Deep Learning Blog - Data-driven discovery of coordinates and governing equations

Ana Maria Mekerishvili [5322065]
Boyan Kolev [5243416]
Jasper Geijsberts [5331587]
Nikolaus Oliver Ricker Chong [5274737]

April 14, 2024

1 Introduction

As part of the course CS4240 Deep Learning, we need to reproduce the results from a deep learning paper. Reproducing a paper is key for science. Papers are meant to gain insights, and potentially be used for further research and developments. If the paper is not reproducible, then it is not useful. In deep learning projects, one-to-one reproducibility is hard, as there are are random aspects that can be difficult to take care of, but similar results can usually be achieved following the same algorithm.

The paper we selected is "Data-driven discovery of coordinates and governing equations", by Kathleen Champion, Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton [2]. In this paper, the authors aim to create an autoencoder capable of finding near-optimal equations for representing non-linear measurable phenomena. A clear view of the architecture is presented in Figure 1.

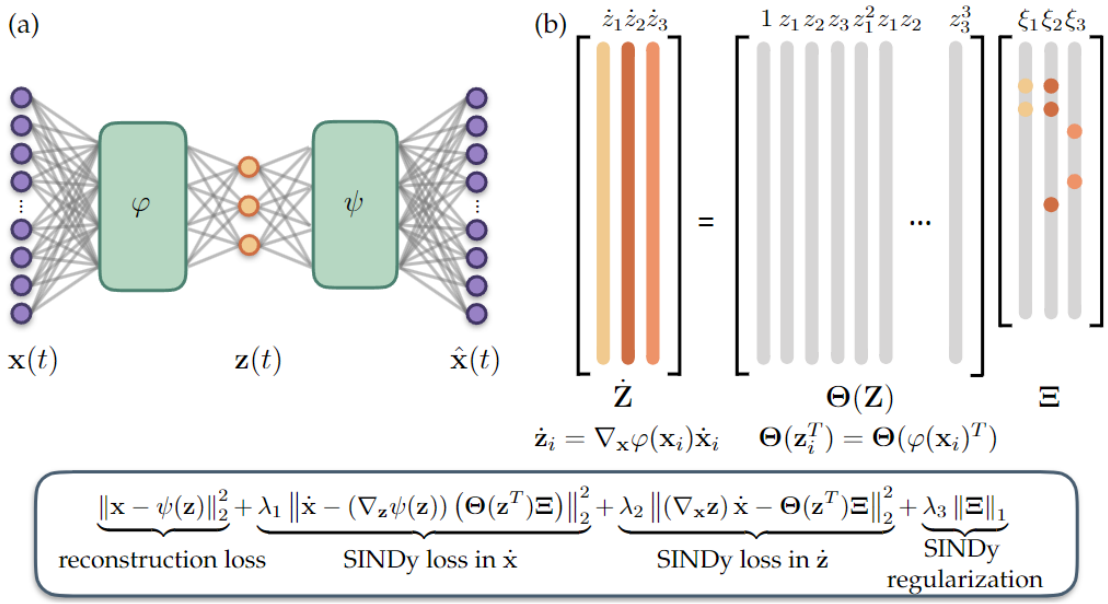


Figure 1: Sindy Autoencoder Architecture

The process consists of using sampled states of a measurable phenomenon as input $\mathbf{x}$, then perform a coordinate transformation ϕ to encode in a latent space. Using this latent space, we can utilize the SINDy algorithm[1] to predict the time derivative $\dot{\mathbf{z}}$ as a function of the nominal state of the latent space $\mathbf{z}$. It does this by creating a library of the latent space and approximating the coefficients as other learnable parameters of the network. Using the prediction for the derivative and the latent space, it is further decoded in ψ to create an approximation of the original sampled state. The figure also shows the loss function terms used. To ensure a good list of coefficients,

there is a mask filtering out irrelevant or insignificant coefficients.

The paper selected already has a written code in Tensorflow. The team decided that it would be a nice idea to reproduce the algorithm using Pytorch, which is another commonly used framework in deep learning applications. There was also an attempt to make the code faster, optimizing small parts and making it more readable (new code variant) than the code by the authors. Furthermore, we wanted to validate this using both dummy data (new data) and one of the examples from the paper. Finally, we were interested in the hyperparameters chosen and their effects on the loss (Hyperparameters checked). All this is discussed further in the next sections.

2 Running the code as is

The first part of the reproduction was to try and run the code on its own. For this, the team cloned the git repository, after which the code was run. However, at first it was unclear which code to run. There are multiple Python files (this can be seen in the tree of the original code, which is visible in subsection A.1). Three of them are located in `src`: `autoencoder.py`, `sindy_utils.py`, and `training.py`. These files turn out to contain the code necessary to run the training loop on different kinds of data. In the examples folder, there are the three examples, which are the three examples used in the paper. The example used for running the code in this reproduction was the **lorenz** example. When running the `train_lorenz.ipynb`, it was quickly found that Python modules were deprecated and the code could therefore not be run. After some searching, the necessary packages to run the code could be found. Then, a conda virtual environment can be created which includes the necessary packages. For this, we used the following code:

```
conda create -n tfv1 python=3.7
conda activate tfv1
pip install tensorflow-gpu==1.15
conda install protobuf=3.20 scipy matplotlib pandas
conda install -n tfv1 ipykernel --update-deps --force-reinstall
```

Using the new conda enviroment, the `train_lorenz.ipynb` can be run. When running, the team noticed a couple things:

1. The code gives many deprecation warnings, which do not stop the code from running, but they do giving worries on the team's end.
2. The amount of time it takes to finish 100 epochs is around 8 minutes.

Using this, it could be calculated that the time it takes to run the whole code would be more than 400 minutes, corresponding to between 6 and 7 hours. This fell in line with the timeframe of training given by the supervisor of our reproduction project.

