UNIVERSITY OF CAMBRIDGE

**King Fai Yeh**
**葉景輝**

# Analysing Neural Network Training with Koopman Operator Theory

Downing College

A dissertation presented on May 13, 2022
for Part II, Computer Science Tripos

## Declaration of originality

I, King Fai Yeh of Downing College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I am content for my dissertation to be made available to the students and staff of the University.

Signed: *King Fai Yeh*

Date: *May 13, 2022*


## Acknowledgements

I would like to thank Prof. Pietro Liò and Paris Flood for originating and supervising the project, and Prof. Robert Harle for his guidance and support for my studies throughout the last three years.

I would also like to offer my heartfelt gratitude to Ingrid and my family for their unconditional love and support.

# Proforma

| | |
|---|---|
| **Candidate number** | 2336F |
| **Project title** | Analysing Neural Network Training with Koopman Operator Theory |
| **Examination** | Computer Science Tripos - Part II, 2022 |
| **Word count** | 11538[1] |
| **Line count** | 3020[2] |
| **Project originators** | Prof. Pietro Liò, Paris Flood |
| **Project supervisors** | Prof. Pietro Liò, Paris Flood |

## Original aims of the project

The project aims to leverage Koopman Operator Theory to represent the training trajectories of neural networks with a linear framework, a new application of the theory. The core part brings over a deep learning approach from Lusch *et al.* (2018) and evaluates the model on the trajectories of different neural architectures. The extension is to improve the existing approach or design a new approach with better performance.

## Work completed

All core and extension criteria have been achieved. A deep autoencoder is implemented and tested against various baselines and datasets for the core part. On the extension, a novel architecture, `KoopmaNNet`, is proposed. `KoopmaNNet` is shown to be outperforming current approaches in the literature in linearising neural network training trajectories, including the core approach. The proposed architecture has demonstrated the possibility of representing training trajectories in a smaller linear space while extracting dominant features that preserve the loss dynamics during training for non-trivial neural architectures and are robust across different architectural variations and training set-ups.

[1] The word count was computed by Overleaf `texcount`.
[2] The line count was computed using the command `find . -name '*.py' | sed 's/.*/"&"/' | xargs wc -l`.

## Special difficulties

None.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

The success of neural networks (NN) in machine learning has brought us a new set of challenges. Although NNs can capture many complex relations in data, the inner working of NNs remains impenetrable to humans. It is not exactly understood how NNs "learn" nor what they have learnt. The lack of information has created considerable impedance in:

1. *Pruning the structure of NNs*

   The over-parametrisation of NNs is one of the primary reasons for the lack of understanding. Bigger models take longer to operate, but it is not straightforward to recognise the importance of each weight and bias and remove the unnecessary ones. It is, therefore, an active field of research to prune the structure of NNs while maintaining their performance.

2. *Controlling the learning trajectory of NNs*

   Much effort has been devoted to controlling how NNs learn, such as regularisation and normalisation. However, the exact impact of these techniques is not fully understood due to the complex non-linearity involved. The process of hyperparameter tuning has also been more of trial-and-error. The control of learning trajectories could help improve the convergence and generlisation properties of NNs.

3. *Interpreting the relations captured by NNs*

   Even though NNs can capture many complex relations, there has not been any general methodology for interpreting and reasoning these relations. It prevents the application of NNs to many safety-critical use cases, while also resulting in a lack of informative feedback on the assumptions and design decisions made when building the models.

In this dissertation, the training process of NNs will be analysed. In particular, this project aims to leverage a mathematical theory named *Koopman Operator Theory* (KOT) to project the training trajectories of the weights and biases (*"training dynamics"*), which are non-linear, into a linear space.

Before diving into the details in the next chapter, it is essential to bring in the perspective of viewing NN training as a form of *discrete-time dynamical system*. Informally, a dynamical system is a system that evolves in time. It comprises a *state space* holding the set of possible states and a *dynamical map* that advances a state to its next state in time. At any given time, a dynamical system is specified by a current state, chosen from its state space. A list of future states can then be obtained by iteratively applying the dynamical map to the current state, obtaining a collection of states that is referred to as a *trajectory* of the system.

The training of NNs can be regarded as a transition of states, where each state is specified as the NN's weights and biases. The state space would be the real space with a dimension equal to its weights and biases. The dynamical map would then be the backpropagation algorithm that "trains" a NN to the next state. The training trajectory of a NN would be a list of states that the NN goes through during training. This perspective fits the process of NN training into a dynamical system, which is a prerequisite for analysing with Koopman Operator Theory.

Notice that the dynamical map in NN training is non-linear due to the activation functions used in NNs. With KOT, it would be possible to transform a non-linear dynamical system into a linear one ("*linearise*"). Linear dynamical systems are helpful since they are intrinsically simpler and better studied than their non-linear counterparts. There have been established optimal estimation and control methodologies for linear dynamics. If the training of NNs can be modelled with linear dynamics, the problems discussed at the start of the section could be drastically simplified.

## 1.2 Contributions

The ease of analysing linear dynamics has attracted some attempts to linearise non-linear dynamics, most of which are data-driven. However, previous approaches either require an in-depth theoretical analysis of the dynamical system or do not perform well when this is not the case. The lack of success of conventional approaches has driven the field towards deep learning, which is well-suited for approximating arbitrarily complex functions. Lusch *et al.* (ibid.) has taken the first step in using deep autoencoders to linearise physical dynamical systems.

The main contributions of this dissertation are as follows:

1. The core part brings over a deep learning approach from Lusch *et al.* (ibid.), initially developed for physical systems, to linearise the training dynamics of NNs. It will be shown that the vanilla autoencoder in the paper can indeed find a linear representation for the training trajectories of simpler NNs.
2. The extension part proposes `KoopmaNNet`, a novel architecture based on graph neural networks. It will be demonstrated that `KoopmaNNet` can cope with more complex NN architectures, outperforming state-of-the-art approaches in Lusch *et al.* (ibid.) and Dogra and Redman (2020) in both accuracy and versatility. `KoopmaNNet` has also shown the possibility of projecting the non-linear training

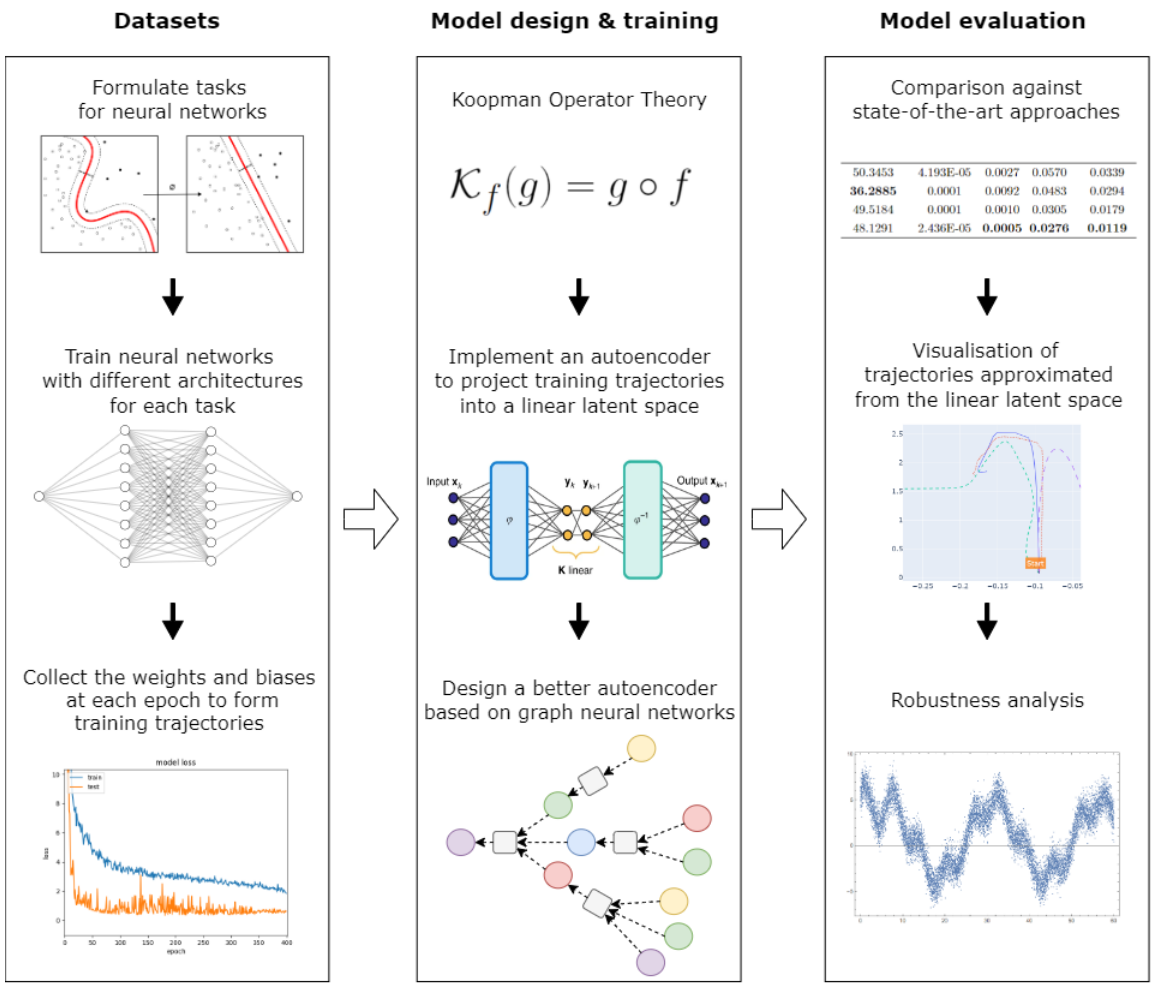



Figure 1.1: A summary of the approach taken in this project for projecting neural network training trajectories into a linear space.

dynamics into a low-dimensional linear space, opening up a new direction for network pruning.

A summary of the approach taken in this project is outlined in Figure 1.1.

## 1.3 Overview of the dissertation

The dissertation begins in chapter 2 with an overview of the theories required to understand this project, an introduction of popular Koopman-analysis approaches in the field, and a description of the project's requirements. Chapter 3 gives a more exact formulation of the problem, introduces the datasets involved, and details the design and implementation of the Lusch approach and the novel `KoopmaNNet`. Chapter 4 evaluates the two models and compares them with selected baselines. Chapter 5 concludes the findings of this project and discusses possible future directions of the work.

# Chapter 2

# Preparation

*This chapter introduces the concepts supporting the work of this project, including (2.1) Koopman Operator Theory, (2.2) autoencoder, and (2.3) graph neural network. Then in section 2.4, several data-driven Koopman analysis methodologies are discussed, including previous KOT applications to NN training trajectories. The remaining sections of the chapter describe the project's starting point, analyse the project requirements, and summarise the software engineering approaches taken in the project.*

## 2.1 Koopman Operator Theory (KOT)

Following the description in section 1.1, this section introduces Koopman Operator Theory, a theory that enables non-linear dynamical systems to be represented linearly. The theories in this section provide essential theoretical support for the deep learning approaches explored in this dissertation.

### 2.1.1 Definitions

A neural network (NN), with fixed topology, can be specified by its set of weights/biases $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^n$, which we will later refer to it as the *state* of the NN. As the NN is trained through each time step, the state is updated with respect to the training function $f : \mathcal{X} \to \mathcal{X}$, i.e. $f(\mathbf{x}_k) = \mathbf{x}_{k+1}$. In the notion of dynamical systems, $\mathcal{X}$ is the state space, and $f$ is the dynamical map. The evolution of states throughout the training process is referred to as the *training trajectory* (or *training dynamics*) of the NN. However, for most NNs, $f$ is composed of numerous non-linear relations, making further analyses hard.

Define $\mathcal{G}$ as a class of *measurement functions* for the training dynamics of NNs, such that each member $g : \mathcal{X} \to \mathbb{C}$ maps a NN state to a complex number (*measurement*). In the context of NNs, the measurement functions here are often the identity function since the NN states are always accessible. However, the quantity measured does not necessarily need to carry an intuitive meaning to humans. The reason for this setup will soon be explained.

### 2.1.2 Koopman operator

NN training has recently been discovered to be intimately connected with KOT, as described in Dogra (2020). It turns out that KOT might provide a possible path for linearising non-linear training dynamics.

The Koopman operator $\mathcal{K}_f$ for a function $f$ is defined by:

$$\mathcal{K}_f(\phi) := \phi \circ f$$

which takes in a function $\phi$ and compose it with $f$.

Recall $f$ as the training function of the NN. Define $\mathcal{K}_f : \mathcal{G} \to \mathcal{G}$ as the Koopman operator for $f$, which takes in a measurement function $g$ and compose it with the training function $f$, i.e. $\mathcal{K}_f(g) := g \circ f$. Intuitively, since $g$ produces a measurement of the current state, $\mathcal{K}_f(g)$ produces a measurement of the next state in time:

$$\mathcal{K}_f(g)(\mathbf{x}_k) = g \circ f(\mathbf{x}_k) = g(\mathbf{x}_{k+1})$$

Therefore, ${\mathcal{K}_f}^t(g)$ takes in a state and produces a measurement of the state $t$ time steps after.

The Koopman operator is linear, since:

$$\begin{aligned}\mathcal{K}_f(\alpha g_1 + \beta g_2)(\mathbf{x}) &= (\alpha g_1 + \beta g_2) \circ f(\mathbf{x}) \\ &= \alpha g_1 \circ f(\mathbf{x}) + \beta g_2 \circ f(\mathbf{x}) \\ &= \alpha \mathcal{K}_f(g_1)(\mathbf{x}) + \beta \mathcal{K}_f(g_2)(\mathbf{x})\end{aligned}$$

However, $\mathcal{K}_f$ is infinite-dimensional since infinite degrees of freedom are needed to characterise the entire set of measurement functions. Thus, it is not straightforward to apply $\mathcal{K}_f$, and a finite-dimensional approximation would be required to put KOT into practice.

### 2.1.3 Koopman-invariant subspace

Consider a (possibly infinite) set of measurement functions $\{g_i\}$. It spans a Koopman-invariant subspace if and only if all functions $g$ in this subspace remain within the subspace after being acted on by the Koopman operator $\mathcal{K}_f$. We call this set of measurement functions a *Koopman-invariant set*. For any function $g$ that can be represented as a linear combination of $\{g_i\}$, ${\mathcal{K}_f}^t(g)$ can also be. This is a useful property.

Consider a finite vector of measurement functions $\mathbf{g}$ that span a Koopman-invariant subspace $\hat{\mathcal{G}}$:

$$\mathbf{g} = \begin{bmatrix} g_1 & g_2 & \dots & g_m \end{bmatrix}^\intercal$$

Note that $\mathbf{g}$ can also be viewed as a function $\mathbf{g} : \mathcal{X} \to \mathbb{C}^m$.

Let's extend the Koopman operator $\mathcal{K}_f$ to act of a vector of measurement functions as

well:

$$\mathcal{K}_f(\mathbf{g}) = \begin{bmatrix} \mathcal{K}_f(g_1) & \mathcal{K}_f(g_2) & \dots & \mathcal{K}_f(g_m) \end{bmatrix}^\intercal$$

As $\hat{\mathcal{G}}$ is Koopman-invariant, $g_i \in \hat{\mathcal{G}}$ implies $\mathcal{K}_f(g_i) \in \hat{\mathcal{G}}$, which means $\mathcal{K}_f(g_i)$ can be represented as a linear combination of $\mathbf{g}$. Hence, when the measurement functions are constrained to the subspace $\hat{\mathcal{G}}$, $\mathcal{K}_f$ can be represented by a finite-dimensional matrix $\mathbf{K}$, such that:

$$\mathcal{K}_f(\mathbf{g}) = \mathbf{K}\mathbf{g}$$

Therefore, by restricting the domain and co-domain of $\mathcal{K}_f$ from $\mathcal{G}$ to $\hat{\mathcal{G}}$, we can get around with the infinite-dimensional property of the Koopman operator and use a finite matrix representation instead.

