

CS-568 Deep Learning

Backpropagation and Vanishing Gradients

Nazar Khan

Department of Computer Science

University of the Punjab

Backpropagation

Learning Algorithm

1. Forward propagate the input vector \mathbf{x}_n to compute *and store* activations and outputs of every neuron in every layer.
2. Evaluate $\delta_k = \frac{\partial L_n}{\partial a_k}$ for every neuron in output layer.
3. Evaluate $\delta_j = \frac{\partial L_n}{\partial a_j}$ for every neuron in *every* hidden layer via backpropagation.

$$\delta_j = h'(a_j) \sum_{k=1}^K \delta_k w_{kj}$$

4. Compute derivative of each weight $\frac{\partial L_n}{\partial w}$ via $\delta \times \text{input}$.
5. Update each weight via gradient descent $w^{\tau+1} = w^\tau - \eta \frac{\partial L_n}{\partial w}$.

Tanh

$A(-1, 1)$ sigmoidal function

- ▶ Since range of logistic sigmoid $\sigma(a)$ is $(0, 1)$, we can obtain a function with $(-1, 1)$ range as $2\sigma(a) - 1$.
- ▶ Another related function with $(-1, 1)$ range is the **tanh** function.

$$\tanh(a) = 2\sigma(2a) - 1 = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

where σ is applied on $2a$.

- ▶ Preferred¹ over logistic sigmoid as activation function $h(a)$ of hidden neurons.
- ▶ Just like the logistic sigmoid, derivative of $\tanh(a)$ is simple: $1 - \tanh^2(a)$. (Prove it.)

¹LeCun et al., 'Efficient backprop'.

A Simple Example

- ▶ Two-layer MLP for multivariate regression from $\mathbb{R}^D \rightarrow \mathbb{R}^K$.
- ▶ Linear outputs $y_k = a_k$ with half-SSE $L = \frac{1}{2} \sum_{k=1}^K (y_k - t_k)^2$.
- ▶ M hidden neurons with $\tanh(\cdot)$ activation functions.

Forward propagation

$$a_j = \sum_{i=0}^D w_{ji}^{(1)} x_i$$

$$z_j = \tanh(a_j)$$

$$z_0 = 1$$

$$y_k = \sum_{j=0}^M w_{kj}^{(2)} z_j$$

$$\delta_k = y_k - t_k$$

- ▶ Compute derivatives $\frac{\partial L}{\partial w_{ji}^{(1)}} = \delta_j x_i$ and $\frac{\partial L}{\partial w_{kj}^{(2)}} = \delta_k z_j$.

Backpropagation

$$\delta_j = (1 - z_j^2) \sum_{k=1}^K w_{kj}^{(2)} \delta_k$$

Backpropagation

Verifying Correctness

- ▶ Any implementation of analytical derivatives (not just backpropagation) must be compared with numerical derivatives.
- ▶ *Numerical derivatives* can be computed via finite *central differences*

$$\frac{\partial L_n}{\partial w_{ji}} = \frac{L_n(w_{ji} + \epsilon) - L_n(w_{ji} - \epsilon)}{2\epsilon} + O(\epsilon^2)$$

- ▶ *Analytical derivatives* computed via backpropagation **must be compared** with numerical derivatives for a few examples to verify correctness.

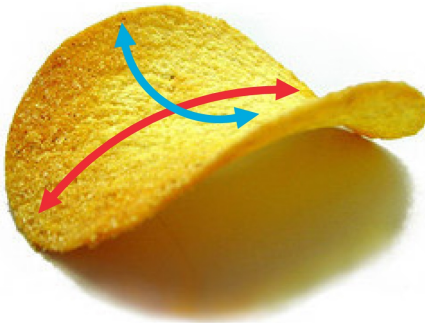
Backpropagation

Efficiency

- ▶ Notice that we could have avoided backpropagation and computed all required derivatives numerically.
- ▶ But cost of numerical differentiation is $O(|W|^2)$.
 - ▶ Two fprops per weight and each fprop has $O(|W|)$ cost. [Why?](#)
- ▶ While cost of backpropagation is $O(|W|)$.

