UCLouvain

epl

École polytechnique de Louvain

In-Context Self-Supervised learning for Stochastic Optimal Control with Transformer architecture

Author: **Margot DEVILLERS**
Supervisor: **Raphaël JUNGERS**
Readers: **Johan SUYKENS, Gianluca BIANCHIN**
Academic year 2024–2025
Master [120] in Mathematical Engineering

They say that diving into Deep Learning reveals the real beauty of Nature.
The more I tuned my Transformers, the more I learned to enjoy Belgium's gentle rain.

Contents

Acknowledgments

I would like to express my sincere gratitude to all those who have contributed to the realization of this master's thesis.

Foremost, I am deeply grateful to my supervisor, Raphaël Jungers, for introducing me to this exciting research topic and for his invaluable guidance and support throughout this journey. His insightful feedback, dedication, and encouragement have played a key role in shaping both the direction and the quality of this research.

I would also like to extend my thanks to my friends and family for their support, patience, and encouragement during this final year of my studies. Their presence has been a source of strength and motivation.

Thank you all for your contributions.

Abstract

The past decade has witnessed remarkable advances in *Artificial Intelligence* (AI), largely driven by the rise of Transformer architectures powered by self-attention mechanisms. These architectures have set new standards in tasks involving natural language processing, reasoning, and decision making. As their remarkable performance extends beyond language, an important question arises: can the models based on such architecture be effectively applied to the control of physical systems operating in uncertain, nonlinear, or partially known environments?

This thesis lies at the intersection of AI and control theory and investigates how the in-context learning capabilities of Transformer models can be leveraged to learn control strategies directly from data. Building upon a recent study demonstrating that Transformers can learn optimal estimation policies from trajectories without explicit system knowledge, we extend this approach to the domain of control. Our work explores whether these models can similarly learn to generate optimal control inputs in a data-driven and generalizable way.

We begin by adapting the original output estimation framework to a supervised control setting, where the model is trained to map state trajectories to control sequences. This first contribution demonstrates the model's robustness to dynamic and uncertain environments, and its capacity to implicitly infer relationships between a system's internal state and control input from brief contextual prompts. However, it also exposes key limitations in estimation accuracy, stemming from the absence of a system-specific optimization objective. To address these challenges, we propose a novel self-supervised framework in which the Transformer learns control laws by interacting with the system and minimizing a predefined cost function over time, without requiring any labeled supervision. This approach enables the model to learn and adapt dynamically, offering a more flexible and scalable alternative to classical control strategies in complex, real-world environments.

The results of this thesis suggest that large Transformer-based models hold promise as general-purpose controllers for safety-critical systems.

Chapter 1

Introduction

The rapid integration of computational systems capable of mimicking human intelligence is transforming everyday life across industries. In healthcare, for example, medical professionals are supported by intelligent systems that assist in organizing patient records, aiding diagnoses and, in more advanced applications, analyzing X-rays [1]. In addition to these supervised tasks, such systems are also used to accelerate patient care and medical discovery by leveraging large datasets—for instance, by predicting the progression of diseases, identifying risk factors, and even accelerating drug discovery. In a similar effort to enhance protection, intelligent technologies are also employed in environmental monitoring—modeling climate phenomena like glacier change, protecting pollinating animals, optimizing energy consumption, and improving climate forecasting. On yet another front, AI has become instrumental in space exploration. From analyzing astronomical data to guiding exploration robots, these systems assist in mission planning, anomaly detection, and ensuring astronaut safety [2]. These are just a few glimpses of how smart systems are shaping the modern world.

What exactly are these systems, how do they manage to achieve such remarkable feats, and to what extent can we integrate them into our lives?

Figure 1.1: AI-assisted diagnostics in healthcare [1].

These questions are at the heart of a multidisciplinary field at the intersection of computer science, mathematics, statistics, and biology: *Artificial Intelligence* (AI). This field aims to create systems that perform tasks typically associated with human intelligence—learning, reasoning, and decision-making—using algorithms, statistical models, and rule-based logic [3]. Although already widespread, today's AI systems are still far from reaching the elusive goal of *general intelligence*. To better appreciate both their current capabilities and future potential, a historical overview of AI's evolution, milestones, and paradigm shifts is given in Appendix 7.1 for the interested reader, or in classical textbooks like [3].

One of the most transformative developments in the recent years has been the emergence of Transformer-based models [4], and in particular, *Large Language Models* (LLMs). These models have redefined what AI systems can do—achieving remarkable results in *Natural Language Processing* (NLP) tasks, code generation, planning, and decision making. But could their success extend beyond language? That is, could we leverage LLMs in systems that interact with and control the physical world — like those behind autonomous vehicles, medical devices, or robotic platforms?

On the other hand, traditional control systems operate within well-defined boundaries. They rely on mathematical models, tuned controllers, feedback mechanisms, and known specifications. In these environments, control strategies are typically engineered with great care and rely on clear system dynamics and target behavior. However, when the environment becomes uncertain, nonlinear, or partially unknown—due to factors like sensor noise, unexpected disturbances, or evolving dynamics—these handcrafted strategies may struggle to adapt. This is a major challenge that has given rise to a growing interest in combining AI and control, leading for example to learning-based approaches such as *Reinforcement Learning* (RL), where a model interacts with the environment and learns control policies to optimize its performance [5] [6]. While powerful, such approaches often require extensive training and are sensitive to changes in dynamics.

But what if we could go further? What if we could leverage the generalization abilities of large pre-trained Transformer-based models to handle such uncertainty without extensive retraining? These models, trained on massive amounts of data, have shown a surprising capacity for reasoning and adaptation through *in-context learning*—solving new problems simply by observing a few examples embedded in the input. Imagine a model that, when presented with the current state of a system and a desired outcome, could infer the best sequence of control actions without requiring detailed knowledge of the underlying system dynamics. Could it generalize to novel situations, like avoiding an unforeseen car accident, by drawing on prior experiences of similar—but not identical—scenarios?

In this thesis, we aim to investigate these questions. The remainder of this work can be divided as follows:

- **Theoretical Background (Chap 2) :** We introduce the main concepts of AI and control theory considered in this work, their current methods and algorithms, how they relate to each other and how we can bridge the gap between them.
- **Reproduction of Existing Results (Chap 3) :** We begin by reproducing and verifying the experimental results presented in the paper *Can Transformers Learn Optimal Filtering for Unknown Systems?* [7]. Since this work serves as the theoretical and methodological foundation for our research, ensuring the correctness and reproducibility of their results is a necessary and rigorous starting point.
- **Supervised Learning Framework: Model and Limitations (Chap 4) :** We explore the use of Transformer-based models as controllers for dynamical systems trained in a supervised manner to perform in-context learning. Through numerical simulations, we identify key limitations and performance bottlenecks of such an approach. This will not only highlight critical challenges but also lay the groundwork for the improvements introduced in the next chapter.
- **From Supervised to Self-Supervised (Chap 5) :** We propose the core contribution of this thesis: a self-supervised in-context learning framework for Transformer-

based control. This novel method allows the model to learn optimal control laws purely through interaction with the system, by minimizing a quadratic cost function over trajectories—without requiring access to labeled supervision. This framework provides a more flexible and scalable alternative for control in uncertain or dynamic environments.

1.1 Related Work

The application of Transformer-based models to dynamical systems has gained increasing attention in recent years. Several works have explored their in-context learning (ICL) capabilities for auto-regressive forecasting of system states or outputs [8, 9, 10]. In particular, the study in [9] provides a theoretical framework for understanding ICL as a form of implicit algorithm learning, in which a Transformer constructs a predictive function at inference time. Their analysis, grounded in multi-task learning theory, links generalization ability to the stability of the learned algorithm. Empirical results support these findings, demonstrating that Transformers can achieve near-optimal performance on both classical regression tasks and time-series prediction in dynamical settings.

Building on this perspective, the paper [7] propose a novel approach for optimal output estimation in dynamical systems using Transformer-based models. In their framework, the Transformer is trained on a distribution of systems sampled from a shared class and is expected to generalize to new, unseen systems within the same family. This work establishes a methodology for system identification through in-context learning and serves as the theoretical and experimental foundation for the present thesis.

While much of the research at the intersection of AI and control theory has focused on Reinforcement Learning (RL) methods [5, 6], the explicit use of Transformer architectures for control tasks remains mainly underexplored. The most recent paper that has been found relating to this subject is [10], which presents a novel approach to *Model Predictive Control* (MPC) by leveraging Transformer architectures for trajectory optimization via sequence modeling. The authors demonstrate that their Transformer-based MPC framework can effectively handle complex control tasks, offering improved performance over traditional methods.

Chapter 2

Theoretical Background

The work presented in this thesis lies at the intersection of two traditionally distinct yet increasingly interconnected fields: *Artificial Intelligence* (AI) and *Control Theory*. Each of these domains has evolved through its own history, methodologies, and objectives—AI focusing on the development of systems capable of mimicking or augmenting human intelligence, and Control Theory focusing on the mathematical modeling and regulation of dynamic systems. Over the past decade, however, these fields have started to converge in meaningful ways. As AI systems become more capable of perception, reasoning, and decision-making, they offer promising tools for addressing the limitations of conventional control techniques, especially in uncertain, nonlinear, or high-dimensional environments. At the same time, the principles and rigor of control theory provide a solid framework to bring structure, safety, and interpretability to AI-driven decision processes.

This chapter aims to provide the necessary theoretical background on both fields, highlighting their foundations, respective strengths, and how recent advances—especially in learning-based approaches—are beginning to bridge the gap between them. By doing so, we lay the groundwork for understanding how modern AI models, particularly Transformer-based architectures, can be leveraged to support and potentially revolutionize control system design.

2.1 Artificial Intelligence

2.1.1 The Role of Machine Learning in Modern AI

The evolution of AI has given rise to a variety of sub-fields, each addressing specific challenges and needs. For instance, the sub-field of *Computer Vision* (CV) emerged to enable machines to interpret and understand visual data, a capability well-needed for applications such as autonomous driving, medical image analysis, and facial recognition.

Another sub-field - and probably the most influential one in the recent years - is *Machine Learning* (ML). It focuses on developing adaptive algorithms and models that are able to learn, take decisions and make predictions from data, without being explicitly programmed for every possible scenarios. With that comes the ability to automatically improve and adapt their performance through experience.
Related fields such as *Data Sciences* (DS) rely on ML techniques to analyze and extract patterns from large datasets. *Natural Language Processing* (NLP), although historically rooted in symbolic and statistical methods, is now also considered a sub-field of ML due to the impressive success of data-driven deep learning models in tasks like machine translation, question answering, sentiment analysis, and summarization. Similarly, while early computer vision systems were built upon principles of geometry, signal processing, and handcrafted features,

modern approaches predominantly leverage ML models—especially convolutional neural networks (CNNs) and, more recently, vision transformers (ViTs) [11] [12]—to solve visual recognition tasks more robustly and efficiently.

It is thus evident that Machine Learning has become a foundational component across most AI domains. At the heart of ML lies the notion of *learning* from data [3], which can take multiple forms depending on the nature of the task and the type of supervision available. The main categories of learning are described below:

- **Supervised Learning:** where an algorithm is trained on a labeled dataset. Each input is paired with a corresponding output (target), and the goal is to learn a mapping from inputs to outputs, allowing the model to generalize to unseen data.
- **Unsupervised Learning:** where the data is unlabeled, and the goal is to identify hidden patterns or structures within the data. *Clustering* and dimensionality reduction techniques such as *Principal Component Analysis* (PCA) are typical applications. This type of learning is useful when manual labeling is costly or impractical.
- **Self-Supervised Learning:** where pseudo-labels are created from seeing the data, allowing the model to learn meaningful representations without the need for human-annotated labels. Self-supervised learning lies between supervised and unsupervised learning and has gained significant attention by leveraging large-scale unlabelled datasets.
- **Reinforcement Learning:** where an agent learns to make sequential decisions by interacting with an environment. It receives feedback in the form of rewards and learns a policy to maximize long-term cumulative reward.

2.1.2 Deep Learning and Transformer Architecture

Deep Learning (DL) refers to a subset of machine learning techniques that approximate functions using deep, multi-layered neural networks. These networks can be *feedforward*—processing input in a single direction—or *recurrent*—incorporating feedback to model temporal dependencies.

Recurrent structures like RNNs and LSTMs became essential for *Natural Language Processing* (NLP) problems - where the input is represented by a sequence of embedded words or word fragments - enabling for instance systems like Apple's Siri, released in 2011, to handle sequences of human language. They use what is called their internal state (memory) to process sequences of inputs. That way, they can model sequence of data so that each sample can be assumed to be dependent on previous ones. However, RNNs suffer from vanishing gradient and, while LSTM takes care of that problem, they both process sentences sequentially, i.e. word by word, making them unsuitable for handling dependencies in very long sequences. The other popular networks during that period, the CNNs, were also not suited for overcoming those limitations. Indeed, unlike images, text sequences vary in length, making traditional *fully connected neural networks* (FCNNs) inefficient due to their fixed input size and quadratic parameter growth. Applying padding would introduce meaningless zeros, increasing memory/computation costs. Truncating the inputs would on the other side risk losing key information, and resizing is not as straight forward as it is for images.

The Transformer architecture, introduced in 2017, brought a paradigm shift in NLP by overcoming these limitations. The reason is that it totally avoids recursion by enabling parallel computation and parameter sharing. Moreover, it learns long-range relationships between words using multi-head self-attention mechanisms and positional embeddings. A common

attention mechanism is applied across input positions in order to ensure scalability and consistency.

The core innovation that enables Transformers to achieve remarkable performance - and to overcome the limitations mentioned here-above - is the *Dot-Product Self-Attention* mechanism [13] [4]. This operation is computer within a key part of each Transformer layer: the *Multi-Head Self-Attention* (MhSa) unit. The other main unit operating in a Transformer layer is a fully connect network (MLP), applied separately on each word. Both units are equipped with residual connections - their output is added back to the original input, e.g. $X \leftarrow X + \mathbf{MhSA}[X]$ - and are typically followed by a LayerNorm operation to ensure stable training [14]. A complete visual representation of the architecture of a Transformer layer is displayed in Figure 2.1.

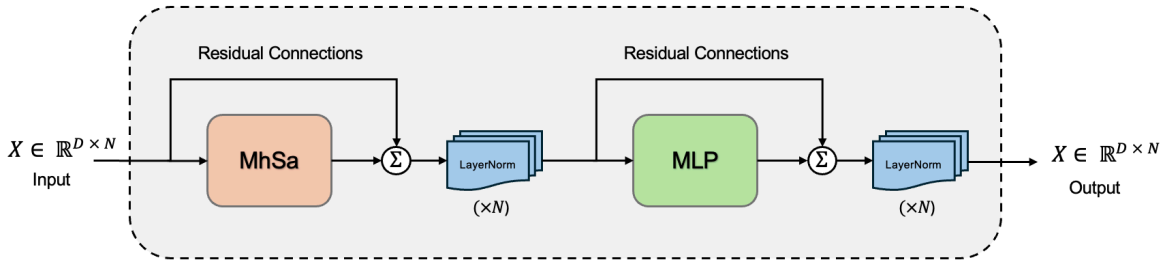


Figure 2.1: Architecture of a Transformer layer. The input X is a $D \times N$ matrix with D the dimension of word embeddings and N the number of tokens. The output is a matrix of the same size.

The MHSA module applies H parallel *Self-Attention* (SA) heads to the input, each designed to learn distinct representations. The outputs of the H heads are concatenated and linearly projected via a matrix Ω_c:

$$\mathbf{MHSA}[X] = \Omega_c \left[\mathbf{SA}_1[X]^T, \mathbf{SA}_2[X]^T, \ldots, \mathbf{SA}_H[X]^T\right]. \tag{2.1}$$

Each self-attention head operates as follows. Given N input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^D$, which form the columns of the matrix $X \in \mathbb{R}^{D \times N}$, the output of the h-th head is computed as:

$$\mathbf{SA}_h[X] = V_h[X] \cdot \mathbf{Softmax}\left(\frac{K_h[X]^T Q_h[X]}{\sqrt{D_q}}\right), \tag{2.2}$$

where $Q_h[X]$, $K_h[X]$, and $V_h[X]$ are the *queries*, *keys*, and *values*, respectively, obtained by linear transformations of the input:

$$\begin{aligned} Q_h[X] &= \beta_{qh}\mathbf{1}^T + \Omega_{qh}X, \\ K_h[X] &= \beta_{kh}\mathbf{1}^T + \Omega_{kh}X, \\ V_h[X] &= \beta_{vh}\mathbf{1}^T + \Omega_{vh}X, \end{aligned}$$

with $\mathbf{1} \in \mathbb{R}^{N \times 1}$ a vector of ones. The dimension $D_q = D_k$ denotes the size of the queries and keys, and the scaling factor $\sqrt{D_q}$ ensures that the dot-product values remain within a numerically stable range.

One of the key strengths of the self-attention mechanism is its ability to allow each token to selectively attend to all others in the sequence, thereby capturing contextual dependencies

across arbitrary distances. This property is particularly powerful in modeling long-range relationships that are challenging for traditional sequence models like RNNs or CNNs.
However, self-attention also introduces a fundamental limitation: the computation is *equivariant* to permutations of the input tokens. In other words, it treats the inputs as an unordered set, overlooking the sequential nature of the data. This is problematic for tasks like language modeling or time series prediction, where word or event order is critical.

To address this, Transformers inject information about token positions into the input representations via *positional embeddings*, which can be either *absolute* (e.g., sinusoidal functions or learned vectors) or *relative* (e.g., distance-based embeddings between tokens). These embeddings are typically added to the input embeddings before entering the first Transformer layer, enabling the model to differentiate tokens not just by their identity but also by their position in the sequence. If the total embedding dimension is D and there are H attention heads, each head typically operates on a subspace of dimension D/H. This design allows each head to specialize in capturing different patterns of interactions across the sequence. Consequently, the MHSA module produces richer and more expressive representations than a *Single-Head Self-Attention* (SHSA) module.

However, this expressivity comes at a cost: each head has its own set of parameters, significantly increasing the total number of weights. Additionally, each attention head requires computation of a full $N \times N$ attention matrix, resulting in a quadratic time and space complexity with respect to the sequence length N. A formal derivation of this complexity can be found in [13].

Depending on the task, different variants of the Transformer architecture are used: *encoder-only*, *decoder-only*, and *encoder-decoder* [13]. Each of these architectures leverages the self-attention mechanism differently to process inputs and generate outputs.

Encoder-only models. These models consist solely of Transformer encoder layers and are typically used for understanding tasks such as classification or regression. BERT (Bidirectional Encoder Representations from Transformers) is a representative encoder-only architecture. It allows each token to attend to all others in the input simultaneously (i.e., full self-attention) and is designed to produce contextual embeddings that capture bidirectional dependencies.

Decoder-only models. These models, such as GPT-2 and GPT-3, consist only of Transformer decoder layers and are used for generative tasks like language modeling, text generation, or autoregressive prediction. A general visual representation is given in Figure 2.2. In this setting, the model predicts the next token in a sequence given the previous ones. To enforce causality and prevent the model from seeing future tokens during training, a *causal mask* is applied in the self-attention mechanism. This ensures that for each position t, the token can only attend to positions $\leq t$ which makes decoder-only Transformers naturally suited for sequence generation and forecasting tasks. This is denoted by "Masked MhSA" in the figure below.

In this work, we will focus on the decoder-only architecture, specifically GPT-2, due to its effectiveness in autoregressive modeling and its simplicity compared to full encoder-decoder designs.

Encoder-decoder models. These models combine both encoder and decoder components and are primarily used for sequence-to-sequence tasks such as machine translation or summarization. The encoder processes the input sequence into a contextual representation, which

the decoder then uses, through a cross-attention mechanism, to generate the output sequence. The Transformer model proposed by Vaswani et al. [4] is an encoder-decoder model and forms the basis for architectures such as T5 and BART.

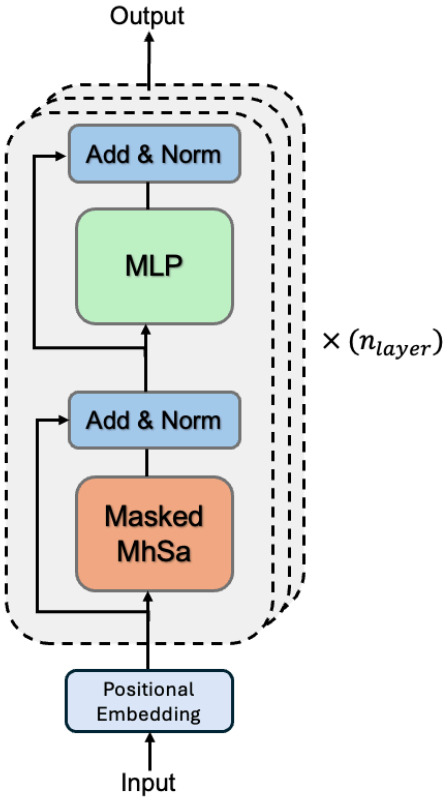


Figure 2.2: Illustration of decoder-only Transformer architectures consisting of n_{layer} Transformer layers with a masked multi-head self-attention unit, residual connections and normalization layers, and finally a fully connected neural network

2.1.3 In-Context Learning

The self-attention mechanism introduced in the layers of the Transformer architecture enabled what is called *in-context learning*(ICL). This approach allows for instance pre-trained LLMs to address new tasks without requiring fine-tuning of their parameters.

Although not yet fully understood, this type of learning - currently only observed in Transformer-based models - empowers a model with the ability to solve new tasks by being conditioned on a few examples only, embedded in the input prompt. In other words, they can construct new predictors from sequences of labeled examples $(x, f(x))$ presented within their context window without modifying the internal weights. This stands in contrast to traditional learning approaches, which typically require parameter updates via backpropagation when adapting to new tasks.

Unlike conventional machine learning models that are trained for specific tasks and whose generalization capabilities are limited by the training data, Transformer-based models with ICL can generalize to a broader class of tasks with minimal examples. This form of learning allows for *zero-shot* or *few-shot* generalization—learning to perform a task "on the fly" by interpreting examples provided in the prompt. This makes ICL both more computationally efficient and more flexible at inference time, enhancing generalization without retraining.

However, it relies on very large pre-trained models endowed with a form of *meta-learning*, which will be discussed in the next section.

To better understand this phenomenon, several studies have investigated the behavior of Transformer models on tractable mathematical problems, such as solving linear systems [15] and performing regression in context [16]. The latter empirically examined the ability of Transformer architectures to in-context learn various function classes. When trained on random instances of linear regression, the model's predictions closely matched those of ordinary least squares. Furthermore, the study showed that Transformers can in-context learn two-layer ReLU networks and decision trees, demonstrating that when trained on structurally diverse data, Transformers are capable of implementing distinct learning algorithms. Additional works have further analyzed the nature of algorithms learned by Transformers in the context of linear models.

2.1.4 Meta-Learning

Although the self-attention mechanism is likely a key factor enabling Transformer-based models to generalize effectively—without requiring fine-tuning or updates to their internal weights when encountering new problems—these models must still have learned how to quickly interpret the structure and relationships within sequences. This ability stems from their initial training on vast and diverse datasets.

The kind of training that allows such generalization is known as *meta-learning*, where the model is explicitly optimized for adaptability, generalization, and few-shot learning. This contrasts with traditional machine learning, where a model is trained on a fixed dataset for a single specific task, often limiting its performance outside that task. In meta-learning, by contrast, the objective is to learn a learning algorithm itself, capable of handling an entire *family* of tasks—each with its own dataset—rather than a single one [8]. This process is often referred to as "learning to learn" [17]: improving a learning algorithm over multiple training episodes. Through these episodes, the model acquires the ability to generalize across tasks and adapt quickly to new scenarios, even when only limited data is available—an ability also discussed in the previous section on in-context learning.

Like conventional machine learning, meta-learning typically involves a *meta-training* phase, during which the model (often called the base learner) is exposed to a wide range of tasks [18]. It learns to identify shared structure and general patterns across these tasks, effectively acquiring a form of transferable knowledge. This is followed by a *meta-testing* phase, in which the model is evaluated on entirely new tasks it has not encountered before. The model's performance during meta-testing provides insight into how well it has learned to generalize and adapt. Throughout both phases, the model's weights are updated to explicitly optimize for these two key aspects: adaptability and generalization across tasks. A common requirement for effective meta-learning is that both training and testing tasks are drawn from the same distribution or task family. That is, the tasks used during meta-training must sufficiently represent the variety and structure of the tasks expected at test time.

