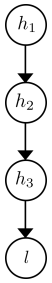
Exercises
Deep Learning
Fall 2020

Machine Learning Institute
Dept. of Computer Science, ETH Zürich
Professor Thomas Hofmann
Web http://www.da.inf.ethz.ch/teaching/2020/DeepLearning

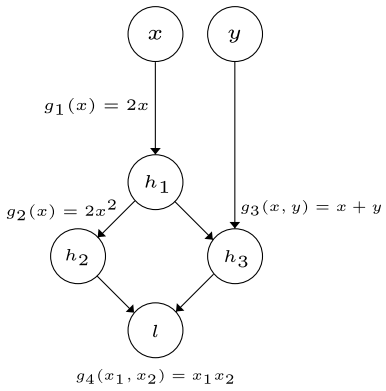
Series Monday, Oct 12, 2020
(Deep Learning, Exercise series 4)

Problem 1 (Backpropagation and Computational Graphs):

a) Most deep learning frameworks provide an automatic differentiation procedure to compute gradients based on the backpropagation algorithm introduced in the lecture. In these frameworks, all computations are represented as a graph and therefore also the gradient computation becomes a graph. Below you find a simple network. Use backpropagation to derive the gradient $\frac{\partial l}{\partial h_1}$ as a function of the intermediate (symbolic) gradients. Now add a node for each gradient that contributes to $\frac{\partial l}{\partial h_1}$ and connect them according to their dependencies.



b) Often you will see backpropagation applied to directed graphs that are trees. However, backpropagation can be applied to any directed acyclic graph (DAG). Below you see a simple DAG with two one-dimensional inputs $x, y \in \mathbb{R}$ and a final layer l. Derive $\frac{\partial l}{\partial x}$.



Problem 2 (Forward- vs. Backward differentiation):

Consider the simple feed-forward network $F_{nn} : \mathbb{R}^{2\times 3} \to \mathbb{R}$ depicted in Figure 1

a) Write $\boldsymbol{y}$ as a function of $\boldsymbol{w}_1$, $\boldsymbol{w}_2$, and $\boldsymbol{w}_3$.

b) Draw the corresponding computational graph

c) Compute $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{w}_1}$ in forward- and reverse (backward) mode differentiation

d) How do the cost of computing the entire gradient in each mode scale in the number of parameters and outputs of F_{nn}?

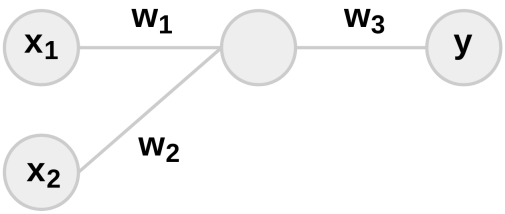


Figure 1: A simple feed-forward network

Problem 3 (Practical backpropagation for a feed-forward neural network):

We now study backpropagation in a more practical setting, and in a way that resembles the implementation of automatic differentiation in deep learning frameworks. When you define the architecture of a model, the framework constructs a *computational graph*: a directed acyclic graph where nodes correspond to tensors (inputs, outputs, learnable parameters, intermediate results). For each node (including intermediate ones), the framework stores a *forward buffer* and a *backward buffer*, which respectively store the result of the forward pass and the result of the backward pass. At the cost of a significant memory usage (depending on the complexity of the model), automatic differentiation can reuse partial computations and can be seen as a form of *dynamic programming*.

Consider the computational graph in Figure 2, which depicts a feed-forward neural network. This network consists of 3 learnable layers parameterized by the matrices $\mathbf{W}_1$, $\mathbf{W}_2$, and $\mathbf{W}_3$, without biases. We use ReLU activation functions $f(x) = \max(0, x)$ in all layers except the last one. We want to optimize the mean squared error loss (MSE) between our predicted output $\hat{\mathbf{y}} \in \mathbb{R}^d$ and the ground truth $\mathbf{y} \in \mathbb{R}^d$, which is defined as $L = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$ for a single data point.

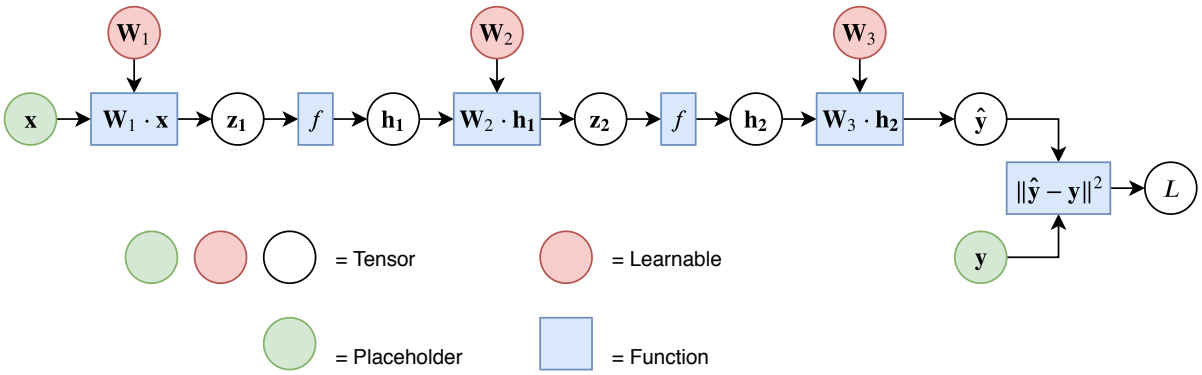


Figure 2: Computational graph for a feed forward network. Circles correspond to tensors, rectangles to functions.

