Optimization for Training Deep Models

Lecture slides for Chapter 8 of *Deep Learning*

www.deeplearningbook.org

Ian Goodfellow

Adapted by m.n. for CMPS 392

Goal

- The network training problem is important and expensive
  - It is quite common to invest days to months of time on hundreds of machines in order to solve even a single instance of this problem
- Finding the parameters θ of a neural network that significantly reduce a cost function $J(\theta)$,
  - Which typically includes a performance measure evaluated on the entire training set
  - As well as additional regularization terms.
- Optimization algorithms:
  - adapt their learning rates during training
  - or leverage information contained in the second derivatives of the cost function.

Optimization vs. Machine learning

Optimization	Machine Learning
0-1 Loss (accuracy)	Surrogate loss function (negative log likelihood)
p_{data} known	Reduce generalization error p_{data} unknown $\hat{p}_{data}$ known
Direct	Indirect
Halt at a local minimum (gradient very small)	Early stopping (convergence criterion is met) (gradient can still be large)

Why Minibatch algorithms?

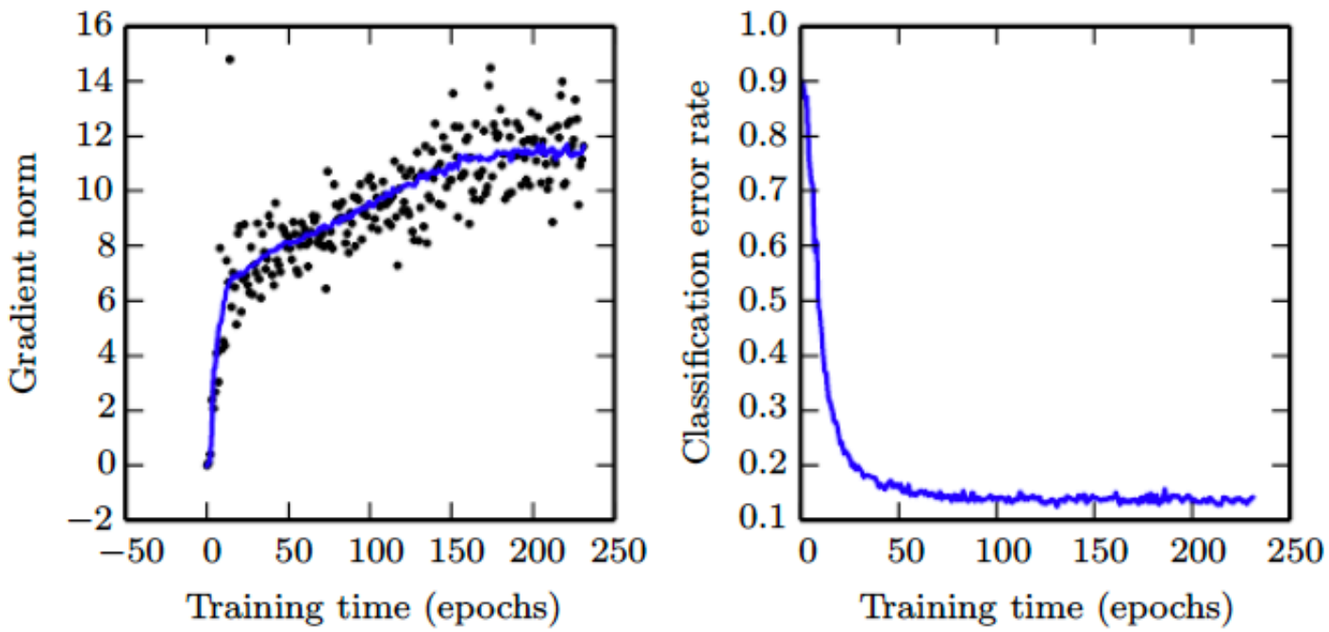
1. The objective function decomposes as a sum over the training examples
2. Estimating the mean has a standard error of $\frac{\sigma}{\sqrt{n}}$
   - Training with a batch 100 times bigger reduces the error by a factor of only 10
3. Redundancy in the training set

What is a good batch size?

