Tutorial: **Hands-On Deep Learning using pytorch** (BVM 2019)

Mattias Heinrich, Christian Lucas, Max Blendowski

Overview / learning outcomes

1) understand the principles of deep learning and its implementation with pytorch
2) build and train your own network and use autograd for iterative image registration (unsupervised)

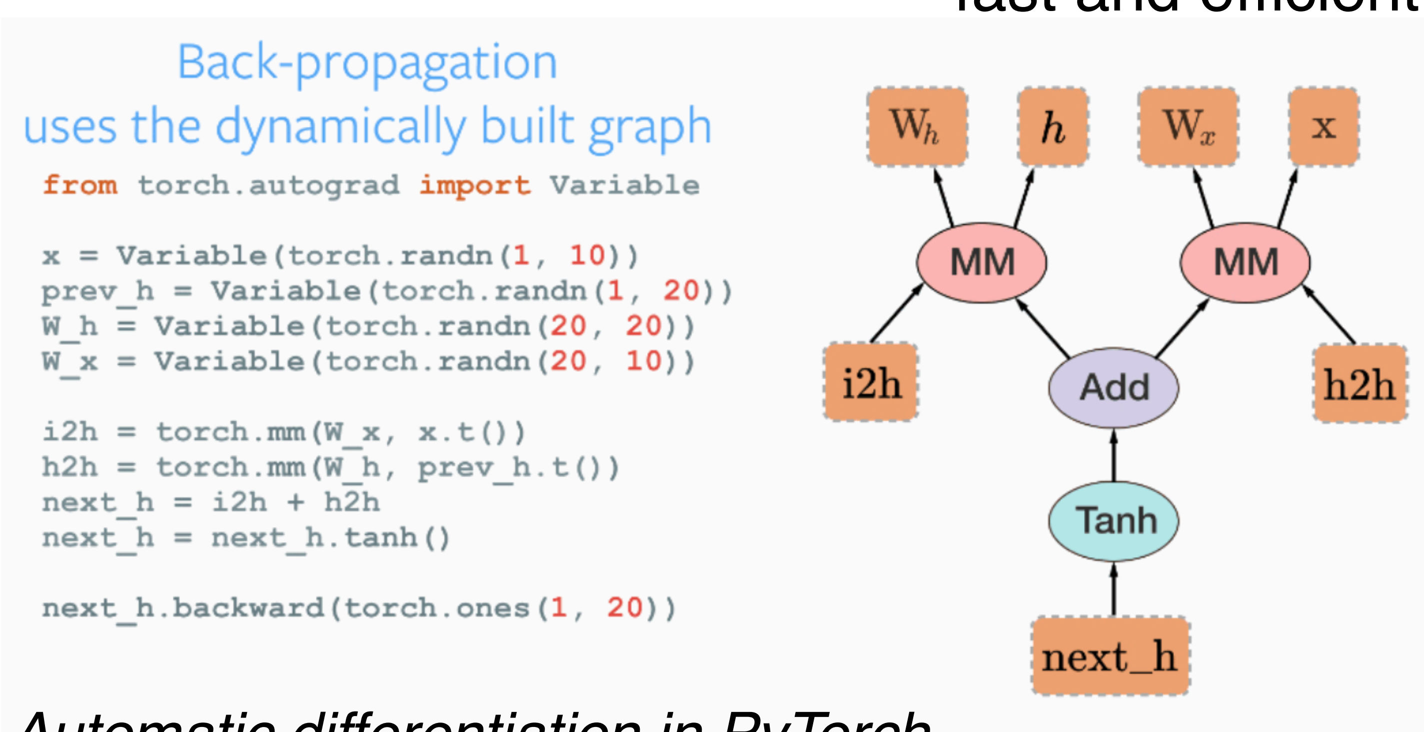
Time	
14:00 - 14:40	Introduction to pytorch and image
14:40 - 15:45	**Practical Part I: build and train U-Net**
15:45 - 16:00	Coffee break
16:00 - 16:20	Intro to autograd and image registration
16:20 - 17:00	**Practical Part II: metrics + iterative alignment**
from 17:00	mingle with free BVM beer/wine

