Optimization Algorithms for Deep Learning Models

Sept 28, Oct 8, 8, 2021

Optimizing Deep Learning Models

- Training Deep Neural Network is one of the most difficult problems.

- Specialized optimization techniques for solving the problem of improving NN's performance.

How Optimization Differs from Learning

- Optimization problem deals with finding the minimum of an objective function.
- For example in NN-training the cost function minimization is an optimization problem.
- In a typical machine learning task, one is interested in improving the performance of the model on unseen data (test data) with respect to some evaluation criterion (performance measure).
- So one looks for optimizing the cost function to indirectly deal with the performance improvement of the model.

How Optimization Differs from Learning…

- Practically, the samples come from an unknown data distribution say, $p_{data}(\boldsymbol{x}, y)$.
- But we train the network to reduce the cost function on training samples.
- That means we are actually training the algorithm to learn the data distribution $\hat{p}(\boldsymbol{x}, y)$ of samples, not the actual distribution $p_{data}(\boldsymbol{x}, y)$.
- Reducing the error on test set or unseen samples may be intractable.

Risk and Empirical Risk

- The problem of reducing the error expectation function over the true data distribution is called '**risk minimization**' problem. That is true generalization error minimization.

- In machine learning algorithms, one minimizes the error expectation function over the training sample distribution. This is known as '**empirical risk minimization**'.

Empirical Risk Minimization

- Empirical risk minimization is prone to overfitting.
  - Models with high capacity do memorize the training set.
- Many loss functions may not be having derivatives.
- In deep learning, the empirical risk minimization is rarely used for performance improvement.

Surrogate Loss Functions

- Instead of minimizing empirical risk, we minimize a surrogate loss function.

- A surrogate loss function acts as a proxy to empirical risk while having enough smoothness for optimization.

- Example :
  - Use of negative log-likelihood in place of 0-1 loss function in binary classification problem.
  - Use of Hinge loss function in Support Vector Machines.

Early Stopping

- In contrast to standard optimization, training algorithms do not stop at local minima, they stop when an early stopping criterion is satisfied.
- Early stopping criterion is based on loss function measured on the validation set.
- The criterion halts the algorithm just before overfitting takes place.
- It means, we try to approximate the true loss function in the learning process, by giving experience of validation data to the learning algorithm.

Early Stopping …

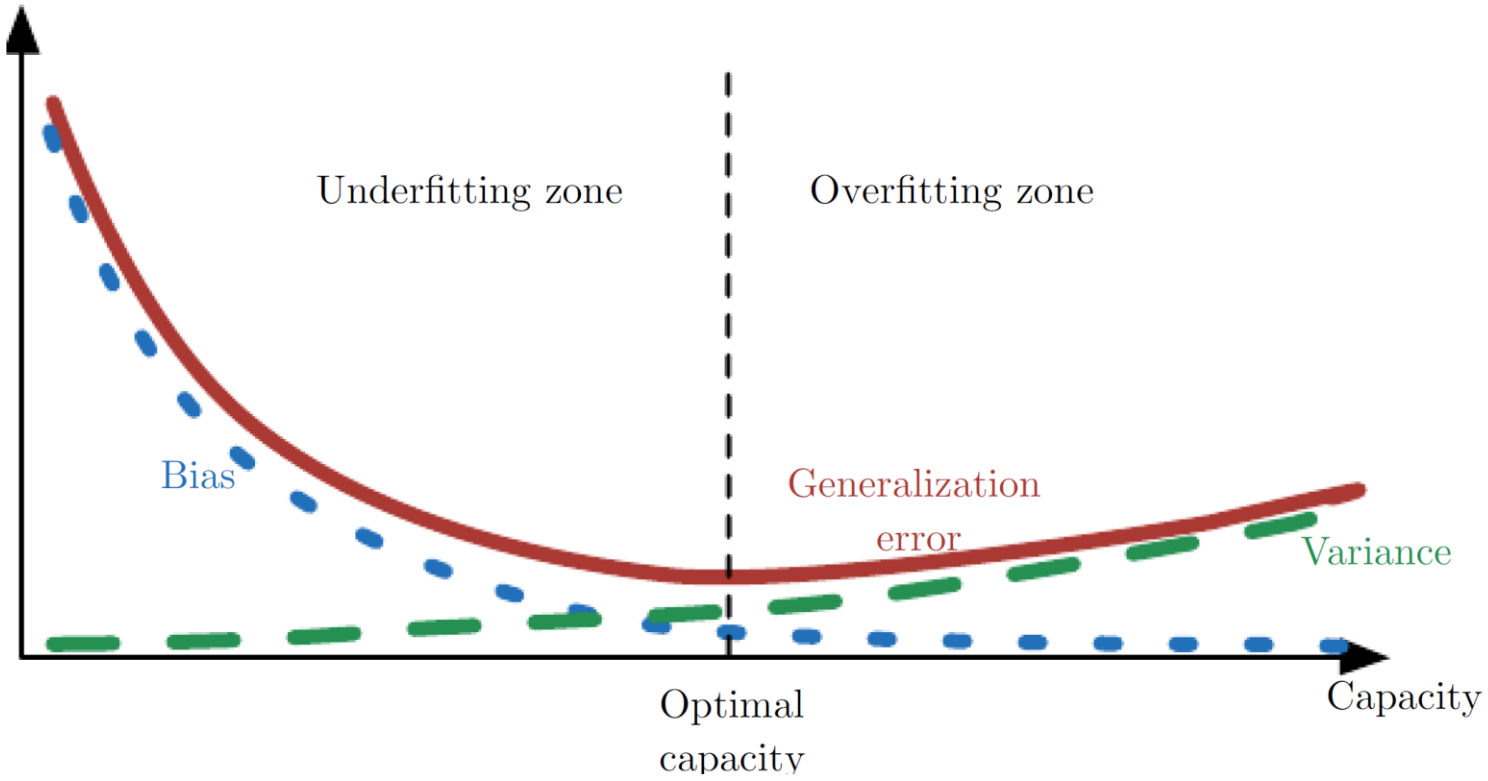


Figure source: © Ian Goodfellow et al., Deep Learning, MIT Press, 2016

Early Stopping …

- Training often stops while the surrogate loss function has large derivatives.

- In contrast, a pure optimization algorithm is said to converge when the gradients become very small.

Mini-Batches

- Consider the optimization techniques using gradient descent.
- Suppose we have m training examples. To compute the gradient, we actually compute the derivative of the loss for each and every example.
- This is an expensive process especially if the dataset is large.
- we compute these derivatives and their average values by randomly sampling a small number of examples from the dataset. Then taking the average of derivatives of loss functions.
- In practice, we divide the entire dataset into small subsets called **mini-batches** to speed up training and optimization.

Mini-Batches

- As an example consider the problem of estimating the mean of the actual data distribution from which our data samples have arrived.
- Then the standard error = $\sigma/\sqrt{n}$, where σ is the actual standard deviation of the data distribution and number of samples = n.
- Now imagine the cases when $n = 100$ and $n = 10{,}000$.
- The second case requires 100 times more computations than the first case.
- But the reduction in standard error of the mean will be only by a factor of 10.
- It is therefore a good idea to use small batches of samples.

Mini-Batches

- Another motivation to estimate the gradient of cost from a small random number of samples is that sometimes there are redundancies in the training set.

A small bird perched on a tree branch		A small bird with yellow flower

Mini-Batches

- Worst case scenario
  - All *m* samples in the training set are identical copies of each other!
- In practice, this does not happen, but a large number of examples may give rise to almost same value of the gradient.
- Thus we do repetitive computations in large dataset without much gain.

Mini-Batches…

- Mini-batches are usually better than online(SGD with single sample).
  - Less computation is required to update the weights.
  - Full advantage of GPUs
    - Computing the gradient for many batches simultaneously using matrix-multiplications are very efficiently done on GPUs.

Deterministic and Stochastic Algorithms

- If an optimization algorithm uses all the training examples from the training set to compute the mean cost function (and its derivatives) then we call it a **deterministic** gradient method or **batch gradient descent** method.

- If only one example is randomly chosen each time to update the gradient value, it is called **stochastic gradient descent (SGD)** or **online** method.

- If we take a small number of randomly selected sampels, then it is called **mini-batch gradient descent**.

- But more recently, mini-batch gradient descent are also termed as SGD, when samples are chosen in a truly random fashion.

