MET CS 777 - Big Data Analysis

Module 3:
Optimization Methods - Gradient Descent

Dimitar Trajanov

Table of contents

Approaches to Learning a Model

There many different ones, including:

- Optimization based (For example: least squares)
- Probabilistic: Maximum Likelihood Estimation (MLE)
- Probabilistic: Bayesian

Optimization based

Optimization-Based

Goal is to reduce some error metric on example/training data.

There is no direct probabilistic motivation.

Example: Least Squares Regression

Example: I observe {18, 22, 45, 49, 86} ... predict next item?.

You can consider this as tuples of
{(1, 18), (2, 22), (3, 45), (4, 49), (5, 86) } ...

- ▷ Might fit a line to the data
- ▷ So $f(t) = m \times t$

We need to choose m

- ▷ Might choose least-squares fit
- ▷ For example, choose m to min $l(m) = \sum_i (f(t_i) - x_i)^2$
- ▷ $l(m)$ often referred to as a **"Loss function"**

Computing Least-Squares Fit

- This loss function is **"Convex"**
- So just choose unique m where $l'(m) = 0$

$$
\begin{aligned}
l(m) &= \sum_i (f(t_i) - x_i)^2 \\
&= \sum_i (m \times t - x_i)^2 \\
l'(m) &= \sum_i 2(m \times t_i^2 - x_i \times t_i) \\
&= 2m(1 + 4 + 9 + 16 + 25) - 2(18 + 44 + 135 + 196 + 430) \\
&= 110m - 1646 = 0
\end{aligned}
$$

So loss minimized at m = 14.96 ; Next value should be 89.8

Other Loss Functions

View the list of prediction errors $(f(t_i) - x_i)$ as a vector

Can have many loss functions, corresponding to norms

Given a vector of errors $\langle \epsilon_1, \epsilon_2, ..., \epsilon_n, \rangle$, l_p norm defined as:

$$\left(\sum_{i=1}^{n} |\epsilon_i| \right)^{\frac{1}{p}}$$

Common loss functions correspond to various norms:

- l_1 corresponds to mean absolute error
- l_2 to mean squared error/least squares
- l_∞ corresponds to minimax

Optimization

At the center of all model "learning" frameworks is optimization!

Why?

- ▷ It is more explicit in case of a cost function
- ▷ Implicitly in case of Bayesian

Solving optimization problems is a fundamental question in data science.

Desired Properties for Optimization Framework

To be useful for data science, optimization framework should be

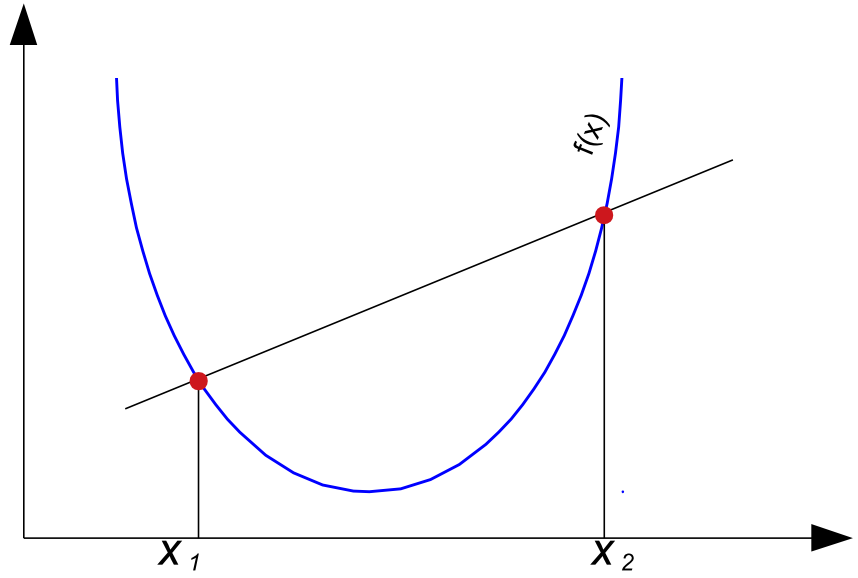
- **easy to apply** to many types of optimization problems,
- **scalable** and easy to implemented in map reduce paradigm
- **fast**

Some Terminology

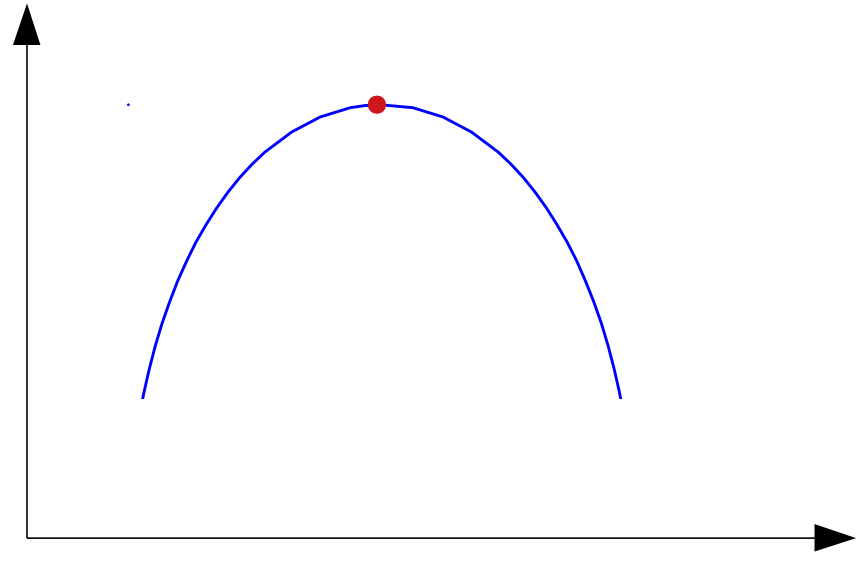
Convex Function

A function $f(x)$ is defined to be convex when

$$\forall x_1, x_2 \in X,\ \forall t \in [0, 1] : f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$



