Optimization in Deep Learning

Computer Science and Engineering, IIIT Dharwad

Arun Chauhan

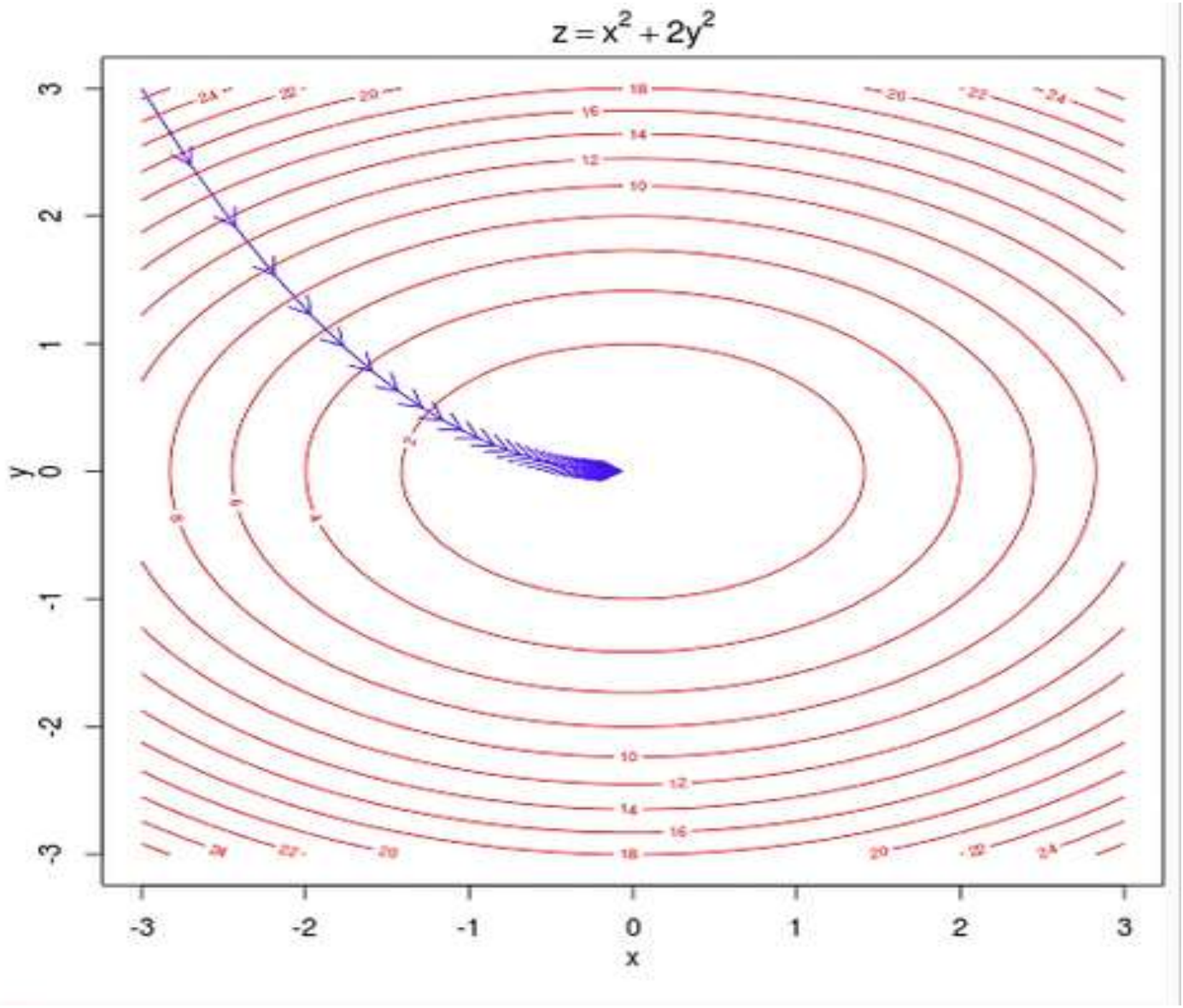
Optimization is essential for DL

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

- The recipe for linear regression
  1. Data set : X and y
  2. Cost function: $J(\boldsymbol{w}) = -E_{x,y\sim\hat{p}_{data}} \log p_{\text{model}}(y \mid \boldsymbol{x}) + \lambda \left\|\boldsymbol{w}\right\|_2^2$ Includes regularization
  3. Model specification: $p_{\text{model}}(y \mid \boldsymbol{x}) = N(y; \boldsymbol{x}^T\boldsymbol{w} + \boldsymbol{b}, 1)$
  4. Optimization algorithm

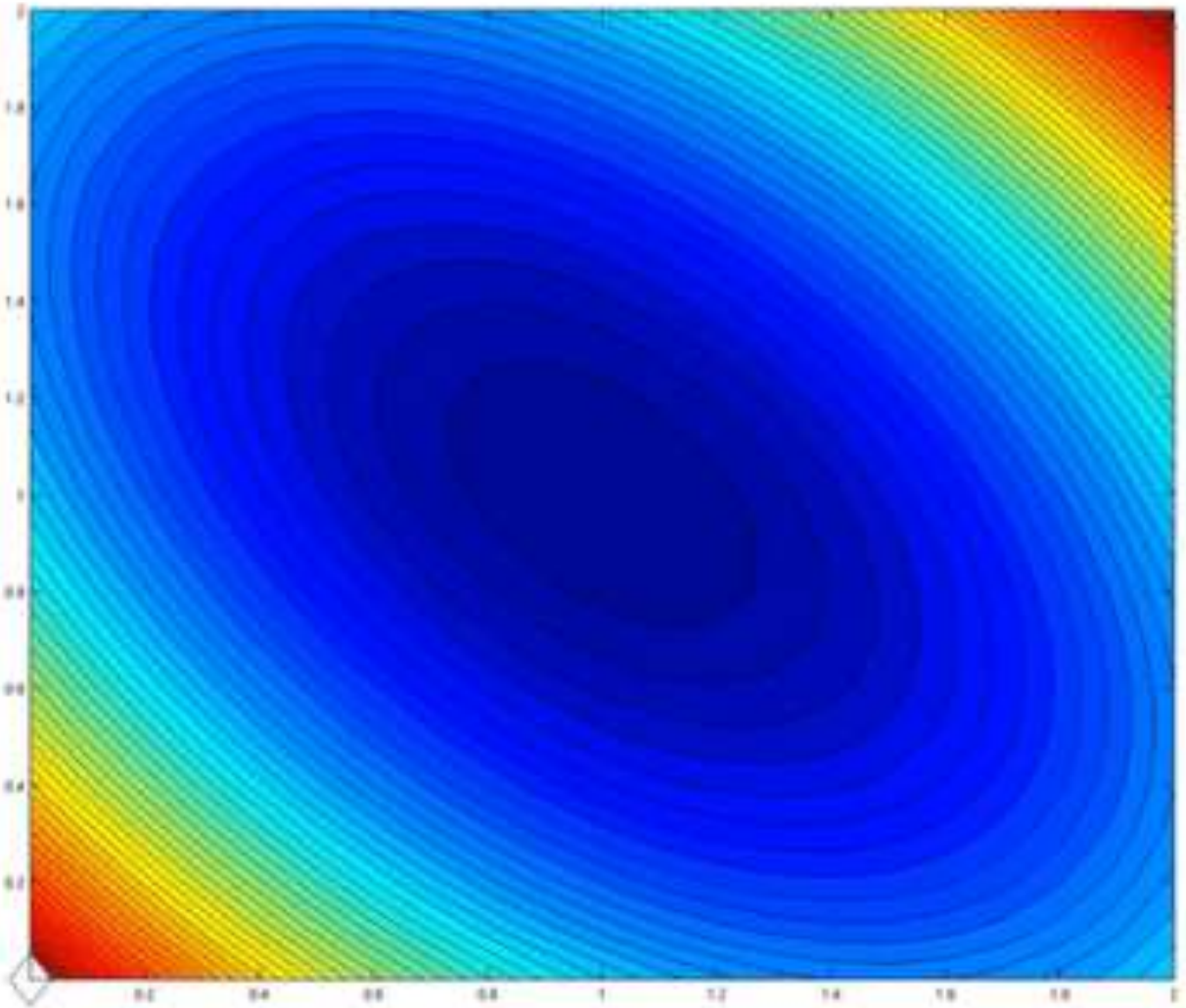
Gradient Descent

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

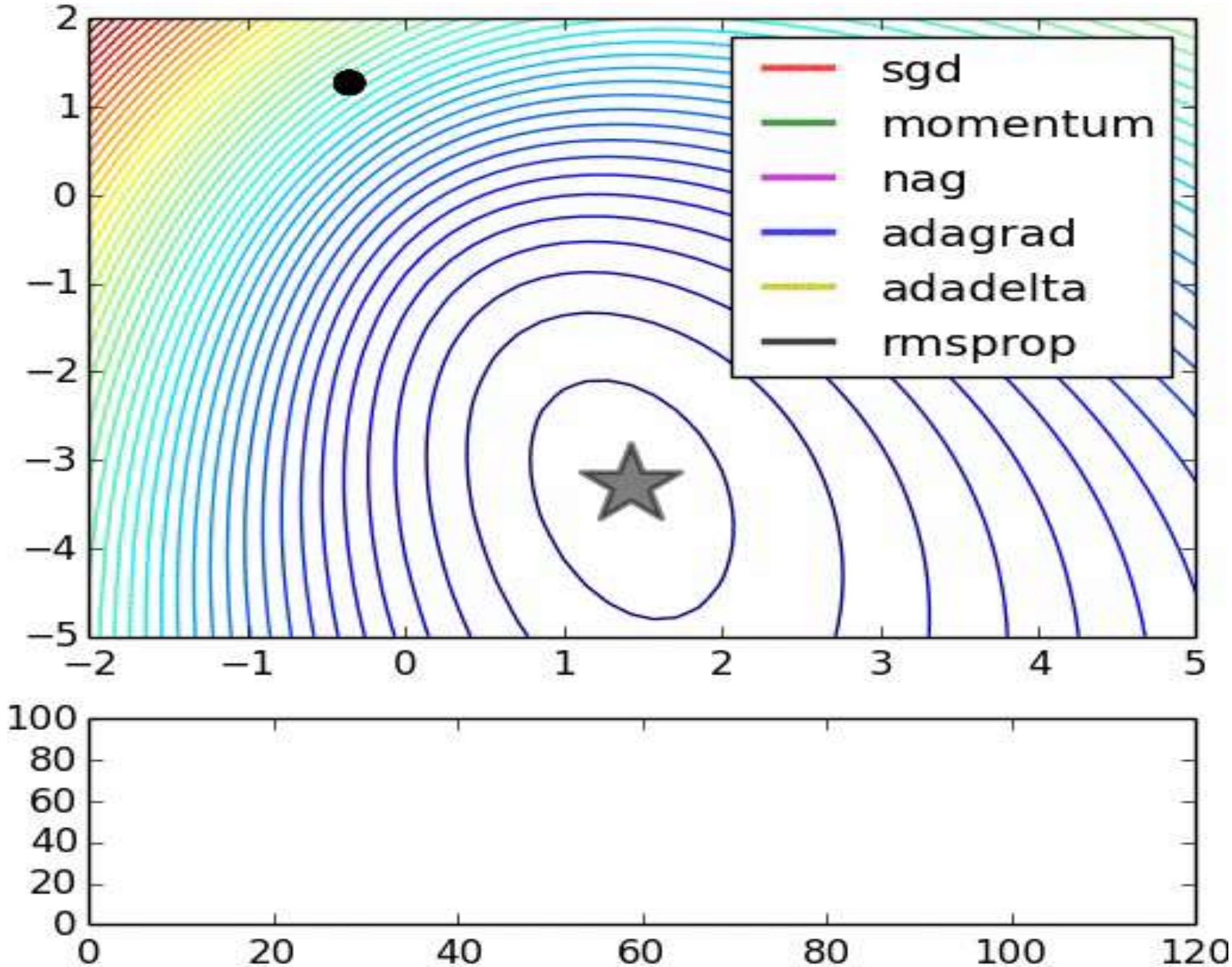


Coordinate Descent

$$f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 & 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 & 0 \end{pmatrix}^T$$