To Apply Mini-Batches

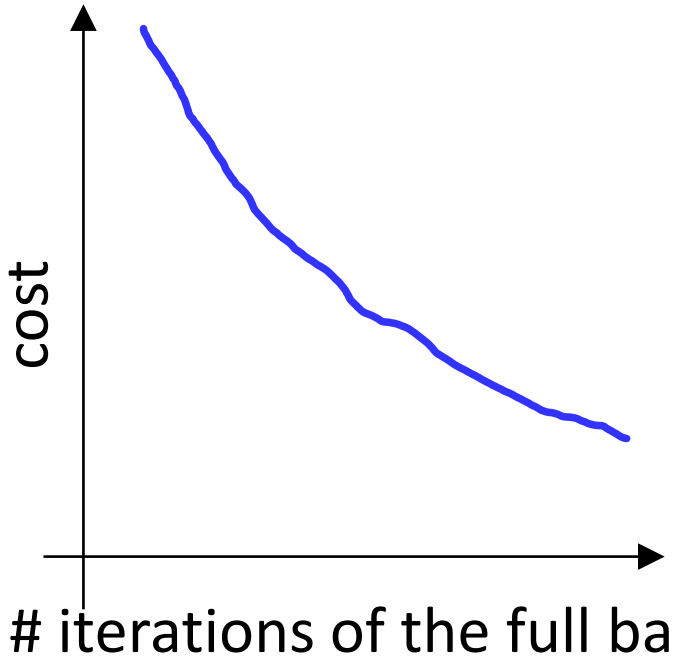
- For large neural networks with very large and highly redundant training sets, it is always preferred to use mini-batch learning.
- Suppose we have m training examples. We divide the set into M subsets of the same size, called mini-batches.
- Then we pass each mini-batch through the network, called an epoch.
- Apply forward and backward propagation.
- Update weights
- Iterate the process for all mini-batches.

Processing with Mini-Batches

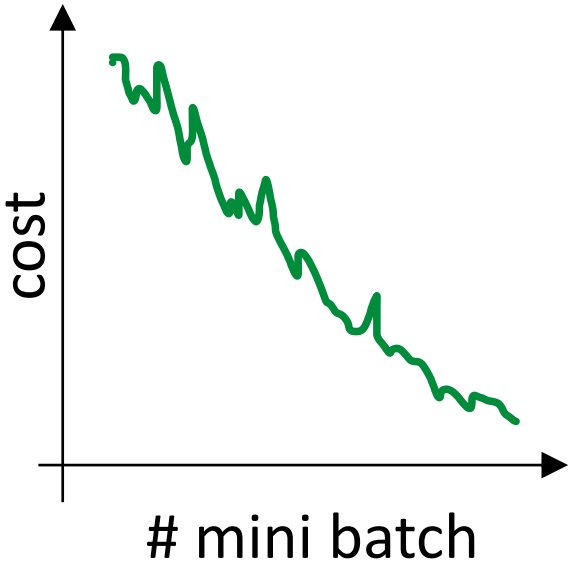
- Guess an initial learning rate.
  - If the error goes up or shows wild oscillations, reduce the learning rate.
  - If the error is decreasing but slowly, increase the learning rate.
- Various methods are proposed in the literature to 'learn' the learning rate.

Effect on Convergence with Mini Batches

Batch gradient descent



Mini-batch gradient descent



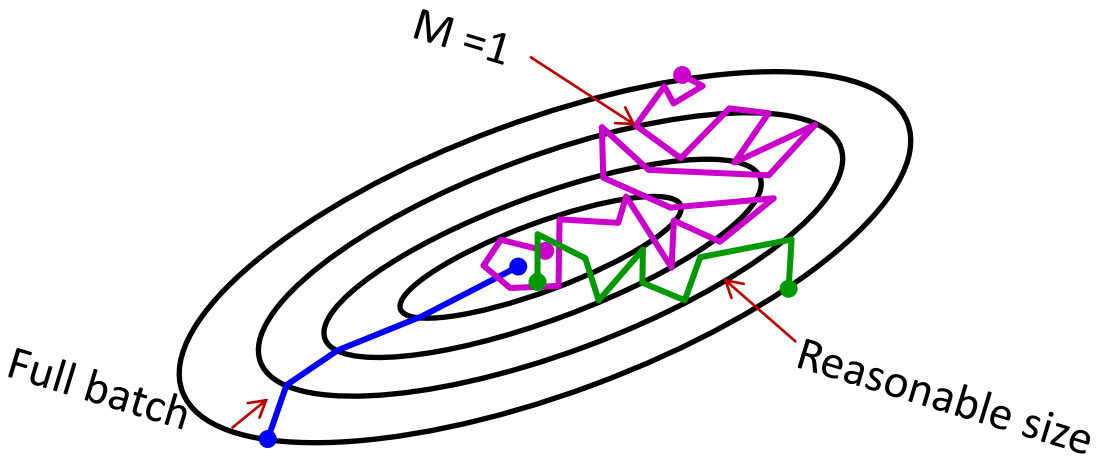
Although there are fluctuations in the cost graph because the samples in one batch may be having different characteristics than others. But the algorithm learns faster.

Mini-Batch Size

- Not too small, not too large
- Example :
  - If M =m, then it reduces to original full batch gradient descent.
  - If M =1, then it reduces to what is called as stochastic gradient descent.
  - Typically mini-batches are chosen as 64, 128, 256 ..

Mini-Batch Size

- Larger batches provide a more accurate estimate of the gradient, but are slow.
- If very small batches are used, multicore architectures remain underutilized.
- So, some absolute minimum batch sizes are selected that lead to optimum time to process a minibatch.
  - Not much reduction if you use a smaller minibatch.



Sampling Mini-Batches

- Random sampling of mini-batches is important.
- To compute an unbiased estimate of the expected gradient from a set of samples, samples need to be independent.
- Two subsequent gradient estimates will be independent only when two subsequent minibatches of examples are independent from each other.

Sampling Mini-Batches

- Most datasets are naturally correlated.
- Example: Consecutive video frames in a video for object detection in a video.
- So it is necessity to shuffle the samples before selecting every minibatch. Else the algorithm's performance may be compromised.
- In practice shuffling the dataset before every minibatch selection is not feasible, especially if the dataset is large. So some wise methods need to be chosen.

Main Challenges in Dealing with Deep Learning Models

- Optimization with an objective function of several variables is an extremely difficult task.
- Traditionally, machine learning approaches avoid this problem by carefully designing the objective function and constraints to ensure that the problem is **convex**.
- Convex optimization itself is a complex problem. However, the problem that we face in deep learning models is that the objective functions are **non-convex**.
- Let us discuss some of the most prominent problems in deep model training.

Main Challenges in Dealing with Deep Learning Models

- Ill-conditioning
- Local minima
- Plateaus, saddle points and other flat regions
- Cliffs and exploding gradients
- Long-term dependencies
- Inexact gradients
- Poor correspondence between local & global structure
- Theoretical limits of optimization

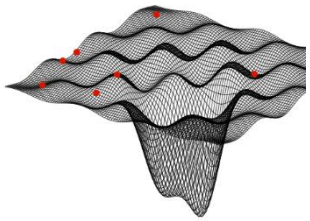
Main Challenges in Dealing with Deep Learning Models

- Ill- Conditioning
  - Like the gradient tells us which direction is leading to the maximum rate of change in function,
  - The second order derivative tells us whether the first derivative is increasing or decreasing.
  - This tells about the function's curvature.
  - Second Order Derivative provides us with a quadratic surface which touches the curvature of the error surface.
  - Recall the Taylor expansion to approximate a function at a point.

$$f(x) \approx f(a) + \nabla f(a) \cdot (x-a) + (x-a)^T Hf(a) \cdot (x-a) + \cdots$$

III- Conditioning

- Since the objective function is having a large number of independent parameters, second derivatives in each direction are different at a single point.
- The condition number of the Hessian at this point measures how much the second derivatives differ from each other.
- When the Hessian has a large condition number, gradient descent performs poorly.
- This is because in one direction, the derivative increases rapidly, while in another direction, it increases slowly.



What is Condition Number?

- A condition number for a matrix related with a computational task measures how sensitive the output is, to a small perturbation in the input data, and to roundoff errors made during the solution process.

- In terms of eigenvalues of a matrix A, its condition number can be represented as

$$\kappa(A) = \frac{|\lambda_{max}|}{|\lambda_{min}|}.$$

- λ_{max} and λ_{min} are the maximum and minimum eigenvalues.

- Larger the number $\kappa(A)$, harder it makes to converge to the solution . This is called **ill conditioning**.

Ill Conditioning

- Although ill-conditioning problem exists in other optimization problems also, techniques used to combat it in other types of models are not so useful to neural networks.
- For example, Newton's method is an excellent tool for minimizing convex functions with poorly conditioned Hessian matrices, but it does not apply directly to solve ill condition problem in the context of NNs.
  - Significant modifications needed in the Newton's method.

Local Minima

- A convex optimization problem can be reduced to the problem of finding a local minimum.

- But in NN, there are several local minima due to highly non-convex nature of its objective function.

