ECBM E4040
Neural Networks and Deep Learning

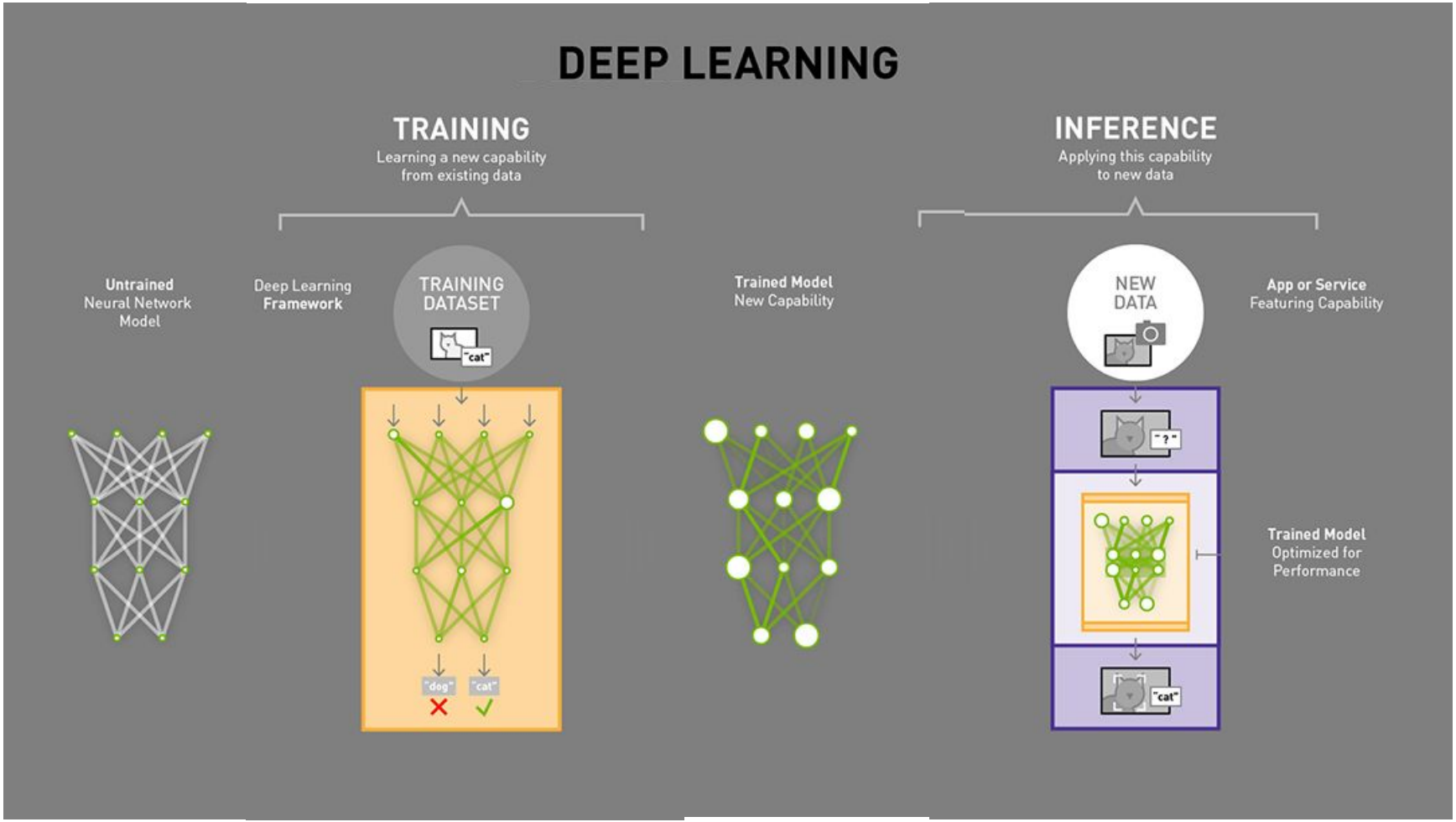
Optimization for Deep Learning

Zoran Kostić
Columbia University
Electrical Engineering Department
& Data Sciences Institute

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

References and Acknowledgments

- Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville, http://www.deeplearningbook.org/ , chapters 4 and 8.
- Lecture material by bionet group / Prof. Aurel Lazar (http://www.bionet.ee.columbia.edu/).
- Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).



https://blogs.nvidia.com/blog/2016/08/22/difference-deep-learning-training-inference-ai/

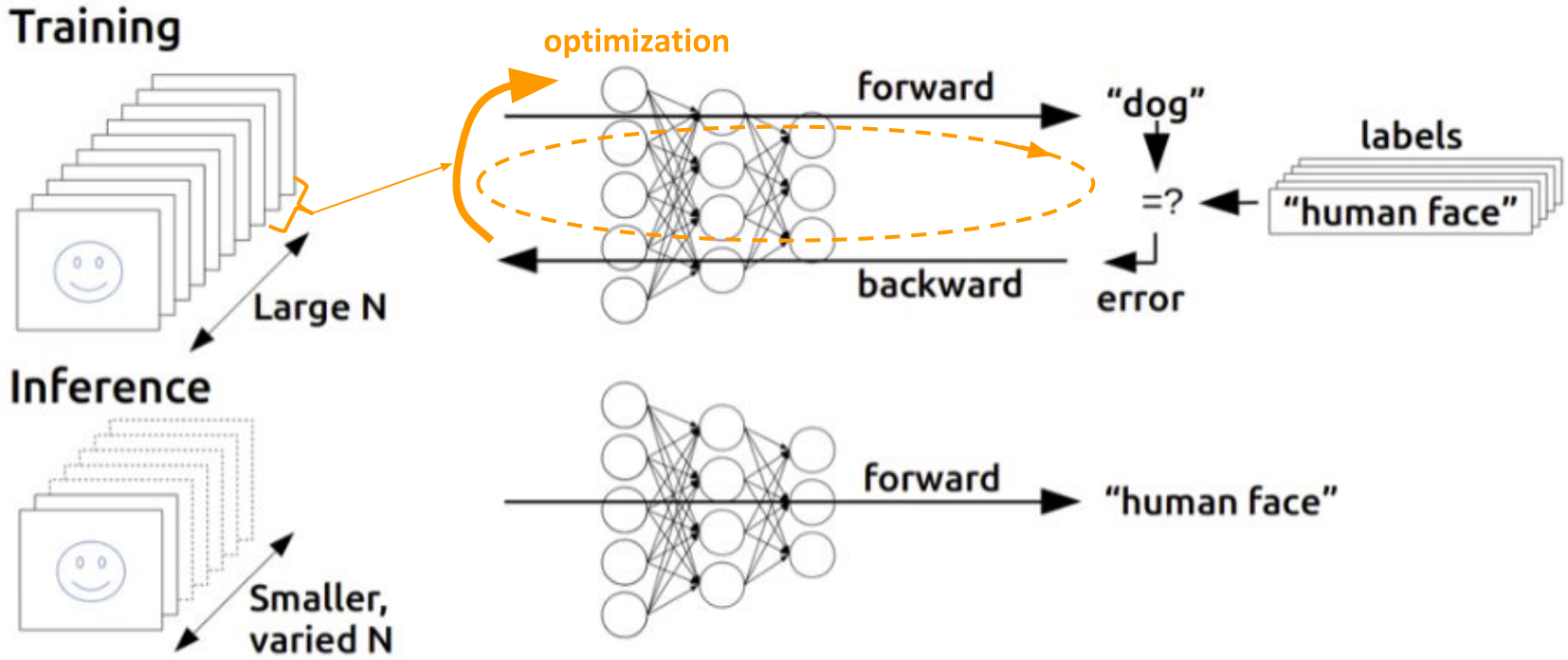


Figure 1: Deep learning training compared to inference. In training, many inputs, often in large batches, are used to train a deep neural network. In inference, the trained network is used to discover information within new inputs that are fed through the network in smaller batches.

https://devblogs.nvidia.com/parallelforall/inference-next-step-gpu-accelerated-deep-learning/

Iterative Learning Process

Given sample points $(x^{(i)}, y^{(i)})$

Build the model which uses coefficients θ

Define the loss function $$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Compute the gradient $$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)}$$

Update the coefficients $$\theta(t+1) = \theta(t) - \alpha \nabla_\theta J(\theta(t))$$

OPTIMIZATION

Optimization for Deep Learning

Searching for Minima

Gradient Descent Methods

Optimization in Deep Learning

Most deep learning algorithms involve the minimization or maximization of a function called objective function or criterion. In minimization, these functions are also called cost functions, loss functions or error functions.

The value that minimizes or maximizes a function is denoted by a superscript $*$:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x}).$$

Assume that $y = f(x)$ with $x, y \in \mathbb{R}$. Points where

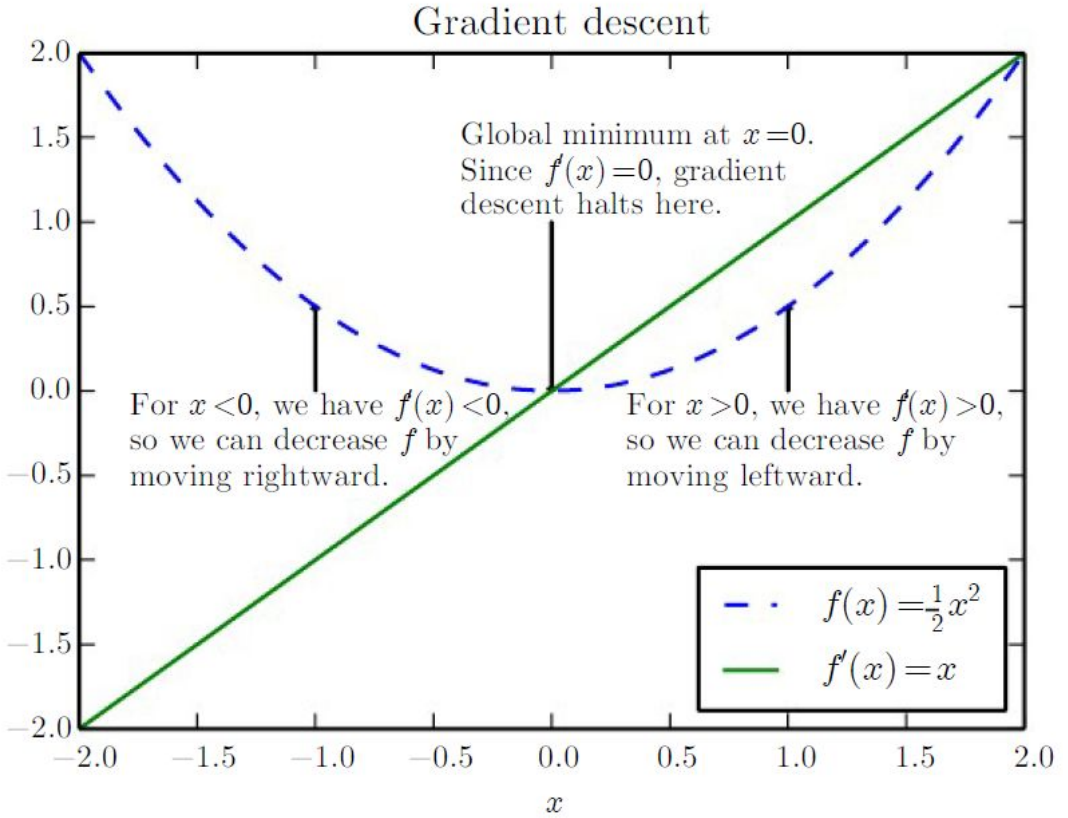
$$\frac{df(x)}{dx} = 0$$

are called critical or stationary points. These can be local minima, local maxima, saddle points, or global minima/maxima.