Why pytorch is a good choice for deep learning:

"An open source deep learning platform that provides a seamless path from research prototyping to production deployment."

originated from torch (lua-based) which offered:

- a powerful N-dimensional array
- lots of routines for indexing, slicing, transposing, …
- linear algebra routines
- neural network, and energy-based models
- numeric optimisation routines
- fast and efficient GPU support



→ great flexibility (dynamic graphs)
→ very fast (C++/CUDA backends)
→ immediate debugging python
→ seamless installation (conda, ..)

Automatic differentiation in PyTorch
https://openreview.net/pdf?id=BJJsrmfCZ

Medical Deep Learning using PyTorch

Three main layers of the framework

`tensor`
data container / variable
B x C x H x W x D
tracked by autograd
supports nearly all mathematical numpy functions (sum, exp, mm, etc.) and low-level operations: view, squeeze, broadcasting, slicing, typecasting

`nn`
neural network modules
with trainable parameters
interacts with optimiser
supports vast majority of (2D/3D) deep network operations: conv, batch-norm, relu, interpolate, pooling, etc.

`nn.functional`
similar functionality but without optimising parameters, + *grid_sample*

`optim`
stochastic gradient descent
optimisers with momentum, e.g. *Adam.* Uses `autograd` to calculate gradient update steps w.r.t. to loss:

```
net.train()
optimizer.zero_grad()
output = net(input)
loss = criterion(output,
label)
loss.backward()
optimizer.step()
```

`torchvision.datasets & .models`
pre-processed and annotated computer vision datasets and pre-trained 2D models
`torch.utils.data.Dataset & .checkpoint`
abstract class for defining a dataset with batch-loader, checkpointing for saving memory

PyTorch A simplistic introduction

```
A = torch.Tensor([2,4])
A.requires_grad = True
B = 1/3*A**3
B.sum().backward()
print(A.grad)
```

```
tensor([ 4., 16.])
```

pytorch's most important functionality is **automatic differentiation**

Some helpful pytorch commands

Create new **Tensors** with: `torch.zeros(2,3) randn() ones() linspace arange` etc

use broadcasting and views:

```
import torch
```

```
x_grid = torch.linspace(-1,1,5)
y_grid = torch.arange(0,7)
xy_grid = x_grid.view(-1,1) + y_grid.float().view(1,-1)
print(xy_grid)
```

```
tensor([[-1.0000,  0.0000,  1.0000,  2.0000,  3.0000,  4.0000,  5.0000],
        [-0.5000,  0.5000,  1.5000,  2.5000,  3.5000,  4.5000,  5.5000],
        [ 0.0000,  1.0000,  2.0000,  3.0000,  4.0000,  5.0000,  6.0000],
        [ 0.5000,  1.5000,  2.5000,  3.5000,  4.5000,  5.5000,  6.5000],
        [ 1.0000,  2.0000,  3.0000,  4.0000,  5.0000,  6.0000,  7.0000]])
```

~~**Docstring** can be displayed when cursor is on function (1. in line) with Shift+Tab~~ (not in colab)

```
x_grid = torch.linspace(-1,1,5)

Docstring:
linspace(start, end, steps=100, out=None, dtype=None, layout=torch.strided, device
=None, requires_grad=False) -> Tensor
```

```
torch.nn.functional.gr
torch.nn.functional.grad
torch.nn.functional.grid_sample
torch.nn.functional.group_norm
```

Tab auto-extends function names

helpful python / pytorch commands

Indentation is always necessary in python (but no semi colons needed),
→ Tab or Shift-Tab with cursor at beginning of line,
methods can be called from Tensor object with .operator()

```
with torch.no_grad():
    print(xy_grid.sum())
```

```
tensor(105.)
```

