Bauhaus-Universität Weimar

1 Computational Mechanics Meets Artificial Intelligence: PyTorch

Leon Herrmann

Stefan Kollmannsberger

Chair of Data Engineering in Construction

Bauhaus-Universität Weimar

Deep Learning in Computational Mechanics – an introductory course, Herrmann et al. 2025

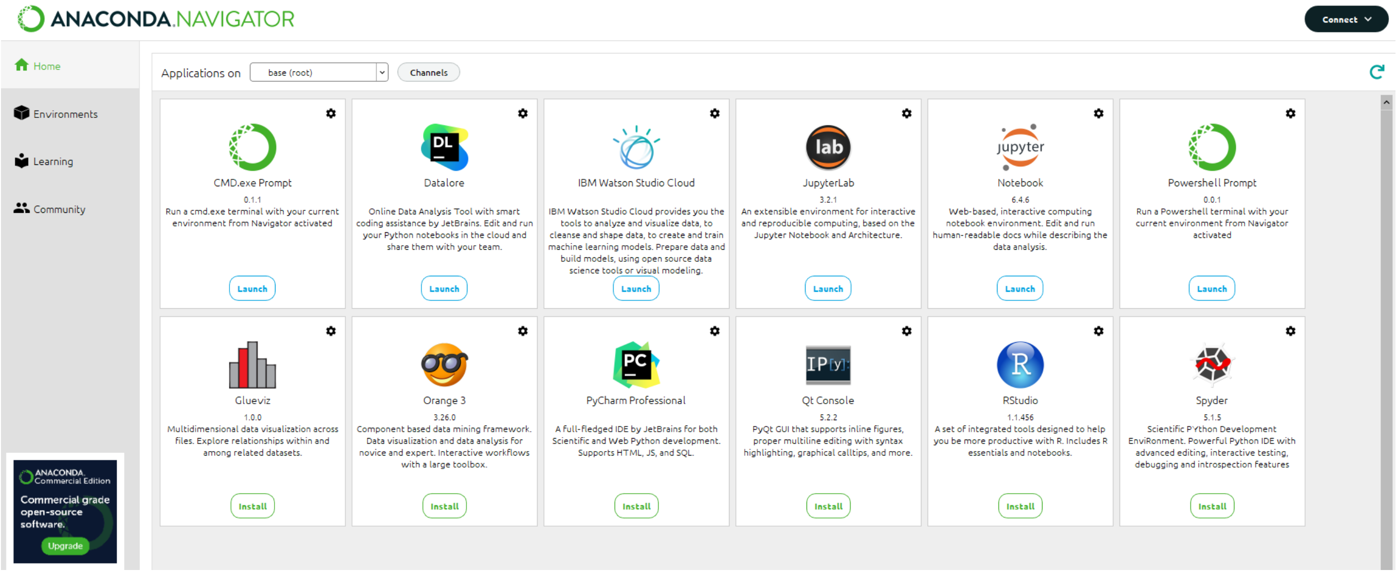
website

book

Contents

Installation

- Download Anaconda from https://www.anaconda.com/products/individual
- Install Anaconda
- Open the Anaconda Navigator
- In the ”Home” tab
  - Install Spyder
  - Install JupyterLab



Installation of modules via command prompt

- Open the Anaconda prompt
- Install PyTorch (tensors and neural networks)

  `conda install pytorch cpuonly -c pytorch`

  (or `conda install pytorch -c pytorch` if you have a GPU)
- Install SciPy (scientific computing)

  `conda install -c anaconda scipy`
- Install Matplotlib (visualization)

