CENG 783 – Deep Learning
Week 4: Artificial neural networks
Notes

• HW
  • Extension to 18th of May.

• Project proposals

• Opportunity to work in a TUBITAK project
Now

• Details of an ANN
  • Activation function
  • Gradient Descent strategies
  • Setting the learning rate
  • Representational capacity
  • Overfitting/generalization
  • When to stop training
Problems of back propagation with sigmoid

• It is extremely slow, if it does converge.
• It may get stuck in a local minima.
• It is sensitive to initial conditions.
• It may start oscillating.
Backprop in deep networks

• Local minima may not be as severe as it is feared
  – If one weight gets into local minima, other weights provide escape routes
  – The more weights, the more escape routes

• Add a momentum term

• Use stochastic gradient descent, rather than true gradient descent
  – This means we have different error surfaces for each data
  – If stuck in local minima in one of them, the others might help

• Train multiple networks with different initial weights
  – Select the best one
Backprop

- Very powerful - can learn any function, given enough hidden units!
- Have the same problems of Generalization vs. Memorization.
  - With too many units, we will tend to memorize the input and not generalize well. Some schemes exist to “prune” the neural network.
- Networks require extensive training, many parameters to fiddle with. Can be extremely slow to train. May also fall into local minima.
- Inherently parallel algorithm, ideal for multiprocessor hardware.
- Despite the cons, a very powerful algorithm that has seen widespread successful deployment.
Now, let us look at alternative aspects

• Loss functions
  – Hinge-loss, softmax loss, squared-error loss, ...
  – We will not look at them here again

• Activation functions
  – Sigmoid, tanh, ReLU, Leaky ReLU, parametric ReLU, maxout

• Backpropagation strategy:
  – True Gradient Descent, Stochastic Gradient Descent, Mini-batch Gradient Descent, RMSprop, AdaDelta, AdaGrad, Adam
Activation Functions
Activation function

• Sigmoid / logistic function

The computational power is increased by the use of a squashing function. In the original paper the logistic function:

\[ f(x) = \frac{1}{1 + e^{-x}} \]

is used.
Activation function

Logistic function $o_{pj} = \frac{1}{1+e^{-net_{pj}}}$ has nice features:

- The derivative is expressable in terms of the function itself:
  \[
  \frac{\partial o_{pj}}{\partial net_{pj}} = o_{pj}(1 - o_{pj})
  \]

- The derivative is a “bump” that pushes uncommitted nodes to change weights.
- Likewise, weights are prevented from blowing up.
- The downside is that it is hard to change large weights!

How about $f(\cdot) = tanh(\cdot)$ which squashes the input into the $[-1 : +1]$?

All we need to make sure it that $f(\cdot)$ is differentiable.
Activation Functions

• sigmoid vs tanh

Derivative: $\sigma(x)(1 - \sigma(x))$

Derivative: $(1 - \tanh^2(x))$
Pros and Cons

• Sigmoid is an historically important activation function
  – But nowadays, rarely used

• Sigmoid drawbacks
  1. It gets saturated, if the activation is close to zero or one
     • This leads to very small gradient, which disallows “transfer”ing 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: \( \tanh(x) = 2\sigma(2x) - 1 \)

Rectified Linear Units (ReLU)

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

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

Derivative: \( 1(x > 0) \)

[Image: Comparison of ReLU and Logistic functions]

[Image: Training error rate over epochs]

[Krizhevsky et al., NIPS12]
ReLU: biological motivation

\[
f(I) = \begin{cases} 
\tau \log \left( \frac{E + RI - V_r}{E + RI - V_{th}} \right) + t_{ref}^{-1}, & \text{if } E + RI > V_{th} \\
0, & \text{if } E + RI \leq V_{th} 
\end{cases}
\]

where \( t_{ref} \) is the refractory period (minimal time between two action potentials), \( I \) the input current, \( V_r \) the resting potential and \( V_{th} \) the threshold potential (with \( V_{th} > V_r \)), and \( R, E, \tau \) the membrane resistance, potential and time constant. The most commonly used activation func-

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).

Rectified Linear Units: Another Perspective

Hinton argues that this is a form of model averaging

A fast approximation

$$\sum_{n=1}^{\infty} \text{logistic}(x + 0.5 - n) = \log(1 + e^x)$$
output = max(0, input)

- Rectified linear units are much faster to compute than the sum of many logistic units.
- They learn much faster than ordinary logistic units and they produce sparse activity vectors.
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
ReLU

• See the following site for more in-depth analysis

http://www.jefkine.com/general/2016/08/24/formulating-the-relu/
Leaky ReLU

- $f(x) = \mathbf{1}(x < 0)(\alpha x) + \mathbf{1}(x \geq 0)(x)$
  - 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
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.
Softplus

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

- Its derivative is the sigmoid function:
  \[ f'(x) = \frac{1}{1 + e^{-x}} \]
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.”