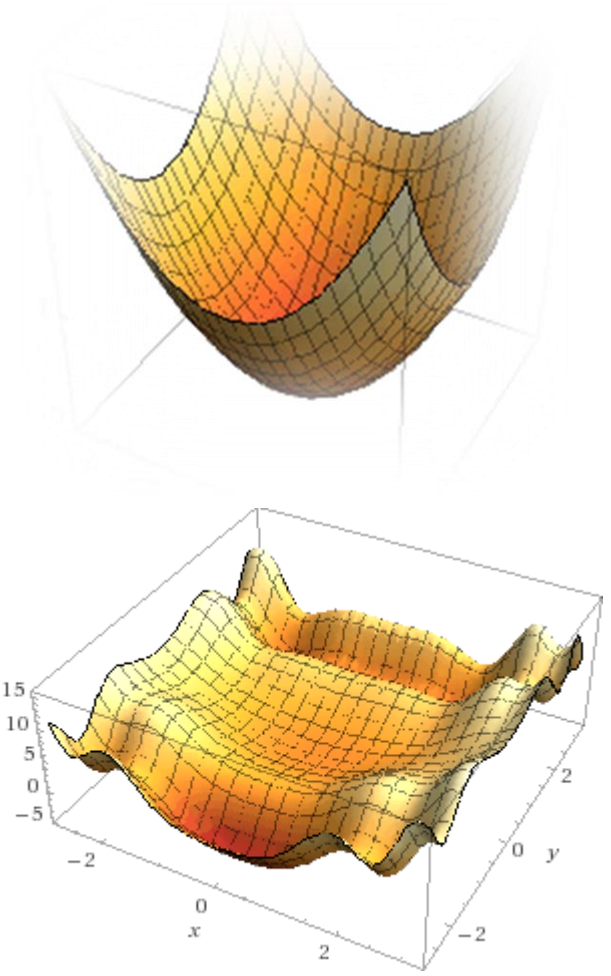


Figure source: https://www.svm-tutorial.com

Plateaus, Saddle Points and Other Flat Regions

- Many researchers and developers attribute nearly all difficulty with neural network optimization to local minima.

- But in high dimensional spaces, it is very difficult to establish that local minima are the problem.

- Many structures other than local minima also have small gradients. These include **Plateaus, Saddle Points and Other Flat Regions**.

Saddle Points

- A saddle point is a point where the Hessian matrix has both positive and negative eigenvalues.
- We can think of a saddle point as being a local minimum along one cross-section of the cost function and a local maximum along another cross-section.

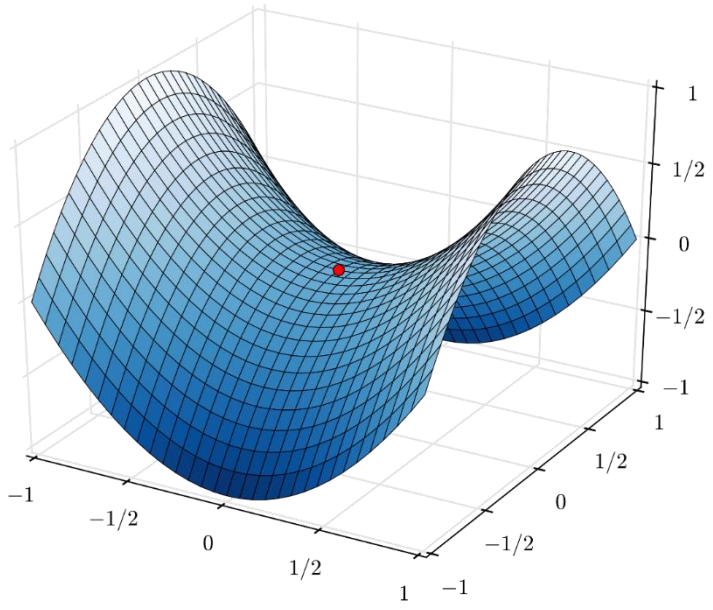


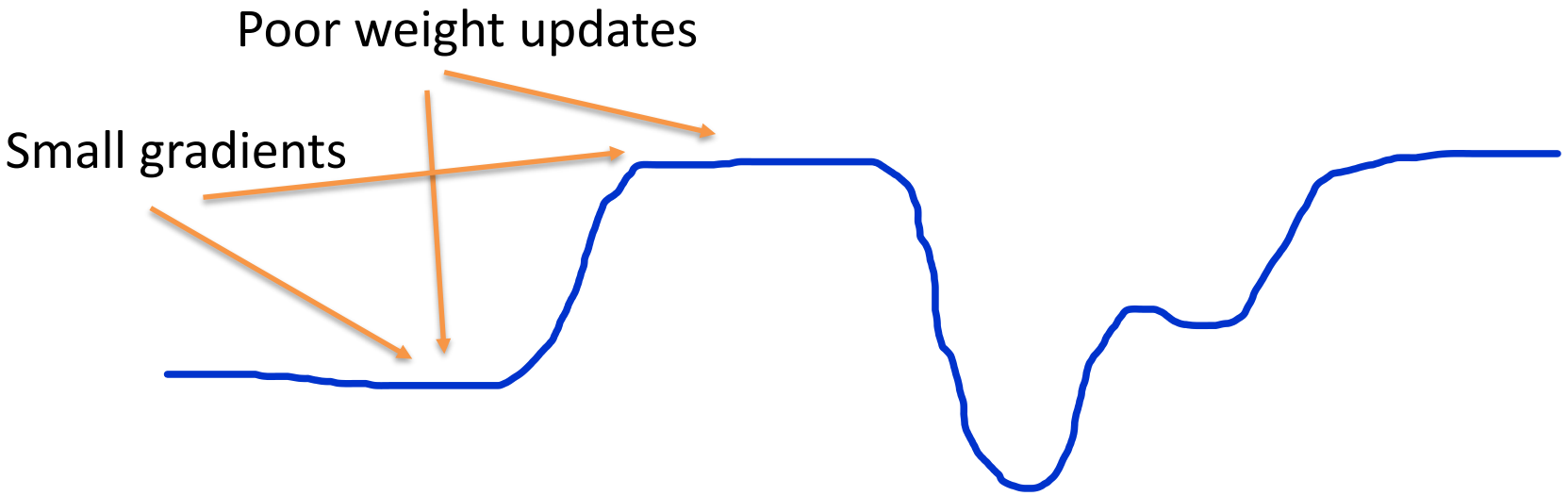
Figure source: https://www.svm-tutorial.com

Saddle Points

- Many classes of random functions exhibit the following behaviour:
- In lower dimensional spaces, local minima and maxima are common.
- In higher dimensional spaces, local minima and maxima are not that important, saddle points are much more common problems.
- For a function of n-variables , the expected ratio of the number of saddle points to local minima grows exponentially with *n*.

Saddle Points, Plateaus and Valleys

- Primary obstacle is not multiple minima but saddle points.
- Most of training time in NN is spent on traversing flat valley of the Hessian matrix or circumnavigating tall "mountain" via an indirect arcing path.



Cliffs and Exploding Gradients

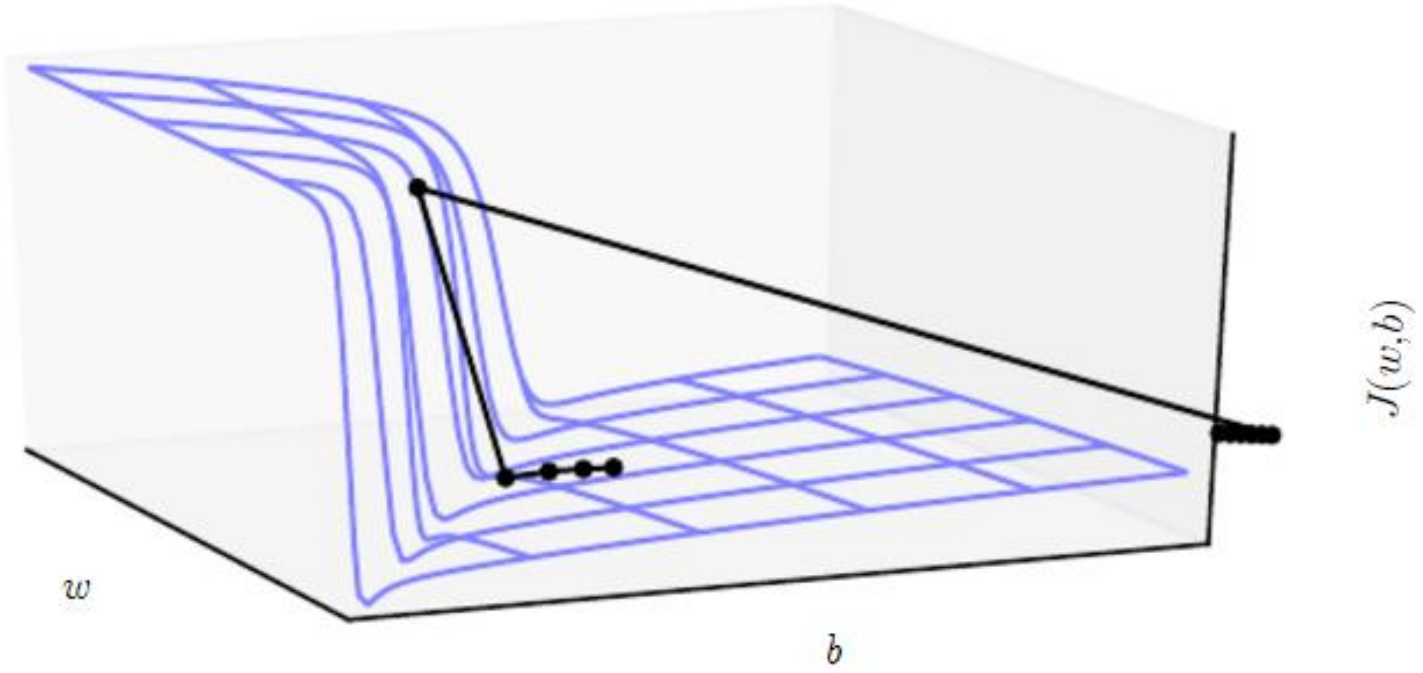


Figure source: Ian Goodfellow et al. , Deep Learning, MIT Press, 2017

Cliffs and Exploding Gradients

- Neural networks with large number of hidden layers
  - Have steep regions resembling cliffs.
  - Result from multiplying several large weights.
  - For example, RNNs with many factors at each time step.

