# Optimization for Training Deep Models

# Topics in Optimization

- Role of Optimization in
- 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

## Optimization is essential for DL

- All is an instance of a recipe:
  1. Specification of a dataset
  2. A cost function
  3. A model
  4. An optimization procedure
- The recipe for linear regression
  1. Data set : $X$ and $y$
  2. Cost function: $\boxed{J(\boldsymbol{w}) = -E_{x,y\sim \hat{p}_{data}} \log p_{\text{model}}(y \mid \boldsymbol{x}) + \lambda \|\boldsymbol{w}\|_2^2}$ Includes regularization
  3. Model specification: $\boxed{p_{\text{model}}(y \mid \boldsymbol{x}) = N(y; \boldsymbol{x}^T \boldsymbol{w} + \boldsymbol{b}, 1)}$
  4. Optimization algorithm
     - solving for where the cost is minimal

## Our focus is on one case of optimization

- To 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

# Optimization in

- There are many contexts for optimization in DL
  1. Inference with PCA requires optimization
     - Encoding: $f(\boldsymbol{x})=\boldsymbol{c}$, Decoding: $\boldsymbol{x} \approx g(f(\boldsymbol{x})), g(\boldsymbol{c})=D\boldsymbol{c}$
     - Optimal $\boldsymbol{c}^*=\mathrm{argmin}_{\boldsymbol{c}}\|\boldsymbol{x}-g(\boldsymbol{c})\|_2$, Reconstruction: $g(f(\boldsymbol{x}))=DD^{\mathrm{T}}\boldsymbol{x}$
  2. Analytical Optimization to write proofs/design algorithms
     - Squared error objective is same as maximum likelihood
  3. Neural network training
     - Most difficult optimization of all is neural network training
       - Weight decay minimization:

$$\boxed{J(\boldsymbol{w}) = -E_{x,y\sim\hat{p}_{data}} \log p_{\text{model}}(y\,|\boldsymbol{x}) + \lambda \|\boldsymbol{w}\|_2^2}$$

## Importance of Optimization

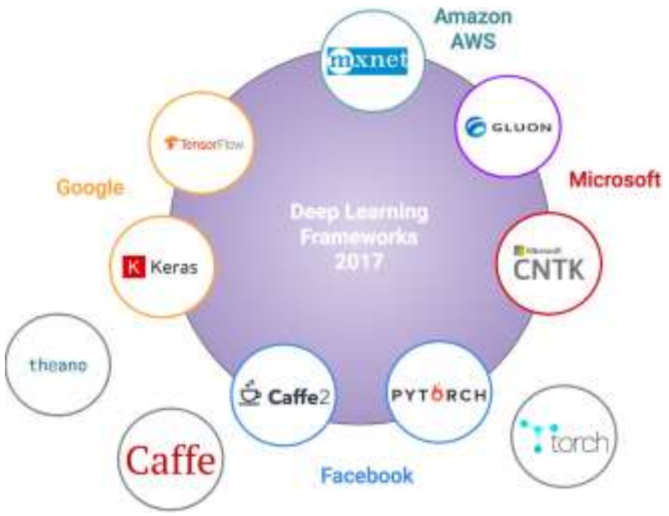
- Common to invest days to months of time on 100s of machines to solve a single instance of neural network training problem
- Because the problem is so important and so expensive
  - Specialized optimization techniques have been developed for solving it

# Plan of Discussion of Optimization

1. How training optimization differs from pure optimization
2. Challenges that make optimization of neural networks difficult
3. Several practical algorithms including
   1. Optimization algorithms
   2. Strategies for initializing parameters
      - Most advanced algorithms
        - adapt learning rates or
        - leverage *second derivatives* of cost function
4. Combine simple optimization algorithms into higher-level procedures

# DL Frameworks include Optimization

- Frameworks offer building blocks
  - For designing, training, validating deep nets through a high level programming interface
- Optimized for performance
  - Provide parallelization for GPUs
  - Visualization of network modeling & interface



- Example: Keras
  - An open source neural network library written in Python
  - It is capable of running on top of TensorFlow
  - Contains implementations of building blocks, such as
    1. Layers
    2. Objectives
    3. Activation functions
    4. Optimizers
    5. Tools to make working with image and text data easier

# Keras for MNIST Neural Network

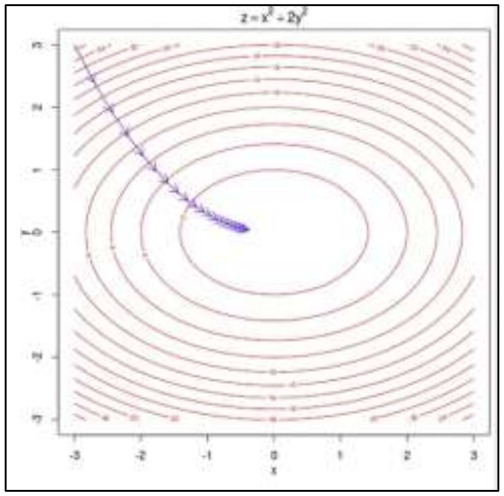
- # Neural Network
- import keras
- from keras.datasets import mnist
- from keras.layers import Dense
- from keras.models import Sequential
- (x_train, y_train), (x_test, y_test) = mnist.load_data()
- num_classes=10
- image_vector_size=28*28
- x_train = x_train.reshape(x_train.shape[0], image_vector_size)
- 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)

# Summary of Optimization Methdos

- Movies:

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

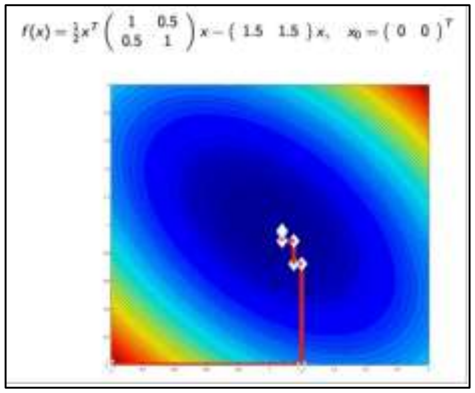
## Gradient Descent



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

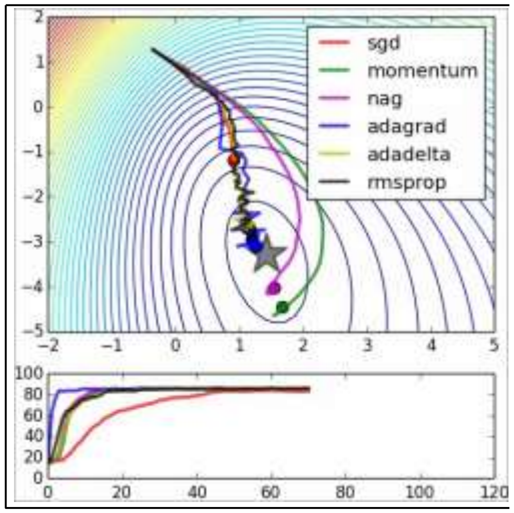
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \boldsymbol{g}$$

## Coordinate Descent



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

## SGD



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

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \boldsymbol{g}$$

## How Learning Differs from Optimization

# 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  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 $J(\boldsymbol{\theta})$ 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

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

  - Where
    - $L$ is the per-example loss function
    - $f(\mathbf{x};\boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$
    - $\hat{p}_{data}$ 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 $f(\boldsymbol{x}; \boldsymbol{\theta})$ and $y$
- Trivial to extend to cases:
  - Where parameters $\boldsymbol{\theta}$ and input $\boldsymbol{x}$ 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

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

$L$ is the per-example loss function

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

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

  – 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 $\boxed{J^*(\theta) = E_{(\mathbf{x},y)\sim p_{\text{data}}}\Big(L(f(\mathbf{x};\theta),y)\Big)}$
  - If we knew $p_{\text{data}}(\mathbf{x},y)$ it would be optimization solved by an optimization algorithm
  - When we do not know $p_{\text{data}}(\mathbf{x},y)$ but only have a training set of samples, we have a machine learning problem
- Empirical risk, with $m$ training examples, is

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

# Empirical Risk Minimization

- Empirical risk, with $m$ training examples, is

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

  - 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

## Ex: Decomposition into a sum

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

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

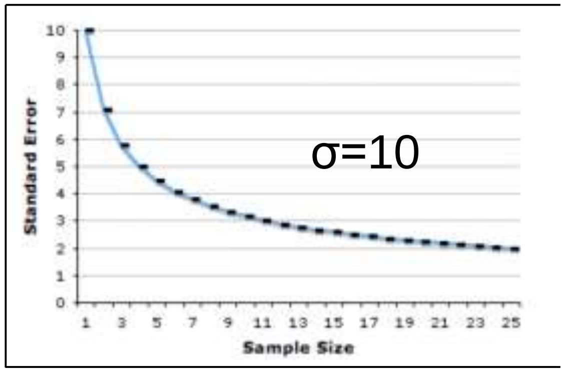
    - Maximizing this sum is equivalent to maximizing the *expectation* over the empirical distribution defined by the training set $\boxed{J(\theta) = E_{\boldsymbol{x},y \sim \hat{p}_{data}} \log p_{model}\left(\boldsymbol{x}, y; \theta\right)}$
    - Commonly used property of $J(\boldsymbol{\theta})$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

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

  - 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 $\boxed{\frac{\sigma}{\sqrt{n}}}$
  - where $\sigma$ is std dev of samples
  - Denominator shows that error decreases less than linearly with no. of samples
    - Ex: 100 samples vs 10,000 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

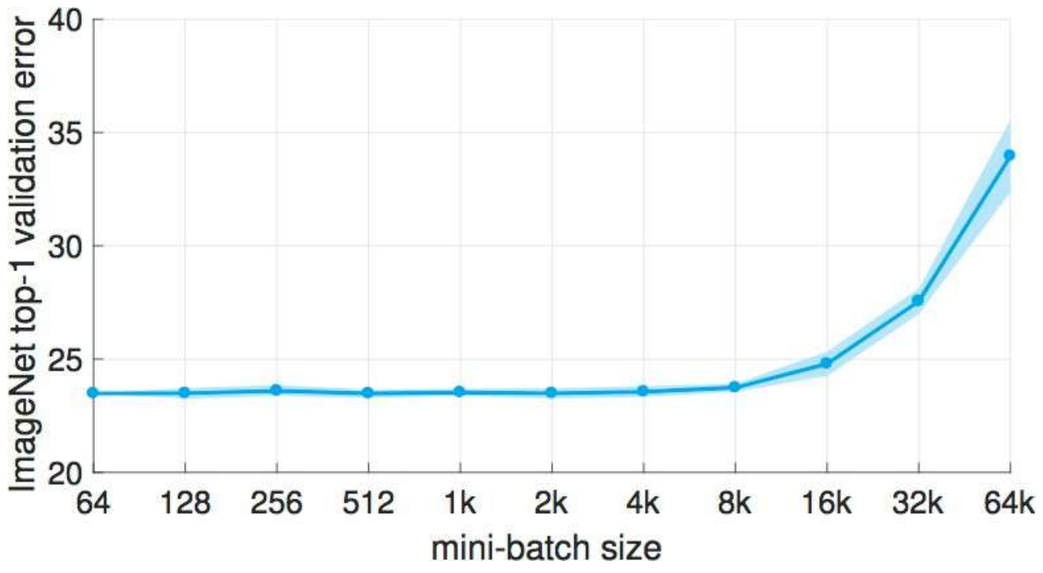
## Stochastic or online methods

- 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
- 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.

