Softmax classifier – cross-entropy loss

- Cross-entropy: $H(p, q) = E_p[- \log q] = -\sum_j p_j \log q_j$
- In our case,
  - $p$ denotes the correct probabilities of the categories. In other words, $p_j = 1$ for the correct label and $p_j = 0$ for other categories.
  - $q$ denotes the estimated probabilities of the categories
- But, our scores are not probabilities!
  - One solution: Softmax function: $sm(s_i) = \frac{e^{s_i}}{\sum_j e^{s_j}}$
  - It maps arbitrary ranges to probabilities
- Using the normalized values, we can define the cross-entropy loss for classification problem now:

$$L_i = -\log e \left( \frac{e^{s_{yi}}}{\sum_j e^{s_j}} \right) = -s_{yi} + \log e \sum_j e^{s_j}$$

http://cs231n.github.io/
logistic loss

• A special case of cross-entropy for binary classification:

\[ H(p, q) = - \sum_{j} p_j \log q_j = -p \log q - (1 - p) \log(1 - q) \]

• Softmax function reduces to the logistic function (see [1] for the derivation):

\[ \frac{1}{1 + e^{-x}} \]

Softmax classifier:
Another interpretation

• Probabilistic view

\[ P(y_i \mid x_i; W) = \frac{e^{s_{y_i}}}{\sum_{j} e^{s_j}}. \]

• In our case, we are minimizing the negative log likelihood.

• This corresponds to Maximum Likelihood Estimation (MLE).

• See the following for a quick look at MLE:
  https://wiseodd.github.io/techblog/2017/01/01/mle-vs-map/

http://cs231n.github.io/
SVM loss vs. cross-entropy loss

• SVM is happy when the classification satisfies the margin
  – Ex: if score values = [10, 9, 9] or [10, -10, -10]
    • SVM loss is happy if the margin is 1

• Cross-entropy is more ambitious: it wants more than a margin
0-1 Loss

• Minimize the # of cases where the prediction is wrong:

\[ L = \sum_{i} 1(f(x_i; W, b)_{y_i} \neq \hat{y}_i) \]

Or equivalently,

\[ L = \sum_{i} 1(\hat{y}_i f(x_i; W, b)_{y_i} < 0) \]
You see that this is a convex function.
  – Nice and easy for optimization

When you combine many of them in a neural network, it becomes non-convex.
Another approach for visualizing loss functions

- 0-1 loss:
  \[ L = 1(f(x) \neq y) \]
  or equivalently as:
  \[ L = 1(yf(x) > 0) \]
- Square loss:
  \[ L = (f(x) - y)^2 \]
  in binary case:
  \[ L = (1 - yf(x))^2 \]
- Hinge-loss
  \[ L = \max(1 - yf(x), 0) \]
- Logistic loss:
  \[ L = (\ln 2)^{-1}\ln(1 + e^{-yf(x)}) \]

Various loss functions used in classification. Here \( t = yf(x) \).
Rosacco et al., 2003
Activation Functions: Sigmoid vs. tanh

- Sigmoid is a historically important activation function
  - But nowadays, rarely used
    - Drawbacks:
      1. It gets saturated, if the activation is close to zero or one
         - This leads to very small gradient, which affects the feedback to earlier layers
         - Initialization is also very important for this reason
      2. It is not zero-centered (not very severe)

- Tanh
  - Similar to the sigmoid, it saturates
  - However, it is zero-centered.  
  - Tanh is always preferred over sigmoid
  - Note: \( \text{tanh}(x) = 2\sigma(2x) - 1 \)

Activation Functions: **Rectified Linear Units (ReLU)**

Vinod Nair and Geoffrey Hinton (2010). Rectified linear units improve restricted Boltzmann machines, ICML.

\[ f(x) = \max(0, x) \]

Derivative: \( 1(x > 0) \)
Activation Functions:
ReLU – biological motivation

Figure 1: Left: Common neural activation function motivated by biological data. Right: Commonly used activation functions in neural networks literature: logistic sigmoid and hyperbolic tangent (tanh).