2.2 Feedback Systems & Control Engineering

Feedback systems lie at the heart of control engineering, enabling robust and automated responses across a wide variety of applications—from cruise control in vehicles and thermostats in homes to the adaptive brightness settings of smartphones. These systems rely on the concept of measuring outputs, comparing them to a reference, and adjusting inputs to reduce errors—forming what is known as a closed-loop system.

Control engineering, as a formal discipline, provides the mathematical and algorithmic tools to design such systems. Classical approaches rely on modeling the underlying dynamics of a physical system using differential equations, and designing controllers to ensure stability, performance, and robustness. Over time, the field has developed a rich set of methodologies supported by strong theoretical foundations and proven effectiveness in real-world applications, especially in the industrial, aerospace, and automotive fields.

2.2.1 Optimal Filtering

In control systems, the design of a controller often assumes that the internal states of the system are fully known. However, in practice, states are usually not directly measurable or are corrupted by random noise and uncertainties. When the exact state is unknown, the controller must rely on estimates derived from imperfect and noisy sensor measurements. This necessitates the use of *optimal filtering* techniques to infer the best possible estimate of the system states given available data.

Optimal filtering for dynamical systems extends the general concept of *filtering*, which is initially referred to the notion of something passing a barrier for the purpose of separating what is desired from what might be called "contamination". In this case, filtering aims to reconstruct the true state of a dynamical system by filtering out information from a mathematical model contaminated with noisy observations, therefore enabling effective feedback control based on accurate state knowledge.

Optimal filtering extends the general concept of filtering, which involves separating a desired signal from noise or contamination. In dynamical systems, this means estimating the true system state by combining a mathematical model with noisy observations, effectively extracting accurate information to enable reliable feedback control.

Several filtering approaches exist, ranging from classical methods such as the Kalman Filter for linear Gaussian systems to more advanced nonlinear and adaptive filtering methods. In general, those methods assume that the model is known and the key question they address is

Given a stochastic system observed through noisy measurements, how can we construct an optimal estimate of the hidden state?

This section focuses primarily on the Kalman Filter, the cornerstone of optimal filtering for linear dynamical systems. It provides a recursive algorithm that fuses system model predictions with sensor data to minimize the estimation error variance under Gaussian noise assumptions. Extensions to nonlinear systems and more complex noise models are briefly discussed.

Kalman Filter

In the context of dynamical systems, uncertainty is an inherent characteristic—arising from sensor noise, model inaccuracies, and limited state observability. Among those challenges, the Kalman Filter serves as an effective algorithm for estimating the state of a system, offering clarity in the presence of stochastic disturbances.

Originally developed by Rudolf E. Kalman in the 1960s, this algorithm provides an optimal solution—under linear-Gaussian assumptions—for estimating the hidden state of a dynamical system. It combines two powerful sources of information:

- The **predictive power** of a system's model, and
- The **corrective insight** of noisy observations.

Through a seamless cycle of prediction and correction, the Kalman Filter enables accurate, real-time state estimation, and serves as a foundational block in modern control theory, signal processing, navigation systems, and beyond.
A basic representation of the mechanism behind the Kalman filter is given next, for the following - general - dynamical system:

$$\begin{cases} \mathbf{x}_{k+1} &= A\mathbf{x}_k + B\mathbf{u}_k + \mathbf{w}_k \\ \mathbf{y}_k &= C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{v}_k \end{cases} \tag{2.3}$$

where $\mathbf{x}_k \in \mathbb{R}^n$ is the state vector, $\mathbf{u}_k \in \mathbb{R}^m$ the control input, $\mathbf{y}_k \in \mathbb{R}^l$ the observation vector. The process and measurement noises are $w_k \sim \mathcal{N}(0, Q)$ and $v_k \sim \mathcal{N}(0, R)$ respectively At each time step k, the Kalman Filter performs two main steps: prediction and update.

1. Prediction Step:

$$\hat{\mathbf{x}}_{k|k-1} = A\hat{x}_{k-1|k-1} + Bu_{k-1} \tag{2.4}$$
$$P_{k|k-1} = AP_{k-1|k-1}A^\top + Q \tag{2.5}$$

   where:

   - $\hat{x}_{k|k-1}$: predicted state estimate
   - $P_{k|k-1}$: predicted error covariance

2. Update (Correction) Step

$$K_k = P_{k|k-1}C^\top (CP_{k|k-1}C^\top + R)^{-1} \tag{2.6}$$
$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \left(y_k - C\hat{x}_{k|k-1} - Du_k\right) \tag{2.7}$$
$$P_{k|k} = (I - K_k C)P_{k|k-1} \tag{2.8}$$

   where:

   - K_k: Kalman gain
   - $\hat{x}_{k|k}$: updated state estimate
   - $P_{k|k}$: updated error covariance

The intuition behind the prediction step is that it used the model to project the state forward. Then, the estimation is corrected by incorporating the measurement. More information and complete proves about the mechanics behind that algorithm can be found in [19].
However, despite its wide applicability and optimality in linear-Gaussian settings, the Kalman Filter has important limitations. Its performance significantly degrades when:

- The system dynamics are nonlinear or unknown,
- The noise is non-Gaussian or time-correlated,
- The underlying model structure changes over time.

Extensions like the Extended Kalman Filter (EKF) or Particle Filters attempt to address some of these limitations. For example, the EKF deals with non-linear systems by linearizing its equations around the current estimate. But very often those methods require additional assumptions, are sensitive to initial conditions, or impose considerable computational cost.

These shortcomings have raised interest in data-driven approaches, particularly in the use of Transformer-based models. Recent research explores whether such models can learn to perform state estimation by leveraging large amounts of observed trajectories—without explicit access to system dynamics. In particular, [7] demonstrates the ability of Transformer architectures to perform accurate system output prediction in complex, non-Markovian environments. Similarly, [9] investigates theoretical properties of in-context learning using Transformers, focusing on generalization guarantees under i.i.d. and Markovian data.

Together, these works mark a promising direction: using Transformers not only for NLP, but as powerful approximators for inference and decision-making in dynamical systems—potentially overcoming the rigidity of classical filtering techniques.

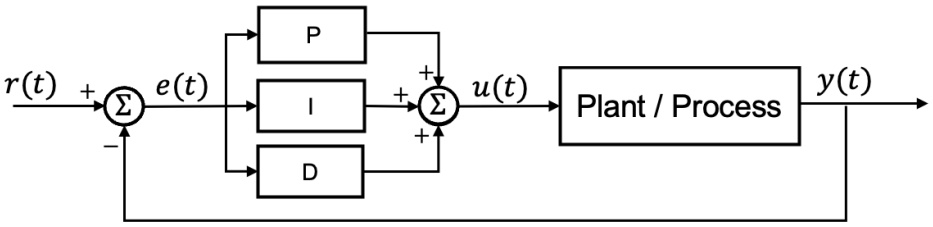


Figure 2.3: Block diagram of a PID (Proportional-Integral-Derivative) controller in a standard feedback loop. The controller takes the error signal $e(t)$ and applies a weighted combination of proportional, integral, and derivative terms to generate the control input $u(t)$. This input drives the plant to minimize the error and improve system performance over time.

2.2.2 Proportional-Integral-Derivative (PID)

Proportional-Integral-Derivative (PID) controllers are among the most widely used feedback controllers for linear and nonlinear systems in engineering due to their simplicity and effectiveness, and are particularly used in industrial automation. A PID controller generates a manipulated variable - the control input $u(t)$ - based on the error $e(t)$, defined as the difference between the set point and the actual process variable - the output $y(t)$. A general block diagram illustrating this controller is given in Figure (2.3), and Equation (2.9) gives its mathematical expression.

$$u(t) = \underbrace{K_p e(t)}_{\text{P}} + \underbrace{K_i \int_0^t e(\tau)d\tau}_{\text{I}} + \underbrace{K_d \frac{de(t)}{dt}}_{\text{D}}, \tag{2.9}$$

where K_p, K_i, and K_d are the proportional, integral, and derivative gains, respectively. The three control actions in the above equation contribute differently to the overall behavior of the

closed-loop system. The *proportional* (P) term provides an immediate response proportional to the current error, offering fast correction but potentially causing overshoot if too large. The *integral* (I) term accumulates past errors to eliminate steady-state offsets, though excessive use may slow the response and cause overshoot. The *derivative* (D) term anticipates future error trends by considering the rate of change, adding a damping effect that improves stability and reduces oscillations [20] [21].

Together, these three components allow the PID controller to react to the present, learn from the past, and anticipate the future, making it a versatile and widely used control strategy for, e.g., regulating the temperature, the pressure, the motor speed and any other variable used in various industries. However, optimal tuning of the gains can be challenging and if they are poorly selected, PID control can lead to an unstable closed-loop system, especially in complex linear system.

2.2.3 Linear Quadratic Regulator (LQR)

As mentioned, the objective of a controller operating a dynamical system is to minimize a cost function. The case where the system dynamics are described by a set of linear difference equations

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k, \quad k \in \{0, 1, ..., N-1\} \tag{2.10}$$

where x_0 is given, and the cost is described by a quadratic function

$$J_N(U) = \frac{1}{2}\sum_{k=0}^{N-1}[\mathbf{x}_k^T Q\mathbf{x}_k + \mathbf{u}_k^T R\mathbf{u}_k] + \frac{1}{2}\mathbf{x}_N^T Q_N \mathbf{x}_N \quad , \tag{2.11}$$

is called the *LQ problem*. The goal of the *linear quadratic regulator* (LQR) is to minimize that functional w.r.t. the control input sequence $U := (\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{N-1})$, where N is the time horizon, Q is the state cost matrix, R is the control cost matrix and Q_N is the terminal cost matrix. The state term $\mathbf{x}_t^T Q\mathbf{x}_t$ measures the state deviation cost at time $k = t$, and thus penalizes deviations from the desired state $\mathbf{x}^* = 0$, while the control term $\mathbf{u}_t^T R\mathbf{u}_t$ measures the input authority cost at time$k = t$ and thus penalizes large control efforts. Finally, $\mathbf{x}_N^T Q_N \mathbf{x}_N$ measures the terminal cost.

The optimal control sequence minimizing (2.11) is given by the following state feedback law:

$$\mathbf{u}_k = -K_k\mathbf{x}_k, \quad k \in \{0, 1, ..., N-1\} \tag{2.12}$$

where the gain matrix K_k is computed by solving the following equation:

$$K_k = \underbrace{\left(R + B^\top S_{k+1} B\right)^{-1}}_{P_{k+1}^{-1}} B^\top S_{k+1} A. \tag{2.13}$$

The matrix S_k is computed backwards in time using the discrete-time Riccati recursion:

$$S_k = A^\top [S_{k+1} - S_{k+1} B P_{k+1}^{-1} B^\top S_{k+1}] A + Q, \tag{2.14}$$

with terminal condition:

$$S_N = Q_N. \tag{2.15}$$

Plugging the solution of (2.14) back in (2.13) gives a sequence of optimal gains $\{K_k\}_{k=0}^{N-1}$, which define the optimal control sequence in (2.12) that minimizes the quadratic cost J_N subject to the system dynamics (2.10). A complete proof of this process using a *Dynamic Programming* (DP) framework can be found at [22].

2.2.4 Model Predictive Control (MPC)

The LQR controller, although presented as optimal so far for linear systems without noise, can face some limitations. For instance, how do we tell it to maintain a voltage input between -5 and 5 volts when moving a drone to a desired state? If we do not, it might compute a control signal of 7 volts, but that input will either be clipped or saturated by the drone's hardware. Such saturation can degrade performance or even destabilize the system, and this situation is therefore not desired. Another example arises in autonomous vehicles, where not only the control inputs but also the system states must remain within safety limits. For instance, a self-driving car must remain within lane boundaries and avoid obstacles. The standard LQR framework is not designed to enforce such *state constraints*.

It is clear that the LQR does not explicitly handle any constraint in its standard formulation. It assumes that the optimal control action can take any value in $\mathbb{R}^n$ as well as the state of the system. All that "really" matters is the minimizing of the loss function, which can therefore be very problematic in real-world systems where actuators have physical limits as mentioned in the two examples. Moreover, LQR generates a fixed feedback law optimized over that function. It might therefore not support trajectory planning in time-varying environments of when facing abrupt and sudden change or disturbance. Suppose the drone or the self-driving car must follow a specific path while avoiding obstacles — LQR will struggle in such scenarios.

These practical limitations motivate the use of more advanced control strategies capable of explicitly handling both input and state constraints, adapting to changing objectives, and operating in dynamic environments. This leads us to *Model Predictive Control* (MPC), a powerful framework where an optimization problem is solved at each time step to compute an optimal control sequence, taking into account the full set of constraints and future reference trajectories.

At each time step k, MPC solves online an optimization problem that minimizes a cost function over a finite prediction horizon N, based on the current state $\mathbf{x}_k$ and predicted future states and inputs:

$$\min_{\mathbf{u}_k,\dots,\mathbf{u}_{k+N-1}} \quad \frac{1}{2}\sum_{i=1}^{N} (\mathbf{y}_{\text{ref}} + \mathbf{y}_{k+i})^2 \tag{2.16}$$

$$\text{s.t.} \quad \mathbf{x}_{k+i+1} = f(\mathbf{x}_{k+i}, \mathbf{u}_{k+i}), \quad \forall i = 0, \dots, N-1 \tag{2.17}$$

$$\mathbf{x}_{k+i} \in \mathcal{X}, \quad \mathbf{u}_{k+i} \in \mathcal{U}, \quad \forall i = 0, \dots, N-1, \tag{2.18}$$

where f represents the system dynamics, and $\mathcal{X}$, $\mathcal{U}$ define admissible sets for the state and control inputs. In the case of linear systems where the references are set to zero, the problem becomes

$$\min_{\mathbf{u}_k,\dots,\mathbf{u}_{k+N-1}} \quad \frac{1}{2}\sum_{i=0}^{N-1} \left(\mathbf{x}_{k+i}^\top Q \mathbf{x}_{k+i} + \mathbf{u}_{k+i}^\top R \mathbf{u}_{k+i}\right) + \frac{1}{2}\mathbf{x}_{k+N}^\top Q_N \mathbf{x}_{k+N} \tag{2.19}$$

$$\text{s.t.} \quad \mathbf{x}_{k+i+1} = A\mathbf{x}_{k+i} + B\mathbf{u}_{k+i}, \quad \forall i = 0, \dots, N-1 \tag{2.20}$$

$$\mathbf{x}_{k+i} \in \mathcal{X}, \quad \mathbf{u}_{k+i} \in \mathcal{U}, \quad \forall i = 0, \dots, N-1, \tag{2.21}$$

Only the first input of the optimal sequence solution of the optimization problem is applied to the system, and the optimization is repeated at the next time step with updated state information—a process known as *receding horizon control*.

[1]Graph taken from lecture slides of the course *H0M82A - Methods and Algorithms for Advanced Process Control*, KU Leuven

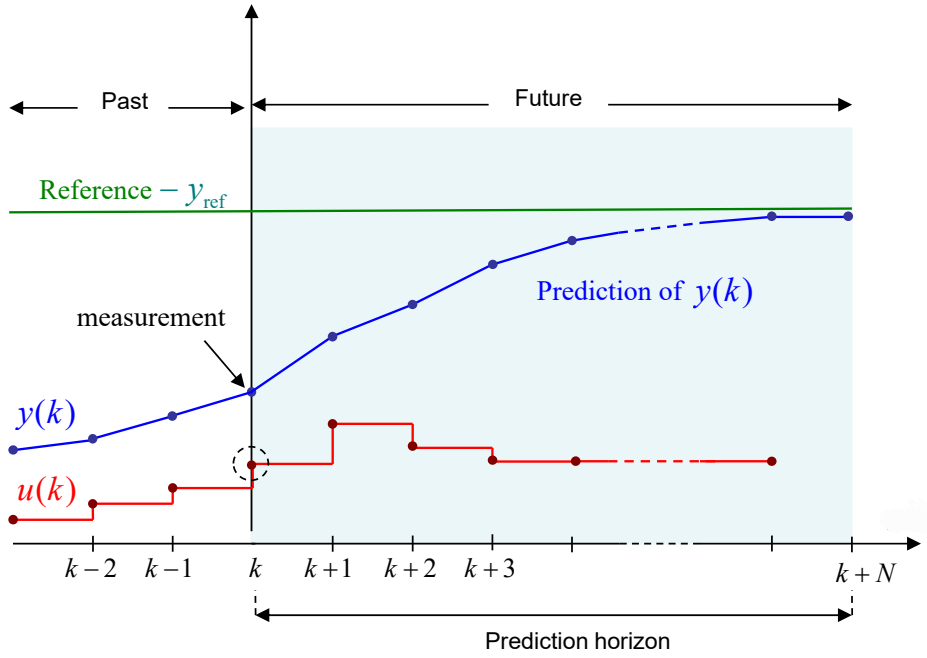


Figure 2.4: Illustration of the principle of Model Predictive Control (MPC). At each control step, the controller uses a dynamic model of the system to predict future behavior over a finite time horizon N. It then solves an optimization problem to find a sequence of control inputs that minimizes a predefined cost function, subject to system constraints. Only the first control input $u(k)$ at each time step, implementing a receding horizon strategy.[1]

The strength of MPC lies in its ability to plan ahead by leveraging a predictive model, while remaining reactive to changes in the system or environment through feedback. Compared to PID and LQR, MPC is more computationally demanding, but offers superior performance and flexibility, especially in systems with multi-variable interactions, time-varying references, or operational constraints. If we remove constraints and assume linear dynamics and quadratic cost, MPC reduces to LQR with a finite horizon, and if we use a very short prediction horizon and specific weighting, MPC behavior is similar to that of a tuned PID controller.

2.2.5 Summary

To conclude this section about control theory, we can affirm that traditional control methods also come with significant limitations that restrict their application in more complex, high-dimensional, or data-driven environments like those explored in this thesis. These limitations include:

- **Model dependency**: Most classical control algorithms assume that the system dynamics are fully known and accurately modeled. In real-world scenarios, these dynamics may be only partially known or subject to change.
- **Linearity assumptions**: Controllers such as LQR and estimators such as Kalman Filters are optimal only under linear system assumptions and Gaussian noise. Their performance degrades when applied to nonlinear or stochastic systems.
- **Limited adaptability**: Classical methods lack the ability to learn from data or adapt in

real time to unforeseen dynamics, making them less suitable in uncertain or unstructured environments.

These constraints, linked with the previous theoretical foundation about AI, motivate the exploration of alternative, data-driven approaches that can operate with less prior information about the system and potentially generalize across tasks. Reinforcement learning, for instance, offers powerful tools for learning control policies directly from interaction data. Yet despite its promise, RL has seen limited adoption in industry, due in part to challenges in stability, interpretability, and sample efficiency—areas where traditional control still holds an advantage. Another emerging direction leverages Transformer-based models to approximate control and filtering tasks, as it will be explained more detail in the remainder of this thesis.

Chapter 3

Reproducing Prior Work on Transformers for System Output Prediction

3.1 Introduction

As starting point for our study on Transformers for dynamical system's optimal control, we empirically reproduce the work of [7]. The approach explored by the authors, as mentioned in Section 1.1, consists of numerically demonstrate the capabilities of a Transformer-based model to perform in-context learning in the context of dynamical system's output prediction. Following the idea of the learning frameworks introduced in Sections 2.1.4 (meta learning) and 2.1.3 (in-context learning), this model is first trained on a large set of distinct source systems drawn from a common class. Then, is it expected to adapt and generalize to previously unseen systems, but drawn from the same class as the training systems, applying zero-shot learning. The main point of this framework is that the model manages to autonomously adapt to new, unseen systems and generate output predictions using only a limited number of past observations from those systems. The authors refer to a Transformer trained that way as a *meta-output-predictor* (MOP), and this approach differs from conventional methods since they typically train a predictor using data from a single, specific system. In contrast, this model allows for generalization to new systems, provided they belong to the same class as those seen during training and that the training set sufficiently captures the class's diversity. This flexibility offers a significant advantage over traditional techniques, which often rely on prior knowledge of system dynamics — such as linearity, time-invariance, or Gaussian noise assumptions. In complex or uncertain environments, satisfying these assumptions can be challenging and potentially leading to a degradation of performance.

In the following sections, in order to make clear the link between the work in [7] and this thesis, we first explain the methodology employed in the former to empirically demonstrate the versatility of MOP and to show that the Transformer-based model reaches the optimal performance of a Kalman filter in the case of linear systems while demonstrating promising results in other settings with non-i.i.d. noise, time-varying dynamics, and nonlinear dynamics. In other words, using exactly the model from [7] as well as its experimental settings, we show that the results and findings are indeed reproducible in each case.
Furthermore, statistical guarantees provided in the original work support these findings by quantifying the amount of training data required to achieve a given level of performance. In particular, it is shown that the excess risk associated with the MOP decays at a rate of $\mathcal{O}(1/\sqrt{MT})$, where T denotes the prediction horizon and M the number of source systems used during training.

3.2 Methodology

The first contribution consists of numerically demonstrating the capabilities of MOP by conducting evaluations across several challenging scenarios. This approach assumes prior access to a collection of M source systems $\{\mathcal{S}_i\}_{i=1}^{M}$, each drawn from a common distribution denoted $\mathcal{D}_{sys}$. For the i-th system, the corresponding output trajectory is given by $\{\mathbf{y}_{i,t}\}_{t=0}^{T-1}$. During training, the model receives past outputs $\mathbf{y}_{i,0:t-1}$ from these source trajectories and learns to generate an estimate $\hat{\mathbf{y}}_{i,t}$ of the true output $\mathbf{y}_{i,t}$, as illustrated in Figure 3.1. In the testing phase, the trained transformer is evaluated on a previously unseen system, also sampled from $\mathcal{D}_{sys}$, by feeding its trajectory into the model and assessing the quality of the predicted outputs with respect to the true outputs.

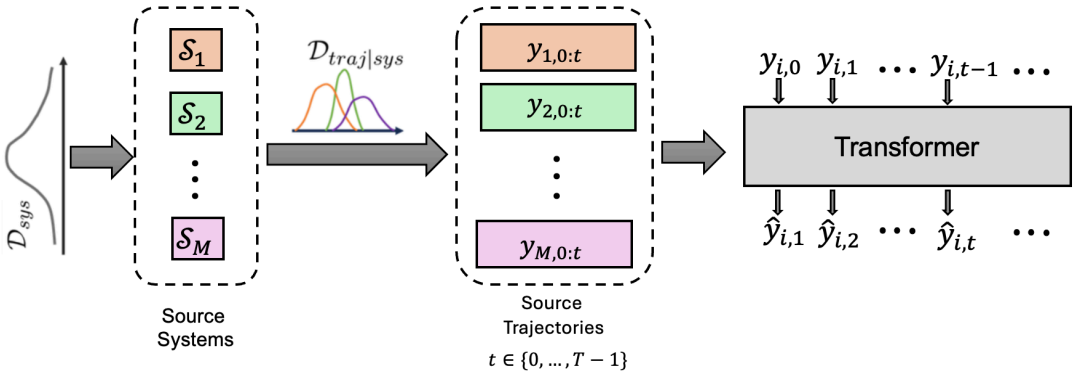


Figure 3.1: Training a transformer model to predict the next output of a dynamical system based on its past observed trajectory. The model is trained on a distribution of M source systems and evaluated on unseen systems sampled from the same distribution $\mathcal{D}_{sys}$.

