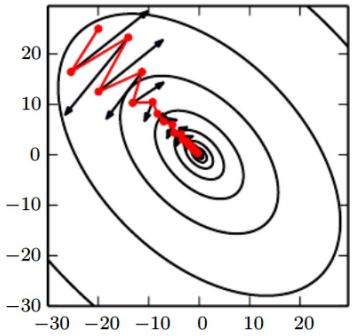
Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Keras: The learning rate can be specified via the *lr* argument and the momentum can be specified via the *momentum* argument.

```
from keras.optimizers import SGD
...
opt = SGD(lr=0.01, momentum=0.9)
model.compile(..., optimizer=opt)
```

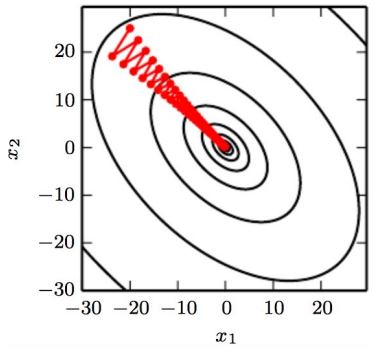
Momentum

- SGD with momentum



Contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix.
Red path cutting across the contours depicts path followed by momentum learning rule as it minimizes this function

- Comparison to SGD without momentum



At each step we show path that would be taken by SGD at that step
Poorly conditioned quadratic objective
Looks like a long narrow valley with steep sides
Wastes time

Nesterov Momentum

- A variant to accelerate gradient, with update

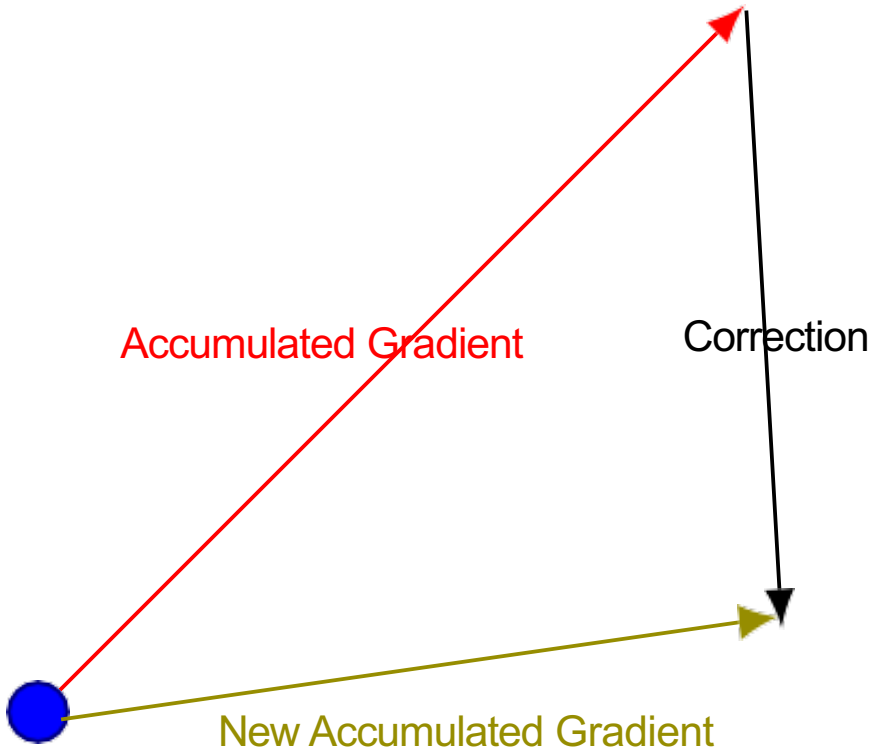
$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)} \right) \right],$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v},$$

  - where parameters α and ε play a similar role as in the standard momentum method