Optimization Algorithms



Learning vs Pure Optimization

- Optimization algorithms for deep learning differ from traditional optimization in several ways:
  - Machine learning acts indirectly
    - We care about some performance measure P defined wrt the training set which may be intractable
    - We reduce a different cost function $J(\boldsymbol{\theta})$ in the hope that doing so will reduce P
- Pure optimization: minimizing J is a goal in itself

Typical cost function

Typically the cost function can be written as an average over a training set

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

- Where
  - L is the per-example loss function
  - $f(\boldsymbol{x}\,;\boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$
  - $\hat{p}_{data}$ is the empirical distribution
- In supervised learning y is target output

Objective wrt data generation is risk

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

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

  - The goal of a machine learning algorithm is to reduce this expected generalization error
- This quantity is known as *risk*

Empirical Risk

- Empirical risk, with m training examples, is

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

  - Still similar to straightforward optimization
- But *empirical risk minimization* is not very useful:
  1. Prone to overfitting: can simply memorize training set
  2. SGD is commonly used, but many useful loss functions have 0-1 loss, with no useful derivatives (derivative is either 0 or undefined everywhere)
- We must use a slightly different approach
  - Quantity we must optimize is even more different from what we truly want to optimize

Surrogate Loss: Log-likelihood

- Sometimes, the loss function we actually care about (say, classification error) is not one that can be optimized efficiently
- In such situations use a surrogate loss function (Log-likelihood)
- Surrogate may learn more
  - one can improve classifier robustness by further pushing the classes apart
- Learning does not stop at minimum like pure optimization problem
  - Often stops when derivatives are still large (Early stopping)

Batch and Minibatch Algorithms

- Objective function usually decomposes as a sum over the training examples

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})$$

- Typically, update on parameters are based on expected value of cost function
  - estimated using only a subset of the terms (mini batch gradient decent)

https://www.quora.com/How-does-one-show-that-the-expected-value-of-a-mini-batch-in-SGD-is-equal-to-the-true-empirical-gradient

https://math.stackexchange.com/questions/1962991/expectation-of-gradient-in-stochastic-gradient-descent-algorithm

Batch and Minibatch Algorithms

- Maximum likelihood estimation problems, when viewed in log space

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})$$

Maximizing this sum is equivalent to maximizing the expectation over the empirical distribution defined by the training set:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}).$$

- Also, most of our optimization algorithms are also expectations over the training set

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta})$$

EXPENSIVE !!!

Batch and Minibatch Algorithms

The standard error of the mean is given by

$$\mathrm{SE}(\hat{\mu}_m) = \sqrt{\mathrm{Var}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}$$

m examples: Less than linear

- Ex: 100 samples vs 10,000 samples
  - Computation increases by factor of 100 but
  - Error decreases by only a factor of 10

- Optimization algorithms converge much faster
  - if allowed to rapidly compute approximate estimates
  - rather than slowly compute exact gradient

Motivation: Estimate of gradient from sampling is redundancy

- Training set may be redundant
  - Worst case: all *m* examples are identical
    - Sampling based estimate could use *m* times less computation
  - In practice
    - unlikely to find worst case situation but
    - likely to find large no. of examples that all make similar contribution to gradient

