Neural Networks: Optimization Part 1

Intro to Deep Learning, Fall 2020
Story so far

• Neural networks are universal approximators
  – Can model any odd thing
  – Provided they have the right architecture
• We must *train* them to approximate any function
  – Specify the architecture
  – Learn their weights and biases
• Networks are trained to minimize total “loss” on a training set
  – We do so through empirical risk minimization
• We use variants of gradient descent to do so
• The gradient of the error with respect to network parameters is computed through backpropagation
Recap: Gradient Descent Algorithm

- In order to minimize any function $f(x)$ w.r.t. $x$

- Initialize:
  - $x^0$
  - $k = 0$

- Do
  - $k = k + 1$
  - $x^{k+1} = x^k - \eta \nabla_x f^T$

- while $|f(x^k) - f(x^{k-1})| > \varepsilon$
Recap: Training Neural Nets by Gradient Descent

Total training error:

\[ \text{Loss} = \frac{1}{T} \sum_t \text{Div}(Y_t, d_t; W_1, W_2, ..., W_K) \]

- Gradient descent algorithm:
- Initialize all weights \( W_1, W_2, ..., W_K \)
- Do:
  - For every layer \( k = 1 \ldots K \) compute:
    - \( \nabla_{W_k} \text{Loss} = \frac{1}{T} \sum_t \nabla_{W_k} \text{Div}(Y_t, d_t) \)
    - \( W_k = W_k - \eta \nabla_{W_k} \text{Loss}^T \)
- Until \( \text{Loss} \) has converged
Recap: Training Neural Nets by Gradient Descent

Total training error:

\[
Loss = \frac{1}{T} \sum_t \text{Div}(Y_t, d_t; W_1, W_2, ..., W_K)
\]

• Gradient descent algorithm:
  • Initialize all weights \( W_1, W_2, ..., W_K \)
  • Do:
    – For every layer \( k \), compute:
      • \( \nabla_{W_k} Loss = \frac{1}{T} \sum_t \nabla_{W_k} \text{Div}(Y_t, d_t) \)
      • \( W_k = W_k - \eta \nabla_{W_k} Loss^T \)
  • Until \( Loss \) has converged
Issues

• Convergence: How well does it learn
  – And how can we improve it
• How well will it generalize (outside training data)
• What does the output really mean?
• *Etc.*
Onward
Onward

• Does backprop always work?
• Convergence of gradient descent
  – Rates, restrictions,
  – Hessians
  – Acceleration and Nestorov
  – Alternate approaches
• Modifying the approach: Stochastic gradients
• Speedup extensions: RMSprop, Adagrad
Does backprop do the right thing?

• Is backprop always right?
  – Assuming it actually finds the minimum of the divergence function?
Recap: The differentiable activation

- Threshold activation: Equivalent to counting errors
  - Shifting the threshold from $T_1$ to $T_2$ does not change classification error
  - Does not indicate if moving the threshold left was good or not

- Differentiable activation: Computes “distance to answer”
  - “Distance” == divergence
  - Perturbing the function changes this quantity,
    - Even if the classification error itself doesn’t change
Does backprop do the right thing?

• **Is backprop always right?**
  – Assuming it actually finds the global minimum of the divergence function?

• In classification problems, the classification error is a non-differentiable function of weights

• The divergence function minimized is only a proxy for classification error

• Minimizing divergence may not minimize classification error
Backprop fails to separate where perceptron succeeds

- Brady, Raghavan, Slawny, ’89
- Simple problem, 3 training instances, single neuron
- Perceptron training rule trivially find a perfect solution
Backprop vs. Perceptron

- Back propagation using logistic function and $L_2$ divergence ($Div = (y - d)^2$)
- Unique minimum trivially proved to exist, backprop finds it
Unique solution exists

• Let $u = f^{-1}(1 - \varepsilon)$
  – E.g. $u = f^{-1}(0.99)$ representing a 99% confidence in the class

• From the three points we get three independent equations:

  \[ w_x \cdot 1 + w_y \cdot 0 + b = u \]
  \[ w_x \cdot 0 + w_y \cdot 1 + b = u \]
  \[ w_x \cdot -1 + w_y \cdot 0 + b = -u \]

• Unique solution \((w_x = u, w_x = u, b = 0)\) exists
  – represents a unique line regardless of the value of \(u\)
- Now add a fourth point
- $t$ is very large (point near $-\infty$)
- Perceptron trivially finds a solution (may take $t^2$ iterations)
Consider backprop:

**Contribution of fourth point to derivative of $L_2$ error:**

$$div_4 = \left(1 - \varepsilon - \sigma(-w_yt + b)\right)^2$$

$$\frac{d \, div_4}{dw_y} = 2 \left(1 - \varepsilon - \sigma(-w_yt + b)\right) \sigma'(-w_yt + b)t$$

$$\frac{d \, div_4}{db} = -2 \left(1 - \varepsilon - \sigma(-w_yt + b)\right) \sigma'(-w_yt + b)$$
Backprop

Notation:
\( y = \sigma(z) = \text{logistic activation} \)
\[ d_4 = \left(1 - \varepsilon - \sigma(-w_y t + b)\right)^2 \]
\[
\frac{d}{d w_y} d_4 = 2 \left(1 - \varepsilon - \sigma(-w_y t + b)\right) \sigma'(-w_y t + b) t
\]
\[
\frac{d}{d b} d_4 = 2 \left(1 - \sigma(-w_y t + b)\right) \sigma'(-w_y t + b) t
\]

- For very large positive \( t, |w_y| > \varepsilon \) (where \( w = [w_x, w_y, b] \))
- \( \left(1 - \varepsilon - \sigma(-w_y t + b)\right) \to 1 \) as \( t \to \infty \)
- \( \sigma'(-w_y t + b) \to 0 \) exponentially as \( t \to \infty \)
- Therefore, for very large positive \( t \)

\[
\frac{d}{d w_y} d_4 = \frac{d}{d b} d_4 = 0
\]
• The fourth point at \( (0, -t) \) does not change the gradient of the \( L_2 \) divergence near the optimal solution for 3 points.
• The optimum solution for 3 points is also a broad local minimum (0 gradient) for the 4-point problem!
  – Will be found by backprop nearly all the time
    • Although the global minimum with unbounded weights will separate the classes correctly.
• Local optimum solution found by backprop

• Does not separate the points *even though the points are linearly separable!*
• Solution found by backprop
• Does not separate the points *even though the points are linearly separable!*
• Compare to the perceptron: *Backpropagation fails to separate where the perceptron succeeds*
Backprop fails to separate where perceptron succeeds

- Brady, Raghavan, Slawny, ’89
- Several linearly separable training examples
- Simple setup: both backprop and perceptron algorithms find solutions
A more complex problem

- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
A more complex problem

• Adding a “spoiler” (or a small number of spoilers)
  – Perceptron finds the linear separator,
  – Backprop does not find a separator
    • A single additional input does not change the loss function significantly
      – Assuming weights are constrained to be bounded
A more complex problem

- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - For bounded $w$, backprop does not find a separator
    - A single additional input does not change the loss function significantly
A more complex problem

- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - For bounded $w$, backprop does not find a separator
    - A single additional input does not change the loss function significantly
A more complex problem

- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - For bounded $w$, backprop does not find a separator
    - A single additional input does not change the loss function significantly
So what is happening here?

