Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. Matthias Schubert
Maximilian Bernhard, Niklas Strauß

Munich, 04.11.2024

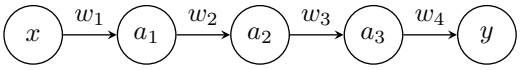
Deep Learning and Artificial Intelligence

WS 2024/25

Exercise 3: Computational Graphs and Vanishing Gradients

Exercise 3-1 Vanishing Gradients Problem

Consider a network with input $x \in \mathbb{R}$, 3 hidden layers each having only one node, and one output $y \in \mathbb{R}$:



In the network each node corresponds to a nonlinear function of the preceding node multiplied with some weight: $a_i = \sigma(w_i \cdot a_{i-1}),\ i = 1, \dots 4$, with $\sigma(x) = \frac{1}{1+e^{-x}}$ where a_0 corresponds to the input x and a_4 corresponds to the output y.

Task:

(a) By using the chain rule, calculate the derivative $\frac{\partial y}{\partial x}$!
$\Big[$Solution: $\frac{\partial y}{\partial x} = \sigma'(z_4)\ w_4 \cdot \sigma'(z_3)\ w_3 \cdot \sigma'(z_2)\ w_2 \cdot \sigma'(z_1)\ w_1\Big]$

(b) Calculate the maximum of the derivative $\sigma' = \frac{\partial}{\partial z}\ \frac{1}{1+e^{-z}}$!
Hint: The derivative can be written as $\sigma' = \sigma(1-\sigma)$. $\big[$Solution: $\frac{1}{4}\big]$

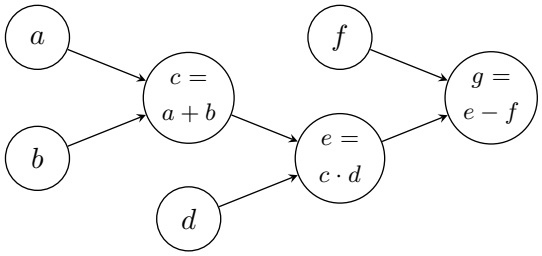
(c) How does this result relate to the vanishing gradients problem?

(d) How can we avoid the vanishing gradients problem during weight initilization?

(e) What are advantages of using the ReLU activation funcition instead of s-shaped ones? What could be disadvantages?

Exercise 3-2 Computational Graphs

Computational graphs are directed graphs that represent the dependencies between the variables and operations within a model or, more generally, a mathematical expression. As an example, consider the expression: $g = (a + b) \cdot d - f$. To build the computational graph for this example we represent each of the operations as well as all of the input variables as nodes and draw an arrow from one node to another if the first is the input to the latter (see figure below). Such a node is called *gate* or *layer* in common. Note that we introduced 2 intermediate variables c and e so that every node has a name.

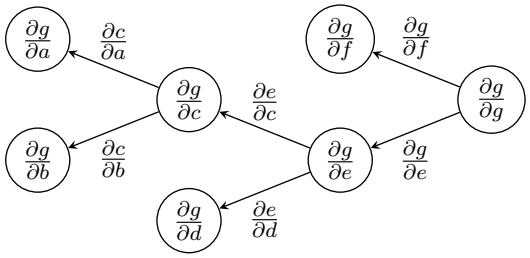


Computational graphs are used by popular deep learning frameworks like Theano and Tensorflow in order to optimize execution, for example, through parallelizing or fusing calculations.

Task: Given an input $\mathbf{x} \in \mathbb{R}^2$, a weight vector $\mathbf{w} \in \mathbb{R}^2$ and a bias $p_0 \in \mathbb{R}$. Draw the computation graph for the mean squared error $L = MSE(\hat{y}, y)$ of a prediction $\hat{y} = \sigma(\mathbf{w}^T\mathbf{x} + p_0)$ [1] with respect to the true value y.

Exercise 3-3 Derivatives on Computational Graphs

Most deep learning frameworks provide an automatic differentiation procedure to compute the gradients based on the backpropagation algorithm introduced in the lecture. Those gradients can be written as a computational graph as well. Consider the example from exercise 2 again. The computational graph for the gradients (with respect to g) would look as follows:



Task:

(a) Given $\mathbf{x} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $\mathbf{w} = \begin{pmatrix} \frac{4}{5} \\ -\frac{7}{5} \end{pmatrix}$, $p_0 = \frac{3}{5}$, $y = 1$ and the loss function $L = (\hat{y} - y)^2$. Calculate the missing values in the computation graph of exercise 2.

$\left[\text{Solution: } p_1 = \frac{4}{5},\ p_2 = -\frac{7}{5},\ z = 0,\ \hat{y} = \frac{1}{1+e^{-0}} = \frac{1}{2},\ L = (\frac{1}{2} - 1)^2 = \frac{1}{4}\right]$

(b) Draw the corresponding computational gradient graph for the computational graph from exercise 2.

(c) Calculate the gradient values for each edge and node in the computational gradient graph.

Exercise 3-4 Computational Graphs in Python

In this exercise you will implement a computational graph in python. For this purpose please use the corresponding jupyter notebook for this exercise. There you will find a template for the implementation of an abstract gate. Every gate has a set of inputs (input_nodes) and consumers. Additionally a gate has to implement the methods **forward** and **backward**. The **forward** method computes the result with respect to the given input nodes (use the *out* field) of the input gates and stores the value in the field *out*. The **backward** function computes and propagates the gradient for the given gate. On call of the **backward** function, the gate uses the incoming gradient *dz* and adds to all input nodes the corresponding gradient. In the template you will find two input gates (*InputGate* and *AddGate*) as simple example.

[1] The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ is often used in logistic regression and binary classification tasks.

In addition, the template provides a *ComputationalGraph* class. This class implements the **backward** and **forward** function as well, but for the whole graph. Both methods return a graphviz object visualizing the respective steps. To draw the computational graphs in jupyter notebook you can for instance use the imported *display* function.

(a) Implement a gate that represents a weight (*WeightGate*). The constructor shall take the parameter α that represents the learning rate of this weight.

(b) Implement a gate that multiplies the outputs of a set of input gates (*MultiplyGate*).

(c) Implement a sigmoid gate that computes the sigmoid σ of one input (*SigmoidGate*). *Hint:* The derivative can be written as $\sigma' = \sigma(1 - \sigma)$.

(d) Implement a gate (*SquaredLossGate*) with the following loss function $L(y, \hat{y}) = (\hat{y} - y)^2$.

(e) Build the computational graph from exercise 3-1 in python and compute display the computational graph after forward and backward. *Hint:* You can validate your calculation with this implementation.

(f) Construct and train a computational graph / network that can classify the XOR dataset with stochastic gradient descent and the already implemented squared loss function.