3 Errors in the code

After we managed to run the code for the first time, it was time to understand what was happening behind it. This was not an easy task, as there were many relations between scripts. It can be seen in the subsection A.1. In order to understand the example `analyze_lorenz_model1.ipynb`, we had to read the `train_lorenz.ipynb`, `autoencoder.py`, `sindy_utils.py`, and `training.py`. Even though it was extremely useful for our own reproduction of the code, there were several sources of confusion, errors, and things that if improved could make the reading much easier.

3.1 Jupyter Notebooks

Even though Jupyter Notebooks are generally easier to understand, in this code they were not used to their full potential. There were no markdowns other than the (sub-)title, and no context was given on why it worked in certain ways. It could have nicely complemented the paper otherwise. Furthermore, the hyperparameters are selected here, but there is no explanation regarding them, neither on the paper nor on the code. It would be appreciated to see a grid search file, or some reasoning behind it.

3.2 Main Scripts

The main scripts are the autoencoder and the training files. These were written correctly, with a good structure. However, the documentation of the functions was incomplete or nonexistent. This made it much more difficult to review. We also found it inconsistent that there was a `sindy_utils.py` script, but most of the functions were already written in the autoencoder file, making it useless except for when plotting functions.

The final set of problems we encountered is that there is some code commented out without any explanation. This was distracting and left us unsure of its relevance. Finally, there was the following function:

```
    def linear_autoencoder(x, input_dim, d):
    # z,encoder_weights,encoder_biases = encoder(x, input_dim, latent_dim,
        [], None, 'encoder')
    # x_decode,decoder_weights,decoder_biases = decoder(z, input_dim,
        latent_dim, [], None, 'decoder')
    z,encoder_weights,encoder_biases = build_network_layers(x, input_dim,
        latent_dim, [], None, 'encoder')
    x_decode,decoder_weights,decoder_biases = build_network_layers(z,
        latent_dim, input_dim, [], None, 'decoder')

    return z, x_decode, encoder_weights, encoder_biases,decoder_weights,
        decoder_biases
```

This function has an error, since there is no `latent_dim`. It is an option to call it, and even though it is not used for these examples, it would break for other scenarios.

With all this in mind, we set ourselves to reproduce the paper in PyTorch.

4 Conversion to PyTorch and running in PyTorch

The main task we proposed ourselves was to reproduce the given paper/code in a more commonly used framework: PyTorch. Performing this would make it more easily readable, modifiable and not reduce the performance greatly if done correctly. Our approach to it consisted of taking the general structure set by the author in their own code and replicate it without "copy-paste". We sometimes had to look into the approach to make the code faster or for debugging. It was also part of our objectives to write it in a simpler and, hopefully, more effective way. We reproduced the autoencoder structure and the training loop.

4.1 Autoencoder

The autoencoder was created using OOP, rather than a set of functions. This made it more consistent, and had easier access to all variables. The autoencoder structure is the following:

```
class Autoencoder(torch.nn.Module):
    def __init__(self, params):
        # ...
        self.decoder = torch.nn.Sequential(*decoder_layers)

    def forward(self, state, mask=None):
        # ...
        return self.x_hat, self.dx_hat

    # Derivatives
    def compute_derivatives_x2z(self, x, dx, ddx=None):
        # ...
        return dz, ddz

    def compute_derivatives_z2x(self, z, dz, ddz=None):
        # ...
        return dx, ddx

    # Sindy Coefficients
    def sindy(self, mask, secondOrder=False):
```

```
        # ...
        return torch.matmul(Theta, self.sindy_coefficients*self.mask)

    # Loss
    def custom_loss_unrefined(self):
        # ..
        return loss

    def custom_loss_refined(self):
        # ...
        return loss
```

In general, it has a similar behaviour to the TensorFlow code, but it is structured in a much easier way. The key points for each function are listed below.

4.1.1 __init__

- The initializer takes the parameters dictionary and assigns them to several attributes. It also computes the library size.
- The SINDy coefficients are initialized based on the desired method and the library-latent size.
- If the sequential thresholding is active, then a mask is created, which can be updated as needed.
- The loss function is also saved as part of the autoencoder, and set by default to the version with regularization.
- The encoder and decoder are created separately using a Sequential object.

4.1.2 forward

- The forward function takes a state as input, which can be of first or second order, and stores it in memory until the next set of states replace it.
- It computes the latent space state (`z`) and the reconstructed state (`x_hat`), followed by computing the derivatives of the latent space using `compute_derivatives_x2z`, transforming the largest derivative to a prediction of the SINDY coefficients, and computes the reconstructed derivatives running new approximations through `compute_derivatives_z2x`.

4.1.3 compute_derivatives_x2z & compute_derivatives_z2x

This was the trickiest function to compute. The objective was to transform the time derivatives `dx` and `ddx` into the latent space. To do this, we made use of the formula $dz = \nabla_x \phi(x) \cdot dx$. There were two approaches to try, one was to compute the gradient of the encoder/decoder with respect to the states using `grad.autograd`, while the second option was to compute the derivatives of the layers, and run the time derivative through them. Both options worked, however, the former was significantly slower than the latter one. For simplicity, only the $dx \to dz$ for batch conversions are shown here:

```
    dz = torch.zeros_like(self.z)
    for i in range(x.shape[0]):
        x_temp = x[i].unsqueeze(0)
        x_temp.requires_grad = True
        z_temp = self.encoder(x_temp)
        # Compute gradient for each output element
        grads_list = []
        for j in range(z_temp.size(1)):
            grad_outputs = torch.zeros_like(z_temp)
            grad_outputs[:,j] = 1
            grads = torch.autograd.grad(outputs=z_temp, inputs=x_temp,
                grad_outputs=grad_outputs, create_graph=True, retain_graph=
                True)[0]
            grads_list.append(grads)
        first_grads = torch.cat(grads_list, dim=0)
        dz[i] = torch.matmul(first_grads, dx[i].unsqueeze(0).T).squeeze()
```