Optimization - Gradient Descent Method

Goal: Find minimum

Method: Use the derivative of the function to follow the function downhill to a minimum.



Optimization - Gradient Descent for Functions with Multiple Inputs

Here we consider functions of the form $f : \mathbb{R}^n \to \mathbb{R}$. The partial derivative $\frac{\partial}{\partial x_i} f(\mathbf{x})$ measures how f changes as a function of x_i at point $\mathbf{x}$. The gradient generalizes to the notion of derivative with respect to a vector: the gradient of f is the vector containing all the partial derivatives, denoted by $\nabla f(\mathbf{x})$.

The directional derivative in direction $\mathbf{u}$ (a unit vector) is the slope of the function f in direction $\mathbf{u}$, i.e., the derivative of the function $f(\mathbf{x} + \alpha\mathbf{u})$ with respect to α, evaluated at $\alpha = 0$.

Using the chain rule, the directional derivative amounts to

$$\mathbf{u}^T \nabla f(\mathbf{x}).$$

Optimization - Gradient Descent for Functions with Multiple Inputs

To minimize f, we have to find the direction in which f decreases the fastest. Using the directional derivative we have

$$\min_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} \mathbf{u}^T \nabla f(\mathbf{x}) = \min_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} ||\mathbf{u}||_2 ||\nabla f(\mathbf{x})||_2 \cos\theta,$$

where θ is the angle between $\mathbf{u}$ and the gradient. Since $||\mathbf{u}||_2 = 1$ the minimization above can be reduced to

$$\min_{\mathbf{u}} \cos\theta.$$

The minimum of the cos function above is -1 and is achieved when $\mathbf{u}$ points in the opposite direction of the gradient.

Optimization - Gradient Descent
The Method of Steepest Descent

Steepest descent chooses a new point

$$\mathbf{x}' = \mathbf{x} - \varepsilon \nabla f(\mathbf{x}),$$

where ε is the size of the step. There are many ways to choose ε (it is an art ...).

Constrained Optimization
Karush-Kuhn-Tucker Approach

Often, we may wish to find the maximal or minimal value of $f(\mathbf{x})$ for values of $\mathbf{x}$ in some set $\mathbb{S}$. This is known as constrained optimization. Points $\mathbf{x}$ that lie within the set $\mathbb{S}$ are called feasible points in constrained optimization terminology.

A very general solution to constrained optimization problem above is provided by the KarushKuhnTucker (KKT) approach. The KKT approach, is based on introducing a new function called the generalized Lagrangian or generalized Lagrange function.

Constrained Optimization
Karush-Kuhn-Tucker Approach

To define the Lagrangian, we'll first describe $\mathbb{S}$ in terms of equations and inequalities. We want a description of $\mathbb{S}$ in terms of m functions g_i and n functions h_j so that

$$\mathbb{S} = \{\mathbf{x} | \forall i, g_i(\mathbf{x}) = 0 \text{ and } \forall j, h_j(\mathbf{x}) \leq 0\}.$$

The equations involving g_i are called the equality constraints and the inequalities involving h_j are called inequality constraints.

Constrained Optimization
Karush-Kuhn-Tucker Approach

The generalized Lagrangian is then defined as

$$L(\mathbf{x}, \lambda, \alpha) = f(\mathbf{x}) + \sum_i \lambda_i g_i(\mathbf{x}) + \sum_j \alpha_j h_j(\mathbf{x}).$$

We solve the constrained minimization problem using unconstrained optimization of the generalized Lagrangian. Observe that, so long as at least one feasible point exists and $f(\mathbf{x})$ is not permitted to have the value ∞, then

$$\min_{\mathbf{x}} \max_{\lambda} \max_{\alpha, \alpha \geq 0} L(\mathbf{x}, \lambda, \alpha),$$

has the same optimal objective function value and set of optimal points $\mathbf{x}$ as

$$\min_{\mathbf{x} \in \mathbb{S}} f(\mathbf{x}).$$

Constrained Optimization
Karush-Kuhn-Tucker Approach

The above follows because any time the constraints are satisfied,

$$\max_{\lambda} \max_{\alpha, \alpha \geq 0} L(\mathbf{x}, \lambda, \alpha) = f(\mathbf{x}),$$

while any time a constraint is violated,

$$\max_{\lambda} \max_{\alpha, \alpha \geq 0} L(\mathbf{x}, \lambda, \alpha) = \infty,$$

These properties guarantee that no infeasible point will ever be optimal, and that the optimum within the feasible points is unchanged.

Gradient-Based Optimization for Unconstrained Least Squares Problems

Find the value of $\mathbf{x}$ that minimizes

$$f(\mathbf{x}) = \frac{1}{2}||\mathbf{Ax} - \mathbf{b}||_2^2.$$

To apply the gradient-based optimization, we derive the gradient

$$\nabla f(\mathbf{x}) = \mathbf{A}^T(\mathbf{Ax} - \mathbf{b}) = \mathbf{A}^T\mathbf{Ax} - \mathbf{A}^T\mathbf{b}.$$

We now just apply the standard steepest decent algorithm

$$\mathbf{x}' = \mathbf{x} - \varepsilon(\mathbf{A}^T\mathbf{Ax} - \mathbf{A}^T\mathbf{b}).$$

Gradient-Based Optimization for Constrained Least Squares Problems

We will minimize the same function $f(\mathbf{x})$, but subject to the constraint $\mathbf{x}^T\mathbf{x} \leq 1$. We introduce the Lagrangian

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda(\mathbf{x}^T\mathbf{x} - 1).$$

We can now solve the problem

$$\min_{\mathbf{x}} \max_{\lambda, \lambda \geq 0} L(\mathbf{x}, \lambda).$$

The solution to the unconstrained least squares problem is given by $\mathbf{x} = \mathbf{A}^+\mathbf{b}$. If this point is feasible, then it is the solution to the constrained problem. Otherwise, we must find a solution where the constraint is active. By differentiating the Lagrangian with respect to $\mathbf{x}$, we obtain the equation

$$\mathbf{A}^T\mathbf{A}\mathbf{x} - \mathbf{A}^T\mathbf{b} + 2\lambda\mathbf{x} = 0.$$

Gradient-Based Optimization for Constrained Least Squares Problems

This tells us that the solution will take the form

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A} + 2\lambda\mathbf{I})^{-1}\mathbf{A}^T\mathbf{b}.$$

The magnitude of λ must be chosen such that the result obeys the constraint. We can find this value by performing gradient ascent on λ. To do so, observe that

$$\frac{\partial}{\partial\lambda}L(\mathbf{x},\lambda) = \mathbf{x}^T\mathbf{x} - 1.$$

When the norm of $\mathbf{x}$ exceeds 1, this derivative is positive, so to ascend the gradient and increase the Lagrangian with respect to λ, we increase λ. This will in turn shrink the optimal $\mathbf{x}$. The process continues until $\mathbf{x}$ has the correct norm and the derivative on λ is 0.

Optimization for Deep Learning

Optimization for Model Training

Challenges in Neural Network Optimization

Basic Learning Algorithms

Algorithms with Adaptive Learning Rates

Optimization - Empirical Risk Minimization

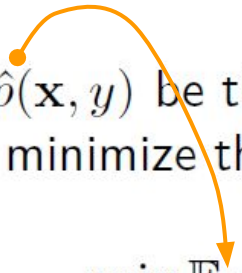
Assume that the input feature vector $\mathbf{x}$ and targets y, sampled from some unknown joint distribution $p(\mathbf{x}, y)$, as well as some loss function $L(f(\mathbf{x}; \boldsymbol{\theta}, y)$. Our ultimate goal is to minimize the risk

$$\min \mathbb{E}_p L(f(\mathbf{x}; \boldsymbol{\theta}), y).$$

The expectation is taken over the true underlying distribution p , so the risk is a form of generalization error. If we knew the true distribution $p(\mathbf{x}, y)$, this would be an optimization task solvable by an optimization algorithm. However, when we do not know $p(\mathbf{x}, y)$ but only have a training set of samples from it, we have a machine learning problem.

Solution: Minimize the expected loss on the training set.

Optimization - Empirical Risk Minimization



Let $\hat{p}(\mathbf{x}, y)$ be the empirical distribution on the training set. We now minimize the empirical risk

$$\min \mathbb{E}_{\hat{p}} L(f(\mathbf{x}; \boldsymbol{\theta}), y) = \min \frac{1}{m} \sum_{i=1}^{m} L(f(\mathbf{x}^i; \boldsymbol{\theta}), y^i),$$

where m is the number of training examples.

Optimization - Empirical Risk Minimization

Note that rather than optimizing the risk directly, we optimize the empirical risk, and hope that the risk decreases significantly as well. However,empirical risk minimization is rarely used because:

- it is prone to overfitting; models with high capacity can simply memorize the training set.
- in many cases it is not feasible; many useful loss functions, such as $0-1$ loss, have no useful derivatives (the derivative is either zero or undefined everywhere).

Deep Learning Algorithms

Deep learning algorithms depend on the values of model parameters (weights/coefficients).

Model parameters are established (calculated) using the input data and corresponding labels (output values associated with particular inputs):

- by calculating the loss (measure of error)
- minimizing the loss

The back-propagation algorithm (backprop) efficiently computes the gradient of the loss with respect to the model parameters.

The learning algorithm utilizes the gradients of the loss to (re-) calculate the parameters.

Optimization Algorithms
(A) Batch Algorithm and (B) Stochastic Algorithms

(A) Optimization algorithms that use the entire training set are called batch or deterministic gradient methods, because they process all of the training examples simultaneously in a large batch.

- **with every new input data (training example), all of the previous examples are reused for recalculation**

(B) Optimization algorithms that use only a single (or a subset) of examples at a time are sometimes called stochastic or online methods

- **only the new input data is used for recalculation**

Optimization Algorithms
(B^{++}) Mini-Batch Algorithm

(B^{++}) Most algorithms used for deep learning fall somewhere between batch algorithms and (true) stochastic algorithms. They use more than one but less than all of the training examples, for every recalculation.