A state-space representation of the M source systems is expressed as follows, for $i \in \{1, \cdots, M\}$:

$$\mathcal{S}_i : \begin{cases} \mathbf{x}_{i,t+1} = f_i(\mathbf{x}_{i,t}) + \mathbf{w}_{i,t+1} \\ \mathbf{y}_{i,t} = g_i(\mathbf{x}_{i,t}) + \mathbf{v}_{i,t} \end{cases}, \tag{3.1}$$

where $\mathbf{x}_{i,t} \in \mathbb{R}^{n_x}$ and $\mathbf{y}_{i,t} \in \mathbb{R}^{n_y}$ are the state and output vectors at the t-th time step for the i-th system. The functions $f_i(.)$ and $g_i(.)$ are the state and output dynamics, with $f_i(0) = g_i(0) = 0$. In particular, for simplicity it is assumed that $\mathbf{x}_{i,0} = 0$. The process and output noises are defined by $\mathbf{w}_{i,t} \sim \mathcal{N}(0, \sigma_{i,w}^2 \mathbf{I}_{n_x})$ and $\mathbf{v}_{i,t} \sim \mathcal{N}(0, \sigma_{i,v}^2 \mathbf{I}_{n_y})$ respectively - the matrix $\mathbf{I}$ denotes the identity matrix - and are mutually independent for all system i and time step t.

It is further assumed that the function $g_i(\cdot)$ is Lipschitz continuous with a constant $L_g > 0$, meaning that, for every system, the inequality $\|g_i(\mathbf{x}) - g_i(\mathbf{x}')\| \leq L_g \|\mathbf{x} - \mathbf{x}'\|$ holds for all states $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^{n_x}$.
Moreover, the state evolution functions $f_i^{(t)}$ of all systems are assumed to be contractive, by enforcing that the following stability condition is satisfied by all M source systems.

Assumption 1 (*Incremental Stability*) *: Let $f_i^{(t)}(\cdot, \cdot)$ denote the t-step state evolution function such that $f_i^{(t)}(\mathbf{x}_{i,\tau}, \mathbf{w}_{i,\tau+1:\tau+t}) = \mathbf{x}_{i,\tau+t}$ for all $\tau \in \mathbb{N}$. Then, there exists constants $\rho \in [0, 1)$ and $C_\rho > 0$ such that for any system i and time step t, for any $\mathbf{x}, \mathbf{x}'$*

and noise sequence $\mathcal{W} := \{\mathbf{w}_{(\tau)}\}_{\tau\in[t]}$*, we have*

$$||f_i^{(t)}(\mathbf{x}, \mathcal{W}) - f_i^{(t)}(\mathbf{x}', \mathcal{W})|| \leq C_\rho \rho^t ||\mathbf{x} - \mathbf{x}'||. \tag{3.2}$$

For instance, when considering linear systems, the authors scale the state matrix A such that it has a spectral radius of 0.95, ensuring that the stability condition (3.2) is satisfied.

More generally, Assumption 1 requires that the state function must be contractive, meaning that small differences in initial conditions do not grow unbounded over time. It is a desirable property for the systems, making them stable and leading repeated application of the state function $f_i(\cdot)$ to a unique fixed point. It therefore guarantees the convergence of iterative methods like the Kalman Filter or the auto-regressive predictor, and makes it easier for a Transformer to learn overtime the mapping from past system outputs to future outputs than if it had an unstable system. Indeed, with an unstable system, there could be highly unpredictable trajectories since small differences in the initial conditions can lead to very different output values, and the complexity of the set of functions that the transformer needs to learn would increase significantly, making it harder for the transformer to learn the optimal mapping based on a finite number of training systems, and therefore giving larger prediction errors.

Training the Transformer such that it can learn to predict the correct system output means that we want to train its internal weights, denoted by $\theta \in \Theta$ for some parameter set Θ, in such a way that, at each time step t, it minimizes a certain loss function. More precisely, the transformer can be seen as a deep sequence model - denoted $\texttt{TF}_\theta(\cdot)$ in the original work - and the estimation of the true system output $\mathbf{y}$ at time $t+1$ is thus given by $\hat{\mathbf{y}}_{t+1} := \texttt{TF}_\theta(\mathcal{Y}_t)$, with $\mathcal{Y}_t := \mathbf{y}_{0:t}$. The loss function that is minimized is denoted $l(\cdot,\cdot) \geq 0$, and the transformer is trained by solving

$$\widehat{\texttt{TF}} = \text{argmin}_{\texttt{TF}\in\mathcal{A}} \frac{1}{MT} \sum_{i=1}^{M} \sum_{t=1}^{T} l(\mathbf{y}_{i,t}, \texttt{TF}(\mathcal{Y}_{i,t-1})) \tag{3.3}$$

where $\mathcal{A} := \{\texttt{TF}_\theta : \theta \in \Theta\}$ represents the *hypothesis class* of the Transformer - that is, the set of all Transformer models parameterized by θ from the parameter space Θ.

3.3 Replication and Evaluation of Original Results

In order to evaluate the performance of the proposed Transformer-based model, its generalization ability is tested across different system distributions and compared to the performance of traditional methods such as the Kalman filter and a linear auto-regressive predictor, which serve as baselines. As a reminder, the Kalman Filter uses exact system dynamics for optimal estimation, while the linear auto-regressive predictor is defined by $\hat{\mathbf{y}}_{t+1} = \alpha_1 \mathbf{y}_t + \alpha_2 \mathbf{y}_{t-1}$ and learns the parameters α_1 and α_2 online using the *Recursive Least Squares* (RLS) algorithm, with the initial covariance set to the identity matrix.

The experiments are conducted on four distinct types of systems, each designed to evaluate the model's robustness under different conditions.

First, a linear system with i.i.d. Gaussian noise represents the classical setting with well-understood stochastic disturbances.
Second, a linear system with colored (temporally correlated) noise introduces a deviation from the standard i.i.d. assumption, testing the model's ability to handle temporal dependencies

in the noise process.
Third, a linear system with time-varying dynamics assesses how well the model generalizes when the system parameters evolve over time.
Finally, a nonlinear quadrotor system provides a more complex, real-world-inspired scenario, involving nonlinear state evolution.

All four settings follow the setup detailed in Table 3.1.

In addition to the figures displaying the prediction errors over the entire horizon for each method, we also compute the total prediction error across all test systems. This error is obtained by summing over all time steps and averaging over the test set, and is given by:

$$\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\|\hat{\mathbf{y}}_{i,t}-\mathbf{y}_{i,t}\| \tag{3.4}$$

where N denotes the number of systems in the test set.
A low value of this metric indicates a strong overall predictive performance by the method under consideration.

Training Parameters	
Number of training (source) systems	20,000
Number of validation systems	5,000
Prediction trajectory length	50
Model architecture	GPT-2 (12 layers, 8 heads, 256-dim)
Number of training steps	10,000
Batch size	64
Training loss function	*Mean Squared Error* (MSE)
Evaluation Parameters	
Number of test systems	1,000
Prediction trajectory length	50
Prediction error metric	$\|\hat{y}_t - y_t\|$

Table 3.1: Summary of the training and evaluation settings used in the experiments.

In Table 3.1, the training loss function is defined by

$$\mathcal{L}_{\text{MSE}} = \frac{1}{BT}\sum_{i=1}^{B}\sum_{t=0}^{T-1}\|\hat{\mathbf{y}}_{i,t}-\mathbf{y}_{i,t}\|^2$$

where B represents the batch size, referring to the number of training systems processed simultaneously in a single forward and backward pass during optimization — in this case, 64 trajectories. Moreover, the number of training steps corresponds to the total number of parameter updates performed by the optimizer, which is set to 10,000 in the experiments.

- **Linear system with i.i.d. Gaussian noise**
  The first, simplest, experiment considered in [7] is described by the following system dynamics
  $$\mathbf{x}_{i,t+1} = A_i\mathbf{x}_{i,t} + \mathbf{w}_{i,t}, \quad \mathbf{y}_{i,t} = C_i\mathbf{x}_{i,t} + \mathbf{v}_{i,t}$$
  with $A_i \in \mathbb{R}^{n_x\times n_x}, C_i \in \mathbb{R}^{n_y\times n_x}$ randomly generated, and with dimensions $n_x = 10$, $n_y = 5$. Both process and measurement noise covariances are equal to $\sigma^2 = 0.01$.

As shown in Figure 3.2, we confirm that MOP approaches the Kalman filter's performance after approximately 20 time steps - which corresponds to the model's data-driven "burn-in" period - as stated in the original paper. This delay contrasts with the Kalman filter's immediate optimality due to its exact knowledge of the system.

Furthermore, the total average prediction errors computed for each method using (3.4) are summarized in the table below:

Kalman	**MOP**	**RLS**
25.87	26.58	28.73

Table 3.2: Average total prediction error over all test systems for *linear systems with i.i.d. Gaussian noise*. Results are reported for the Kalman filter, the MOP model, and the linear auto-regressive predictor based on Recursive Least Squares (RLS).

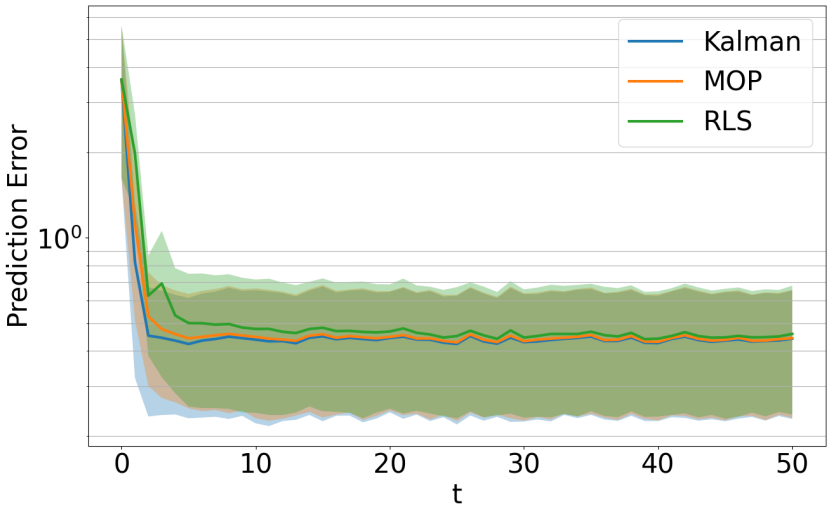


Figure 3.2: Output prediction error trajectories for *linear systems with i.i.d. Gaussian noise*.

- **Linear system with colored noise**
  The next experiment considers the case where the systems have memory through non-i.i.d. noise processes. Specifically, the process and measurement noises in (3.1) become $\mathbf{w}_t = \sum_{t'=t-4}^{t} \eta_{t'}$ and $\mathbf{v}_t = \sum_{t'=t-4}^{t} \gamma_{t'}$, where $\eta_{t'}, \gamma_{t'} \sim \mathcal{N}(0, 0.01)$.

  It is known however that the Kalman filter assumes independent noise. This new setting therefore results in suboptimal performance from this filter, as it does not account for temporal dependencies in the noise processes. This represents a key difference from the previous case, as illustrated in Figure 3.3, which shows the results obtained by reproducing the experiment from the original work. On the bright side, we notice that MOP can actually learn the non-i.i.d. noise structure from training data as it shows its optimality over all the trajectory length.

Note that, in this case, the graphical results obtained when reproducing the experiment are not entirely consistent with those presented in the original publication. Specifically, the prediction errors here decrease steadily as t increases, with a clear ranking in filter performance: MOP consistently outperforms the others, the Kalman filter stabilizes around 10^0, and the auto-regressive predictor remains between the two, though slowly improving over time. In contrast, the original results show a slight increase in all three prediction error curves before decreasing, starting around $t \approx 5$.

Table 3.3 regroups the averages total prediction errors for this experiment, as described in the previous section, clearly confirming the Transformer's dominance over the other methods.

Kalman	**MOP**	**RLS**
52.39	31.27	41.00

Table 3.3: Average total prediction error over all test systems for *linear systems with colored noise*. Results are reported for the Kalman filter, the MOP model, and the linear auto-regressive predictor based on Recursive Least Squares (RLS).

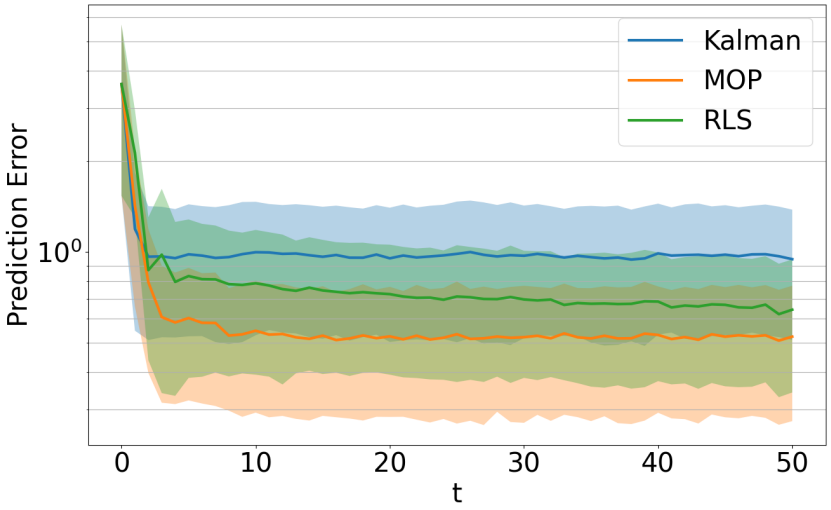


Figure 3.3: Output prediction error trajectories for *linear systems with colored noise*.

- **Linear system with i.i.d. noise and changing dynamics**
  This third experimental setting assesses the ability of MOP to adapt to run-time changes in the system dynamics. Specifically, when generating the test trajectories, the underlying dynamics are suddenly changed to new, randomly generated ones at time $t = 50$.

  As illustrated in Figure 3.4, all estimators initially show a spike in prediction error due to the abrupt change. MOP gradually adapts to the new dynamics, regaining prior performance levels by around $t = 100$. In contrast, the Kalman filter — due to its lack of adaptive mechanisms — remains suboptimal. The slower adaptation of MOP is partly due to its fixed-length context window, which initially still contains data from

the previous system.

From Table 3.4, it can be observed that although the error values are relatively close across all methods, the MOP approach yields the lowest overall prediction error. This marks a notable shift from the result of the first experiment, where the Kalman filter performed best due to its optimal handling of i.i.d. noise. In the current setting with changing dynamics, however, the Kalman filter's inability to adapt causes its performance to stagnate over time, while the more adaptive nature of MOP allows it to better track the evolving system behavior. The auto-regressive predictor remains very to MOP during the second half of the trajectory.

Kalman	**MOP**	**OLS**
50.88	49.42	51.97

Table 3.4: Average total prediction error over all test systems for *linear systems with i.i.d. Gaussian noise and changing dynamics*. Results are reported for the Kalman filter, the MOP model, and the linear auto-regressive predictor based on Recursive Least Squares (RLS).

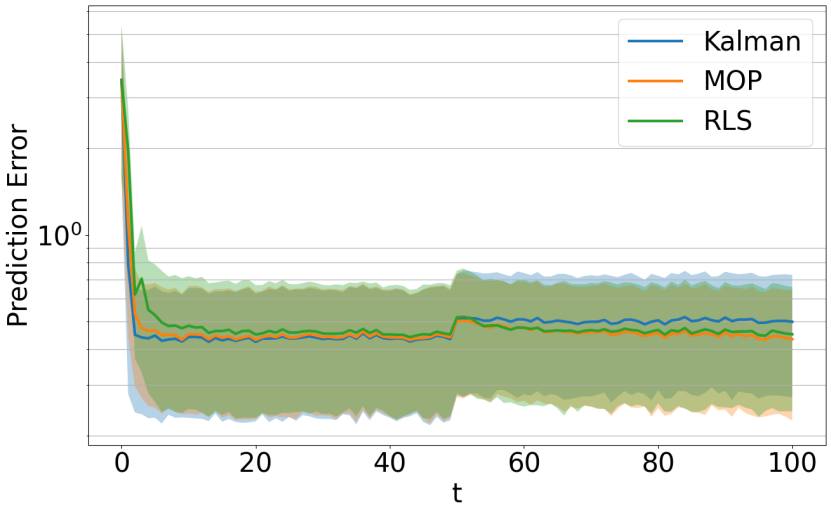


Figure 3.4: Output prediction error trajectories for *linear systems with i.i.d. Gaussian noise and changing dynamics.*

- **Nonlinear quadrotor system**
  Finally, the Transformer-based model is evaluated on a more challenging class of systems: nonlinear, underactuated planar quadrotors. The discrete-time dynamics of the systems are given by:

$$\mathbf{x}_{t+1} = \begin{bmatrix} x_{t+1} \\ z_{t+1} \\ \phi_{t+1} \\ \dot{x}_{t+1} \\ \dot{z}_{t+1} \\ \dot{\phi}_{t+1} \end{bmatrix} = \begin{bmatrix} x_t + (\dot{x}_t \cos(\phi_t) - \dot{z}_t \sin(\phi_t))\tau \\ z_t + (\dot{x}_t \sin(\phi_t) + \dot{z}_t \cos(\phi_t))\tau \\ \phi_t + \dot{\phi}_t \tau \\ \dot{x}_t + (\dot{z}_t \dot{\phi}_t - g \sin(\phi_t))\tau \\ \dot{z}_t + (-\dot{x}_t \dot{\phi}_t - g \cos(\phi_t) + (u_{0t} + u_{1t})/m)\tau \\ (u_{0t} - u_{1t}) l \tau / J \end{bmatrix} + \mathbf{w}_t$$

$$y_t = C\mathbf{x}_t + \mathbf{v}_t.$$

The state vector includes planar positions (x_t, z_t), pitch angle ϕ_t, and their corresponding velocities $(\dot{x}_t, \dot{z}_t, \dot{\phi}_t)$, all taking values in $\mathbb{R}$. The control inputs are the two thrust forces u_0 and u_1, which are also real-valued. Moreover, the systems satisfy the following configuration:

- The mass m, arm length l, and moment of inertia J are independently sampled from the interval $[0.5, 2]$.
- Process and measurement noises are drawn from $\mathcal{N}(0, 0.01)$.
- The discretization time step is $\tau = 0.1$.
- The observation matrix $C \in \mathbb{R}^{3\times 6}$ has elements sampled uniformly from $[0, 1]$.
- Inputs u_0 and u_1 are constrained to the interval $[-0.5, 0.5]$.

When the system is nonlinear, the Kalman filter considered so far is not optimal anymore. This is why the Extended Kalman Filter (EKF) - a standard nonlinear extension of the Kalman filter- is used as baseline for comparison instead. The experimental results, shown in Figure 3.5, confirm the findings from the original study: MOP significantly outperforms the EKF.
This degraded prediction quality from the EKF can be explained by the fact that this method relies on linearized dynamics, therefore introducing approximation errors that increase over time. On the other hand MOP, trained on diverse nonlinear source systems, seems to have learned a flexible, data-driven prediction model that generalizes better to new nonlinear environments.

The average total prediction errors for this experiment are given in Table 3.5, supporting the comments.

EKF	**MOP**
35.57	18.72

Table 3.5: Average total prediction error over all test systems for *nonlinear quadrotor systems with i.i.d. Gaussian noise*. Results are reported for the Extended Kalman filter (EKF) and the MOP model.

3.4 Summary

The work of [7] lays the foundation for a convincing approach to optimal filtering using Transformer-based models within an in-context learning framework. By training on a large and diverse collection of dynamical systems, MOP demonstrates remarkable generalization capabilities to previously unseen systems, requiring only a short sequence of past outputs for accurate prediction. Crucially, this is achieved without explicit knowledge of the system dynamics, assumptions of linearity, or restrictive noise models marking its significant advantage compared to the baseline methods.

Their empirical evaluation across a variety of settings, including nonlinear dynamics and time-varying systems, supports the claim that Transformers can match or even surpass classical estimators, especially in complex scenarios. This foundational work therefore serves not only

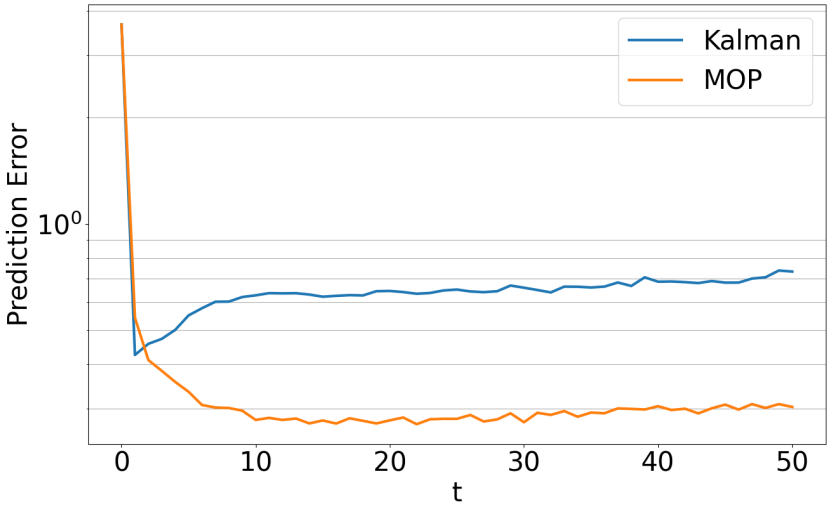


Figure 3.5: Output prediction error trajectories for *nonlinear quadrotor systems with i.i.d. Gaussian noise.*

as a benchmark but also as a motivation for extending Transformer-based approaches beyond output prediction. Indeed, the adaptability and generalization exposed by MOP suggest that such architectures may also perform in more ambitious tasks, such as control or filtering in safety-critical environments as considered in this thesis. In the next chapters, we investigate the performance of similar Transformer-based models in the context of optimal control of simple dynamical systems, based on the insights obtained from this chapter's results.

Chapter 4

Transformers in the Loop: Supervised In-Context Learning with Transformers for Control

4.1 Introduction

The previous chapter was about reproducing and validating the results of the existing work [7], which explores the use of Transformer-based models for predicting the outputs of dynamical systems. In this chapter, we take a step further and ask ourselves:

Can a Transformer-based model be similarly applied to the problem of optimal control?

In other words, can a Transformer implicitly learn the optimal control law that governs a dynamical system, allowing it to drive the system to a desired state? This objective differs from the one in [7] since the goal in the latter is to learn the optimal predictor, as given in (3.3), that uses past and current information to make prediction about the future value of the output variable. As illustrated in Figure 4.1, we now want our model to be able to learn an optimal law that maps the state history to a control input.

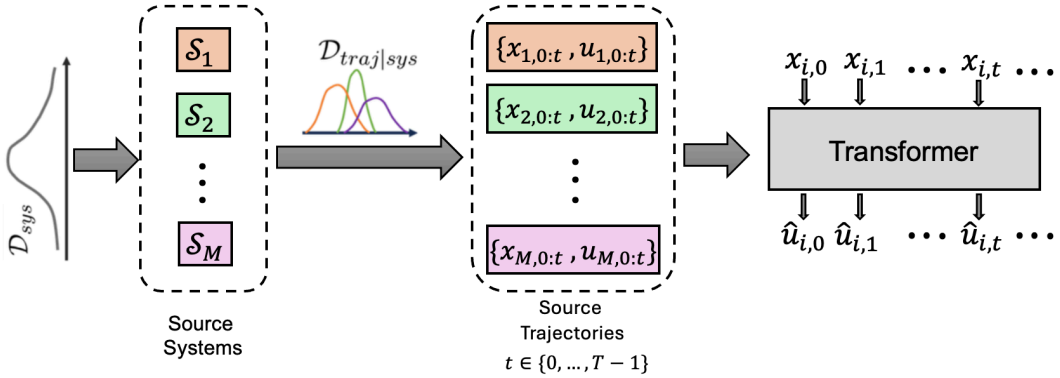


Figure 4.1: Training a transformer model to estimate the control input of a dynamical system based on its past and current states. The model is trained on a distribution of M source systems and evaluated on unseen systems sampled from the same distribution $\mathcal{D}_{sys}$.

Of course, we are facing a major challenge since in most real-world scenarios the optimal controller is not known. This raises two key questions: What should the input-output pairs

for training the Transformer be, i.e. what is the ground truth data? And given the high data requirements for training Transformers, how do we obtain enough meaningful data to ensure a good generalization?

Fortunately, there is one well-known case where the optimal controller is known, which allows us to easily generate synthetic systems and gain preliminary numerical insights: the LQR for fully observable linear systems. When the system is only partially observable and affected by Gaussian noise, the *Linear Quadratic Gaussian* (LQG) controller provides the optimal solution by combining the LQR with a Kalman filter to estimate the states. For systems with constraints, the optimal control law is given by the MPC policy. Therefore, focusing our study on linear systems to establish a solid foundation, we refine our research question to:

Can a Transformers-based model implicitly learn the optimal control law given by the LQR ?

