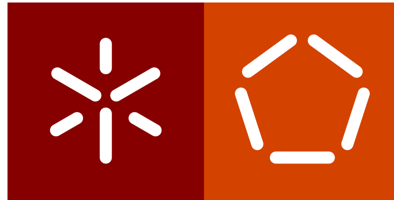
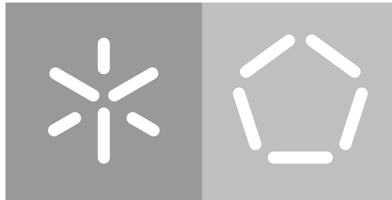


Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Pedro Milhazes Carvalho Pinto

Analysis of Human-Computer Interaction Time Series using Deep Learning

November 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Pedro Milhazes Carvalho Pinto

Analysis of Human-Computer Interaction Time Series using Deep Learning

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Paulo Novais
André Pimenta

November 2021

AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

This is an academic work which can be utilized by third parties given that the rules and good practices internationally accepted, regarding author copyrights and related copyrights.

Therefore, the present work can be utilized according to the terms provided in the license bellow.

If the user needs permission to use the work in conditions not foreseen by the licensing indicated, the user should contact the author, through the RepositóriUM of University of Minho.

License provided to the users of this work

Attribution-NonCommercial

CC BY-NC

https://creativecommons.org/licenses/by-nc/4.0/

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Insert name

____

ACKNOWLEDGEMENTS

This dissertation was one of the biggest challenges in my academic journey and would not have been successful without the help of many people.

I thank my supervisors André Pimenta and Paulo Novais. Besides guiding me towards success in this project, they provided me with countless opportunities to grow technically, professionally, and personally. They inspire me in the sense that both of them actively pursue knowledge and use it for the greater good.

A big thank you to all of my former colleagues at Anybrain. Their hard work and the product they're building offered the necessary context to develop the work in this dissertation. Being able to contribute with my knowledge and learning ability to a venture such as Anybrain was an impactful experience.

I would also like to express my appreciation for the friendship and adventures experienced alongside my friends and colleagues at the University of Minho. These were some truly incredible years which I will never forget.

A deep feeling of love and gratitude goes towards my girlfriend Inês. We trod this path together while giving advice, motivation, support, and love to each other. Whenever I had a difficult time, she was the first - and sometimes the only one - to listen to my problems and the one helping me surpass whatever challenge was ahead.

Finally and most importantly, the ultimate feeling of appreciation is for my family. They nurtured my love for knowledge and science, taught me personal values, and contributed everything they had to my success. I am humbled and aware that I can never repay them, so all I can do is my best to make them proud, fulfilling the dreams they have for me. My deepest wish is to, someday, do the same as my parents did.

ABSTRACT

The collection and use of data resulting from human-computer interaction are becoming more and more common. These have been allowing for the birth of intelligent systems that extract powerful knowledge, potentially improving the user experience or even originating various digital services. With the rapid scientific advancements that have been taking place in the field of Deep Learning, it is convenient to review the underlying techniques currently used in these systems.

In this work, we propose an approach to the general task of analyzing such interactions in the form of time series, using Deep Learning. We then rely on this approach to develop an anti-cheating system for video games using only keyboard and mouse input data. This system can work with any video game, and with minor adjustments, it can be easily adapted to new platforms (such as mobile and gaming consoles).

Experiments suggest that analyzing HCI time series data with deep learning yields better results while providing solutions that do not rely highly on domain knowledge as traditional systems.

Keywords: Deep Learning, Time Series, Human-Computer Interaction, Ambient Intelligence, Fraud Detection

RESUMO

A recolha e a utilização de dados resultantes da interação humano-computador estão a tornar-se cada vez mais comuns. Estas têm permitido o surgimento de sistemas inteligentes capazes de extrair conhecimento extremamente útil, potencialmente melhorando a experiência do utilizador ou mesmo originando diversos serviços digitais. Com os acelerados avanços científicos na área do Deep Learning, torna-se conveniente rever as técnicas subjacentes a estes sistemas.

Neste trabalho, propomos uma abordagem ao problema geral de analisar tais interações na forma de séries temporais, utilizando Deep Learning. Apoiamo-nos então nesta abordagem para desenvolver um sistema de anti-cheating para videojogos, utilizando apenas dados de input de rato e teclado. Este sistema funciona com qualquer jogo e pode, com pequenos ajustes, ser adaptado para novas plataformas (como dispositivos móveis ou consolas).

As experiências sugerem que analisar dados de séries temporais de interação humano-computador produz melhores resultados, disponibilizando soluções que não são altamente dependentes de conhecimento de domínio como sistemas tradicionais.

Palavras-Chave: Deep Learning, Séries Temporais, Interação Humano-Computador, Inteligência Ambiente, Deteção de Fraude

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation 2
- 1.2 Objectives 3
- 1.3 Why Deep Learning and Sequential Data 3
- 1.4 Document Structure 4

2 STATE OF ART 5
- 2.1 Human-Computer Interaction 5
  - 2.1.1 Behavior Analysis in HCI 6
  - 2.1.2 Video Games as HCI 6
- 2.2 Ambient Intelligence 8
  - 2.2.1 Sensorization and Data Collection 8
  - 2.2.2 Reasoning 9
- 2.3 Machine Learning 10
  - 2.3.1 Learning Paradigms 10
    - 2.3.1.1 Supervised Learning 10
    - 2.3.1.2 Unsupervised Learning 11
  - 2.3.2 Generalization 12
  - 2.3.3 Estimators and Maximum Likelihood 12
  - 2.3.4 Optimization 13
  - 2.3.5 Motivations for Deep Learning 14
- 2.4 Deep Learning 15
  - 2.4.1 Neural Networks 15
  - 2.4.2 Regularization 16
    - 2.4.2.1 Parameter Norm Penalties 17
    - 2.4.2.2 Parameter Sharing 17
    - 2.4.2.3 Semi-Supervised and Multitask Learning 18

2.4.2.4 Early Stopping 19
2.4.2.5 Ensemble Methods and Dropout 19
2.4.2.6 Dataset Augmentation and Noise Introduction 20
2.4.2.7 Adversarial Training 21
2.4.3 Optimization 21
2.4.3.1 Stochastic Gradient Descent 22
2.4.3.2 Challenges in Gradient-based Methods 22
2.4.3.3 Gradient Descent Variants 23
2.4.4 Tools and Frameworks 24
2.5 Sequence Modeling With Deep Learning 28
2.5.1 Sequential Data 28
2.5.2 Models For Sequential Data 29
2.5.2.1 Recurrent Neural Networks 29
2.5.2.2 Convolutional Neural Networks 33
2.5.3 Sequence Modeling Tasks 35
2.5.3.1 Sequence Labeling 35
2.5.3.2 Sequential Anomaly Detection 36
2.5.4 Work in Sequential Data Analysis 37
2.6 Machine Learning in Videogames 39
3 PROPOSED APPROACH 42
3.1 Data Collection 43
3.2 Data Processing Pipeline 45
3.3 Modeling 47
3.3.1 General Model Architecture 49
3.3.2 Hyperparameter Tuning 52
3.3.3 User-centered Cross-validation 53
3.4 Deployment 55
4 FRAUD DETECTION IN VIDEO GAMES 56
4.1 Existing Solutions and Requirements 56
4.2 Dataset Description 57
4.3 Experiments 59

4.3.1 Hyperparameter Search 59
4.3.2 Player-Based Cross-validation 63
4.4 Results Discussion and Possible Improvements 65
5 CONCLUSION 67

LIST OF FIGURES

Figure 1.0.1	Dissertation focus area	2
Figure 2.4.1	A Basic MLP	16
Figure 2.4.2	Multitask Learning	18
Figure 2.4.3	Dropout in MLP	20
Figure 2.5.1	Recurrent Neural Network	30
Figure 2.5.2	Long Short-Term Memory	32
Figure 2.5.3	Sparsity in a CNN	34
Figure 2.5.4	Sequence Labelling Tasks	36
Figure 3.2.1	Unprocessed multivariate time series example	47
Figure 3.2.2	Processed multivariate time series example	47
Figure 3.3.1	Modeling Process	49
Figure 3.3.2	Generic CNN architecture	52
Figure 3.3.3	User-centered cross-validation	54
Figure 3.4.1	Deployment example	55
Figure 4.2.1	Dataset label distribution	58
Figure 4.2.2	Dataset hardware events distribution	58
Figure 4.2.3	Two time series examples from our dataset	58
Figure 4.3.1	Search for number of layers, number of filters, and filter size	60
Figure 4.3.2	Search for number of layers, batch size, and L2 regularization	61
Figure 4.3.3	Pooling vs. no pooling AUC	61
Figure 4.3.4	CNN architecture for cheat detection	62
Figure 4.3.5	Results validated for each user	66

LIST OF TABLES

Table 2.4.1 Deep Learning Frameworks 26
Table 2.5.1 Related Work in Sequential Anomaly Detection 38
Table 2.6.1 Machine Learning in Video Games 40
Table 3.1.1 Keyboard and Mouse Events 43
Table 3.1.2 Event Stream 44
Table 3.2.1 Multivariate time series of the events in Table 3.1.2. 45
Table 3.3.1 Modeling approach axes 48
Table 3.3.2 RNNs vs. CNNs in our use-case 50
Table 3.3.3 CNN architecture features varying in hyperparameter selection. 51
Table 4.3.1 Hyperparameter search space and chosen values. 59
Table 4.3.2 Results of Player-Based Cross-Validation in Triggerbot detection. 64
Table 4.3.3 Results of Player-Based Cross-Validation in Aimbot detection. 64

LIST OF ALGORITHMS

1 Stochastic Gradient Descent 22
2 Stochastic Gradient Descent with momentum 23
3 Adam algorithm 24
4 User-centered Cross-validation. 54

ACRONYMS

A

AD Anomaly Detection.

AMI Ambient Intelligence.

C

CNN Convolutional Neural Networks.

D

DL Deep Learning.

DNN Deep Neural Networks.

H

HCI Human-Computer Interaction.

HMM Hidden Markov Models.

M

ML Machine Learning.

MLE Maximum Likelihood Estimation.

MSE Mean Squared Error.

R

RNN Recurrent Neural Networks.

S

SGD Stochastic Gradient Descent.

SVM Support Vector Machines.

U

UAD Unsupervised Anomaly Detection.

1

INTRODUCTION

The analysis of the interactions taking place between us humans and our technological artifacts is a subject of high relevance. Human-Computer Interaction (HCI) makes use of that analysis to improve those tools and design a better user experience. Ambient Intelligence (AmI), on the other hand, focuses on creating a technological environment that adapts to the user's needs.

The increasingly popular field of Machine Learning has been improving our ability of extracting knowledge related to these interactions, playing a major role in the success of tasks such as sentiment analysis, human activity recognition, the development of automated medical diagnosis, or the development of health-related decision support systems.

The rapid development observed in Machine Learning, and especially in the sub-field of Deep Learning, has been giving birth to ever more diverse and sophisticated tools. Some deep learning models specialize in processing data with a specific structure (images or sequences, for example). To take advantage of these models, we need to formulate our problems accordingly.

We can use deep learning models such as convolutional neural networks to take advantage of the sequential context in human-computer interaction data. In this work, we propose a framework that uses such models to analyze human-computer interaction and apply it to the real-world scenario of fraud detection in video-games, a highly interactive domain.

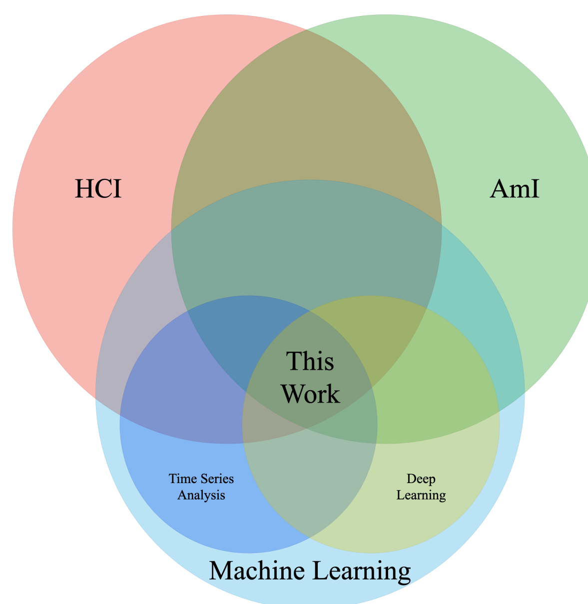


Figure 1.0.1: This diagram illustrates the focus area of our work, at the intersection of human-computer interaction, ambient intelligence, and time-series analysis with deep learning.

1.1 MOTIVATION

Anybrain is a company focused on providing services to the video game industry. The revolutionary aspect of the company's approach is the non-invasive analysis of the interaction taking place between the player and the gaming platform (usually the computer). One of the main ambitions of the company is to create an approach that can be applied to any game, thus being much more scalable than traditional solutions. The data resulting from player interactions with the platform is highly complex, requiring a careful analysis of appropriate techniques and algorithms, which constitutes the main motivation for this work.

Some of the main difficulties this type of analysis are

- Inadequate modeling of the problem
- Scarcity of labeled data

We aim to mitigate the first issue by proposing a set of tools and strategies that are more appropriate for analyzing sequential data of user input.

Many of the existing solutions to these types of tasks involve traditional machine learning models that do not capture temporal patterns in the data. In this work, we will focus primarily on deep models that handle time-series well and have shown positive results in related tasks such as human activity recognition.

1.2 OBJECTIVES

The main goal of this project was to propose a Deep Learning approach for the analysis of human-computer interactions in the form of time-series. We applied that approach to the solutions developed at Anybrain.

At the beginning of this project, we set the following objectives:

- Review scientific concepts and literature related to HCI, AmI, and Sequential Data Analysis with Deep Learning.
- Define a domain that successfully describes the analysis of human-computer interactions in the form of time series. Propose a systematic approach to problems that fit in the established domain.
- Develop an anti-cheating system for video games that is game-agnostic and relies on the proposed deep learning approach to HCI.
- Disseminate our findings and contribute to the scientific knowledge on the fields related to this work.

1.3 WHY DEEP LEARNING AND SEQUENTIAL DATA

Human-computer interaction data can be very complex. For example, if we're analyzing keyboard mouse input data (one of the most common input formats in gaming), there are hundreds of possible events, such as pressing a mouse button or any key of the keyboard.

The curse of dimensionality hinders the use of traditional machine learning methods with data this complex. Many solutions require a preprocessing of the data that extracts a set of human-engineered features.

These approaches have their limitations. Some of them are:

1. The manual feature design by humans is a time-demanding task.
2. The features we come up with might not capture relationships between events.
3. The length of the periods for which we calculate the features might dissolve important context such as order or concentration in time.

Deep Learning models are much more resistant to the curse of dimensionality. Furthermore, there are models such as recurrent neural networks (RNNs) or convolutional neural networks (CNNs) that benefit from structured data. This allows us to address the problems listed above, respectively:

1. Using input data as close as possible to the raw interaction that takes place between the user and the platform, and letting the feature extraction be done automatically by the models.
2. Work with snapshots of the user activity without reducing the dimensionality of the data (losing less information).
3. Preserve the sequential structure of the data (ordinal or temporal context) by using time-series.

The research hypothesis driving this work was:

"If we apply Deep Learning to the analysis of human-computer interaction, we will be able to attain better results in tasks such as behavior analysis-based fraud detection."

1.4 DOCUMENT STRUCTURE

In section 1, we explained the background, motivation and main objectives of this work. In section 2, we explore the research fields related to this work, introducing their key concepts as well as the state of the art in subjects of interest to our work. In section 3 we describe a framework to analyze HCI time-series data. In section 4, we explain how we implemented the generic anti-cheating system based on preciously explained concepts. Finally, in section 5, we state our concluding remarks and suggest future work directions on this subject.

2

STATE OF ART

2.1 HUMAN-COMPUTER INTERACTION

Human-Computer Interaction (HCI) is *a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them* (Hewett et al. (1992)).

Being at the intersection of computer science, behavioral sciences, and design, HCI is a multi-disciplinary field with many applications.

HCI and cognitive science are closely related. While the design of appropriate interfaces benefits from the understanding of the human cognitive process, we can also expand that same understanding by studying the way humans interact with computers (Boring (2002)).

In the past years, **progress in human-computer interface design has been towards multi-modal, active, and adaptive systems**.

Multi-modal interfaces are those in which the user can interact with the system in various ways. Multiple input/output sources represent an increase in complexity if one wishes to analyze the interaction taking place.

Active interfaces interact with the user spontaneously, by offering some recommendation or support an implicit request. The successful implementation of such functionality often requires the analysis of the user's behavior.

Adaptivity is required because users don't all behave in the same way. A user's behavior can even change in the span of a few minutes depending on the task he's performing.

2.1.1 *Behavior Analysis in HCI*

Research in HCI behavior analysis has been increasing in the last years, likely due to the availability of machine learning methods to analyze data.

One of the main applications for this knowledge is informing system design, as seen in applications that apply HCI theory to shape its interaction with the user (Consolvo et al. (2009)).

Another good example is the development of systems that rely on affective computing to provide useful features. These tools might provide helpful insight into the user's health state (Thieme et al. (2020); Pimenta et al. (2014); Carneiro et al. (2015)) or recommendations regarding productivity (Carneiro et al. (2017, 2016); Pimenta et al. (2015)), for example.

As (Hekler et al. (2013)) pointed out, despite the growing interest in this area, much of the research still lacks a base framework to establish standard methods to evaluate or classify behavior.

2.1.2 *Video Games as HCI*

Video games can are a particular case-study in HCI, however, there are some key aspects in which they differ from other types of software (such as productivity tools). As described by (Pagulayan et al. (2003)), we can list some of the main differences as:

- Aiming to provide entertainment (and not to make a certain task easier).
- The tendency to reach a greater variety in methods to complete a task or to interact with the system.
- A game defines it's own goal (to complete/win the game) while most software exists to fulfill an external objective.
- Video games usually generate a higher variability in user experience.
- Usage of more diverse input/output methods.

These aspects certainly must inform software design and even motivate a new framework within HCI to address video games, such as suggested by Barr et al. (2007). Despite the great variety of interaction domains in which we can apply behavior analysis (even among games), we argue that we can use the same tools (similar data structures and models) to analyze HCI.

There is a multitude of possible applications for behavior analysis in video games. Sykes and Brown (2003) analyzed the use of a gamepad and to correlate player arousal with the difficulty of the game. Ravaja et al. (2004) studied the self-awareness state and emotional response of users when playing different types of games. Mandryk and Inkpen (2004) compared players' physiological measures (such as heartbeat rate) when playing against a computer versus when playing against another human.

In this work, we analyze player behavior in the form of multi-modal (keyboard and mouse) input data. If we can model player behavior, we can implement functionalities such as cheat detection solely based on the user's behavior.

2.2 AMBIENT INTELLIGENCE

Ambient Intelligence (AmI) is a term that describes an environment populated with an ensemble of electronic devices that are sensitive and responsive to the presence of people. These devices should cooperate seamlessly to help humans carry out their everyday tasks. Some applications for AmI are smart homes, health monitoring and assistance, hospitals, transportation, emergency services, schools, and workplaces (Cook et al. (2009)).

According to Ducatel et al. (2001), there are certain key technological requirements in AmI:

- Unobtrusive hardware
- Seamless web-based communications infrastructure
- Massively distributed dynamic device networks
- Natural feeling human interface
- Dependability and security

Ambient Intelligence is tangential to our work. Our analysis is not focused on the physical environment but rather on user interaction with digital devices. In this sense, the concepts in AmI that are most relevant to this work are sensorization and reasoning.