- Larger batches provide a more accurate estimation fo the gradient, but with **less than linear** returns
- Examples of the batch are typically processed in parallel using multi-core architectures which get under-utilized by small size batches.
- For many hardware setups, the limiting factor is most often the memory to store the batch
- If using GPUs, it is common for power of 2 batch sizes to offer better runtime (ranging from 32 to 256, 16 for large models)
- Small batches can offer a regularization effect
  - At the cost of a small learning rate to maintain stability
  - And a large total training runtime
- It is also very important to shuffle the data

Challenges in Neural Network optimization

- Ill-conditioning:
  - a step of $-\epsilon g$ adds $\frac{1}{2}\epsilon^2 g^T H g - \epsilon g^T g$ to the cost
  - Ill conditioning is a problem when $\frac{1}{2}\epsilon^2 g^T H g > \epsilon g^T g$
  - To detect the problem, monitor both $g^T g$ and $g^T H g$

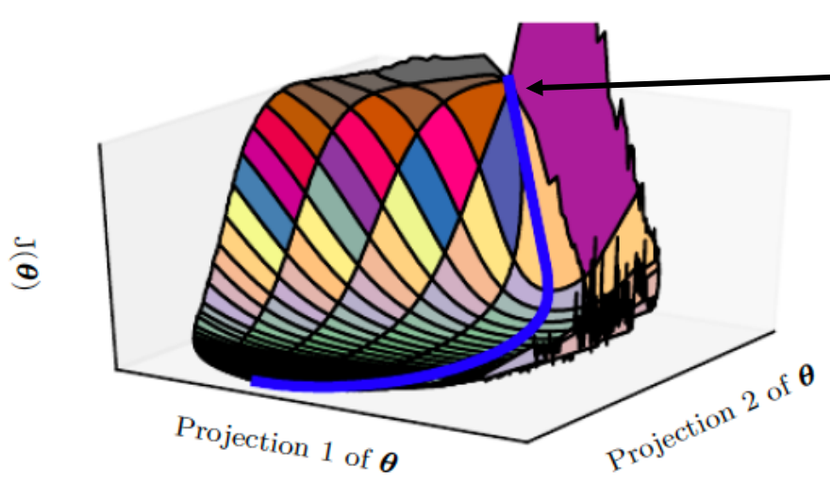


Local minima in neural networks

- Local minima with equivalent loss function value:
  - Weight space symmetry:
    - take a neural network and modify layer 1 by swapping the incoming weight vector for unit i with the incoming weight vector for unit j, then doing the same for the outgoing weight vectors
    - There are $n!^m$ ways of arranging the hidden units (depth m, width n)
  - scale all of the incoming weights and biases of a unit by α and scale all of its outgoing weights by $1/\alpha$.
- Today, experts suspect that, for sufficiently large neural networks, most local minima have a low cost function value
  - A test that can rule out local minima as the problem is to plot the norm of the gradient over time.
    - the gradient does not shrink to insignificant size → the problem is not getting stuck at a critical point!

Saddle points

- In high dimensional spaces, local minima are rare and saddle points are more common.
  - Imagine that the sign of each eigenvalue (of H) is generated by flipping a coin.
  - Hessian becomes more likely to be positive as we reach regions of lower cost
  - gradient descent empirically seems to be able to escape saddle points
    - For Newton's method, it is clear that saddle points constitute a problem
  - Flat regions of high cost remains a challenge
    - gradient and hessian are 0



