ECBM E4040 Neural Networks and Deep Learning

Lecture: Optimization for Training Deep Models

Instructor: Zoran Kostic

Columbia University
Department of Electrical Engineering

Oct. 19th, 2016

Outline of Part I

Outline of Part II

Part I

Optimization - Part I

Empirical Risk Minimization

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

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

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

Solution: Minimize the expected loss on the training set.

Empirical Risk Minimization (cont'd)

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

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

where m is the number of training examples.

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

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

Batch and Minibatch Algorithms

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

Optimization algorithms that use only a single example at a time are sometimes called stochastic or sometimes online methods

Most algorithms used for deep learning fall somewhere in between, using more than one but less than all of the training examples. These were traditionally called minibatch or minibatch stochastic methods and it is now common to simply call them stochastic methods.

Batch and Minibatch Algorithms (cont'd)

Minibatch sizes are generally driven by the following factors:

- larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- multicore architectures are usually underutilized by extremely small batches; use some absolute minimum batch size.
- if batches are processed in parallel, then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor.
- on GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect. Generalization error is often best for a batch size of 1, though this might take a very long time to train and require a small learning rate to maintain stability.

Ill-Conditioning

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

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

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

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

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

Ill-Conditioning (cont'd)

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

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

Local Minima

Neural networks and any models with multiple equivalently parameterized latent variables all have multiple local minima because of the model identifiability problem. A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters.

Causes of non-identifiability:

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

Plateaus, Saddle Points and Other Flat Regions

Many classes of random functions exhibit the following behavior: in low-dimensional spaces, local minima are common. In higher dimensional spaces, local minima are rare and saddle points are exponentially more common.

To understand the intuition behind this, observe that a local minimum has only positive eigenvalues. A saddle point has a mixture of positive and negative eigenvalues. Imagine that the sign of each eigenvalue is generated by flipping a coin. In 1-dimensional space, it is easy to obtain a local minimum by tossing a coin and getting heads once. In n-dimensional space, it is exponentially unlikely that all n coin tosses will be heads.

Gradient-Based Learning Algorithms

The back-propagation algorithm (backprop) efficiently computes the gradient of the loss with respect to the model parameters. Backprop does not specify how we use this gradient to update the weights of the model.

Here, we describe/discuss a number of gradient-based learning algorithms that have been proposed to optimize the parameters of deep learning models.

The Gradient Descent Algorithm

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

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

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

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

The Gradient Descent Algorithm (cont'd)

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

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

The Stochastic Gradient Descent Algorithm

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

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

Batch gradient descent most often works with a fixed learning rate. To converge to a minimum, the learning rate of the SGD has to decrease at an appropriate rate during training. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not become 0 even when we arrive at a minimum (whereas the true gradient becomes small and then 0 when we approach and reach a minimum).

Algorithm 1: Stochastic Gradient Descent (SGD)

Algorithm 1 Stochastic gradient descent update at training iteration k

```
Require: Learning rate $\eta_k$.
Require: Initial parameter $\boldsymbol{\theta}$.
  while Stopping criterion not met do
      Sample a minibatch of $m$ examples from the training set $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
      Set $\hat{\mathbf{g}} = \mathbf{0}$.
      for $i = 1$ to $m$ do
          Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} + \frac{1}{m}\nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)$
      end for
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_k \hat{\mathbf{g}}$
  end while
```

Convergence of the SGD Algorithm

A sufficient condition to guarantee convergence is that

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

- stochastic gradient converges initially much faster than batch gradient descent; (many more stochastic updates can be performed for the price of one deterministic update)
- batch gradient descent (with larger and larger minibatches) will converge to lower values of the objective function, because of its faster rate; (after some number of iterations).

For large neural networks that are trained on large datasets, SGD remain the algorithm of choice. Training error can in principle be greatly improved by following SGD by a deterministic gradient-based optimization method, but that may be at the cost of worse generalization error.