Activation Functions:

ReLU – biological motivation

Hinton argues that this is a form of model averaging
Activation Functions:
ReLU: Pros and Cons

• **Pros:**
  – It converges much faster (claimed to be 6x faster than sigmoid/tanh)
    • It overfits very fast and when used with e.g. dropout, this leads to very fast convergence
  – It is simpler and faster to compute (simple comparison)

• **Cons:**
  – A ReLU neuron may “die” during training
  – A large gradient may update the weights such that the ReLU neuron may never activate again
    • Avoid large learning rate

• **See also:**
  http://www.jefkine.com/general/2016/08/24/formulating-the-relu/
Activation Functions: Leaky ReLU

- \( f(x) = 1(x < 0)(\alpha x) + 1(x \geq 0)(x) \)
  - When \( x \) is negative, have a non-zero slope (\( \alpha \))

- If you learn \( \alpha \) during training, this is called parametric ReLU (PReLU)

Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng (2014). Rectifier Nonlinearities Improve Neural Network Acoustic Models

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2015) Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification
Activation Functions: maxout

- $\max(w_1^T x + b_1, w_2^T x + b_2)$

- ReLU, Leaky ReLU and PReLU are special cases of this
- Drawback: More parameters to learn!

“Maxout Networks” by Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, Yoshua Bengio, 2013.
Activation Functions: **Softplus**

- A smooth approximation to the ReLU unit:
  \[ f(x) = \ln(1 + e^x) \]

- Its derivative is the sigmoid function:
  \[ f'(x) = \frac{1}{1 + e^{-x}} \]
Activation Functions:
Swish: A Self-Gated Activation Function

“The choice of activation functions in deep networks has a significant effect on the training dynamics and task performance. Currently, the most successful and widely-used activation function is the Rectified Linear Unit (ReLU). Although various alternatives to ReLU have been proposed, none have managed to replace it due to inconsistent gains. In this work, we propose a new activation function, named Swish, which is simply \( f(x) = x \cdot \text{sigmoid}(x) \). Our experiments show that Swish tends to work better than ReLU on deeper models across a number of challenging datasets. For example, simply replacing ReLUs with Swish units improves top-1 classification accuracy on ImageNet by 0.9% for Mobile NASNet-A and 0.6% for Inception-ResNet-v2. The simplicity of Swish and its similarity to ReLU make it easy for practitioners to replace ReLUs with Swish units in any neural network.”
Activation Functions:

Exponential Linear Unit

• Similar to the Swish function

\[ \text{ELU} = \begin{cases} 
  x & x \geq 0 \\
  \alpha(e^x - 1) & x < 0 
\end{cases} \]
Today

- Towards deep learning
  - Loss functions
  - A general look at optimization
  - Gradient Descent strategies
  - Challenges of the loss surface
  - Momentum
  - Setting the learning rate

- Representational Capacity
- Overfitting, convergence, when to stop training
- Preprocessing and initialization
- Issues and practical advices

These slides at: http://kovan.ceng.metu.edu.tr/~sinan/DL/week_6.pdf
Administrative Issues

• Programming assignment 1: Deadline 25\textsuperscript{th} of April
• Take-Home Exam 1 coming in two weeks
• Project paper selection
  – https://docs.google.com/spreadsheets/d/1tzPHq_Vgu6gCwNyXJHGvqeA6pgU67H0nKYjqkisWfKc/edit?usp=sharing
  – Deadline: 19\textsuperscript{th} of April
LOSS FUNCTIONS, AGAIN
Loss functions

• **A single correct** label case (classification):
  – Hinge loss:
    • \( L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1) \)
  – Cross-entropy (negative log-likelihood) loss:
    • \( L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) \)
Loss functions

• **Many correct** labels case:
  – Binary prediction for each label, independently:
    • \( L_i = \sum_j \max(0, 1 - y_{ij}f_j) \)
    • \( y_{ij} = +1 \) if example \( i \) is labeled with label \( j \); otherwise \( y_{ij} = -1 \).

  – Alternatively, train logistic loss for each label (0 or 1):
    \[
    L_i = \sum_j y_{ij} \log(\sigma(f_j)) + (1 - y_{ij}) \log(1 - \sigma(f_j))
    \]
Loss functions

• What if we want to predict a graph, tree etc.? Something that has structure.
  – **Structured loss**: formulate loss such that you minimize the distance to a correct structure
Courses/tutorials:

• METU IAM771: Optimization Methods for Machine Learning
  https://catalog.metu.edu.tr/course.php?course_code=9700771

• EPFL: Optimization for Machine Learning:
  https://github.com/epfml/OptML_course

• Optimization Algorithms in Machine Learning:
  http://videolectures.net/nips2010_wright_oaml/

