Optimization for Training Deep Models

Nathan Bartley
nbartley@usc.edu

Deep learning algorithms involve optimization in many contexts

- Inference in PCA involves an optimization problem
- Restricted Boltzmann Machines are learned via approximation of maximum likelihood
- In many probabilistic graphical models, we maximize likelihood by maximizing the evidence lower bound to do inference

Most of these optimization problems are concerned with training deep neural networks, and because training is so costly we have a specialized set of methods developed for this

We will be concerned primarily for one particular case of optimization:

$$\arg\min_{\Theta} J(\Theta)(+\alpha\Omega(\Theta))$$

The layout of the lecture will be as follows:

1. How training optimization for machine learning differs from pure optimization
2. Concrete challenges for training deep models
3. Defining several algorithms (& strategies for Θ initialization)
4. Review several higher order strategies that combine simple ones
5. Practical overall strategies for an entire deep learning pipeline (Ch. 11)

1 How Learning differs from Pure Optimization

The primary difference between machine learning and pure optimization is that machine learning often acts indirectly Typically the machine learning learning set up is as follows:

- **Goal**: increase some performance metric P
- **Method**: minimize some $J(\Theta)$ with hopes of optimizing P

Pure optimization is often just concerned with optimizing $J(\Theta)$ directly

Notation. we discuss $J(\Theta)$ as a supervised learning unregularized cost function. The most straightforward function is the **empirical risk**

$$J(\Theta) = \mathop{\mathbb{E}}_{(x,y)\sim\hat{p}_{data}} L(f(x;\Theta), y)$$

- L is the per-example loss function
- $f(x;\Theta)$ is the predicted output for x
- $\hat{p}_{data}$ is the empirical distribution

In a perfect world we would try to use p_{data} the data-generating distribution s.t. $J^*(\Theta) = \mathbb{E}_{(x,y)\sim p_{data}}$ to compute the **expected generalization error**.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

As a comparison, an unsupervised cost function (e.g., for k-Means) might be:

$$J(\Theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} \mathbb{1}_{[x_n \in k]} \|x_n - \mu_k\|^2$$

And the regularized version of the supervised empirical risk might look like:

$$J(\Theta) = \mathop{\mathbb{E}}_{(x,y)\sim\hat{p}_{data}} L(f(x;\Theta), y) + \alpha\Omega(\Theta)$$

When using a set of m training examples, the empirical risk can be defined as the average loss over that set:

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

Optimizing this is called **empirical risk minimization**. Two problems arise with ERM:

1. Empirical risk is a proxy for true risk, but ERM is prone to overfitting
2. Many popular loss functions have no useful (or tractable) gradient

1.1 Surrogate Loss Functions & Early Stopping

Often exactly minimizing a loss function (e.g., expected 0-1 loss) is intractable. Because of this we often use a proxy, or **surrogate loss function**, that is easier to minimize. For example, we can use negative log-likelihood of the correct class as a surrogate for the 0-1 loss.

A benefit of using surrogates is that they can actually learn more

- When using a log-likelihood surrogate the test set 0-1 loss might keep decreasing after training set 0-1 loss reaches 0 but continuing minimizing negative log-likelihood

Another difference between pure and machine learning optimization is that we will often do something like **early stopping** based on the true underlying loss function. This helps to prevent overfitting and leaves large gradients in the surrogate loss (which in pure optimization you usually leave small gradients in the function).

1.2 Batch and Mini-batch

Optimization in machine learning typically computes each Θ pdate based on an expected value of the cost function estimated on only a subset of x.

For example we can decompose MLE as:

$$\Theta^{MLE} = \arg\max_{\Theta} \sum_{i=1}^{M} \log p_{model}(x^{(i)}, y^{(i)}; \Theta)$$

Maximizing this is equivalent to maximizing:

$$J(\Theta) = \mathop{\mathbb{E}}_{(x,y)\sim\hat{p}_{data}} [\log p_{model}(x, y; \Theta)]$$

but if we want to maximize it via the gradient

$$\nabla_\Theta J(\Theta) = \mathop{\mathbb{E}}_{(x,y)\sim\hat{p}_{data}} [\nabla_\Theta \log p_{model}(x, y; \Theta)]$$

Not only is this hard to compute sometimes, but it's expensive to do it over the entire dataset

Recall: the standard error of the mean $= \frac{\sigma}{\sqrt{n}}$.

If your mean is the gradient, using 10,000 examples only reduces the standard error by a factor of 10 when compared to 100 examples. You may also find many examples that provide very similar contributions to ∇_Θ.

These two ideas motivate the **Minibatch / Minibatch Stochastic Gradient Descent** approach: one can relatively cheaply compute an approximation of the gradient by using $m < n$ training examples. **Deterministic / Batch Gradient Descent** uses the whole training set. **Stochastic Gradient Descent** is usually defined as using just one training example, but in this book sometiems SGD and Minibatch are conflated.