(Batch)Gradient Decent Vs Stochastic Gradient Decent /online

- **(Batch) Gradient Descent**:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost = compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters
    # Update parameters.
    parameters = update_parameters(parameters, grads)
```

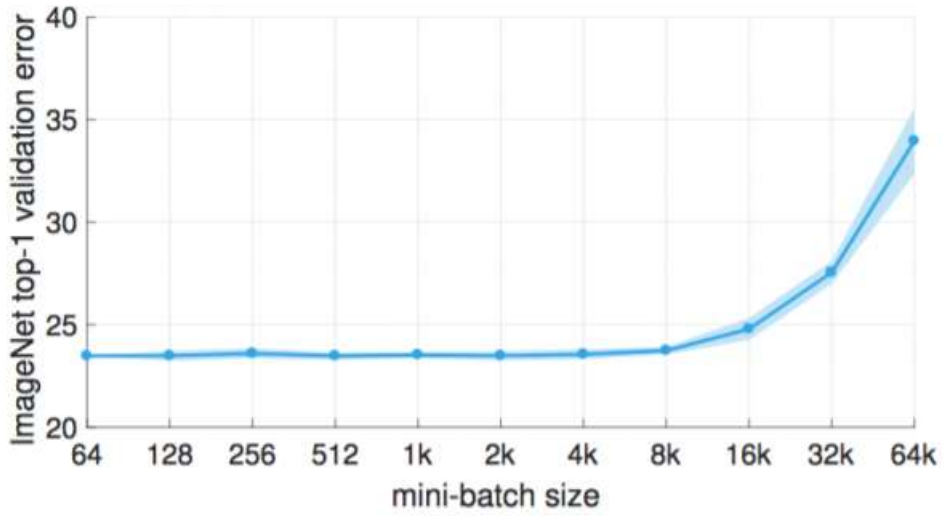
- **Stochastic Gradient Descent**:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    for j in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,j], parameters)
        # Compute cost
        cost = compute_cost(a, Y[:,j])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

Minibatch Size

- Driven by following factors
  - Larger batches→more accurate gradient but with less than linear returns
  - Multicore architectures are underutilized by extremely small batches
    - Use some minimum size below which there is no reduction in time to process a minibatch
  - If all examples processed in parallel, amount of memory scales with batch size
    - This is a limiting factor in batch size
  - GPU architectures more efficient with power of 2
    - Range from 32 to 256, sometimes with 16 for large models

Minibatch Size Vs Regularization



ImageNet dataset large minibatches cause optimization difficulties, but when these are addressed the trained networks exhibit good generalization. Specifically, no loss of accuracy when training with large minibatch sizes up to 8192 images.

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

Regularizing effect of small batches

- Small batches offer regularizing effect due to noise added in process
- Generalization is best for batch size of 1
- Small batch sizes require small learning rate
  - To maintain stability due to high variance in estimate of gradient
- Total run time can be high
  - Due to reduced learning rate and
  - Requires more time to observe entire training set

Use of minibatch information

- Different algorithms use different information from the minibatch
  - Some algorithms more sensitive to sampling error
- Algorithms using gradient $\boldsymbol{g}$ are robust and can handle smaller batch sizes like 100
- Second order methods using Hessian H and compute updates such as $\mathrm{H}^{-1}\boldsymbol{g}$ require much larger batch sizes like $10{,}000$

Random selection of minibatches

- Crucial to select minibatches randomly
- Computing expected gradient from a set of samples requires that sample independence
- Many data sets are arranged with successive samples highly correlated
  - E.g., blood sample data set has five samples for each patient
- Necessary to shuffle the samples
  - For a data set with billions of samples shuffle once and store it in shuffled fashion

https://www.quora.com/Why-do-we-need-to-shuffle-inputs-for-stochastic-gradient-descent

$$J(\theta) = \sum_{x \in \text{training set}} \mathcal{L}(\theta; x)$$

$$g_1 = \nabla_\theta \left[\frac{1}{M} \sum_{i=1}^{M} \mathcal{L}(\theta; x_i) \right]$$

$$g_2 = \nabla_\theta \left[\frac{1}{M} \sum_{i=M+1}^{2M} \mathcal{L}(\theta; x_i) \right]$$

$$g_3 = \nabla_\theta \left[\frac{1}{M} \sum_{i=2M+1}^{3M} \mathcal{L}(\theta; x_i) \right]$$

...

Each of these g_j is a **biased** estimate of $\nabla_\theta J(\theta)$. Why so? Take, for example, $\mathbb{E}[g_2]$, we have

$$\mathbb{E}_{x_i}[g_2] = \mathbb{E}_{x_i} \left[\nabla_\theta \left[\frac{1}{M} \sum_{i=M+1}^{2M} \mathcal{L}(\theta; x_i) \right] \right] = \nabla_\theta \left[\frac{1}{M} \sum_{i=M+1}^{2M} \mathcal{L}(\theta; x_i) \right]$$

Now let's see what happens if you permute the training data x_i. First, note that none of the gradients g_i is determined any more. All of them have the expected value

$$\mathbb{E}_{i \sim \text{Uniform}\{1,2,\ldots,N\}}[g]$$

$$= \mathbb{E}_{i \sim \text{Uniform}\{1,2,\ldots,N\}} \left[\nabla_\theta \left[\frac{1}{M} \sum_{i=1}^{M} \mathcal{L}(\theta; x_i) \right] \right]$$

$$= \nabla_\theta \frac{1}{M} \sum_{i=1}^{M} \mathbb{E}_{i \sim \text{Uniform}\{1,2,\ldots,N\}}[\mathcal{L}(\theta; x_i)]$$

$$= \nabla_\theta \frac{1}{M} \cdot M \cdot \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\theta; x_i)$$

$$= \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\theta; x_i)$$

SGD and generalization error

- Minibatch SGD follows the gradient of the true generalization error

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

  - As long as the examples are not repeated
- Implementations of minibatch SGD
  - Shuffle once and pass through multiple no. of times
    - On the first pass: each minibatch computes unbiased estimate of true generalization error
    - Second pass: estimate is more biased because it is formed by resampling values already used rather than fair samples from data generating distribution
- Easiest to see equivalence in online learning

Impact of growing data sets

- Data sets are growing more rapidly than computing power
- More common to use each training example only once
  - Or even make an incomplete pass through the data set
- With a large training set overfit is not an issue
  - Underfitting and computational efficiency become predominant concerns

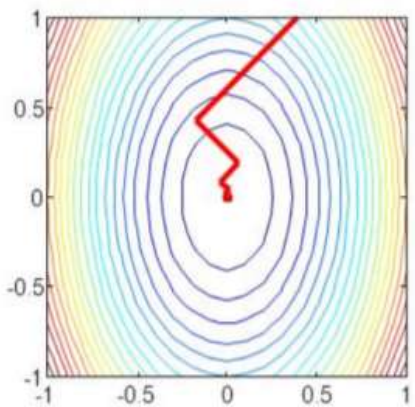
Challenges in Neural Network Optimization

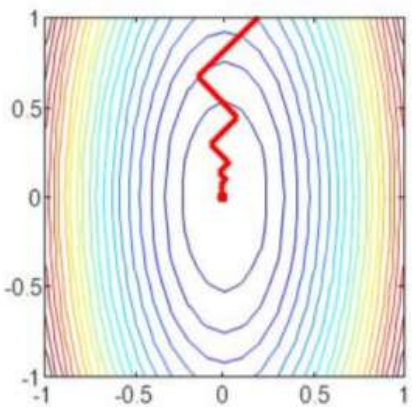
1. Ill-conditioning of the Hessian

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

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

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





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

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

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

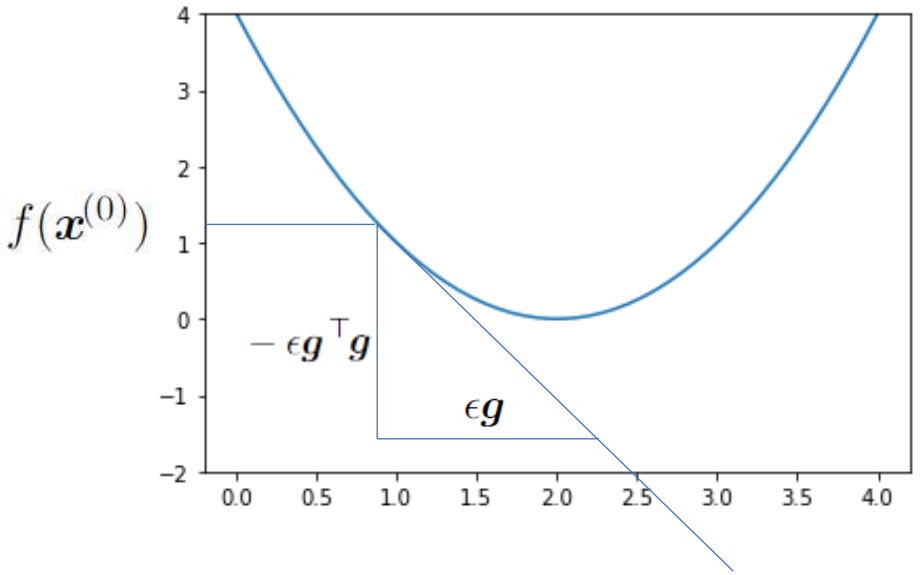
Taylor series approximation

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

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

$$f(\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \epsilon \boldsymbol{g}^\top \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}$$

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



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

2. Local Minima

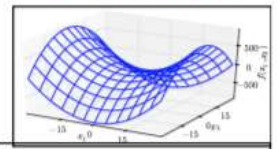
- In convex optimization, problem is one of finding a local minimum
- Some convex functions have a flat region rather than a global minimum point
- Any point within the flat region is acceptable
- With non-convexity of neural nets many local minima are possible
- Many deep models are guaranteed to have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

Model Identifiability

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

3. Plateaus, Saddle Points etc

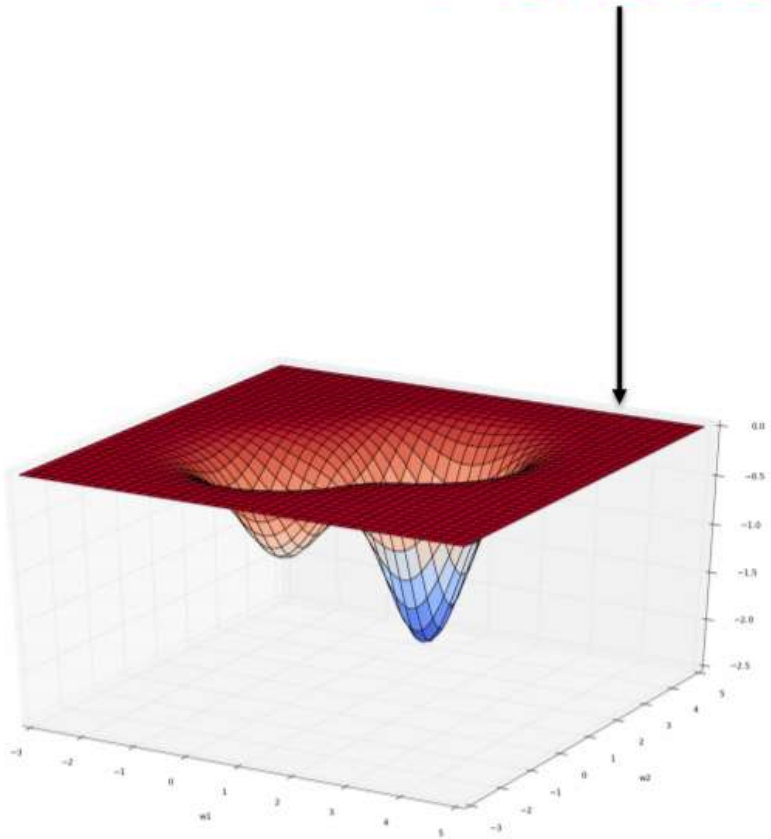
- More common than local minima/maxima are:
  - Another kind of zero gradient points: saddle points
    - At saddle, Hessian has both positive and negative values
      - Positive: cost greater than saddle point
      - Negative values have lower value
    - In low dimensions:
      - Local minima are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

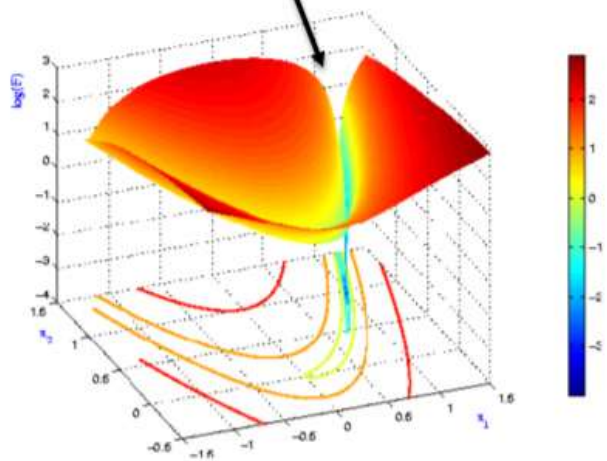


Contains both positive and negative curvature
Function is $f(x)=x_1^2-x_2^2$

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

Plateau and Ravine





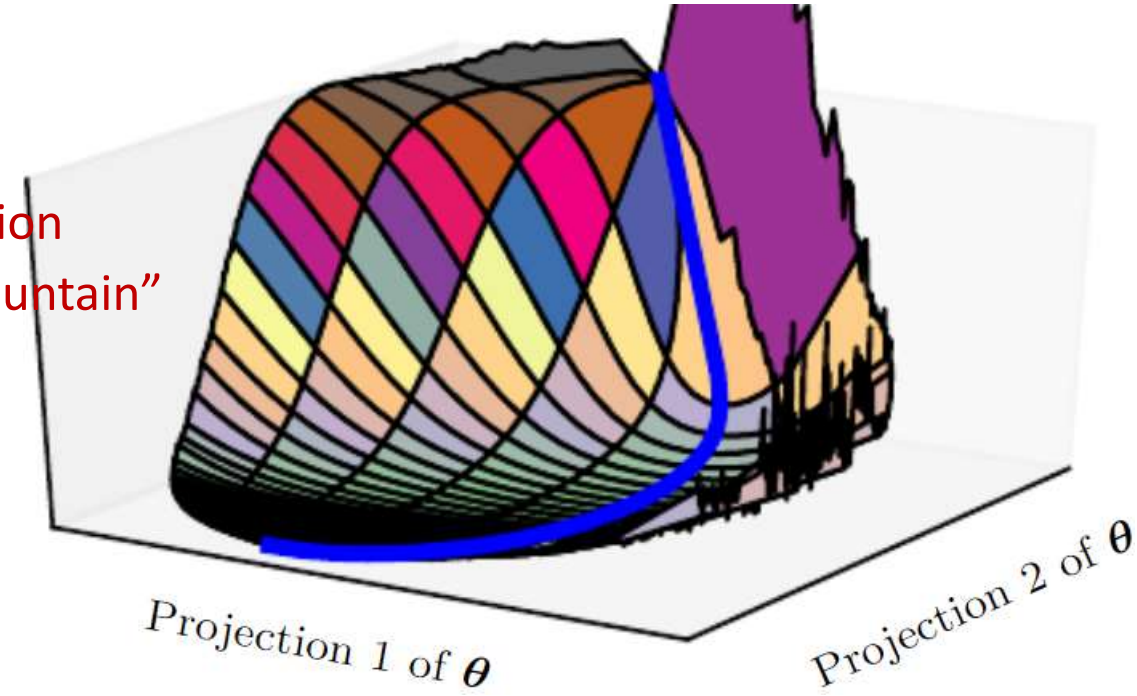
Visualization of the cost function of a NN

Goodfellow .et .al (2015)

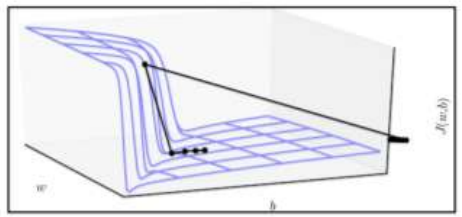
- Primary Obstacle:

  Saddle point of high cost near where the parameters are initialized.

- But, SGD training trajectory escapes this saddle point readily.
- Most time spent in traversing a flat valley.
  - Because of:
    - High noise in the gradient
    - Poor conditioning of the Hessian matrix in this region
    - Or, simply the need to circumnavigate the tall “mountain” visible in the figure via an indirect arcing path.



4.Cliffs and Exploding Gradients



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

5. Long-Term Dependencies

- When the computational graph becomes extremely deep.
  - Very deep feed forward network.
  - Recurrent neural network.

After t steps, $\boldsymbol{W}^t = \left(\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}\right)^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}$

$\lambda_1 = 1.01; \quad \lambda_1^{365} = 37.78$

$\lambda_2 = 0.99; \quad \lambda_2^{365} = 0.02$

vanishing and exploding gradient problem ➡ **Gradients also scaled according to** $\text{diag}(\boldsymbol{\lambda})^t$.

Recurrent networks use the same matrix $\boldsymbol{W}$ at each time step, but feedforward networks do not, so even very deep feedforward networks can largely avoid the vanishing and exploding gradient problem (Sussillo, 2014).

6. Inexact Gradients

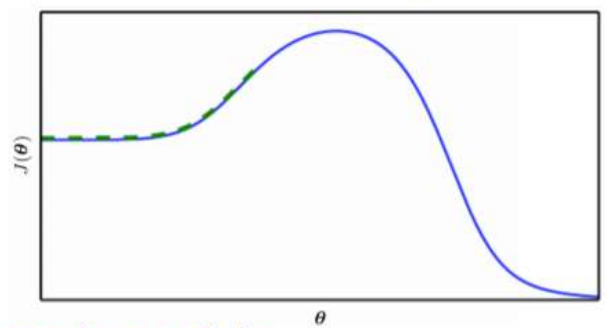
- Optimization algorithms assume we have access to exact gradient or Hessian matrix
- In practice we have a noisy or biased estimate
  - Every deep learning algorithm relies on sampling-based estimates
    - In using minibatch of training examples
  - In other case, objective function is intractable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

7. Poor Correspondence between Local and Global Structure

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

Need for good initial points

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



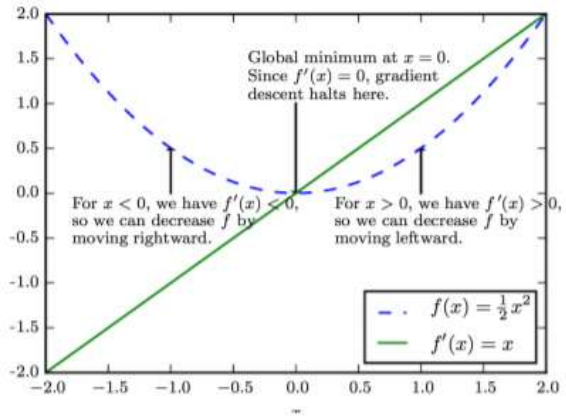
8. Theoretical Limits of Optimization

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

Basic Optimization Algorithms

1. Stochastic Gradient Descent

- Gradient descent follows gradient of entire training set downhill



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

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

SGD follows gradient estimate downhill

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$

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

$k \leftarrow 1$

while stopping criterion not met **do**

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

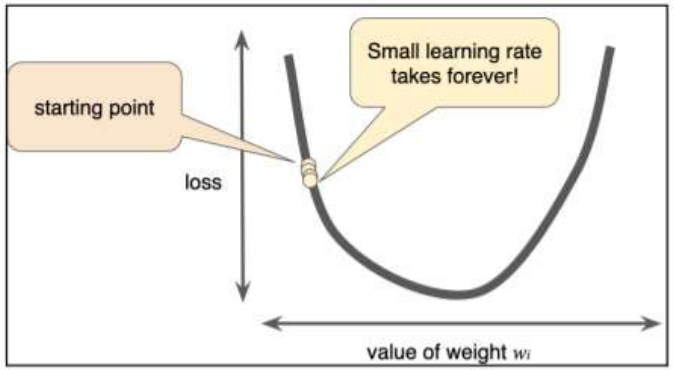
Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

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

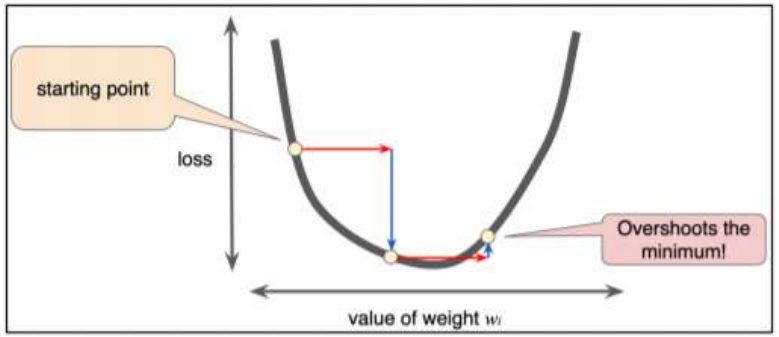
$k \leftarrow k + 1$

end while

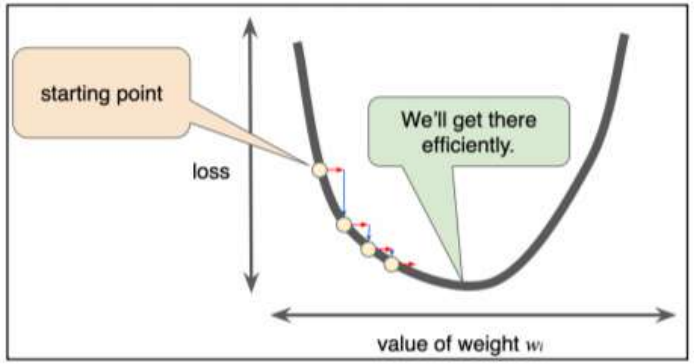
Choice of learning rate



Too small learning rate will take too long



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



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

Need for decreasing learning rate

- True gradient of total cost function
  - Becomes small and then 0
    - We can use a fixed learning rate
- But SGD has a source of noise
  - Random sampling of m training samples
    - Gradient does not vanish even when arrive at a minimum
  - Sufficient condition for SGD convergence

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

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