https://arxiv.org/pdf/1706.02677.pdf

## 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 $\boldsymbol{g}$ are robust and can handle smaller batch sizes like 100
- Second order methods using Hessian H and compute updates such as $H^{-1}\boldsymbol{g}$ 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

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

  – 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 $(\boldsymbol{x},y)$ coming from data generating distribution $p_{\text{data}}(\boldsymbol{x},y)$
      - » Examples never repeated, every example is a fair sample
- Equivalence is easy to derive when $\boldsymbol{x}$ and $y$ are discrete
  - As seen next

# Discrete case with loss function

- Generalization error (in terms of loss function) is $\boxed{J^*(\theta) = E_{(\boldsymbol{x},y)\sim p_{data}}\Big(L(f(\boldsymbol{x};\theta),y)\Big)}$
  - Which can be written as a sum

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

  - with exact gradient

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

Implies that derivative can be computed in batches

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

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

- 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

# Challenges in Neural Network Optimization

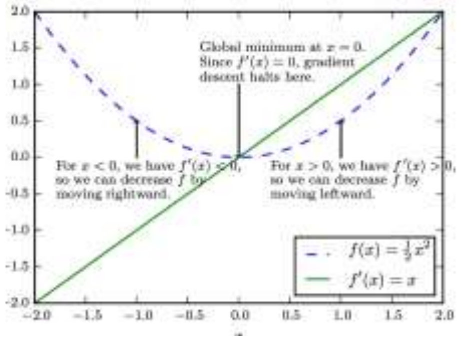
# Topics in Optimization for DL

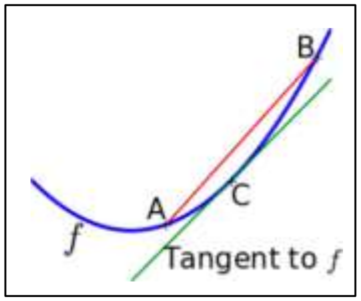
- Importance of Optimization in machine learning

1. How learning differs from optimization
2. Challenges in neural network optimization
3. Basic Optimization Algorithms
4. Parameter initialization strategies
5. Algorithms with adaptive learning rates
    1. AdaGrad, RMSPro, Adam
    2. Choosing the right optimization algorithm
6. Approximate second-order methods
7. Optimization strategies and meta-algorithms

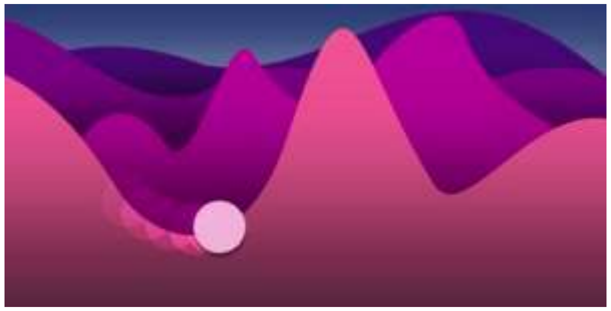
# Optimization is a difficult task

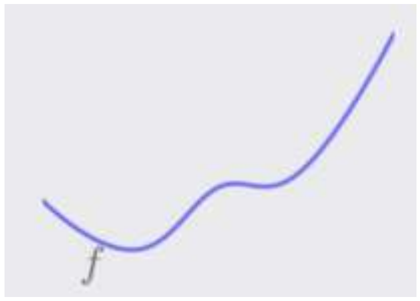
- Optimization is an extremely difficult task
  - Traditional ML: careful design of objective function and constraints to ensure convex optimization





  - When training neural networks, we must confront the nonconvex case





# Challenges in Optimization

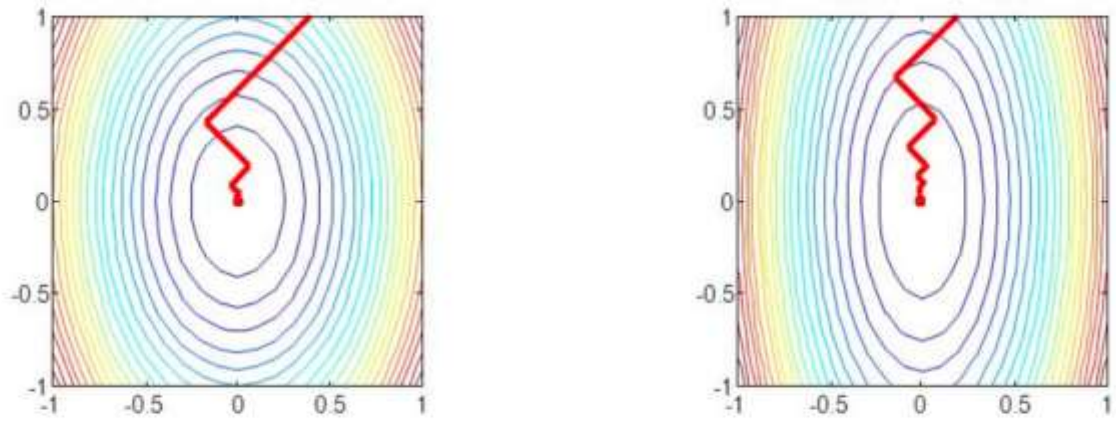
- Summary of challenges in optimization
  1. Ill-conditioning
  2. Local minima
  3. Plateaus, saddle points and other flat regions
  4. Cliffs and exploding gradients
  5. Long-term dependencies
  6. Inexact gradients
  7. Poor correspondence between local & global structure
  8. Theoretical limits of optimization

## 1. Ill-conditioning of the Hessian

- Even when optimizing convex functions one problem is an ill conditioned Hessian matrix, $H$
  - Very general problem in optimization, convex or not

**Condition number** is the ratio of maximal and minimal eigenvalues of the Hessian $\nabla^2 f(x)$,

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$$



Problem with large condition number is called **ill-conditioned**

Steepest descent **convergence rate is slow** for ill-conditioned problems

Source: https://slideplayer.com/slide/4916524/

## Result of Ill-conditioning

- Causes SGD to be stuck: even very small steps cause increase in cost function

$$\boxed{f(x) \approx f(x^{(0)}) + (x - x^{(0)})^T g + \frac{1}{2}(x - x^{(0)})^T H(x - x^{(0)})}$$

Substituting $x = x^{(0)} - \varepsilon g$

$$\boxed{f(x^{(0)} - \varepsilon g) \approx f(x^{(0)}) - \varepsilon g^T g + \frac{1}{2}\varepsilon^2 g^T H g}$$

  - Gradient descent step of $-\varepsilon g$ will add to the cost $\boxed{-\varepsilon g^T g + \frac{1}{2}\varepsilon^2 g^T H g}$
  - Ill conditioning becomes a problem when $\boxed{\frac{1}{2}\varepsilon^2 g^T H g > \varepsilon g^T g}$
  - To determine whether ill-conditioning is detrimental monitor $g^T g$ and $g^T H g$ terms
    - Gradient norm doesn't shrink but $g^T H g$ grows order of magnitude
  - Learning becomes very slow despite a strong gradient

## 2. Local Minima

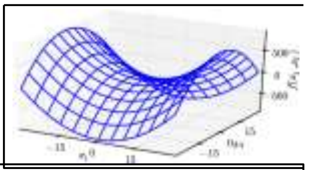
- In convex optimization, problem is one of finding a local minimum
- Some convex functions have a flat region rather than a global minimum point
- Any point within the flat region is acceptable
- With non-convexity of neural nets many local minima are possible
- Many deep models are guaranteed to have an extremely large no. of local minima
- This is not necessarily a major problem

# Model Identifiability

- Model is identifiable if large training sample set can rule out all but one setting of parameters
  - Models with latent variables are not identifiable
    - Because we can exchange latent variables
      - If we have $m$ layers with $n$ units each there are $n!^m$ ways of arranging the hidden units
    - This non-identifiability is *weight space symmetry*
  - Another is scaling incoming weights and biases
    - By a factor $\alpha$ and scale outgoing weights by $1/\alpha$
- Even if a neural net has uncountable no. of minima, they are equivalent in cost
  - So not a problematic form of non-convexity

# 3. Plateaus, Saddle Points etc

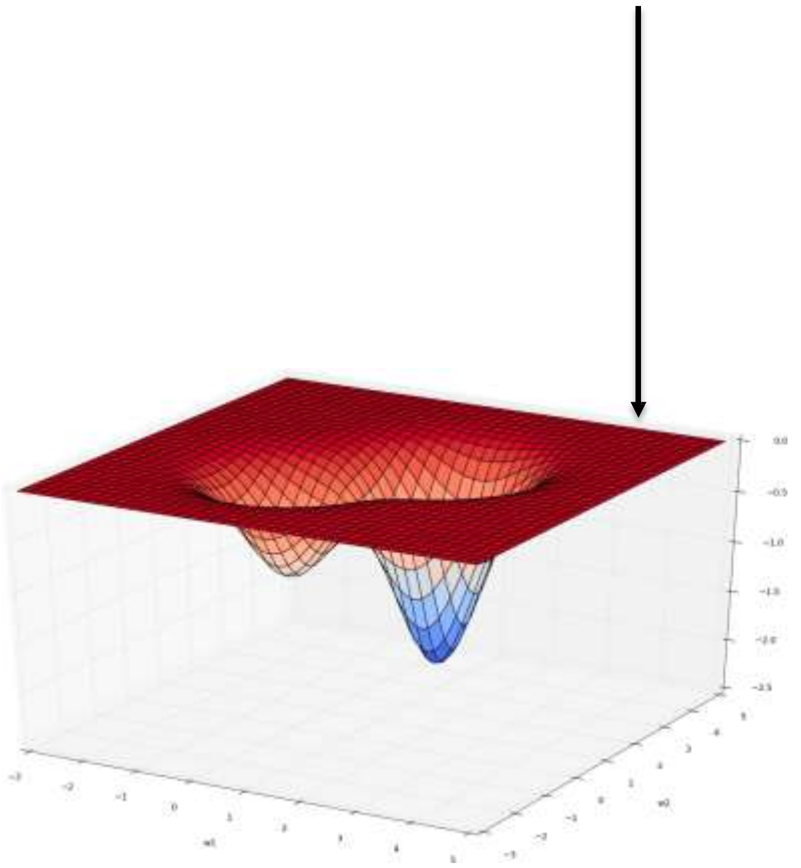
- More common than local minima/maxima are:
  - Another kind of zero gradient points: saddle points
    - At saddle, Hessian has both positive and negative values
      - Positive: cost greater than saddle point
      - Negative values have lower value
    - In low dimensions:
      - Local minima are more common
    - In high dimensions:
      - Local minima are rare, saddle points more common