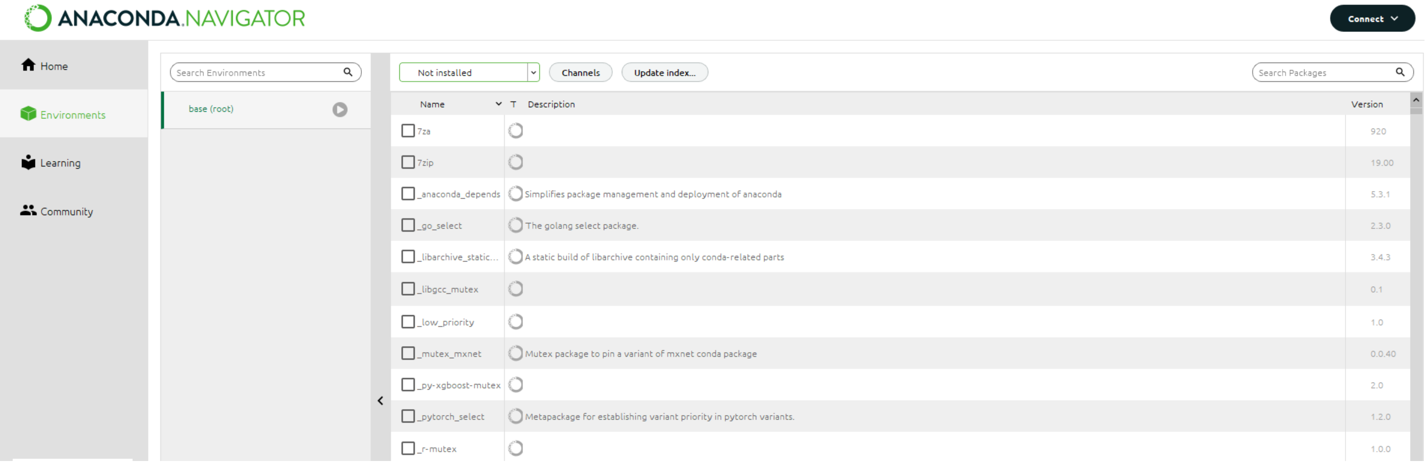
  `conda install -c conda-forge matplotlib`
- Modules that will be used in the course and are preinstalled with the default anaconda installation
  - NumPy (arrays and matrices)
  - Pillow (imaging)
- (If not)

  `conda install -c anaconda numpy`

  `conda install -c anaconda pillow`

Alternatively – Installation of Modules via GUI

- In the ”Environment” tab
  - Navigate to the base (root) and toggle to the ”Not installed” modules



  - Select and install the modules (with the Apply button)
    - PyTorch
    - SciPy

Introduction to Python

- This is not a course on Python.
- For an introduction to the basics of Python, see https://docs.python.org/3/tutorial/index.html

Arrays in Python with NumPy

- NumPy is a library for multi-dimensional arrays and matrices in Python
- Good introduction available at https://numpy.org/doc/stable/user/absolute_beginners.html

- Import of the library

```
import numpy as np
```

- Array creation

```
a = np.zeros((5,5))     # array filled with zeros (5, 5)
b = np.ones(5)      # array filled with ones (5)
c = np.array([1,2,3,4,5])   # array created from a list (5)
d = np.linspace(0,1,10) # linearly spaced array in range [0,1] (10)
```

- Array transformations

```
np.concatenate((b,c), axis=0)   # concatenation of arrays b and c
np.reshape(a, 25)       # reshape into an array of dimension (25)
a.flatten()     # flattens an array the dimension (25)
```

Arrays in Python with NumPy

- Indexing
- In Python the index starts at 0 and thus ends at n-1

```
c = np.array([1,2,3,4,5])
c[-1]        # last element
c[:2]        # the first two elements (excluding i=2)
c[2:]        # elements from the three onward (including i=2)
c[1:3]       # second element until third element (i=1,2)
c[::2]       # every second element starting from i=0
c[::-1]      # flips the order of the elements
```

- Linear Algebra

```
np.dot(b,c) # vector product
np.matmul(a,b)  # matrix multiplication
```

- Elementwise math operations (most math expressions from the math library are available)

```
np.sin(d*np.pi) # elementwise application of the sine
b*c          # elementwise multiplication of vectors b and c
```

