Lecture 11:

Convolution and Image Classification & Segmentation

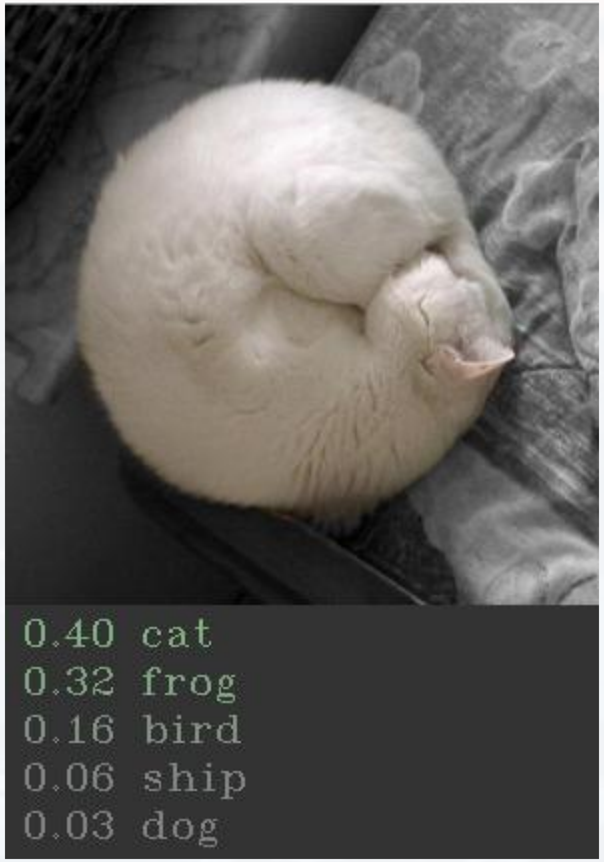
Markus Hohle

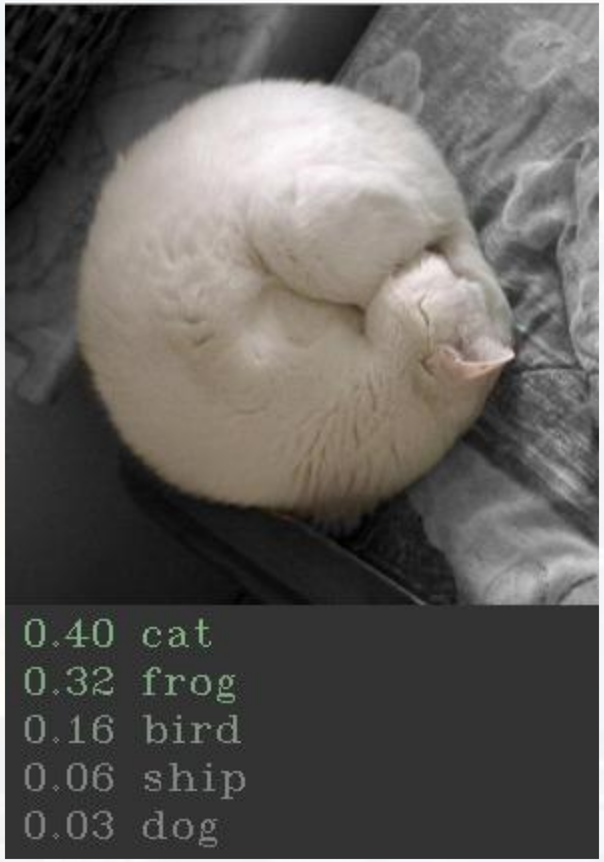
University California, Berkeley

Bayesian Data Analysis and Machine Learning for Physical Sciences

Course Map

Part III



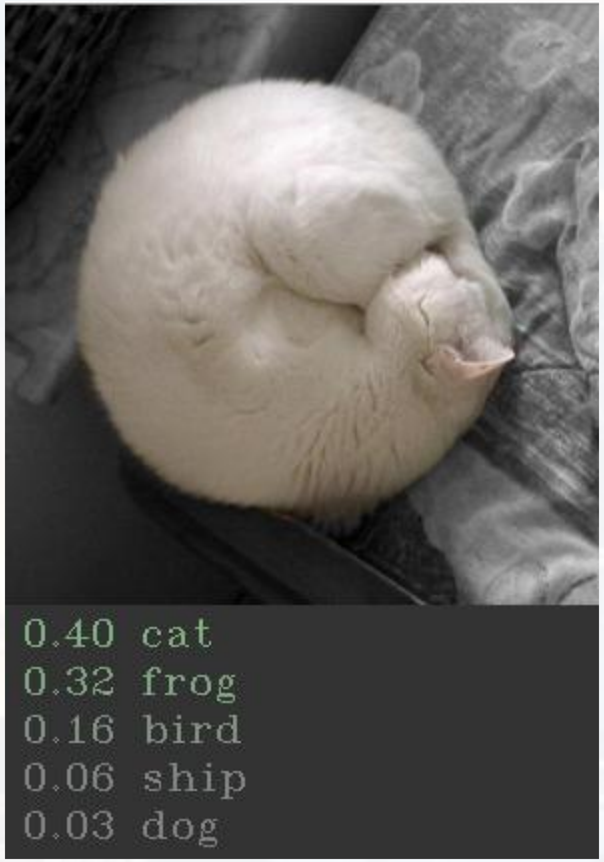


Outline

PyTorch & Cuda

Generative Adversarial Network (GAN)

Variational AutoEncoder (VAE)



Outline

PyTorch & Cuda

Generative Adversarial Network (GAN)

Variational AutoEncoder (VAE)

problem: even for moderate setups, computational time becomes the limiting factor

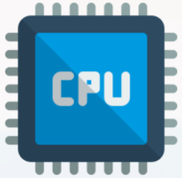
self made ANN → keras/tensor flow → pytorch → pytorch on GPU (Cuda)

speed

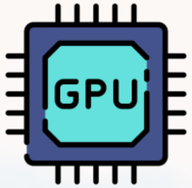
```
512/512 [==============================] - 4072s 8s/step - loss: 3.4534 - accuracy: 0.3338
Epoch 4/5
512/512 [==============================] - ETA: 0s - loss: 3.1926 - accuracy: 0.3980    saved  ../data/segmentation
pics/checkpoints//.3
512/512 [==============================] - 3363s 7s/step - loss: 3.1926 - accuracy: 0.3980
Epoch 5/5
512/512 [==============================] - ETA: 0s - loss: 3.0071 - accuracy: 0.4318   saved  ../data/segmentation
pics/checkpoints//.4
512/512 [==============================] - 3616s 7s/step - loss: 3.0071 - accuracy: 0.4318
```

Lenovo T14, NVIDIA GeForce MX450: (simple LSTM)

Keras (CPU):	**300 sec**
PyTorch (CPU):	11 sec
PyTorch (GPU):	**3 sec**



training AI → mainly matrix operations
→ GPUs are a lot better at it!