How is this useful in the context of NN training? If we can find such a finite vector of measurement functions $\mathbf{g}$ spanning a Koopman-invariant subspace $\hat{\mathcal{G}}$, we can apply $\mathbf{g}$ to transform NN states $\mathbf{x}$ into latent variables $\mathbf{y} \in \mathcal{Y} \subseteq \mathbb{C}^m$. This is beneficial since:

$$\begin{aligned} \mathbf{g}(\mathbf{x}_{k+t}) &= \mathbf{g} \circ f^t(\mathbf{x}_k) \\ &= ({\mathcal{K}_f}^t(\mathbf{g}))(\mathbf{x}_k) \\ &= (\mathbf{K}^t\mathbf{g})(\mathbf{x}_k) \\ &= \mathbf{K}^t(\mathbf{g}(\mathbf{x}_k)) \end{aligned}$$

Rearranging the terms, we have:

$$\mathbf{x}_{k+m} = \mathbf{g}^{-1}(\mathbf{K}^t(\mathbf{g}(\mathbf{x}_k)))$$

Therefore, if we can find such $\mathbf{g} : \mathcal{X} \to \mathcal{Y}$ that can preserve the features in $\mathcal{X}$, NN states in $\mathcal{X}$ can be represented with latent variables in $\mathcal{Y}$. As the latent coordinates $\mathbf{g}(\mathbf{x}_k)$ can be advanced linearly with $\mathbf{K}$, this linear framework would come in handy for predicting/approximating the trajectories of training dynamics. Eventually, $\mathbf{g}^{-1}$ could reconstruct the NN states from the latent coordinates in $\mathcal{Y}$.

However, finding a set of measurement functions spanning a Koopman-invariant subspace is not always straightforward. Many traditional approaches usually suffer from the lack of guarantee that the generated set is Koopman-invariant (see S. L. Brunton, B. W. Brunton, *et al.* (2015)). This brings us to the eigenfunction approach in the following subsection.

### 2.1.4 Koopman eigenfunction

A Koopman eigenfunction $\varphi$ is a special type of measurement function $g : \mathcal{X} \to \mathbb{C}$ that advances linearly upon being acted on by the Koopman operator $\mathcal{K}_f$:

$$\mathcal{K}_f(\varphi)(\mathbf{x}_k) = \lambda \cdot \varphi(\mathbf{x}_k)$$

Which implies:

$$\varphi(\mathbf{x}_{k+1}) = \lambda \cdot \varphi(\mathbf{x}_k)$$

Since any set of Koopman eigenfunctions must span a Koopman-invariant subspace, we can construct a finite vector of eigenfunctions $\boldsymbol{\varphi} : \mathcal{X} \to \mathcal{Y}$:

$$\boldsymbol{\varphi} = \begin{bmatrix} \varphi_1 & \varphi_2 & \dots & \varphi_p \end{bmatrix}^\intercal$$

and use $\boldsymbol{\varphi}$ in place of $\mathbf{g}$ as described in the subsection above. The corresponding linear advancement matrix $\mathbf{K}$ would be diagonal, filled with its eigenvalues.

Why would the measurements have to be complex? Consider if the measurement functions $g \in \mathcal{G}$ only map to a real measurement space; the corresponding $\mathbf{K}$ for any Koopman-invariant function set would also be a real matrix. By restricting ourselves to using eigenfunction sets only, $\mathbf{K}$ would become a real diagonal matrix.

Notice that *periodic orbits*, which repeat themselves as the system evolves, are common in dynamical systems. Thus, for the eigenfunction approach to be valid, it must be able to express periodicity well. Since $\boldsymbol{\varphi}$ is non-linear, it is theoretically possible to represent periodic orbits with a real diagonal $\mathbf{K}$. However, this means that the burden of handling periodicity would rest on $\boldsymbol{\varphi}$ and $\boldsymbol{\varphi}^{-1}$. If we would like to approximate $\boldsymbol{\varphi}$ and $\boldsymbol{\varphi}^{-1}$ with deep learning, it would be unwise to rely on non-periodic activation functions to account for periodic events. If the measurements are complex, the handling of periodicity can then be easily delegated to $\mathbf{K}$, leaving less for the deep models to learn. Therefore, operating in a complex space would help models learn better by utilising a better-suited component to capture periodicity.

Of course, if we are to estimate $\mathbf{K}$ for any Koopman-invariant function set not necessarily being eigenfunctions, $\mathbf{K}$ would not have to be diagonal. In that case, we can use a real measurement space without compromising its expressiveness of latent dynamics.

To conclude this section, we have the following equation:

$$\mathbf{x}_{k+t} = \boldsymbol{\varphi}^{-1}(\mathbf{K}^t(\boldsymbol{\varphi}(\mathbf{x}_k)))$$

Breaking it down, we have:

$$\mathbf{y}_k = \boldsymbol{\varphi}(\mathbf{x}_k) \qquad \mathbf{y}_{k+t} = \mathbf{K}^t(\mathbf{y}_k) \qquad \mathbf{x}_{k+t} = \boldsymbol{\varphi}^{-1}(\mathbf{y}_{k+t})$$

where in the context of NN training:

- $\mathbf{x}_k$ and $\mathbf{x}_{k+t}$ are NN states (in $\mathcal{X}$) at time step $k$ and $k+t$
- $\mathbf{y}_k$ and $\mathbf{y}_{k+t}$ are their corresponding latent representation in $\mathcal{Y}$
- $\mathbf{K}$ is a diagonal matrix

The challenge now is to get a good set of eigenfunctions $\boldsymbol{\varphi}$ whose span preserves most features in the original subspace $\mathcal{X}$.

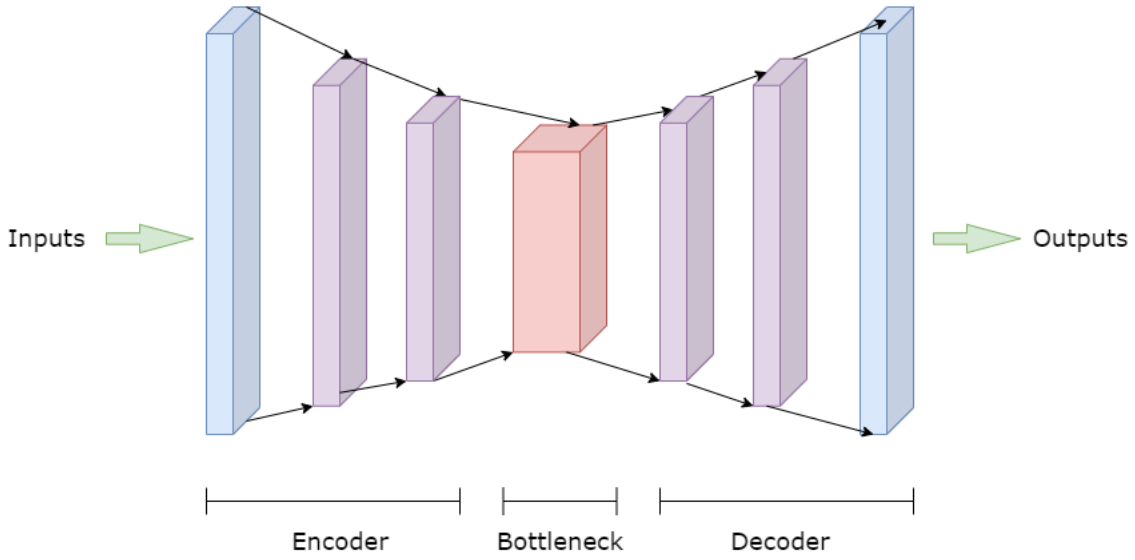



Figure 2.1: Figure showing the structure of a feedforward autoencoder.

## 2.2 Autoencoder

Autoencoders are a class of self-supervised neural networks trained for finding an informative representation of the data such that the representation is helpful for other applications. In particular, an autoencoder encodes the inputs into a latent representation and decodes the representation back into the original inputs:

$$\mathbf{y} = \varphi(\mathbf{x})$$
$$\tilde{\mathbf{x}} = \varphi^{-1}(\mathbf{y})$$

Formally, the *encoder* $\varphi : \mathbb{R}^n \to \mathbb{R}^m$ maps the inputs $\mathbf{x} \in \mathbb{R}^n$ into latent coordinates $\mathbf{y} \in \mathbb{R}^m$, whereas the *decoder* $\varphi^{-1} : \mathbb{R}^m \to \mathbb{R}^n$ attempts to reconstruct the inputs $\tilde{\mathbf{x}}$ from $\mathbf{y}$. Autoencoders are usually trained to minimise the differences between the original inputs and the reconstructed inputs:

$$\underset{\varphi,\varphi^{-1}}{\operatorname{argmin}} < x, \varphi^{-1}(\varphi(\mathbf{x})) >$$

with $< . >$ being a designated loss function. Figure 2.1 shows an example of a feedforward autoencoder.

## 2.3 Graph neural network (GNN)

Graph neural networks are a class of neural networks taking graph structures as inputs. Before the emergence of GNNs, most deep learning architectures could not exploit the graph structures in data. Feedforward MLPs flatten graphs into linear arrays, dropping their intrinsic structures. CNNs are designed for data modelled as grids in the Euclidean space, which is a particular case of graphs in a broader sense, making them unsuitable for generic graphs. As a graph shall not be treated differently even if its nodes and

edges are permuted, neural networks for graphs shall have a *permutation-invariant* architecture. We shall start with the concept of *message passing*, which is centric on the permutation invariance of GNNs today.

### 2.3.1 Message passing

Message passing, also known as *neighbourhood aggregation*, generalise the convolutional operator for regular grids in CNN to irregular neighbourhoods in GNN. In each iteration, a *message* is generated on each edge using its features and the features of its two endpoint nodes. Then for each node, all messages on its incoming edges are aggregated. The aggregated message is then used to update the features of the node. In essence, message passing aggregates the information of each node's neighbourhood and updates the node. The aggregator must be permutation-invariant, ensuring the permutation invariance for the entire GNN.

Formally, denote a graph by $G = (V, E)$, with $n = |V|$ and $e = |E|$. Denote the features of a node $u$ by $\mathbf{x}_u \in \mathbb{R}^c$, an edge $u \to v$ by $\mathbf{e}_{u,v} \in \mathbb{R}^d$.

At iteration $t$ of message passing, each node feature $\mathbf{x}_u^{(t)}$ is updated with information aggregated from its neighbours $\mathcal{N}(u) = \{v \mid v \to u \in E\}$:

$$\mathbf{x}_u^{(t+1)} = \gamma(\mathbf{x}_u^{(t)}, \square_{v \in \mathcal{N}(u)}\ \phi(\mathbf{x}_u^{(t)}, \mathbf{x}_v^{(t)}, \mathbf{e}_{v,u}))$$

For each node $u$, the steps are:

1. For each edge $v \to u$, $\phi$ generates a message $\phi(\mathbf{x}_u^{(t)}, \mathbf{x}_v^{(t)}, \mathbf{e}_{v,u})$ using the node features of its endpoints $\mathbf{x}_u^{(t)}$ and $\mathbf{x}_v^{(t)}$ and the features of the edge $\mathbf{e}_{v,u}$.
2. $\square$ aggregates the messages for $u$. The aggregator must be permutation-invariant, e.g., sum, average, max, min.
3. $\gamma$ uses the aggregated message to update the node features $\mathbf{x}_u^{(t)}$ to $\mathbf{x}_u^{(t+1)}$.

Therefore, after $t$ iterations of message passing, the features of any node $u$ will represent the features of its radius-$k$ neighbourhood. Figure 2.2 provides an example of how message passing works in a graph.

### 2.3.2 Graph convolutional layers

Following the introduction of message passing in the last subsection, this section will introduce the graph convolutional layers/operators relevant to this project. They are used within different architectural variations of `KoopmaNNet`.

**Graph convolutional network (GCN)** GCN is a classical GNN architecture proposed in Kipf *et al.* (2016). In GCN, messages for each node are generated by linearly transforming the features of its incoming neighbours. The average of these messages then becomes the new features of the node.

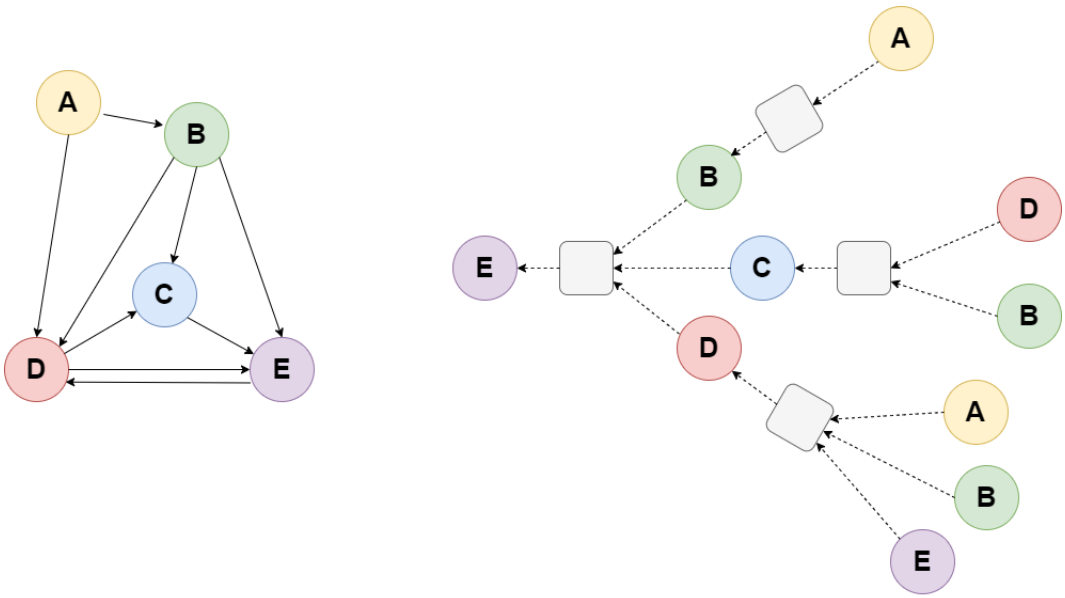



Figure 2.2: Figure showing the process of message passing. The graph on the left is an example of an input graph $G$, while the figure on the right shows two iterations of message passing, with E being the node in focus.

**Graph attention network (GAT)** GAT, proposed in Veličković *et al.* (2017), generalises GCN by imposing weights on messages. In other words, GAT takes note of the relative importance of the neighbours by a weighted aggregation. The weights are computed by a self-attention mechanism, depending on the projection magnitude of each message to a trainable attention vector specific to the current node. The new features of the node are given by a weighted sum of its features and its neighbours' features.

**Graph transformer operator** Shi *et al.* (2020) takes a further step in weighted message passing of GAT by bringing over the attention mechanism and residual connections from classic transformers in Vaswani *et al.* (2017). In particular, to calculate the attention weights for a node, a *query* vector is generated for the node, whereas *key* and *value* vectors are generated for each neighbour. The relative weights of the neighbours depend on the projection magnitudes of their key vectors on the node's query vector. The new features of the node are then created by summing its linearly-transformed features and the weighted aggregation of its neighbours' value vectors.

**Interaction network (IN)** Interaction networks in Battaglia *et al.* (2016) are an early proposal of the message passing paradigm. Being aware that the paradigm is agnostic to the choice of message passing functions, the paper uses MLPs for both message generation and node update while aggregating the messages with a summation.

## 2.4 Data-driven Koopman analysis

Putting KOT into practice, data-driven Koopman analysis has long been applied to real-world dynamical systems to obtain a globally linear representation of their dynamics. This section will introduce a few common approaches for identifying linearised dynamics

from non-linear systems. Some of these approaches inspire the methodologies developed in this project, while others are used as evaluation baselines.