Jupyter notebooks are based on cell execution (similar to %% in Matlab)
Ctrl+Enter or Alt+Enter runs code of current cell
(Alt+Enter in addition jumps into a new line)

Funktionen werden in python mit `def name(arguments, ..):` deklariert
Vorsicht keine klare Unterscheidung zwischen globalen und lokalen Variablen (Lesezugriff auf beides möglich)

```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
data = torch.load('uebung6_data.pth')
img_train = data['img_train']
```

```
def resample(img):
    x_grid = torch.linspace(-1,1,img.size(2)//3).view(1,img.size(2)//3,1,1)\
                .repeat(1,1,img.size(3)//3,1)
    y_grid = torch.linspace(-1,1,img.size(3)//3).view(1,1,img.size(3)//3,1)\
                .repeat(1,img.size(2)//3,1,1)
    xy_grid = torch.cat((y_grid,x_grid),3)
    img_resampled = F.grid_sample(img,xy_grid)
    return img_resampled

img_resampled = resample(img_train[0:1,:,:,:])
plt.imshow(img_resampled[0,0,:,:].detach(),'gray')
plt.show()
```

`tensor`
data container / variable

list of widely used functions

see more details in documentation [https://pytorch.org/docs/stable/index.html)](https://pytorch.org/docs/stable/index.html)

matrix multiply: `torch.matmul(tensor1, tensor2)`

clip the values of a tensor: `torch.clamp(input, min, max)`

swap two specified dimensions: `torch.transpose(input, dim0, dim1)`

short hand for swapping Dim=0 and Dim=1 is .t()

for >3 dimensions: `.permute(*dims) : *dims (int...)` – the desired ordering of dimensions

reshape the size of dimensions, same memory (total number of elements must be equal): **`.view(*shape)`**

select with tensor of integer indices: `torch.index_select(input, dim, index)`

slicing: similar to MatLab, but declared as (start:end:stride) e.g. select every second row of 2D Tensors starting from 3:

`x_ = x[3::2,:]`

repeat a tensor along an axis: `expand(*sizes) : *sizes (torch.Size or int...)` – the desired expanded size

remove a (any) dimension of size 1: `.squeeze()`

add a dimension of size 1: `.unsqueeze(dim)` (useful for broadcasting)

concatenate list of tensors: `torch.cat(tensors, dim) : y = torch.cat((x,x),0)`

return number of dimensions `.dim();` and **return their sizes** `.size(dim);`

change datatype of tensor: `.type(dtype=None) : string e.g. 'float'` or simply `.float().cuda()`

nn
neural network

common deep learning functions

see more details in documentation https://pytorch.org/docs/stable/nn.html

nonlinearity ReLU (with extra argument for slope default=0):

`torch.nn.LeakyReLU()`

convert predictions into pseudo probabilities (for classification)

`torch.nn.LogSoftmax(dim=None)` including logarithm

`torch.nn.Softmax(dim=None)`

convolutional layers for learning features and pooling layers for smoothing / downsampling:

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1,
groups=1, bias=True)
```

```
torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False,
count_include_pad=True)
```

transposed convolution (used in backward path of Conv2d and classical U-Nets):

```
torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
output_padding=0, groups=1, bias=True, dilation=1)
```

loss function for categorical classification / segmentation (use together with log_softmax)

```
torch.nn.NLLLoss(weight=None, size_average=None, ignore_index=-100, reduce=None,
reduction='mean')
```

and loss function for continuous regression (L1 norm, absolute differences):