Convex Function



Concave Function

- For maximum of a Concave function, we calculate minimum of $-f(x)$

None Convex Functions - Global and Local Minimum

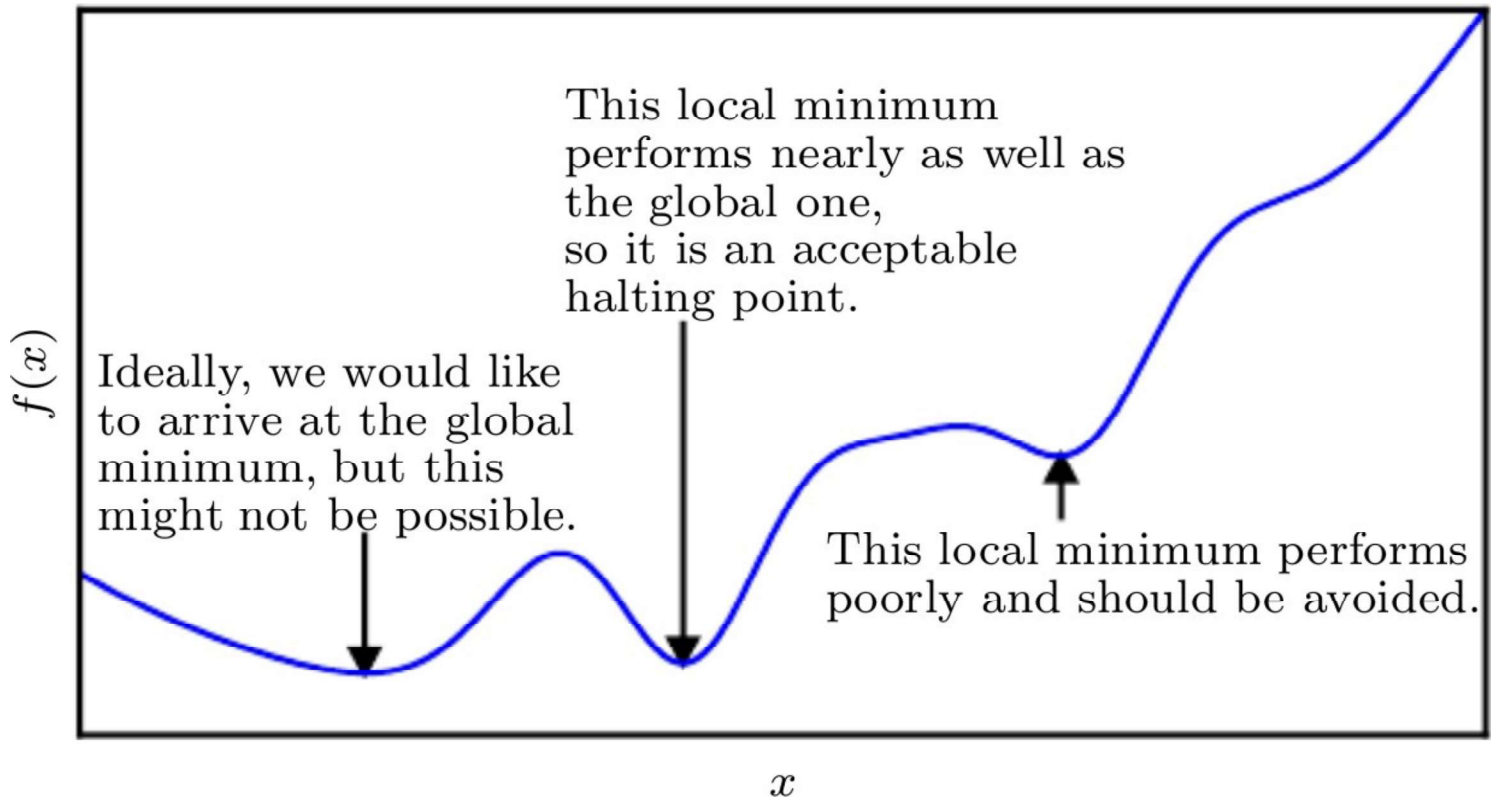


Image from Book: Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning - The MIT Press (2016)

Example - Doing Regression
Computation can be expensive

Example - Doing Regression

We have a set of training data like

- Data Matrix of $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, with Labels $\boldsymbol{Y} \in \mathbb{R}^{n \times 1}$
- Linear Regression Model :

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_d x_d + e$$

The same in matrix form: $Y_{n \times 1} = X_{n \times d} \Theta_{d \times 1} + e_{n \times 1}$

- We are looking for a vector of parameters (set of weights) $\Theta \in \mathbb{R}^{d \times 1}$ to minimize

$$MSE = L(\Theta) = \frac{1}{N} e^T e = \frac{1}{N} (Y^T Y - 2\Theta^T X^T Y + \Theta^T X^T X \Theta)$$

Finding exact solution:

This function is a **convex function**.

- Take the derivative and set it to zero

$$\frac{d}{d\Theta} L(\Theta) = -2X^T (Y - X\Theta) = 0$$

We set the above to zero and will get

$$\Theta = (X^T X)^{-1} X^T Y$$

Doing Regression - Computation Costs

In computing exact solution:

$$\Theta = (X^T X)^{-1} X^T Y$$

We know that n **number of data rows can be large**

- **If dimension d is small**, then $X^T X \in \mathbb{R}^{d \times d}$ and $X^T Y \in \mathbb{R}^{d \times 1}$ are in good size to compute in parallel on a single machine.

Nice! This can be one line code in R Program
solve(t(X) %*% X) %*% t(X) %*% y.

But what if the dimension d is large?

We would need a large size of memory for computing matrix operations (Matrix-Matrix, Matrix-Vector).

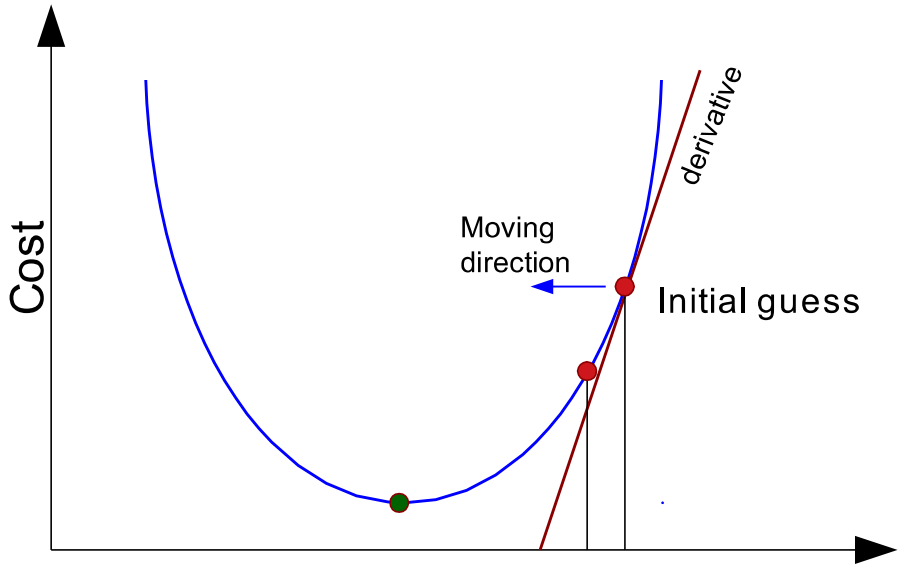
Gradient Descent

Gradient Descent

Most widely used optimization framework for - at least - **"Big Data" science** is **gradient descent**.