- Gradient update step is likely to move parameters extremely far, just like jumping off the cliff.

- Cliffs are always problematic from either direction.

- Solution ?
  - *Gradient clipping* heuristics can be used.

Long-Term Dependencies

- When computational graphs become extremely deep.
  - Example: 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 **Exploding or Vanishing Gradient Problems**.
- Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function
- Exploding gradients can make learning unstable. The cliff structures lead to exploding gradient phenomenon as discussed earlier.

Long-Term Dependencies

- Suppose there are t layers.
- Hypothetically, assume that the same weight matrix W is used in all the layers.
- Therefore there will be t multiples of W in the end.
- Eigenvalue decomposition gives the following form of W.

$$W = V^{-1} diag(\lambda) V$$

$$\therefore\ W^t = V^{-1} diag(\lambda)^t V$$

- An eigenvalue λ_i that is larger than 1 in magnitude will result in exploding gradient and those with magnitude less than 1 will result in vanishing gradient problems.
- Actually the **vanishing and exploding gradient problem** arise due to the fact that gradients through such a computational graph are scaled according to diag(λ).

Inexact Gradients

- Optimization algorithms assume that the gradient or Hessian matrix is exact.
- In practice we have a noisy or biased estimate due to noise in samples and round off errors.
- In some cases the objective function is intractable, so the gradient is intractable as well.
- Various neural network optimization algorithms are designed to account for imperfections in the gradient estimate.
- One can also avoid the problem by choosing a surrogate loss function that is easier to approximate than the true loss.

Poor Correspondence between Local and Global Structure

- It can be difficult to proceed in the right direction if
- $J(\theta)$ is poorly conditioned at the current point θ ;
  - θ lies on a cliff
  - θ is a saddle point hiding the opportunity to make progress downhill from the gradient
- If the direction that makes most improvement locally does not point towards distant regions of much lower cost or a global minimum, then the algorithm may fail to provide an optimum value.

Poor Correspondence between Local and Global Structure

- 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.

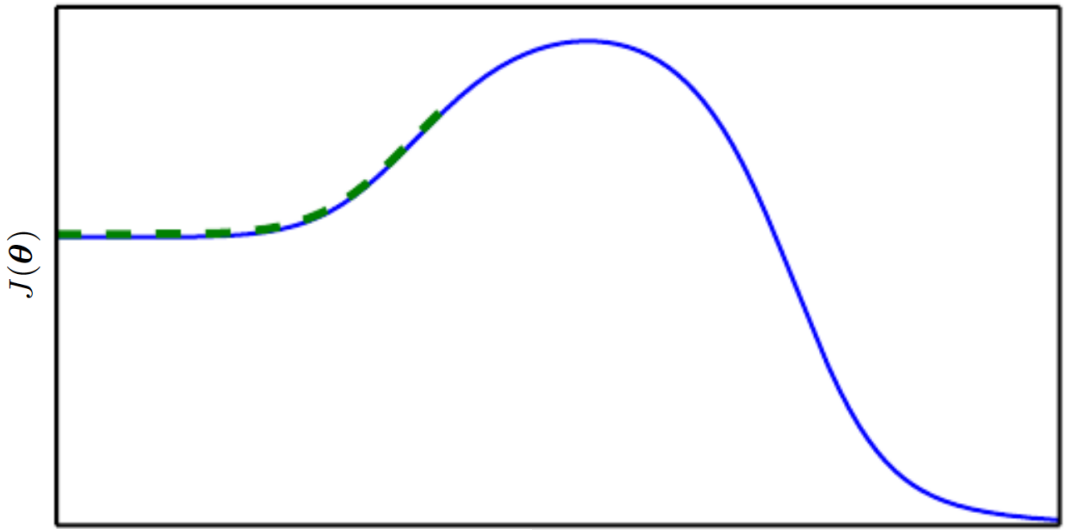


Figure source: Ian Goodfellow et al., Deep Learning, MIT Press, 2017

Theoretical Limits

- There are limits on the performance of any optimization algorithm we might design for neural networks.
- Some show that there exist problem classes that are intractable.
  - But difficult to tell whether a given problem falls in that class.
- Theoretical analysis of an optimization algorithm is difficult.
- More realistic bounds on the performance of optimization algorithms is a research problem.

Optimization in Neural Networks

- Mini-batch Gradient Descent
- A few optimization algorithms faster than gradient descent:
  - Gradient Descent with Momentum
  - RMSProp (Root Mean Square Prop)
  - Adam Optimization

Recall…

- Batch gradient descent can be very slow
  - One needs to calculate the gradients for the whole dataset to perform just *one* update of the parameter.
- Batch gradient descent also doesn't allow to update the model for new examples – *online*, while it is working.
- Stochastic gradient descent on the other hand updates the parameter after each training example.
- So the algorithm learns through examples more quickly than the batch gradient descent.
- But its cost function converges with lot of fluctuations.

Recall…

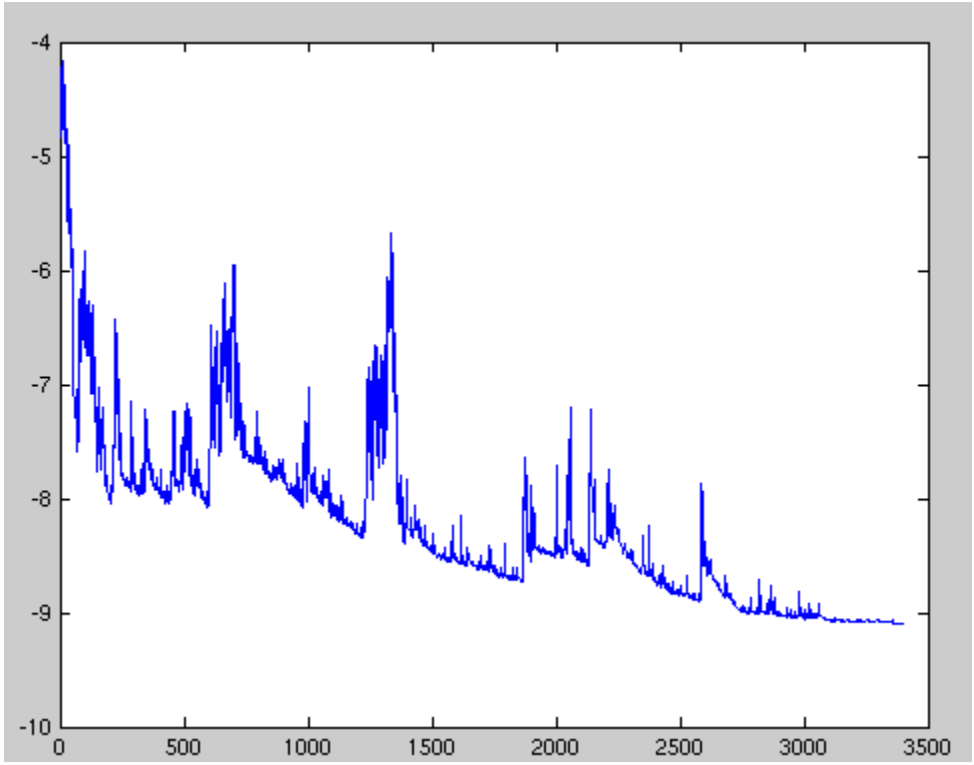


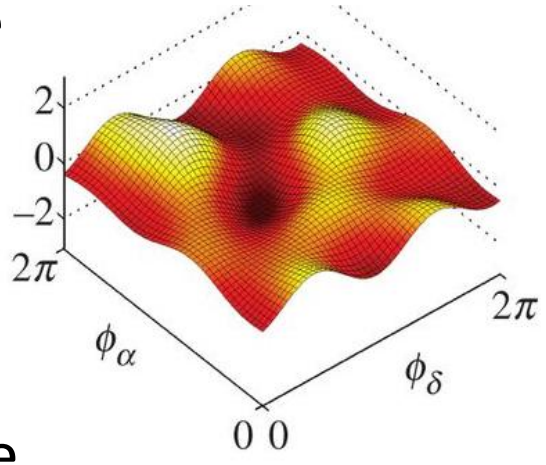
Figure from Wikipedia

Recall…

- Mini-Batch gradient descent is an alternative that divides the whole data into mini-batches.
- So it is faster than the batch gradient descent and also provides opportunity to update parameters after each minibatch.
- Challenges with minibatch gradient descent
  - Choosing a proper learning rate.
  - The same learning rate applies to different types of minibatches.
  - how to deal with getting stuck in local minima
- So gradient descent optimization algorithms are applied.