2.2.1 *Sensorization and Data Collection*

We can use several sensors to collect data regarding the user and their context. These often generate large volumes of data. If they are not completely reliable, that data might be noisy and can even contain missing values. A common practice is to preprocess data and reduce its volume by calculating descriptive statistical measures. Despite this being an efficient approach, one can argue that it does not provide much flexibility to experiment with other techniques, with a different preprocessing of the raw sensor data or which might require finer granularity. In this sense, while prototyping different reasoning mechanisms, it is often worth to store raw data.

There are many types of sensors used in AmI. Audiovisual, passive infrared (PIR), radio frequency identification (RFID), and multi-modal wearable sensors are some of the most popular, as pointed out by Pauwels et al. (2007).

As we intend to analyze HCI data, audiovisual, wearable, and virtual sensors make the most sense. Frameworks such as proposed in Baltrušaitis et al. (2016, 2018) can be of use to study the user's facial expressions while interacting with the computer. In other approaches, such as Sykes and Brown (2003) and Kołakowska (2013), behavior analysis is based on peripheral input data.

As previously mentioned, our framework for analyzing human-computer interaction relies on peripheral input data (e.g., mouse and keyboard).

2.2.2 *Reasoning*

We can divide reasoning into tasks such as user behavior modeling, activity recognition, activity prediction, and decision making.

In this work, we focus on modeling and activity recognition. Since we plan to approach both tasks using a general and unified data collection and processing approach, we can use similar pattern recognition methods in these tasks.

There are two major approaches to activity recognition: data-driven and knowledge-driven. Knowledge-driven systems rely on real-world observations and our understanding of a specific domain. Data-driven methods are more flexible since they rely on probabilistic methods and data availability, requiring much less domain-specific knowledge.

As pointed out by Chen et al. (2012), traditional machine learning models have been used for sensor-based activity recognition. For example, in the task of WiFi sensor-based human activity recognition, Wang et al. (2015) use Hidden Markov Models (HMMs), while Yin et al. (2008) use a one-class support vector machine (SVM) for anomaly detection.

In the last few years, with the popularization of deep learning, many solutions use deep models to automate feature extraction (instead of relying on manually extracted features based on domain-specific knowledge). Convolutional neural networks (CNNs), recurrent neural networks (RNNs), deep belief networks (DBNs), deep Boltzmann machines (DBMs), and autoencoders are among the most popular deep learning models in activity recognition, according to Nweke et al. (2018).

2.3 MACHINE LEARNING

In this section and the following, we present an overview of Machine Learning and Deep Learning, with help from Goodfellow et al. (2016), where a good introduction on these topics can be found.

A machine learning algorithm is an algorithm that learns from experience. According to Mitchell (1997), *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

The task T performed with the help of machine learning is usually too complex for traditional fixed programs designed by humans. Some examples of possible tasks are classification, regression, transcription, translation, anomaly detection, and denoising.

The performance measure P (e.g. accuracy or error rate) is usually specific to the task to be performed by the system and should reflect the desired behavior of the model.

The experience E can be seen as the exposure to data. Differences in data structures, differences in label availability, and dimensionality are some of the dataset characteristics that might shape the learning paradigm and the statistical models to adopt.

2.3.1 *Learning Paradigms*

We often hear that a machine learning algorithm is supervised, or unsupervised, among other possible denominations. Those terms refer to the learning paradigm that stems from the type of dataset used.

2.3.1.1 *Supervised Learning*

Supervised learning algorithms use datasets that contain a label y for each example $\boldsymbol{x}$. Usually, these algorithms try to correlate the features with a given label (for example, in classification) by estimating $p(y|\boldsymbol{x})$.

Perhaps the simplest example of a supervised learning algorithm is linear regression. In linear regression, we try to predict a value y from an n-dimensional feature vector $\boldsymbol{x}$. For each feature x_i in $\boldsymbol{x}$, our model has a parameter (or weight) w_i. Our prediction $\hat{y}$ can be defined as $\hat{y} = \boldsymbol{w}^{\top}\boldsymbol{x} + b$, where b is an additional parameter called bias, which allows our model to represent affine functions (instead of only allowing for hyperplanes passing in the origin).

Another simple example, closely related to linear regression, is logistic regression. Despite its name, this model is generally used for classification, since it outputs probabilities by applying a logistic function to the output of a linear regression model.

We won't delve into the formulation of additional and more complicated algorithms, since that is out of the scope of this review of the field of machine learning. The examples of linear and logistic regression models go about to show that machine learning models are nothing more than parameters and functions that transform the input into an answer in a given format.

One of the most influential models in traditional machine learning is Support Vector Machines (SVM). This kind of model behaves similarly to linear regression, incorporating non-linear kernels (instead of just multiplying the input by weights). Until the rebirth of the interest in neural networks, SVMs were used in a wide range of tasks to achieve state-of-the-art results.

Another very important family of machine learning models are tree-based models. A decision tree is a rule-based model that successively divides space into smaller regions. A leaf is a region that contains examples of the same class (in classification). Decision trees alone are not usually strong models, however, some of the most used traditional machine learning models consist of training an ensemble of shallow trees (weak learners) and combining their answers to achieve good generalization. Random forests, AdaBoost and Gradient Boosting are some of the most popular of these methods.

2.3.1.2 *Unsupervised Learning*

Unsupervised algorithms don't have access to a label. Their goal is usually to create a representation of the probabilistic distribution that generated the dataset. In this sense, we can say that they attempt to estimate $p(\boldsymbol{x})$ (as opposed to $p(\boldsymbol{y}|\boldsymbol{x})$ in supervised learning).

A popular unsupervised learning task is clustering, which consists of grouping similar examples drawn from a population. K-means clustering is one of the simplest and most popular clustering methods. It works by initializing k centroids randomly and iteratively correcting their position to the mean of every training example assigned to the respective cluster. An example is assigned to the cluster with the closest centroid.

Principal Component Analysis (PCA) is another example of an unsupervised learning algorithm (although it is often mentioned as a dimensionality reduction method). This method learns a linear transformation T that transforms input $\boldsymbol{x}$ into a lower-dimensional representation $\boldsymbol{z}$ such that $T(\boldsymbol{x}) = \boldsymbol{z}$. It achieves so by minimizing the error $\boldsymbol{\varepsilon} = T^{-1}(T(\boldsymbol{x})) - \boldsymbol{x}$.

These two examples illustrate two ways to extract information from a dataset that doesn't contain labels. There are many more methods to perform similar tasks. Some of these methods involve deep learning models called autoencoders.

2.3.2 *Generalization*

Generalization is the ability of a model to perform well on data it has not previously seen (used in training). As we usually have a measure that quantifies the training error, there is also a test error (performance on a testing dataset), also referred to as generalization error.

The generalization error should be lesser or equal to the training error. Our goal is to minimize the gap between these two measures. When the gap is great, it means that our model is overfitting. Underfitting occurs when the model isn't able to achieve a low enough train error.

Several factors influence the model's likeliness to underfit or overfit. Perhaps the easiest way to control it is to change the model's capacity. A model that is too complex (high capacity), might overfit by learning properties that are specific to the training set. A model that is not complex enough might underfit by not being able to learn the properties of the probabilistic distribution that generates the data.

In parametric models such as the ones we've been mentioning, the most straight forward method for controlling capacity is changing the number of parameters. Using a kernel of different complexity in SVMs, limiting the depth of decision trees, or changing the number of layers in a neural network are some examples of ways to control representational capacity for different models.

2.3.3 *Estimators and Maximum Likelihood*

Statistics provides us with several tools that help us to assess the performance of machine learning algorithms. Some of the most important of these tools are estimators. Estimators allow us to generalize in the sense that they predict the value of some property of the data generating distribution from a sample of m examples.

An example of a simple estimator is the sample mean, which is an estimator for the mean parameter of a Gaussian distribution, defined as $\hat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$. Estimators can be described through some properties such as bias or variance.

In machine learning, we are often concerned with estimating an entire function, by estimating the set of parameters $\hat{\boldsymbol{\theta}}$ that better approximates the unknown true set of parameters $\boldsymbol{\theta}$ (which describe the true probabilistic distribution of data). These sets of parameters describe a function that maps a given input $\boldsymbol{x}$ to a result y.

The most common method to derive estimators in machine learning is called Maximum Likelihood Estimation, which consists of maximizing a likelihood function to choose the model that is most likely to have generated the training data. In other words, if we have a set of m independent examples $\boldsymbol{X}$ and a probability function $p_{model}(\boldsymbol{X};\hat{\boldsymbol{\theta}})$ that estimates the true probability of $\boldsymbol{X}$ occurring in a space described by $\boldsymbol{\theta}$, the maximum likelihood estimation of the set of parameters $\boldsymbol{\theta}$ is given by

$$\hat{\boldsymbol{\theta}} = \underset{\hat{\boldsymbol{\theta}}}{\operatorname{argmax}}\, p_{model}(\boldsymbol{X};\hat{\boldsymbol{\theta}})$$

.

Providing additional details or developing this formula is beyond the scope of this review, however, it is worth stating that many of the evaluating functions using in machine learning, such as the mean squared error (MSE) in regression, or cross-entropy in classification, are derived from the principle of maximum likelihood.

2.3.4 *Optimization*

Machine Learning often involves solving optimization problems, such as seen with maximum likelihood estimation, or with the example of linear regression.

We can solve linear regression by simply solving a system of linear equations. In more complex models, however, the optimization problem is usually much harder. In this sense, there are specific algorithms or heuristics to update the model parameters, according to a given evaluation function.

One of the most influential optimization algorithms in machine learning is Stochastic Gradient Descent, which consists of calculating the gradient of the error in the model's predictions, and then updating its parameters according to that gradient. We'll make a more detailed description of this algorithm in the context of Deep Learning.

It is noteworthy that some models, such as decision trees and their ensembles, require very specific algorithms as their parameters interact in a way that doesn't allow for standard gradient-based optimization.

2.3.5 *Motivations for Deep Learning*

Although traditional Machine Learning models are extremely useful and used extensively in a wide range of tasks, they have failed in providing solutions for more complex problems such as object detection or voice recognition.

Deep Learning has been widely adopted in the past decade because it provides answers to some of the problems that impair traditional machine learning methods to solve complex tasks.

One of these problems is the curse of dimensionality. As the number of features or dimensions in input data increases, the number of possible input configurations usually grows exponentially. In this sense, either the volume of training data also grows exponentially, or we are left with a model of a space filled with empty regions (not populated by any training example).

Another problem with traditional machine learning methods is that they tend to incorporate a limited set of prior beliefs (such as local constancy). These fail to generalize for complex probabilistic distributions. When there aren't enough training examples to cover regions of space with abrupt changes, many models might fail to generalize to those regions, returning answers close to the labels of the nearest training examples (an extreme case being the k-nearest neighbors algorithm).

2.4 DEEP LEARNING

"Deep learning is a particular kind of machine learning that achieves great power and flexibility by representing the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones." (Goodfellow et al. (2016)).

Deep models provide a greater statistical capacity and statistical efficiency that allow us to solve otherwise intractable tasks. Additionally, Deep Learning includes the study of proper evaluation functions and metrics, optimization algorithms, and regularization methods from a statistical and empirical perspective.

Although some of the theoretical foundations for Deep Learning had already been developed in the last century (Fukushima (1980)), recent breakthroughs in fields such as Computer Vision (Krizhevsky et al. (2012)) captured the scientific community's attention and revived the interest in the subject.

Although the theoretical foundations for Deep Learning had already been born in the last century, recent breakthroughs in fields such as Computer Vision captured the scientific community's attention and revived the interest in the subject.

In this section, we firstly review some of the main concepts to understand in deep learning. Secondly, we explore models specialized in the processing of sequential data and time-series. Finally, we briefly review the main tools and frameworks that support the development of deep learning-based solutions.

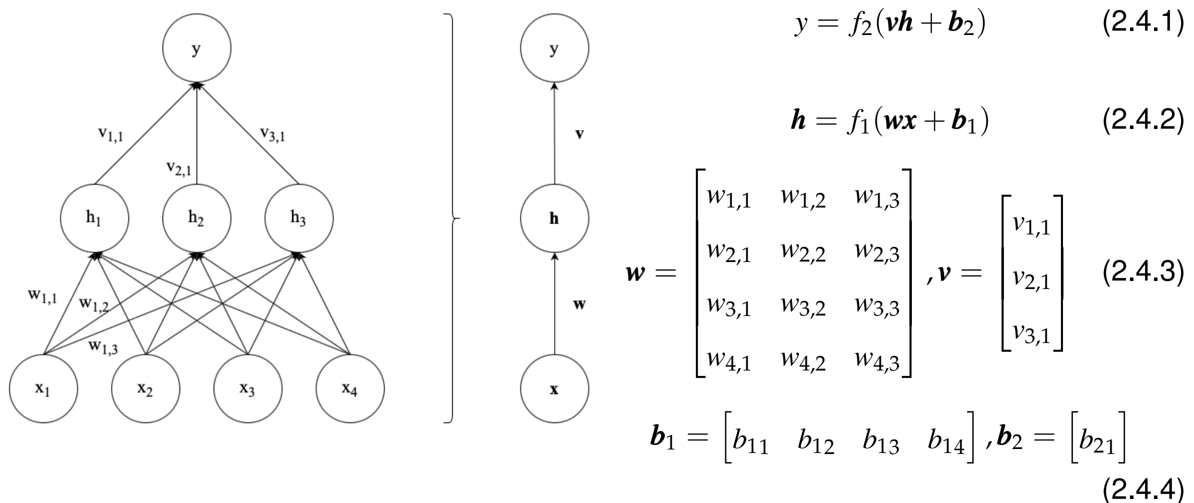
2.4.1 *Neural Networks*

Artificial Neural Networks are the most used model in Deep Learning. The basic building blocks of a neural network are fairly simple. In this sense, it is possible to increase the complexity of models without any additional effort.

The name Neural Networks stems from the biological inspiration of these models. Each unit resembles a neuron in the sense that it receives input from (either from the input data or other neurons) and computes its output.

The most basic (and common) type of neural network is the Multilayer Perceptron (MLP). This type of model is organized in groups of units called layers. Each layer's neurons connect to all of the neurons in the previous and next layers. For each connection in the neural network, we have a parameter that stores it's weight or strength.

A way to understand how neural networks work is to think of them as a chain of functions, where each layer behaves similarly to the linear models described in section 2.3.1.1. We should notice that neural networks can learn non-linear distributions because each unit applies a non-linear function to its output.



$$y = f_2(\boldsymbol{v}\boldsymbol{h} + \boldsymbol{b}_2) \tag{2.4.1}$$

$$\boldsymbol{h} = f_1(\boldsymbol{w}\boldsymbol{x} + \boldsymbol{b}_1) \tag{2.4.2}$$

$$\boldsymbol{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix}, \boldsymbol{v} = \begin{bmatrix} v_{1,1} \\ v_{2,1} \\ v_{3,1} \end{bmatrix} \tag{2.4.3}$$

$$\boldsymbol{b}_1 = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \end{bmatrix}, \boldsymbol{b}_2 = \begin{bmatrix} b_{21} \end{bmatrix} \tag{2.4.4}$$

Figure 2.4.1: A basic MLP with 4 input features, 3 hidden units, and an 1 output unit. The functions f_1 and f_2 are the activation functions of the units in the hidden layer and the output unit, respectively. $\boldsymbol{w}, \boldsymbol{v}$ are the arrays of parameter that specify the strength of the connections between units. $\boldsymbol{b}_1, \boldsymbol{b}_2$ are the arrays of parameters that define the bias for each unit in the hidden and output layers, respectively.

MLPs are one of the most general deep learning models because they use general matrix multiplication to propagate information between layers. According to the universal approximation theorem, an MLP with a single layer, a large enough number of non-linear units can represent any function in $f\colon \mathbb{R}^n \mapsto \mathbb{R}^m | n, m \in \mathbb{N}$ with an arbitrarily low error rate (Hornik et al. (1989)). However, we have no guarantee that our optimization algorithm will be able to converge to a good solution.

In this sense, there are specialized types of neural networks such as CNNs and RNNs, which impose stronger restrictions on the weights and structure of the network, in order to perform well on specific tasks related to computer vision and sequence modeling, respectively.

2.4.2 *Regularization*

As discussed in section 2.3.2, one of the main concerns when implementing a machine learning algorithm is its ability to generalize to new data. The implementation of strategies that aim to improve model generalization is

called regularization. There are many methods applied to deep learning models in order to lower their generalization error without harming their ability to learn complex functions. Here we review some of the most common and successful regularization methods.

2.4.2.1 *Parameter Norm Penalties*

Parameter norm penalties consist of introducing a penalty in the objective function of the algorithm. If our model is described by the set of parameters $\boldsymbol{\theta}$, and we have an objective function $J(\boldsymbol{\theta};\boldsymbol{X})$, the regularized function $\tilde{J}$ can be defined as

$$\tilde{J}(\boldsymbol{\theta};\boldsymbol{X}) = J(\boldsymbol{\theta};\boldsymbol{X}) + \alpha\Omega(\boldsymbol{\theta})$$

where Ω is the function we use to calculate parameter norm and α is a hyperparameter that defines how strong we want the penalty to be.

The choice of the function represented by Ω gives birth to different methods. Explaining how these different choices prove useful to lower the generalization error of machine learning algorithms is well beyond the scope of this review. The most popular of these methods are

- L^2 Regularization, where $\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{w}||_2^2$
- L^1 Regularization, where $\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1$

2.4.2.2 *Parameter Sharing*

Such as in parameter norm penalties, we might benefit from imposing additional constraints to the model parameters. Parameter sharing consists of forcing parameters to share their value, which means that they should encode a relation of the same type.

Parameter sharing helps the model to achieve better generalization because it doesn't learn separate parameters to describe the same data transformation. It also makes the model considerably more computationally efficient, since we don't need to store multiple variables for parameters with the same value.

Convolutional neural networks are one of the best examples of parameter sharing since they use the same parameters across several regions of an image. This allows CNNs to significantly decrease the number of parameters needed to process images, making training considerably easier. As we'll see, parameter sharing is also present in recurrent neural networks.

2.4.2.3 *Semi-Supervised and Multitask Learning*

The paradigm of semi-supervised learning consists of using labeled and unlabeled examples to perform a given task. In deep learning, this usually involves learning a representation of the data distribution that can be shared by an unsupervised and a supervised model.

To achieve this goal we can make the supervised and unsupervised models share part of their parameters, instead of having two completely separate models. In this way, both labeled and unlabeled examples contribute to learning the data distribution.

A more general concept is multitask learning (Caruana (1993)), in which we have a group of parameters shared across several models that perform different tasks. If the representation expressed in the shared parameters is relevant to the multiple tasks, we can achieve better generalization.