1.2.1 Practical Tips

- Larger batch sizes provide more accurate estimates of the gradient
- When training in parallel, use some absolute minimum batch size
  - too small and you get no reduction in time per minibatch
- Max batch size will be limited by available memory
- Sometimes you get hardware acceleration with specific sizes of arrays
  - GPUs often get better runtimes on powers of 2 (16-256)
- Small batches can offer a regularizing effect
  - Generalization error is often best with batch size of 1
  - To run this might require a small learning rate, as to maintain stability (there will be high variance in the estimate of the gradient)

A good motivation for this last point is that the first pass over each minibatch follows the gradient of the *true* generalization error. On subsequent passes the estimate becomes biased (due to sampling already seen values)

Similarly, some algorithms are more sensitive to sampling error than others, either because they use information that is difficult to estimate accurately with few examples, or because they use information in ways that amplifies sampling error

For instance, methods that compute updates based only on the gradient **g** are usually robust and can handle smaller batch sizes (e.g., 100). Second-order methods, those that rely on the Hessian H and computing updates like $H^{-1}g$ typically need larger batch sizes like 10,000 to minimize fluctuations in the estimate

Practical considerations. It is important to sample minibatches randomly to ensure independent samples. It is also important that two subsequent minibatches be independent. For example, make sure to shuffle sentences in your document!

As datasets grow in size, it is more common to make only one pass or even an incomplete pass over the training set

- Overfitting is not an issue when using **extremely** large datasets, but underfitting and computational complexity are

2 Challenges in Neural Network Optimization

Even the shortcut of trying to turn to convex optimization has complications in machine learning. Non-convex optimization is an important problem in deep learning.

2.1 Ill Conditioning

One of the most general problems in numerical optimization is the ill-conditioning of the Hessian Matrix H.

- This can manifest by getting SGD stuck in the way that even very small steps increase the cost function.

Recall: The second-order Taylor series expansion of a cost function

$$\begin{aligned} x &= x^0 - \epsilon g f(x^{(0)} - \epsilon g) \\ &\approx f(x^{(0)}) - \epsilon g^T g + \frac{1}{2}\epsilon^2 g^T H g \end{aligned}$$

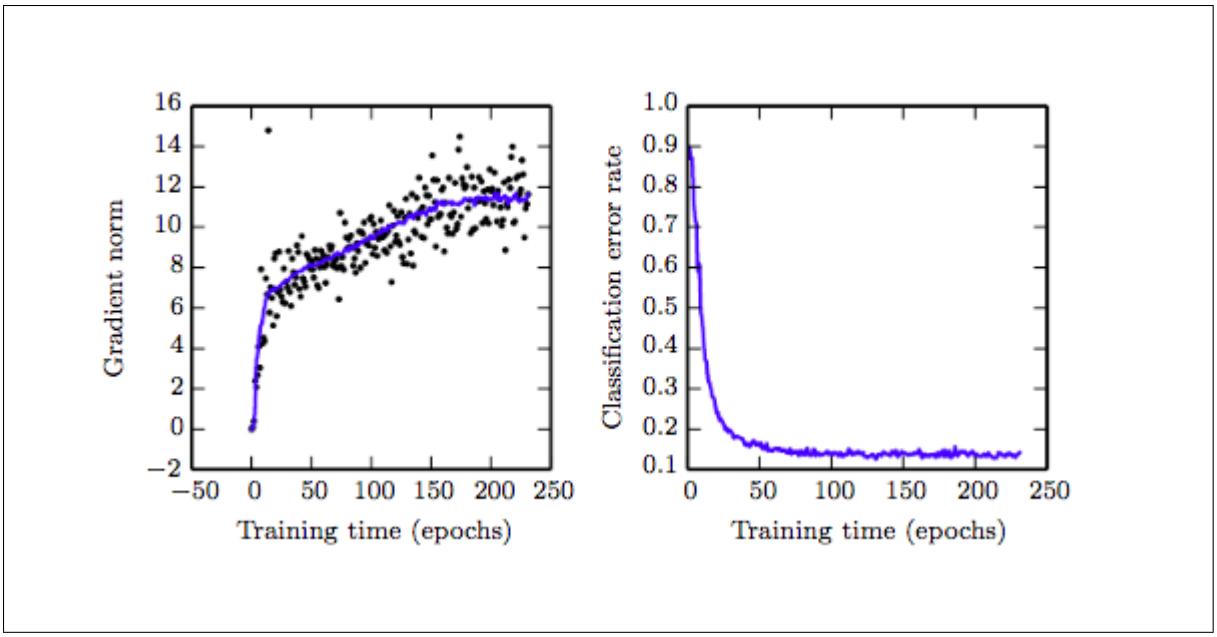

Figure 1: A growing gradient norm does not necessarily mean learning improves.

Ill-conditioning becomes a problem when $\frac{1}{2}\epsilon^2 g^T H g > \epsilon g^T g$, which means that learning will become very slow despite having a strong gradient. This is because ϵ will have to be small enough to compensate for strong curvature.

There are two ways we can deal with it:

1. Monitor $g^T g$ and $g^T H g$ throughout training and can probably stop when $g^T H g > g^T g$
2. Newton's method modified for neural networks (which we describe later)

2.2 Local Minima

In the convex case, any local minimum is guaranteed to be a global minimum. For nonconvex optimization functions (and generally for neural networks) there can be an extremely large number of local minima.

In the neural network case in particular, this can seem like an issue because any model with multiple equivalently parameterized latent variables have many local minima. This is largely because of the **model identifiablity problem**, where a model if defined as being **identifiable** if after training we can rule out all but one setting of a model's parameters. In the neural network with m layers and n nodes, this means that there are $n!^m$ ways of rearranging the hidden units.