Cliffs and Exploding Gradients

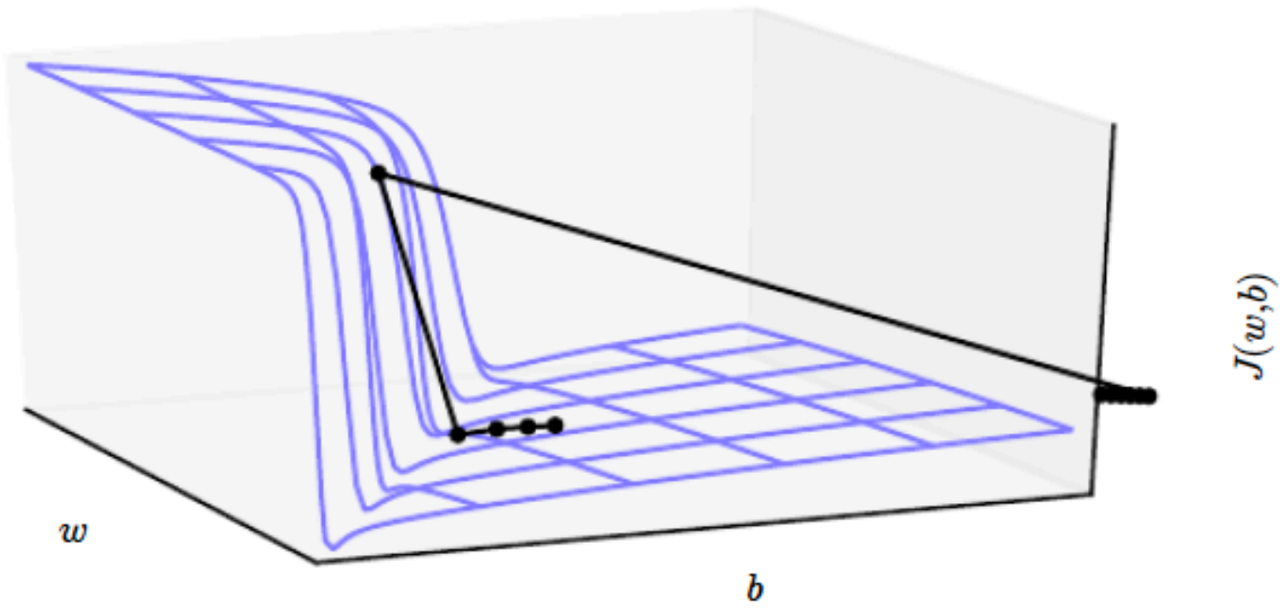


Figure 8.3: The objective function for highly nonlinear deep neural networks or for recurrent neural networks often contains sharp nonlinearities in parameter space resulting from the multiplication of several parameters. These nonlinearities give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly losing most of the optimization work that had been done. Figure adapted with permission from Pascanu *et al.* (2013).

Vanishing and exploding gradient

- Suppose that a computational graph contains a path that consists of repeatedly multiplying by a matrix W.
  - After t steps, this is equivalent to multiplying by W^t.
- Suppose that W has an eigen decomposition

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

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

  - eigenvalue $\lambda_i > 1$will explode
  - eigenvalue $\lambda_i < 1$will vanish