Momentum

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

Formally, we introduce a variable $\mathbf{v}$ that plays the role of velocity (or momentum) that accumulates gradient. The update rule is given by:

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

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

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

Algorithm 2: Stochastic Gradient Descent (SGD)

Algorithm 2 Stochastic gradient descent update with momentum

```
Require: Learning rate $\eta$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial vector $\mathbf{v}$.
  while Stopping criterion not met do
      Sample a minibatch of $m$ examples from the training set
      $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
      Set $\mathbf{g} = \mathbf{0}$.
      for $i = 1$ to $m$ do
          Compute gradient estimate:
          $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$
      end for
      Compute velocity update: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \mathbf{g}$
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$
  end while
```

Nesterov Momentum

The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard momentum method. The update rule is given by:

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

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

Algorithm 3: Stochastic Gradient Descent (SGD)

Algorithm 3 Stochastic gradient descent with Nesterov momentum

```
Require: Learning rate $\eta$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial vector $\mathbf{v}$.
  while Stopping criterion not met do
      Sample a minibatch of $m$ examples from the training set
      $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
      Apply interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\mathbf{v}$
      Set $\mathbf{g} = \mathbf{0}$.
      for $i = 1$ to $m$ do
          Compute gradient (at interim point):
          $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$
      end for
      Compute velocity update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\mathbf{g}$
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$
  end while
```

The AdaGrad Algorithm

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

In the convex optimization context, the AdaGrad algorithm enjoys some desirable theoretical properties. However, empirically it has been found that for training deep neural network models the accumulation of squared gradients from the beginning of training results in a premature and excessive decrease in the effective learning rate.

Algorithm 4: The AdaGrad Algorithm

Algorithm 4 The AdaGrad Algorithm

```
Require: Global learning rate $\eta$.
Require: Initial parameter $\boldsymbol{\theta}$.
  Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$,
  while Stopping criterion not met do
      Sample a minibatch of $m$ training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
      Set $\mathbf{g} = \mathbf{0}$.
      for $i = 1$ to $m$ do
          Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$
      end for
      Accumulate gradient: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g}^2$ (square element-wise)
      Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\mathbf{r}}}$ element-wise)
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

The RMSprop Algorithm

The RMSprop algorithm addresses the deficiency of AdaGrad by changing the gradient accumulation into an exponentially weighted moving average. In deep networks, the optimization surface is far from convex. Directions in parameter space with strong partial derivatives early in training may flatten out as training progresses.

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

Algorithm 5: The RMSprop Algorithm

Algorithm 5 The RMSprop Algorithm

```
Require: Global learning rate $\eta$, decay rate $\rho$.
Require: Initial parameter $\boldsymbol{\theta}$.
  Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$,
  while Stopping criterion not met do
      Sample a minibatch of $m$ training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
      Set $\mathbf{g} = \mathbf{0}$.
      for $i = 1$ to $m$ do
          Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$
      end for
      Accumulate gradient: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\mathbf{g}^2$
      Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\mathbf{r}}}$ element-wise)
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

The RMSprop Algorithm with Nesterov Momentum

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

Empirically RMSprop has shown to be an effective and practical optimization algorithm for deep neural networks:

- it is easy to implement and relatively simple to use; there does not appear to be a great sensitivity to the algorithm's hyperparameters,
- it is currently one of the "go to" optimization methods being employed routinely by deep learning researchers.

Algorithm 6: RMSprop with Nesterov Momentum

Algorithm 6 The RMSprop Algorithm with Nesterov Momentum

```
Require: Global learning rate $\eta$, decay rate $\rho$, momentum coeff. $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\mathbf{v}$.
  Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$,
  while Stopping criterion not met do
      Sample a minibatch of $m$ training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
      Compute interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\mathbf{v}$
      Set $\mathbf{g} = \mathbf{0}$.
      for $i = 1$ to $m$ do
          Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$
      end for
      Accumulate gradient: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\mathbf{g}^2$
      Compute update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\mathbf{r}}}$ element-wise)
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$
  end while
```

The Adam Algorithm

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

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

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

Algorithm 7: The Adam Algorithm

Algorithm 7 The Adam Algorithm

```
Require: Step-size $\alpha$.
Require: Decay rates $\rho_1$ and $\rho_2$, constant $\epsilon$.
Require: Initial parameter $\boldsymbol{\theta}$.
    Initialize 1st and 2nd moment variables $\mathbf{s} = \mathbf{0}$, $\mathbf{r} = \mathbf{0}$.
    Initialize timestep $t = 0$.
    while Stopping criterion not met do
        Sample a minibatch of $m$ training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
        Set $\mathbf{g} = \mathbf{0}$.
        for $i = 1$ to $m$ do
            Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$
        end for
        $t \leftarrow t + 1$
```

Algorithm 7: The Adam Algorithm (cont'd)

Algorithm 7 The Adam Algorithm (cont'd)

```
        Get biased first moment: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\mathbf{g}$
        Get biased second moment: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g}^2$
        Compute bias-corrected first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}$
        Compute bias-corrected second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$
        Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\alpha \frac{\mathbf{s}}{\sqrt{\mathbf{r}}+\epsilon} \odot \mathbf{g}$ (element-wise)
        Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
    end while
```