What to do …

- If the surface is highly undulated, then the stochastic gradient descent or minibatch gradient descent methods get stuck in local minima.
- To avoid this situation, a fraction of the past update vector(s) is combined with the current update vector.
- This results in
  - increase in the momentum for dimensions where gradients point in the same directions.
  - reducing the updates for dimensions whose gradients change directions.
  - faster convergence and reduced oscillation is achieved by this way.



Exponentially Decaying Moving Average

Exponentially Decaying Moving Average

- Taking weighted average of the data that has exponential decay in weights.

- For example, if the data is given as follows
  - $x_1, x_2, \dots, x_m$
- Define a factor $0 < \beta < 1$ to take average.
- Define $v_0 = 0$, $v_1 = \beta v_0 + (1 - \beta)x_1$

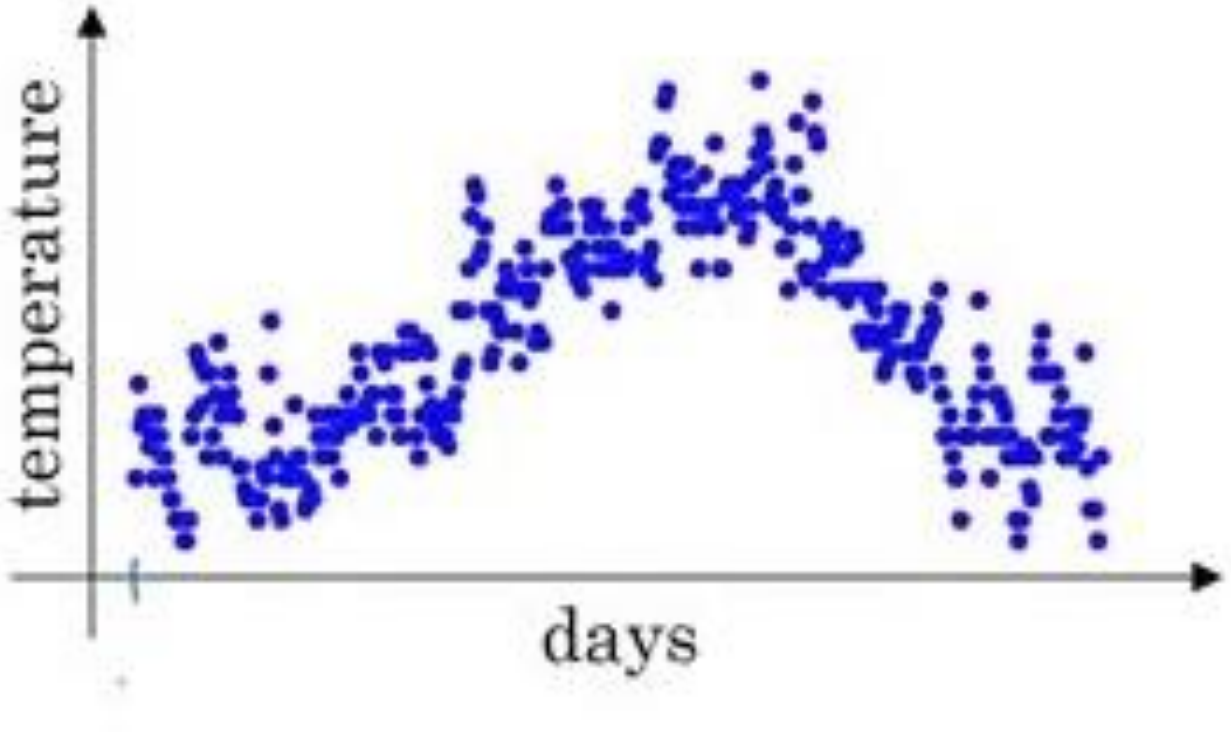
$$v_2 = \beta v_1 + (1 - \beta)x_2$$

$$\dots$$

$$v_t = \beta v_{t-1} + (1 - \beta)x_t$$

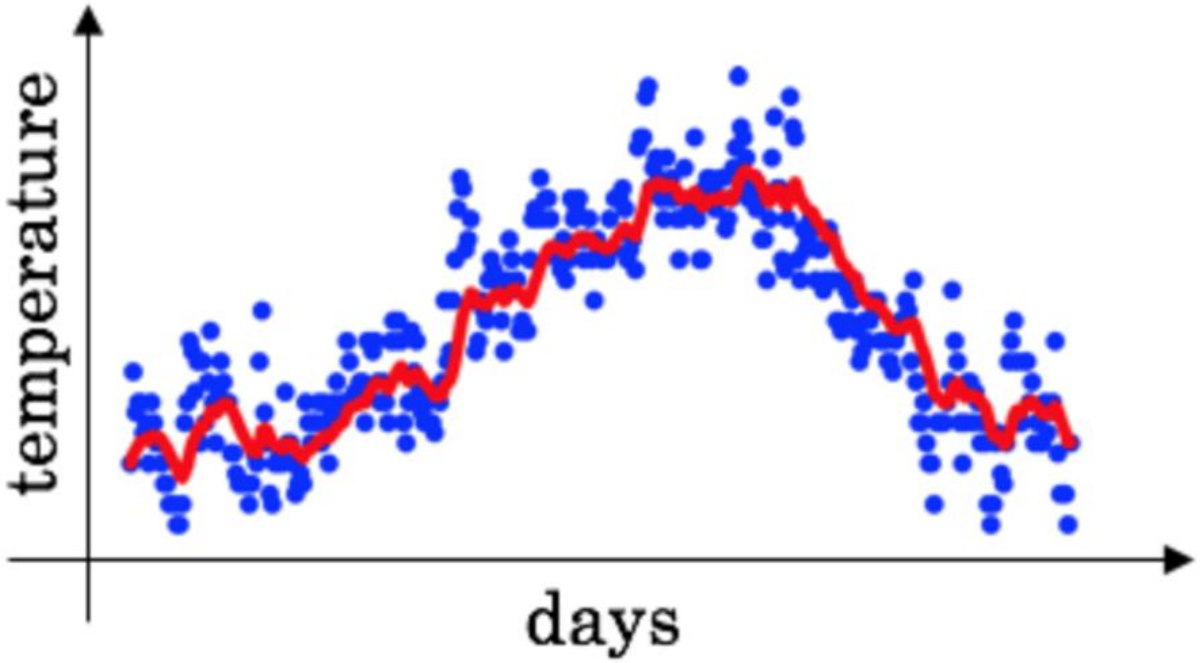
Exponentially Decaying Moving Average

- Data distribution