What is the idea?

- Gradient Descent is an iterative algorithm
- Goal: **choose θ^* to minimize cost function $L(\theta)$**
- Start from an initial guess and try to incrementally improve current solution
- At iteration step $\theta^{(iter)}$ is the current guess for θ^*



What is a Gradient?

Gradient is the multi-dimensional analog to a **derivative**

$\nabla L(\theta)$ is a vector-valued function

- It is a vector whose ***i*th** entry is ***i*th** partial derivative evaluated at $\boldsymbol{\theta_i}$

$$\nabla L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial \theta_0} \\ \frac{\partial L(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial L(\theta)}{\partial \theta_d} \end{bmatrix}$$

Negative of gradient indicates direction of steepest descent

- We use the gradient to find out the direction of steepest descent in multidimensional space.

Gradient Descent - Basic algorithm

$\theta^{(iter)} \leftarrow$ an initial guess for θ^*

$iter \leftarrow 1$

repeat

$\quad \theta^{(iter+1)} \leftarrow \theta^{(iter)} - \lambda \nabla L(\theta^{(iter)})$

$\quad iter \leftarrow iter + 1$

until (Stop Condition)

- Here λ is the **"learning rate"** and controls speed of convergence
- $\nabla L(\theta_{iter})$ is the **gradient** of L evaluated at iteration "*iter*" with parameter of θ_{iter}
- Stop conditions can be different

When to Stop?

$\theta^{(iter)} \leftarrow$ an initial guess for θ^*

$iter \leftarrow 1$

repeat

$\quad \theta^{(iter+1)} \leftarrow \theta^{(iter)} - \lambda \nabla L(\theta^{(iter)})$

$\quad iter \leftarrow iter + 1$

until (Stop Condition)

Stop condition can be different, for example:

- **Maximum number of iteration is reached** $(iter < MaxIteration)$
- **Gradient $\nabla L(\theta^{(iter)})$ or parameters are not changing**
  $(\|\theta^{(iter+1)} - \theta^{(iter)}\| < precisionValue)$
- **Cost is not decreasing** $(\|L(\theta^{(iter+1)}) - L(\theta^{(iter)})\| < precisionValue)$
- **Combination of the above**

Mostly we stop it based on number of iterations
(Early stop to avoid "Over-fitting").

Example - Back to Multiple Linear Regression Model

$$y = \theta_0 + \theta_1 x_1 + \cdots + \theta_d x_d$$

$$MSE = L(\Theta) = \frac{1}{2N} \sum_{i=1}^{n} (y^i - (\theta_0 + \theta_1 x_1 + \ldots + \theta_d x_d))^2$$

How to compute the gradient with many dimensions?

Example - Back to Multiple Linear Regression Model

$$y = \theta_0 + \theta_1 x_1 + \cdots + \theta_d x_d$$

$$MSE = L(\Theta) = \frac{1}{2N} \sum_{i=1}^{n} (y^i - (\theta_0 + \theta_1 x_1 + \ldots + \theta_d x_d))^2$$

How to compute the gradient with many dimensions?

Compute partial derivatives

$$\frac{\partial L}{\partial \theta_1} = \frac{-1}{N} \sum_{i=1}^{n} x_1^{(i)} (y^{(i)} - (\theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_d x_d^{(i)}))$$

$$\frac{\partial L}{\partial \theta_2} = \frac{-1}{N} \sum_{i=1}^{n} x_2^{(i)} (y^{(i)} - (\theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_d x_d^{(i)}))$$

$\ldots$

Compute components of the gradients **(map operation)** and then sum them up and update weights in the next iteration **(reduce operation)**

Map Reduce Implementation

```
// initialize parameters
iter = 0
learning Rate = 0.01
num Iteration = 400
theta = np. zeros ( no Parameters )

while ( iter < max Num Iteration ):
  reduce Data = my Data . map (
           // Calculate the gradients
      )
      . reduce (
           // update model parameters
           // lambda theta :  - learning Rate * theta
      )
  iter = iter +1
```

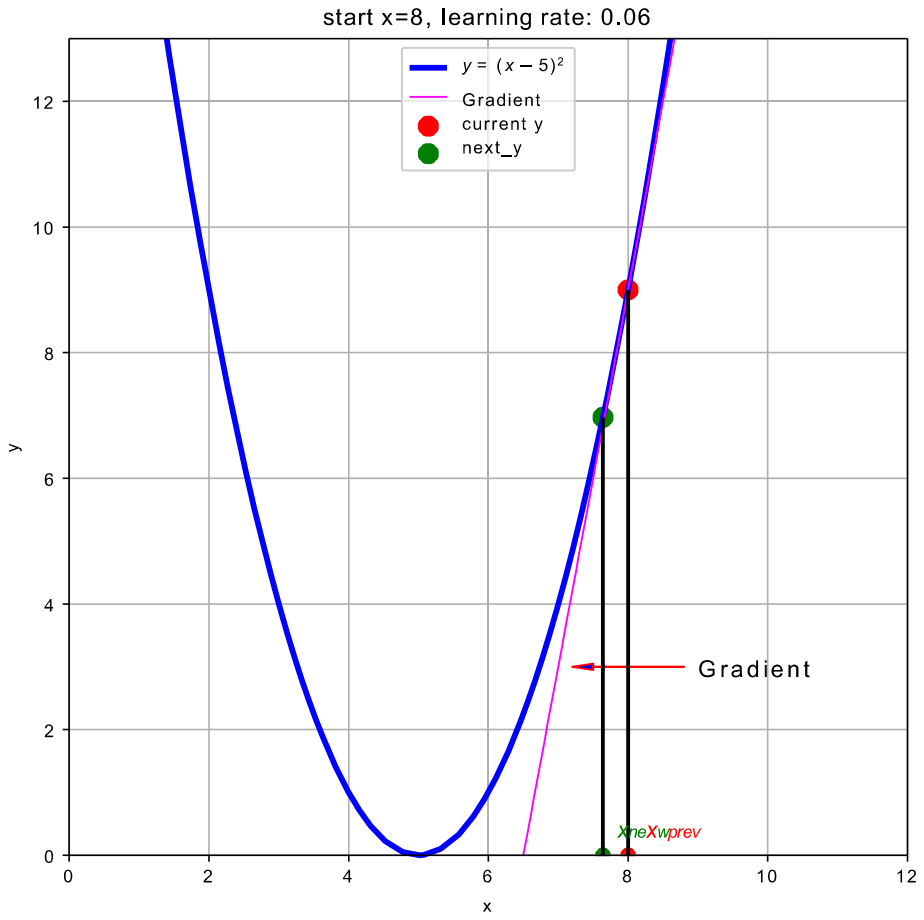
In Spark you can **reduceByKey()** or better **treeAggregate()**