Why this is not a problem, however, is that all of the local minima arising from nonidentifiability are equivalent to each other in terms of cost function value, which is not a problem when trying to reach convergence practically.

Local minima more generally are particularly problematic when they have high cost compared to the global minimum. However, in neural networks, it remains an open question how local minima relate to the global minimum. Researchers largely suspect that these local minima are not a problem, and that most are going to be low cost anyway, which is cheaper to find than the global minimum.

2.2.1 Practical Tips

A test that can rule out local minima as the problem is to plot the norm of the gradient over time. If it does not shrink and become insignificant than the problem is not related to local minima or critical points.

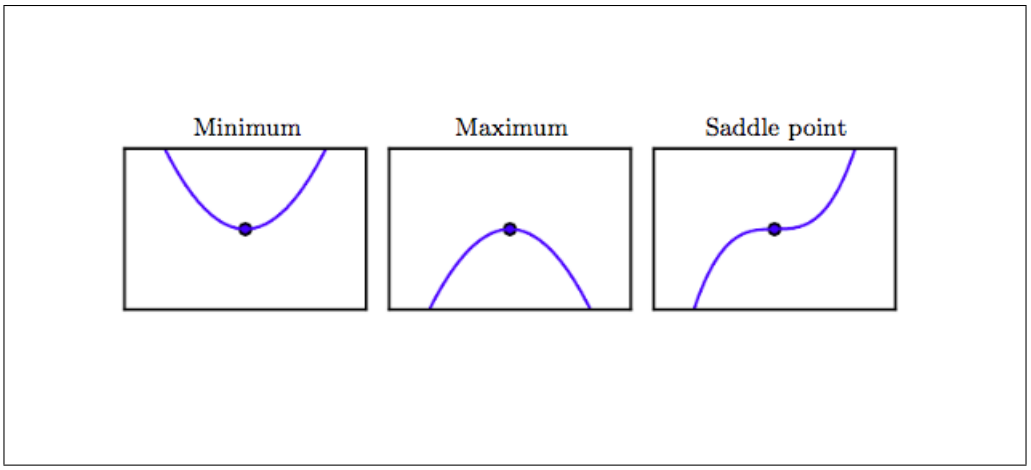

Figure 2: Types of critical points in one dimension.

NB. Positively testing that local minima are the problem in high dimensional space is very hard.

2.3 Plateaus, Saddle Points, and Other Flat Regions

At a saddle point, the Hessian matrix H has both positive and negative eigenvalues

To understand high dimensional issues, we consider random functions. For many classes of random functions $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ local minima are common in low dimension, but much rarer in high dimension. In other words, for many classes of random functions, the expected ratio of the number of saddle points to the number of local minima grows expontentially with n.

Intuition. Observe that the Hessian only has positive eigenvalues at a local minimum. At a saddle point, we have a mix of positive and negative. Consider the sign of each eigenvalue as the result of a fair coin toss. In one dimension, it is easy to get a positive value, but in n dimensions it is much harder to get all positive values.

An important property of many random functions is that the eigenvalues of H are more likely to be positive as we reach regions of lower cost. This means:

1. Local minima are more likely to have low cost than higher cost
2. Saddle points are likely to have high cost
3. Local maxima are likely to have extremely high cost

Does this behavior in random functions happen in neural networks? Dauphin et al. [2014] showed experimentally that real neural networks contain very many high cost saddle points. Compounded with some other work, this suggests that similar behavior might happen in neural networks.

Implications. Having many saddle points will make first-order methods struggle. Second-order methods (e.g., Newton's Method) may work to find a point with gradient 0, but will likely find saddle points.

Dauphin et al. [2014] describes a method for a saddle free Newton's Method that works much better in these situations, but it is difficult to scale.

NB. Wide flat regions with both gradient and H equal to zero exist as well, but these are a problem for all numerical optimization algorithms.

2.4 Cliffs and Exploding Gradients

Deep models often have very steep regions resembling cliffs. These usually result from the multiplication of several large weights together.

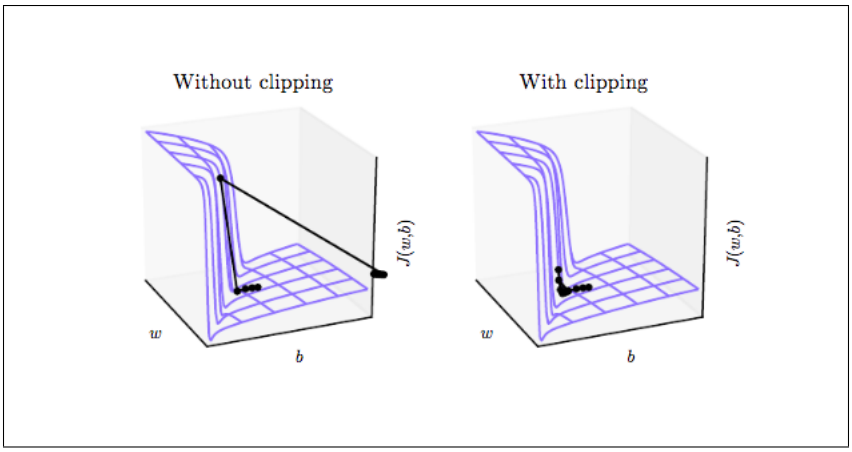

Figure 3: Cliff structures and gradient clipping.