• The perceptron may change greatly upon adding just a single new training instance
  – But it fits the training data well
  – The perceptron rule has *low bias*
    • Makes no errors if possible
  – But high variance
    • Swings wildly in response to small changes to input

• Backprop is minimally changed by new training instances
  – Prefers consistency over perfection
  – It is a *low-variance* estimator, at the potential cost of bias
Backprop fails to separate even when possible

- This is not restricted to single perceptrons
- An MLP learns non-linear decision boundaries that are determined from the entirety of the training data
- Adding a few “spoilers” will not change their behavior
Backprop fails to separate even when possible

- This is not restricted to single perceptrons
- An MLP learns non-linear decision boundaries that are determined from the entirety of the training data
- Adding a few “spoilers” will not change their behavior
**Backpropagation: Finding the separator**

- Backpropagation will often not find a separating solution *even though the solution is within the class of functions learnable by the network*

- This is because the separating solution is not a feasible optimum for the loss function

- One resulting benefit is that a backprop-trained neural network classifier has lower variance than an optimal classifier for the training data
Variance and Depth

- Dark figures show desired decision boundary (2D)
  - 1000 training points, 660 hidden neurons
  - Network heavily overdesigned even for shallow nets

- **Anecdotal: Variance decreases with**
  - Depth
  - Data

10000 training instances
The Loss Surface

• The example (and statements) earlier assumed the loss objective had a single global optimum that could be found
  – Statement about variance is assuming global optimum

• What about local optima
The Loss Surface

• **Popular hypothesis:**
  – In large networks, saddle points are far more common than local minima
    • Frequency of occurrence exponential in network size
  – Most local minima are equivalent
    • And close to global minimum
  – This is not true for small networks

• **Saddle point:** A point where
  – The slope is zero
  – The surface increases in some directions, but decreases in others
    • Some of the Eigenvalues of the Hessian are positive; others are negative
  – Gradient descent algorithms often get “stuck” in saddle points
The Controversial Loss Surface

• **Baldi and Hornik (89),** “Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima” : An MLP with a single hidden layer has only saddle points and no local Minima

• **Dauphin et. al (2015),** “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization” : An exponential number of saddle points in large networks

• **Chomoranksa et. al (2015),** “The loss surface of multilayer networks” : For large networks, most local minima lie in a band and are equivalent
  – Based on analysis of spin glass models

• **Swirszc et. al. (2016),** “Local minima in training of deep networks”, In networks of finite size, trained on finite data, you can have horrible local minima

• Watch this space...
Story so far

• Neural nets can be trained via gradient descent that minimizes a loss function

• Backpropagation can be used to derive the derivatives of the loss

• Backprop is not guaranteed to find a “true” solution, even if it exists, and lies within the capacity of the network to model
  – The optimum for the loss function may not be the “true” solution

• For large networks, the loss function may have a large number of unpleasant saddle points
  – Which backpropagation may find
Convergence

• In the discussion so far we have assumed the training arrives at a local minimum

• Does it always converge?
• How long does it take?

• Hard to analyze for an MLP, but we can look at the problem through the lens of convex optimization
A quick tour of (convex) optimization

The streetlight effect is a type of observational bias where people only look for whatever they are searching by looking where it is easiest.

“I’m searching for my keys.”
Convex Loss Functions

• A surface is “convex” if it is continuously curving upward
  – We can connect any two points on or above the surface without intersecting it
  – Many mathematical definitions that are equivalent

• Caveat: Neural network loss surface is generally not convex
  – Streetlight effect
Convergence of gradient descent

• An iterative algorithm is said to converge to a solution if the value updates arrive at a fixed point
  – Where the gradient is 0 and further updates do not change the estimate

• The algorithm may not actually converge
  – It may jitter around the local minimum
  – It may even diverge

• Conditions for convergence?
Convergence and convergence rate

• Convergence rate: How fast the iterations arrive at the solution
• Generally quantified as

\[
R = \frac{|f(x^{(k+1)}) - f(x^*)|}{|f(x^{(k)}) - f(x^*)|}
\]

– \(x^{(k+1)}\) is the k-th iteration
– \(x^*\) is the optimal value of \(x\)

• If \(R\) is a constant (or upper bounded), the convergence is *linear*
  – In reality, its arriving at the solution exponentially fast

\[
|f(x^{(k)}) - f(x^*)| \leq R^k |f(x^{(0)}) - f(x^*)|
\]
Convergence for quadratic surfaces

Minimize $E = \frac{1}{2} aw^2 + bw + c$

$w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw}$

- Gradient descent to find the optimum of a quadratic, starting from $w^{(k)}$
- Assuming fixed step size $\eta$
- What is the optimal step size $\eta$ to get there fastest?

