Learning Deep Feature Spaces for Nonparametric Inference

Lernen von Merkmalsräumen für nichtparametrische Inferenz mittels Deep Learning
Bachelor-Thesis von Philipp Becker aus Groß-Umstadt
Tag der Einreichung:

1. Gutachten: Prof. Dr. techn. Gerhard Neumann
2. Gutachten: M.Sc. Gregor Gebhardt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Learning Deep Feature Spaces for Nonparametric Inference
Lernen von Merkmalsräumen für nichtparametrische Inferenz mittels Deep Learning

Vorgelegte Bachelor-Thesis von Philipp Becker aus Groß-Umstadt

1. Gutachten: Prof. Dr. techn. Gerhard Neumann
2. Gutachten: M.Sc. Gregor Gebhardt

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 18. Oktober 2016

_____

(Philipp Becker)

Abstract

An important part of control and planning in robotics is the ability to infer the next system states given previous system states and the actions applied. In the case were the true system state is not accessible and can only be perceived through observations, this is done using models describing the system dynamics and which observations are emitted by the system in a certain state. Formally, these models can be expressed as functions. Deriving these models analytically is almost impossible for real world systems. Another way to obtain them is Machine Learning, a field which provides various tools to learn functions based solely on example data.

Two recent approaches using machine learning tools to perform inference for dynamical systems are the Kernel Kalman Filter (KKF) [Gebhardt et al., 2016] and Embed to Control (E2C) [Watter et al., 2015]. While the former employs kernel methods and nonparametric representations of probability distributions to solve the task, the latter uses Deep Learning methods to perform inference.

We propose the Deep Nonparametric Kalman Filter (DNKF), an approach performing nonparametric inference in a Deep Learning setting by combining ideas from E2C and the KKF. Further, we evaluate our approach and compare it with E2C and the KKF.

Zusammenfassung

Ein wichtiger Aspekt von Planung und Steuerung in der Robotik ist die Fähigkeit die nächsten Systemzustände anhand von vorherigen Systemzuständen und angewendeten Aktionen vorherzusagen. In dem Fall in dem der echte Systemzustand unbekannt ist und nur durch Beobachtungen wahrgenommen werden kann, bedient man sich dazu Modellen welche die Beobachtungen die in einem bestimmten Zustand gemacht werden sowie die Systemdynamik beschreiben. Formal können solche Modelle als Funktionen aufgefasst werden. Für reale Systeme ist es fast unmöglich diese Modelle analytisch herzuleiten. Eine Alternative bietet das Machine Learning welches zahlreiche Methoden bereitstellt um Funktionen einzig anhand von Beispieldaten zu erlernen.

Der Kernel Kalman Filter (KKF) [Gebhardt et al., 2016] und Embed to Control (E2C) [Watter et al., 2015] sind zwei aktuelle Ansätze welche sich Machine Learning Methoden bedienen um Inferenz in dynamischen Systemen durchzuführen. Während erstgenannter Kernel Methoden und nichtparametrische Repräsentationen von Wahrscheinlichkeitsverteilungen nutzt, setzt letzterer Deep Learning Methoden zur Inferenz ein.

Wir schlagen den Deep Nonparametric Kalman Filter (DNKF) vor, einen Ansatz welcher nicht parametrische Inferenz mittels Deep Learning Methoden ermöglicht indem er Ideen von E2C und dem KKF verbindet. Des weiteren evaluieren wir unseren Ansatz und vergleichen ihn mit E2C und dem KKF .

Acknowledgments

I thank my supervisors Gregor Gebhardt and Gerhard Neumann for their support, advice and patience.

Furthermore, I thank Jan Hohgräfe and Simon Ramstedt for proofreading this thesis, providing feedback and ideas for improvement.

Last, I would like to thank my family, especially my parents, for supporting me during my studies in the last years.

Calculations for this research were conducted on the Lichtenberg high performance computer of the TU Darmstadt.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations

List of Abbreviations

Notation	Description
BPTT	backpropagation trough time
DNKF	Deep Nonparametric Kalman Filter
E2C	Embed to Control
KKF	Kernel Kalman Filter
MSE	Mean Squared Error
NN	Neural Network
RKHS	Reproducing Kernel Hilbert Space
RKL	Recurrent Kalman Layer
RNN	Recurrent Neural Network
VAE	Variational Autoencoder

1 Introduction

One of the main goals of robotics is to perform control and planning for robots living in and interacting with real world environments. The difficulty of that task is determined by the nature of the environment itself. When building an assembly line, e.g., in a car factory, engineers try to build an environment that is as deterministic as possible. Furthermore, they can use sufficiently many sensors in order to enable the robot of perceiving every information necessary for the given task, making the environment fully observable.

However, often it is not possible to build a deterministic and fully observable environment for the robot, which makes the problem of control and planning much harder. Imagine for example a vacuum robot that has the task to clean a house. For the robot it is not possible to perceive anything outside the room it is currently in. Furthermore, the environment is highly nondeterministic and constantly changed by factors outside of the robots influence.

An important aspect of control and planning is the ability to infer future system states. If the true state of the system is known, it is sufficient to model the system dynamics using a transition model. This model predicts future system states given the current state. Depending on the system at hand the future states may also depend on actions applied to it. However, as described above, the robot usually has no access to the true state of the environment, or even its own state, and can only partially perceive it through noisy observations. In those cases an observation model, i.e., a model describing which observations are emitted by the system in a certain state, is needed.

It is almost impossible to obtain accurate dynamics models for dynamical systems, due to factors such as friction and material fatigue, causing inherent model errors. Those model errors, together with partial information and the stochasticity of the environment, make exact predictions impossible, hence only state estimates are available for the given task. In order to account for that and additionally providing information about the uncertainty of the estimates, it is beneficial to model the dynamics using probability distributions.

Another way to obtain the models is Machine Learning, a branch of Artificial Intelligence, concerned with finding models and solving a given task, only using data. This is achieved by using a parameterized model, whose parameters are optimized using a function assessing the current performance of the model given the data.

In Machine Learning linear models are desirable since those are particularly easy to work with and to optimize. However, the system dynamics and the observation generation process are often highly nonlinear, even for very simple systems. Hence, it is not reasonable to approximate them using a linear model. One common approach to deal with this is to learn the models not directly in the observation or state space but in some latent space in which the dynamics and observation generation process are approximately linear.

With Embed to Control (E2C) a model for parametric inference using Machine Learning methods was introduced by Watter et al. [2015]. They map the observations into a low dimensional latent space, using a Variational Autoencoder (VAE). In that space a locally linear transition model is learned simultaneously to the VAE. Additionally, another constraint is added to the optimized cost function in order to achieve a dynamics model corresponding well with the real system dynamics.

Both the VAE and the transition model are learned using Deep Neural Networks (NNs), powerful Machine Learning models inspired by the brain. Those Deep NNs are capable of solving complex tasks and learning useful abstractions. Recently, they have been employed to solve a variety of tasks that were considered to be unsolvable for computers in the near future, like winning the game of Go against the best human players [Silver et al., 2016]. This has been possible due to new theoretical insights

and progress in computer technology making very big data sets available and making it feasible to train models with a large amount of parameters on these data sets.

However, parametric representations of probability distribution are inherently restricted in their ability to represent complex statistical features such as multimodality, skewness and rich dependency structures. To cope with this, several approaches providing nonparametric representations of probability distributions were introduced. One of them, a framework surveyed by [Song et al., 2013], is based on the idea of embedding probability distributions into very high, potentially infinite, dimensional Reproducing Kernel Hilbert Spaces (RKHSs).

Based on that framework, Gebhardt et al. [2016] introduced the Kernel Kalman Filter (KKF). Like the classical Kalman Filter [Kálmán, 1960] it can be employed to form state estimates for dynamical systems using sequences of observations, together with an observation and a transition model.

In this work we will combine the ideas of nonparamtetric inference using Kalman Filters and finding latent representations of the state space in which the transition and observation model become linear. Further, we will make use of Deep Learning methods, exploiting their capability of dealing with large data sets.

After introducing all necessary preliminaries, we will propose our own approach, the Deep Nonparametric Kalman Filter (DNKF) and finally evaluate its performance with experiments in simulation using visual observations of a pendulum and the endeffector positions of a quad link, i.e., a pendulum with four links.

2 Preliminaries

2.1 Formalization of the Problem

We assume a time discrete system whose system dynamics are modelled by

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t + \boldsymbol{\epsilon}_t,$$

with system matrix $\mathbf{A}$, control matrix $\mathbf{B}$, transition noise $\boldsymbol{\epsilon}_t$ and applied action $\mathbf{u}_t$. This is also referred to as transition model. Further, it is assumed that there is no access to the true state $\mathbf{x}_t$ but only to some observation (measurement) $\mathbf{o}_t$ of the system. The process generating these observations is also linear, i.e.,

$$\mathbf{o}_t = \mathbf{C}\mathbf{x}_t + \boldsymbol{\gamma}_t,$$

with an observation matrix $\mathbf{C}$ and observation noise $\boldsymbol{\gamma}_t$. However, we have full access to the actions $\mathbf{u}_t$ applied to the system.

Often it is useful to redefine those models as probability distributions. In this case the transition model is given by

$$\mathbf{x}_{t+1} \sim P_t\left(X|\mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t\right)$$

and the observation model is given by

$$\mathbf{o}_t \sim P_o\left(O|\mathbf{C}\mathbf{x}_t\right).$$

2.2 Kalman Filter and Embeddings

Introduced by and named after Kálmán [1960], the Kalman Filter is used to obtain state estimates of a system whose true state is hidden. It is an iterative algorithm working in two steps. First, it uses an observation together with the observation model to update its estimate. Second, it employs the transition model to predict an estimate of the next state given the executed action.

The Kalman Filter assumes the transition dynamics as well as the observation model to be known. Further, it assumes both, the transition noise and the observation noise to be Gaussian distributed with zero mean and known covariances denoted by $\mathbf{H}_t$ for the transitions and $\mathbf{R}_t$ for the observations. If all those assumptions are satisfied the estimates produced by the Kalman Filter are optimal in the least squares sense.

During the update step an estimate, a-posterior to the observation, of the current state is inferred. This is done by computing a weighted average of the Kalman Filters a-priori estimate and the current observation. The weights are represented by the Kalman gain $\mathbf{Q}_t$ and depend on the uncertainty of both the a-prior estimate and the observation. A higher uncertainty about one of the quantities yields smaller influence on the a-posteriori estimate. The formulas of the update step are given by

Measurement Residual:	$\mathbf{y}_t = \mathbf{o}_t - \mathbf{C}_t\mathbf{x}_t^-$
Resuidual Covariance:	$\mathbf{S}_t = \mathbf{C}_t\mathbf{P}_t^-\mathbf{C}_t^T + \mathbf{R}_t$
Kalman Gain:	$\mathbf{Q}_t = \mathbf{P}_t^-\mathbf{C}_t^T\mathbf{S}_t^{-1}$
Mean Update:	$\mathbf{x}_t^+ = \mathbf{x}_t^- + \mathbf{Q}_t\mathbf{y}_t$
Covariance Update:	$\mathbf{P}_t^+ = (\mathbf{I} - \mathbf{Q}_t\mathbf{C}_t)\,\mathbf{P}_t^-,$

where $\mathbf{I}$ denotes the identity matrix. While $\mathbf{x}_t^-$ and $\mathbf{P}_t^-$ denote the mean and covariance of the estimated state a-priori to observation $\mathbf{o}_t$, the mean and covariance of the estimated state a-posteriori to observation $\mathbf{o}_t$ are denoted by $\mathbf{x}_t^+$ and $\mathbf{P}_t^+$ respectively.