```
torch.nn.L1Loss(size_average=None, reduce=None, reduction='mean')
```

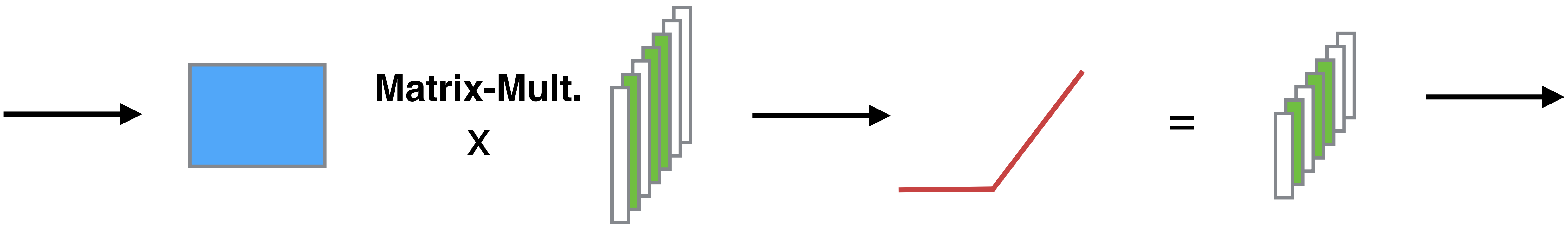
extract all overlapping patches (n-D) e.g. to create sparse networks or implement own filters:

```
torch.nn.Unfold(kernel_size, dilation=1, padding=0, stride=1)
```

some words on the difference to nn.functiona

Stochastic Gradient Descent (SGD)

conventional optimisation would sum gradient over all data points
inefficient for larger datasets, **cancelling out** of opposing gradients → **mini-batches**



Parametermatrix **W**
to learn: **5 x 10 + Bias**

Data **X**
Feature-Dimension 10

Nonlinearity
ReLU

Output Z
(next layer)

find **solution** starting with random init of **w**

Gradient Descent: $$\vec{w}^{(k)} = \vec{w}^{(k-1)} - \eta \frac{dL}{d\vec{w}}$$

important: manual tuning of **learning rate *η***

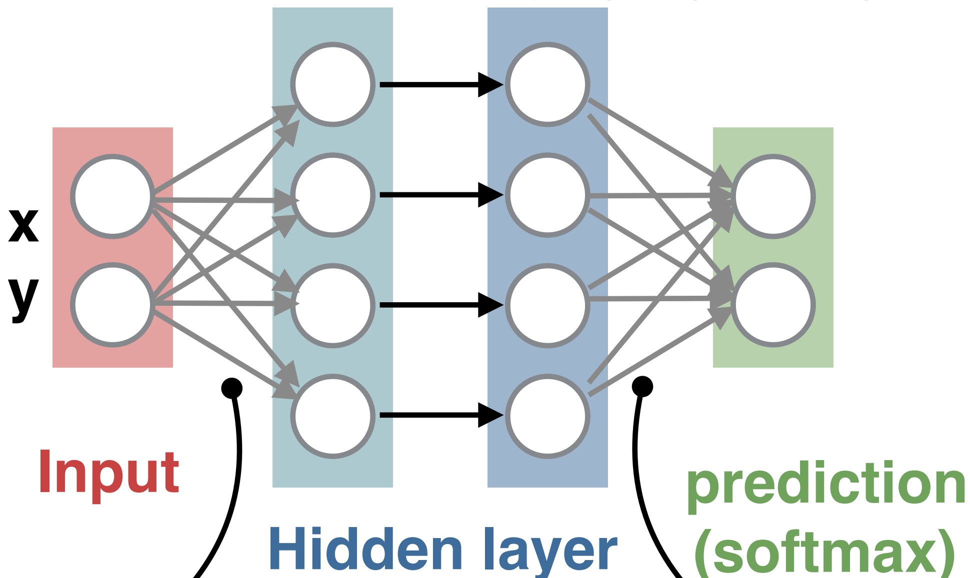
improved learning with **momentum**

$$\vec{w}^{(k)} = \vec{w}^{(k-1)} + \eta \underbrace{\left(\mu \vec{m}^{(k-1)} - \frac{dL}{d\vec{w}}\right)}_{\text{Momentum } \vec{m}^{(k)}}$$

→ **pytorch perform automatic differentiation for you, only forward path definition is necessary**

Multilayer Perceptron

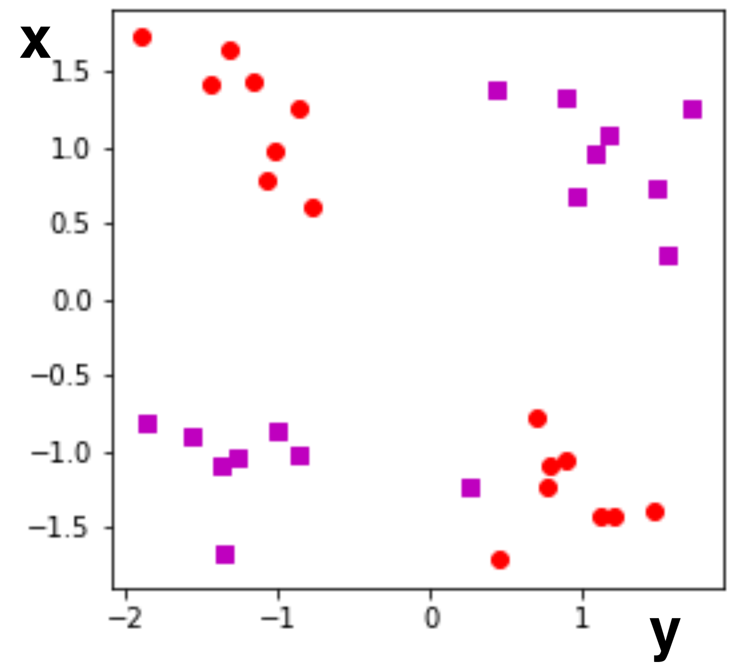
ReLU activation



weight matrix
(4x2) $\mathbf{W_1}$ (+ B_1 bias)

weight matrix
(2x4) $\mathbf{W_2}$ (+ B_2 bias)

XOR problem
→ not linearly separable

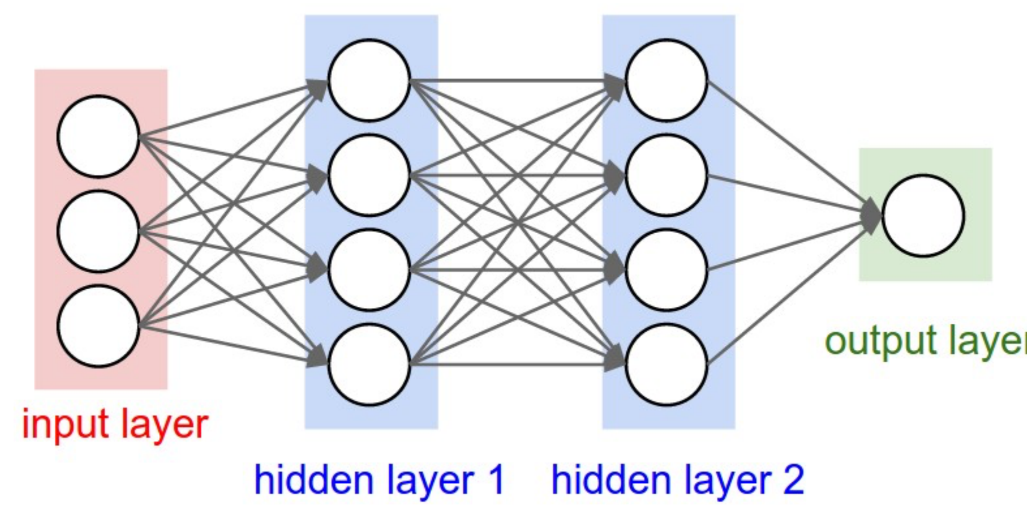


complex non-linear transfer function

$$f = W_2 \max(0, W_1\vec{x}+B_1)+B_2$$

→ differentiable with chain rule (autograd)

→ neural networks (including CNNs) comprise of arbitrarily many hidden layers >100



Example: Training MLP for XOR classification

pytorch implementation of two-layer MLP with 4 hidden neurons