Classes in Python

- A class is an extensible program-code-template for creating objects
- Good introduction to classes in Python at https://docs.python.org/3/tutorial/classes.html
- Basic class outline in Python

```python
class MyClass:
    def __init__(self, y):  # constructor
        self.a = y  # member variable a

    def f(self, b):     # member function
        return self.a + b
```

- Instantiation of MyClass

```python
x = MyClass(3)          # construction
x.a                # acess of member variable a
x.f(2)                 # call on member function f
```

Lambda Functions in Python

- Lambda functions are a convenient way of expressing functions.
- Basic lambda syntax in Python

```
f = lambda x : x**2 + 3
f(2)              # calling the lambda function
```

- This is equivalent to defining the function as

```
def f(x):
    return x**2 + 3
```

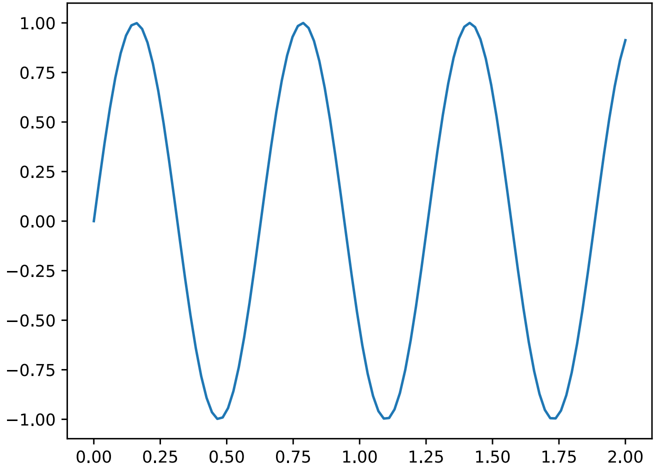
- Lambda syntax for multiple inputs

```
g = lambda x, y : x * y
g(2, 3)           # calling the lambda function
```

Plotting in Python with Matplotlib

- Matplotlib is a plotting library for Python
- Good introduction available at https://matplotlib.org/stable/tutorials/introductory/usage.html
- Basic 2d plot

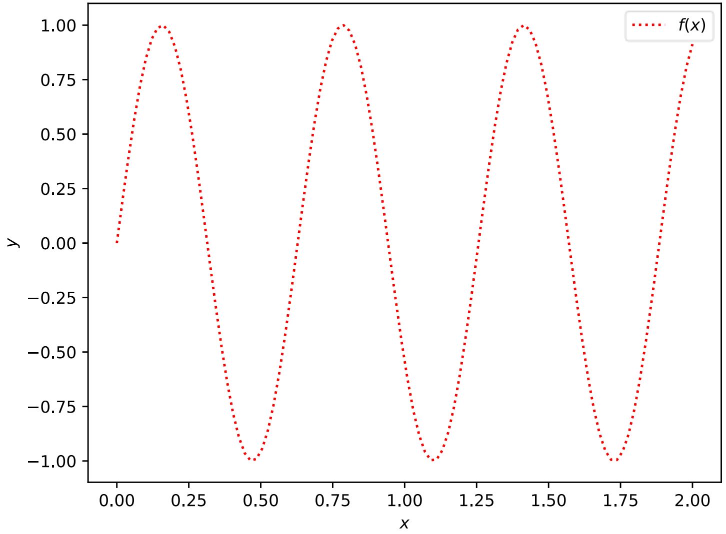
```
import matplotlib.pyplot as plt
import numpy as np

f = lambda x : np.sin(x)
x = np.linspace(0, 2, 100)

plt.plot(x, f(x))
```



Plotting in Python with Matplotlib

- For more control, an object-oriented interface is typically used

```
f = lambda x : np.sin(x)
x = np.linspace(0, 2, 100)

fig, ax = plt.subplots()
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.plot(x, f(x), linestyle=':',
    color='r', label="$f(x)$")
ax.legend()
fig.tight_layout()
plt.show()
```



- For more options for lineplots, see https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html