The AdaDelta Algorithm

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

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

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

The AdaDelta Algorithm

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

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

or

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

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

Algorithm 8: The AdaDelta Algorithm

Algorithm 8 The AdaDelta Algorithm

```
Require: Decay rate $\rho$, constant $\epsilon$.
Require: Initial parameter $\boldsymbol{\theta}$.
  Initialize accumulation variables $\mathbf{r} = \mathbf{0}$, $\mathbf{s} = \mathbf{0}$.
  while Stopping criterion not met do
    Sample a minibatch of $m$ training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
    Set $\mathbf{g} = \mathbf{0}$.
    for $i = 1$ to $m$ do
      Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$
    end for
    Accumulate gradient: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\mathbf{g}^2$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\sqrt{\mathbf{s}+\epsilon}}{\sqrt{\mathbf{r}+\epsilon}} \odot \mathbf{g}$ (element-wise)
    Accumulate update: $\mathbf{s} \leftarrow \rho\mathbf{s} + (1-\rho)[\Delta\boldsymbol{\theta}]^2$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Choosing the Right Optimization Algorithm

A natural question is: which algorithm should one choose?
Unfortunately, there is currently no consensus on this point.
A comparison of a large number of optimization algorithms across a wide range of learning tasks suggests that

- the family of algorithms (represented by RMSprop and AdaDelta) performed fairly robustly; however, no single best algorithm has emerged,
- the most popular optimization algorithms actively in use include SGD, SGD+momentum, RMSprop, RMSprop+momentum, AdaDelta and Adam.
- the choice of which algorithm to use, as this point, seems to depend as much on the users familiarity with the algorithm (for ease of hyperparameter tuning) as it does on any established notion of superior performance.

Part II

Optimization for Training Deep Models - Part II

Overview

We'll investigate second-order methods to the training of deep networks by examining the empirical risk:

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

where m is the number of training examples and $\hat{p}(\mathbf{x}, y)$ is the empirical distribution.

Newton's Updating Algorithm

Given the objective function $J(\boldsymbol{\theta})$, the Hessian matrix of J with respect to $\boldsymbol{\theta}$ is defined by

$$[\mathbf{H}(J)(\boldsymbol{\theta})]_{ij} = \frac{\partial^2}{\partial\theta_i \partial\theta_j} J(\boldsymbol{\theta}).$$

Newton's method to be developed here is based on using a second-order Taylor series expansion to approximate $J(\boldsymbol{\theta})$ near some point $\boldsymbol{\theta}_0$, ignoring derivatives of higher order, i.e.,

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(J)(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0).$$

At the critical point we obtain the Newton parameter update rule:

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

If the objective function is convex but not quadratic, this update can be iterated (see Algorithm 1.)

Algorithm 1: Newton's method with objective $J(\boldsymbol{\theta})$

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