Gradient descent with fixed step size $\eta$ to estimate scalar parameter $w$
Convergence for quadratic surfaces

- Any quadratic objective can be written as:
  \[ E(w) = E(w^{(k)}) + E'(w^{(k)})(w - w^{(k)}) + \frac{1}{2} E''(w^{(k)})(w - w^{(k)})^2 \]
  - Taylor expansion

- Minimizing w.r.t \( w \), we get (Newton’s method):
  \[ w_{min} = w^{(k)} - E''(w^{(k)})^{-1} E'(w^{(k)}) \]

- Note:
  \[ \frac{dE(w^{(k)})}{dw} = E'(w^{(k)}) \]

- Comparing to the gradient descent rule, we see that we can arrive at the optimum in a single step using the optimum step size:
  \[ \eta_{opt} = E''(w^{(k)})^{-1} = \alpha^{-1} \]
With non-optimal step size

\[ w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw} \]

Gradient descent with fixed step size \( \eta \) to estimate scalar parameter \( w \)

- For \( \eta < \eta_{opt} \) the algorithm will converge monotonically
- For \( 2\eta_{opt} > \eta > \eta_{opt} \) we have oscillating convergence
- For \( \eta > 2\eta_{opt} \) we get divergence
For generic differentiable convex objectives

- Any differentiable convex objective $E(w)$ can be approximated as

  $$E \approx E(w^{(k)}) + (w - w^{(k)}) \frac{dE(w^{(k)})}{dw} + \frac{1}{2} (w - w^{(k)})^2 \frac{d^2E(w^{(k)})}{dw^2} + \ldots$$

  — Taylor expansion

- Using the same logic as before, we get (Newton’s method)

  $$\eta_{opt} = \left(\frac{d^2E(w^{(k)})}{dw^2}\right)^{-1}$$

- We can get divergence if $\eta \geq 2\eta_{opt}$
For functions of multivariate inputs

\[ E = g(w), \ w \text{ is a vector } w = [w_1, w_2, ..., w_N] \]

• Consider a simple quadratic convex (paraboloid) function

\[ E = \frac{1}{2} w^T A w + w^T b + c \]

  – Since \( E^T = E \) (\( E \) is scalar), \( A \) can always be made symmetric
    • For convex \( E \), \( A \) is always positive definite, and has positive eigenvalues

• When \( A \) is diagonal:

\[ E = \frac{1}{2} \sum_i (a_{ii} w_i^2 + b_i w_i) + c \]

  – The \( w_i \)s are uncoupled
  – For convex (paraboloid) \( E \), the \( a_{ii} \) values are all positive
  – Just a sum of \( N \) independent quadratic functions
Multivariate Quadratic with Diagonal $A$

\[
E = \frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w} + \mathbf{w}^T \mathbf{b} + c = \frac{1}{2} \sum_i (a_{ii}w_i^2 + b_iw_i) + c
\]

- Equal-value contours will ellipses with principal axes parallel to the spatial axes
Multivariate Quadratic with Diagonal $A$

\[
E = \frac{1}{2} w^T A w + w^T b + c = \frac{1}{2} \sum_i (a_{ii} w_i^2 + b_i w_i) + c
\]

- Equal-value contours will be parallel to the axes
  - All “slices” parallel to an axis are shifted versions of one another

\[
E = \frac{1}{2} a_{ii} w_i^2 + b_i w_i + c + C(\neg w_i)
\]
Multivariate Quadratic with Diagonal $A$

$E = \frac{1}{2} w^T A w + w^T b + c = \frac{1}{2} \sum_i (a_{ii} w_i^2 + b_i w_i) + c$

- Equal-value contours will be parallel to the axis
  - All “slices” parallel to an axis are shifted versions of one another

$E = \frac{1}{2} a_{ii} w_i^2 + b_i w_i + c + C(\neg w_i)$
"Descents" are uncoupled

- The optimum of each coordinate is not affected by the other coordinates
  - I.e. we could optimize each coordinate independently
- Note: Optimal learning rate is different for the different coordinates

\[
E = \frac{1}{2} a_{11} w_1^2 + b_1 w_1 + c + C(\neg w_1) \\
\eta_{1, opt} = a_{11}^{-1} \\
E = \frac{1}{2} a_{22} w_2^2 + b_2 w_2 + c + C(\neg w_2) \\
\eta_{2, opt} = a_{22}^{-1}
\]
Vector update rule

\[
\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla_w E^T
\]

\[
\mathbf{w}_i^{(k+1)} = \mathbf{w}_i^{(k)} - \eta \frac{\partial E \left( \mathbf{w}_i^{(k)} \right)}{\partial \mathbf{w}}
\]

- Conventional vector update rules for gradient descent: update entire vector against direction of gradient
  - Note: Gradient is perpendicular to equal value contour
  - The same learning rate is applied to all components
Problem with vector update rule

\[ \mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla_{\mathbf{w}} E^T \]

\[ w_i^{(k+1)} = w_i^{(k)} - \eta \frac{\partial E \left( w_i^{(k)} \right)}{\partial \mathbf{w}} \]

\[ \eta_{i, opt} = \left( \frac{\partial^2 E \left( w_i^{(k)} \right)}{\partial w_i^2} \right)^{-1} = a_{ii}^{-1} \]

- The learning rate must be lower than twice the *smallest* optimal learning rate for any component

\[ \eta < 2 \min_i \eta_{i, opt} \]

- Otherwise the learning will diverge

- This, however, makes the learning very slow

- And will oscillate in all directions where \( \eta_{i, opt} \leq \eta < 2\eta_{i, opt} \)
Dependence on learning rate

• \( \eta_{1, opt} = 1; \eta_{2, opt} = 0.33 \)
• \( \eta = 2.1\eta_{2, opt} \)
• \( \eta = 2\eta_{2, opt} \)
• \( \eta = 1.5\eta_{2, opt} \)
• \( \eta = \eta_{2, opt} \)
• \( \eta = 0.75\eta_{2, opt} \)
Dependence on learning rate

- $\eta_{1,\text{opt}} = 1; \eta_{2,\text{opt}} = 0.91$; $\eta = 1.9 \eta_{2,\text{opt}}$
Convergence

• Convergence behaviors become increasingly unpredictable as dimensions increase.

• For the fastest convergence, ideally, the learning rate $\eta$ must be close to both, the largest $\eta_{i,\text{opt}}$ and the smallest $\eta_{i,\text{opt}}$.
  – To ensure convergence in every direction.
  – Generally infeasible.