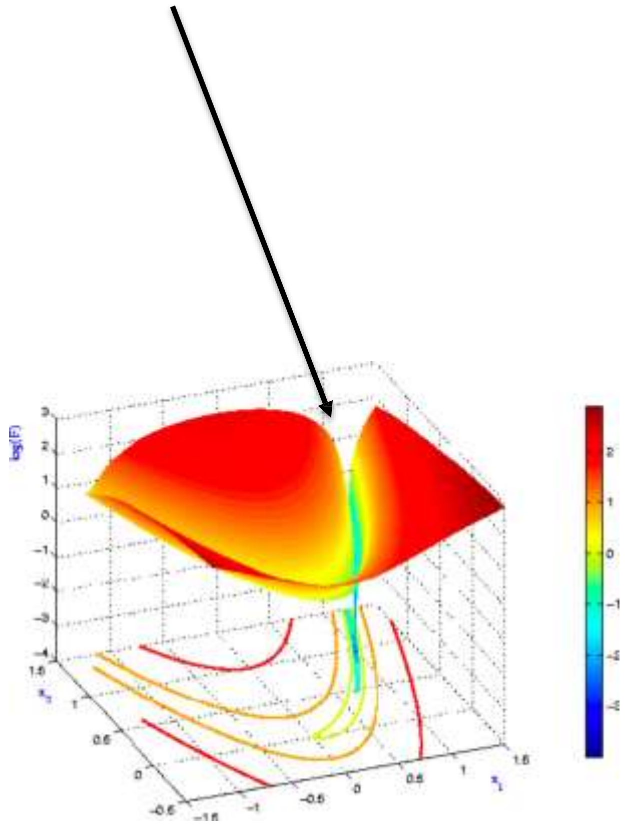


Contains both positive and negative curvature
Function is $f(\boldsymbol{x})=x_1^2-x_2^2$

- For Newton's saddle points pose a problem
  - Explains why second-order methods have not replaced gradient descent

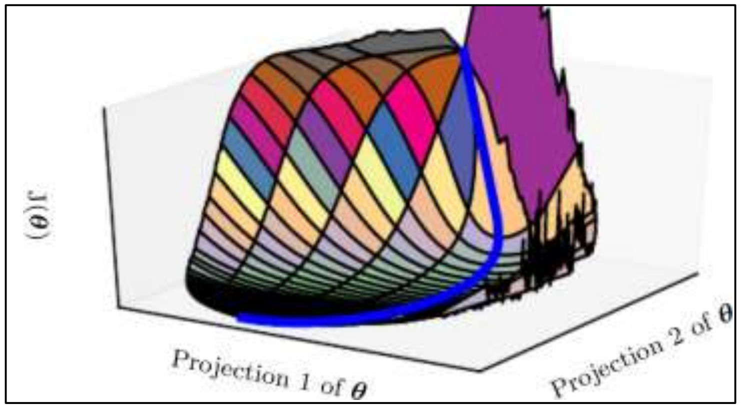
## Plateau and Ravine





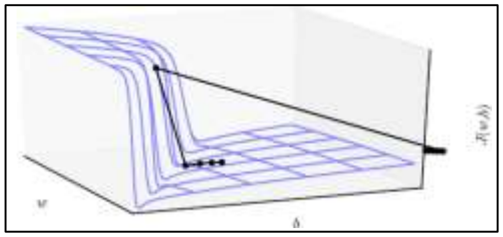
## Cost Function of Neural Network

- Visualizations are similar for
  - Feedforward networks
  - Convolutional networks
  - Recurrent networks



- Applied to object recognition and NLP tasks
- Primary obstacle is not multiple minima but saddle points
- Most of training time spent on traversing flat valley of the Hessian matrix or circumnavigating tall "mountain" via an indirect arcing path

# 4.Cliffs and Exploding Gradients



- Neural networks with many layers
  - Have steep regions resembling cliffs
    - Result from multiplying several large weights
    - E.g., RNNs with many factors at each time step
- Gradient update step can move parameters extremely far, jumping off cliff altogether
- Cliffs dangerous from either direction
- *Gradient clipping* heuristics can be used

## 5. Long-Term Dependencies

- When computational graphs become extremely deep, as with
  - feed-forward networks with many layers
  - RNNs which construct deep computational graphs by repeatedly applying the same operation at each time step
- Repeated application of same parameters gives rise to difficulties
- Discussed further with RNNs in 10.7

# 6. Inexact Gradients

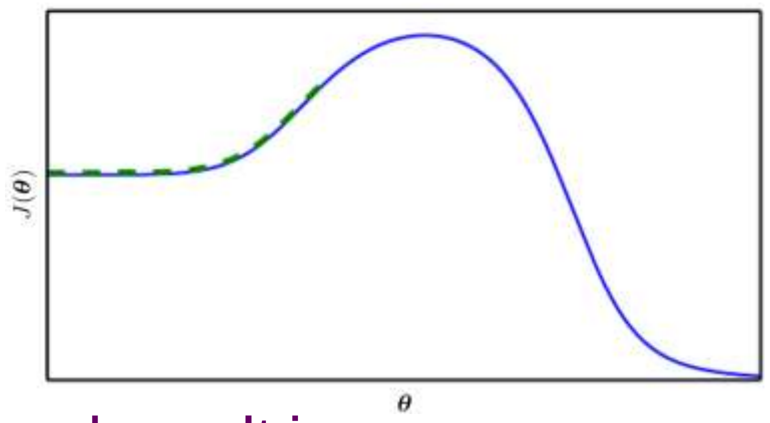
- Optimization algorithms assume we have access to exact gradient or Hessian matrix
- In practice we have a noisy or biased estimate
  - Every algorithm relies on sampling- based estimates
    - In using minibatch of training examples
  - In other case, objective function is intractable
    - In which case gradient is intractable as well
    - Contrastive Divergence gives a technique for approximating the gradient of the intractable log-likelihood of a Boltzmann machine

## 7. Poor Correspondence between Local and Global Structure

- It can be difficult to make a single step if:
  - $J(\boldsymbol{\theta})$ is poorly conditioned at the current point $\boldsymbol{\theta}$
  - $\boldsymbol{\theta}$ lies on a cliff
  - $\boldsymbol{\theta}$ is a saddle point hiding the opportunity to make progress downhill from the gradient
- It is possible to overcome all these problems and still perform poorly
  - if the direction that makes most improvement locally does not point towards distant regions of much lower cost

# Need for good initial points

- Optimization based on local downhill moves can fail if local surface does not point towards the global solution
- Research directions are aimed at finding good initial points for problems with a difficult global structure
  - Ex: no saddle points or local minima
    - Trajectory of circumventing such mountains may be long and result in excessive training time



## 8. Theoretical Limits of Optimization

- There are limits on the performance of any optimization algorithm we might design for neural networks
- These results have little bearing on the use of neural networks in practice
  - Some apply only to networks that output discrete values
    - Most neural networks output smoothly increasing values
  - Some show that there exist problem classes that are intractable
    - But difficult to tell whether problem falls in thet class

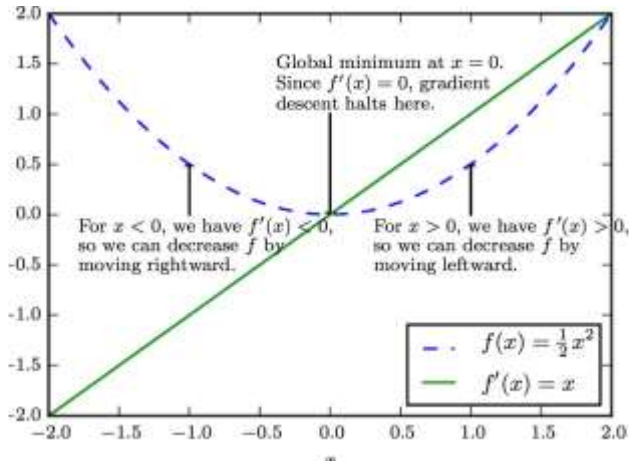
## Basic Optimization Algorithms

## Topics

- Importance of Optimization in machine learning
- How learning differs from optimization
- Challenges in neural network optimization
- Basic Optimization Algorithms
  - SGD, Momentum, Nesterov Momentum
- Parameter initialization strategies
- Algorithms with adaptive learning rates
  - AdaGrad, RMSProp, Adam
- Approximate second-order methods
- Optimization strategies and meta-algorithms

## 1. Stochastic Gradient Descent

- Gradient descent follows gradient of entire training set downhill

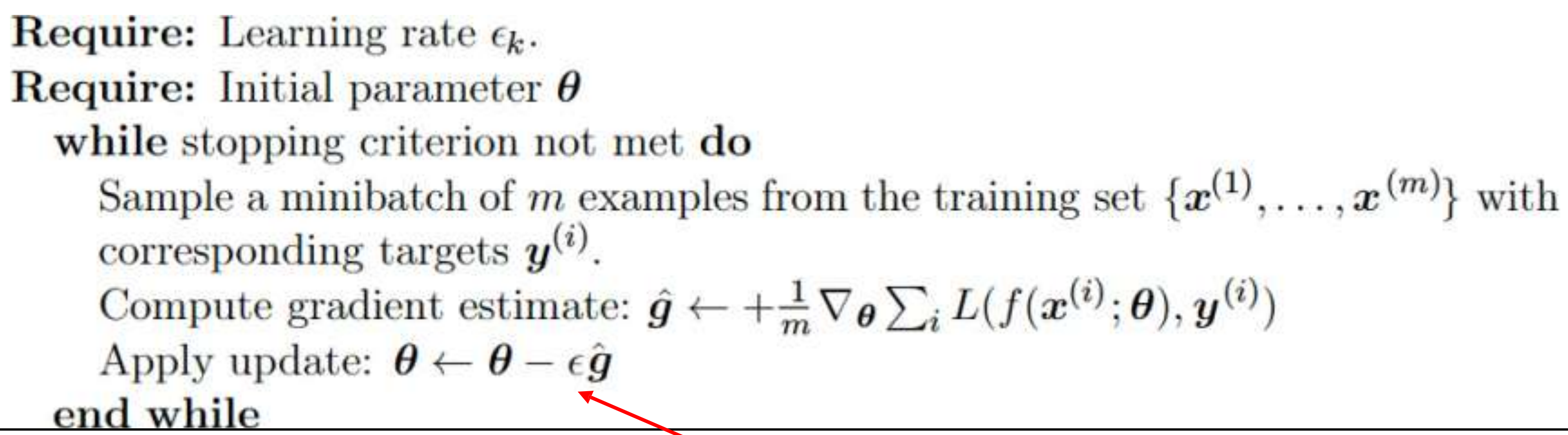


Criterion $f(x)$ minimized by moving from current solution in direction of the negative of gradient