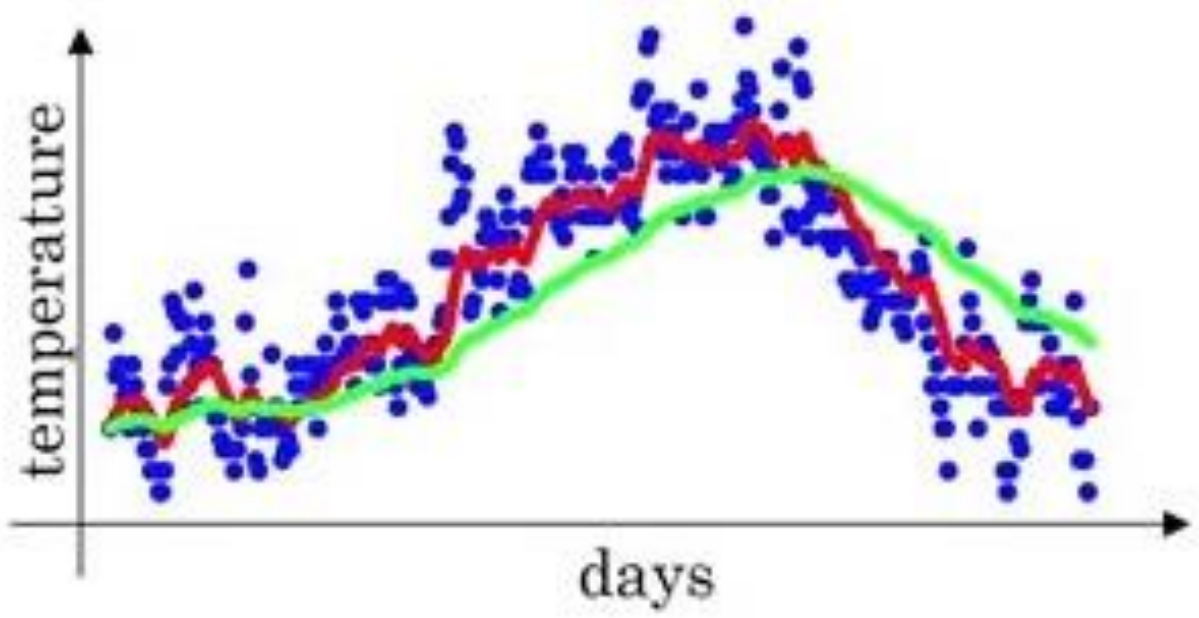
Exponentially Decaying Moving Average

- Weighted average for $\beta = 0.9$



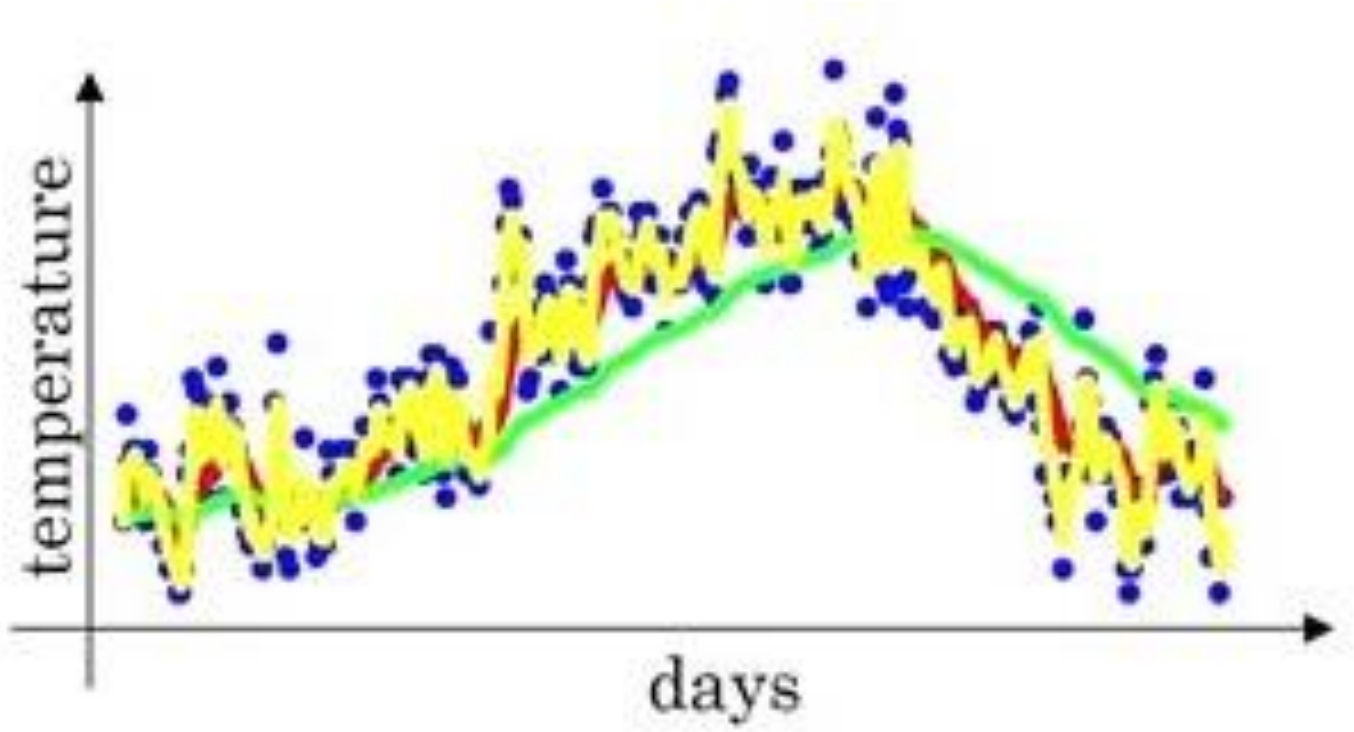
Exponentially Decaying Moving Average

- Weighted average for $\beta = 0.98$



Exponentially Decaying Moving Average

- Weighted average for $\beta = 0.5$



Understanding Exponentially Decaying Moving Average

$$v_t = \beta v_{t-1} + (1-\beta)x_t$$

$$v_{100} = 0.9v_{99} + 0.1x_{100}$$

$$v_{99} = 0.9v_{98} + 0.1x_{99}$$

$$v_{98} = 0.9v_{98} + 0.1x_{99} \quad \ldots.$$

$$v_{100} = 0.1x_{100} + 0.9v_{99}$$

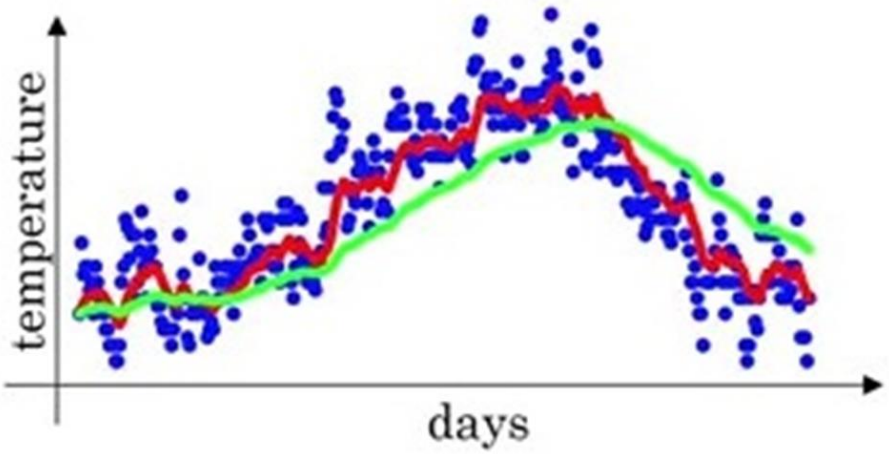
$$= 0.1x_{100} + 0.9(0.1x_{99} + 0.9v_{98})$$

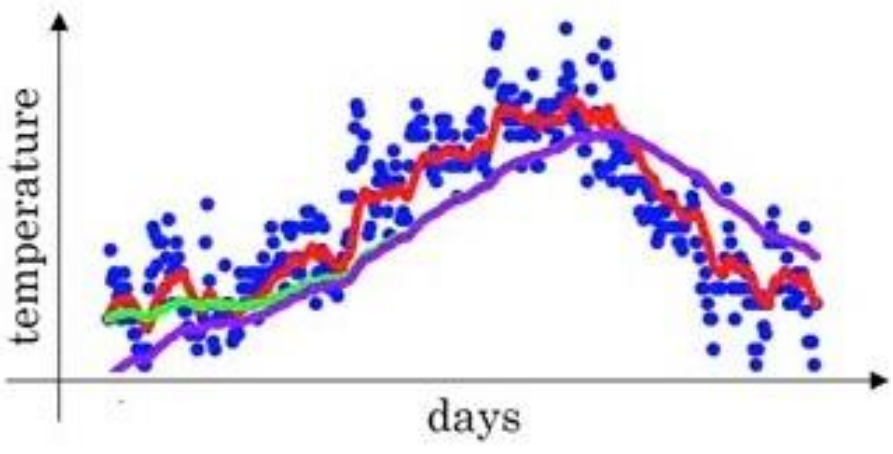
$$= 0.1x_{100} + 0.9(0.1x_{99} + 0.9(0.1x_{98} + 0.9v_{97}))$$

$$= 0.1x_{100} + 0.1(0.9)x_{99} + 0.1(0.9)^2x_{98} + 0.1(0.9)^3v_{97} \quad +0.1(0.9)^4v_{96} + \ldots$$

Bias Correction

- Weighted average for $\beta = 0.98$





$$v_t = \beta v_{t-1} + (1 - \beta) x_t$$

- Bias correction helps improve initial estimations and averaging in exponential weighted average.

