Ausgewählte Kapitel sozialer Webtechnologien

Computational Graphs, Backpropagation and Automatic Differentiation

Oliver Fischer

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Agenda

- **Vorwissen**
  - Ableitungen
  - Ableitungsregeln
- **Computational Graph**
- **Backpropagation**
- **Automatic Differentiation**

Vorwissen

Suppose that we have a function $f : \mathbb{R} \rightarrow \mathbb{R}$, whose input and output are both scalars. The *derivative* of f is defined as

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}, \tag{2.4.1}$$

if this limit exists. If $f'(a)$ exists, f is said to be *differentiable* at a. If f is differentiable at every number of an interval, then this function is differentiable on this interval. We can interpret the derivative $f'(x)$ in (2.4.1) as the *instantaneous* rate of change of $f(x)$ with respect to x. The so-called instantaneous rate of change is based on the variation h in x, which approaches 0.

Let $y = f(x_1, x_2, \ldots, x_n)$ be a function with n variables. The *partial derivative* of y with respect to its i^{th} parameter x_i is

$$\frac{\partial y}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{2.4.7}$$

To calculate $\frac{\partial y}{\partial x_i}$, we can simply treat $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ as constants and calculate the derivative of y with respect to x_i. For notation of partial derivatives, the following are equivalent:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{2.4.8}$$

Vorwissen

To differentiate a function that is formed from a few simpler functions such as the above common functions, the following rules can be handy for us. Suppose that functions f and g are both differentiable and C is a constant, we have the *constant multiple rule*

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \tag{2.4.3}$$

the *sum rule*

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \tag{2.4.4}$$

the *product rule*

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{2.4.5}$$

and the *quotient rule*

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{2.4.6}$$

Vorwissen

Let us first consider functions of a single variable. Suppose that functions $y = f(u)$ and $u = g(x)$ are both differentiable, then the chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{2.4.10}$$

Now let us turn our attention to a more general scenario where functions have an arbitrary number of variables. Suppose that the differentiable function y has variables $u_1, u_2, \ldots, u_m$, where each differentiable function u_i has variables $x_1, x_2, \ldots, x_n$. Note that y is a function of $x_1, x_2, \ldots, x_n$. Then the chain rule gives

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_i} \tag{2.4.11}$$

for any $i = 1, 2, \ldots, n$.