• Convergence is particularly slow if $\frac{\max_i \eta_{i,\text{opt}}}{\min_i \eta_{i,\text{opt}}}$ is large.
  – The “condition” number is small.
Comments on the quadratic

• Why are we talking about quadratics?
  – Quadratic functions form some kind of benchmark
  – Convergence of gradient descent is linear
    • Meaning it converges to solution exponentially fast

• The convergence for other kinds of functions can be viewed against this benchmark

• Actual losses will not be quadratic, but may locally have other structure
  – Local between current location and nearest local minimum

• Some examples in the following slides..
  – Strong convexity
  – Lifschitz continuity
  – Lifschitz smoothness

  – ..and how they affect convergence of gradient descent
Quadratic convexity

- A quadratic function has the form $\frac{1}{2} w^T A w + w^T b + c$
  - Every “slice” is a quadratic bowl
- In some sense, the “standard” for gradient-descent based optimization
  - Others convex functions will be steeper in some regions, but flatter in others
- Gradient descent solution will have linear convergence
  - Take $O(\log 1/\varepsilon)$ steps to get within $\varepsilon$ of the optimal solution
A strongly convex function is \textit{at least} quadratic in its convexity
  
  - Has a lower bound to its second derivative

The function sits within a quadratic bowl
  
  - At any location, you can draw a quadratic bowl of fixed convexity (quadratic constant equal to lower bound of 2\textsuperscript{nd} derivative) touching the function at that point, which contains it

Convergence of gradient descent algorithms at least as good as that of the enclosing quadratic
Strong convexity

• A strongly convex function is *at least* quadratic in its convexity
  – Has a lower bound to its second derivative

• The function sits within a quadratic bowl
  – At any location, you can draw a quadratic bowl of fixed convexity (quadratic constant equal to lower bound of 2\textsuperscript{nd} derivative) touching the function at that point, which contains it

• Convergence of gradient descent algorithms at least as good as that of the enclosing quadratic
Types of continuity

• Most functions are not strongly convex (if they are convex)
• Instead we will talk in terms of Lifschitz smoothness
• But first: a definition

• **Lifschitz continuous**: The function always lies outside a cone
  – The slope of the outer surface is the Lifschitz constant
  – $|f(x) - f(y)| \leq L|x - y|$
Lifschitz smoothness

- Lifschitz smooth: The function’s derivative is Lifschitz continuous
  - Need not be convex (or even differentiable)
  - Has an upper bound on second derivative (if it exists)
- Can always place a quadratic bowl of a fixed curvature within the function
  - Minimum curvature of quadratic must be $\geq$ upper bound of second derivative of function (if it exists)
• Lifschitz smooth: The function’s derivative is Lifschitz continuous
  – Need not be convex (or even differentiable)
  – Has an upper bound on second derivative (if it exists)
• Can always place a quadratic bowl of a fixed curvature within the function
  – Minimum curvature of quadratic must be >= upper bound of second derivative of function (if it exists)
Types of smoothness

• A function can be both strongly convex and Lipschitz smooth
  – Second derivative has upper and lower bounds
  – Convergence depends on curvature of strong convexity (at least linear)

• A function can be convex and Lipschitz smooth, but not strongly convex
  – Convex, but upper bound on second derivative
  – Weaker convergence guarantees, if any (at best linear)
  – This is often a reasonable assumption for the local structure of your loss function
Types of smoothness

• A function can be both strongly convex and Lipschitz smooth
  – Second derivative has upper and lower bounds
  – Convergence depends on curvature of strong convexity (at least linear)

• A function can be convex and Lipschitz smooth, but not strongly convex
  – Convex, but upper bound on second derivative
  – Weaker convergence guarantees, if any (at best linear)
  – This is often a reasonable assumption for the local structure of your loss function
Convergence Problems

• For quadratic (strongly) convex functions, gradient descent is exponentially fast
  – Linear convergence
    • Assuming learning rate is non-divergent

• For generic (Lifschitz Smooth) convex functions however, it is very slow
  \[ |f(w^{(k)}) - f(w^*)| \propto \frac{1}{k} |f(w^{(0)}) - f(w^*)| \]
  – And inversely proportional to learning rate
  \[ |f(w^{(k)}) - f(w^*)| \leq \frac{1}{2\eta k} |w^{(0)} - w^*| \]
  – Takes $O(1/\epsilon)$ iterations to get to within $\epsilon$ of the solution
  – An inappropriate learning rate will destroy your happiness

• Second order methods will locally convert the loss function to quadratic
  – Convergence behavior will still depend on the nature of the original function

• Continuing with the quadratic-based explanation...
Convergence

• Convergence behaviors become increasingly unpredictable as dimensions increase

• For the fastest convergence, ideally, the learning rate $\eta$ must be close to both, the largest $\eta_{i,\text{opt}}$ and the smallest $\eta_{i,\text{opt}}$
  – To ensure convergence in every direction
  – Generally infeasible

• Convergence is particularly slow if $\frac{\max_i \eta_{i,\text{opt}}}{\min_i \eta_{i,\text{opt}}}$ is large
  – The “condition” number is small
One reason for the problem

- The objective function has different eccentricities in different directions
  - Resulting in different optimal learning rates for different directions
  - The problem is more difficult when the ellipsoid is not axis aligned: the steps along the two directions are coupled! Moving in one direction changes the gradient along the other

- Solution: *Normalize* the objective to have identical eccentricity in all directions
  - Then all of them will have identical optimal learning rates
  - Easier to find a working learning rate
Solution: Scale the axes

• Scale (and rotate) the axes, such that all of them have identical (identity) “spread”
  – Equal-value contours are circular
  – Movement along the coordinate axes become independent

• Note: equation of a quadratic surface with circular equal-value contours can be written as

\[ E = \frac{1}{2} \mathbf{\hat{w}}^T \mathbf{\hat{w}} + \mathbf{b}^T \mathbf{\hat{w}} + c \]
Scaling the axes

• Original equation:

\[ E = \frac{1}{2} w^T A w + b^T w + c \]

• We want to find a (diagonal) scaling matrix \( S \) such that

\[
S = \begin{bmatrix}
    s_1 & \cdots & 0 \\
    \vdots & \ddots & \vdots \\
    0 & \cdots & s_N \\
\end{bmatrix}, \quad \hat{w} = S w
\]

• And

\[ E = \frac{1}{2} \hat{w}^T \hat{w} + \hat{b}^T \hat{w} + c \]
Scaling the axes

• Original equation:

\[ E = \frac{1}{2} w^T A w + b^T w + c \]

• We want to find a (diagonal) scaling matrix \( S \) such that

\[
S = \begin{bmatrix}
s_1 & \cdots & 0 \\
\vdots & \ddots & \vdots \\
0 & \cdots & s_N
\end{bmatrix}, \quad \hat{w} = Sw
\]

• And

\[
E = \frac{1}{2} \hat{w}^T \hat{w} + \hat{b}^T \hat{w} + c
\]

By inspection:
\[ S = A^{0.5} \]
Scaling the axes

• We have

\[ E = \frac{1}{2} w^T A w + b^T w + c \]

\[ \hat{w} = Sw \]

\[ E = \frac{1}{2} \hat{w}^T \hat{w} + \hat{b}^T \hat{w} + c \]

\[ = \frac{1}{2} w^T S^T S w + \hat{b}^T S w + c \]