- **These were traditionally called minibatch or minibatch stochastic gradient methods**
- **It is now common to call them stochastic gradient methods as well.**

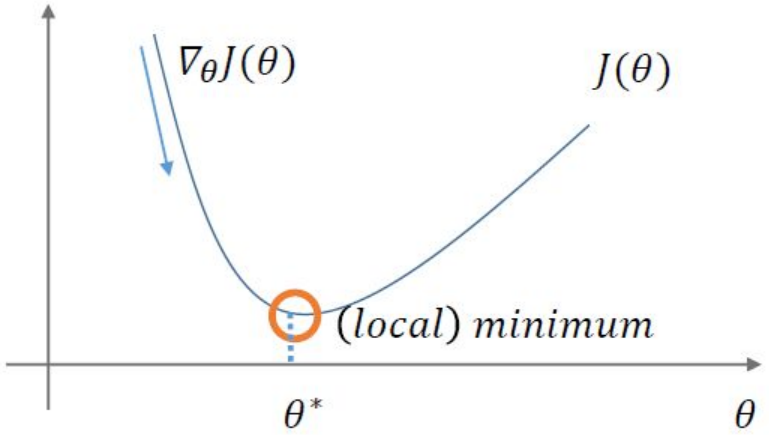
Optimization Algorithms
Gradient Descent Variants

Gradient descent is a way to minimize an objective function $J(\theta)$

- $J(\theta)$: **Objective function**
- $\theta \in Rd$: **Parameters of the model**
- ε: **Learning rate. This determines the size of the steps we take to reach a (local) minimum.**

Update equation

$$\theta = \theta + \varepsilon * \nabla_{\theta} J(\theta)$$



Optimization Algorithms
Gradient Descent Variants

Three variants of gradient descent:

- **Batch gradient descent (deterministic)**
- **Stochastic gradient descent (SGD)**
- **Mini-batch stochastic gradient descent (often called SGD as well)**

The difference between these algorithms is the amount of data used for update

Update equation

$$\theta = \theta + \varepsilon * \nabla_\theta J(\theta)$$

Optimization Algorithms - Batch Gradient Descent

Gradient descent also called batch gradient descent or deterministic gradient descent updates the parameters only after having seen a batch of all the training examples. The gradient is computed exactly and deterministically.

The algorithm calls for updating the model parameters $\boldsymbol{\theta}$ (the weights and biases) with a small step in the direction of the gradient of the objective function that includes the terms of all the training examples. For the case of supervised learning with data pairs $[\mathbf{x}^t, \mathbf{y}^t]$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \nabla_{\boldsymbol{\theta}} \sum_t L(f(\mathbf{x}^t; \boldsymbol{\theta}), \mathbf{y}^t),$$

where ϵ is the learning rate, an optimization hyperparameter that controls the size of the step the parameters take in the direction of the gradient.

Optimization Algorithms
(Batch) Gradient Descent

The gradient descent algorithm guarantees to reduce the loss if ϵ is smaller than some threshold value. If we assume the gradient is Lipschitz continuous, then a fixed step size less than the reciprocal of the Lipschitz constant $\mathcal{L}$ guarantees convergence.

Batch gradient descent is rarely used in machine learning because it does not exploit the particular structure of the objective function, which is written as a large sum of generally i.i.d. terms associated with each training example. Exploiting this structure is what allows stochastic gradient descent to achieve much faster practical convergence.

Optimization Algorithms
Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) and its variants are the most frequently used optimization algorithms for neural networks.

SGD is conceptually similar to the (batch) gradient descent except that it uses a stochastic (i.e., noisy) estimator of the gradient to perform its update. In machine learning, this is typically obtained by sampling one or a small subset of the training examples and computing the corresponding gradient.

Batch gradient descent most often works with a fixed learning rate.

To converge to a minimum, the learning rate of the SGD should decrease at some appropriate rate during training. This is because the SGD gradient estimator introduces noise (m random training examples) which does not become 0 even when we arrive at a minimum (the true gradient becomes small and reaches 0).

Stochastic Gradient Descent (SGD, with mini-batch)

Algorithm 1 Stochastic gradient descent update at training iteration k

Require: Learning rate η_k.

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

while Stopping criterion not met **do**

  Sample a minibatch of m examples from the training set $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

  Set $\hat{\mathbf{g}} = \mathbf{0}$.

  for $i = 1$ to m **do**

    Compute gradient estimate:

    $\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} + \frac{1}{m} \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)$

  end for

  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_k \hat{\mathbf{g}}$

end while

if *m*=1 -> single example SGD

Optimization Algorithms
Stochastic Gradient Descent (SGD), Single Example

This method performs a parameter update for each (one) training example $x^{(i)}$ and label $y^{(i)}$.

Update equation:

$$\theta = \theta + \varepsilon * \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

Note in various literature: ε or η or μ

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params + learning_rate * params_grad
```

Optimization Algorithms
Batch vs. Stochastic Gradient Descent

Batch gradient descent converges to the minimum. The fluctuation is small.

SGD fluctuation is large but it can jump to new local minima.

Mini-batch reduces the variance of the parameter updates compared to SGD.

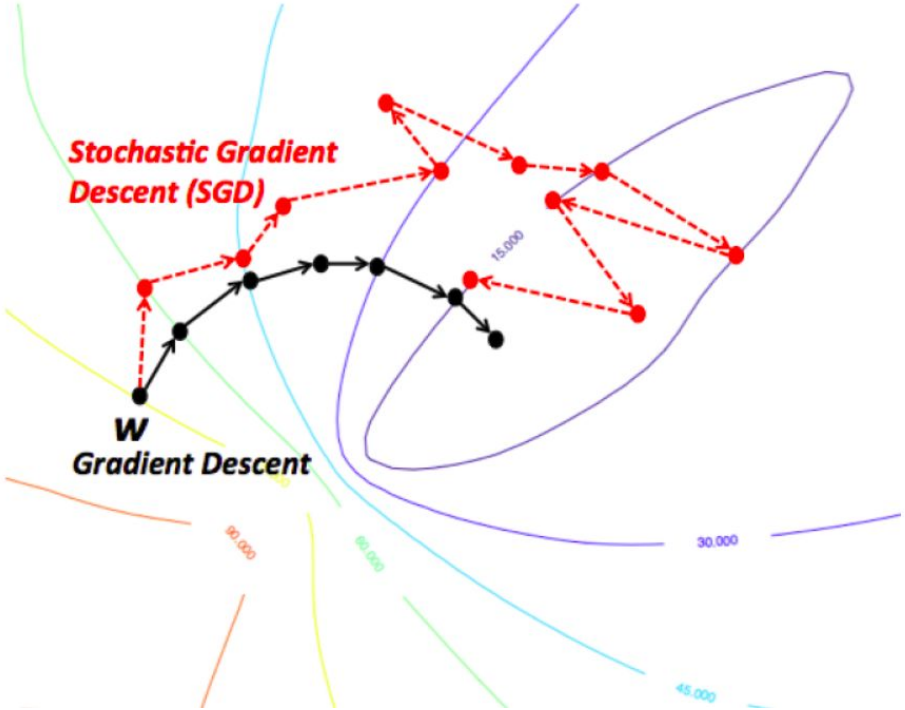


figure https://wikidocs.net/3413

Optimization Algorithms
Mini-Batch Stochastic Gradient Descent

"A compromise" between batch and single example SGD.

Performs an update for every mini-batch of m elements.

Update equation:

$$\theta = \theta + \varepsilon * \nabla_\theta J(\theta; x^{(i:i+m)}; y^{(i:i+m)})$$

Code

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params + learning_rate * params_grad
```

Optimization Algorithm (Granularity)
Mini-Batch Stochastic Gradient Descent

Minibatch sizes are generally driven by the following factors:

- **Larger batches (m) provide a more accurate estimate of the gradient, but with less than linear returns**
- **Can make use of highly optimized matrix optimizations common to deep learning libraries**
- **Multicore GPU architectures are usually underutilized by extremely small batches -> motivation to use some absolute minimum batch size**
- **If batches are processed in parallel, then the amount of memory scales with the batch size. For many hardware setups, this is the limiting factor**

Optimization Algorithm (Granularity)
Mini-Batch Stochastic Gradient Descent

Minibatch sizes are generally driven by the following factors:

- **On GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.**
- **Small batches can offer a regularizing effect. Generalization error is often best for a batch size of 1, though this might take a very long time to train and require a small learning rate to maintain stability.**

Optimization Algorithms
Gradient Descent Learning Rate ε

The choice of learning rate ε impacts the ability to avoid numerous suboptimal local minima for highly non-convex error functions.

Learning rate ε challenges:

- **Choosing the optimal (or just good) learning rate**
  - **large value -> fast convergence & high noise**
  - **small value -> slow convergence & low noise, local minima possibility**
- **Settings of the learning rate schedule**
  - **Adapting the value of the learning rate: time-based, drop-based**
  - **Choosing thresholds when to change ε is hard**
- **Changing the learning rate for each parameter is difficult**

Optimization Algorithms (Granularity) Tradeoffs

Trade-off between:

- **The accuracy of the parameter update**
- **The time it takes to perform an update.**

Method	Accuracy	Time	Memory Usage	Online Learning
Batch gradient descent	high	slow	high	no
Stochastic gradient descent SGD	low	fast	low	yes
Mini-batch SGD	variable	variable	variable	yes

Challenges with Neural Network Optimization

Convexity

Ill-conditioning

Local minima vs. other regions

Challenges in Neural Network Optimization

Optimization in general is an extremely difficult task.

Traditionally, machine learning has avoided the difficulty of general optimization by carefully designing the objective function and constraints to ensure that the optimization problem is made to be convex.

For neural networks, one must confront the general non-convex case.

Note that even convex optimization is not without complications.

Challenges in Neural Network Optimization
Ill-Conditioning

Assume that we update our parameters using a gradient descent step $\boldsymbol{\theta}' = \boldsymbol{\theta} - \alpha \mathbf{g}$, where α is a learning rate and $\mathbf{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.

A second-order Taylor series expansion predicts that the value of the cost function at the new point is given by

$$J(\boldsymbol{\theta}') = J(\boldsymbol{\theta}) - \alpha \mathbf{g}^T \mathbf{g} + \frac{1}{2} \mathbf{g}^T \mathbf{H} \mathbf{g},$$

where $\mathbf{H}$ is the Hessian of J with respect to $\boldsymbol{\theta}$.

The $-\alpha \mathbf{g}^T \mathbf{g}$ term is always negative - if the cost function were a linear function of the parameters, gradient descent would always move downhill. However, the second-order term $\frac{1}{2} \mathbf{g}^T \mathbf{H} \mathbf{g}$ can be negative or positive depending on the eigenvalues of H and the alignment of the corresponding eigenvectors with $\mathbf{g}$.

Hessian: Matrix of Second Derivatives

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\, \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\, \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\, \partial x_1} & \dfrac{\partial^2 f}{\partial x_n\, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Challenges in Neural Network Optimization
ILL-Conditioning

On steps where $\mathbf{g}$ aligns closely with large, positive eigenvalues of $\mathbf{H}$, the learning rate must be very small, or the second-order term will result in gradient descent accidentally moving uphill.

The ill-conditioning problem is generally believed to be present in neural networks. It can manifest by causing SGD to get "stuck" in the sense that even very small steps increase the cost function. We can monitor the squared gradient norm $\mathbf{g}^T\mathbf{g}$ and the $\mathbf{g}^T\mathbf{H}\mathbf{g}$ term. In many cases, the gradient norm does not shrink significantly throughout learning, but the $\mathbf{g}^T\mathbf{H}\mathbf{g}$ term grows by more than an order of magnitude.

Challenges in Neural Network Optimization
Local Minima

Neural networks and any models with multiple equivalently parameterized latent variables all have multiple local minima because of the model identifiability problem.