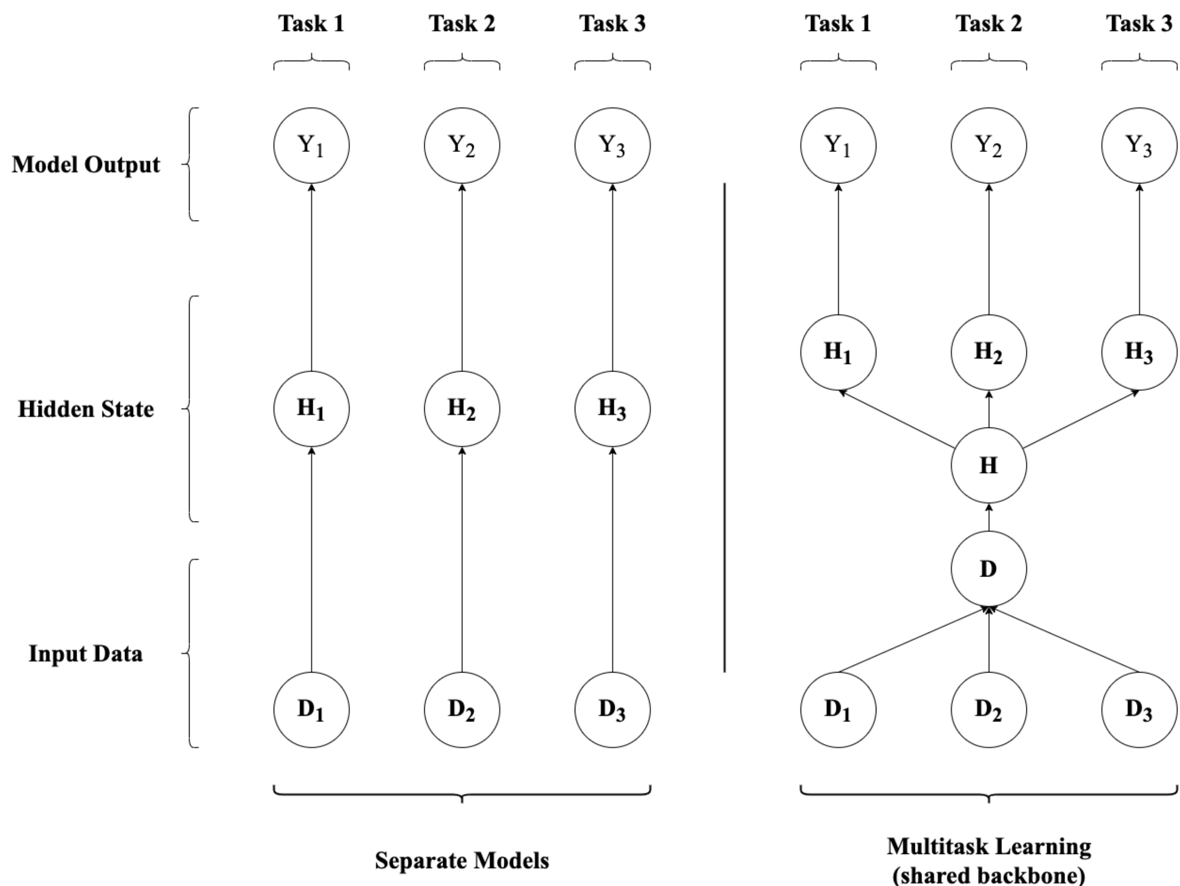


Figure 2.4.2: Suppose we have to solve three tasks that operate on the same type of data (for example, images). For each task, we have a model with a hidden state $\boldsymbol{H}_n$ and a dataset $\boldsymbol{D}_n$. In multi-task learning, we force the three models to share part of their hidden state $\boldsymbol{H}$. This allows backpropagation from all three outputs to benefit the representation learned by the three models and the separate tasks to benefit from the entire dataset $\boldsymbol{D}$.

2.4.2.4 *Early Stopping*

One of the most common forms of regularization in deep learning is early stopping. This technique consists of storing a copy of the model's parameters every time its validation performance measures improve. We then use the model with the lowest generalization error.

The main advantage of this method is the fact that it doesn't require any modification to the model's parameters or the objective function.

2.4.2.5 *Ensemble Methods and Dropout*

Given the stochastic factors in the training of machine learning models (in parameter initialization or data sampling, for example), it is often unlikely that an algorithm converges to the same model twice.

In this sense, we can train several models using the same process. If these models make independent errors, we can combine their answers to make even better predictions.

This is the core idea in bagging (Breiman (1996)), which is a method that aggregates the answers of several models under the assumption that they make independent errors. These models are trained with datasets randomly selected with replacement from the original dataset. If the errors are correlated, bagging performs on average at least as well as any of the models in the ensemble.

There are other types of ensemble methods, such as boosting, which doesn't exhibit regularization properties, since it works by building increasingly complex models.

Another regularization method that can be seen as bagging is called dropout (Srivastava et al. (2014)), which works by randomly dropping connections in a neural network at train time. Dropout provides a way to train many less dense models without the significant computational overhead associated with other bagging methods.

The ability to train these models simultaneously and implicitly is due to the parameter sharing between them. This parameter sharing motivates the whole network to learn redundant units since if a certain feature is not present, the network can still make a correct prediction based on other information.

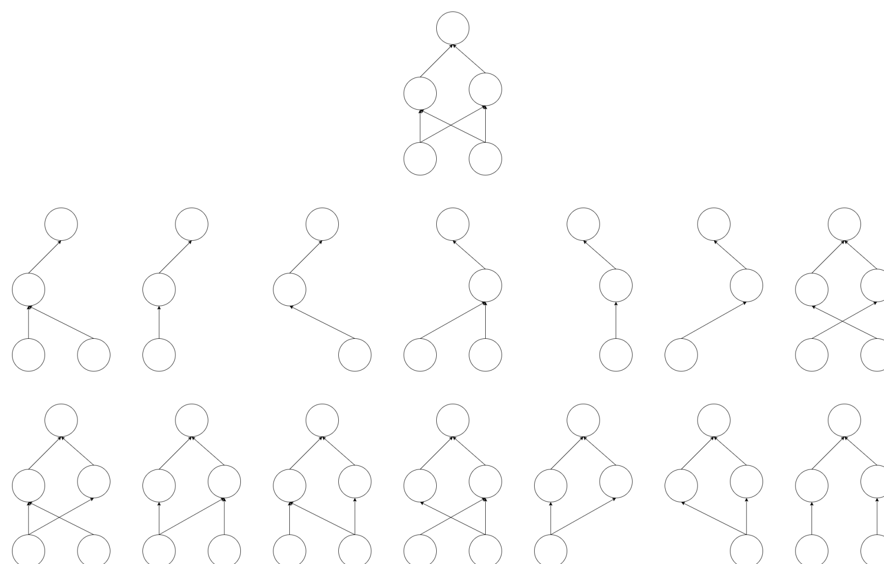


Figure 2.4.3: This picture illustrates the possible set of graphs obtainable by applying dropout to the simple MLP above. For models with a high number of parameters (modern deep learning models can have millions of parameters), we cannot possibly keep track of the possible variations.

As dropout affects the structure of the network, some models might benefit from a slightly different approach. Some research aims to create variants of dropout that attend to this problem, such as Ghiasi et al. (2018) in the case of CNNs.

2.4.2.6 *Dataset Augmentation and Noise Introduction*

The volume of training data is one of the most important factors for an algorithm to generalize well. Data augmentation consists of applying some transformation to training data to generate synthetic training examples. It is one of the main methods for overcoming limited training data and increase model robustness.

Although his approach is not applicable to all machine learning tasks, it is effective in classification or any other tasks where we can ensure the data transformation does not also alter labels.

The specific transformations vary with the type of data used by the models. For example, if we are dealing with image classification, one way of performing data augmentation is applying geometric transformations such as rotation or reflection. Coming up with new and more effective data augmentation methods through geometric transformation is an active research subject (Devries and Taylor (2017); Zhang et al. (2017); Yun et al. (2019)).

Some transformations work for almost any data format, such as the introduction of noise in training data. This method is particularly useful to address the vulnerability of neural networks to noise in the input.

Noise can also be applied directly to the weights of the neural network, which is equivalent to applying a norm penalty according to Bishop (1995).

2.4.2.7 *Adversarial Training*

Let us have a classifier f that maps a data example $\boldsymbol{x}$ into a class y. An adversarial example is an input $\boldsymbol{x'}$ that is indistinguishable to the human eye, and for which $f(\mathbf{x'}) \neq y$.

As noise can highly affect neural network predictions, Szegedy et al. (2014) points out that we can use an algorithm to maximize the classifier's error by training a generative model that applies noise to existing training examples to generate adversarial examples.

This is interesting in the context of regularization because one can use these adversarial examples to train a model and increase its robustness to noise.

As suggested by Goodfellow et al. (2014), we can train a generative (the adversarial model) and a discriminative (the classifier) model simultaneously to iteratively improve the ability to generate adversarial models, which can be used to improve the robustness of the discriminative model.

2.4.3 *Optimization*

As discussed in section 2.3.4, a machine learning algorithm usually involves optimizing the model parameters according to an objective function.

What distinguishes machine learning from optimization is that we don't want to minimize the training error. What we want is to minimize the test error. We assume we can do this indirectly because the data generating process is the same for the training and test data.

Gradient-based optimization is the universally adopted family of methods used to optimize neural networks (and other machine learning models). Stochastic Gradient descent is the base for almost all of the optimization algorithms in deep learning.

In this section, we firstly introduce stochastic gradient descent (SGD), then list some of the main challenges in gradient-based optimization, and finally present some of the variations of this algorithm that are currently most used and aim to solve some of the aforementioned challenges.

2.4.3.1 *Stochastic Gradient Descent*

Stochastic Gradient Descent (SGD, described in algorithm 1) works by iteratively sampling minibatches of m training examples and subtracting the gradient of the objective function $f(\boldsymbol{x};\boldsymbol{\theta})$ to the parameters of the model.

Algorithm 1: Stochastic Gradient Descent

$\varepsilon \leftarrow$ learning rate;

$\theta \leftarrow$ initial model parameters;

while *stopping criterion not met* **do**

  Sample minibatch $\boldsymbol{x}$ of size m from training set $\boldsymbol{X}$;

  Calculate loss function: $L(\boldsymbol{x};\boldsymbol{\theta})$;

  Calculate gradient of loss function: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x};\boldsymbol{\theta})$;

  Update model parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \boldsymbol{g}$;

end

The main innovation of gradient descent (relative to standard gradient descent, that exists since the 19th century) is the random sampling of minibatches. Minibatch sampling allows us to obtain an unbiased estimate of the gradient for the whole training set.

It is also important to notice that the learning rate usually decreased throughout the iterations, to minimize noise introduced by the random sampling of the minibatches.

2.4.3.2 *Challenges in Gradient-based Methods*

The optimization of neural networks is a nonconvex problem, which means that a local minimum is not guaranteed to be a global minimum. The high dimensional spaces objective functions in deep learning are extremely complex and pose some challenges to the learning process:

- Local minima
- Other zero-gradient points
- Cliffs and exploding gradients

As suggested by Goodfellow et al. (2015), local minima are empirically shown not to be a major problem to neural networks trained with stochastic gradient descent. A way to detect local minima would be to track the norm of the model parameters, which would shrink to a very small size when trapped in a region with a null gradient.

In the case of exploding gradients, common in recurrent neural networks, we can apply techniques such as gradient clipping, which consists of imposing a limit to the gradient norm, preventing potentially destructive updates to the parameters.

There are other challenges to gradient-based optimization, but these suffice to understand the motivation behind the most widely used gradient descent variations.

2.4.3.3 *Gradient Descent Variants*

MOMENTUM As standard SGD can be slow, the method of momentum (Polyak (1964)) became common to accelerate learning when facing consistent gradients. This is accomplished by accumulating past gradients in a velocity variable that decays exponentially.

Algorithm 2: Stochastic Gradient Descent with momentum

$\varepsilon \leftarrow$ learning rate;
$\boldsymbol{v} \leftarrow$ initial velocity;
$\alpha \leftarrow$ momentum parameter;
$\theta \leftarrow$ initial model parameters;
while *stopping criterion not met* **do**
  Sample minibatch $\boldsymbol{x}$ of size m from training set $\boldsymbol{X}$;
  Calculate loss function: $L(\boldsymbol{x};\boldsymbol{\theta})$;
  Calculate gradient of loss function: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x};\boldsymbol{\theta})$;
  Update velocity: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \boldsymbol{g}$;
  Update model parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$;
end

Introducing this concept analogous to velocity in gradient descent is useful to overcome obstacles such as local minima that could otherwise disturb the optimization process.

ADAPTIVE LEARNING RATES Some optimization implements more sophisticated ways of adapting the learning rate and the velocity of the algorithm.

RMSProp is an improvement of the AdaGrad algorithm proposed by (Duchi et al. (2011)). Both these algorithms reduce the learning rate according to the history of the squared gradient. The improvement introduced by RMSProp is an exponential decay applied to that history so that the learning rate doesn't shrink when little the gradients are too small.

RMSProp is usually combined with momentum, being one of the most popular optimization algorithms for deep neural networks.

Another widely used algorithm is Adam (Kingma and Ba (2017)), which gets this name because it tries to implement adaptive moments for both velocity and learning rate. Adam is usually very robust to different hyper-parameter choices.

Algorithm 3: Adam algorithm

$\varepsilon \leftarrow$ step size;

$\rho_1, \rho_2 \leftarrow$ exponential decay rates;

$\theta \leftarrow$ initial model parameters;

$\boldsymbol{s}, \boldsymbol{r} \leftarrow$ 1st and 2nd moment variables initialized to $\mathbf{0}$;

$t \leftarrow$ time step initialized to 0;

$\delta \leftarrow$ small stabilization factor;

while *stopping criterion not met* **do**

Sample minibatch $\boldsymbol{x}$ of size m from training set $\boldsymbol{X}$;

Calculate loss function: $L(\boldsymbol{x}; \boldsymbol{\theta})$;

Calculate gradient of loss function: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}; \boldsymbol{\theta})$;

Increment time step: $t \leftarrow t + 1$;

Update biased first moment: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$;

Update biased second moment: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2 \boldsymbol{g} \odot \boldsymbol{g}$;

Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1{}^t}$;

Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2{}^t}$;

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\varepsilon \frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$;

Update model parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$;

end

2.4.4 *Tools and Frameworks*

Implementing all of the concepts we've been describing is time-consuming and arguably beyond the skill set of most deep learning practitioners.

In this sense, many machine learning and deep learning libraries have been created. Most of the algorithms (when available) in these libraries usually comply with the scientific work supporting them.

The choice of tools for deep learning should be constrained by what functionalities or algorithms the practitioner needs. Although some frameworks and libraries provide a wide variety of tools, we should also consider how lively the community around a certain tool is, as well as other factors such as supported language and hardware (GPU) support. Most deep learning libraries provide a Python API and are written in a lower-level language such as C/C++.

Tool	Type	Language Support	Advantages
Tensorflow (Abadi et al. (2015))	Numerical Computation Framework	Python, C++, JavaScript, Java, Go, Swift	Extremely popular and actively maintained; Provides a rich set of functionalities for model deployment
CNTK (Seide and Agarwal (2016))	Deep Learning Framework	Python, C++, BrainScript	Supports ONNX (neural network file format shared by several frameworks); Faster than the other frameworks supported by Keras when working with RNN/LSTM
PyTorch (Paszke et al. (2019))	Deep Learning Framework	Python	Dynamic Computational graphs; Support for ONNX; Growing popularity for research
MXNet Chen et al. (2015)	Deep Learning Framework	Python, C++, Julia, MatLab, Go, R, Scala, Perl	Vast language support; Supports ONNX; Good computational scalability
Keras (Chollet et al. (2015))	High-Level Deep Learning API (providing bindings for TensorFlow, Theano, MXNet or CNTK)	Python	Extremely popular and actively maintained; Consistent, clean and simple to use API allows fast development of DL models

Table 2.4.1: An overview of the most popular frameworks for deep learning. All of these tools provide GPU support.

Nguyen et al. (2019) compare different deep learning libraries, showing that Tensorflow is the most popular deep learning framework (across several GitHub statistics such as number of contributors and commits), followed by Pytorch.

The main differences between these two frameworks are their approach to the implementation of computational graphs. In most situations, this difference won't cause any discrepancy in results.

In this work, we will use TensorFlow along with its Keras API because we're most familiar with those tools, which are the industry standard for deployment of deep learning models.

2.5 SEQUENCE MODELING WITH DEEP LEARNING

In the last section, we explored a subfield of machine learning distinguished by the **structure of representations**. In this section, we explore another factor: the **structure of data**.

The structure of the analyzed data often has implications on several parts of a machine learning algorithm, such as the model topology, the objective function, or the regularization methods used. In this sense, we can group machine learning tasks by their type of data. Tasks using data with similar structures often use the same similar models and require similar methodologies.

For instance, computer vision problems are currently most frequently approached by using CNNs. The success of CNNs can be attributed to these models' strong prior beliefs regarding the extraction of features in images represented by a two-dimensional array of features. Another example is the processing of time-series, with the use of recurrent neural networks, among other models capable of capturing sequential dependencies.

This section reviews concepts regarding the analysis of sequential data with deep learning, the main focus of this work.

2.5.1 *Sequential Data*

Sequential data is structured in a way that expresses ordinal dependence between observations.

For example, let's assume we have a certain system that we need to classify as being in a normal or abnormal state. Additionally, assume we have two consecutive measurements of the system state: $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$.

In the non-sequential approach, we apply a model f for both states, and our answers are given by $y_1 = f(\boldsymbol{x_1})$ and $y_2 = f(\boldsymbol{x_2})$.

The sequential approach consists of building a model g that takes the array $\boldsymbol{X} = \{\boldsymbol{x_1}, \boldsymbol{x_2}\}$ as input, so that our answer is $\boldsymbol{y} = g(\boldsymbol{X})$. There are several options for the shape of $\boldsymbol{y}$. g can give a single answer for the whole sequence or a sequence of answers, for example.

Contrarily to f, g can retrieve information from the **transition** between states, which sometimes is a great advantage. Furthermore, maximum likelihood estimation makes the **i.i.d.** assumption (independent and identically distributed examples), which might not hold when a system's state influences future states and invalidates the non-sequential approach.

It is important to note that sequences are not always related to time. For instance, sequential data is widely used in bioinformatics to encode molecular structures such as proteins. Natural Language Processing (NLP) also makes heavy use of sequential data to represent sentences, which can be seen as sequences of words.

Sequential data that express temporal order are called time series. Models used for time series and other types of sequential data don't usually differ, since the information lies upon the concept of order and not time itself.

As we've discussed in section 2.1, human behavior analysis can benefit a great deal from the introduction of temporal context. In this sense, this work will focus on time series.

A time series is a sequence of τ timesteps. For example, for a length of $\tau = 3$, we can have a sequence $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3\}$. Each timestep $\boldsymbol{x}_t$ is indexed by its order in the sequence.

$\boldsymbol{x}_t$ is an array of real values. When we have non-numeric data, it is necessary to perform some sort of encoding so that $\boldsymbol{x}_t \in \mathbb{R}^M$, where M is the number of features in every timestep.