```
@author: arun
"""
t = 100
e_0 = 1 # crucial
e_t = 0.01
for k in range(200):
    if(k<t):
        a = k/t
        e_k = (1-a)*e_0+a*e_t
    else:
        e_k = 0.01
    print(e_k)
```

Convergence

$$\text{excess error } J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

- Generalization error cannot decrease faster than $O(\frac{1}{k})$

Cramér-Rao bound (Cramér, 1946; Rao,1945)

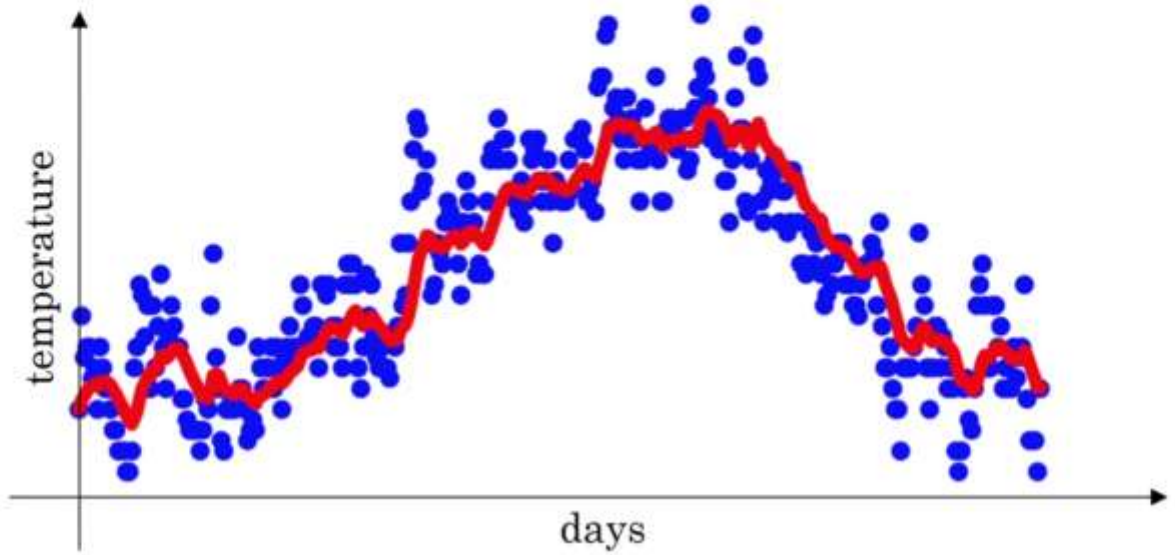
- SGD: Excess error is $O\left(\frac{1}{\sqrt{k}}\right)$ (For convex), after k iterations

- SGD: Excess error is $O(\frac{1}{k})$ (For strongly convex), after k iterations

The Momentum Method

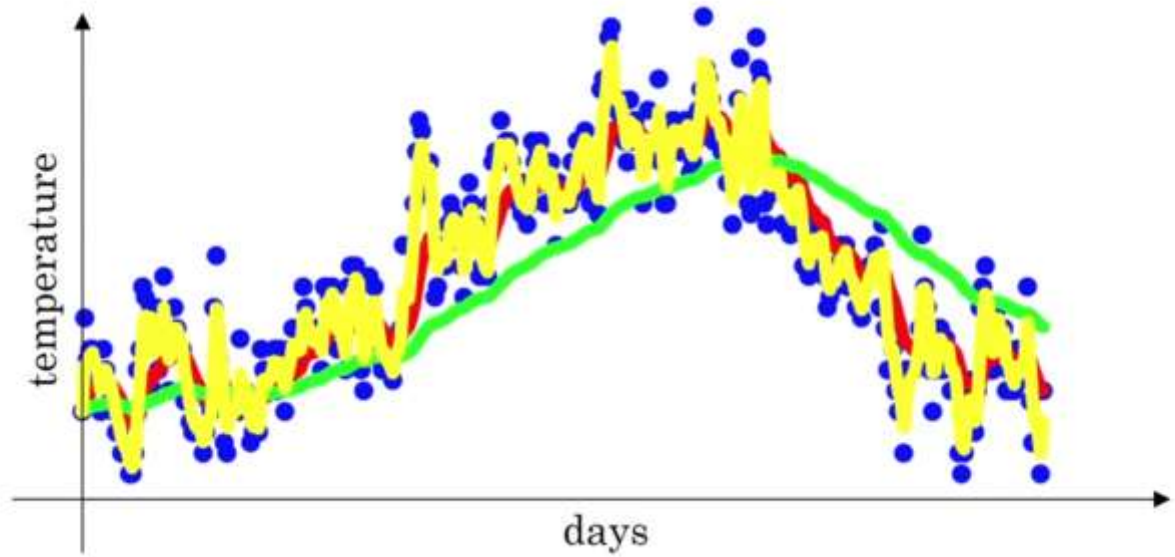
Exponentially weighted averages

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



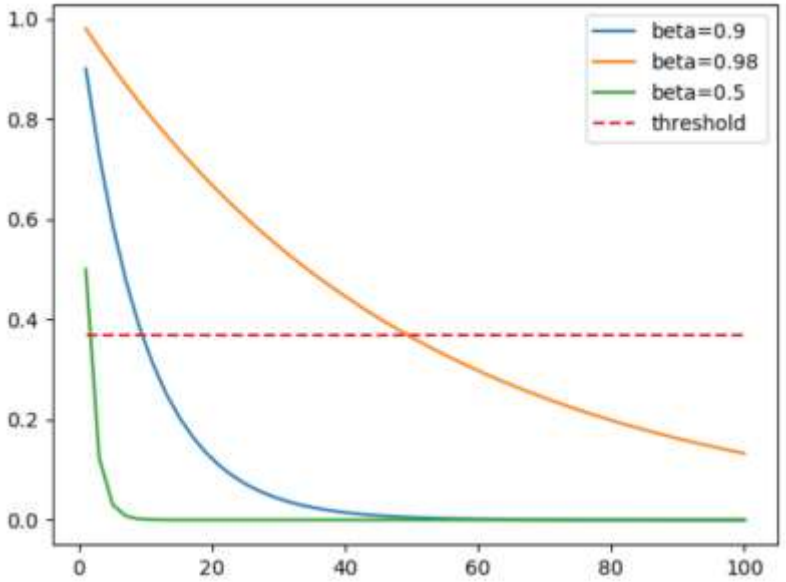
Exponentially weighted averages

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$ β = 0.9, 0.98, 0.5



Exponentially weighted averages

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



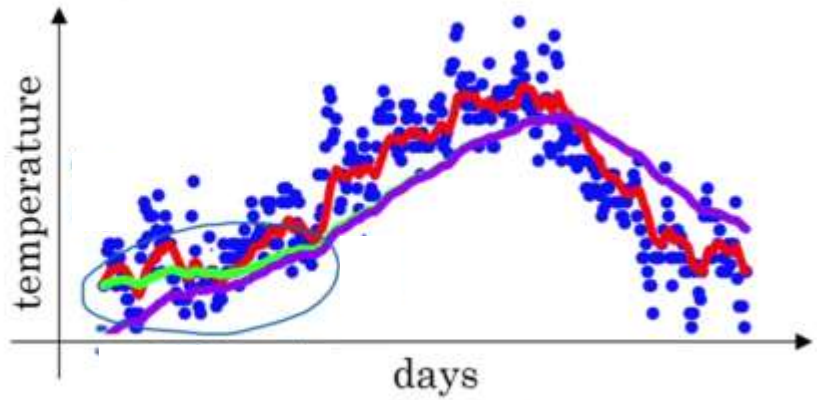
$v_{100} = 0.9\, v_{99} + 0.1\theta_{100}$
$v_{99} = 0.9\, v_{98} + 0.1\, \theta_{99}$
$v_{98} = 0.9\, v_{97} + 0.1\, \theta_{98}$
....

$v_{100} = 0.10_{100} + 0.9\, v_{99}(0.1\, \theta_{99} + 0.9\, v_{98}(0.1\, \theta_{98} + 0.9\, v_{97}(\ldots)))$

$v_{100} = 0.10_{100} + 0.9 * 0.1\, \theta_{99} + 0.9^2 * 0.1\, \theta_{98} + 0.9^3 * 0.1\theta_{97} + 0.9^4 * 0.1\theta_{96} \ldots)$

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$ $\quad (1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e}$

Bias correction



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

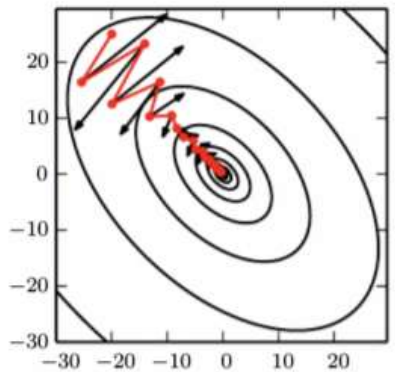
$$\frac{v_t}{1-\beta^t}$$

2. The Momentum method

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

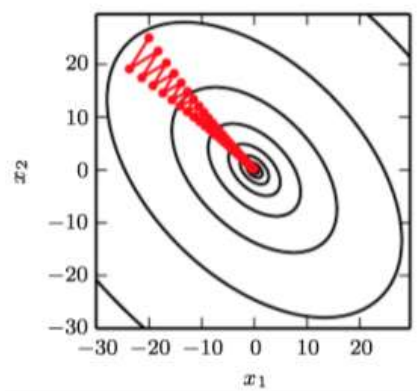
Momentum

- SGD with momentum



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

- Comparison to SGD without momentum



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

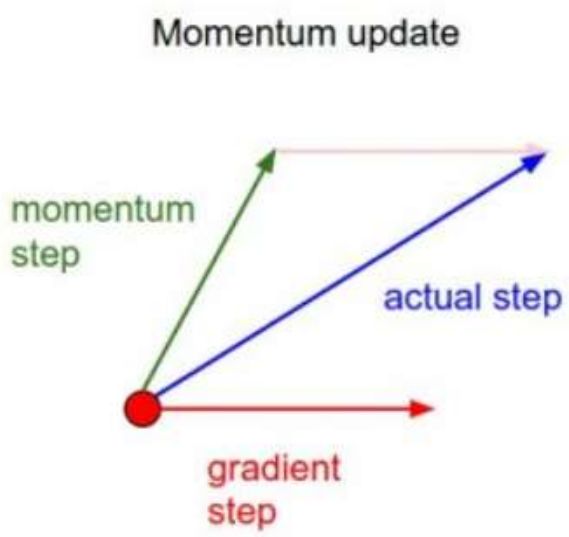
Stochastic gradient descent with momentum

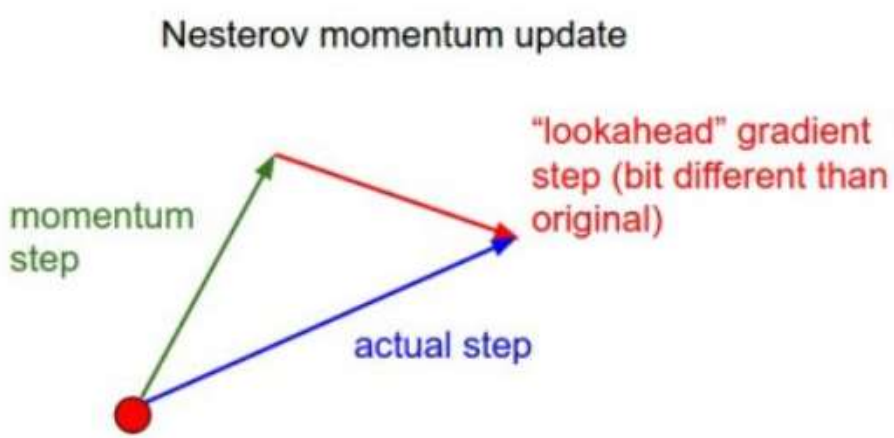
Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

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

https://cs231n.github.io/neural-networks-3/

SGD with Nesterov Momentum





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

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

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

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

(SGD) with Nesterov momentum

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α

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

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.

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

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

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

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

end while

(SGD) with Nesterov momentum

In the convex batch gradient case, Nesterov momentum brings the rate of convergence of the excess error from $O(1/k)$ (after k steps) to $O(1/k^2)$ as shown by Nesterov (1983). Unfortunately, in the stochastic gradient case, Nesterov momentum does not improve the rate of convergence.

Initialization of Parameters

Role of Initialization

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

Known property: Break Symmetry

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

Weights drawn from Gaussian

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

https://www.deeplearning.ai/ai-notes/initialization/

https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79

Heuristics for initial scale of weights

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

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