Next, the posterior estimate together with the transition model is used to compute the prior estimate of the next state.

$$\begin{aligned} &\text{Next Mean:} && \mathbf{x}_{t+1}^- = \mathbf{A}\mathbf{x}_t^+ + \mathbf{B}\mathbf{u}_t \\ &\text{Next Covariance:} && \mathbf{P}_{t+1}^- = \mathbf{A}\mathbf{P}_t^+\mathbf{A}^T + \mathbf{H}_t. \end{aligned}$$

If for the current time step no observation is available the update step is omitted and the next prior directly inferred from the current prior.

2.2.1 Kernel Embedding of Distributions

There are several methods to represent probability distributions in a nonparamatric manner. One of them, a framework surveyed by Song et al. [2013], is based on the embedding of distributions into a Reproducing Kernel Hilbert Space (RKHS). Embedding a probability distribution into an RKHS means representing the distribution as an element in the RKHS. Such embeddings were derived for marginal, joint and conditional distributions as well as kernel versions of the sum, product and Bayes rule.

A Hilbert space is a space that has an inner (scalar) product $\langle\cdot,\cdot\rangle$ defined and is complete with respect to the norm induced by that inner product. Let $\mathcal{F}$ be a Hilbert space of functions $f : \mathcal{X} \mapsto \mathbb{R}$. If for each $x \in \mathcal{X}$ there exists an element $k_x(\cdot) \in \mathcal{F}$ satisfying the reproducing property, i.e., $\langle f(\cdot), k_x(\cdot)\rangle_{\mathcal{F}} = f(x)$, then $\mathcal{F}$ is an RKHS and $k(x,x') = \langle k_x(\cdot), k_{x'}(\cdot)\rangle_{\mathcal{F}}$ is the corresponding reproducing kernel. This kernel can now be used to implicitly map values $x, x' \in X$ to some feature space using a mapping $\phi : \mathcal{X} \mapsto \mathcal{F}$ and evaluating a inner product in that space by computing $k(\phi(x), \phi(x')) = \langle\phi(x), \phi(x')\rangle_{\mathcal{F}}$.

In order to embed a marginal distribution $P(X)$ over a random variable X the so called mean map [Smola et al., 2007] is used,

$$\boldsymbol{\mu}_X = \mathbb{E}_{P(X)}[\phi(X)] = \int_{\mathcal{F}} \phi(x)p(x)dx.$$

Often one has no access to the distribution $P(X)$ but only to a finite amount of samples $x_i \sim P(X)$. In such cases a finite sample estimator is used. The estimator of the mean map is given by

$$\hat{\boldsymbol{\mu}}_X = \frac{1}{m}\sum_{i=1}^{m}\phi(\mathbf{x}_i).$$

Similarly to marginal distributions, joint distributions can be embedded into a tensor product (outer product) RKHS $\mathcal{F} \otimes \mathcal{F}$ [Smola et al., 2007]. The embedding of a joint distribution of two random variables X and Y is defined as

$$\mathcal{C}_{XY} = \mathbb{E}_{P(X,Y)}[\phi(X) \otimes \phi(Y)] = \int_{\mathcal{F}\times\mathcal{F}} \phi(x) \otimes \phi(y)p(x,y)dxdy.$$

Again a finite sample estimator exists, which is given by

$$\hat{\mathcal{C}}_{XY} = \frac{1}{m}\sum_{i=1}^{m}\phi(x_i) \otimes \phi(y_i).$$

As a next step, conditional distributions of the form $P(X|Y)$ are embedded into the RKHS [Song et al., 2009]

$$\boldsymbol{\mu}_{X|y} = \mathbb{E}_{P(X|Y)}[\phi(X)] = \int_{\mathcal{F}} \phi(x)p(x|y)dx.$$

Note that this is not a single element in the RKHS but a family of elements with one member for each possible value y of Y. To condition X on a particular y and obtain a single RKHS element the conditional embedding operator $\mathcal{C}_{X|Y}$ is used. This operator takes y as input and outputs the corresponding member of the family. Formally it satisfies

$$\boldsymbol{\mu}_{X|y} = \mathcal{C}_{X|Y}\phi(y).$$

Song et al. [2009] derived this conditional embedding operator as

$$\mathcal{C}_{X|Y} = \mathcal{C}_{XY}\mathcal{C}_{YY}^{-1}, \quad \text{hence} \quad \boldsymbol{\mu}_{X|y} = \mathcal{C}_{XY}\mathcal{C}_{YY}^{-1}\phi(y).$$

Finally the kernel sum rule (marginalization) is introduced [Fukumizu et al., 2013]. A marginal distribution $Q(X)$ can be obtained from a conditional distribution $P(X|Y)$ and a prior $\pi(Y)$ on the conditioning variable Y via

$$Q(X) = \int_{\mathcal{F}} p(x|y)\pi(y)dy.$$

Embedding $Q(X)$ to the RKHS yields

$$\boldsymbol{\mu}_X = \mathbb{E}_{Q(X)}[\phi(X)] = \mathbb{E}_{\pi(Y)}\mathbb{E}_{P(X|Y)}[\phi(X)] = \mathbb{E}_{\pi(Y)}[\mathcal{C}_{X|Y}\phi(Y)] = \mathcal{C}_{X|Y}\mathbb{E}_{\pi(Y)}[\phi(Y)] = \mathcal{C}_{X|Y}\boldsymbol{\mu}_Y,$$

where $\boldsymbol{\mu}_Y$ denotes the mean map of the prior $\pi(Y)$ and $\mathcal{C}_{X|Y}$ can be moved outside of the expected value $\mathbb{E}_{\pi(Y)}$ since both are linear operators.

The kernel product rule as well as the kernel Bayes rule are omitted here since they are not needed for further derivations.

2.2.2 Kernel Kalman Filter

Based on this framework, Gebhardt et al. [2016] derived the Kernel Kalman Filter (KKF), a nonparametric version of the Kalman Filter. In order to do this they derive kernel versions of the update and the prediction rule.

The current state estimate is modelled with embedded marginal distributions $\boldsymbol{\mu}_{X,t}^-$ and $\boldsymbol{\mu}_{X,t}^+$. Again $\boldsymbol{\mu}_{X,t}^-$ denotes the estimate a-priori to the observation $\mathbf{o}_t$ and $\boldsymbol{\mu}_{X,t}^+$ denotes the observation a-posteriori to $\mathbf{o}_t$. Both, the observation model and the transition model are embedded conditional distributions with conditional operators $\mathcal{T}$ for the transition and $\mathcal{C}_{O|X}$ for the observation. Further, the current observation $\mathbf{o}_t$ is embedded into an RKHS and the embedded observation denoted by $\phi(\mathbf{o}_t)$. Note that the state estimates and observations in general are not embedded into the same RKHS and $\mathcal{C}_{O|X}$ is used to map a state estimate into the observation RKHS.

Corresponding to the update step in the classical Kalman Filter they introduce the kernel Kalman rule. Following a recursive least squares approach they chose the posterior estimate $\mu_{X,t}^+$ such that it minimizes the squared distance between the true observation, embedded into the RKHS, $\phi(\mathbf{o}_t)$, and the inferred observation, $\mathcal{C}_{O|X}\mu_{X,t}^+$. This yields the following objective

$$\min_{\mu_{\mathbf{x}}^+} \sum_t \left(\phi(\mathbf{o}_t) - \mathcal{C}_{O|X}\boldsymbol{\mu}_x^+\right)^T \mathcal{R}^{-1} \left(\phi(\mathbf{o}_t) - \mathcal{C}_{O|X}\boldsymbol{\mu}_x^+\right),$$

with some metric $\mathcal{R}$. With the recursive least squares solution the following rule is obtained

$$\boldsymbol{\mu}_{\mathbf{x},t}^+ = \boldsymbol{\mu}_{\mathbf{x},t}^- + \mathcal{Q}_t \left(\phi(\mathbf{o}_t) - \mathcal{C}_{O|X}\boldsymbol{\mu}_{\mathbf{x},t}^-\right),$$

where $\mathcal{Q}_t$ corresponds to the Kalman gain in the Kalman Filter and is hence referred to as kernel Kalman gain operator.

The RKHS equivalent to the prediction step is obtained by using the kernel sum rule. With the embedded transition model $\mathcal{T}$ the next prior state estimate $\mu^-_{X,t+1}$ is obtained from the current posterior state estimate $\mu^+_{X,t}$ by marginalization.

$$\mu^-_{X,t+1} = \mathcal{T}\mu^+_{X,t}.$$

Together those two rules yield the KKF. Like with the classical Kalman Filter the update step is omitted if no observation is present.

2.2.3 Practical Aspects

In practice $\mathcal{Q}_t$, $\mathcal{C}_{O|X}$ and $\mathcal{T}$ need to be learned from finite data sets. Due to the high, possible infinite, dimensionality of the RKHS used, it is computationally intractable to work directly with them. However, since all necessary calculations can be expressed using inner products the kernel can be used to work implicitly in the RKHS, making computation tractable.

Employing the kernel makes the usage of the Gram matrix necessary. This matrix contains the evaluated kernel for every pair of the n data points in the set, hence it is of size $n \times n$. In particular a matrix of the same size, computed from the Gram matrix, needs to be inverted in order to estimate the kernel Kalman gain operator. This yields a runtime growing cubically in the number of data points, making it hard to train the KKF for large data sets.

Gebhardt et al. [2016] alleviated this drawback by introducing the Subspace Kernel Kalman Filter. However, it remains hard to train the model on a data sets with sufficiently many samples to cover the complete state space for systems like the quad link.

2.3 Deep Learning

Deep Learning is a subfield of Machine Learning employing Neural Networks (NNs). Recently, Deep Learning methods have been used to develop algorithms that achieve state of the art and sometimes even human like performance in a variety of tasks like playing the game of Go [Silver et al., 2016], image recognition [Szegedy et al., 2015], image captioning [Vinyals et al., 2015] and natural language processing [Goldberg, 2015].

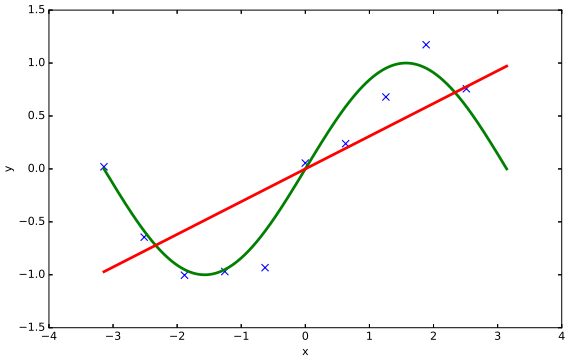
Machine Learning in general tries to find patterns and underlying structures in data and uses that knowledge to build a model capable of solving a specific task. Such models are represented by parameterized functions and the parameters are optimized by another function assessing how the model is currently performing. This process is called training or fitting the model.

Depending on the task at hand, Machine Learning approaches are categorized into supervised, unsupervised and reinforcement learning. Supervised learning models are trained with a set of the form $\mathcal{D} = \{X, Y\} = \{(\mathbf{x}_i, \mathbf{y}_i) | i = 1, \ldots, n\}$ with inputs $\mathbf{x}_i$ and corresponding desired outputs $\mathbf{y}_i$. For unsupervised models we only have a set of inputs $X = \{(\mathbf{x}_i | i = 1, \ldots, n\}$.