2.5.2 *Models For Sequential Data*

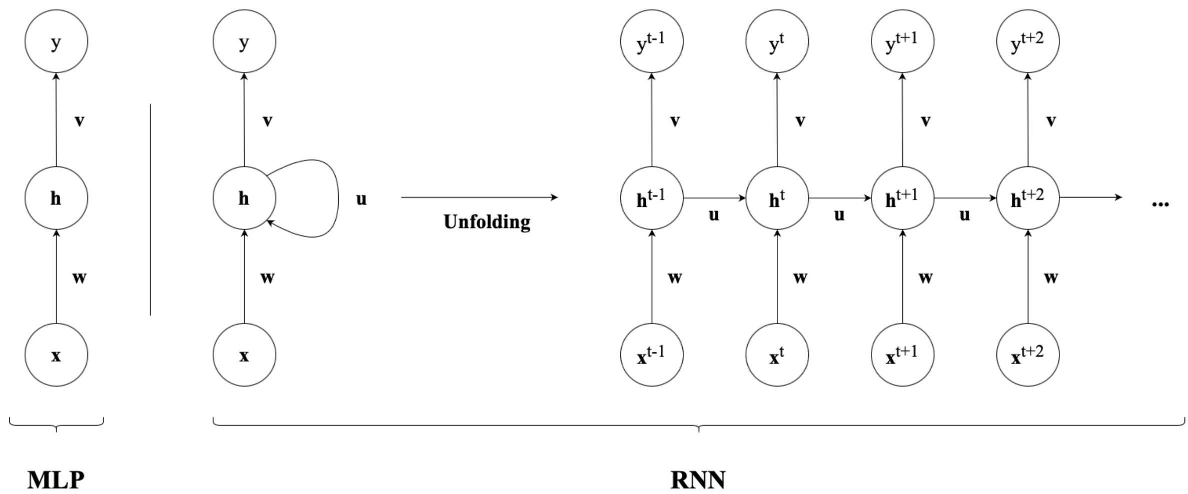
Before the advent of deep learning, machine learning was already applied to sequential data in many domains (Dietterich (2002)). Some of the most popular methods involved Hidden Markov Models (HMMs) or the use of a sliding window alongside a non-sequential model. Recurrent neural networks were also already known, but their computational cost was an impairment to the existence of the deep architectures we know today.

In this section, we explore some of the most popular deep learning methods for handling sequential data.

2.5.2.1 *Recurrent Neural Networks*

We've presented in section 2.4.1 what arguably is the most basic structure for a neural network. In an MLP, a given unit only receives input from units in the previous layer and only outputs to units in the next layer.

In recurrent neural networks (RNNs), a unit can be connected to itself. The representation of the transition from a timestep to the next relies on the self-connections in units.



$$\boldsymbol{h}^t = f(\boldsymbol{w}\boldsymbol{x}^t + \boldsymbol{u}\boldsymbol{h}^{t-1} + \boldsymbol{b}) \tag{2.5.1}$$

Figure 2.5.1: Comparison between a basic RNN and the MLP seen in section 2.4.1. Notice that the RNN features an additional set of parameters $\boldsymbol{u}$ which creates a connection between hidden units that should represent the state in consecutive time steps. All the parameters are shared through time. Equation (2.5.1) expresses the computation of a hidden unit in the RNN (analogous to equation (2.4.2) for the MLP). We can see that this recurrent neural network returns output for every timestep, but that is not always the case.

Self-connections are implemented through an operation called unfolding. Unfolding means extending the computational graph by repeating the self-connection for the number of timesteps in the sequence. Depending on the number of timesteps, the unfolding might produce extremely deep graphs, which means RNNs can be quite computationally expensive.

After unfolding, the computational graph of a recurrent neural network is similar to that of any other non-recurrent neural network. This means we need few to no adjustments in the algorithms that calculate gradients or perform parameter updates.

Although we might initially think of self-connections as only a way to introduce the past context, we can also introduce future context. Self-connections can be oriented from the future to the past, which has proven to be more effective in text-to-speech tasks (Bakiri and Dietterich (1999)), as in many other non-time-related tasks. Bidirectional recurrent neural networks are RNNs that include sets of parameters for recurrent connections in both ways.

As we recall from section 2.4.1, a connection between two units involves multiplying the input by a parameter, meaning that in RNNs we often have the same parameter multiplied by itself several times. Some of the main problems in traditional RNNs arise from this successive multiplication:

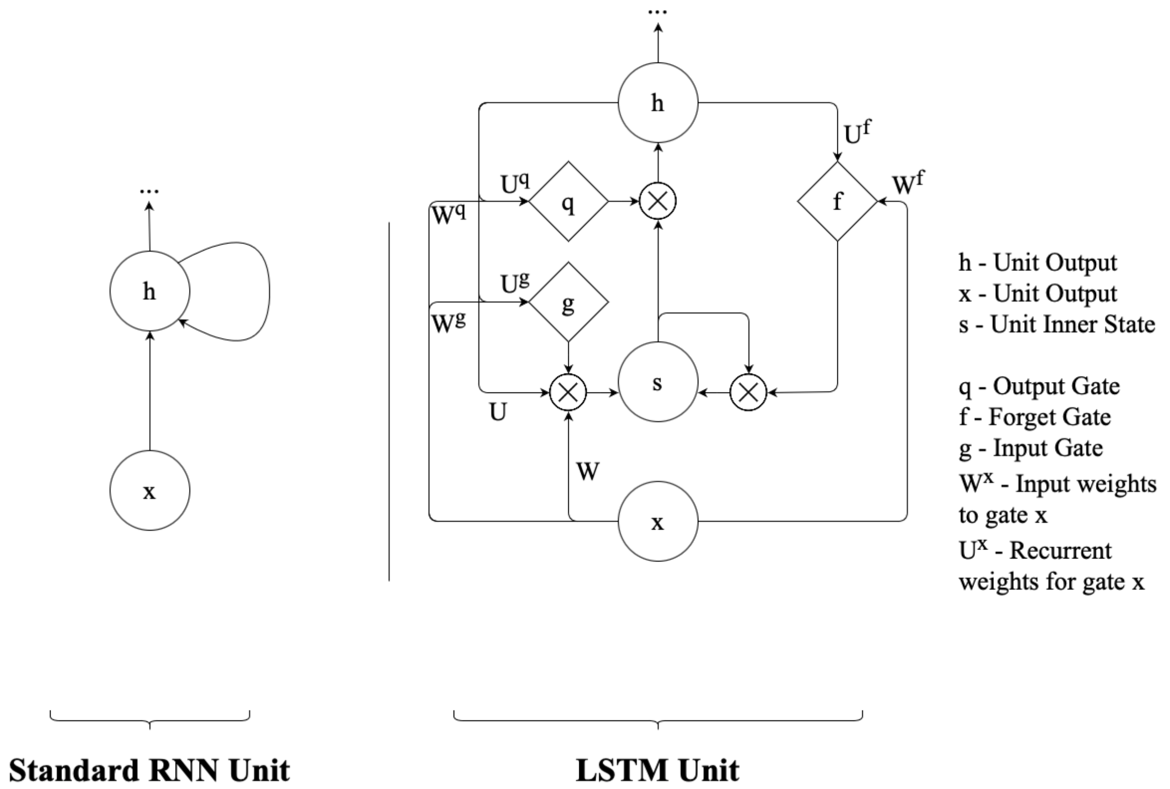
- Exploding gradients - when the multiplying parameter has a value much larger than 1, it's repetitive multiplication can result in a numerical overflow
- Vanishing gradients - the same as an exploding gradient, but when a parameter much smaller than 1 causes an underflow
- High nonlinearity - some activation functions, when repeatedly applied in successive timesteps, are completely deformed and lose some of their properties. Such is the example of the hyperbolic tangent

These obstacles make it difficult to operate on large sequences and to learn long-term dependencies in data (Bengio et al. (1994)). Some more sophisticated RNNs address these issues, such those based in gated recurrent units, which are some of the most successful models today.

LONG SHORT-TERM MEMORY Hochreiter and Schmidhuber (1997) introduced the Long Short-Term Memory (LSTM) with the idea of creating paths through which the gradient neither explodes nor vanishes.

From an outer perspective, an LSTM unit behaves as the basic recurrent unit described in section 2.5.2.1. Its inner state, however, is controlled by three gates defined by separate parameters. These gates are:

- Input gate - Controls the weight of each input to the unit, similar to what we already find in regular RNNs.
- Output Gate - Controls how much of the unit's inner state is allowed to pass on to the output.
- Forget Gate - Inserted in a self-loop inside the unit, controls how much of the unit's inner state passes on to the next inner state.



$$h_i^t = tanh(s_i^t)q_i^t \tag{2.5.2}$$

$$s_i^t = f_i^t s_i^{t-1} + g_i^t \sigma\left(b_i + \sum_j W_{i,j} x_j^t + \sum_j U_{i,j} h_j^{t-1}\right) \tag{2.5.3}$$

$$q_i^t = \sigma\left(b_i^q + \sum_j W_{i,j}^q x_j^t + \sum_j U_{i,j}^q h_j^{t-1}\right) \tag{2.5.4}$$

$$f_i^t = \sigma\left(b_i^f + \sum_j W_{i,j}^f x_j^t + \sum_j U_{i,j}^f h_j^{t-1}\right) \tag{2.5.5}$$

$$g_i^t = \sigma\left(b_i^g + \sum_j W_{i,j}^g x_j^t + \sum_j U_{i,j}^g h_j^{t-1}\right) \tag{2.5.6}$$

Figure 2.5.2: Unlike in a regular RNN, the flow of information in an LSTM unit is controlled by these gates. Every gate is updated depending on the last hidden state $\boldsymbol{h}^{t-1}$ and the current input $\boldsymbol{x}^t$. There are separate $\boldsymbol{W}$ and $\boldsymbol{U}$ parameters for each gate and also the inner state.

OTHER GATED RECURRENT UNITS Some argue that some of the components of the LSTM unit are redundant and introduce unnecessary complexity in models. In this sense, less complex approaches, still relying on the concept of gates, were created.

Cho et al. (2014) proposed a unit architecture that only has two gates, known as Gated Recurrent Unit (GRU). The main difference between GRU and LSTM is that one of the gates simultaneously performs functions similar to the input and forget gates of LSTM.

LSTM and GRU remain the most popular gated units for recurrent neural networks, although it is possible to come up with many more unit architectures. Jozefowicz et al. (2015) conducted an extensive study on different gated unit architectures, with the LSTM and GRU as starting points. They found that although the GRU outperforms the LSTM in most tasks, the performance gap can be closed with proper initialization of the forget gate bias parameters in the LSTM. The best-performing architectures in the experiment were similar to the GRU.

In this sense, to choose the unit architecture in a recurrent neural network, one should experiment with several possibilities for each different task.

2.5.2.2 *Convolutional Neural Networks*

Convolutional neural networks (CNNs) are neural networks that employ the convolution operation instead of general matrix multiplication (in at least one of its layers). The convolution is a linear operation on two functions that operate on the same domain. It consists of multiplying the values of a function f in an infinitely wide vincinity of a point t by a function g that takes the distance of any given point to t.

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)dt \tag{2.5.7}$$

The convolution operation has properties that are desirable to machine learning. Unlike in dense layers in MLPs, the convolutional layers in a CNN guarantee that certain features only interact with a limited number of features in their neighborhood. Since the same function g can be applied to any point in the domain of f, we can express the same interaction occurring in different parts of the domain while using the same parameters (recall parameter sharing as a means to regularization in section 2.4.2.2).

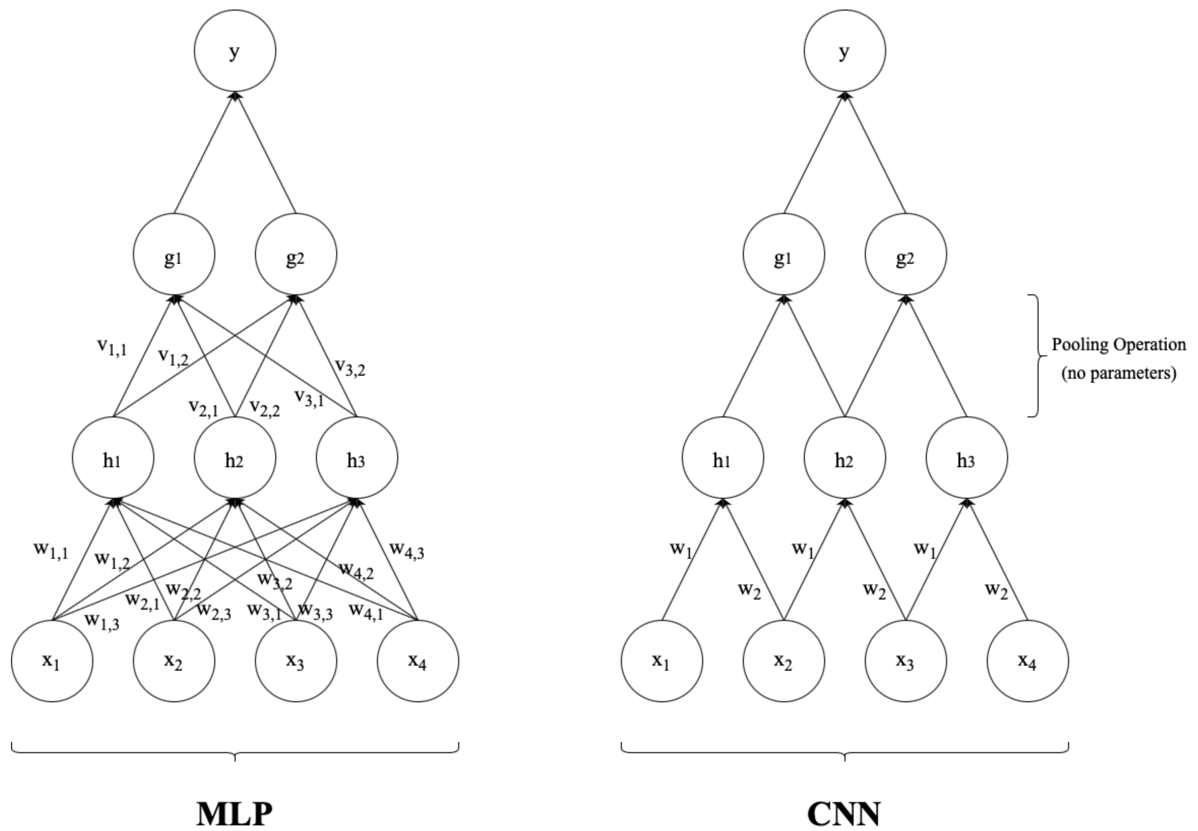


Figure 2.5.3: On the left, we have a simple MLP with two fully connected hidden layers. Notice that a feature interacts with every feature in its layer. On the right, we have a simple CNN with a convolutional layer (only one kernel is illustrated) and a max-pooling layer. The parameters of a convolution kernel are shared across the whole input, so a given kernel learns a way in which a feature interacts with its neighbor features.

Another operation frequently used in CNNs is pooling, which consists of summarizing a group of close features with a given function. Perhaps one of the most popular types of pooling is max pooling, which returns the maximum value in the group of features to be summarized.

The convolution and pooling operations are infinitely strong priors forcing input features to only interact with features in their vicinity. These priors work particularly well in computer vision, hence it's extensive use with images.

Because those assumptions regarding proximity also hold true in a temporal perspective (events closer in time are most likely more strongly correlated), CNNs can also be used with sequential data.

To better understand how CNNs work, one needs to be familiar with the concepts of:

- **Filter** (or convolution kernel) - the matrix of weights that describe the convolution performed through the whole data (e.g. across an entire picture, or a time series). Analog to the convolution operation

described by Equation 2.5.7. In a single convolutional layer, we can have many filters, which means several convolutions performed on the same data.

- **Filter size** - the size of the rolling window of features that directly interact to yield the result of the convolution. Analog to the τ in Equation 2.5.7.

2.5.3 *Sequence Modeling Tasks*

Sequence modeling is a broad term through which we can refer to the extraction of information from sequential data. Numerous tasks can fit into this description, and with new machine learning methods, the variety of uses one can find for sequential data is increasing. A complete review of the universe of problems related to sequential data is well beyond the scope of this work. We will focus on two main areas: sequence labeling and sequential outlier detection.

2.5.3.1 *Sequence Labeling*

Sequence labeling is the task of establishing a correspondence between a sequence of labels and a sequence of input data (Graves (2012)). This is a broad definition because it doesn't constrain the shape or alignment of input or output sequences.

In this sense, there are progressively stricter concepts that impose increasingly strong prior assumptions regarding output shape.

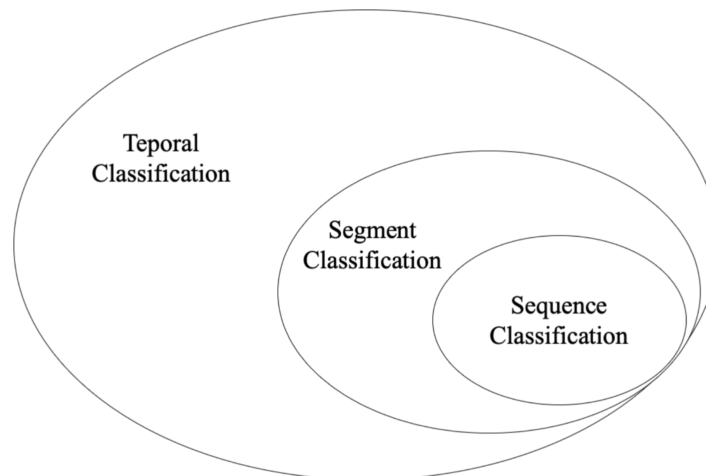


Figure 2.5.4: The groups of sequence labeling tasks, according to John. Temporal classification is the looser concept, where we do not have any constraints for the input and output sequence alignments (for example, translating a sentence). Segment classification is a subcase of the previous, where the input/output alignment is known in advance. Sequence classification is the stricter case, where we classify a whole sequence with only one label.

2.5.3.2 *Sequential Anomaly Detection*

Anomaly Detection consists of identifying observations that do not conform with the normal expectations in a certain domain. From a statistical perspective, anomalies (often referred to as outliers) seem to arise from a probabilistic distribution different from the one known or inferred from past observations.

Anomalies are generally classified as either contextual (or conditional), collective, or point anomalies. Point anomalies simply do not fit in the context of previously observed cases, while contextual anomalies might appear normal in the absence of context. Collective anomalies, on the other hand, seem irregular when occurring in a group. Although the implementation of a point anomaly detection system might appear more simple, the introduction of context (even if it is a more complex solution) might benefit the success at identifying outliers (Hayes and Capretz (2014)).

This context might be some order (sequential data), space (images or 3D models, for example), or any variable or structure that contributes to the ability to build a representation where the outliers stand out.

One of the main challenges in anomaly detection is the difficulty in collecting labeled data from outliers. The main reason for this is that these events are rare.

In this sense, some of the most successful approaches to anomaly detection fall onto unsupervised anomaly detection.

Unsupervised anomaly detection usually involves modeling the domain data distribution and then evaluating the behavior of the model. The model should perform consistently with normal data and erratically when exposed to outliers. This representational model is usually trained for tasks in which we can leverage input data as its label. Two examples are forecasting (the last element in a sequence is the label) and input reconstruction (as seen in autoencoders).

Sequential anomaly detection can be seen as a binary sequence labeling problem because it generally consists of producing output that labels data as normal or anomalous.

2.5.4 *Work in Sequential Data Analysis*

In this section, we present a (by no means systematic) review of work done in sequential data analysis to attain a satisfactory understanding of the methods currently used in this field. Our review prioritized work related to anomaly detection since it is related to the fraud-detection challenge we seek to solve and usually represents a greater challenge. The model architectures used in anomaly detection are often easily applicable to other sequence labeling tasks.