A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters.

Challenges in Neural Network Optimization
Local Minima

Causes of non-identifiability:

- Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other.
- In any rectified linear or maxout network, we can scale all of the incoming weights and biases of a unit by α if we also scale all of its outgoing weights by $1/\alpha$. Thus, every local minimum of a rectified linear or maxout network lies on an $(m \times n)$-dimensional hyperbola of equivalent local minima.

Challenges in Neural Network Optimization
Plateaus, Saddle Points and Other Flat Regions

Many classes of random functions exhibit the following behavior:

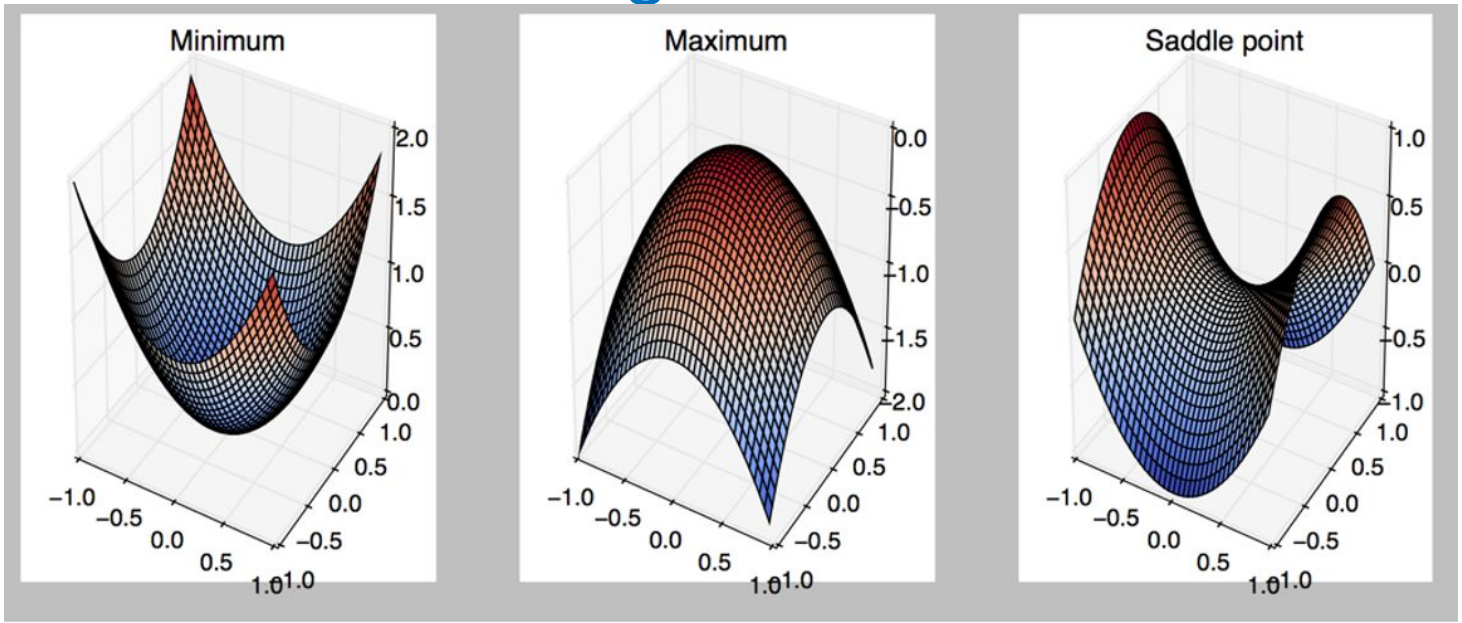
- **In low-dimensional spaces, local minima are common.**
- **In higher dimensional spaces, local minima are rare and saddle points are exponentially more common.**

To understand the intuition behind this, observe that a local minimum has only positive eigenvalues.

A saddle point has a mixture of positive and negative eigenvalues.

Imagine that the sign of each eigenvalue is generated by flipping a coin. In 1-dimensional space, it is easy to obtain a local minimum by tossing a coin and getting heads once. In n-dimensional space, it is exponentially unlikely that all n coin tosses will be heads.

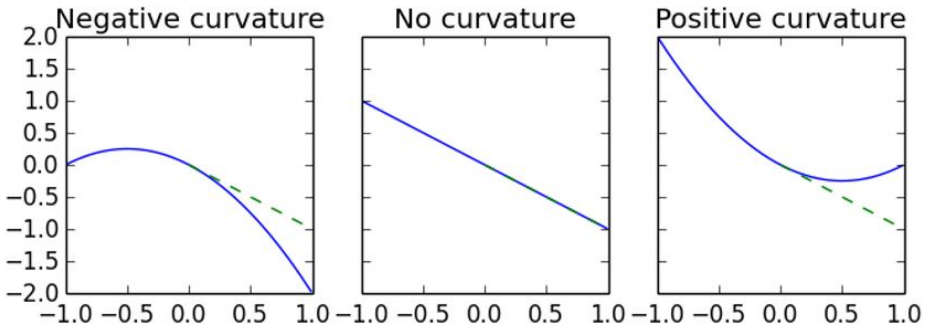
Critical Points where Gradient is zero and Hessian has "..." Eigenvalues



(Goodfellow 2015)

Cost
The Effect of Higher Order Derivatives

Cost function	Gradient	Hessian
$J(\boldsymbol{\theta})$	$\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$	$\boldsymbol{H}$
	$g_i = \frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta})$	$H_{i,j} = \frac{\partial}{\partial \theta_j} g_i$



Goodfellow 2015

Cost
Taylor Series Expansion

$$f(x) = f(x_0) + (x - x_0) f'(x) + \frac{1}{2}(x - x_0)^2 f''(x) + \dots$$

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{g} + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \dots$$

Baseline

Linear change due to gradient

Correction from directional curvature

Goodfellow 2015

Cost
Does (Every?) Gradient Step Reduce the Cost

2nd-order Taylor series prediction:

$$J(\boldsymbol{\theta} - \epsilon \boldsymbol{g}) \approx J(\boldsymbol{\theta}) - \epsilon \boldsymbol{g}^\top \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}$$

The improvement in the worst case when $\boldsymbol{g}$ aligns with $\lambda_{\max}$:

$$(\epsilon - \frac{1}{2}\epsilon^2 \lambda_{\max}) \boldsymbol{g}^\top \boldsymbol{g}$$

When $\boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g} \leq 0$, Taylor series predicts that all step sizes improve. Otherwise, optimal step size is

$$\epsilon^* = \frac{\boldsymbol{g}^\top \boldsymbol{g}}{\boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}}$$

Goodfellow 2015

Stochastic Gradient Descent Convergence

A sufficient condition to guarantee convergence is that

$$\sum_{k \in \mathbb{N}} \eta_k = \infty \quad \text{and} \quad \sum_{k \in \mathbb{N}} \eta_k^2 < \infty.$$

Stochastic gradient algorithm initially converges much faster than the batch gradient algorithm (many more stochastic updates can be performed for the computational price of one deterministic update).

Batch gradient descent will converge to lower values of the objective function.

For large neural networks which are trained on large datasets, SGD algorithms remain the algorithms of choice.

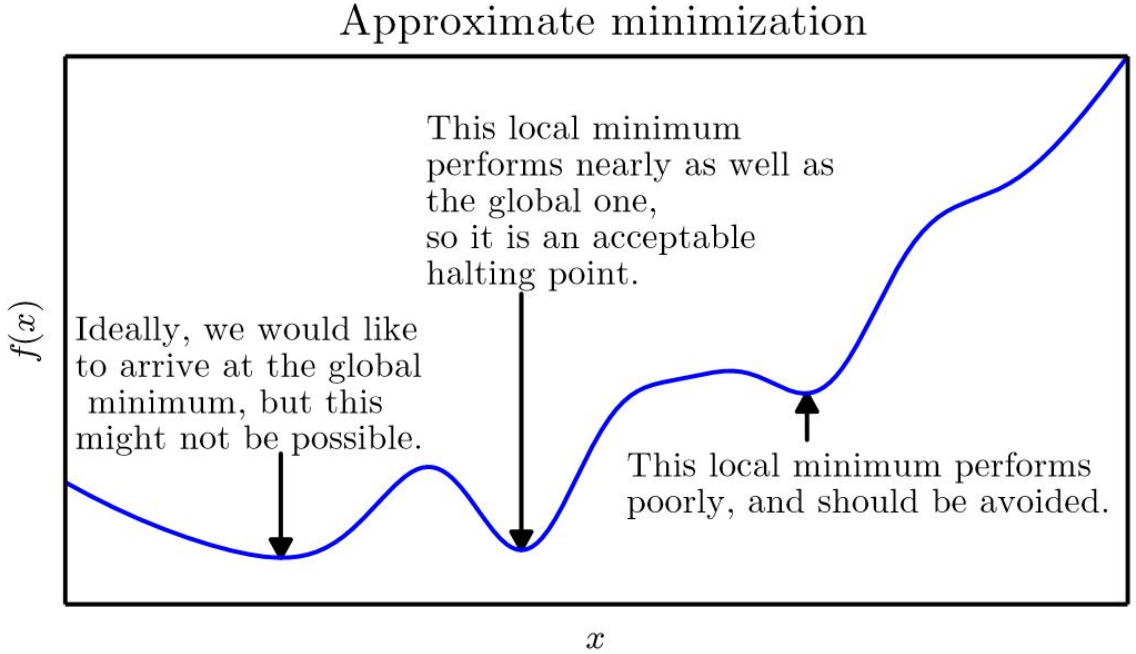
Key Variants of Stochastic Gradient Descent

Challenges with SGD

Momentum

Nesterov Momentum

Optimization: Seek Minimum
SGD Tries to Find the Minimum



www.deeplearningbook.org

Deep Learning, Goodfellow, Bengio, and Courville 2016

SGD Faces Problems

Traditional view:

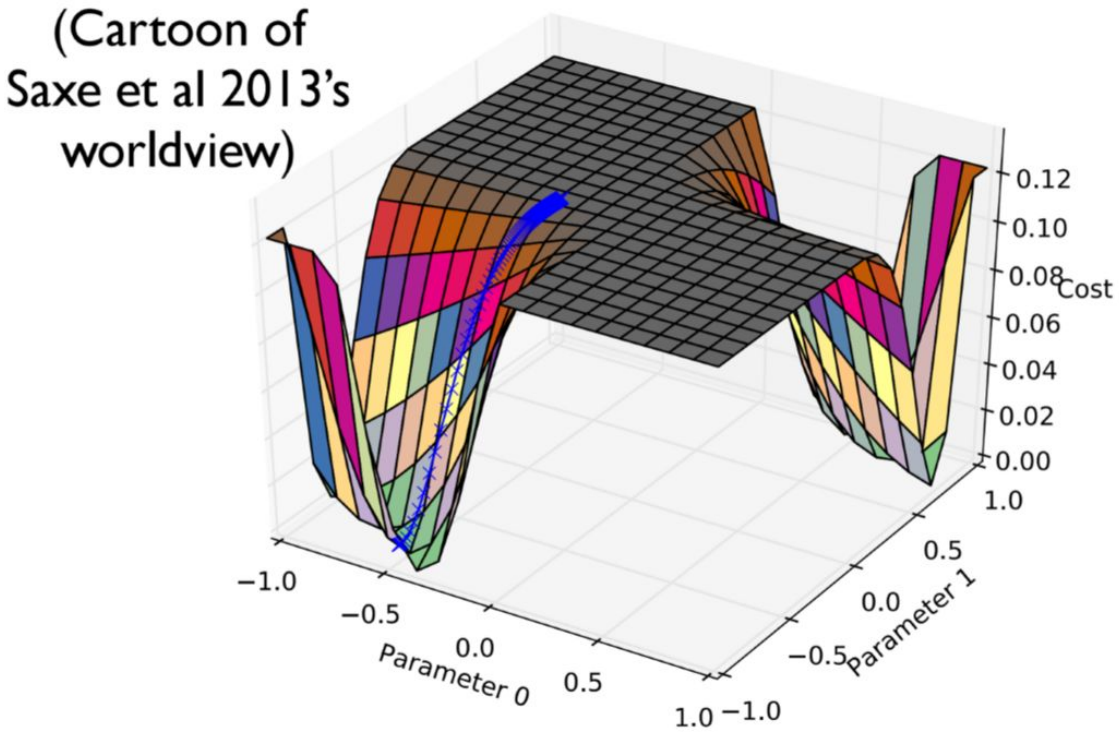
- **SGD usually moves downhill**
- **SGD eventually encounters a critical point**
- **Usually this is a minimum**
- **However, it is a local minimum**
- **J has a high value at this critical point**
- **Some global minimum is the real target, and has a much lower value of J**

SGD Faces Problems

Hyper-dimensional view:

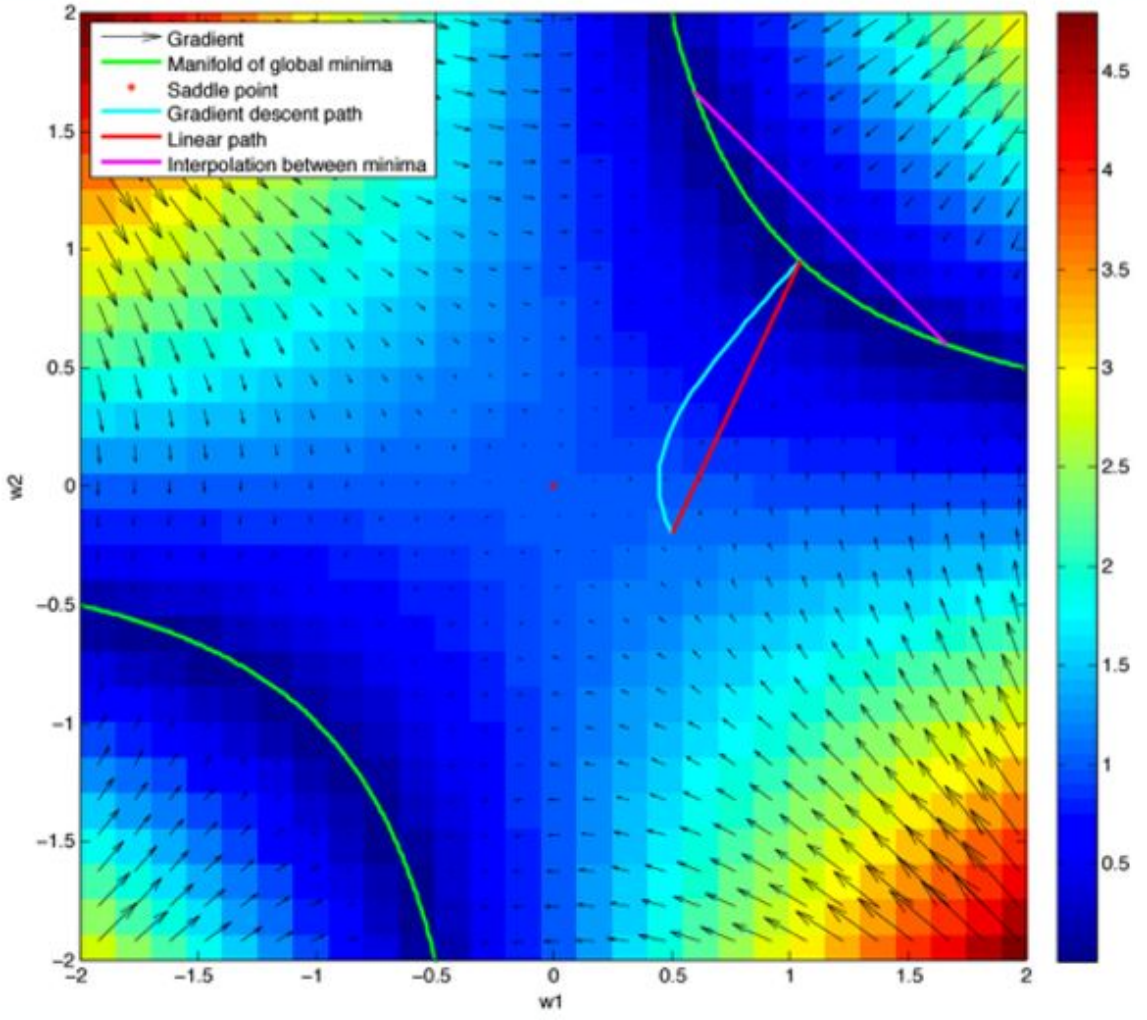
- **SGD usually moves downhill**
- **SGD eventually encounters a critical point**
- **Usually this is a saddle point**
- **SGD is stuck, and the main reason it is stuck is that it fails to exploit negative curvature**
  - **(as we will see, this happens to Newton's method, but not very much to SGD)**
  - **Never stop if function doesn't have a local minimum**
  - **Possibly still moving but not improving**
  - **Too bad of conditioning**
  - **Too much gradient noise**
  - **Overfitting**

Gradient Descent Example 1



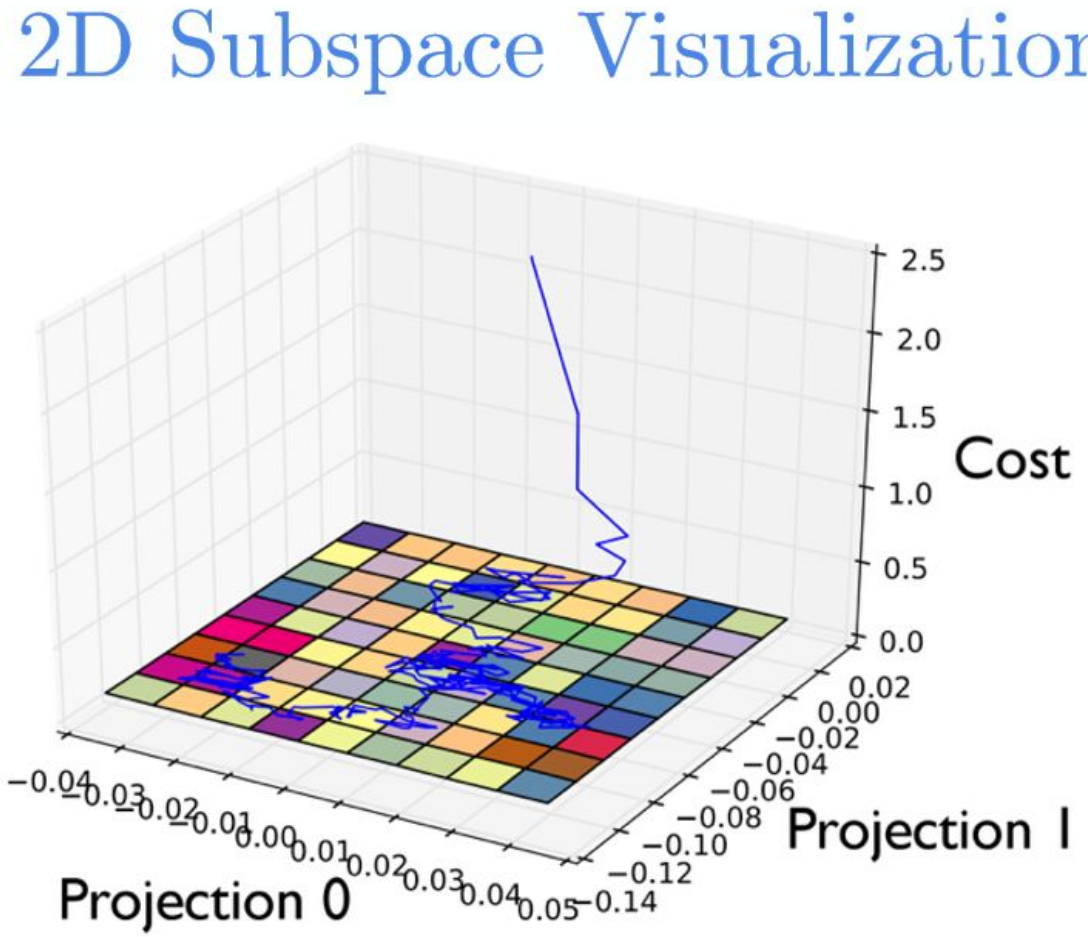
Goodfellow 2015

Gradient Descent Example 2



Goodfellow 2015

Gradient Descent Example 3



Goodfellow 2015

State-of-the-Art Optimizers

- **We can optimize most classifiers, autoencoders, or recurrent nets if they are based on linear layers**
- **Especially true of LSTM, ReLU, maxout**
- **It may be much slower than we want**
- **Even depth does not prevent success, Sussillo 14 reached 1,000 layers**
- **We may not be able to optimize more exotic models**
- **Optimization benchmarks are usually not done on the exotic models**

SGD Challenges
Target for Improvement

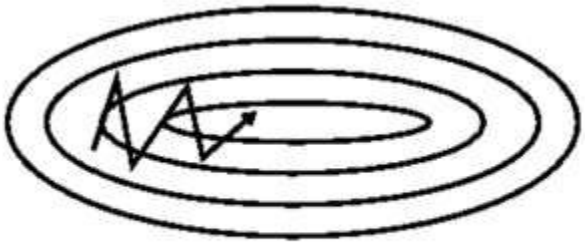
Optimization surfaces can be non-convex and irregular, with many local minima:

- **SGD has trouble navigating ravines which are common around local minima**
- **Areas where the surface curves much more steeply in one dimension than in another is very difficult for SGD**

(a) SGD without momentum

Momentum Accelerates SGD

(a) SGD without momentum



(b) SGD with momentum

Momentum accelerates SGD by adding a fraction α of the update vector of the past time-step to the current update vector. Typical α = 0.9.

Analogy: Think of a speedup of a ball rolling down a steep slope, going faster and faster.

(SGD with) Momentum

Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient.

Formally, we introduce a variable *v* that plays the role of velocity (or momentum) which accumulates the gradient. The update rule is given by:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \nabla_{\boldsymbol{\theta}} \frac{1}{m} \sum_{t=1}^{m} L(f(\mathbf{x}^t; \boldsymbol{\theta}), \mathbf{y}^t)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v},$$

where *v* is the direction and speed at which the parameters move through the parameter space. Note that the velocity is set to an exponentially decaying average of the negative gradient. The larger α is relative to η, the more previous gradients affect the current direction.