Plotting in Python with Matplotlib

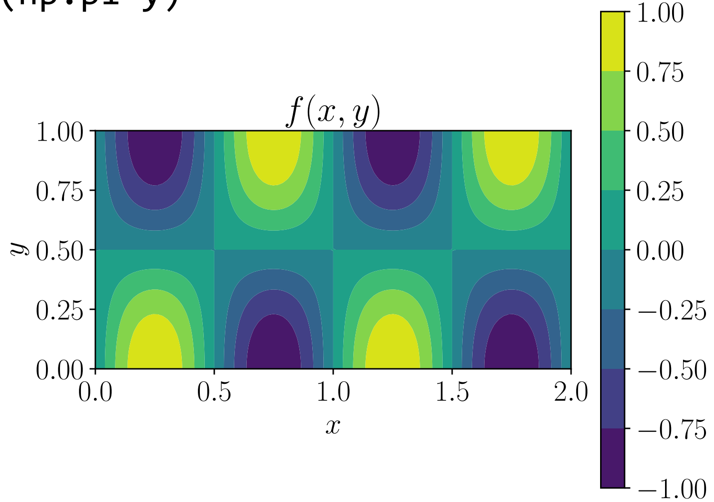
- Contourplots

meshgrid

$$x_i = [3, 4, 5] \qquad y_i = [1, 2]$$

$$x_o = \begin{pmatrix} 3 & 4 & 5 \\ 3 & 4 & 5 \end{pmatrix} \qquad y_o = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix}$$

```python
x = np.linspace(0, 2, 100)
y = np.linspace(0, 1, 100)
x, y = np.meshgrid(x, y)
g = lambda x, y : np.sin(2*np.pi*x)*np.cos(np.pi*y)

fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_title("$f(x, y)$")
cp = ax.contourf(x, y, g(x, y))
fig.colorbar(cp)
fig.tight_layout()
plt.show()
```



Plotting in Python with Matplotlib

- Pseudocolorplot (no interpolation)

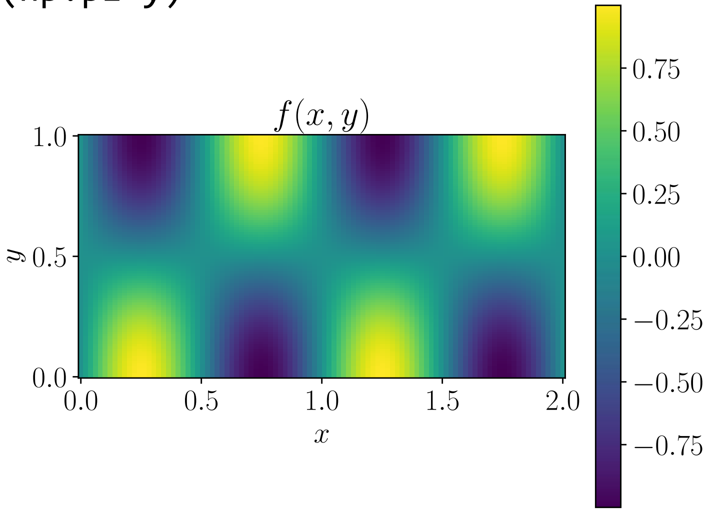
```
x = np.linspace(0, 2, 100)
y = np.linspace(0, 1, 100)
x, y = np.meshgrid(x, y)
g = lambda x, y : np.sin(2*np.pi*x)*np.cos(np.pi*y)

fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_title("$f(x, y)$")
cp = ax.pcolormesh(x, y, g(x, y))
fig.colorbar(cp)
fig.tight_layout()
plt.show()
```