- Recurrent neural networks use the same matrix W at each time step

Poor Correspondence between Local and Global Structure

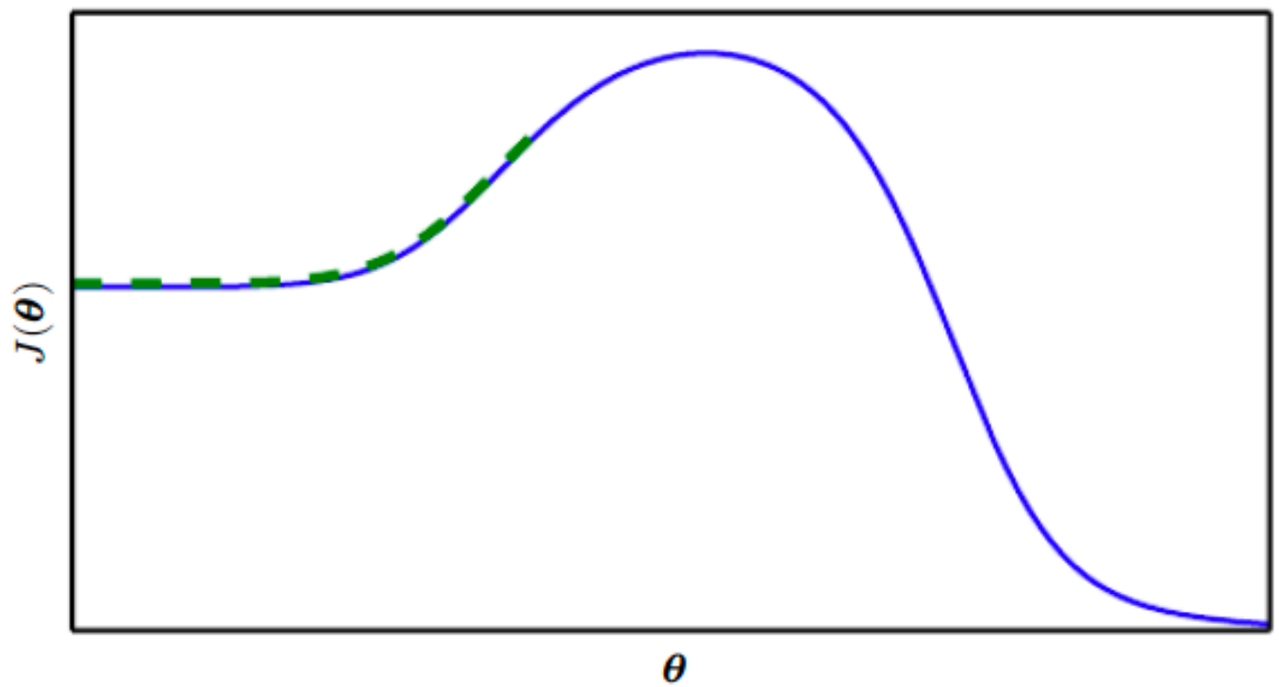


Figure 8.4: Optimization based on local downhill moves can fail if the local surface does not point toward the global solution. Here we provide an example of how this can occur, even if there are no saddle points and no local minima. This example cost function contains only asymptotes toward low values, not minima. The main cause of difficulty in this case is being initialized on the wrong side of the "mountain" and not being able to traverse it. In higher dimensional space, learning algorithms can often circumnavigate such mountains but the trajectory associated with doing so may be long and result in excessive training time, as illustrated in figure 8.2.

What is a good learning rate?

- It is common to decay the learning rate linearly until iteration τ
  - $\epsilon_k = \left(1 - \frac{k}{\tau}\right)\epsilon_0 + \frac{k}{\tau}\epsilon_\tau$
  - After iteration τ, it is common to leave ϵ constant
  - Parameters to choose are $\epsilon_0,\ \epsilon_\tau,\ \tau$:
    - τ is usually set as set to the number of iterations required to make a few hundred epochs
    - ϵ_τ should be set to roughly 1% the value of ϵ_0
    - To set ϵ_0, monitor the oscillations in the learning curve:
      - High learning rate may cause instability
      - Gentle oscillations are ok especially if dropout is used
      - Low learning rate: learning proceeds slowly, and may become stuck
  - Monitor the first iterations and choose an initial rate higher than the best performing rate

Momentum

- From physics: Momentum = mass x velocity
  - Mass $= 1$
  - $v(t) = \frac{\partial \theta(t)}{\partial t}$
  - $f(t) = \frac{\partial v(t)}{\partial t}$
    - One force is $-\nabla_\theta J(\theta)$
    - Another is $-v(t)$ (viscous drag)
  - $v \leftarrow \alpha\, v \; - \epsilon \nabla_\theta \left(\frac{1}{m} \sum L\left(f\left(x^{(i)}; \theta \right), y^{(i)} \right) \right) \qquad \alpha \in [0,1)$
  - $\theta \leftarrow \theta + v$
- Imagine you are pushing a particle through parameter space
  - The previous gradients contribute to the new direction that should be taken
- We can think of the particle as being like a hockey puck sliding down an icy surface.
  - Whenever it descends a steep part of the surface, it gathers speed and continues sliding in that direction until it begins to go uphill again.

Momentum analysis

- If the momentum always observes gradient g
  - $\mathrm{v}_0 = -\epsilon \mathrm{g} \Rightarrow \mathrm{v}_\infty = -\epsilon \frac{g}{1-\alpha}$
  - $\alpha = 0.9 \Rightarrow v_\infty = 10\, v_0$

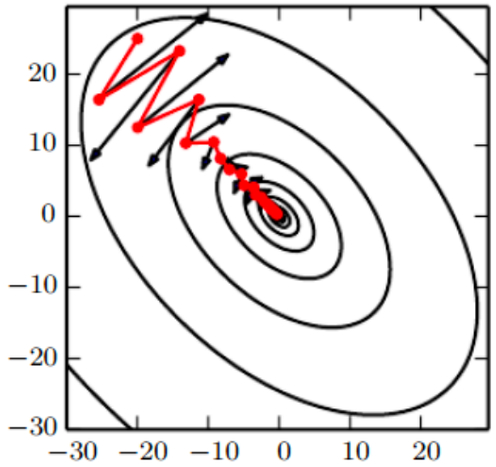


Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon. Compare also figure 4.6, which shows the behavior of gradient descent without momentum.