(SGD with) Momentum

Algorithm 2 Stochastic gradient descent update with momentum

Require: Learning rate η, momentum parameter α.

Require: Initial parameter $\boldsymbol{\theta}$, initial vector $\mathbf{v}$.

while Stopping criterion not met **do**

  Sample a minibatch of m examples from the training set $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

  Set $\mathbf{g} = \mathbf{0}$.

  for $i = 1$ to m **do**

    Compute gradient estimate:

    $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

  end for

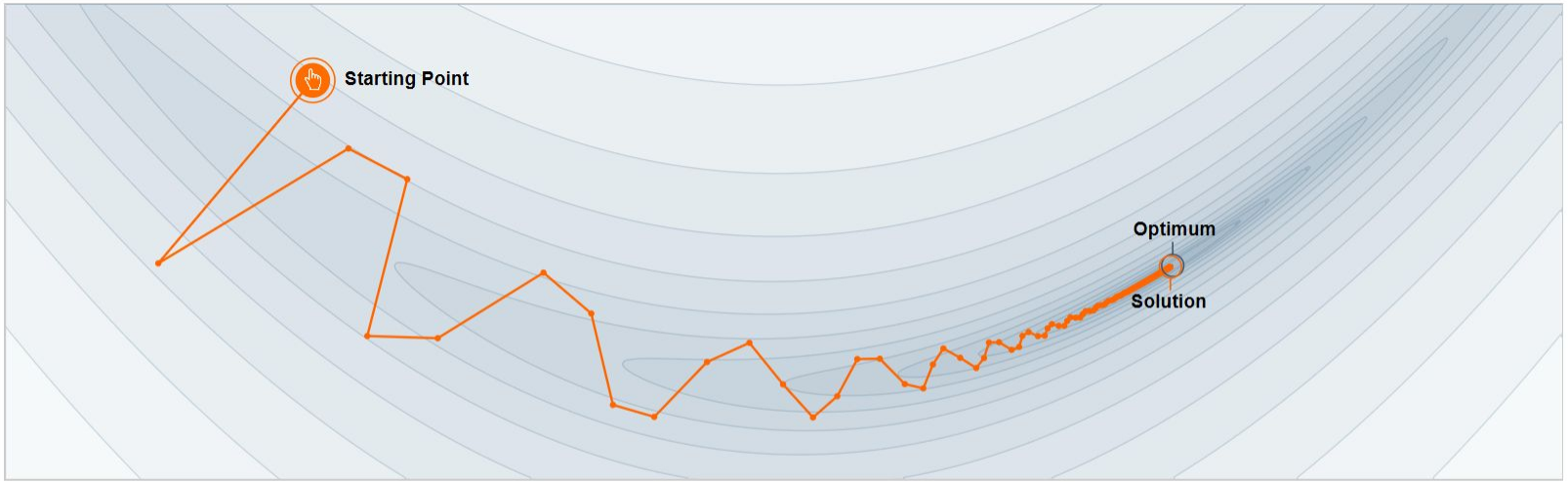
  Compute velocity update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\mathbf{g}$

  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$

end while

Momentum - Why does it Work

The momentum term reduces (cancels out) updates for dimensions whose gradients change directions.



Step-size α = 0.0039

0 0.003 0.006

Momentum β = 0.80

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

The momentum term increases for dimensions whose gradients point in the same direction.

Interactive treatise: Goh, "Why Momentum Really Works", Distill, 2017. http://doi.org/10.23915/distill.00006

Momentum - Critique

A ball which rolls down a hill, blindly following the slope, may be unsatisfactory.

- **We would like to have a smarter ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.**
- **Nesterov Momentum (accelerated gradient) improves on the momentum method.**

Nesterov Momentum

The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied.

Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard momentum method.

The update rule is given by:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \nabla_{\boldsymbol{\theta}} \frac{1}{m} \sum_{t=1}^{m} L(f(\mathbf{x}^t; \boldsymbol{\theta} + \alpha \mathbf{v}), \mathbf{y}^t)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}.$$

Nesterov Momentum

Algorithm 3 Stochastic gradient descent with Nesterov momentum

Require: Learning rate η, momentum parameter α.

Require: Initial parameter $\boldsymbol{\theta}$, initial vector $\mathbf{v}$.

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Apply interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\mathbf{v}$

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient (at interim point):

$\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Compute velocity update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\mathbf{g}$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$

end while

Nesterov Momentum

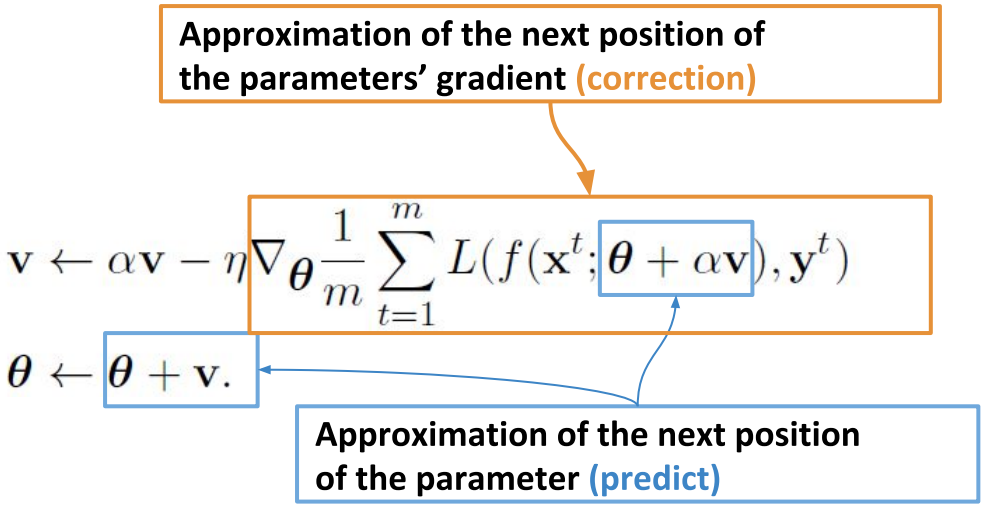
$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\nabla_{\boldsymbol{\theta}}\frac{1}{m}\sum_{t=1}^{m} L(f(\mathbf{x}^t; \boldsymbol{\theta} + \alpha\mathbf{v}), \mathbf{y}^t)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}.$$

Approximation of the next position of the parameter (predict)

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Nesterov Momentum



Nesterov Momentum

Blue line: predict

Beige line: correction

Green line: accumulated gradient

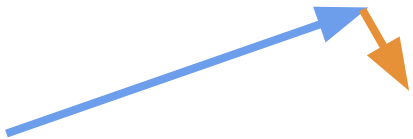
Approximation of the next position of the parameters' gradient (correction)

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\nabla_{\boldsymbol{\theta}}\frac{1}{m}\sum_{t=1}^{m} L(f(\mathbf{x}^t;\boldsymbol{\theta}+\alpha\mathbf{v}),\mathbf{y}^t)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}.$$

Approximation of the next position of the parameter (predict)

Nesterov Momentum



Blue line: predict

Beige line: correction

Green line: accumulated gradient

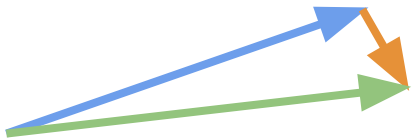
Approximation of the next position of the parameters' gradient (correction)

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\nabla_{\boldsymbol{\theta}}\frac{1}{m}\sum_{t=1}^{m} L(f(\mathbf{x}^t;\boldsymbol{\theta}+\alpha\mathbf{v}),\mathbf{y}^t)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}.$$

Approximation of the next position of the parameter (predict)

Nesterov Momentum



Blue line: predict

Beige line: correction

Green line: accumulated gradient

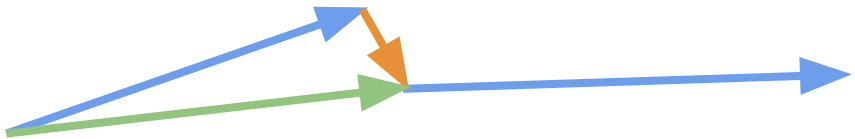
Approximation of the next position of the parameters' gradient (correction)

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\nabla_{\boldsymbol{\theta}}\frac{1}{m}\sum_{t=1}^{m} L(f(\mathbf{x}^t;\boldsymbol{\theta}+\alpha\mathbf{v}),\mathbf{y}^t)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}.$$

Approximation of the next position of the parameter (predict)

Nesterov Momentum

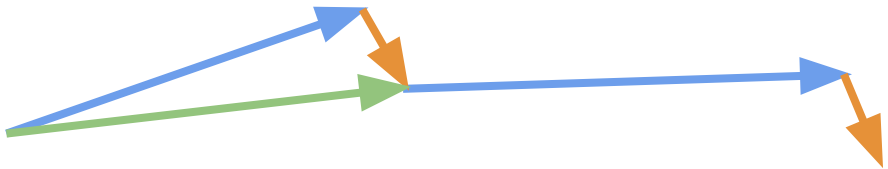


Blue line: predict

Beige line: correction

Green line: accumulated gradient

Nesterov Momentum

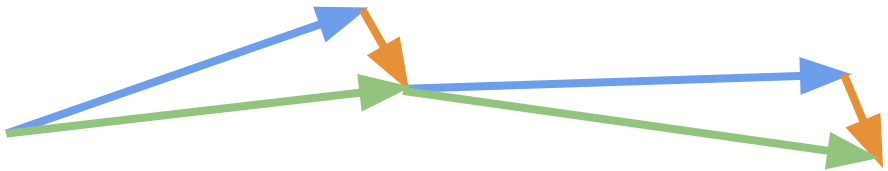


Blue line: predict

Beige line: correction

Green line: accumulated gradient

Nesterov Momentum

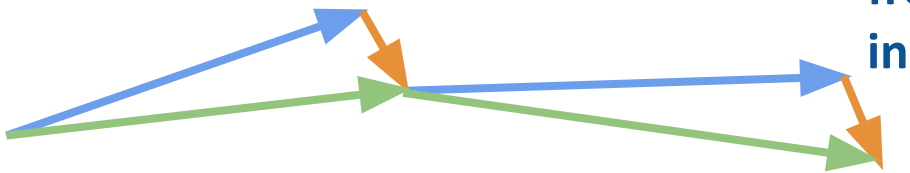


Blue line: predict

Beige line: correction

Green line: accumulated gradient

Nesterov Momentum



This anticipatory update prevents from going too fast and results in increased responsiveness.

Blue line: predict

Beige line: correction

Green line: accumulated gradient

Now, updates can be adapted to the slope of the error hypersurface, and speed up SGD in turn.

Variants of Stochastic Gradient Descent with Adaptive Learning Rates

AdaGrad

RMSprop

Adam

AdaDelta

AdamW

More Control of Learning Rates

Previous methods :

- **Used the same learning rate η for all parameters θ**

We want to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.