### 2.4.1 Dynamic-mode decomposition (DMD)

Dynamic-mode decomposition[1], in the context of Koopman Operator theory[2], is an algorithm that identifies a best-fit linear approximation to the Koopman operator of a dynamical system. Consider two matrices $\mathbf{X}$ and $\mathbf{X}'$:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k \end{bmatrix} \qquad \mathbf{X}' = \begin{bmatrix} \mathbf{x}_2, \mathbf{x}_3, \dots, \mathbf{x}_{k+1} \end{bmatrix}$$

DMD find a best-fit matrix $\mathbf{K}$ such that:

$$\mathbf{X}' \approx \mathbf{K}\mathbf{X}$$

In other words, it treats the measurement function $\mathbf{g}$ in section 2.1.3 as the identity function $\mathbf{I} : \mathcal{X} \to \mathcal{X}$. DMD utilises the proper orthogonal decomposition (POD) via the computationally-efficient singular-value decomposition (SVD) to perform dimensionality reduction on $\mathbf{K}$. With SVD/POD, DMD finds the dominant eigenvalues and eigenvectors of $\mathbf{K}$ without computing the matrix directly, allowing its spectral decomposition to be approximated. The spectral decomposition can be used to advance $\mathbf{x}_i$ without involving expensive matrix operations via $\mathbf{K}$.

**Extended DMD (eDMD)** Extended DMD[3] augments the identity measurements in DMD by non-linear measurements before performing a linear regression on them. To put it simply, instead of approximating $\mathbf{K}$ in $\mathbf{X}' \approx \mathbf{K}\mathbf{X}$, eDMD approximates the $\mathbf{K}$ in:

$$\mathbf{g}(\mathbf{X}') = \mathbf{K}\mathbf{g}(\mathbf{X})$$

where:

$$\mathbf{g}(\begin{bmatrix} \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k \end{bmatrix}) \approx \begin{bmatrix} \mathbf{g}(\mathbf{x}_1), \mathbf{g}(\mathbf{x}_2), \dots, \mathbf{g}(\mathbf{x}_k) \end{bmatrix}$$

which corresponds to the concept outlined in section 2.1.3. The *SINDy regression*[4], also known as "Sparse identification of non-linear dynamical systems", identifies a sparse matrix $\mathbf{K}$ from a library of non-linear candidate functions $\mathbf{g}$ to avoid overfitting.

### 2.4.2 Deep learning

Traditional algorithms of finding a finite approximation of the Koopman operator suffer from several problems. Theoretical frameworks like DMD, using a best-fit linear matrix to advance the dynamics, are not expressive enough to represent more complex non-linear systems. Extended DMD attempts to improve the expressiveness of the framework by incorporating non-linear measurement functions, yet the set of non-linear measurements is not exhaustive, and they are no longer guaranteed to be Koopman-invariant. With deep neural networks being universal function approximators, some

[1] Tu *et al.* (2013)
[2] Rowley *et al.* (2009)
[3] Williams *et al.* (2014)
[4] S. L. Brunton, Proctor, *et al.* (2015)

effort has been devoted to approximating the measurement functions with deep learning.

Recall in section 2.1.3, we had the following equation:

$$\mathbf{x}_{k+t} = \mathbf{g}^{-1}(\mathbf{K}^t(\mathbf{g}(\mathbf{x}_k)))$$

Our aim is to find a good approximation to $\mathbf{g}$, $\mathbf{K}$ and $\mathbf{g}^{-1}$. The task of identifying an appropriate set of measurement functions $\mathbf{g}$ and using them to transform and reconstruct the NN states fits naturally with the structure and use case of autoencoders.

Think of $g : \mathcal{X} \to \mathcal{Y}$ as a function transforming the non-linear dynamics into their latent-space embeddings. The advancement of embeddings in $\mathcal{Y}$ is the latent representation of the original dynamics. Applying $\mathbf{K}$ repeatedly to the latent-space embeddings produces a linear trajectory, from which the original dynamics can be reconstructed. Note that a bidirectional mapping must be constructed between the original dynamics and its embeddings, since for the embeddings to qualify as the "latent representation", it must be shown possible to recover training dynamics from them. These requirements resonate with the functioning of autoencoders.

A few attempts have been made to use deep neural networks to analyse physical nonlinear dynamical systems. Here are two of the attempts more relevant to this dissertation:

- Lusch *et al.* (2018) adopted the Koopman eigenfunction approach as described in section 2.1.4. The paper employs an autoencoder to approximate the eigenfunctions and the transition matrix $\mathbf{K}$, using feedforward MLPs for the encoder and decoder and a diagonal matrix for $\mathbf{K}$. By restricting $\mathbf{K}$ to be diagonal, $\mathbf{g}$ naturally becomes eigenfunctions $\boldsymbol{\varphi}$ of $\mathcal{X}$.

  The paper also proposed an auxiliary network to handle systems with continuous spectra. In continuous spectra, dynamics are characterised by a continuous range of frequencies instead of discrete spectra with isolated and fixed frequencies. An example of a system with continuous spectra would be the classical pendulum:

  $$\ddot{x} = -\sin(\omega x)$$

  The pendulum frequency can be any number within the continuous spectra $(0, \omega)$. An infinite Fourier sum has to be used to express the shift in frequency within a linear framework, defying low-dimensional Koopman approximations. The auxiliary network, in the form of an MLP, accounts for the varying frequency by freshly computing a new pair of conjugate eigenvalues $\lambda_\pm$ for each point in the trajectory. The linear operator $\mathbf{K}(\lambda_\pm)$ is then constructed from perfect sines and cosines with their frequency derived from the eigenvalues. The auxiliary network could hence avoid a high-dimensional approximation for the harmonic series.

- Li *et al.* (2019) uses the Koopman-invariant subspace approach in section 2.1.3. The paper proposes to use *compositional Koopman operators* to prune the search space of $\mathbf{K}$. Specifically, by encoding the world state into object-centric embeddings of dimension $k$ and categorising the objects' relations into $h$ types, the relations between any two objects can be modelled as a smaller matrix $\tilde{\mathbf{K}}_i \in \mathbb{R}^{k \times k}$, with $i \in \{1, \ldots, h\}$ denoting the type of the relation. With this approach, the overall

Koopman matrix $\mathbf{K}$ can be formulated as a block matrix composed by $\tilde{\mathbf{K}}_i$, reducing the number of parameters in $\mathbf{K}$ from $n^2$ to $hk^2$. This approach is also applicable to physical systems with a variable number of objects.

The paper uses a GNN-based autoencoder to compute the embeddings. $\mathbf{K}$ is identified by a linear regression instead of training it along with the graph autoencoder.

### 2.4.3 Application of Koopman analysis to neural networks

There have not been many attempts to apply the Koopman Operator Theory to analyse neural network training trajectories. One notable application would be Dogra and Redman (2020), which proposed to partition a NN architecturally into smaller partitions for more efficient analysis. In the paper, NNs are split into perceptrons, where each partition corresponds to a perceptron's incoming weights and biases. The paper aims to identify a Koopman-invariant subspace approximating the trajectory of the partition, with the measurement function $\mathbf{g}$ being the identity function $\mathbf{I} : \mathcal{X} \to \mathcal{X}$, similar to DMD in section 2.4.1. When the training of NN has reached near regions of local minima, a best-fit linear matrix $\mathbf{K}$ is used to capture the underlying trajectory trends of the partition ($\mathbf{x}_{k+1} \approx \mathbf{K}\mathbf{x}_k$). The approach has resulted in faster training and convergence than traditional backpropagation with a comparable loss performance.

Tano *et al.* (2020) proposed a similar approach to accelerate NN training by alternating between backpropagation and DMD. In each iteration, a few backpropagation steps are performed, after which the training trajectory of each NN layer is fed into the cheaper DMD to advance for a few more steps. The iterations are repeated until convergence.

## 2.5 Starting point

To the best of my knowledge, there have not been any documented attempts to adopt deep neural networks to linearise the training trajectories of NNs. This project begins with reproducing the vanilla autoencoder proposed in Lusch *et al.* (2018) and applying it to the training dynamics of neural networks. Although there is a public repository for the implementation, the code is not directly adaptable as physical systems and neural network dynamics are significantly different. Therefore, the project was started entirely from scratch.

Before the project, I had only a trivial understanding of linear algebra given by the Part IA Mathematics course for the Natural Sciences Tripos. The Part IB Artificial Intelligence course provided a brief theoretical overview of neural networks and back-propagation regarding deep learning. The Part IB group project was my only hands-on deep learning experience, in which I utilised Convolutional Neural Networks (CNN) for computer vision. Besides these, I had no other experience relevant to this project. Thus, I had to spend the first few weeks familiarising myself with the mathematical theories in this project and general deep learning approaches.

## 2.6 Requirement analysis

Upon getting a more thorough understanding of the relevant theories and approaches for this project, the core success criteria were distilled into a list of workflow specifications:

1. Generate suitable datasets covering a variety of tasks and NN architectures. At least regression and classification should be included. The datasets should also span across architectures on different scales.
2. Implement a pipeline for training (and testing) models on different datasets.
3. Implement the autoencoder proposed in Lusch *et al.* (2018) and adapt the architecture for this problem.
4. Select and implement a set of common baselines for comparison. They should include both numerical and deep learning baselines.
5. Interpret the results and evaluate the model's performance in predicting the training trajectories of NNs.

Since the project's objective is to bring over an existing approach to a new problem on which there has not been any published work, the last two points are essential in showing the ability and limits of the approach.

The extension part of the project is to improve the existing approach or even design a new approach with better performance. As the extension requires innovating ideas without similar approaches for the same problem as a reference, it is more of a research-flavoured part and, therefore, subject to higher risks.

The above workflow can be further modularised into deliverables, with their priorities and difficulties evaluated. The deliverables are listed in Table 2.1, and their dependencies are plotted in Figure 2.3.

| **Deliverable** | **Project** | **Priority** | **Difficulty** |
|---|---|---|---|
| Trajectory datasets | Core | **Must have** | **Easy** |
| Model training pipeline | Core | **Must have** | **Medium** |
| Lusch autoencoder | Core | **Must have** | **Easy** |
| Baseline models | Core | **Should have** | **Medium** |
| Model evaluation framework | Core | **Must have** | **Medium** |
| Trajectory visualisation framework | Core | **Should have** | **Easy** |
| Better prediction model | Extension | **Could have** | **Hard** |

Table 2.1: A list of deliverables of this project with their priorities and difficulties.

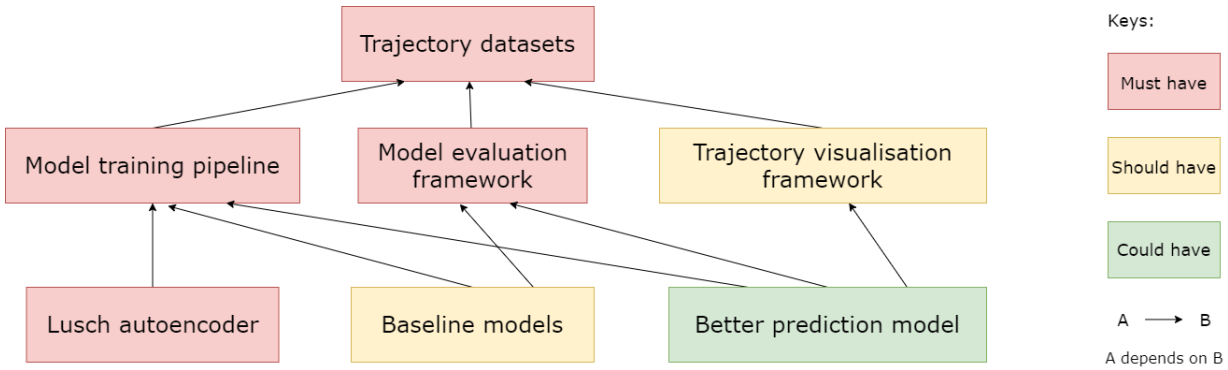



Figure 2.3: The dependency relations between different deliverables in the project.

## 2.7 Software engineering

### 2.7.1 Methodology

As the main focus of this project lies in deep learning, experiments and feedback are centric on the direction of model development. For this reason, the agile model was used instead of the traditional waterfall model to keep the development process iterative and incremental.

With the dependency graph above, it is possible to identify the components in the project that can be worked on independently. Before the components were implemented, a brief sketch of APIs was drawn from the components' relations. With a brief plan, large-scale structural modifications could be avoided later down the workflow, which unnecessarily lengthens development time and potentially gives rise to compatibility issues. As such, the development process could be streamlined.

For training, datasets were pickled at the time of generation for fast retrieval and neat management. For testing, unit tests were written for some necessary modules, whereas integration tests were executed every time the modules are modified. During development, object-oriented programming (OOP) was utilised with automatic formatting for the ease of logical comprehension and maintenance.

### 2.7.2 Languages and frameworks

For this project, I chose Python to implement the models and generate the datasets for two reasons. Firstly, the intrinsic design of Python, together with the variety of highly-optimised data manipulation and deep learning frameworks, can vastly simplify implementations. Secondly, as Python is utilised in most machine learning projects, more community support is available during development, and my code can potentially be reused more conveniently.

For implementing the deep neural networks, PyTorch was chosen over TensorFlow as the dynamic computational graphs in PyTorch offer more flexibility and convenience than the static ones in TensorFlow. With PyTorch, the input size does not have to be specified beforehand, while in TensorFlow, it is required. As the hyperparameters are

often changed during development, more flexibility would come in handy. Also, PyTorch simplifies debugging. In TensorFlow, the static computational graphs separate the location of where an error is made and where it is observed, which complicates bug tracking. In PyTorch, resources are created dynamically under the dynamic computational graphs, so bugs can be observed right at where they are made.

Table 2.2 lists out the libraries used with the reasons for using them:

| **Library** | **Citation** | **Usage** | **License** |
|---|---|---|---|
| PyTorch | Paszke *et al.* (2019) | Deep learning | BSD License |
| PyTorch Geometric | Fey *et al.* (2019) | Graph neural networks (GNN) | MIT License |
| NumPy | Harris *et al.* (2020) | Vectorised operations | BSD License |
| Pandas | McKinney *et al.* (2010) | Data manipulation | BSD License |
| Scikit-learn | Pedregosa *et al.* (2011) | Machine learning | BSD License |
| SciPy | Virtanen *et al.* (2020) | Optimisation | BSD License |
| Plotly | Inc. (2015) | Data visualisation | MIT License |
| Project Jupyter | Kluyver *et al.* (2016) | Interactive computing | BSD License |
| tqdm | Costa-Luis (2019) | Progress visualisation | MIT License |
| PySINDy | Silva *et al.* (2020) | Library for extended DMD | MIT License |

Table 2.2: A list of libraries used in this project with their usage and license.

### 2.7.3 Development environment

I used Visual Studio Code as the IDE for this project, mainly for the rich set of extensions available on the platform. The dissertation was written in LaTeX with Overleaf, an online cloud-based editor.

The project was developed on my desktop computer (Intel i7-9700K 4.9GHz with 32GB RAM, running Windows 10). The models were trained on 4 GPUs (NVIDIA Titan Xp) from the Computational Biology group, generously offered by Prof. Pietro Liò.

### 2.7.4 Redundancy and backup

Redundancy is crucial in minimising disruptions caused by unlikely but catastrophic events. For version control of the source code, `git` was used with the repository hosted on *GitHub*. For redundancy, the source code and datasets are automatically synchronised with my network-attached storage (NAS) at home and my laptop disk in real-time.

I would like to note that the SSD of my computer died in mid-April, just as I was writing my dissertation and benchmarking my models. All project data in the disk were inaccessible. The redundancy measures above were successful in helping me to seamlessly switch to my laptop and continue with my project without disruptions.

# Chapter 3

# Implementation

*This chapter begins with a more specific formulation of the problem, followed by a description of the datasets used in this project. The remaining sections detail the design and implementation of the Lusch autoencoder and the proposed* `KoopmaNNet`.

