Exercise 3: Computational Graphs and Vanishing Gradients

Exercise 3-1 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.

\[
\begin{align*}
    a & \rightarrow c = a + b \\
    b & \rightarrow c \\
    d & \rightarrow e = c \cdot d \\
    e & \rightarrow f = e - f \\
    f & \rightarrow g = f
\end{align*}
\]

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 \( x \in \mathbb{R}^2 \), a weight vector \( w \in \mathbb{R}^2 \) 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 \( \hat{y} = \sigma(w^T x + p_0) \)\(^1\) with respect to the true value \( y \).

**Solution:**

\[
\begin{align*}
    p_0 & \rightarrow z = p_0 + p_1 + p_2 \\
    x_1 & \rightarrow p_1 = w_1 x_1 \\
    w_1 & \rightarrow p_1 \\
    x_2 & \rightarrow p_2 = w_2 x_2 \\
    w_2 & \rightarrow p_2 \\
    \hat{y} & \rightarrow y = \sigma(z) \\
    L & \rightarrow L = MSE(\hat{y}, y)
\end{align*}
\]

\(^{1}\)The sigmoid function \( \sigma(z) = \frac{1}{1 + e^{-z}} \) is often used in logistic regression and binary classification tasks.
Exercise 3-2  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 1 again. The computational graph for the gradients (with respect to g) would look as follows:

![Computational Graph](image)

Task:

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

Solution: \( p_1 = \frac{4}{5}, p_2 = -\frac{7}{5}, z = 0, \hat{y} = \frac{1}{1+e^{-w}} = \frac{1}{2}, L = (\frac{1}{2} - 1)^2 = \frac{1}{4} \).

(b) Draw the corresponding computational gradient graph for the example in exercise 1.

Solution:

![Gradient Graph](image)

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

Solution:
\[ \frac{\partial L}{\partial L} = 1; \quad \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y) = -1; \quad \frac{\partial L}{\partial y} = -2(\hat{y} - y) = 1; \]

\[ \frac{\partial \hat{y}}{\partial z} = \frac{1}{(1 + e^{-z})} = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{e^0}{(1 + e^0)^2} = \frac{1}{4}; \]

\[ \frac{\partial L}{\partial z} = -\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = -\frac{1}{4}; \]

\[ \frac{\partial z}{\partial \partial p_0} = \frac{\partial z}{\partial p_1} = \frac{\partial z}{\partial p_2} = 1; \]

\[ \frac{\partial p_1}{\partial x_1} = w_1 = \frac{4}{5}; \quad \frac{\partial p_2}{\partial x_2} = w_2 = -\frac{7}{5}; \]

\[ \frac{\partial p_1}{\partial w_1} = x_1 = 1; \quad \frac{\partial p_2}{\partial w_2} = x_2 = 1; \]

\[ \frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial p_1} \cdot \frac{\partial p_1}{\partial x_1} = -\frac{1}{4} \cdot \frac{4}{5} = -\frac{1}{5}; \quad \frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial p_2} \cdot \frac{\partial p_2}{\partial x_2} = -\frac{1}{4} \cdot \left(-\frac{7}{5}\right) = \frac{7}{20}; \]

\[ \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial p_1} \cdot \frac{\partial p_1}{\partial w_1} = -\frac{1}{4} \cdot 1 = -\frac{1}{4} = \frac{\partial L}{\partial w_2}; \]

(Results can just be drawn into the graph from exercise 2b))

**Exercise 3-3 Computational Graphs in Python**

In this exercise you will implement a computational graph in python. For this purpose please use the corresponding jupyter notebook from the lecture website. 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.

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 $\alpha$ 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 $\sigma$ 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.

**Exercise 3-4   Vanishing Gradients Problem**

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

![Computational Graph](attachment:image.png)

In the network each node corresponds to the sigmoid of the preceding node multiplied with some weight:

$$a_i = \sigma(w_i \cdot a_{i-1}), \ i = 1, \ldots, 4,$$

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 gradient $\frac{\partial y}{\partial x}$! 

**Solution:** Applying the chain rule we get

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a_4} \frac{\partial a_4}{\partial a_3} \frac{\partial a_3}{\partial a_2} \frac{\partial a_2}{\partial a_1} \frac{\partial a_1}{\partial x}.$$ 

For the hidden layers we have

$$\frac{\partial y}{\partial a_4} = \sigma'(z_4) \ w_4.$$ 

Thus:

$$\frac{\partial y}{\partial x} = \sigma'(z_4) \ w_4 \cdot \sigma'(z_3) \ w_3 \cdot \sigma'(z_2) \ w_2 \cdot \sigma'(z_1) \ w_1.$$ 

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

**Solution:**
\[ \sigma' = \frac{e^{-z}}{(1+e^{-z})^2} \]

We want to find the maximum of \( \sigma' = \sigma(1 - \sigma) \). Thus we need to set its derivative to zero:

\[ \sigma'' = \sigma'(1 - \sigma) + \sigma(-\sigma') \]

\[ = \sigma'(1 - 2\sigma) \]

\[ \overset{!}{=} 0 \]

Since \( \sigma' > 0 \) for all \( z \), the second term has to be zero:

\[ (1 - 2\sigma) = 0 \]

\[ \iff \]

\[ \sigma = \frac{1}{1 + e^{-z}} = \frac{1}{2} \]

\[ \iff \]

\[ z = 0 \]

Finally, we calculate \( \sigma'(0) = \frac{e^{0}}{(1+e^{0})^2} = \frac{1}{4} \). This is the maximal value \( \sigma' \) can take.

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

Solution:

If we use a uniform distribution over the interval \([0, 1]\) or a normal distribution with mean 0 and standard deviation 1 to initialize the weights in a network, then most of the factors in the product \( \frac{\partial y}{\partial x} = \sigma'(z_4) \ w_4 \cdot \sigma'(z_3) \ w_3 \cdot \sigma'(z_2) \ w_2 \cdot \sigma'(z_1) \ w_1 \) will satisfy: \( \left| \sigma'(z_j)w_j \right| \leq \frac{1}{4} \). Multiplying many such terms (as it is the case in deep networks), its value will exponentially decrease, i.e. the gradients will become very small and learning will become very slow or practically stop. This is also the case for other (squashing) activation functions like the \( \tanh \) function.

The main problem here is that if the gradients in early layers of a network are very small and the ones in later layers are big (less multiplications), this leads to an unstable situation in which different layers in the network learn at different speeds. One approach to solve this problem is the Xavier Weights initialization which allows for the variance of the input and the output (and the layers in between) to be constant so that the network will learn optimally.

Exploding gradients: Similarly, if the weights in the network are very large and the derivatives of the activation functions are not too small (e.g. using RELUs), we get exploding gradients.