- **Adagrad**
- **Adadelta**
- **RMSprop**
- **Adam, Adagrad, AdamW**

The AdaGrad Algorithm

- Uses a different learning rate for every parameter θ_i at every time step t.
- Individually adapts the learning rate for each model parameter θ_i, by scaling it inversely proportional to an accumulated sum of squared partial derivatives over all training iterations.
- The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate.
- The net effect is greater progress in the more gently sloped directions of parameter space.

The AdaGrad Algorithm

In the convex optimization context, the AdaGrad algorithm enjoys some desirable theoretical properties.

However it has been empirically found that for training deep neural network models, the accumulation of squared gradients from the beginning of training results in premature and excessive decrease in the effective learning rate.

The AdaGrad Algorithm

Algorithm 4 The AdaGrad Algorithm

Require: Global learning rate η.

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

Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$,

while Stopping criterion not met **do**

Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Accumulate gradient: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g}^2$ (square element-wise)

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\mathbf{r}}}$ element-wise)

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

end while

The AdaGrad Algorithm - Alternative Notation

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$$\mathbb{R}^{d \times d}$$

$$G_t = \begin{pmatrix} \blacksquare & \cdots & \bullet & \cdots & \bullet \\ \vdots & & \vdots & & \vdots \\ \bullet & \cdots & \blacksquare & \cdots & \bullet \\ \vdots & & \vdots & & \vdots \\ \bullet & \cdots & \bullet & \cdots & \blacksquare \end{pmatrix}$$

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

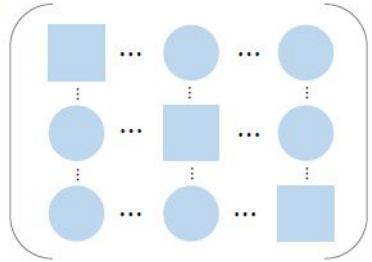
Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

The AdaGrad Algorithm - Alternative Notation

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$$G_t = \in \mathbb{R}^{d \times d}$$



Adagrad modifies the general learning rate η based on the past gradients that have been computed for θ_i

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

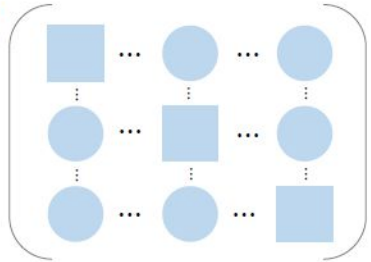
Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

The AdaGrad Algorithm - Alternative Notation

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$$G_t = \mathbb{R}^{d \times d}$$



G_t is a diagonal matrix where each diagonal element (i,i) is the sum of the squares of the gradients θ_i up to time step t.

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

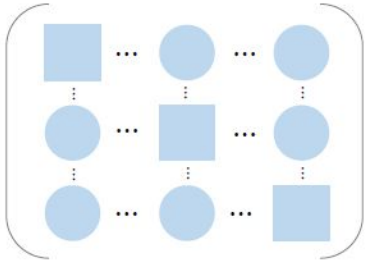
Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

The AdaGrad Algorithm - Alternative Notation

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$$G_t = \in \mathbb{R}^{d \times d}$$



ε is a smoothing term that avoids division by zero (usually on the order of 1e − 8).

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

The AdaGrad Algorithm

Advantages:

- **Well-suited for dealing with sparse data**
- **Greatly improves the robustness of SGD**
- **Eliminates manual tuning of the learning rate**

Disadvantage:

- **Accumulation of the squared gradients in the denominator. This causes the learning rate to shrink and become infinitesimally small. The algorithm can no longer acquire additional knowledge.**

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Adadelta

Adadelta is an extension of Adagrad.

- **Adagrad accumulates all past squared gradients -> problem**

Adadelta:

- **Restricts the window of accumulated past gradients to a fixed size w**

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Adadelta

- Instead of inefficiently storing, the sum of gradients is recursively defined as a decaying average of all past squared gradients.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

- $E[g^2]_t$: The running average at time step t.
- γ : A fraction similarly to the Momentum term, around 0.9

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Adadelta

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Adagrad

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

SGD

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Adadelta

Adagrad

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

SGD

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Adadelta

Adagrad

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

SGD

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Adadelta

- The units in this update do not match and the update should have the same hypothetical units as the parameter.
  - As well as in SGD, Momentum, or Adagrad

- To realize this, first defining another exponentially decaying average

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Adadelta

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma) \Delta\theta_t^2$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

Adadelta

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma) \Delta\theta_t^2$$

We approximate RMS with the RMS of parameter updates until the previous time step.

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

Adadelta

- Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_{t-1}$ finally yields the Adadelta update rule:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

- Note : **we do not even need to set a default learning rate**

Adadelta - Alternative Notation

Algorithm 8 The AdaDelta Algorithm

Require: Decay rate ρ, constant ϵ.

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

Initialize accumulation variables $\mathbf{r} = \mathbf{0}$, $\mathbf{s} = \mathbf{0}$.

while Stopping criterion not met **do**

Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Accumulate gradient: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\mathbf{g}^2$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\sqrt{\mathbf{s}+\epsilon}}{\sqrt{\mathbf{r}+\epsilon}} \odot \mathbf{g}$ (element-wise)

Accumulate update: $\mathbf{s} \leftarrow \rho\mathbf{s} + (1-\rho)[\Delta\boldsymbol{\theta}]^2$

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

end while

The RMS Prop Algorithm

The RMSprop algorithm addresses the deficiency of AdaGrad by changing the gradient accumulation into an exponentially weighted moving average.

In deep networks, the optimization surface is far from convex. Directions in parameter space with strong partial derivatives early in training may flatten out as training progresses.

The introduction of the exponentially weighted moving average allows the effective learning rates to adapt to the changing local topology of the loss surface.

The RMS Prop Algorithm

Algorithm 5 The RMSprop Algorithm

Require: Global learning rate η, decay rate ρ.

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

Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$,

while Stopping criterion not met **do**

Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Accumulate gradient: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1 - \rho)\mathbf{g}^2$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\mathbf{r}}}$ element-wise)

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

end while

The RMS Prop Algorithm - Alternative Notation

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients.

RMSprop

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

The RMS Prop with Nesterov Momentum

RMSprop combined with Nesterov momentum introduced a new hyperparameter ρ, which controls the length scale of the moving average.

Empirically RMSprop has shown to be an effective and practical optimization algorithm for deep neural networks: it is easy to implement and relatively simple to use; there does not appear to be a great sensitivity to the algorithm's hyperparameters.

RMSprop is currently one of the "go to" optimization methods being employed routinely by deep learning researchers.

The RMS Prop with Nesterov Momentum

Algorithm 6 The RMSprop Algorithm with Nesterov Momentum

Require: Global learning rate η, decay rate ρ, momentum coeff. α.

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\mathbf{v}$.

Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$,

while Stopping criterion not met **do**

Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Compute interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\mathbf{v}$

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Accumulate gradient: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1 - \rho)\mathbf{g}^2$

Compute update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\mathbf{r}}}$ element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$

end while

The Adam Algorithm

Adam is a variant on RMSprop+momentum with important distinctions:

- **Momentum is incorporated directly as an estimate of the first order moment with exponential weighting of the gradient. The most straightforward way to add momentum to RMSprop is to apply momentum to the rescaled gradients (not particularly well motivated).**
- **Includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second order moments to account for their initialization at the origin.**

Note that RMSprop also incorporates an estimate of the (uncentered) second order moment; however, it lacks the correction term of Adam. Unlike in Adam, the RMSprop second-order moment estimate may have high bias early in training.

The Adam Algorithm

Algorithm 7 The Adam Algorithm

Require: Step-size α.
Require: Decay rates ρ_1 and ρ_2, constant ϵ.
Require: Initial parameter $\boldsymbol{\theta}$.
  Initialize 1st and 2nd moment variables $\mathbf{s} = \mathbf{0}$, $\mathbf{r} = \mathbf{0}$.
  Initialize timestep $t = 0$.
  while Stopping criterion not met **do**
    Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
    Set $\mathbf{g} = \mathbf{0}$.
    for $i = 1$ to m **do**
      Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$
    end for
    $t \leftarrow t + 1$

The Adam Algorithm

Algorithm 7 The Adam Algorithm (cont'd)

Get biased first moment: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\mathbf{g}$

Get biased second moment: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g}^2$

Compute bias-corrected first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}$

Compute bias-corrected second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\alpha \frac{\mathbf{s}}{\sqrt{\mathbf{r}}+\epsilon} \odot \mathbf{g}$ (element-wise)

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

end while

The Adam Algorithm - Alternative Notation

- Adam's feature :
  - Storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop
  - Keeping an exponentially decaying average of past gradients m_t, similar to momentum.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \longrightarrow$$ The first moment (the mean)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \longrightarrow$$ The second moment (the uncentered variance)

The Adam Algorithm - Alternative Notation

- As m_t and v_t are initialized as vectors of 0's, they are biased towards zero.
  - Especially during the initial time steps
  - Especially when the decay rates are small
    - (i.e. β1 and β2 are close to 1).

- Counteracting these biases in Adam

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

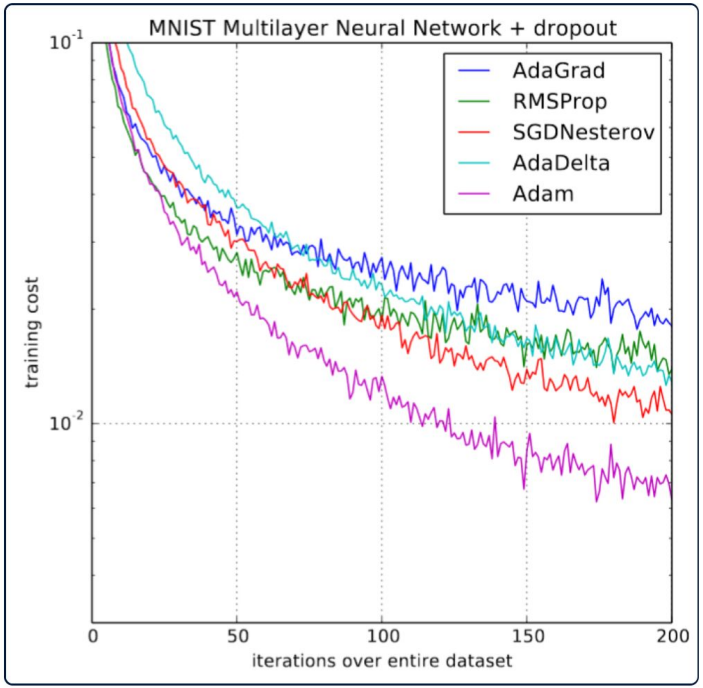
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Adam

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Note : default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε

Comparison of Optimizers



But in practice, this does not really pan out.

Why?

AdamW Algorithm

Are weight decay and L2 regularization the same?