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

Relu: $\frac{\sqrt{2}}{\sqrt{n}}$

Understanding the difficulty of training deep feedforward neural networks
Glorot et. al.

Initialization for the biases

- Bias settings must be coordinated with setting weights
- Setting biases to zero is compatible with most weight initialization schemes
- Situations for nonzero biases:
  - Choose bias to causing too much saturation at initialization

Algorithms with Adaptive Learning Rates

Learning Rate is Crucial

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

Heuristic Approaches

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

https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827

The AdaGrad algorithm: Intution

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\varepsilon I + diag(G_t)}} \cdot g_t$$

where; $$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^\top$$

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t)$$

We can expand the equation as follows:

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_t^{(1)} \\ \theta_t^{(2)} \\ \vdots \\ \theta_t^{(m)} \end{bmatrix} - \eta \left(\begin{bmatrix} \varepsilon & 0 & \cdots & 0 \\ 0 & \varepsilon & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varepsilon \end{bmatrix} + \begin{bmatrix} G_t^{(1,1)} & 0 & \cdots & 0 \\ 0 & G_t^{(2,2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & G_t^{(m,m)} \end{bmatrix} \right)^{-1/2} \cdot \begin{bmatrix} g_t^{(1)} \\ g_t^{(2)} \\ \vdots \\ g_t^{(m)}, \end{bmatrix}$$

https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827

The AdaGrad algorithm

$$\eta \left(\begin{bmatrix} \varepsilon & 0 & \cdots & 0 \\ 0 & \varepsilon & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varepsilon \end{bmatrix} + \begin{bmatrix} G_t^{(1,1)} & 0 & \cdots & 0 \\ 0 & G_t^{(2,2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & G_t^{(m,m)} \end{bmatrix} \right)^{-1/2}$$

$$= \eta \begin{bmatrix} \varepsilon + G_t^{(1,1)} & 0 & \cdots & 0 \\ 0 & \varepsilon + G_t^{(2,2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varepsilon + G_t^{(m,m)} \end{bmatrix}^{-1/2}$$

$$= \eta \begin{bmatrix} \frac{1}{\sqrt{\varepsilon + G_t^{(1,1)}}} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sqrt{\varepsilon + G_t^{(2,2)}}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sqrt{\varepsilon + G_t^{(m,m)}}} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\eta}{\sqrt{\varepsilon + G_t^{(1,1)}}} & 0 & \cdots & 0 \\ 0 & \frac{\eta}{\sqrt{\varepsilon + G_t^{(2,2)}}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\eta}{\sqrt{\varepsilon + G_t^{(m,m)}}} \end{bmatrix}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\varepsilon I + diag(G_t)}} \cdot g_t$$

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_t^{(1)} \\ \theta_t^{(2)} \\ \vdots \\ \theta_t^{(m)} \end{bmatrix} - \begin{bmatrix} \frac{\eta}{\sqrt{\varepsilon + G_t^{(1,1)}}} & 0 & \cdots & 0 \\ 0 & \frac{\eta}{\sqrt{\varepsilon + G_t^{(2,2)}}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\eta}{\sqrt{\varepsilon + G_t^{(m,m)}}} \end{bmatrix} \cdot \begin{bmatrix} g_t^{(1)} \\ g_t^{(2)} \\ \vdots \\ g_t^{(m)} \end{bmatrix}$$

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_t^{(1)} \\ \theta_t^{(2)} \\ \vdots \\ \theta_t^{(m)} \end{bmatrix} - \begin{bmatrix} \frac{\eta}{\sqrt{\varepsilon + G_t^{(1,1)}}} g_t^{(1)} \\ \frac{\eta}{\sqrt{\varepsilon + G_t^{(2,2)}}} g_t^{(2)} \\ \vdots \\ \frac{\eta}{\sqrt{\varepsilon + G_t^{(m,m)}}} g_t^{(m)} \end{bmatrix}.$$

The AdaGrad algorithm

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

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

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

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

while stopping criterion not met **do**