CUDA is the link of your GPU to Python (PyTorch)

check, if graphic card is on list

check your graphics device:

→ Windows command shell prompt

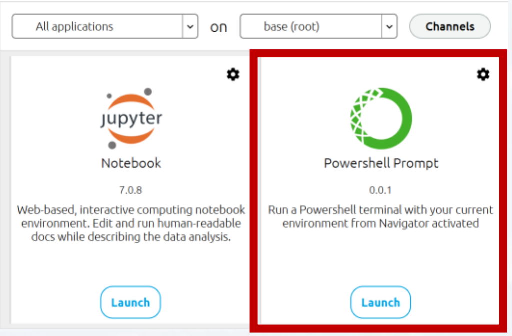
→ type `nvidia-smi`

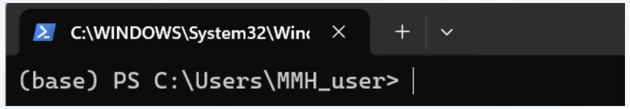
→ press *Enter*

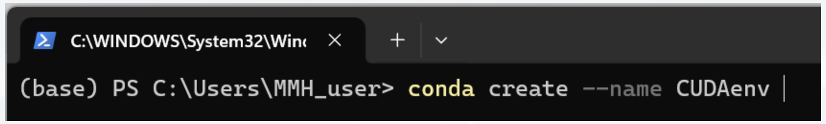
```
(base) C:\Users\MMH_user>nvidia-smi
Tue Jan 16 20:09:26 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 537.79                 Driver Version: 537.79       CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                     TCC/WDDM  | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce MX450         WDDM  | 00000000:01:00.0 Off |                  N/A |
| N/A   49C    P0              N/A /   9W |      0MiB /  2048MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

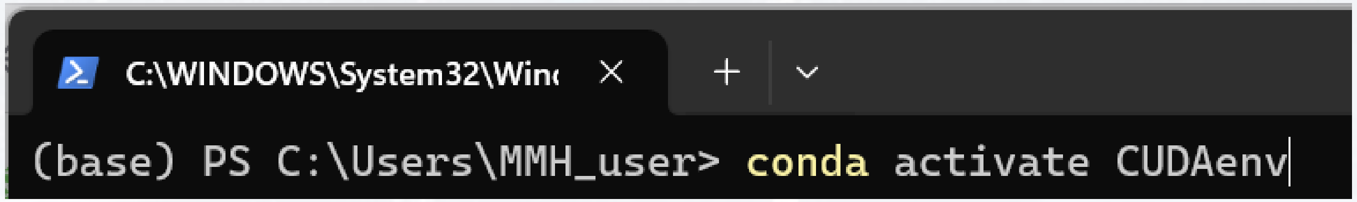
Installing CUDA

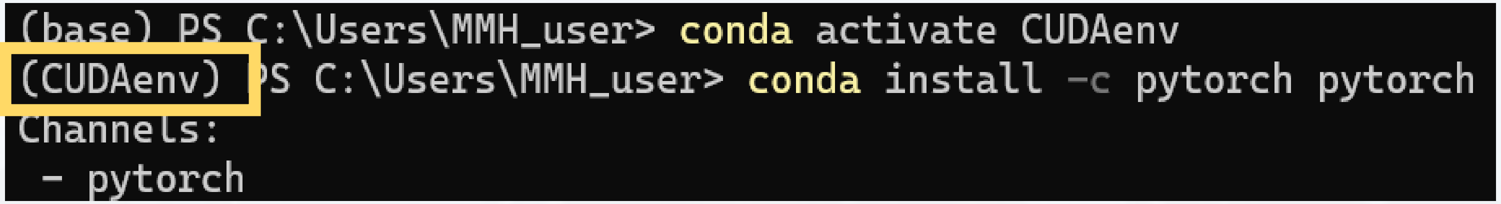
conda environment











Installing CUDA

cuda toolkit

```
(CUDAenv) PS C:\Users\MMH_user> conda install -c anaconda cudatoolkit
```

check libraries

→ type: `conda list`

```
(CUDAenv) PS C:\Users\MMH_user> conda list
# packages in environment at C:\Users\MMH_user\anaconda3\envs\CUDAenv:
#
# Name                    Version                   Build  Channel
blas                      1.0                         mkl
bzip2                     1.0.8                h2bbff1b_6
ca-certificates           2024.7.2             haa95532_0
cudatoolkit               11.8.0               hd77b12b_0
expat                     2.6.2                hd77b12b_0
filelock                  3.13.1          py312haa95532_0
intel-openmp              2023.1.0          h59b6b97_46320
```

Installing CUDA

usually, a few libraries are missing

check again graphics card:	type in anaconda	`nvidia-smi`

check libraries:	type in anaconda	`conda list cudnn`
		`conda list cudatoolkit`
		`conda list torch`

	if not:	`conda install <library>`

check Python:	type in anaconda	`python`
		`import torch`
		`torch.cuda.is_available()`

open Spyder	run in Spyder	`pip install` (see the commented line in `CheckMyCuda.py`)

Installing CUDA

usually, a few libraries are missing

CheckMyCuda.py

```
import torch

def test_cuda():
    print("PyTorch version: ", torch.__version__)
    print("CUDA version: ", torch.version.cuda)
    print("CUDA Available: ", torch.cuda.is_available())
    if torch.cuda.is_available():
        print("Number of GPUs: ", torch.cuda.device_count())
        print("GPU Name: ", torch.cuda.get_device_name(0))

if __name__ == "__main__":
    test_cuda()
```

```
PyTorch version:  2.3.1+cu118
CUDA version:  11.8
CUDA Available:  True
Number of GPUs:  1
GPU Name:  NVIDIA GeForce MX450
```

The key part in PyTorch is to set all matrices and the model **to the device (CPU or GPU)**

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
```

```
In [13]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    ...: print("Using device:", device)
Using device: cuda
```

Congratulation! If you see this, you are ready to go!

The key part in PyTorch is to set all matrices and the model **to the device (CPU or GPU)**

```
In [13]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    ...: print("Using device:", device)
Using device: cuda
```

Torch objects like **`model`** or **`torch.tensor`** have the property **`.to`**

```
TrainX = torch.tensor(TrainX, dtype = torch.float32)
TrainY = torch.tensor(TrainY, dtype = torch.float32)
```

```
TrainX = TrainX.to(device)
TrainY = TrainY.to(device)
```

turning numpy array into torch.tensor

```
model  = model.to(device)
```

allocating objects to the device

The key part in PyTorch is to set all matrices and the model **to the device (CPU or GPU)**

```
In [13]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    ...: print("Using device:", device)
Using device: cuda
```

When running the training, we need to **synchronize** between GPU (for training the model) and CPU (for everything else)…

```
torch.cuda.synchronize()

      #training loop
      for epoch in range(n_epochs):
            outputs = model(TrainX)

            ... #do stuff...

torch.cuda.synchronize()
```