A GENERAL LOOK AT OPTIMIZATION
Mathematical Optimization

Nonlinear Optimization

Convex Optimization

Least-squares

LP
(mathematical) optimization problem

minimize \quad f_0(x) \\
subject to \quad f_i(x) \leq b_i, \quad i = 1, \ldots, m

- \( x = (x_1, \ldots, x_n) \): optimization variables
- \( f_0 : \mathbb{R}^n \rightarrow \mathbb{R} \): objective function
- \( f_i : \mathbb{R}^n \rightarrow \mathbb{R}, i = 1, \ldots, m \): constraint functions

**optimal solution** \( x^* \) has smallest value of \( f_0 \) among all vectors that satisfy the constraints
Convex Optimization

minimize \( f_0(x) \)
subject to \( f_i(x) \leq b_i, \quad i = 1, \ldots, m \)

- objective and constraint functions are convex:

\[
f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y)
\]

if \( \alpha + \beta = 1, \alpha \geq 0, \beta \geq 0 \)

- includes least-squares problems and linear programs as special cases
Interpretation

• Function’s value is below the line connecting two points
Another interpretation

A differentiable function $f$ is convex if for all $x$ and $y$ we have

$$f(y) \geq f(x) + \nabla f(x)^T (y - x).$$

- The function is globally above the tangent at $x$. 

![Diagram showing a convex function and its tangent line at a point](image.png)
Convex vs. Non-convex Ex.

- Convex, min. easy to find

Affine – border case of convexity
Convex vs. Non-convex Ex.

- Non-convex, easy to get stuck in a local min.
- Can’t rely on only local search techniques
Example convex functions

Some simple convex functions:

- $f(x) = c$
- $f(x) = a^T x$
- $f(x) = x^T A x$ (for $A \succeq 0$)
- $f(x) = \exp(ax)$
- $f(x) = x \log x$ (for $x > 0$)
- $f(x) = \|x\|^2$
- $f(x) = \|x\|_p$
- $f(x) = \max_i \{x_i\}$

Some other notable examples:

- $f(x, y) = \log(e^x + e^y)$
- $f(X) = \log \det X$ (for $X$ positive-definite).
- $f(x, Y) = x^T Y^{-1} x$ (for $Y$ positive-definite)
Operations that conserve convexity

1. Non-negative weighted sum:
   \[ f(x) = \theta_1 f_1(x) + \theta_2 f_2(x). \]

2. Composition with affine mapping:
   \[ g(x) = f(Ax + b). \]

3. Pointwise maximum:
   \[ f(x) = \max_i \{f_i(x)\}. \]
Deep learning functions

• $Wx$ - convex
• ReLU – convex
• Softmax – convex
• Sigmoid, tanh – non-convex
• Loss functions (cross-entropy, max-margin, squared-error loss) are convex
• How about NNs?
  – NNs without non-linearities are convex. The parameters just model a hyper-plane.
  – NNs with non-linearities are convex.
Why convex optimization?

• Can’t solve most OPs
  – E.g. NP Hard, even high polynomial time too slow

• Convex OPs
  – (Generally) No analytic solution
  – Efficient iterative algorithms to find (global) solution

• Easy to see why convexity allows for efficient solution
  – Just “slide” down the objective function as far as possible and will reach a minimum
Non-convex Problems

• Some non-convex problems highly multi-modal, or NP hard
• Could be forced to search all solutions, or hope stochastic search is successful
• Cannot guarantee best solution, inefficient
• Harder to make performance guarantees with approximate solutions
Mathematical Optimization

Nonlinear Optimization

Convex Optimization

Least-squares

LP

- Analytical solution
- Good algorithms and software
- High accuracy and high reliability

minimize $\|Ax - b\|_2^2$

A mature technology!
Mathematical Optimization

Nonlinear Optimization

Convex Optimization

Least-squares

LP

minimize \( c^T x \)
subject to \( a_i^T x \leq b_i \)
\( i = 1, \ldots, m \)

- No analytical solution
- Algorithms and software
- Reliable and efficient

Also a mature technology!
Almost a mature technology!