- SGD: Accelerated by minibatches downhill
  - Wide use for ML in general and for
  - Average gradient on a minibatch is an estimate of the gradient

# SGD follows gradient estimate downhill

**Algorithm:** SGD update at training iteration $k$

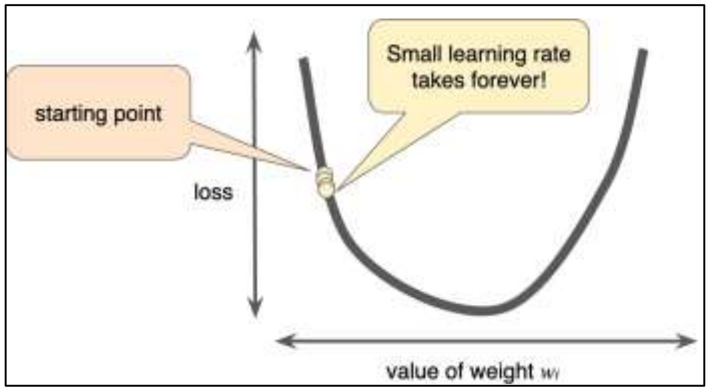


```
Require: Learning rate $\epsilon_k$.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    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
```

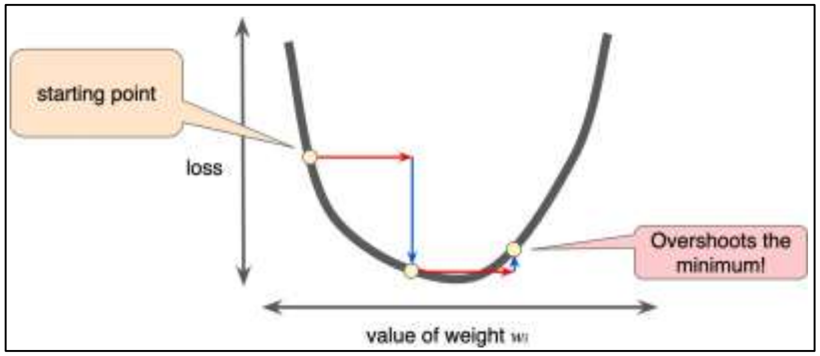
A crucial parameter is the learning rate $\varepsilon$

At iteration $k$ it is $\varepsilon_k$

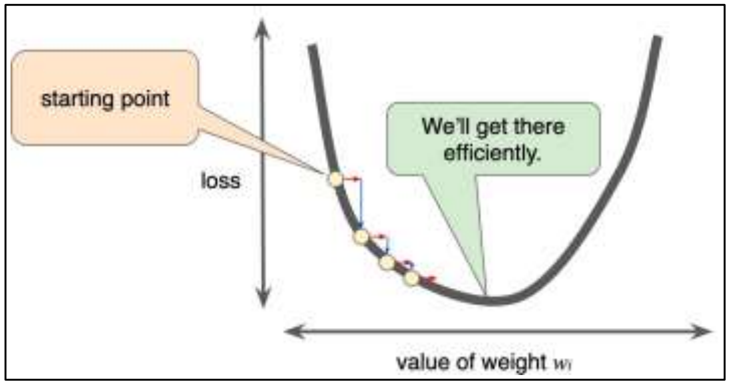
# Choice of learning rate



Too small learning rate
will take too long



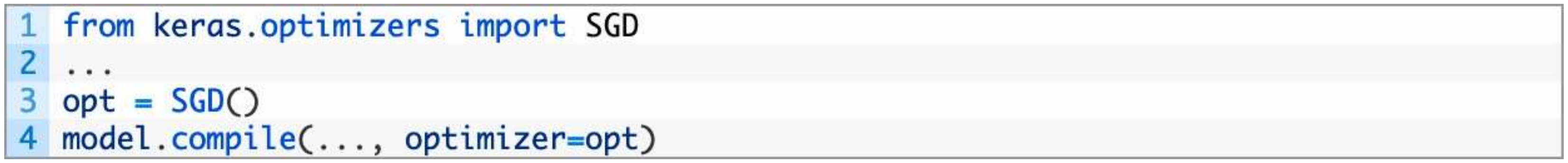
Too large, the next point will
perpetually bounce haphazardly
across the bottom of the well



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

## Learning rate in Keras

- Keras provides SGD class to implement SGD optimizer with learning rate and momentum
    - The default learning rate is 0.01 and no momentum used



```python
from keras.optimizers import SGD
...
opt = SGD()
model.compile(..., optimizer=opt)
```

# Need for decreasing learning rate

- True gradient of total cost function
  - Becomes small and then $0$
    - We 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 arrive at a minimum
  - Sufficient condition for SGD convergence

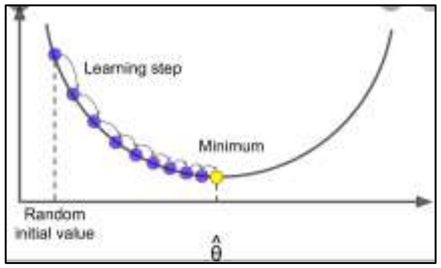
$$\boxed{\sum_{k=1}^{\infty} \varepsilon_k = \infty \qquad \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty}$$

  - Common to decay learning rate linearly until iteration $\tau$: $\varepsilon_k=(1-\alpha)\varepsilon_0+\alpha\varepsilon_\tau$ with $\alpha=k/\tau$
  - After iteration $\tau$, it is common to leave $\varepsilon$ constant
    - Often a small positive value in the range $0.0$ to $1.0$

# Learning Rate Decay

- Decay learning rate

  $\tau$: $\varepsilon_k=(1-\alpha)\varepsilon_0+\alpha\varepsilon_\tau$ 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

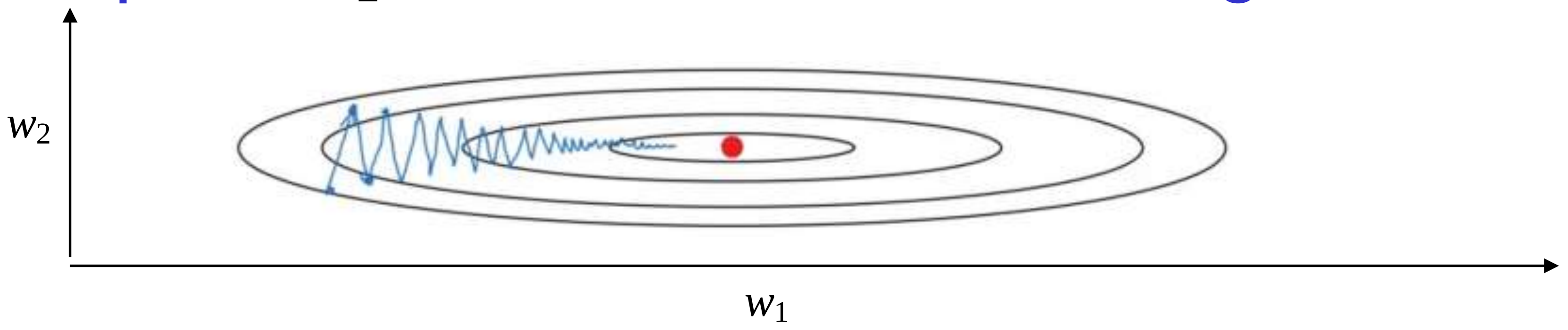
```
from keras.optimizers import SGD
...
opt = SGD(lr=0.01, momentum=0.9, decay=0.01)
model.compile(..., optimizer=opt)
```

## 2. The Momentum method

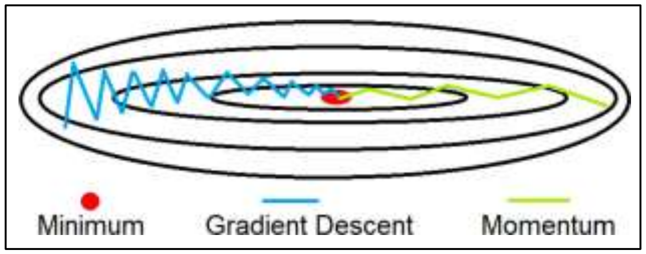
- SGD is a popular optimization strategy
- But it can be slow
- Momentum method accelerates learning, when:
  - Facing high curvature
  - Small but consistent gradients
  - Noisy gradients
- Algorithm accumulates moving average of past gradients and move in that direction, while exponentially decaying