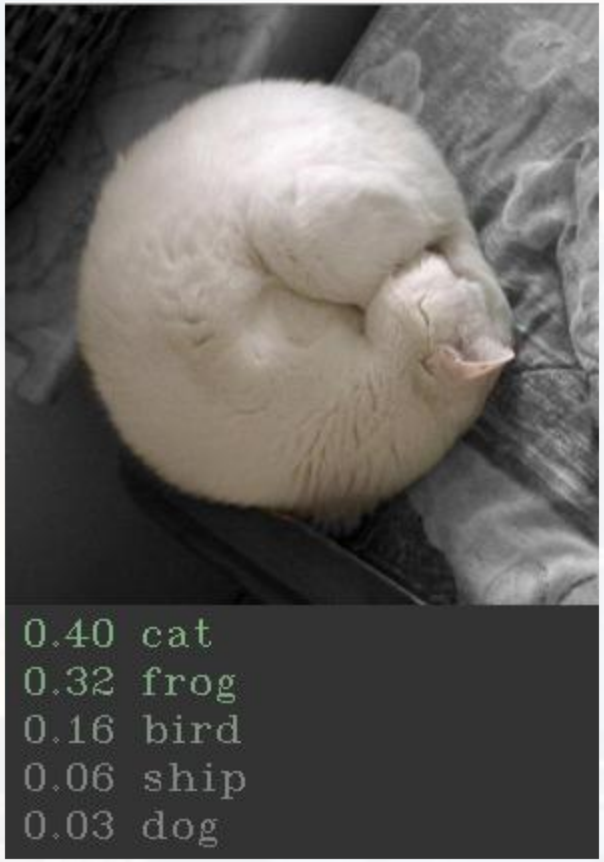
The key part in PyTorch is to set all matrices and the model **to the device (CPU or GPU)**

```
In [13]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    ...: print("Using device:", device)
Using device: cuda
```

When running the training, we need to **synchronize** between GPU (for training the model) and CPU (for everything else)…

…and later detach the model from the GPU

```
PredY = model(TestX).detach().to('cpu').numpy()
```



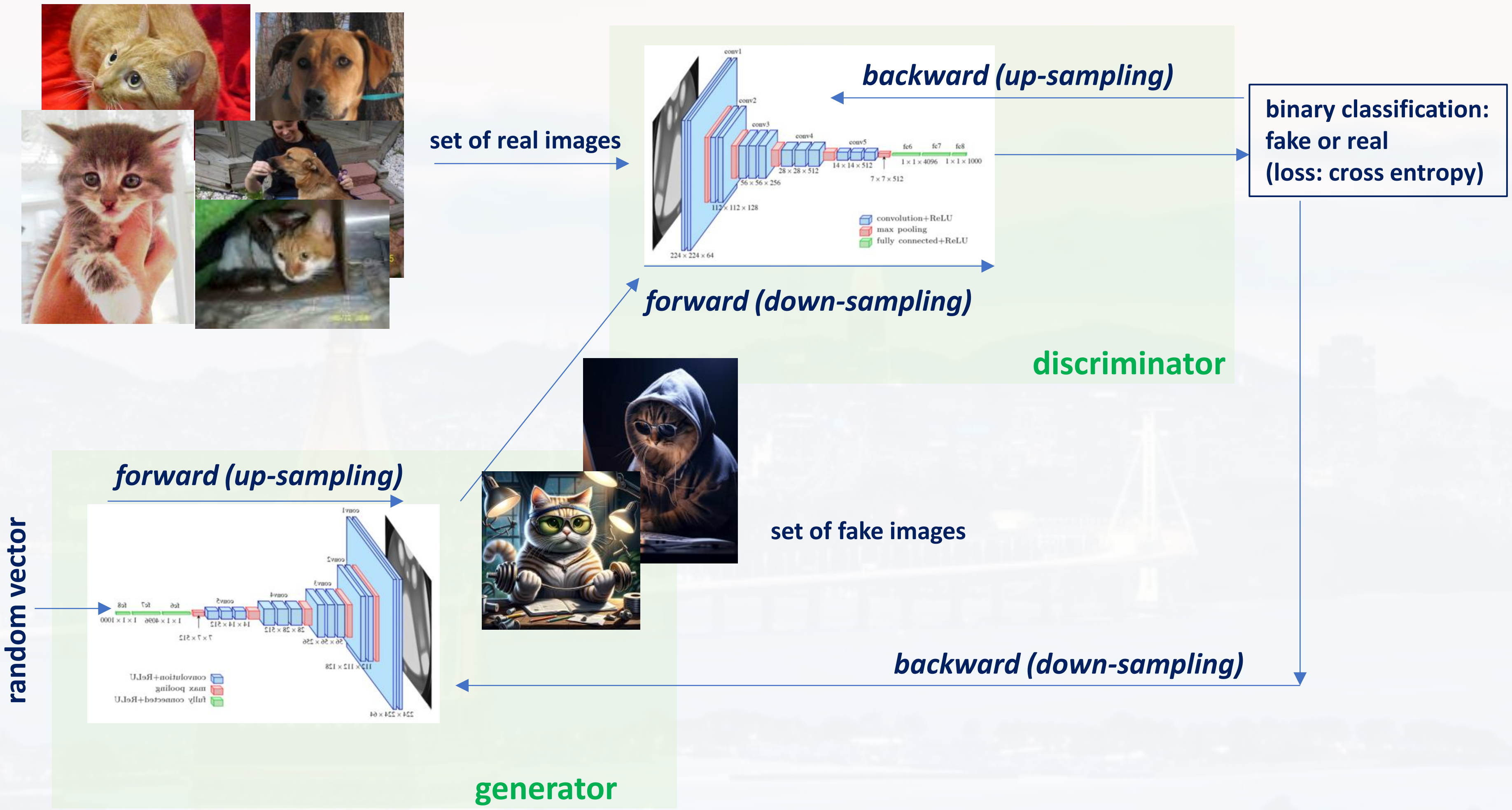
Outline

PyTorch & Cuda

Generative Adversarial Network (GAN)

Variational AutoEncoder (VAE)

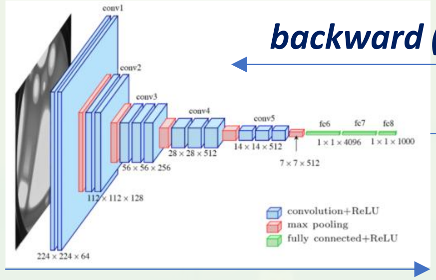
Convolution, Image Classification & Segmentation



Convolution, Image Classification & Segmentation

set of real images

backward (up-sampling)



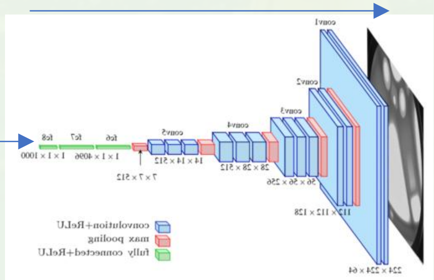
forward (down-samplin

binary classification:
fake or real
(loss: cross entropy)

set of fake images

forward (up-sampling)

random vector



generator

bac

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()

        self.model = nn.Sequential(
        nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1),
        nn.LeakyReLU(0.2),
        nn.Dropout(0.25),
        nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
        nn.ZeroPad2d((0, 1, 0, 1)),
        nn.BatchNorm2d(64, momentum=0.82),
        nn.LeakyReLU(0.25),
        nn.Dropout(0.25),
        nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
        nn.BatchNorm2d(128, momentum=0.82),
        nn.LeakyReLU(0.2),
        nn.Dropout(0.25),
        nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256, momentum=0.8),
        nn.LeakyReLU(0.25),
        nn.Dropout(0.25),
        nn.AdaptiveAvgPool2d((1, 1)),
```

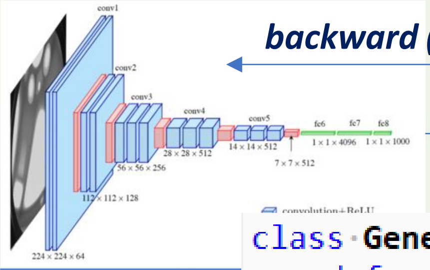
```
nn.Sigmoid()
```

standard classification CNN

Convolution, Image Classification & Segmentation

set of real images

backward (up-sampling)



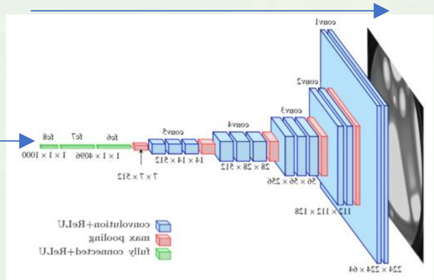
forward (down-

binary classification:
fake or real
(loss: cross entropy)

set of fake images

random vector

forward (up-sampling)



generator