Mathematical Optimization

Convex Optimization

Least-squares

LP

Nonlinear Optimization

Minimize $f_0(x)$ subject to $f_i(x) \leq b_i, \ i = 1, \ldots, m$

- No analytical solution
- Algorithms and software
- Reliable and efficient
Sadly, no effective methods to solve
Only approaches with some compromise
Local optimization: “more art than technology”
Global optimization: greatly compromised efficiency
Help from convex optimization
1) Initialization 2) Heuristics 3) Bounds

Far from a technology! (something to avoid)
Why Study Convex Optimization

With only a bit of exaggeration, we can say that, if you formulate a practical problem as a convex optimization problem, then you have solved the original problem. If not, there is little chance you can solve it.

-- Section 1.3.2, p8, Convex Optimization
Recommended:

Course on Neural Net Training Dynamics
https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2021/
Schemes of training

• True/Standard Gradient Descent
• Stochastic Gradient Descent
• Steepest Gradient Descent
• Momentum Gradient Descent
Stochastic Gradient Descent

Batch Gradient Descent

F. Bach
Large vs. small batch sizes

Forth the following as possible causes for this phenomenon: (i) LB methods over-fit the model; (ii) LB methods are attracted to saddle points; (iii) LB methods lack the explorative properties of SB methods and tend to zoom-in on the minimizer closest to the initial point; (iv) SB and LB methods converge to qualitatively different minimizers with differing generalization properties. The data presented in this paper supports the last two conjectures.

The main observation of this paper is as follows:

The lack of generalization ability is due to the fact that large-batch methods tend to converge to sharp minimizers of the training function. These minimizers are characterized by a significant number of large positive eigenvalues in $\nabla^2 f(x)$, and tend to generalize less well. In contrast, small-batch methods converge to flat minimizers characterized by having numerous small eigenvalues of $\nabla^2 f(x)$. We have observed that the loss function landscape of deep neural networks is such that large-batch methods are attracted to regions with sharp minimizers and that, unlike small-batch methods, are unable to escape basins of attraction of these minimizers.

Figure 1: A Conceptual Sketch of Flat and Sharp Minima. The Y-axis indicates value of the loss function and the X-axis the variables (parameters)
Large vs. small batch sizes

• Stability [1]:
  – Large batch sizes introduce stability in terms of gradient directions. But this increases the changes of getting stuck in local minima.
  – Small batch sizes introduce noisy gradients which make it difficult to get stuck in local minima.

• Local convergence & width of the minima [2]:
  – Small batch sizes tend to converge to solutions that are farther away from the initial position whereas large batch sizes lead to solutions close to the initial position.


Gradient descent

https://en.wikipedia.org/wiki/Gradient_descent
Second order methods

• Newton’s method for optimization:
  
  \[ w \leftarrow w - [Hf(w)]^{-1} \nabla f(w) \]
  
  where \( Hf(w) \) is the Hessian

  \[ H = \begin{bmatrix}
  \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
  \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
  \vdots & \vdots & \ddots & \vdots \\
  \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2}
  \end{bmatrix} \]

• Hessian gives a better feeling about the surface
  
  – It gives information about the curvature of surface
Intuition behind Newton’s method

- Newton’s method assumes that the function \( f(x) \) we are trying to minimize is quadratic, and aims to find the minimum \( (x + \delta) \), where \( f'(x + \delta) = 0 \).
- From Taylor expansion:
  \[
  f(x + \delta) = f(x) + f'(x)\delta + \frac{1}{2} f''(x)\delta^2
  \]
- Solving for \( \delta \) using \( f'(x + \delta) = 0 \):
  \[
  \frac{d}{d\delta} \left[ f(x) + f'(x)\delta + \frac{1}{2} f''(x)\delta^2 \right] = 0
  \]
  which yields:
  \[
  \delta = f'(x) / f''(x)
  \]
- In high-dimensional cases, \( f'(x) \) is replaced by \( \nabla f(x) \) and \( f''(x) \) by \( Hf(x) \).
Compare this to Newton’s method for finding the roots

- To find a root \( r \) of a function \( (f(x)) \), i.e., \( f(r) = 0 \):
  \[
x_{k+1} = x_k + \frac{f(x_k)}{f'(x_k)}
  \]

- In optimization, we wish to end up with \( f'(x) = 0 \) with:
  \[
x_{k+1} = x_k + \frac{f'(x_k)}{f''(x_k)}
  \]
Newton’s method for optimization

- \( w \leftarrow w - [Hf(w)]^{-1} \nabla f(w) \)
  - Makes bigger steps in shallow curvature
  - Smaller steps in steep curvature

- **Note that there is no (learning rate) hyper-parameter!** (if you wish you can add a step size, but this is not necessary)