We can deal with these cliff structures using **gradient clipping.** One way to do gradient clipping is to normal before the parameter update: if

$$\|g\| > vg \qquad \leftarrow \frac{gv}{\|g\|}$$

NB. Cliffs are common in RNNs due to multiplication of many factors (1 / timestep)

2.5 Long-Term Dependencies

Consider a deep feedforward network or an RNN with a high number of timesteps.

$\rightarrow$ The computational graph gets deep

Suppose a computational graph contains a path that consists of multiplying by a matrix Wt times

$$W^t = (vdiag(\lambda)V^{-1})^t = Vdiag(\lambda)^t V^{-1}$$

Any eigenvalues λ_i will explode if > 1 and vanish if < 1. In this way, cliffs result in exploding gradients.

This issue of long-term dependencies is really more of a problem for RNNS since feedforward networks use a different W for each layer. (Ch 10).

2.6 Inexact Gradients

Most optimization algorithms assume we have access to the exact gradient or to H. Often we only have a biased estimate. Similarly, sometimes the objective function is intractable (which implies the gradient is too). Because of this we do one of two things:

- approximate the gradient (e.g., contrastive divergence for the log-likelihood of a Boltzmann machine)
- we can choose a surrogate loss function that is easier to approximate than the true loss.

2.7 Poor Correspondence between Local and Global Structure

FIGURE

Up to now we have considered the properties of the loss function at a single point– if Θ lies on a cliff, if $J(\Theta)$ is poorly conditioned at Θ, if Θ is a single point, etc.

How does this impact global structure? Neural networks often do not arrive at a region of small gradient. Similarly, sometimes critical points do not exist (e.g., negative log-likelihood asymptotically

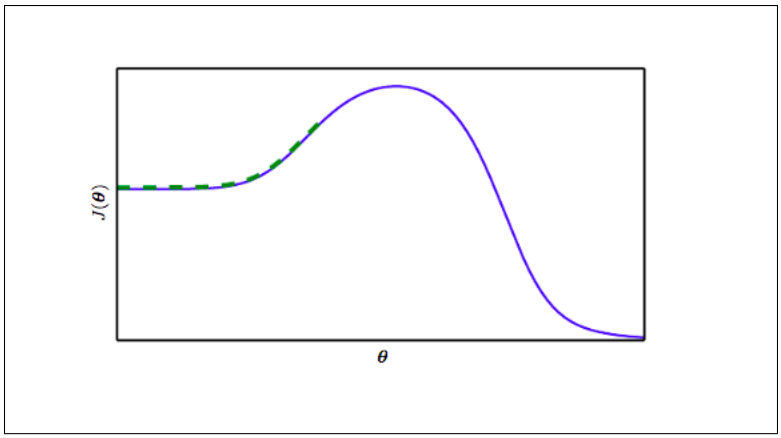

Figure 4: Local and Global correspondence. Local downhill moves on such a one-dimensional cost-function will not result in a local minimum.

approaches some value). Sometimes functions like $P(y|x) = N(y; f(\Theta), \beta^{-1})$ might have negative log-likelihood that approach $-\infty$.

More research needs to be done in understanding the factors that affect the length of the learning trajectory.

Regardless, all problems might be avoided if there exists a region of space connected reasonably directly to a solution by a path that local descent can follow, and if we can initialize learning to be in that region.

2.8 Theoretical limits to Optimization

There are several theoretical results that show that there are limits on the performance of any optimization algorithm we might design for neural networks, but these results typically have little bearing on the use of neural networks in practice.

For instance, some results only apply when the units output discrete values (not that common). Some describe problem classes that are intractable, but it is difficult to tell if a problem falls into that class.

3 Basic Optimization algorithms

3.1 Stochastic Gradient Descent (SGD)

SGD and its variants are probably the most used optimization algorithms for machine learning and deep learning. It takes an unbiased estimate of the gradient over m samples.

The crucial parameter is the learning rate ϵ. Often we will **decay** the learning rate such that $\epsilon_k > \epsilon_0 fork > 0$. We decay because the variance of the random sampling of m examples does not vanish over the course of training.

In practice, we use a linear decay:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau$$

where $\alpha = \frac{k}{\tau}$. After τ iterations we usually leave ϵ constant

Algorithm 1: Stochastic Gradient Descent Update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, ...$
Require: Initial parameter Θ
$k \leftarrow 1$;
while *stopping criterion not met* **do**
  Sample a minibatch of *m* examples from the training set $x^{(1)}, ..., x^{(m)}$ with corresponding targets $y^{(i)}$;
  Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{m} \nabla_\Theta \sum_i L(f(x^{(i)}; \Theta), y^{(i)})$;
  Apply update: $\Theta \leftarrow \Theta - \epsilon_k \hat{g}$;
  $k \leftarrow k + 1$;
end

3.1.1 Strategy

Usually choose your learning rate parameters by monitoring the learning curves (objective function values over time). This is an art more than a science.