## 3.1 Problem formulation

As discussed in section 1.1, the project aims to learn a set of measurement functions that transforms the non-linear training trajectories of neural networks into a linear representation. In line with Lusch *et al.* (2018), this project adopts the eigenfunction approach. As outlined in section 2.1.4, the equation for the transformation is:

$$\mathbf{x}_{k+t} = \boldsymbol{\varphi}^{-1}(\mathbf{K}^t(\boldsymbol{\varphi}(\mathbf{x}_k)))$$

From which $\boldsymbol{\varphi}$, $\mathbf{K}$ and $\boldsymbol{\varphi}^{-1}$ have to be estimated.

The problem specification above can be framed as a trajectory prediction task. Consider the NN state $\mathbf{x}_k$. The objective is to compute the future states $\mathbf{x}_{k+t}$ as the NN is trained. There are two routes of doing it:

1. Using the non-linear backpropagation to train the NN directly:

$$\mathbf{x}_{k+t} = f^t(\mathbf{x}_k)$$

2. Transforming the NN state into a latent representation, advancing the latent variables with a linear operator, and reconstructing the NN state from the latent space:

$$\mathbf{x}_{k+t} = \boldsymbol{\varphi}^{-1}(\mathbf{K}^t(\boldsymbol{\varphi}(\mathbf{x}_k)))$$

Suppose we treat the trajectory produced from the first route as the ground truth and the second route as an approximation. In that case, we can formulate the task as finding an approximation for $\boldsymbol{\varphi}$, $\mathbf{K}$ and $\boldsymbol{\varphi}^{-1}$ such that the difference between the two trajectories is minimised.

As in Lusch *et al.* (2018), the dynamics prediction models are trained on three types of loss functions:

1. **Reconstruction loss**

$$\mathcal{L}_{recon} = || \mathbf{x} - \boldsymbol{\varphi}^{-1}(\boldsymbol{\varphi}(\mathbf{x})) ||$$

The reconstruction loss encourages the model to find a set of latent-space coordinates representing each point in the training dynamics. In formal terms, the loss fits $\boldsymbol{\varphi}$ and $\boldsymbol{\varphi}^{-1}$ such that there exists a set of $\{\mathbf{y}_k\}$ satisfying $\mathbf{y}_k = \boldsymbol{\varphi}(\mathbf{x}_k)$ and $\mathbf{x}_k = \boldsymbol{\varphi}^{-1}(\mathbf{y}_k)$. This ensures that the training dynamics must be recoverable from its latent representation.

2. **Linear prediction loss**

$$\mathcal{L}_{linear} = || \boldsymbol{\varphi}(\mathbf{x}_{k+t}) - \mathbf{K}^t(\boldsymbol{\varphi}(\mathbf{x}_k)) ||$$

The linear prediction loss enforces the structure of the latent space by ensuring that the latent representations advance linearly with $\mathbf{K}$, i.e. $\mathbf{y}_{k+t} = \mathbf{K}^t(\mathbf{y}_k)$.

3. **Dynamics prediction loss**

$$\mathcal{L}_{pred} = || \mathbf{x}_{k+t} - \boldsymbol{\varphi}^{-1}(\mathbf{K}^t(\boldsymbol{\varphi}(\mathbf{x}_k))) ||$$

The dynamics prediction loss ensures that future states can be predicted by advancing the latent representation of the current state using the equation $\mathbf{x}_{k+t} = \boldsymbol{\varphi}^{-1}(\mathbf{K}^t(\boldsymbol{\varphi}(\mathbf{x}_k)))$. In other words, it enforces a bidirectional mapping between the linear latent dynamics and the non-linear training dynamics.

where $|| \cdot ||$ is a symbol for the mean-squared error, averaged over time steps and the number of samples.

The overall loss function on which the models are trained on will be a linear combination of the three:

$$\mathcal{L} = \alpha_1 \cdot \mathcal{L}_{recon} + \alpha_2 \cdot \mathcal{L}_{linear} + \alpha_3 \cdot \mathcal{L}_{pred}$$

and with $L_2$-regularisation.

**Models** For this task, two autoencoders have been implemented for training dynamics prediction. The first autoencoder is implemented based on the architecture proposed in Lusch *et al.* (ibid.), with some adaptations to fit into the context of NN training. The second autoencoder is a novel approach based on graph neural networks (GNN). They will be discussed in more detail in the following subsections.

## 3.2 Datasets

Different tasks are formulated to study the dynamics prediction models' ability and limitations, and neural networks of various depths, equipped with different types of non-linearity, are trained to tackle the tasks. Datasets are created by training the NNs with different initial conditions for a fixed number of epochs. After each epoch, weights, biases, and training losses are collected to form the training trajectories.

This is a list of the tasks studied in this project:

**Linear regression** The linear regression problem is based on a synthetic dataset generated using the formula:

$$y_i = \mathbf{w} \cdot \mathbf{x}_i + b + \epsilon_i$$

with $\epsilon_i$ sampled from a Gaussian distribution. There are 1000 samples, each with 20 features. A **1-layer** feedforward NN is trained for this problem.

**Classification** The task is based on the classic wine dataset, which are the results of a chemical analysis of wines grown in the same region in Italy but derived from 3 different cultivars[1]. The task is to classify each wine into 1 of the 3 classes using the quantities of its 13 constituents. The dataset consists of 178 samples.

Two feedforward NNs are trained for this problem:

- **1-layer** Linear → Softmax
- **2-layer** Linear → Activation → Linear → Softmax

**Image classification** The task is based on the digits dataset, which includes 1800 samples of handwritten digits in 8x8 bitmaps[2]. The objective is to classify each bitmap as 1 of the 10 digits.

This task aims to study the models' performance in predicting the dynamics of convolutional neural networks (CNN). In order to reduce memory usage, only images from 2 of the classes are included so that the CNN can be slightly scaled down.

Define a *convolutional layer* as "Convolution → Activation → Average Pooling". Two CNNs are trained for this problem:

- **2-layer CNN** Convolutional layer → Linear
- **3-layer CNN** Convolutional layer → Convolutional layer → Linear

**DE solver** NN Differential equation (DE) solvers have emerged as a new class of unsupervised approaches tackling complex systems of ordinary/partial differential equations. They have shown to be performing significantly better than traditional numerical integrators in approximating the functional solutions.

In this project, a DE solver will be trained to solve for $f(x)$, where:

$$f'(x) - f(x) = y$$

with $f(1) = 0$ as the initial conditions.

The NN DE solver for this project is composed of 3 layers, each separated by a nonlinear activation layer. The inputs are uniformly sampled over a range on which the loss is evaluated.

[1] *UCI Machine Learning Repository: Wine data set* (1991)
[2] *UCI Machine Learning Repository: Optical recognition of handwritten digits data set* (1998)

The tasks are selected to include both synthetic and real-world problems. Input normalisation (i.e., standardising each dimension to a mean of 0 and variance of 1) is applied to some task datasets to make the weights and biases evolve within a narrower range in training.

The initial weights and biases of the NNs are sampled from $(-0.1, 0.1)$. The NNs are then trained with standard gradient descent (non-stochastic) so that the training process corresponds precisely to the mathematical framework outlined in the last chapter. To avoid complex dynamics (e.g., involving momentum), the SGD (Stochastic Gradient Descent) optimiser is used in the study. In chapter 4.3, some of these configurations are altered to analyse the models' robustness to variations. All NNs are trained for 400 epochs. Each trajectory dataset comprises 10k training trajectories, with a train/validation/test split of 8k/1k/1k.

However, training the dynamics prediction models with 400-epoch trajectories takes far more time than it is possible to keep the project iterative. To keep the training time within a reasonable limit, the weights and biases are sampled per 10 epochs, shrinking the length of the trajectories to 40. This augmentation would have a minimal effect on the predictions, as the objective of the predictions is to capture the overall trends instead of transient fluctuations.

## 3.3 Vanilla autoencoder

### 3.3.1 Architecture

The architecture proposed in Lusch *et al.* (2018) uses MLPs for both its encoder and decoder. The architecture of the encoder is identical to the decoder. Non-linear activations are placed between neighboring layers in the encoder and decoder. Their output layers are linear. The autoencoder is then trained on the loss function detailed in the last subsection. The architecture of the autoencoder is shown in Figure 3.1.

Note that the dimension $m$ of the latent space is a hyperparameter of the model.

### 3.3.2 Adapatation for this project

The autoencoder in Lusch *et al.* (ibid.) was designed for low-dimensional physical systems with well-studied relations between quantities. Thus, the paper's methodology and architectural hyperparameters cannot be directly adopted for this problem.

**Nature of the dynamics** In the paper, the autoencoder was trained on three continuous-time dynamical systems, two of which are in the continuous spectra. The systems have at most three dimensions and are governed by relatively more straightforward physical equations. The dynamics are captured by snapshots separated by a fixed time interval.

In contrast, the training process of a NN is a discrete-time dynamical system, so snapshots of the weights and biases after each training epoch form its trajectory. The

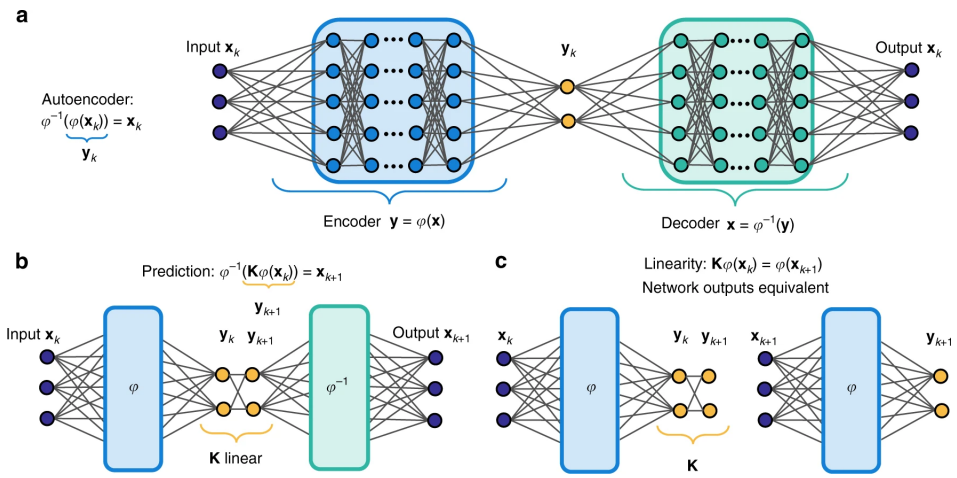



Figure 3.1: Diagram showing the architecture of the Lusch autoencoder, taken from Lusch *et al.* (2018). (a) The autoencoder is trained on reconstruction loss to identify $\boldsymbol{\varphi}$ and $\boldsymbol{\varphi}^{-1}$ which transforms between NN states and their latent representations. (b) The matrix $\mathbf{K}$ is used to linearly advance the latent coordinates, with the autoencoder trained on dynamics prediction loss to predict future NN states via the latent dynamics. (c) Linear prediction loss is used to identify the linear dynamics in the latent space.

training trajectory of a NN depends on many factors that affect backpropagation, such as training data, the types of non-linearity imposed, and the current state of the NN. An exact system of equations can always be derived to describe the training dynamics. However, the expressions would comprise of (1) numerous terms, since the gradients are averaged over the entire set of training data, and (2) "deep" terms, as the gradients stack up as they are backpropagated into earlier layers. Also, most NNs have many more weights and biases than the number of quantities in the physical systems above, resulting in training trajectories with dimensions magnitudes higher. Thus, the non-linear relations to be captured in this project are significantly more complex than the physical systems studied in the paper.

**Continuous spectra** Recall that in the paper, an auxiliary network was proposed to handle systems with continuous spectra (details in section 2.4.2). However, neither the loss functions, the activations, or the SGD optimiser has a continuous spectrum of frequencies. Therefore, the auxiliary network will not be needed for this problem.

**System-specific optimisations** One interesting observation is that the autoencoder in the paper was trained with a vital constraint derived from the specific physical properties of the systems. To make this possible, the relations in the dynamical systems must have been well studied and interpreted. Similar optimisations can hardly be replicated for this task due to the complexity of interpreting NN training dynamics.

### 3.3.3 Implementation

**Complex eigenfunctions** In the paper, real numbers are used in the autoencoder. To represent complex eigenvalues with real numbers, the Jordan block $\mathbf{B}_{\mu,\omega}$ was used:

$$\mathbf{B}_{\mu,\omega} = \exp(\mu) \begin{bmatrix} \cos(\omega) & -\sin(\omega) \\ \sin(\omega) & \cos(\omega) \end{bmatrix}$$

and hence the diagonal matrix $\mathbf{K}$ becomes a block diagonal matrix:

$$\mathbf{K} = \begin{bmatrix} \mathbf{B}_{\mu_1,\omega_1} & 0 & \dots & 0 \\ 0 & \mathbf{B}_{\mu_2,\omega_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \mathbf{B}_{\mu_m,\omega_m} \end{bmatrix}$$

Instead of using Jordan blocks in the paper, this project uses the inbuilt support for complex numbers in PyTorch. It is believed that the use of inbuilt autograd features would unleash more low-level optimisations than implementing Jordan blocks at a higher level.

Precisely, in my implementation, the encoder and decoder use real numbers, and $\mathbf{K}$ uses complex numbers. The outputs of the encoder are first grouped into pairs $(\mu_i, \omega_i)$, representing a complex latent coordinate $\exp(\mu_i) \cdot (\cos(\omega_i) + i \sin(\omega_i))$, before advanced linearly by $\mathbf{K}$. After the latent coordinates are advanced, the complex coordinates are decoupled back into real and imaginary parts before passing into the real decoder. The motivation for using complex numbers in $\mathbf{K}$ has been discussed in section 2.1.4.

**Loss function** Recall that the loss function is a combination of the three losses: $\mathcal{L}_{recon}$, $\mathcal{L}_{linear}$ and $\mathcal{L}_{pred}$. In the implementation, $\mathcal{L}_{recon}$ is trained with NN states at every time step, taking linear time. In contrast, training $\mathcal{L}_{linear}$ and $\mathcal{L}_{pred}$ with trajectories spanning all time intervals $[k, k+t]$ requires quadratic time, significantly lengthening the training. Therefore, predictions for $\mathcal{L}_{linear}$ and $\mathcal{L}_{pred}$ always start at time 0, so the overall training is linear to the trajectory lengths.

**Architectural hyperparameters** The architectural hyperparmaters for the autoencoder include (1) the number of hidden layers $h$ (in the encoder and decoder), (2) the width of hidden layers $w$ (in the encoder and decoder), and (3) the dimension of latent space $m$.

Recall that each trajectory dataset is generated from the training trajectories of an architecturally-different NN. As the architecture of a NN defines its dynamics, each dataset characterises the dynamics of a different dynamical system. Therefore, the architectural hyperparameters of the autoencoder have to be separately determined for each dataset.

The search of architectural hyperparameters is done via *hill climbing*:

1. Recall $n$ being the dimension of the NN states. Let $m_0$ be the closest power of

2 to $n$. Start the hyperparameter search with $h = 1$ and $w = m = m_0$. Define $(h, w, m)$ as a *hyperparameter set*.

2. In each iteration, train the autoencoder with a neighbouring hyperparameter set. A neighbouring set is defined as a hyperparameter set that is identical to the current set, except that one of the hyperparameters is modified. The modification of each hyperparameter follows the rules below:

$$h \leftarrow h \pm 1 \qquad w \leftarrow 2w \ \texttt{or} \ \frac{w}{2} \qquad m \leftarrow 2m \ \texttt{or} \ \frac{m}{2}$$

   with the contraint that $h, w, m \geq 0$.

   If the neighbouring hyperparameter set results in a better validation loss, adopt it in place of the current set. Otherwise, keep the current set.

3. Repeat step 2 until all neighbouring hyperparameter sets are no better than the current set. When this is reached, stop the search and adopt the current set as the final hyperparameters for the model.

Note that both the width of hidden layers $w$ and the dimension of latent space $m$ are always in powers of 2.