- **Disadvantage:**
  - Too much memory requirement
  - For 1 million parameters, this means a matrix of 1 million x 1 million \( \Rightarrow \sim 3725 \text{ GB} \) RAM
  - Alternatives exist to get around the memory problem (quasi-Newton methods, Limited-memory BFGS -- short for Broyden–Fletcher–Goldfarb–Shanno)
Whitening and second order optimization both destroy information about the dataset, and can make generalization impossible

Abstract

Machine learning is predicated on the concept of generalization: a model achieving low error on a sufficiently large training set should also perform well on novel samples from the same distribution. We show that both data whitening and second order optimization can harm or entirely prevent generalization. In general, model training harnesses information contained in the sample-sample second moment matrix of a dataset. We prove that for models with a fully connected first layer, the information contained in this matrix is the only information which can be used to generalize. Models trained using whitened data, or with certain second order optimization schemes, have less access to this information; in the high dimensional regime they have no access at all, producing models that generalize poorly or not at all. We experimentally verify these predictions for several architectures, and further demonstrate that generalization continues to be harmed even when theoretical requirements are relaxed. However, we also show experimentally that regularized second order optimization can provide a practical tradeoff, where training is still accelerated but less information is lost, and generalization can in some circumstances even improve.
RPROP (Resilience Propagation)

• Instead of the magnitude, use the sign of the gradients

\[
\Delta_{ij}^{(t)} = \begin{cases} 
\eta^+ \cdot \Delta_{ij}^{(t-1)}, & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\
\eta^- \cdot \Delta_{ij}^{(t-1)}, & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\
\Delta_{ij}^{(t-1)}, & \text{else}
\end{cases}
\]

(4)

where \( 0 < \eta^- < 1 < \eta^+ \)

• Motivation: If the sign of a gradient has changed, that means we have “overshot” a minima

• Advantage: Faster to run/converge

• Disadvantage: More complex to implement
Gradient Descent with Line Search

• Gradient descent:
  \[ w_{ij}^t = w_{ij}^{t-1} + s \, dir_{ij}^{t-1} \]
  where \( dir_{ij}^{t-1} = -\frac{\partial L}{\partial w_{ij}} \)

• Gradient descent with line search:
  – Choose \( s \) such that \( L \) is minimized along \( dir_{ij}^{t-1} \).
  – Set \( \frac{dL(w_{ij}^t)}{ds} = 0 \) to find the optimal \( s \).
Figure 6: The method of Steepest Descent. (a) Starting at $[-2, -2]^T$, take a step in the direction of steepest descent of $f$. (b) Find the point on the intersection of these two surfaces that minimizes $f$. (c) This parabola is the intersection of surfaces. The bottommost point is our target. (d) The gradient at the bottommost point is orthogonal to the gradient of the previous step.
Gradient Descent with Line Search

\[ w_{ij}^t = w_{ij}^{t-1} + s \, \text{dir}_{ij}^{t-1} \]

- Set \( \frac{dL(w_{ij}^t)}{ds} = 0 \) to find the optimal \( s \).

\[
\frac{dL(w_{ij}^t = w_{ij}^{t-1} + s \, \text{dir}_{ij}^{t-1})}{ds} = \frac{dL}{dw_{ij}^t} \frac{dw_{ij}^t}{ds} = \frac{dL}{dw_{ij}^t} \text{dir}_{ij}^{t-1} = 0
\]

\[
\frac{dL}{dw_{ij}^t} \frac{dw_{ij}^t}{ds} = \frac{dL}{dw_{ij}^t} \text{dir}_{ij}^{t-1} = 0
\]

- Interpretation:
  - Choose \( s \) such that: the gradient direction at the new position is orthogonal to the current direction
- This is called **steepest gradient descent**
- Problem: makes zig-zag
Figure 8: Here, the method of Steepest Descent starts at $[-2, -2]^T$ and converges at $[2, -2]^T$. 
Conjugate Gradient Descent

• Motivation
Conjugate Gradient Descent

- Two vectors are conjugate (A-orthogonal) if:
  \[ u^T A v = 0 \]
- We assume that the error surface has the quadratic form:
  \[ f(x) = \frac{1}{2} x^T A x - b^T x + c \]

Figure 22: These pairs of vectors are A-orthogonal... because these pairs of vectors are orthogonal.
Conjugate Gradient Descent

\[ \text{dir}_{ij}^t = - \frac{\partial E(w_{ij}^t)}{\partial w_{ij}^t} + \beta \text{dir}_{ij}^{t-1} \]