Linear schedule strategy. For linear decay usually τ may be set to the number of iterations required to make a few hundred passes over the training set. Usually ϵ_τ is set to $0.01 * \epsilon_0$. In order to set ϵ_0:

1. too large if there are violent oscillations in the learning curve
2. too small if the learning curve is stuck
3. monitor the first several iterations and use a learning rate that is higher than the best performing rate at the time (but not too high)

This last idea arises from the observation that the optimal initial learning rate, in terms of training time and final cost value, is usually higher than the learning rate that yields the best performance after 100 iterations or so

3.1.2 Convergence

The most important property of SGD is that the computation time per update does not grow with n. SGD might converge to some fixed tolerance of its final test set error before finishing the training set.

We can measure the convergence rate by measuring the **excess error**

$$J(\Theta) - \min_\Theta J(\Theta)$$

In the convex case this excess error is $O(\frac{1}{\sqrt{K}})$ after k iterations. In the strongly convex case, excess error is $O(\frac{1}{k}$ after k iterations.

In the nonconvex case we refer to the Cramér-Rao bound, which states that generalization error cannot decrease faster than $O(\frac{1}{k}$ (and which any faster implies overfitting).

However, this treatment of the problem ignores the constants obscured in the asymptotic analysis. Often SGD makes rapid initial progress while evaluating the gradient using very few examples, which is useful when trying to find a "good enough" generalization error quickly.

3.2 Momentum

Momentum accelerates learning especially when:

1. there is high curvature
2. there are small consistent gradients
3. there are noisy gradients

Intuition. The momentum algorithm accumulates an exponentially decaying moving average of the past gradients and continues to move in their direction.

Formally. The momentum algorithm introduces v the velocity of the gradient. We assume the particle under inspection has unit mass. To compute momentum we take mass$*v$. The parameter update is as follows:

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

Algorithm 2: Stochastic Gradient Descent with momentum

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

The step size is going to be the largest when many successive gradients point in the same direction. In other words, if momentum algorithm always observes g, it will accelerate towards $-g$ until it reaches terminal velocity where each step is $\frac{\epsilon \|g\|}{1-\alpha}$.

This implies that α scales the gradient like $\frac{1}{1-\alpha}$. For instance, when $\alpha = 0.9$ this means a 10x speed up as compared to regular SGD. As such, common values of ϵ are 0.5, 0.9, and 0.99.

3.2.1 Physical Intuition for Momentum

We are simulating a particle subject to continuous time Newtonian Dynamics:

- $\Theta(t)$:= position of the particle at any point in time
- $f(t)$:= net force the particle experiences at time t = $\frac{d^2}{dt^2}\Theta(t)$

To remove the complexity of a second derivative we introduce $v(t)$:= velocity of the point at time t. This yields:

$$v(t) = \frac{d}{dt}\Theta(t) f(t) \qquad\qquad = \frac{d}{dt} v(t)$$

We can solve these differential equations via Euler's Method / numerical simulation.

What are the forces acting on the particle?

1. One is proportional to $-\nabla_\Theta J(\Theta)$. If the particle were a hockey puck on ice, this force would be the acceleration the puck feels when it reaches a slope on the surface.
2. The other is $-v(t)$ viscous drag, to ensure the particle loses energy over time.

3.2.2 Nesterov Momentum

Sutskever et al. [2013] introduced a variant of momentum inspired by Nesterov's accerlated gradient method:

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

The gradient in this case is evaluated after the current velocity is applied. In the convex batch gradient case, this is shown to bring convergence of excess error down from $O(\frac{1}{k}) \rightarrow O(\frac{1}{k^2})$. It does not seem to improve SGD generally.

Algorithm 3: Stochastic Gradient Descent with momentum

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

4 Parameter Intialization Strategies

Training algorithms for deep learning models are usually iterative and thus need some initialized point to start from. This initial point can determine whether you converge at all, how fast you converge, and generalization error.

Modern strategies for initializing parameters are simple and heuristic (it's not a well understood problem!).

The only property known with complete certainty is the Θ_0 needs to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, they need different Θ_0. Otherwise a deterministic learning algorithm will update the units the same way!

To break symmetry, the goal is to have each unit compute a different function (which suggests they should all start at different places). A good strategy for doing so is to randomly initialize each unit from a high-entropy distribution over a high-dimensional space.

4.1 General Initialization Strategies

There are number of different strategies towards initialization:

1. Set biases to heuristically chosen constants, not random (same with extra parameters)
2. Initialize all weights from a Gaussian or Uniform distribution
   - The scale of the distribution has an impact on optimization
   - Larger initial weights yield a stronger symmetry breaking effect
     - Too big of weights might explode the gradient, or saturate the unit (causing gradient loss). Here, saturation means to squeeze the input (i.e., ReLU = max(0,x) is non-saturating, but $\sigma(x)$ is)
3. Initialize the weights of a fully connected layer with m inputs and n outputs by sampling from $U(-\frac{1}{\sqrt{m}}, +\frac{1}{\sqrt{m}})$
4. Glorot and Bengio [2010] introduced the method of **normalized initialization**:

$$W_{ij} \sim U(-\sqrt{\frac{6}{m+n}}, +\sqrt{\frac{6}{m+n}}$$

   - This assumes no nonlinearities, but empirically seems to work well on nonlinear models
5. Saxe et al. [2013] initializes to random orthogonal matrices with a **gain** factor g that accounts for nonlinearity at each layer. Increasing this gain pushes the network toward the regime where activations increase in norm as they propagate forward and gradients increase in norm backward.