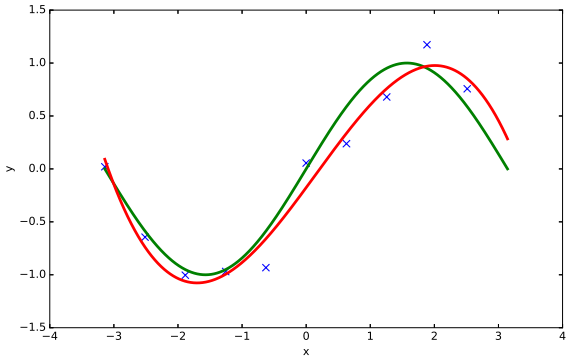
The performance of a Machine Learning model is significantly influenced by the way the data is represented. For an example of the importance of a good representation see Figure 2.1. Usually a good representation is obtained by extracting features from the data and then map each data point to the feature space. Finding good features is often a cumbersome and time consuming task that requires severe domain knowledge and can only be performed by (systematic) trial and error.

The main reason for the recent success of Deep Learning is that the task of finding suitable features is taken away from the developer. Instead, the features are learned as part of the model directly from the data during training.

As stated above Deep Learning models are going back to NNs, introduced by Rosenblatt [1958] and back then referred to as Perceptrons.



(a) Linear function trained on raw data, $\phi(x) = x$



(b) Linear function trained on data transformed to polynomial (degree 3) feature space, $\phi(x) = (x^3, x^2, x, 1)^T$

Figure 2.1: The model is tasked to learn the sin function (green) based on noisy samples (blue) using a function linear in the parameters θ, i.e., $f(x) = \theta^T \phi(x)$. The results (red) obtained with a suitable feature map **(b)** are clearly better than the results obtained by just feeding in the raw data **(a)**.

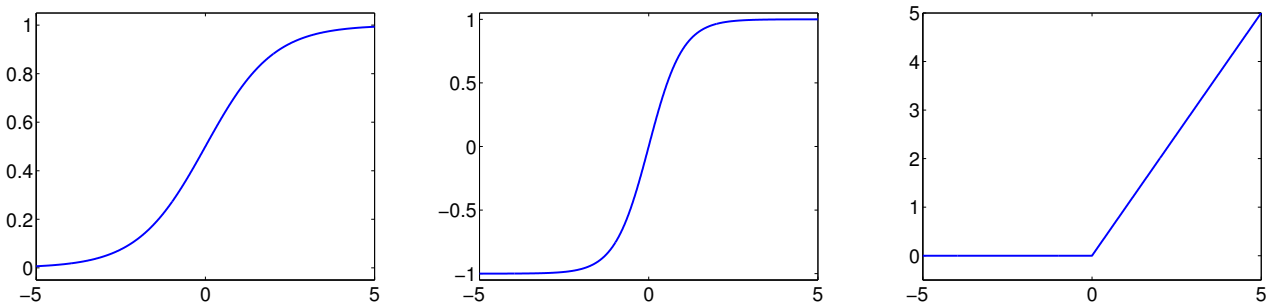
2.3.1 Neural Networks

Originally, Neural Networks (NNs) were inspired by the brain in which neurons propagate information from one to another in order to represent abstract situations and make complex decisions. In contrast to the brain, where neurons can form arbitrary connections the neurons in a NN are organized in layers. There are no connections between the neurons in a single layer and information, represented as real numbers, flows from one layer to the next in a specific direction. Each neuron accumulates the information received from the neurons in a previous layer, transforms it, and then propagates it to the next layer. Classically, each neuron is connected to all the neurons in the previous and next layer, this is called densely connected. However, modern Deep Learning approaches use a variety of specialized layers, such as convolutional or recurrent layers for which this is not always the case.

Often the inputs themselves as well the outputs are considered to be layers, referred to as input and output layer. All layers in between are called hidden layers. The amount of hidden layers, as well the width, i.e., the number of neurons of each layer are hyperparameters of the model and have significant influence on its performance. The term Deep Learning refers to a recent trend of using multiple hidden layers. While in classical works rarely more than one hidden layer was used, Szegedy et al. [2015] for example used 22 layers.

More formally a NN is a nonlinear, parameterized function of special form. Each neuron is a scalar product of its input and a weight vector of trainable parameters. The result of the scalar product is then transformed using a nonlinear activation function. While this could be any function there are several very commonly used ones, visualized in Figure 2.2. A sketch and the formulas for a very simple NN can be found in Figure 2.3

In practice it is more efficient to compute the values for a whole layer with a single matrix vector multiplication by concatenating the weight vectors into a matrix. Further, often an additional neuron is added to each layer. This neuron outputs a constant 1 and serves as a bias, allowing for constant offsets in the representation learned by the following layer. Note that, in this work, such a bias neuron is assumed to be present in every layer unless stated otherwise.

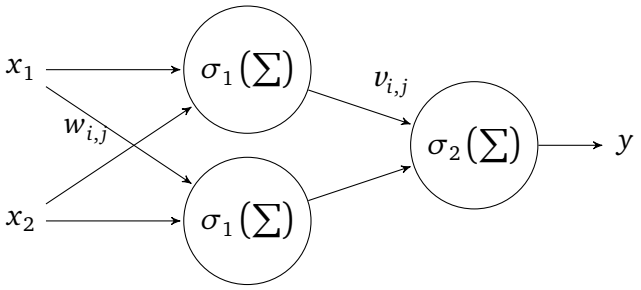


(a) Sigmoid:
$$\sigma(x) = \frac{1}{1+\exp -x}$$

(b) Hyperbolic Tangent:
$$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$

(c) Rectified Linear unit (ReLU):
$$\text{relu}(x) = \max(0, x)$$

Figure 2.2: Examples of common activation functions. While in early works in the field mostly the sigmoid and hyperbolic tangent were used nowadays the ReLU is the most common activation functions since it has some nice properties regarding optimization and does not saturate.



(a) Sketch of the NN as actual network of neurons. Those visual metaphors are are often used to explain NNs. The weights $w_{i,j}$ making up the weight matrix W. Similar, the weights $v_{i,j}$ make up a weight matrix $\mathbf{V}$.

$$\mathbf{h} = \sigma_1(\mathbf{W}\mathbf{x})$$
$$y = \sigma_2(\mathbf{V}\mathbf{h})$$

(b) Formulas computing the values of each layer, where σ_1 and σ_2 represent the nonlinearities of each layer and the weight matrices $\mathbf{W}$ and $\mathbf{V}$ are the trainable parameters.

Figure 2.3: Example of a very simple NN with one hidden layer of width two.

2.3.2 Training Deep Neural Networks

As stated above, in order to train a parameterized model $m_\theta : \mathcal{X} \mapsto \mathcal{Y}$ on a data set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i) | i = 1, \ldots, n\}$, one optimizes a function evaluating its performance. In the context of Machine Learning such functions are often referred to as loss functions[1]. For the supervised methods discussed here, this function is usually chosen to measure the distance between the output predicted by the model and the desired output given by the dataset. The most prominent example of such a loss function is the Mean Squared Error (MSE), which, for a single sample $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$, is given by

$$\text{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\sum_{j=1}^{N}(y_j - \hat{y}_j)^2 = \| \mathbf{y} - \hat{\mathbf{y}} \|_2^2,$$

where $\hat{\mathbf{y}} = m_\theta(\mathbf{x})$ and N denotes the dimension of $\mathcal{Y}$.

[1] The terms loss function, cost function and error function are used equivalently.

Often it is assumed that the output data $\mathbf{y}_i$ was produced by a probability distribution P and the model is tasked with learning a distribution Q that is as close to P as possible. In such cases measures justified by probability theory are used. Examples of such measures are the Cross Entropy (CE)

$$\mathrm{H}(P,Q) = \mathbb{E}_P[-\log(Q)] = -\int_X p(x)\log q(x)dx$$

and the Kullback-Leibler (KL) Divergence

$$\mathrm{KL}(P \parallel Q) = \int_X p(x)\log\frac{p(x)}{q(x)}dx.$$

Note that neither is a metric in a mathematical sense since they are not symmetric. Usually P and Q are not explicitly known and only samples, in form of desired values in the data set or as outputs of the model

$$\mathbf{y}_i \sim P \quad \text{and} \quad \hat{\mathbf{y}}_i \sim Q,$$

are available. In this case

$$H(\mathbf{y},\hat{\mathbf{y}}) = \sum_{j=1}^{N} y_j \log(\hat{y}_j) \quad \text{and} \quad KL(\mathbf{y} \parallel \hat{\mathbf{y}}) = \sum_{i=j}^{N} y_j \log\frac{y_j}{\hat{y}_j}$$

are used as to approximate the Cross Entropy and the KL-Divergence based on a single sample. Since usually not single samples but data sets are considered, the errors are summed or averaged over the whole data set, both yielding identical results regarding the optimization.

A common way to optimize functions is gradient descent, an iterative algorithm following the negative gradient until a local minimum is reached. For a loss function $L(\theta,\mathcal{D}) \mapsto \mathbb{R}$ this looks as follows

$$\theta_{t+1} = \theta_t - \alpha_t \frac{\partial L(\theta,\mathcal{D})}{\partial\theta},$$

where α_t denotes the learning rate, a factor scaling the gradient to ensure convergence. For large data sets $\mathcal{D}$ it is infeasible to calculate the gradient for the whole set and sufficient approximations can be obtained with only a subset. Hence a method called stochastic gradient descent is used. For this $\mathcal{D}$ is split into several disjoint subsets $d_i \subset \mathcal{D}$, referred to as (mini)batches, and the parameters are updated after each batch. Optimizing once with each batch is referred to as a training epoch and the networks are usually trained for several epochs. The update is performed according to

$$\theta_{t+1} = \theta_t - \alpha_t \frac{\partial L(\theta,d_i)}{\partial\theta}.$$

If additional assumptions are made about α_t, namely

$$\sum_{t=1}^{\infty}\alpha_t > \infty \quad \text{and} \quad \sum_{t=1}^{\infty}\alpha_t^2 < \infty,$$

stochastic gradient descent provably converges to a local minimum. For practical use, a variety of more sophisticated algorithms for stochastic gradient descent exists such as Adam [Kingma and Ba, 2014]. It uses exponentially decaying averages over the gradients and their squares to compute adaptive learning rates, aiming at faster convergence and avoiding getting stuck in saddle points.

It remains to calculate the gradients of the model. For (Deep) NN this is usually done using the backpropagation algorithm [Rumelhart et al., 1986]. However, modern Deep Learning frameworks, as used for this work, provide automatic calculation of gradients employing a technique known as automatic differentiation. Since every computer program can be written as a sequence of elementary arithmetic operations a derivative of that program can be obtained with the derivatives of the elementary operations by repeated application of the chain rule for derivatives given by

$$\frac{dx}{dy} = \frac{dx}{dw}\frac{dw}{dy}.$$

This idea is in fact very similar to the backpropagation algorithm.

2.3.3 Recurrent Neural Networks and Backpropagation trough Time

Often it is not possible to infer the whole system state from a single observation. As an example, imagine a model that should learn to infer the velocity and acceleration of an object solely on visual data. That is obviously not possible based on a single image. This makes models capable of dealing with sequential data, i.e., data where the data points are not independent but each point may depend on previous points, necessary. As illustrated by the example above, in such cases it is in general not sufficient to look at each data point individually and the model has to learn the temporal dependencies between the single data points in order to fully understand the data. Thus, the model needs to be enabled to take previously seen data points into account in order to solve such problems.

The naive idea to deal with this would be to give the model all the information at once, e.g., by concatenating the elements of the sequence. This however has a couple of disadvantages. First, the concatenated input vector may become very high dimensional, since its dimensionality increases with the length of the input sequence. This yields a high number of model parameters and hence makes learning more difficult. Second, since the whole data sequence needs to be present before it can be given to the model it is not possible to perform any kind of online task. Third, it is often not guaranteed that all sequences have the same length and shorter sequences need to be padded which increases the amount of data without adding any additional information.