By assuming quadratic form etc.:

\[ \beta = \frac{\sum_{i,j} \left( \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} - \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)} \right) \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)}}{\sum_{i,j} \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)} \cdot \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}} \]
Conjugate Gradient Descent

• Or simply as:

\[ \beta = \frac{(\nabla E_{\text{new}} - \nabla E_{\text{old}}) \cdot \nabla E_{\text{new}}}{(\nabla E_{\text{old}})^2} \]

• Interpretation:
  – Rewrite this as:

\[ \beta = \frac{\nabla E_{\text{new}}^2}{\nabla E_{\text{old}}^2} - \frac{\nabla E_{\text{old}} \cdot \nabla E_{\text{new}}}{\nabla E_{\text{old}}^2} \]

• For more detailed motivation and derivations, see:
Steepest and Conjugate Gradient Descent: Cons and Pros

• **Pros:**
  – Faster to converge than, e.g., stochastic gradient descent (even mini-batch)

• **Cons:**
  – They don’t work well on saddle points
  – Computationally more expensive
  – In 2D:
    • Steepest descent is $O(n^2)$
    • Conjugate descent is $O(n^{3/2})$

Online Interactive Tutorial

http://www.benfrederickson.com/numerical-optimization/
• http://bair.berkeley.edu/blog/2017/09/12/learning-to-optimize-with-rl
CHALLENGES OF THE LOSS SURFACE
Challenges

- Local minima
- Saddle points
- Cliffs
- Valleys
Local minima

• Solutions
  – Momentum
    • Make weight update depend on the previous one as well:
      \[ \Delta w_{ij}^t = -\eta \nabla L^t + \alpha \Delta w_{ji}^{t-1} \]
    • \(0 \leq \alpha < 1\): momentum (constant)

  – Incremental update
  – Large training data
  – Adaptive learning rate
  – Good initialization
  – Different minimization strategies
• For smaller networks, local minima are more problematic

• For large-size networks, most local minima are equivalent and yield similar performance on a test set.

• The probability of finding a “bad” (high value) local minimum is non-zero for small-size networks and decreases quickly with network size.

• Struggling to find the global minimum on the training set (as opposed to one of the many good local ones) is not useful in practice and may lead to overfitting.
Do neural nets have saddle points?

- Saxe et al, 2013:
  - neural nets without non-linearities have many saddle points
- all the minima are global
- all the minima form a connected manifold
Do neural nets have saddle points?

- Dauphin et al 2014: Experiments show neural nets do have as many saddle points as random matrix theory predicts.
- Choromanska et al 2015: Theoretical argument for why this should happen.
- Major implication: most minima are good, and this is more true for big models.
- Minor implication: the reason that Newton’s method works poorly for neural nets is its attraction to the ubiquitous saddle points.
Valleys, Cliffs and Exploding Gradients

Figure 8.1: One theory about the neural network optimization is that poorly conditioned Hessian matrices cause much of the difficulty in training. In this view, some directions have a high curvature (second derivative), corresponding to the quickly rising sides of the valley (going left or right), and other directions have a low curvature, corresponding to the smooth slope of the valley (going down, dashed arrow). Most second-order methods, as well as momentum or gradient averaging methods are meant to address that problem, by increasing the step size in the direction of the valley (where it pays off the most in the long run to go) and decreasing it in the directions of steep rise, which would otherwise lead to oscillations (blue full arrows). The objective is to smoothly go down, staying at the bottom of the valley (green dashed arrow).
Figure 8.2: Contrary to what is shown in Figure 8.1, the objective function for highly non-linear deep neural networks or for recurrent neural networks is typically not made of symmetrical sides. As shown in the figure, there are sharp non-linearities that give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly ruining a lot of the optimization work that had been done. Figure graciously provided by Razvan Pascanu (Pascanu, 2014).
Figure 8.3: To address the presence of cliffs such as shown in Figure 8.2, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is above a threshold (which is a hyperparameter, although not a very critical one). Using such a gradient clipping heuristic (dotted arrows trajectories) helps to avoid the destructive big moves which would happen when approaching the cliff, either from above or from below (bold arrows trajectories). Figure graciously provided by Razvan Pascanu (Pascanu, 2014).
USING MOMENTUM TO IMPROVE STEPS
Momentum