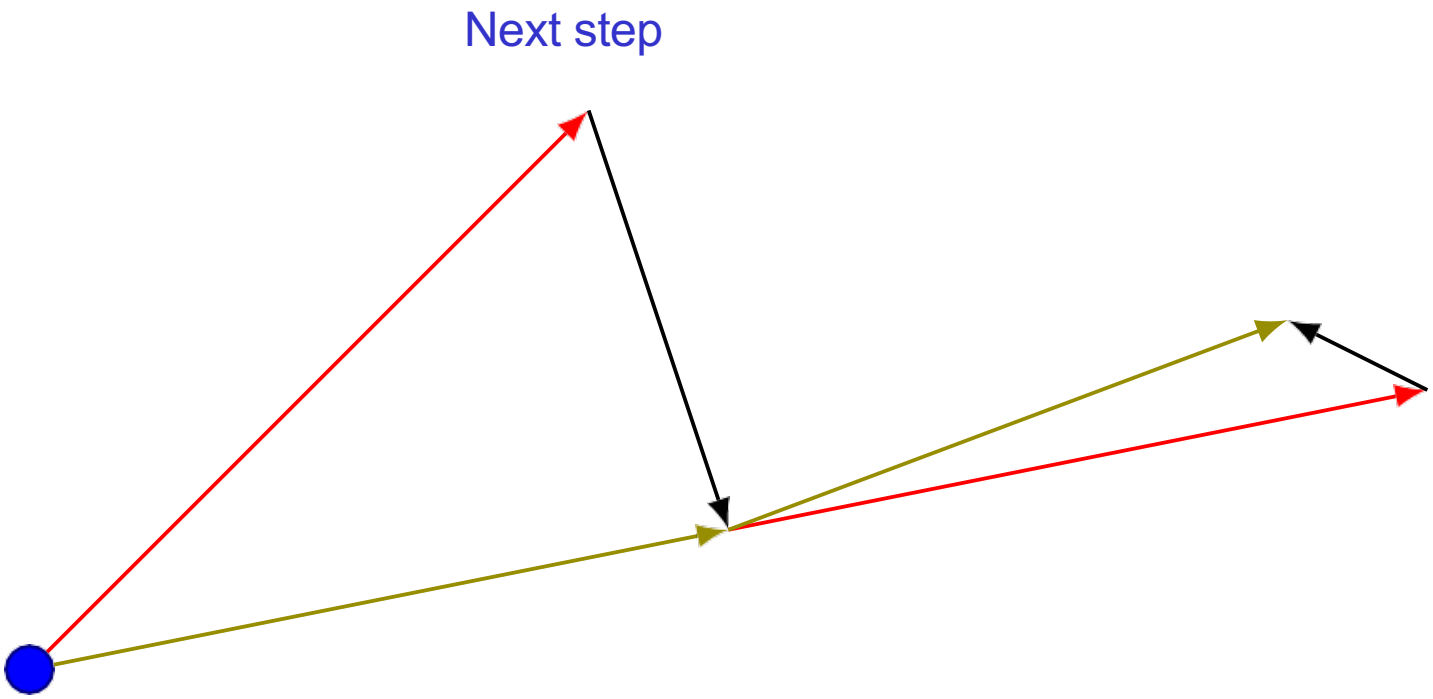
– Difference between Nesterov and standard momentum is where gradient is evaluated.
  - Nesterov gradient is evaluated after the current velocity is applied.
  - One can interpret Nesterov as attempting to add a correction factor to the standard method of momentum

Nesterov Momentum

- First take a step in the direction of the accumulated gradient
- Then calculate the gradient and make a correction



Nesterov Momentum



SGD with Nesterov Momentum

- A variant of the momentum algorithm
  - Nesterov's accelerated gradient method
- Applies a correction factor to standard method