Work	Task	Paradigm	Methods	Data Type	Obs.
Pereira and Silveira (2019)	Anomaly Detection in electrocardiogram (ECG) data	Unsupervised	LSTM	Univariate Time Series	Variational Recurrent Autoencoder, Bidirectional LSTM Layers, Sequence Classification, Biometric Sensors
Rajpurkar et al. (2017)	Anomaly Detection in ECG data	Supervised	CNN	Univariate Time Series	Sequence Classification, Biometric Sensors
Chauhan and Vig (2015)	Anomaly Detection in ECG data	Unsupervised	LSTM	Univariate Time Series	Sequence Classification, Biometric Sensors
Malhotra et al. (2017)	General Sequence to Sequence encoding	Unsupervised	GRU	Univariate Time Series	Autoencoder, Fixed-length sequence encoding
Karim et al. (2018)	General Time Series Classification	Unsupervised	CNN + LSTM	Univariate Time Series	Sequence Classification
Siegel (2020)	Anomaly Detection in Industrial Sensor Networks	Unsupervised	RNN, CNN	Multivariate Time Series	Sequence Classification, Sensor Data
Cherdo et al. (2020)	General Unsupervised Anomaly Detection in Time Series	Unsupervised	LSTM	Univariate Time Series	Spectral Analysis, Fourier Transform Preprocessing, Sequence Classification
Lu et al. (2017)	General Unsupervised Anomaly Detection in Time Series	Unsupervised	RNN	Multivariate Time Series	Autoencoder, Sequence Classification
Lin et al. (2020)	General Unsupervised Anomaly Detection in Time Series	Unsupervised	LSTM	Multivariate Time Series	Variational Autoencoder, Segment Classification
Munir et al. (2019)	General Unsupervised Anomaly Detection in Time Series	Unsupervised	CNN	Multivariate Time Series	Prediction-based Anomaly Detection, Segment Classification

Table 2.5.1: This table presents work done in sequence labeling, mostly related to anomaly detection. We can notice the prevalence of unsupervised learning, due to the difficulty of gathering labeled data for large volumes of data. Unsupervised methods are extensively used because they allow us to harness the information in of data that is straight-forward to collect and store but would be extremely hard to label.

2.6 MACHINE LEARNING IN VIDEOGAMES

As seen in section 2.1.2, we can see video games as a particular case of HCI. In section 2.2.2, we discussed how machine learning plays a major role in modern AmI solutions.

In this section, we review the use of machine learning in video games. Although our main interest is deep learning as a reasoning layer for data collected through the sensorization of human-computer interaction, we provide a broader perspective of the use-cases for ML in video games.

Work	Type of Model(s) Used	Data Type	Paradigm	Use Case Description	Observation
Islam et al. (2020)	Novel Kernel Machine Variant (Gaussian Kernel)	Tabular Data	Unsupervised	Cheat detection using pattern recognition in network traffic data	Tests ran on data collected from a small sample, might not be representative of the population
Galli et al. (2011)	Decision Trees; Random Forest; Neural Networks; SVM	Tabular Data	Supervised	Cheat detection using in-game contextual data	Extremely small test dataset. Rudimentary reporting of results.
Alayed et al. (2013)	SVM; Logistic regression	Tabular Data	Supervised	Cheat detection using domain knowledge and in-game contextual data	Extremely small test dataset. Only three players participated in data collection.
Pao et al. (2010)	SVM variant; K-Nearest Neighbors	Tabular Data	Supervised	Detection of automated gameplay (bots) using in-game data	Extremely small test dataset. Structure of used data is unclear.
Pluskal and Sedivý (2014)	Random Forest	Tabular Data	Supervised	Recommendation system for game item microtransactions using in-game data	-

Yeung et al. (2006)	Dynamic Bayesian Networks	Time Series	Supervised	Cheat detection using in-game data	Reduced dataset (10 matches and only 6 players). Missing quantitative evaluation.
Alkhalifa (2016)	HMMs	Time Series	Supervised	Cheat detection using in-game data	Reduced dataset. Missing quantitative evaluation.
Willman (2020)	LSTM	Time Series	Supervised	Cheat detection using in-game data	Reduced and unbalanced dataset. As pointed out by author, low accuracy of the models.
Jain et al. (2016)	Autoencoders	2-D Bitmap	Unsupervised	Content generation using generative models (recognizing, generating, or repairing game levels)	-
Chen et al. (2018)	CNN	Sequential Data	Supervised	Predicting player lifetime spendings in game micro-transactions	-
Partlan et al. (2019)	LSTM	Sequential Data	Supervised	Imitating player behavior using in-game data to automate QA testing	-

Table 2.6.1: This table is representative of the current landscape of machine learning in video games. Cheat detection is one of the most sought-after use cases, and in-game data remains the most popular source of data.

From our review of machine learning in video games, summarized in Table 2.6.1, we can observe that:

- There is an extreme difficulty in gathering labeled data for research. This makes a good case for the usage of unsupervised methods (and still, most work does not tackle unsupervised learning).

- Most work doesn't reach meaningful conclusions or runs experiments on an extremely small test data set. This problem can be solved if the research results from a collaborative process between researchers and the video game industry, as is our case.
- Most of the experiements to evaluate proposed machine learning systems consist of collecting data from a very limited sample of players during a few matches of a single game. We plan to validate our approaches across several games and hundreds or even thousands of players.
- There is almost no scientific work applying deep learning to video games. Nearly all approaches focus on simple models and simple data structures.
- Previous approaches only consider in-game data. In this sense, our work is a novel approach to applying deep learning to video games (because it relies only on human-computer interaction data).

3

PROPOSED APPROACH

In this section, we present our general approach for a data-driven system dealing with HCI. Firstly, we introduce our data collection method. Secondly, we proceed to describe a data pipeline supporting multiple approaches with varying use of domain knowledge. Then, we provide details on our deep learning modeling approach, and finally, we describe a possible deployment method for the resulting models.

The main features of our approach can be summarized as follows:

- Our data collection process is agnostic to the platform or the peripherals being used, provided that we can collect the stream of events corresponding to the user's actions;
- Since we use deep neural networks, our input and output data can take many shapes, which contributes to the two following properties:
  - We can apply various sets of features, sources of behavioral events, and time granularity degrees the input data for our solutions;
  - We can also formulate in various ways the answers to the problems we're trying to solve, this being able to use this approach for several problem types (e.g. classification, forecasting, segmentation, or even generative applications);
- Our approach inclines towards modeling with close-to-raw data, this not requiring great efforts in manual feature engineering. This also means we rely less on domain knowledge (commonly critical to inform feature engineering);
- The main architectural features of the models resulting from our approach are faced as hyperparameters and subject to an automatic optimization process. This way, the architecture of the models is automatically optimized for each case study where we apply this framework (instead of arbitrarily picked and tested based on the practitioner's experience).

3.1 DATA COLLECTION

The success of a data project is often more impacted by the quality of data rather than by how sophisticated the modeling process actually is. In this sense, it is important to devise a data collection method that is accurate and effective.

Humans can interact with a computer in many ways (especially now that technologies such as voice recognition are starting to work well), but the most common is through peripherals such as keyboards, mouses, and trackpads.

We start by defining our most basic form of interaction, which is an event E. An event occurs at a given time t, possesses a code c drawn from an alphabet $\boldsymbol{A}$ and a real value v which describes the state or intensity for that event. The set of possible values $\boldsymbol{v}_c$ for the event is a function of c, since different types of events can be described with a different set of values.

Mean of Interaction	Alphabet	Value Range
Keyboard	$\boldsymbol{A}_{keyboard} = \{Q, W, E, R, T, Y, ...\}$	$\boldsymbol{V}_{keyboard} = \{\boldsymbol{v}_Q, \boldsymbol{v}_W, \boldsymbol{v}_E, \boldsymbol{v}_R, \boldsymbol{v}_T, \boldsymbol{v}_Y, ...\} = \{\{0, 1\}\}$ (up or down)
Mouse	$\boldsymbol{A}_{mouse} = \{R, L, M_WHEEL, CURSOR_X, CURSOR_Y\}$	$\boldsymbol{V}_{mouse} = \{\boldsymbol{v}_R, \boldsymbol{v}_L, \boldsymbol{v}_{M_WHEEL}, \boldsymbol{v}_{CURSOR_X}, \boldsymbol{v}_{CURSOR_Y}, ...\} = \{\{0, 1\}, \mathbb{R}\}$ (up or down for buttons, rotation angle of scroll wheel, cursor position)

Table 3.1.1: Example of the event alphabet and value ranges for the specific case of mouse and keyboard. When we have a small set of values for a given event type, we can also divide it into dedicated codes for each value (for example K_down and K_up when $\boldsymbol{v}_K = \{0, 1\}$).

Timestamp (milliseconds)	Code	Value	Event Description
1574365839854	cursorX	960	The cursor's horizontal location
1574365839854	D	1	The D keyboard key was pressed
1574365839854	cursorY	540	The cursor's vertical location
1574365839854	MOUSE_LEFT	1	The left mouse button was pressed
1574365840188	A	1	The A keyboard key was pressed
1574365840196	D	0	The D keyboard key was released
1574365840392	cursorX	960	The cursor's horizontal location
1574365840392	cursorY	540	The cursor's vertical location
1574365840392	MOUSE_LEFT	0	The left mouse button was released
...	...	...	...

Table 3.1.2: A possible sequence of keyboard and mouse events. This table contains roughly 500 milliseconds of a real interaction from our data.

Table 3.2.1: Multivariate time series of the events in Table 3.1.2.

Timestamp	A	D	MOUSE_LEFT	cursorX_var	cursorY_var
1574365839854	0	1	1	0	0
1574365839954	0	1	1	0	0
1574365840054	0	1	1	0	0
1574365840154	0.66	0.42	1	0	0
1574365840254	1	0	1	0	0
1574365840354	1	0	0.38	0	0
...	...	...	...	...	...

The first stage of our data pipeline consists of capturing and storing these events. Especially in the case of events that require a fine-grained description (mouse movement, for example), an immense volume of data is generated. In this sense, we need to use techniques to reduce this volume and store data efficiently, discussed in the following section.

3.2 DATA PROCESSING PIPELINE

Although we use raw event data with minor preprocessing, we still need to produce regular structures that serve as input to the deep neural networks. Table 3.1.2 illustrates the information we collect for each event. Each event is characterized by three attributes: a timestamp, a code, and a value.

Once we have the collection of events described above, we process them to obtain a data structure as seen in the sample in Table 3.2.1. Each column represents an event code and each row represents a timestep (each timestep corresponds to 100 milliseconds).

We aggregate events in timesteps according to the value range of each event code.

For binary events (such as pressing or releasing keys), the value of each timestep is the amount of time between values 1 and 0 and the total duration of the timestep given by the formula

$$x_t = \frac{\sum_n \left[\min(t+T,\ e_{0_n}) - \max(t,\ e_{1_n})\right]}{T}, \quad \forall\, n\ :\ t < e_{0_n} < t+T \ \vee\ t < e_{1_n} < t+T \tag{3.2.1}$$

where e_{0_n} and e_{1_n} are the timestamps of the nth events with the keycode of the column being calculated and values 0 and 1 respectively. T is the duration of a timestep and t is the beginning of the timestep.

For real-valued columns, we follow a different approach. For each possible event code, we generate two features expressing the variance and the amplitude of the values in each timestep. Following the same language as before, the two columns can be defined with the formulae

$$\begin{cases} x_{t_{var}} = \sigma\left(\frac{\sum_{n=1}^{N}(e_{n_{value}} - \overline{e_{values}})^2}{N-1}\right) \\ x_{t_{amp}} = \sigma\left(\max_n e_{n_{value}} - \min_n e_{n_{value}}\right) \end{cases}, \quad \forall\, n\ :\ t < e_n < t+T \tag{3.2.2}$$

where $e_{n_{value}}$ is the value of the nth event with the keycode of the column being calculated. The function σ serves the purpose of keeping the values of these columns in the same range as the binary coluns, and is defined by the formula

$$\sigma :\ \ ]-\infty,\ +\infty[\ \rightarrow\ ]-1,\ 1[\qquad \sigma(x) = \frac{2}{1+\exp(-x)} - 1 \tag{3.2.3}$$

We can visualize the transformation applied by σ in Figures 3.2.1 (before) and 3.2.2 (after). The mouse movement features we used in this work assume values in $[0,\ +\infty[$. When we apply the σ function, these features are squashed into the interval $[0,\ 1[$, which is nearly same domain seen in the features resulting from binary events.

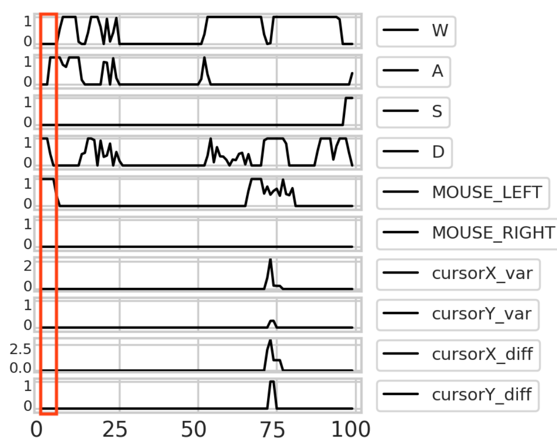


Figure 3.2.1: Example of 10 seconds of interaction between the player and the computer. The red rectangle highlights the data sample seen in Tables 3.1.2 and 3.2.1.

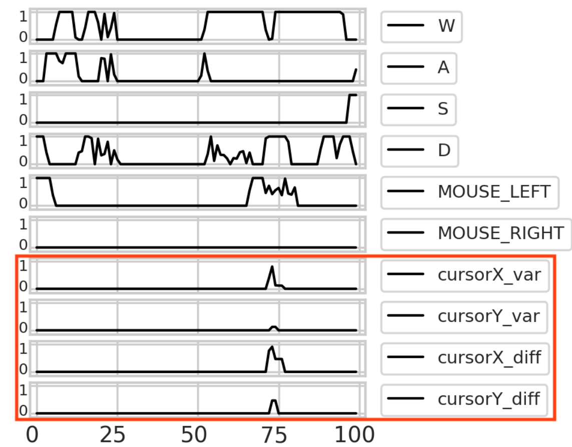


Figure 3.2.2: The same data seen in Figure 3.2.1, after applying the function σ to the features outlined by the red rectangle.

3.3 MODELING

Modeling is the central part of our work. We seek to find models with meaningful representations of the human-computer interaction.

We can place our modeling approach along two axes:

- The degree of feature engineering (using raw data or domain knowledge).
- The existence of temporal connectivity (if we consider only an isolated time window or a sequence).

		Temporal Connectivity	
		Single Time Window	Time Series
Feature Engineering	Raw Data	(1) Using an array of raw features for an isolated time window. This approach is not viable because without temporal connectivity, raw features or event occurrences have no meaning.	(2) Using time series of raw features. In this approach, we rely on sophisticated models to extract sequential dependencies in the multivariate stream of events.
	Domain Knowledge	(3) Using domain knowledge to condense the stream of events in a single array of hand-engineered features. This approach is most commonly used with traditional machine learning models and generally consists of calculating statistical aggregations of the event stream.	(4) Using domain knowledge to produce hand-engineered features while still taking advantage of temporal connectivity between the intervals for which the bioml etric aggregations are calculated.

Table 3.3.1: This table summarizes the approaches we can take regarding the data described in the previous section. Our work focuses on approach (2). We should note that it is also possible to combine these approaches. For example, we might want to concatenate an array of raw data and hand-engineered features, thus combining (3) and (4).

As seen in table 3.3.1, we can take several approaches to analyze HCI data. Using time-series demands more complex models, capable of capturing temporal dependencies, as seen in section 2.5.2. The downside of using time-series is that it requires storing and processing much larger volumes of data.

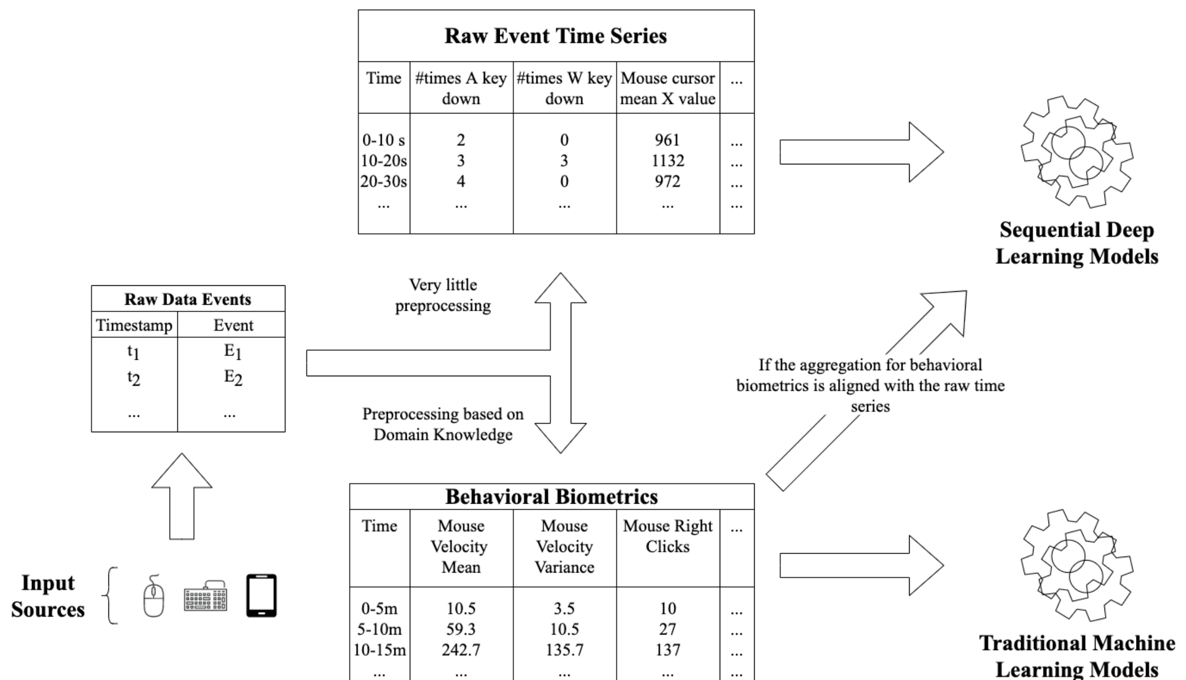


Figure 3.3.1: As explained in table 3.3.1, we can take several modeling approaches. This diagram illustrates the data pipeline from the moment we record the events. The less preprocessing we apply to data, the more complex models we need to automatically learn features. The diagonal arrow represents the possibility of joining approaches (2) and (4) from table 3.3.1.

We only explore the approach of using raw data instead of domain knowledge. That decision was to allow the deep learning models to automatically extract the features (instead of manually engineering them). In this sense, we can apply the same algorithms to different domains where different features are created.

3.3.1 *General Model Architecture*

As discussed in section 2.5, Deep Learning allows us to take advantage of the sequential structure of time series. Although it is theoretically possible to process this data with a simple architecture such as an MLP, training those models or even creating them with high enough capacity to achieve good results would be intractable.

In this sense, one must resort to more complex neural network architectures such as RNNs or CNNs, which rely on assumptions that allow them to process complex data structures while not exploding in model complexity (and computational cost).

As reviewed in table 2.5.4, the two main contenders to be the cornerstone of our models' architectures are RNNs (especially gated unit architectures) and CNNs.

Table 3.3.2: RNNs vs. CNNs in our use-case

RNNs	CNNs
• Difficult to parallelize • Complexity (number of parameters) explodes with time series length and number of features • Work better with univariate time series • Struggle with irregular patterns (that do not repeat)	• Easily parallelizable • Complexity (number of parameters) does not depend on time series length nor number of features (but rather on the structure of the layers, i.e. number of kernels and kernel dimensions) • Perform well with high-dimensional data • Deals with irrelgular patterns

Gated unit architectures such as LSTM and GRU thrive in problems with univariate time series and regular patterns, as seen in Chauhan and Vig (2015). In more irregular and high-dimensional data, CNNs tend to be a better choice, as seen in Siegel (2020).