The main issue here was that the grad function would not always return what we expected. At first, it would return a combination of all samples (x) with all latent spaces (z), which made it extremely confusing and not useful, since we only needed to pair the corresponding values. Therefore, we had to use a for-loop to filter and append, which fixed the issue, but reduced the code speed drastically. Therefore, after discussing with another team, we tried the layers' derivative approach shown here:

```
def derivative(input, dx, layers, activation, act_func):
    dz = dx
    for i, layer in enumerate(layers):
        if i < len(layers) - 1:
            input = layer(input)
            if activation == 'elu':
                dz = torch.where(input < 0, torch.exp(input), torch.
                    ones_like(input)) * layer(dz)
            elif activation == 'relu':
                dz = (input > 0).float() * layer(dz)
            elif activation == 'sigmoid':
                dz = act_func(input) * (1 - torch.sigmoid(input)) *
                    layer(dz)
            input = act_func(input)
        else:
            dz = layer(dz)
    return dz

#########################################################
layers = list(self.encoder.children())
if self.params['model_order'] == 1:
    dz = derivative(x, dx, layers, self.activation, self.act_func)
```

This variation ran the code in a fraction of a second and allowed us to continue with the reproduction. A similar process occurred for the second derivatives and from z to x. If the system was of first order, the second derivatives were set to None.

4.1.4 sindy

A library is made using the transformed state vector and derivatives, followed by a coefficient mask constantly updated in the training loop. Finally, these are multiplied with the SINDy coefficients, and an approximation of the model is obtained in the latent space.

We struggled understanding the math behind this. We continuously struggled to figure out if the coefficients should change in time or not. We had them constant in our code, with the network changing around it, but then realised we had to set them as `nn.Parameter` to update them in time. This proved to be quite complex, as the SINDy parameters did not update. In the end it was just issues with converting the parameters to NumPy and then detaching them from the network when multiplying with the mask. It was fixed as follows:

```
### __init__ ###
if self.seq_thresholding:
    self.mask = torch.tensor(params['coefficient_mask'])
else:
    self.mask = None

### sindy ###
 if self.seq_thresholding:
    if mask is not None:
        self.mask = torch.tensor(mask)
    return torch.matmul(Theta, (self.sindy_coefficients * self.mask).
        float())
else:
    return torch.matmul(Theta, self.sindy_coefficients.float())
```

We also had a problem with the library creation here, as the SINDy coefficients were not of the same size. This was because the function given by the author did not include enough documentation, which made us think we had to give as input the number of sample states, but not the derivatives.

The libraries were just converted from the TensorFlow version, since they were more math focused, rather than AI.

4.1.5 custom_loss_unrefined & custom_loss_refined

- Two loss functions were created, and they would be set to the main function `loss_func()` for ease of access.
- The mean squared loss is computed for the state and main derivative, such that we have x, dx/ddx and dz/ddz compared to the reconstructed or predicted values.
- All losses are added with a predefined weight to a total scalar loss.
- The refined loss also includes a regularisation term around the SINDy coefficients.

4.2 Training Loop

For the training loop, a standard loop is used, with a small difference: the coefficient mask is updated every batch. This is a mask stating which coefficients shall be used in the model and training and is a key part of the Autoencoder which is used in both the TensorFlow and PyTorch code. The conversion of the training loop from TensorFlow to PyTorch was a relatively simple one. The biggest changes between the two loops are the way they are set up:

- **TensorFlow**: For the TensorFlow loop, all necessary parameters are initialized and set up, after which a TensorFlow Session is started. With this session, a loop can be initiated, where in each iteration the session can be run.

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(params['max_epochs']):
        for j in range(params['epoch_size']//params['batch_size']):
            batch_idxs = np.arange(j*params['batch_size'], (j+1)*
                params['batch_size'])
            train_dict = create_feed_dictionary(training_data, params,
                 idxs=batch_idxs)
            sess.run(train_op, feed_dict=train_dict)

        if params['print_progress'] and (i % params['print_frequency']
             == 0):
            validation_losses.append(print_progress(sess, i, loss,
                losses, train_dict, validation_dict, x_norm,
                sindy_predict_norm_x))

        if params['sequential_thresholding'] and (i % params['
            threshold_frequency'] == 0) and (i > 0):
            params['coefficient_mask'] = np.abs(sess.run(
                autoencoder_network['sindy_coefficients'])) > params['
                coefficient_threshold']
            validation_dict['coefficient_mask:0'] = params['
                coefficient_mask']
            print('THRESHOLDING:␣%d␣active␣coefficients' % np.sum(
                params['coefficient_mask']))
            sindy_model_terms.append(np.sum(params['coefficient_mask'
                ]))
```

- **PyTorch**: For the PyTorch loop, the network or model is first initialized and set up, after which the loop is immediately started. in the loop, the model will go through the following:

  1. A forward pass will be performed.
  2. The loss will be calculated.
  3. The gradients are set to zero
  4. The loss is propagated backward
  5. The optimizer step is performed