```python
class Generator(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()

        self.model = nn.Sequential(
            nn.Linear(latent_dim, 128 * 8 * 8),
            nn.ReLU(),
            nn.Unflatten(1, (128, 8, 8)),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128, momentum=0.78),
            nn.ReLU(),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64, momentum=0.78),
            nn.ReLU(),
            nn.Conv2d(64, 3, kernel_size=3, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        return img
```

**up-sampling:
like in standard
segmentation CNN**

Convolution, Image Classification & Segmentation

x: real image
z: latent vector (random numbers)

$p_D(x)$: probability that discriminator *D* classifies *x* as real image
$G(z)$: fake image generated from generator *G*

$\mathcal{L}_D$: loss function of discriminator *D*
$\mathcal{L}_G$: loss function of generator *G*

(mean) binary cross entropy (BCE): $\mathcal{L}_D = -\frac{1}{N}\sum_{i=1}^{N} \log[p_D(x_i)] + \log[1 - p_D(G(z_i))]$

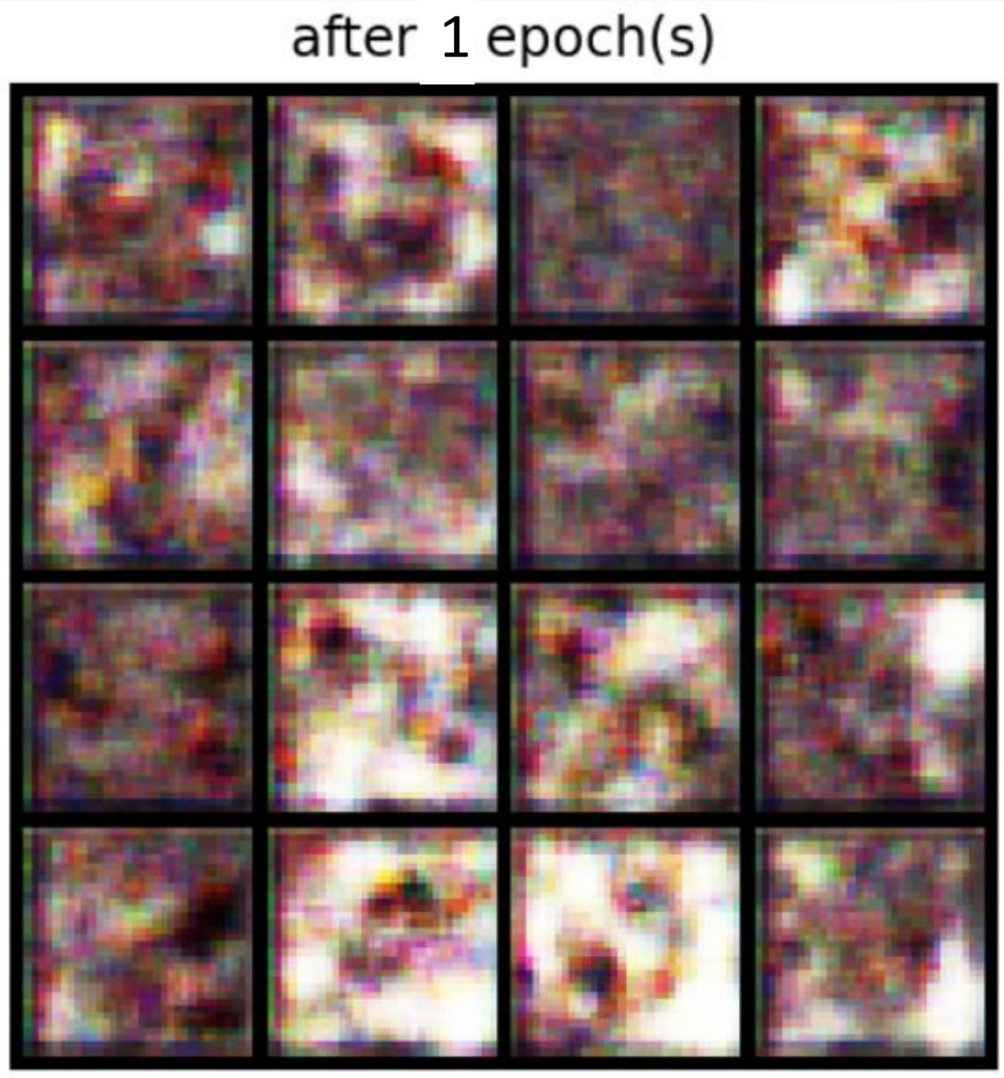
over all *N* images $\mathcal{L}_G = -\frac{1}{N}\sum_{i=1}^{N} \log[1 - p_D(G(z_i))]$

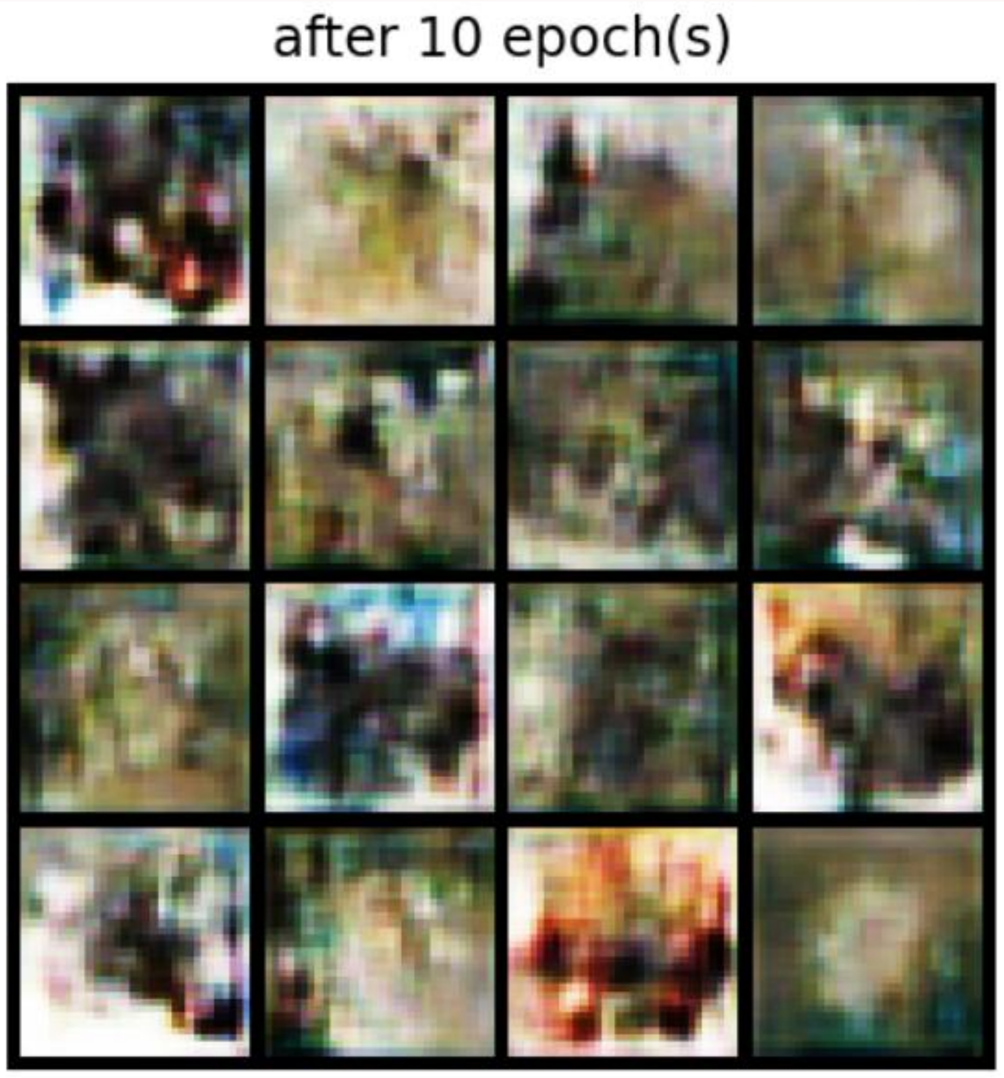
ideal case for *D*: $\mathcal{L}_D = -\log[1] - \log[1 - 0] = 0$
worst case for *D*: $\mathcal{L}_D = -\log[1] - \log[1 - \mathbf{0.5}] = \mathbf{0.69}$

ideal case for *G*: $\mathcal{L}_D = \log[1 - \mathbf{0.5}] = \mathbf{-0.69}$
worst case for *G*: $\mathcal{L}_D = \log[1 - 0] = 0$

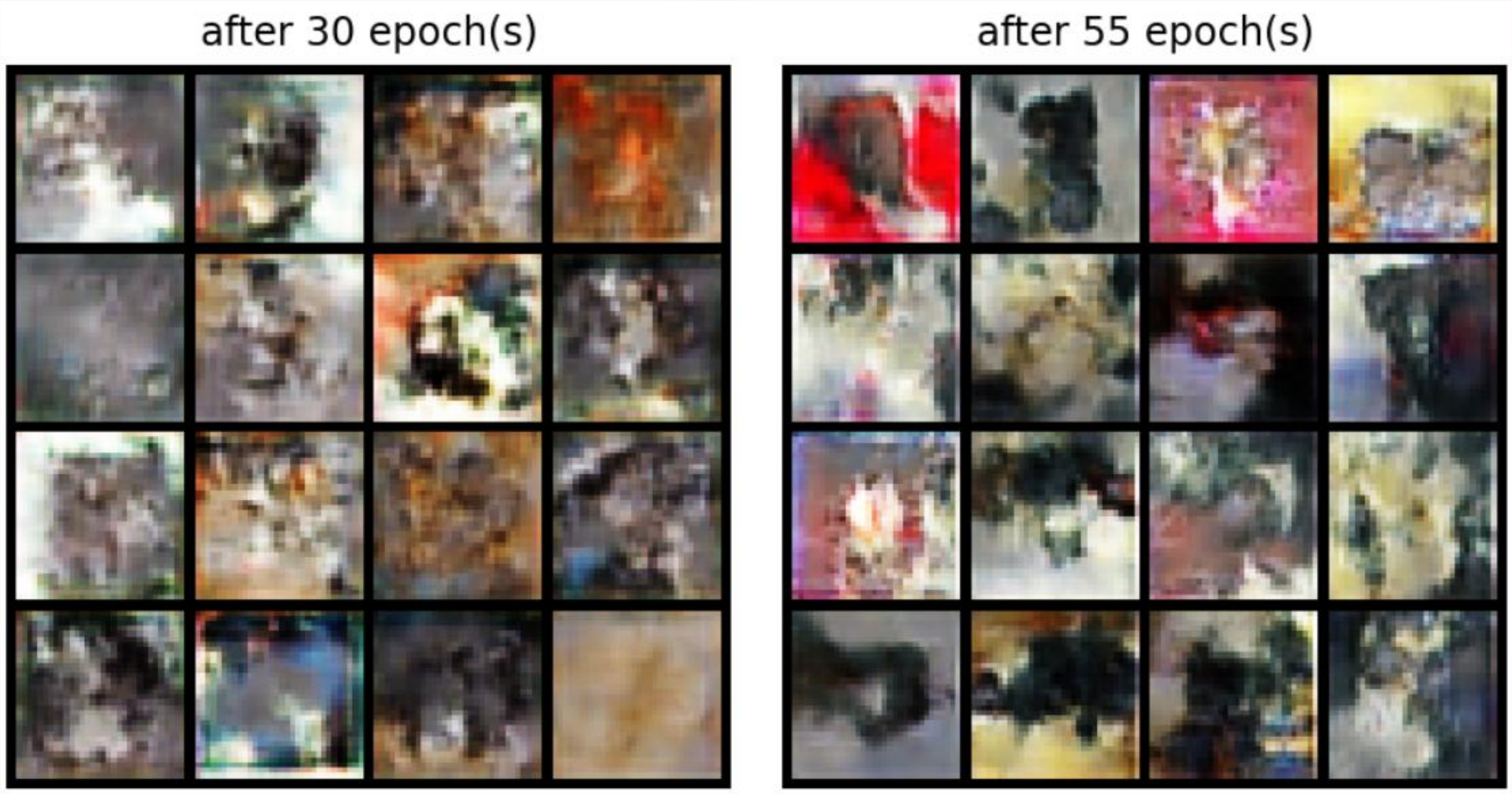
Convolution, Image Classification & Segmentation

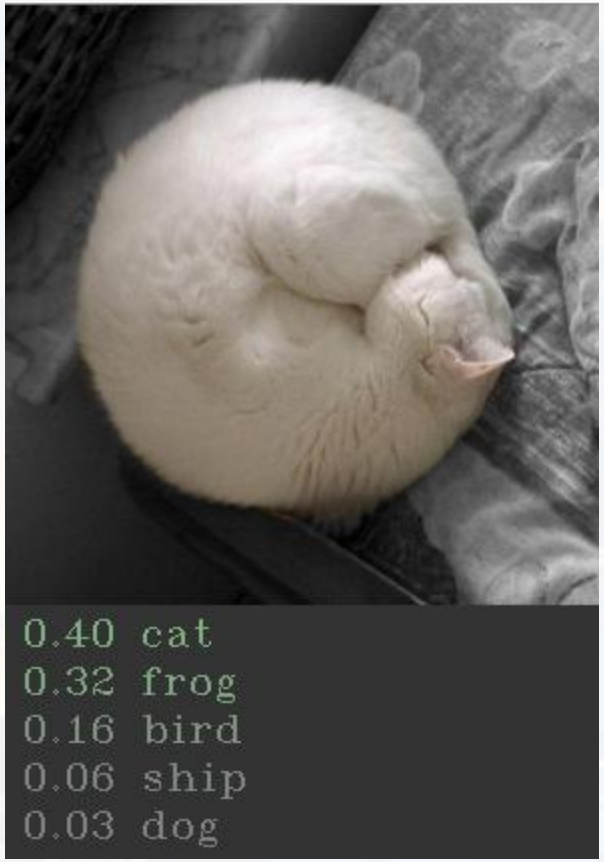
check out `GAN architecture.ipynb`





check out `GAN architecture.ipynb`





Outline

PyTorch & Cuda

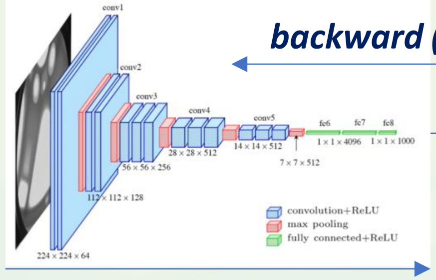
Generative Adversarial Network (GAN)

Variational AutoEncoder (VAE)

GAN:

set of real images

backward (up-sampling)



forward (down-sampling)

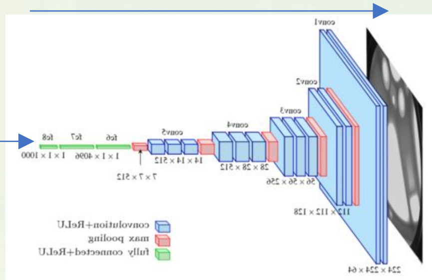
binary classification:
fake or real