Algorithm: SGD with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

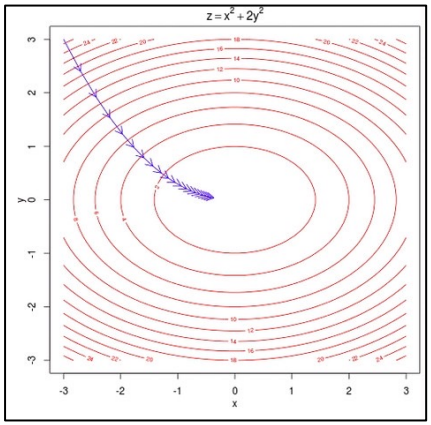
This line is added from plain momentum

Summary of Optimization Methods

- Movies:

http://hduongtrong.github.io/2015/11/23/coordinate-descent/

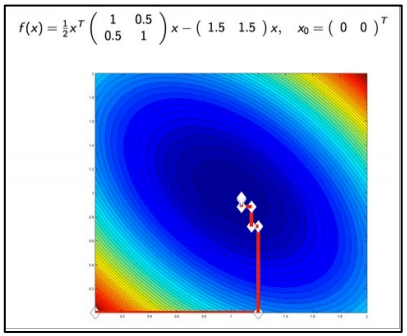
Gradient Descent



$$g = \frac{1}{M} \nabla_{\vartheta} \sum_{i=1}^{M} L\left(\boldsymbol{x}^{(i)}, y^{(i)}, \vartheta\right)$$

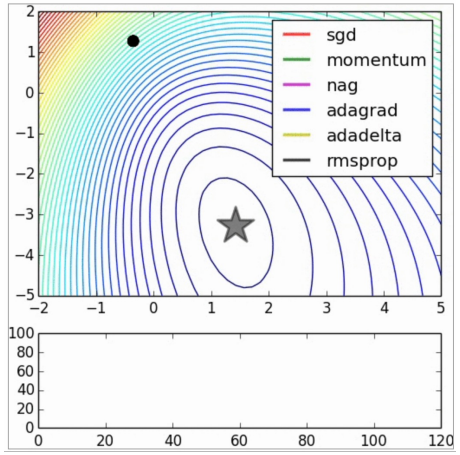
$$\vartheta \leftarrow \vartheta - \varepsilon g$$

Coordinate Descent



Minimize $f(\boldsymbol{x})$ wrt a single variable, x_i, then wrt x_j etc

SGD



$$g = \frac{1}{m'} \nabla_{\vartheta} \sum_{i=1}^{m'} L\left(\boldsymbol{x}^{(i)}, y^{(i)}, \vartheta\right)$$

$$\vartheta \leftarrow \vartheta - \varepsilon g$$

UCMERCED

Topics in Optimization

- Role of Optimization in Deep Learning
- Basic Algorithms
- Algorithms with adaptive learning rates
  1. AdaGrad
  2. RMSProp
  3. Adam
  4. Choosing the right optimization algorithm
- Optimization strategies and meta-algorithms

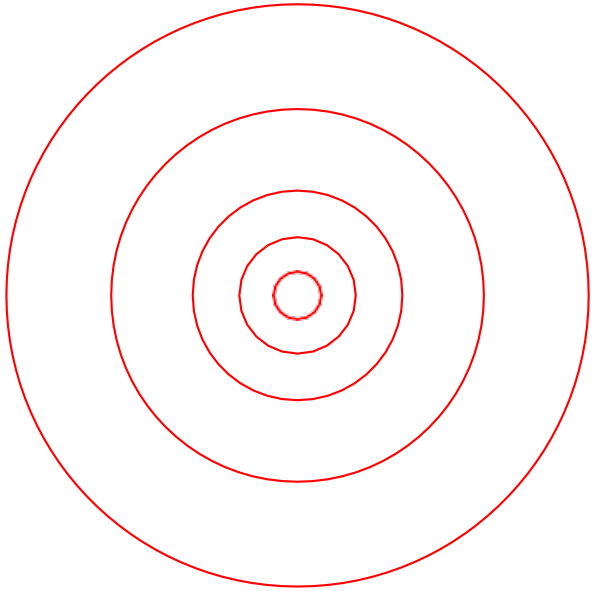
Importance of Learning Rate

- Learning rate is the most difficult hyperparameter to set
  - It significantly affects model performance
- Cost is highly sensitive to some directions in parameter space and insensitive to others
  - Momentum helps but introduces another hyperparameter
  - Other approach
    - If direction of sensitivity is axis aligned, have a separate learning rate for each parameter and adjust them throughput learning