```
for i in tqdm(range(params['max_epochs']), desc='Epochs_Loop'):
    for j in (range(batch_iter)):
        batch_idxs = np.arange(j*params['batch_size'], (j+1)*params['
            batch_size'])
        train_dict = create_feed_dictionary(training_data, params, idxs=
            batch_idxs)
        x          = train_dict['x']
        dx         = train_dict['dx']

        model.forward([x, dx])
        loss = model.loss_func()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if params['print_progress'] and (i % params['print_frequency'] == 0):
        validation_losses.append(loss.detach().cpu().numpy())

    if params['sequential_thresholding'] and (i % params['
        threshold_frequency'] == 0) and (i > 0):
        model.mask = np.abs(model.sindy_coefficients) > params['
            coefficient_threshold']
        params['coefficient_mask'] = model.mask
        validation_dict['coefficient_mask:0'] = model.mask
        sindy_model_terms.append(np.sum(model.mask))

    final_losses.append(loss.detach().cpu().numpy())
```

The most important difference between the two loops is that for TensorFlow a Session needs to be started, while for PyTorch, the individual forward passes, loss calculations, etc. need to be performed in the loop explicitly. As the team was mostly comfortable and familiar with PyTorch, the loop structure for TensorFlow was hard to understand at first. This turned out to be one of the biggest struggles with transferring the code.

Other than the different training loops, the struggles the team had with transferring the code to PyTorch was the way to handle the Autoencoder as well as the PKL files. The errors from this were, however, through debugging and other methods, fixed over multiple days.

4.2.1 Getting to run the code on the GPU

After the PyTorch code could be run fully without any errors, the next step was to optimize the code for runtime. An optimization would help with allowing faster training speeds, which in turn would allow for more iterations and an easier hyperparameter tuning (which is done in section 6). The best way one can do this is by running complex and big calculations on the GPU of the computer. The GPU computes tensor and matrix operations more quickly than a CPU.

So, how do we get the code to run on the GPU?

To run the matrix and tensor operations, we need to implement CUDA. CUDA "is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs)."[1]. To implement CUDA in our Jupyter notebook, we basically need to tell our computer that a GPU is available, and then tell it to use it on certain data. Lets check if we have a GPU and then tell our computer to use it. Note that if you want to use CUDA, you will need an NVIDIA GPU.

```
os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"]="0" # GPU index
```

These two lines tell our computer the following. The first line states that the order of the GPU devices should be by their PCI bus ID, giving them a logical order. The second line state that the only visible device should be the device with ID "0", resulting in the fact that no other devices can be used on accident. We then do the following lines of code:

[1] https://developer.nvidia.com/cuda-zone

```
print("Available GPUs:", torch.cuda.device_count())
for i in range(torch.cuda.device_count()):
    print(f"GPU {i}: {torch.cuda.get_device_name(i)}")

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
torch.cuda.set_device(device)
```

This code shows a list of the GPUs that we have available, prints their name, and then sets the device that we use to be this GPU. If it is not available, it will use the CPU.

Next up, we put a random seed for initialization.

```
torch.manual_seed(seed=42)
torch.cuda.manual_seed(seed=42)
```

After implementing this code, we finally can use CUDA, but we are not immediately using it everywhere. In fact, we need to specify what exact operations and data we want to put on the GPU. we do this by passing the device we just defined: Whenever we want to put some data x on the GPU, we simply write the following:

```
    x.to(device)
```

Note that we do need to make sure that wherever we use this we pass the previously defined device and that we also do the calculations on only the GPU or CPU: we cant mix two data in an operation with one part of the data on the CPU and the other on the GPU.

We will now showcase how we put some data on the GPU. Because the training loop is the most demanding for computations, the GPU is best used here. Therefore, all the data that is passed to the training loop will be used on the GPU. However, it turns out we cannot just put the `training_data` and the other data to the GPU directly. Instead, we put the individual elements of the data on the GPU. For this we use a new function:

```
def move_tensors_to_device(data_dict, device):
"""
Move all tensors within a dictionary to the specified device.
"""
for key in data_dict:
    if isinstance(data_dict[key], torch.Tensor):
        data_dict[key] = data_dict[key].to(device)
return data_dict
```

With this new function, we can then transform the data we put into the training loop on the GPU:

```
def train_network(training_data, val_data, params, device: torch.device =
    device):
    training_data   = move_tensors_to_device(training_data ,device)
    val_data        = move_tensors_to_device(val_data, device)
    params          = move_tensors_to_device(params, device)
```

Like this, and by also putting other data, if necessary, on the GPU, we are able to run the training loop much more quickly.

4.2.2 GPU running problems

Although technically the training loop should be able to run on the GPU with these prerequisites, in reality this proved more difficult. When tried on one of our laptops that included a GPU, the code stated that it could recognize the GPU, and that this GPU was the active CUDA device. However, after putting all the data that goes into the model on the GPU, and training, it was found that the GPU was not used, when looking at the task manager. This strange occurrence was investigated by the team but could not be explained. Therefore, the team was not able to, locally, use the GPU available for the training loop.

What we discovered was that we tried to move things to device, but it was not done properly. Every operation should have GPU as device, and what we were doing was only moving the object itself. As a potential fix, we could go through each of the functions using tensors, and ensure they are done in the right device. This could be time-consuming but save time in the long run.

5 Results in PyTorch

Due to time constraints, the training of the models and generation of the results was performed only for the Lorenz example. However, the training procedure structure and function would remain identical for the other two examples, and in the autoencoder code, second derivatives have been included to account for second-order models.

5.1 Data Generation

It was decided that the structure of the data generation for the models training will be identical to the original paper [2], the detailed description of which can be found on pp 19-26 [2]. As such, data generation functions for the training data as well as those for the simulation data were directly taken from the source code provided by Champion et al. [2]. These functions include the ones in the *example_lorenz.py* file and the *sindy_simulate()* function, which propagates an initial state using the SINDy library and the provided coefficients (coming either from the trained model or from the testing data expected coefficients).

5.2 Running in Kaggle

To train the models using better resources than what a typical student has on their local machine, we recommend transferring the code to Kaggle. The process is relatively simple, especially if the code is created with modularity in mind, and consists of 4 steps:

1. Upload a main running script (from which the training is executed and results are saved) and all the supporting scripts which need to be set as utility scripts by opening them in an edit window and clicking File > Set as Utility Script. When setting the scripts, do not add the .py extension at the end, i.e. keep it file_name instead of file_name.py. Before proceeding to the next step, use the quick save option from the save menu on all the files such that version 1 appears instead of version 0 on the right. Note, use the quick save version to save time.

2. Any time a module/utility script (A) is used by another utility script (B), open (B) in an editing window, then click File > Add Input. Once the Add Input window opens, click Your Work > (A) > $\oplus$

3. Navigate to the main script, open it in an edit window and start a session. Search for an extensible window on the right-hand side for which in session options you may select an accelerator, Language=Python, Environment=Always Use Latest Environment.

4. Run the main script and download the results from the output folder '/kaggle/working'. Don't forget to refresh the output folder after each run to get the latest data. We recommend running in commit mode and closing the laptop or having the laptop connected to a power outlet in a never-sleep mode and dark screen while running all cells (no resources of the laptop are used except some battery power from the not sleep mode).

Using Kaggle, we were able to run the full model training using the same parameters and data sizes as in the ones used by Champion et al. [2], the training took between 8 and 9 hours.

5.3 Results

Before presenting the results, it is important to clarify that the training process is not guaranteed to produce a sparse coefficient matrix, but it is also possible for it to produce a sparse matrix which does mirror the true dynamics. Indeed, this is something Champion et al. are aware of and have communicated transparently in their work [2], "The learning procedure discovers a dynamical model by fitting coefficients that predict the continuous-time derivatives of the variables in a dynamical system. Thus, it is possible for the training procedure to discover a model with unstable dynamics or which is unable to predict the true dynamics through simulation. " Furthermore, this is the reason they trained 20 models with the same hyperparameters before arriving at a satisfactory result. Hence, with the limited time resources, we were only able to train 5 'full models' i.e. with the exact same model parameters as were provided in the authors' code:

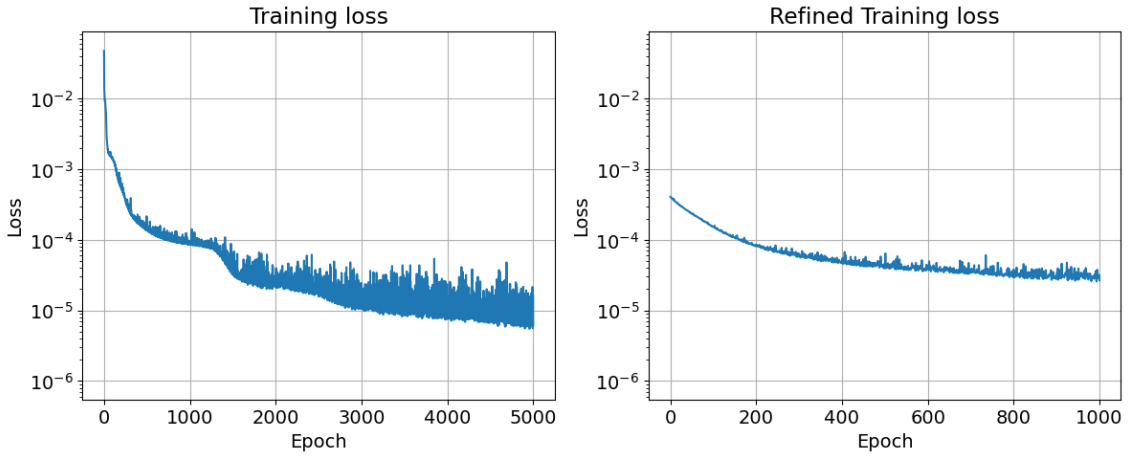
```
noise           = 1e-6
training_data   = get_lorenz_data(1024, noise)
validation_data = get_lorenz_data(20, noise)
```

```
params = {}

params['input_dim']      = 128
params['latent_dim']     = 3
params['model_order']    = 1
params['poly_order']     = 3
params['include_sine']   = False
params['library_dim']    = library_size(params['latent_dim'], params['
    poly_order'], params['include_sine'], True)

# sequential thresholding parameters
params['sequential_thresholding']     = True
params['coefficient_threshold']       = 0.2
params['threshold_frequency']         = 50
params['coefficient_mask']            = np.ones((params['library_dim'],
    params['latent_dim']))
params['coefficient_initialization']  = 'constant'

# loss function weighting
params['loss_weight_decoder']                        = 1.0
params['loss_weight_sindy_z']                        = 0.0
params['loss_weight_sindy_x']                        = 1e-4
params['loss_weight_sindy_regularization']           = 1e-5

params['activation']                                 = 'sigmoid'
params['widths']                                     = [64,32]

# training parameters
params['epoch_size']            = training_data['x'].shape[0]
params['batch_size']            = 1024
params['learning_rate']         = 1e-3

params['data_path']             = os.getcwd() + '/'
params['print_progress']        = True
params['print_frequency']       = 100