Parameter initialization strategy

- Break symmetry
  - initialize each unit to compute a different function from all of the other units.
- Set the biases for each unit to heuristically chosen constants (0 is good),
- initialize the weights randomly (Gaussian, truncate Gaussian or uniform)
  - Initial weights that are too large may result in exploding values during forward propagation or back-propagation, saturation or extreme sensitivity.
  - Initial weight that are reasonably large helps in breaking symmetry and propagate information successfully
    - For a layer with m inputs, n outputs: $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$
    - Glorot and Bengio: $U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$
- Another choice is to transfer the weights from another machine learning task (e.g. unsupervised learning)

Algorithms with adaptive learning rates

- If we believe that the directions of sensitivity are somewhat axis-aligned,
  - it can make sense to use a separate learning rate for each parameter,
  - and automatically adapt these learning rates throughout the course of learning.
- Example: AdaGrad

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

RMSProp
(Root Mean Square Propagation)

- The RMSProp algorithm (Hinton, 2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average.
- RMSProp is currently one of the go-to optimization methods being employed routinely by deep learning practitioners

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 with Nesterov momentum

- The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied ($\theta + \alpha v$).

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

Adam
(Adaptive Moments)

combination of RMSProp and momentum

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

2016)

Choosing the Right Optimization Algorithm

- Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, Adam, and AdaDelta.
- The choice depends largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning)

Batch normalization the problem

- Consider this very simple, yet deep, neural network
- $\hat{y} = x w_1 w_2 w_3 \dots w_l = f(\boldsymbol{w})$
  - $\boldsymbol{g} = \nabla_{\boldsymbol{w}} \hat{y}$
  - $\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \boldsymbol{g}$
  - $f(\boldsymbol{w} - \epsilon \boldsymbol{g}) \approx \hat{y} - \epsilon \boldsymbol{g}^{\boldsymbol{T}} \boldsymbol{g}$
  - If $g = 1$ and we want to decrease $\hat{y}$ by 0.1
    - Take $\epsilon = \frac{0.1}{g^T g}$
  - In reality: $f(\boldsymbol{w} - \epsilon \boldsymbol{g}) = x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l)$
  - One of the second order terms: $\epsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i$
    - Might be negligible if $\prod_{i=3}^{l} w_i$ is small
    - Or might be exponentially large
    - This makes it very hard to choose an appropriate learning rate

Batch normalization the solution

- Provides an elegant way of reparametrizing almost any deep network
- Can be applied to any input or hidden layer in the network
- The goal is to isolate the updates across many layers
- Let $\boldsymbol{H}$ be a minitbatch of activations of the layer to normalize

$$\boldsymbol{H}' = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

  - $\boldsymbol{\mu}$ contains the mean of each unit $\boldsymbol{\mu} = \frac{1}{m}\sum_i \boldsymbol{H}_{i,:}$
  - $\boldsymbol{\sigma}$ contains the standard deviation of each unit $\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m}\sum_i \left(\boldsymbol{H}_{i,:} - \mu_i\right)^2}$ $\delta = 10^{-8}$

Batch normalization interpretation

- Crucially we backpropagate through these operations
  - They are part of the computation graph
- At test time, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ may be replaced by running averages that were collected during training time.
  - This allows the model to be evaluated on a single example
- Revisiting $\hat{y} = xw_1w_2w_3 \ldots w_l = f(\boldsymbol{w})$
  - Now we can say: $\hat{y} = \hat{h}^{(l-1)}w_l$ (where $\hat{h} = \frac{h-\mu}{\sigma}$)
  - The parameters at the lower layers have mostly no effect
- In non-linear deep networks,
  - batch normalization acts to standardize only the mean and variance of each unit in order to stabilize learning, (first and second order statistics)
  - but allows the relationships between units and the nonlinear statistics of a single unit to change, (higher order statistics)

Batch normalization expressive power

- To maintain the expressive power, we replace $\boldsymbol{H}$ by:

$$\boldsymbol{\gamma H' + \beta}$$

  - We can represent the same family of functions of the input as the old parametrization
  - And have different learning dynamics in the same time

Checkpoint with Mean 0, Stddev 1

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

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

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

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

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

Coordinate descent

- We minimize $f(x)$ with respect to a single variable x_i, then minimize it with respect to another variable x_j and so on,
- Block coordinate descent refers to minimizing with respect to a subset of the variables simultaneously
- Example: sparse coding
- Find W that can linearly decode a matrix of activation values H to reconstruct the training set X.
  - we can divide the inputs to the training algorithm into two sets:
    - the dictionary parameters W
    - and the code representations H .

$$J(\boldsymbol{H}, \boldsymbol{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} \left(\boldsymbol{X} - \boldsymbol{W}^\top \boldsymbol{H} \right)^2_{i,j}.$$