• Equating linear and quadratic coefficients, we get

\[ S^T S = A, \quad \hat{b}^T S = b^T \]

• Solving:

\[ S = A^{0.5}, \quad \hat{b} = A^{-0.5} b \]
Scaling the axes

• We have

\[ E = \frac{1}{2} w^T A w + b^T w + c \]

\[ \hat{w} = S w \]

\[ E = \frac{1}{2} \hat{w}^T \hat{w} + \hat{b}^T \hat{w} + c \]

• Solving for \( S \) we get

\[ \hat{w} = A^{0.5} w, \quad \hat{b} = A^{-0.5} b \]
Scaling the axes

• We have

\[ E = \frac{1}{2} w^T A w + b^T w + c \]

\[ \hat{w} = S w \]

\[ E = \frac{1}{2} \hat{w}^T \hat{w} + \hat{b}^T \hat{w} + c \]

• Solving for \( S \) we get

\[ \hat{w} = A^{0.5} w, \quad \hat{b} = A^{-0.5} b \]
The Inverse Square Root of $A$

• For any positive definite $A$, we can write
  \[ A = E \Lambda E^T \]
  – Eigen decomposition
  – $E$ is an orthogonal matrix
  – $\Lambda$ is a diagonal matrix of non-zero diagonal entries

• Defining $A^{0.5} = E \Lambda^{0.5} E^T$
  – Check $\left(A^{0.5}\right)^T A^{0.5} = E \Lambda E^T = A$

• Defining $A^{-0.5} = E \Lambda^{-0.5} E^T$
  – Check: $\left(A^{-0.5}\right)^T A^{-0.5} = E \Lambda^{-1} E^T = A^{-1}$
Returning to our problem

\[ E = \frac{1}{2} \hat{w}^T \hat{w} + \hat{b}^T \hat{w} + c \]

- Computing the gradient, and noting that \( A^{0.5} \) is symmetric, we can relate \( \nabla_{\hat{w}} E \) and \( \nabla_{w} E \):

\[
\nabla_{\hat{w}} E = \hat{w}^T + \hat{b}^T \\
= w^T A^{0.5} + b^T A^{-0.5} \\
= (w^T A + b^T) A^{-0.5} \\
= \nabla_{w} E \cdot A^{-0.5}
\]
Returning to our problem

- Gradient descent rule:
  
  \[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \nabla_{\mathbf{w}} E(\mathbf{w}^{(k)})^T \]

  - Learning rate is now independent of direction

- Using \( \hat{\mathbf{w}} = A^{0.5} \mathbf{w} \), and \( \nabla_{\hat{\mathbf{w}}} E(\hat{\mathbf{w}})^T = A^{-0.5} \nabla_{\mathbf{w}} E(\mathbf{w})^T \)

\[
\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta A^{-1} \nabla_{\mathbf{w}} E(\mathbf{w}^{(k)})^T
\]
Modified update rule

\[ E = \frac{1}{2} w^T A w + b^T w + c \]

**Leads to the modified gradient descent rule**

\[ \hat{w}^{(k+1)} = \hat{w}^{(k)} - \eta \nabla_{\hat{w}} E(\hat{w}^{(k)})^T \]

\[ w^{(k+1)} = w^{(k)} - \eta A^{-1} \nabla_w E(w^{(k)})^T \]
For non-axis-aligned quadratics..

$$E = \frac{1}{2} w^T A w + w^T b + c$$

$$E = \frac{1}{2} \sum_i a_{ii} w_i^2 + \sum_{i \neq j} a_{ij} w_i w_j + \sum_i b_i w_i + c$$

- If $A$ is not diagonal, the contours are not axis-aligned
  - Because of the cross-terms $a_{ij} w_i w_j$
  - The major axes of the ellipsoids are the *Eigenvectors* of $A$, and their diameters are proportional to the Eigen values of $A$

- But this does not affect the discussion
  - This is merely a rotation of the space from the axis-aligned case
  - The component-wise optimal learning rates along the major and minor axes of the equal-contour ellipsoids will be different, causing problems
    - The optimal rates along the axes are Inversely proportional to the *eigenvalues* of $A$
For non-axis-aligned quadratics..

• The component-wise optimal learning rates along the major and minor axes of the contour ellipsoids will differ, causing problems
  – Inversely proportional to the eigenvalues of $A$

• This can be fixed as before by rotating and resizing the different directions to obtain the same normalized update rule as before:
  $$w^{(k+1)} = w^{(k)} - \eta A^{-1}b$$
Generic differentiable multivariate convex functions

- Taylor expansion

\[ E(w) \approx E(w^{(k)}) + \nabla_w E(w^{(k)})(w - w^{(k)}) + \frac{1}{2} (w - w^{(k)})^T H_E(w^{(k)})(w - w^{(k)}) + \ldots \]
Generic differentiable *multivariate* convex functions

• Taylor expansion

\[ E(w) \approx E(w^{(k)}) + \nabla_w E(w^{(k)})(w - w^{(k)}) + \frac{1}{2} (w - w^{(k)})^T H_E(w^{(k)})(w - w^{(k)}) + \ldots \]

• Note that this has the form \( \frac{1}{2} w^T A w + w^T b + c \)

• Using the same logic as before, we get the normalized update rule

\[ w^{(k+1)} = w^{(k)} - \eta H_E(w^{(k)})^{-1} \nabla_w E(w^{(k)})^T \]

• For a quadratic function, the optimal \( \eta \) is 1 (which is exactly Newton’s method)
  – And should not be greater than 2!
Minimization by Newton’s method \((\eta = 1)\)

- Iterated localized optimization with quadratic approximations

\[
{\bf w}^{(k+1)} = {\bf w}^{(k)} - \eta H_E({\bf w}^{(k)})^{-1} \nabla_w E({\bf w}^{(k)})^T \\
- \eta = 1
\]
Minimization by Newton’s method \((\eta = 1)\)

- Iterated localized optimization with quadratic approximations

\[ w^{(k+1)} = w^{(k)} - \eta H_E(w^{(k)})^{-1} \nabla_w E(w^{(k)})^T - \eta = 1 \]
Minimization by Newton’s method \((\eta = 1)\)

- Iterated localized optimization with quadratic approximations

\[
\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_w E(\mathbf{w}^{(k)})^T
\]

\(- \eta = 1\)
Minimization by Newton’s method \((\eta = 1)\)