As the changes are made incrementally, the search would stop at a locally optimal hyperparameter set. Given that the loss landscape here is anticipated to be approximately convex, the solution found via hill-climbing should be very close (if not identical) to the global optimum. More complex randomised techniques (e.g., simulated annealing) would likely be overkill.

### 3.3.4 Weaknesses

After some benchmarking, it is found that the vanilla autoencoder proposed in Lusch *et al.* (2018) is quite restrictive in training dynamics prediction. As soon as more non-linear layers are included in the NNs, the predictions become inaccurate. Thus, the use of the autoencoder is somewhat limited for this task, especially where non-linear activations are centric in the vast majority of NN architecture. A more comprehensive evaluation of the model in comparison with other baselines will be available in chapter 4.

Up to this point, the core implementation criteria of the project have been accomplished. Recall that the extension part of the project is to devise an approach to improve the training dynamics predictions. For this purpose, a novel architecture, `KoopmaNNet`, will be presented in the next section. It will be shown that `KoopmaNNet` can predict the training dynamics of more complex NNs, on which the vanilla autoencoder has failed.

## 3.4 KoopmaNNet

In response to the weaknesses of the vanilla autoencoder above, I would like to propose a novel architecture, `KoopmaNNet`, to improve the predictions of NN training dynamics.

`KoopmaNNet` is a graph autoencoder based on Interaction Networks in Battaglia *et al.* (2016). It leverages the forward and backpropagation relations of the weights/biases in the NNs and performs a trainable message-passing protocol in between.

### 3.4.1 Motivation

The weak performance of the vanilla autoencoder in training dynamics prediction may be due to its failure to exploit the intrinsic graphical structure of NNs. The weights and biases in NNs are flattened before feeding into the autoencoder, dropping all relations between nodes and edges. Therefore, injecting an inductive bias into the architecture utilising these relations would be desirable. This observation calls for the use of graph neural networks (GNN).

Nevertheless, it is not entirely straightforward to directly employ the graphical structure of NNs as inputs of GNNs. In neural networks, all edges carry weights, while perceptrons may or may not have a bias. However, most current GNN architectures focus more on node predictions than edge weight predictions. Hence, the proposed model, `KoopmaNNet`, would need to start with some form of graph transformation.

The following subsections will detail the inner workings of the architecture.

### 3.4.2 Graph transformation

Each feedforward NN can be modelled as a directed acyclic graph, with the direction of each edge following the flow of data during forward propagation. We regard this as the *"NN graph"* for any feedforward NN.

Before feeding the NN graph into `KoopmaNNet`, we first need to transform it to be more "GNN-friendly". Most of the current GNN architectures focus heavily on nodes, so each edge weight and node bias in the NN should ideally rest on separate nodes in the transformed graph. How should the graph be transformed?

Consider the flow of data when NNs are trained. Forward propagation passes input samples from the first to the last layer of the network, at which data is read out and loss evaluated. Backpropagation passes the gradients backward and aggregates them following the chain rule. Since training trajectories are defined by the results of forward and backpropagation, we could hopefully capture the dynamics better by constructing a graph showing these relations.

This inspires the graph transformation algorithm below:

1. Let the transformed graph be $G = (V, E)$. $G$ is directed.
2. For every edge and biased node in the NN graph, create a corresponding node in $V$. Create one additional node in $V$ representing the training loss of the NN under the current state. The attributes of each node consist of (1) its (unique) identifier and (2) its weight/bias/loss.

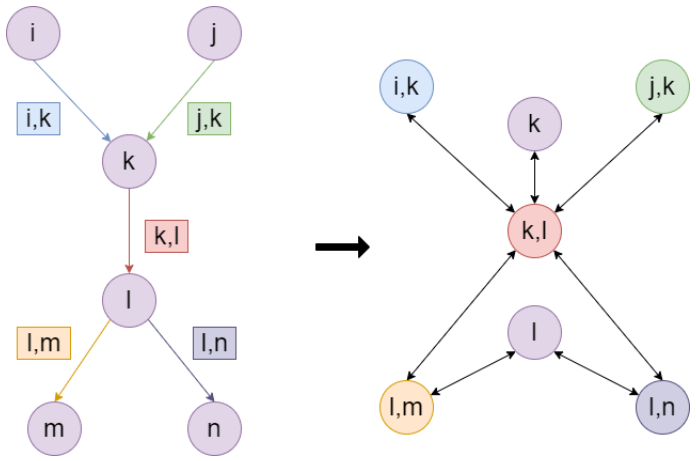



Figure 3.2: An example graph transformed by the algorithm. The graph on the left represents a NN, and the graph on the right represents the transformed graph. Each bidirectional edge shown in the right graph symbolises two directed edges in the transformed graph.

Label $u_i$ as the node in $G$ corresponding to biased node $i$ in the NN graph, and $e_{i,j}$ the node for edge $i \to j$ in the NN graph.

3. For any three nodes $i, j, k$ in the NN graph, if there exists edges $i \to j$ and $j \to k$, create two edges $e_{i,j} \to e_{j,k}$ and $e_{j,k} \to e_{i,j}$ in $G$. These edges shall be attributed with their (unique) identifiers. The former edge represents their relation in forward propagation and the latter in backpropagation.

4. For any two nodes $i, j$ in the NN graph, if $i$ has a bias and there exists an edge $i \to j$, create two edges $u_i \to e_{i,j}$ and $e_{i,j} \to u_i$ in $G$. The rest is analogous to (3).

5. For each edge and biased node in the last layer of the NN graph, make an edge from and to the loss node in $G$.

Figure 3.2 shows an example of the transformation.

In the transformed graph $G$, each node could represent an edge, a biased node, or the training loss of the NN. An edge exists from $u$ to $v$ if and only if:

- During forward propagation, data fed into $v$ is directly computed from $u$, or
- During backpropagation, $u$ directly contributes to the gradient of $v$.

Under this setting, an edge in $G$ loosely corresponds to the effect of an activation function in the NN. The nodes and edges in $G$ together represent all the data flows in the network when a NN is trained. To distinguish between the data flows of forward and backpropagation, edges in $G$ should be directed. Also, all nodes and edges in $G$ should be uniquely identified since the data flowing through the NN components they represent are different.

Although the training loss is not a part of the NN graph, backpropagation always begins with the loss, forming part of the training dynamics. The loss should also be included in $G$ to complete the dynamical system.

This algorithm transforms a NN graph into a graph that captures the forward and backpropagation relations during training. The following subsection will illustrate how node and edge embeddings are created in the transformed graph.

### 3.4.3 Embeddings

The transformed graph is not taken directly as the input of `KoopmaNNet`, as the raw attributes may not be spanning a good space. Higher-dimensional embeddings are generated for each node and edge based on their attributes to put them into a more expressive and flexible space.

For each node $u_i \in V$ identified as $id_i$, let $v_i$ be the weight/bias/loss (later referred as "value") it is carrying, its embedding $\mathbf{x}_i$ is constructed as follows:

$$\mathbf{x}_i = \sigma_N(id_i) \mid\mid w(v_i)$$

where $\sigma_N$ is a trainable lookup table encoding node ids, and $w$ is an MLP encoding node values. In other words, the embedding of a node is the concatenation of its node-unique and weight-dependent representations. Since they represent information from different dimensions, the two representations are concatenated rather than added together.

The embedding of an edge $u_i \to u_j \in E$ is constructed in a similar way:

$$\mathbf{e}_{i,j} = \sigma_E(id_{i \to j})$$

where $\sigma_E$ is a trainable lookup table encoding edge ids.

Recall that each edge in the transformed graph corresponds to a node in the NN graph and represents the effect of its activation. Although activations are categorised (e.g., linear, ReLU, sigmoid), their categories are not included as edge attributes since category-unique properties would have already been included in the more granular edge-unique representations.

After creating embeddings for the nodes and edges in the transformed graph, the graph is fed into graph convolutional layers below.

### 3.4.4 Graph convolutions

Recall the equation for message passing layers:

$$\mathbf{x}_i^{(t+1)} = \gamma(\mathbf{x}_i^{(t)}, \square_{j \in \mathcal{N}(i)}\ \phi(\mathbf{x}_i^{(t)}, \mathbf{x}_j^{(t)}, \mathbf{e}_{j,i}))$$

The graph convolutions in `KoopmaNNet` are based on the ideas of Interaction Networks from Battaglia *et al.* (2016), where (1) the message construction function $\phi$ is an MLP, (2) $\square$ aggregates messages from neighbours with a "sum", and (3) the node update function $\gamma$ is also an MLP. Traditional convolutional layers, such as GCN and GAT, are not used here as their flexibility is rather insufficient for this task. More discussion about their performance will be left for the next chapter.

### 3.4.5 The proposed model

Formally, suppose there are $n$ nodes and $m$ edges in the transformed graph $G$. The graph can be represented by an adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, a list of node attributes $(id_i, v_i)$, and a list of edge attributes $(id_{i \to j})$. Create trainable $c$-dimensional embeddings for each node and edge by:

$$\begin{aligned} \mathbf{x}_i &= \sigma_N(id_i) \mid\mid w(v_i) \\ \mathbf{e}_{i,j} &= \sigma_E(id_{i \to j}) \end{aligned}$$

Denote the node embedding matrix by $\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \mathbf{x}_2 \dots \mathbf{x}_n \end{bmatrix}^\intercal \in \mathbb{R}^{n \times c}$ and the edge embedding matrix by $\mathbf{E} = \begin{bmatrix} \mathbf{e}_{i_1,j_1} \mathbf{e}_{i_2,j_2} \dots \mathbf{e}_{i_m,j_m} \end{bmatrix}^\intercal \in \mathbb{R}^{m \times c}$.

For simplicity, the above convolution can be abstracted as:

$$\mathbf{X}^{(t+1)} = conv_{\mathbf{A}}(\mathbf{X}^{(t)}, \mathbf{E})$$

since $\mathbf{A}$ is constant for a given dataset.

Using this we can define the graph encoder $\boldsymbol{\varphi}$:

$$\begin{aligned} \mathbf{X}^{(1)} &= conv_{\mathbf{A}}(\mathbf{X}, \mathbf{E}) \\ \mathbf{X}^{(2)} &= conv_{\mathbf{A}}(\mathbf{X}^{(1)}, \mathbf{E}) \\ \mathbf{y} &= \phi(\mathbf{X}^{(2)}) \end{aligned}$$

with $\phi : \mathbb{R}^{n \times c} \to \mathbb{R}^h$ being an MLP, where $h$ is the dimension of the latent space.

The latent coordinates $t$ time steps after are obtained in the same way as the vanilla autoencoder:

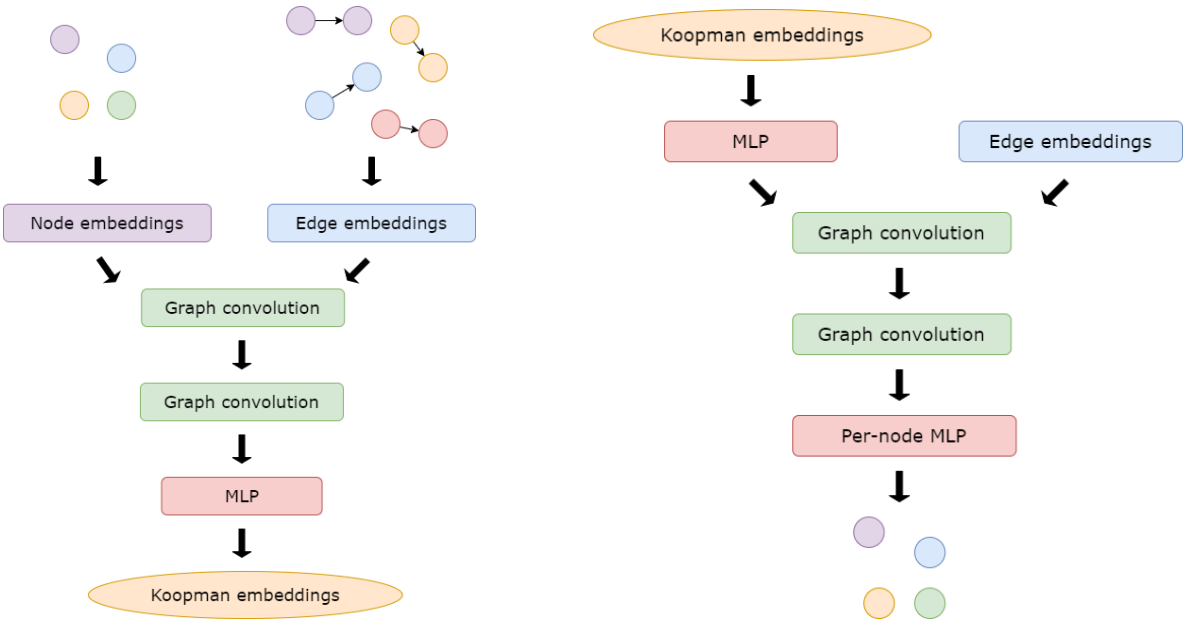
$$\widetilde{\mathbf{y}} = \mathbf{K}^t \mathbf{y}$$

Note that the notation here is a little sloppy: Before $\mathbf{y}$ is linearly advanced by $\mathbf{K}$, $\mathbf{y}$ is grouped into (real, imaginary) pairs to form complex numbers and decomposed after, as described in section 3.3.3. For ease of reading, these operations are omitted from the equation above.

The graph decoder $\boldsymbol{\varphi}^{-1}$ can be defined in a similar way:

$$\begin{aligned} \widetilde{\mathbf{X}}^{(2)} &= \phi^{-1}(\widetilde{\mathbf{y}}) \\ \widetilde{\mathbf{X}}^{(1)} &= conv_{\mathbf{A}}(\widetilde{\mathbf{X}}^{(2)}, \mathbf{E}) \\ \widetilde{\mathbf{X}} &= conv_{\mathbf{A}}(\widetilde{\mathbf{X}}^{(1)}, \mathbf{E}) \end{aligned}$$

with $\phi^{-1} : \mathbb{R}^h \to \mathbb{R}^{n \times c}$ being an MLP as well. Finally, a per-node MLP is applied to the features of every node to obtain a prediction value for each node, representing its weight/bias/loss. This concludes `KoopmaNNet`.




(a) Encoder of `KoopmaNNet` (b) Decoder of `KoopmaNNet`

Figure 3.3: The architectures of the graph encoder and decoder of `KoopmaNNet`.

## 3.5 Architectural evaluation

There are several drawbacks for the Lusch autoencoder to using MLP for its encoder and decoder. MLPs stack over several fully-connected layers, where the number of parameters in each layer is quadratic to the input dimension. The high dimension of the autoencoder is likely to result in a large number of local minima, which impedes optimisation. Due to many parameters and the lack of inductive bias, large MLPs are prone to overfitting. Thus, it should be reasonable to expect that the Lusch autoencoder does not scale to bigger NNs.

In contrast, based on graph neural networks, `KoopmaNNet` utilises the intrinsic graph structure of NNs as an inductive bias. With the inductive bias, not only the number of parameters in `KoopmaNNet` can be reduced, but they can also be devoted to specific operations aiming to capture more complex relations (e.g., message passing). `KoopmaNNet` also uses graph convolutional layers to capture local neighbourhood features. Each graph convolutional layer uses MLPs for generating messages and updating nodes for flexibility. As several convolutional layers are stacked one after another, each layer is tasked to analyse a larger neighbourhood for each node compared with its previous layer. Eventually, after the graph convolutions, an MLP is used to condense the node components into a much smaller Koopman embedding in the latent space. Only by considering the graph structure of NNs is it possible to exploit the concept of "neighbourhood" and thus model the relations of forward and backpropagation.

In `KoopmaNNet`, there is no local attention mechanism like the one seen in graph attention networks (GAT). It is proposed that the relative importance of each edge and neighbour has already been accounted for during message generation and node updates with flexible MLPs. In other words, GAT is a more restricted formulation of the convolutional layers in `KoopmaNNet`. Therefore a separate attention mechanism would not be

required. The experiments also empirically show that `KoopmaNNet` with local attention performs even worse than its counterpart without it.