6. Martens [2010] introduced sparse initialization in which each unit is initialized to have k nonzero weights.
    - The idea here is to keep the total amount of input to the unit independent from the total amount of inputs m
    - This imposes a strong prior for weights drawn from large Gaussians (since it might drop a value entirely instead of shrinking it). This can cause issues for units like maxout units.
7. Look at the range or standard deviation of the activations or gradients on a single minibatch of data. If weights are too small, range of activations will shrink as the activations propagate forward.

NB. Often the optimal criteria for initial weights do not lead to optimal performance. This might be because:

1. We may be using the wrong criteria
2. Properties impose at initialization may not hold for training
3. We might be speeding up optimization but hurting generalization.

Tips. Treat the initial scale of the weights for each layer as a hyperparameter by themselves (search by using random search). Simiarly, treat whether you use dense or sparse initialization the same.

4.2 Initialization other parameters

4.2.1 Strategies

1. Setting biases to 0 is compatible with most weight intiialization schemes
2. If a bias is for an output unit, assume that the initial weights are small enough that the output of the unit is determined only by the bias
3. Sometimes we choose the bias to avoid causing too much saturation at initialization.
    - Setting a ReLU bias to 0.1 instead of 0
    - This is not compatible with initialization schemes that do not expect strong input from biases (e.g., random walk)
4. Sometimes set biases ≈ 1 so that other units have a chance to learn (e.g., forget gate in LSTM)
5. Another common parameter is variance / precision

$$p(y|x) = N(y|w^T x + b, \frac{1}{\beta})$$

   We can usually set $\beta = 1$.
6. Initialize a supervised model with parameters learned by an unsupervised one trained on same inputs. This encodes information about the distribution into the initial parameters.

5 Adaptive Learning Rates

There a number of different learning algorithms that relax the assumption that we need a fixed learning rate (nor one that decays, etc). Here we describe a number of different adaptive learning rate algorithms.

5.1 Delta-bar-delta

This is an early heuristic approach introduced by Jacobs [1988] to adapting the learning rate over time. If the partial derivatives of the loss function with respect to some parameter remains the same sign, we increase Θ. If the partials change sign, Θ decreases.

5.2 AdaGrad

This approach introduced by Duchi et al. [2011] individually adapts the learning rates of all the parameters by scaling them inversley proporitional to the square root of the sum of all historical squared values of the gradient. This gives us greater progress in the gentle sloped directions of parameter space, and performs well for some but not all deep learning models.

Algorithm 4: AdaGrad Algorithm

Require: Global learning rate ϵ
Require: Initial parameter Θ
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable r = 0;
while *stopping criterion not met* **do**
  Sample a minibatch of *m* examples from the training set $x^{(1)}, ..., x^{(m)}$ with corresponding targets $y^{(i)}$;
  Compute gradient: $g \leftarrow \frac{1}{m} \nabla_\Theta \sum_i L(f(x^{(i)}; \Theta), y^{(i)})$;
  Accumulate squared gradient: $r \leftarrow r + g \odot g$;
  Compute update: $\triangle\Theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$ (division and square root applied element-wise);
  Apply update: $\Theta \leftarrow \Theta + \triangle\Theta$;
end

5.3 RMSProp

AdaGrad is designed to converge rapidly when applied to a convex function. RMSProp, introduced by Hinton et al. [2012], modifies AdaGrad at the gradient accumulation step into an exponentially

weighted moving average to work better in a nonconvex setting. This weighted moving average discards values at the extreme past, and as such is one of the goto training algorithms.

Algorithm 5: RMSProp Algorithm

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameter Θ
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers
Initialize gradient accumulation variable r = 0;
while *stopping criterion not met* **do**
  Sample a minibatch of *m* examples from the training set $x^{(1)}, ..., x^{(m)}$ with corresponding targets $y^{(i)}$;
  Compute gradient: $g \leftarrow \frac{1}{m} \nabla_\Theta \sum_i L(f(x^{(i)}; \Theta), y^{(i)})$;
  Accumulate squared gradient: $r \leftarrow \rho r + (1 - \rho) g \odot g$;
  Compute parameter update: $\triangle\Theta = -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$ (division and square root applied element-wise);
  Apply update: $\Theta \leftarrow \Theta + \triangle\Theta$;
end

Algorithm 6: RMSProp Algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α
Require: Initial parameter Θ, initial velocity **v**
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers
Initialize gradient accumulation variable r = 0;
while *stopping criterion not met* **do**
  Sample a minibatch of *m* examples from the training set $x^{(1)}, ..., x^{(m)}$ with corresponding targets $y^{(i)}$;
  Compute interim update: $\tilde{\Theta} \leftarrow \Theta + \alpha v$;
  Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\tilde{\Theta}} \sum_i L(f(x^{(i)}; \tilde{\Theta}), y^{(i)})$;
  Accumulate gradient: $r \leftarrow \rho r + (1 - \rho) g \odot g$;
  Compute velocity update: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot g$ ($\frac{1}{\sqrt{r}}$ applied element-wise;
  Apply update: $\Theta \leftarrow \Theta + v$;
end

5.4 Adam

Named for "Adaptive Moments", Adam can be considered a variant of RMSProp with momentum. Introduced by Kingma and Ba [2014], one of the major differences is that Adam incorporates momentum directly as an estimate of the first order moment (rather than applying momentum to the

rescaled gradients). Adam also has a bias correction not really found in regular SGD + Momentum. Given this, Adam is generally regarded as robust to hyperparameters.