meshgrid

$y_i = [1, 2]$

$x_i = [3, 4, 5]$

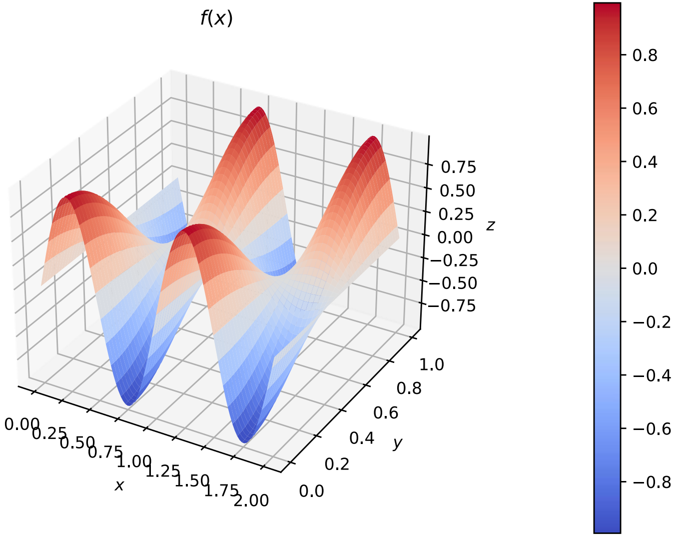
$$x_o = \begin{pmatrix} 3 & 4 & 5 \\ 3 & 4 & 5 \end{pmatrix} \qquad y_o = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix}$$



Plotting in Python with Matplotlib

- Surfaceplots

```
fig = plt.figure()

ax = fig.add_subplot(projection='3d')
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_zlabel("$z$")

ax.set_title("$f(x)$")
cp = ax.plot_surface(x, y, g(x, y), cmap=plt.cm.coolwarm,
    antialiased=False)
fig.colorbar(cp, pad=0.2, ticks=np.linspace(-0.8, 0.8, 9))
fig.tight_layout()
plt.show()
```



Scientific Computing in Python with SciPy

- SciPy is a Python library for scientific computing
- It contains modules for
  - Optimization
  - Linear algebra
  - Integration
  - Interpolation
  - FFT
  - ODE solvers
  - Image processing
  - And many more
- It will be used in the last part of the course, when talking about data-driven approaches

- For an introduction to the library, check out https://scipy.github.io/devdocs/tutorial/index.html

What is PyTorch

- Open source machine learning library
- Primarily developed by Facebooks's AI Research lab
- Used in softwares, such as Tesla's autopilot, Disney's face recognition, OpenAI's projects

- Two high-level features
  - Tensor computing with acceleration via GPU's (similar to numpy)
  - Neural networks with automatic differentiation

- Introductory tutorial: https://pytorch.org/tutorials/beginner/nlp/pytorch_tutorial.html
- Find help here: https://pytorch.org/docs/
- Alternative libraries: Tensorflow, Keras

- In Python:

```
import torch
```

PyTorch

Computing with Tensors – Creation

- Creation of arbitrary tensors

```
a = torch.tensor([[0., 2., 4.], [3., 4., 2.]]) # shape (2, 3)
b = torch.zeros(3, 3, 3) # shape (3, 3, 3)
c = torch.linspace(0, 1, 10) # shape (10)
```

- Indexing

```
print(a[0, 0]) # to access first element
a[:, 0] # to access first order tensor in the second dimension
b[0, :, :] # to access first second order tensor
b[-1, -1, -1] # to access last element
a[1:3, 0] # to access last two elements of the first first order tensor
```

- Functions operating on tensors take one or multiple tensors as input and return the modified tensor
- Useful operations

```
print(a)     #print the tensor to console
a.shape or a.size()
torch.cat((a, b[0, :, :]),0) # concatenation in dimension 0
```

Computing with Tensors – Manipulation

```
a = torch.tensor([[0., 2., 4.], [3., 4., 2.]]) # (2, 3)
b = torch.zeros(3, 3) # (3, 3)
c = torch.tensor([[[0., 1.], [4., 2.]], [[2., 3.], [3., 2.]]]) #(2,2,2)
d = torch.ones(1,4) # (1, 4)
```