```
import torch
import torch.nn as nn
import torch.nn.functional as F

#define multilayer perceptron
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.linear1 = nn.Linear(2,4)
        self.linear2 = nn.Linear(4,2)
    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x

net = MLP()
#loss criterion and SGD optimizer
crossEntropy = nn.CrossEntropyLoss()
```

import necessary torch libraries with shorthands nn and F

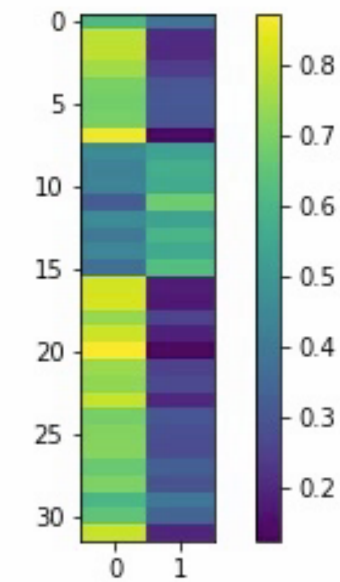
initial definition has to contain all trainable parameters

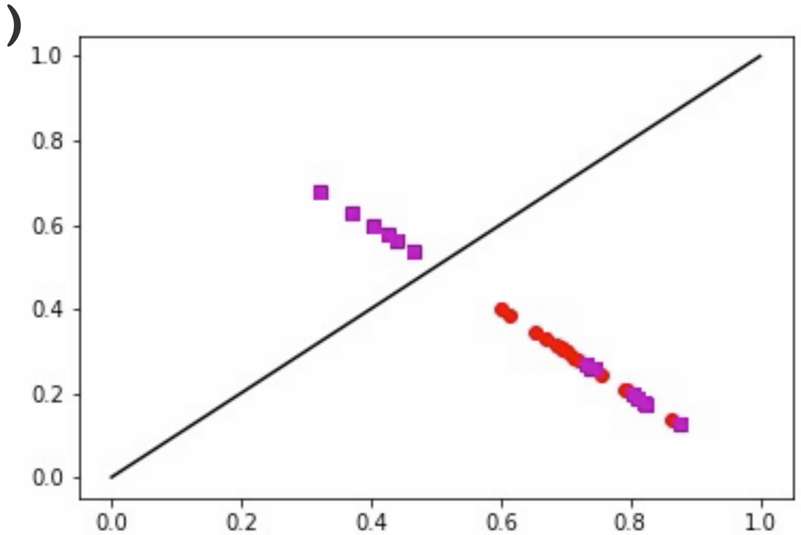
network operations are only defined for forward path (classic relu has no trainable weight →

Example: Training MLP for XOR classification

```
#SGD optimizer and training procedure
optimizer = optim.SGD(net.parameters(), lr=0.025, momentum=0.95)
run_loss = np.zeros(20)
#iterate over several epochs
for epoch in range(20):
    run_loss[epoch] = 0.0
    idx_epoch = torch.randperm(32).view(4,8)
    #mini-batches consider subsets of whole dataset
    for iter in range(8):
        idx_iter = idx_epoch[:,iter]
        optimizer.zero_grad()
        #forward path and loss
        input = data[idx_iter,:,:,:]
        outputs = net(input)
        loss = crossEntropy(outputs, label[idx_iter,:,:])
        #backward path and weight updates
        loss.backward()
        optimizer.step()
        run_loss[epoch] += loss.item()
```

Softmax output (during learning iterations)





evolution of decision boundary