(loss: cross entropy)

discriminator

set of fake images

forward (up-sampling)

random vector



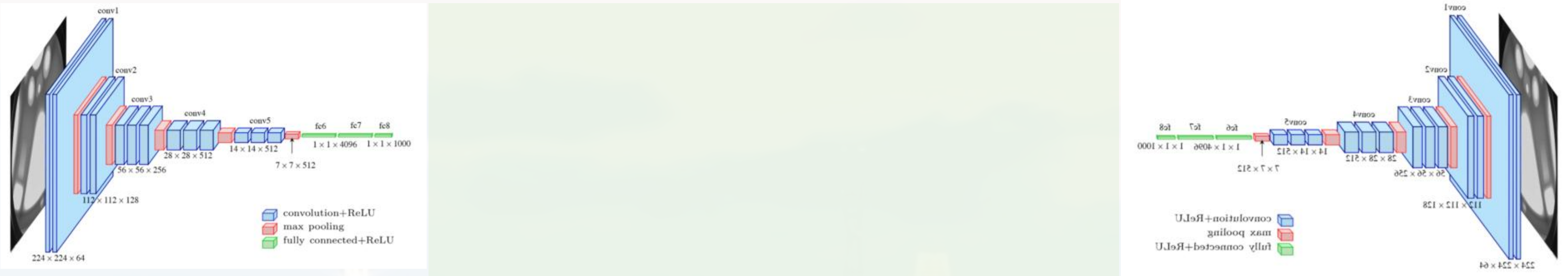
backward (down-sampling)

generator

each image x is represented by **one single** vector z

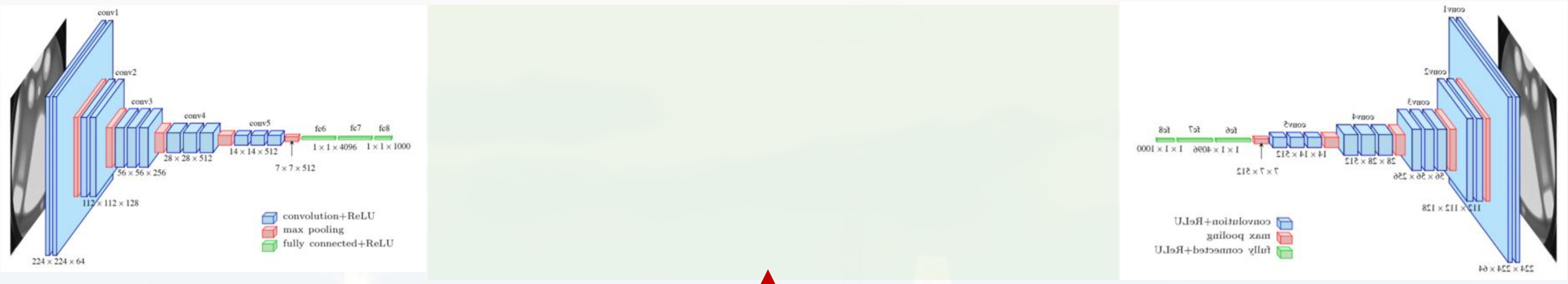
Convolution, Image Classification & Segmentation

now:



Convolution, Image Classification & Segmentation

now:



GAN: each image x is represented by **one single** vector z

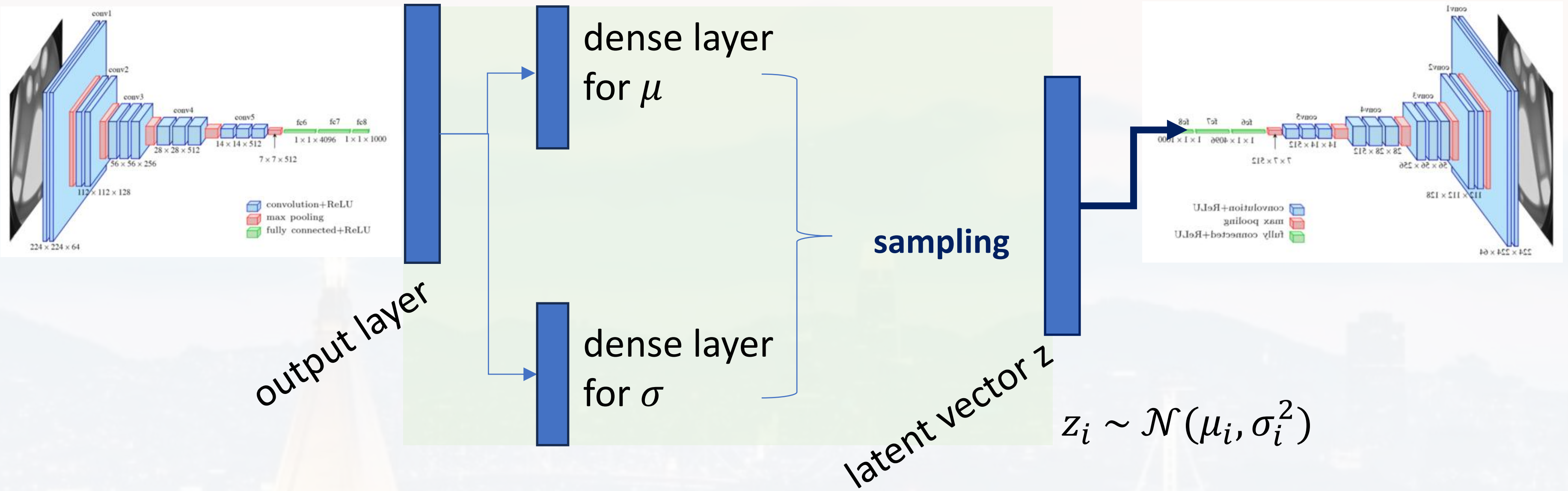
VAE: each image x is represented by **a distribution** we sample from in order **to get z**

i: index over dimensions of z

$z_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$ (max entropy!)

→ generate image from up-sampling as before

→ randomnes accounts for diverisity in generated data now:



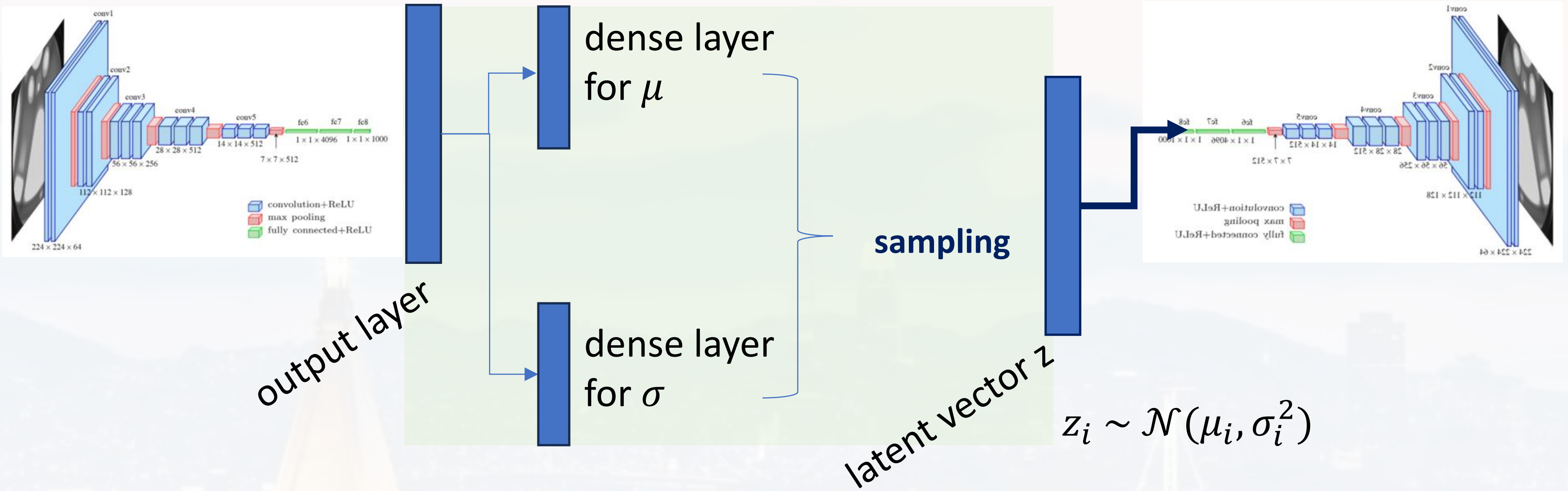
example: output layer has shape `(None, N)`

we want M dimensional encoding for z

→ weights in dense layer for σ and μ have shape `(N, M)`

→ latent vector has length N

now:



$z_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$

backprop:

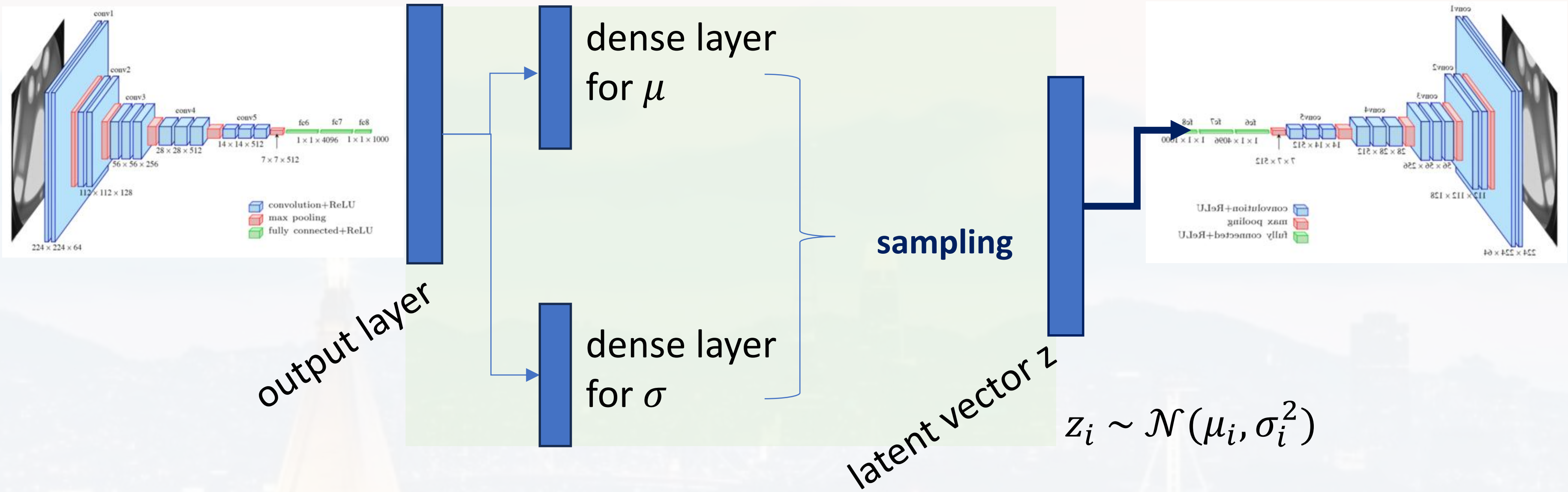
- μ_i and σ_i^2 have been generated deterministically from a dense layer
- we need to run backpropagation through μ_i and σ_i^2 in order to adjust the weights/biases
- in order to keep randomness, we actually sample via

$z_i = \mu_i + \sigma_i \cdot \epsilon$ where $\epsilon \sim \mathcal{N}(0,1)$ is not affected by backpropagation

Convolution, Image Classification & Segmentation

VAE

now:



$z_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$

loss: 1) **reconstruction loss $\boldsymbol{L_R}$**, i. e. the difference between input image x and reconstructed image $\boldsymbol{p(x|z)}$. Note: L_R alone would **lead to overfitting**, exact image gets reproduced!

2) we need a **regularization term** that counteracts $\boldsymbol{L_R}$:

KL-divergence $\boldsymbol{L_{KL} = KL[q(z|x)|p(z)]}$, which increases if learned distribution $\boldsymbol{q(z|x)}$ gets too far from prior $\boldsymbol{p(z)} \sim \mathcal{N}(0,1)$