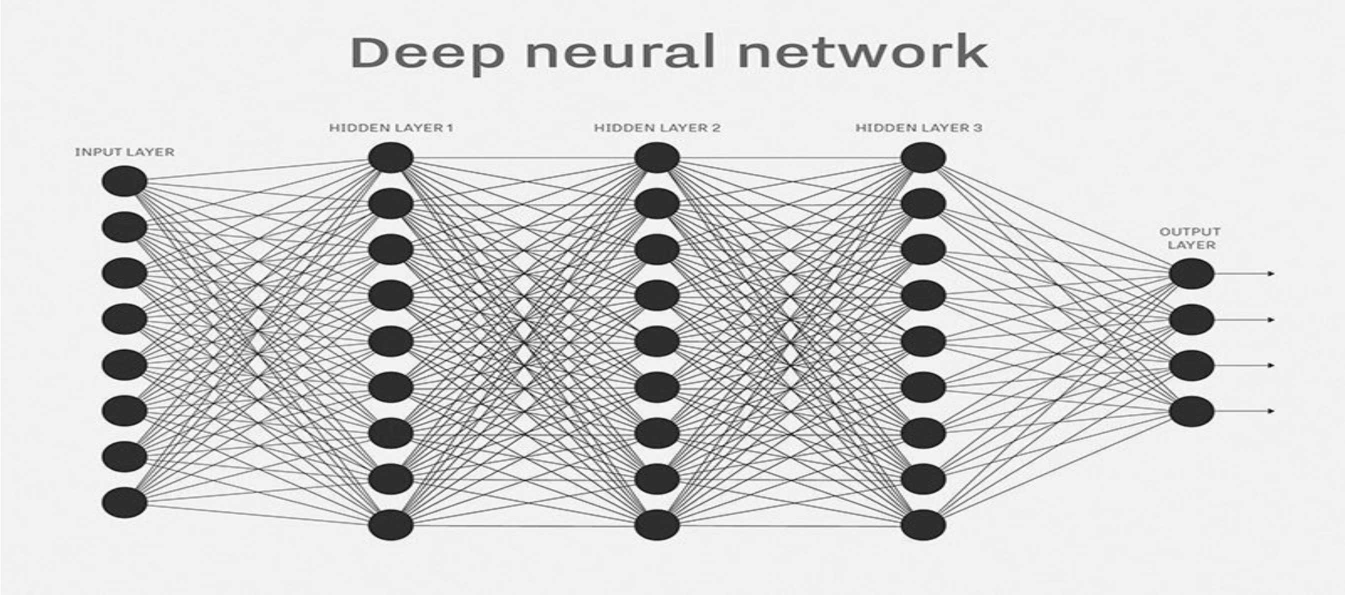
Computational Graphs (CG) - Hintergrund

Ziel: Verstehen der Grundbausteine und Funktion Neuronaler Netze

$Y = f(\theta, X)$

- θ Model Parameter
- X Input
- Y Output

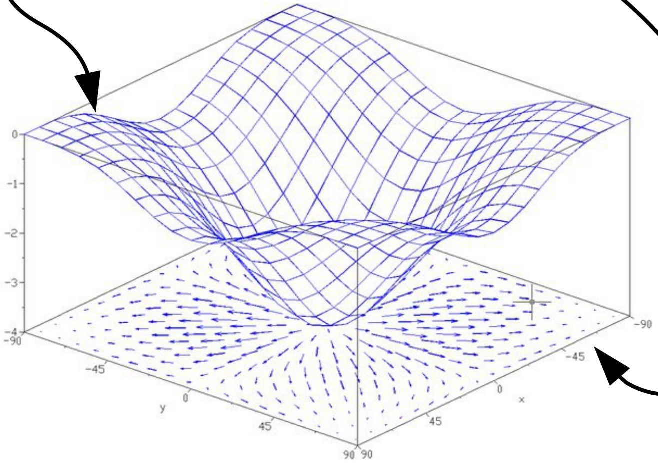
Lernen entspricht dem Anpassen aller θ's an eine Aufgabe!



Computational Graphs - Hintergrund

$f(\theta,X)$ beinhaltet das Model und die Fehlerfunktion für den Lernprozess

Input Vektoren θ,X → Skalarfeld $f(\theta,X)$ → Gradienten Feld $grad_{\theta} f(\theta,X)$



Computational Graphs - Hintergrund

Letzte Vorlesung:

GD for Multivariate Regression in Vector Form (1/4)

- Hypothesis: $h_\Theta(\vec{x}) = \vec{\Theta}^T \vec{x} = \Theta_0 x_0 + \Theta_1 x_1 + \ldots \Theta_n x_n$
  with $x_0 = 1$
- N + 1 Parameter: $\vec{\Theta}^T = (\Theta_0, \Theta_1, \ldots, \Theta_n)$

- **Minimize cost function J:**

$$J(\vec{\Theta}) = \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(\vec{x}^{(i)}) - y^{(i)})^2$$

Computational Graphs - Hintergrund

Letzte Vorlesung:

GD for Multivariate Regression in Vector Form (2/4)

- Repeat until convergence is reached

$$\Theta_j \leftarrow \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta)$$

- Simultaneous update of all Θ_j

Computational Graphs - Hintergrund

Letzte Vorlesung:

GD for Multivariate Regression in Vector Form (3/4)

- Gradient Definition:

$$grad(J(\Theta)) = \nabla J(\Theta) = \begin{pmatrix} \frac{\partial J(\Theta)}{\partial \Theta_0} \\ \frac{\partial J(\Theta)}{\partial \Theta_1} \\ \dots \\ \frac{\partial J(\Theta)}{\partial \Theta_n} \end{pmatrix}$$

$$\vec{\Theta}^{neu} \leftarrow \vec{\Theta}^{alt} - \alpha \cdot grad(J(\Theta^{alt}))$$

Computational Graphs - Hintergrund

Letzte Vorlesung:

GD for Multivariate Regression in Vector Form (4/4)

$$\frac{\partial}{\partial \Theta_j} J(\Theta) = \frac{\partial}{\partial \Theta_j} \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(\vec{x}^{(i)}) - y^{(i)})^2$$

$$= \frac{\partial}{\partial \Theta_j} \frac{1}{2m} \sum_{i=1}^{m} (\vec{\Theta}^T \cdot \vec{x}^{(i)} - y^{(i)})^2$$