• Maintain a “memory”

\[ \Delta w(t + 1) \leftarrow \mu \Delta w(t) - \eta \nabla L \]

where \( \mu \) is called the momentum weight

• Momentum filters oscillations on gradients (i.e., oscillatory movements on the error surface)

• \( \mu \) is typically initialized to 0.9.
  – It is better if it anneals from 0.5 to 0.99 over multiple epochs
Momentum

Figure 8.5: The effect of momentum on the progress of learning. Momentum acts to accumulate gradient contributions over training iterations. Directions that consistently have positive contributions to the gradient will be augmented.
Nesterov Momentum

• Use a “lookahead” step to update:

\[
\begin{align*}
\mathbf{w}_{\text{ahead}} & \leftarrow \mathbf{w} + \mu \Delta \mathbf{w}(t) \\
\Delta \mathbf{w}(t+1) & \leftarrow \mu \Delta \mathbf{w}(t) - \eta \nabla L_{\text{ahead}} \\
\mathbf{w} & \leftarrow \mathbf{w} + \Delta \mathbf{w}(t+1)
\end{align*}
\]

Nesterov Momentum
(alternative formulation)

Classical Momentum

\[ v_{t+1} = \mu_t v_t - \alpha_t J' (\theta_t) \]
\[ \theta_{t+1} = \theta_t + v_{t+1} \]

Nesterov’s Momentum

\[ y_{t+1} = (1 + \mu_t) \theta_t - \mu_t \theta_{t-1} \]
\[ \theta_{t+1} = y_{t+1} - \alpha_t J' (y_{t+1}) \]

- Uses smoothed weights
- Uses future gradient to update
- Guaranteed optimal convergence rate for convex functions (if first-order gradient based methods are used)


Momentum vs. Nesterov Momentum

• When the learning rate is very small, they are equivalent.
• When the learning rate is sufficiently large, Nesterov Momentum performs better (it is more responsive).
• See for an in-depth comparison:

On the importance of initialization and momentum in deep learning

Ilya Sutskever
James Martens
George Dahl
Geoffrey Hinton

ILYASU@GOOGLE.COM
JMARTENS@CS.TORONTO.EDU
GDAHL@CS.TORONTO.EDU
HINTON@CS.TORONTO.EDU
Demo (and further reading)

http://distill.pub/2017/momentum/
SETTING THE LEARNING RATE
Alternatives

• Single global learning rate
  – Adaptive Learning Rate
  – Adaptive Learning Rate with Momentum

• Per-parameter learning rate
  – AdaGrad
  – RMSprop
  – Adam
  – AdaDelta
Global Methods: Scheduling the learning rate

- **Step decay**
  - $\eta' \leftarrow \eta \times c$, where $c$ could be 0.5, 0.4, 0.3, 0.2, 0.1 etc.

- **Exponential decay:**
  - $\eta = \eta_0 e^{-kt}$, where $t$ is iteration number
  - $\eta_0, k$: hyperparameters

- **1/t decay:**
  - $\eta = \eta_0 / (1 + kt)$

- If you have time, keep decay small and train longer
Global Methods: Scheduling the learning rate

Figure 2. Triangular learning rate policy. The blue lines represent learning rate values changing between bounds. The input parameter `stepsizes` is the number of iterations in half a cycle.

Global Methods: learning rate & the batch size

• Bigger batch size, bigger learning rate
  – If you increase batch size to kN, learning rate should be scaled by:
    • $\sqrt{k}$ [1]
    • $k$ [2]

• Two interpretations:
  – Bigger batch means more stable gradient => Safer to make large steps.
  – Bigger batch means less number of update steps => increase learning rate to compensate.


Global Methods: learning rate & the batch size

BATCH SIZE:

Global Methods: warm-up

• Start with a small learning rate [1]
  – Constant learning rate
  – Gradually increasing

• Why? The first steps of learning appear to be very critical [2]


Per-parameter Methods: Adagrad

- Higher the gradient, lower the learning rate

- Accumulate square of gradients elementwise (initially $r = 0$):
  
  $$ r \leftarrow r + \left( \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W} \right)^2 $$

- Update each parameter/weight based on the gradient on that:
  
  $$ \Delta W \leftarrow -\frac{\eta}{\sqrt{r}} \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W} $$