- Useful operations in general

```
torch.cat((a, b),0) # concatenation in dimension 0
torch.transpose(a, 0, 1) # dimensions 0 and 1 are transposed
torch.permute(c, (2, 1, 0)) # dimensions 0 and 2 are permuted
torch.reshape(c, (1, 8)) # or c.view(1, 8)  reshape to shape (1, 8)
torch.unsqueeze(a, 0) # creates an additional empty dimension (1, 2, 3)
torch.squeeze(d, 0) # removes first empty dimension (4)
torch.flatten(c) # flattens c (8)
torch.ones_like(c) # creates a tensor of ones with the shape of c
torch.cos(a) # cosine, mathematical operations as in numpy: apply cos()
             to each entry
```

Computing with Tensors – Manipulation

- Often the same Pytorch functions can be called in two ways

```
torch.flatten(c) or c.flatten()
```

- Creating a grid

```
x = torch.linspace(0, 1, 10) # (10)
y = torch.linspace(0, 2, 5) # (5)
x, y = torch.meshgrid(x, y) # creates a grid (10,5)
```

- Conversion to and from numpy

```
import numpy as np
a = np.array([2., 3.])
b = torch.from_numpy(a) # convert numpy array to tensor
b.numpy() # convert tensor to numpy array
```

Computing with Tensors – Gradient

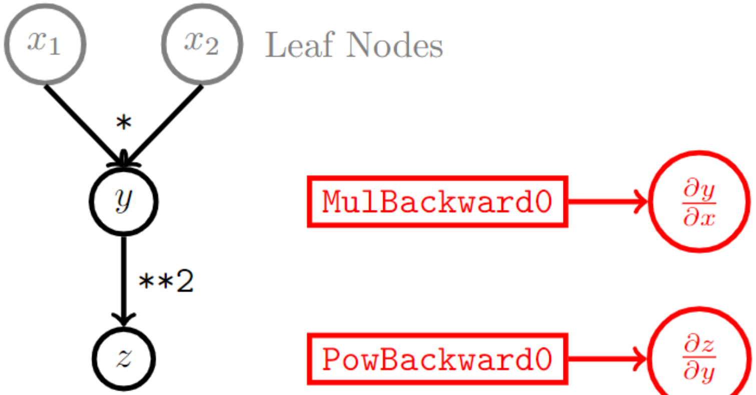
- The computation of gradients occurs via automatic differentiation by constructing a computational graph

The construction of the computational graph is enabled with `requires_grad = True`

```
x.requires_grad = True
y = torch.tensor([0.], requires_grad = True)
```

- The computational graph saves all operations as function handles, e.g. MulBackward0 for the first multiplication that occurs in the graph
- Each node, i.e. tensor of the graph contains

```
data # tensor
requires_grad # boolean for gradient
grad # gradient
grad_fn # function handle
is_leaf # boolean for leaf nodes
```



Computing with Tensors – Gradient

- The construction of the graph occurs during tensor manipulations, i.e. during the forward propagation
- The gradients are computed with the chain rule during the backward propagation
- Consider the following example

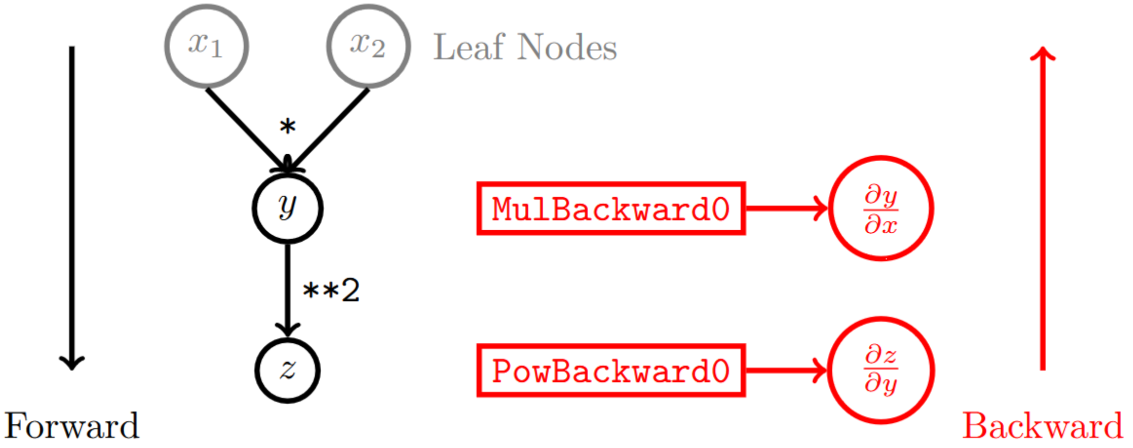
$$x_1, x_2$$
$$y = x_1 \cdot x_2$$
$$z = y^2$$