A more sophisticated approach which overcomes the mentioned issues are so called Recurrent Neural Networks (RNNs). Those are special NNs whose output at a certain time step t does not only depend on the current input $\mathbf{x}_t$ but also on an internal state $\mathbf{h}_{t-1}$. In the most simplest case a recurrent layer can be formulated as follows

$$\mathbf{h}_t = \sigma(\mathbf{W}_{in}\mathbf{x} + \mathbf{W}_h\mathbf{h}_{t-1}) \quad \text{and} \quad \mathbf{y}_t = \mathbf{h}_t,$$

with weight matrices $\mathbf{W}_{in}$ and $\mathbf{W}_h$, an arbitrary activation function σ and an output identical to the hidden state. However, learning long term dependencies with this model is difficult [Bengio et al., 1994], hence nowadays often more sophisticated recurrent models are used. Examples for such models are the Gated Recurrent Unit (GRU) [Cho et al., 2014] and the Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997].

In order to train such a network, a loss on the whole sequence needs to be defined. Usually this loss is just defined as the sum of errors for each data point. Using this error, the RNN can be trained with the standard backpropagation algorithm by a simple trick called unfolding. While unfolding the RNN, it is rewritten as a deep standard NN with one layer for each data point in the input sequence. The layer corresponding to timestep t produces two outputs, the internal state $\mathbf{h}_t$ and a predicted output $\mathbf{y}_t$ (note that, as in the example above, they may be the same) and takes two inputs, the internal state from the previous layer $\mathbf{h}_{t-1}$ as well as the actual input $\mathbf{x}_t$. This is visualized in Figure 2.4. Note however that the weights $\mathbf{W}_{in}$ and $\mathbf{W}_h$ are the same for all layers, this is known as weight sharing, a common trick in Deep Learning to reduce the number of parameters and make a model invariant to the exact position of a particular part of information. Unfolding an RNN and applying backpropagation is often referred to as backpropagation trough time (BPTT) [Werbos, 1990]. To account for the weight sharing the losses, and hence the gradients, are summed up over the elements of the entire sequence.

Since an update step can be performed only after the outputs for the whole sequence are computed, training an RNN becomes more costly with increasing sequence length. One approach to deal with that is splitting each sequence into parts of length k and training the RNN on each of this parts. However, this prevents the model from learning dependencies reaching further back in time than k steps. To cope with this the truncated BPTT algorithm was introduced [Williams and Peng, 1990]. The main idea of this algorithm is to process the whole sequence, one step after another, but run BPTT not only once at the end but every k steps. Since the internal state is not reset and exposed to the whole sequence until the current time step, the model is capable of learning dependencies reaching further back than k time steps.

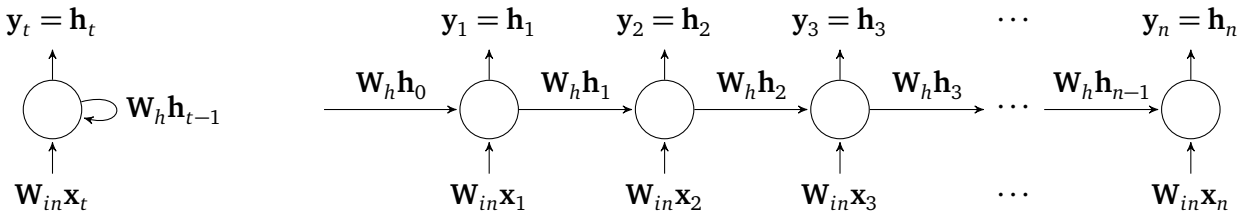


Figure 2.4: Left: The basic RNN introduced above. **Right:** By unfolding the RNN through time it was transformed to a NN with weights ($\mathbf{W}_{in}$ and $\mathbf{W}_h$) that are shared between all time steps. Unfolding an RNN is very similar to unwinding a loop in a program.

2.3.4 Autoencoder

As stated above many approaches rely on good representations. One common Machine Learning tool to find such representations is the autoencoder. Note that in the context of autoencoders the representation is often referred to as encoding. The goal of an autoencoder is to copy the input to the output. Formally, this is described by two parameterized functions (more specific NNs), an encoder $f_\theta : \mathcal{X} \mapsto \mathcal{Z}$ and a decoder $g_\theta : \mathcal{Z} \mapsto \mathcal{X}$, where $\mathcal{X}$ denotes the input space while $\mathcal{Z}$ denotes the space the representations live in, also referred to as latent space. The goal now is to find θ such that

$$\mathbf{x} = g_\theta(\mathbf{z}), \quad \text{with} \quad \mathbf{z} = f_\theta(\mathbf{x}).$$

While this technically is an unsupervised learning problem it can be solved by taking the inputs as both inputs and desired outputs and train the model by optimizing one of the supervised learning cost functions introduced above.

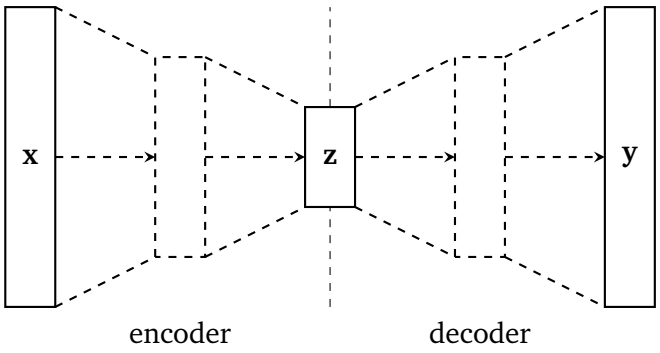


Figure 2.5: Schematic view of an undercomplete autoencoder. The output $\mathbf{y}$ should be a reconstruction of the input $\mathbf{x}$. The goal is to find a lower dimensional representation (encoding) $\mathbf{z}$ containing maximal information about $\mathbf{x}$.

However, in order to ensure that the learned latent representation is actually useful, the autoencoder needs to be further constrained. The most common method of such a constraint is to chose $\mathcal{Z}$ such that its is of a lower dimensionality than $\mathcal{X}$. This forces the model to discard some of the information present in the data and to focus on the most significant features of the data which are then contained in $\mathbf{z}$. Such an autoencoder is referred to as undercomplete.

2.4 Embed to Control (E2C)

Embed to Control (E2C) is an approach for inference in dynamical systems recently proposed by Watter et al. [2015]. Their idea is to find a low dimensional latent representation from high dimensional observation data using a Variational Autoencoder (VAE). Subsequently, this latent representation is used to perform optimal control. In order for that to work, the latent space dynamics need to correspond well with the actual observed dynamics, which is achieved by an additional constraining term in the cost function.

2.4.1 Variational Autoencoder

The Variational Autoencoder (VAE) was introduced by Kingma and Welling [2013]. The main difference to a standard autoencoder is that the VAE represents elements of the latent space not as points in that space but rather as parameterized distributions over it. In order to achieve that they assume that each $\mathbf{x}_i$ is generated by a random process, in which first a latent variable $\mathbf{z}_i$ is sampled from a prior $p_\theta(\mathbf{z})$ and then $\mathbf{x}_i$ is sampled from some conditional likelihood $p_\theta(\mathbf{x}|\mathbf{z})$. In order to encode some $\mathbf{x}$ to its latent representation the posterior distribution $p_\theta(\mathbf{z}|\mathbf{x})$ is needed. However computation of this posterior is often intractable since neither the true parameters θ nor the values of the latent variables $\mathbf{z}_i$ are known. Thus it has to be approximated using another distribution $q_\phi(\mathbf{z}|\mathbf{x})$. To obtain such a distribution that is close to the true posterior the variational lower bound is maximized.

For a single $\mathbf{x}_i$ the marginal log likelihood can be rewritten as

$$\begin{aligned}\log(P_\theta(\mathbf{x}_i)) &= KL(q_\phi(\mathbf{z}|\mathbf{x}_i)||p_\theta(\mathbf{z}|\mathbf{x}_i) - KL(q_\phi(\mathbf{z}|\mathbf{x}_i)||p_\theta(\mathbf{z})) + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}_i)}[\log p_\theta(\mathbf{x}_i|\mathbf{z})] \\ &= KL(q_\phi(\mathbf{z}|\mathbf{x}_i)||p_\theta(\mathbf{z}|\mathbf{x}_i)) + \mathcal{L}(\theta, \phi, \mathbf{x}_i),\end{aligned}$$

where $\mathcal{L}(\theta, \phi, \mathbf{x}_i)$ is the variational lower bound. It bounds the log-likelihood (of $\mathbf{x_i}$) from below since the KL-Divergence is always non-negative. Because the log-likelihood is fixed, maximizing this lower bound minimizes the KL-Divergence between the true and the approximated posterior. Thus the objective function of the VAE is given by

$$\min_{\theta,\phi} -\mathcal{L}(\theta, \phi, \mathbf{x}_i) = \min_{\theta,\phi} KL(q_\phi(\mathbf{z}|\mathbf{x}_i)||p_\theta(\mathbf{z})) - \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}_i)}[\log p_\theta(\mathbf{x}_i|\mathbf{z})].$$

The expected value term in this objective represents the expected reconstruction loss of the VAE while the KL term acts as a regularization by constraining the approximated posterior to be close to the prior.

This objective is then optimized using stochastic gradient descent, however, since the naive Monte Carlo estimator of the gradient suffers from high variance [Paisley et al., 2012] it can not be used. Kingma and Welling [2013] circumvent this problem by introducing the reparameterization trick.

2.4.2 The Embed to Control Model

The first part of Embed to Control (E2C) is a VAE used to encode an observation $\mathbf{o}_t$ to a lower dimensional latent representation $\mathbf{z}_t$ and reconstruct (decode) observations $\tilde{\mathbf{o}}_t$ from a given latent representation $\mathbf{z}_t$. Formally the encoding and decoding are given by two probability distributions

$$\begin{aligned}\mathbf{z}_t &\sim Q_\phi(Z|O) = \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) \\ \tilde{\mathbf{o}}_t &\sim P_\theta(O|Z) = Bernoulli(\mathbf{p}_t),\end{aligned}$$

learned by the VAE.

In addition to the VAE, another normal distribution is learned, this distribution represents the locally linear transition dynamics

$$\hat{\mathbf{z}}_{t+1} \sim \hat{Q}_\psi(\hat{Z}|Z, \mathbf{u}_t) = \mathcal{N}(\mathbf{A}_t\boldsymbol{\mu}_t + \mathbf{B}_t\mathbf{u}_t + \mathbf{b}_t, \mathbf{C}_t).$$

To estimate the parameters $\mathbf{A}_t, \mathbf{B}_t, \mathbf{b}_t$ of this model, another neural network is trained simultaneously to the VAE. The variance of the transition is given by

$$\mathbf{C}_t = \mathbf{A}_t\boldsymbol{\Sigma}_t\mathbf{A}_t^T + \mathbf{H}_t,$$

where $\mathbf{H}_t$ is an estimate of the covariance of the system noise which is modelled by a zero mean Gaussian distribution $\mathcal{N}(\mathbf{0}, \mathbf{H}_t)$.