Additionally, CNNs are more parallelizable than RNNs because (according to the unfolding operation explained in section 2.5.2.1) many calculations need to be sequential.

Taking into account the comparison in Table 3.3.2, we opted to use convolutional neural networks.

Most deep learning frameworks provide similar terms and APIs to easily define different architectures. In this sense, part of our modeling pipeline is dedicated to choosing the architecture that yields the best results for a given dataset. Table 3.3.3 introduces the architectural features that vary in our hyperparameter optimization process.

Table 3.3.3: CNN architecture features varying in hyperparameter selection.

Characteristic	Type	Effect
Number of Layers	Architecural Characteristic	The number of convolutional layers in our deep neural network.
Number of Filters	Architecural Characteristic	The number of filters (or kernels) in each layer. This parameter could unique to each layer, so that different layers could have a different number of filters.
Filter Size	Architecural Characteristic	The size of each filter (i.e. the number of time steps that interact to originate a feature in the following layer). As in the filter number, this parameter could be unique to each layer, if we have the means to perform an hyperparameter search with that number of parameters.
Use of Pooling Layers	Architecural Characteristic	In our approach, the use of pooling layers is itself automatically determined by the pipeline. When pooling layers are used, every convolutional layer is followed by a max pooling layer.
L2 Regularization λ	Optimization Parameter	The λ parameter of the L2 regularization as described in Section 2.4.2. The higher the value, the more constrained the network weights will be.
Batch Size	Optimization Parameter	The size of every minibatch in the gradient descent algorithm variant used, as described in 2.4.3.3.

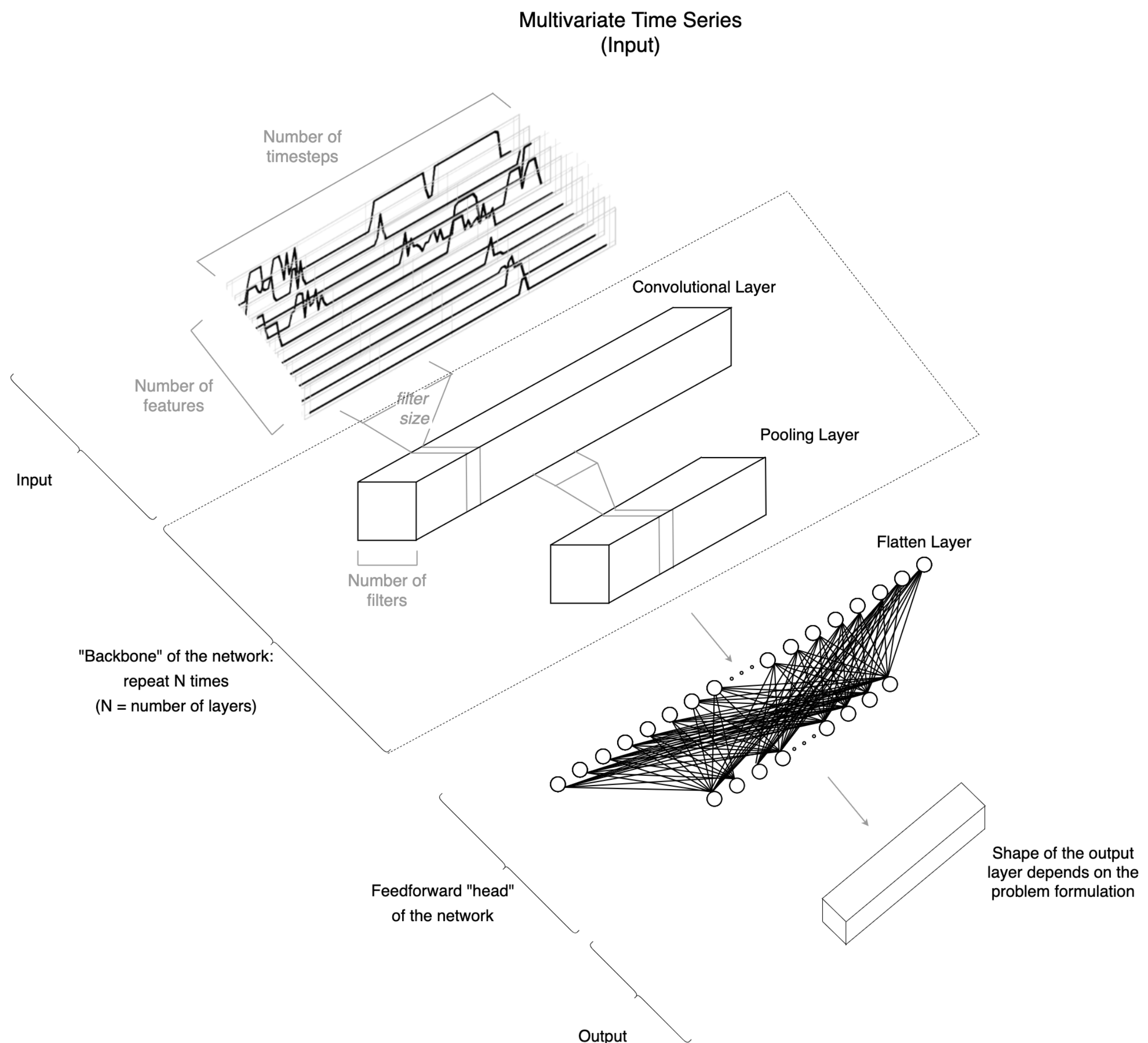


Figure 3.3.2: This is the generic architecture of the convolutional neural networks in our approach. We define the number of layers as the number of convolutional layers (that may be followed by pooling operations). The properties of each convolutional layer are also subject to optimization, and would ideally be independent between layers. The main focus of the architectural optimization pipeline is the network's backbone since the head of the network is highly dependant on the problem.

3.3.2 *Hyperparameter Tuning*

Hyperparameter optimization allows us to automatically find a good architecture for any given domain, especially considering that we treat some of the fundamental characteristics of our models (such as the number of layers) as hyperparameters.

In this sense, we ought to follow an efficient search heuristic or algorithm, so that we're able to experiment with enough configurations to reasonably cover our hyperparameter search space.

As it is well beyond the scope of our work to create a hyperparameter optimization method from scratch, we relied on the framework introduced in John. Optuna is a minimalistic framework that implements efficient hyperparameter search and pruning strategies, supporting dynamically defined search spaces and distributed computing.

In terms of the search strategy Optuna uses, there are two main components:

- Sampling - the strategy for choosing the next hyperparameter configuration to be tested.
- Pruning - the decision process to discard trials unlikely to yield good results, thus avoiding unnecessary computational cost.

Sampling methods usually fall into one of these two categories: relational and independent. While the first studies the correlation between different hyperparameters, the second bases the choice of the parameter independently from the others. Optuna features both types of methods, including sophisticated techniques such as covariance matrix adaptation Hansen and Ostermeier (2001) and bayesian optimization Shahriari et al. (2016).

Optuna's pruning algorithm is an extension (due to parallelization concerns) of the successive halving algorithm Jamieson and Talwalkar (2016), which iteratively discards the worst half of hyperparameter configurations in each training step.

3.3.3 *User-centered Cross-validation*

Cross-validation techniques provide a way of ensuring that our models are not overfitting to the training dataset.

Usually, methods such as K-fold cross-validation are simple to implement and effective at securing key assumptions such as:

- Train and validation examples are independent.
- Train and validation datasets follow similar probabilistic distributions.

In some domains, however, we ought to account for other factors that may determine how different data points relate to each other. In HCI, we should consider the possibility of a wide variety of behaviors and usage patterns. In this sense, a method to evaluate if our models remain accurate for different user groups (with varying habits and behaviors) is needed.

With this concern in mind, we developed a user-centric adaptation of the K-fold cross-validation method.

User-centered cross-validation not only tests our approach's ability to perform well in unseen data but most importantly of detecting cheating interactions in unknown players.

Algorithm 4: User-centered Cross-validation.

$\boldsymbol{X} \leftarrow$ dataset of multivariate time series
$\boldsymbol{y} \leftarrow$ labels for each example in the dataset
$\boldsymbol{players} \leftarrow$ list of users with records in $\boldsymbol{X}$
for *p in* ***users*** **do**
  $\boldsymbol{X}_u \leftarrow$ records of user u
  $\boldsymbol{y}_u \leftarrow$ labels for user u
  $\boldsymbol{X}_o \leftarrow$ records of other users
  $\boldsymbol{y}_o \leftarrow$ labels of other users
  $model \leftarrow$ new instance of proposed model
  Train$(model, \ \boldsymbol{X}_o, \boldsymbol{y}_o)$
  Evaluate$(model, \boldsymbol{X}_u, \boldsymbol{y}_u)$
end

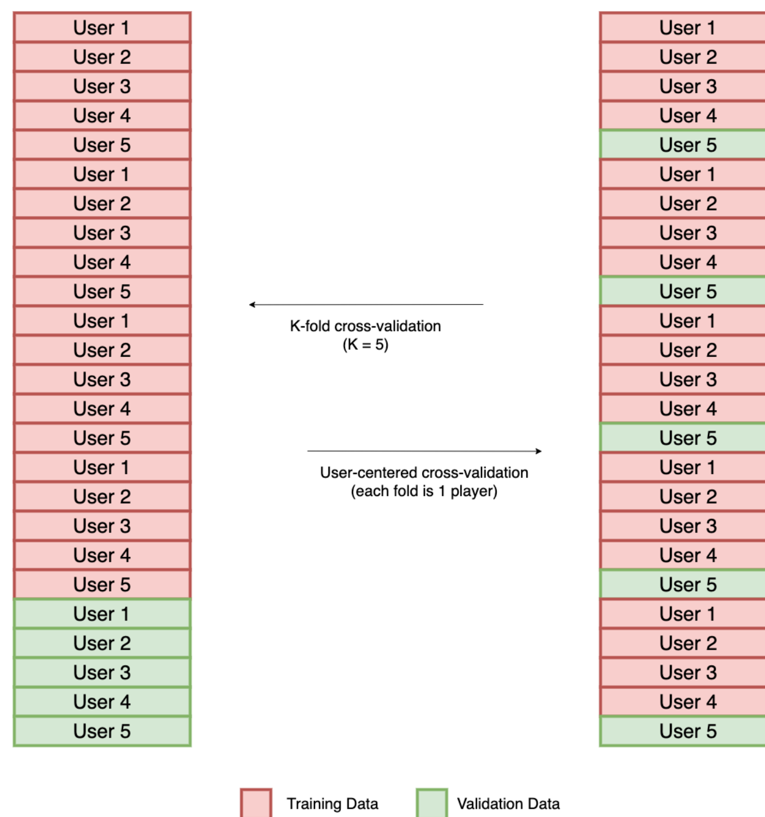


Figure 3.3.3: In this picture we can visually compare our proposed method for user-centered cross-validation with K-fold cross-validation. While in K-fold the data splits don't take into account the origin of the data examples, in user-centered cross-validation there is no leakage of data from a given user between train and validation data.

3.4 DEPLOYMENT

Although deployment isn't usually a major focus in scientific research in machine learning, we can not overlook it if we want to take advantage of the models resulting from the previously described approach.

As discussed in Section 2.4.4, we use TensorFlow to implement our deep learning models. This framework provides a straightforward method of deployment, which consists of saving models on any file system.

Since TensorFlow provides APIs in several languages, the deployment can be completely independent of the platform used to train and test the models.

The specific architecture used to serve the models can vary depending on the available infrastructure in each project. Figure 3.4.1 provides an overview of a possible training and deployment system.

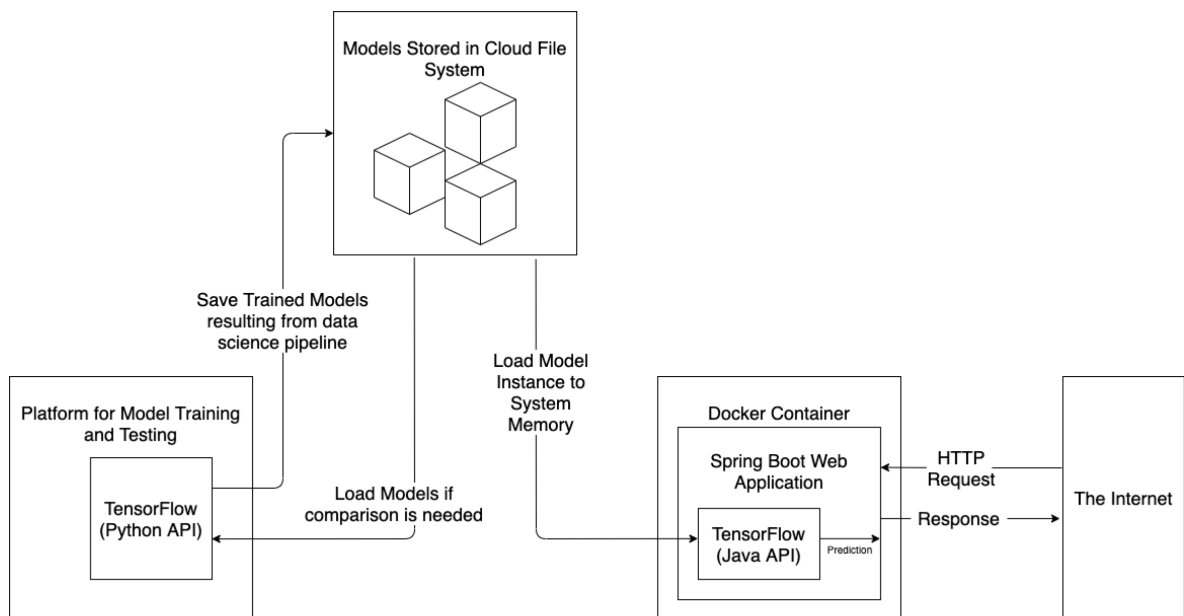


Figure 3.4.1: An example of how models resulting from the training pipeline can be served trough HTTP requests.

4

FRAUD DETECTION IN VIDEO GAMES

Online video games drive a multi-billion dollar industry dedicated to maintaining a competitive and enjoyable experience for players. Traditional cheat detection systems struggle when facing new exploits or sophisticated fraudsters. More advanced solutions based on machine learning are more adaptive but rely heavily on in-game data, which means that each game has to develop its own cheat detection system.

In this section, we present a novel approach to cheat detection that is based in the concepts discussed in Section 3 and that doesn't require in-game data. Our models achieve an average accuracy of respectively 99.2% and 98.9% in triggerbot and aimbot (two widespread cheats), in an experiment to validate the system's ability to detect cheating in players never seen before.

4.1 EXISTING SOLUTIONS AND REQUIREMENTS

Anti-cheating systems help to provide a better experience to the players. By keeping the players engaged in a competitive environment, the game communities can keep growing, providing entertainment, and generating profit.

The problem with traditional anti-cheating systems is that they have a history of always being one step behind the most sophisticated fraudsters and cheaters. Most of them consist of searching for malware or evidence that the game software has been tampered altered.

In this sense, machine learning has helped by providing a statistical approach and tools to predict if a player is cheating based on his data (Yeung et al., 2006; Galli et al., 2011; Alkhalifa, 2016; Islam et al., 2020; Alayed et al., 2013).

Nearly all machine learning approaches to anti-cheating in video games consist of analyzing in-game data, which is information regarding the game environment (such as the player's avatar positioning or activity during

gameplay). Analyzing this data demands domain knowledge and a process of feature engineering for each game.

A system capable of analyzing gameplay without relying on in-game data would hold great value since it could be applied to several games without modification and be adaptive to new types of cheats.

4.2 DATASET DESCRIPTION

The dataset we used in our experiments was collected in a real-world context of players in the game Counter-Strike: Global Offensive, a first-person shooter. Players installed an application that collected keyboard and mouse events as previously described. Since we intended our dataset to be as realistic as possible, most data resulted from normal players (not cheaters) engaging in matches on the game's official servers. Players were given full freedom to play as they intended (as long as they were not cheating) to maximize behavior variety in the dataset.

We tested two types of cheating:

- Aimbot - a cheat that automatically aims towards the cheater's target, thus greatly reducing the need to perform mouse movements;
- Triggerbot - a script that automatically fires the weapon as soon as the crosshair reaches an opponent, thus reducing the need for a fast reaction.

These cheats greatly alter player behavior and allow much better performance in the game. The hypothesis motivating our approach is that these different behavioral patterns reflect in the multivariate time series, and are detected by the deep learning models.

As shown in Figure 4.2.1, our dataset contains interaction data from 118 players, 8 of whom have engaged in cheating. Labels of cheating interactions are rare, which is coherent with a real-world scenario where cheaters represent a small minority of players.

The labels were generated in scheduled matches by altering the cheating software to produce a timestamp for every cheat activation. Each record in our dataset corresponds to 5 minutes of gameplay. We label a record as a cheating record if there is at least one activation of the cheat during that interaction. In Figure 4.2.3 we can visualize two samples from our training dataset. Each record is a multivariate time series with 3000 time steps and 10 values in each timestep. We selected those 10 variables based on the frequence with which they occur, as seen in Figure 4.2.2, and on their respective performed function in the game.

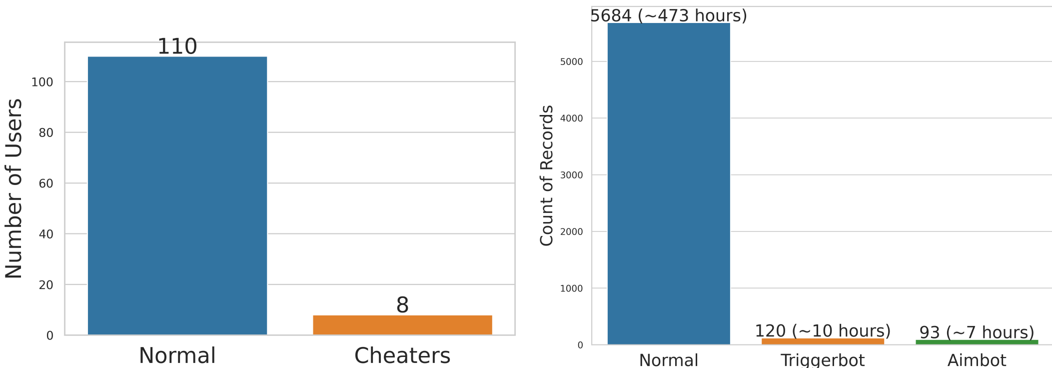


Figure 4.2.1: Distribution of players and data records by cheat.

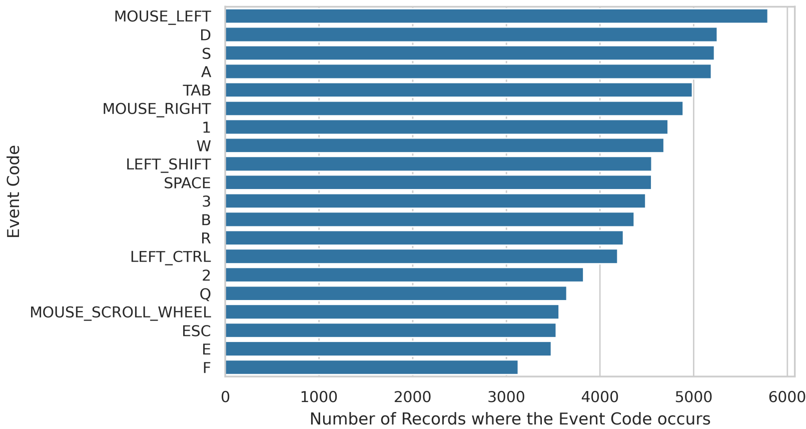


Figure 4.2.2: Event codes that appear the most often in our dataset.