# 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



- We can set a higher learning rate

## Momentum definition

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

# Momentum update rule

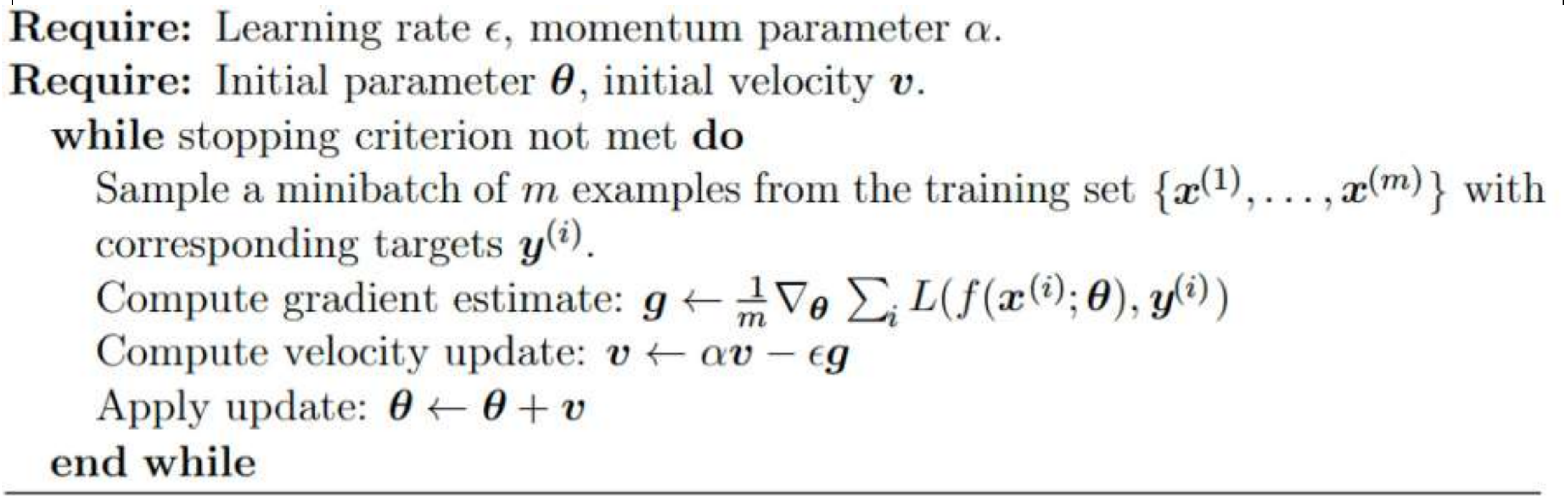
- The update rule is given by

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

- The velocity $\boldsymbol{v}$ accumulates the gradient elements $\boxed{\nabla_{\theta}\left(\frac{1}{m}\sum_{i=1}^{m} L\left(f(x^{(i)};\theta), y^{(i)}\right)\right)}$
- The larger $\alpha$ is relative to $\varepsilon$, the more previous gradients affect the current direction
- The SGD algorithm with momentum is next

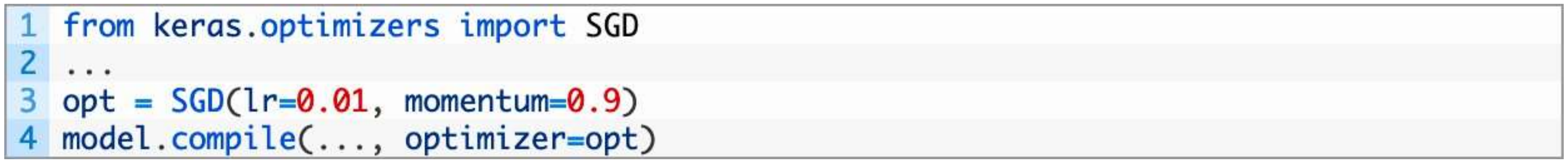
# SGD algorithm with momentum

Algorithm: SGD with momentum



```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$.
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 $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ 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
```

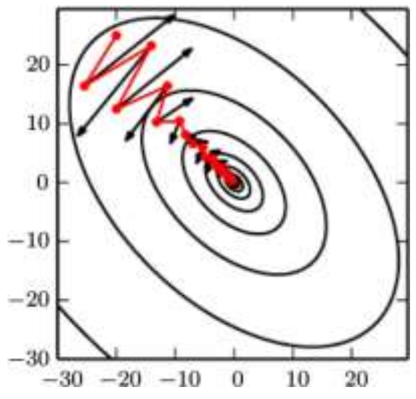
Keras: The learning rate can be specified via the *lr* argument and the momentum can be specified via the *momentum* argument.



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

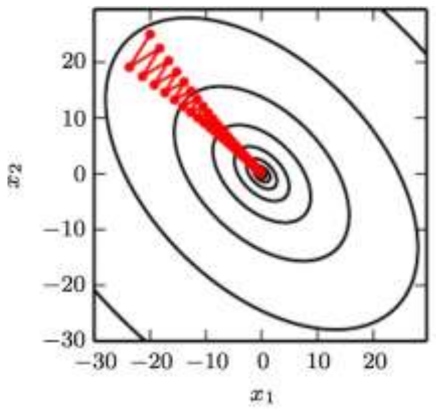
# Momentum

- 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

## 3. 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 $\alpha$ and $\varepsilon$ play a similar role as in the standard momentum method

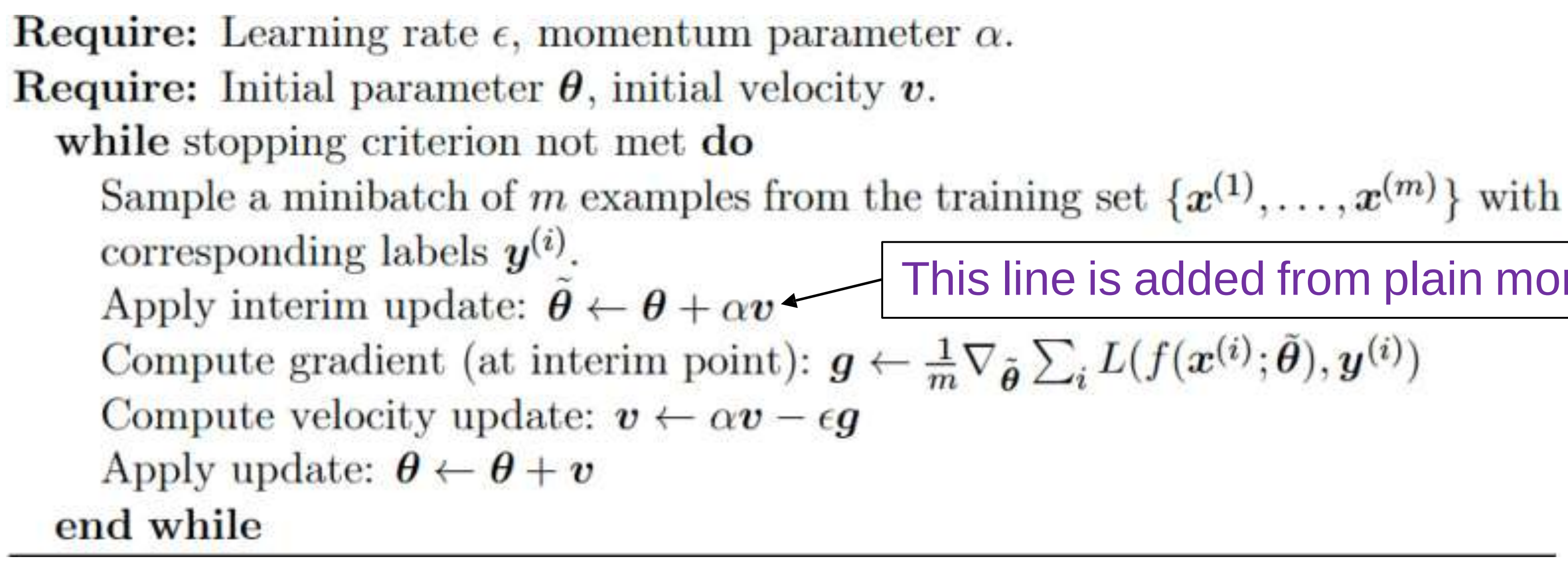
– Difference between Nesterov and standard momentum is where gradient is evaluated.

  - Nesterov gradient is evaluated after the current velocity is applied.
  - Thus one can interpret Nesterov as attempting to add a correction factor to the standard method of 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

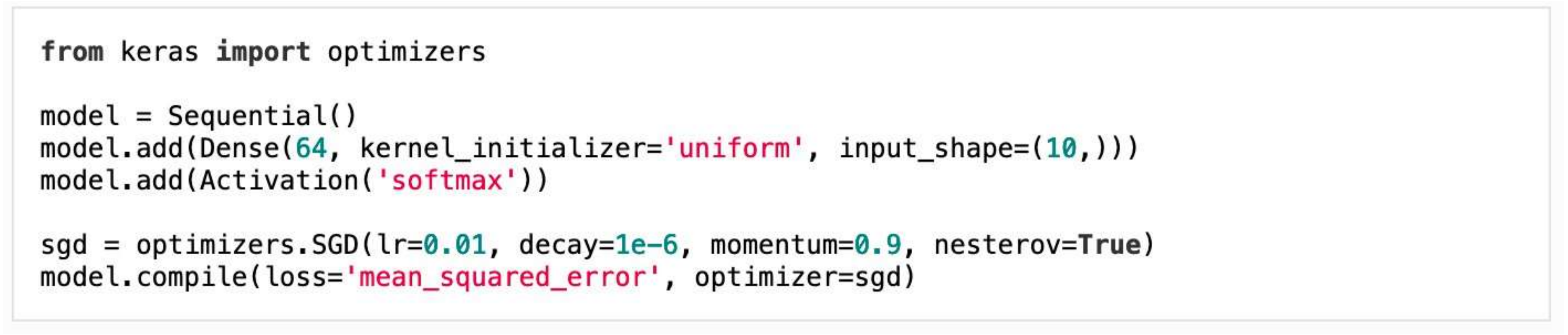


```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 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

## Keras SGD with Nesterov Momentum

- An optimizer is one of the two arguments required for compiling a Keras model:



```
from keras import optimizers

model = Sequential()
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(Activation('softmax'))

sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

**Arguments**
**learning_rate**: float >= 0. Learning rate.
**momentum**: float >= 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
**nesterov**: boolean. Whether to apply Nesterov momentum.