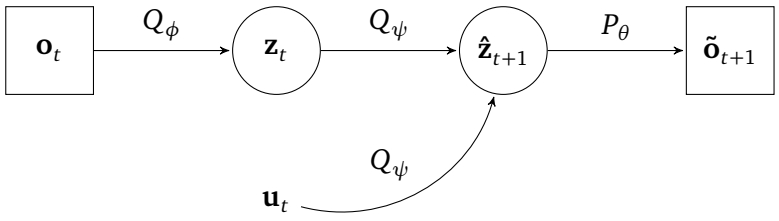


Figure 2.6: Visualization of the E2C-Model. First an observation is encoded, after the next latent state is predicted and finally decoded to get the next predicted observation

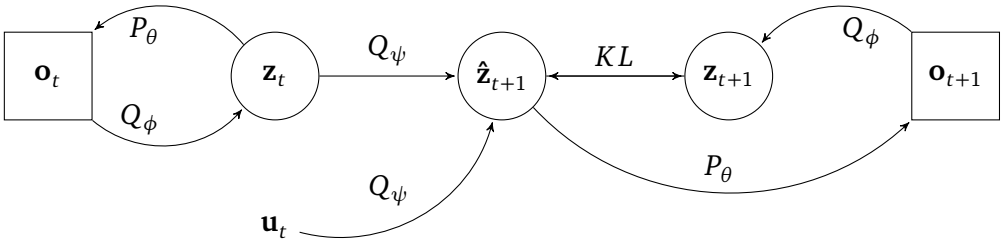


Figure 2.7: Information flow in the E2C model during training. The current observation is encoded and decoded in order to train the VAE. The VAE is also trained on decoding the next latent state, as predicted by the transition model. Last, the next observation is encoded and the obtained latent representation forced to be similar to the predicted next latent state by an KL-Divergence term in the cost function.

2.4.3 Training the Model

For training a data set of the form $D = \{(\mathbf{o}_1, \mathbf{u}_1, \mathbf{o}_2), \cdots, (\mathbf{o}_{T-1}, \mathbf{u}_{T-1}, \mathbf{o}_T)\}$ is used. The parameters are optimized according to the following objective function

$$L(D, \phi, \psi, \theta) = \sum_{(\mathbf{o}_t, \mathbf{u}_t, \mathbf{o}_{t+1}) \in D} -\mathcal{L}(\phi, \theta, \mathbf{o}_t, \mathbf{u}_t, \mathbf{o}_{t+1}) + \lambda \mathrm{KL}\left(Q_\psi(\hat{Z}|\boldsymbol{\mu}_t, \mathbf{u}_t) \parallel Q_\phi(Z|\mathbf{o}_{t+1})\right). \tag{2.1}$$

The first part is the negative variational lower bound, the cost function training the VAE. Note that not only the reconstruction of the current image is learned but the model is also trained to reconstruct the next image from the latent representation of the next state as given by the transition model.

$$\begin{aligned} &-\mathcal{L}(\phi, \theta, \mathbf{o}_t, \mathbf{u}_t, \mathbf{o}_{t+1}) = \\ &\mathbb{E}_{Q_\phi(\mathbf{z}_t|\mathbf{o}_t)}[-\log P_\theta(\mathbf{o}_t|\mathbf{z}_t)] + \mathbb{E}_{Q_\psi(\hat{\mathbf{z}}_{t+1}|\mathbf{z}_t)}[-\log P_\theta(\mathbf{o}_{t+1}|\hat{\mathbf{z}}_{t+1})] + \mathrm{KL}(Q_\phi(\mathbf{z}_t|\mathbf{o}_t) \parallel P(Z)). \end{aligned} \tag{2.2}$$

The second part of the cost function (2.1) places an additional constraint on the latent representation, forcing the encoder and the transition model to produce similar results. It is vital for control that the trajectories planned out in the latent space Z are also valid in the observation space O. This is ensured by the similarity in the outputs produced by the encoder and the transition model since it allows the transition model to work with its own outputs. This can be used to predict states for multiple time steps into the future by feeding the last state predicted by the transition model and the current action back into the transition model. The influence of this term is weighted by a scalar λ, a hyperparameter of the model.

Figure 2.7 shows the information flow in the E2C model during training. In the case of Gaussian distributions both the KL terms in Equations 2.1 and 2.2 can be computed analytically.

Since the E2C model has no internal memory it only works if the observation data has the Markov property, i.e.,

$$P(\mathbf{o}_n|\mathbf{o}_{n-1}, \mathbf{o}_{n-2}, \ldots, \mathbf{o}_0) = P(\mathbf{o}_n|\mathbf{o}_{n-1}).$$

This is often not true for observation data, thus the Markov property has to be restored, e.g., by concatenating sufficiently many observations into a single input.

3 Deep Nonparametric Kalman Filter

We propose the Deep Nonparametric Kalman Filter (DNKF), a model to perform nonparametric inference for dynamical systems, tasked with predicting the next observations, given previous observations and actions applied. It combines ideas of both, the Kernel Kalman Filter (KKF) and Embed to Control (E2C) in a Deep Learning setting. High dimensional, nonparametric representations of probability distributions are used to form estimates over the system state. Instead of obtaining these representations by embedding the probability distributions into a Reproducing Kernel Hilbert Space (RKHS) an encoder-decoder structure is used to learn them directly from the data. The usage of a Deep Learning setting alleviates the scaling problems of the KKF and allows the usage of a transition model that, unlike the KKFs transition model, is capable of dealing with actions applied to the system.

To form the state estimates we introduce a special recurrent model, the Recurrent Kalman Layer (RKL). The recurrent nature of that model introduces internal memory in form of the state estimate into the model, hence, unlike E2C, the DNKF does not rely on the Markov property of the observations. Besides that, the RKL learns linear transition and observation models, used to incorporate observations into the estimate and predict next states. The set of all parameters of the RKL is denoted by ψ.

Since it is unreasonable to assume that the system dynamics and observation generation process are linear in the observation space, an encoding function enc_ϕ is trained, mapping the observations into a more suitable latent space. With that encoding function embeddings, similar to those introduced in Section 2.2.1 are obtained. Additionally, a decoding function dec_θ is learned, mapping latent state estimates, produced by the RKL, back to observations.

Note that, similar to the KKF, in general the latent observations and the latent state estimates do not live in the same space. Hence the encoding and the decoding function do not form an autoencoder.

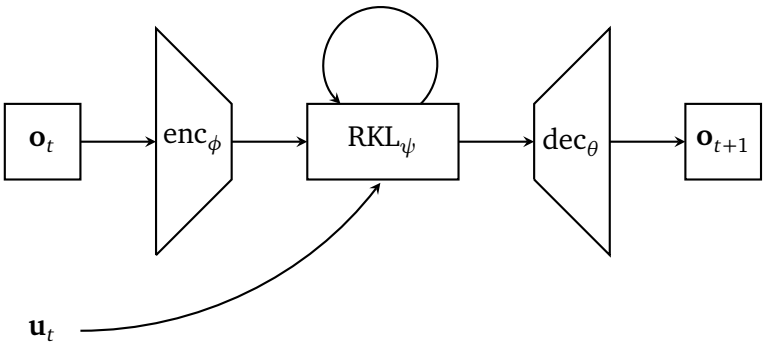


Figure 3.1: Sketch of the whole model $\mathcal{M}_{\phi,\psi,\theta}$. The observations, encoded by enc_ϕ, are fed into the RKL together with the actions. Observations are reconstructed from the estimates produced by the RKL.

3.1 Learned Embeddings

The proposed encoding and decoding functions enc_ϕ and dec_θ are currently modelled by densely connected multilayer Neural Networks (NNs). However, depending on the type of observation at hand, more sophisticated models, like convolutional NNs may be used.

Unlike the latent representation used in E2C the learned embeddings are very high dimensional. Due to the high dimensionality it is possible to approximate complex, nonlinear processes with a linear model.

Note that, due to the high latent dimension, in general neither the dimensionality of the encoded observations nor the dimensionality of the latent estimated states is smaller than the dimensionality of the observation and the models are hence not undercomplete. To cope with this, other regularization methods for autoencoders, like sparsity constrains [Ng, 2011] may be used.

3.2 Recurrent Kalman Layer

We propose the Recurrent Kalman Layer (RKL), a special recurrent layer, in order to use the idea of the KKF in a Deep Learning setting. Its internal state as well as its output is the prior state estimate $\boldsymbol{\mu}_t^-$. Given the current action $\mathbf{u}_t$ and the encoding of the current observation $\text{enc}_\phi(\mathbf{o}_t)$ the RKL follows the update and predict scheme to form its estimate over the system state using formulas similar to those of the KKF.

First the encoded observation is used to update the state estimate, forming a posterior estimate $\boldsymbol{\mu}_t^+$.

$$\text{Update:} \quad \boldsymbol{\mu}_t^+ = \boldsymbol{\mu}_t^- + \mathbf{Q}\left(\text{enc}_\phi(\mathbf{o}_t) - \mathbf{C}\boldsymbol{\mu}_t^-\right)$$

This posterior is used together with the transition model and the current action to predict the next prior.

$$\text{Predict:} \quad \boldsymbol{\mu}_{t+1}^- = \mathbf{A}\boldsymbol{\mu}_t^+ + \mathbf{B}\mathbf{u}_t + \mathbf{c}.$$

All variables in the formulas above, namely the observation model $\mathbf{C}$, the parameters of the transition model, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{c}$, as well as an additional matrix $\mathbf{Q}$, corresponding to the Kalman gain, are considered to be model parameters and are optimized during the training process.

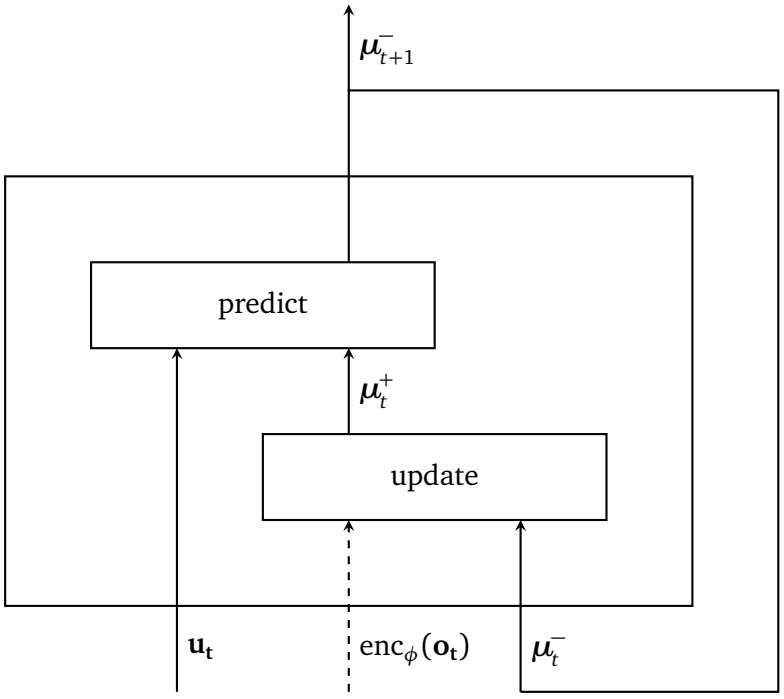


Figure 3.2: Sketch of the Recurrent Kalman Layer

Again the update step is only performed if a valid observation is available for the current time step. If this is not the case, the posterior is just set to be the prior.

Note, that the prior and the posterior are normalized after each update and prediction step in order to ensure numerical stability.

3.3 Training the Model

The model is trained using a data set

$$\mathcal{D} = \{(\mathbf{o}_i, \mathbf{u}_i)_{i=0,\dots,T_m}^{(m)} \,|m = 1, \dots, M\}$$

with M sequences of observation-action pairs. The sequences are not necessarily of equal length and the length of sequence m is denoted by T_m.

The objective of the model is to minimize the distances between given observations $(\mathbf{o}_{t-1}, \mathbf{o}_t)$ and the output of the DNKF

$$\left(\tilde{\mathbf{o}}_{t-1}^{+}, \tilde{\mathbf{o}}_t^{-}\right) = \mathcal{M}_{\phi,\psi,\theta}\left((\mathbf{o}_i, \mathbf{u}_i)_{i=0\cdots t-1}\right),$$

where $\tilde{\mathbf{o}}_{t-1}^{+}$ is obtained by decoding the a-posteriori estimate $\boldsymbol{\mu}_{t-1}^{+}$ while $\tilde{\mathbf{o}}_t^{-}$ is obtained by decoding the a-priori estimate $\boldsymbol{\mu}_t^{-}$.