Polyak averaging

- Polyak averaging consists of averaging together several points in the trajectory through parameter space visited by an optimization algorithm.
- If t iterations of gradient descent visit points $\theta^{(1)}, \ldots, \theta^{(t)}$, then the output of the Polyak averaging algorithm is

$$\hat{\theta}^{(t)} = \frac{1}{t} \sum_{i} \hat{\theta}^{(i)}$$

- On some problem classes, such as gradient descent applied to convex problems, this approach has strong convergence guarantees.
- When applied to neural networks, its justification is more heuristic, but it performs well in practice.
- The basic idea is that the optimization algorithm may leap back and forth across a valley several times without ever visiting a point near the bottom of the valley.
- The average of all of the locations on either side should be close to the bottom of the valley though.
- We can also use the running average:

$$\hat{\boldsymbol{\theta}}^{(t)} = \alpha \hat{\boldsymbol{\theta}}^{(t-1)} + (1 - \alpha)\boldsymbol{\theta}^{(t)}.$$

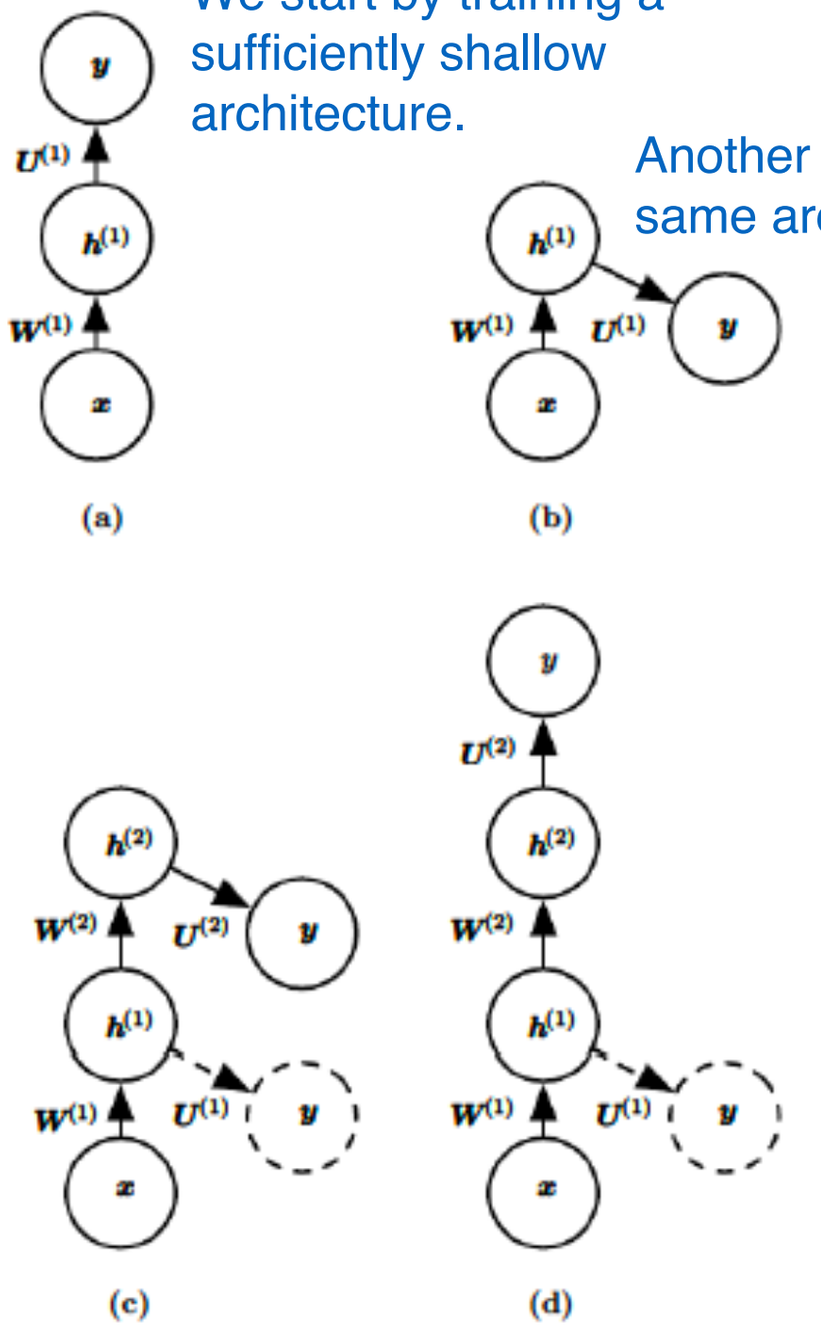
Supervised Pretraining

- Pretraining algorithms break supervised learning problems into other simpler supervised learning problems.
- Greedy algorithms break a problem into many components, then solve for the optimal version of each component in isolation

Greedy supervised pretraining

We start by training a sufficiently shallow architecture.