# Parameter Initialization Strategies

## Topics

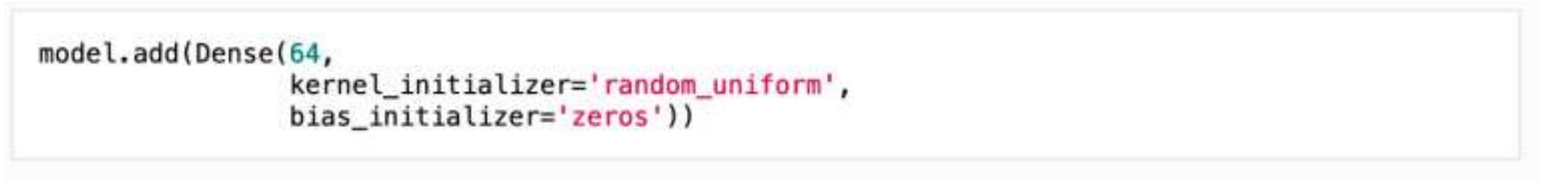
- Importance of Optimization in machine learning
- How learning differs from optimization
- Challenges in neural network optimization
- Basic Optimization Algorithms
  - SGD, Momentum, Nesterov Momentum
- Parameter initialization strategies
- Algorithms with adaptive learning rates
- Approximate second-order methods
- Optimization strategies and meta-algorithms

## Role of Initialization

1. Non-iterative optimization requires no initialization
   - Simply solve for solution point
2. Iterative but converge regardless of initialization
   - Acceptable solutions in acceptable time
3. Iterative but affected by choice of Initialization
   - training algorithms are iterative
     - Initialization determines whether it converges at all
     - Can determine how quickly learning converges

# Keras Initialization

- Initializations define the way to set the initial random weights of Keras layers
- The keyword arguments used for passing initializers to layers will depend on the layer
- Usually it is simply kernel_initializer and bias_initializer:



```
model.add(Dense(64,
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
```

# Available Initializers in Keras

- Zeros `keras.initializers.Zeros()`
- Ones `keras.initializers.Ones()`
- Constant `keras.initializers.Constant(value=0)`
- Random Normal `keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)`
- Random Uniform `keras.initializers.RandomUniform(minval=-0.05, maxval=0.05, seed=None)`
- Truncated Normal `keras.initializers.TruncatedNormal(mean=0.0, stddev=0.05, seed=None)`
- Variance Scaling `keras.initializers.VarianceScaling(scale=1.0, mode='fan_in', distribution='normal', seed=None)`
- Orthogonal `keras.initializers.Orthogonal(gain=1.0, seed=None)`
- Identity `keras.initializers.Identity(gain=1.0)`
- Lecun_uniform `keras.initializers.lecun_uniform(seed=None)`

## Modern Initialization Strategies

- They are simple and heuristic
- Based on achieving nice properties
- But problem is a difficult one
  - Some initial points are beneficial for optimization but detrimental to generalization

## Known property: Break Symmetry

- Only property known with certainty: Initial parameters must be chosen to break symmetry
- If two hidden units have the same inputs and same activation function then they must have different initial parameters
- Usually best to initialize each unit to compute a different function
- This motivates use random initialization of parameters

## Choice of biases

- Biases for each unit are heuristically chosen constants
- Only the weights are initialized randomly
- Extra parameters such as conditional variance of a prediction are constants like biases

## Weights drawn from Gaussian

- Weights are almost always drawn from a Gaussian or uniform distribution
  - Choice of Gaussian or uniform does not seem to matter much but not studied exhaustively
- Scale of the initial distribution does have an effect on outcome of optimization and ability to generalize
  - Larger initial weights will yield stronger symmetry-breaking effect, helping avoid redundant units
  - Too large may result in exploding values

# Heuristics for initial scale of weights

- One heuristic is to initialize the weights of a fully connected layer with $N_{in}$ inputs and $N_{out}$ outputs by sampling each weights from $\mathrm{Uniform}(-r, r)$ where $r = \frac{1}{\sqrt{N_{in}}}$
- Another heuristic is normalized initiation with

$$r = \sqrt{\frac{6}{N_{in} + N_{out}}}$$

  - Which is a compromise between the goal of initializing all layers to have the same *activation* variance and the goal of having all layers having the same *gradient* variance

## Initialization for the biases

- Bias settings must be coordinated with setting weights
- Setting biases to zero is compatible with most weight initialization schemes
- Situations for nonzero biases:
  - Bias for an output unit: initialize to obtain right marginal statistics for output
    - Set bias to inverse of activation function applied to the marginal statistics of the output in the training set
  - Choose bias to causing too much saturation at initialization

## Algorithms with Adaptive Learning Rates

# Topics

- Importance of Optimization in machine learning

1. How learning differs from optimization
2. Challenges in neural network optimization
3. Basic Optimization Algorithms
   - SGD, Momentum, Nesterov Momentum
4. Parameter initialization strategies
5. Algorithms with adaptive learning rates
   1. AdaGrad
   2. RMSProp
   3. Adam
   4. Choosing the right optimization algorithm
6. Approximate second-order methods
7. Optimization strategies and meta-algorithms

## Learning Rate is Crucial

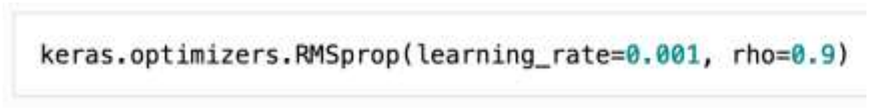
- Learning rate: most difficult hyperparam to set
- It significantly affects model performance
- Cost is highly sensitive to some directions in parameter space and insensitive to others
  - Momentum helps but introduces another hyperparameter
  - Is there another way?
    - If direction of sensitivity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

## Heuristic Approaches

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

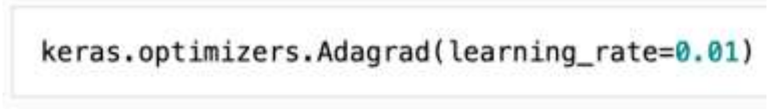
## Keras Adaptive Optimizers

- RMSprop



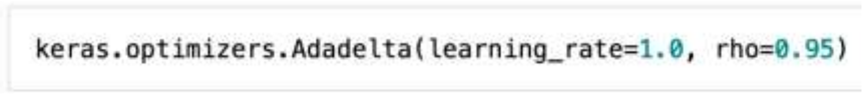
```
keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)
```

- Adagrad



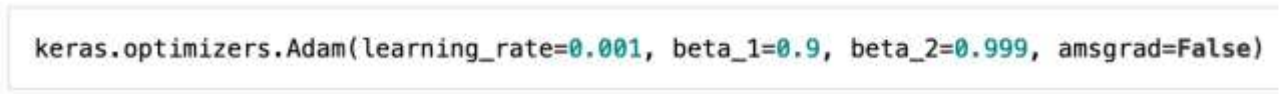
```
keras.optimizers.Adagrad(learning_rate=0.01)
```

- Adadelta



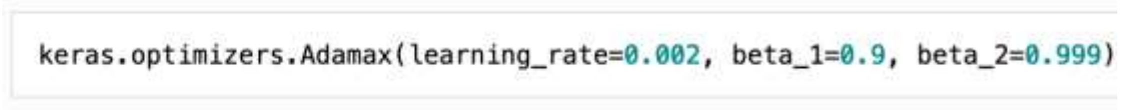
```
keras.optimizers.Adadelta(learning_rate=1.0, rho=0.95)
```

- Adam



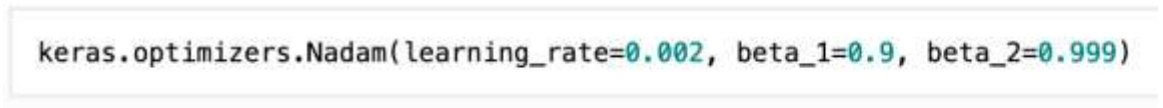
```
keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=False)
```

- Adamax



```
keras.optimizers.Adamax(learning_rate=0.002, beta_1=0.9, beta_2=0.999)
```

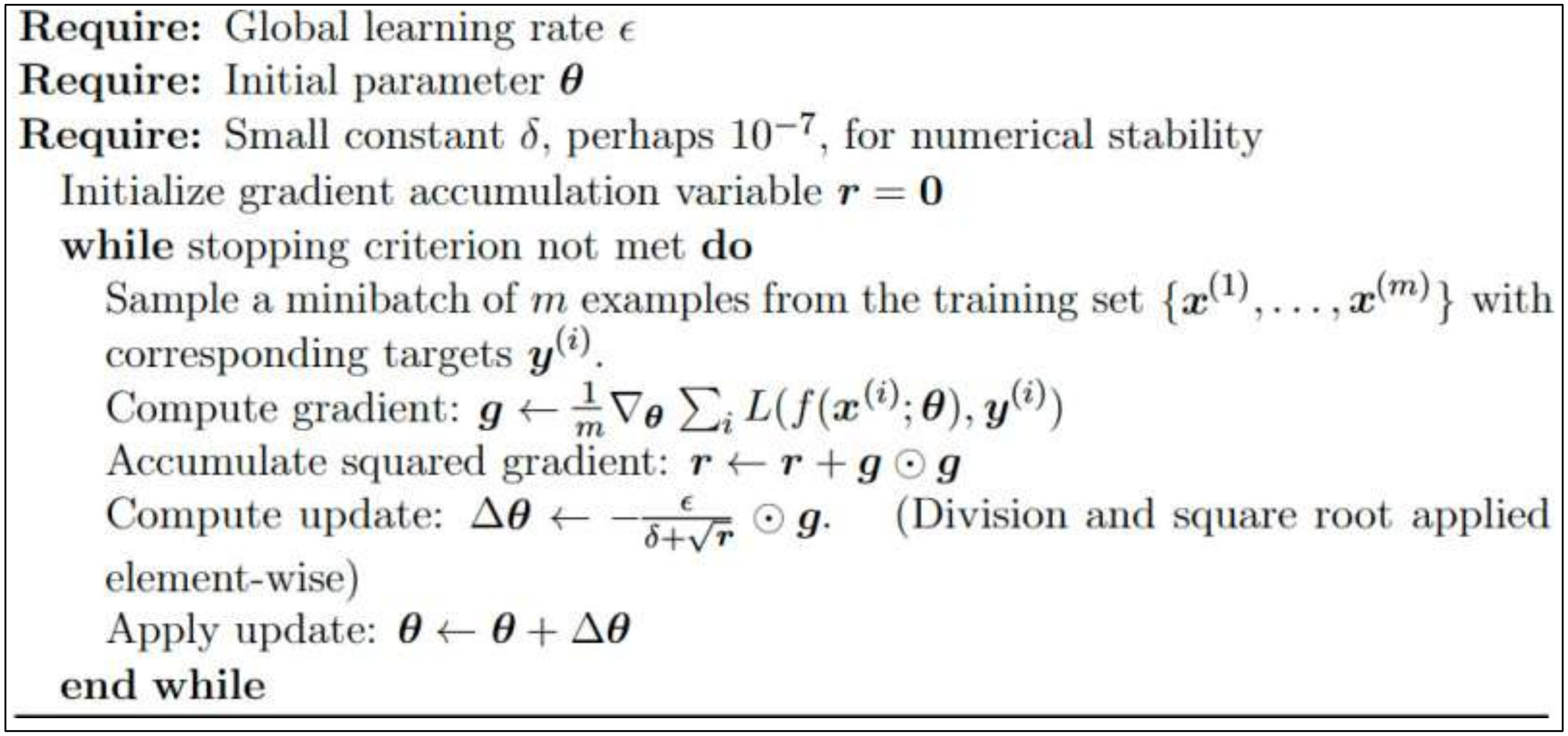
- Nadam



```
keras.optimizers.Nadam(learning_rate=0.002, beta_1=0.9, beta_2=0.999)
```

