P04 – Unsupervised Learning

1. Predictive maintenance with autoencoders

1.1. Background

During the lecture, you got familiar with different types of autoencoders as well as different use cases. In this exercise, you will have a close look at an application of novelty detection to condition monitoring.

Condition monitoring is the process of monitoring a parameter associated with a certain condition in machinery (e.g., vibration or temperature that are related with bearing malfunctions). The goal is to identify a significant change which is indicative of a developing fault. In this exercise, you will monitor the reconstruction error of a measured vibration signal to produce a «novelty signal» (see Fig. 1).

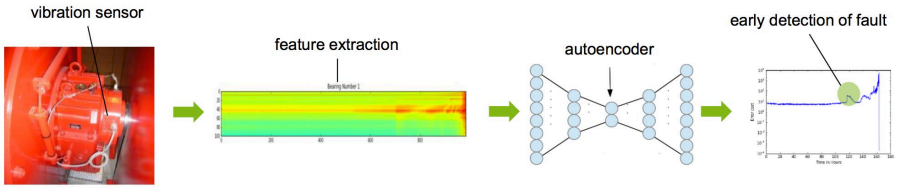

Figure 1: The process of novelty detection.

You will work with a bearing data set from NASA, provided by the Centre for Intelligent Maintenance Systems (IMS) [1]. In order to focus on autoencoders, you are provided with extracted features (spectrograms) as well as code templates. In order to run the template, you need TensorFlow (with Keras) installed. Therefore, update your Anaconda installation using the Anaconda prompt and the following command:

```
pip install tensorflow
conda install matplotlib
```

1.2. Novelty detection for predictive maintenance

1.2.1 NASA data exploration:

a. Explore the NASA dataset:

Open the notebook `nasa_data_exploration.ipynb` and analyse the spectrograms from the training and testing set.

- *What is the difference between the training and test set?*
- *Can you find any errors or anomalies?*
  (Hints: Have a closer look to the sensor 0 from the testing set.)

b. Normalize the data set and plot the spectrograms
(*Hint: use the plot_spectrogram_features and scale function)*
   - *What do you observe?*

1.2.2. Autoencoder implementation:

c. Complete the notebook for the compressing autoencoder (`nasa_autoencoder_compression.ipynb`):
   - Assign the right value to the *n_features* variable
   - Add a Dense layer as output layer to the model.
     i) What is the output dimension?
   - Analyse the fitting method (cell 13):
     i) What do you observe?
   - Analyse the reconstruction error plot.
     Hint: You should see a fault on t>600.

d. Create a sparse autoencoder:
   - Use the compressing autoencoder as a template and create a sparse autoencoder.
     i) Which variable do you have to change?
     ii) *How does the reconstruction error plot change?*

e. Create a denoising autoencoder:
   - Use the compression autoencoder as template and add a noise layer.
   - *Hint: Use the GaussianNoise[1] layer from Keras*

f. [Optional] Create a deep autoencoder:
   - *Do you think you can improve the result by adding additional dense layers?*
   - Use the compressing autoencoder as a template and add additional dense layers.

[1] https://keras.io/api/layers/regularization_layers/gaussian_noise/

[1] J. Lee, H. Qiu, G. Yu, and J. Lin, "Rexnord technical services", Bearing Data Set, IMS, University of Cincinnati, NASA Ames Prognostics Data Repository, NASA Ames, Moffett Field, CA, https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/ [Online; accessed 2021-09-15]

2. Image synthesis with GANs

2.1. Make it run

We use the concise DCGAN template of Arthur Juliani's Medium post[2]. You can either download and use the IPython notebook from his Github account[3] or take the derived Python 3 script (`DCGAN.py`) that comes with this lab description.

```
pip install tf_slim
conda install scipy scikit-learn imageio
```

To run either script, you need Python 3 with TensorFlow and some other libraries up and running. Ideally, you take an existing environment via a Docker container:

1. Install Docker and get a suitable image with the deep learning software stack installed (see appendix A)
2. Start your Docker container (see appendix B)
3. Install additional libraries (see above) *within the running Docker container* by executing a shell (docker exec… → see appendix B

Copy your scripts/notebooks into the running Docker container. Execute them. If you don't get errors but occasional outputs (loss values, synthesized images), your setup is fine.

2.2. Get acquainted with the code

Train the two networks for a few thousand iterations (<50'000) and inspect the generated images: How does training improve over time? Does it? Is the generator learning useful things? How would you characterize the learnt information?

Generally, training on the CPU is extremely slow compared to using a GPU: While training on a laptop's CPU might take 10 Minutes per 100 iterations, it is 10-100 faster on a Cuda-enabled GeForce-GPU. Consider training on fast hardware as you will need more than 5'000 iterations to draw first conclusions.

[2] https://medium.com/p/54deab2fce39
[3] https://gist.github.com/awjuliani/8ebf356d03ffee139659807be7fa2611

2.3. Analyse the training process

Background: Salimans et al.'s paper "Improved Techniques for Training GANs" shows the potential of better insight into what is going wrong and how to remedy these errors; here's another idea, out of intuitive appeal:

Usually, a mini batch for GAN training is comprised of m random images from the generator $G(z)$, as well as m randomly drawn images from the training set for $D(x)$. The $G(z)$ and x values are thus unrelated, potentially leading to a situation where (say, for a dataset with images of fruits) the generator outputs apples, but the dataset (by chance) includes only oranges. Using this data to train, it teaches D to tell apples from oranges instead of fake from real, and G to paint oranges instead of more realistic apples.