## 3.6 Training

The weights and biases of the models were initialised randomly in the range $(-w, w)$ for $w = \frac{1}{\sqrt{d}}$, where $d$ is the dimension of the input of their layers. For the learning process, the Adam optimiser[3] was used with learning rate starting at $1e{-}3$. A waterfall schedule was adopted, multiplying the learning rate by 0.7 whenever the validation loss did not decrease for 30 epochs. $L_2$-regularisation was also applied, with the weight decay searched among powers of 10. The final model had the lowest validation loss among all epochs. The test loss of the final model was then evaluated and reported.

The models were trained on an NVIDIA Titan Xp GPU. The vanilla autoencoders were trained for 6-12 hours each, while `KoopmaNNet` models were trained for 8-16 hours. The limiting factor for training time for bigger models is usually the amount of GPU memory available.

## 3.7 Repository overview

An overview of the repository used in this project is shown in Table 3.1. All project code is listed under the directory `src/`. Since the relations between entities are not complex, the project directory has at most two layers, making it easier to look up and reference.

A directory is created for each task outlined in section 3.2. There are routines retrieving datasets and specifying the NN architectures used within each task directory. The APIs are standardised across different tasks for ease of access by the main routines.

The main routines for dataset generation and model training are in the root directory, with their specific parameters for different tasks and architectures.

| Directory | Details |
|---|---|
| `src/` | Main routines for dataset generation and model training |
| `src/models` | Dynamics prediction models used within this dissertation |
| `src/reports` | Model evaluation and trajectory visualisation scripts |
| `src/util` | Utilities for main routine |
| `src/linear` | Routines related to the linear regression task |
| `src/logistic` | Routines related to the classification task |
| `src/ode` | Routines related to the DE solver task |
| `src/digits` | Routines related to the image classification task |

Table 3.1: Repository overview

[3] Kingma *et al.* (2014)

# Chapter 4

# Evaluation

*This chapter begins with an evaluation of the deep learning models in comparison with the theoretical baselines. The performance of* `KoopmaNNet` *is then contrasted with the approaches of two recent papers and tested for robustness across variations of training environments. The properties of the predicted trajectories are also analysed. At the end of the chapter, the project is evaluated against the success criteria coined in section 2.6.*

## 4.1 Model comparison

In this section, the Lusch autoencoder and `KoopmaNNet` will be compared against two conventional Koopman analysis baselines, evaluated on the deviations of the weight/bias predictions. It will be shown that:

- The deep learning approaches outperform the theoretical ones
- The proposed `KoopmaNNet` in this dissertation wins over the Lusch autoencoder by a significant margin

This section will also contrast `KoopmaNNet` with the methodology proposed in Dogra and Redman (2020) in identifying the systems in which NNs train.

### 4.1.1 Baselines

To investigate the performance of deep learning approaches for linearising NN training trajectories, theoretical baselines are used for comparison. However, since data-driven Koopman analysis is a newly arising field, few conventional baselines are available. The most suitable theoretical baselines, DMD and SINDy, are thus chosen for this study. DMD finds the best linear fit to the dynamics directly. SINDy realises eDMD by augmenting the dynamics with a selected few non-linear candidate functions and performing a best linear fit on the augmented dynamics. Their inner workings have been described in section 2.4.1.

| | Regression | Classification | | | DE solver | Image Classification | | |
|---|---|---|---|---|---|---|---|---|
| Model | 1-fc | 1-fc | 2-fc | 3-fc | 3-fc | 1-conv-1-fc | 1-conv-2-fc | 2-conv-1-fc |
| DMD | 48.1506 | 0.0002 | 0.0591 | 0.0886 | 0.0671 | 0.6268 | 0.4245 | 0.3687 |
| eDMD (SINDy) | - | 0.0791 | 0.0710 | - | 0.1001 | - | - | - |
| Lusch (2-layer MLP) | 48.4885 | 2.676E-05 | 0.0051 | 0.0419 | 0.0353 | 0.0336 | 0.1603 | 0.2365 |
| Lusch (3-layer MLP) | 48.1532 | **2.345E-05** | 0.0020 | 0.0370 | 0.0323 | 0.0208 | 0.1203 | 0.2463 |
| Lusch (4-layer MLP) | 49.2249 | 7.275E-06 | 0.0124 | 0.0649 | 0.0367 | 0.0177 | 0.1257 | 0.2437 |
| `KoopmaNNet` (GCN) | 50.3453 | 4.193E-05 | 0.0027 | 0.0570 | 0.0339 | 0.0266 | 0.2799 | 0.4153 |
| `KoopmaNNet` (GAT) | **36.2885** | 0.0001 | 0.0092 | 0.0483 | 0.0294 | 0.0307 | 0.1522 | 0.2108 |
| `KoopmaNNet` (Transformer) | 49.5184 | 0.0001 | 0.0010 | 0.0305 | 0.0179 | 0.0246 | 0.0785 | 0.1285 |
| `KoopmaNNet` (IN) | 48.1291 | 2.436E-05 | **0.0005** | **0.0276** | **0.0119** | **0.0122** | **0.0366** | **0.1153** |

Table 4.1: Prediction losses on the trajectory datasets. The NN architecture used in each dataset is labelled with either $x$-fc or $y$-conv-$z$-fc. The former refers to architecture with $x$ layers of fully-connected layers, and the latter with $y$ convolutional layers followed by $z$ fully-connected layers. The latent space dimensions of the deep learning models are set to be the closest power of 2 greater than the number of weights and biases in the respective NNs. Some entries of SINDy are left out as the identified systems produced out-of-scale predictions.

### 4.1.2 Trajectory prediction

To better understand the performances of the deep learning models, multiple set-ups have been trained and tested for each model and dataset. For the Lusch autoencoder, MLPs of various depths were tested. For `KoopmaNNet`, graph convolutional layers with different levels of complexity were tested. For each model and dataset, the prediction loss $\mathcal{L}_{pred}$ (see section 3.1) is reported. It can be interpreted as the average squared deviation in the weight/bias predictions. The results are shown in Table 4.1.

**Theoretical vs. Deep learning approaches** The following table compares the performance of DMD with the 3-layer Lusch autoencoder using the one-tailed t-test ($\alpha = 0.01$). The 3-layer Lusch autoencoder is chosen because (1) MLP is a simpler architecture than all other deep learning approaches, and (2) the 3-layer model is the best performing MLP autoencoder on average.

| Regression | Classification | | | DE solver | Image Classification | | |
|---|---|---|---|---|---|---|---|
| 1-fc | 1-fc | 2-fc | 3-fc | 3-fc | 1-conv-1-fc | 1-conv-2-fc | 2-conv-1-fc |
| 5.055E-01 | 2.360E-05 | 1.975E-06 | 5.124E-05 | 3.319E-04 | 2.565E-06 | 1.346E-06 | 8.116E-04 |

Table 4.2: p-values comparing DMD and the 3-layer Lusch autoencoder on trajectory predictions.

The significance values in Table 4.2 assert that deep learning has a substantial advantage over theoretical approaches. Theoretical baselines are often unable to capture more complex relations. DMD fits a linear model to the dynamics, which is somewhat naive when non-linearity is ubiquitous in NNs. SINDy assumes that a Koopman-invariant subspace can be approximated with a library of candidate functions. However, as backpropagation works by stacking gradient terms one-within-another, the gradient expressions quickly suffer from a combinatorial explosion. Thus the library is unlikely to be sufficiently expressive to approximate such a subspace, identifying an "invariant" system that is largely spurious. This accounts for its inability to produce sound predictions.


(a) DE solver: 3-fc

(b) Image classification: 1-conv-2-fc

Figure 4.1: Example of weight/bias trajectories and the corresponding predictions made by DMD (purple dashed line), Lusch autoencoder (green dashed line), and `KoopmaNNet` (red dashed line). The x-axis and y-axis each represents a weight/bias.

**Lusch vs. KoopmaNNet** The following table compares the performance of the 3-layer Lusch autoencoder and `KoopmaNNet`:

| **Regression** | **Classification** | | | **DE solver** | **Image Classification** | | |
|---|---|---|---|---|---|---|---|
| 1-fc | 1-fc | 2-fc | 3-fc | 3-fc | 1-conv-1-fc | 1-conv-2-fc | 2-conv-1-fc |
| 2.862E-01 | 5.688E-01 | 2.576E-03 | 6.713E-04 | 8.022E-04 | 6.560E-03 | 4.380E-04 | 1.549E-04 |

Table 4.3: p-values comparing the 3-layer Lusch autoencoder and `KoopmaNNet` (IN) on trajectory predictions.

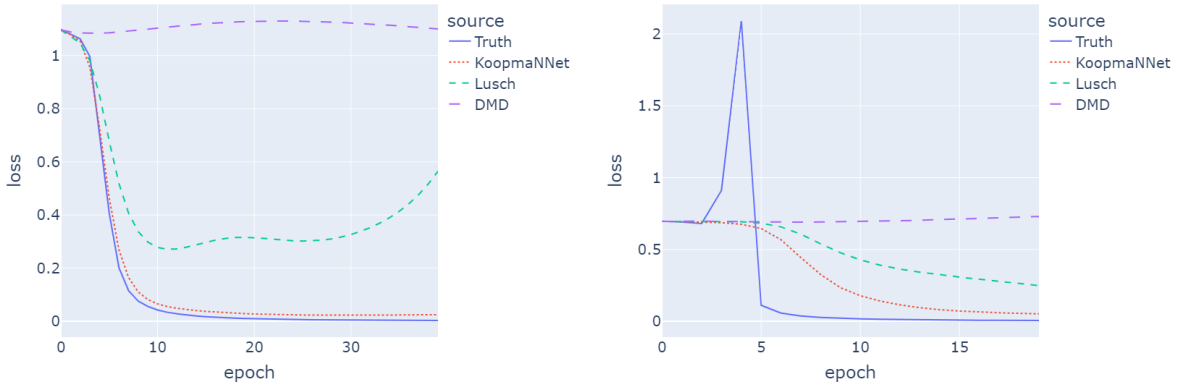
For most of the datasets, the proposed `KoopmaNNet` in this dissertation significantly outperforms the Lusch autoencoder in predicting NN training trajectories via a linear representation. The exceptions are the two datasets corresponding to the dynamics of 1-layer NNs. They could be explained by the fact that the Lusch autoencoder is already capable of making excellent predictions with MLPs, and thus it is not easy to improve significantly further. Figure 4.1 contrasts the actual and predicted trajectories of two weights/biases on two different datasets.

The results in Table 4.1 have shown that as NNs grow in size and depth, the predicted trajectories of some weights and biases may become less accurate. Despite the inaccuracies, the predicted trajectories of the training dynamics follow some valuable patterns when viewed as a whole. These patterns will be detailed in the following subsection.

### 4.1.3 Loss performance

For more complex NNs, it may seem that the average deviations in Table 4.1 are too large to be helpful. However, when the predicted sets of weights and biases are plugged into the NNs, they produce loss trajectories closely resembling the ground truth. Figure 4.2 shows a comparison between the actual training loss and the loss of the NN using the predicted weights and biases in each step.

Therefore, despite not being able to predict the exact trajectory on which the weights




(a) Classification: 3-fc

(b) Image classification: 2-conv-1-fc

Figure 4.2: Example of loss trajectories and the corresponding predictions made by DMD (purple dashed line), Lusch autoencoder (green dashed line), and `KoopmaNNet` (red dashed line).

and biases evolve, `KoopmaNNet` can find a "linearisable" trajectory having a similar (if not better) loss performance as the actual trajectory. The predicted trajectories can often avoid training instability, converging along a smoother path than the actual ones. From the perspective of the individual weights and biases, this implies that the model can recognise the dominant features deciding the loss dynamics of the NN and leave out the rest so that the predicted trajectory is simple enough to be projected into a linear space.

To quantify the loss performance of `KoopmaNNet`, a similar evaluation methodology as Dogra and Redman (2020) is adopted. We look at the *loss performance* $r$, the ratio of the differences between the initial loss and the final loss in the actual trajectory to that of the predicted trajectory:

$$r = \frac{l_0 - l_{pred}}{l_0 - l_{true}}$$

where $l_0$ stands for the initial training loss, and $l_{pred}, l_{true}$ stands for the final training loss in the predicted trajectory and the actual trajectory. The closer $r$ is to 1, the closer the final loss of the predicted trajectory is to the actual trajectory. The mean values for $r$ (in percentages) are tabulated as follows:

| **Regression** | **Classification** | | | **DE solver** | **Image Classification** | | |
|---|---|---|---|---|---|---|---|
| 1-fc | 1-fc | 2-fc | 3-fc | 3-fc | 1-conv-1-fc | 1-conv-2-fc | 2-conv-1-fc |
| 100.3% | 99.2% | 98.6% | 92.2% | 87.6% | 98.0% | 100.6% | 90.5% |

This confirms that the trajectories predicted by `KoopmaNNet`, which can be projected into a linear latent space (*"linearisable"*), indeed have a similar loss performance as the non-linear training trajectories computed via backpropagation.

| | 128 | 64 | 32 | 16 | 8 | 4 |
|---|---|---|---|---|---|---|
| Classification: 2-fc | 0.0005 | 0.0006 | 0.0015 | 0.0022 | 0.0030 | 0.0130 |
| Image: 1-conv-2-fc | 0.0393 | 0.0513 | 0.0805 | 0.0647 | 0.0562 | 0.4536 |

Table 4.4: Prediction losses of `KoopmaNNet` evaluated on `Classification: 2-fc` and `Image: 1-conv-2-fc` with latent dimension varying among powers of 2.

### 4.1.4 Comparison with previous approaches

Recall that Dogra and Redman (2020) and Tano *et al.* (2020) also attempt to identify a linear relationship on which the weights and biases evolve, as discussed in section 2.4.3. It is proposed that the methodology discussed in this dissertation has produced superior results compared to the papers:

1. In the papers, the linear Koopman operator identified is local to the weights and bias of a perceptron/layer. However, `KoopmaNNet` can identify a globally linear operator (together with a set of eigenfunctions) that applies to the entire network.
2. In Tano *et al.* (ibid.), the Koopman operator identified is only applicable for a few training epochs. In Dogra and Redman (2020), the operator identified is valid only after the weights and biases have entered basins of local loss minima. In comparison, the operator identified in `KoopmaNNet` is valid throughout the entire training process without requiring proximity to any loss minima.
3. The Koopman operator in the papers applies only to a specific training instance. In contrast, the operator in `KoopmaNNet` can be applied to any training trajectories randomly initialised with a uniform distribution of weights and biases.

In the benchmarks above, the dimension of the latent space is always the closest power of 2 greater than the number of weights and biases in the NN. In the following subsection, it will be shown that to yield a similar loss performance, the number of eigenfunctions needed for each NN is much smaller.

## 4.2 Latent space dimensions

In section 4.1.3, it was shown that the trajectories predicted by `KoopmaNNet` evaluate to a similar loss of the actual trajectories. It would be natural to ask: What is the smallest number of eigenfunctions required to capture the training dynamics? This section tries to answer this question.

Two NN architectures are chosen for evaluation, including (1) a 2-layer feedforward NN trained for the classification problem and (2) a CNN with one convolutional layer and two fully-connected layers for the image classification problem. `KoopmaNNet` with a latent space dimension varying along powers of 2 is trained on their trajectories. The prediction losses are reported in Table 4.4, and the loss performances $r$ are in Table 4.5.

| | 128 | 64 | 32 | 16 | 8 | 4 |
|---|---|---|---|---|---|---|
| Classification: 2-fc | 98.0% | 98.4% | 98.5% | 98.5% | 98.2% | 90.1% |
| Image: 1-conv-2-fc | 100.6% | 100.7% | 100.5% | 100.6% | 100.7% | 0.1% |

Table 4.5: Loss performances of `KoopmaNNet` evaluated on `Classification: 2-fc` and `Image: 1-conv-2-fc` with latent dimension varying among powers of 2.