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

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

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

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

end while

Caveat: AdaGrad performs well for some but not all deep learning models.

The RMSProp algorithm

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ

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

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

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

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

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

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

end while

RMSProp algorithm with Nesterov momentum

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α

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

Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$

while stopping criterion not met **do**

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

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

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

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

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

end while

RMSProp is popular

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

Adam: Adaptive Moments

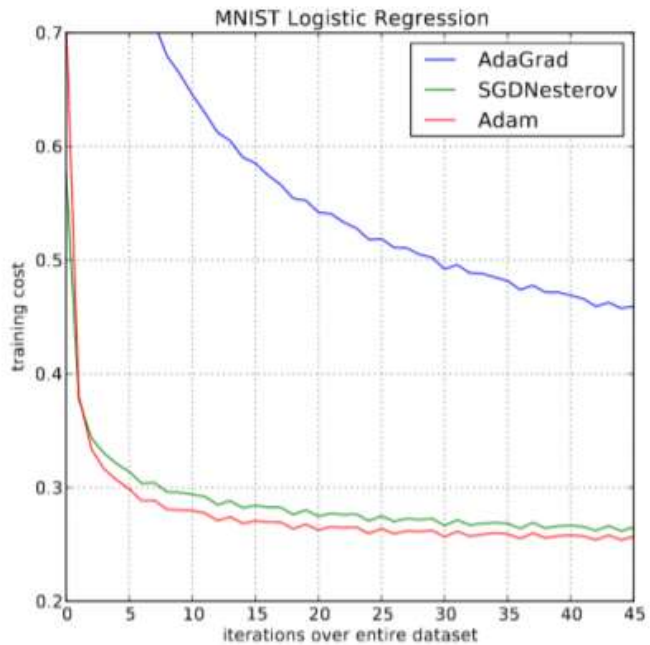
- Yet another adaptive learning rate optimization algorithm
- Variant of RMSProp with momentum
- Generally robust to the choice of hyperparameters

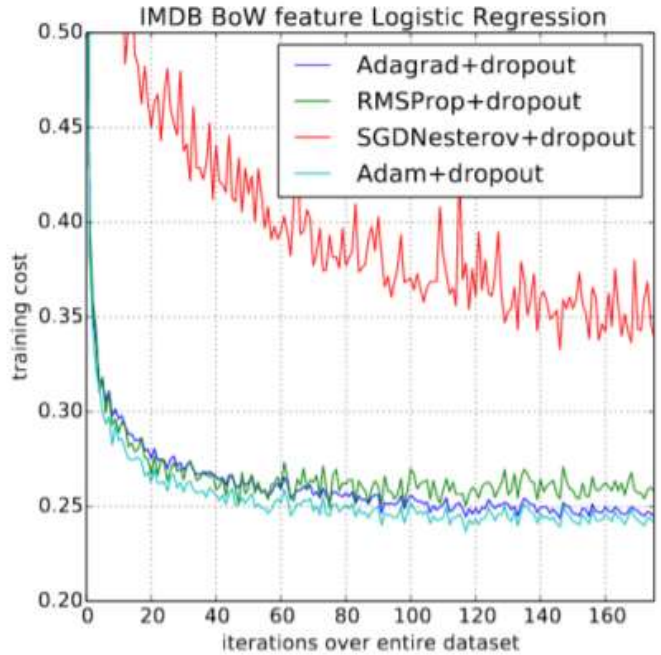
The Adam Algorithm

Algorithm 8.7 The Adam algorithm

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

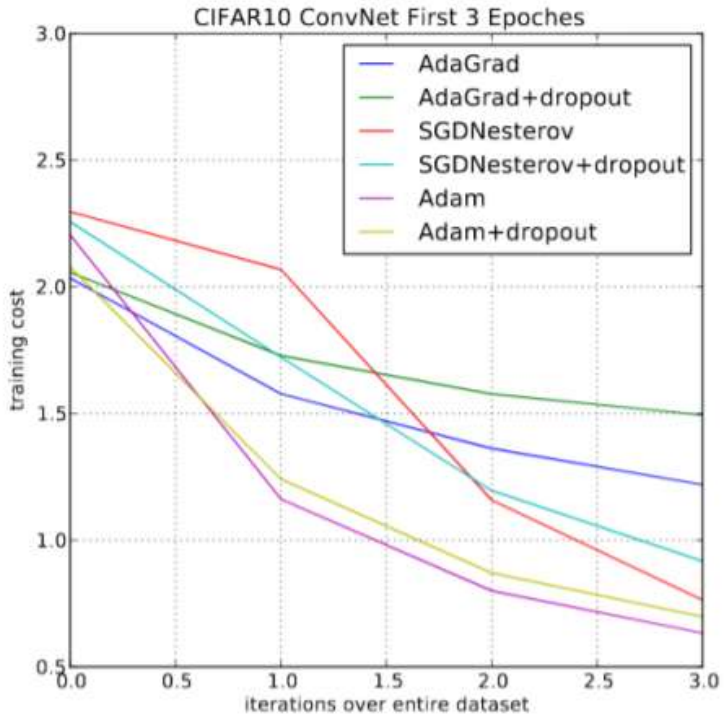
Performance on Multilayer NN

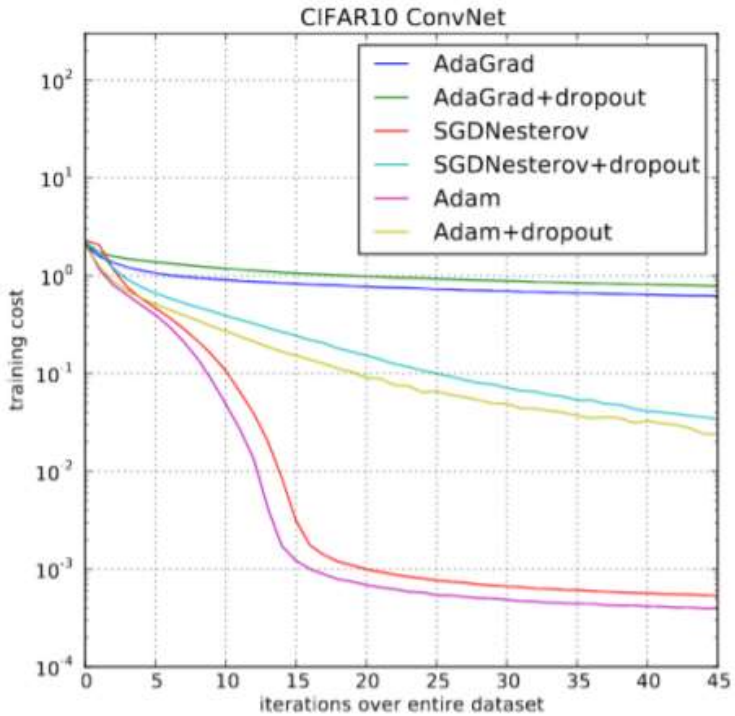




Training of multilayer neural networks on MNIST images. (a) Neural networks using dropout stochastic regularization. (b) Neural networks with deterministic cost function.

Performance with CNN





Choosing the Right Optimizer

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

Approximate Second Order Methods

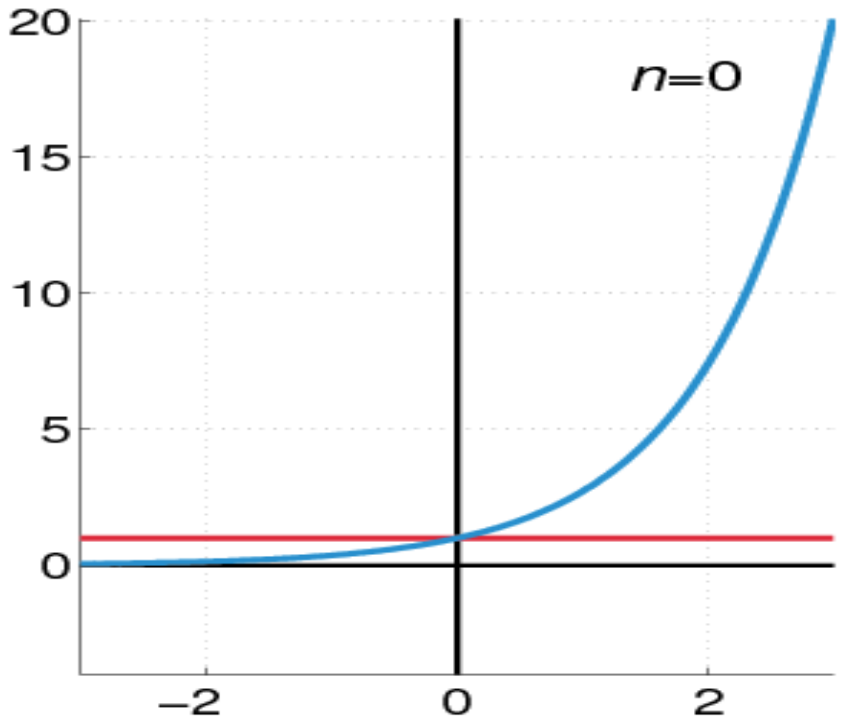
Newton's Method

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

Newton Update Rule

- Taylor expansion till second order

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

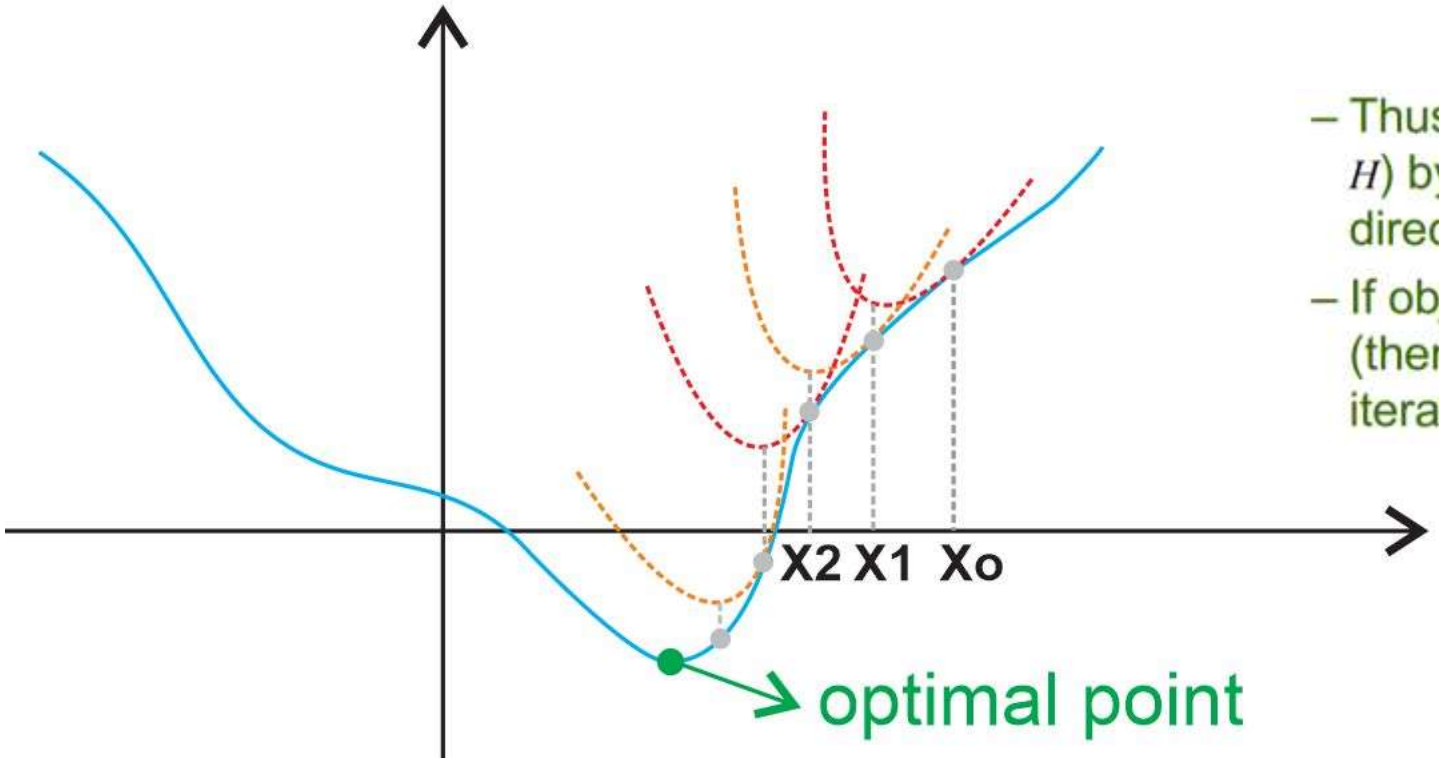


Newton Update Rule

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

- Solving for the critical point of this function at local approx.

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$



– Thus for a quadratic function (with positive definite H) by rescaling the gradient by H^{-1} Newton's method directly jumps to the minimum

– If objective function is convex but not quadratic (there are higher-order terms) this update can be iterated yielding the training algorithm given next

Newton Method: Alogorithm

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$

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

Require: Training set of m examples

while stopping criterion not met **do**

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

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

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

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

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

end while

Positive Definite Hessian

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

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

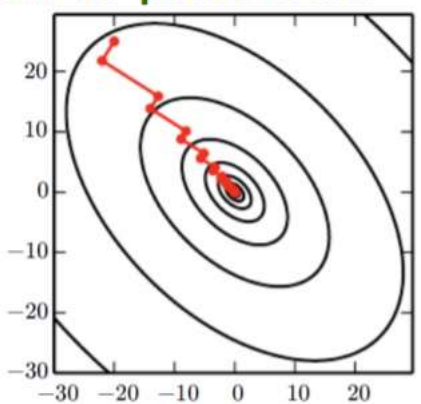
Regularizing the Hessian

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

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

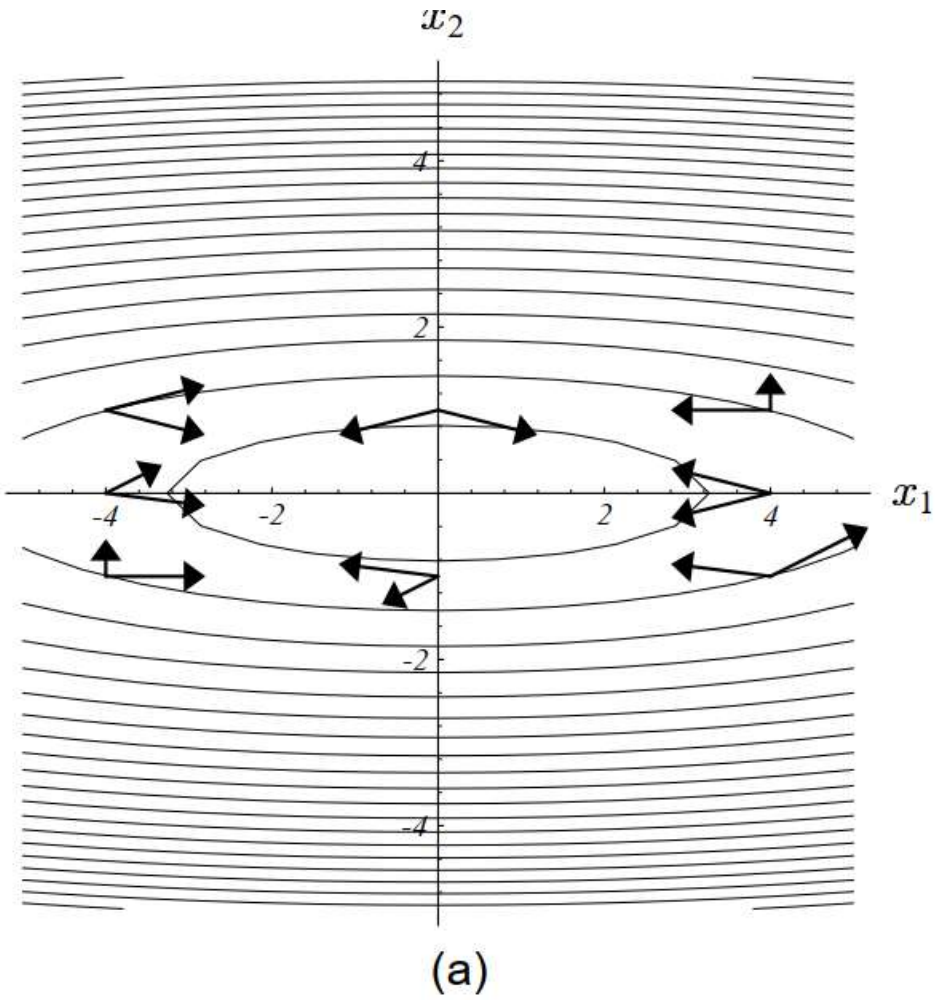
Motivating Conjugate Gradients