Learning Rate is Super Important

Gradient Descent - learning rate too Small (Slow convergence)

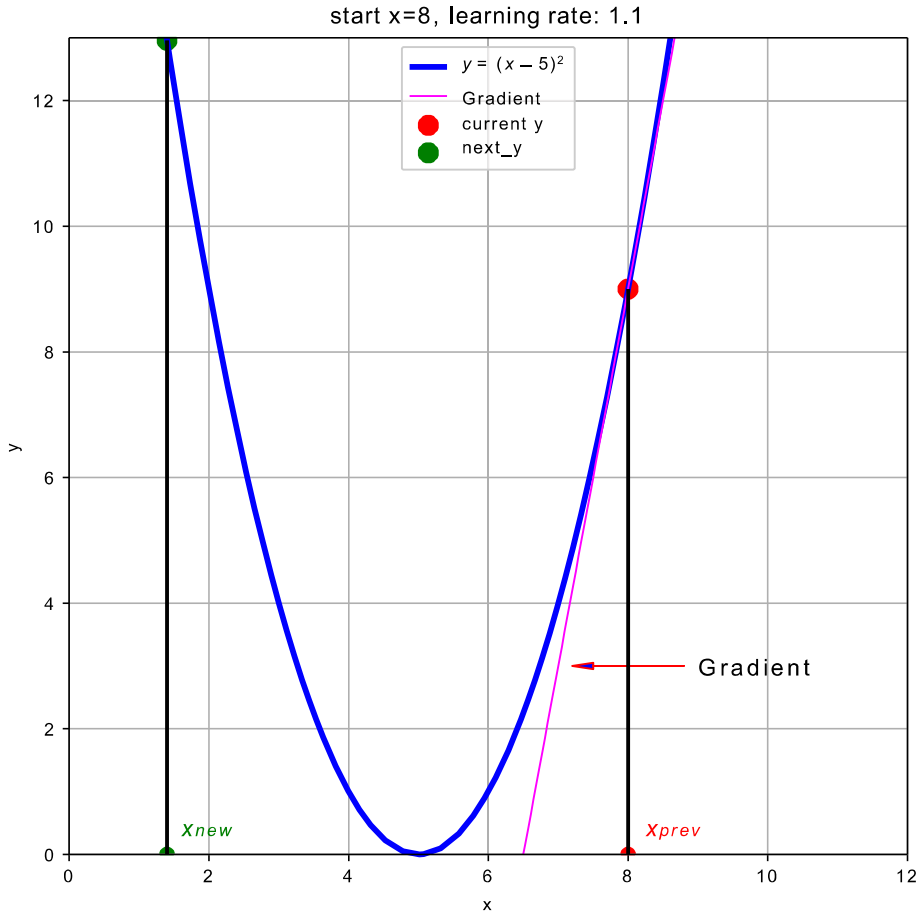
The learning rate is Super important.