This might be exploited for a potentially improved training process as follows:

Algorithm: For the next mini batch, …

- Randomly draw m images from the training data
- For each x_i in the m images, let G generate a visually fitting fake image by
    1. Fixing G's weights
    2. Optimizing the output $G(z)$ (via gradient descent, by finding a suitable z) to best match the current x_i…
    3. …by e.g. minimizing the L1 norm between the pixel values of both images.
- Proceed as usual by feeding D with the m training and m generated images, compute the usual loss and update D's and G's parameter according to the respective gradients

This way, the m generated images per mini batch are the best the generator can do in its current form to mimic this particular batch of training data. The discriminator should be able to learn subtle features much more quickly, and the generator could be driven to learn improve mimicking in general: instead of alternating the kind of objects produced in each mini batch, it learns to improve the appearance of objects themselves.

Task: Implement the algorithm above and perform a short experiment indicating if the sketched idea improves GAN training in general. Does the intuitive idea stand the experimental test? How do you interpret your results? Do they support your intuition? What conclusions do your draw? Based on what evidence? What would be needed to convince a broader audience?

Hints:

- An implementation for synthesizing the best training images might be based on the respective code parts for image inpainting (optimizing z) by Brandon Amos

(https://bamos.github.io/2016/08/09/deep-completion/, https://github.com/bamos/dcgan-completion.tensorflow)

- Improved training could be shown experimentally by quicker convergence (visually better synthesis after considerably less training *iterations*) or generally higher synthesis quality
- Training *time* in general might not be a good indicator for improvement as the synthesis process of each example per mini batch will be considerably slower.

Appendix A: Installing Docker (potentially outdated!)

This[4] works on Linux, OS X and Windows.

1. Install the Docker engine: See https://docs.docker.com/engine/installation/
2. Download a Docker image from "Docker hub": Start the Docker engine and execute within its shell
   ```
   docker pull IMAGE_NAME
   ```
   with `IMAGE_NAME` being for example:
   a. `floydhub/dl-docker:cpu`
      (CPU-only; includes e.g. TensorFlow, Caffe, Theano, Keras, Lasagne, Torch)
   b. `oduerr/tf_docker`
      (includes TensorFlow and some examples)
   c. `gcr.io/tensorflow/tensorflow:latest-gpu`
      (includes GPU support and TensorFlow)
3. Run the container (see appendix B for explanation of commands; the shared directory of the host machine has to be listed under „Preferences->File Sharing" in Docker):
   ```
   docker run -it -p 8888:8888 -p 6006:6006
              -v /Users/xxx/sharedfolder:/root/sharedfolder
              floydhub/dl-docker:cpu bash
   ```
4. Start the Jupyter notebook within the Docker container by executing the command `jupyter notebook` inside the Docker container.
5. Open a Webbrowser and navigate to http://127.0.0.1:8888/
6. Navigate to your shared directory and create an IPython notebook (or load one)

For more background on Docker, see also https://www.dataquest.io/blog/docker-data-science/.

[4] See also https://github.com/saiprashanths/dl-docker

Appendix B: A Docker cheat sheet (maybe outdated!)

Essential commands within the shell (on Linux / MacOS) or the Docker quickstart terminal (on Windows):

`docker` **`pull`** `IMAGE`

downloads the image from "Docker hub", given by the path in `IMAGE`
example: `docker pull oduerr/tf_docker`

`docker` **`run`** `IMAGE [COMMAND]`

starts the image as a container (optional: executes the `COMMAND` within)
`-p` sets a port mapping
`-d` runs the container as a background process ("detached")
`-it` keeps an input session open
`-v` specifies which directory on the local machine to store your notebooks in
example:
```
docker run -d -p 8888:8888
           -v LOCAL_FOLDER:/home/ds/notebooks
           dataquestio/python3-starter
```
example:
```
docker run -d -p 8080:8080 -p 8081:8081
           -it oduerr/tf_docker bash
```
→ the output string contains the `CONTAINER_ID` used later

`docker` **`cp`** `SRC DEST`

copies a local file into the running container
`SRC` and `DEST` may be local files/paths (given by a usual path) or
files within the container (prefixed by "`CONTAINER_ID:`")
example:
```
docker cp /home/vik/data.csv
          4greg24134:/home/ds/notebooks
```

`docker` **`exec`** `-it CONTAINER_ID COMMAND`

executes shell commands inside a running container
`-it` keeps an input session open
example: `docker exec -it 4greg24134 /bin/bash`

`docker` **`rm`** `-f CONTAINER_ID`

stops the container

`docker` **`ps`**

lists running containers to get their `CONTAINER_ID`s

`docker` **`commit`** `CONTAINE_ID IMAGE_NAME`

save a running container as a new image (persisting all the changes you made)

`IMAGE_NAME` must be lowercase (otherwise "invalid reference format" error)
example: `docker commit 30fc8b8d0d53 dcgan-playground`

Windows-specific commands within the standard Windows shell:

`docker-machine` **`ip`** `NAME`
returns the IP address of the Docker machine that runs containers as VMs
`NAME` usually is "`default`", and returns "`192.168.99.100`"
example: `docker-machine ip default`

Linux-specific commands within the standard linux shell to supervise the system during long-running jobs:

`htop -d 10`
CPU process viewer/manager, refresh every 1 seconds
`-d` delay in 10th of seconds

`nvidia-smi`
GPU process viewer