The remainder of this chapter is structured as follows. We begin by outlining the problem setup and the class of systems under consideration. Next, we delve into the architecture of the custom Transformer model used in this thesis. This is followed by a presentation of numerical evaluations comparing the model's performance with traditional control methods used as baselines. Finally, we discuss the emerging need for theoretical guarantees.

4.2 Problem Setup

The framework adopted here is similar to that of Chapter 3. In particular, our objective is still to develop a Transformer-based model that generalizes well to unseen target systems by being trained it on a large number of source systems — a setting known as *meta-learning*, as explained in Section 2.1.4 — with all systems drawn from a similar distribution. The model still does not adapt its parameters during the testing phase and is expected to generalize well by being conditioned only on a short prompt of few input/output pairs of the target system. This behavior is referred to as *in-context learning*, as previously discussed. However, unlike before where the focus was on output prediction, the current objective shifts to solving the problem of optimal control. Finally, it is worth emphasizing that our training approach falls under *supervised learning*, since the model's estimated control inputs are compared to ground-truth data.

Building on this framework, our goal now is to compute the optimal control inputs for an unknown target system, denoted $\mathcal{S}_0$. To enable this, we train the model using a set of M source systems $\{\mathcal{S}_i\}_{i=1}^{M}$. In line with the work in Chapter 3, both the source and target systems are assumed to be sampled from the same distribution $\mathcal{D}_{\text{sys}}$, and each system is described by the general state-space formulation:

$$\mathcal{S}_i = \begin{cases} \mathbf{x}_{i,t+1} &= f_i(\mathbf{x}_{i,t}, \mathbf{u}_{i,t}) \\ \mathbf{y}_{i,t} &= g_i(\mathbf{x}_{i,t}, \mathbf{u}_{i,t}) \end{cases} \qquad t \in \{0, 1, \dots, T-1\}$$

To simplify the analysis and build foundational understanding, we begin by restricting our study to linear dynamical systems, which results in the following simplified model:

$$\mathcal{S}_i = \begin{cases} \mathbf{x}_{i,t+1} &= A_i\mathbf{x}_{i,t} + B_i\mathbf{u}_{i,t} \\ \mathbf{y}_{i,t} &= C_i\mathbf{x}_{i,t} + D_i\mathbf{u}_{i,t} \end{cases} \qquad t \in \{0, 1, \dots, T-1\} \tag{4.1}$$

with $A_i \in \mathbb{R}^{n_x \times n_x}, B_i \in \mathbb{R}^{n_x \times n_u}, C_i \in \mathbb{R}^{n_y \times n_x}$ and $D_i \in \mathbb{R}^{n_y \times n_u}$, and where $\mathbf{x}_{i,t} \in \mathbb{R}^{n_x}$, $\mathbf{u}_{i,t} \in \mathbb{R}^{n_u}$, $\mathbf{y}_{i,t} \in \mathbb{R}^{n_y}$ denote the state, input, and output vectors at time step t for the i^{th} system, respectively. The constant T denotes the trajectory length.
Later in this chapter, we extend the study to more complex settings, such as systems subject

to noise, to further evaluate the robustness and generalization ability of the Transformer in optimal control tasks.

To further simplify our setting, we assume that the transmission matrices D_i are zero. Since the outputs are then linearly dependent on the states, it is valid—and computationally convenient—to work directly with the states in our subsequent analysis. This simplification allows us to apply state-feedback control theory more directly. In particular, the general form of a state feedback control law is given by the expression

$$\mathbf{u}_{i,t} = -K_{i,t}\mathbf{x}_{i,t},$$

where $K_{i,t}$ is a scalar matrix of size $n_u \times n_x$ associated with the i-th systems and time step t. Figure 4.2 presents a visual representation of that structure applied to our framework, i.e. considering the dynamics of the i-th source system.

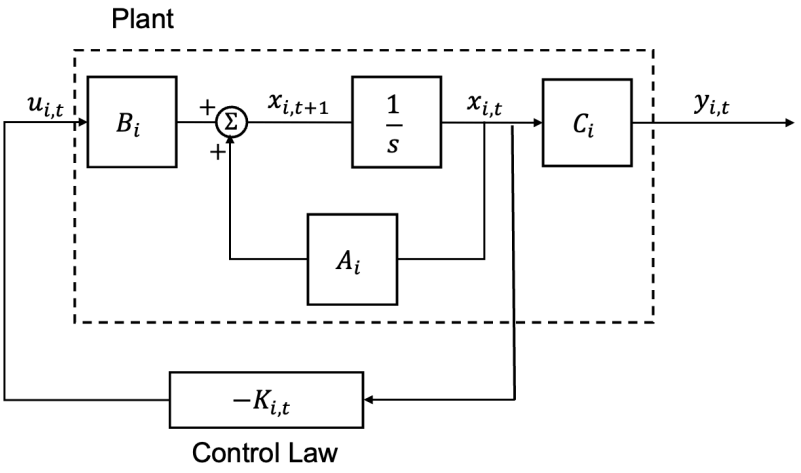


Figure 4.2: Schematic representation of a linear dynamical system with state-feedback control. The control input $\mathbf{u}_{i,t}$ is computed as a linear function of the state $\mathbf{x}_{i,t}$ using the gain matrix $K_{i,t}$, i.e., $\mathbf{u}_{i,t} = -K_{i,t}\mathbf{x}_{i,t}$.

As explained in Section 2.2.3, LQR is optimal for linear systems. In the following experiments, except if mentioned otherwise, the M state-input trajectory pairs $\{(\mathbf{x}_{i,0:T}, \mathbf{u}_{i,0:T-1})\}_{i=1}^{M}$ are therefore generated by simulating i.i.d. linear systems using a LQR controller. The generated state trajectory for the i-th system is then given as input - also known as the *prompt* - to the model in the form $\mathcal{X}_{i,t} := \{\mathbf{x}_{i,k}\}_{k=0}^{t}$, denoting the states up to time t, and the corresponding input trajectory is given to the model as well in order to compute the estimation error between those ground-truth values and the ones estimated by the Transformer.

Our goal is then to train a Transformer model capable of predicting optimal control inputs based on observed system states. The model, already viewed in Chapter 3 as a deep sequence learner $\mathrm{TF}_\theta(\cdot)$, now maps sequences of states $\mathcal{X}_{i,t}$ to estimated input $\hat{u}_{i,t} := \mathrm{TF}_\theta(\mathcal{X}_{i,t})$, aiming to approximate the true control input $u_{i,t}$ at each time step. The Transformer's trainable parameters are denoted by $\theta \in \Theta$ for some parameter set Θ, and the optimal model is obtained by solving the following equation after each pass through the network:

$$\widehat{\mathrm{TF}} = \arg\min_{\mathrm{TF}\in\mathcal{A}} \frac{1}{MT} \sum_{i=1}^{M} \sum_{t=0}^{T-1} \ell\left(\mathbf{u}_{i,t}, \mathrm{TF}(\mathcal{X}_{i,t})\right), \tag{4.2}$$

where $\mathcal{A} := \{\mathrm{TF}_\theta : \theta \in \Theta\}$ and $\ell(\cdot,\cdot) \geq 0$ is the training loss function. During the testing phase, for the target system S_0 defined above, we predict its control inputs by applying the optimized model to its state sequence, i.e., $\widehat{\mathrm{TF}}(\mathcal{X}_{0,t})$ serves as the estimation for $u_{0,t}$.

4.3 Model Architecture

Before performing the experiments, let us first describe the Transformer-based model proposed for this work. We mentioned that the framework of this chapter follows up the one in [7], and the model we consider here is thus greatly inspired from the model proposed by the authors of that work. In particular, the model is based on the GPT-2 architecture - introduced in Section 2.1.2 - to perform sequence-to-sequence regression tasks in the context of dynamical systems with time-dependent inputs and outputs. The implementation builds upon the `transformers` library developed by Hugging Face, incorporating custom modifications to support auto-regressive prediction tailored to our control setting. The full implementation of the class defining the model is provided in Listing 7.3, while the remainder of this section highlights and explains its most important parts.

The model, embedded in a class named `GPT2`, is structured as a subclass of a base class inheriting the *LightningModule* module from the open-source Python library *PyTorch Lightning*. The complete implementation of that base model is given in Listing 7.4. It wraps the GPT-2 architecture, modified for continuous-valued input/output data. More precisely, in addition to the GPT-2 Transformer backbone - which can be described as a stack of n_{layer} Transformer decoder blocks, each consisting of masked multi-head self-attention and feed-forward sub-layers as in Figure 2.2 - it possesses an input projection layer and an output projection layer, which simply consist of a linear layer `Linear` that maps input features of dimension n_x to the embedding space of the GPT-2 backbone and of a linear layer that maps the output embeddings from the Transformer to the desired output dimension n_u, respectively.

The model implements a `forward` method that handles training-time predictions. The model's inputs $\mathbf{x} \in \mathbb{R}^{B\times T\times n_x}$ are first projected into the embedding space, passed through the Transformer model, and projected back to the output space to produce predictions $\hat{\mathbf{u}} \in \mathbb{R}^{B\times T\times n_u}$, where B is the batch size and T remains the trajectory length. The mean squared error (MSE) is used as the training loss function:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{BT}\sum_{i=1}^{B}\sum_{t=0}^{T-1} \|\hat{\mathbf{u}}_{i,t} - \mathbf{u}_{i,t}\|^2 \tag{4.3}$$

This choice is standard in regression tasks because it penalizes larger errors more heavily due to the squaring, improving convergence and stability. Moreover, it is both differentiable and convex, which facilitates optimization with gradient-based methods. In this chapter, we use the AdamW optimizer to minimize (4.3). Averaging the loss over the batch size and trajectory length helps normalize its magnitude, improving training stability and making it comparable across different runs.

On the other hand, the evaluation error metric used for model assessment is the total prediction error per sample across all test systems, summed over the time steps and averaged over the testing set:

$$\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \|\hat{\mathbf{u}}_{i,t} - \mathbf{u}_{i,t}\| \tag{4.4}$$

This metric directly measures the magnitude of the error vector at each time step without squaring, providing an interpretable notion of the total control input deviation per trajectory.

Summing over time captures the cumulative deviation across the entire prediction horizon for each sample, and averaging over the batch provides an overall measure of performance.

Throughout all the experiments of this chapter, we fix some of the parameters and hyper-parameters for the training of the Transformer-based model. Those parameters are given in Table 4.1.

Fixed Parameters and Hyper-Parameters	
Number of test systems N	1,000
Model architecture	GPT-2
Embedding dimension:	$n_{embd} = 256$
Number of transformer layers:	$n_{layer} = 12$
Number of attention heads:	$n_{head} = 8$
Batch size:	$B = 64$
Positional embedding limit:	$n_{\text{positions}} = 2048$

Table 4.1: Fixed model parameters and hyper-parameters used across all experiments in this chapter. These include architectural choices for the Transformer model ($n_{layer}, n_{head}, n_{embd}$) as well as batch size and evaluation settings.

Moreover, dropout rates for embeddings, residuals, and attention set to 0 to ensure deterministic behavior. The number of training samples used as well as the number of epochs will be specific to each experiment, where the *number of epochs* can be defined as the number of times the model has seen the entire training set. It depends on the batch size, training steps and the dataset size, and is in general approximately given by the following ratio:

$$\text{Number of epochs} = \frac{\text{Training steps} \times \text{Batch size}}{\text{Number of training samples}}$$

4.4 Numerical Evaluations

The length of the trajectories is set to $T = 30$ and we fix the state dimension to $n_x = 10$, the input dimension to $n_u = 3$ and the output dimension to $n_y = 5$. The matrices for each system are generated as follows:

- $A \in \mathbb{R}^{n_x \times n_x}$ is randomly generated and scaled to have a spectral radius less than 1 (specifically $\rho(A) < 0.95$), ensuring stability.
- $B \in \mathbb{R}^{n_x \times n_u}$ is randomly generated such that the controllability matrix
$$\mathcal{C} = \begin{bmatrix} B & AB & A^2B & \dots & A^{n_x-1}B \end{bmatrix}$$
has full rank ($\text{rank}(\mathcal{C}) = n_x$).
- $C \in \mathbb{R}^{n_y \times n_x}$ is randomly generated such that the observability matrix
$$\mathcal{O} = \begin{bmatrix} C & CA & CA^2 & \dots & CA^{n_x-1} \end{bmatrix}^T$$
has full rank ($\text{rank}(\mathcal{O}) = n_x$).
- $D = \mathbf{0} \in \mathbb{R}^{n_y \times n_u}$ is the zero matrix.

Moreover, the initial state vector $\mathbf{x}_0$ is sampled from a standard multivariate normal distribution with zero mean and identity covariance matrix, i.e., $\mathbf{x}_0 \sim \mathcal{N}(0, \mathbf{I}_{n_x})$.

4.4.1 First experiment - Deterministic LTI system

As explained, the M source systems are deterministic *Linear Time Invariant* (LTI) systems generated using the LQR controller. A pseudo-code of the implementation used in this experiment is given below, but all the code for this thesis can be accessed at `https://github.com/devillersmar/master_thesis_code.git`.

Algorithm 1 Simulate LTI System Trajectories using LQR

function SIMULATEWITHLQR($fsim$, n_{traj}, Q, R, Q_N)
  $K_s \leftarrow$ SOLVERICCATI($fsim$, Q, R, Q_N, n_{traj})
  $x \leftarrow$ random vector $\sim \mathcal{N}(0, I)$
  $x_s \leftarrow [x]$
  $u_s \leftarrow [\,]$
  for $t = 0$ to $n_{\text{traj}} - 1$ **do**
    $u \leftarrow -K_s[t] \cdot x$
    Append u to u_s
    $x \leftarrow Ax + Bu$
    Append x to x_s
  end for
  return x_s, u_s
end function

In order to evaluate the performance of the Transformer, we consider 2 baselines: the MPC as described in Section (2.2.4), and a simple -naive - proportional controller of the form $\mathbf{u}_{i,t} = -K\mathbf{x}_{i,t}$, where K is a constant matrix fixed over the estimation horizon.

The proportional controller is known to not be optimal since it doesn't explicitly use the system dynamics to optimize control, and is thus used as a naive control strategy that does not leverage model knowledge nor explicit cost minimization. It is also memoryless – it doesn't account for where the system is going, just where it is- which is in contrast to MPC and LQR where the former looks ahead over a moving finite horizon and the latter accounts for the entire future via the Riccati recursion. Since the horizon is finite in this case, we differentiate the LQR from the MPC by setting the prediction horizon of the latter to a quarter of the total trajectory length. On the bright side, it is straightforward to implement the proportional controller - as mentioned in Section (2.3) - typically just requiring to manually tune the gain matrix K. This simplicity makes it easy to understand and use as a reference, setting a low baseline that any well-designed optimal controller (such as LQR or MPC) should surpass.

The following pseudo-code describes the computation of control input sequences based on the optimal sequence of states returned by the LQR and using the naive proportional control strategy, where the gain matrix is randomly initialized for each system. The function applying the MPC policy, also based on the sequence of states returned by the LQR, is given in Listing 7.6.

Algorithm 2 Control Inputs Generation using Proportional Controller

x_s = sequence of states returned by LQR
Initialize empty list u_P
for $i = 0$ to $M - 1$ **do**
    K_p = random proportional gain matrix
    for $t = 0$ to $T - 1$ **do**
        $u_t \leftarrow -K_p \cdot (C_i \cdot x_{i,t})$
    end for
    Append control sequence $\{u_t\}_{t=0}^{T-1}$ to u_P
end for

Since we want to compare the performance of the Transformer-based model to the ones of the selected baselines, it is essential to first identify suitable values for the two most critical training hyper-parameters: the number of training samples and the number of training epochs. Due to practical limitations in training capacity, the model is trained with number of source systems varying from $5,000$ to $80,000$. Specifically, we have that $M \in \{5 \times 10^3, 10 \times 10^3, 20 \times 10^3, 40 \times 10^3, 80 \times 10^3\}$. In parallel, the number of epochs is set to $n_{\text{epoch}} = 2^k$ for $k \in \{0, 1, 2, 3, 4\}$, resulting in a logarithmic progression from 1 to 16 epochs.

Figure 4.3 presents a heatmap of the prediction errors computed using (4.4) across all $(n_{\text{epoch},i}, M_j)$ combinations, for $i, j \in \{1, 2, 3, 4, 5\}$. Complementarily, Figure 4.4 illustrates the averaged estimation errors — along with their standard deviations — first as a function of the number of training samples (left), and then as a function of the number of training epochs (right).

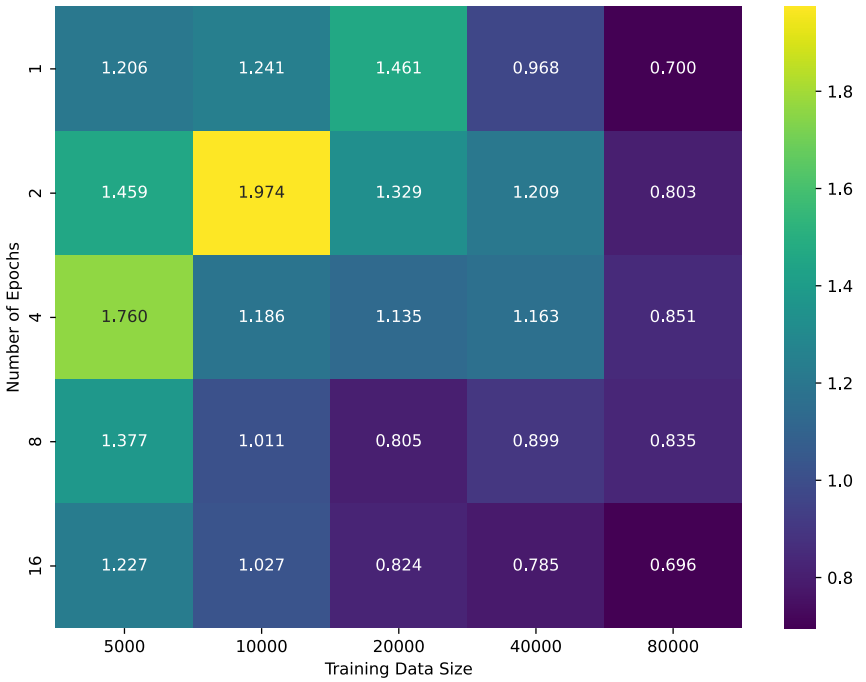
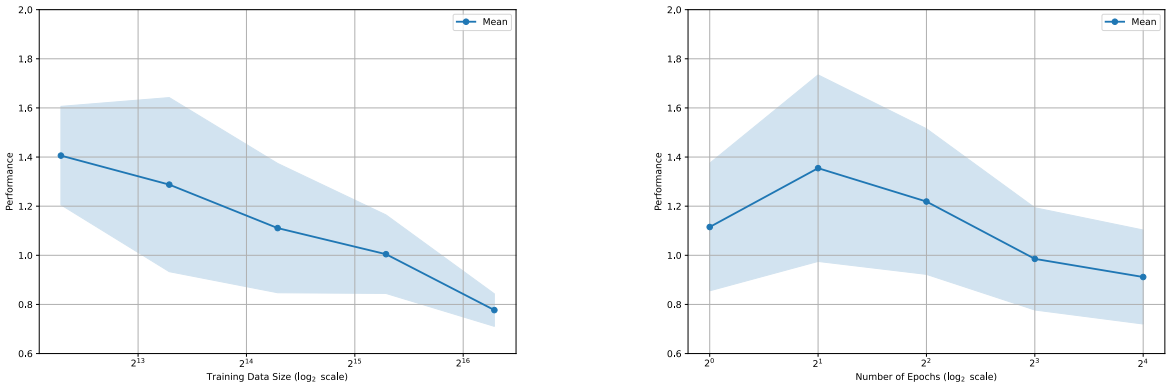


Figure 4.3: Heatmap showing the prediction errors of the Transformer-based model for a deterministic LTI system using (4.4) , evaluated across combinations of training epochs (n_{epoch}) and number of training samples (M). Lower values indicate better performance.



(a) (b)

Figure 4.4: Average estimation errors (with corresponding standard deviation) of the Transformer-based model for the deterministic LTI system: (left) as a function of the number of training samples, showing the impact of dataset size on estimation accuracy; (right) as a function of the number of training epochs, illustrating the effect of training duration on model performance.

The results clearly indicate that increasing the number of training samples consistently improves model performance. This trend, highlighted in both the heatmap and Figure 4.4(a), is aligned with existing literature on Transformer-based architectures. Indeed, models such as GPT-2 and BERT owe much of their generalization power to the availability of massive datasets during pretraining. These findings reinforce the hypothesis that data abundance is a key enabler of strong performance—not only in natural language processing tasks, but also in estimation tasks involving dynamical systems such as LTI models.

In contrast, the impact of the number of epochs is less straightforward. While one might expect a steady performance improvement with longer training durations, the results in Figure 4.4(b) reveal a more nuanced behavior. Specifically, the model's accuracy appears to degrade when trained for only 2 or 4 epochs, with performance starting to recover and eventually improve only after reaching 8 or more epochs. This suggests that very short training durations may lead to underfitting, where the model fails to capture the system dynamics effectively.

This result, while intuitive, naturally prompts deeper questions regarding the data efficiency and generalization capability of Transformer-based models in control tasks. In particular, one might ask whether there exists an optimal amount of training data beyond which the estimation error asymptotically approaches zero. Could the excess risk be theoretically bounded, as explored in [7]? Is it even possible for a Transformer architecture to fully learn the optimal control policy induced by the LQR controller, thus bypassing the need for explicit computation of the discrete-time Riccati equation?

This latter question resonates with one of the key challenges encountered during the experiments: as the trajectory horizon extends beyond 30 time steps, controllers relying on solutions of the Riccati recursion begin to produce unstable and oscillatory system behavior. This degradation is likely due to the well-known numerical sensitivity of the backward Riccati recursion [23] [24]. As the recursion proceeds over long time horizons, the involved matrix inversions—especially those of the form $(R+B^\top P_{k+1}B)^{-1}$—can become ill-conditioned. Combined with the accumulation of floating-point round-off errors, this often leads to inaccurate or

diverging gain matrices K_k, ultimately compromising the stability and accuracy of the system.

These observations not only highlight a practical limitation of classical optimal control in digital settings, but also emphasize the potential advantage of data-driven models such as Transformers, which might inherently regularize such numerical issues through learning. If such models can approximate the optimal control law with sufficient accuracy from data, they may offer a robust alternative when traditional analytical methods become numerically unreliable.

Unless mentioned otherwise, all subsequent experiments and evaluations are conducted using the model trained with $M = 80{,}000$ samples and $n_{\text{epoch}} = 16$, as this combination yields the lowest estimation error according to the results summarized above.

Staying in the context of average estimation error between the estimated control inputs and the optimal input returned by LQR, Figure 4.5 displays for each method the estimation errors for each time step, averaged with respect to the number of testing samples.

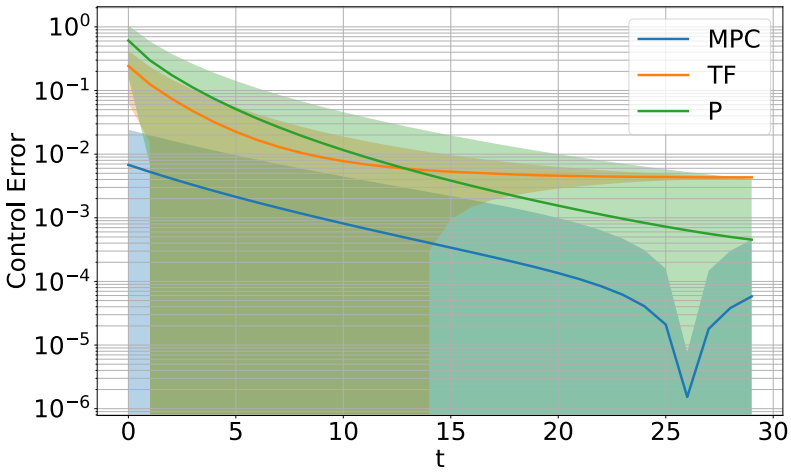


Figure 4.5: Average Control input estimation errors over time steps averaged over the testing set for deterministic LTI systems, comparing the Transformer-based model (TF), Model Predictive Control (MPC), and a naive proportional controller (P).