Too small: many, many passes through the data to converge



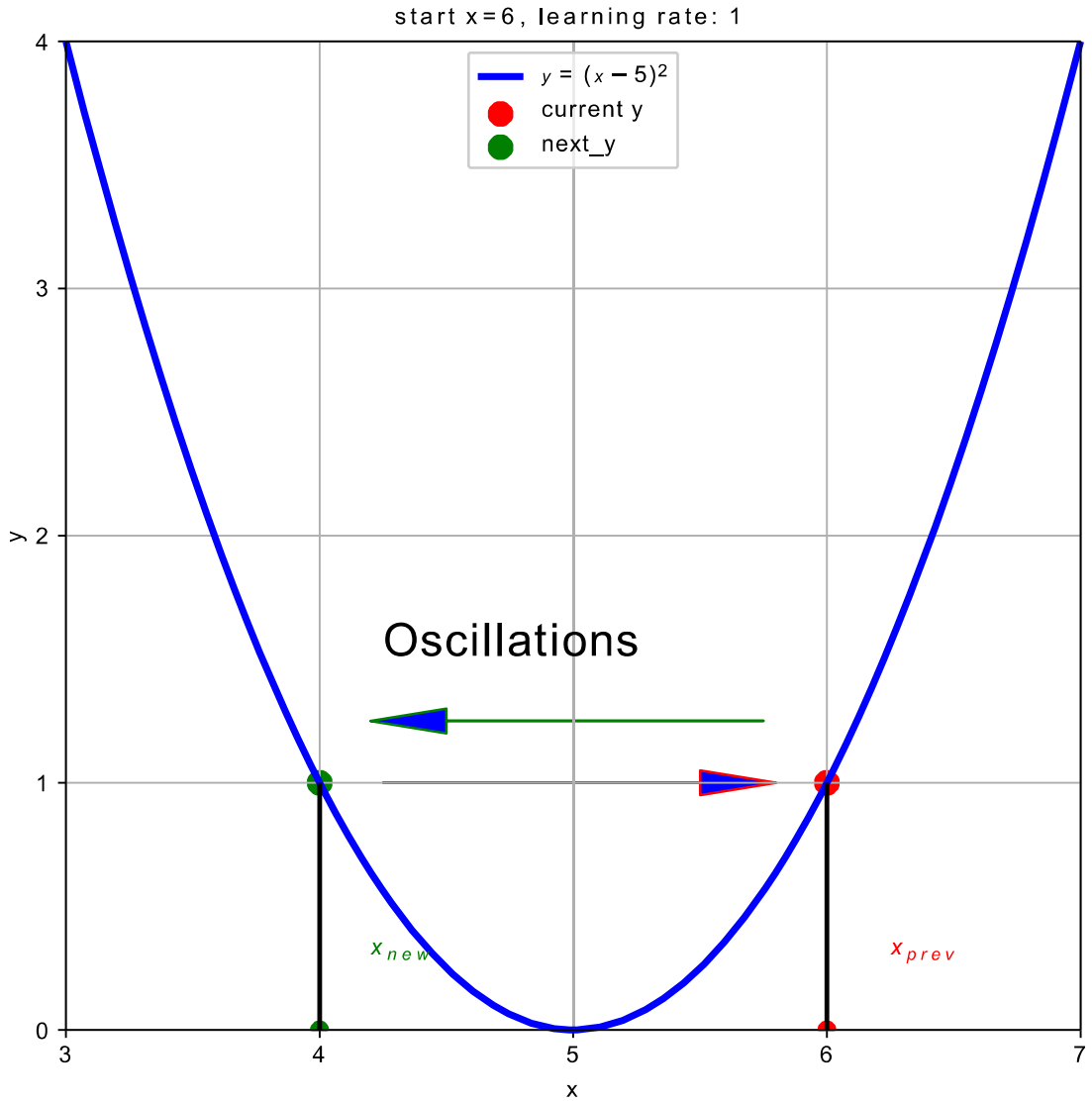
Gradient Descent

Too large: Jumping around



Gradient Descent - Oscillations

Too large: oscillate into oblivion



Choose Learning Rate - Line Search

Best option (in terms of results) but **most expensive**:

- Solve another mini-optimization problem
- Select λ so as to minimize $L(\theta^{(iter)})$
- It's a 1-dimensional optimization problem!
- Called a line search

Line Search

$l \leftarrow 0$

$h \leftarrow 999999$

while $(h - l > e)$ **do**

  $h' \leftarrow l + \frac{1}{c}(h - l)$

  $l' \leftarrow h - \frac{1}{c}(h - l)$

  $goodness_h \leftarrow L(\theta^{(i)} - h' \nabla L(\theta^{(iter)}))$

  $goodness_l \leftarrow L(\theta^{(i)} - l' \nabla L(\theta^{(iter)}))$

  if $(goodness_h < goodness_l)$ **then**

    $l \leftarrow l'$

  else

    $h \leftarrow h'$

  end if

end while

"Golden Section Search" $c = \frac{1}{2}(1 + \sqrt{5}) = 1.618$

Other Ways To Choose Learning Rate - ”Bold Driver”

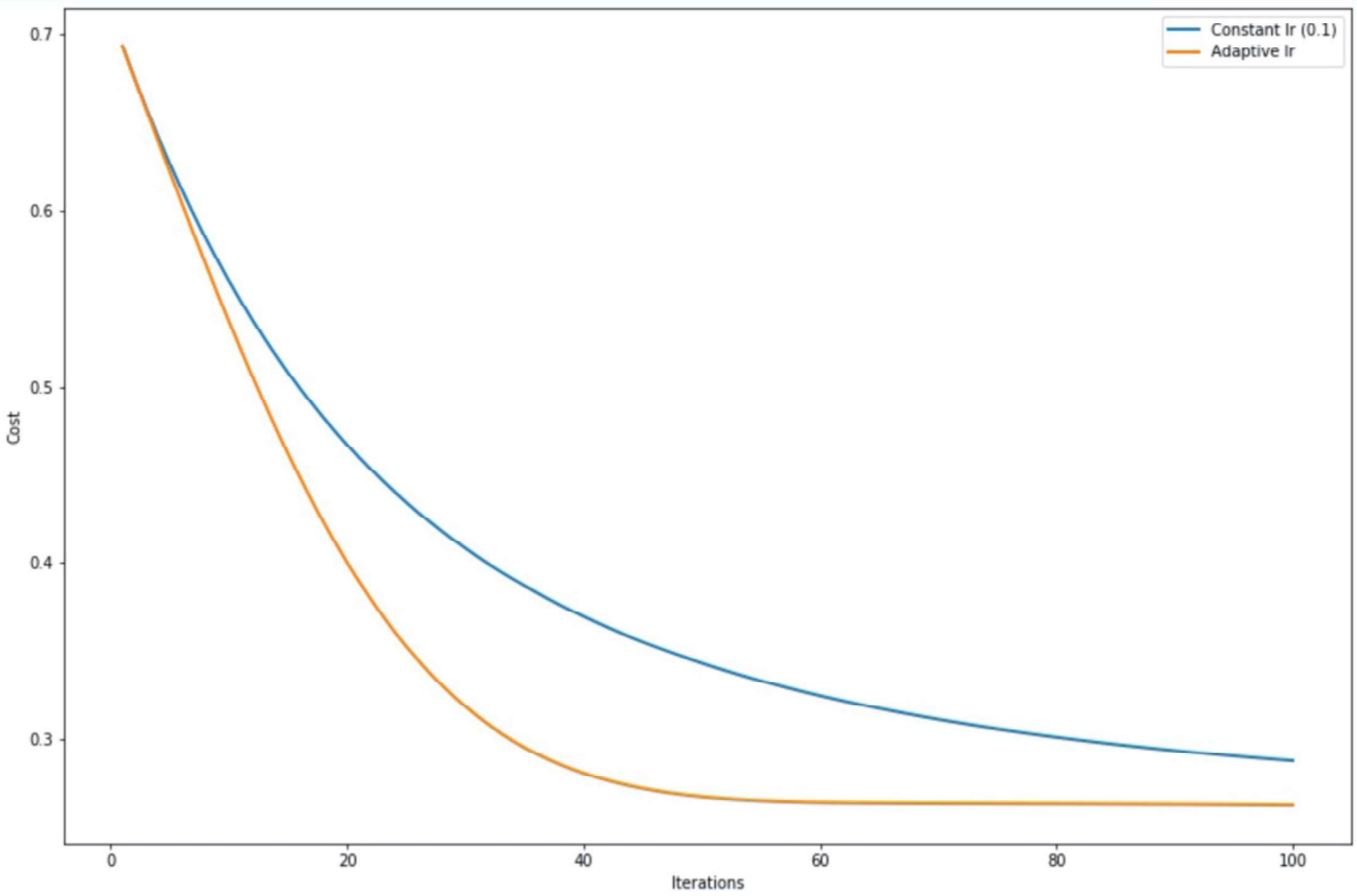
Line search is very costly!