We will show in Chapter 4 that the choice of the distance measure d impacts performance of the model and depends on the type of observations at hand. In order to account for the recurrent parts of the model backpropagation trough time (BPTT) is used for training. The calculated gradients are propagated through the whole network, training all parts of the model simultaneously. This yields the following objective for a single sequence $s_m \in \mathcal{D}$,

$$Ł(s_m, \phi, \psi, \theta) = \sum_{i=1}^{T_m} d\left(\mathbf{o}_i^{(m)}, \tilde{\mathbf{o}}_i^{-(m)}\right) + d\left(\mathbf{o}_{i-1}^{(m)}, \tilde{\mathbf{o}}_{i-1}^{+(m)}\right). \tag{3.1}$$

The first part of that cost function trains the model to infer the next observation. The second part is necessary to enforce similarity between the prior and posterior estimates by forcing both to be compatible with the decoder. This is needed in the case when no observation is present and the prior and posterior estimates are identical.

4 Evaluation and Experiments

The Deep Nonparametric Kalman Filter (DNKF) is evaluated on two dynamical systems, a pendulum and a quad link. After describing our set up in detail, the results of an evaluation of different hyperparameters are presented. A comparison of our model with both, Embed to Control (E2C) and the Kernel Kalman Filter (KKF) follows on data sets generated without and with noise.

4.1 Set Up

4.1.1 Data sets

Both the pendulum and the quad link were simulated with initial positions uniformly sampled from the entire space and initial velocities of 0.

Following the experiments conducted by Watter et al. [2015], black and white images were generated as observations for the pendulum. Each image has a resolution of 48×48 pixels, yielding a 2304 dimensional observation, and the pendulum is rendered as a simple line. See Figure 4.1a for example images. Both, the training and test set, consist of $15,000$ samples, split up in 100 episodes with 150 steps each.

Similar to the pendulum, trajectories for the quad link were sampled. The endeffector position i.e., the end point of the outermost link was used as observation, yielding a two dimensional observation. See Figure 4.1b for an example of a whole trajectory. Both data sets consist of $60,000$ samples each, split up in 200 episodes with 300 steps each.

Due to the cubic scaling of the KKF, subsets had to be used. To make up for the disadvantage caused by the smaller data sets the initial positions were not sampled from the entire space but from a subspace whose size corresponded to the size of the data set. Furthermore, only sets without actions were used since the KKF is not capable of dealing with actuated systems.

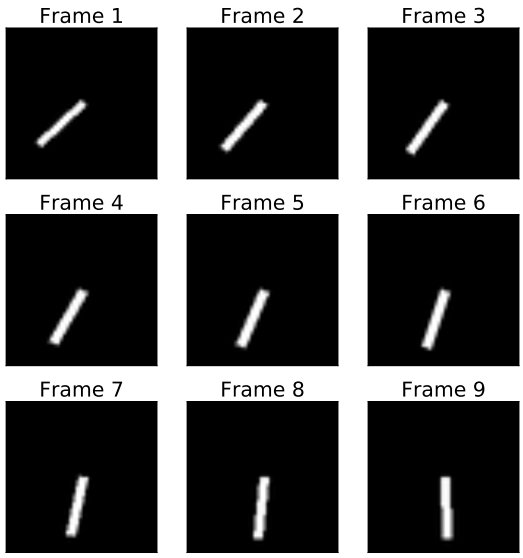
Note that neither the images nor the endeffector positions have the Markov property, since in both cases it is impossible to infer the velocity from a single observation.

4.1.2 Implementation

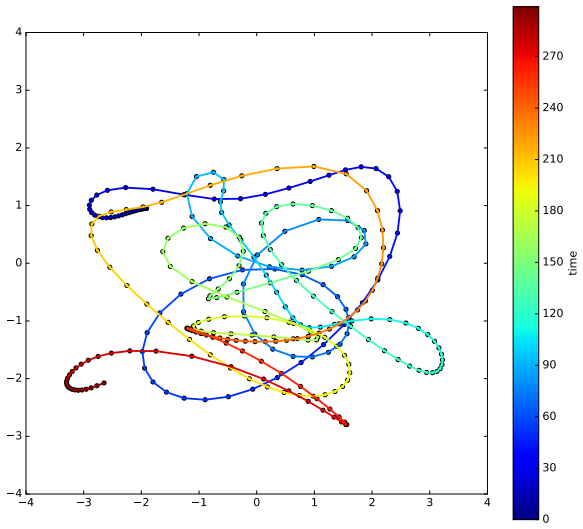
Both the DNKF and E2C were implemented in Python using Tensorflow [Abadi et al., 2015], a modern Deep Learning framework making use of highly parallel hardware and accelerators. Further, automated differentiation is a build in feature, hence no gradients were calculated manually.

For E2C the encoder and decoder were modelled by densely connected Neural Networks (NNs) with two hidden layers each, both with a width of 800 neurons. The transition model was also modelled by densely connected NNs with two hidden layers, which have a width of 100 neurons each. This set up was described by Watter et al. [2015] in their experiments with a pendulum.

Those densely connected NNs with two hidden layers were also used for the encoder and decoder of the DNKF. Different values of their width were evaluated, together with the dimensions of the latent observations and latent states used by the Recurrent Kalman Layer (RKL). Further, in order to increase learning speed, truncated backpropagation trough time (BPTT) was used. Again, different values for k, the number of steps after which the error is truncated, were compared. The results of this hpyerparameter evaluation can be found in Section 4.2.



(a) Example observations of the pendulum. Each image has a size of 48×48 pixels



(b) Example trajectory of the quad link endeffector position. Colour corresponding to time steps for better visibility

Figure 4.1: Example observations

Two different loss functions were considered, the Mean Squared Error (MSE) and the Cross Entropy. If images are used as observations, as done for the pendulum, both are possible. To justify the Cross Entropy each pixel value p is interpreted as a Bernoulli distribution with mean p of that pixel to be black (0) or white (1). It turned out that the Cross Entropy in fact worked better, hence it was used to train the DNKF for the pendulum. For the endeffector positions of the quad link the Cross Entropy is unsuitable since they are not interpreted as probability distribution, thus the MSE was used.

Both, E2C and the DNKF were trained using Adam [Kingma and Ba, 2014]. A learning rate of $\alpha = 0.0005$, an exponential decay rate of the gradients average, $\beta_1 = 0.9$ and exponential decay rate of the average over the squared gradients, $\beta_2 = 0.999$, were used. All weight matrices were initialized using the uniform Glorot initialization[1] as introduced by Glorot and Bengio [2010].

The KKF implementation was provided and is also written in Python.

4.1.3 Errors Measures for Evaluation

Three different errors were used to compare the models. First the current state error, measuring the distance between input and its reconstruction. For the DNKF this error is the second term of Equation 3.1, for E2C the first expectation term in Equation 2.2. Second, the next state error, measuring the distance between the next inferred state and the corresponding target. For the DNKF this is the first term of Equation 3.1, for E2C the second expectation term in Equation 2.2.

Last we used a MSE evaluation error to enable comparison of approaches with different cost functions. This error is just the mean of the MSE of all target in the test set and the corresponding predictions. For E2C this error is measured only for the last element of the output sequence, since all previous elements of that sequence are part of the input and were only added to restore the Markov property.

[1] Often also referred to as uniform Xavier initialization, after Glorots first name

4.2 Hyperparameter Evaluation

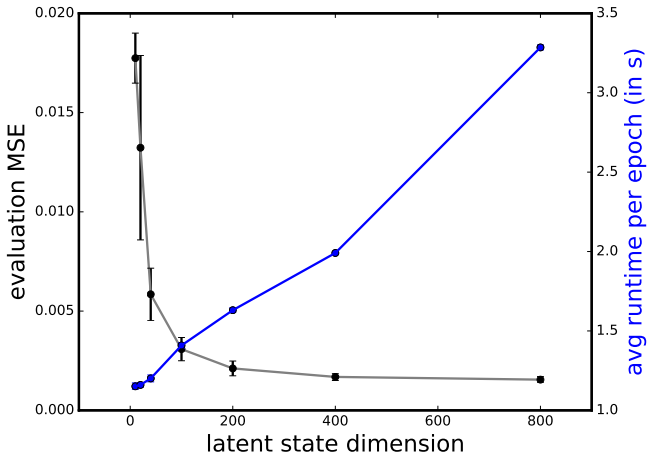
Due to the vast space of hyperparameters and the long runtime of a single experiment, only a limited evaluation of hyperparameters could be performed.

4.2.1 Latent State Dimension

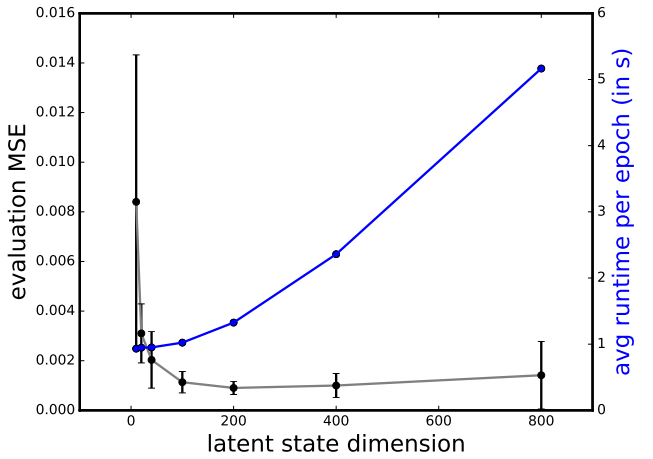
The latent state dimension was evaluated for the following values:

$$10, 20, 40, 100, 200, 400, 800.$$

During each experiment the dimension of the latent observations was set to one half of the latent state dimension. The number of neurons in all hidden layers, in both the encoder and decoder, was set to two times the latent state dimension. The minimal loss was obtained for latent state dimensions of 800 for the pendulum and 200 for the quad link. Detailed results, together with the runtime for each case, can be found in Figure 4.2.



(a) Pendulum: Minimum at 800 (MSE: 0.00154). Standard deviation to small to be visible for high values.



(b) Quad Link: Minimum at 200 (MSE: $9.02 \cdot 10^{-4}$)

Figure 4.2: Evaluation of different latent state dimensions. Evaluation MSE results averaged over 20 trials. Error bars show two times standard deviation. For most runtime values the standard deviation is to small to be visible.