It is evident that the MPC consistently outperforms the other methods throughout the trajectory, yielding the smallest estimation errors. The Transformer initially outperforms the proportional controller, but its error decay rate slows over time, eventually allowing the proportional controller—which maintains a more consistent decay rate—to surpass it halfway through the trajectory.

While the average Euclidean error provides insight into how closely the predicted control inputs match the optimal ones at each time step, it does not fully capture the impact of those predictions on the system's overall performance. In optimal control problems, it might be more informative to evaluate the actual closed-loop performance of the controller. To this end, a crucial complementary metric is the *total quadratic cost* incurred by the trajectories generated by each method, here denoted J. This cost reflects how effectively the controller

regulates the system state while balancing control effort, and is defined as:

$$J(\mathbf{x}, \mathbf{u}) = \frac{1}{2NT} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \left[\sum_{k=0}^{t-1} \mathbf{x}_{i,k}^T Q_i \mathbf{x}_{i,k} + \mathbf{u}_{i,k}^T R_i \mathbf{u}_{i,k} + \mathbf{x}_{i,t}^T Q_{i,t} \mathbf{x}_{i,t} \right], \tag{4.5}$$

where Q_i and R_i are positive semi-definite weighting matrices that penalize state deviations and control efforts, respectively, and $Q_{i,t}$ denotes the terminal state cost. This total quadratic cost is a function of both the state trajectories $\{\mathbf{x}_{i,t}\}_{t=0}^{T}$ and control inputs $\{\mathbf{u}_{i,t}\}_{t=0}^{T-1}$, as the control inputs determine the state evolution through the system dynamics.

This cost function serves as a direct measure of the control quality achieved by a method, rather than merely the accuracy of its predictions. By comparing that cost across different approaches, we obtain evaluation that better aligns with the true objective of optimal control, i.e. minimizing long-term cumulative costs while ensuring system stability and efficiency.

Figure 4.6 shows box plots of the quadratic cost (4.5) across the test set for each of the four methods. The LQR controller achieves the lowest median loss, with a compact interquartile

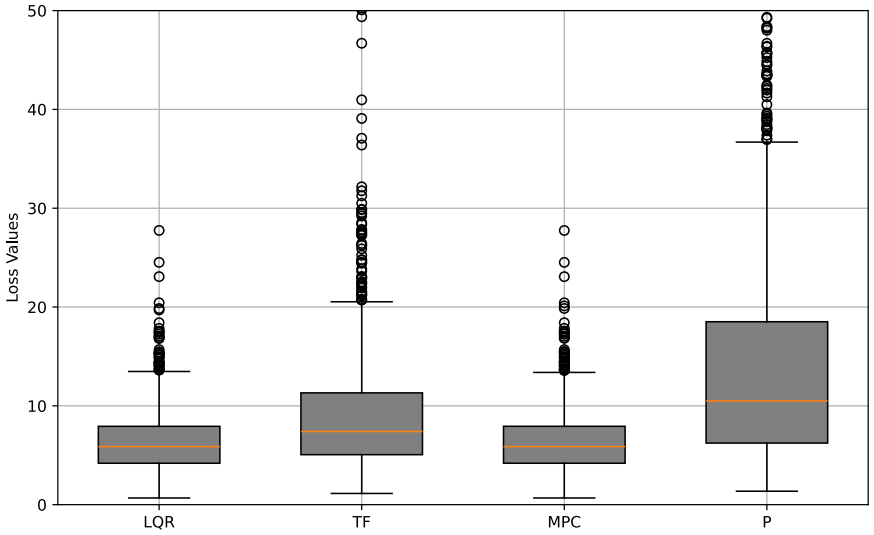


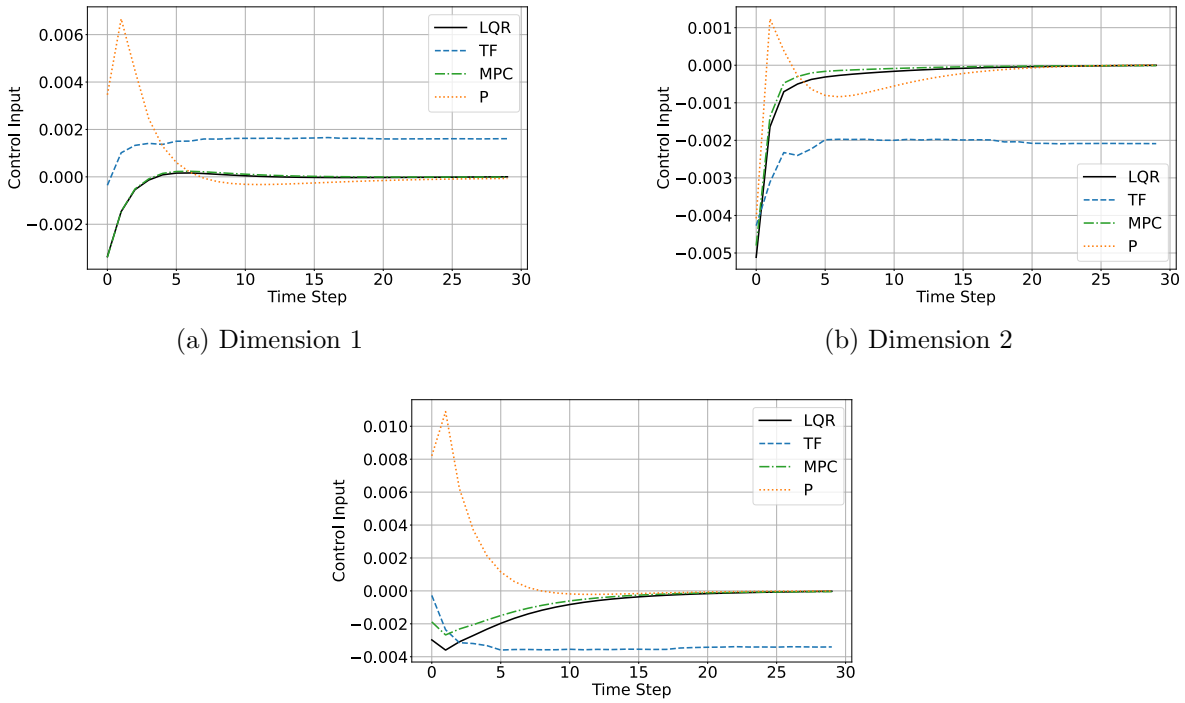
Figure 4.6: Box plots of the total quadratic cost J incurred by each method across the test systems. This cost quantifies the overall closed-loop performance for the LQR, the Transformer-based (TF), the MPC and a naive proportional (P) controller.

range and few outliers, indicating both high performance and low variability. This result is expected, as LQR represents an optimal control policy when the system dynamics are perfectly known. The MPC controller demonstrates similar results to the LQR controller for this experiment, while the Transformer-based method yields a slightly higher median loss with a wider spread, suggesting generally good performance but with greater variability. In contrast, the naive proportional controller exhibits the highest median loss, the largest spread, and a significant number of outliers, indicating poor and inconsistent control performance. Overall, these results highlight the robustness of LQR, the promising potential of Transformer-based control, and the limitations of more simplistic control strategies.

These results can be interpreted in two ways. On one hand, one might view them positively: the Transformer is able to implicitly learn features of the system class during training, yielding a lower quadratic cost than a simple proportional controller—a baseline that remains widely

used in industrial applications. On the other hand, one could find the results unsatisfactory, as the Transformer still does not reach the average performance levels of the LQR and MPC controllers.

To bring more insight into our reflection, let us now analyze the trajectories taken by each dimension of the control inputs returned by the methods. This is given in Figure 4.7.



(a) Dimension 1

(b) Dimension 2

(c) Dimension 3

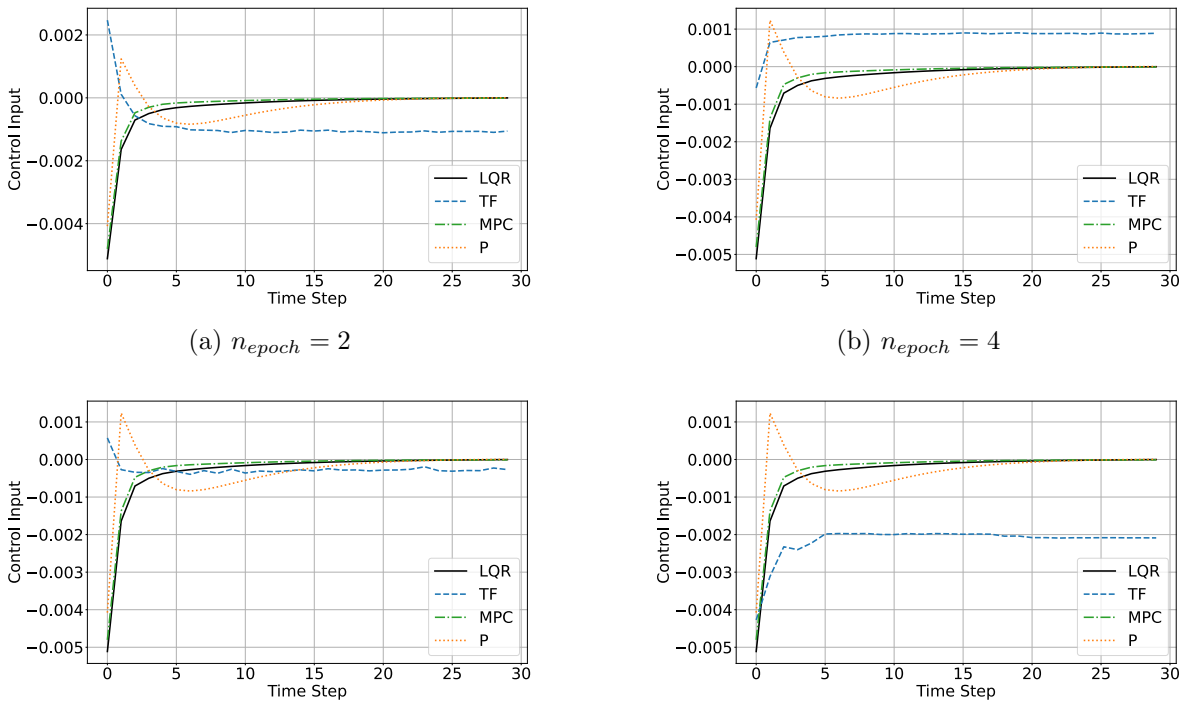
Figure 4.7: Trajectories of control input components over the trajectory horizon and shown separately for three representative input dimensions for each method: the LQR, the Transformer-based (TF), the MPC and a naive proportional (P) controller.

Clearly, none of the trajectories predicted by the Transformer converge to the steady-state value of zero for any input dimension. In contrast, the MPC closely follows the optimal LQR trajectories, converging smoothly to zero, while the proportional controller takes roughly half the horizon to settle. The first half of the trajectories shows the typical behavior of a proportional controller, as discussed in Section 2.2.2. This result makes us reassess our expectations for the Transformer: is it truly a positive outcome that its total quadratic cost J in (4.5) is lower than that of the proportional controller, even though it fails to drive the system to the desired steady state value? After all, the proportional controller — although slowly — ultimately achieves the control objective.

This raises a deeper question: what kind of relationship between the states and inputs is the Transformer actually learning? Although it deviates from the behavior of the baseline methods, could the policy it learns still be meaningful in some alternative sense?

Let us now investigate the trajectories for a specific input dimension, say dimension 2, at varying numbers of training epochs n_{epoch} for a training dataset of size $M = 80{,}000$. We already know from the heatmap that the error values do not vary a lot from one value of

number of epochs to another for this number of training data so we would expect quite similar trajectories. Figure 4.8 displays the trajectories for $n_{epoch} \in 2, 4, 8, 16$.



(a) $n_{epoch} = 2$

(b) $n_{epoch} = 4$

(c) $n_{epoch} = 8$

(d) $n_{epoch} = 16$

Figure 4.8: Control input trajectories for dimension 2 at varying numbers of training epochs $n_{\text{epoch}} \in \{2, 4, 8, 16\}$ with a fixed training dataset size of $M = 80{,}000$ for each method: the LQR, the Transformer-based (TF), the MPC and a naive proportional (P) controller.

Despite similar overall error metrics from the heatmap, the input trajectories differ substantially from one number of epochs to another. For $n_{epoch} = 2$ and 8, trajectories start positive, then decay below zero, settling at biased steady-state values. Conversely, the two other values of n_{epoch} lead to an increasing curve of values over time, but again fail to converge to zero. It looks like the model does not remember anything about the systems from one epoch to another.

This divergence highlights an important nuance: although the Transformer achieves average quadratic loss values that stand in the range of the other methods, it does not explicitly optimize for driving the control inputs to zero steady-state values. But what it *does* optimize seems to not even be similar from one training to another. Moreover, unlike LQR or MPC which inherently incorporate such constraints via the cost function design, the Transformer's training objective (MSE over predicted inputs) does not enforce these properties. This could explain why the Transformer's control trajectories deviate significantly from zero in steady state, resulting in different input dynamics despite comparable overall cost performance. This suggests that future improvements may require incorporating more explicit constraints or loss terms that emphasize steady-state behavior and system stability, and this is exactly what the next chapter is about. Before diving into this new aspect of training the Transformer, let us first make a few more experiments in our current setup challenging the abilities of the controllers considered so far.

4.4.2 Second experiment - LTI system with i.i.d. Gaussian noise

In this second experiment, we make the system's dynamics (4.1) become stochastic by adding additive i.i.d. Gaussian noise to its equations. The state-space model representation thus becomes

$$\mathcal{S}_i = \begin{cases} \mathbf{x}_{i,t+1} & = A_i\mathbf{x}_{i,t} + B_i\mathbf{u}_{i,t} + \mathbf{w}_{i,t+1} \\ \mathbf{y}_{i,t} & = C_i\mathbf{x}_{i,t} + D_i\mathbf{u}_{i,t} + \mathbf{v}_{i,t+1} \end{cases} \tag{4.6}$$

where the additive noise is defined by $\mathbf{w}_t = \sum_{t'=t-n_{\text{noise}}+1}^{t} \eta_{t'}$ and $\mathbf{v}_t = \sum_{t'=t-n_{\text{noise}}+1}^{t} \gamma_{t'}$. The sum of the terms $\eta_t \sim \mathcal{N}(0, \sigma_{\mathbf{w}}^2 I)$ represents the process noise affecting the states, and the variable n_{noise} specifies whether we are dealing with non-i.i.d noise ($n_{\text{noise}} > 1$) or not ($n_{\text{noise}} = 1$). Similarly, $\gamma_t \sim \mathcal{N}(0, \sigma_{\mathbf{v}}^2 I)$ represents the measurement noise affecting the outputs. The noise covariances are initially fixed to $\sigma_{\mathbf{w}}^2 = 0.01$ and $\sigma_{\mathbf{v}}^2 = 0.01$.

However, the LQR controller is not robust anymore since it is designed under the assumption of an accurate knowledge of the system dynamics. It therefore does not explicitly take into account uncertainties and it is sensitive to deviations induced by the noise. We thus need to introduce a compensator structure composed of an estimator and a controller that will be able to perform optimal control while filtering out the noisy signals. This is done by combining the Kalman filter and the LQR controller, leading to the LQG controller.

The function simulating noisy dynamical system trajectories used in this experiment therefore apply LQG to obtain the optimal control inputs and a pseudo-code is given in the following algorithm. The complete implementation of the function can be found at Listing (7.8).

Algorithm 3 Noisy Dynamical System's Trajectories Generation using LQG Controller

Compute LQR gains K_s using Riccati recursion
Initialize Kalman filter with (A, B, Q, R, Q_T) and initial state estimate x_{est}
Initialize noise buffers w_t, v_t
for $t = 0$ to $T - 1$ **do**
    $u_t \leftarrow -K_t \cdot x_{\text{est}}$
    $x_{t+1} \leftarrow Ax_t + Bu_t + \sum w_{t-i}$
    $y_t \leftarrow Cx_{t+1} + \sum v_{t-i}$
    KalmanFilter.update(y_t)
    KalmanFilter.predict(u_t)
    Store x_{t+1}, x_{est}, u_t
end for

While the MPC implementation remains identical, the naive proportional controller is adapted in order to incorporate the noise in the system and therefore becomes:

Algorithm 4 Control Inputs Generation using Proportional Controller

x_s = sequences of states returned by LQG
Initialize empty list u_P
for $i = 0$ to $M - 1$ **do**
    K_p = random proportional gain matrix
    Initialize noise buffer v_t
    for $t = 0$ to $T - 1$ **do**
        $y_t \leftarrow C_i \cdot x_t + \sum v_{t-j}$
        $u_t \leftarrow -K_p \cdot y_t$
        Append new noise sample to v_t
    end for
    Append control sequence $\{u_t\}_{t=0}^{T-1}$ to u_P
end for

From Figure 4.9, several key observations emerge regarding the robustness of each controller in the presence of i.i.d. Gaussian noise:

First, the performance of the MPC controller deteriorates compared to its behavior in the noise-free setting. This degradation can be attributed to the finite prediction horizon of the MPC, which limits its ability to anticipate disturbances introduced by the noise.

Then, the proportional controller performs poorly in this setting and maintains throughout the whole trajectory the highest error curve as expected. Its simplistic design, which lacks any internal representation of the system dynamics or estimation mechanism, makes it highly sensitive to uncertainties.

Finally, the Transformer finds itself in a quite good position in-between those two baselines proving that, while not achieving the accuracy of the MPC, it clearly outperforms the proportional controller. This indicates that it has captured some understanding of the system's underlying structure. Nevertheless, its inability to fully close the performance gap with MPC suggests that it does not yet model the temporal dependencies and noise statistics with sufficient precision. This highlights a current limitation of the model considered in this section.

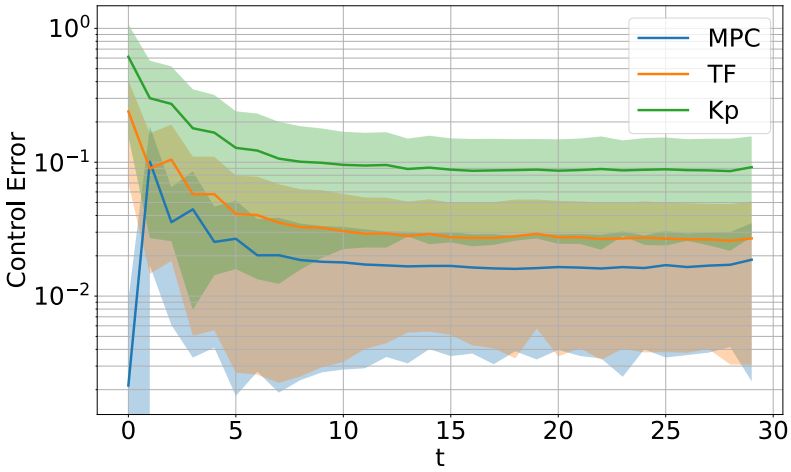


Figure 4.9: Average control input estimation errors over trajectory length over the testing set for LTI systems with i.i.d. Gaussian noise, comparing the Transformer-based model (TF), MPC, and a naise proportional controller (P).

4.4.3 Third Experiment: Response to Changing Dynamics

In this final experiment, we are still considering the noisy dynamical systems as introduced in the previous section, but we now evaluate the robustness of the Transformer-based controller when faced with abrupt changes in the dynamics of those systems. Specifically, the underlying system switches to a different set of dynamics at time step $t = 30$, simulating a sudden and significant change in the environment.

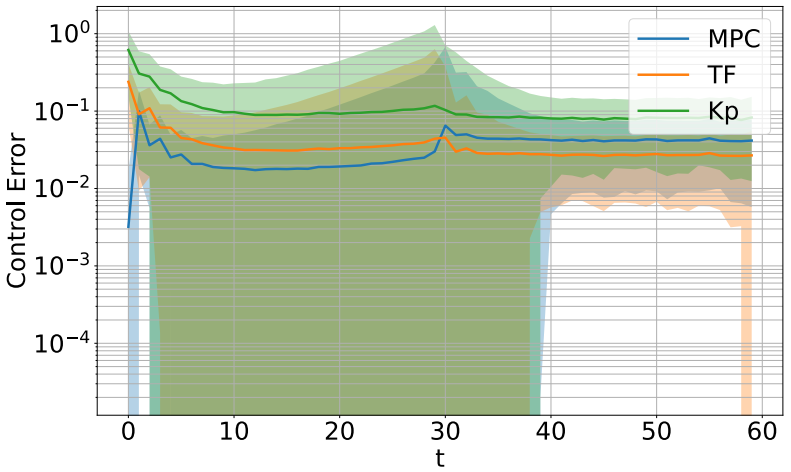


Figure 4.10: Average control input estimation errors over trajectory length over the testing set for LTI systems with i.i.d. Gaussian noise under changing dynamics at $t = 30.$, comparing the Transformer-based model (TF), MPC, and a naise proportional controller (P).

As shown in Figure 4.10, both the Transformer and MPC policies experience a sharp increase in estimation error immediately following the change in dynamics. However, a key difference

emerges in their ability to recover: the Transformer quickly adapts to the new dynamics and even achieves better performance in the second half of the trajectory than in the first. In contrast, the MPC policy struggles to recover and fails to reach back its previous level of performance.

These results highlight a noteworthy strength of the Transformer-based approach. While previous experiments suggested that the Transformer might not yet be leveraging sufficient model information for accurate estimation of the optimal control inputs, this experiment demonstrates its superior adaptability. This robustness to sudden system changes could offer a valuable advantage in real-world applications where unexpected perturbations or modeling inaccuracies are common.

4.5 Summary

In conclusion, this chapter provided us with results demonstrating a significant strength of Transformer-based models: their robustness and adaptability to abrupt system changes and modeling uncertainties. Although first tests proved a certain limitation in accurately estimating optimal control inputs — likely due to incomplete exploitation of system dynamics — the third experiment reveals that Transformers effectively generalize from learned patterns to maintain performance under perturbations. This behavior aligns with established AI research showing Transformers' superior ability to capture complex temporal dependencies and tolerate distribution shifts, explaining for instance its remarkable performance in most NLP or computer vision tasks compared to previous methods. Such resilience is critical for safety-critical control applications, where unexpected disturbances and model inaccuracies frequently occur.

The limitations of the Transformer-based model as optimal controller for dynamical systems, as identified in this chapter, motivate however the need to re-think and modify the way the training of the model is performed. In particular, we observed that while these models perform reasonably well, they lack the ability to explicitly account for control-optimality principles — such as those employed in LQR — when trained using conventional supervised learning. Supervised training typically focuses on minimizing a prediction error between estimated and ground truth trajectories, rather than optimizing a performance-oriented cost function grounded in control theory as it will be done in the next chapter.

Chapter 5

Beyond Labels: Self-Supervised In-Context Learning with Transformers for Control

5.1 Introduction

In this chapter, we present the core contribution of this thesis: a self-supervised in-context learning framework for Transformer-based controllers. Instead of minimizing the discrepancy between predicted and reference data, like it was done in Chapter 4, the model is now trained to directly minimize the following total quadratic cost:

$$J(\mathbf{x}, \mathbf{u}) = \frac{1}{2MT} \sum_{i=1}^{M} \sum_{t=0}^{T-1} \left[\sum_{k=0}^{t-1} \mathbf{x}_{i,k}^T Q_i \mathbf{x}_{i,k} + \mathbf{u}_{i,k}^T R_i \mathbf{u}_{i,k} + \mathbf{x}_{i,t}^T Q_{i,t} \mathbf{x}_{i,t} \right], \tag{5.1}$$

where Q_i and R_i are positive semi-definite weighting matrices that penalize state deviations and control efforts, respectively, and $Q_{i,t}$ denotes the terminal state cost.