One other standard method is ”Bold Driver”

- Make a very conservative initial guess for λ
  At each iteration, compute the cost $L(\Theta^{(iter)})$
- Better than last time?
  $\lambda \leftarrow 1.05 \times \lambda$
  If cost decreases, increase learning rate
- Worse than last time?
  $\lambda \leftarrow 0.5 \times \lambda$
  If cost increases, decrease rate

This would be then just one evaluation of loss function per iteration!

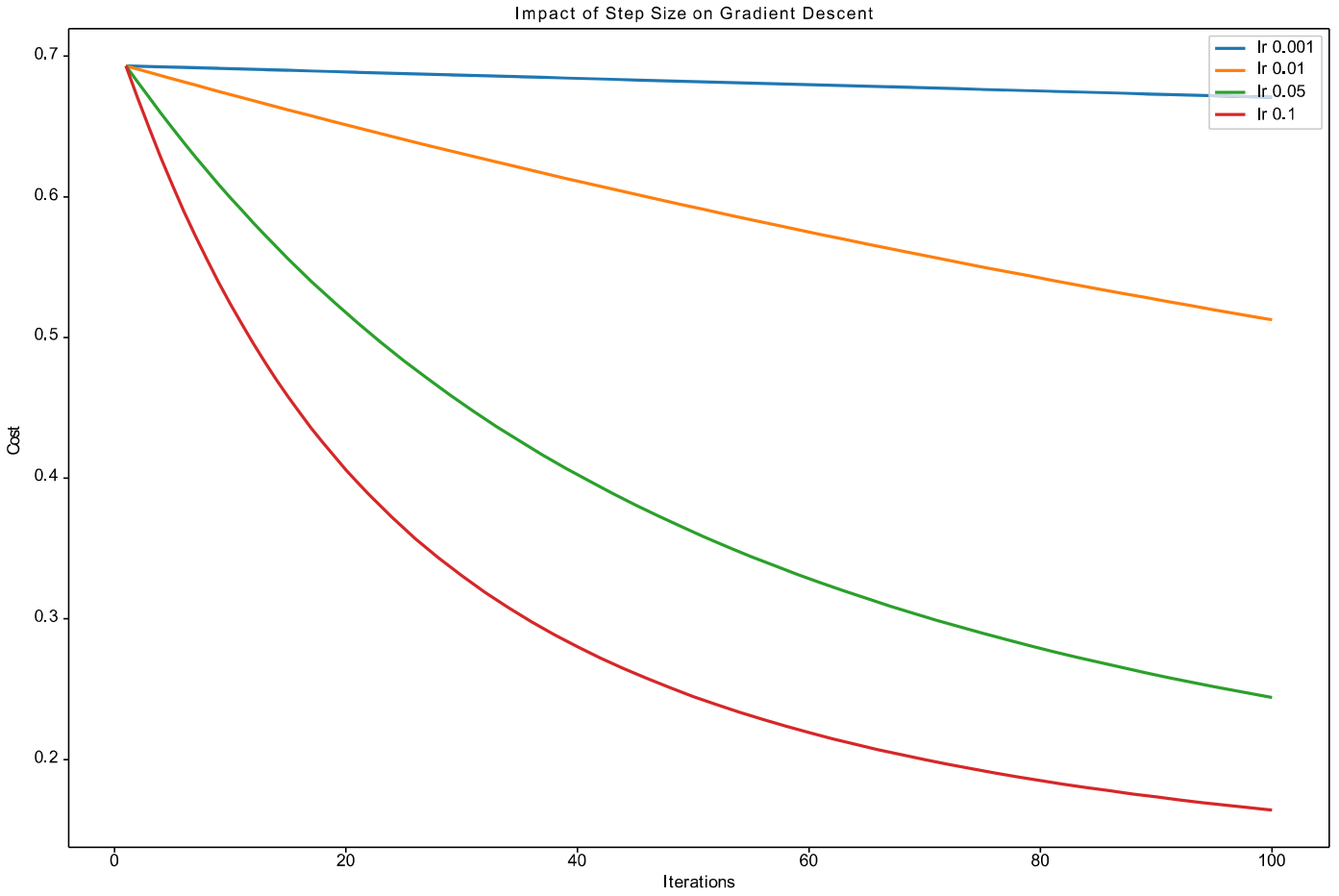
Cost Monitoring - ”Bold Driver” vs. Constant Learning Rates



Text Classification with Logistic regression (20k Dimensions Term Frequencies) .

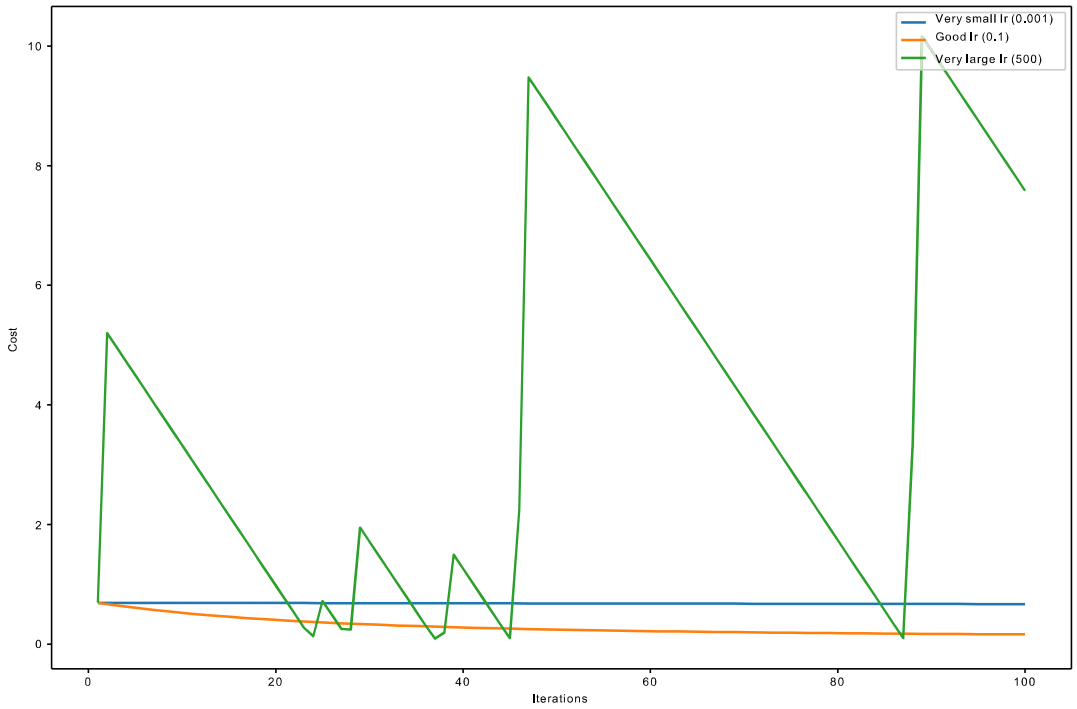
- Horizontal axis is number of iterations
- Vertical axis shows value of negative log-likelihood