loss: 1) **reconstruction loss $\boldsymbol{L_R}$**, i. e. the difference between input image x and reconstructed image $p(x|z)$. Note: L_R alone would **lead to overfitting**, exact image gets reproduced!

2) we need a **regularization term** that counteracts $\boldsymbol{L_R}$:

KL-divergence $\boldsymbol{L_{KL} = KL[q(z|x)|p(z)]}$, which increases if learned distribution $\boldsymbol{q(z|x)}$ gets too far from prior $\boldsymbol{p(z)}\sim\mathcal{N}(0,1)$

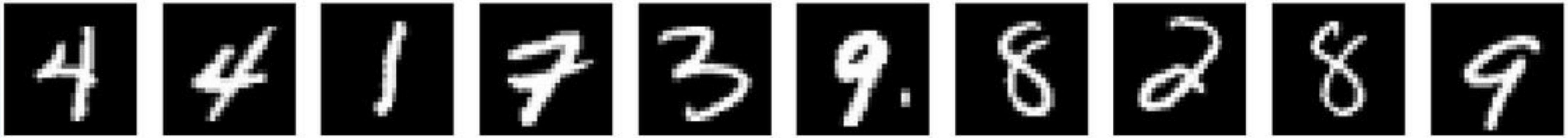
see here and **module 6 (variational Bayes)**

check out `VAR architecture.ipynb`

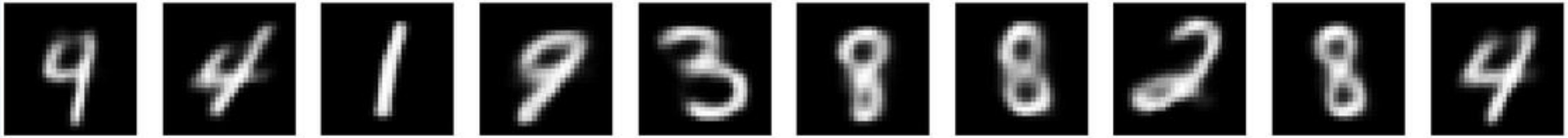
VAE

check out VAR architecture.ipynb

original



reconstruction



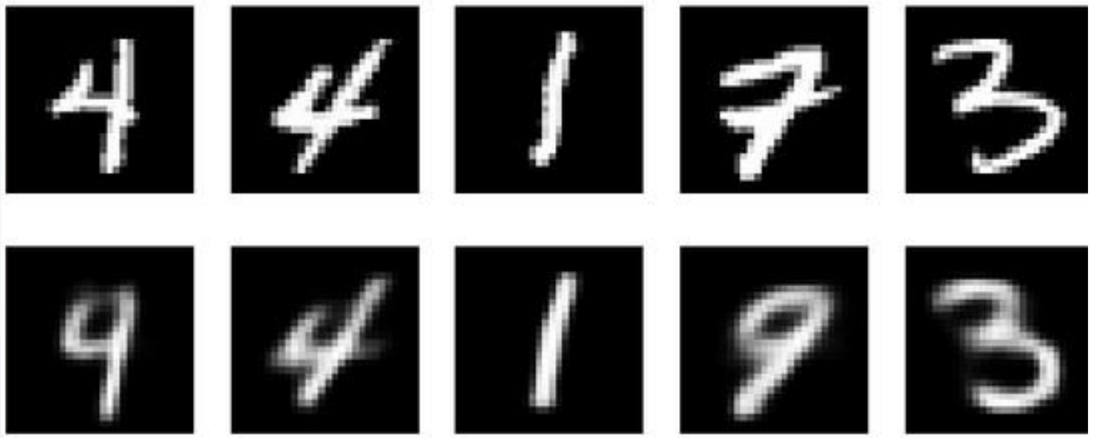
generated

Convolution, Image Classification & Segmentation

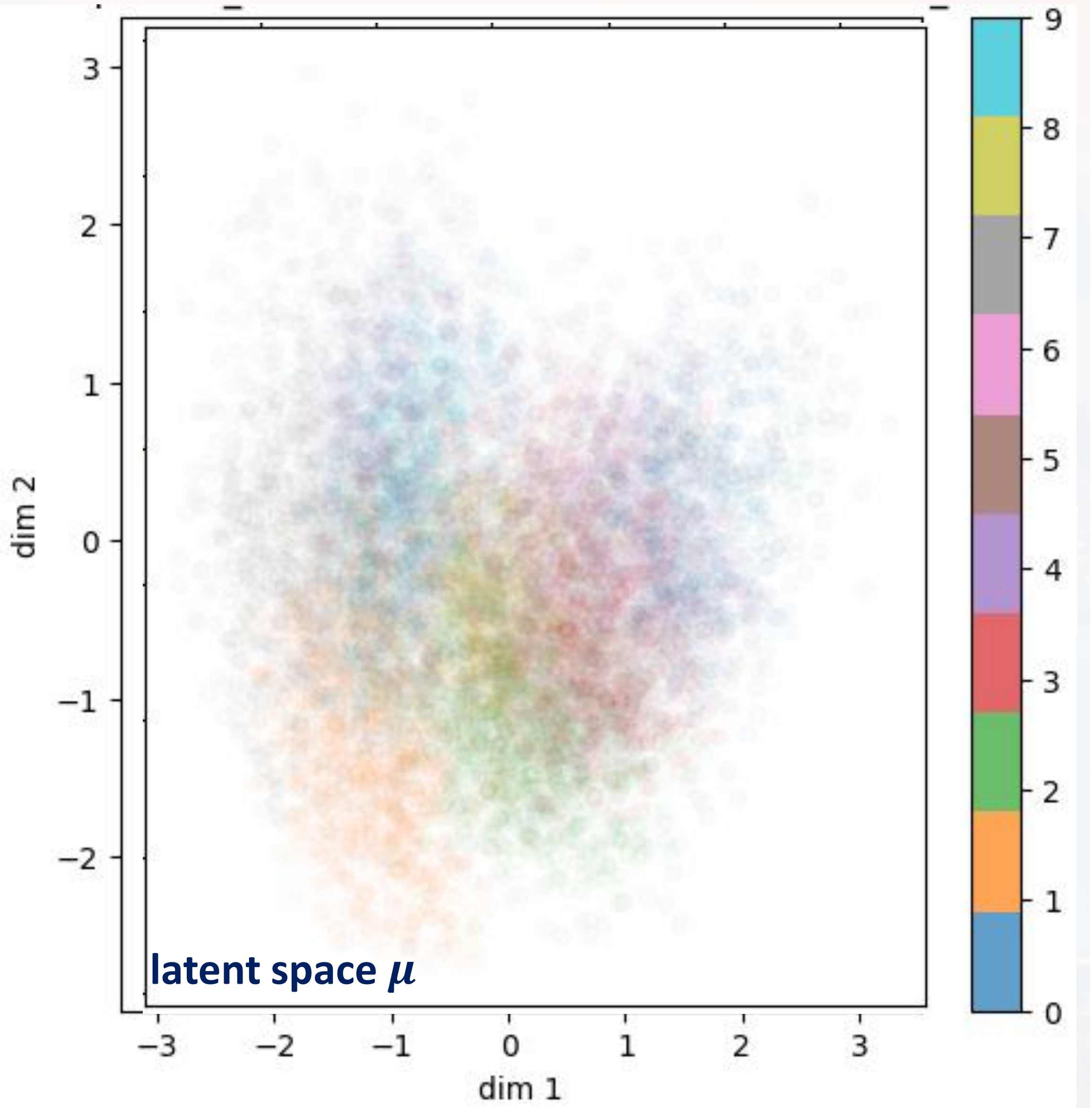
check out `VAR architecture.ipynb`

original

reconstruction



generated



Thank you very much for your attention!