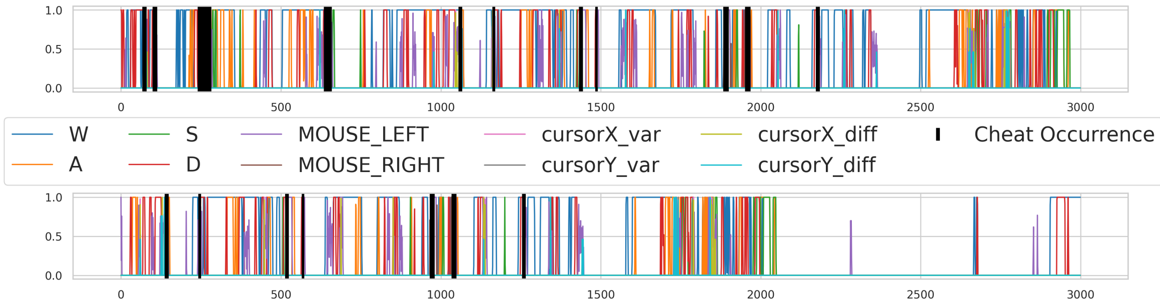


Figure 4.2.3: Two examples of records in our dataset.

Table 4.3.1: Hyperparameter search space and chosen values.

Hyperparameter	Type	Search Interval or Set	Chosen Value
L2 regularization λ	Real-valued	$[5 \times 10^{-5},\ 5 \times 10^{-3}]$	3×10^{-3}
Number of Convolutional Layers	Integer	$[2,\ 8]$	3
Number of Filters	Integer	$[4,\ 16]$	6
Filter Size	Integer	$[10,\ 50]$	25
Max Pooling	Boolean	$\{True,\ False\}$	$False$
Batch Size	Integer	$[32,\ 512]$	64

4.3 EXPERIMENTS

In this section, we explain our experiments. First, we introduce the hyperparameter search that, along with the strategy described in Section 3.3.2, resulted in the final architecture of our models. Then, we explain how we used the cross-validation method described in Section 3.3.3 and why it is superior to the methodology seen in previous work.

4.3.1 *Hyperparameter Search*

To arrive at our proposed architecture, we conducted several trials to search for a combination of hyperparameters that maximized the area under the receiver operating characteristic curve (AUC) metric.

We used TensorFlow (Abadi et al., 2015) and Keras (Chollet et al., 2015) for model implementation and Optuna (Akiba et al., 2019) for hyperparameter optimization. We used the Adam algorithm (Kingma and Ba, 2017) to optimize our models by minimizing L2 regularized cross-entropy, defined by

$$L(\boldsymbol{\theta};\boldsymbol{X};\boldsymbol{y}) = \lambda \|\boldsymbol{w}_{\boldsymbol{\theta}}\|_2^2 - \frac{\sum_{i=1}^{N} [\boldsymbol{y}_i \times \log(p(\boldsymbol{X}_i)) + (1-\boldsymbol{y}_i) \times \log(1-p(\boldsymbol{X}_i))]}{N} \tag{4.3.1}$$

where $\boldsymbol{y}_i$ and $p(\boldsymbol{x}_i)$ are the true and predicted labels of the sample $\boldsymbol{X}_i$, respectively, N is the number of samples in $\boldsymbol{X}$, λ is the L2 regularization hyperparameter and $\boldsymbol{w}_{\boldsymbol{\theta}}$ is the trainable subset of the model's parameters $\boldsymbol{\theta}$.

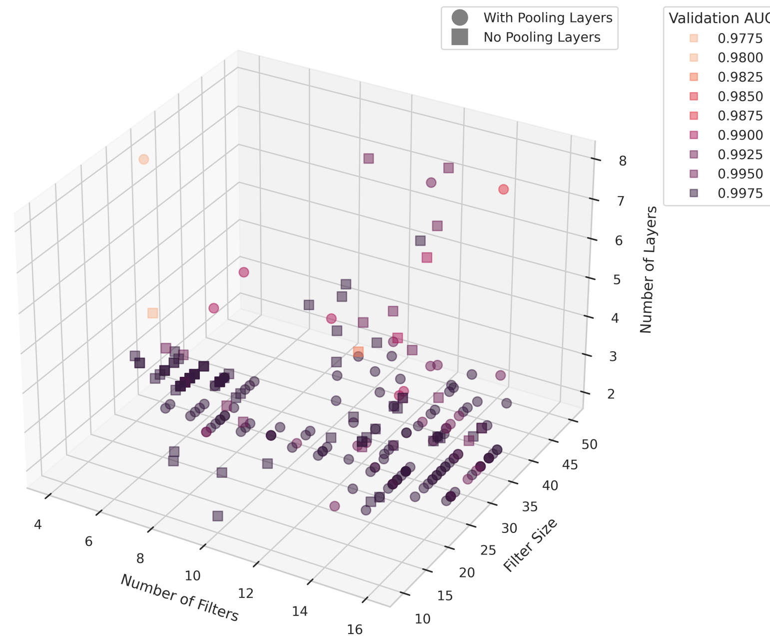


Figure 4.3.1: Distribution of the experimental results by number of layers, number of filters in each layer, and filter size.

Table 4.3.1 describes our hyperparameter search space. The number of layers and filters in each layer was intended to regulate complexity in our model. The filter size has an impact on model complexity, but it also determines the temporal range of the interactions between features. The batch size and the L2 λ value were the variable sources of regularization (we also used a fixed 0.5 dropout rate in every layer). We also tested the use of max-pooling to explore the potential benefits of dimensionality reduction in hidden features. In those models, we applied the max-pooling operation following each convolutional layer.

In Figures 4.3.1 and 4.3.2, we can observe the results of the hyperparameter optimization. It appears that two separate clusters are being formed according to the usage of max-pooling. In figure 4.3.3, we examine the distribution of the AUC metric and the loss function values in models with or without max-pooling layers. Models without max-pooling seem to achieve better results in both metrics while being more consistent in terms of loss function values.

Taking into account the results of the hyperparameter optimization, we went on to validate our approach with the values shown in the last column of Table 4.3.1.

Our proposed CNN architecture is illustrated in Figure 4.3.4. We follow each layer except for the last with a dropout mask to achieve better generalization. The use of a small number of filters can also be seen as a form of obtaining better generalization.

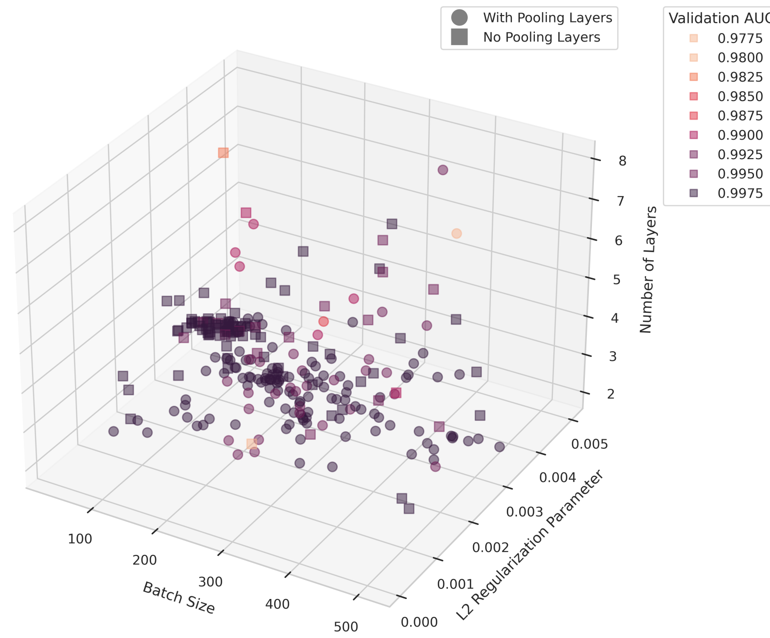


Figure 4.3.2: Distribution of the experimental results by batch size, L2 Regularization parameter, and number of layers.

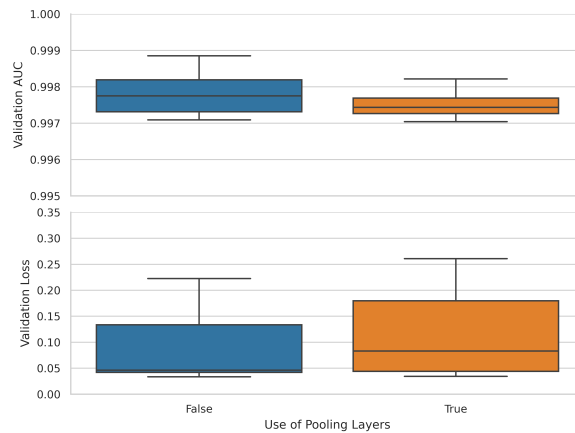


Figure 4.3.3: Distribution of the validation AUC and loss for the 100 best trials, by use of pooling layers.

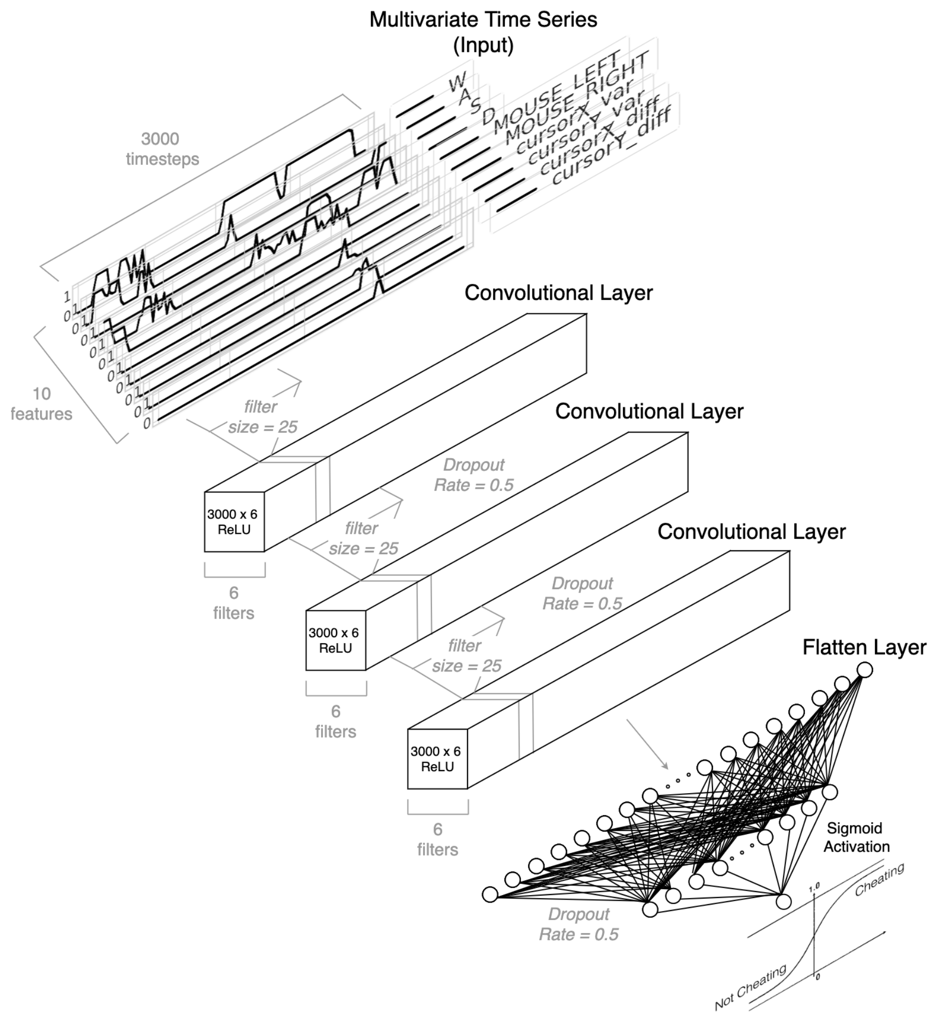


Figure 4.3.4: Proposed architecture for the CNN classifier.

Filter size has a strong meaning in this context because the range of timesteps interacting to originate a hidden feature determines the time period that influences that feature.

We can multiply that range by applying some sort of pooling operation, such as max-pooling, and although this can also mean fewer parameters and a model that is easier to train, models using max-pooling layers did not perform as well as the ones that don't use pooling.

Perhaps in tasks with a necessity for detecting patterns with a wide temporal range, deeper models that make use of pooling might be the better choice.

4.3.2 *Player-Based Cross-validation*

Previous works conducted experiments using rudimentary validation techniques and very small datasets.

Alayed et al. (2013) performed 10-fold cross-validation in 7.6 hours of data from only 2 players. Galli et al. (2011) used a simple train-test split with a dataset of also 2 players, but only 1 hour of data. In Islam et al. (2020), interactions from 20 players were collected but there is no reference to the volume of their dataset. Yeung et al. (2006) collected a mere 1.7 hours of data from 3 players and divided it into training, validation, and test datasets. As seen in Figure 4.2.1, our dataset consists of roughly 490 hours of gameplay and contains data from 128 players in total.

To quickly take action against fraudulent behavior, cheat detection systems must make accurate classifications for new players. No previous work took this necessity into account, so we employed the cross-validation method described in Section 3.3.3 to address it.

User-centered cross-validation not only tests our approach's ability to perform well in unseen data but most importantly of detecting cheating interactions in unknown players.

We repeated this experiment for both cheats in our dataset: triggerbot and aimbot. The results are presented in Tables 4.3.2 and 4.3.3 respectively. Triggerbot results were better, which might be due to the patterns associated with this type of cheat being easier to detect or the fact that there is more data on this type of cheat.

We chose the best by maximizing the validation true positive rate (TPR), as long as the false positive rate (FPR) didn't exceed 5%, which did not occur for any instance of our experiment.

The left column in Figure 4.3.5 shows the validation receiver operating characteristics curve for aimbot detection on four players. The right column shows the models' output distribution according to the ground truth. We can see that our models produce a very clear distinction between fraudulent and legitimate interactions.

Table 4.3.2: Results of Player-Based Cross-Validation in Triggerbot detection.

Player	Number of Records	AUC	Best Threshold t	FPR for t	TPR for t	Accuracy for t
1	40	0.997	0.080	0.047	1	0.975
2	375	1	0.381	0	1	1
3	121	0.996	0.027	0.010	0.95	0.992
4	129	1	0.54	0	1	1
5	126	1	0.348	0	1	1
7	26	1	0.591	0	1	1
8	320	0.992	0.005	0.027	1	0.978
AVG	**162.429**	**0.998**	**0.282**	**0.012**	**0.993**	**0.992**

Table 4.3.3: Results of Player-Based Cross-Validation in Aimbot detection.

Player	Number of Records	AUC	Best Threshold t	FPR for t	TPR for t	Accuracy for t
1	29	1	0.049	0	1	1
2	381	0.954	0.160	0.011	0.895	0.984
3	109	0.996	0.073	0.030	1	0.972
4	133	0.991	0.124	0.032	1	0.970
5	110	1	0.214	0	1	1
6	25	1	0.225	0	1	1
7	29	1	0.054	0	1	1
8	314	0.995	0.225	0.016	1	0.984
AVG	**137.625**	**0.992**	**0.141**	**0.011**	**0.987**	**0.989**

The magnitude of the predictions varied mostly due to the stochastic nature of the model's parameter initialization.

4.4 RESULTS DISCUSSION AND POSSIBLE IMPROVEMENTS

Results show that our models were able to establish a clear distinction between legitimate and fraudulent gameplay.

As previously mentioned, max-pooling can enable learning longer patterns, and according to our hyperparameter search, it allows for deeper networks. Different games or applications may require this additional capacity. For each new domain where this approach is applied, a new hyperparameter search should be performed to find a well-performing architecture.

The presented cross-validation method allowed us to show that our models learn patterns that are not player-specific. Our models can detect fraudulent players even if they weren't exposed to their behavior. We suspect that the principal source of variation in our results is the fact that some of the tested players (such as player #2) represent a significant portion of our dataset.

We used a reduced set of event types (movement keys, mouse buttons, and mouse movements). The number of necessary event types to include in the multivariate time series might vary for different domains, which influences the architecture resulting from the hyperparameter search.

Our models perform better than those in previous related work (Galli et al., 2011; Alayed et al., 2013; Islam et al., 2020; Alkhalifa, 2016) while also being tested in a much larger dataset. As showed in previous sections, our models achieved an accuracy of 98.9% and 99.2% in triggerbot and aimbot, respectively.

These results validate our ground-breaking approach to cheat detection in video games. The ability to analyze player behavior based solely on input data is remarkable, allowing for the application of this system in many different video games, perhaps with different input methods, thus greatly reducing the need for manual feature engineering.

This dataset has a large volume of data, collected from a wide variety of players in a real-world context. In this sense, we are confident that these results will translate to (or even improve with) communities with thousands or even millions of players.

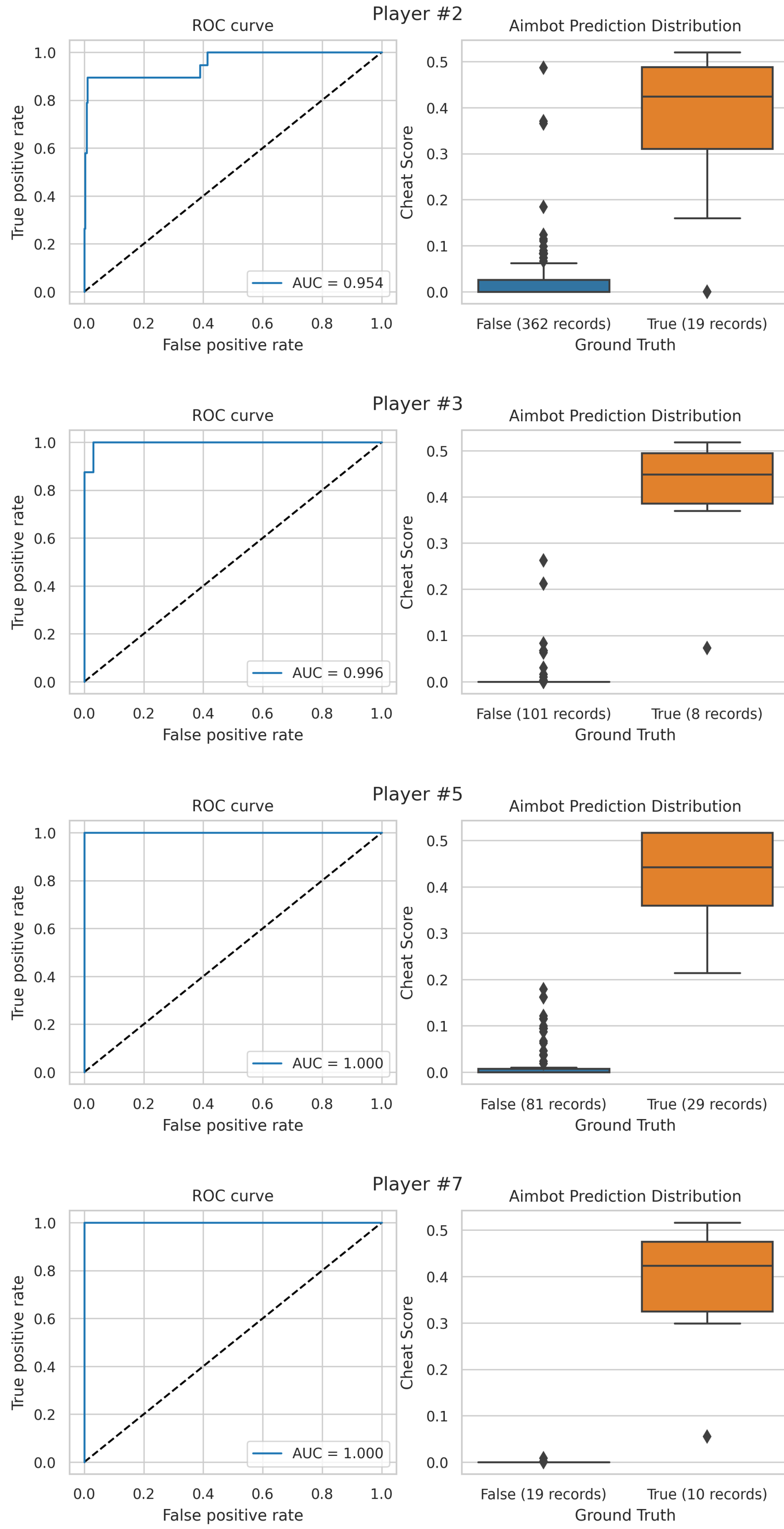


Figure 4.3.5: Validation AUC and prediction distribution for 4 of the tested players.