Searching for Activation Functions

Prajit Ramachandran, Barret Zoph, Quoc V. Le
Google Brain
{prajit, barretzoph, qvl}@google.com
Activation Functions: To sum up

• Don’t use sigmoid
• If you really want, use tanh but it is worse than ReLU and its variants
• ReLU: be careful about dying neurons
• Leaky ReLU and Maxout: Worth trying
DEMO

- [Link](http://playground.tensorflow.org/#activation=tanh&regularization=L2&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.24725&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification)
Interactive introductory tutorial

BACK PROPAGATION / MINIMIZATION STRATEGIES
Schemes of training

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

• Curricular training
Stochastic Gradient Descent

Batch Gradient Descent
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 \textit{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 \textit{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)
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 \( \delta \):

\[
\frac{d}{d\delta}\left[f(x) + f'(x)\delta + \frac{1}{2} f''(x)\delta^2\right] = 0
\]

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 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 \( \times \) 1 million \( \Rightarrow \sim 3725 \text{ GB RAM} \)
  – Alternatives exist to get around the memory problem (quasi-Newton methods, Limited-memory BFGS)

• Active research area \( \Rightarrow \) A suitable project topic ☺️
RPROP (Resilience Propagation)

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

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

(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
RPROP (Resilience Propagation)

For all weights and biases:

\[
\text{if } (\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) > 0) \text{ then } \{
\Delta_{ij}(t) = \text{minimum } (\Delta_{ij}(t-1) * \eta^+, \Delta_{max}) \\
\Delta w_{ij}(t) = - \text{sign } (\frac{\partial E}{\partial w_{ij}}(t)) * \Delta_{ij}(t) \\
w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)
\}\n\]

\[
\text{else if } (\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) < 0) \text{ then } \{
\Delta_{ij}(t) = \text{maximum } (\Delta_{ij}(t-1) * \eta^-, \Delta_{min}) \\
w_{ij}(t+1) = w_{ij}(t) - \Delta w_{ij}(t-1) \\
\frac{\partial E}{\partial w_{ij}}(t) = 0
\}\n\]

\[
\text{else if } (\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) = 0) \text{ then } \{
\Delta w_{ij}(t) = - \text{sign } (\frac{\partial E}{\partial w_{ij}}(t)) * \Delta_{ij}(t) \\
w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)
\}\n\]

A Direct Adaptive Method for Faster Backpropagation Learning: 
The RPROP Algorithm

Martin Riedmiller  Heinrich Braun
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

\[ w_{ij}^t = w_{ij}^{t-1} + s \, dir_{ij}^{t-1} \]

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

\[ \frac{dL(w_{ij})}{ds} = \frac{dL}{dw_{ij}} \frac{dw_{ij}}{ds} = \frac{dL}{dw_{ij}} \, dir_{ij}^{t-1} = 0 \]

\[ \frac{dL}{dw_{ij}^t} \frac{dw_{ij}^t}{ds} = \frac{dL}{dw_{ij}^t} \, dir_{ij}^{t-1} = 0 \]

- 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.](image)
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.:

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

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

  – If the new direction suggests a radical turn, rely more on the old direction!

• 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
Genetic Algorithms

General strategy:
- Randomly choose weights and encode them on a string of bits (chromosomes).
- Determine a “fitness” function (e.g. error function).
- Use genetic operators *mutation, crossover* to construct new strings.
- Use “survival of the fittest” to produce better and better strings.

Some observations/comments:
- Selection of fitness and operators is crucial to its effectiveness.
- Search is global, not fooled by local minima.
- Fitness (or error in this case) function need not be differentiable.
- Search is rather blind, since it does not use the $\nabla$ info.
- It can be a good method for initialization, to be used for a gradient method.

Slide credit: E. Sahin
CHALLENGES OF THE ERROR SURFACE
Challenges

• Local minima
• Saddle points
• Cliffs
• Valleys
Local minima

• Solutions
  – Momentum
    • Make weight update depend on the previous one as well:
      \[ \Delta w_{ij}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n - 1) \]
    • \( 0 \leq \alpha < 1 \): 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).
Valleys, Cliffs and Exploding Gradients

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).
Valleys, Cliffs and Exploding Gradients

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”
  \[ \Delta w(t + 1) \leftarrow \mu \Delta w(t) - \eta \nabla E \]
  where \( \mu \) is called the momentum term

• 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
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:
  \[ w_{\text{ahead}} \leftarrow w + \mu \Delta w(t) \]
  \[ \Delta w(t + 1) \leftarrow \mu \Delta w(t) - \eta \nabla E_{\text{ahead}} \]
  \[ w \leftarrow w + \Delta w(t + 1) \]

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
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
Annealing the learning rate (Global)

- **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
Adagrad (Per parameter)

- 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 $\eta$.
**Require:** Initial parameter $\theta$

Initialize gradient accumulation variable $r = 0$.