Another drawing of the same architecture



We keep only the input-to-hidden layer of the original network and discard the hidden-to-output layer. We send the output of the first hidden layer as input to another supervised single hidden layer MLP that is trained with the same objective as the first network was, thus adding a second hidden layer. This can be repeated for as many layers as desired

Another drawing of the result, viewed as a feedforward network. To further improve the optimization, we can jointly fine-tune all the layers, either only at the end or at each stage of this process

Transfer learning

- Yosinski et al. (2014) pretrain a deep convolutional net with 8 layers of weights on a set of tasks:
  - a subset of the 1000 ImageNet object categories
- and then initialize a same-size network with:
  - the first k layers of the first net
  - the upper layers initialized randomly
- All the layers of the second network are then jointly trained to perform a different set of tasks
  - another subset of the 1000 ImageNet object categories,
  - with fewer training examples than for the first set of tasks.

Designing Models to Aid Optimization

- It is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm
- Most of the advances in neural network learning over the past 30 years have been obtained by changing the model family rather than changing the optimization procedure
  - Stochastic gradient descent with momentum, which was used to train neural networks in the 1980s, remains in use in modern state of the art neural network applications.
- Specifically, modern neural networks reflect a design choice to use:
  - linear transformations between layers
  - and activation functions that are differentiable almost everywhere and have significant slope in large portions of their domain.
- Other ideas:
  - Fitnets: one network is a teacher, the other is a student
  - skip connections: mitigate the vanishing gradient problem
  - GoogLeNet: adding extra copies of the output to the intermediate hidden layers of the network, these extra copies are removed after the training

Continuation Methods

- Continuation methods are a family of strategies that can make optimization easier by choosing initial points to ensure that local optimization spends most of its time in well-behaved regions of space.
- The idea behind continuation methods is to construct a series of objective functions over the same parameters. In order to minimize a cost function $J(\theta)$, we will construct new cost functions $\{J^{(0)}, \ldots, J^{(n)}\}$.
- These cost functions are designed to be increasingly difficult, with $J^{(0)}$ being fairly easy to minimize, and $J^{(n)}$, the most difficult, being $J(\theta)$, the true cost function motivating the entire process.
- When we say that $J^{(i)}$ is easier than $J^{(i+1)}$, we mean that it is well behaved over more of θ space.
- Example:

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\theta' \sim \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}, \sigma^{(i)2})} J(\boldsymbol{\theta}')$$

  - The intuition for this approach is that some non-convex functions become approximately convex when blurred

Curriculum learning

- Curriculum learning is based on the idea of planning a learning process to begin by learning simple concepts and progress to learning more complex concepts that depend on these simpler concepts.
- Earlier $J^{(i)}$ are made easier by increasing the influence of simpler examples
  - either by assigning their contributions to the cost function larger coefficients,
  - or by sampling them more frequently
- the way humans teach:
  - teachers start by showing easier and more prototypical examples
  - and then help the learner refine the decision surface with the less obvious cases.
- stochastic curriculum: a random mix of easy and difficult examples is always presented to the learner, but where the average proportion of the more difficult examples (for example sequences with longer-term dependencies) is gradually increased.

Useful links

- Ian's lecture on batch normalization (minutes 3:40 → 13)
  - https://www.youtube.com/watch?v=Xogn6veSyxA
- Series: Optimization (A collection of four posts)
  - https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/
  - https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam
  - https://blog.paperspace.com/vanishing-gradients-activation-function/
  - https://blog.paperspace.com/busting-the-myths-about-batch-normalization/
- ADAM's Story and Proof
  - https://www.youtube.com/watch?v=n65pEIMp6x8
  - https://www.youtube.com/watch?v=q3zHYfaKg4k

Optional
Approximate Second-Order Methods

- Newton's Method
- Conjugate Gradients
- Non-linear conjugate gradients
- Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm

Newton's method

- For a locally quadratic function (with positive definite $\boldsymbol{H}$), by rescaling the gradient by H^{-1}, Newton's method jumps directly to the minimum.
- If the objective function is convex but not quadratic (there are higher-order terms), this update can be iterated

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),y^{(i)}).$$

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

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

Newton's method algorithm

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

If the eigenvalues of the Hessian are not all positive, for example, near a saddle point, then Newton's method can actually cause updates to move in the wrong direction. This situation can be avoided by regularizing the Hessian.