Neural Networks and Stationary Points

- ▶ For optimisation, we notice that W^* must be a *stationary point* of $L(W)$.
 - ▶ Minimum, maximum, or saddle point.
 - ▶ A saddle point is where gradient vanishes but point is not an extremum.



Neural Network training finds local minimum

- ▶ The goal in neural network minimisation is to find a local minimum.
- ▶ A global minimum, *even if found*, cannot be verified as globally minimum.
- ▶ Due to symmetry, there are multiple equivalent local minima.
- ▶ Reaching *any suitable* local minimum is the goal of neural network optimisation.
- ▶ Since there are no analytical solutions for W^* , we use iterative, numerical procedures.

Optimisation Options

- ▶ Options for iterative optimisation
 - ▶ Online methods (using partial training data)
 - ▶ Stochastic gradient descent
 - ▶ Stochastic gradient descent using mini-batches
 - ▶ Batch methods (using all training data)
 - ▶ Batch gradient descent
 - ▶ Conjugate gradient descent
 - ▶ Quasi-Newton methods

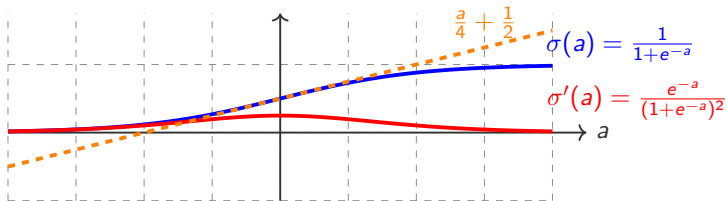
Online Methods

- ▶ Online methods converge faster since parameter updates are more frequent.
- ▶ Have greater chance of escaping local minima because stationary point w.r.t to whole data set will generally not be a stationary point w.r.t an individual data point.

Batch Methods

- ▶ Batch methods are practical for small datasets only.
- ▶ Deep Learning datasets are increasingly becoming larger and larger.
- ▶ Conjugate gradient descent and quasi-Newton methods
 - ▶ are more robust and faster than batch gradient descent, and
 - ▶ decrease loss at each iteration until arriving at a minimum.

Problems with sigmoidal neurons



- ▶ For large $|a|$, sigmoid value approaches either 0 or 1. This is called *saturation*.
- ▶ When the sigmoid saturates, the gradient approaches zero.
- ▶ Neurons with sigmoidal activations stop learning when they saturate.
- ▶ When they are not saturated, they are **almost linear**.
- ▶ There is another reason for the gradient to approach zero during backpropagation.

Vanishing Gradient

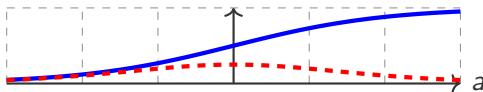
- ▶ Notice that gradient of the sigmoid is always between 0 and $\frac{1}{4}$.
- ▶ Now consider the backpropagation equation.

$$\delta_j = \underbrace{h'(a_j)}_{\leq \frac{1}{4}} \sum_{k=1}^K w_{kj} \delta_k$$

where δ_k will also contain *at least* one factor of $\leq \frac{1}{4}$.