The two are the same for (plain) SGD:

- **L2 regularization: final_loss = loss + wd * all_weights.pow(2).sum() / 2**
- **For SGD -> weight loss: w = w - lr * w.grad - lr * wd * w**

But they are not the same for more sophisticated optimization algorithms!

https://www.fast.ai/2018/07/02/adam-weight-decay/

AdamW Algorithm

Weight decay and L2 regularization are not always the same - example:

SGD with momentum:

- **Using L2 regularization consists in adding wd*w to the gradients, but the gradients aren't subtracted from the weights directly.**
- **First, one computes a moving average**

This leads to AdamW:

- **Use weight decay with Adam, and not the L2 regularization (that classic deep learning libraries implement).**

https://www.fast.ai/2018/07/02/adam-weight-decay/

AdamW Algorithm

Summary:

- **Inside the step function of the optimizer, only the gradients are used to modify the parameters, the value of the parameters themselves isn't used at all (except for the weight decay, but that will be done outside).**
- **One can then implement weight decay by simply doing it before the step of the optimizer. It still has to be done after the gradients are computed (otherwise it would impact the gradients values) so inside the training loop.**
- **The optimizer should have been set with wd=0 otherwise it will do some L2 regularization**

https://www.fast.ai/2018/07/02/adam-weight-decay/

AdamW Algorithm

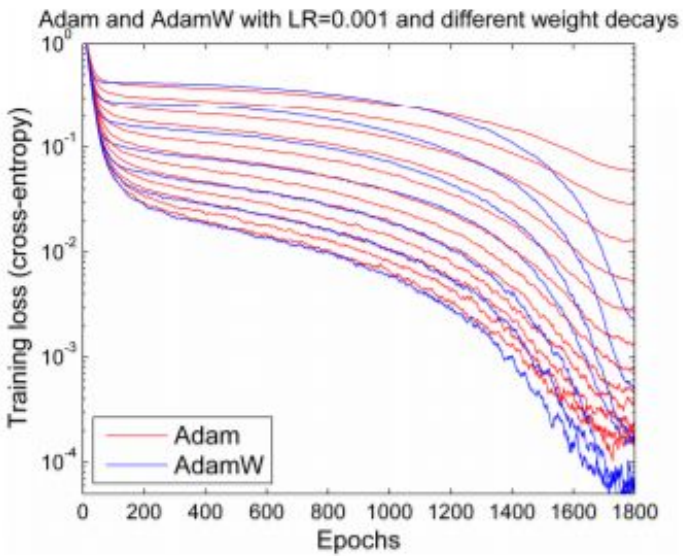
Summary:

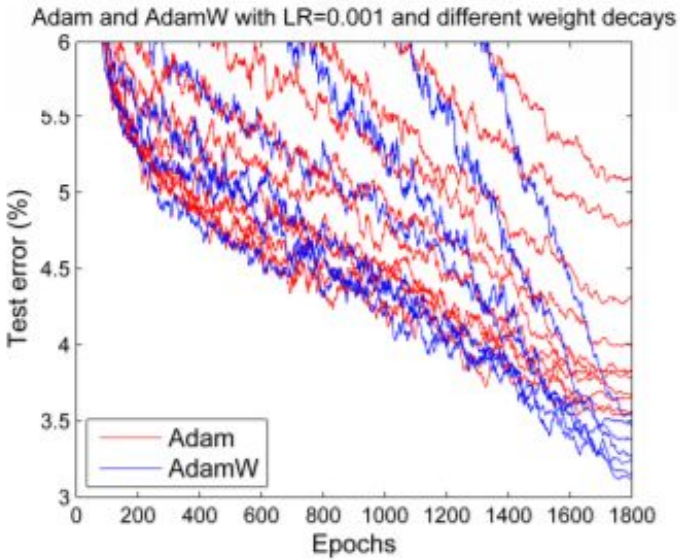
```
loss.backward()
#Do the weight decay here!
optimizer.step()

loss.backward()
for group in optimizer.param_groups():
    for param in group['params']:
        param.data = param.data.add(-wd * group['lr'], param.data)
optimizer.step()
```

https://www.fast.ai/2018/07/02/adam-weight-decay/

AdamW Algorithm





Choosing the Optimization Algorithm

There is no consensus on what is “the best” algorithm.

A comparison of optimization algorithms across a wide range of learning tasks suggests that the family of algorithms (represented by RMSprop and AdaDelta) perform robustly; however, no single best algorithm has emerged)

The most popular optimization algorithms actively in use include SGD, SGD+momentum, RMSprop, RMSprop+momentum, AdaDelta and Adam.

The choice of which algorithm to use, as this point, seems to depend as much on the users familiarity with the algorithm (for ease of hyperparameter tuning) as it does on any established notion of superior performance.

Although, there is evidence that properly implemented “advanced algorithms” may be achieving “super-convergence”.

Optimization Trajectories

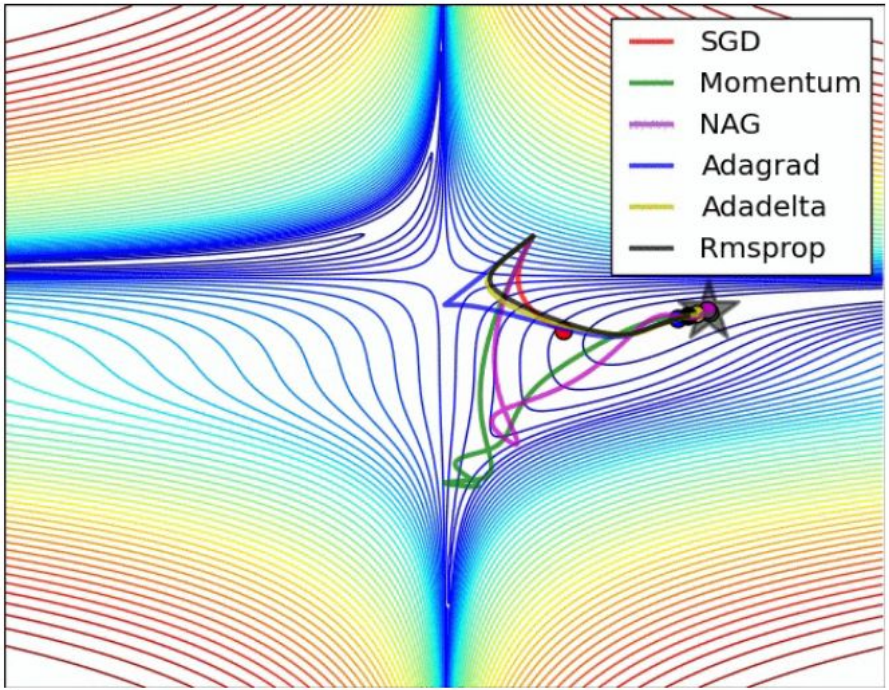


Image 5: SGD optimization on loss surface contours

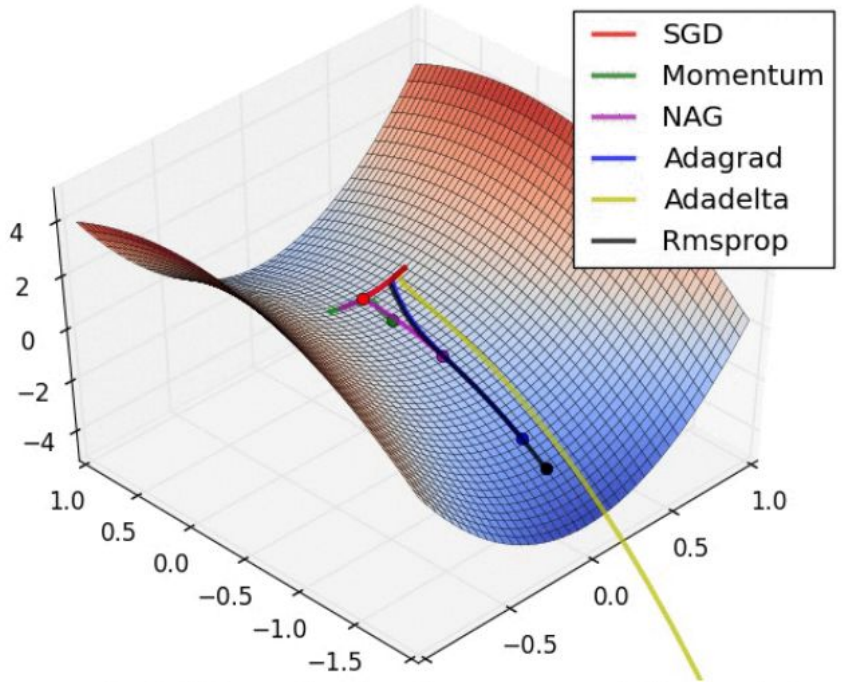


Image 6: SGD optimization on saddle point

Sebastian Ruder (2016). An overview of gradient descent optimisation algorithms. arXiv preprint arXiv:1609.04747. **http://ruder.io/optimizing-gradient-descent/**

Parallelizing and Distributing the SGD Algorithm

SGD is inherently sequential, when it is run step by step:

- **It has good convergence**
- **It is slow**

Parallelization/distributed execution could be used to make it faster:

- **But, suboptimal communication between workers can lead to poor convergence.**

Parallelizing and distributing SGD

The following algorithms and architectures are designed for optimizing parallelized and distributed SGD:

- **Hogwild!**
- **Downpour SGD**
- **Delay-tolerant Algorithms for SGD**
- **TensorFlow**
- **Elastic Averaging SGD**

Initialization

Training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization.

The initial point can determine whether the algorithm converges at all, if it is stable or not and if the algorithm fails altogether.

When learning does converge, the initial point can determine how quickly learning converges, and whether it converges to a point with high or low cost. Also, points of comparable cost can have wildly varying generalization errors.

Backup Slides

Optional material

Adadelta Alternative Description

AdaDelta seeks to directly address problems with AdaGrad by incorporating some second-order gradient information into the optimization algorithm.

The Taylor series around the current point $\boldsymbol{\theta}^0$ for the single dimension θ_j is given by:

$$L(f(\mathbf{x}^i;\boldsymbol{\theta}^0 + \mathbf{e}_j\Delta\theta_j), \mathbf{y}^i) = L(f(\mathbf{x}^i;\boldsymbol{\theta}^0), \mathbf{y}^i) + \\ + \mathbf{e}_j\frac{\partial}{\partial\theta_j}L(f(\mathbf{x}^i;\boldsymbol{\theta}^0), \mathbf{y}^i)\Delta\theta_j + +\mathbf{e}_j\frac{1}{2}\frac{\partial^2}{\partial\theta_j^2}L(f(\mathbf{x}^i;\boldsymbol{\theta}^0), \mathbf{y}^i)\Delta\theta_j^2$$

Adadelta Alternative Description

The Taylor series reaches is extremum with respect to $\Delta\theta_j$ when its derivative is equal to zero:

$$\Delta\theta_j = \frac{1}{\frac{\partial^2}{\partial\theta_j^2} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i)} \frac{\partial}{\partial\theta_j} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i),$$

or

$$\frac{1}{\frac{\partial^2}{\partial\theta_j^2} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i)} = \frac{\Delta\theta_j}{\frac{\partial}{\partial\theta_j} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i)}.$$

AdaDelta estimates the LHS as the ratio of separate estimates of $\Delta\theta_j$ and $\frac{\partial}{\partial\theta_j} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i)$, using an exponentially weighted RMS estimates of both the parameter increments (in the numerator) and partial derivatives (in the denominator).

Approximate Second-Order Optimization Methods

Newton's Method

Conjugate Gradients

BFGS

Optimization Strategies and Meta-Algorithms

Coordinate Descent