- Iterated localized optimization with quadratic approximations

\[
\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_w E(\mathbf{w}^{(k)})^T
\]

\(- \eta = 1\)
Minimization by Newton’s method

- Iterated localized optimization with quadratic approximations

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_w E(\mathbf{w}^{(k)})^T \]

- \( \eta = 1 \)
Minimization by Newton’s method

- Iterated localized optimization with quadratic approximations

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \nabla \mathbf{w} E (\mathbf{w}^{(k)}) - \eta = 1 \]
Minimization by Newton’s method

- Iterated localized optimization with quadratic approximations

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_w E(\mathbf{w}^{(k)})^T \]

- \( \eta = 1 \)
Minimization by Newton’s method

- Iterated localized optimization with quadratic approximations

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_w E(\mathbf{w}^{(k)})^T \]

- \( \eta = 1 \)
Minimization by Newton’s method

- Iterated localized optimization with quadratic approximations

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_w E(\mathbf{w}^{(k)})^T \]

- \( \eta = 1 \)
Minimization by Newton’s method

- Iterated localized optimization with quadratic approximations

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_w E(\mathbf{w}^{(k)})^T \]

\[ - \eta = 1 \]
Minimization by Newton’s method

- Iterated localized optimization with quadratic approximations

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_w E(\mathbf{w}^{(k)})^T \]

\[- \eta = 1 \]
Issues: 1. The Hessian

- Normalized update rule

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_\mathbf{w} E(\mathbf{w}^{(k)})^T \]

- For complex models such as neural networks, with a very large number of parameters, the Hessian \( H_E(\mathbf{w}^{(k)}) \) is extremely difficult to compute
  - For a network with only 100,000 parameters, the Hessian will have \( 10^{10} \) cross-derivative terms
  - And its even harder to invert, since it will be enormous
Issues: 1. The Hessian

• For non-convex functions, the Hessian may not be positive semi-definite, in which case the algorithm can **diverge**
  – Goes away from, rather than towards the minimum
Issues: 1. The Hessian

- For non-convex functions, the Hessian may not be positive semi-definite, in which case the algorithm can *diverge*
  - Goes away from, rather than towards the minimum
  - Now requires additional checks to avoid movement in directions corresponding to −ve Eigenvalues of the Hessian
Issues: 1 – contd.

• A great many approaches have been proposed in the literature to approximate the Hessian in a number of ways and improve its positive definiteness
  – Boyden-Fletcher-Goldfarb-Shanno (BFGS)
    • And “low-memory” BFGS (L-BFGS)
    • Estimate Hessian from finite differences
  – Levenberg-Marquardt
    • Estimate Hessian from Jacobians
    • Diagonal load it to ensure positive definiteness
  – Other “Quasi-newton” methods

• Hessian estimates may even be local to a set of variables

• Not particularly popular anymore for large neural networks.
Issues: 2. The learning rate

• Much of the analysis we just saw was based on trying to ensure that the step size was not so large as to cause divergence within a convex region

  – $\eta < 2\eta_{opt}$
Issues: 2. The learning rate

• For complex models such as neural networks the loss function is often not convex
  – Having $\eta > 2\eta_{opt}$ can actually help escape local optima

• However always having $\eta > 2\eta_{opt}$ will ensure that you never ever actually find a solution
Decaying learning rate

- Start with a large learning rate
  - Greater than 2 (assuming Hessian normalization)
  - Gradually reduce it with iterations

Note: this is actually a reduced step size
Decaying learning rate

• Typical decay schedules

  – Linear decay: \( \eta_k = \frac{\eta_0}{k+1} \)

  – Quadratic decay: \( \eta_k = \frac{\eta_0}{(k+1)^2} \)

  – Exponential decay: \( \eta_k = \eta_0 e^{-\beta k} \), where \( \beta > 0 \)

• A common approach (for nnets):
  1. Train with a fixed learning rate \( \eta \) until loss (or performance on a held-out data set) stagnates
  2. \( \eta \leftarrow \alpha \eta \), where \( \alpha < 1 \) (typically 0.1)
  3. Return to step 1 and continue training from where we left off
Story so far: Convergence

- Gradient descent can miss obvious answers
  - And this may be a *good* thing

- Convergence issues abound
  - The loss surface has many saddle points
    - Although, perhaps, not so many bad local minima
    - Gradient descent can stagnate on saddle points
  - Vanilla gradient descent may not converge, or may converge tooooooo slowly
    - The optimal learning rate for one component may be too high or too low for others
Story so far: Second-order methods

- Second-order methods “normalize” the variation along the components to mitigate the problem of different optimal learning rates for different components
  - But this requires computation of inverses of second-order derivative matrices
  - Computationally infeasible
  - Not stable in non-convex regions of the loss surface
  - Approximate methods address these issues, but simpler solutions may be better
Divergence-causing learning rates may not be a bad thing
   – Particularly for ugly loss functions

Decaying learning rates provide good compromise between escaping poor local minima and convergence

Many of the convergence issues arise because we force the same learning rate on all parameters
Let's take a step back

- Problems arise because of requiring a fixed step size across all dimensions
  - Because step are “tied” to the gradient
- Let's try releasing this requirement

\[
\begin{align*}
\mathbf{w}^{(k+1)} & \leftarrow \mathbf{w}^{(k)} - \eta (\nabla_{\mathbf{w}} E)^T \\
 w_{i}^{(k+1)} & = w_{i}^{(k)} - \eta \frac{dE}{dw}(w_{i}^{(k)})
\end{align*}
\]
Derivative-inspired algorithms

• Algorithms that use derivative information for trends, but do not follow them absolutely

• Rprop
• Quick prop
RProp

- *Resilient* propagation
- Simple algorithm, to be followed *independently* for each component
  - i.e. steps in different directions are not coupled

- At each time
  - If the derivative at the current location recommends continuing in the same direction as before (i.e. has not changed sign from earlier):
    - *increase* the step, and continue in the same direction
  - If the derivative has changed sign (i.e. we’ve overshot a minimum)
    - *reduce* the step and reverse direction
Rprop

- Select an initial value $\hat{w}$ and compute the derivative
  - Take an initial step $\Delta w$ against the derivative
    - In the direction that reduces the function
      - $\Delta w = \text{sign} \left( \frac{dE(\hat{w})}{dw} \right) \Delta w$
      - $\hat{w} = \hat{w} - \Delta w$

Orange arrow shows direction of derivative, i.e. direction of increasing $E(w)$
• Compute the derivative in the new location
  – If the derivative has not changed sign from the previous location, increase the step size and take a longer step

\[ \alpha > 1 \]