- Method to efficiently avoid calculating H^{-1}
  - By iteratively descending conjugate directions
- Arises from steepest descent for quadratic bowl has an ineffective zig-zag pattern
  - Since each line direction is orthogonal to previous
    - Let previous search direction be $\boldsymbol{d}_t$
    - Then $\nabla_\theta J(\theta)\boldsymbol{d}_{t-1} = 0$
    - Current search direction will have no contribution in direction $\boldsymbol{d}_{t-1}$
    - Thus $\boldsymbol{d}_t$ is orthogonal to $\boldsymbol{d}_{t-1}$
    - Method of conjugate gradients addresses this problem

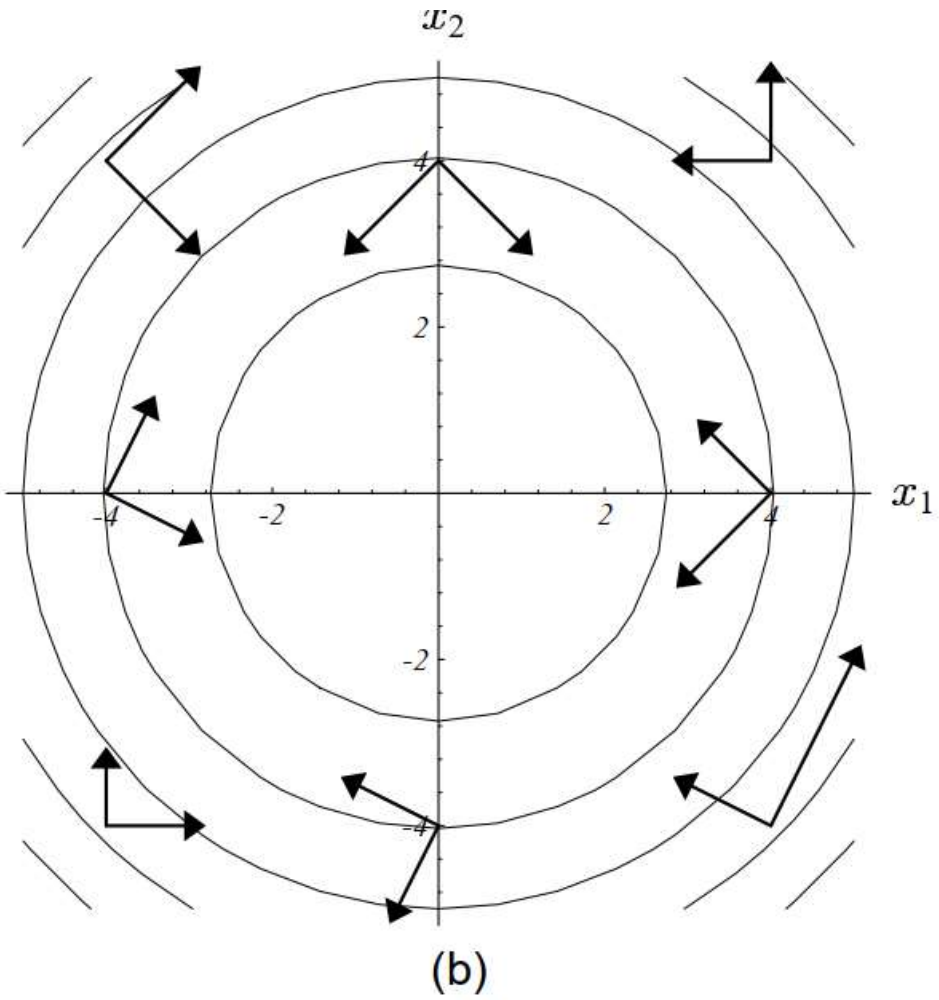


https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

Conjugate Vectors



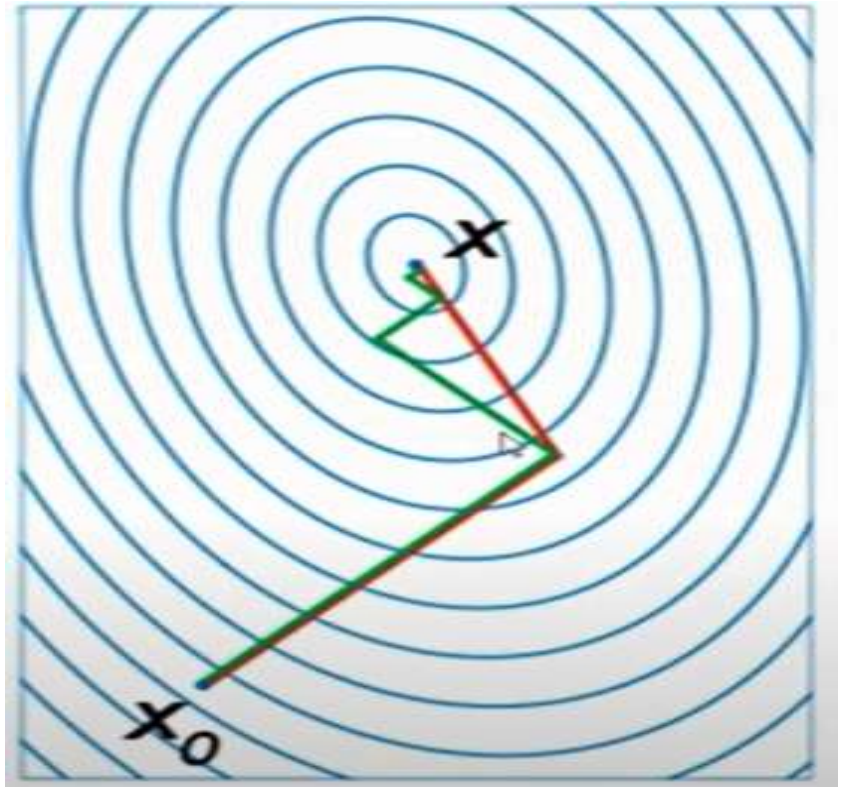
(a)



(b)

Conjugate Gradient Vs Steepest Decent

https://www.youtube.com/watch?v=eAYohMUpPMA&ab_channel=TomCarlone



Imposing Conjugate Directions

- We seek to find a search direction that is conjugate to the previous line search direction
- At iteration t the next search direction $\boldsymbol{d}_t$ takes the form $\boldsymbol{d}_t = \nabla_\theta J(\theta) + \beta_t \boldsymbol{d}_{t-1}$
- Directions $\boldsymbol{d}_t$ and $\boldsymbol{d}_{t-1}$ are conjugate if $\boldsymbol{d}_t H \boldsymbol{d}_{t-1} = 0$
- Methods for imposing conjugacy
  - Fletcher-Reeves $\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$
  - Polak-Ribiere $\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$

The conjugate gradient method

Algorithm 8.9 The conjugate gradient method

Require: Initial parameters $\boldsymbol{\theta}_0$
Require: Training set of m examples
  Initialize $\boldsymbol{\rho}_0 = \mathbf{0}$
  Initialize $g_0 = 0$
  Initialize $t = 1$
  while stopping criterion not met **do**
    Initialize the gradient $\boldsymbol{g}_t = \mathbf{0}$
    Compute gradient: $\boldsymbol{g}_t \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute $\beta_t = \frac{(\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^\top \boldsymbol{g}_t}{\boldsymbol{g}_{t-1}^\top \boldsymbol{g}_{t-1}}$ (Polak-Ribière)
    (Nonlinear conjugate gradient: optionally reset β_t to zero, for example if t is a multiple of some constant k, such as $k = 5$)
    Compute search direction: $\boldsymbol{\rho}_t = -\boldsymbol{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$
    Perform line search to find: $\epsilon^* = \text{argmin}_\epsilon \frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_t + \epsilon \boldsymbol{\rho}_t), \boldsymbol{y}^{(i)})$
    (On a truly quadratic cost function, analytically solve for ϵ^* rather than explicitly searching for it)
    Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t$
    $t \leftarrow t + 1$
  end while

The BFGS Algorithm

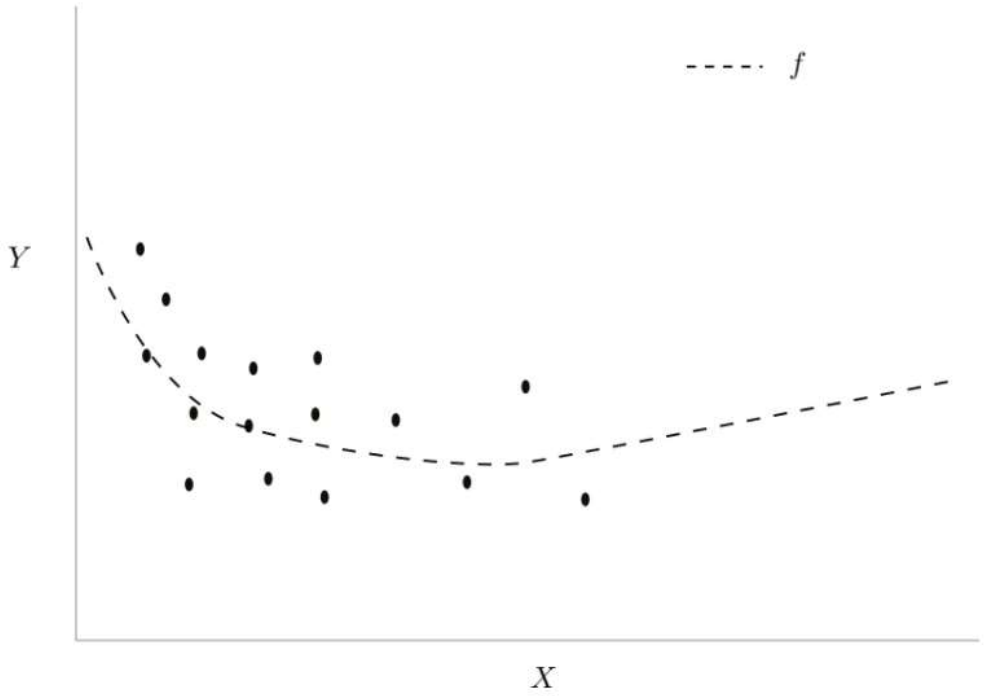
- Broyden-Fletcher-Goldfarb-Shanno (BFGS)
  - Newton's method without the computational burden
  - It is similar to the conjugate gradient method
    - More direct approach to approximating Newton's update
- Recall Newton's update: $\theta^* = \theta_0 - H^{-1}\nabla_\theta J(\theta_0)$
    - where H is the Hessian of J wrt θ evaluated at θ_0
  - Primary difficulty is computation of H^{-1}
  - BFGS is quasi Newton: approximates H^{-1} by matrix M_t that is iteratively refined by low-rank updates
    - Once the inverse Hessian M_t is updated, the direction of descent ρ_t is determined by $\rho_t = M_t g_t$
    - Final update to parameters is $\theta_{t+1} = \theta_t + \varepsilon^* \rho_t$

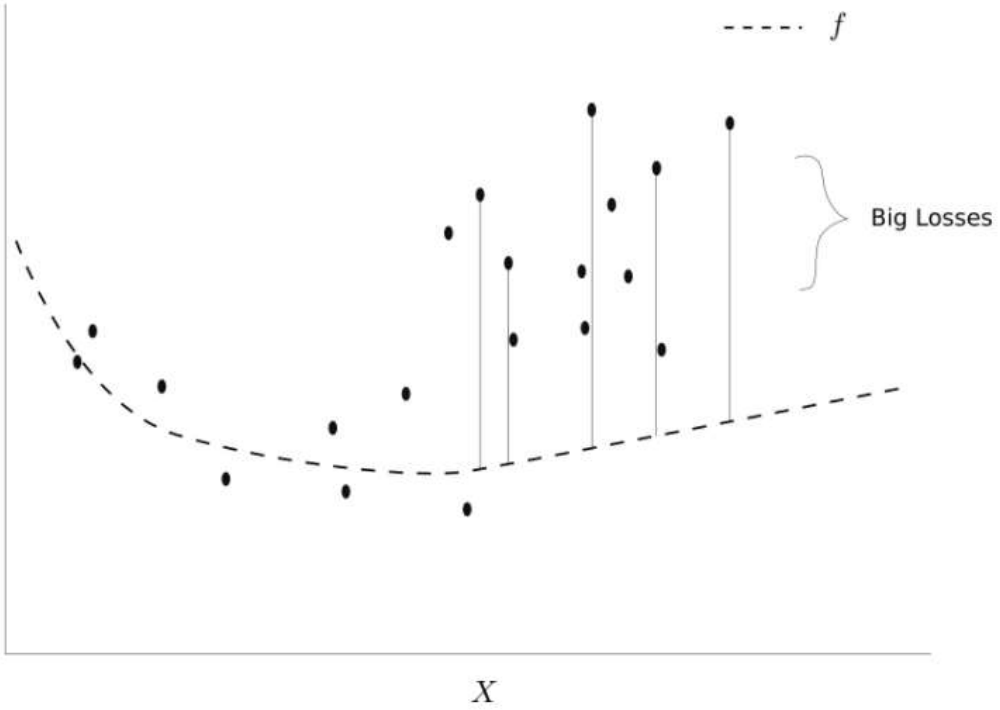
Optimization Strategies and Meta-Algorithms

Batch Normalization

- Batch normalization: exciting recent innovation
- Motivation is difficulty of choosing learning rate ε in deep networks
- Method is to replace activations with zero-mean with unit variance activations