Text Classification with Logistic regression - Different Learning Rate



The same Text Classification with 20k Dimensions, term frequencies of words

Example - Very Large Learning Rate

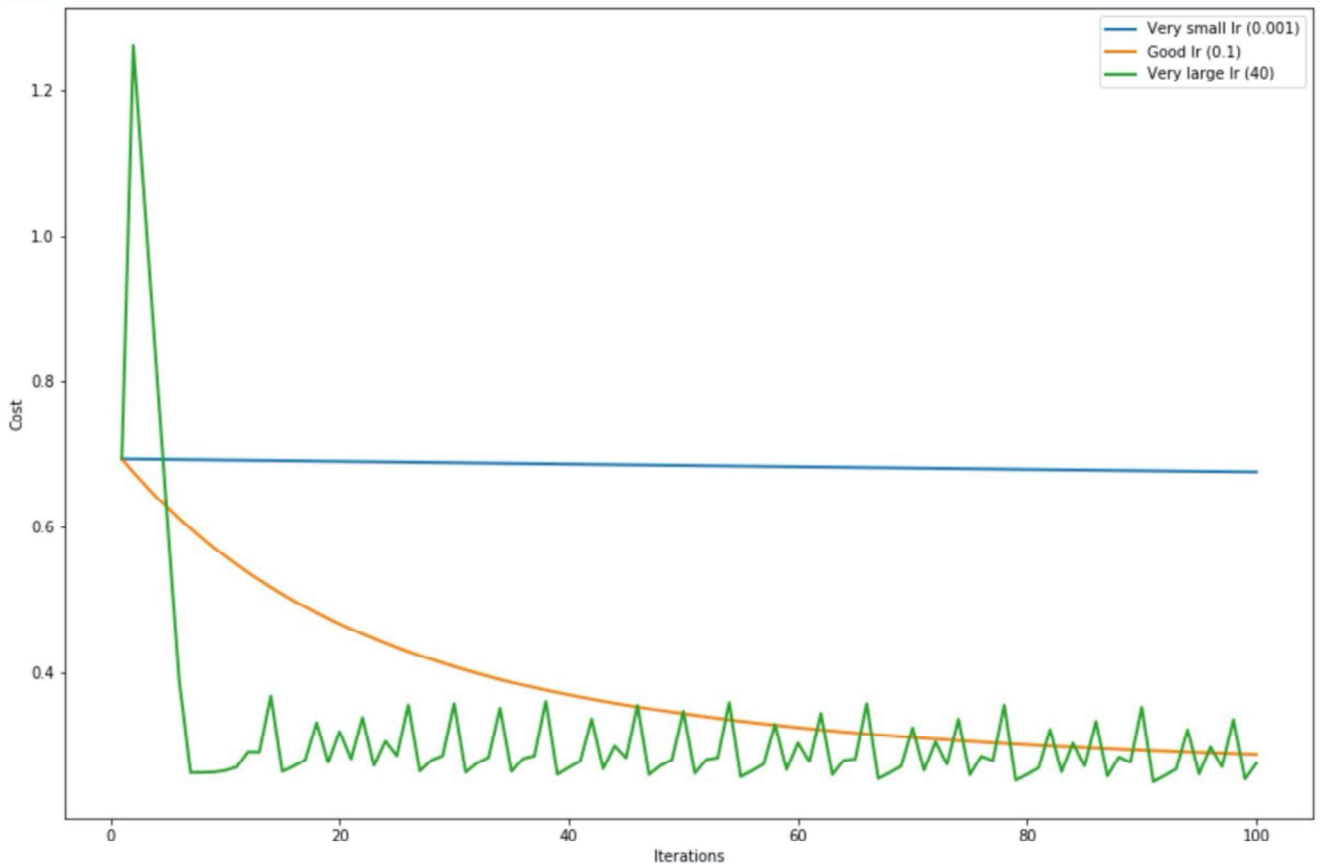


The same Text classification with 20k Dimensions, term frequencies of words

Very small learning rate of 0.001

- Very Large learning rate
- Good working learning rate.

Text Classification with Logistic regression - Large Learning Rate



20k Dimensions, term frequencies of words

Very small learning rate of 0.001

- Large learning rate
- Good working learning rate.

Variations of Gradient Descent

Variations of Gradient Descent

Depending on Size of data that we use in each iteration:

- **Full Batch Gradient Descent** (Using the whole data set (size n))
- **Stochastic Gradient Descent (SGD)** (Using one sample per iteration (size 1))
- **Mini Batch Gradient Descent** (Using a mini batch of data (size $m < n$))

Some times people refer to SGD as mini batch.

Stochastic Gradient Descent

Calculate the gradient of a single sample and update parameters in each iterations

- Pick up samples for update by **sequential read (pass over the data** - do not take random sample from a big data set in each iterations - computationally expensive)
- It is noisy and sometimes will do lots of iterations
- Do not be afraid of **non-convex functions**, SGD can help to **get out of local minimums**.
- It is a widely applied approach
- SGD is often faster in producing good results.

Mini-Batch Gradient Descent

Something between Stochastic and Full-batch

- Pick up mini batch samples for update by sequential pass over the data and use it to update
- Batch gradient descent has better convergence rates than stochastic gradient descent in theory.
- But we may not to pursue to converge fast (presumably results in overfitting)
- Depending on size, can help to out of local minimums.