# AdaGrad

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



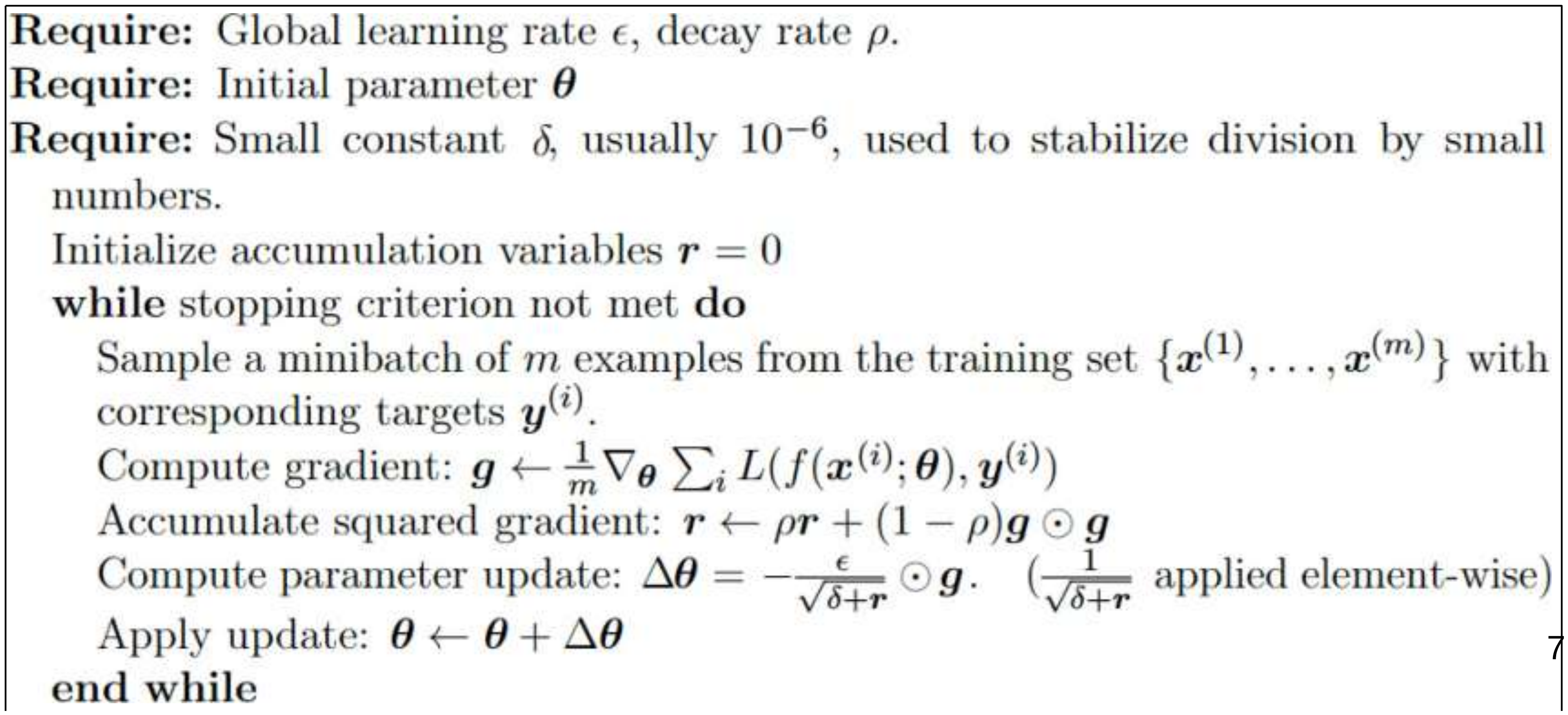
```
Require: Global learning rate $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ 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

## RMSProp

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

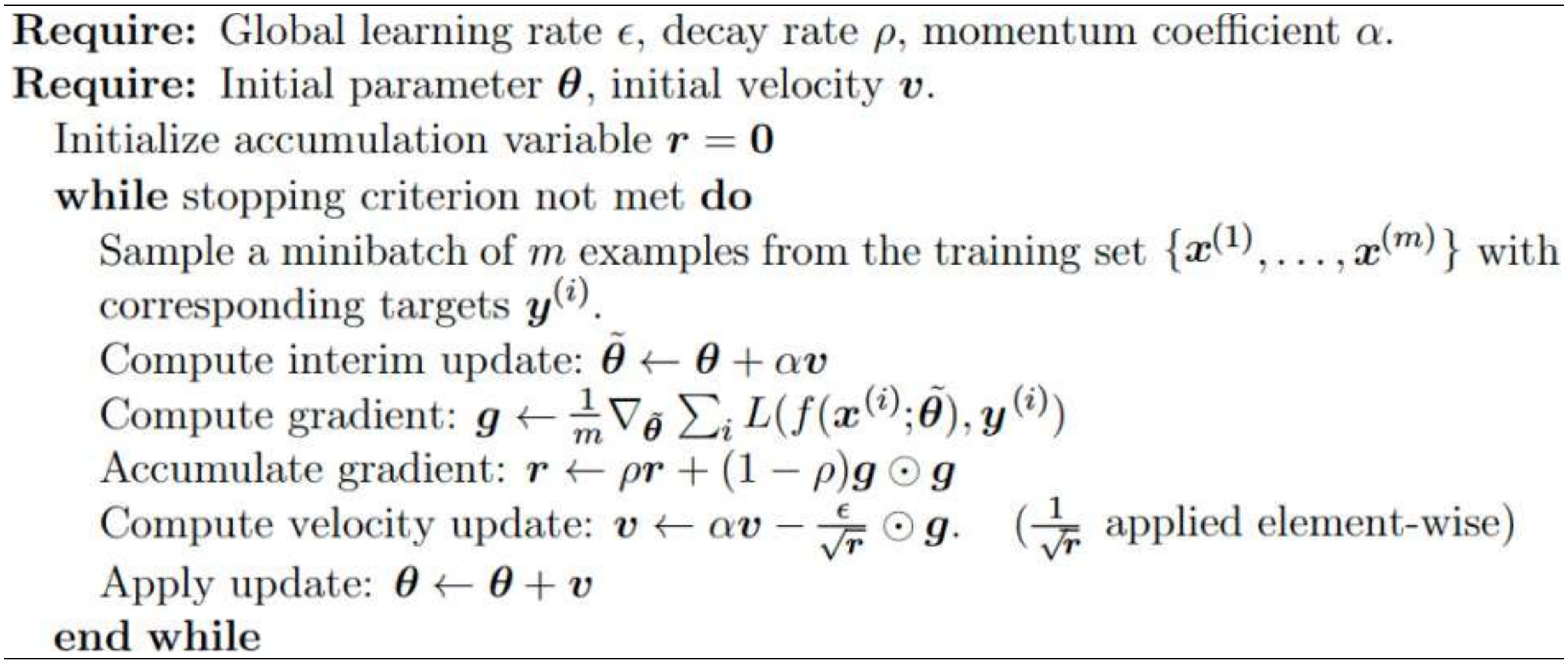
The RMSProp Algorithm



```
Require: Global learning rate $\epsilon$, decay rate $\rho$.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, 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
```

## RMSProp combined with Nesterov

Algorithm: RMSProp with Nesterov momentum



```
Require: Global learning rate $\epsilon$, decay rate $\rho$, momentum coefficient $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  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

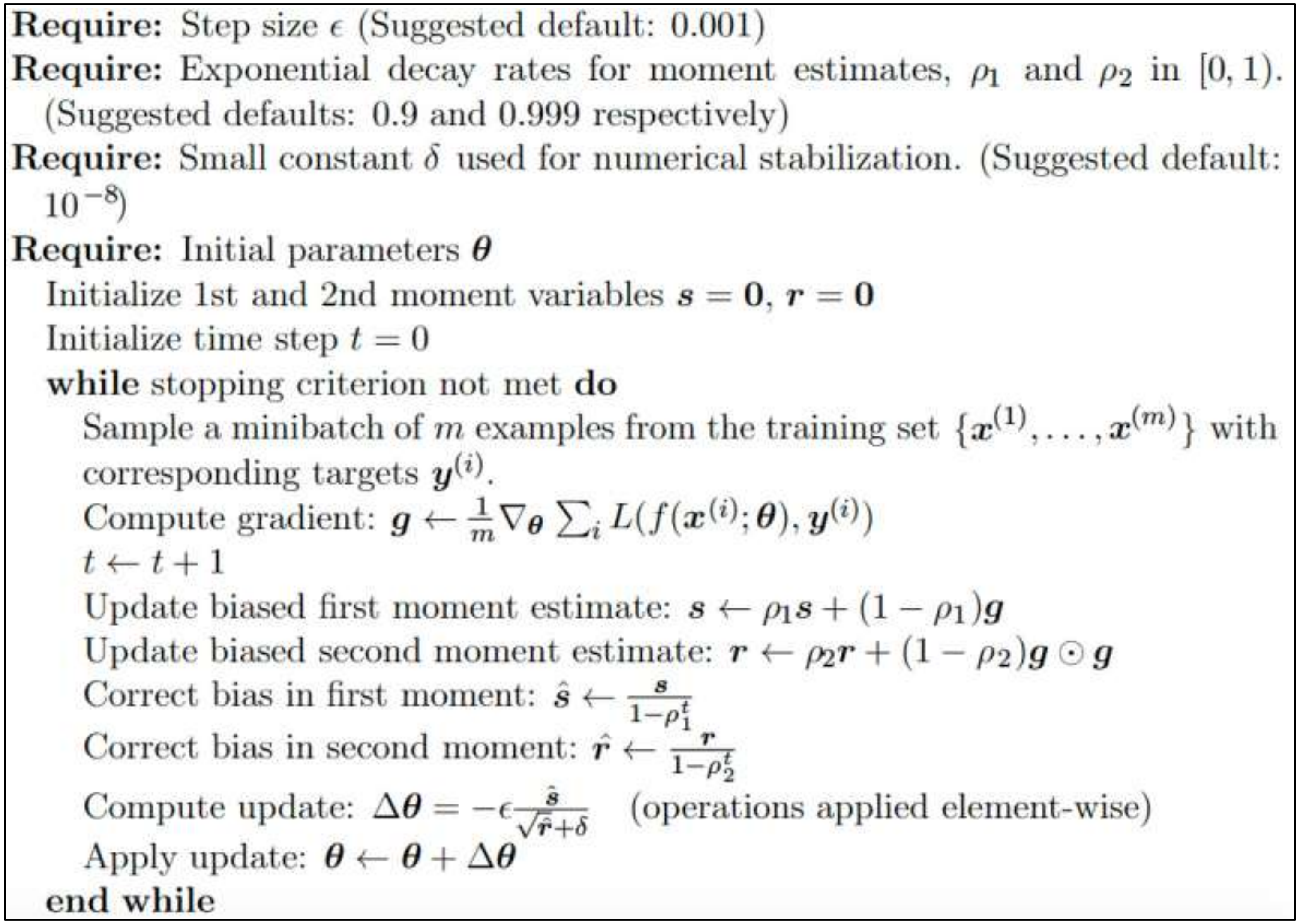
- RMSProp is an effective practical optimization algorithm
- Go-to optimization method for  practitioners