Our final goal is to compute the gradient of the loss with respect to the weights, namely $\frac{\partial L}{\partial \mathbf{W}_1}$, $\frac{\partial L}{\partial \mathbf{W}_2}$, and $\frac{\partial L}{\partial \mathbf{W}_3}$.

1. Show simbolically (without actually computing any gradient) how we can compute $\frac{\partial L}{\partial \mathbf{W}_1}$, $\frac{\partial L}{\partial \mathbf{W}_2}$, and $\frac{\partial L}{\partial \mathbf{W}_3}$ by chaining the Jacobians of intermediate computations, i.e. as products of partial derivatives.
2. Compute the Jacobians of the following operations:
   a) Matrix multiplication, with respect to the input vector: $\frac{\partial}{\partial \mathbf{x}}(\mathbf{W}\mathbf{x})$
   b) General element-wise function f (e.g. activation function): $\frac{\partial}{\partial \mathbf{x}} f(\mathbf{x})$
   c) ReLU activation function: $\frac{\partial}{\partial \mathbf{x}} \max(\mathbf{0}, \mathbf{x})$
   d) Squared error, with respect to the predicted output: $\frac{\partial}{\partial \hat{\mathbf{y}}} \|\hat{\mathbf{y}} - \mathbf{y}\|^2$

We still need one building block to solve the task: the Jacobian of matrix multiplication with respect to the weight matrix $\frac{\partial}{\partial \mathbf{W}}(\mathbf{W}\mathbf{x})$. Since $\mathbf{W}$ is an $M \times N$ matrix (for N inputs and M outputs, yielding a $\mathbb{R}^N \to \mathbb{R}^M$

map), its Jacobian would result in a 3D tensor which is difficult to represent on paper and is very inefficient to compute – especially with regard to memory requirements. Also note that $\mathbf{x}$ is a constant here.

In practice, deep learning frameworks circumvent this problem entirely by observing that the loss is always a scalar, which means that the last operation (which corresponds to the leftmost Jacobian in the backward pass) is a map from a vector to a scalar ($\mathbb{R}^M \to \mathbb{R}$). Therefore, the leftmost Jacobian is a vector. More formally, a backpropagation chain from the loss to a parameter can be expressed as

$$\mathbf{j_1} \cdot \mathbf{J_2} \cdot \mathbf{J_3} \cdot \cdots \cdot \mathbf{J_n}$$

Using the associative property of matrix multiplication, we are free to choose the order of computations. Instead of computing Jacobian-Jacobian products, we start from the left and always compute vector-Jacobian products:

$$(\overbrace{(\underbrace{(\mathbf{j_1} \cdot \mathbf{J_2})}_{\text{vector}} \cdot \mathbf{J_3})}^{\text{vector}} \cdot \cdots \cdot \mathbf{J_n})$$

Therefore, for each operator, deep learning frameworks define how to perform this vector-Jacobian chaining instead of defining the full Jacobian. Formally, given a chain $f \to g \to \cdots \to L$, we define how to compute $\frac{\partial L}{\partial f}$ as a function of $\frac{\partial L}{\partial g}$, as opposed to defining $\frac{\partial g}{\partial f}$.

3. Derive $\frac{\partial L}{\partial \mathbf{W}_3}$ directly, without using backpropagation. Then, express $\frac{\partial L}{\partial \mathbf{W}_3}$ as a function of $\frac{\partial L}{\partial \hat{\mathbf{y}}}$. We now know how to backpropagate through a linear layer.
   Hint: while this operation is independent of the kind of loss function, for simplicity you can derive the gradient using the MSE and then abstract it away.

4. Using this knowledge, show how to compute $\frac{\partial L}{\partial \mathbf{W}_2}$ and $\frac{\partial L}{\partial \mathbf{W}_1}$ (you can assume that all intermediate states have already been computed).

5. Previously, you computed the full Jacobian for element-wise functions. Can we find a more efficient approach? Compute $\frac{\partial L}{\partial \mathbf{z_1}}$ given $\frac{\partial L}{\partial \mathbf{h_1}}$.

6. To show how automatic differentiation works on any DAG, assume that we want to optimize two losses, $L_1 = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$ (as before) and $L_2 = \|\hat{\mathbf{y}} - \mathbf{y}\|_1$ (mean absolute error), such that the final loss is $L = L_1 + L_2$. Write down the chain to compute $\frac{\partial L_1}{\partial \mathbf{W}_1}$ and $\frac{\partial L_2}{\partial \mathbf{W}_1}$ separately. Finally, show how you can efficiently calculate $\frac{\partial L}{\partial \mathbf{W}_1}$ by reusing intermediate computations.