Implementation Tips:

- Use mini-batch data size of 64, 256, 512, ... (power of 2))
- Make sure that mini-batch data fits in CPU/GPU memory

Implementation Tips:

- **Monitor the costs in each iteration to check if it decreases.**
- **Map** to calculate gradients, **Reduce** to update the weights
- Use vectorization of computations (run bulk operations, e.g., use numpy inside Spark RDDs or Spark Dataframes)

Issues of Gradient Descent

- You can find local min/max and not Global
- There is no grantee that you can find the global minimum
- All depends on Starting point, Step-Size and Function type.

The main problem is :

Oscillate into oblivion

Gradient Descent with Momentum

A momentum is a moving average of our gradients.

$$V^{(iter)} = \beta V^{iter-1} + (1-\beta)\nabla L(W, X, y)$$
$$W^{(iter+1)} = W^{(iter)} - \lambda V^{(iter)}$$

β is a positive number

λ is learning rate

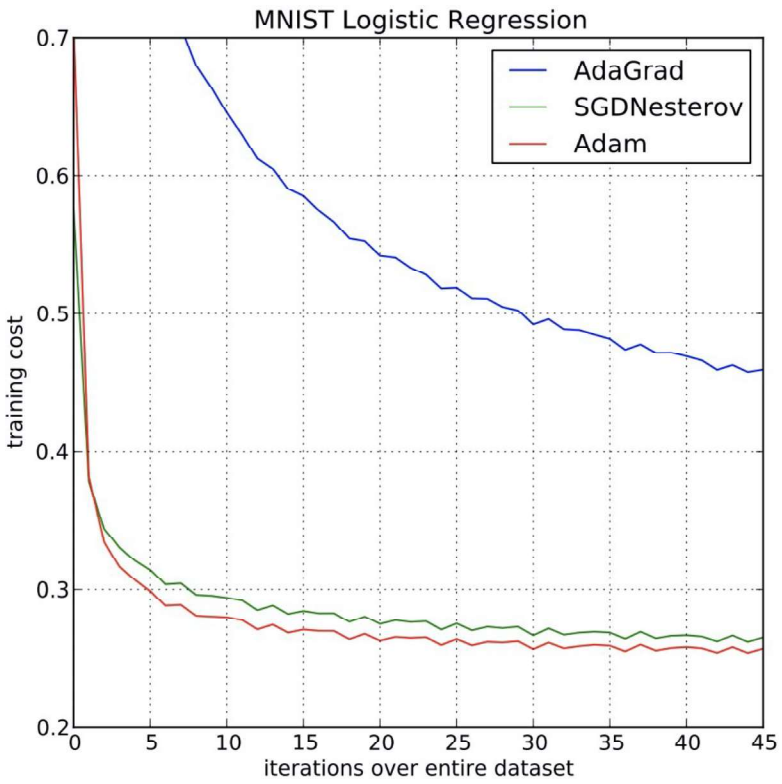
$L(W, X, y)$ is cost function

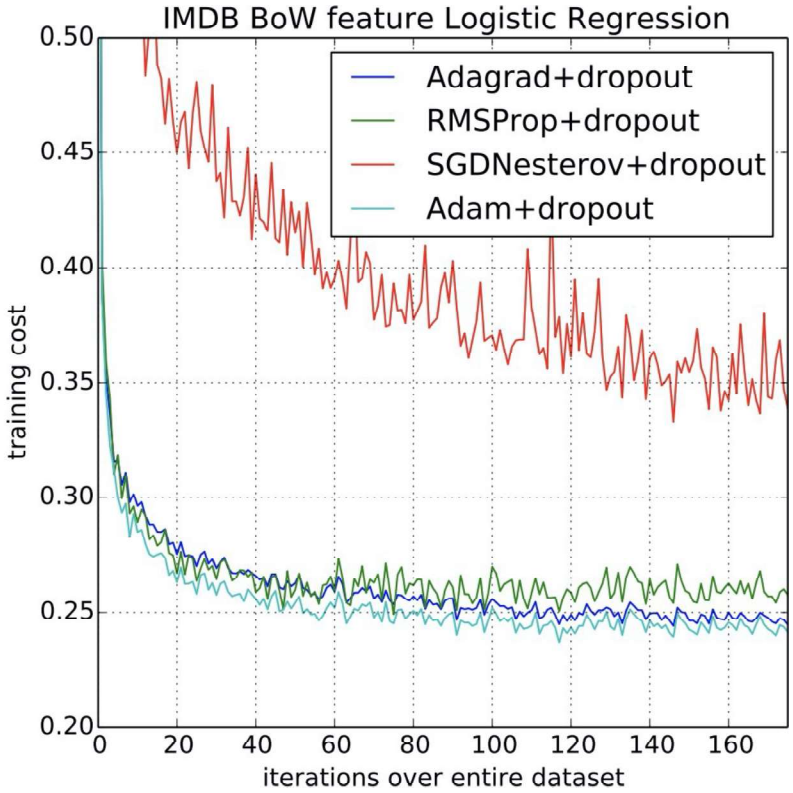
W is vector of weights (model parameters)

Further methods

- Adaptive gradient algorithm (AdaGrad)
- Root Mean Square Propagation (RMSProp)
- **Adaptive Moment Estimation (Adam)**
- ...

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION





Logistic regression training negative log likelihood on MNIST images and IMDB movie reviews with 10,000 bag-of-words (BoW) feature vectors.

(Image from Kingma et. al 2017)

Summary

Gradient Descent is a great optimization algorithm

- Gradient descent is a **first-order** iterative optimization algorithm and is easy to use.
- Widely applicable in many different machine learning methods
- But convergence can be slow

Some of the related publications

- Diederik P. Kingma, Jimmy Ba (2015). *Adam: A Method for Stochastic Optimization.* In proceeding of ICLR 2015: San Diego, CA, USA
- John Duchi, Elad Hazan, and Yoram Singer (2011). *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.* Journal of Machine Learning Research, 12:2121–2159, 2011.
- Timothy Dozat (2016). *Incorporating Nesterov Momentum into Adam.* ICLR Workshop, (1):2013–2016, 2016.
- Bottou, L´eon (1998). *Online Algorithms and Stochastic Approximations.* Online Learning and Neural Networks. Cambridge University Press. ISBN 978-0-521-65263-6
- Bottou, L´eon (2010). *Large-scale machine learning with SGD.* Proceedings of COMPSTAT'2010. Physica-Verlag HD, 2010. 177-186.
- Bottou, L´eon (2012). *SGD tricks. Neural Networks: Tricks of the Trade.* Springer Berlin Heidelberg, 2012. 421-436.