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [H(f(\boldsymbol{\theta}_0)) + \alpha \boldsymbol{I}]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0).$$

The method of steepest descent

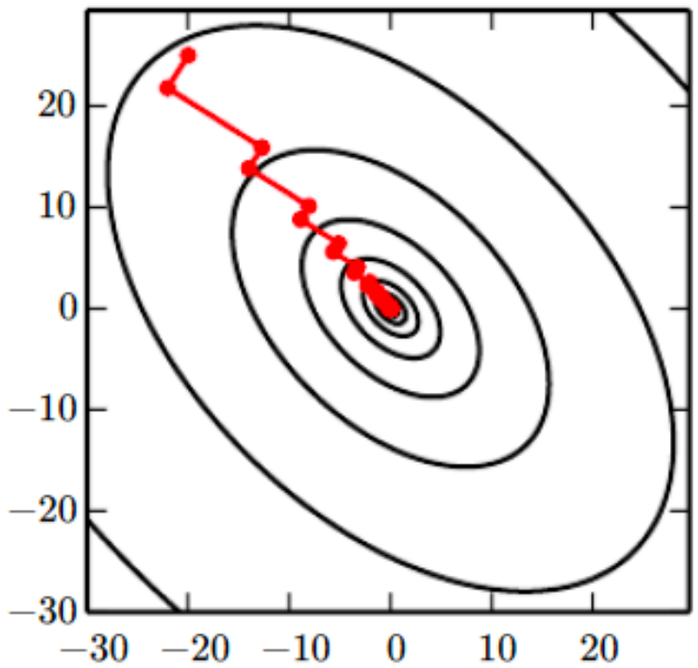


Figure 8.6: The method of steepest descent applied to a quadratic cost surface. The method of steepest descent involves jumping to the point of lowest cost along the line defined by the gradient at the initial point on each step. This resolves some of the problems seen with using a fixed learning rate in figure 4.6, but even with the optimal step size the algorithm still makes back-and-forth progress toward the optimum. By definition, at the minimum of the objective along a given direction, the gradient at the final point is orthogonal to that direction.

Method of conjugate gradients

- In the method of conjugate gradients, we seek to find a search direction that is conjugate to the previous line search direction, i.e. it will not undo progress made in that direction.
- At training iteration t, the next search direction $\boldsymbol{d}_t$ takes the form:

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1}$$

- Two directions, $\boldsymbol{d}_t$ and $\boldsymbol{d}_{t-1}$, are defined as conjugate if $\boldsymbol{d}_t \boldsymbol{H} \boldsymbol{d}_{t-1} = 0$

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$$

2. Polak-Ribière:

$$\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$$

Conjugate gradient algorithm

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

Nonlinear conjugate gradients

- The nonlinear conjugate gradients algorithm includes occasional resets where the method of conjugate gradients is restarted with line search along the unaltered gradient.
- Practitioners report reasonable results in applications of the nonlinear conjugate gradients algorithm to training neural networks,
  - though it is often beneficial to initialize the optimization with a few iterations of stochastic gradient descent before commencing nonlinear conjugate gradients.
  - Also, while the (nonlinear) conjugate gradients algorithm has traditionally been cast as a batch method, minibatch versions have been used successfully for the training of neural networks

BFGS

- The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm attempts to bring some of the advantages of Newton's method without the computational burden.
  - In that respect, BFGS is similar to the conjugate gradient method.
- However, BFGS takes a more direct approach to the approximation of Newton's update. Recall that Newton's update is given by:

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

- The primary computational difficulty in applying Newton's update is the calculation of the inverse Hessian H^{-1} (which is $O(n^3)$).
  - The approach adopted by quasi-Newton methods (of which the BFGS algorithm is the most prominent) is to approximate the inverse with a matrix M_t that is iteratively refined by low rank updates to become a better approximation of H^{-1}.
- BFGS algorithm must store the inverse Hessian matrix, M, that requires $O(n^2)$ memory, making BFGS impractical for most modern deep learning models that typically have millions of parameters.