# training time cutoffs
params['max_epochs']            = 5001
params['refinement_epochs']     = 1001

n_experiments                   = 1
```

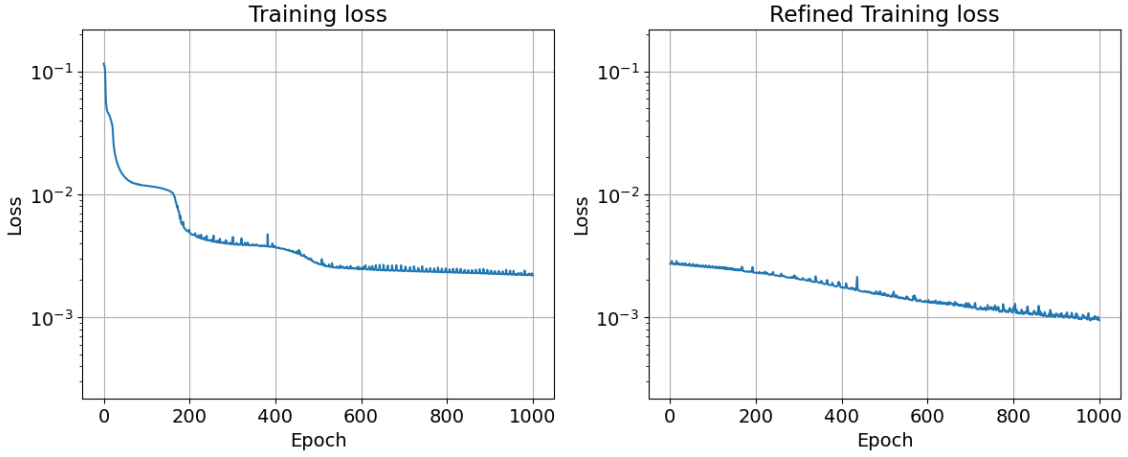
and two models with 1/10, 1/2 of the training data and 1001 epochs/refined epochs. From here on out for simplicity the 5 fully trained models will be referred to as $M_{F,i}$ with $i \in \{1, 2, 3, 4, 5\}$, the 1/10 model as M_{01} and the 1/2 as M_{05} The first results which were always checked to confirm the training had been achieved successfully from a purely mathematical standpoint was to check the evolution of the total loss both for the normal and refined epochs. The expectation is to see a decreasing trend and for the refined total loss to be smaller in magnitude, which is indeed the case. The results for this analysis can be found in Table 1 and representative plots of the evolution of the loss can be seen in Figure 2.

Table 1: Model total training losses at epochs of interest

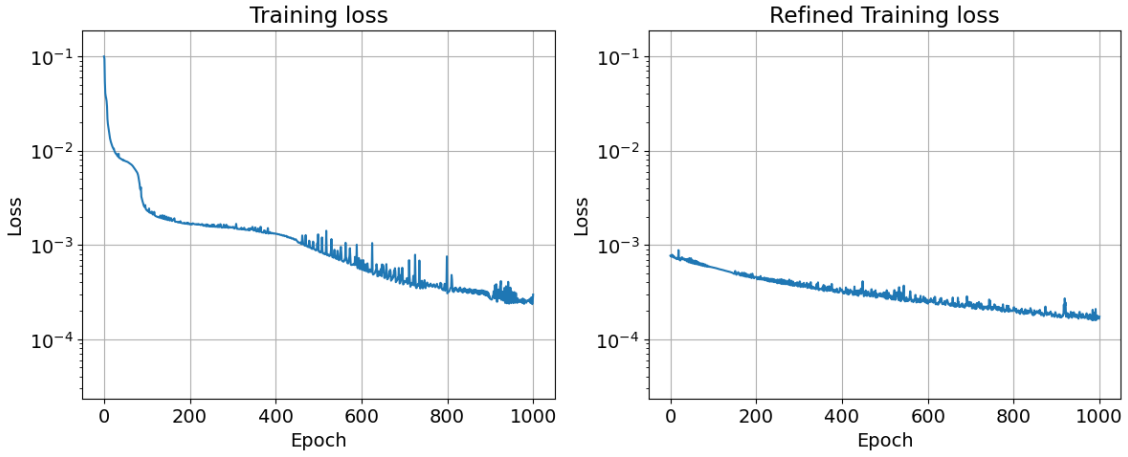
Model	**Loss last epoch**	**Training Loss epoch 1000**	**Refined training Loss epoch 1000**
Model	Loss last epoch	Training Loss epoch 1000	Refined training Loss epoch 1000
$M_{F,1}$	6.0268×10^{-6}	8.7214×10^{-5}	2.6654×10^{-5}
$M_{F,2}$	3.0077×10^{-6}	1.5238×10^{-5}	1.4563×10^{-5}
$M_{F,3}$	2.1779×10^{-5}	1.2895×10^{-4}	5.5268×10^{-5}
$M_{F,4}$	3.8146×10^{-6}	1.5277×10^{-5}	1.1620×10^{-5}
$M_{F,5}$	6.5085×10^{-6}	1.1393×10^{-5}	2.1174×10^{-5}
M_{01}	2.1896×10^{-3}	2.1896×10^{-3}	9.4836×10^{-4}
M_{05}	2.9770×10^{-4}	2.9770×10^{-4}	1.7357×10^{-4}



((a)) $M_{F,1}$



((b)) M_{01}



((c)) M_{05}

Figure 2: Total loss evolution over epoch for the different training

The next step after observing the models is to display the model's SINDy coefficient matrix multiplied by the coefficients mask and propagate the simulation based on some test data generated.

```
# Generate data
t                   = np.arange(0,20, 0.01)
z0                  = np.array([[-8, 7, 27]])
test_data           = generate_lorenz_data(z0, t, params['input_dim'],
    linear=False, normalization=np.array([1/40,1/40,1/40]))
test_data['x']      = test_data['x'].reshape((-1,params['input_dim']))
test_data['dx']     = test_data['dx'].reshape((-1,params['input_dim']))
test_data['z']      = test_data['z'].reshape((-1,params['latent_dim']))
```

```
test_data['dz']      = test_data['dz'].reshape((-1,params['latent_dim']))
```