- Partial derivatives saved in the graph

$$\frac{\partial z}{\partial y} = 2y, \frac{\partial y}{\partial x_1} = x_2, \frac{\partial y}{\partial x_2} = x_1$$

- The backward propagation is called for z
- The derivative of z w.r.t. each variable is computed and saved at the nodes
- Example for what is saved at x_1.grad

$$\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x_1} = 2y \cdot x_2 = 2x_1 x_2^2$$



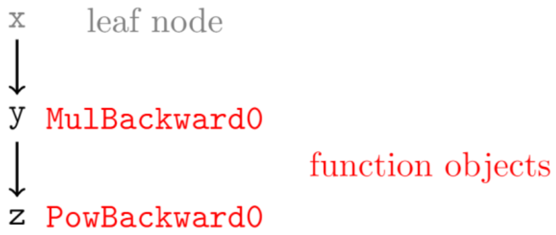
Computing with Tensors – Gradient

- The gradient of each tensor can be computed with automatic differentiation
  - By building up a computational graph with each tensor manipulation
  - Using the backward propagation (an algorithm to compute the gradient, more on this in Chapter 3)

- Manipulation

```
x = torch.tensor([1.], requires_grad=True)
y = 2 * x
z = y ** 2
```

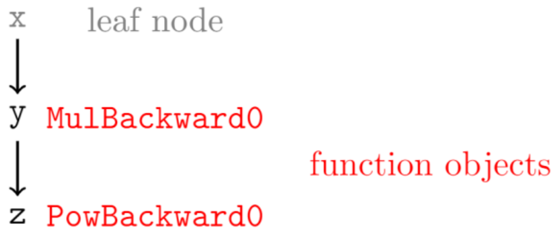
Corresponding Graph



Print out `x, y, z` to check the function objects or use `x.grad_fn`

- A tutorial for more details: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

Computing with Tensors – Gradient



- Sampling

```
x = torch.linspace(0., 1., 10)
x.requires_grad = True # initiates the graph
```

- Manipulation (to see the function objects, print out `y` and `z`)

```
y = 2 * x
z = y ** 2
```

- The backward propagation only saves the gradients at the leaf node. To save the gradient at non-leaf nodes, this must be specified

```
y.retain_grad()
```

- Backward propagation computes the partial derivatives of `z` with respect to all operations and saves them at the leaf nodes `x` unless specified otherwise as e.g. `y`

```
z.backward(torch.ones_like(z), retain_graph=True)
```

- Access the gradients

```
dz_dx = x.grad # gradient of z with respect to x (here 8x)
dz_dy = y.grad # gradient of z with respect to y (here 2y so 4x)
```

Computing with Tensors – Gradient

Alternative more convenient gradient computation with `autograd` (complete code below)

```
import torch
from torch.autograd import grad
x = torch.linspace(0., 1., 10)
x.requires_grad = True # initiates the graph
y = 2 * x
z = y ** 2
dz_dx = grad(z, x, torch.ones_like(z), retain_graph=True)[0]
dz_dy = grad(z, y, torch.ones_like(z), retain_graph=True)[0]
```

- Detachement from graph (important for performance, conversion to numpy, and plotting)

```
z.detach()
```