Heuristic Approaches

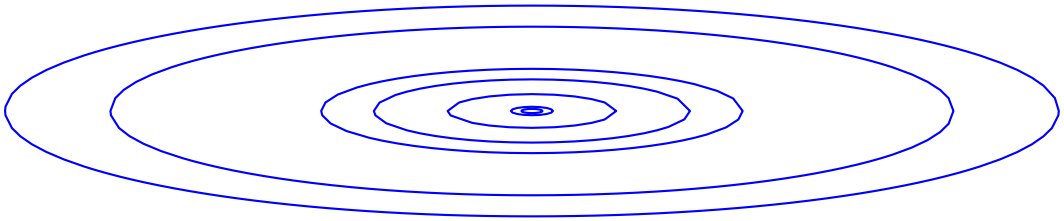
- Delta-bar-delta Algorithm (1988)
  - Applicable to only full batch optimization
    - If partial derivative of the loss wrt to a parameter remains the same sign, the learning rate should increase
    - If the partial derivative changes sign, the learning rate should decrease
- Recent Incremental mini-batch methods
  - Adapt learning rates of model parameters
    1. AdaGrad
    2. RMSProp
    3. Adam

Motivation



Nice (all features are equally important)

Motivation



Harder

AdaGrad

- Individually adapts learning rates of all parameters
  - Scale them inversely proportional to the sum of the historical squared values of the gradient
- The AdaGrad Algorithm:

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Performs well for some but not all deep learning

RMSProp

- AdaGrad is good when the objective is convex.
- AdaGrad might shrink the learning rate too aggressively, we want to keep the history in mind
- We can adapt it to perform better in non-convex settings by accumulating an exponentially decaying average of the gradient

RMSProp

- Modifies AdaGrad for a nonconvex setting
  - Changes gradient accumulation into a exponentially weighted moving average
  - Converges rapidly when applied to a convex function

The RMSProp Algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

UCMERCED

RMSProp Combined with Nesterov

Algorithm: RMSProp with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
Initialize accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

RMSProp is Popular

- RMSProp is an effective practical optimization algorithm
- Common optimization method for deep learning practitioners

Adam: Adaptive Moments

- Another adaptive learning rate optimization algorithm
- Variant of RMSProp with momentum
- Generally robust to the choice of hyperparameters

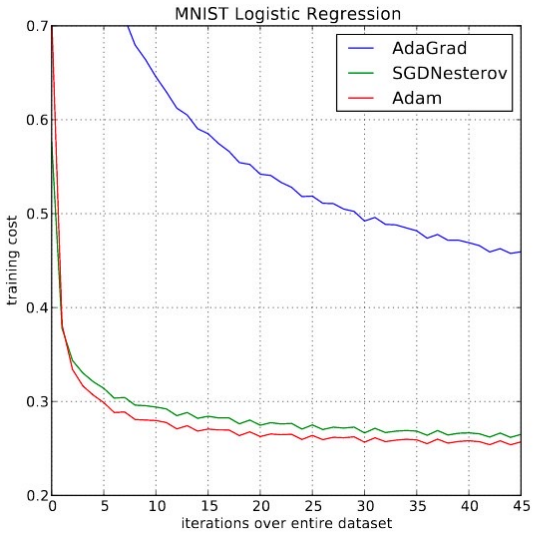
Adam Optimizer

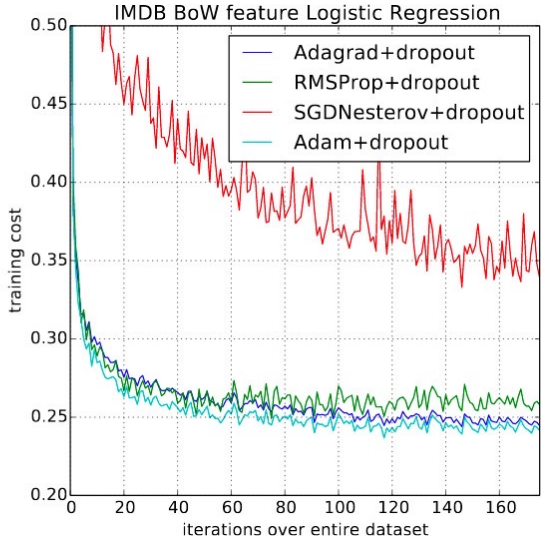
Adam Algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