```
sindy = params['sindy_coefficients'].detach().numpy()
params['sindy_coefficients'] = sindy
lorenz_sim = sindy_simulate(
    test_data['z'][0], t, test_data['sindy_coefficients'], params['
        poly_order'], params['include_sine']
)
z_sim = sindy_simulate(
    test_data['z'][0], t, sindy*params['coefficient_mask'], params['
        poly_order'], params['include_sine']
)
```

Ideally, one would want to obtain something similar to the coefficient matrix and simulation results displayed in Figure 3, which represent the ground truth conditions (GT). The coefficient matrix results are shown in **??**, from which it can be seen that Champion et al. were indeed correct in warning their readers that the training is not guaranteed to produce a sparse coefficient matrix or to obtain physically meaningful results. Furthermore, the comparison between observed and expected dynamics is shown in Figure 5

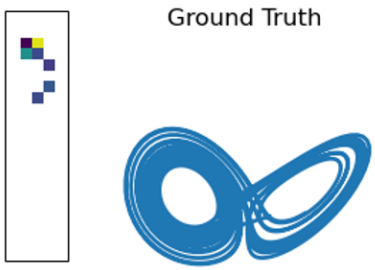


Figure 3: Ground Truth results (GT)

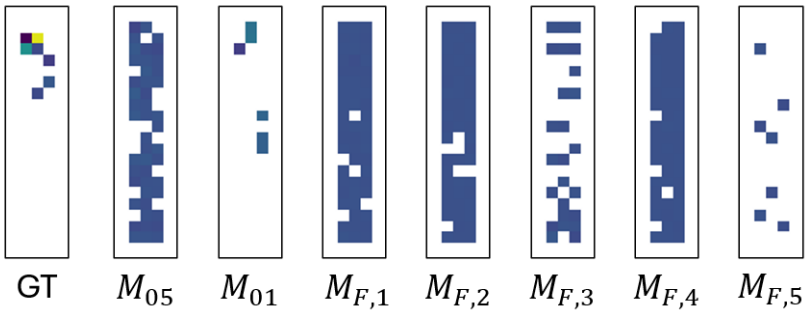


Figure 4: Comparison of coefficient matrix structure and sparsity for the different models

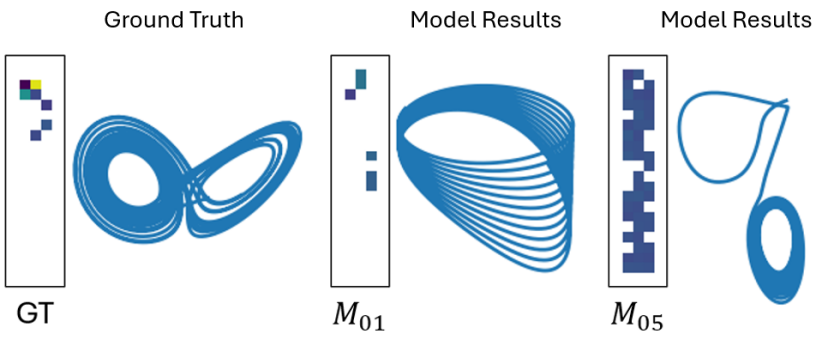


Figure 5: Comparison of propagated simulation and observed dynamics

It should be noted that the expected attractor dynamic behaviour was only observed for M_{05} and M_{01}. However, we do believe that the reproduction of the code was successful, but we were just not 'lucky' enough to obtain the 1 model which was sparse enough with appropriate coefficients such as the one out of the ten tries of Champion et al. [2]. Some interesting recommendations for future work would include analysing the training data generated in more depth by perhaps varying the noise level and distribution, as well as performing an in-depth hyperparameter tuning, which the next section of this blog will cover partially.

6 Hyperparameter Tuning

In this section, we will explain how we performed a grid search to see how the hyperparameters can be tuned. For the grid search we used the scikit-learn module, and specifically the `ParameterGrid` class. We also used the time module for capturing the amount of time spent training each training loop.

```
from sklearn.model_selection import ParameterGrid
from time import time
```

The grid search works as follows: one first states the hyperparameters one wants to change and tune, and one specifies multiple values for that hyperparameter. This is done for multiple hyperparameters after which all possibilities of combinations of these hyperparameters are checked. As an example we can use the batch size. In this example we can try to run the code with no batches, with 4 batches, and with 8 batches, while keeping all the other parameters the same. Then the experiments (which are the full training loops) can be dumped into a csv file, after which analysis can be performed on which hyperparameters are the most optimal. What must be noted, is that the amount of experiments needed to cover all different possibilities of hyperparameters increases very rapidly. If one is changing 5 hyperparameters which all have 3 different possibilities, the number of experiments becomes $3^5 = 243$. Therefore, with the limited training time, the most important hyperparameters should be chosen.

6.1 Parameter Analysis

To choose the most important hyperparameters, a decision had to be made by the team. The hyperparameters considered were the following:

- `latent_dim` (the latent dimension)
- `poly_order` (the polynomial order)
- `include_sine` (include a sine function or not)
- `loss_weight_decoder` (weight of decoder loss)
- `loss_weight_sindy_z` (weight of sindy z loss)
- `loss_weight_sindy_x` (weight of sindy x loss)
- `loss_weight_sindy_regularization` (weight of sindy regularization loss)
- `activation` (which activation function is used)
- `widths` (the amount of layers and how many neurons per layer)
- `batch_size` (the batch size)
- `learning_rate` (the learning rate)

However, after considering the amount of time it would take to perform all the possibilities, it became clear that the amount of hyperparameters had to be decreased. Afterward, the list was reduced to the following hyperparameters, where the values considered can also be seen.

- `'latent_dim': [1, 3, 6],`
- `'loss_weight_sindy_z': [0.0, 1e-4]`
- `'loss_weight_sindy_x': [0.0, 1e-4]`

- `'widths' : [[64],[64, 32],[64,32,16]]`
- `'batch_size': [128, 32]`

So why are these parameters chosen?

The fundamental thoughts behind the decision on certain hyperparameters over others is that the team wanted to investigate hyperparameters that are key to the SindyAutoencoder, and ones that generally can have a big impact.

Latent dimension Latent dimension is a key parameter in an Autoencoder, and therefore also in the SindyAutoencoder. Therefore, this parameter was deemed as very interesting to perform hyperparameter tuning on.

Poly order, or the polynomial order, determines to which polynomial order the training is performed. This will change the training very predictably, namely a higher order takes more time but will be more accurate.

Include sine determines if a sine is included in the analysis and training. The importance of this is known, therefore this was not deemed an important hyperparameter.

The weight of the decoder loss is an important hyperparameter, but since the loss depends on multiple values, it was decided to best keep this hyperparameter constant and then vary the other loss weights.

Loss weight Sindy z and x both were chosen as hyperparameters for the reasons given in the explanation of the previous hyperparameter. These losses are therefore chosen as hyperparameters. They directly influence the loss and are therefore interesting.

Loss weight Sindy regularization was not chosen as a hyperparameter because of its low influence, as well as the training time needed for the grid search.

Which **activation function** is used is an interesting hyperparameter, as deemed by the team, however, all activation functions introduce non-linearity and therefore they are all assumed to perform similarly.

The widths include the number of neurons per layer and the number of layers. This is of course an incredibly important hyperparameter of the Autoencoder, so this was taken as a hyperparameter to be varied.

The batch size speaks for itself: it adjusts how large the batch size is. Here it was seen as interesting to vary if batches are used or not and how this affects the results. Therefore, this parameter was chosen.

Lastly, the **learning rate** is a parameter that influences how quick the model learns. This parameter was not chosen due to the fact that it is not as interesting as Autoencoder specific hyperparameters and due to the fact the optimal learning rate varies depending on the size and accuracy of data that you are using. This was therefore not chosen.

6.2 Code Creation

The code creation, as mentioned, was done using the scikit learn `ParameterGrid` class. Using this, a parameter grid with all possibilities could be formed:

```
param_grid = {
    'latent_dim': [1, 3, 6],
    'loss_weight_sindy_z': [0.0, 1e-4],
    'loss_weight_sindy_x': [0.0, 1e-4],
    'widths' : [[64],[64, 32],[64,32,16]],
    'batch_size': [128, 32]
}
grid = ParameterGrid(param_grid)
```

Afterwards, a for loop over all the possibilities in this grid is performed:

```
for i, changed_params in enumerate(grid):
    params_run = params.copy()
    params_run.update(changed_params)
    params_run['library_dim'] = library_size(params_run['latent_dim'],
        params_run['poly_order'], params_run['include_sine'], True)
    params_run['coefficient_mask'] = np.ones((params_run['library_dim'],
        params_run['latent_dim']))

    print(f'EXPERIMENT:{i+1}')
    start = time()

    tf.reset_default_graph()
    results_dict = train_network(training_data, validation_data, params_run
        )
    end = time()
    dt = end-start

    df = pd.DataFrame()
    df = df.append({**results_dict, **params_run}, ignore_index=True)

    df2save = df[['loss_decoder','loss_sindy_x', 'loss_sindy_z', '
        loss_sindy_regularization','validation_losses']]
    changed_params['widths'] = str(changed_params['widths'])
    df_params = pd.DataFrame(changed_params, index=[0])

    df_final = pd.concat([df2save, df_params], axis=1)
    df_final['dt'] = dt

    dftocsv(df_final)

print("DONE")
```

This for loop takes the original parameters, copies them to `params_run`, after which the chosen hyperparameters are updated. The `library_dim` and `coefficient_mask` also need to be updated as they depend on the changed hyperparameters. Then the network is trained, after which the relevant information is put in a pandas DataFrame, which then is dumped into a CSV file, using the `dftocsv` function. This function can be seen in the next block of code:

```
def dftocsv(df):
    if os.path.isfile('gridsearch_results.csv'):
        df.to_csv('gridsearch_results.csv', mode='a', header=False)
    else:
        df.to_csv('gridsearch_results.csv', mode='w', header=True)
```

6.3 Results

In total, the amount of experiments needed (and thus possibilities of combinations of hyperparameters) turned out to be 72. To run this, it took multiple hours.

For the results, the combination of hyperparameters that had the lowest loss was the following:

Table 2: The best hyperparameters from the grid search

Validation loss	Batch size	Latent dimension	loss weight sindy x	loss weight sindy z	widths	dt [s]
0.00123	128	3	0	0	64	55.225

7 Task Division

Task	**Assigned Team Member**
Reproducing the autoencoder	Nikolaus, Ana
Hyperparameter Tuning	Jasper, Nikolaus
Training and results	Boyan, Jasper
Sindy library and simulation	Boyan, Jasper
Reporting	Boyan, Jasper, Nikolaus
Poster	Ana, Boyan, Jasper, Nikolaus

References

[1] Sparse identification of non-linear dynamics, Sep 2023.

[2] Kathleen Champion, Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*, 116(45):22445–22451, October 2019.

A Appendix A

A.1 Tensorflow Original Code Structure

```
C:.
|   LICENSE
|   README.md
|
|---examples
|   |---lorenz
|   |   |---analyze_lorenz_model1.ipynb
|   |   |---analyze_lorenz_model2.ipynb
|   |   |---checkpoint
|   |   |---example_lorenz.py
|   |   |---model1.data-00000-of-00001
|   |   |---model1.index
|   |   |---model1.meta
|   |   |---model1_params.pkl
|   |   |---model2.data-00000-of-00001
|   |   |---model2.index
|   |   |---model2.meta
|   |   |---model2_params.pkl
|   |   |---train_lorenz.ipynb
|   |
|   |---pendulum
|   |       ...
|   |
|   |---rd
|           ...
|
|---rd_solver
|   |---reaction_diffusion.m
|   |---reaction_diffusion_rhs.m
|
|---src
    |---autoencoder.py
    |---sindy_utils.py
    |---training.py
```