• \( \Delta w = \alpha \Delta w \)
• \( \hat{w} = \hat{w} - \Delta w \)
Rprop

• Compute the derivative in the new location
  – If the derivative has not changed sign from the previous location, increase the step size and take a step

\[ \alpha > 1 \]

• \( \Delta w = \alpha \Delta w \)
• \( \hat{w} = \hat{w} - \Delta w \)
Rprop

- Compute the derivative in the new location
  - If the derivative has changed sign

Orange arrow shows direction of derivative, i.e. direction of increasing $E(w)$

$\Delta w_0$, $\alpha \Delta w_0$, $\alpha^2 \Delta w_0$
Rprop

- Compute the derivative in the new location
  - If the derivative has changed sign
  - Return to the previous location
    - $\hat{w} = \hat{w} + \Delta w$

Orange arrow shows direction of derivative, i.e. direction of increasing $E(w)$
• Compute the derivative in the new location
  – If the derivative has changed sign
  – Return to the previous location
    • $\hat{w} = \hat{w} + \Delta w$
  – Shrink the step
    • $\Delta w = \beta \Delta w$

$\beta < 1$
Rprop

- Compute the derivative in the new location
  - If the derivative has changed sign
  - Return to the previous location
    - \( \hat{w} = \hat{w} + \Delta w \)
    - \( \beta < 1 \)
      - Shrink the step
        - \( \Delta w = \beta \Delta w \)
        - Take the smaller step forward
          - \( \hat{w} = \hat{w} - \Delta w \)

Orange arrow shows direction of derivative, i.e. direction of increasing \( E(w) \)
Rprop (simplified)

• Set $\alpha = 1.2$, $\beta = 0.5$
• For each layer $l$, for each $i, j$:
  – Initialize $w_{l,i,j}, \Delta w_{l,i,j} > 0$,
  – $\text{prevD}(l, i, j) = \frac{d\text{Err}(w_{l,i,j})}{dw_{l,i,j}}$
  – $\Delta w_{l,i,j} = \text{sign}(\text{prevD}(l, i, j))\Delta w_{l,i,j}$
  – While not converged:
    • $w_{l,i,j} = w_{l,i,j} - \Delta w_{l,i,j}$
    • $D(l, i, j) = \frac{d\text{Err}(w_{l,i,j})}{dw_{l,i,j}}$
    • If $\text{sign}(\text{prevD}(l, i, j)) = \text{sign}(D(l, i, j))$:
      – $\Delta w_{l,i,j} = \min(\alpha \Delta w_{l,i,j}, \Delta_{\text{max}})$
      – $\text{prevD}(l, i, j) = D(l, i, j)$
    • else:
      – $w_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$
      – $\Delta w_{l,i,j} = \max(\beta \Delta w_{l,i,j}, \Delta_{\text{min}})$
Rprop (simplified)

- Set $\alpha = 1.2$, $\beta = 0.5$
- For each layer $l$, for each $i, j$:
  - Initialize $w_{l,i,j}, \Delta w_{l,i,j} > 0$,
  - $\text{prevD}(l, i, j) = \frac{d\text{Err}(w_{l,i,j})}{dw_{l,i,j}}$
  - $\Delta w_{l,i,j} = \text{sign}(\text{prevD}(l, i, j))\Delta w_{l,i,j}$
  - While not converged:
    - $w_{l,i,j} = w_{l,i,j} - \Delta w_{l,i,j}$
    - $D(l, i, j) = \frac{d\text{Err}(w_{l,i,j})}{dw_{l,i,j}}$
    - If $\text{sign}(\text{prevD}(l, i, j)) = \text{sign}(D(l, i, j))$:
      - $\Delta w_{l,i,j} = \alpha \Delta w_{l,i,j}$
      - $\text{prevD}(l, i, j) = D(l, i, j)$
    - else:
      - $w_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$
      - $\Delta w_{l,i,j} = \beta \Delta w_{l,i,j}$
RProp

• A remarkably simple first-order algorithm, that is frequently much more efficient than gradient descent.
  – And can even be competitive against some of the more advanced second-order methods

• Only makes minimal assumptions about the loss function
  – No convexity assumption
QuickProp

• Quickprop employs the Newton updates with two modifications

\[ \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E \left( \mathbf{w}^{(k)} \right)^{-1} \nabla_w E ( \mathbf{w}^{(k)} )^T \]

• But with two modifications
QuickProp: Modification 1

- It treats each dimension independently
- For \( i = 1 : N \)
  \[
  w_i^{k+1} = w_i^k - E''(w_i^k | w_j^k, j \neq i)^{-1} E'(w_i^k | w_j^k, j \neq i)
  \]
- This eliminates the need to compute and invert expensive Hessians
QuickProp: Modification 2

• It approximates the second derivative through finite differences

\[ w_{i}^{k+1} = w_{i}^{k} - D(w_{i}^{k}, w_{i}^{k-1})^{-1} E'(w_{i}^{k} | w_{j}^{k}, j \neq i) \]

• This eliminates the need to compute expensive double derivatives
QuickProp

\[ w^{(k+1)} = w^{(k)} - \left( \frac{E'(w^{(k)}) - E'(w^{(k-1)})}{\Delta w^{(k-1)}} \right)^{-1} E'(w^{(k)}) \]

Finite-difference approximation to double derivative obtained assuming a quadratic \( E() \)

- Updates are independent for every parameter
- For every layer \( l \), for every connection from node \( i \) in the \((l - 1)\)th layer to node \( j \) in the \( l \)th layer:

\[
\Delta w^{(k)}_{l,ij} = \frac{\Delta w^{(k-1)}_{l,ij}}{Err'(w^{(k)}_{l,ij}) - Err'(w^{(k-1)}_{l,ij})} \cdot Err'(w^{(k)}_{l,ij})
\]

\[ w^{(k+1)}_{l,ij} = w^{(k)}_{l,ij} - \Delta w^{(k)}_{l,ij} \]
QuickProp

\[ w^{(k+1)} = w^{(k)} - \left( \frac{E'(w^{(k)}) - E'(w^{(k-1)})}{\Delta w^{(k-1)}} \right)^{-1} E'(w^{(k)}) \]

Finite-difference approximation to double derivative obtained assuming a quadratic \( E() \)