UCMERCED

Performance on Multilayer NN

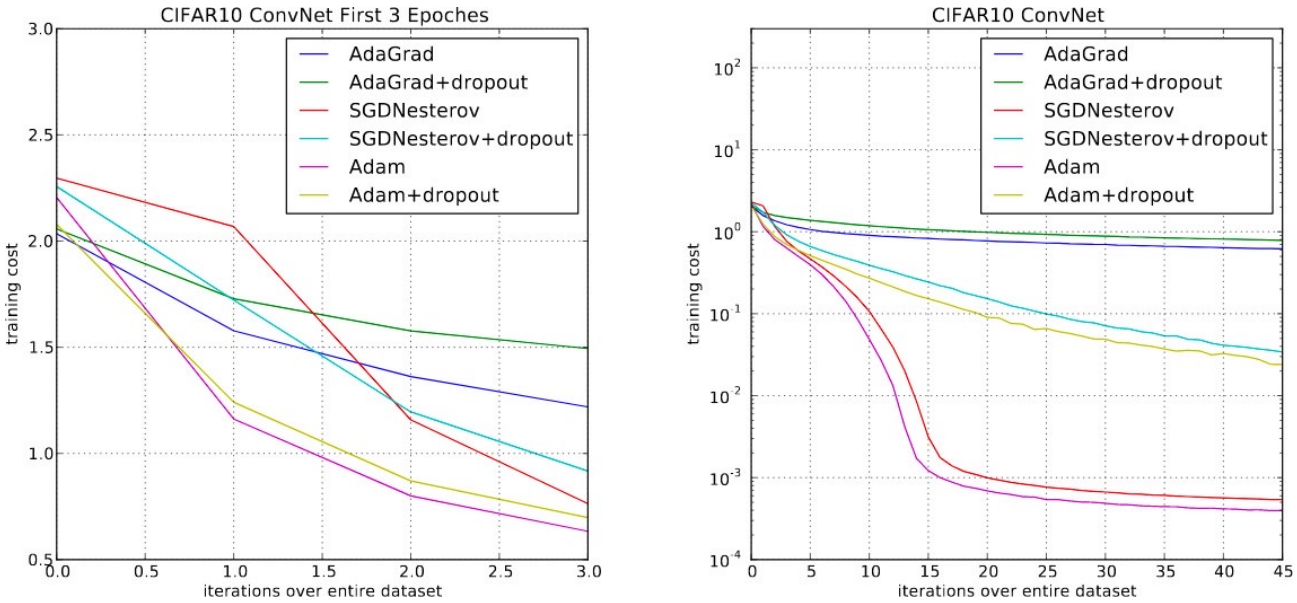




Training of multilayer neural networks on MNIST images. (a) Neural networks using dropout stochastic regularization. (b) Neural networks with deterministic cost function.

UCMERCED

Performance with CNN



Convolutional neural networks training cost.
(left) Training cost for the first three epochs.
(right) Training cost over 45 epochs.
CIFAR-10 with c64-c64-c128-1000 architecture.