---

**Algorithm 8.4 The AdaGrad algorithm**

**Require:** Global learning rate $\epsilon$

**Require:** Initial parameter $\theta$

**Require:** Small constant $\delta$, perhaps $10^{-7}$, for numerical stability

- Initialize gradient accumulation variable $r = 0$
- while stopping criterion not met do
  - Sample a minibatch of $m$ examples from the training set \{$(x^{(1)}, \ldots, x^{(m)})$\} with corresponding targets $y^{(i)}$.
  - Compute gradient: $g \leftarrow \frac{1}{m} \nabla \theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$.
  - Accumulate squared gradient: $r \leftarrow r + g \odot g$.
  - Compute update: $\Delta \theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$. (Division and square root applied element-wise)
  - Apply update: $\theta \leftarrow \theta + \Delta \theta$.
- end while


Algorithm taken from: Goodfellow et al., Deep Learning, 2016.
Per-parameter Methods:

Root-Mean-Squared Propagation (RMSprop)

- Similar to Adagrad. Adagrad uses the whole history of gradients, which can be a limitation when training converges to a nice “bowl”.

- RMSprop handles this by weighted averaging:
  \[ r \leftarrow \rho r + (1 - \rho) \left( \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W} \right)^2 \]

- \( \rho \) is typically \([0.9, 0.99, 0.999]\)

- Update each parameter/weight based on the gradient on that:
  \[ \Delta W \leftarrow -\eta \frac{\sqrt{r}}{\sqrt{\delta + r}} \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W} \]

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate \( \epsilon \), decay rate \( \rho \)
Require: Initial parameter \( \theta \)
Require: Small constant \( \delta \), usually \( 10^{-6} \), used to stabilize division by small numbers

Initialize accumulation variables \( r = 0 \)
while stopping criterion not met do
  Sample a minibatch of \( m \) examples from the training set \( \{x^{(1)}, \ldots, x^{(m)}\} \) with corresponding targets \( y^{(i)} \).
  Compute gradient: \( g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)}) \).
  Accumulate squared gradient: \( r \leftarrow \rho r + (1 - \rho) g \circ g \).
  Compute parameter update: \( \Delta \theta = -\frac{\epsilon}{\sqrt{\delta + r}} \odot g \) \((\frac{1}{\sqrt{\delta + r}} \text{ applied element-wise})\)
  Apply update: \( \theta \leftarrow \theta + \Delta \theta \).
end while

Currently, unpublished. Proposed by Hinton in one of his lectures.

Algorithm taken from: Goodfellow et al., Deep Learning, 2016.
Per-parameter Methods:

RMSprop with Nesterov Momentum

**Algorithm 8.6** RMSProp algorithm with Nesterov momentum

**Require:** Global learning rate $\epsilon$, decay rate $\rho$, momentum coefficient $\alpha$

**Require:** Initial parameter $\theta$, initial velocity $v$

Initialize accumulation variable $r = 0$

while stopping criterion not met do

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

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

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

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

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

Apply update: $\theta \leftarrow \theta + v$.

end while

Algorithm taken from: Goodfellow et al., Deep Learning, 2016.
Per-parameter Methods:
Adaptive Moments (Adam)

- A variation of RMSprop + momentum
- Incorporates first & second order moments
- Bias correction needed to get rid of bias towards zero at initialization

Algorithm 8.7 The Adam algorithm

Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization (Suggested default: $10^{-8}$)

Require: Initial parameters $\theta$
Initialize 1st and 2nd moment variables $s = 0$, $r = 0$
Initialize time step $t = 0$

while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
    Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1) g$
    Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$
    Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$
    Compute update: $\Delta \theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$ (operations applied element-wise)
    Apply update: $\theta \leftarrow \theta + \Delta \theta$
end while

Algorithm taken from:
Goodfellow et al., Deep Learning, 2016.
Comparison

NAG: Nesterov’s Accelerated Gradient

https://twitter.com/alecrad
Comparison

![Comparison Graph]

Legend:
- sgd
- sgd-momentum
- adagrad
- adam
- adadelta
- rmsprop

X-axis: Iterations (0k to 16k)
Y-axis: Loss Values (0.47 to 2.83)
• When SGD+momentum is tuned for hyperparameters, it can outperform Adam etc.
• There are methods that try to finetune the hyper-parameters:

**YellowFin and the Art of Momentum Tuning**

To sum up

- Different problems seem to favor different per-parameter methods
- Adam seems to perform better among per-parameter adaptive learning rate algorithms
- SGD+Nesterov momentum seems to be a fair alternative