- Plotting with matplotlib

```
import matplotlib.pyplot as plt
plt.plot(x.detach(), dz_dx.detach())
```

Computing with Tensors – Gradient

`autograd.grad` returns a tuple, where each entry corresponds to the quantities to be derived with

```
dz_dx = grad(z, x, torch.ones_like(z), retain_graph=True)[0]
dz_dy = grad(z, y, torch.ones_like(z), retain_graph=True)[0]
```

- Can be rewritten as

```
dz = grad(z, (x,y), torch.ones_like(z), retain_graph=True)
dz_dx = dz[0]
dz_dy = dz[1]
```

In general, `autograd.grad` is an engine for computing the vector-Jacobian product $\boldsymbol{J}^T\boldsymbol{v}$, which is useful in the context of backpropagation, i.e., applying the chain-rule

$$\boldsymbol{J}^T\boldsymbol{v} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial z}{\partial y_1} \\ \vdots \\ \frac{\partial z}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial z}{\partial x_1} \\ \vdots \\ \frac{\partial z}{\partial x_n} \end{pmatrix}$$

Computing with Tensors – Gradient

- Consider the following example $\boldsymbol{Ax} = \mathbf{y}$ where

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \boldsymbol{x} = (2, 1)^T, \boldsymbol{y} = (y_1, y_2)^T$$

- The Jacobian of y w.r.t. x can be computed with

```
A = torch.tensor([[1., 2.], [3., 4.]])
x = torch.tensor([[2.], [1.]], requires_grad=True)
y = torch.matmul(A, x) # alternative is A @ x
dy_dx = torch.zeros((2,2))
dy_dx[0] = grad(y, x, torch.tensor([[1.], [0.]]), retain_graph=True)[0][:,0]
dy_dx[1] = grad(y, x, torch.tensor([[0.], [1.]]), retain_graph=True)[0][:,0]
```

Explanation for the first row dy_dx[0]

$$\left(\frac{\partial y}{\partial x}\right)_1 = \begin{pmatrix} \dfrac{\partial y_1}{\partial x_1} & \dfrac{\partial y_2}{\partial x_1} \\ \dfrac{\partial y_1}{\partial x_2} & \dfrac{\partial y_2}{\partial x_2} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \dfrac{\partial y_1}{\partial x_1} \\ \dfrac{\partial y_1}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Computing with Tensors – Gradient

- Alternative with the inbuilt Jacobian function

```
y = lambda x : torch.matmul(A, x)
dy_dx = torch.autograd.functional.jacobian(y, x)
```

- For element-wise operations

```
x = torch.tensor([1., 2.], requires_grad=True)
y = x**2  #element wise sqaring! y=[x_1^2, x_2^2]=[1,4]
dy_dx1 = grad(y, x, torch.tensor([1., 0.]),retain_graph=True)
dy_dx2 = grad(y, x, torch.tensor([0., 1.]),retain_graph=True)
dy_dx  = grad(y, x, torch.tensor([1., 1.]),retain_graph=True)
```

- Explanation: The fact, that only diagonal entries are non-zero can be exploited

$$\frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} \\ \frac{\partial y_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

Exercises

- E.1 Introduction to PyTorch (C)
  - Familiarize yourself with the PyTorch syntax, including tensor initialization & manipulation, gradient computation, neural networks, GPU acceleration.
- E.2 Computing Stress Fields with Tensors (C)
  - Given a displacement field, compute the corresponding stress field using automatic differentiation.
- E.3 Stress Fields from Perceptrons (C)
  - Given a displacement field expressed via a perceptron, compute the corresponding stress field using automatic differentiation.

Contents

1 Computational Mechanics Meets Artificial Intelligence: PyTorch

Leon Herrmann

Stefan Kollmannsberger

Chair of Data Engineering in Construction

Bauhaus-Universität Weimar

Deep Learning in Computational Mechanics – an introductory course, Herrmann et al. 2025

website

book