UCMERCED

Choosing the Right Optimizer

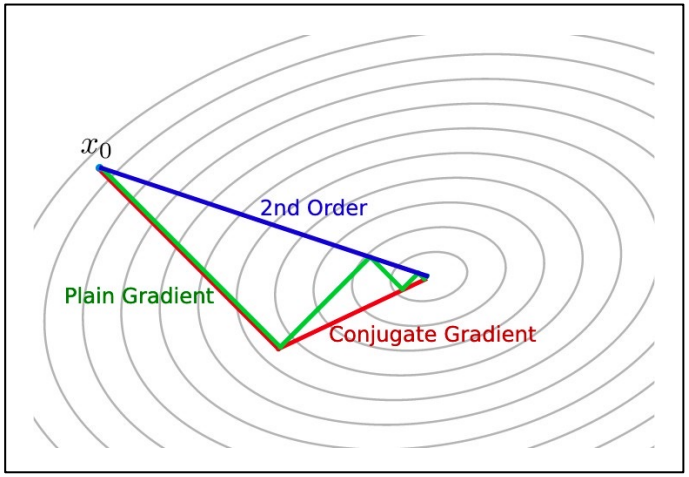
- We have discussed several methods of optimizing deep models by adapting the learning rate for each model parameter
- Which algorithm to choose?
  - No consensus
- Most popular algorithms actively in use:
  - SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta and Adam
  - Choice depends on user's familiarity with algorithm

Topics in Second Order Methods

1. Overview
2. Newton's Method
3. Other: Conjugate Gradients
   - Nonlinear Conjugate Gradients
4. Other: BFGS
   - Limited Memory BFGS

Why Second-order Methods?

- Better direction



- Better step-size
  - A full step jumps directly to the minimum of the local squared approx.
  - often a good heuristic
  - additional step size reduction and dampening are straight-forward

Issues with Second-order Methods

- Computational expensive, even for approximation methods
- Rarely used for large scale problems

- Methods include:
  - Newton's method
  - Conjugate gradients
  - Nonlinear conjugate gradients
  - Broyden-Fletcher-Goldfar-Shanno (BFGS)
  - Limited Memory BFGS

- Major issue is size of the Hessian

Overview

- Second order methods for training deep networks
- Objective function examined is empirical risk:
  - Empirical risk, with m training examples, is

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

$f(\boldsymbol{x};\boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$
y is target output
L is the per-example loss function

  - Methods extend readily to other objective functions such as those that include parameter regularization

Newton's Method

- In contrast to first order gradient methods, second order methods make use of second derivatives to improve optimization
- Most widely used second order method is Newton's method
- It is described in more detail here emphasizing neural network training
- Based on Taylor's series expansion to approximate $J(\theta)$ near some point θ_0 ignoring derivatives of higher order

Newton Update Rule

- Taylor's series to approximate $J(\theta)$ near θ_0

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta)_0 + \frac{1}{2}(\theta - \theta_0)^T H (\theta - \theta_0)$$

  - where H is the Hessian of J wrt θ evaluated at θ_0

- Solving for the critical point of this function we obtain the Newton parameter update rule

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0)$$

  - Thus for a quadratic function (with positive definite H) by rescaling the gradient by H^{-1} Newton's method directly jumps to the minimum
  - If objective function is convex but not quadratic (there are higher-order terms) this update can be iterated yielding the training algorithm given next

Training Algorithm associated with Newton's Method

Algorithm: Newton's method with objective:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$$

Require: Initial parameter $\boldsymbol{\theta}_0$

Require: Training set of m examples

while stopping criterion not met **do**

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m} \nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Compute Hessian inverse: $\boldsymbol{H}^{-1}$

Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$

Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

Positive Definite Hessian

- For surfaces that are not quadratic, as long as the Hessian remains positive definite, Newton's method can be applied iteratively
- This implies a two-step procedure:
  - First update or compute the inverse Hessian (by updating the qudratic approximation)
  - Second, update the parameters according to

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0)$$

Regularizing the Hessian

- Newton's method is appropriate only when the Hessian is positive definite
  - In deep learning the surface of the objective function is nonconvex
    - Many saddle points: problematic for Newton's method
- Can be avoided by regularizing the Hessian
  - Adding a constant α along the Hessian diagonal

$$\theta^* = \theta_0 - [H(f(\theta_0)) + \alpha I]^{-1} \nabla_\theta f(\theta_0)$$

Topics in Optimization

- Role of Optimization in Deep Learning
- Basic Algorithms
- Algorithms with adaptive learning rates
- Approximate second-order methods