This total quadratic cost, function of both the state trajectories $\{\mathbf{x}_{i,t}\}_{t=0}^{T}$ and control inputs $\{\mathbf{u}_{i,t}\}_{t=0}^{T-1}$, captures the cumulative trade-off between state deviation and control effort, guiding the model toward learning an optimal feedback policy. Indeed, in optimal control problems - as already mentioned in Section 4.4.1 - evaluating the controller's closed-loop performance is more informative than measuring prediction accuracy alone. To this end, the total quadratic cost J serves as a direct metric of control quality, reflecting how well the controller regulates the system while balancing actuation effort. Training the model to minimize this cost should therefore offer a more principled evaluation aligned with the true objectives of optimal control, i.e. minimizing long-term cumulative costs while ensuring stability and efficiency.

Importantly, this framework no longer relies on supervision: the model is not provided with optimal state or control trajectories during training. Instead, it receives only an initial state — common to all evaluated controllers in the following experiments — and must auto-regressively generate the full sequence of system states and control inputs. This self-supervised setup better reflects practical control settings, where policies must infer actions in real-time based on partial observations and internal dynamics models, without access to a ground truth trajectory.

The remainder of this chapter is structured as follows. We first describe the problem setup and the class of dynamical systems considered. It is followed by a section describing briefly the updated architecture of the Transformer-based controller, emphasizing the transition from

supervised to unsupervised learning. Finally, we introduce two key experimental settings and the baseline methods against which our approach is evaluated, highlighting the potential of Transformer models to learn optimal control laws solely from interaction and initial state information.

5.2 Problem Set Up

The type of systems considered in this chapter are stochastic LTI dynamical systems with additive Gaussian noise, similar to the ones introduced in Section 4.4.2. Specifically, every i-th source system $\mathcal{S}_i$, for $i \in \{1, 2, \cdots, M\}$ can be represented by the following discrete-time state-space equations:

$$\mathcal{S}_i = \begin{cases} \mathbf{x}_{i,t+1} &= A_i \mathbf{x}_{i,t} + B_i \mathbf{u}_{i,t} + \mathbf{w}_{i,t+1} \\ \mathbf{y}_{i,t} &= C_i \mathbf{x}_{i,t} + D_i \mathbf{u}_{i,t} + \mathbf{v}_{i,t+1} \end{cases} \qquad t \in \{0, 1, \ldots, T-1\} \tag{5.2}$$

with $A_i \in \mathbb{R}^{n_x \times n_x}, B_i \in \mathbb{R}^{n_x \times n_u}, C_i \in \mathbb{R}^{n_y \times n_x}$ and $D_i \in \mathbb{R}^{n_y \times n_u}$, and where $\mathbf{x}_{i,t} \in \mathbb{R}^{n_x}$, $\mathbf{u}_{i,t} \in \mathbb{R}^{n_u}$, $\mathbf{y}_{i,t} \in \mathbb{R}^{n_y}$ denote the state, input, and output vectors at time step t for the i^{th} system, respectively. The constant T denotes the trajectory length.

The additive noise is defined by $\mathbf{w}_t = \sum_{t'=t-n_{\text{noise}}+1}^{t} \eta_{t'}$ and $\mathbf{v}_t = \sum_{t'=t-n_{\text{noise}}+1}^{t} \gamma_{t'}$. The sum of the terms $\eta_t \sim \mathcal{N}(0, \sigma_{\mathbf{w}}^2 I)$ represents the process noise affecting the states, and the variable n_{noise} specifies whether we are dealing with non-i.i.d noise ($n_{\text{noise}} > 1$) or not ($n_{\text{noise}} = 1$). Similarly, $\gamma_t \sim \mathcal{N}(0, \sigma_{\mathbf{v}}^2 I)$ represents the measurement noise affecting the outputs. The noise covariances are initially fixed to $\sigma_{\mathbf{w}}^2 = 0.01$ and $\sigma_{\mathbf{v}}^2 = 0.01$.
Both i.i.d. and temporally correlated (non-i.i.d.) noise settings are considered to assess the controller's robustness under different uncertainty models.

While our focus is restricted to linear systems in this work, the methodology lays the groundwork for future research into more complex nonlinear control problems. The unsupervised in-context learning framework introduced here is designed to be flexible and generalizable, offering a foundation for extending Transformer-based controllers to broader classes of systems.

5.3 Model Architecture

The model considered in this chapter is very similar to the one used in the previous chapter, and the complete updated implementation is given in Listing 7.5.
The architecture of our custom model is still built on the GPT-2 architecture - visually represented in Figure 2.2 - from Hugging Face's `transformers` library, and inherits from a base class model following the *LightningModule* module of the Pytorch Lightning library. The implementation of the base class is the same as in the previous chapter in given in the Listing 7.4.
Only a brief overview of the modified or new parts will be given in this section.

Simulating Linear Dynamics A method `simulate_linear_dynamics` is added in order to model the evolution of a linear dynamical system under the influence of Gaussian noise: $\mathbf{x}_{i,t+1} = A_i \mathbf{x}_{i,t} + B_i \mathbf{u}_{i,t} + \mathbf{w}_{i,t+1}$.
This function models the type of interaction that the model would potential have with real-world - complex - systems, and therefore stands on its own, not hard-coded in the model's main functionalities enabling therefore generalization to other type of models by simply replacing it with the desired dynamics.

Auto-regressive Simulation A method denoted `predict_ar` auto-regressively simulates the system trajectory by iteratively applying the GPT-2 model to predict control inputs. It operates in the following way:

1. Use the current context window of past states.
2. Predict the control input using GPT-2.
3. Simulate the next state using the linear dynamics model.
4. Append the new state to the context.

One of the main points of this new version of the model is the addition of a context window, and it is fixed to one quarter of the total trajectory length in the following experiments.

Forward Method The `forward` method integrates the core functionality:

- Performs autoregressive prediction via `predict_ar`.
- Computes the loss via `calculate_losses_and_metrics`.

It returns the loss and optionally intermediate states and predictions, giving the optimizer all the information needed to adjust the weights of the model in the best possible way.

The following table give the fixed parameters used throughout all the experiments. Note that due to the increased complexity of our model, a trade-off between efficiency and computational cost had to be found. That way, the architecture of the model is modified is more suited for performing the experiments.

Fixed Parameters and Hyper-Parameters	
Number of test system N	1,000
Model architecture	GPT-2
Embedding dimension:	$n_{embd} = 16$
Number of transformer layers:	$n_{layer} = 3$
Number of attention heads:	$n_{head} = 2$
Batch size:	$B = 64$
Positional encoding limit:	$n_{\text{positions}} = 2 * T$

Table 5.1: Fixed model parameters and hyper-parameters used across all experiments in this chapter. These include architectural choices for the Transformer model ($n_{layer}, n_{head}, n_{embd}$) as well as batch size and evaluation settings.

Moreover, dropout rates for embeddings, residuals, and attention set to 0 to ensure deterministic behavior. The number of training samples used as well as the number of epochs will be specific to each experiment.

Regarding the evaluation error metric used for model assessment, it remains the total prediction error per sample across all test systems, summed over the time steps and averaged over the testing set as given in:

$$\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\|\hat{\mathbf{u}}_{i,t} - \mathbf{u}_{i,t}\| \tag{5.3}$$

5.4 Experiments

For both experiments presented in this section, the states and inputs trajectories returned by the Transformer are compared to 3 different models: the LQR - whose implementation is given in Listing (7.7) and known to be sub-optimal for systems that present noisy data - the LQG - whose implementation is given in Listing (7.8) and known to be the optimal controller for systems that present noisy data with i.i.d. Gaussian noise - and the simple - naive - proportional controller.

The choice of LQG is justified by the fact that we want to compare the Transformer to what is known to be optimal - at least in the first experiment we consider - and the choice of LQR will allow us to see if the Transformer can at least be as good as a method known to not be optimal in those cases. Finally, the third baseline controller, the proportional controller, is known to not be optimal and any case so, if the Transformer-based model improves indeed its estimations thanks to the new training set up presented in the introductory section, then it should be much better than the proportional controller. Moreover, we know that this type of controller is still often used in industries so outperforming it would already be a sign of good performance.

As the implementation of the proportional controller considered in these experiments implementation diverge from the previously introduced proportional controllers in Chapter 4, we give its complete implementation in the Listing 7.9. But the general form is still given by

$$\mathbf{u}_{i,t} = -K\mathbf{x}_{i,t},$$

where K is a constant matrix fixed over the estimation horizon and specific to each system.

5.4.1 First experiment: LTI system with i.i.d. Gaussian noise

Following the idea of the first experiment of Chapter 4, we first study the performance of the Transformer model in this framework across different pairs of values $(n_{\text{epoch},i}, M_j)$, for $i, j \in \{1, 2, 3, 4\}$, which are feasible within the computational limitations. More precisely, we have here that $M \in \{5 \times 10^3, 10 \times 10^3, 20 \times 10^3, 40 \times 10^3\}$ and the numbers of epochs are set as $n_{\text{epoch}} = 2^k$ for $k \in \{0, 1, 2, 3\}$. The resulting errors and means of errors are presented in Figure 5.1 and Figure 5.2.

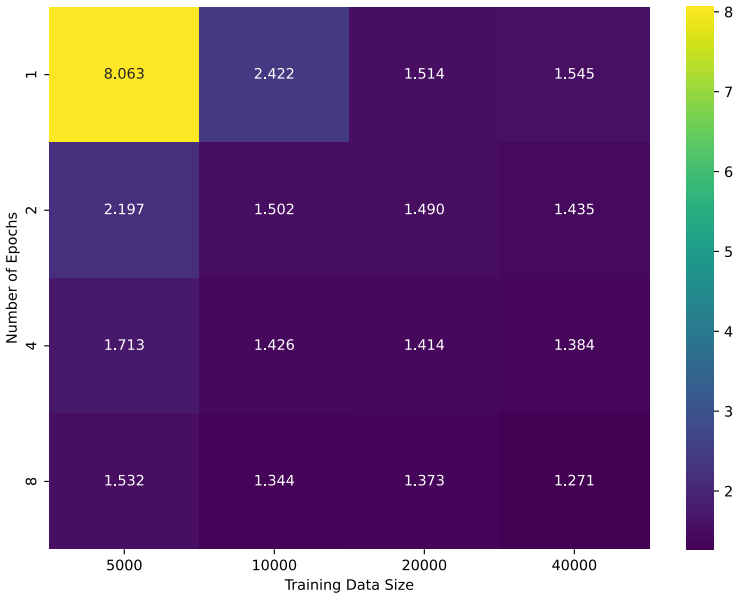
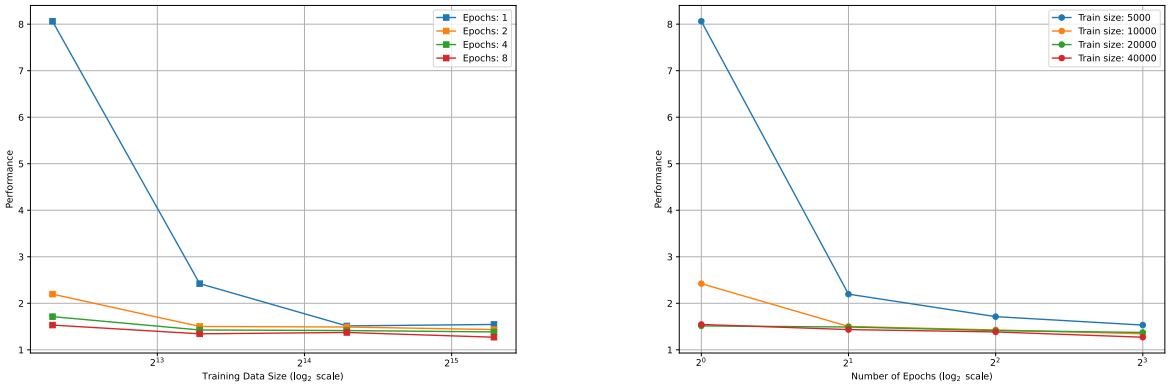


Figure 5.1: Heatmap showing the prediction errors of the Transformer-based model for LTI system with i.i.d. Gaussian noise using (5.3), evaluated across combinations of training epochs (n_{epoch}) and number of training systems (M). Lower values indicate a better performance.



(a) (b)

Figure 5.2: Average estimation errors (with corresponding standard deviation) of the Transformer-based model for the LTI systems with i.i.d. Gaussian noise: (left) as function of number of training data, (right) as a function of number of training epochs

The heatmap reveals that the model performance improves significantly when increasing either the number of training epochs from 1 to 2 epochs or the training data size from 5 000 to 10 000 samples. Beyond this region, the errors stay close to each other, even when increasing the number of source systems or the number of epochs. For instance, while the estimation error drops drastically from 8.06 to 2.20 when increasing the number of epochs from 1 to 2 with 5 000 samples, the error only marginally decreases from 1.514 to 1.271 when increasing the dataset size from 10 000 to 40 000 at 8 training epochs. This saturation behavior is further supported by the error curves Figure 5.2, where the subfigure (a) shows the error decreasing steeply for all epoch counts up to 10 000–20 000 samples, after which the curves flatten. Simi-

larly, subfigure (b) shows that most of the learning happens within the first few epochs, with error curves converging around epoch 4.

In brief, it seems like we are facing an "error floor" for large data and training regimes. Further research could investigate this situation in more details in order to determine if those results reflect modeling limitations or simply caused by the physical/computational limitations mentioned above. The latter explanation would obviously be preferable.

Following a similar reasoning as in the previous chapter, the next results concern only the ones obtained with the model trained on $M = 80{,}000$ source systems and $n_{\text{epoch}} = 16$ number of epochs, as it is the one associated with the smallest estimation error in the heatmap. Note that a similar map could be done by comparing the total cost value for each pair, but these values result in a similar conclusion and we therefore stick to a heatmap of the estimation errors.

We remain also within the framework of evaluating the average estimation errors between the optimal control inputs returned by LQR and those produced by the 3 other methods. Figure 5.3 displays the estimation errors for each time step, averaged across the test samples.

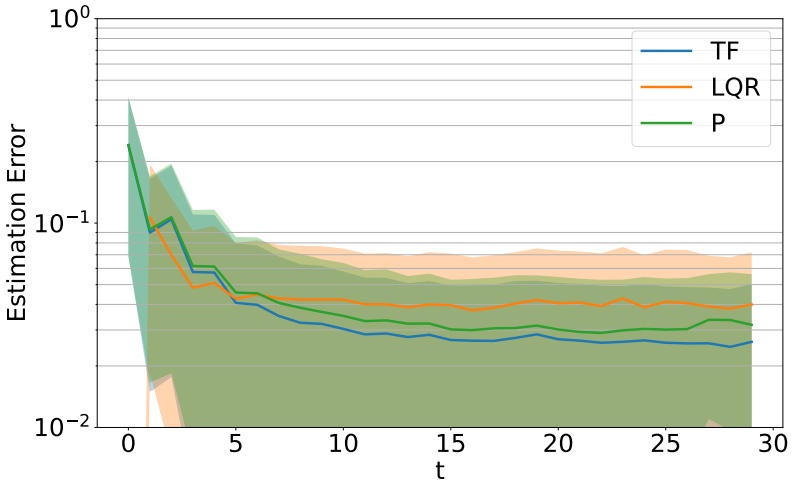


Figure 5.3: Control input estimation errors over the trajectory length averaged over the testing set for LTI systems with i.i.d. Gaussian noise, comparing the Transformer-base model (TF), LQR, and the naive proportional controller (P).

From those curves, we notice that it only takes a short burn-in time ($\sim$ 5 steps) for the Transformer to outperform the baseline methods, in terms of the error relative to the LQG-generated optimal controls.

This rapid convergence gives more hope about the potential of a Transformer-based model as optimal controller and its ability to learn useful representations of the control law even in noisy settings.
However, a complementary perspective is given in Figure 5.4, which presents boxplots corresponding to the average Euclidean error as given in Equation 4.5 for each method, mitigate the previous results. Indeed, it looks like the LQR still slightly outperforms the Transformer in terms of the total induced cost, and the latter is comparable to the proportional controller in this metric, though with better stability as exposed in the next figures.

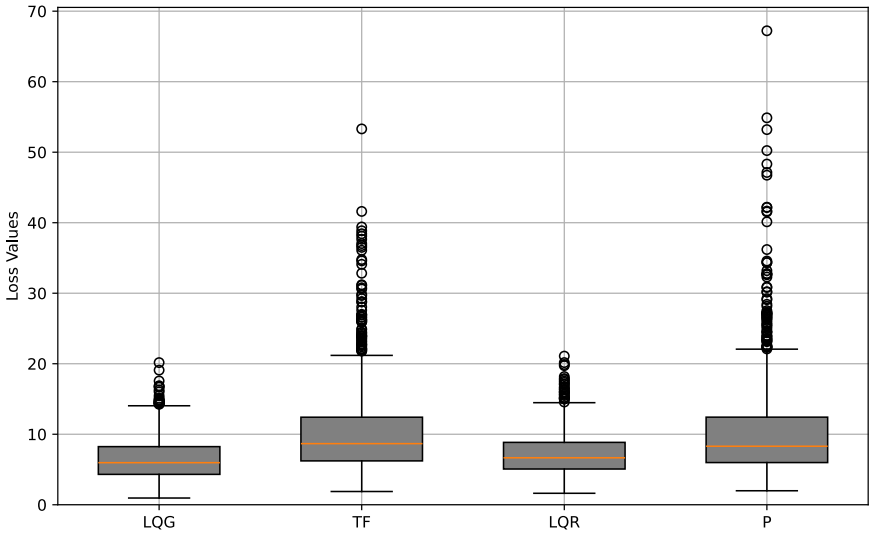
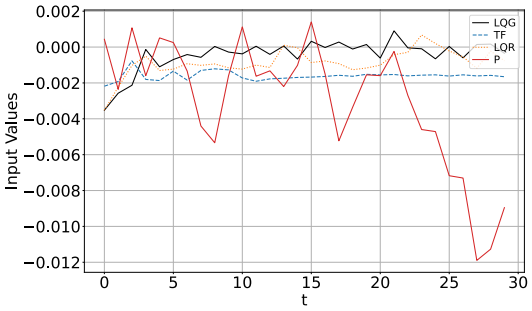


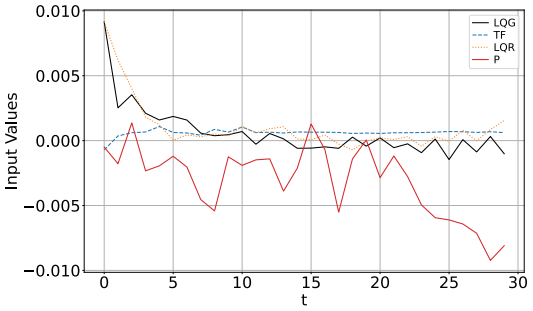
Figure 5.4: Box plots of the total quadratic cost J incurred by each method across the test systems, for the optimal LQG controller, the Transformer-base model (TF), LQR, and the naive proportional controller (P).

Figures 5.5 and 5.6 provide a more detailed view of the control input and state trajectories. It is clearly from those plots that the Transformer is still not yet able to reach the desired steady state value like the LQG - and even the LQR - controller does. It is worth mentioning however that its performance has improved over the one of the experiments of the previous section, suggesting that the model now indeed exhibits greater awareness and takes the system dynamics into account, likely through the minimization of the quadratic loss function during training. On the other hand, it is much more stable than any other trajectories, and especially the ones obtained with the proportional controller which are characterized by high oscillations and instability.

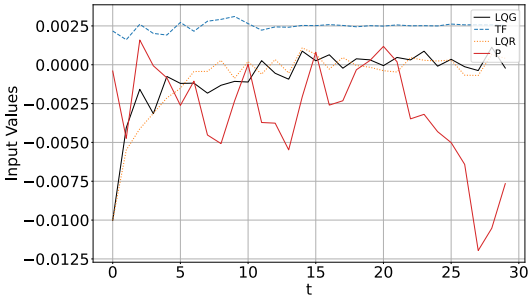
Figure 5.6 confirms those observations by showing evidence that the Transformer generates states trajectories that closely match those of the LQR and LQG controllers. On that figure, the state values of the proportional controller are not shown due to their significantly different scale, but Figure 5.7 displays them for completeness.



(a) Dimension 1



(b) Dimension 2



(c) Dimension 3

Figure 5.5: Trajectories of control input components over the trajectory horizon and shown separately for the three input dimensions for each method: the optimal LQG controller, the Transformer-base model (TF), LQR, and the naive proportional controller (P).

Figure 5.6: Trajectories of internal state components over the trajectory horizon and shown separately for the n_x state dimensions for each method **except** the the naive proportional controller: the optimal LQG controller, the Transformer-base model (TF) and LQR.

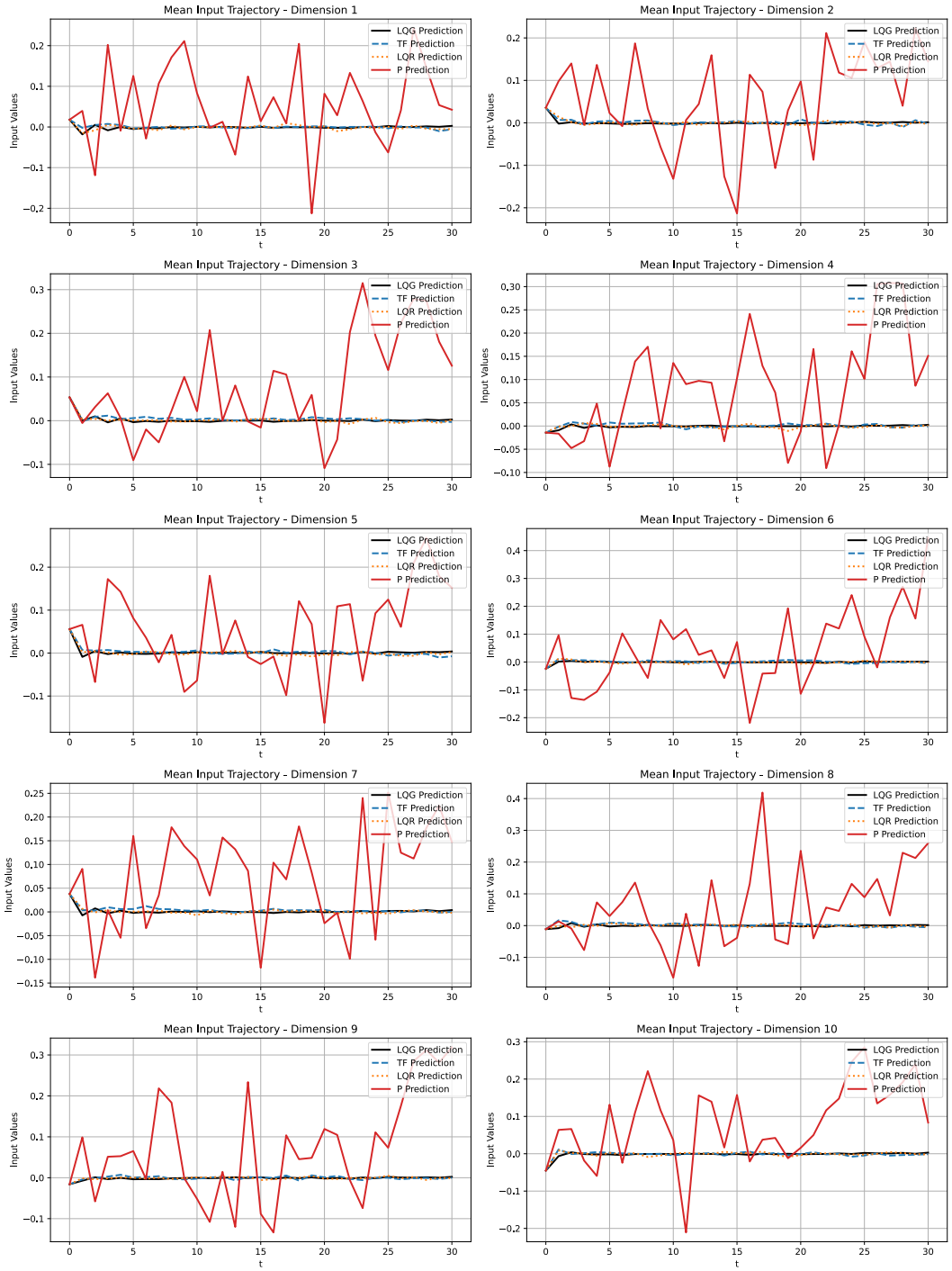


Figure 5.7: Trajectories of internal state components over the trajectory horizon and shown separately for the n_x state dimensions for each method: the optimal LQG controller, the Transformer-base model (TF), LQR, and the naive proportional controller (P).