The tables have shown that 8 eigenfunctions are sufficient to capture the dynamics of both NNs. Comparing this to the number of weights and biases in these NNs (106 for the classification NN and 194 for the image classification NN), the number of eigenfunctions is smaller by a magnitude.

As the latent dimensions are reduced, the prediction losses show an increasing trend, but $r$ does not follow a corresponding decreasing trend. It could be due to the redundancy of eigenfunctions. Once there are sufficient eigenfunctions to capture the dominant trends in the training dynamics, even though a further increase in the number of eigenfunctions could capture more subtle dynamics, they do not have a noticeable impact on the loss of the NNs.

The results in this subsection highlight the possibility of representing the training of NNs with over a hundred parameters using less than ten linearly-evolving eigenfunctions. This provides some evidence on the over-parametrisation of NNs, while also bringing some new insights for network pruning.

## 4.3 Robustness to variations

This section aims to showcase the robustness of `KoopmaNNet` when confronted with training trajectories with various specifications.

### 4.3.1 Non-linearity

To investigate the versatility of `KoopmaNNet` on different non-linearity types, the model is evaluated on the training trajectories of the same NN architectures but with different non-linear activation functions. The same datasets from the last subsection are used. The loss performances $r$ are as follows:

| | ELU ($\alpha = 1$) | ReLU | Leaky ReLU | Sigmoid | Tanh |
|---|---|---|---|---|---|
| Classification: 2-fc | 98.6% | 97.3% | 96.7% | 99.2% | 98.5% |
| Image: 1-conv-2-fc | 96.9% | 83.6% | 93.5% | 100.6% | 98.0% |

In general, the losses of NNs equipped with sigmoid, tanh, and ELU are better predicted than ReLU and leaky ReLU. The smoothness of functions might explain it: sigmoid, tanh, and ELU (with $\alpha = 1$) are smooth functions, while the gradients of ReLU and leaky ReLU are discontinuous at $x = 0$. For NNs with non-smooth activations, their training would follow a "rougher" trajectory, and therefore it would be harder to linearise the dynamics. Nevertheless, the loss performances are robust regardless of the

choice of activation functions.

### 4.3.2 Optimisers

In this section, `KoopmaNNet` is evaluated on the training trajectories produced by 4 different optimisers, which include SGD, Adagrad[1], Adadelta[2] and Adam[3]. The loss performances are:

| | SGD | Adagrad | Adadelta | Adam |
|---|---|---|---|---|
| Classification: 2-fc | 98.6% | 98.1% | 94.2% | 98.8% |
| Image: 1-conv-2-fc | 100.6% | 95.7% | 97.7% | 99.2% |

As shown, `KoopmaNNet` is robust across the choice of optimisers.

### 4.3.3 Stochastic training

Recall that all datasets used in previous sections are generated with batch gradient descent, i.e., all training data is grouped into a single batch to produce more stable trajectories. However, due to many training samples, most NNs nowadays are trained with stochastic gradient descent to reduce training time. To show that `KoopmaNNet` is robust regardless of the stochasticity of training, this section reports the loss performances of `KoopmaNNet` when applied to stochastic trajectories:

| | SGD | Adagrad | Adadelta | Adam |
|---|---|---|---|---|
| Classification: 2-fc | 99.9% | 97.4% | 99.7% | 99.6% |
| Image: 1-conv-2-fc | 97.8% | 99.4% | 99.3% | 99.1% |

The results show that `KoopmaNNet` can handle stochastic trajectories as well.

## 4.4 Success criteria

In section 2.6, a list of workflow specifications has been specified for this project. This section compares the work accomplished against the specifications in the success criteria.

1. *Generate suitable datasets covering a variety of tasks and NN architectures. At least regression and classification should be included. The datasets should also span across architectures on different scales.* ✓

   The datasets generated for this project have been described in section 3.2. One regression and two classification tasks have been studied, where NN architectures spanning across one layer to three layers (with and without convolutions) have been trained for the tasks. Datasets also include trajectories of architectures with different activation functions, optimisers, and stochasticity.

[1] Duchi *et al.* (2011)
[2] Zeiler (2012)
[3] Kingma *et al.* (2014)

2. *Implement a pipeline for training (and testing) models on different datasets.* ✓

   General training and testing environments have been implemented, allowing an orthogonal choice of model and dataset for each run.

3. *Implement the autoencoder proposed in Lusch* et al. *(2018) and adapt the architecture for this problem.* ✓

   As described in section 3.3, the Lusch autoencoder has been implemented and adapted for NN trajectory prediction.

4. *Select and implement a set of common baselines for comparison. They should include both numerical and deep learning baselines.* ✓

   Two numerical baselines (DMD and SINDy) and two deep learning baselines (Lusch autoencoder and `KoopmaNNet`) have been implemented. 3 to 4 architectural variations have been evaluated for each deep learning baseline.

5. *Interpret the results and evaluate the model's performance in predicting the training trajectories of NNs.* ✓

   The prediction losses of the models were reported and compared, where the predicted trajectories of some weights/biases were plotted and contrasted with the actual trajectories. The models were also evaluated on the loss performance of the NNs using the predicted weights and biases, spanning different architectures, latent space dimensions, activation functions, optimisers, and stochasticity.

6. **Extension** *Improve the existing approach or design a new approach with better performance.* ✓

   A novel architecture, `KoopmaNNet`, has been designed and implemented to improve upon the performance of the Lusch autoencoder. The evaluation in this chapter has shown that `KoopmaNNet` outperforms the Lusch autoencoder in predicting the training trajectories of all non-trivial NN architectures.

To conclude, the project has **achieved and exceeded** all of the core and extension success criteria outlined in chapter 2.6.

# Chapter 5

# Conclusion

## 5.1 Major findings

This project explores the possibility of representing non-linear neural network training trajectories with a linear representation by leveraging Koopman Operator Theory and deep learning. Due to the lack of published attempts for this problem, the approach from Lusch *et al.* (2018) was brought over and adapted. However, the approach quickly reached its limits when confronted with more complex NNs.

To address its weaknesses, a novel architecture, `KoopmaNNet` has been proposed in this dissertation. Based on graph neural networks, `KoopmaNNet` can exploit the intrinsic graph structure of NNs by modelling the data flows between nodes and edges during training. Using the results of `KoopmaNNet`, I have shown that:

1. The training of neural networks can indeed be represented with a **linear** framework using a latent space that is at least **a magnitude smaller** than the NNs.
2. `KoopmaNNet` can **extract the dominant features** affecting the loss dynamics of the NNs and dispose of the rest so that the trajectories can be well expressed with a linear latent space. Trajectories predicted by `KoopmaNNet` still have a highly similar loss performance to the actual ones generated with backpropagation, while at times being **more stable** than the actual training.
3. `KoopmaNNet` can cope with **non-trivial NN architectures**. The proposed model is also **robust** across different activation functions, optimisers, and capable of predicting stochastic training trajectories.

As far as the author is aware, these observations are novel in the field's literature.

It has been shown that the proposed `KoopmaNNet` outperforms Lusch *et al.* (ibid.) in predicting training trajectories of all non-trivial NNs. The latent-space embeddings produced by `KoopmaNNet` are also more versatile than that in Dogra and Redman (2020) and Tano *et al.* (2020). Therefore, to the best of the author's knowledge, the novel `KoopmaNNet` proposed in this dissertation performs better than most of the current approaches in the field.

## 5.2 Lessons learnt

This core part of the project entails using an existing approach in an unexplored context with limited direct support from the current literature. Therefore, there is a need to develop an original interpretation of the paper and adapt its approach. However, there is no guarantee that the approach would work for this problem.

The extension part of the project is a research project by nature. To improve upon the current approaches, a significant amount of time has to be devoted to researching and understanding relevant topics across different fields, selecting techniques from the vast amount available, and innovations targeting the weaknesses of current approaches. The extension part has naturally emerged as the focus of this dissertation due to its performance surpassing the current approaches.

Overall, the experience in the project has provided me with a taste of research, and it has led me to recognise the skills and perseverance required for research. It has also provoked my interest in graph neural networks and the integration of mathematical theories and machine learning to solve problems.

## 5.3 Limitations and future work

Throughout the project, several major shortcomings of `KoopmaNNet` have been identified. These limitations, together with the potential solutions, are explained as follows:

1. *There are too many edges in the transformed graph.*

   The graph transformation algorithm constructs an edge between every pair of weights and biases directly related during forward propagation or backpropagation. This implies an edge between every pair of incoming and outgoing weights for each perceptron, producing a graph with a quadratic number of edges. This construction enlarges the search space of edge embeddings and thus makes the model less scalable.

   One solution would be to insert another node in between, directing all incoming weights of the perceptron into the node and connecting the node to all outgoing weights. This refinement reduces the number of edges from quadratic to linear in the number of weights and biases.

2. `KoopmaNNet` *does not involve graph pooling.*

   In `KoopmaNNet`, every convolutional layer passes messages between every pair of neighbouring nodes. However, not every weight/bias is equally important to the overall dynamics. It is anticipated that the most significant dynamics features arise from clusters instead of individual weights and biases. This calls for graph pooling to collect clusters' features while reducing the size of the graph. Graph pooling devotes more computational resources to analysing higher-level features and shrinks the graph size bit-by-bit; hence it is a natural replacement of the MLP at the end of the encoder of `KoopmaNNet`.

Popular graph pooling approaches include top-k pooling and edge contraction pooling, introduced in Gao *et al.* (2019) and Diehl (2019) respectively. However, most current pooling approaches do not take edge weights. To use pooling in `KoopmaNNet`, either the transformed graph shall be unweighted, or a weighted graph pooling approach has to be designed.

3. *Graph convolutions lack a global perspective of graphs.*

   Graph convolutions are good at gathering local neighbourhood features but are less effective in obtaining global characteristics. For global attention, transformers could be a good alternative. However, transformers were built initially for sequences. To use transformers with graphs, the existence of a structural encoding for graphs would likely be a prerequisite. A recent paper Ying *et al.* (2021) has taken a step on this, which might serve as a good starting point for this direction.

Although the core and extension success criteria have been vastly exceeded, given the directions outlined above, it is anticipated that the project will still go on after the dissertation is submitted in the hope of better results. If `KoopmaNNet` can be scaled further, the linearised training trajectories could potentially bring extensive insights for pruning neural networks, controlling training dynamics, or even interpreting neural networks.

# Bibliography


Battaglia, Peter W. *et al.* (2016). *Interaction Networks for Learning about Objects, Relations and Physics*. arXiv: 1612.00222.

Brunton, Steven L., Bingni W. Brunton, *et al.* (2015). "Koopman invariant subspaces and finite linear representations of nonlinear dynamical systems for control". In: DOI: 10.1371/journal.pone.0150171. arXiv: 1510.03007.

Brunton, Steven L., Joshua L. Proctor, and J. Nathan Kutz (2015). "Discovering governing equations from data: Sparse identification of nonlinear dynamical systems". In: DOI: 10.1073/pnas.1517384113. eprint: arXiv:1509.03580.

Costa-Luis, Casper O. da (2019). "'tqdm': A Fast, Extensible Progress Meter for Python and CLI". In: *Journal of Open Source Software* 4.37, p. 1277. DOI: 10.21105/joss.01277. URL: https://doi.org/10.21105/joss.01277.

Diehl, Frederik (2019). "Edge Contraction Pooling for Graph Neural Networks". In: DOI: https://doi.org/10.48550/arXiv.1905.10990. eprint: arXiv:1905.10990.

Dogra, Akshunna S. (2020). *Dynamical Systems and Neural Networks*. arXiv: 2004.11826.

Dogra, Akshunna S. and William T Redman (2020). "Optimizing Neural Networks via Koopman Operator Theory". In: DOI: arXiv:2006.02361. eprint: arXiv:2006.02361.

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61, pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

Fey, Matthias and Jan E. Lenssen (2019). "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

Gao, Hongyang and Shuiwang Ji (2019). "Graph U-Nets". In: DOI: https://doi.org/10.48550/arXiv.1905.05178. eprint: arXiv:1905.05178.

Harris, Charles R. *et al.* (Sept. 2020). "Array programming with NumPy". In: *Nature* 585.7825, pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

Inc., Plotly Technologies (2015). *Collaborative data science*. URL: https://plot.ly.

Kingma, Diederik P. and Jimmy Ba (2014). "Adam: A Method for Stochastic Optimization". In: DOI: 10.48550/arXiv.1412.6980. eprint: arXiv:1412.6980.

Kipf, Thomas N. and Max Welling (2016). "Semi-Supervised Classification with Graph Convolutional Networks". In: DOI: 10.48550/arXiv.1609.02907. eprint: arXiv:1609.02907.

Kluyver, Thomas *et al.* (2016). "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press, pp. 87–90.

Li, Yunzhu *et al.* (2019). "Learning Compositional Koopman Operators for Model-Based Control". In: DOI: 10.48550/arXiv.1910.08264. eprint: arXiv:1910.08264.

Lusch, Bethany, J. Nathan Kutz, and Steven L. Brunton (2018). "Deep Learning for Universal Linear Embeddings of Nonlinear Dynamics". In: *Nature Communications* 9.4950. DOI: 10.1038/s41467-018-07210-0.

McKinney, Wes *et al.* (2010). "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX, pp. 51–56.

Paszke, Adam *et al.* (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach *et al.* Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Pedregosa, F. *et al.* (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Rowley, CLARENCE W. *et al.* (2009). "Spectral analysis of nonlinear flows". In: *Journal of Fluid Mechanics* 641, pp. 115–127. DOI: 10.1017/S0022112009992059.

Shi, Yunsheng *et al.* (2020). *Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification*. DOI: 10.48550/arXiv.2009.03509. eprint: arXiv:2009.03509.

Silva, Brian de *et al.* (2020). "PySINDy: A Python package for the sparse identification of nonlinear dynamical systems from data". In: *Journal of Open Source Software* 5.49, p. 2104. DOI: 10.21105/joss.02104. URL: https://doi.org/10.21105/joss.02104.

Tano, Mauricio E., Gavin D. Portwood, and Jean C. Ragusa (2020). "Accelerating Training in Artificial Neural Networks with Dynamic Mode Decomposition". In: *CoRR* abs/2006.14371. arXiv: 2006.14371. URL: https://arxiv.org/abs/2006.14371.

Tu, Jonathan H. *et al.* (2013). "On Dynamic Mode Decomposition: Theory and Applications". In: DOI: 10.3934/jcd.2014.1.391. eprint: arXiv:1312.0041.

*UCI Machine Learning Repository: Optical recognition of handwritten digits data set* (1998). URL: https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits.

*UCI Machine Learning Repository: Wine data set* (1991). URL: https://archive.ics.uci.edu/ml/datasets/wine.

Vaswani, Ashish *et al.* (2017). "Attention Is All You Need". In: DOI: 10.48550/arXiv.1706.03762. eprint: arXiv:1706.03762.

Veličković, Petar *et al.* (2017). "Graph Attention Networks". In: DOI: 10.48550/arXiv.1710.10903. eprint: arXiv:1710.10903.

Virtanen, Pauli *et al.* (2020). "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17, pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

Williams, Matthew O., Ioannis G. Kevrekidis, and Clarence W. Rowley (2014). "A Data-Driven Approximation of the Koopman Operator: Extending Dynamic Mode Decomposition". In: DOI: 10.1007/s00332-015-9258-5. eprint: arXiv:1408.4408.

Ying, Chengxuan *et al.* (2021). "Do Transformers Really Perform Bad for Graph Representation?" In: DOI: https://doi.org/10.48550/arXiv.2106.05234. eprint: arXiv:2106.05234.

Zeiler, Matthew D. (2012). "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701. arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701.

# Proposal: Analysing Neural Network Training with Koopman Operator Theory

2336F

## 1 Overview

The success of neural networks (NNs) in machine learning has brought us with a new set of challenges to address. The complexity of non-linear training dynamics of NNs makes them unintuitive to humans. What if we can represent the non-linear training dynamics using a linear framework? If we can transform the non-linear training trajectory into a linear representation (in another space), it opens up a whole new world of possibilities to predict, control and even interpret NN training trajectories with the rich repertoire of linear dynamical analysis tools.