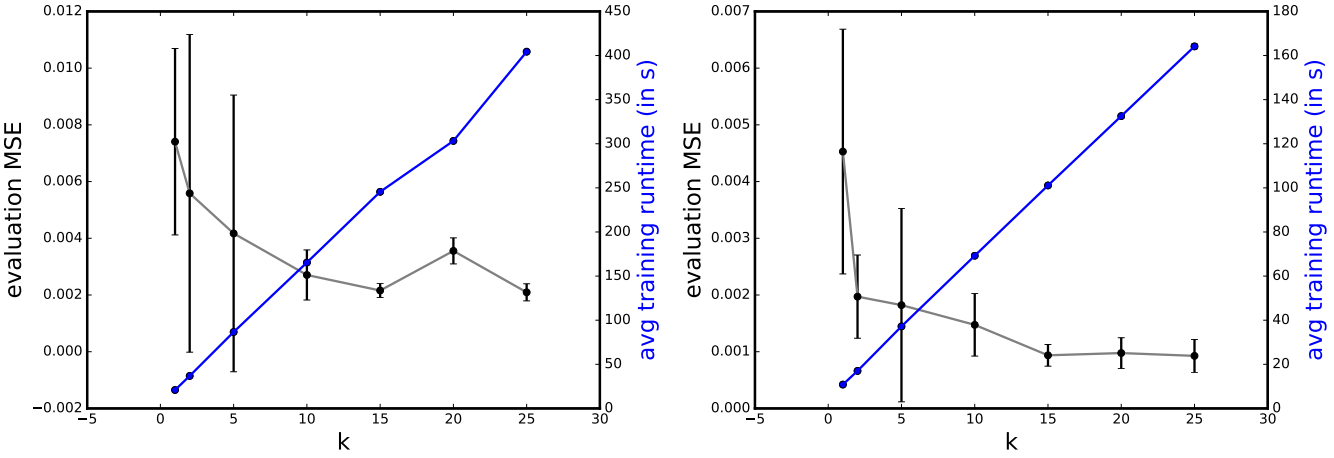
From the smallest to the largest value of the latent state dimension, the runtime roughly increases by a factor 3 for the pendulum and 5 for the quad link, while the dimensions increase by a factor 80. This is due to the system used to execute the experiments, which employs two highly parallel hardware accelerators (Nvidia Tesla k20). The usage of such hardware accelerators is common for Deep Learning approaches and allows efficient training of models with a large amount of parameters on large data sets.

4.2.2 Truncation of BPTT

Furthermore, the number of steps after which the error is truncated and propagated back trough the network, k, was evaluated for the values

$$1, 2, 5, 10, 15, 20, 25.$$

To cope with the fact that smaller values for k yield more weight updates per epoch, the models were trained for $5 \cdot k$ training epochs. As determined in the previous section, the latent space dimension was set to 800 for the pendulum and 200 for the quad link. The results can be found in Figure 4.3. For both, the pendulum and the quad link, the minimum was obtained for $k = 25$. No bigger values were evaluated since they lead to unreasonable high runtimes.



(a) Pendulum: Minimum at 25 (MSE: 0.00209)

(b) Quad Link: Minimum at 25(MSE: $9.26 \cdot 10^{-4}$)

Figure 4.3: Evaluation of different values of k (number of steps between truncations). Evaluation MSE results averaged over 20 trials. Error bars show two times standard deviation. For the runtime the standard deviation is to small to be visible.

The parameter k has almost no influence on the runtime of a single epoch, however, as stated above, for larger values of k more epochs are needed to train the model since fewer update steps are performed per epoch. Because of that not the runtime per epoch is displayed in Figure 4.3 but the runtime of the whole training process.

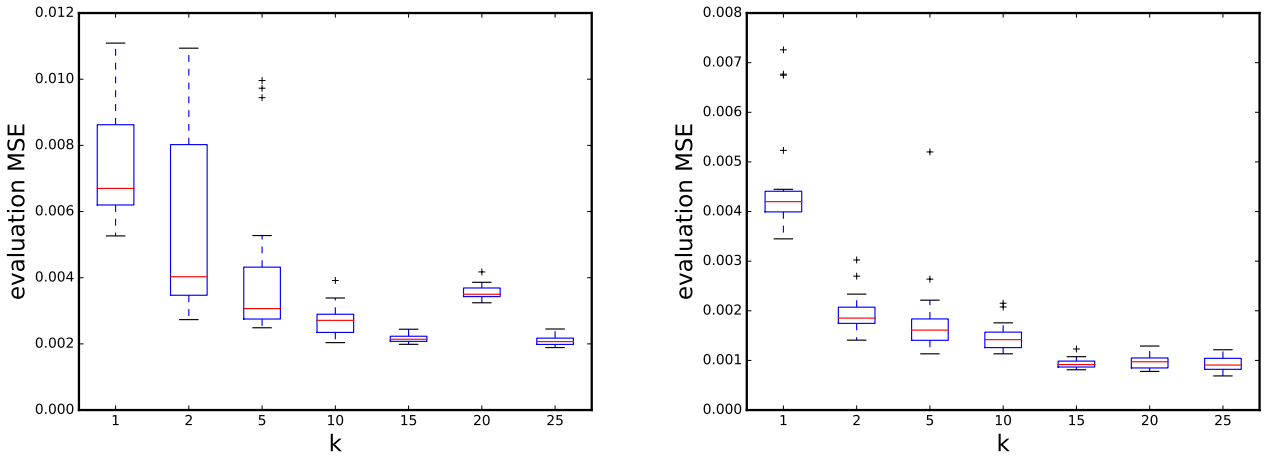
The high standard deviation for low values of k is explained by outliers, performing worse than the mean. This can be seen in the corresponding box plots shown in Figure 4.4. For further experiments the value of $k = 15$ was chosen for the pendulum. This value produced only slightly worse results (MSE: 0.00215) in three fifths of the time.

4.3 One Step Prediction Error

The DNKF, E2C and the KKF were compared on the task of predicting the next state on both the pendulum and the quad link. While the DNKF and E2C were evaluated on data sets with and without actions, the KKF was evaluated only on the smaller data sets without actions, as described above.

4.3.1 Pendulum

Our E2C implementation achieved slightly better results as those reported by Watter et al. [2015]. They reported a next state prediction loss, measured with the Cross Entropy, of 89.3 for the pendulum, we archived 67.1. Despite, the fact that is was tried to reproduce the experiments as good as possible, using the same network architecture and latent space dimension as reported in the E2C paper, those two values are not completely comparable since the error is highly depended on the time steps between single observations and the magnitude of the actions applied. Both were not reported in the original paper.



(a) Pendulum **(b)** Quad Link

Figure 4.4: Evaluation of different values of k. Red horizontal line marks median, blue box marks the first and third quartile. The difference between those is referred to as inner quartile range (IQR). Horizontal black lines mark the minimum and maximum value that still lie in the range between the first quartile minus the IQR and the third quartile plus the IQR. Outliers, i.e., points outside of that range, are marked by $+$. Note that both systems have outliers, especially for low values of k, and all those outliers are worse than the mean. This causes the big standard deviation seen in Figure 4.3.

Note that for comparison with the DNKF, these values need to be halved since E2C works with sequences of two consecutive images. The weighting factor of the additional constraining KL-term was set to $\lambda = 0.25$, also according to the original paper.

Table 4.1 and Table 4.2 show the results of the evaluation. Further, Table 4.1 shows that the DNKF works better with the Cross Entropy loss than with the MSE. By using the Cross Entropy loss results similar to E2C were achieved in the actuated case. In the unactuated case the DNKF works slightly better than E2C. Note that for all experiments the next state error is only slightly worse than the current state error. The KKF performs worse than E2C and the DNKF.

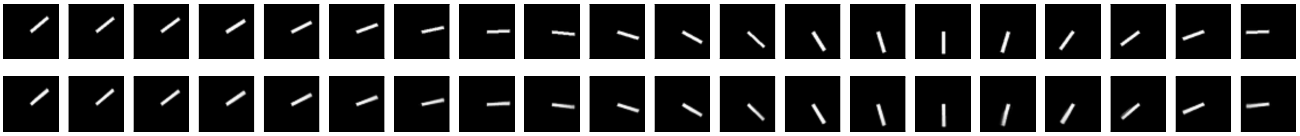


Figure 4.5: Upper row: Part of true trajectory. **Lower row:** One step predictions, produced by the DNKF

4.3.2 Quad Link

Unlike the images used as observations for the pendulum, the endeffector positions of the quad link are not interpretable as Bernoulli distributions. Hence, it was not possible to use the original E2C Cross Entropy loss function. Both Cross Entropy terms were replaced by square loss functions, making it necessary to rescale the regularization terms. However, this turned out to be problematic since on the one hand less regularization clearly yields better results while on the other side it alleviates the similarity between the transition models and the encoders outputs, which is one of the reasons why E2C works in the first place. Eventually, the regularizations terms were scaled down by a factor of 100. To restore the Markov property we used sequences of length 5 to train the model. Note the difference between the next

Pendulum (with actions)	E2C	DNKF (CE)	DNKF (MSE)
Current State (CE)	30.049 ± 0.731	28.397 ± 0.615	-
Next State (CE)	33.155 ± 0.756	34.621 ± 1.355	-
Evaluation Error (MSE)	$0.00124 \pm 1.17 \cdot 10^{-4}$	$0.00100 \pm 1.77 \cdot 10^{-4}$	0.00979 ± 0.00119

Table 4.1: Comparison of E2C and the DNKF on images of the actuated pendulum. The DNKF was once trained with Cross Entropy as error and once with MSE. Results averaged over 20 trails, ± two times standard deviation.

Pendulum (without actions)	E2C	KKF	DNKF (CE)
Current State (CE)	28.975 ± 0.733	—	27.887 ± 0.369
Next State (CE)	31.044 ± 0.449	—	31.880 ± 0.611
Evaluation Error (MSE)	$9.27 \cdot 10^{-4} \pm 6.04 \cdot 10^{-5}$	0.00219 ± 0.00201	$6.84 \cdot 10^{-4} \pm 8.15 \cdot 10^{-5}$

Table 4.2: Comparison of E2C and the DNKF, evaluated on the pendulum with no actions applied. Evaluation of the DNKF with MSE as loss function is omitted, since it has been already shown that it performs worse. Results averaged over 20 trails, ± two times standard deviation.

state error, measured on the whole target sequence and the evaluation error measured only on the last element of the target sequence.

The DNKF instead showed no difficulties working with the MSE instead of the Cross Entropy and outperformed E2C, as shown in Table 4.3 and Table 4.4. For the DNKF the next state prediction error and the evaluation error are the same. Note that for the DNKF the next state error is worse than the current state error by factors of roughly 4 and 12 in the unactuated and actuated case respectively. We try to explain this behaviour in Section 5.1.

The KKF produced results with a high variance and performs worse than the DNKF if the mean over all 20 trails is considered. However, this high variance is caused by a few trails with poor performance and if the medians over all trials are compared, the KKF slightly outperforms the DNKF.

Quad Link (without actions)	E2C	KKF	DNKF (MSE)
Current State (MSE)	$0.00394 \pm 5.35 \cdot 10^{-4}$	-	$2.05 \cdot 10^{-4} \pm 1.38 \cdot 10^{-4}$
Next State (MSE)	$0.00246 \pm 4.53 \cdot 10^{-4}$	-	$8.17 \cdot 10^{-4} \pm 2.58 \cdot 10^{-4}$
Evaluation Error	0.00448 ± 0.00106	0.00510 ± 0.0191	$8.17 \cdot 10^{-4} \pm 2.58 \cdot 10^{-4}$
Evaluation Error (median)	-	$7.46 \cdot 10^{-4}$	$7.91 \cdot 10^{-4}$

Table 4.3: Comparison of E2C, the KKF and the DNKF on the unactuated quad link. Results averaged over 20 trails, ± two times standard deviation.

Quad Link (with actions)	E2C	DNKF
Current State (MSE)	0.00695 ± 0.00141	$3.58 \cdot 10^{-4} \pm 3.04 \cdot 10^{-4}$
Next State (MSE)	0.00658 ± 0.00144	0.00420 ± 0.00107
Evaluation Error (MSE)	0.0179 ± 0.00447	0.00420 ± 0.00107

Table 4.4: Comparison of E2C and the DNKF, evaluated on the quad link with uniform sampled actions applied to all four links. Results averaged over 20 trails, ± two times standard deviation.