```
Require: Initial parameter $\boldsymbol{\theta}_0$.
  while Stopping criterion not met do
    Initialize the gradient $\mathbf{g} = \mathbf{0}$
    Initialize the Hessian $\mathbf{H} = \mathbf{0}$
    for i=1 to m do
      Compute gradient $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i;\boldsymbol{\theta}), y^i)/m$
      Compute Hessian $\mathbf{H} \leftarrow \mathbf{H} + \nabla^2_{\boldsymbol{\theta}} L(f(\mathbf{x}^i;\boldsymbol{\theta}), y^i)/m$
    end for
    Compute Hessian inverse: $\mathbf{H}^{-1}$
    Compute update: $\Delta\boldsymbol{\theta} = \mathbf{H}^{-1}\mathbf{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \Delta\boldsymbol{\theta}$
  end while
```

Intuition Behind the Newton Algorithm

Change of Variables

The Hessian can be interpreted as a transformation from the Euclidean space where the problem is defined to a space where gradient optimization is simplified. Since $\mathbf{H}$ is a real symmetric matrix, it can be represented as $\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$, where $\mathbf{Q}$ is an orthogonal matrix.

We consider the alternative parametrization

$$\boldsymbol{\phi} = \mathbf{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta}$$

and note that by the chain rule

$$\nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0) = \left[\frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{\phi}}\right]^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) = \mathbf{Q}\mathbf{\Lambda}^{-1/2}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0).$$

or equivalently, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) = \mathbf{\Lambda}^{1/2}\mathbf{Q}^T\nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0)$.

Intuition Behind the Newton Algorithm (cont'd)

Change of Variables

$$
\begin{aligned}
J(\boldsymbol{\theta}) &= J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(J)(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \\
&= J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{Q}\boldsymbol{\Lambda}^{1/2} \nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^T(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \\
&= J(\boldsymbol{\theta}_0) + (\boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta} - \boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0) + \\
&\qquad + \frac{1}{2}(\boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta} - \boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta}_0)^T(\boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta} - \boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\boldsymbol{\theta}_0) \\
&= J(\boldsymbol{\theta}_0) + (\boldsymbol{\phi} - \boldsymbol{\phi}_0)^T \nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\phi} - \boldsymbol{\phi}_0)^T(\boldsymbol{\phi} - \boldsymbol{\phi}_0).
\end{aligned}
$$

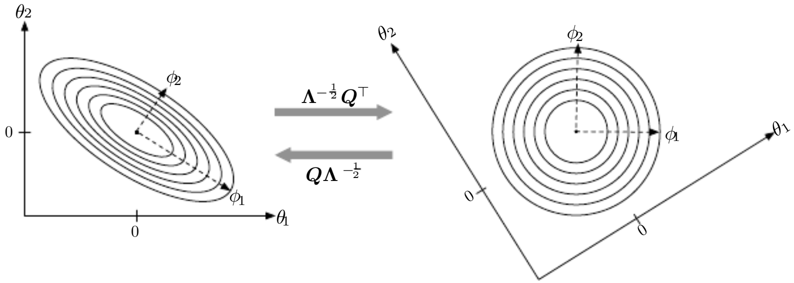
Computing the gradient with respect to $\boldsymbol{\phi}$ and setting this to zero yields the Newton update in $\boldsymbol{\phi}$-space:

$$\boldsymbol{\phi}^* = \boldsymbol{\phi}_0 - \nabla_{\boldsymbol{\phi}} J(\boldsymbol{\theta}_0).$$

That is, in $\boldsymbol{\phi}$-space, the Newton update is transformed into a standard gradient step with unit learning rate.

Intuition Behind the Newton Algorithm (cont'd)

Change of Variables



Newton's method maps an arbitrary and possibly ill-conditioned quadratic objective function in parameter space $\boldsymbol{\theta}$ into an alternative parameter space ϕ where the quadratic objective function is isometric.

Regularizing the Hessian

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

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

The algorithm works fairly well as long as the negative eigenvalues of the Hessian are still relatively close to zero. In cases where there are more extreme directions of curvature, the value of α would have to be sufficiently large to offset the negative eigenvalues. However, as α increases in size, the Hessian becomes dominated by the $\alpha\mathbf{I}$ diagonal and the direction chosen by Newton's method converges to the standard gradient.

Limitations of Newton's Method

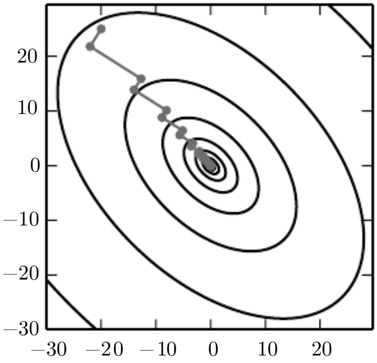
Newton's method for training large neural networks is limited by the significant computational burden it imposes as the inverse Hessian has to be computed at every training iteration.

Newton's method would require the inversion of a $K \times K$ matrix - with computational complexity of $\mathcal{O}(K^3)$.

As a consequence, only networks with very small number of parameters can be practically trained via Newton's method.

The Method of Conjugate Gradients

In the method of steepest descent, line searches are applied iteratively in the direction associated with the gradient.



When applied in a quadratic bowl, steepest descent progresses in a rather ineffective back-and-forth, zig-zag pattern as each line search direction (the gradient) is guaranteed to be orthogonal to the previous line search direction.

Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending conjugate directions.

The Method of Conjugate Gradients (cont'd)

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 $\mathbf{d}_t$ takes the form:

$$\mathbf{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \beta_t \mathbf{d}_{t-1}$$

were β_t is a coefficient whose magnitude controls how much of the direction, $\mathbf{d}_{t-1}$, we should add back to the current search direction.

Two directions, $\mathbf{d}_t$ and $\mathbf{d}_{t-1}$, are defined as conjugate if

$$\mathbf{d}_t^T \mathbf{H}(J) \mathbf{d}_{t-1} = 0.$$

The Method of Conjugate Gradients (cont'd)

We have

$$\begin{aligned} \mathbf{d}_t^T \mathbf{H}\, \mathbf{d}_{t-1} &= 0 \\ \mathbf{d}_t^T \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^T\, \mathbf{d}_{t-1} &= 0 \\ \left(\boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\, \mathbf{d}_t\right)^T \left(\boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\, \mathbf{d}_{t-1}\right) &= 0 \\ \left(\mathbf{d}_t^{\phi}\right)^T \mathbf{d}_{t-1}^{\phi} &= 0, \end{aligned}$$

where $\mathbf{d}_t^{\phi} = \boldsymbol{\Lambda}^{1/2}\mathbf{Q}^T\, \mathbf{d}_t$ as the direction $\mathbf{d}_t$ transformed to ϕ-space. The method of conjugate gradients can be interpreted as the application of the method of steepest descent in ϕ-space. The conjugate directions can be computed using the eigen-decomposition of the Hessian, $\mathbf{H}$.

The Method of Conjugate Gradients (cont'd)

Can we calculate the conjugate directions without computing the Hessian, its inverse or its decomposition?

1 Fletcher-Reeves:

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

2 Polak-Ribiere:

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

Algorithm 2: Conjugate Gradient Method

Algorithm 10 Conjugate Gradient Method