Overall, this experiment demonstrates encouraging progress in the Transformer-based controller's performance. While it has not yet surpassed traditional model-based methods in total cost or steady-state accuracy, it exhibits a strong ability to adapt and learn stable, near-optimal control policies from data alone. Its superior stability in the input values over simpler baselines like the proportional controller, alongside the improvements in terms of errors and steady state value, suggests that further experiments could be performed - such as enhancing

model awareness or incorporating different training approaches - and make it an interesting alternative for real-world control tasks. Focusing on the stabilization of state trajectories could also contribute significantly to improving the method.

5.4.2 Second experiment: LTI system with non-i.i.d. Gaussian noise

For the second experiment of this chapter, we consider exactly the same setup and systems as in the previous section, but now introduce an additive non-i.i.d. Gaussian noise. As previously mentioned, this type of structured noise is implemented by setting the variable n_{noise} to a value strictly greater than 1, therefore giving the systems a "memory" of the noise at the previous time steps. In this experiment, we set $n_{noise} = 5$.
Moreover, we keep considering only the results obtained from the model trained on a source system dataset of size $M = 80{,}000$ and trained for $n_{epoch} = 8$ epochs.

Since the Kalman filter used in the LQG controller assumes i.i.d. noise for optimal state estimation, its performance is no longer theoretically guaranteed under non-i.i.d. conditions. Consequently, plotting the state estimation error over time for LQG becomes less meaningful, as its underlying assumptions are violated. Instead, we focus directly on analyzing the overall cost induced by each controller under these new conditions. This is presented in Figure 5.8.

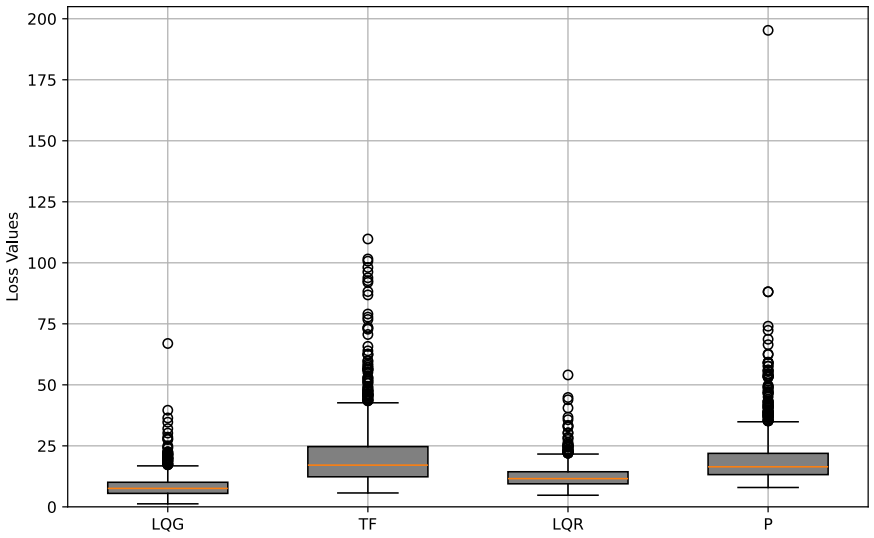
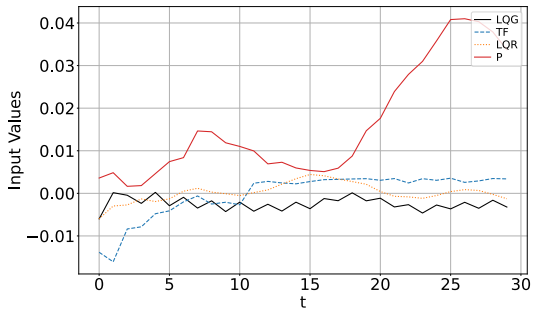


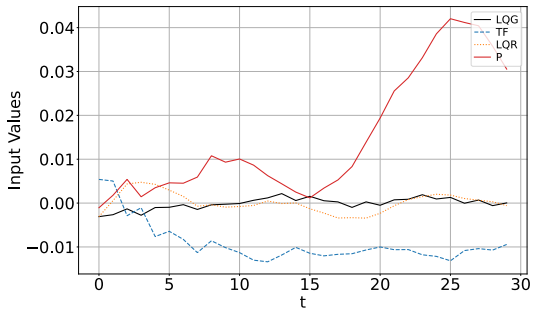
Figure 5.8: Box plots of the total quadratic cost J incurred by each method across the test systems: the LQG, the Transformer-based (TF), the LQR and the naive proportional (P), and for LTI systems with non-i.i.d. Gaussian noise.

The LQG and LQR controllers achieve the lowest median losses, with LQG slightly outperforming LQR overall. Both exhibit tight interquartile ranges, suggesting consistent performance despite the violation of the i.i.d. noise assumption. In contrast, the Transformer-based controller shows a notably wider interquartile range, indicating greater variability in performance across test cases. This variability suggests that while the Transformer can occasionally perform competitively, it lacks robustness under structured noise. Lastly, the proportional controller has again both a high median and wide spread in cost, reinforcing its limitations in noisy environments. Overall, while the Transformer shows potential, it still requires improvements to match the stability and reliability of model-based approaches under complex noise conditions.

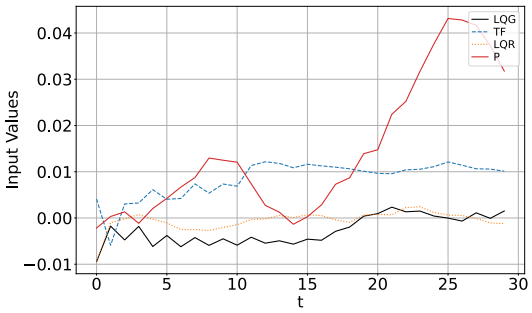
The following figures illustrate the control input and state trajectories for each method. These visualizations complement the boxplot analysis, offering deeper insights into performance degradation. In particular, they confirm that the Transformer occasionally produces unstable behavior but remains more consistent than the proportional controller. Furthermore, they reveal that both LQG and LQR also suffer from increased fluctuations in state trajectories compared to the previous experiment, underscoring the impact of noise correlation on all methods.



(a) Dimension 1



(b) Dimension 2



(c) Dimension 3

Figure 5.9: Trajectories of control input components over the trajectory horizon and shown separately for the three input dimensions for each method: the optimal LQG controller, the Transformer-base model (TF), LQR, and the naive proportional controller (P), and for LTI systems with non-i.i.d. Gaussian noise.

Figure 5.10: Trajectories of internal state components over the trajectory horizon and shown separately for the n_x state dimensions for each method **except** the the naive proportional controller: the optimal LQG controller, the Transformer-base model (TF) and LQR, and for LTI systems with non-i.i.d. Gaussian noise.

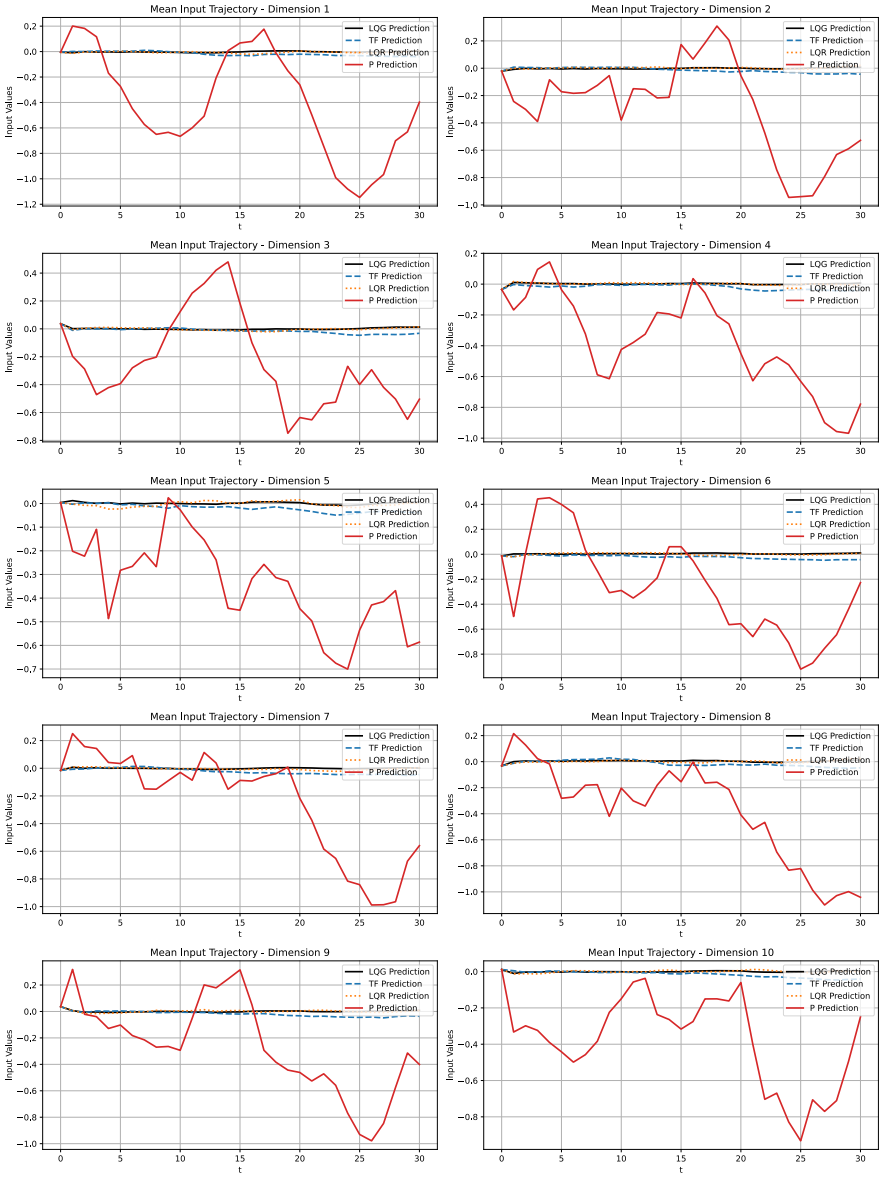


Figure 5.11: Trajectories of internal state components over the trajectory horizon and shown separately for the n_x state dimensions for each method **except** the the naive proportional controller: the optimal LQG controller, the Transformer-base model (TF) and LQR, and for LTI systems with non-i.i.d. Gaussian noise.

Overall, this experiment underscores the challenge posed by non-i.i.d. Gaussian noise in control settings. While model-based controllers like LQG and LQR remain relatively robust, their performance does degrade, especially in terms of trajectory stability. The Transformer-based controller shows promise, achieving competitive performance in some cases, but lacks the reliability needed for deployment in safety-critical scenarios. Improving the robustness of data-driven controllers under temporally correlated disturbances remains an important direction for future work.

Chapter 6

Conclusion

This thesis set out to explore the capabilities of Transformer-based models for learning and controlling complex dynamical systems through in-context learning and under the self-supervised learning setting introduced in Chapter 5. Across a range of experiments, we demonstrated that Transformers, when trained appropriately, can capture rich representations of system dynamics and adapt to varying control scenarios, even in the face of perturbations or modeling uncertainties. While these results are promising, they also underscore the challenges inherent in relying solely on in-context learning for control synthesis, particularly when optimality is desired. On the other hand, it was easy to notice that various setting choices can be modified and improved in order to continue the research and ameliorate the results. For instance, this thesis is based on the architecture of GPT-2 as it is the latest open-source version published by OpenAI, but we could enhance the study by using newest - and more efficient, at least in NLP tasks - versions like GPT-3 or GPT-4.

Two perspectives from the author emerge from this work. On one hand, one may envision a future where Transformers are capable of real-time, in-context control—acting as flexible, data-driven agents that adapt to new environments without explicit retraining. On the other hand, this thesis can also be seen as a step toward better understanding of *what* Transformers can learn about complex phenomena, beyond conventional supervised tasks. This lens treats the Transformer not just as a tool for control, but as a scientific instrument—revealing patterns, structure, and generalization behavior in high-dimensional, dynamical contexts.

Rather than favoring one view over the other, this work suggests that both perspectives are valuable and mutually enriching. The potential of in-context learning for control remains largely open, but the insights gained here point to exciting directions: using Transformers to uncover latent structure in dynamical systems, informing model-based control design, or complementing traditional control techniques with learned representations.

In this sense, the contribution of this thesis is not only technical but also conceptual—inviting a dialogue between modern AI models and classical control theory, and encouraging further exploration at their intersection.

6.1 Use of AI

In addition to being the core model used in all experiments presented in this thesis, GPT-2 (and its newest version) was also leveraged as a writing assistant for correcting punctuation and spelling errors, as well as a debugging aid during the implementation of the experiments through the use of the app ChatGPT.

Chapter 7

Appendix

7.1 Appendix A

The emergence of the field of AI is deeply related to the advances in the field of *Computer Science* (CS). While the development of the computer has now been evolving over more than two centuries, the development of AI only started in the early 20^{th} century when new discoveries in neurology had shown that the brain functioned as an electrical network of neurons. This understanding, alongside other theories discovered at that time, laid the theoretical ground for AI as a field. In particular, it contributed to the creation of the Turing test - a benchmark for evaluating machine intelligence - introduced in 1950 in Alan Turing's paper *Computational Machinery and Intelligence*. Around the same time, Warren McCulloch and Walter Pitts (1943) proposed the first mathematical model of an artificial neuron [25], but it was only in 1956 that the term "*Artificial Intelligence*" was officially introduced at the Dartmouth Workshop [26] , hosted by John McCarthy and Marvin Minsky, marking the birth of AI as an academical field. Shortly after, Frank Rosenblatt (1957) introduced the first ever implementation of a McCulloch& Pitts neuron: the *Perceptron*. It was an early neural network model inspired by the visual cortex, capable of learning through weight adjustments. However, by 1969, a book [27] published by Minksy and Seymour Papert exposed its limitations - such as the inability to solve the XOR problem - leading to the first AI winter.

Despite that setback, computing technologies were still improving, and with them, progress in AI also continued. In particular, Paul Werbos introduced in 1974 an essential algorithm for training deep neural networks, though not immediately recognized, named *backpropagation* [28]. Backpropagation was finally popularized in the $80's$, enabling multi-layer networks to learn more complex patterns. This renewed interest lead to theoretical advances, like the *Universal Approximation Theorem* [29] in 1989 or the *Long-Short-Term Memory* [30] networks in 1991. The latter was introduced in a paper of Sepp Hochreiter and Jürgen Schmidhuber, and addressed the *Vanishing Gradient problem*. The difficulties due to this problem when training deep networks, added to the lack of computational power at that time, led to a second AI winter. As a result, simpler and more efficient machine learning methods, such as *Support Vector Machines* (SVMs) and *Random Forest*, became dominant due to their superior performance on many real-world tasks.

AI's second resurgence began in the mid-2000s, driven by advances in computational hardware, data availability, and algorithmic innovations. It started when Hinton, Osindero, and Teh introduced *Deep Belief Networks* (DBNs) in 2006, showing that deep networks could be efficiently trained using unsupervised pretraining [31]. This approach helped mitigate the vanishing gradient problem, making deep networks trainable again. Around the same time, the explosion of big data and the rise of GPUs enabled large-scale *Deep Learning* (DL). In

2012, a deep CNN named AlexNet dominated the ImageNet competition, demonstrating the power of DL in computer vision and therefore introducing a new wave of interest and rapid breakthroughs in DL. For instance, in 2014, *Generative Adversarial Networks* (GANs) revolutionized high-quality synthetic image generation and *Residual Networks* (ResNets) solved the vanishing gradient problem in very deep architectures by skipping connections. Those residual connections stabilize the training and convergence of deep neural networks with hundreds of layers, making them common motif in current deep neural networks. In parallel, advances in *natural language processing* (NLP) were made possible by *Recurrent Neural Networks* (RNN) and in 2017, Vaswani introduced in his well-known paper [4] the *Transformer* (TF) architecture, eliminating sequential processing limitations in language models. This led to GPT (OpenAI), BERT (Google), and other large-scale NLP models, transforming AI's ability to process and generate text. Today, AI is the dominant paradigm in computing, enabling breakthroughs in computer vision, speech recognition, robotics, and generative AI. Figure 7.1 summarizes the different milestones mentioned here above.

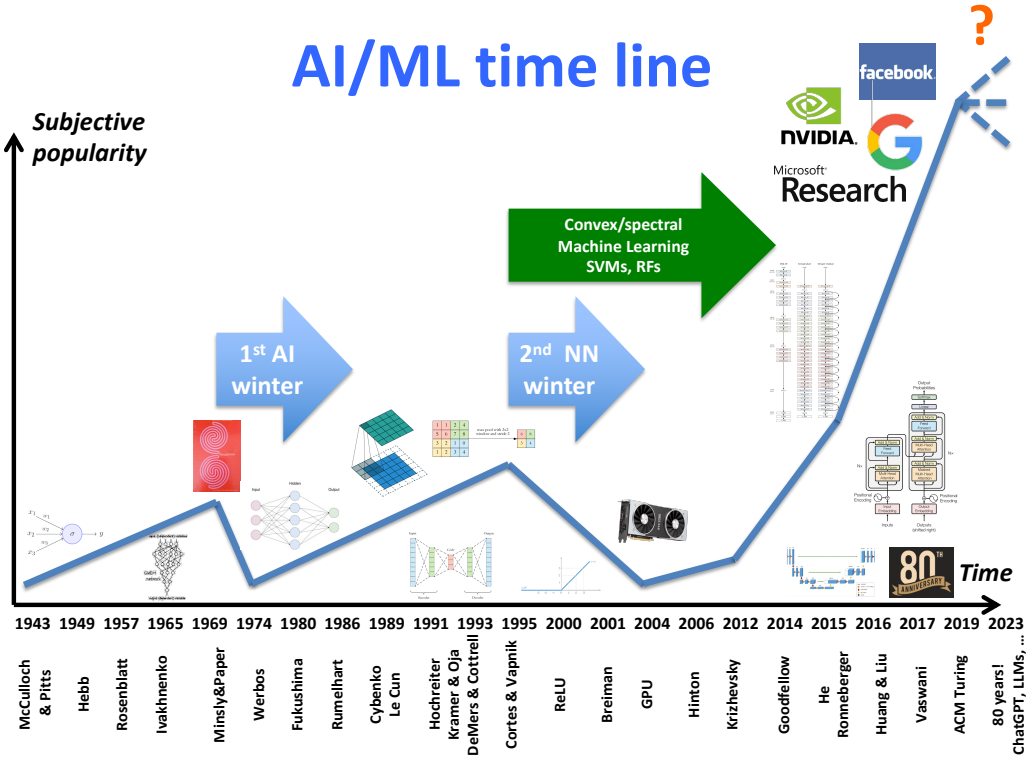


Figure 7.1: Timeline of major breakthroughs in the field AI.[1]

[1]Graph taken from lecture slides of the course *LELEC2870 - Machine learning : regression, deep networks and dimensionality reduction*, UCLouvain

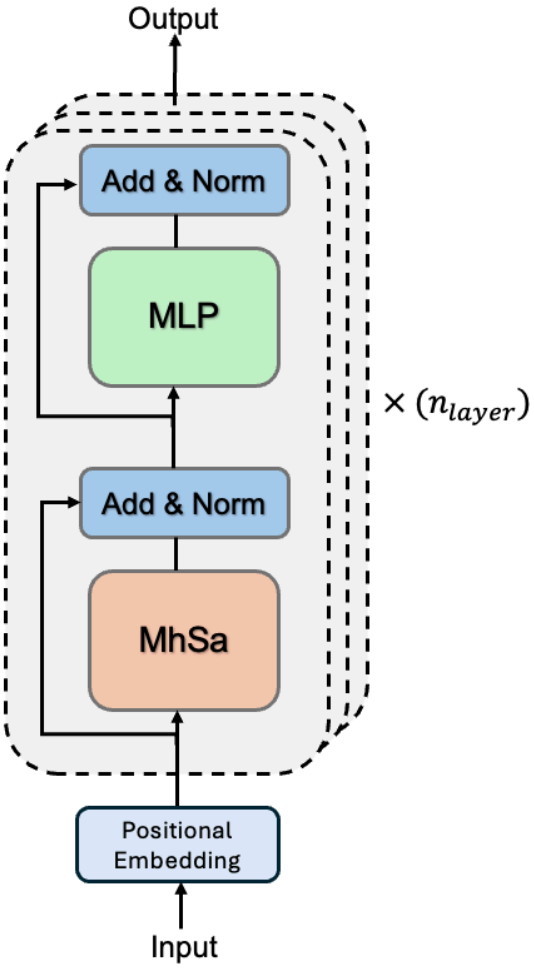


Figure 7.2: Illustration of encoder-only Transformer architectures consisting of n_{layer} Transformer layers with a multi-head self-attention unit, residual connections and normalization layers, and finally a fully connected neural network

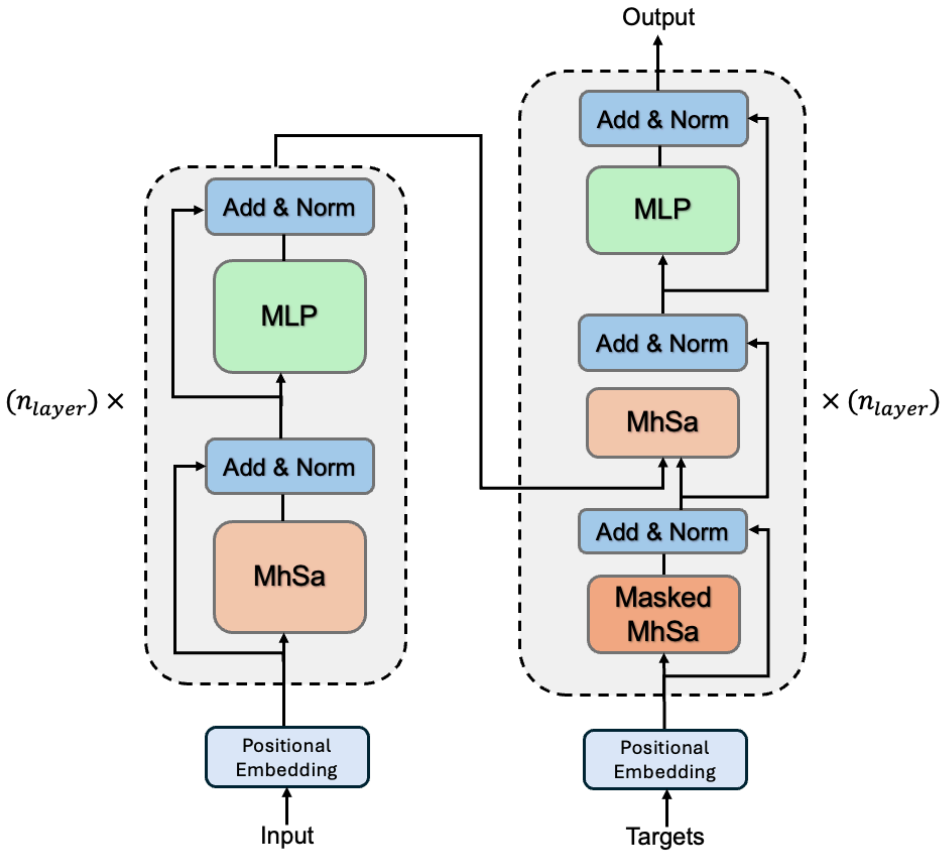


Figure 7.3: Illustration of encoder-decoder Transformer architectures consisting of n_{layer} Transformer layers with a (masked) multi-head self-attention unit for the (decoder) encoder layers, residual connections and normalization layers, and finally a fully connected neural network

7.2 Appendix B

```
def solve_riccati(self, Q, R, QN, n_traj):

    S_list = [None]*(n_traj + 1)
    K_list = [None]*n_traj

    S_list[n_traj] = QN

    # Backward iteration
    for k in range(n_traj-1, -1, -1):
        S_next = S_list[k+1]

        P_inv = np.linalg.inv(R + self.B.T @ S_next @ self.B)

        S_k = self.A.T@(S_next-S_next@self.B@P_inv@self.B.T@S_next)@self.A + Q
        S_list[k] = S_k

        K_k = P_inv @ self.B.T @ S_k @ self.A
        K_list[k] = K_k

    return K_list
```