5

CONCLUSION

With deep learning and multivariate time series, we were able to create useful representations of human behavior. These models can capture complex patterns by analyzing a stream of signals originated from the interactions taking place. The decision to structure these streams into multivariate time series and the composable nature of deep learning models were essential to this approach's success.

We can summarize our achievements by revisiting the initial objectives for this project.

- *Review scientific concepts and literature related to HCI, AmI, and Sequential Data Analysis with Deep Learning.*

  We provided an in-depth literature review of these topics, which informed our ideation process for the proposed framework and allowed us to draw some conclusions regarding their current landscape. Our main observation is that HCI is currently not experiencing the same momentum as Machine Learning and Deep Learning, partly because its literature predominantly addresses user experience and design concerns. We believe that works such as ours, where we explore HCI towards its use in data analytics and behavioral models, should gain popularity in the following years.

- *Define a domain that successfully describes the analysis of human-computer interactions in the form of time series. Propose a systematic approach to problems that fit in the established domain.*

  We laid the foundations for a unified framework for performing multivariate time series analysis in HCI data. Our work relied on the generic concepts of event and sources of interaction allowing for a wide variety of types of interaction (with different peripherals and platforms) to be addressed with our proposed approach.

- *Develop an anti-cheating system for video games that is game-agnostic and relies on the proposed deep learning approach to HCI.*

We used deep learning to capture behavioral patterns in HCI data and proved the effectiveness of our proposed framework in the case study of fraud detection in video games.

Our models showed extremely positive results compared to any previous work, with accuracy levels of roughly 99% detecting aimbot and triggerbot. Our cheat detection system can be applied to any game and any input method. This system should be useful to the video-game industry, which is in great expansion and requiring such technologies.

Additionally, we conclude that multivariate time series are data structures rich enough to bear information regarding HCI, even in highly interactive settings such as video games.

This conclusion suggests further experimentation with other domains beyond entertainment, such as user segmentation or wellbeing monitorization (useful in remote work or e-learning, for example). In this sense, we can point towards several directions of future work.

- *Disseminate our findings and contribute to the scientific knowledge on the fields related to this work.*

  We wrote a scientific paper documenting our work in fraud detection in video games, titled *"Deep Learning and Multivariate Time Series for Fraud Detection in Video Games"*. This work was accepted at the Machine Learning Journal, one of the top journals in the field of Machine Learning and Artificial Intelligence. We'll also be presenting this paper at the IEEE Data Science and Advanced Analytics '21 conference, which is also an internationally renowned forum for sharing knowledge in the aforementioned fields.

Considering that we successfully fulfilled all of the initial objectives and that we achieved remarkable results, we find that this project was extremely successful and that it is appropriate to discuss future work directions.

FUTURE WORK

It will be interesting to observe the behavior of the developed anti-cheating system in communities with thousands or even millions of players.

Another important path to investigate is the use of unsupervised methods to leverage great volumes of unlabeled data, not just in cheat detection but also in other use-cases such as grouping players in clusters based on their experience.

Related to the first point, we should also improve the capability of our data pipeline to handle big data.

One limitation of the framework we presented is the arbitrary choice of the time series' length and granularity. Ideally, these two characteristics would be hyperparameters in our pipeline, which is a non-trivial goal because

these influence the data processing. A possible solution would be to implement several levels of hyperparameter optimization.

Implementing some additional (while still rudimentary) features calculated based on the mouse movements, along with the introduction of a feature selection mechanism, is another promising path for improvement.

One of the main research areas in deep learning, which we did not address in our work, is model explainability. Some insight regarding which behaviors or patterns most influence the models' predictions could be extremely helpful and even allow us to learn more about our interactions with computers.

Finally, we envision the core ideas in this approach extrapolating beyond video games and finding uses in HCI applied to wellbeing. In this sense, implementing the same data collection described in various platforms (such as web and mobile) would allow us to extend the range of this approach to case studies such as social networks and other massively popular occurrences of HCI.

BIBLIOGRAPHY

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

H. Alayed, F. Frangoudes, and C. Neuman. Behavioral-based cheating detection in online first person shooters using machine learning techniques. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, 2013. doi: 10.1109/CIG.2013.6633617.

Salman Alkhalifa. *Machine Learning and Anti-Cheating in FPS Games*. PhD thesis, 09 2016.

Ghulum Bakiri and Thomas G Dietterich. Achieving high-accuracy text-to-speech with machine learning. *Data mining in speech synthesis*, 10, 1999.

T. Baltrušaitis, P. Robinson, and L. Morency. Openface: An open source facial behavior analysis toolkit. In *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1–10, 2016. doi: 10.1109/WACV.2016.7477553.

T. Baltrušaitis, A. Zadeh, Y. C. Lim, and L. Morency. Openface 2.0: Facial behavior analysis toolkit. In *2018 13th IEEE International Conference on Automatic Face Gesture Recognition (FG 2018)*, pages 59–66, 2018. doi: 10.1109/FG.2018.00019.

P. Barr, J. Noble, and R. Biddle. Video game values: Human–computer interaction and games. *Interacting with Computers*, 19(2):180–195, 2007. doi: 10.1016/j.intcom.2006.08.008.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. doi: 10.1109/72.279181.

C. M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Computation*, 7(1):108–116, 1995. doi: 10.1162/neco.1995.7.1.108.

Ronald Boring. Human-computer interaction as cognitive science. *Proceedings of the Human Factors and Ergonomics Society*, 46:1767–1771, 09 2002. doi: 10.1177/154193120204602103.

Leo Breiman. Bagging predictors. In *Machine Learning*, volume 24, pages 123–140. Morgan Kaufmann, 1996. doi: https://doi.org/10.1023/A:1018054314350.

Davide Carneiro, Paulo Novais, José Miguel Pêgo, Nuno Sousa, and José Neves. Using mouse dynamics to assess stress during online exams. In Enrique Onieva, Igor Santos, Eneko Osaba, Héctor Quintián, and Emilio Corchado, editors, *Hybrid Artificial Intelligent Systems*, pages 345–356, Cham, 2015. Springer International Publishing.

Davide Carneiro, André Pimenta, Sérgio Gonçalves, José Neves, and Paulo Novais. Monitoring and improving performance in human–computer interaction. *Concurrency and Computation: Practice and Experience*, 28 (4):1291–1309, 2016.

Davide Carneiro, André Pimenta, José Neves, and Paulo Novais. A multi-modal architecture for non-intrusive analysis of performance in the workplace. *Neurocomputing*, 231:41 – 46, 2017. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2016.05.105. URL http://www.sciencedirect.com/science/article/pii/S0925231216311614. Neural Systems in Distributed Computing and Artificial Intelligence.

Richard Caruana. Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 41–48. Morgan Kaufmann, 1993.

S. Chauhan and L. Vig. Anomaly detection in ecg time signals via deep long short-term memory networks. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–7, 2015. doi: 10.1109/DSAA.2015.7344872.

L. Chen, J. Hoey, C. D. Nugent, D. J. Cook, and Z. Yu. Sensor-based activity recognition. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):790–808, 2012. doi: 10.1109/TSMCC.2012.2198883.

P. P. Chen, A. Guitart, A. F. del Río, and Á. Periáñez. Customer lifetime value in video games using deep learning and parametric models. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 2134–2140, 2018. doi: 10.1109/BigData.2018.8622151.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL http://arxiv.org/abs/1512.01274.

Y. Cherdo, P. d. Kerret, and R. Pawlak. Training lstm for unsupervised anomaly detection without a priori knowledge. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4297–4301, 2020. doi: 10.1109/ICASSP40776.2020.9053744.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

François Chollet et al. Keras. https://keras.io, 2015.

Sunny Consolvo, David W. McDonald, and James A. Landay. Theory-driven design strategies for technologies that support behavior change in everyday life. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, page 405–414, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605582467. doi: 10.1145/1518701.1518766. URL https://doi.org/10.1145/1518701.1518766.

Diane J. Cook, Juan C. Augusto, and Vikramaditya R. Jakkula. Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, 5(4):277 – 298, 2009. ISSN 1574-1192. doi: https://doi.org/10.1016/j.pmcj.2009.04.001. URL http://www.sciencedirect.com/science/article/pii/S157411920900025X.

Terrance Devries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout. *CoRR*, abs/1708.04552, 2017. URL http://arxiv.org/abs/1708.04552.

Thomas G. Dietterich. Machine learning for sequential data: A review. In Terry Caelli, Adnan Amin, Robert P. W. Duin, Dick de Ridder, and Mohamed Kamel, editors, *Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-70659-5.

Ken Ducatel, Marc Bogdanowicz, Fabiana Scapolo, Jos Leijten, and Jean Burgelman. Scenarios for ambient intelligence in 2010. 01 2001.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. 12(null):2121–2159, July 2011. ISSN 1532-4435.

Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. 1980.

L. Galli, D. Loiacono, L. Cardamone, and P. L. Lanzi. A cheating detection framework for unreal tournament iii: A machine learning approach. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 266–272, 2011. doi: 10.1109/CIG.2011.6032016.

Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V. Le. Dropblock: A regularization method for convolutional networks. *CoRR*, abs/1810.12890, 2018. URL http://arxiv.org/abs/1810.12890.

Ian Goodfellow, Oriol Vinyals, and Andrew Saxe. Qualitatively characterizing neural network optimization problems. In *International Conference on Learning Representations*, 2015. URL http://arxiv.org/abs/1412.6544.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*, volume 385. 01 2012. ISBN 978-3-642-24796-5. doi: 10.1007/978-3-642-24797-2.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001. doi: 10.1162/106365601750190398.

M. A. Hayes and M. A. M. Capretz. Contextual anomaly detection in big sensor data. In *2014 IEEE International Congress on Big Data*, pages 64–71, 2014. doi: 10.1109/BigData.Congress.2014.19.

Eric B. Hekler, Predrag Klasnja, Jon E. Froehlich, and Matthew P. Buman. Mind the theoretical gap: Interpreting, using, and developing behavioral theory in hci research. CHI '13, page 3307–3316, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318990. doi: 10.1145/2470654.2466452. URL https://doi.org/10.1145/2470654.2466452.

Thomas Hewett, Ronald Baecker, Stuart Card, Jean Gasen, Marilyn Tremaine, Gary Perlman, Gary Strong, and William Verplank. Acm sigchi curricula for human-computer interaction, 09 1992.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL http://www.sciencedirect.com/science/article/pii/0893608089900208.

M. S. Islam, B. Dong, S. Chandra, L. Khan, and B. M. Thuraisingham. Gci: A gpu based transfer learning approach for detecting cheats of computer game. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2020. doi: 10.1109/TDSC.2020.3013817.

Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC Workshop on Computational Creativity and Games*, page 9, 2016.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248. PMLR, 2016.

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International conference on machine learning*, pages 2342–2350, 2015.

F. Karim, S. Majumdar, H. Darabi, and S. Chen. Lstm fully convolutional networks for time series classification. *IEEE Access*, 6:1662–1669, 2018. doi: 10.1109/ACCESS.2017.2779939.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

A. Kołakowska. A review of emotion recognition methods based on keystroke dynamics and mouse movements. In *2013 6th International Conference on Human System Interactions (HSI)*, pages 548–555, 2013. doi: 10.1109/HSI.2013.6577879.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. 2012.

S. Lin, R. Clark, R. Birke, S. Schönborn, N. Trigoni, and S. Roberts. Anomaly detection for time series using vae-lstm hybrid model. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4322–4326, 2020. doi: 10.1109/ICASSP40776.2020.9053558.

Weining Lu, Yu Cheng, Cao Xiao, Shiyu Chang, Shuai Huang, Bin Liang, and Thomas Huang. Unsupervised sequential outlier detection with deep architectures. *IEEE transactions on image processing*, 26(9):4321–4330, 2017.

Pankaj Malhotra, Vishnu TV, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. Timenet: Pre-trained deep recurrent neural network for time series classification. *CoRR*, abs/1706.08838, 2017. URL http://arxiv.org/abs/1706.08838.

Regan Mandryk and Kori Inkpen. Physiological indicators for the evaluation of co-located collaborative play. pages 102–111, 01 2004. doi: 10.1145/1031607.1031625.

Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math;, 1997.

M. Munir, S. A. Siddiqui, A. Dengel, and S. Ahmed. Deepant: A deep learning approach for unsupervised anomaly detection in time series. *IEEE Access*, 7:1991–2005, 2019. doi: 10.1109/ACCESS.2018.2886457.

Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52(1):77–124, Jun 2019. ISSN 1573-7462. doi: 10.1007/s10462-018-09679-z. URL https://doi.org/10.1007/s10462-018-09679-z.

Henry Friday Nweke, Ying Wah Teh, Mohammed Ali Al-garadi, and Uzoma Rita Alo. Deep learning algorithms for human activity recognition using mobile and wearable sensor networks: State of the art and research challenges. *Expert Systems with Applications*, 105:233 – 261, 2018. ISSN 0957-4174. doi: https:

//doi.org/10.1016/j.eswa.2018.03.056. URL http://www.sciencedirect.com/science/article/pii/S0957417418302136.

Randy Pagulayan, Kevin Keeker, Dennis Wixon, Ramon Romero, and Thomas Fuller. User-centered design in games. *Human-Computer Interact. Handb*, pages 883–906, 01 2003. doi: 10.1201/b11963-39.

H. Pao, K. Chen, and H. Chang. Game bot detection via avatar trajectory analysis. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(3):162–175, 2010. doi: 10.1109/TCIAIG.2010.2072506.

Nathan Partlan, Abdelrahman Madkour, Chaima Jemmali, Josh Miller, Christoffer Holmgård, and Magy El-Nasr. Player imitation for build actions in a real-time strategy game. 11 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

E. J. Pauwels, A. A. Salah, and R. Tavenard. Sensor networks for ambient intelligence. In *2007 IEEE 9th Workshop on Multimedia Signal Processing*, pages 13–16, 2007. doi: 10.1109/MMSP.2007.4412806.

J. Pereira and M. Silveira. Learning representations from healthcare time series data for unsupervised anomaly detection. In *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 1–7, 2019. doi: 10.1109/BIGCOMP.2019.8679157.

André Pimenta, Davide Carneiro, Paulo Novais, and José Neves. Analysis of human performance as a measure of mental fatigue. In Marios Polycarpou, André C. P. L. F. de Carvalho, Jeng-Shyang Pan, Michał Woźniak, Héctor Quintian, and Emilio Corchado, editors, *Hybrid Artificial Intelligence Systems*. Springer International Publishing, 2014.

André Pimenta, Davide Carneiro, Paulo Novais, and José Neves. Detection of distraction and fatigue in groups through the analysis of interaction patterns with computers. In David Camacho, Lars Braubach, Salvatore

Venticinque, and Costin Badica, editors, *Intelligent Distributed Computing VIII*, pages 29–39, Cham, 2015. Springer International Publishing.

Ondrej Pluskal and J. Sedivý. Predicting players behavior in games with microtransactions. In *STAIRS*, 2014.

B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1 – 17, 1964. ISSN 0041-5553. doi: https://doi.org/10.1016/0041-5553(64)90137-5. URL http://www.sciencedirect.com/science/article/pii/0041555364901375.

Pranav Rajpurkar, Awni Y Hannun, Masoumeh Haghpanahi, Codie Bourn, and Andrew Y Ng. Cardiologist-level arrhythmia detection with convolutional neural networks. *arXiv preprint arXiv:1707.01836*, 2017.

Niklas Ravaja, Mikko Salminen, Jussi Holopainen, Timo Saari, Jari Laarni, and Aki Järvinen. Emotional response patterns and sense of presence during video games: Potential criterion variables for game design. volume 82, pages 339–347, 01 2004. doi: 10.1145/1028014.1028068.

Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. KDD '16, page 2135, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2945397. URL https://doi.org/10.1145/2939672.2945397.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016. doi: 10.1109/JPROC.2015.2494218.

B. Siegel. Industrial anomaly detection: A comparison of unsupervised neural network architectures. *IEEE Sensors Letters*, 4(8):1–4, 2020. doi: 10.1109/LSENS.2020.3007880.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. 15(1):1929–1958, January 2014. ISSN 1532-4435.

Jonathan Sykes and Simon Brown. Affective gaming: measuring emotion through the gamepad. pages 732–733, 01 2003. doi: 10.1145/765891.765957.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014. URL http://arxiv.org/abs/1312.6199.

Anja Thieme, Danielle Belgrave, and Gavin Doherty. Machine learning in mental health: A systematic review of the hci literature to support the development of effective and implementable ml systems. *ACM Trans. Comput.-Hum. Interact.*, 27(5), August 2020. ISSN 1073-0516. doi: 10.1145/3398069. URL https://doi.org/10.1145/3398069.

Wei Wang, Alex X. Liu, Muhammad Shahzad, Kang Ling, and Sanglu Lu. Understanding and modeling of wifi signal based human activity recognition. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, page 65–76, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336192. doi: 10.1145/2789168.2790093. URL https://doi.org/10.1145/2789168.2790093.

M. Willman. Machine learning to identify cheaters in online games, 2020. URL http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-170973.

S. F. Yeung, J. C. S. Lui, Jiangchuan Liu, and J. Yan. Detecting cheaters for multiplayer games: theory, design and implementation[1]. In *CCNC 2006. 2006 3rd IEEE Consumer Communications and Networking Conference, 2006.*, volume 2, pages 1178–1182, 2006. doi: 10.1109/CCNC.2006.1593224.

J. Yin, Q. Yang, and J. J. Pan. Sensor-based abnormal human-activity detection. *IEEE Transactions on Knowledge and Data Engineering*, 20(8):1082–1090, 2008. doi: 10.1109/TKDE.2007.1042.

Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. *CoRR*, abs/1905.04899, 2019. URL http://arxiv.org/abs/1905.04899.

Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *CoRR*, abs/1710.09412, 2017. URL http://arxiv.org/abs/1710.09412.