- Updates are independent for every parameter
- For every layer \( l \), for every connection from node \( i \) in the \((l-1)\)th layer to node \( j \) in the \( l \)th layer:

\[ \Delta w_{l,ij}^{(k)} = \frac{\Delta w_{l,ij}^{(k-1)}}{Err'(w_{l,ij}^{(k)}) - Err'(w_{l,ij}^{(k-1)})} Err'(w_{l,ij}^{(k)}) \]

\[ w_{l,ij}^{(k+1)} = w_{l,ij}^{(k)} - \Delta w_{l,ij}^{(k)} \]

Computed using backprop
Quickprop

• Employs Newton updates with empirically derived derivatives

• Prone to some instability for non-convex objective functions

• But is still one of the fastest training algorithms for many problems
Story so far: Convergence

- Gradient descent can miss obvious answers
  - And this may be a good thing

- Vanilla gradient descent may be too slow or unstable due to the differences between the dimensions

- Second order methods can normalize the variation across dimensions, but are complex

- Adaptive or decaying learning rates can improve convergence

- Methods that decouple the dimensions can improve convergence
A closer look at the convergence problem

- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others.

---

Proposal:
- Keep track of oscillations
- Emphasize steps in directions that converge smoothly
- Shrink steps in directions that bounce around.
A closer look at the convergence problem

- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others

- **Proposal:**
  - Keep track of oscillations
  - Emphasize steps in directions that converge smoothly
  - Shrink steps in directions that bounce around.
The momentum methods

• Maintain a running average of all past steps
  – In directions in which the convergence is smooth, the average will have a large value
  – In directions in which the estimate swings, the positive and negative swings will cancel out in the average

• Update with the running average, rather than the current gradient
Momentum Update

The momentum method maintains a running average of all gradients until the current step

\[
\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla W \text{Loss}(W^{(k-1)})^T
\]

\[
W^{(k)} = W^{(k-1)} + \Delta W^{(k)}
\]

- Typical $\beta$ value is 0.9

The running average steps
- Get longer in directions where gradient retains the same sign
- Become shorter in directions where the sign keeps flipping
Training by gradient descent

- Initialize all weights $W_1, W_2, \ldots, W_K$
- Do:
  - For all $i, j, k$, initialize $\nabla_{W_k} \text{Loss} = 0$
  - For all $t = 1 : T$
    - For every layer $k$:
      - Compute $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      - Compute $\nabla_{W_k} \text{Loss} + = \frac{1}{T} \nabla_{W_k} \text{Div}(Y_t, d_t)$
    - For every layer $k$:
      \[ W_k = W_k - \eta (\nabla_{W_k} \text{Loss})^T \]
- Until $\text{Loss}$ has converged
Training with momentum

• Initialize all weights $W_1, W_2, \ldots, W_K$

• Do:
  – For all layers $k$, initialize $\nabla_{W_k} \text{Loss} = 0, \Delta W_k = 0$
  – For all $t = 1: T$
    • For every layer $k$:
      – Compute gradient $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      – $\nabla_{W_k} \text{Loss} += \frac{1}{T} \nabla_{W_k} \text{Div}(Y_t, d_t)$
    – For every layer $k$
      $\Delta W_k = \beta \Delta W_k - \eta (\nabla_{W_k} \text{Loss})^T$
      \[ W_k = W_k + \Delta W_k \]

• Until $\text{Loss}$ has converged
Momentum Update

- The momentum method

\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)})^T \]

- At any iteration, to compute the current step:
Momentum Update

- The momentum method
  \[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)})^T \]
- At any iteration, to compute the current step:
  - First computes the gradient step at the current location
Momentum Update

- The momentum method

\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)})^T \]

- At any iteration, to compute the current step:
  - First computes the gradient step at the current location
  - Then adds in the scaled previous step
    - Which is actually a running average
Momentum Update

The momentum method

\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_w \text{Loss}(W^{(k-1)})^T \]

At any iteration, to compute the current step:

– First computes the gradient step at the current location
– Then adds in the scaled previous step
  • Which is actually a running average
– To get the final step
Momentum update

• Momentum update steps are actually computed in two stages
  – First: We take a step against the gradient at the current location
  – Second: Then we add a scaled version of the previous step

• The procedure can be made more optimal by reversing the order of operations..
Nestorov’s Accelerated Gradient

• Change the order of operations

• At any iteration, to compute the current step:
Nestorov’s Accelerated Gradient

• Change the order of operations

• At any iteration, to compute the current step:
  – First extend the previous step
Nestorov’s Accelerated Gradient

• Change the order of operations
• At any iteration, to compute the current step:
  – First extend the previous step
  – Then compute the gradient step at the resultant position
Nestorov’s Accelerated Gradient

• Change the order of operations
• At any iteration, to compute the current step:
  – First extend the previous step
  – Then compute the gradient step at the resultant position
  – Add the two to obtain the final step
Nestorov’s Accelerated Gradient

- Nestorov’s method

\[
\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)} + \beta \Delta W^{(k-1)})^T
\]

\[
W^{(k)} = W^{(k-1)} + \Delta W^{(k)}
\]
Nestorov’s Accelerated Gradient

- Comparison with momentum (example from Hinton)
- Converges much faster
Training with Nestorov

• Initialize all weights $W_1, W_2, ..., W_K$
• Do:
  – For all layers $k$, initialize $\nabla_{W_k} Loss = 0$, $\Delta W_k = 0$
  – For every layer $k$
    \[ W_k = W_k + \beta \Delta W_k \]
  – For all $t = 1: T$
    • For every layer $k$:
      – Compute gradient $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      \[ \nabla_{W_k} Loss += \frac{1}{T} \nabla_{W_k} \text{Div}(Y_t, d_t) \]
    – For every layer $k$
      \[ W_k = W_k - \eta (\nabla_{W_k} Loss)^T \]
      \[ \Delta W_k = \beta \Delta W_k - \eta (\nabla_{W_k} Loss)^T \]
• Until $Loss$ has converged
Momentum and trend-based methods..

• We will return to this topic again, very soon..
Story so far

• Gradient descent can miss obvious answers
  – And this may be a good thing

• Vanilla gradient descent may be too slow or unstable due to the differences between the dimensions

• Second order methods can normalize the variation across dimensions, but are complex

• Adaptive or decaying learning rates can improve convergence

• Methods that decouple the dimensions can improve convergence

• Momentum methods which emphasize directions of steady improvement are demonstrably superior to other methods
Coming up

• Incremental updates
• Revisiting “trend” algorithms
• Generalization
• Tricks of the trade
  – Divergences..
  – Activations
  – Normalizations