**while** Stopping criterion not met **do**

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

Set $g = 0$

for $i = 1$ to $m$ **do**

Compute gradient: $g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$

end for

Accumulate gradient: $r \leftarrow r + g^2$ (square is applied element-wise)

Compute update: $\Delta \theta \leftarrow -\frac{\eta}{\sqrt{r}} g$. % ($\frac{1}{\sqrt{r}}$ applied element-wise)

Apply update: $\theta \leftarrow \theta + \Delta \theta$

end while
RMSprop (Per parameter)

- Similar to Adagrad
- Calculates a moving average of square of the gradients
- Accumulate square of gradients (initially \( r = 0 \)):

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

- \( \rho \) is typically \([0.9, 0.99, 0.999]\)
- Update each parameter/weight based on the gradient on that:

\[
\Delta W \leftarrow -\frac{\eta}{\sqrt{r}} \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W}
\]

---

**Algorithm 8.5 The RMSprop algorithm**

**Require**: Global learning rate \( \eta \), decay rate \( \rho \).

**Require**: Initial parameter \( \theta \)

Initialize accumulation variables \( r = 0 \)

while Stopping criterion not met do

Sample a minibatch of \( m \) examples from the training set \( \{x^{(1)}, \ldots, x^{(m)}\} \).

Set \( g = 0 \)

for \( i = 1 \) to \( m \) do

    Compute gradient: \( g \leftarrow g + \nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)}) \)

end for

Accumulate gradient: \( r \leftarrow \rho r + (1 - \rho)g^2 \)

Compute parameter update: \( \Delta \theta = -\frac{\eta}{\sqrt{r}} \odot g \) \% \( \frac{1}{\sqrt{r}} \) applied element-wise

Apply update: \( \theta \leftarrow \theta + \Delta \theta \)

end while

Currently, unpublished.
Algorithm 8.6 RMSprop algorithm with Nesterov momentum

Require: Global learning rate $\eta$, 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 $\{x^{(1)}, \ldots, x^{(m)}\}$.
Compute interim update: $\theta \leftarrow \theta + \alpha v$
Set $g = 0$

for $i = 1$ to $m$ do

Compute gradient: $g \leftarrow g + \nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$

end for

Accumulate gradient: $r \leftarrow \rho r + (1 - \rho) g^2$
Compute velocity update: $v \leftarrow \alpha v - \frac{\eta}{\sqrt{r}} \odot g$. \hspace{1em} \% (\frac{1}{\sqrt{r}} \text{ applied element-wise})
Apply update: $\theta \leftarrow \theta + v$

end while
Adam (per parameter)

- Similar to 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 $\alpha$

**Require:** Decay rates $\rho_1$ and $\rho_2$, constant $\epsilon$

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

Initialize 1st and 2nd moment variables $s = 0$, $r = 0$,
Initialize timestep $t = 0$

**while** Stopping criterion not met **do**

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

Set $g = 0$

**for** $i = 1$ to $m$ **do**

Compute gradient: $g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$

end **for**

$t \leftarrow t + 1$

Get biased first moment: $s \leftarrow \rho_1 s + (1 - \rho_1)g$

Get biased second moment: $r \leftarrow \rho_2 r + (1 - \rho_2)g^2$

Compute bias-corrected first moment: $\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$

Compute bias-corrected second moment: $\hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$

Compute update: $\Delta \theta = -\alpha \frac{s}{\sqrt{r} + \epsilon} g$  \% (operations applied element-wise)

Apply update: $\theta \leftarrow \theta + \Delta \theta$

end **while**
Adadelta (per parameter)

• Incorporates second-order gradient information

Algorithm 8.8 The Adadelta algorithm

Require: Decay rate $\rho$, constant $\epsilon$
Require: Initial parameter $\theta$

Initialize accumulation variables $r = 0$, $s = 0$, 
while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set \{x^{(1)}, \ldots, x^{(m)}\}.
    Set $g = 0$
    for $i = 1$ to $m$ do
        Compute gradient: $g \leftarrow g + \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$
    end for
    Accumulate gradient: $r \leftarrow \rho r + (1 - \rho)g^2$
    Compute update: $\Delta \theta = -\sqrt{\frac{s + \epsilon}{r + \epsilon}} g$ \% (operations applied element-wise)
    Accumulate update: $s \leftarrow \rho s + (1 - \rho) [\Delta \theta]^2$
    Apply update: $\theta \leftarrow \theta + \Delta \theta$
end while
Comparison

NAG: Nesterov’s Accelerated Gradient
Comparison
• 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