## Adam: Adaptive Moments

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

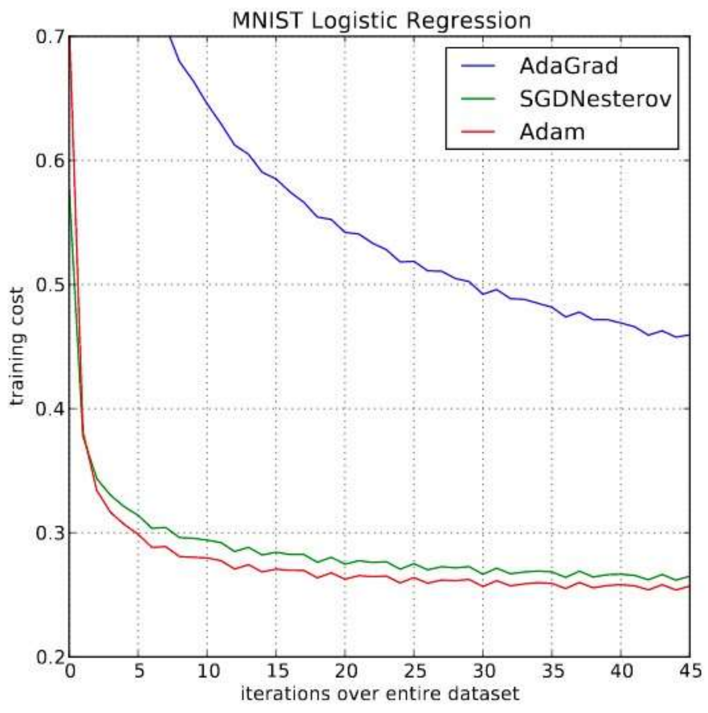
## The Adam Optimizer

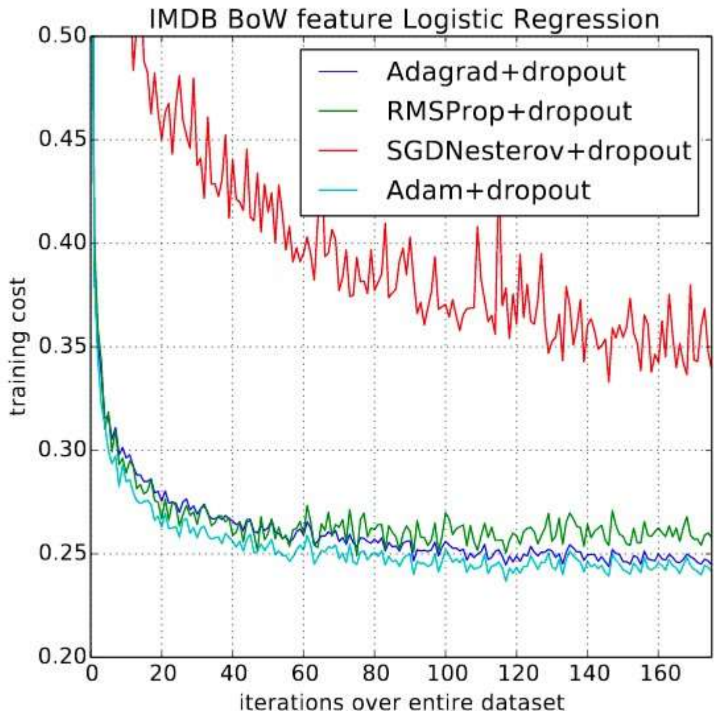
### The Adam Algorithm



```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization. (Suggested default:
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ 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
```

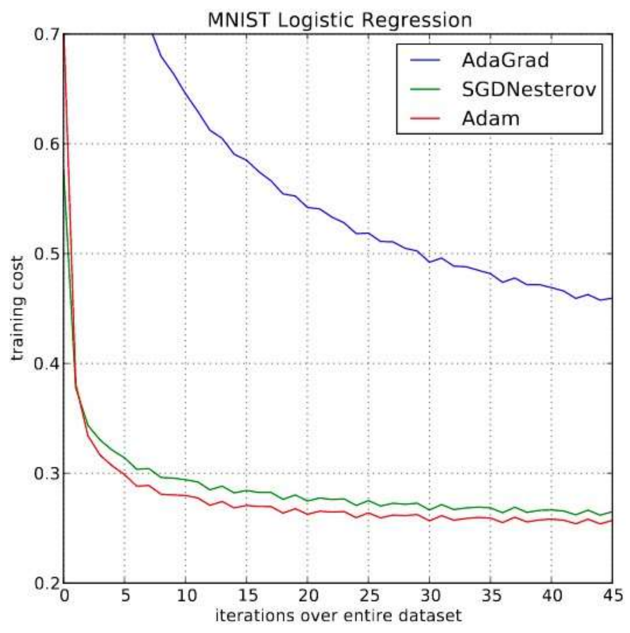
# Performance in Logistic Regression

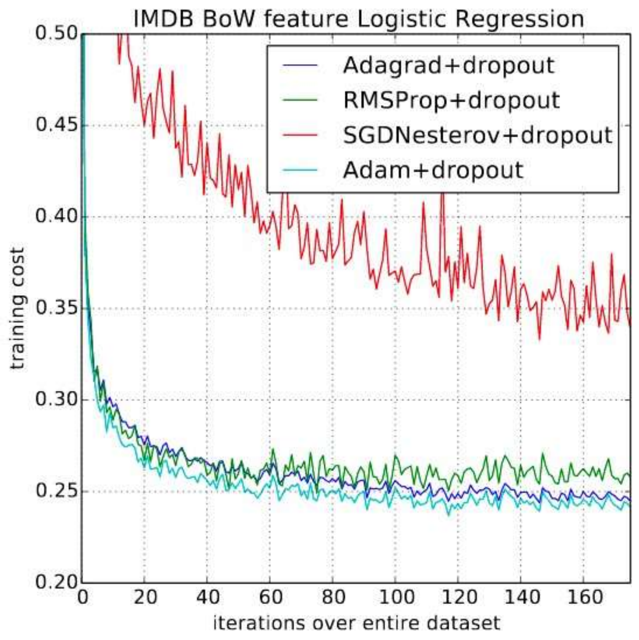




Logistic regression training negative log likelihood on MNIST images and IMDB movie reviews with 10,000 bag-of-words (BoW) feature vectors.

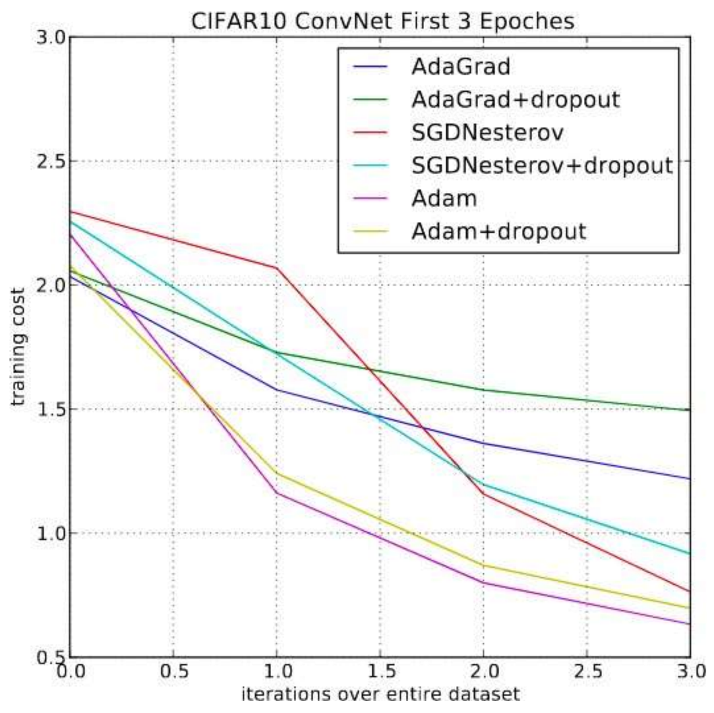
# Performance on Multilayer NN

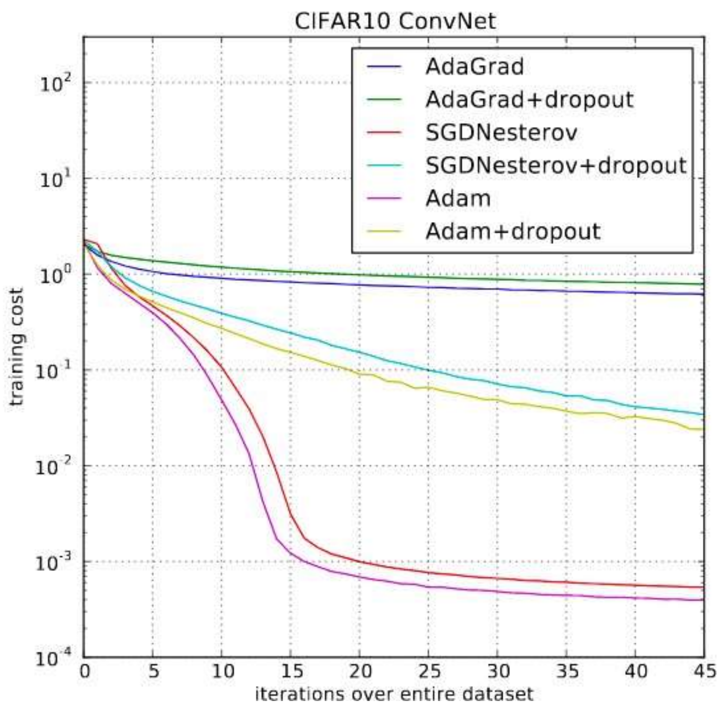




Training of multilayer neural networks on MNIST images. (a) Neural networks using dropout stochastic regularization. (b) Neural networks with deterministic cost function.

# 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.

## 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?
  - There is 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