Bias Correction

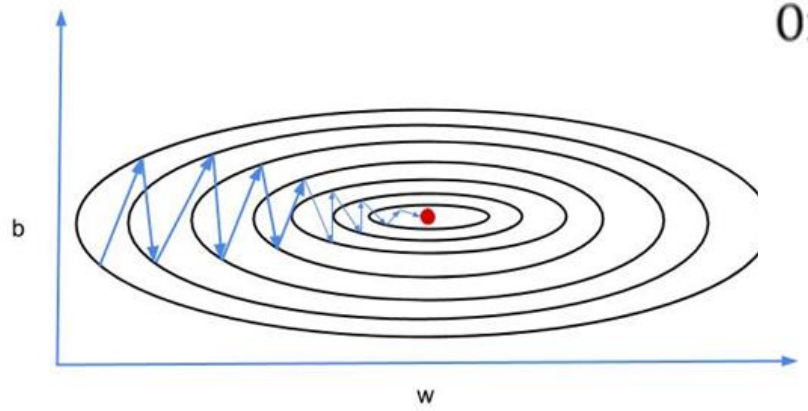
- $v_t = \beta v_{t-1} + (1-\beta)x_t$
- v_t is replaced by $\frac{v_t}{(1-\beta^t)}$
- As a result, bias correction is done right from the initial value of $v_t = 0$.
- Example: if $\beta = 0.98$ and $t = 2$ then
- $v_t = 0.98v_1 + 0.02x_2$ and $\frac{v_2}{(1-\beta^2)} = \frac{0.98(0.02x_1)+0.02x_2}{0.0396}$
- When t is large enough, the bias correction makes almost no difference.

Gradient Descent with Momentum

Gradient Descent with Momentum

- The method of gradient descent with momentum is designed to accelerate learning.
- When the error surface has high curvature, small but consistent gradients. Sometimes noisy gradients.
- The algorithm accumulates an **exponentially decaying moving average** of past gradients and continues to move in their direction.

Gradient Descent with Momentum



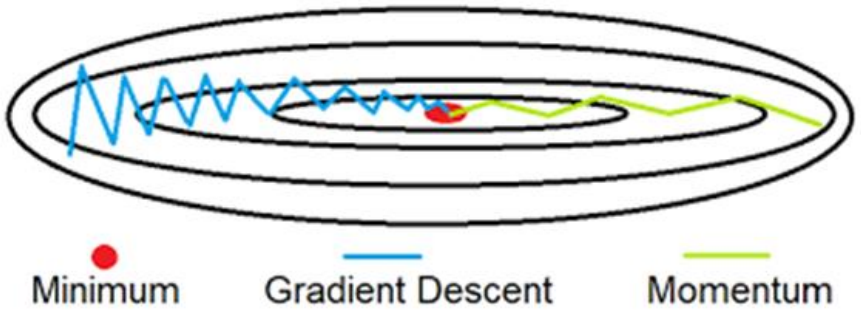
On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta)dW$$

$$v_{db} = \beta v_{db} + (1-\beta)db$$

- We have discussed batch gradient descent earlier. In batch gradient descent we use $W = W - \alpha v_{dW},\ b = b - \alpha v_{db}$



α determines the size of the step taken in reaching the minimum.

Gradient Descent with Momentum

On iteration t:

Compute dW, db on the current mini-batch

$$v_{dw} = \beta \ \ v_{dW} + \cancel{(1 - \beta_1)}. dW$$

$$v_{db} = \beta \ \ v_{db} + \cancel{(1 - \beta_1)}. db$$

$$W = W \ - \ \alpha v_{dW} \qquad b = b \ - \ \alpha v_{db}$$

Can be written as:

$$v \ = \ \beta \, v \ - \alpha \nabla J(\theta)$$

Hyperparameters: α, β

Gradient Descent with Momentum: Algorithm

- **Inputs:** Learning rate α, momentum parameter β, initial parameter θ and b, initial velocity v.

while stopping criterion not met **do**

Sample a minibatch of k examples $\{\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^k\}$ from the training set with corresponding targets $\{y^1, y^2, \ldots, y^k\}$.

Compute gradient estimate: $\nabla J(\theta) = \frac{1}{k} \nabla_\theta \sum_{i=1\ldots k} L(\hat{y}^i, y^i)$

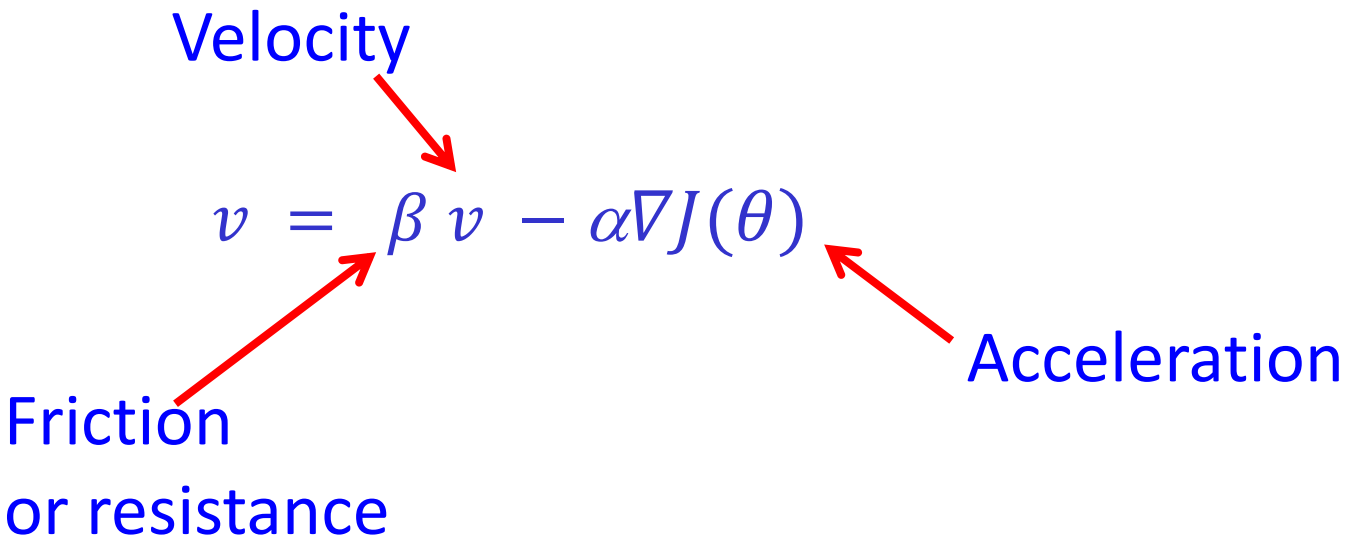
Compute velocity update: $v = \beta v - \alpha \nabla J(\theta)$

Apply update: $\theta = \theta + v$

end while

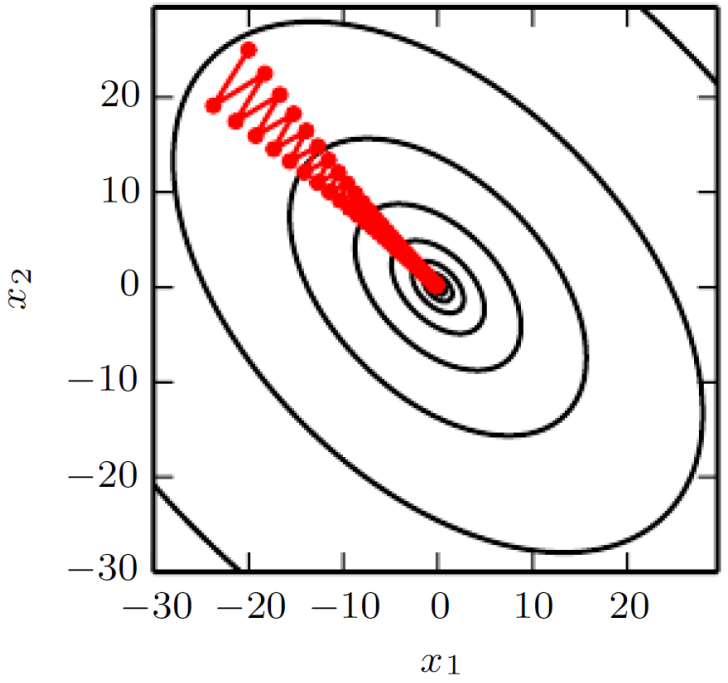
Hyperparameters: α, β

Gradient Descent with Momentum: Algorithm

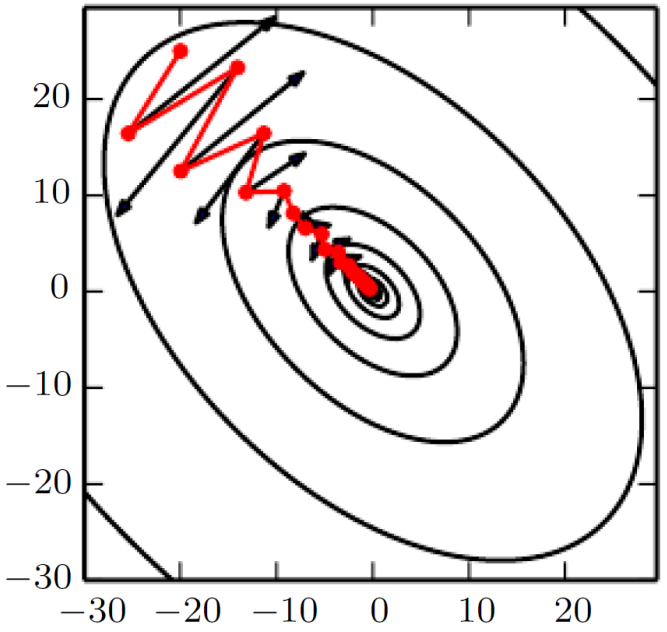


- The gradient descent algorithm simply moves the solution a single step based on each gradient.
- The momentum algorithm instead alters the velocity of the motion at each step.