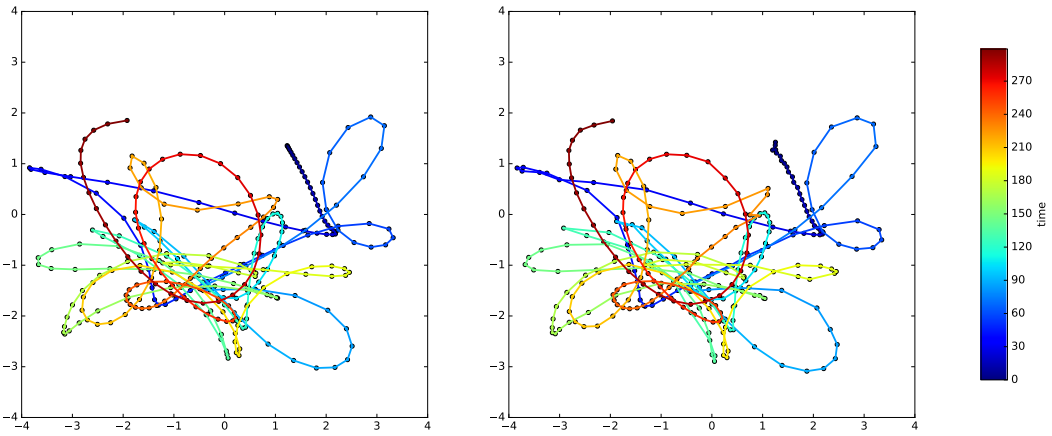


Figure 4.6: Left: True trajectory. **Right:** Predicted trajectory, the trajectory was predicted one step at a time with observations after each step. Colour corresponding to time for better visibility

4.4 One Step Prediction Error with Noise

In real world scenarios the observations as well as the transitions of the system are often noisy, hence experiments were conducted to see how DNKF, E2C and the KKF perform in the presence of noise.

Two forms of noise were used. First, observation noise, added on the generated observations. For the pendulum the observation noise was added on the current angle before the image was rendered. For the quad link the observation noise was added to the endeffector position. Second, transition noise was added to the true state after every transition.

While the system was trained with noisy samples as well as noisy target the evaluation on the test set was conducted using only the noisy inputs but targets free of observation noise. All noise was sampled from zero mean Gaussian distributions with different standard deviations and the unactuated pendulum and quad link were evaluated with different combinations of observation and transition noise. The results can be found in Table 4.5 and Table 4.6 respectively.

First, we note that performance loss is roughly equal for both, observation and transition noise using the pendulum, on the quad link all approaches suffered more from transition noise than from observation noise. For the pendulum, the DNKF works best for low amounts of noise, however for increasing amount

of noise the KKF outperforms the DNKF. In general the KKF suffers less from the added noise than the other approaches. E2C performed slightly worse than the DNKF for all experiments.

The experiments on the quad link show that the KKF handles the noise better and outperforms the other approaches. In fact the issue with the outliers, performing much worse than the median, reported in Section 4.3.2 is not longer present.

observation noise	transition noise	E2C	KKF	DNKF
0.05	0.0	$0.00617 \pm 2.12 \cdot 10^{-4}$	0.00647 ± 0.00722	$0.00338 \pm 3.26 \cdot 10^{-4}$
0.1	0.0	$0.00899 \pm 1.91 \cdot 10^{-4}$	0.00708 ± 0.00730	$0.00646 \pm 6.90 \cdot 10^{-4}$
0.25	0.0	$0.0122 \pm 1.96 \cdot 10^{-4}$	0.00740 ± 0.00572	$0.0117 \pm 6.25 \cdot 10^{-4}$
0.0	0.05	$0.0047 \pm 1.39 \cdot 10^{-4}$	0.00730 ± 0.00686	$0.00354 \pm 1.39 \cdot 10^{-4}$
0.0	0.1	$0.00833 \pm 2.26 \cdot 10^{-4}$	0.00805 ± 0.00666	$0.00695 \pm 1.58 \cdot 10^{-4}$
0.05	0.05	$0.00745 \pm 2.09 \cdot 10^{-4}$	0.00773 ± 0.00597	$0.00512 \pm 1.19 \cdot 10^{-4}$
0.1	0.05	$0.00971 \pm 1.75 \cdot 10^{-4}$	0.00742 ± 0.00559	$0.00759 \pm 2.30 \cdot 10^{-4}$
0.25	0.05	$0.0125 \pm 2.70 \cdot 10^{-4}$	0.00726 ± 0.00612	$0.0119 \pm 5.75 \cdot 10^{-4}$
0.05	0.1	$0.00929 \pm 2.19 \cdot 10^{-4}$	0.00749 ± 0.00479	$0.00761 \pm 2.54 \cdot 10^{-4}$
0.1	0.1	$0.0106 \pm 1.64 \cdot 10^{-4}$	0.00781 ± 0.00461	$0.00912 \pm 2.57 \cdot 10^{-4}$

Table 4.5: Evaluation Error for noisy unactuated pendulum. First two columns show the standard deviation of the Gaussian distributions the noise was sampled from. Results averaged over 10 trails, $\pm$ two times standard deviation.

4.5 Efficiency

An empirical runtime comparison of the DNKF and E2C is not reasonable, since the runtime is mainly determined by the network architecture and the width of the individual layers for both. However, on the one hand, during training the DNKF need only one pass trough the encoder, while E2C needs two. On the other hand the DNKF needs more passes per weight update than E2C due to the usage of (truncated) BPTT.

It is mentionable that in the experiments performed in Section 4.3, the DNKF needed roughly one fifth of the update steps needed by E2C for the pendulum to achieve the reported results. For the quad link the DNKF needed roughly one fifteenth of the update steps needed by E2C.

observation noise	transition noise	E2C	KKF	DNKF
0.05	0.0	0.0124 ± 0.00118	$9.00 \cdot 10^{-4} \pm 0.00114$	$0.0100 \pm 4.32 \cdot 10^{-4}$
0.1	0.0	0.0254 ± 0.00182	0.00109 ± 0.00143	$0.0221 \pm 8.05 \cdot 10^{-4}$
0.25	0.0	0.0851 ± 0.00222	0.00153 ± 0.00272	0.0695 ± 0.00208
0.0	0.05	0.0161 ± 0.00171	0.00417 ± 0.0148	$0.0137 \pm 8.52 \cdot 10^{-4}$
0.0	0.1	0.0555 ± 0.00563	0.00508 ± 0.0152	0.0533 ± 0.00456
0.05	0.05	0.0221 ± 0.00127	0.00466 ± 0.00840	0.0204 ± 0.00119
0.1	0.05	0.0364 ± 0.00185	0.00493 ± 0.00843	$0.0333 \pm 8.59 \cdot 10^{-4}$
0.25	0.05	0.101 ± 0.00426	0.00651 ± 0.0111	0.0870 ± 0.00317
0.05	0.1	0.0625 ± 0.00529	0.00612 ± 0.0177	0.0603 ± 0.00456
0.1	0.1	0.0808 ± 0.00532	0.00672 ± 0.0199	0.0769 ± 0.00508

Table 4.6: Evaluation Error for noisy unactuated quad link. First two columns show the standard deviation of the Gaussian distributions the noise was sampled from. Results averaged over 10 trails, $\pm$ two times standard deviation.

5 Conclusion

In this thesis we introduced the Deep Nonparametric Kalman Filter (DNKF), our own approach for nonparametric inference. The Recurrent Kalman Layer, a special recurrent layer based on the formulas of the Kernel Kalman Filter (KKF), was derived. This was combined with the approach of using an encoder-decoder structure to find latent spaces for the system to operate in, an idea taken from the Embed to Control (E2C) paper by Watter et al. [2015]. The usage of a Deep Learning setting alleviates the KKFs problems with big data sets. Further, the approach is capable of performing inference for actuated systems, since a corresponding term was added to the transition model.

Different hyperparameters were evaluated for that approach. However, due to the high computational cost of a single model evaluation only a restricted subset of the hyperparameters could be evaluated.

In comparison with E2C we saw that the DNKF performed roughly equally well on the images of a pendulum. However, using E2C, we were not able to achieve satisfying results with the quad link. The DNKF on the other hand performed well with the quad link. When compared to the KKF the DNKF performs better in cases with no noise. On the other side our experiments showed that the KKF is more capable of dealing with noise and outperformed the DNKF on the pendulum if sufficient noise was added and in all quad link experiments with noise.

Additionally, we were able to reproduce some of the results reported in the original E2C paper, more particular, for the observations of a pendulum roughly the same prediction error was obtained.

5.1 Issues with the Quad Link

In section 4.3.2 a big difference between the current state error and the next state error was noted for the quad link. This difference is absent for the pendulum. This problem may arise since the encoder-decoder structure is not undercomplete for the quad link and it is possible to just copy the current observation trough the encoder, update step and decoder to the output, yielding a very small current state error.

For the one step prediction evaluated in that section this does not poses a problem, however it becomes problematic when the model is tasked with predicting multiple steps without observations. In this case the transition model needs to work with the a-priori estimate and the model fails since that estimate is not compatible with the learned transition model.

5.2 Inferring multiple Steps

Even without the issue stated above, which is not present for the pendulum, the model is not yet able to predict multiple steps into the future. This is likely caused by a difference in the prior and posterior representation that is not handleable by the transition model. It was tried to train the transition model on performing transitions with both the prior and the posterior by adding a corresponding term to the cost function. This term measures the distance between a decoded prediction obtained by transitioning the prior and the next state observation. This however prevented the model from learning anything but averaging over all images, yielding the same output for every input, this output is shown in Figure 5.1.

5.3 Future Work

Once the above mentioned issues are resolved it remains to evaluate the DNKF on more complex and real world systems. In order to find suitable latent representations for such systems, more sophisticated

(a) Average image over test set

(b) Results produced by DNKF with modified cost function. The result is almost identical for all inputs

Figure 5.1: When the DNKF is trained with the modified cost function discribed in Section 5.2 it learns to average over all pictures.

encoder-decoder structures, such as deeper Neural Network (NN) and convolutional NN, need to be used.

Additionally, in the introduction it was stated that inference is an important aspect of control. It remains to evaluate how the models learned by the DNKF perform when employed to solve a controlling task, especially in comparison with E2C, whose purpose is not inference but control. For example the model learned from the pendulum observations can be combined with a linear quadratic regulator in order to solve tasks like a pendulum swing up.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Fukumizu, K., Song, L., and Gretton, A. (2013). Kernel bayes' rule: Bayesian inference with positive definite kernels. *Journal of Machine Learning Research*, 14(1):3753–3783.

Gebhardt, G. H., Kupcsik, A., and Neumann, G. (2016). The kernel kalman rule – efficient nonparametric inference with recursive least squares.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256.

Goldberg, Y. (2015). A primer on neural network models for natural language processing. *arXiv preprint arXiv:1510.00726*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Kálmán, R. E. (1960). A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Ng, A. (2011). Sparse autoencoder. *CS294A Lecture notes*, 72:1–19.

Paisley, J., Blei, D., and Jordan, M. (2012). Variational bayesian inference with stochastic search. *arXiv preprint arXiv:1206.6430*.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503.

Smola, A., Gretton, A., Song, L., and Schölkopf, B. (2007). A hilbert space embedding for distributions. In *International Conference on Algorithmic Learning Theory*, pages 13–31. Springer.

Song, L., Fukumizu, K., and Gretton, A. (2013). Kernel embeddings of conditional distributions: A unified kernel framework for nonparametric inference in graphical models. *IEEE Signal Processing Magazine*, 30(4):98–111.

Song, L., Huang, J., Smola, A., and Fukumizu, K. (2009). Hilbert space embeddings of conditional distributions with applications to dynamical systems. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 961–968. ACM.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015). Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3156–3164.

Watter, M., Springenberg, J., Boedecker, J., and Riedmiller, M. (2015). Embed to control: A locally linear latent dynamics model for control from raw images. In *Advances in Neural Information Processing Systems 28*, pages 2728–2736.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Williams, R. J. and Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501.