Algorithm 7: 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, usually 10^{-8}, for numerical stabilization
Require: Initial parameters Θ
Initialize 1st and 2nd moment variables $s = 0, r = 0$;
Initialize time step t = 0;
while *stopping criterion not met* **do**
  Sample a minibatch of *m* examples from the training set $x^{(1)}, ..., x^{(m)}$ with corresponding targets $y^{(i)}$;
  Compute gradient: $g \leftarrow \frac{1}{m}\nabla_\Theta \sum_i L(f(x^{(i)}; \Theta), y^{(i)})$;
  $t \leftarrow t + 1$;
  Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1)g$;
  Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2)g \odot g$;
  Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$;
  Correct bias in second moment: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$;
  Compute update: $\triangle\Theta = -\epsilon\frac{s}{\sqrt{\hat{r}}+\delta}$ (operations applied element-wise);
  Apply update: $\Theta \leftarrow \Theta + \triangle\Theta$;
end

5.5 AdaDelta

AdaDelta (introduced by Zeiler [2012]) is an extension of AdaGrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Like RMSprop, it does not keep the entire history, but some fixed size window *w*.

It keeps a running average of the accumulated past gradients $\mathbb{E}[g^2]_t$ that depends only on the previous average and the current gradient:

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

We treat γ similar to a momentum term, and can set it around 0.9. To ensure the update has the same hypothetical units as the parameters, AdaDelta keeps track of another running average of the squared parameter updates $\mathbb{E}[\triangle\Theta]_t$.

Algorithm 8: AdaDelta

Require: Constant ϵ, Decay rate ρ
Require: Initial parameters Θ
Initialize accumulation variables $\mathbb{E}[g^2]_0 = 0, \mathbb{E}[\triangle\Theta^2]_0 = 0$;
Initialize time step $t = 1$;
while *stopping criterion not met* **do**
  Sample a minibatch of *m* examples from the training set $x^{(1)}, ..., x^{(m)}$ with corresponding targets $y^{(i)}$;
  Compute gradient: $g \leftarrow \frac{1}{m}\nabla_\Theta \sum_i L(f(x^{(i)}; \Theta), y^{(i)})$;
  Accumulate gradient: $\mathbb{E}[g^2]_t = \rho\,\mathbb{E}[g^2]_{t-1} + (1 - \rho)g_t^2$;
  Compute update: $\triangle\Theta_t = -\frac{\sqrt{\mathbb{E}[\triangle\Theta^2]_{t-1}+\epsilon}}{\sqrt{\mathbb{E}[g^2]_t+\epsilon}}g_t$;
  Accumulate Updates: $\mathbb{E}[\triangle\Theta^2]_t = \rho\,\mathbb{E}[\triangle\Theta^2]_{t-1} + (1 - \rho) \triangle \Theta_t^2$;
  Apply Update: $\Theta_{t+1} = \Theta_t + \triangle\Theta_t$;
  $t \leftarrow t + 1$;
end

5.6 Choosing the Right Optimization Algorithm

There is no clear consensus what the right optimization algorithm is. Schaul et al. [2013] presents a comparison of algorithms across a number of different tasks. That being said, the most popular algorithms are:

- SGD
- SGD with momentum
- RMSProp
- RMSProp with momentum
- AdaDelta
- Adam

6 Approximate Second-order Methods

Recall: we are primarily considering empirical risk:

$$J(\Theta) = \mathop{\mathbb{E}}_{(x,y)\sim\hat{p}_{data}}[L(f(x;\Theta),y)] \qquad = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\Theta),y^{(i)})$$

Here we consider second-order methods, or those methods that make use of the second derivatives of the objective function to improve optimization. The most prominent methods are Newton's method, those involving conjugate gradients, and BFGS.

6.1 Newton's Method

This method is based on a second-order Taylor series expansion to approximate $J(\Theta)$ near some Θ_0:

$$J(\Theta) \approx J(\Theta_0) + (\Theta-\Theta_0)^T\nabla_\Theta J(\Theta_0) + \frac{1}{2}(\Theta-\Theta_0)^T H(\Theta-\Theta_0)$$

When we solve for the critical point we get:

$$\Theta^* = \Theta_0 - H^{-1}\nabla_\Theta J(\Theta_0)$$

Locally quadratic functions (and those with positive definite H) will jump straight to the minimum. If the function is convex but not quadratic we can iterate using this method (as long as H is positive definite)

Intuition. In deep learning we usually have a nonconvex objective function surface, e.g., one with lots of saddle points. If eigenvalues of H are not all positive (for instance, near a saddlepoint), then we can update in the wrong direction.