This project has two parts: The core part aims to apply the Koopman autoencoder to predict NN training trajectories, while the extension wishes to leverage on the Koopman eigenfunctions produced by the autoencoder to explore further opportunities. The following subsections will go over the key concepts involved.

### 1.1 Definitions

A NN can be specified by its set of weights/biases $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^n$, which we will later refer to it as the *state* of the NN. As the NN is trained through each time step, the state is updated with respect to the training function $f : \mathcal{X} \to \mathcal{X}$. In other words, $f(\mathbf{x}_k) = \mathbf{x}_{k+1}$. The evolution of the state throughout the training process is referred to as the *training trajectory* (or *training dynamics*) of the NN. However, $f$ is non-linear for most NNs, which makes further analyses hard.

Define $\mathcal{G}$ as a class of *measurement functions* for NNs, such that each member $g : \mathcal{X} \to \mathcal{R}$ maps a NN state to a real number (*measurement*). The quantity measured does not necessarily need to carry an intuitive meaning to humans. The reason for this will soon be explained.

### 1.2 Koopman Operator

NN training has recently been discovered to be intimately connected with the Koopman Operator Theory (KOT) [*Dynamical Systems and Neural Networks*, Akshunna S. Dogra]. It turns out that KOT could be a feasible way for linearising non-linear training dynamics.

Briefly, the Koopman operator for a function $h$ is defined by the rule $\mathcal{K}_h(\phi) := \phi \circ h$, which takes in a function $\phi$ and compose it with $h$.

Recall $f$ as the training function of the NN. Define $\mathcal{K}_f : \mathcal{G} \to \mathcal{G}$ as the Koopman operator for $f$, which takes in a measurement function $g$ and compose it with the training function $f$, i.e. $\mathcal{K}_f(g) := g \circ f$. Intuitively, since $g$ produces a measurement of the current state, $\mathcal{K}_f(g)$ produces a measurement of the next state in time:

$$\mathcal{K}_f(g)(\mathbf{x}_k) = g \circ f(\mathbf{x}_k) = g(\mathbf{x}_{k+1})$$

Therefore, ${\mathcal{K}_f}^m(g)$ takes in a state and produces a measurement of the state $m$ time steps after.

The Koopman operator is linear, since:

$$\begin{aligned}\mathcal{K}_f(\alpha g_1 + \beta g_2)(\mathbf{x}) &= (\alpha g_1 + \beta g_2) \circ f(\mathbf{x}) \\ &= \alpha g_1 \circ f(\mathbf{x}) + \beta g_2 \circ f(\mathbf{x}) \\ &= \alpha \mathcal{K}_f(g_1)(\mathbf{x}) + \beta \mathcal{K}_f(g_2)(\mathbf{x})\end{aligned}$$

Yet, the Koopman operator is infinite-dimensional, impeding its direct use in applied circumstances.

### 1.3 Koopman-invariant Subspace

Consider a (possibly infinite) set of measurement functions $\mathcal{S}$. $\mathcal{S}$ spans a Koopman-invariant subspace if and only if all functions $g$ in this subspace remain within the subspace after being acted on by the Koopman operator $\mathcal{K}$. Equivalently, for any function $g$ that can be represented as a linear combination of $\mathcal{S}$, ${\mathcal{K}_f}^m(g)$ can also be. This is a useful property.

Consider a finite vector of measurement functions that span a Koopman-invariant subspace $\mathcal{G}'$:

$$\mathbf{g} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_m \end{bmatrix}$$

Let's extend the Koopman operator $\mathcal{K}_f$ to act of a vector of measurement functions as well:

$$\mathcal{K}_f(\mathbf{g}) = \begin{bmatrix} \mathcal{K}_f(g_1) \\ \mathcal{K}_f(g_2) \\ \vdots \\ \mathcal{K}_f(g_m) \end{bmatrix}$$

By the property of Koopman-invariant subspace, we know $\mathcal{K}_f(g_i)$ is also in $\mathcal{G}'$, which means $\mathcal{K}_f(g_i)$ can be represented as a linear combination of $\mathbf{g}$. In such case, $\mathcal{K}_f$ can then be represented by a finite-dimensional matrix $\mathbf{K}$, such that:

$$\mathcal{K}_f(\mathbf{g}) = \mathbf{K}\mathbf{g}$$

Therefore, by restricting the domain and co-domain of $\mathcal{K}_f$ from the function space $\mathcal{G}$ to a Koopman-invariant subspace $\mathcal{G}'$ spanned by a finite set of measurement functions, we can get around with the infinite-dimensional property of the Koopman operator and use a finite matrix representation instead.

How is this useful in the context of NNs? If we can find such finite vector of measurement functions $\mathbf{g}$ spanning a Koopman-invariant subspace $\mathcal{G}'$, we can transform the NN state space into another space $\mathcal{Y} \subseteq \mathbb{R}^m$. This is beneficial since:

$$\begin{aligned}\mathbf{g}(\mathbf{x}_{k+m}) &= {\mathcal{K}_f}^m(\mathbf{g})(\mathbf{x}_k) \\ &= (\mathbf{K}^m\mathbf{g})(\mathbf{x}_k) \\ &= \mathbf{K}^m(\mathbf{g}(\mathbf{x}_k))\end{aligned}$$

Which means:

$$\mathbf{x}_{k+m} = \mathbf{g}^{-1}(\mathbf{K}^m(\mathbf{g}(\mathbf{x}_k)))$$

Therefore, if we can find such $\mathbf{g} : \mathcal{X} \to \mathcal{Y}$ that is able to preserve the features in $\mathcal{X}$, we can transform NN states into points in $\mathcal{Y}$. Since the coordinates $\mathbf{g}(\mathbf{x}_k)$ can be advanced linearly in $\mathcal{Y}$, this linear

framework would come in handy for predicting/approximating the trajectories of training dynamics. Note that $\mathbf{g}^{-1}$ reconstructs the NN state from the coordinates in $\mathcal{Y}$.

However, finding a set of measurement functions spanning a Koopman-invariant subspace isn't always straightforward. This gives rise to the eigenfunction approach in the following subsection.

### 1.4 Koopman Eigenfunction

A Koopman eigenfunction $\varphi$ is a special type of measurement function $g : \mathcal{X} \to \mathbb{R}$ that advances linearly upon being acted on by the Koopman operator $\mathcal{K}_f$:

$$\mathcal{K}_f(\varphi)(\mathbf{x}_k) = \lambda \cdot \varphi(\mathbf{x}_k)$$

Which implies

$$\varphi(\mathbf{x}_{k+1}) = \lambda \cdot \varphi(\mathbf{x}_k)$$

Since any set of Koopman eigenfunctions will span a Koopman-invariant subspace, we can construct a finite vector of eigenfunctions $\boldsymbol{\varphi}$:

$$\boldsymbol{\varphi} = \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ \vdots \\ \varphi_p \end{bmatrix}$$

and use $\boldsymbol{\varphi}$ in place of $\mathbf{g}$ as described in the subsection above.

But how do we get an appropriate set of eigenfunctions $\boldsymbol{\varphi}$ whose span preserves most features in the original subspace $\mathcal{X}$?

### 1.5 Koopman Autoencoder

To conclude, we need to estimate $\boldsymbol{\varphi}$, $\boldsymbol{\varphi}^{-1}$ and $\mathbf{K}$ in the following equation:

$$\mathbf{x}_{k+m} = \boldsymbol{\varphi}^{-1}(\mathbf{K}^m(\boldsymbol{\varphi}(\mathbf{x}_k)))$$

Or:

$$\mathbf{y}_k = \boldsymbol{\varphi}(\mathbf{x}_k) \qquad \mathbf{y}_{k+m} = \mathbf{K}^m(\mathbf{y}_k) \qquad \mathbf{x}_{k+m} = \boldsymbol{\varphi}^{-1}(\mathbf{y}_{k+m})$$

where:

- $\mathbf{x}_k$ and $\mathbf{x}_{k+m}$ are NN states (in $\mathcal{X}$) at time step $k$ and $k+m$
- $\mathbf{y}_k$ and $\mathbf{y}_{k+m}$ are the corresponding coordinates in $\mathcal{Y}$
- $\mathbf{K}$ is a diagonal matrix

The task of identifying an appropriate set of Koopman eigenfunctions and using them to transform and reconstruct the NN states fits naturally with the structure and use case of autoencoders. The paper *Deep learning for universal linear embeddings of nonlinear dynamics*, Bethany Lusch, J. Nathan Kutz, Steven L. Brunton has provided a methodology for such an autoencoder. The architecture of the autoencoder is shown in Figure 1.

The autoencoder is trained on three types of loss functions:

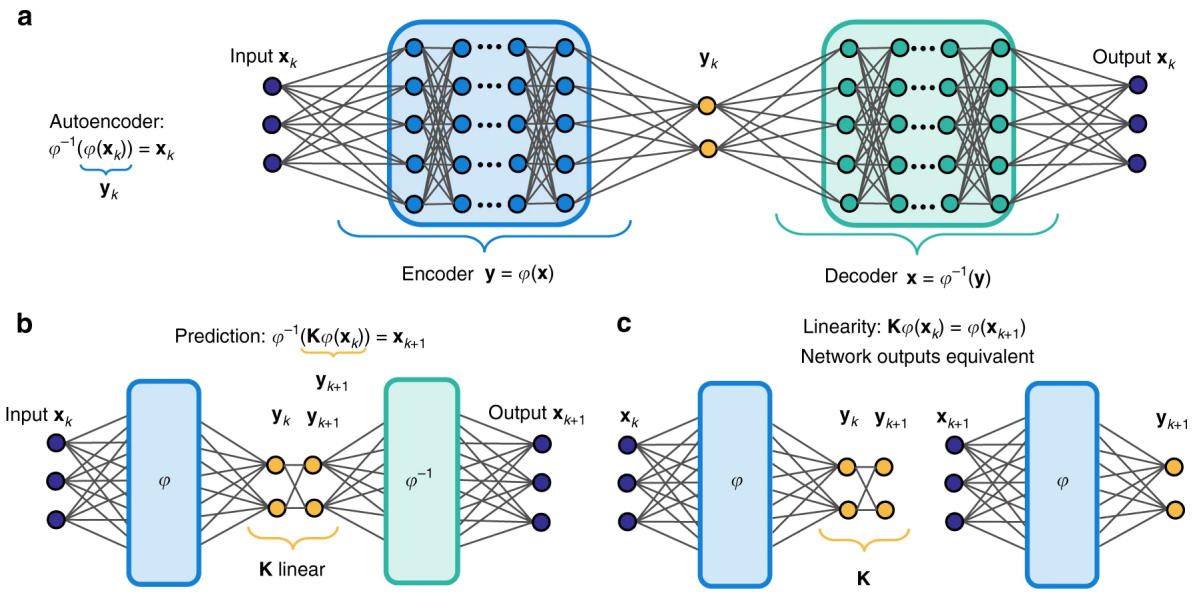



Figure 1: Diagram showing the architecture of the Koopman Autoencoder. (a) The autoencoder is trained on reconstruction accuracy, so as to identify $\varphi$ and $\varphi^{-1}$ which transforms NN states in $\mathcal{X}$ into coordinates in $\mathcal{Y}$ and the other way round. (b) The matrix $\mathbf{K}$ is added to advance coordinates in $\mathcal{Y}$, and the autoencoder is trained on state prediction accuracy to predict future NN states in $\mathcal{X}$. (c) Linear prediction accuracy is used as the loss function to identify the linear dynamics in $\mathcal{Y}$.

1. Reconstruction accuracy: $||\mathbf{x} - \varphi^{-1}(\varphi(\mathbf{x}))||$

   We seek to identify the coordinates $\mathbf{y} = \varphi(\mathbf{x})$ where the linear dynamics evolve in $\mathcal{Y}$, along with an inverse $\mathbf{x} = \varphi^{-1}(\mathbf{y})$ so that state $x$ can be recovered. In the autoencoder, $\varphi$ is the encoder and $\varphi^{-1}$ is the decoder.

2. Linear prediction accuracy: $||\varphi(\mathbf{x}_{k+m}) - \mathbf{K}^m(\varphi(\mathbf{x}_k))||$

   We learn the matrix $\mathbf{K}$ for predicting the linear dynamics in $\mathcal{Y}$, i.e. $\mathbf{y}_{k+m} = \mathbf{K}^m(\mathbf{y}_k)$.

3. State prediction accuracy: $||\mathbf{x}_{k+m} - \varphi^{-1}(\mathbf{K}^m(\varphi(\mathbf{x}_k)))||$

   Finally, we need $\varphi$, $\varphi^{-1}$ and $\mathbf{K}$ to be able to predict future states in the original non-linear dynamics.

where $||\cdot||$ is mean-squared error, averaged over dimension then number of samples.

The paper trained and tested the autoencoder on three continuous dynamical systems, which makes the dataset not directly applicable to this project. Nevertheless, the same methodology can be used to generate the dataset specific for this project.

# 2 Project Description

## 2.1 Core

Re-implement, train and evaluate a Koopman autoencoder to predict NN training trajectories. At least 2 NNs will be studied, including a NN differential equation solver and a MNIST NN. Both are fully-connected and contain only feed-forward connections. For each of these NNs, the process will likely be:

- Train up the NN with different weight initialisations. Collect the state (i.e. weights/biases) at every training time step. Each data point corresponds to the set of NN states evolved from one particular set of initial weights. This forms a dataset of training trajectories starting from various sets of initial weights.
- Subdivide into training, validation and test folds for the autoencoder. The exact distribution is yet to be decided.
- The autoencoder is considered successful if the training trajectory predicted by the autoencoder matches the actual trajectory up to a certain accuracy. In particular, information loss between the original and predicted training trajectories can be used as an evalution metric.

Since applied Koopman analysis has also shown to be versatile across different architectures [*Optimizing Neural Networks via Koopman Operator Theory*, Akshunna S. Dogra, William T Redman], we could train and evaluate the autoencoder on more complicated NNs (having more neurons, more deep layers, recurrent connections. etc.) should time allows, so as to better demonstrate the power of KOT in NN training dynamics prediction.

### 2.2 Extension

If the implemented autoencoder is sufficiently effective, we can leverage on the potential of the linear framework for extensions such as:

- **Control** - Control the training processes of NNs based on the insights offered in the linear framework; and
- **Interpretation** - Interpret the eigenfunctions/eigenvalues and the transformed training trajectories with tools from classical linear dynamics analysis and neural network interpretation literature

Another direction could be trying to seek further improvements of the autoencoder on top of the one proposed in the *Lusch et al.* paper.

## 3 Starting Point

Very brief background on linear algebra, given by the Math B course in Natural Sciences Tripos in my first year. For deep learning, my only experience is from my second year group project on Computer Vision, using Convolutional Neural Networks (CNN). Beside these, I have no prior experience with the theories and technology in this project.

## 4 Timetable

- 1 Nov - 14 Nov: **Background reading**

  Data-driven KOT; Relations of KOT to NN training; Autoencoders
- 15 Nov - 28 Nov: **Background reading**

  Linear dynamics analysis; NN interpretations
- 29 Nov - 12 Dec: **Implementation**

  Implement and evaluate the autoencoder

- 12 Dec - 18 Jan (Start of Lent): **Implementation & Evaluation**
  Write up the core section of the dissertation & the progress report
- 18 Jan (Start of Lent) - 28 Feb: **Extensions**
- 28 Feb - 26 Apr (Start of Easter): **Append extension findings into the dissertation**
- 26 Apr (Start of Easter) - 13 May (Dissertation Deadline): **Buffer**

# 5 Resource Declaration

I will be using the GPUs from the Computer Science and Technology Department to train up the NNs in my project. My supervisor, Prof. Pietro Lio, will help with securing access for these.

Other than the above, I will be using my personal computer for development and research. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.