Results in Update Rule:

$$\Theta_j \leftarrow \Theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (\vec{\Theta}^T \cdot \vec{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

or

$$\vec{\Theta}^{neu} \leftarrow \vec{\Theta}^{alt} - \alpha \frac{1}{m} \sum_{i=1}^{m} (\vec{\Theta}^T \cdot \vec{x}^{(i)} - y^{(i)}) \vec{x}^{(i)}$$

Computational Graphs - Hintergrund

- Manuelles Ableiten für sehr komplexe Funktionen, siehe NN und CNN, nicht möglich

  Brauchen eine Automatisierung zum Ableitung
  ☐ DeepLearning Framework, siehe PyTorch

Computational Graphs - Hintergrund

Typische Trainingsschleife in PyTorch

PyTorch Cifar10 Training

.backward() beinhaltet automatisiertes Ableiten und ist das Herzstück moderner DeepLearning Frameworks

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

print('Finished Training')
```

Model+Loss Function bilden CG

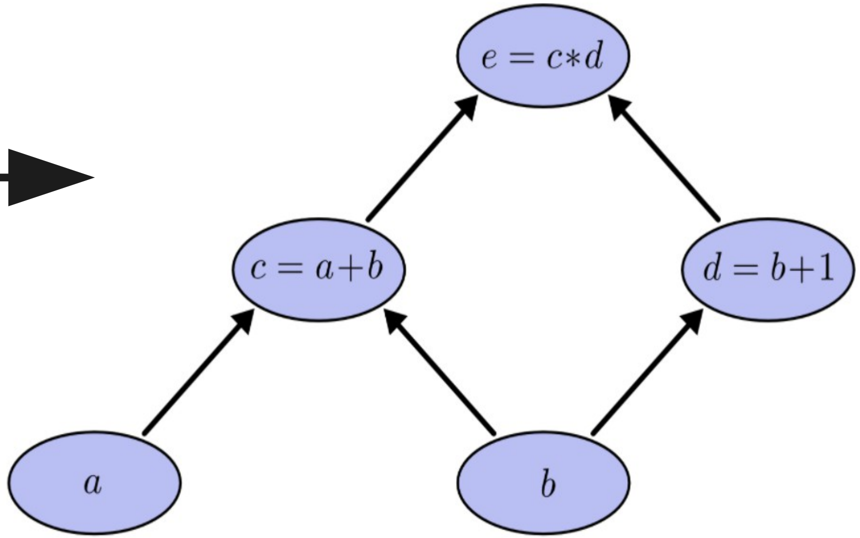
Model Training

Computational Graphs

- grafische Darstellung einer Funktion

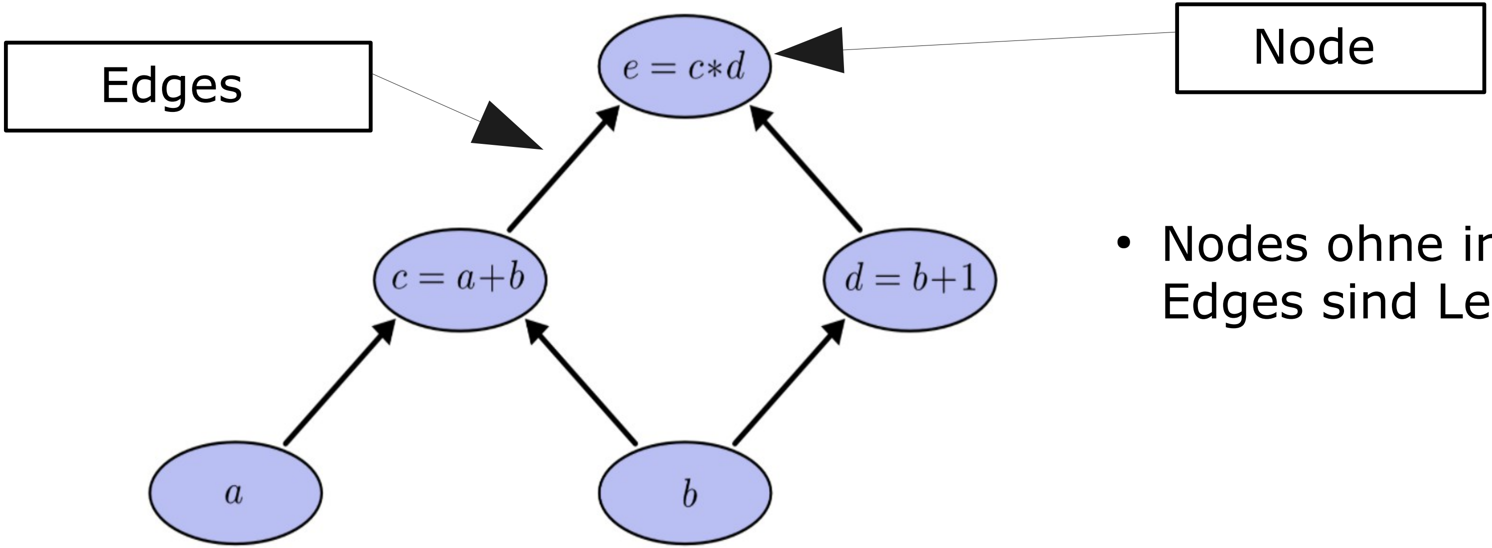
e(a,b) = (a+b)*(b+1)

c = a+b
d = b +1
e = c * d



Computational Graphs

- besteht aus Nodes und directed Edges



- Nodes ohne incomming Edges sind Leaf-Nodes

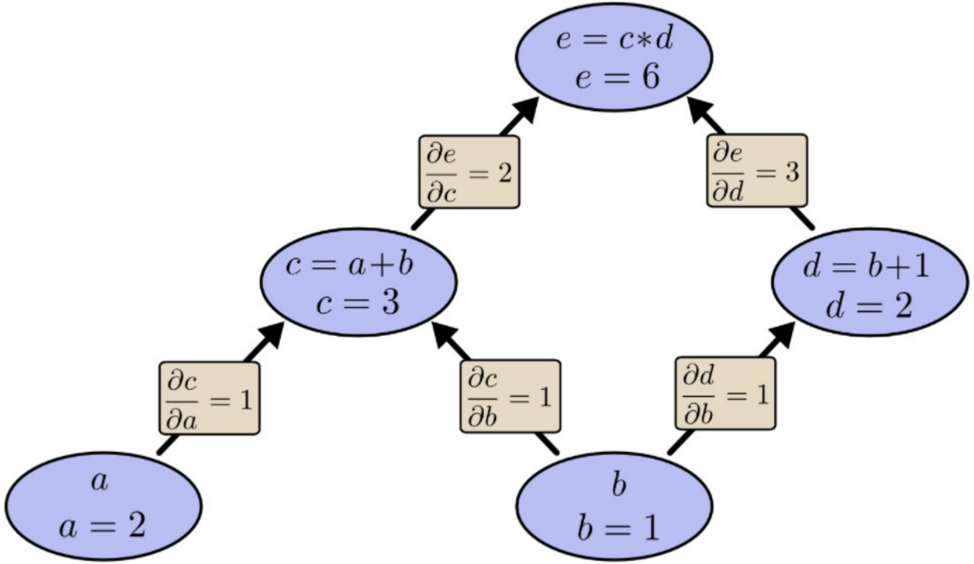
Computational Graphs

Vorhin: Lernen entspricht dem Anpassen aller θ's (Parameter) an eine Aufgabe!

→ Zum Anpassen der Parameter einer Funktionen werden deren Ableitungen benutzt.

Computational Graphs

Derivatives on Computational Graphs

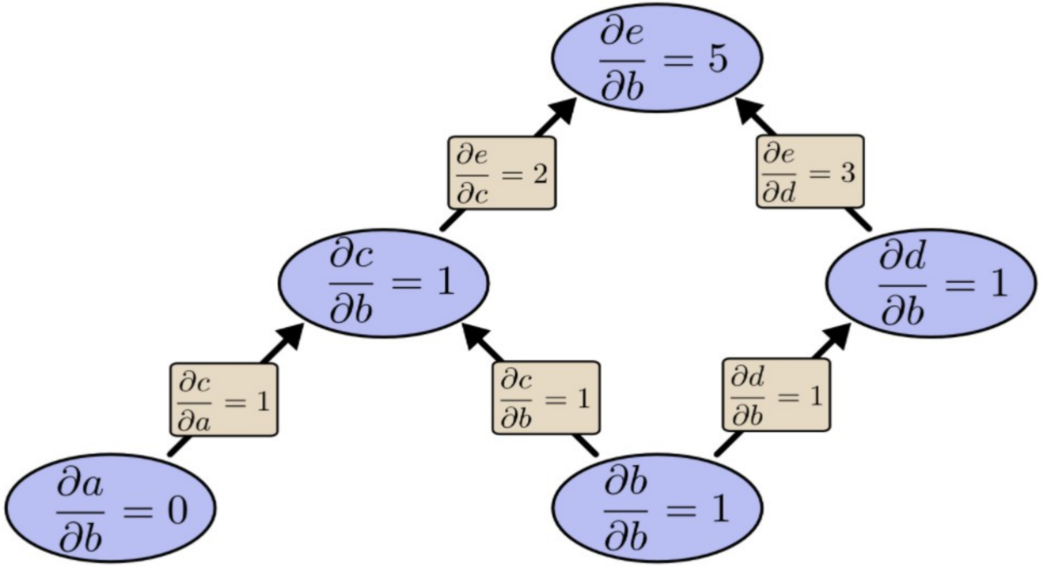


Computational Graphs

Forward-mode differentiation