```
Require: Initial parameter $\boldsymbol{\theta}_0$.
  Initialize $\boldsymbol{\rho}_0 = \mathbf{0}$
  while Stopping criterion not met do
      Initialize the gradient $\mathbf{g}_t = \mathbf{0}$
      for i=1 to m do % loop over the training set.
          Compute gradient $\mathbf{g}_t \leftarrow \mathbf{g}_t + \frac{1}{m}\nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), y^i)$
      end for
      Compute: $\beta_t = \frac{(\mathbf{g}_t - \mathbf{g}_{t-1})^T \mathbf{g}_t}{\mathbf{g}_{t-1}^T \mathbf{g}_{t-1}}$ (Polak - Ribière)
      Compute search directions: $\boldsymbol{\rho}_t = -\mathbf{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$
      Perform line search to find:
      $\eta^* = \mathrm{argmin}_\eta \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^i; \boldsymbol{\theta}), y^i)$
      Apply update: $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$
  end while
```

The Broyden - Fletcher - Goldfarb - Shanno Algorithm

The Newton algorithm

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

can be re-written as

$$\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t = -\mathbf{H}^{-1} \left(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t+1}) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) \right).$$

The BFGS method is based on approximating the inverse of the Hessian with a matrix $\mathbf{M}$ that is updated as

$$\mathbf{M}_t = \mathbf{M}_{t-1} + (1 + \frac{\boldsymbol{\phi}^T \mathbf{M}_{t-1} \boldsymbol{\phi}}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}) \frac{\boldsymbol{\phi}\boldsymbol{\phi}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}} - \frac{\boldsymbol{\Delta}\boldsymbol{\phi}^T \mathbf{M}_{t-1} + \mathbf{M}_{t-1} \boldsymbol{\phi} \boldsymbol{\Delta}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}},$$

where $\boldsymbol{\phi} = \mathbf{g}_t - \mathbf{g}_{t-1}$, $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ and $\boldsymbol{\Delta} = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$.

Algorithm 3: The BFGS Method

Algorithm 11 The BFGS Method