Algorithm 9: Newton's method with objective $J(\Theta) = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\Theta),y^{(i)}$

Require: Initial parameter Θ_0
Require: Training set of *m* examples
while *stopping criterion not met* **do**
  Compute gradient: $g \leftarrow \frac{1}{m}\nabla_\Theta \sum_i L(f(x^{(i)};\Theta),y^{(i)})$;
  Compute Hessian: $H \leftarrow \frac{1}{m}\nabla^2_\Theta \sum_i L(f(x^{(i)};,\Theta),y^{(i)})$;
  Compute Hessian inverse: H^{-1};
  Compute update: $\triangle\Theta = -H^{-1}g$;
  Apply update: $\Theta = \Theta + \triangle\Theta$;
end

Something we can do to address this is to regularize the Hessian:

$$\Theta^* = \Theta_0 - [H(f(\Theta)) + \alpha I]^{-1}\nabla_\Theta f(\Theta_0)$$

This regularization allows us to use Newton's method on nonconvex functions, as long as any negative eigenvalues of H are close to 0.

Practical concerns. Newton's method is computationally complex (holding H is $O(|\Theta|^2)$ in size and inverting it is $O(|\Theta|^3)$. Variants of this allow us to sidestep this burden by avoiding computing H.

6.2 Conjugate gradients

We can avoid calculating H^{-1} by iteratively descending conjugate directions. We seek to find a search direction that is conjugate to the previous line search direction, that is to say, it will not undo progress made in that direction.

A search direction in the conjugate gradient methods is $d_t = \nabla_\Theta J(\Theta) + \beta_t d_{t-1}$. d_t and d_{t-1} are conjugate if $d_t^T H d_{t-1} = 0$

How do we compute β_t without computing H?

1. Fletcher-Reeves

$$\beta_t = \frac{\nabla_\Theta J(\Theta_t)^T \nabla_\Theta J(\Theta_t)}{\nabla_\Theta J(\Theta_{t-1})^T \nabla_\Theta J(\Theta_{t-1})}$$

2. Polak-Ribière

$$\beta_t = \frac{(\nabla_\Theta J(\Theta_t) - \nabla_{Theta} J(\Theta_{t-1}))^T \nabla_\Theta J(\Theta_t)}{\nabla_\Theta J(\Theta_{t-1})^T \nabla \Theta J(\Theta_{t-1})}$$

Algorithm 10: Conjugate gradient method

Require: Initial parameters Θ_0
Require: Training set of *m* examples
Initialize $\rho_0 = 0$;
Initialize $g_0 = 0$;
Initialize $t = 1$;
while *stopping criterion not met* **do**
  Initialize the gradient $g_t = 0$;
  Compute gradient: $g_t \leftarrow \frac{1}{m} \nabla_\Theta \sum_i L(f(x^{(i)}; \Theta), y^{(i)})$;
  Compute $\beta_t = \frac{(g_t - g_{t-1})^T g_t}{g_{t-1}^T g_{t-1}}$ (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: $\rho_t = -g_t + \beta_t \rho_{t-1}$;
  Perform line search to find: $\epsilon^* = \arg\min_\epsilon \frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}; \Theta_t + \epsilon \rho_t), y^{(i)})$;
  (On a truly quadratic cost function, analytically solve for ϵ^* rather than explicitly searching for it);
  Apply update: $\Theta_{t+1} = \Theta_t + \epsilon^* \rho_t$;
  $t \leftarrow t + 1$;
end

Using an approach like this, on a quadratic surface we ensure that we do not increase the gradient along the previous direction. This means that we stay at the minimum along previous directions, and that in a k-dimensional parameter space we require at most k line searches.

6.2.1 Nonlinear conjugate gradients

The method is largely applicable to complicated objective functions. With no assurance that the objective function is quadratic however, the conjugate directions are no longer assured to remain at the minimum. To account for this, we reset the method occasionally with a line search along the unaltered gradient.

Practical tips. It is often beneficial to to initialize the optimization with a few iterations of SGD.

Minibatch versions of such a method exist, as proposed by Le et al. [2011]. Scaled conjugate gradients, proposed specifically for neural networks have also been proposed Møller [1993].

6.3 BFGS Algorithm

The Broyden-Fletcher-Goldfarb-Shanno Algorithm takes a more direct approach to the approximation of the Newton's Method update than conjugate gradient.

Recall in Newton's method that $\Theta^* = \Theta_0 - H^{-1}\nabla_\Theta J(\Theta_0)$. In the BFGS algorithm, we approximate the inverse of a matrix M_t that is iteratively refined by low rank updates to $\approx H^{-1}$.

This defines a direction of descent $\rho_t = M_t g_t$. We then perform a line search to find ϵ^* s.t.

$$\Theta_{t+1} = \Theta_t + \epsilon^* \rho_t$$

BFGS spends less time refining each line search than Newton's method but has to store the matrix M, which is $O(n^2)$

6.3.1 Limited Memory (L-BFGS)

We can avoid storing M by assuming M_{t-1} is the identity matrix. If used with exact line searches, directions defined by L-BFGS are mutually conjugate. Two benefits of this approach are that it remains well behaved even when the minimum of each line search is approximate, and the memoryless version can be expanded to store some vectors used to update M at each timestep (which is $O(n)$.

References

Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14:8, 2012.

Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307, 1988.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Quoc V Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 265–272. Omnipress, 2011.

James Martens. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.

Martin Fodslette Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks*, 6(4):525–533, 1993.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

Tom Schaul, Ioannis Antonoglou, and David Silver. Unit tests for stochastic optimization. *arXiv preprint arXiv:1312.6055*, 2013.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.