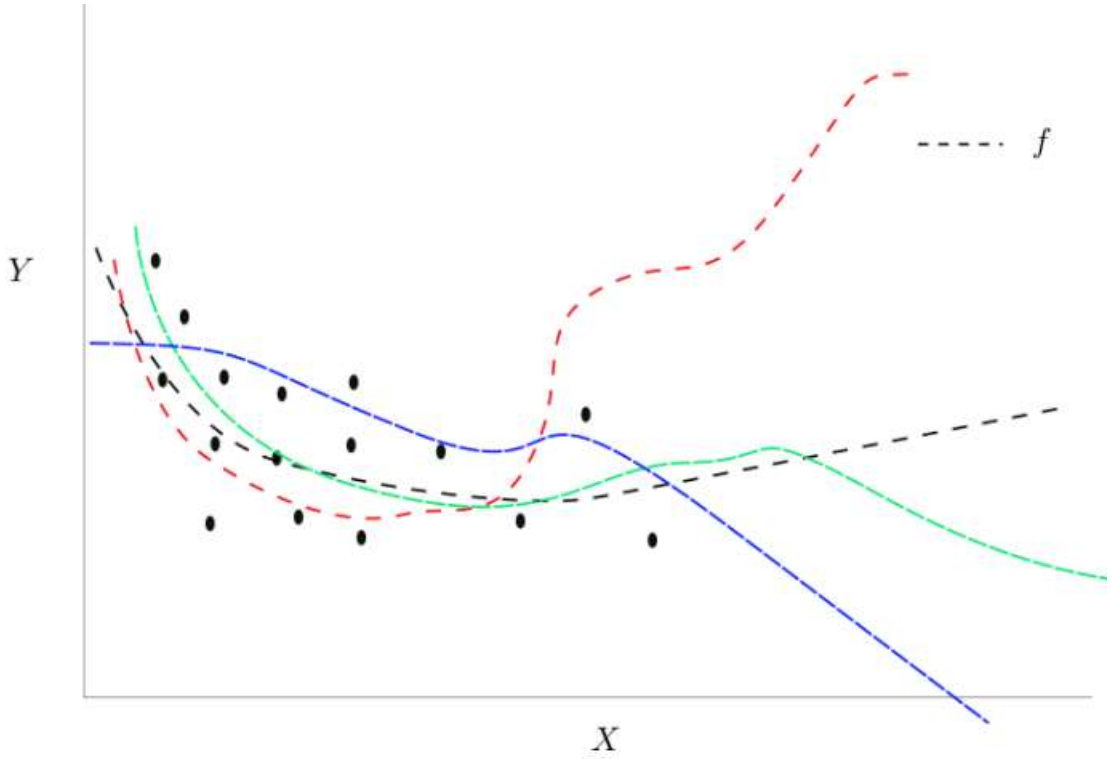
Covariate Shift

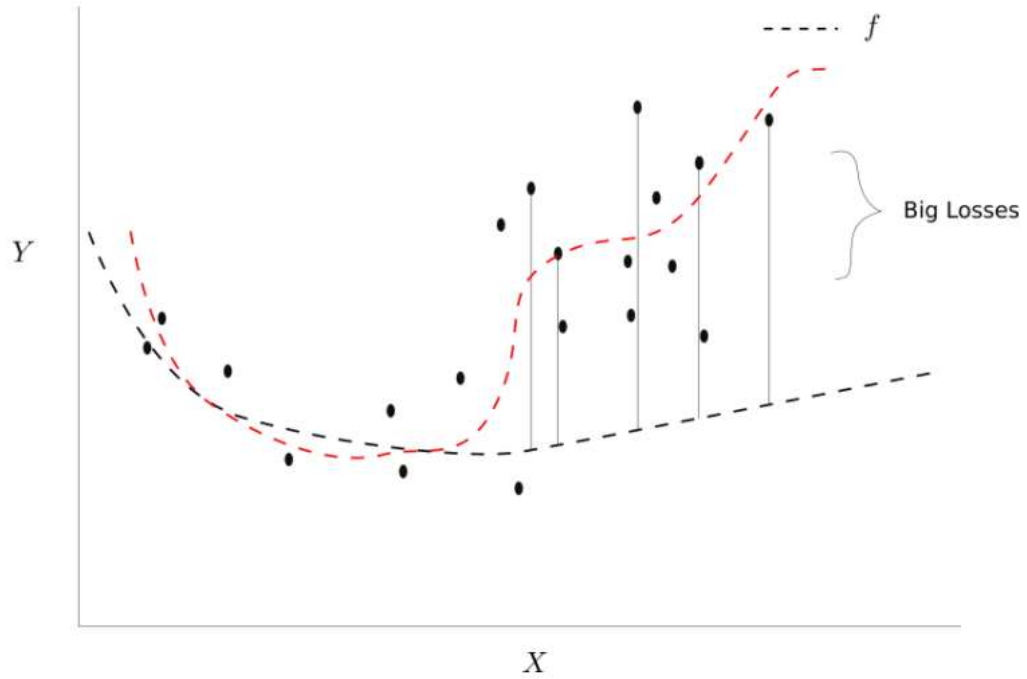




Covariate Shift

https://blog.paperspace.com/busting-the-myths-about-batch-normalization/



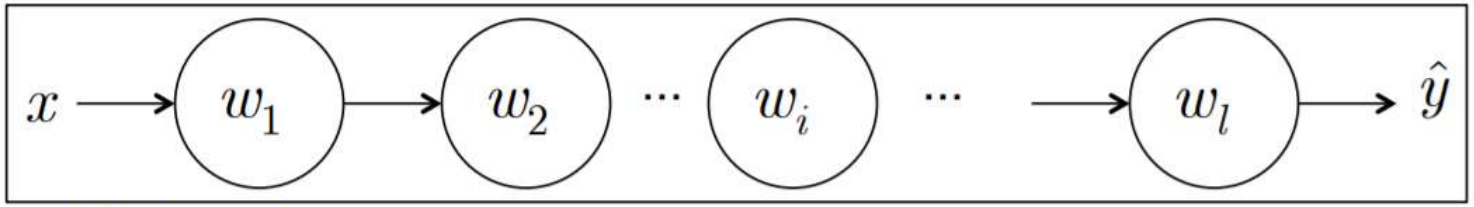


Motivation: Difficulty of composition

- Very deep models involve compositions of several functions or layers
  - Ex: $f(\boldsymbol{x},\boldsymbol{w}) = f^{(l)}[\ldots f^{(3)}[f^{(2)}[f^{(1)}(\boldsymbol{x})]]]$ has depth of l
- The gradient tells us how to update each parameter $\boldsymbol{w}^{\tau+1} = \boldsymbol{w}^{\tau} - \varepsilon \nabla_{\boldsymbol{w}} E_{\boldsymbol{x},y \in \hat{p}_{data}} L\left(f(\boldsymbol{x};\boldsymbol{w}), y\right)$
  - Under assumption that other layers do not change
  - We update all l layers simultaneously
  - When we make the change unexpected results can happen
    - Because many functions are changed simultaneously
    - Under the assumption that other functions remain constant

Choosing learning rate ϵ in multilayer

- Simple example:
  - l layers, one unit per layer, no activation function

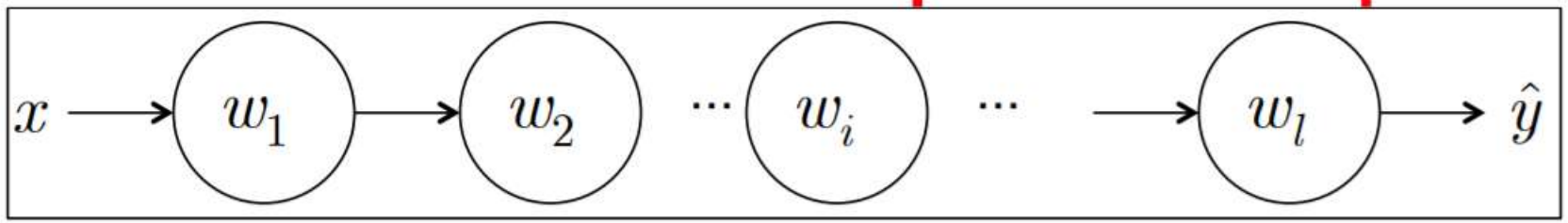


  - Network computes

$$\hat{y} = x \cdot w_1 \cdot w_2 \cdot w_3 \ldots \cdot w_i \ldots \cdot w_l$$

  - where w_i provides weight of layer i
  - Output of layer i is $h_i = h_{i-1} w_i$
    - Output is a linear function of input x but a nonlinear function of the weights w_i

Gradient in Simple example



- Suppose cost has put a gradient of 1 on $\hat{y}$, so we wish to decrease $\hat{y}$ slightly
- Backpropagation can compute a gradient $\boldsymbol{g} = \nabla_{\boldsymbol{w}} \hat{y}$
- Consider update $\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \boldsymbol{g}$
  - First order Taylor's series approx of $\hat{y}$ predicts that the value of $\hat{y}$ will decrease by $\varepsilon \boldsymbol{g}^{\mathrm{T}} \boldsymbol{g}$, because:

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

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

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

Difficulty of Multilayer learning Rate

- To decrease $\hat{y}$ by 0.1 we could set ϵ to $\frac{0.1}{g^T g}$

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

- However, update includes 2^{nd},3^{rd} .. order effects
  - Since $w \leftarrow w - \epsilon g$ new value of $\hat{y}$ is $x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2)\dots(w_l - \epsilon g_l)$
    - A second order term arising from this update is $\varepsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i$
    - This term can be negligible if $\prod_{i=3}^{l} w_i$ is small or exponentially large if weights on layers 3 to l are greater than 1
- This makes it very hard to choose ϵ because the effects of an update for one layer depend so strongly on all other layers
  - Second-order optimization algorithms address this issue but it seems hopeless for $l > 2$

The Batch Normalization Solution

- Provides an elegant way of reparameterizing almost any network
- Significantly reduces the problem of coordinating updates across many layers
- Can be applied to any input or hidden layer in a network

Batch Normalization

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{W}$$

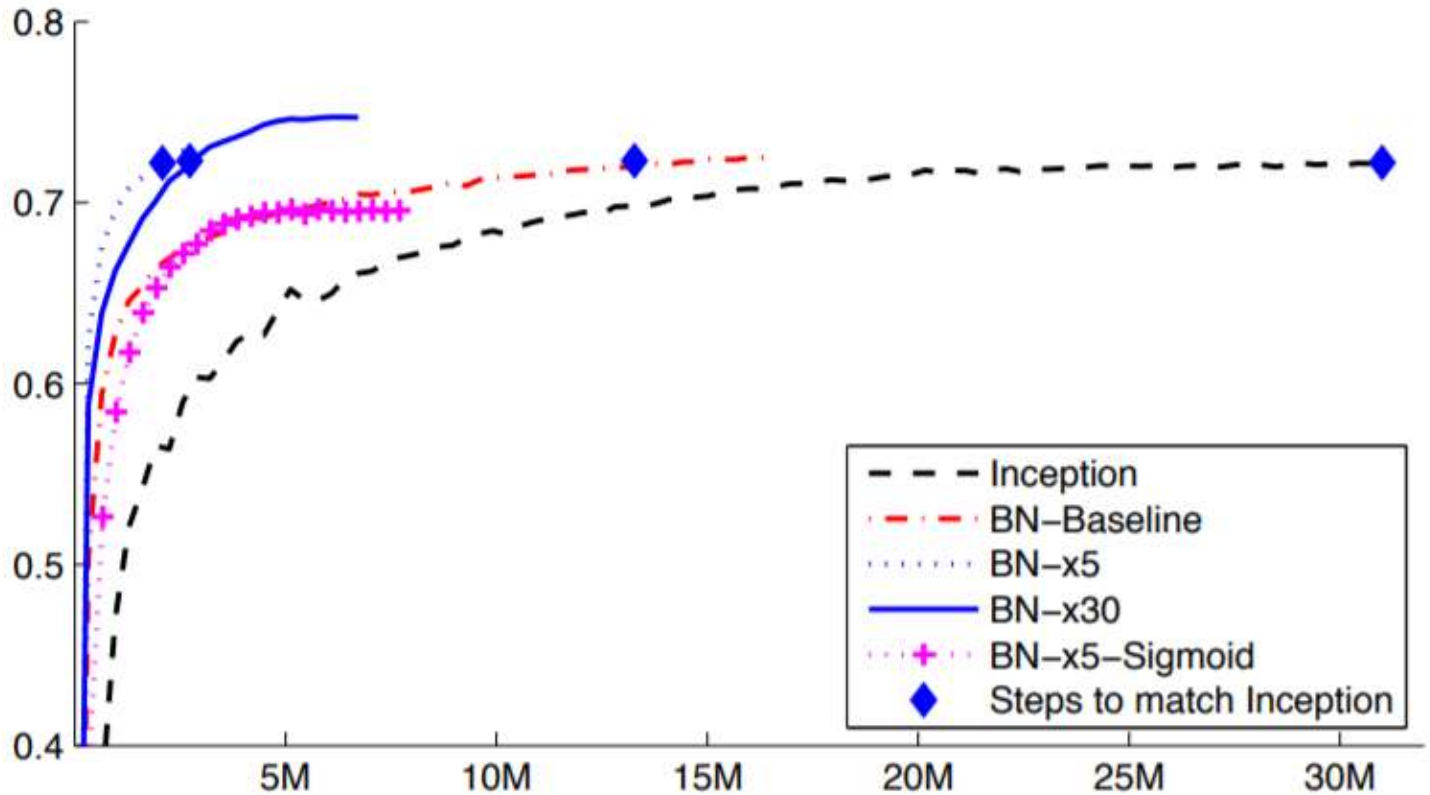
$$\tilde{\boldsymbol{Z}} = \boldsymbol{Z} - \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{Z}_{i,:}$$

$$\hat{\boldsymbol{Z}} = \frac{\tilde{\boldsymbol{Z}}}{\sqrt{\epsilon + \frac{1}{m}\sum_{i=1}^{m} \tilde{\boldsymbol{Z}}_{i,:}^2}}$$

$$\boldsymbol{H} = \max\{0, \boldsymbol{\gamma}\hat{Z} + \boldsymbol{\beta}\}$$

"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Ioffe and Szegedy 2015

Batch Normalization



"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Ioffe and Szegedy 2015

References

- Chapter 8, Deep Learning, Ian et.el.