Listing 7.1: Solve the Riccati Equation

```
def apply_lqr(fsim, n_traj, x0=None, Q=None, R=None, QN=None, return_obj=False
    ):

    Q = np.eye(fsim.nx) if Q is None else Q
    R = np.eye(fsim.nu) if R is None else R
    QN = np.eye(fsim.nx) if QN is None else QN

    us = []
    xs = [np.random.rand(fsim.nx) if x0 is None else x0]
    x = xs[0]

    if fsim.noise == True:
       ws = [np.random.randn(fsim.nx)*fsim.sigma_w for _ in range(fsim.n_noise
           )]

    Ks = solve_riccati(fsim, Q, R, QN, n_traj)

    for t in range(n_traj):
        u_opt = -Ks[t] @ x
        us.append(u_opt)

        # Update state with system dynamics
        x = fsim.A @ x + fsim.B @ u_opt
        if fsim.noise == True:
            x += sum(ws[-fsim.n_noise:])
            ws.append(np.random.randn(fsim.nx) * fsim.sigma_w)
        xs.append(x)

    us = np.array(us).astype("f")
    xs = np.array(xs).astype("f")

    return (us, Ks, xs) if return_obj else us
```

Listing 7.2: LQR controller implementation

```
class GPT2(BaseModel):
    def __init__(self, n_dims_in, n_positions, n_embd=256, n_layer=12, n_head
        =8, n_dims_out=1):
```

```
        super(GPT2, self).__init__()
        configuration = GPT2Config(
            n_positions=2048,
            n_embd=n_embd,
            n_layer=n_layer,
            n_head=n_head,
            resid_pdrop=0.0,
            embd_pdrop=0.0,
            attn_pdrop=0.0,
            use_cache=False,
        )
        self.n_positions = n_positions
        self.n_dims_in = n_dims_in
        self.n_dims_out = n_dims_out
        self._read_in = nn.Linear(n_dims_in, n_embd)
        self._backbone = GPT2Model(configuration)
        self._read_out = nn.Linear(n_embd, n_dims_out)

    def predict_step(self, input_dict, batch_idx=None):
        xs = input_dict["xs"]
        embeds = self._read_in(xs)
        output = self._backbone(inputs_embeds=embeds).last_hidden_state
        prediction = self._read_out(output)
        return input_dict, {"preds": prediction}

    def forward(self, input_dict, batch_idx=None, return_intermediate_dict=
        False):
        input_dict, intermediate_dict = self.predict_step(input_dict,
            batch_idx)

        output_dict = self.calculate_loss(input_dict, intermediate_dict)

        optimized_loss = 0
        for key, loss in output_dict.items():
            if "loss_" in key:
                optimized_loss += loss
        output_dict["optimized_loss"] = optimized_loss
        return (intermediate_dict, output_dict) if return_intermediate_dict
            else output_dict

    def calculate_loss(self, input_dict, intermediate_dict):
        ys = input_dict["ys"]
        preds = intermediate_dict["preds"]
        res_sq = (preds - ys) ** 2

        output_dict = {}
        output_dict["loss_mse"] = torch.mean(res_sq)
        return output_dict

    def predict_ar(self, ins, fix_window_len=True):
        ins = torch.from_numpy(ins).float().to(self.device)
        one_d = False
        if ins.ndim == 2:
            one_d = True
            ins = ins.unsqueeze(0)
        bsize, points, _ = ins.shape
        d_o = self.n_dims_out
        outs = torch.zeros(bsize, 1, d_o).to(self.device)
        with torch.no_grad():
            for i in range(1, points+1):
                I = ins[:, :i]
                if fix_window_len and I.shape[1] > self.n_positions:
                    I = I[:, -self.n_positions:]
```

```
                _, interm = self.predict_step({"xs": I})
                pred = interm["preds"][:, -1:]
                outs = torch.cat([outs, pred], dim=1)
        outs = outs.detach().cpu().numpy()
        if one_d:
            outs = outs[0]
        return outs
```

Listing 7.3: GPT2 model - Supervised ICL

```
class BaseModel(pl.LightningModule):
    def __init__(self):
        super(BaseModel, self).__init__()

    def forward(self, input_dict, current_epoch=None):
        raise NotImplementedError("Base Class")

    def log_output_dct(self, loss_dict, typ):
        for k in loss_dict:
            if "loss" in k or "metric" in k:
                self.log(typ+"_"+k, loss_dict[k], on_step=True, on_epoch=True,
                         prog_bar=True, logger=True)

        if hasattr(config, "loss_weighting_strategy") and config.
            loss_weighting_strategy in ["gradnorm", "my_gradnorm"] and typ ==
            "train":
            for name, param in self.loss_weighting_strategy.lw_dict.items():
                self.log(name, param.detach().item(), on_step=True,
                         on_epoch=True, prog_bar=True, logger=True)

    def training_step(self, input_dict, batch_idx):
        intermediate_dict, loss_dict = self(
            input_dict, batch_idx=batch_idx, return_intermediate_dict=True)
        self.log_output_dct(loss_dict, "train")
        return {"loss": loss_dict["loss_quadratic"],
                "intermediate_dict": intermediate_dict,
                "loss_dict": loss_dict}


    def on_after_backward(self):
        if hasattr(config, "loss_weighting_strategy") and config.
            loss_weighting_strategy in ["gradnorm", "my_gradnorm"]:
            self.loss_weighting_strategy.normalize_coeffs()

    def validation_step(self, input_dict, batch_idx):
        intermediate_dict, loss_dict = self(
            input_dict, batch_idx=batch_idx, return_intermediate_dict=True)
        self.log_output_dct(loss_dict, "val")
        return {"loss": loss_dict["loss_quadratic"],
                "intermediate_dict": intermediate_dict,
                "loss_dict": loss_dict}

    def test_step(self, input_dict, batch_idx):
        loss_dict = self(input_dict, batch_idx=batch_idx)
        self.log_output_dct(loss_dict, "test")

    def configure_optimizers(self):

        optimizer = torch.optim.AdamW(self.parameters(), lr=config.
            learning_rate,
                                      weight_decay=config.weight_decay)
        return optimizer
```

Listing 7.4: Base model inheriting Pytorch Lightning module

```
class GPT2(BaseModel):
    def __init__(self, n_dims_in, n_positions, context_wdw, n_embd=32, n_layer
        =6, n_head=4, n_dims_out=1):
        super(GPT2, self).__init__()
        configuration = GPT2Config(
            n_positions= 2 * n_positions,
            n_embd=n_embd,
            n_layer=n_layer,
            n_head=n_head,
            resid_pdrop=0.0,
            embd_pdrop=0.0,
            attn_pdrop=0.0,
            use_cache=False,
        )
        self.wdw = context_wdw
        self.n_positions = n_positions
        self.n_dims_in = n_dims_in
        self.n_dims_out = n_dims_out
        self._read_in = nn.Linear(n_dims_in, n_embd)
        self._backbone = GPT2Model(configuration)
        self._read_out = nn.Linear(n_embd, n_dims_out)


    def simulate_linear_dynamics(
        self, x_k: torch.Tensor,
        u_k: torch.Tensor,
        A: torch.Tensor,
        B: torch.Tensor,
        sigma_w: torch.Tensor,
        n_noise: torch.Tensor,
        noise_buffer: torch.Tensor,
    ):
        B_sz, nx = x_k.shape

        x = x_k.unsqueeze(-1)
        u = u_k.unsqueeze(-1)

        Ax = torch.bmm(A, x)
        Bu = torch.bmm(B, u)

        noise_avg = noise_buffer[:, -int(n_noise[0].item()):, :].sum(dim=1) #
            (B, nx)
        noise_avg = noise_avg.unsqueeze(-1)

        x_k1 = Ax + Bu + noise_avg
        x_k1 = x_k1.squeeze(-1)
        x_k1 = x_k1.unsqueeze(1)

        new_noise = torch.randn(B_sz, 1, nx, device=self.device) * sigma_w.
            view(-1, 1, 1)
        noise_buffer = torch.cat([noise_buffer, new_noise], dim=1)

        return x_k1, noise_buffer


    def predict_step(self, input_seq: torch.Tensor):

        embeds = self._read_in(input_seq)
        output = self._backbone(inputs_embeds=embeds).last_hidden_state
        prediction = self._read_out(output)

        return prediction
```

```
    def predict_ar(self, input_dict: dict[str, torch.Tensor], fix_window_len:
        Optional[bool] =True):

        x0 = input_dict["xs"].unsqueeze(1)
        input_seq = x0
        bsize, _ , nx = x0.shape

        sigma_w_batch = input_dict["sigma_w"]
        n_noise_batch = input_dict["n_noise"]
        noise_buffer = torch.randn(bsize, 1, nx, device=self.device) *
            sigma_w_batch.view(-1, 1, 1)

        outs = []

        for i in range(1, self.n_positions + 1):
            context = input_seq[:, :i, :]

            wdw = self.wdw
            if fix_window_len and context.shape[1] > wdw:
                context = context[:, -wdw:]

            interm_preds = self.predict_step(context)

            pred = interm_preds[:, -1:, :]
            state = context[:,-1:, :]

            new_state, new_buffer = self.simulate_linear_dynamics(
                state.squeeze(1),
                pred.squeeze(1),
                input_dict["A"],
                input_dict["B"],
                sigma_w_batch,
                n_noise_batch,
                noise_buffer,
            )

            noise_buffer = new_buffer
            input_seq = torch.cat([input_seq, new_state], dim=1)
            outs.append(pred)

        outs = torch.cat(outs, dim=1)  # Shape: (B, T, nu)
        return input_dict, {"inputs": input_seq, "outputs": outs }



    def calculate_losses_and_metrics(self, input_dict: dict[str, torch.Tensor
        ], intermediate_dict: dict[str, torch.Tensor]):

        x_traj = intermediate_dict["inputs"]
        u_seq = intermediate_dict["outputs"]
        Q = input_dict["Q"]
        R = input_dict["R"]
        QN = input_dict["QN"]
        _ , T, _ = u_seq.shape

        cost = None

        for t in range(T):
            if t == 0:
                cost_t = torch.einsum('bi,bij,bj->b', x_traj[:, 0, :], QN,
                    x_traj[:, 0, :])
```

```
            else:
                cost_t_x = torch.einsum('bti,bij,btj->bt', x_traj[:,0:t,:], Q,
                     x_traj[:,0:t,:]).mean(dim=1)*(t+1)
                cost_t_u = torch.einsum('bti,bij,btj->bt', u_seq[:,0:t,:], R,
                    u_seq[:,0:t,:]).mean(dim=1)*(t+1)
                cost_t_xf = torch.einsum('bi,bij,bj->b', x_traj[:, t, :], QN,
                    x_traj[:, t, :])
                cost_t = cost_t_x + cost_t_u  + cost_t_xf
            if cost == None:
                cost = cost_t
            else:
                cost = cost + cost_t # (B)
        cost = cost.mean(dim=0)
        cost = cost/(2*T)

        return {"loss_quadratic": cost}



    def forward(self, input_dict: dict[str, torch.Tensor], batch_idx: Optional
        [int] = None, return_intermediate_dict: Optional[bool] = False):

        if input_dict["dataset_typ"][0] == "drone":
            input_dict, intermediate_dict = self.predict_nonlinear(input_dict)
        else:
            input_dict, intermediate_dict = self.predict_ar(input_dict)
        loss_dict = self.calculate_losses_and_metrics(input_dict,
            intermediate_dict)

        return (intermediate_dict, loss_dict) if return_intermediate_dict else
             loss_dict
```

Listing 7.5: GPT2 model - Unsupervised ICL

```
def apply_mpc(fsim, xs, traj_len, n_horizon):
    Q = np.eye(fsim.nx)
    R = np.eye(fsim.nu)
    QN = np.eye(fsim.nx)

    us = []

    for t in range(xs.shape[0]-1):
        x_var = cp.Variable((fsim.nx, n_horizon + 1))
        u_var = cp.Variable((fsim.nu, n_horizon))

        cost = 0
        constraints = [x_var[:, 0] == xs[t]]

        for k in range(n_horizon):
            cost += cp.quad_form(x_var[:, k], Q) + cp.quad_form(u_var[:, k], R
                )

            constraints += [x_var[:, k + 1] == fsim.A @ x_var[:, k] + fsim.B @
                 u_var[:, k]]
        cost += cp.quad_form(x_var[:, n_horizon], QN)  # Optional terminal
            cost

        prob = cp.Problem(cp.Minimize(cost), constraints)
        prob.solve(solver=cp.OSQP)

        u_opt = u_var[:, 0].value if u_var.value is not None else np.zeros((
            fsim.nu,))
        us.append(u_opt)
```

```
    return us
```

Listing 7.6: Control Input Generation based on LQR states trajectory

```
def apply_lqr(fsim:FilterSim, n_traj, x0=None):
    Q = np.eye(fsim.nx)
    R = np.eye(fsim.nu)
    QN = np.eye(fsim.nx)
    Ks = fsim.solve_riccati( fsim.A, fsim.B, Q, R, QN, n_traj)

    us = []
    xs = [np.random.randn(fsim.nx) if x0 is None else x0]
    ws = [np.random.randn(fsim.nx) * fsim.sigma_w for _ in range(fsim.n_noise)
        ]

    for t in range(n_traj):
        u_opt = -Ks[t] @ xs[-1]
        us.append(u_opt)

        x = fsim.A @ xs[-1] + fsim.B @ u_opt + sum(ws[-fsim.n_noise:])
        ws.append(np.random.randn(fsim.nx) * fsim.sigma_w)
        xs.append(x.copy())

    return us, xs
```

Listing 7.7: Noisy systems trajectories simulation using LQR

```
def simulate_with_lqg(self, traj_len, Q, R, SN, x0=None):

        Ks = self.solve_riccati(self.A, self.B, Q, R, SN, traj_len)

        kf = KalmanFilter(dim_x=self.nx, dim_z=self.ny, dim_u=self.nu)
        kf.F = self.A
        kf.H = self.C
        kf.B = self.B
        kf.Q = np.eye(self.nx) * self.sigma_w ** 2
        kf.R = np.eye(self.ny) * self.sigma_v ** 2
        kf.x = np.random.randn(self.nx) if x0 is None else x0

        vs = [np.random.randn(self.ny) * self.sigma_v for _ in range(self.
            n_noise)]
        ws = [np.random.randn(self.nx) * self.sigma_w for _ in range(self.
            n_noise)]

        x_true = kf.x.copy()
        xs = [x_true.copy()]
        xs_est = [x_true.copy()]

        us = []
        ys = []

        for i in range(traj_len):
            u = -Ks[i] @ kf.x
            us.append(u)

            x = self.A @ xs[-1] + self.B @ u + sum(ws[-self.n_noise:])
            xs.append(x)

            ws.append(np.random.randn(self.nx) * self.sigma_w)
            vs.append(np.random.randn(self.ny) * self.sigma_v)
            y = self.C @ x + sum(vs[-self.n_noise:])
            ys.append(y)
```

```
            kf.update(y)
            kf.predict(u=u)

            xs_est.append(kf.x)

        return xs, xs_est, us, Ks
```

Listing 7.8: Noisy systems trajectories simulation using LQG

```
def simulate_with_pid_batched(traj_len, A, Bmat, C, Kp_val, Ki=0.0, Kd=0.0,
    y_ref=None, x0=None, controller_type="PID"):
    B, nx = x0.shape
    nu = config.nu
    ny = config.ny

    # Ensure gain shapes are (B, ny)
    def ensure_batch_shape(param):
        param = np.asarray(param)
        if param.ndim == 0:
            return np.full((B, nu, ny), param)
        elif param.ndim == 1:
            return np.broadcast_to(param, (B, ny))
        return param

    Kp = np.zeros((B, nu, ny))
    for i in range(min(nu, ny)):
        Kp[:, i, i] = Kp_val
    Ki = ensure_batch_shape(Ki)
    Kd = ensure_batch_shape(Kd)

    if y_ref is None:
        y_ref = np.zeros((B, ny))


    # Initialization
    xs = np.zeros((B, traj_len + 1, nx))
    us = np.zeros((B, traj_len, nu))
    ys = np.zeros((B, traj_len, ny))

    # Noise sequences
    vs = [np.random.randn(config.ny) * 1e-1 for _ in range(config.n_noise)]
    ws = [np.random.randn(config.nx) * 1e-1 for _ in range(config.n_noise)]


    xs[:,0,:] = x0
    error_integral = np.zeros((B, ny))
    error_prev = np.zeros((B, ny))

    for t in range(traj_len):
        xt = xs[:, t, :]  # (B, nx)

        # Output with noise
        yt = np.einsum('bij,bj->bi', C, xt) + sum(vs[-config.n_noise:])
        #yt = (C @ xt) + sum(vs[-config.n_noise:])  # (B, ny)
        ys[:, t, :] = yt

        # Error terms
        err = y_ref - yt
        error_integral += err
        err_deriv = err - error_prev
        error_prev = err

        # Compute control
        u = (
```

```
            np.einsum('bij,bj->bi', Kp, err) +
            (np.einsum('bij,bj->bi', Ki, error_integral) if controller_type in
                ["PI", "PID"] else 0) +
            (np.einsum('bij,bj->bi', Kd, err_deriv) if controller_type == "PID
                " else 0)
        )
        us[:, t] = u

        # System evolution
        xt_next = np.einsum('bij,bj->bi', A, xt) + np.einsum('bij,bj->bi',
            Bmat, u)  + sum(ws[-config.n_noise:])
        xs[:, t + 1, :] = xt_next

        # Append new noise
        ws.append(np.random.randn(config.nx) * 1e-1)
        vs.append(np.random.randn(config.ny) * 1e-1)

    return np.array(xs) , np.array(us) , np.array(ys)
```

Listing 7.9: Noisy systems trajectories simulation using proportional controller

Bibliography

[1] Hewlett Packard Enterprise. *What is AI in Healthcare?* Accessed: 2025-05-16. n.d. URL: https://www.hpe.com/be/en/what-is/ai-healthcare.html.

[2] European Space Agency. *Inside ESA's European Astronaut Centre (EAC): Where space careers begin*. LinkedIn, accessed: 2025-05-16. 2023. URL: https://www.linkedin.com/pulse/inside-esas-european-astronaut-centre-eac-where-azg9e/.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th. Hoboken: Pearson, 2021. ISBN: 978-0-1346-1099-3.

[4] Ashish Vaswani et al. "Attention is All You Need". In: *Advances in Neural Information Processing Systems* 30 (2017). URL: https://arxiv.org/abs/1706.03762.

[5] Swagat Kumar. "Controlling an Inverted Pendulum with Policy Gradient Methods – A Tutorial". In: *arXiv preprint arXiv:2105.07998* (2021). URL: https://arxiv.org/abs/2105.07998.

[6] Lucian Buşoniu et al. "Reinforcement learning for control: Performance, stability, and deep approximators". In: *Annual Reviews in Control* 46 (2018), pp. 8–28. ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2018.09.005. URL: https://www.sciencedirect.com/science/article/pii/S1367578818301184.

[7] Haldun Balim et al. "Can Transformers Learn Optimal Filtering for Unknown Systems?" In: *arXiv preprint* 2308.08536v3 (2023). URL: https://arxiv.org/abs/2308.08536.

[8] Marco Forgione, Filippo Pura, and Dario Piga. "From System Models to Class Models: An In-Context Learning Paradigm". In: *IEEE Control Systems Letters* 7 (2023), pp. 3513–3518. DOI: 10.1109/LCSYS.2023.3335036.

[9] Yingcong Li et al. *Transformers as Algorithms: Generalization and Stability in In-context Learning*. 2023. arXiv: 2301.07067 [cs.LG]. URL: https://arxiv.org/abs/2301.07067.

[10] Davide Celestini et al. "Transformer-based Model Predictive Control: Trajectory Optimization via Sequence Modeling". In: *IEEE Robotics and Automation Letters* 9.11 (2024), pp. 9820–9827. DOI: 10.1109/LRA.2024.3466069. URL: https://ieeexplore.ieee.org/document/10685132.

[11] Bo-Kai Ruan, Hong-Han Shuai, and Wen-Huang Cheng. *Vision Transformers: State of the Art and Research Challenges*. 2022. arXiv: 2207.03041 [cs.CV]. URL: https://arxiv.org/abs/2207.03041.

[12] Lorenzo Papa et al. "A Survey on Efficient Vision Transformers: Algorithms, Techniques, and Performance Benchmarking". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46.12 (Dec. 2024), pp. 7682–7700. ISSN: 1939-3539. DOI: 10.1109/tpami.2024.3392941. URL: http://dx.doi.org/10.1109/TPAMI.2024.3392941.

[13] Simon J. D. Prince. *Understanding Deep Learning*. The MIT Press, Nov. 2024. URL: http://udlbook.com.

[14] Xinyi Wu et al. "On the Role of Attention Masks and LayerNorm in Transformers". In: *arXiv preprint arXiv:2405.18781* (2024). URL: https://arxiv.org/abs/2405.18781.

[15] Frank Cole et al. *In-Context Learning of Linear Systems: Generalization Theory and Applications to Operator Learning*. 2025. arXiv: 2409.12293 [cs.LG]. URL: https://arxiv.org/abs/2409.12293.

[16] Ekin Akyürek et al. *What learning algorithm is in-context learning? Investigations with linear models*. 2023. arXiv: 2211.15661 [cs.LG]. URL: https://arxiv.org/abs/2211.15661.

[17] Timothy Hospedales et al. "Meta-Learning in Neural Networks: A Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.9 (2022), pp. 5149–5169. DOI: 10.1109/TPAMI.2021.3079209.

[18] IBM Editorial Team. *What is Meta-Learning?* https://www.ibm.com/think/topics/meta-learning. Accessed: 2025-05-27. 2022. URL: https://www.ibm.com/think/topics/meta-learning.

[19] Yan Pei et al. *An Elementary Introduction to Kalman Filtering*. 2019. arXiv: 1710.04055 [eess.SY]. URL: https://arxiv.org/abs/1710.04055.

[20] Control Tutorials for MATLAB and Simulink. *PID Control - Control Tutorials for MATLAB and Simulink*. https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID#7. Accessed: 2025-05-15. 2023.

[21] Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Second. Version v3.1.5 (2020-07-24). Princeton University Press, 2020. URL: https://fbsbook.org.

[22] Somil Bansal. *Special Lecture on the Linear Quadratic Regulator*. https://smlbansal.github.io/Papers/lqrlecture.pdf. Accessed: 2025-04-08. 2018.

[23] M.M. Konstantinov and G.B. Pelova. "Sensitivity of the solutions to differential matrix Riccati equations". In: *IEEE Transactions on Automatic Control* 36.2 (1991), pp. 213–215. DOI: 10.1109/9.67297.

[24] P.Hr Petkov, N.D Christov, and M.M Konstantinov. "On the numerical properties of the schur approach for solving the matrix riccati equation". In: *Systems Control Letters* 9.3 (1987), pp. 197–201. ISSN: 0167-6911. DOI: https://doi.org/10.1016/0167-6911(87)90040-5. URL: https://www.sciencedirect.com/science/article/pii/0167691187900405.

[25] Warren S. McCulloch and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

[26] John McCarthy et al. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. Tech. rep. Dartmouth College, 1955. URL: http://jmc.stanford.edu/articles/dartmouth/dartmouth.pdf.

[27] Marvin L. Minsky and Seymour A. Papert. *Perceptrons*. Cambridge, MA: MIT Press, 1969. ISBN: 978-0-262-63022-0.

[28] Paul J. Werbos. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". PhD thesis. Harvard University, 1974.

[29] George Cybenko. "Approximation by Superpositions of a Sigmoidal Function". In: *Mathematics of Control, Signals, and Systems* 2.4 (1989), pp. 303–314. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[30] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". Diploma thesis. PhD thesis. Technische Universität München, 1991. URL: https://www.bioinf.jku.at/publications/older/hochreiter91.pdf.

[31] Geoffrey E. Hinton and Ruslan R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786 (2006), pp. 504–507. DOI: 10.1126/science.1127647. URL: https://www.cs.toronto.edu/~hinton/science.pdf.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**