```
Require: Initial parameter $\boldsymbol{\theta}_0$.
  Initialize inverse Hessian $\mathbf{M}_0 = \mathbf{I}$
  while Stopping criterion not met do
      Compute gradient $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ (via batch back-prop)
      Compute $\boldsymbol{\phi} = \mathbf{g}_t - \mathbf{g}_{t-1}$, $\boldsymbol{\Delta} = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$
      Approx $\mathbf{H}^{-1}$: $\mathbf{M}_t = \mathbf{M}_{t-1} + (1 + \frac{\boldsymbol{\phi}^T \mathbf{M}_{t-1} \boldsymbol{\phi}}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}) \frac{\boldsymbol{\phi}\boldsymbol{\phi}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}} - - \frac{\boldsymbol{\Delta}\boldsymbol{\phi}^T \mathbf{M}_{t-1} + \mathbf{M}_{t-1} \boldsymbol{\phi} \boldsymbol{\Delta}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}$
      Compute search directions: $\boldsymbol{\rho}_t = \mathbf{M}_t \mathbf{g}_t$
      Perform line search to find: $\eta^* = \text{argmin}_\eta \, J(\boldsymbol{\theta}_t + \eta \boldsymbol{\rho}_t)$
      Apply update: $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$
  end while
```

Limited Memory BFGS

The BFGS algorithm must store the inverse Hessian matrix, $\mathbf{M}$, that requires $\mathcal{O}(n^2)$ memory, making BFGS impractical for most modern deep learning models that typically have millions of parameters. We set $\mathbf{M}_{t-1} = \mathbf{I}$ to obtain:

$$\mathbf{M}_t = \mathbf{I} + (1 + \frac{\boldsymbol{\phi}^T\boldsymbol{\phi}}{\boldsymbol{\Delta}^T\boldsymbol{\phi}})\frac{\boldsymbol{\phi}\boldsymbol{\phi}^T}{\boldsymbol{\Delta}^T\boldsymbol{\phi}} - \frac{\boldsymbol{\Delta}\boldsymbol{\phi}^T + \boldsymbol{\phi}\boldsymbol{\Delta}^T}{\boldsymbol{\Delta}^T\boldsymbol{\phi}}.$$

Therefore, the direction update formula becomes

$$\boldsymbol{\rho}_t = \mathbf{g}_t + a\boldsymbol{\phi} + b\boldsymbol{\Delta},$$

where the scalars a and b are given by

$$a = (1 + \frac{\boldsymbol{\phi}^T\boldsymbol{\phi}}{\boldsymbol{\Delta}^T\boldsymbol{\phi}})\frac{\boldsymbol{\phi}^T\mathbf{g}_t}{\boldsymbol{\Delta}^T\boldsymbol{\phi}} - \frac{\boldsymbol{\Delta}^T\mathbf{g}_t}{\boldsymbol{\Delta}^T\boldsymbol{\phi}} \quad \text{and} \quad b = -\frac{\boldsymbol{\phi}^T\mathbf{g}_t}{\boldsymbol{\Delta}^T\boldsymbol{\phi}}$$

Coordinate Descent

Coordinate descent is a methodology of optimizing along one coordinate at a time. Block coordinate descent refers to minimizing with respect to a subset of the variables simultaneously.

Divide and conquer: different variables in the optimization problem are separated into groups that play relatively isolated roles. For example, consider the cost function

$$J(\mathbf{X}, \mathbf{W}) = \sum_{ij} |H_{ij}| + \sum_{ij} \left(\mathbf{X} - \mathbf{W}^T \mathbf{H}\right)^2_{ij} .$$

This function describes a learning problem called sparse coding, where the goal is to find a weight matrix $\mathbf{W}$ that can linearly decode a matrix of activation values $\mathbf{H}$ to reconstruct the training set $\mathbf{X}$.

Coordinate Descent (cont'd)

In full generality finding $\min J$ is hard because J is not convex.

However, we can divide the inputs to the training algorithm into two sets: the dictionary parameters $\mathbf{W}$ and the code representations $\mathbf{H}$. Minimizing the objective function with respect to either one of these sets of variables is a convex problem.

Block coordinate descent thus gives us an optimization strategy that allows us to use efficient convex optimization algorithms.

Initialization Strategies

Training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether

- the algorithm converges at all,
- they are unstable and, thereby, the algorithm encounters numerical difficulties,
- the algorithm fails altogether.

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

Initialization Strategies (cont'd)

The only property known with complete certainty is that the initial parameters need to break symmetry between different units:

- if two units have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way;
- if the model or training algorithm is capable of using stochasticity to compute different updates for different units, it is usually best to initialize each unit to compute a different function from all of the other units.

Weights are usually initialized to values drawn from a Gaussian or uniform distribution. The choice of the distribution does not seem to matter; the scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize.

Initialization Strategies (cont'd)

Regularization and optimization can give very different insights into how we should initialize a network:

- optimization suggests weights that are large enough to propagate information successfully,
- regularization encourages making weigths smaller.

Initializing the parameters $\boldsymbol{\theta}$ to $\boldsymbol{\theta}_0$ is similar to imposing a Gaussian prior $p(\boldsymbol{\theta})$ with mean $\boldsymbol{\theta}_0$. The prior implies that it is more likely that units do not interact with each other than that they do interact.

Units interact only if the likelihood term of the objective function expresses a strong preference for them to interact. If we initialize $\boldsymbol{\theta}_0$ to large values, then our prior specifies which units should interact with each other, and in pre-specified ways.