- ▶ This means that values of δ_j keep getting smaller as we backpropagate towards the early layers.
- ▶ Since gradient = $\delta \times$ input, the gradients also keep getting smaller for the earlier layers. Known as the *vanishing gradient* problem.
- ▶ *Therefore, while the network might be deep, learning will not be deep.*

Logistic Sigmoid



Activation function

$$y(a) = \frac{1}{1+e^{-a}}$$

Derivative

$$y'(a) = y(a)(1 - y(a))$$

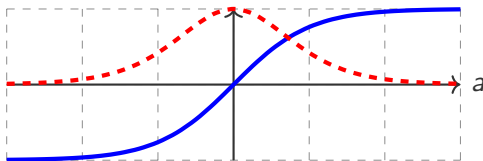
Maximum magnitude of derivative

$$\frac{1}{4}$$

Problem

Cause vanishing gradients

Hyperbolic Tangent



Activation function

$$y(a) = \tanh(a)$$

Derivative

$$y'(a) = 1 - y^2(a)$$

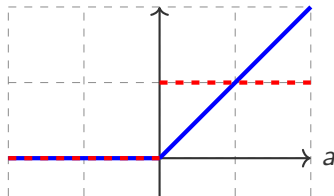
Maximum magnitude of derivative

1

Problem

Cause vanishing gradients

Rectified Linear Unit (ReLU)



Activation function $y(a) = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{if } a \leq 0 \end{cases}$

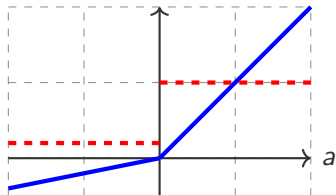
Derivative $y'(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a \leq 0 \end{cases}$

Advantage Avoids vanishing gradients

Problem Dead neurons²

²This can be an advantage as well since death implies fewer neurons.

Leaky ReLU



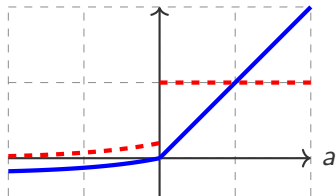
Activation function $y(a) = \begin{cases} a & \text{if } a > 0 \\ ka & \text{if } a \leq 0 \end{cases}$

where $0 \leq k \leq 1$

Derivative $y'(a) = \begin{cases} 1 & \text{if } a > 0 \\ k & \text{if } a \leq 0 \end{cases}$

Advantage Neuron is always learning

Exponential Linear Unit (ELU)



Activation function

$$y(a) = \begin{cases} a & \text{if } a > 0 \\ k(e^a - 1) & \text{if } a \leq 0 \end{cases}$$

where $k > 0$

Derivative

$$y'(a) = \begin{cases} 1 & \text{if } a > 0 \\ y(a) + k & \text{if } a \leq 0 \end{cases}$$

Maximum magnitude of derivative

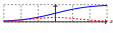
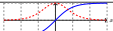
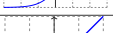

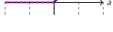
1

Advantage

Neuron is mostly learning

Activation Functions

Summary

Name	$y(a)$	Plot	$y'(a)$	Comments
Logistic sigmoid	$\frac{1}{1+e^{-a}}$		$y(a)(1 - y(a))$	Vanishing gradients
Hyperbolic tangent	$\tanh(a)$		$1 - y^2(a)$	Vanishing gradients
Rectified Linear Unit (ReLU)	$\begin{cases} a & \text{if } a > 0 \\ 0 & \text{if } a \leq 0 \end{cases}$		$\begin{cases} 1 \\ 0 \end{cases}$	Dead neurons. Sparsity.
Leaky ReLU	$\begin{cases} a & \text{if } a > 0 \\ ka & \text{if } a \leq 0 \end{cases}$		$\begin{cases} 1 \\ k \end{cases}$	$0 < k < 1$
Exponential Linear Unit (ELU)	$\begin{cases} a & \text{if } a > 0 \\ k(e^a - 1) & \text{if } a \leq 0 \end{cases}$		$\begin{cases} 1 \\ y(a) + k \end{cases}$	$k > 0.$

- ▶ Saturated sigmoidal neurons stop learning. Piecewise-linear units keep learning by avoiding saturation.
- ▶ ELU has been shown to lead to better accuracy and faster training.
- ▶ *Take home message:* For hidden neurons, use a member of the LU family. They avoid *i)* saturation and *ii)* the vanishing gradient problem.