Ableitung aller Nodes bezüglich **eines** Leaf-Nodes!

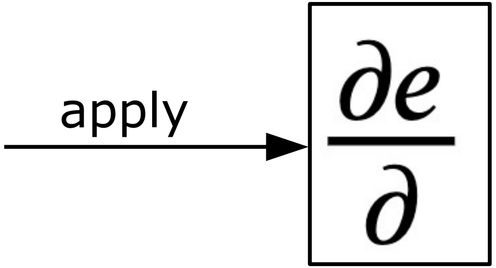
$$\text{apply} \longrightarrow \boxed{\frac{\partial}{\partial b}}$$

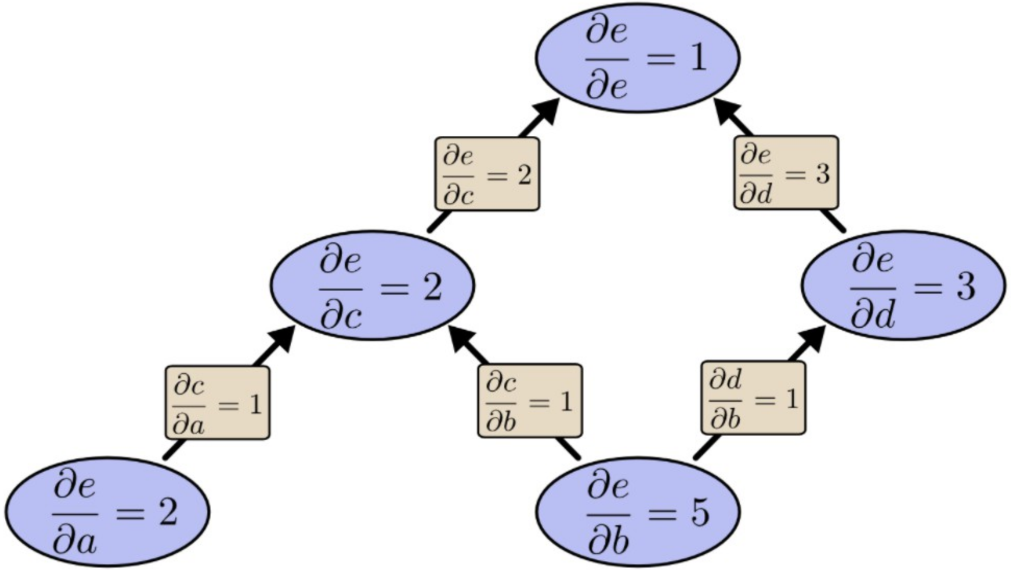


Computational Graphs

Reverse-mode differentiation

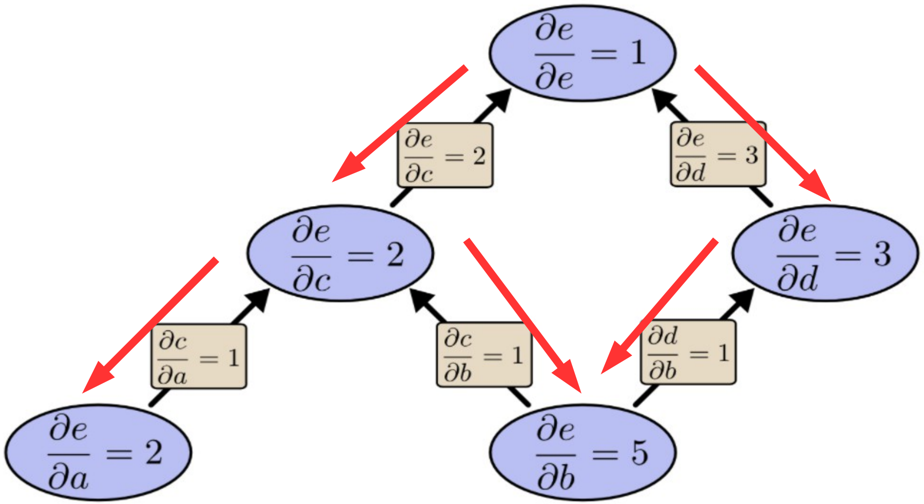
Ableitung des Outputs bezüglich **aller** Nodes!





Backpropagation

Im Kontext Neuronaler Netze entspricht Backpropagation der Reverse-mode differentiation.



Automatic Differentiation (automatisiertes Ableiten)

- für komplexere Funktionen/ Berechnungen ist die manuelle Berechnung der Node Ableitungen und Kettenregeln nicht mehr möglich

- z.B.: typische moderne Neuronale Netze besitzen 10m++ lernbare Parameter

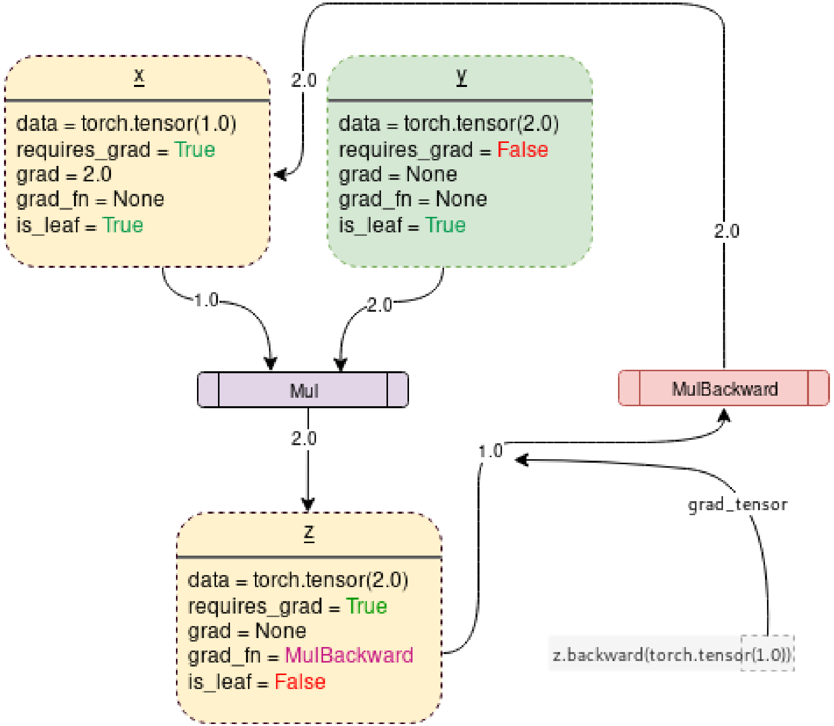
- z.B.: Bild mit Auflösung von 224x224x3 (ImageNet) entspricht 150528 Leaf-Nodes

- brauchen Deep Learning Frameworks welche automatisiert für jede Berechnung einen Graphen erstellen und darauf Ableitungen berechnen können

Automatic Differentiation

$z = x * y$
$2.0 = 1.0 * 2.0$

- für diese Vorlesung: PyTorch
  - Konstruiert für jede Berechnung einen Computational Graph
  - Tracked Node status
  - Kennt für alle in PyTorch implementierten Rechenoperation den Forward sowie Backward path
  - Erlaubt Backpropagation via automatic differentiation



Automatic Differentiation

Für Interessierte: Deep.TEACHING

- Einstieg in Differentiable Programming
- Übungsbegleiteter Kurs zum Erstellen eines eigenen Python Deep Learning Framework
- Von automatic differentiation auf Skalaren bis zum Bau Neuronaler Netze

Weitere Fragen?