Gradient Descent with Momentum …



GD without momentum



GD with momentum

Figure source: © Ian Goodfellow et al. , Deep Learning, MIT Press, 2017

Algorithms with Adaptive Learning Rates

- Learning rate is one of the most difficult hyperparameters to set because it has a significant impact on model performance.
- The momentum algorithm controls the downhill movement by introducing another hyperparameter.
- Can there be another way ?
- Adaptive learning rate are used to mitigate the effect of learning rate initialization leading to probably wrong direction of motion.

RMSProp
Using Adaptive Learning Rate

RMSProp

On iteration t:

Compute the derivative dW, db on the current mini-batch

$$S_{dW} = \beta_1 S_{dW} + (1 - \beta_1).dW^2$$

$$S_{db} = \beta_1 S_{db} + (1 - \beta_1).db^2$$

$$W = W - \alpha.\frac{dW}{\sqrt{S_{dW}}} \qquad b = b - \alpha.\frac{db}{\sqrt{S_{db}}}$$

- Maintain a moving (discounted) average of the square of gradients
- Divide the gradient by the root of this average

RMSProp

On iteration t:

Compute the derivative dW, db on the current mini-batch

$$S_{dW} = \beta_1 S_{dW} + (1 - \beta_1).dW^2$$

$$S_{db} = \beta_1 S_{db} + (1 - \beta_1).db^2$$

$$W = W - \alpha.\frac{dW}{\sqrt{S_{dW}+\epsilon}} \qquad b = b - \alpha.\frac{db}{\sqrt{S_{db}+\epsilon}}$$

RMSProp: Algorithm

- RMSProp also works like Gradient Descent with momentum with the following change –
  - For each mini-batch (t)
    - Compute$\nabla J(\theta)$ on the current minibatch
    - Define $S_{d\theta} = \beta_1 S_{d\theta} + (1 - \beta_1)\nabla J(\theta)^2$
    - $\theta = \theta - \frac{\alpha}{\sqrt{S_{d\theta} + \varepsilon}} \nabla J(\theta)$
    - ε is added to avoid division by zero.

$\nabla J(\theta)^2$ is computed by element wise multiplication of terms

Adam Optimization

Adam Optimization

- Adaptive moment estimation (Adam)
- Combines the concept of RMSProp and momentum
- Works well on a large class of problems

For the t^{th} minibatch

$$v_{dW} = \beta_1 v_{dW} + (1 - \beta_1).dW \qquad v_{db} = \beta_1 v_{db} + (1 - \beta_1).db$$

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2).dW^2 \qquad S_{db} = \beta_2 S_{db} + (1 - \beta_2).db^2$$

Adam Optimization

- Then we also perform bias correction:

$$v_{dW}^{corrected} = \frac{v_{dW}}{1-\beta_1^t} \quad and \quad v_{db}^{corrected} = \frac{v_{db}}{1-\beta_1^t}$$

$$s_{dW}^{corrected} = \frac{s_{dW}}{1-\beta_2^t} \quad and \quad s_{db}^{corrected} = \frac{s_{db}}{1-\beta_2^t}$$

- Finally, perform the update:

$$w = w - \alpha . \frac{v_{dw}^{corrected}}{\sqrt{s_{dw}^{corrected} + \epsilon}} \qquad b = b - \alpha . \frac{v_{db}^{corrected}}{\sqrt{s_{db}^{corrected} + \epsilon}}$$

Adam Optimization Algorithm

- initialize $v_{d\theta}, S_{d\theta} = 0$.
- For the t^{th} mini-batch

$$v_{d\theta} = \beta v_{d\theta} + (1-\beta)\theta$$

$$S_{d\theta} = \beta_1 S_{d\theta} + (1-\beta_1)\nabla J(\theta)^2$$

$$v_{d\theta} = \frac{v_{d\theta}}{(1-\beta^t)}; S_{d\theta} = \frac{S_{d\theta}}{(1-\beta^t)}$$

Finally, $\theta = \theta - \frac{\alpha}{\sqrt{s_{d\theta}+\varepsilon}} \nabla J(\theta)$

Adam Optimization Algorithm

- Hyperparameter selection:
  - α needs to be tuned.
  - $\beta = 0.9$ is a common choice.
  - $\beta_1 = 0.999$ as suggested by authors of Adam algorithm.

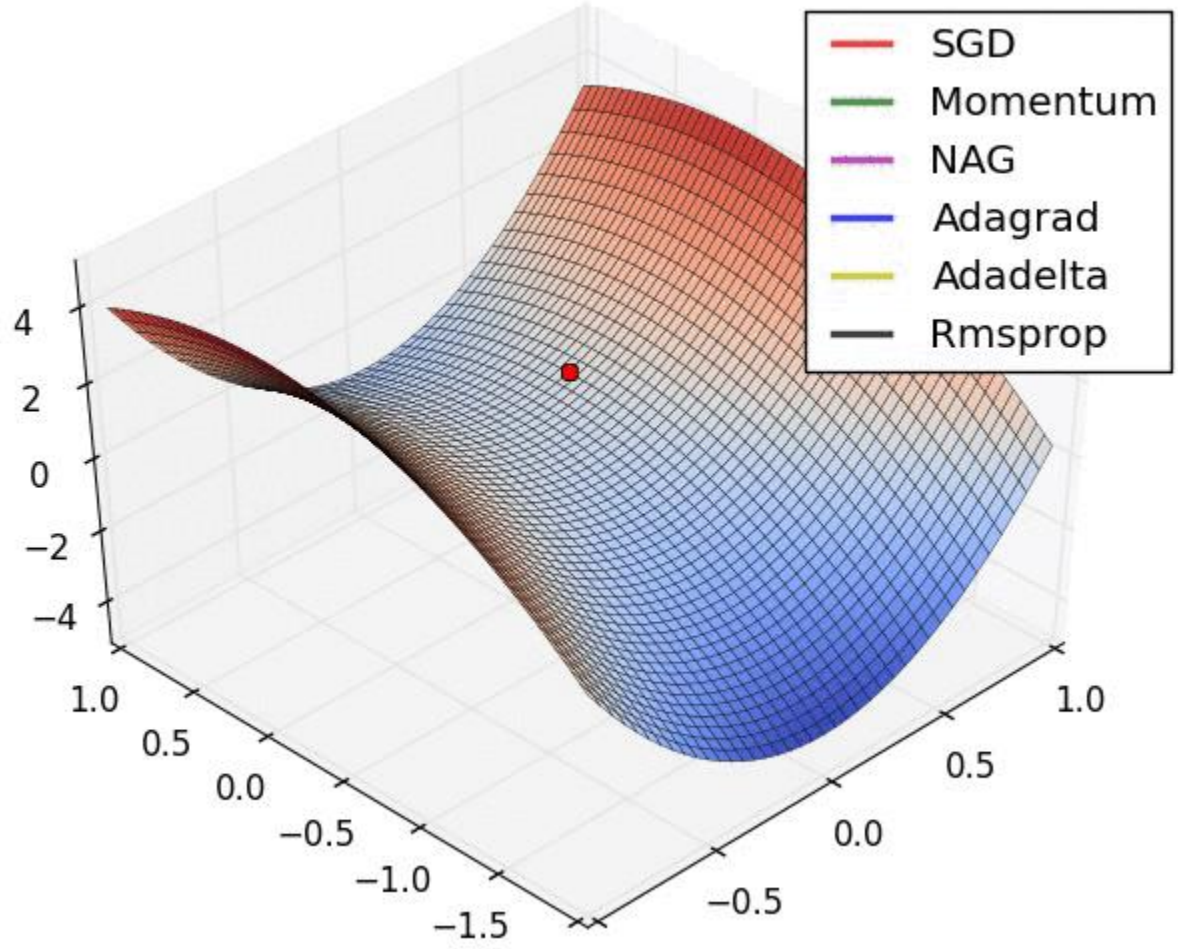
A Comparison

- MNIST dataset: 60,000 instances (10 classes, digits)
- 50,000 Train; 10,000 test

- Results on test set using three optimizers –

Measure	SGD	RMSprop	Adam
Loss	0.153962	0.1019028	0.0605893
Accuracy	0.9546	0.9824	0.9831

A Comparison…



Source: Kevin McGuinness's lecture slide

Reference

- Contents of these slides are taken from the text book.

Ian Goodfellow and Yoshua Bengio and Aaron Courville, Deep Learning, MIT Press, 2016

- Slides (48-71) are created in line with the concepts given by Andrew Ng in Coursera (deeplearning.ai).